

**Dragan Milićev**

**Programiranje u realnom  
vremenu**

***Skripta***

**Beograd, 2002.**



---

# Sadržaj

---

Sadržaj	1
<b>I OSNOVE OBJEKTNO ORIJENTISANOG PROGRAMIRANJA NA JEZIKU C++</b>	<b>4</b>
Pregled osnovnih koncepata OOP-a na jeziku C++	5
Klase	5
Konstruktori i destruktori	6
Nasleđivanje	7
Polimorfizam	8
Vežbe	9
Elementi jezika C++ koji nisu objektno orijentisani	12
Ugrađeni tipovi i deklaracije	12
Pokazivači	12
Nizovi	13
Izrazi	14
Naredbe	15
Funkcije	15
Struktura programa	16
Oblast važenja imena	17
Životni vek objekata	18
Konverzije tipova	20
Dinamički objekti	21
Funkcije	21
Operatori i izrazi	24
Vežbe	25
Klase i preklapanje operatora	27
Pojam i deklaracija klase	27
Pokazivač <code>this</code>	28
Zajednički članovi klase	29
Prijatelji klase	30
Konstruktori	31
Destruktor	33
Pojam preklapanja operatora	33
Operatori <code>new</code> i <code>delete</code>	34
Primeri struktura pogodnih za RT implementacije	35
Nasleđivanje	58
Izvedene klase	58
Polimorfizam	61
Višestruko nasleđivanje	66
<b>II OSNOVE OBJEKTNO ORIJENTISANOG MODELOVANJA NA JEZIKU UML</b>	<b>67</b>
Modelovanje strukture	68
Klasa, atributi i operacije	68
Asocijacija	68

Zavisnost	71
Generalizacija/Specijalizacija	72
Interfejsi	73
<b>Modelovanje ponašanja</b>	<b>74</b>
Interakcije i dijagrami interakcije	74
Aktivnosti i dijagrami aktivnosti	76
Mašine stanja i dijagrami prelaza stanja	78
<b>Organizacija modela</b>	<b>85</b>
Paketi	85
Dijagrami	85
<b>III UVOD U SISTEME ZA RAD U REALNOM VREMENU</b>	<b>86</b>
<b>Definicija sistema za rad u realnom vremenu</b>	<b>87</b>
Podela i terminologija RT sistema	87
Primeri RT sistema	88
<b>Karakteristike RT sistema</b>	<b>89</b>
<b>IV POUZDANOST I TOLERANCIJA OTKAZA</b>	<b>90</b>
<b>Pouzdanost i tolerancija otkaza</b>	<b>91</b>
Pouzdanost, padovi i otkazi	91
Sprečavanje i tolerancija otkaza	92
N-Version Programiranje	93
Dinamička softverska redundansa	95
Blokovi oporavka	98
<b>Izuzeci i njihova obrada</b>	<b>101</b>
Dinamička redundansa i izuzeci	101
Obrada izuzetaka bez posebne jezičke podrške	101
Izuzeci i njihova reprezentacija	102
Obrada izuzetka	104
Propagacija izuzetka	106
Vežbe	107
<b>V OSNOVE KONKURENTNOG PROGRAMIRANJA</b>	<b>110</b>
<b>Konkurentnost i procesi</b>	<b>111</b>
Konkurentno programiranje	111
Pojam procesa	113
Predstavljanje procesa	114
Interakcija između procesa	117
Implementacija niti	118
Vežbe	136
<b>Sinhronizacija i komunikacija pomoću deljene promenljive</b>	<b>138</b>
Međusobno isključenje i uslovna sinhronizacija	138
Uposleno čekanje	140
Semafori	143
Uslovni kritični regioni	148
Monitori	149
Klasifikacija poziva operacija	155
Implementacija sinhronizacionih primitiva	155
Vežbe	160

---

<b>Sinhronizacija i komunikacija pomoću poruka</b>	<b>163</b>
Sinhronizacija procesa	163
Imenovanje procesa i struktura poruke	165
Randevo u jeziku Ada	166
Komunikacija u metodi ROOM	169
Vežbe	170
<b>Kontrola resursa</b>	<b>171</b>
Modeli za pristup deljenim resursima	171
Problemi nadmetanja za deljene resurse	173
Vežbe	180
 <b>VI SPECIFIČNOSTI RT PROGRAMIRANJA</b>	 <b>182</b>
<b>Realno vreme</b>	<b>183</b>
Časovnik realnog vremena	183
Merenje proteklog vremena	186
Vremenske kontrole	187
Kašnjenje procesa	191
Specifikacija vremenskih zahteva	192
Kontrola zadovoljenja vremenskih zahteva	196
Implementacija u školskom Jezgru	198
Vežbe	208
<b>Raspoređivanje</b>	<b>217</b>
Osnovne strategije raspoređivanja	217
Testovi rasporedivosti	221
Opštiji model procesa	227
Projektovanje prema vremenskim zahtevima	234
Vežbe	238

# **I      Osnove objektno          orijentisanog          programiranja          na jeziku C++**

---

# Pregled osnovnih koncepata OOP-a na jeziku C++

---

\* U ovoj glavi biće dat kratak i sasvim površan pregled osnovnih koncepata OOP koje podržava C++. Potpunija i preciznija objašnjenja koncepata biće data kasnije, u posebnim glavama.

\* Primeri koji se koriste u ovoj glavi samo su pokazni, nisu pravljeni da budu upotrebljivi. Iz realizacije primera izbačeno je sve što bi smanjivalo preglednost osnovnih ideja. Zato su primeri često i nekompletni.

\* Čitalac ne treba da se trudi da posle čitanja ove glave zapamti sintaksu rešenja, niti da otkrije sve pojedinosti pokazanih primera. Cilj je da čitalac samo stekne predstavu o osnovnim idejama OOP-a i jezika C++, da vidi šta je novo i šta se sve može uraditi, kao i da nauči da razmišlja na novi, objektni način.

## Klase

\* *Klasa* (engl. *class*) je osnovna organizaciona jedinica programa u OOP jezicima, pa i u jeziku C++. Klasa predstavlja strukturu u koju su grupisani podaci i funkcije:

```
// Deklaracija klase:
```

```
class Osoba {  
public:  
    void koSi();           // funkcija: predstavi se!  
                           // ... i još nešto  
private:  
    char* ime;             // podatak: ime i prezime  
    int  god;              // podatak: koliko ima godina  
};
```

```
// Svaka funkcija se mora i definisati:
```

```
void Osoba::koSi () {  
    cout<<"Ja sam "<<ime<<" i imam "<<god<<" godina.\n";  
}
```

\* Klasom se definiše novi, korisnički tip za koji se mogu kreirati instance (primeri, promenljive). Klasa može da predstavlja realizaciju apstrakcije iz domena problema.

\* Instance klase nazivaju se *objekti* (engl. *objects*). Svaki objekat ima sopstvene elemente koji su navedeni u deklaraciji klase. Ovi elementi klase nazivaju se *članovi* klase (engl. *class members*). Članovima se pristupa pomoću operatora "->":

```
/* Korišćenje klase Osoba:  
   negde u programu se prave objekti tipa Osoba, */
```

```
Osoba* pera = new Osoba;  
Osoba* mojOtac = new Osoba;  
Osoba* direktor = new Osoba;
```

```
/* a onda se oni koriste: */
```

```
pera->koSi();          // poziv funkcije koSi objekta pera
mojOtac->koSi();        // poziv funkcije koSi objekta mojOtac
direktor->koSi();       // poziv funkcije koSi objekta direktor
```

```
/* i na kraju unište kada više nisu potrebni: */
```

```
delete pera;
delete mojOtac;
delete direktor;
```

\* Ako pretpostavimo da su ranije, na neki način, postavljene vrednosti članova svakog od navedenih objekata, ovaj segment programa daje:

```
Ja sam Petar Markovic i imam 25 godina.
Ja sam Slobodan Milicev i imam 58 godina.
Ja sam Aleksandar Simic i imam 40 godina.
```

\* Specifikator `public`: govori prevodiocu da su samo članovi koji se nalaze iza njega dostupni spolja. Ovi članovi nazivaju se *javnim*. Članovi iza specifikatora `private`: su nedostupni korisnicima klase (ali ne i članovima klase) i nazivaju se *privatnim*:

```
/* Izvan članova klase nije moguće: */
```

```
pera->ime="Petar Markovic";      // nedozvoljeno
mojOtac->god=55;                 // takođe nedozvoljeno
```

```
/* Šta bi tek bilo da je ovo dozvoljeno: */
direktor->ime="bu...., kr...., ...";
direktor->god=1000;
/* a onda ga neko pita (što je dozvoljeno): */
direktor->koSi();
/* ?! */
```

## Konstruktori i destruktori

\* Da bi se omogućila inicijalizacija objekta, u klasi se definiše posebna funkcija koja se implicitno (automatski) poziva kada objekat nastaje. Ova funkcija se naziva *konstruktor* (engl. *constructor*) i nosi isto ime kao i klasa.

```
class Osoba {
public:
    Osoba(char* ime, int godine); // konstruktor
    void koSi();                  // funkcija: predstavi se!
private:
    char* ime;                   // podatak: ime i prezime
    int god;                     // podatak: koliko ima godina
};
```

```
/* Svaka funkcija se mora i definisati: */
```

```
void Osoba::koSi () {
    cout<<"Ja sam "<<ime<<" i imam "<<god<<" godina.\n";
}
```

```
Osoba::Osoba (char* i, int g) {
    if (proveriIme(i))           // proveri ime da nije ružno
        ime=i;
    else
```



```

    ime="necu da ti kazem ko";
    god=((g>=0 && g<=100)?g:0); // proveriti godine
}

/* Korišćenje klase Osoba sada je: */

Osoba* pera = new Osoba("Petar Markovic",25); // poziv konstruktora Osoba
Osoba* mojOtac = new Osoba("Slobodan Milicev",58);

pera->koSi();
mojOtac->koSi();

delete pera;
delete mojOtac;
delete direktor;

```

\* Ovakav deo programa može dati ranije navedene rezultate.

\* Moguće je definisati i funkciju koja se poziva uvek kada objekat prestaje da živi. Ova funkcija naziva se *destruktor*.

## Nasleđivanje

\* Pretpostavimo da nam je potreban novi tip, Maloletnik. Maloletnik je "jedna vrsta" osobe, koja "posедуje sve što i osoba, samo ima još nešto", ima staratelja. Ovakva relacija između klasa naziva se *nasleđivanje*.

\* Kada nova klasa predstavlja "jednu vrstu" druge klase (engl. *a-kind-of*), kaže se da je ona izvedena iz osnovne klase.

```

class Maloletnik : public Osoba {
public:
    Maloletnik(char* ime, char* staratelj, int godine); // konstruktor
    void koJeOdgovoran();
private:
    char* staratelj;
};

void Maloletnik::koJeOdgovoran () {
    cout<<"Za mene odgovara "<<staratelj<<".\n";
}

Maloletnik::Maloletnik (char* i, char* s, int g) : Osoba(i,g), staratelj(s)
{}

```

\* Izvedena klasa Maloletnik ima sve članove kao i osnovna klasa Osoba, ali ima još i članove staratelj i koJeOdgovoran. Konstruktor klase Maloletnik definiše da se objekat ove klase kreira zadavanjem imena, staratelja i godina, i to tako da se konstruktor osnovne klase Osoba (koji inicijalizuje ime i godine) poziva sa odgovarajućim argumentima. Sam konstruktor klase Maloletnik samo inicijalizuje staratelja.

\* Sada se mogu koristiti i nasleđene osobine objekata klase Maloletnik a na raspolaganju su i njihova posebna svojstva kojih nije bilo u klasi Osoba:

```

Osoba* otac = new Osoba("Petar Petrovic",40);
Maloletnik* dete = new Maloletnik("Milan Petrovic","Petar Petrovic",12);

otac->koSi();

```

```
dete->koSi();
dete->koJeOdgovoran();
otac->koJeOdgovoran();           // Ovo, naravno, ne može!
```

```
/* Izlaz će biti:
Ja sam Petar Petrovic i imam 40 godina.
Ja sam Milan Petrovic i imam 12 godina.
Za mene odgovara Petar Petrovic.
*/
```

## Polimorfizam

\* Pretpostavimo da nam je potrebna nova klasa žena, koja je "jedna vrsta" osobe, samo što još ima i devojčko prezime. Klasa Zena biće izvedena iz klase Osoba.

\* I objekti klase Zena treba da se "odazivaju" na funkciju koSi, ali je teško pretpostaviti da će jedna dama otvoreno priznati svoje godine. Zato objekat klase Zena treba da ima funkciju koSi, samo što će ona izgledati malo drugačije, svojstveno izvedenoj klasi Zena:

```
class Osoba {
public:
    Osoba(char* ime, int godine); // Konstruktor
    virtual void koSi();          // Virtuelna funkcija
protected:                      // Dostupno naslednicima
    char* ime;                   // Podatak: ime i prezime
    int god;                     // Podatak: koliko ima godina
};

void Osoba::koSi () {
    cout<<"Ja sam "<<ime<<" i imam "<<god<<" godina.\n";
}

Osoba::Osoba (char* i, int g) : ime(i), god(g) {}

class Zena : public Osoba {
public:
    Zena(char* ime, char* devojacko, int godine);
    virtual void koSi();          // Nova verzija funkcije koSi()
private:
    char* devojacko;
};

Zena::Zena (char* i, char* d, int g) : Osoba(i,g), devojacko(d) {}

void Zena::koSi () {
    cout<<"Ja sam "<<ime<<", devojacko prezime "<<devojacko<<".\n";
}
```

\* Funkcija članica koja će u izvedenim klasama imati nove verzije deklarise se u osnovnoj klasi kao *virtuelna funkcija* (virtual). Izvedena klasa može da da svoju definiciju virtuelne funkcije, ali i ne mora. U izvedenoj klasi ne mora se navoditi reč virtual.

\* Da bi članovi osnovne klase Osoba bili dostupni izvedenoj klasi Zena, ali ne i korisnicima spolja, oni se deklarise iza specifikatora protected: i nazivaju *zaštićenim* članovima.

\* Drugi delovi programa, korisnici klase `Osoba`, ako su dobro projektovani, ne vide nikakvu promenu zbog uvođenja izvedene klase. Oni uopšte ne moraju da se menjaju:

```
/* Funkcija "ispitaj" propituje osobe i ne mora da se menja: */

void ispitaj (Osoba* hejTi) {
    hejTi->koSi();
}

/* U drugom delu programa koristimo novu klasu Zena: */

Osoba* otac = new Osoba("Petar Petrovic",40);
Zena* majka = new Zena("Milka Petrovic","Mitrovic",35);
Maloletnik* dete = new Maloletnik("Milan Petrovic","Petar Petrovic",12);

ispitaj(otac);           // Pozvaće se Osoba::koSi()
ispitaj(majka);          // Pozvaće se Zena::koSi()
ispitaj(dete);           // Pozvaće se Osoba::koSi()

delete otac;
delete majka;
delete dete;

/* Izlaz će biti:
Ja sam Petar Petrovic i imam 40 godina.
Ja sam Milka Petrovic, devojacko prezime Mitrovic.
Ja sam Milan Petrovic i imam 12 godina.
*/
```

\* Funkcija `ispitaj` dobija pokazivač na tip `Osoba`. Kako je i žena osoba, C++ dozvoljava da se pokazivač na tip `Zena` (majka) *konvertuje* (pretvori) u pokazivač na tip `Osoba` (`hejTi`). Mehanizam virtuelnih funkcija obezbeđuje da funkcija `ispitaj`, preko pokazivača `hejTi`, pozove pravu verziju funkcije `koSi`. Zato će se za argument majka pozivati funkcija `Zena::koSi`, a za argument otac funkcija `Osoba::koSi`. Za argument dete takođe će se pozvati funkcija `Osoba::koSi`, jer klasa `Maloletnik` nije redefinisala virtuelnu funkciju `koSi`.

\* Navedeno svojstvo da se odaziva prava verzija funkcije klase čiji su naslednici dali nove verzije naziva se *polimorfizam* (*polymorphism*).

## Vežbe

### 1.1

Šta ispisuju dati segmenti programa?

a)

```
class Cube {
public:
    Cube (int initial);
    int throwing();
private:
    int last;
};

Cube::Cube(int i) : last(i) {}

int Cube::throwing() {
    last = (3*last+5)%6 + 1; // %6 je ostatak pri deljenju sa 6
```

```
    return last;
}

Cube* c1 = new Cube(1);
Cube* c2 = new Cube(3);
cout << (c1->throwing()) << (c1->throwing()) << (c1->throwing());
cout << (c2->throwing()) << (c2->throwing()) << (c2->throwing());
cout << (c1->throwing()) << (c2->throwing()) << (c1->throwing());
```

b)

```
class LicnostIzBajke {
public:
    virtual char* umeDaLeti () { return "NE."; }
    //...
};

class Vestica : public LicnostIzBajke {
public:
    virtual char* umeDaLeti () { return "DA, na metli." ; }
    //...
};

LicnostIzBajke* ivica = new LicnostIzBajke;
LicnostIzBajke* marica = new LicnostIzBajke;
LicnostIzBajke* veca = new Vestica;
cout << ivica->umeDaLeti() << marica->umeDaLeti() << veca->umeDaLeti();
LicnostIzBajke* tajna = ivica;
cout << tajna->umeDaLeti();
tajna = veca;
cout << tajna->umeDaLeti();
```

## 1.2

Data je sledeća klasa:

```
class Person {
public:
    Person (char* name);
    char* getName();
protected:
    void setName(char* newName);
private:
    char* name;
};

Person::Person(char* n) : name(n) {}
char* Person::getName() { return name; }
void Person::setName(char* newName) { name = newName; }
```

Šta nije ispravno u datim segmentima programa?

a)

```
Person* father = new Person();
```

b)

```
Person* son = new Person("Bart", "Simpson");
```

c)

```
Person* father = new Person("Homer Simpson");
father->setName("");
```

d)

```
Person* daughter = new Person("Lisa Simpson");  
cout << daughter->name;
```

e)

```
class Woman : public Person {  
public:  
    Woman (char* born);  
private:  
    char* born;  
};  
  
Woman::Woman (char* b) : born(b) {}
```

f)

```
class Woman : public Person {  
public:  
    Woman (char* name, char* born);  
private:  
    char* born;  
};  
  
Woman::Woman (char* n, char* b) : Person(""), born(b) { name = n; }
```

g)

```
class Woman : public Person {  
public:  
    Woman (char* name, char* born);  
private:  
    char* born;  
};  
  
Woman::Woman (char* n, char* b) : name(n), born(b) {}
```

### 1.3

Projektovati klasu `Timer` koja apstrahuje koncept *vremenskog brojača*. Brojač se može inicijalizovati vremenom koje treba da meri. Na svaki otkucaj koji se generiše spolja, pozivom operacije `tick()`, brojač odbrojava jednu jedinicu vremena. Kada dobroji do nule, brojač treba da zaustavi dalje brojanje (ignoriše dalje otkucaje). Brojaču se može postaviti novo vreme, kao i očitati preostalo ili proteklo vreme. Na primeru pokazati upotrebu nekoliko različitih brojača.

### 1.4

Projektovati klase koje apstrahuju koncept *porta* kao sredstva za pristup do ulaznih i/ili izlaznih uređaja računara. Port ima svoju adresu u adresnom prostoru računara (adresa porta je vrednost tipa `void*`). Na port se može upisivati vrednost ili sa njega čitati vrednost (pretpostaviti da je vrednost porta tipa `int`). Posebne vrste portova su *ulazni* portovi, koji ne dozvoljavaju upis. Takođe postoji i posebna vrsta izlaznih portova na koje se upis vrši najpre upisom adrese, a zatim i vrednosti na datu fiksnu adresu u adresnom prostoru računara. Ilustrovati na primerima način korišćenja portova koji ne zavisi od vrste porta, već im svima pristupa na isti način.

---

# Elementi jezika C++ koji nisu objektno orijentisani

---

## Ugrađeni tipovi i deklaracije

- \* Ugrađeni tipovi nisu realizovani kao klase, već kao jednostavne strukture podataka.
- \* *Deklaracija* je iskaz koji uvodi neko ime u program. Ime se može koristiti samo ako je prethodno deklarirano. Deklaracija govori prevodiocu kojoj jezičkoj kategoriji pripada neko ime i šta se sa tim imenom može raditi.
- \* *Definicija* je ona deklaracija koja kreira objekat (alocira memorijski prostor za njega) ili daje telo funkcije.
- \* U OOP, objekat je instanca klase. U jeziku C++, objektom se smatra instanca bilo kog tipa, i ugrađenog tipa i klase.
- \* Neki osnovni ugrađeni tipovi su: ceo broj (`int`), znak (`char`) i racionalni broj (`float` i `double`). Objekat može biti inicijalizovan u deklaraciji; takva deklaracija je i definicija:

```
int i;  
int j=0, k=3;  
float f1=2.0, f2=0.0;  
double PI=3.14;  
char a='a', nul='\0';
```

## Pokazivači

- \* *Pokazivač* je objekat koji *ukazuje* na neki drugi objekat. Pokazivač se realizuje tako što sadrži adresu objekta na koji ukazuje.
- \* Ako pokazivač `p` ukazuje na objekat `x`, onda je rezultat operacije `*p` objekat `x` (operacija dereferenciranja pokazivača).
- \* Rezultat operacije `&x` je pokazivač koji ukazuje na objekat `x` (operacija uzimanja adrese).
- \* Tip "pokazivač na tip `T`" označava se sa `T*`. Na primer:

```
int i=0, j=0; // objekti i i j tipa int;  
int *pi;      // objekat pi je tipa "pokazivač na int" (tip: int*);  
pi=&i;        // vrednost pokazivača pi je adresa objekta i,  
              // pa pi ukazuje na i;  
*pi=2;        // *pi označava objekat i; i postaje 2;  
j=*pi;        // j postaje jednak objektu na koji ukazuje pi,  
              // a to je i;  
pi=&j;        // pi sada sadrži adresu j, tj. ukazuje na j;
```

- \* Na isti način se mogu definisati pokazivači na proizvoljan tip. Ako je `p` pokazivač koji ukazuje na objekat klase sa članom `m`, onda je `(*p).m` isto što i `p->m`:

```
Osoba* po = new Osoba("Petar Simić",40); // po je pokazivač na tip Osoba;
(*po).koSi(); // poziv funkcije koSi objekta *po;
po->koSi(); // isto što i (*po).koSi();
```

\* Tip na koji pokazivač ukazuje može biti proizvoljan, pa i drugi pokazivač:

```
int i=0, j=0; // i i j tipa int;
int *pi=&i; // pi je pokazivač na int, ukazuje na i;
int **ppi; // ppi je tipa "pokazivač na - pokazivač na - int";
ppi=&pi; // ppi ukazuje na pi;
*pi=1; // pi ukazuje na i, pa i postaje 1;
**ppi=2; // ppi ukazuje na pi,
// pa je rezultat operacije *ppi objekat pi;
// rezultat još jedne operacije * je objekat na koji ukazuje
// pi, a to je i; i postaje 2;
*ppi=&j; // ppi ukazuje na pi, pa pi sada ukazuje na j,
// a ppi još uvek na pi;
ppi=&i; // greška: ppi je pokazivač na pokazivač na int,
// a ne pokazivač na int!
```

\* Pokazivač tipa `void*` može ukazivati na objekat bilo kog tipa. Ne postoje objekti tipa `void`, ali postoje pokazivači tipa `void*`.

\* Pokazivač koji ima posebnu, simboličku vrednost `0` ne ukazuje ni na jedan objekat. Ovakav pokazivač razlikuje se od bilo kog drugog pokazivača koji ukazuje na neki objekat. Može se ispitati da li pokazivač ukazuje na neki objekat ili ne (tj. da li je jednak `0` ili ne).

## Nizovi

\* Niz je objekat koji sadrži nekoliko objekata nekog tipa. Niz je, kao i pokazivač, izvedeni tip. Tip "niz objekata tipa `T`" označava se sa `T[]`.

\* Niz se deklarise na sledeći način:

```
int a[100]; // a je objekat tipa "niz objekata tipa int" (tip: int[]);
// sadrži 100 elemenata tipa int;
```

\* Ovaj niz ima 100 elemenata koji se indeksiraju od 0 do 99;  $i+1$ -vi element je `a[i]`:

```
a[2]=5; // treći element niza a postaje 5
a[0]=a[0]+a[99];
```

\* Elementi mogu biti bilo kog tipa, pa čak i nizovi. Na ovaj način se definišu višedimenzionalni nizovi:

```
int m[5][7]; // m je niz od 5 elemenata;
// svaki element je niz od 7 elemenata tipa int;
m[3][5]=0; // pristupa se četvrtom elementu niza m;
// on je niz elemenata tipa int;
// pristupa se zatim njegovom šestom elementu i on postaje 0;
```

\* Nizovi i pokazivači su blisko povezani u jezicima C i C++. Sledeća tri pravila povezuju nizove i pokazivače:

1. Svaki put kada se ime niza koristi u nekom izrazu, osim u operaciji uzimanja adrese (operator `&`), implicitno se konvertuje u pokazivač na svoj prvi element. Na primer, ako je `a` tipa `int[]`, onda se on konvertuje u tip `int*`, sa vrednošću adrese prvog elementa niza (to je početak niza).

2. Definisana je operacija sabiranja pokazivača i celog broja, ako su zadovoljeni sledeći uslovi: pokazivač ukazuje na element nekog niza i rezultat sabiranja je opet pokazivač koji ukazuje na element istog niza ili za jedno mesto iza poslednjeg elementa niza. Rezultat sabiranja `p+i`, gde je `p` pokazivač a `i` ceo broj, jeste pokazivač koji ukazuje `i` elemenata iza

elementa na koji ukazuje pokazivač `p`. Ako navedeni uslovi nisu zadovoljeni, rezultat operacije je nedefinisan. Analogna pravila postoje za operacije oduzimanja celog broja od pokazivača, kao i inkrementiranja i dekrementiranja pokazivača.

3. Operacija `a[i]` je, po definiciji, ekvivalentna sa `*(a+i)`.

Na primer:

```
int a[10]; // a je niz objekata tipa int;
int *p=&a; // p ukazuje na a[0];
a[2]=1;    // a[2] je isto što i *(a+2); a se konvertuje u pokazivač
           // koji ukazuje na a[0]; rezultat sabiranja je pokazivač
           // koji ukazuje na a[2]; dereferenciranje tog pokazivača (*)
           // predstavlja zapravo a[2]; a[2] postaje 1;
p[3]=3;    // p[3] je isto što i *(p+3), a to je a[3];
p=p+1;     // p sada ukazuje na a[1];
*(p+2)=1;  // a[3] postaje sada 1;
p[-1]=0;   // p[-1] je isto što i *(p-1), a to je a[0];
```

## Izrazi

\* *Izraz* je iskaz u programu koji sadrži operande (objekte, funkcije ili literale nekog tipa), operacije nad tim operandima i proizvodi rezultat tačno definisanog tipa. Operacije se zadaju pomoću operatora ugrađenih u jezik.

\* Operator može da prihvata jedan, dva ili tri operanda strogo definisanih tipova, i proizvodi rezultat koji se može koristiti kao operand nekog drugog operatora. Na ovaj način se formiraju složeni izrazi.

\* Prioritet operatora definiše redosled izračunavanja operacija unutar izraza. Podrazumevani redosled izračunavanja može se promeniti pomoću zagrada `()`.

\* C i C++ su veoma bogati operatorima. Zapravo, najveći deo obrade u jednom programu definisano je izrazima.

\* Mnogi ugrađeni operatori imaju sporedni efekat: pored toga što proizvode rezultat, oni menjaju vrednost nekog od svojih operandada.

\* Postoje operatori za inkrementiranje (`++`) i dekrementiranje (`--`), u prefiksnoj i postfixnoj formi. Ako je `i` nekog od numeričkih tipova ili pokazivač, `i++` znači "inkrementiraj `i`, a kao rezultat vrati njegovu staru vrednost"; `++i` znači "inkrementiraj `i` a kao rezultat vrati njegovu novu vrednost". Slično važi za dekrementiranje.

\* Dodela vrednosti vrši se pomoću operatora dodele `=`: `a=b` znači "dodeli vrednost izraza `b` objektu `a`, a kao rezultat vrati tu dodeljenu vrednost". Ovaj operator grupiše zdesna ulevo. Tako:

```
a=b=c; // dodeli c objektu b i vrati tu vrednost, a onda je dodeli a-u;
       // prema tome, c je dodeljen i objektu b i objektu a;
```

\* Postoji i operator složene dodele: `a+=b` znači isto što i `a=a+b`, ali se izraz `a` samo jednom izračunava:

```
a+=b;    // isto što i a=a+b;
a-=b;    // isto što i a=a-b;
a*=b;    // isto što i a=a*b;
a/=b;    // isto što i a=a/b;
```



[illegible]

```
void f(); // a vraća tip int;
          // globalna funkcija bez argumenata
          // koja nema povratnu vrednost;
```

\* Definicija funkcije daje i telo funkcije. Telo funkcije je složena naredba (blok):

```
int Counter::inc () { // definicija funkcije članice; vraća int;
    return count++;   // vraća se rezultat izraza;
}
```

\* Funkcija može vratiti vrednost koja je rezultat izraza u naredbi `return`.

\* Unutar tela funkcije mogu se deklarirati lokalna imena (tačnije unutar svakog ugnežđenog bloka):

```
int Counter::inc () {
    int temp; // temp je lokalni objekat
    temp=count+1; // count je član klase Counter
    count=temp;
    return temp;
}
```

\* Funkcija članica neke klase može pristupiti članovima sopstvenog objekta bez posebne specifikacije. Globalna funkcija mora navesti objekat čijem članu pristupa.

\* Poziv funkcije obavlja se pomoću operatora `()`. Rezultat ove operacije je rezultat poziva funkcije:

```
int f(int); // deklaracija globalne funkcije
Counter* c = new Counter; // pokazivač c na objekat klase Counter
int a=0, b=1;
a=b+c->inc(); // poziv funkcije c->inc() koji vraća int
a=f(b); // poziv globalne funkcije f
delete c; // uništavanje objekta na koji ukazuje c
```

\* Može se deklarirati i pokazivač na funkciju:

```
int f(int); // f je tipa "funkcija koja prima jedan argument tipa int
            // i vraća int";
int (*p)(int); // p je tipa
               // "pokazivač na funkciju
               // koja prima jedan argument tipa int
               // i vraća int";
p=&f; // p ukazuje na f;
int a;
a=(*p)(1); // poziva se funkcija na koju ukazuje p, a to je funkcija f;
```

## Struktura programa

\* Program se sastoji samo od deklaracija (klasa, objekata, ostalih tipova i funkcija). Sva obrada koncentrisana je unutar tela funkcija.

\* Program se fizički deli na odvojene jedinice prevođenja – datoteke. Datoteke se prevode odvojeno i nezavisno, a zatim se povezuju u izvršni program. U svakoj datoteci se moraju deklarirati sva imena pre nego što se koriste.

\* Zavisnosti između modula-datoteka definišu se pomoću *datoteka-zaglavlja*. Zaglavlja sadrže deklaracije svih entiteta koji su definisani u datom modulu, a koriste se u drugim modulima. Zaglavlja (`.h`) se uključuju u tekst datoteke koja se prevodi (`.cpp`) pomoću direktive `#include`.

\* Glavni program (izvor toka kontrole) definiše se kao obavezna funkcija `main`. Primer jednostavnog, ali kompletnog programa:

```

class Counter {
public:
    Counter();
    int inc(int by);
private:
    int count;
};

Counter::Counter () : count(0) {}

int Counter::inc (int by) {
    return count+=by;
}

void main () {
    Counter* a = new Counter;
    Counter* b = new Counter;
    int i=0, j=3;
    i=a->inc(2)+b->inc(++j);
    delete a; delete b;
}

```

## Oblast važenja imena

\* *Oblast važenja imena* (engl. *scope*) je deo teksta programa u kome se deklarirano ime može koristiti.

\* Globalna imena su imena koja se deklariraju van svih funkcija i klasa. Njihova oblast važenja je deo teksta od mesta deklaracije do kraja datoteke.

\* Lokalna imena su imena deklarirana unutar bloka, uključujući i blok tela funkcije. Njihova oblast važenja je od mesta deklariranja, do završetka bloka u kome su deklarirane.

```

int x;                // globalni x

void f () {
    int x;            // lokalni x, sakriva globalni x;
    x=1;              // pristup lokalnom x
    {
        int x;        // drugi lokalni x, sakriva prethodnog
        x=2;          // pristup drugom lokalnom x
    }
    x=3;              // pristup prvom lokalnom x
}

```

```

int *p=&x;            // uzimanje adrese globalnog x

```

\* Globalnom imenu može se pristupiti, iako je sakriveno, navođenjem operatora "::" ispred imena:

```

int x;                // globalni x

void f () {
    int x=0;          // lokalni x
    ::x=1;            // pristup globalnom x;
}

```

\* Za formalne argumente funkcije smatra se da su lokalni, deklarirani u krajnje spoljašnjem bloku tela funkcije:

```
void f (int x) {  
    int x;          // pogrešno  
}
```

\* Prvi izraz u naredbi `for` može da bude definicija objekta. Tako se dobija lokalno ime za blok u kome se nalazi `for`:

```
{  
    for (int i=0; i<10; i++) {  
        //...  
        if (a[i]==x) break;  
        //...  
    }  
    if (i==10) // može se pristupati imenu i  
}
```

\* Oblast važenja klase imaju svi članovi klase. To su imena deklarirana unutar deklaracije klase. Imenu koje ima oblast važenja klase, izvan te oblasti može se pristupiti preko operatora `"."` i `"->"`, gde je levi operand objekat, odnosno pokazivač na objekat date klase ili klase izvedene iz date klase; ili preko operatora `":"`, gde je levi operand ime klase:

```
class X {  
public:  
    int x;  
    void f();  
};  
  
void X::f () { /*...*/ }  
X* xx = new X;  
xx->x=0;  
xx->X::f(); // može i ovako
```

\* Oblast važenja funkcije imaju samo labele (za `goto` naredbe). One se mogu navesti bilo gde (i samo) unutar tela funkcije, a vide se u celoj funkciji.

## Životni vek objekata

\* Životni vek objekta je vreme postojanja (u memoriji) tog objekta tokom izvršavanja programa. Za to vreme se objektu može pristupiti.

\* Na početku životnog veka, objekat se inicijalizuje (poziva se njegov konstruktor ako je to primerak klase), a na kraju se objekat ukida (poziva se njegov destruktor ako je to primerak klase). Zbog toga se nastanak (ili kreiranje) objekta čvrsto vezuje za njegovu inicijalizaciju.

\* U odnosu na životni vek, postoje automatski, statički, dinamički i privremeni objekti, kao i objekti koji su članovi drugih objekata.

\* Životni vek *automatskog* objekta (lokalni objekat koji nije deklarisan kao `static`) traje od nailaska na njegovu definiciju, do napuštanja oblasti važenja tog objekta. Automatski objekat nastaje iznova pri svakom pozivu bloka u kome je deklarisan. Definicija objekta klase je izvršna naredba jer uzrokuje poziv njegovog konstruktora u vreme izvršavanja.

\* Životni vek *statičkih* objekata (globalni i lokalni `static` objekti) traje od izvršavanja njihove definicije do kraja izvršavanja programa. Globalni statički objekti se inicijalizuju samo jednom, na početku izvršavanja programa, pre korišćenja bilo koje funkcije ili objekta iz istog fajla, ne obavezno pre poziva funkcije `main`, a prestaju da žive po završetku funkcije `main`. Lokalni statički objekti počinju da žive pri prvom nailasku toka programa na njihovu definiciju.

\* Primer:

```
int glob=1;           // globalni objekat; životni vek mu
                      // traje do kraja programa;

void f () {
    int lok=2;        // lokalni objekat; životni vek mu je do
                      // izlaska iz spoljnog bloka funkcije;
    static int sl=3;   // lokalni statički objekat; oblast
                      // važenja je funkcija, a životni vek je ceo
                      // program; inicijalizuje se samo jednom;
    for (int i=0; i<sl; i++) {
        int j=i;      // j je lokalni za for blok
        //...
    }
}
```

\* Primer:

```
int a=1;

void f () {
    int b=1;          // inicijalizuje se pri svakom pozivu
    static int c=1;    // inicijalizuje se samo jednom
    printf(" a = %d ",a++);
    printf(" b = %d ",b++);
    printf(" c = %d\n",c++);
}

void main () {
    while (a<4) f();
}

// izlaz će biti:
// a = 1 b = 1 c = 1
// a = 2 b = 1 c = 2
// a = 3 b = 1 c = 3
```

\* Životni vek *dinamičkih* objekata neposredno kontroliše programer. Oni nastaju operacijom `new`, a nestaju operacijom `delete`.

\* Životni vek *privremenih* objekata je kratak i nedefinisan. Ovi objekti nastaju pri izračunavanju izraza, za odlaganje međurezultata ili privremeno smeštanje vraćene vrednosti funkcije. Najčešće se uništavaju čim više nisu potrebni.

\* Životni vek članova klase je isti kao i životni vek objekta kome pripadaju.

\* Formalni argumenti funkcije, pri pozivu funkcije, nastaju kao lokalni automatski objekti i inicijalizuju se stvarnim argumentima. Dakle, na mestu ulaska u funkciju, u trenutku poziva funkcije, nastaju formalni argumenti kao lokalni automatski objekti i inicijalizuju se stvarnim argumentima. Semantika ove inicijalizacije formalnog argumenta ista je kao i inicijalizacija bilo kog objekta u definiciji.

\* Rezultat poziva funkcije predstavlja objekat koji na mestu poziva funkcije, u trenutku povratka iz funkcije, nastaje kao bezimeni privremeni objekat i koji se inicijalizuje izrazom iza naredbe `return`. Semantika te inicijalizacije potpuno je ista kao i inicijalizacija objekta u definiciji:

```
X f (X x1) { // U trenutku poziva iz main(), kao da se radi: X x1=x3;
    ...
    return x2;
}

void main () {
```

```
...f(x3)... // U trenutku povratka, kao da se radi: X temp = x2;
}

*      U jeziku C++, svi objekti, i primerci ugrađenih tipova i primerci klasa, mogu biti svih
kategorija po životnom veku:

// X može biti i ugrađeni tip i klasa:
X xs; // Statički objekat tipa X sa imenom xs

X f (X xal) { // Formalni argument je automatski objekat sa imenom xal
    X xa2; // Automatski objekat sa imenom xa2
    ...
    ...new X...; // Dinamički, bezimeni objekat tipa X
    ...
}

void g () {
    ...f(xs)... // Povratna vrednost je privremeni, bezimeni objekat tipa X
}

class Y {
    ...
    X xm; // Član tipa X sa imenom xm
    ...
};
```

## Konverzije tipova

- \* C++ je strogo tipizirani jezik, što je u duhu njegove objektno orijentacije.
- \* Tipizacija znači da svaki objekat ima tačno određen tip. Svaki put kada se na nekom mestu očekuje objekat jednog tipa, a koristi se objekat drugog tipa, potrebno je izvršiti *konverziju* tipova.
- \* Konverzija tipa znači pretvaranje objekta datog tipa u objekat potrebnog tipa.
- \* Slučajevi kada se može desiti da se očekuje jedan tip, a dostavlja se drugi, odnosno kada je potrebno vršiti konverziju su:
  1. operatori za ugrađene tipove zahtevaju operande odgovarajućeg tipa;
  2. neke naredbe (if, for, do, while, switch) zahtevaju izraze odgovarajućeg tipa;
  3. pri pozivu funkcije, kada su stvarni argumenti drugačijeg tipa od deklariranih formalnih argumenata;
  4. pri povratku iz funkcije, ako se u izrazu iza return koristi izraz drugačijeg tipa od deklarisanog tipa povratne vrednosti funkcije;
  5. pri inicijalizaciji objekta jednog tipa pomoću objekta drugog tipa. Slučaj 3 se može svesti u ovu grupu, jer se formalni argumenti inicijalizuju stvarnim argumentima pri pozivu funkcije. I slučaj 4 se može svesti u ovu grupu, jer se privremeni objekat, koji prihvata vraćenu vrednost funkcije na mestu poziva, inicijalizuje izrazom iza naredbe return.
- \* Konverzija tipa može biti ugrađena u jezik (standardna konverzija) ili je definiše korisnik (programer) za svoje tipove (korisnička konverzija).
- \* Standardne konverzije su, na primer, konverzije iz tipa `int` u tip `float`, ili iz tipa `char` u tip `int`, konverzija pokazivača na izvedenu klasu u pokazivač na osnovnu klasu itd.
- \* Prevodilac može izvršiti konverziju koja mu je dozvoljena, na mestu gde je to potrebno; ovakva konverzija naziva se *implicitnom*. Programer može navesti koja konverzija treba da se izvrši; ova konverzija naziva se *eksplicitnom*.
- \* Jedan način zahtevanja eksplicitne konverzije je pomoću operatora *cast*: *(tip) izraz*.

## Dinamički objekti

\* Operator `new` pravi dinamički objekat, a operator `delete` uništava dinamički objekat nekog tipa `T`.

\* Operator `new` za svoj operand ima identifikator tipa i eventualne inicijalizatore (argumente konstruktora). Operator `new` alokira potreban prostor u slobodnoj memoriji za objekat datog tipa, a zatim poziva konstruktor tipa sa zadatim vrednostima. Operator `new` vraća pokazivač na dati tip:

```
Complex* pc1 = new Complex(1.3, 5.6);
Complex* pc2 = new Complex(-1.0, 0);
```

\* Objekat kreiran pomoću operatora `new` naziva se dinamički objekat, jer mu je životni vek poznat tek u vreme izvršavanja programa. Ovakav objekat nastaje kada se izvrši operator `new`, a traje sve dok se ne ukine operatorom `delete` (može da traje i po završetku bloka u kome je nastao):

```
Complex* pc;

void f() {
    pc=new Complex(0.1, 0.2);
}

void main () {
    f();
    delete pc;    // ukidanje objekta *pc
}
```

\* Operator `delete` ima jedan operand koji je pokazivač na neki tip. Ovaj pokazivač mora da ukazuje na objekat nastao pomoću operatora `new`. Operator `delete` poziva destruktora za objekat na koji ukazuje pokazivač, a zatim oslobađa zauzeti prostor. Ovaj operator vraća `void`.

\* Operatorom `new` može se napraviti i niz objekata nekog tipa. Ovakav niz ukida se operatorom `delete` sa parom uglastih zagrada:

```
Complex* pc = new Complex[10];
//...
delete [] pc;
```

\* Kada se alokira niz, nije moguće zadati inicijalizatore. Ako klasa nema definisan konstruktor, prevodilac obezbeđuje podrazumevanu inicijalizaciju. Ako klasa ima konstruktore, da bi se alocirao niz, potrebno je da postoji konstruktor koji se može pozvati bez argumenata.

\* Kada se alokira niz, operator `new` vraća pokazivač na prvi element alociranih niza. Sve dimenzije niza, osim prve, treba da budu konstantni izrazi. Prva dimenzija može da bude i promenljivi izraz, ali takav da može da se izračuna u trenutku izvršavanja naredbe sa operatorom `new`.

## Funkcije

### *Deklaracije funkcija i prenos argumenata*

\* Funkcije se deklariraju i definišu kao i u jeziku C, samo što je moguće kao tipove argumenata i rezultata navesti korisničke tipove (klase).

- \* U deklaraciji funkcije ne moraju da se navode imena formalnih argumenata.
- \* Pri pozivu funkcije, upoređuju se tipovi stvarnih argumenata sa tipovima formalnih argumenata navedenim u deklaraciji, i, po potrebi, vrši se konverzija. Semantika prenosa argumenata jednaka je semantici inicijalizacije.
- \* Pri pozivu funkcije, inicijalizuju se formalni argumenti, kao automatski lokalni objekti pozvane funkcije. Ovi objekti se konstruišu pozivom odgovarajućih konstruktora, ako ih ima. Pri vraćanju vrednosti iz funkcije, semantika je ista: konstruiše se privremeni objekat koji prihvata vraćenu vrednost na mestu poziva:

```
class Tip {  
    //...  
public:  
    Tip(int i);           // konstruktor  
};  
  
Tip f (Tip k) {  
    //...  
    return 2;            // poziva se konstruktor Tip(2)  
}  
  
void main () {  
    Tip k(0);  
    k=f(1);               // poziva se konstruktor Tip(1)  
    //...  
}
```

### *Neposredno ugrađivanje u kôd*

- \* Često se definišu vrlo jednostavne, kratke funkcije (na primer samo prosleđuju argumente drugim funkcijama). Tada je vreme koje se troši na prenos argumenata i poziv duže od vremena izvršavanja tela same funkcije.

\* Ovakve funkcije se mogu deklarirati tako da se neposredno ugrađuju u kôd (*inline* funkcije). Tada se telo funkcije direktno ugrađuje u pozivajući kôd. Semantika poziva ostaje potpuno ista kao i za običnu funkciju.

- \* Ovakva funkcija deklarira se kao *inline*:

```
inline int inc(int i) {return i+1;}
```

- \* Funkcija članica klase može biti *inline* ako se definiše unutar deklaracije klase, ili izvan deklaracije klase, kada se ispred njene deklaracije nalazi reč *inline*:

```
class C {  
public:  
    int val () {return i;} // ovo je inline funkcija  
private:  
    int i;  
};  
  
// ili:  
  
class D {  
public:  
    int val ();  
private:  
    int i;  
};  
  
inline int D::val() {return i;}
```



\* Prevodilac ne mora da uvaži zahtev za neposredno ugrađivanje u kôd. Programeru ovo ne treba da predstavlja nikakvu prepreku, jer je semantika ista. *Inline* funkcije samo mogu da ubrzaju program, a ne i da izmene njegovo izvršavanje.

\* Ako se *inline* funkcija koristi u više datoteka, u svakoj datoteci mora da postoji njena potpuna definicija (najbolje pomoću datoteke-zaglavlja).

### **Podrazumevane vrednosti argumenata**

\* C++ obezbeđuje i mogućnost postavljanja podrazumevanih vrednosti za argumente. Ako se pri pozivu funkcije ne navede argument za koji je definisana podrazumevana vrednost (u deklaraciji funkcije), kao vrednost stvarnog argumenta uzima se ta podrazumevana vrednost:

```
Complex::Complex (double r=0, double i=0) // podrazumevana
    {real=r; imag=i;} // vrednost za r i i je 0

void main () {
    Complex c; // kao da je napisano "Complex c(0,0);"
    //...
}
```

\* Podrazumevani argumenti mogu da budu samo nekoliko poslednjih iz liste:

```
Complex::Complex(double r=0, double i) // greška
    { real=r; imag=i; }
```

### **Preklapanje imena funkcija**

\* Često se javlja potreba da se u programu naprave funkcije koje realizuju logički istu operaciju, samo sa različitim tipovima argumenata. Za svaki od tih tipova mora, naravno, da se realizuje posebna funkcija. U jeziku C to bi se ostvarilo tako što bi se tim funkcijama dodelila različita imena. To, međutim, smanjuje čitljivost programa.

\* U jeziku C++ moguće je definisati više različitih funkcija sa istim identifikatorom. Ovakav koncept naziva se *preklapanje imena funkcija* (engl. *function overloading*). Uslov je da im se razlikuje broj i/ili tipovi argumenata. Tipovi rezultata ne moraju da se razlikuju:

```
char* max (char *p, char *q) { return (strcmp(p,q)>=0)?p;q; }
double max (double i, double j) { return (i>j) ? i : j; }

double r=max(1.5,2.5); // poziva se max(double,double)
char* q=max("Pera","Mika"); // poziva se max(char*,char*)
```

\* Koja će se funkcija stvarno pozvati, određuje se u fazi prevođenja, prema slaganju tipova stvarnih i formalnih argumenata. Zato je potrebno da prevodilac može jednoznačno da odredi koja se funkcija poziva.

\* Pravila za razrešavanje poziva veoma su složena, pa se u praksi svode samo na dovoljno jasno razlikovanje tipova formalnih argumenata preklapljenih funkcija. Kada razrešava poziv, prevodilac otprilike ovako pravi redosled slaganja tipova stvarnih i formalnih argumenata:

1. Najbolje je potpuno slaganje tipova; tipovi  $T^*$  (pokazivač na  $T$ ) i  $T[]$  (niz elemenata tipa  $T$ ) se ne razlikuju.
2. Sledeće je slaganje tipova korišćenjem standardnih konverzija.
3. Zatim sledi slaganje tipova korišćenjem korisničkih konverzija;
4. Najmanje odgovara slaganje sa tri tačke (. . .).

## Operatori i izrazi

\* Pregled operatora dat je u sledećoj tabeli. Operatori su grupisani po prioritetima, tako da su operatori u istoj grupi istog prioriteta, višeg od operatora koji su u narednoj grupi. U tablici su prikazane i ostale važne osobine: način grupisanja (asocijativnost, L – sleva udesno, D – zdesna ulevo), da li je rezultat lvrednost (D – da, N – nije, D/N – zavisi od nekog operanda, pogledati specifikaciju operatora u knjizi), kao i način upotrebe. Prazna polja ukazuju da svojstvo grupisanja nije primereno datom operatoru.

Operator	Značenje	Grup.	lvred.	Upotreba
::	razrešavanje oblasti važenja	L	D/N	<i>ime_klase :: član</i>
::	pristup globalnom imenu		D/N	<i>:: ime</i>
[ ]	indeksiranje	L	D	<i>izraz [ izraz ]</i>
( )	poziv funkcije	L	D/N	<i>izraz (lista_izraza)</i>
( )	konstrukcija vrednosti		N	<i>ime_tipa (lista_izraza)</i>
.	pristup članu	L	D/N	<i>izraz . ime</i>
->	posredni pristup članu	L	D/N	<i>izraz -&gt; ime</i>
++	postfiksni inkrement	L	N	<i>lvrednost++</i>
--	postfiksni dekrement	L	N	<i>lvrednost--</i>
++	prefiksni inkrement	D	D	<i>++lvrednost</i>
--	prefiksni dekrement	D	D	<i>--lvrednost</i>
sizeof	veličina objekta	D	N	<i>sizeof izraz</i>
sizeof	veličina tipa	D	N	<i>sizeof (tip)</i>
new	stvaranje dinamičkog objekta		N	<i>new tip</i>
delete	ukidanje dinamičkog objekta		N	<i>delete izraz</i>
~	komplement po bitovima	D	N	<i>~izraz</i>
!	logička negacija	D	N	<i>!izraz</i>
-	unarni minus	D	N	<i>-izraz</i>
+	unarni plus	D	N	<i>+izraz</i>
&	adresa	D	N	<i>&amp;lvrednost</i>
*	dereferenciranje pokazivača	D	D	<i>*izraz</i>
( )	konverzija tipa ( <i>cast</i> )	D	D/N	<i>(tip) izraz</i>
. *	posredni pristup članu	L	D/N	<i>izraz . * izraz</i>
-> *	posredni pristup članu	L	D/N	<i>izraz -&gt; * izraz</i>
*	množenje	L	N	<i>izraz * izraz</i>
/	deljenje	L	N	<i>izraz / izraz</i>
%	ostatak	L	N	<i>izraz % izraz</i>
+	sabiranje	L	N	<i>izraz + izraz</i>
-	oduzimanje	L	N	<i>izraz - izraz</i>
<<	pomeranje ulevo	L	N	<i>izraz &lt;&lt; izraz</i>
>>	pomeranje udesno	L	N	<i>izraz &gt;&gt; izraz</i>
<	manje od	L	N	<i>izraz &lt; izraz</i>
<=	manje ili jednako od	L	N	<i>izraz &lt;= izraz</i>
>	veće od	L	N	<i>izraz &gt; izraz</i>
>=	veće ili jednako od	L	N	<i>izraz &gt;= izraz</i>
==	jednako	L	N	<i>izraz == izraz</i>

!=	nije jednako	L	N	<i>izraz != izraz</i>
&	I po bitovima	L	N	<i>izraz &amp; izraz</i>
^	isključivo ILI po bitovima	L	N	<i>izraz ^ izraz</i>
	ILI po bitovima	L	N	<i>izraz   izraz</i>
&&	logičko I	L	N	<i>izraz &amp;&amp; izraz</i>
	logičko ILI	L	N	<i>izraz    izraz</i>
? :	uslovni operator	L	D/N	<i>izraz ? izraz : izraz</i>
=	prosta dodela	D	D	<i>lvrednost = izraz</i>
*=	množenje i dodela	D	D	<i>lvrednost *= izraz</i>
/=	deljenje i dodela	D	D	<i>lvrednost /= izraz</i>
%=	ostatak i dodela	D	D	<i>lvrednost %= izraz</i>
+=	sabiranje i dodela	D	D	<i>lvrednost += izraz</i>
-=	oduzimanje i dodela	D	D	<i>lvrednost -= izraz</i>
>>=	pomeranje udesno i dodela	D	D	<i>lvrednost &gt;&gt;= izraz</i>
<<=	pomeranje ulevo i dodela	D	D	<i>lvrednost &lt;&lt;= izraz</i>
&=	I i dodela	D	D	<i>lvrednost &amp;= izraz</i>
=	ILI i dodela	D	D	<i>lvrednost  = izraz</i>
^=	isključivo ILI i dodela	D	D	<i>lvrednost ^= izraz</i>
,	sekvenca	L	D/N	<i>izraz , izraz</i>

## Vežbe

### 2.1

Napisati definicije sledećih struktura podataka:

- (a) Niz pokazivača na x.      (b) Pokazivač na (prvi element) niza iz tačke a.
- (c) Pokazivač na funkciju koja prima pokazivač na funkciju koja prima pokazivač na niz elemenata tipa x; obe funkcije vraćaju `void`.
- (d) Niz pokazivača na niz elemenata tipa x.
- (e) Niz pokazivača na funkcije koje primaju niz elemenata tipa x i vraćaju pokazivač na niz znakova.

### 2.2

Napisati funkciju koja prebrojava različite elemente u datom celobrojnom nizu.

### 2.3

Napisati funkciju koja iz datog celobrojnog niza izbacuje elemente koji su jednaki zadatoj vrednosti.

### 2.4

Napisati funkciju koja sabira dva pozitivna cela broja u "neograničenoj tačnosti". Brojevi su predstavljeni kao nizovi decimalnih cifara.

**2.5**

Napisati funkciju koja u datom nizu znakova pretvara prva slova reči u velika, a ostala u mala. Reči su rastavljene blanko znacima.

**2.6**

Napisati skup funkcija koje operišu jednostruko ulančanim dinamičkim listama.

---

# Klase i preklapanje operatora

---

## Pojam i deklaracija klase

\* *Klasa* je realizacija apstrakcije koja ima svoju internu predstavu (svoje atribute) i operacije koje se mogu vršiti nad njenim instancama. Klasa definiše tip. Jedan primerak takvog tipa (instanca klase) naziva se *objektom* te klase (engl. *class object*).

\* Podaci koji su deo klase nazivaju se *podaci članovi* klase (engl. *data members*). Funkcije koje su deo klase nazivaju se *funkcije članice* klase (engl. *member functions*).

\* Članovi (podaci ili funkcije) klase iza ključne reči `private`: zaštićeni su od pristupa spolja (enkapsulirani su). Ovim članovima mogu pristupati samo funkcije članice klase. Ovi članovi nazivaju se *privatnim članovima klase* (engl. *private class members*).

\* Članovi iza ključne reči `public`: dostupni su spolja i nazivaju se *javnim članovima klase* (engl. *public class members*).

\* Članovi iza ključne reči `protected`: dostupni su funkcijama članicama date klase, kao i klasa izvedenih iz te klase, ali ne i korisnicima spolja, i nazivaju se *zaštićenim članovima klase* (engl. *protected class members*).

\* Redosled sekcija `public`, `protected` i `private` proizvoljan je, ali se preporučuje baš navedeni redosled. Podrazumeva se da su članovi privatni (ako se ne navede specifikator ispred).

\* Objekat klase ima unutrašnje stanje, predstavljeno vrednostima atributa, koje menja pomoću operacija. Funkcije članice nazivaju se još i *metodima* klase, a poziv ovih funkcija – *upućivanje poruke* objektu klase. Objekat klase menja svoje stanje kada se pozove njegov metod, odnosno kada mu se uputi poruka.

\* Objekat unutar svoje funkcije članice može pozivati funkciju članicu neke druge ili iste klase, odnosno može uputiti poruku drugom objektu. Objekat koji šalje poruku (poziva funkciju) naziva se *objekat-klijent*, a onaj koji je prima (čija je funkcija članica pozvana) je *objekat-server*.

\* Preporučuje se da se klase projektuju bez javnih podataka članova. Podaci članovi treba da budu privatni, osim ukoliko postoje jaki razlozi sa suprotnu odluku. Javne treba da budu samo funkcije članice koje predstavljaju operacije date apstrakcije koje su na raspolaganju korisnicima klase. Zaštićene su obično jednostavne operacije klase koje ne predstavljaju operacije interfejsa date klase, nego su ili proste funkcije za pristup do podataka članova, ili pomoćne funkcije koje služe za implementaciju javnih operacija jer se koriste na više mesta (nastale su algoritamskom dekompozicijom implementacije operacija i lokalizacijom zajedničkih delova). Ovakve jednostavnije operacije su pomoćne (engl. *helper functions*) i često su potrebne i izvedenim klasama, pa su zbog toga zaštićene.

\* Unutar funkcije članice klase, članovima objekta čija je funkcija pozvana pristupa se direktno, samo navođenjem njihovog imena.

\* Kontrola pristupa članovima nije stvar objekta, nego klase: jedan objekat neke klase iz svoje funkcije članice može da pristupi privatnim članovima drugog objekta iste klase. Kontrola pristupa članovima potpuno je odvojena od provere oblasti važenja: najpre se, na osnovu oblasti važenja, određuje entitet na koji se odnosi dato ime na mestu obraćanja u programu, a zatim se određuje da li se tom entitetu može pristupiti.

- \* Moguće je preklopiti (engl. *overload*) funkcije članice, uključujući i konstruktore.
- \* Deklaraciju klase koja nije definicija predstavlja samo deklaracija `class X;`. Pre potpune definicije klase (sa svim njenim članovima) mogu samo da se definišu pokazivači i reference na tu klasu, ali ne i objekti te klase, jer se njihova veličina ne zna.

## Pokazivač `this`

- \* Unutar svake funkcije članice postoji implicitni (podrazumevani, ugrađeni) lokalni objekat `this`. Tip ovog objekta je "konstantni pokazivač na klasu čija je funkcija članica" (ako je klasa `X`, `this` je tipa `X*const`). Ovaj pokazivač ukazuje na objekat čija je funkcija članica pozvana:

```
// definicija funkcije cAdd članice klase complex
complex complex::cAdd (complex c) {
    complex temp=this;
    // u temp se prepisuje objekat koji je prozvan
    temp.real+=c.real;
    temp.imag+=c.imag;
    return temp;
}
```

- \* Pristup članovima objekta čija je funkcija članica pozvana obavlja se neposredno; implicitno je to pristup preko pokazivača `this` i operatora `->`. Može se i eksplicitno pristupati članovima preko ovog pokazivača unutar funkcije članice:

```
// nova definicija funkcije cAdd članice klase complex
complex complex::cAdd (complex c) {
    complex temp;
    temp.real=this->real+c.real;
    temp.imag=this->imag+c.imag;
    return temp;
}
```

- \* Pokazivač `this` je, u stvari, skriveni argument funkcije članice. Poziv objekat.f() prevodilac prevodi u kôd koji ima semantiku kao `f(&objekat)`.

- \* Pokazivač `this` može da se iskoristi prilikom povezivanja (uspostavljanja relacije između) dva objekta. Na primer, neka klasa `X` sadrži objekat klase `Y`, pri čemu objekat klase `Y` treba da "zna" ko ga sadrži (ko mu je "nadređeni"). Veza se inicijalno može uspostaviti pomoću konstruktora:

```
class X {
public:
    X () : y(this) {...}
private:
    Y y;
};
```

```
class Y {
public:
    Y (X* theContainer) : myContainer(theContainer) {...}
private:
    X* myContainer;
};
```

## Zajednički članovi klase

### *Zajednički podaci članovi*

\* Pri kreiranju objekata klase, za svaki objekat se kreira poseban komplet podataka članova. Ipak, moguće je definisati podatke članove za koje postoji samo jedan primerak za celu klasu, tj. za sve objekte klase.

\* Ovakvi članovi nazivaju se *statičkim članovima*, i deklariraju se pomoću reči `static`:

```
class X {
public:
    //...
private:
    static int i;           // postoji samo jedan i za celu klasu
    int j;                 // svaki objekat ima svoj j
    //...
};
```

\* Statički član klase ima životni vek kao i globalni statički objekat: nastaje na početku programa i traje do kraja programa. Statički član klase ima sva svojstva globalnog statičkog objekta, osim oblasti važenja klase i kontrole pristupa.

\* Statički član mora da se inicijalizuje posebnom deklaracijom van deklaracije klase. Obraćanje ovakvom članu van klase vrši se preko operatora `::`. Za prethodni primer:

```
int X::i=5;
```

\* Statičkom članu može da se pristupi iz funkcije članice, ali i van funkcija članica, čak i pre formiranja ijednog objekta klase (jer statički član nastaje kao i globalni objekat), naravno uz poštovanje prava pristupa. Tada mu se pristupa preko operatora `::` (`X::i`).

\* Zajednički članovi se uglavnom koriste kada svi primeri jedne klase treba da dele neku zajedničku informaciju, npr. kada predstavljaju neku kolekciju, odnosno kada je potrebno imati ih "sve na okupu i pod kontrolom". Na primer, svi objekti neke klase se uvezuju u listu, a glava liste je zajednički član klase.

\* Zajednički članovi smanjuju potrebu za globalnim objektima i tako povećavaju čitljivost programa, jer je, za razliku od globalnih objekata, moguće ograničiti pristup do njih. Zajednički članovi logički pripadaju klasi i "upakovani" su u nju.

### *Zajedničke funkcije članice*

\* I funkcije članice mogu da se deklariraju kao zajedničke za celu klasu, dodavanjem reči `static` ispred deklaracije funkcije članice.

\* Statičke funkcije članice imaju sva svojstva globalnih funkcija, osim oblasti važenja i kontrole pristupa. One ne poseduju pokazivač `this` i ne mogu neposredno (bez pominjanja konkretnog objekta klase) koristiti nestatičke članove klase. Statičke funkcije mogu neposredno koristiti samo statičke članove te klase.

\* Statičke funkcije članice mogu se pozivati za konkretan objekat (što nema posebno značenje), ali i pre formiranja ijednog objekta klase, preko operatora `::`.

\* Primer:

```
class X {
    static int x;           // statički podatak član;
    int y;
public:
    static int f(X,X&);     // statička funkcija članica;
```

```
    int g();
};

int X::x=5;           // definicija statičkog podatka člana;

int X::f(X x1, X& x2){ // definicija statičke funkcije članice;
    int i=x;          // pristup statičkom članu X::x;
    int j=y;          // greška: X::y nije statički,
                      // pa mu se ne može pristupiti neposredno!
    int k=x1.y;        // ovo može;
    return x2.x;       // i ovo može,
                      // ali se izraz "x2" ne izračunava;
}

int X::g () {
    int i=x;          // nestatička funkcija članica može da
    int j=y;          // koristi i pojedinačne i zajedničke
    return j;         // članove; y je ovde this->y;
}

void main () {
    X xx;
    int p=X::f(xx,xx); // X::f može neposredno, bez objekta;
    int q=X::g();      // greška: za X::g mora konkretan objekat!
    xx.g();            // ovako može;
    p=xx.f(xx,xx);    // i ovako može,
                      // ali se izraz "xx" ne izračunava;
}
```

\* Statičke funkcije predstavljaju operacije klase, a ne svakog posebnog objekta. Pomoću njih se definišu neke opšte usluge klase (engl. *class utilities*), npr. tipično stvaranje novih, dinamičkih objekata te klase (operator `new` je implicitno definisan kao statička funkcija klase). Na primer, na sledeći način obezbeđuje se da se za datu klasu mogu kreirati samo dinamički objekti:

```
class X {
public:
    static X* create () { return new X; }
private:
    X(); // konstruktor je privatn
};
```

## Prijatelji klasa

\* Prijateljske funkcije (engl. *friend functions*) neke klase nisu članice te klase, ali imaju pristup do privatnih članova te klase. Te funkcije mogu da budu globalne funkcije ili članice drugih klasa.

\* Da bi se neka funkcija proglasila prijateljem klase, potrebno je bilo gde u deklaraciji te klase navesti deklaraciju te funkcije sa ključnom reči `friend` ispred. Prijateljska funkcija se definiše na uobičajen način:

```
class X {
public:
    void f(int ip) {i=ip;}
private:
    friend void g (int,X&); // prijateljska globalna funkcija
    friend void Y::h ();    // prijateljska članica druge klase
    int i;
};
```



```

void g (int k, X &x) {
    x.i=k;           // prijateljska funkcija može da pristupa
}                   // privatnim članovima klase

void main () {
    X x;
    x.f(5);          // postavljanje preko članice
    g(6,x);          // postavljanje preko prijatelja
}

```

\* Globalne funkcije koje predstavljaju usluge neke klase ili operacije nad tom klasom (najčešće su prijatelji te klase) nazivaju se *klasnim uslugama* (engl. *class utilities*).

\* "Prijateljstvo" se ne nasleđuje: ako je funkcija *f* prijatelj klasi *X*, a klasa *Y* izvedena (naslednik) iz klase *X*, funkcija *f* nije prijatelj klasi *Y*.

\* Ako je potrebno da sve funkcije članice klase *Y* budu prijateljske funkcije klasi *X*, onda se klasa *Y* deklarira kao prijateljska klasa (engl. *friend class*) klasi *X*. Tada sve funkcije članice klase *Y* mogu da pristupaju privatnim članovima klase *X*, ali obratno ne važi ("prijateljstvo" nije simetrična relacija):

```

class X {
    friend class Y;
    //...
};

```

\* "Prijateljstvo" nije ni tranzitivna relacija: ako je klasa *Y* prijatelj klasi *X*, a klasa *Z* prijatelj klasi *Y*, klasa *Z* nije automatski prijatelj klasi *X*, već to mora eksplicitno da se naglasi (ako je potrebno).

\* Prijateljske klase se uobičajeno koriste kada dve klase imaju tešnje međusobne veze. Pri tome je nepotrebno (i loše) "otkrivati" delove neke klase da bi oni bili dostupni drugoj klasi, jer će onda biti dostupni i ostalima (ruši se enkapsulacija). U tom slučaju se ove dve klase proglašavaju prijateljskim. Na primer, na sledeći način može se obezbediti da samo klasa *Creator* može da stvara objekte klase *X*:

```

class X {
public:
    ...
private:
    friend class Creator;
    X(); // konstruktor je dostupan samo klasi Creator
    ...
};

```

## Konstruktori

\* Funkcija članica koja nosi isto ime kao i klasa naziva se *konstruktor* (engl. *constructor*). Ova funkcija se uvek poziva prilikom nastanka objekta te klase.

\* Konstruktor nema tip koji vraća. Konstruktor može da ima argumente proizvoljnog tipa. Unutar konstruktora, članovima objekta pristupa se kao i u bilo kojoj drugoj funkciji članici.

\* Konstruktor se uvek implicitno poziva pri nastanku objekta klase, odnosno na početku životnog veka svakog objekta date klase.

\* Konstruktor, kao i svaka funkcija članica, može biti preklopljen (engl. *overloaded*). Konstruktor koji se može pozvati bez stvarnih argumenata (nema formalne argumente ili ima sve argumente sa podrazumevanim vrednostima) naziva se podrazumevanim konstruktorom.

\* Ukoliko u klasi nije eksplicitno deklarisan nijedan konstruktor, prevodilac implicitno generiše podrazumevani konstruktor koji je javni i koji vrši podrazumevanu inicijalizaciju podobjekta osnovne klase i objekata-članova pozivima njihovih podrazumevanih konstruktora, ako ih ima (bilo kao eksplicitno deklarisan ili implicitno generisan). Ako takvih konstruktora u odgovarajućim klasama nema, javlja se greška.

\* Konstruktor se poziva kada se kreira objekat klase. Na tom mestu je moguće navesti inicijalizatore, tj. stvarne argumente poziva konstruktora. Poziva se onaj konstruktor koji se najbolje slaže po broju i tipovima argumenata (pravila su ista kao i kod preklapanja funkcija):

```
class X {
public:
    X ();
    X (double);
    X (char*);
    //...
};

void main () {
    double d=3.4;
    char *p="Niz znakova";
    X a(d),           // poziva se X(double)
      b(p),           // poziva se X(char*)
      c;              // poziva se X()
    //...
}
```

\* Pri definisanju objekta `c` sa zahtevom da se poziva podrazumevani konstruktor klase `X`, ne treba navesti `X c()`; (jer je to deklaracija funkcije), već samo `X c`;

\* Pre izvršavanja samog tela konstruktora klase, pozivaju se konstruktori članova. Argumenti ovih poziva mogu da se navedu iza zaglavlja definicije (ne deklaracije) konstruktora klase, iza znaka : (dvotačka):

```
class YY {
public:
    YY (int j) {...}
    //...
};

class XX {
public:
    XX (int);
private:
    YY y;
    int i;
};

XX::XX (int k) : y(k+1) , i(k-1) {
    // y je inicijalizovan sa k+1, a i sa k-1
    // ... ostatak konstruktora
}
```

\* Prvo se pozivaju konstruktori članova, po redosledu deklarisanja u deklaraciji klase, pa se onda izvršava telo konstruktora klase.

\* Ovaj način ne samo da je moguć, već je i jedino ispravan: navođenje inicijalizatora u zaglavlju konstruktora predstavlja specifikaciju *inicijalizacije* članova (koji su ugrađenog tipa ili su objekti klase), što je različito od *operacije dodele* koja se može vršiti unutar tela konstruktora. Osim toga, kada član korisničkog tipa nema podrazumevani konstruktor, ovaj način je i jedini način inicijalizacije člana.

\* Konstruktor se može pozvati i eksplicitno u nekom izrazu. Tada nastaje privremeni objekat klase pozivom odgovarajućeg konstruktora sa navedenim argumentima. Isto se dešava ako se u inicijalizatoru eksplicitno navede poziv konstruktora:

```
void main () {
    complex c1(1,2.4),c2;
    c2=c1+complex(3.4,-1.5);    // privremeni objekat
    complex c3=complex(0.1,5);  // opet privremeni objekat
                                // koji se kopira u c3
}
```

\* Kada se kreira niz objekata neke klase, poziva se podrazumevani konstruktor za svaku komponentu niza ponaosob, po rastućem redosledu indeksa.

## Destruktor

\* Funkcija članica koja ima isto ime kao klasa, uz znak ~ ispred imena, naziva se *destruktor* (engl. *destructor*). Ova funkcija poziva se automatski, pri prestanku života objekta klase, za sve slučajeve životnog veka objekata:

```
class X {
public:
    ~X () { cout<<"Poziv destruktora klase X!\n"; }
}

void main () {
    X x;
    //...
} // ovde se poziva destruktor objekta x
```

\* Destruktor nema tip koji vraća i ne može imati argumente. Unutar destruktora, privatnim članovima pristupa se kao i u bilo kojoj drugoj funkciji članici. Svaka klasa može da ima najviše jedan destruktor.

\* Destruktor se implicitno poziva i pri uništavanju dinamičkog objekta pomoću operatora `delete`. Za niz, destruktor se poziva za svaki element ponaosob. Redosled poziva destruktora je, u svakom slučaju, obratan od redosleda poziva konstruktora.

\* Ako klasa nema eksplicitno deklarisan destruktor, prevodilac implicitno generiše podrazumevani destruktor koji je javni i koji vrši destrukciju podataka članova i podobjekta osnovne klase pozivom njihovih destruktora.

\* Destruktori se uglavnom koriste kada objekat treba da dealocira memoriju ili neke sistemske resurse koje je konstruktor alocirao; to je najčešće potrebno kada klasa sadrži članove koji su pokazivači na pridružene dinamičke objekte koji ekskluzivno pripadaju datom objektu, pa ih je potrebno uništiti prilikom uništavanja tog objekta.

## Pojam preklapanja operatora

\* Pretpostavimo da su nam u programu potrebni kompleksni brojevi i operacije nad njima. Treba nam struktura podataka koja će, pomoću osnovnih (u jezik ugrađenih) tipova, predstaviti strukturu kompleksnog broja, a takođe i funkcije koje će realizovati operacije nad kompleksnim brojevima.

\* Kada je potrebna struktura podataka za koju nisu bitni detalji implementacije, već operacije koje se nad njom vrše, sve ukazuje na klasu. Klasa upravo predstavlja apstraktni tip podataka za koji su definisane operacije.

\* U jeziku C++, operatori za korisničke tipove su specijalne funkcije koje nose ime `operator@`, gde je @ neki operator ugrađen u jezik:

```
class Complex {
public:
    Complex(double re, double im);           /* konstruktor */
    friend Complex operator+ (Complex, Complex); /* operator + */
    friend Complex operator- (Complex, Complex); /* operator - */
private:
    double real, imag;
};

Complex::Complex (double r, double i) : real(r), imag(i) {}

Complex operator+ (Complex c1, Complex c2) {
    Complex temp(0,0); /* privremena promenljiva tipa Complex */
    temp.real=c1.real+c2.real;
    temp.imag=c1.imag+c2.imag;
    return temp;
}

Complex operator- (Complex c1, Complex c2) {
    /* može i ovako: vratiti privremenu promenljivu
       koja se kreira konstruktorom sa odgovarajućim argumentima */
    return Complex(c1.real-c2.real, c1.imag-c2.imag);
}
```

\* Operatorske funkcije se mogu koristiti u izrazima kao i operatori nad ugrađenim tipovima. Izraz `t1@t2` se tumači kao `t1.operator@ (t2)` ili `operator@ (t1, t2)`:

```
Complex c1(3,5.4), c2(0,-5.4), c3(0,0);
c3=c1+c2;      /* poziva se operator+(c1,c2) */
c1=c2-c3;      /* poziva se operator-(c2,c3) */
```

## Operatori *new* i *delete*

\* Ponekad programer želi da preuzme kontrolu nad alokacijom dinamičkih objekata neke klase, a ne da je prepusti ugrađenom alokatoru.

\* Za ovakve potrebe mogu se preklopiti operatori `new` i `delete` za neku klasu. Operatorske funkcije `new` i `delete` moraju biti statičke (`static`) funkcije članice, jer se one pozivaju pre nego što je objekat stvarno inicijalizovan, odnosno pošto je uništen.

\* Ako je korisnik definisao ove operatorske funkcije za neku klasu, one će se pozivati kad god nastaje dinamički objekat te klase operatorom `new`, odnosno kada se takav objekat dealocira operatorom `delete`.

\* Unutar tela ovih operatorskih funkcija ne treba eksplicitno pozivati konstruktor, odnosno destruktor. Konstruktor se implicitno poziva posle operatorske funkcije `new`, a destruktor se implicitno poziva pre operatorske funkcije `delete`. Ove operatorske funkcije služe samo da obezbede prostor za smeštanje objekta i da ga posle oslobode, a ne da od "presnih" bitova naprave objekat (što rade konstruktori), odnosno pretvore ga u "presne bitove" (što radi destruktor). Operator `new` treba da vrati pokazivač na alocirani prostor.

\* Ove operatorske funkcije deklarišu se na sledeći način:

```
void* X::operator new (size_t velicina);
void X::operator delete (void* pokazivac);
```

Tip `size_t` je celobrojni tip definisan u `<stdlib.h>` i služi za izražavanje veličina objekata. Argument `velicina` daje veličinu potrebnog prostora koji treba alocirati za objekat. Argument `pokazivac` je pokazivač na prostor koji treba osloboditi.

\* Podrazumevani (ugrađeni) operatori `new` i `delete` mogu da se pozivaju unutar tela redefinisanih operatorskih funkcija ili eksplicitno, preko operatora `::`, ili implicitno, kada se dinamički kreiraju objekti tipova za koje su redefinisani ovi operatori.

\* Primer:

```
#include <stdlib.h>

class XX {
public:
    void* operator new (size_t sz)
    { return new char[sz]; } // koristi se ugrađeni new
    void operator delete (void *p)
    { delete [] p; } // koristi se ugrađeni delete
    //...
};
```

## Primeri struktura pogodnih za RT implementacije

### *Kolekcija implementirana kao dvostruko ulančana dinamička lista*

#### *Koncepcija*

\* Za mnoge primene u RT i drugim sistemima potrebna je struktura koja predstavlja kolekciju pokazivača na objekte nekog tipa. *Kolekcija* je linearna struktura elemenata u koju se elementi mogu ubacivati, iz koje se mogu izbacivati, i koji se mogu redom obilaziti.

\* Jedna jednostavna implementacija oslanja se na dinamičku, dvostruko ulančanu listu, čiji su elementi strukture koje sadrže veze (pokazivače) prema susednim elementima i sam sadržaj (pokazivač na objekat u kolekciji). Ove strukture se dinamički alociraju i dealociraju prilikom umetanja i izbacivanja elemenata.

#### *Implementacija*

```
// Project: Real-Time Programming
// Subject: Data Structures
// Module: Collection
// File: collection.h
// Date: October 2002
// Author: Dragan Milicev
// Contents:
//      Class: Collection
//      CollectionIterator

#ifndef _COLLECTION_
#define _COLLECTION_

////////////////////////////////////
// class Collection
////////////////////////////////////

class Object;
class CollectionElement;
class CollectionIterator;
```

```
class Collection {
public:

    Collection ();
    ~Collection ();

    void      append (Object*);
    void      insert (Object*, int at=0);
    void      remove (Object*);
    Object*   remove (int at=0);
    Object*   removeFirst() { return remove(0); }
    Object*   removeLast()  { return remove(size()-1); }
    void      clear  ();

    int       isEmpty ()      { return sz==0; }
    int       size   ()      { return sz; }
    Object*   first   ();
    Object*   last    ();
    Object*   itemAt  (int at);
    int       location(Object*);

    CollectionIterator* createIterator ();
    CollectionIterator* getIterator ()   { return internalIterator; }

protected:

    void remove (CollectionElement*);

private:

    friend class CollectionIterator;
    CollectionElement* head;
    CollectionElement* tail;
    int sz;

    CollectionIterator* internalIterator;

};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// class CollectionIterator
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

class CollectionIterator {
public:

    CollectionIterator (Collection* c) : col(c), cur(0) { reset(); }

    void      reset() { if (col!=0) cur=col->head; }
    int       next  ();

    int       isDone() { return cur==0; }
    Object*   currentItem();

private:

    Collection* col;
    CollectionElement* cur;
};
```

```

};

#endif

// Project: Real-Time Programming
// Subject: Data Structures
// Module: Collection
// File: collection.cpp
// Date: October 2002
// Author: Dragan Milicev
// Contents:
//      Class: Collection
//      CollectionIterator

#include "collection.h"

/////////////////////////////////////////////////////////////////
// class CollectionElement
/////////////////////////////////////////////////////////////////

class CollectionElement {
public:
    Object* cont;
    CollectionElement *prev, *next;

    CollectionElement (Object*);
    CollectionElement (Object*, CollectionElement* next);
    CollectionElement (Object*, CollectionElement* prev, CollectionElement*
next);
};

inline CollectionElement::CollectionElement (Object* e)
    : cont(e), prev(0), next(0) {}

inline CollectionElement::CollectionElement (Object* e, CollectionElement*
n)
    : cont(e), prev(0), next(n) {
    if (n!=0) n->prev=this;
}

inline CollectionElement::CollectionElement (Object* e, CollectionElement*
p, CollectionElement* n)
    : cont(e), prev(p), next(n) {
    if (n!=0) n->prev=this;
    if (p!=0) p->next=this;
}

/////////////////////////////////////////////////////////////////
// class Collection
/////////////////////////////////////////////////////////////////

Collection::Collection ()
    : head(0), tail(0), sz(0),

```

```
    internalIterator(new CollectionIterator(this))
{}

Collection::~~Collection () {
    clear();
    delete internalIterator;
}

void Collection::remove (CollectionElement* e) {
    if (e==0) return;
    if (e->next!=0) e->next->prev=e->prev;
    else tail=e->prev;
    if (e->prev!=0) e->prev->next=e->next;
    else head=e->next;
    if (internalIterator && internalIterator->currentItem()==e->cont)
        internalIterator->next();
    delete e;
    sz--;
}

void Collection::append (Object* e) {
    if (head==0) head=tail=new CollectionElement(e);
    else tail=new CollectionElement(e,tail,0);
    sz++;
}

void Collection::insert (Object* e, int at) {
    if (at<0 || at>size()) return;
    if (at==0) {
        head=new CollectionElement(e,head);
        if (tail==0) tail=head;
        sz++;
        return;
    }
    if (at==size()) {
        append(e);
        return;
    }
    int i=0;
    for (CollectionElement* cur=head; i<at; cur=cur->next, i++);
    new CollectionElement(e,cur->prev,cur);
    sz++;
}

void Collection::remove (Object* e) {
    if (tail && tail->cont==e) {
        remove(tail);
        return;
    }
    for (CollectionElement* cur=head; cur!=0; cur=cur->next)
        if (cur->cont==e) remove(cur);
}

Object* Collection::remove (int at) {
    Object* ret = 0;
    if (at<0 || at>=size()) return 0;
    if (at==0) {
        ret = head->cont;
        remove(head);
    }
}
```



```

        return ret;
    }
    if (at==size()-1) {
        ret = tail->cont;
        remove(tail);
        return ret;
    }
    int i=0;
    for (CollectionElement* cur=head; i<at; cur=cur->next, i++);
    ret = cur->cont;
    remove(cur);
    return ret;
}

void Collection::clear () {
    for (CollectionElement* cur=head, *temp=0; cur!=0; cur=temp) {
        temp=cur->next;
        delete cur;
    }
    head=0;
    tail=0;
    sz=0;
    if (internalIterator) internalIterator->reset();
}

Object* Collection::first () {
    if (head==0) return 0;
    else return head->cont;
}

Object* Collection::last () {
    if (tail==0) return 0;
    else return tail->cont;
}

Object* Collection::itemAt (int at) {
    if (at<0 || at>=size()) return 0;
    int i=0;
    for (CollectionElement* cur=head; i<at; cur=cur->next, i++);
    return cur->cont;
}

int Collection::location (Object* e) {
    int i=0;
    for (CollectionElement* cur=head; cur!=0; cur=cur->next, i++)
        if (cur->cont==e) return i;
    return -1;
}

CollectionIterator* Collection::createIterator () {
    return new CollectionIterator(this);
}

```

```

////////////////////////////////////
// class CollectionIterator
////////////////////////////////////

```

```
int CollectionIterator::next () {
    if (cur!=0) cur=cur->next;
    return !isDone();
}

Object* CollectionIterator::currentItem () {
    return cur?cur->cont:0;
}
```

### *Primer upotrebe*

```
#include "collection.h"
#include <iostream.h>

class Object {
    //...
};

class X : public Object {
public:
    X(int ii) : i(ii) {}
    int i;
    //...
};

void main () {
    X* x1 = new X(1);
    X* x2 = new X(2);
    X* x3 = new X(3);
    X* x4 = new X(4);
    X* x5 = new X(5);

    Collection* coll = new Collection;
    coll->append(x1);
    coll->append(x2);
    coll->insert(x3);
    coll->insert(x4,2);
    X* x = (X*)coll->removeFirst();
    coll->append(x);
    coll->insert(x5,3);

    CollectionIterator* it = coll->getIterator();
    for (it->reset(); !it->isDone(); it->next()) {
        X* x = (X*)it->currentItem();
        cout<<x->i<<" ";
    }
    cout<<"\n";

    Collection* col2 = new Collection;
    col2->append(x1);
    col2->append(x2);
    col2->append(x3);
    col2->append(x4);
    col2->append(x5);
    it = col2->getIterator();
    for (it->reset(); !it->isDone(); it->next()) {
        X* x = (X*)it->currentItem();
        cout<<x->i<<" ";
    }
}
```

```

    cout<<"\n";

    it = coll->createIterator();
    for (it->reset(); !it->isDone(); it->next()) {
        X* x = (X*)it->currentItem();
        cout<<x->i<<" ";
        CollectionIterator* it = coll->createIterator();
        for (it->reset(); !it->isDone(); it->next()) {
            X* x = (X*)it->currentItem();
            cout<<x->i<<" ";
        }
        delete it;
        cout<<"\n";
    }
    delete it;
    cout<<"\n";

    coll->clear();
    delete coll;
    col2->clear();
    delete col2;
    delete x1;
    delete x2;
    delete x3;
    delete x4;
    delete x5;
}

```

### Analiza kompleksnosti

\* Neka je kompleksnost algoritma alokacije prostora ugrađenog alokatora (ugrađene operatorske funkcije `new`)  $C_a$ , a algoritma dealokacije  $C_d$ . Neka je  $n$  veličina kolekcije (broj elemenata u kolekciji). Tada je kompleksnost najznačajnijih operacija prikazane implementacije sledeća:

Operacija	Kompleksnost
<code>Collection::append(Object*)</code>	$C_a$
<code>Collection::insert(Object*)</code>	$C_a$
<code>Collection::insert(Object*, int)</code>	$C_a + O(n)$
<code>Collection::remove(Object*)</code>	$C_d + O(n)$
<code>Collection::remove(int)</code>	$C_d + O(n)$
<code>Collection::removeFirst()</code>	$C_d$
<code>Collection::removeLast()</code>	$C_d$
<code>Collection::clear()</code>	$C_d \cdot O(n)$
<code>Collection::isEmpty()</code>	$O(1)$
<code>Collection::size()</code>	$O(1)$
<code>Collection::first()</code>	$O(1)$
<code>Collection::last()</code>	$O(1)$
<code>Collection::itemAt(int)</code>	$O(n)$
<code>Collection::location(Object*)</code>	$O(n)$
<code>CollectionIterator::reset()</code>	$O(1)$
<code>CollectionIterator::next()</code>	$O(1)$
<code>CollectionIterator::isDone()</code>	$O(1)$
<code>Collection::currentItem()</code>	$O(1)$

\* Prema tome, najkritičnije i najčešće korišćene operacije za umetanje i izbacivanje elemenata jako zavise od kompleksnosti algoritma ugrađenog alokatora i dealokatora. U zavisnosti od implementacije struktura koje vode računa o zauzetom i slobodnom prostoru u dinamičkoj memoriji, ova kompleksnost može različita. Tipično je to  $O(k)$ , gde je  $k$  broj zauzetih i/ili slobodnih segmenata dinamičke memorije, ili u boljem slučaju  $O(\log k)$ . Što je još gore, operacija `Collection::remove(Object*)` ima i dodatnu kompleksnost  $O(n)$ , zbog sekvencijalne pretrage elementa koji se izbacuje.

\* Zbog ovakve orijentacije na ugrađene alokatore, čija se kompleksnost teško može proceniti i kontrolisati, kao i na sekvencijalnu pretragu kod izbacivanja, ova implementacija nije pogodna za RT sisteme.

### ***Kolekcija kao ulančana lista sa vezama ugrađenim u objekte***

#### *Koncepcija*

\* Problem sa prethodnim rešenjem je što se za strukturu veza između elemenata kolekcije alokira poseban prostor, na šta se troši vreme pri umetanju i izbacivanju elemenata. Osim toga, objekat koji je element kolekcije nema nikakvu vezu prema toj strukturi, pa je kod izbacivanja elementa (zadatog kao pokazivač na dati objekat) potrebno vršiti sekvencijalnu pretragu kolekcije.

\* Rešenje koje eliminiše ove probleme oslanja se na to da struktura veza bude ugrađena u same objekte koji se smeštaju u kolekciju. Zbog toga nema potrebe za alokaciju i dealokaciju struktura za veze.

\* Potencijalni problem sa ovakvim pristupom je da može doći do greške ukoliko se objekat koji je već element neke kolekcije pokuša ubaciti u drugu kolekciju, korišćenjem iste strukture za veze, čime dolazi do korupcije prve kolekcije. Zato je u ovu implementaciju ugrađena zaštita od ovakve pogrešne upotrebe.

#### *Implementacija*

```
// Project: Real-Time Programming
// Subject: Data Structures
// Module: Collection
// File: collect.h
// Date: October 2002
// Author: Dragan Milicev
// Contents:
//      Class:
//          CollectionElement
//          Collection
//          CollectionIterator

#ifndef _COLLECTION_
#define _COLLECTION_

////////////////////////////////////
// class CollectionElement
////////////////////////////////////

class Object;
class Collection;

class CollectionElement {
public:
    CollectionElement (Object* holder);
```

```

Object*      getHolder()      { return holder; }
Collection*  getContainer () { return container; }

private:

    friend class Collection;
    friend class CollectionIterator;
    void set (CollectionElement* prev, CollectionElement* next);
    void setContainer (Collection* col) { container = col; }

    CollectionElement *prev, *next;

    Collection* container;
    Object* holder;

};

inline CollectionElement::CollectionElement (Object* h)
    : container(0), holder(h), prev(0), next(0) {}

inline void CollectionElement::set (CollectionElement* p,
CollectionElement* n) {
    prev = p;
    next = n;
    if (n!=0) n->prev=this;
    if (p!=0) p->next=this;
}

////////////////////////////////////
// class Collection
////////////////////////////////////

class CollectionIterator;

class Collection {
public:

    Collection ();
    ~Collection ();

    void      append (CollectionElement*);
    void      insert (CollectionElement*, int at=0);
    void      insertBefore (CollectionElement* newElem, CollectionElement*
beforeThis);
    void      insertAfter  (CollectionElement* newElem, CollectionElement*
afterThis);

    void      remove (CollectionElement*);
    Object*   remove (int at=0);
    Object*   removeFirst() { return remove(0); }
    Object*   removeLast()  { return remove(size()-1); }

    void      clear  ();

    int       isEmpty ()      { return sz==0; }
    int       size   ()      { return sz; }

```

```
Object* first    () { return head->getHolder(); }
Object* last     () { return tail->getHolder(); }
Object* itemAt   (int at);

int            location(CollectionElement*);

CollectionIterator* createIterator ();
CollectionIterator* getIterator  () { return internalIterator; }

private:

    friend class CollectionIterator;
    CollectionElement* head;
    CollectionElement* tail;
    int sz;

    CollectionIterator* internalIterator;

};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// class CollectionIterator
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

class CollectionIterator {
public:

    CollectionIterator (Collection* c) : col(c), cur(0) { reset(); }

    void    reset()  { if (col!=0) cur=col->head; }
    int     next ()  { if (cur!=0) cur=cur->next; return !isDone(); }

    int     isDone() { return cur==0; }

    Object*    currentItem()    { return cur?cur->getHolder():0; }
    CollectionElement* currentElement() { return cur; }

private:

    Collection* col;
    CollectionElement* cur;

};

#endif

// Project:  Real-Time Programming
// Subject:  Data Structures
// Module:   Collection
// File:     collect.cpp
// Date:     October 2002
// Author:   Dragan Milicev
// Contents:
//          Class: CollectionElement
//                  Collection
//                  CollectionIterator
```

```

#include "collect.h"

////////////////////////////////////
// class Collection
////////////////////////////////////

Collection::Collection ()
    : head(0), tail(0), sz(0),
    internalIterator(new CollectionIterator(this))
{}

Collection::~~Collection () {
    clear();
    delete internalIterator;
}

void Collection::append (CollectionElement* e) {
    if (e==0 || e->getContainer()!=0) return;
    if (head==0) {
        e->set(0,0);
        head=tail=e;
    }
    else {
        e->set(tail,0);
        tail=e;
    }
    e->setContainer(this);
    sz++;
}

void Collection::insert (CollectionElement* e, int at) {
    if (e==0 || e->getContainer()!=0 || at<0 || at>size()) return;
    if (at==0) {
        e->set(0,head);
        e->setContainer(this);
        head=e;
        if (tail==0) tail=head;
        sz++;
        return;
    }
    if (at==size()) {
        append(e);
        return;
    }
    int i=0;
    for (CollectionElement* cur=head; i<at; cur=cur->next, i++);
    e->set(cur->prev,cur);
    e->setContainer(this);
    sz++;
}

void Collection::insertBefore (CollectionElement* newElem,
CollectionElement* beforeThis) {
    if (newElem==0 || newElem->getContainer()!=0) return;
    if (beforeThis==0) { append(newElem); return; }
    if (beforeThis->prev==0) { insert(newElem); return; }
    newElem->set(beforeThis->prev,beforeThis);
    newElem->setContainer(this);
    sz++;
}

```

```
}

void Collection::insertAfter (CollectionElement* newElem,
CollectionElement* afterThis) {
    if (newElem==0 || newElem->getContainer()!=0) return;
    if (afterThis==0) { insert(newElem); return; }
    if (afterThis->next==0) { append(newElem); return; }
    newElem->set(afterThis,afterThis->next);
    newElem->setContainer(this);
    sz++;
}

void Collection::remove (CollectionElement* e) {
    if (e==0 || e->getContainer()!=this) return;
    if (e->next!=0) e->next->prev=e->prev;
    else tail=e->prev;
    if (e->prev!=0) e->prev->next=e->next;
    else head=e->next;
    e->set(0,0);
    e->setContainer(0);
    if (internalIterator && internalIterator->currentItem()==e->getHolder())
        internalIterator->next();
    sz--;
}

Object* Collection::remove (int at) {
    CollectionElement* ret = 0;
    if (at<0 || at>=size()) return 0;
    if (at==0) {
        ret = head;
        remove(head);
        return ret?ret->getHolder():0;
    }
    if (at==size()-1) {
        ret = tail;
        remove(tail);
        return ret?ret->getHolder():0;
    }
    int i=0;
    for (CollectionElement* cur=head; i<at; cur=cur->next, i++);
    ret = cur;
    remove(cur);
    return ret?ret->getHolder():0;
}

void Collection::clear () {
    for (CollectionElement* cur=head, *temp=0; cur!=0; cur=temp) {
        temp=cur->next;
        cur->set(0,0);
        cur->setContainer(0);
    }
    head=0;
    tail=0;
    sz=0;
    if (internalIterator) internalIterator->reset();
}

Object* Collection::itemAt (int at) {
    if (at<0 || at>=size()) return 0;
    int i=0;
```



```

    for (CollectionElement* cur=head; i<at; cur=cur->next, i++);
    return cur?cur->getHolder():0;
}

int Collection::location (CollectionElement* e) {
    if (e==0 || e->getContainer()!=this) return -1;
    int i=0;
    for (CollectionElement* cur=head; cur!=0; cur=cur->next, i++)
        if (cur==e) return i;
    return -1;
}

CollectionIterator* Collection::createIterator () {
    return new CollectionIterator(this);
}

```

### *Primer upotrebe*

```

#include "collect.h"
#include <iostream.h>

class Object {
    //...
};

class X : public Object {
public:
    X(int ii) : i(ii), ceForC1(this), ceForC2(this) {}
    int i;

    CollectionElement ceForC1;
    CollectionElement ceForC2;
    //...
};

void main () {
    X* x1 = new X(1);
    X* x2 = new X(2);
    X* x3 = new X(3);
    X* x4 = new X(4);
    X* x5 = new X(5);

    Collection* coll = new Collection;
    coll->append(&x1->ceForC1);
    coll->append(&x2->ceForC1);
    coll->insert(&x3->ceForC1);
    coll->insert(&x4->ceForC1,2);
    X* x = (X*)coll->removeFirst();
    coll->append(&x->ceForC1);
    coll->insert(&x5->ceForC1,3);

    CollectionIterator* it = coll->getIterator();
    for (it->reset(); !it->isDone(); it->next()) {
        X* x = (X*)it->currentItem();
        cout<<x->i<<" ";
    }
    cout<<"\n";

    Collection* col2 = new Collection;
    col2->append(&x1->ceForC2);
    col2->append(&x2->ceForC2);
}

```

```

col2->append(&x3->ceForC2);
col2->append(&x4->ceForC2);
col2->append(&x5->ceForC2);
col2->append(&x3->ceForC1); // Tolerant Error
it = col2->getIterator();
for (it->reset(); !it->isDone(); it->next()) {
    X* x = (X*)it->currentItem();
    cout<<x->i<<" ";
}
cout<<"\n";

it = coll->createIterator();
for (it->reset(); !it->isDone(); it->next()) {
    X* x = (X*)it->currentItem();
    cout<<x->i<<" ";
    CollectionIterator* it = coll->createIterator();
    for (it->reset(); !it->isDone(); it->next()) {
        X* x = (X*)it->currentItem();
        cout<<x->i<<" ";
    }
    delete it;
    cout<<"\n";
}
delete it;
cout<<"\n";

coll->clear();
delete coll;
col2->clear();
delete col2;
delete x1;
delete x2;
delete x3;
delete x4;
delete x5;
}

```

### Analiza kompleksnosti

\* Kompleksnost ove implementacije više ne zavisi od kompleksnosti algoritma alokatora i dealokatora. Kompleksnost najznačajnijih operacija je sada značajno smanjena, naročito za najčešće operacije stavljanja i izbacivanja elementa, i to na početak ili kraj:

Operacija	Kompleksnost
Collection::append(CollectionElement*)	$O(1)$
Collection::insert(CollectionElement*)	$O(1)$
Collection::insert(CollectionElement*,int)	$O(n)$
Collection::insertBefore(...)	$O(1)$
Collection::insertAfter(...)	$O(1)$
Collection::remove(CollectionElement*)	$O(1)$
Collection::remove(int)	$O(n)$
Collection::removeFirst()	$O(1)$
Collection::removeLast()	$O(1)$
Collection::clear()	$O(n)$
Collection::isEmpty()	$O(1)$
Collection::size()	$O(1)$
Collection::first()	$O(1)$
Collection::last()	$O(1)$

Collection::itemAt(int)	$O(n)$
Collection::location(CollectionElement*)	$O(n)$
CollectionIterator::reset()	$O(1)$
CollectionIterator::next()	$O(1)$
CollectionIterator::isDone()	$O(1)$
Collection::currentItem()	$O(1)$
Collection::currentElement()	$O(1)$

## ***FIFO Red***

### *Koncepcija*

\* FIFO (*First-In First-Out*) red (engl. *queue*) je struktura u koju se elementi mogu umetati i vaditi, ali sa sledećim protokolom: operacija vađenja elementa uvek vraća element koji je najdavnije stavljen u red, tako da se elementi vade po redosledu stavljanja.

### *Implementacija*

\* Implementacija se u potpunosti oslanja na realizovanu kolekciju:

```
// Project: Real-Time Programming
// Subject: Data Structures
// Module: FIFO Queue
// File: queue.h
// Date: October 2002
// Author: Dragan Milicev
// Contents:
// Class:
// Queue

#ifndef _QUEUE_
#define _QUEUE_

#include "collect.h"

////////////////////////////////////
// class Queue
////////////////////////////////////

class Queue {
public:

    void put(CollectionElement* e) { col.append(e); }
    Object* get () { return col.removeFirst(); }
    void clear () { col.clear(); }

    int isEmpty () { return col.isEmpty(); }
    int isFull () { return 0; }
    int size () { return col.size(); }

    Object* first () { return col.first(); }
    Object* last () { return col.last(); }
    Object* itemAt (int at) { return col.itemAt(at); }

    int location(CollectionElement* e) { return col.location(e); }

    CollectionIterator* createIterator () { return col.createIterator(); }
    CollectionIterator* getIterator () { return col.getIterator(); }
```

```
private:
    Collection col;
};

#endif
```

### *Primer upotrebe*

\* Najčešće upotrebljavane operacije ove apstrakcije su operacije stavljanja (`put()`) i uzimanja elementa (`get()`). Data klasa koristi se slično kao i ranije realizovana kolekcija.

### *Analiza kompleksnosti*

\* Kako se sve operacije ove klase svode na odgovarajuće operacije klase `Collection`, time je i njihova kompleksnost identična. Treba uočiti da je kompleksnost najčešće korišćenih operacija `put()` i `get()` jednaka  $O(1)$ .

### *Red sa prioritetom*

#### *Koncepcija*

\* *Red sa prioritetom* (engl. *priority queue*) je linearna struktura u koju se elementi mogu smeštati i izbacivati, ali pri čemu elementi imaju svoje *prioritete*. Prioritet je veličina koja se može upoređivati (tj. za koju je definisana neka relacija totalnog uređenja, na primer prirodan broj).

\* Najkritičnije operacije ove strukture su operacije smeštanja i izbacivanja elementa i operacija vraćanja (bez izbacivanja) elementa koji ima trenutno najviši prioritet.

#### *Implementacija*

\* Implementacija se oslanja na postojeću implementaciju kolekcije.

\* Da bi kompleksnost obe operacije vraćanja trenutno najprioritetnijeg elementa i operacije umetanja elementa bila manje od  $O(n)$ , potrebno je da nijedna od njih ne uključuje linearnu pretragu po eventualno uređenoj listi. Zbog toga ova implementacija sadrži pokazivač na trenutno najprioritetniji element, koji se ažurira prilikom promene strukture reda ili promene prioriteta nekog elementa.

```
// Project: Real-Time Programming
// Subject: Data Structures
// Module: Priority Queue
// File: pqueue.h
// Date: October 2002
// Author: Dragan Milicev
// Contents:
//     Class:
//         PriorityElement
//         PriorityQueue
//     Type:
//         Priority

#ifndef _PQUEUE_
#define _PQUEUE_

#include "collect.h"
```

```

////////////////////////////////////
// class PriorityElement
////////////////////////////////////

typedef unsigned int Priority;
const Priority MinPri = 0;

class PriorityQueue;

class PriorityElement : public CollectionElement {
public:
    PriorityElement (Object* holder, Priority p = 0)
        : CollectionElement(holder), pri(p), container(0) {}

    Priority getPriority ()           { return pri; }
    void      setPriority (Priority newPri);

    PriorityQueue* getContainer ()    { return container; }

private:
    Priority pri;

    friend class PriorityQueue;
    void setContainer (PriorityQueue* c) { container = c; }
    PriorityQueue* container;
};

////////////////////////////////////
// class PriorityQueue
////////////////////////////////////

class PriorityQueue {
public:
    PriorityQueue () : col(), highest(0) {}

    Object* first ()    { return highest?highest->getHolder():0; }

    void add (PriorityElement*);
    void remove(PriorityElement*);
    void clear ();

    void notifyPriorityChange (PriorityElement*);

    int  isEmpty () { return col.isEmpty(); }
    int  size      () { return col.size(); }

    CollectionIterator* createIterator ()    { return col.createIterator(); }
    CollectionIterator* getIterator  ()      { return col.getIterator(); }

private:
    Collection col;
    PriorityElement* highest;

```

```
};

#endif

// Project: Real-Time Programming
// Subject: Data Structures
// Module: Priority Queue
// File: pqueue.cpp
// Date: October 2002
// Author: Dragan Milicev
// Contents:
//      Class:
//      PriorityElement
//      PriorityQueue

#include "pqueue.h"

/////////////////////////////////////////////////////////////////
// class PriorityElement
/////////////////////////////////////////////////////////////////

void PriorityElement::setPriority (Priority newPri) {
    if (pri==newPri) return;
    pri = newPri;
    if (container!=0) container->notifyPriorityChange(this);
}

/////////////////////////////////////////////////////////////////
// class PriorityQueue
/////////////////////////////////////////////////////////////////

void PriorityQueue::add (PriorityElement* e) {
    if (e==0 || e->getContainer()!=0) return;
    col.append(e);
    e->setContainer(this);
    notifyPriorityChange(e);
}

void PriorityQueue::remove(PriorityElement* e) {
    if (e==0 || e->getContainer()!=this) return;
    col.remove(e);
    e->setContainer(0);
    if (highest!=e) return;

    Priority maxPri = MinPri;
    highest = 0;
    CollectionIterator* it = getIterator();
    for (it->reset(); !it->isDone(); it->next()) {
        PriorityElement* pe = (PriorityElement*)it->currentElement();
        if (pe->getPriority()>=maxPri) {
            maxPri = pe->getPriority();
            highest = pe;
        }
    }
}

void PriorityQueue::clear () {
    CollectionIterator* it = getIterator();
```

```

        for (it->reset(); !it->isDone(); it->next()) {
            PriorityElement* pe = (PriorityElement*)it->currentElement();
            pe->setContainer(0);
        }
        col.clear();
        highest = 0;
    }

void PriorityQueue::notifyPriorityChange (PriorityElement* e) {
    if (e==0 || e->getContainer()!=this) return;
    if (highest==0 || highest->getPriority()<e->getPriority()) {
        highest = e;
        return;
    }
    if (highest==e) {
        Priority maxPri = e->getPriority();
        CollectionIterator* it = getIterator();
        for (it->reset(); !it->isDone(); it->next()) {
            PriorityElement* pe = (PriorityElement*)it->
>currentElement();
            if (pe->getPriority()>maxPri) {
                maxPri = pe->getPriority();
                highest = pe;
            }
        }
        return;
    }
}

```

### *Primer upotrebe*

```

#include "pqueue.h"
#include <iostream.h>

class Object {
    //...
};

class X : public Object {
public:

    X(int ID, Priority pri) : id(ID), peForPQ(this) {
        peForPQ.setPriority(pri); }
    int id;

    PriorityElement* getPriorityElement () { return &peForPQ; }

    Priority getPriority () { return peForPQ.getPriority(); }
}

    void setPriority (Priority pri) { peForPQ.setPriority(pri); }

private:

    PriorityElement peForPQ;

    //...
};

void main () {
    X* x1 = new X(1,1);
    X* x2 = new X(2,2);
    X* x3 = new X(3,3);
}

```

```

X* x4 = new X(4,4);
X* x5 = new X(5,5);

PriorityQueue* pq = new PriorityQueue;
pq->add(x1->getPriorityElement());
pq->add(x3->getPriorityElement());
pq->add(x4->getPriorityElement());
pq->add(x2->getPriorityElement());
pq->add(x5->getPriorityElement());

X* top = 0;

top = (X*)pq->first();
cout<<top->getPriority()<<" "<<top->id<<"\n";

x1->setPriority(6);
top = (X*)pq->first();
cout<<top->getPriority()<<" "<<top->id<<"\n";

x5->setPriority(3);
x1->setPriority(0);
top = (X*)pq->first();
cout<<top->getPriority()<<" "<<top->id<<"\n";

pq->remove(top->getPriorityElement());
top = (X*)pq->first();
cout<<top->getPriority()<<" "<<top->id<<"\n";

CollectionIterator* it = pq->getIterator();
for (it->reset(); !it->isDone(); it->next()) {
    X* x = (X*)it->currentItem();
    cout<<x->id<<" ";
}
cout<<"\n";

pq->clear();
delete pq;
delete x1;
delete x2;
delete x3;
delete x4;
delete x5;
}

```

### Analiza kompleksnosti

Operacija	Kompleksnost
PriorityElement::getPriority()	$O(1)$
PriorityElement::setPriority()	Najčešći slučaj: $O(1)$ Najgori slučaj: $O(n)$
PriorityQueue::first()	$O(1)$
PriorityQueue::add(PriorityElement*)	$O(1)$
PriorityQueue::remove(PriorityElement*)	Najgori slučaj: $O(n)$
PriorityQueue::clear()	$O(n)$
PriorityQueue::notifyPriorityChange(PriorityElement*)	Najgori slučaj: $O(n)$
PriorityQueue::isEmpty()	$O(1)$
PriorityQueue::size()	$O(1)$



## Efikasna alokacija memorije

### Koncepcija

\* Intenzivno korišćenje dinamičkih objekata u OOP podrazumeva oslanjanje na algoritme alokacije i dealokacije ugrađenih menadžera dinamičke memorije. Kao što je već rečeno, kompleksnost ovih algoritama je tipično  $O(n)$  i  $O(1)$ , odnosno  $O(\log n)$  za obe operacije u najboljem slučaju. Zbog toga kreiranje i uništavanje dinamičkih objekata klasa može da bude režijski jako skupo u RT programima.

\* Osim toga, dugotrajno korišćenje (tipično za RT sisteme) jedinstvene dinamičke memorije za objekte svih klasa može da dovede do fragmentacije dinamičke memorije.

\* Ideja rešenja oba problema zasniva se na tome da se dealocirani objekti ne vraćaju na korišćenje ugrađenom menadžeru, već se čuvaju u listi pridruženoj datoj klasi kako bi se njihov prostor ponovo koristio za nove objekte iste klase ("reciklaža"). Za neku klasu *X* za koju se želi brza alokacija i dealokacija objekata, treba obezbediti jedan objekat klase *RecycleBin*. Ovaj objekat funkcioniše tako što u svojoj listi čuva sve dealocirane objekte date klase. Kada se traži alokacija novog objekta, *RecycleBin* prvo vadi iz svoje liste već spreman reciklirani objekat, tako da je alokacija u tom slučaju veoma brza. Ako je lista prazna, tj. ako nema prethodno recikliranih objekata date klase, *RecycleBin* pri alokaciji koristi ugrađeni alokator *new*. Pri dealokaciji, objekat se ne oslobađa već se reciklira, tj. upisuje se u listu klase *RecycleBin*.

### Implementacija

```
// Project: Real-Time Programming
// Subject: Data Structures
// Module: Recycle Bin
// File: recycle.h
// Date: October 2002
// Author: Dragan Milicev
// Contents:
//      Class:
//      RecycleBin
//      Macro:
//      RECYCLE_DEC(X)
//      RECYCLE_DEF(X)
//      RECYCLE_CON(X)

#ifndef _RECYCLE_
#define _RECYCLE_

#include <stdlib.h>
#include "collect.h"

////////////////////////////////////
// class RecycleBin
////////////////////////////////////

class RecycleBin {
public:

    void recycle (CollectionElement* e) { col.append(e); }
    void* getNew (size_t size);

    int isEmpty () { return col.isEmpty(); }
    int size () { return col.size(); }

private:
```



```
void* RecycleBin::getNew (size_t size) {
    Object* obj = col.removeFirst();
    if (obj!=0) return obj;
    return new char[size];
}
```

### Primer upotrebe

```
#include "recycle.h"

class Object {};

class Y : public Object {
public:
    Y() : RECYCLE_CON(Y) {}
    int i,j,k;
    RECYCLE_DEC(Y)
};

// To be put into a .cpp file:
RECYCLE_DEF(Y);

void main () {
    Y* p1 = new Y;
    Y* p2 = new Y;
    delete p1;
    Y* p3 = new Y;
    delete p3;
    delete p2;
}
```

### Analiza kompleksnosti

\* Implementacija klase `RecycleBin` se oslanja na implementaciju kolekcije. U većini slučajeva, posle dovoljno dugog rada programa i ulaska u stacionarni režim kreiranja i uništavanja dinamičkih objekata neke klase, logično je očekivati da se alokacija objekta svodi na uzimanje elementa iz kolekcije, što je izuzetno efikasno. U najgorem slučaju, alokacija se svodi na ugrađeni algoritam alokatora. Dealokacija se u svakom slučaju svodi na ubacivanje elementa u kolekciju:

Operacija	Kompleksnost u najčešćem slučaju	Kompleksnost u najgorem slučaju
<code>X::new()</code> / <code>RecycleBin::getNew()</code>	$O(1)$	$C_a$
<code>X::delete()</code> / <code>RecycleBin::recycle()</code>	$O(1)$	$O(1)$

# Nasleđivanje

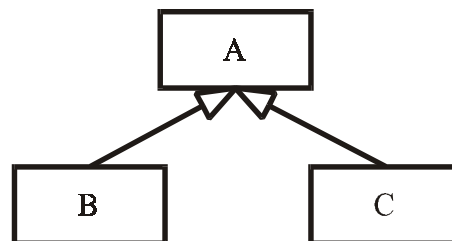
## Izvedene klase

### Šta je nasleđivanje i šta su izvedene klase?

\* U praksi se često sreće slučaj da je jedna klasa objekata (klasa B) podvrsta neke druge klase (klasa A). To znači da su objekti klase B " (specijalna) vrsta" (engl. "*a-kind-of*") objekata klase A, ili da objekti klase B "imaju sve osobine klase A, i još neke, sebi svojstvene". Ovakva relacija između klasa naziva se *nasleđivanje* (engl. *inheritance*): klasa B nasleđuje klasu A.

\* Relacija nasleđivanja se u programskom modelu definiše u odnosu na to šta želimo da klase rade, odnosno koja svojstva i servise da imaju. Primer: da li je krug vrsta elipse, ili je elipsa vrsta kruga, ili su i krug i elipsa podvrste ovalnih figura?

\* Ako je klasa B nasledila klasu A, kaže se još da je klasa A *osnovna klasa* (engl. *base class*), a klasa B *izvedena klasa* (engl. *derived class*). Može se reći i da je klasa A nadklasa (engl. *superclass*), a klasa B podklasa (engl. *subclass*), ili da je klasa A roditelj (engl. *parent*), a klasa B dete (engl. *child*). Relacija nasleđivanja se najčešće prikazuje (usmerenim acikličnim) grafom:



### Kako se definišu izvedene klase u jeziku C++?

\* Da bi se klasa izvela iz postojeće klase, nije potrebno menjati postojeću klasu, niti je ponovo prevoditi. Izvedena klasa se deklarise navođenjem reči `public` i naziva osnovne klase, iza znaka `:` (dvotačka):

```

class Base {
public:
    void f();
private:
    int i;
};

class Derived : public Base {
public:
    void g();
private:

```

```
int j;
```

\* Objekti izvedene klase imaju sve članove osnovne klase, i svoje posebne članove koji su navedeni u deklaraciji izvedene klase.

\* Objekti izvedene klase definišu se i koriste na uobičajen način:

```
void main () {
    Base b;
    Derived d;
    b.f();
    b.g(); // ovo, naravno, ne može
    d.f(); // d ima i funkciju f,
    d.g(); // i funkciju g
}
```

### Prava pristupa

\* Ključna reč `public` u zaglavlju deklaracije izvedene klase znači da su svi javni članovi osnovne klase ujedno i javni članovi izvedene klase.

\* Privatni članovi osnovne klase uvek to i ostaju. Funkcije članice izvedene klase ne mogu da pristupaju privatnim članovima osnovne klase. Nema načina da se "povredi privatnost" osnovne klase (ukoliko neko nije prijatelj te klase, što je zapisano u njenoj deklaraciji), jer bi to značilo da postoji mogućnost da se probije enkapsulacija koju je zamislio projektant osnovne klase.

\* Javnim članovima osnovne klase se iz funkcija članica izvedene klase pristupa neposredno, kao i sopstvenim članovima:

```
class Base {
private:
    int pb;
public:
    int jb;
    void put(int x) {pb=x;}
};

class Derived : public Base {
private:
    int pd;
public:
    void write(int a, int b, int c) {
        pd=a;
        jb=b;
        pb=c; // ovo ne može,
        put(c); // već mora ovako
    }
};
```

\* Deklaracija člana izvedene klase sakriva istoimeni član osnovne klase. Sakrivenom članu osnovne klase može da se pristupi pomoću operatora `::`. Na primer, `Base::jb`.

\* Često postoji potreba da nekim članovima osnovne klase pristupaju funkcije članice izvedenih klasa, ali ne i korisnici klasa. To su najčešće funkcije članice koje direktno pristupaju privatnim podacima članovima. Članovi koji su dostupni samo izvedenim klasama, ali ne i korisnicima spolja, navode se iza ključne reči `protected`: i nazivaju se *zaštićeni članovi* (engl. *protected members*).

\* Zaštićeni članovi ostaju zaštićeni i za sledeće izvedene klase pri sukcesivnom nasleđivanju. Uopšte, ne može se povećati pravo pristupa nekom članu koji je privatn, zaštićen ili javni.

```
class Base {
    int pb;
protected:
    int zb;
public:
    int jb;
    //...
};

class Derived : public Base {
    //...
public:
    void write(int x) {
        jb=zb=x; // može da pristupi javnom i zaštićenom članu,
        pb=x;    // ali ne i privatnom: greška!
    }
};

void f() {
    Base b;
    b.zb=5; // odavde ne može da se pristupa zaštićenom članu
}
```

### ***Konstruktori i destruktori izvedenih klasa***

\* Prilikom nastanka objekta izvedene klase, poziva se konstruktor te klase, ali se pre izvršavanja njegovog tela poziva konstruktor osnovne klase. U zaglavlju definicije konstruktora izvedene klase, u listi inicijalizatora, moguće je navesti i inicijalizator osnovne klase (argumente poziva konstruktora osnovne klase). To se radi navođenjem imena osnovne klase i argumenata poziva konstruktora osnovne klase:

```
class Base {
private:
    int bi;
    //...
public:
    Base(int); // konstruktor osnovne klase
    //...
};

Base::Base (int i) : bi(i) { /*...*/ }

class Derived : public Base {
private:
    int di;
    //...
public:
    Derived(int);
    //...
};

Derived::Derived (int i) : Base(i), di(i+1) { /*...*/ }
```

- \* Pri inicijalizaciji objekta izvedene klase redosled poziva konstruktora je sledeći:
1. Inicijalizuje se podobjekat osnovne klase, pozivom konstruktora osnovne klase.
  2. Inicijalizuju se podaci članovi, eventualno pozivom njihovih konstruktora, po redosledu njihovog deklarisanja.
  3. Izvršava se telo konstruktora izvedene klase.
- \* Pri uništavanju objekta, redosled poziva destruktora uvek je obratan.

```

class XX {
    //...
public:
    XX() {cout<<"Konstruktor klase XX.\n";}
    ~XX() {cout<<"Destruktor klase XX.\n";}
};

class Base {
    //...
public:
    Base() {cout<<"Konstruktor osnovne klase.\n";}
    ~Base() {cout<<"Destruktor osnovne klase.\n";}
    //...
};

class Derived : public Base {
    XX xx;
    //...
public:
    Derived() {cout<<"Konstruktor izvedene klase.\n";}
    ~Derived() {cout<<"Destruktor izvedene klase.\n";}
    //...
};

void main () {
    Derived d;
}

/* Izlaz će biti:
Konstruktor osnovne klase.
Konstruktor klase XX.
Konstruktor izvedene klase.
Destruktor izvedene klase.
Destruktor klase XX.
Destruktor osnovne klase.
*/

```

## Polimorfizam

### *Šta je polimorfizam?*

\* Pretpostavimo da smo projektovali klasu geometrijskih figura sa namerom da sve figure imaju funkciju `crtaj()` kao članicu. Iz ove klase izveli smo klase kruga, kvadrata, trougla itd. Naravno, svaka izvedena klasa treba da realizuje funkciju crtanja na sebi svojstven način (krug se sasvim drugačije crta od trougla). Sada nam je potrebno da u nekom delu programa iscrtamo sve figure koje se nalaze na našem crtežu. Ovim figurama pristupamo preko niza pokazivača tipa `Figura*`. C++ omogućava da figure iscrtamo prostim navođenjem:

```

void crtanje () {
    for (int i=0; i<brojFigura; i++)
        nizFigura[i]->crtaj();
}

```

\* Iako se u ovom nizu mogu naći različite figure (krugovi, trouglovi itd.), mi im pristupamo kao figurama, jer sve vrste figura imaju zajedničku osobinu "da mogu da se nacrtaju". Ipak, svaka od figura svoj zadatak ispuniće onako kako joj to i priliči, odnosno svaki objekat će "prepoznati" kojoj izvedenoj klasi pripada, bez obzira na to što mu se obraćamo "uopšteno", kao objektu osnovne klase. To je posledica naše pretpostavke da su i krug i kvadrat i trougao vrste figura.

\* Svojstvo da svaki objekat izvedene klase, čak i kada mu se pristupa kao objektu osnovne klase, izvršava metod tačno onako kako je to definisano u njegovoj izvedenoj klasi naziva se *polimorfizam* (engl. *polymorphism*).

### ***Virtuelne funkcije***

\* Funkcije članice osnovne klase koje se u izvedenim klasama mogu realizovati specifično za svaku izvedenu klasu, nazivaju se *virtuelne funkcije* (engl. *virtual functions*).

\* Virtuelna funkcija se u osnovnoj klasi deklariše pomoću ključne reči `virtual` na početku deklaracije. Prilikom definisanja virtuelnih funkcija u izvedenim klasama ne mora se stavljati reč `virtual`, ali se to preporučuje radi povećanja čitljivosti i razumljivosti programa.

\* Prilikom poziva odaziva se ona funkcija koja pripada klasi kojoj i objekat koji prima poziv.

```
class ClanBiblioteke {
public:
    virtual void platiClanarinu () // virtuelna funkcija
    { racun-=clanarina; }
    //...
private:
    int racun;
    //...
};

class PocasniClan : public ClanBiblioteke {
public:
    virtual void platiClanarinu () { }
    //...
};

void main () {
    ClanBiblioteke *clanovi[100];
    //...
    for (int i=0; i<brojClanova; i++)
        clanovi[i]->platiClanarinu();
    //...
}
```

\* Virtuelna funkcija osnovne klase ne mora da se redefiniše u svakoj izvedenoj klasi. U izvedenoj klasi u kojoj virtuelna funkcija nije definisana, važi značenje te virtuelne funkcije iz osnovne klase.

\* Deklaracija neke virtuelne funkcije u svakoj izvedenoj klasi mora da se u potpunosti slaže sa deklaracijom te funkcije u osnovnoj klasi (broj i tipovi argumenata, kao i tip rezultata).

### ***Dinamičko vezivanje***

\* Pokazivač na objekat izvedene klase može se implicitno konvertovati u pokazivač na objekat osnovne klase (pokazivaču na objekat osnovne klase može se dodeliti pokazivač na



objekat izvedene klase direktno, bez eksplicitne konverzije). Ovo je interpretacija činjenice da se objekat izvedene klase može smatrati i objektom osnovne klase.

\* Pokazivaču na objekat izvedene klase može se dodeliti pokazivač na objekat osnovne klase samo uz eksplicitnu konverziju. Ovo je interpretacija činjenice da objekat osnovne klase nema sve osobine izvedene klase.

\* Objekat osnovne klase može se inicijalizovati objektom izvedene klase, i objektu osnovne klase može se dodeliti objekat izvedene klase bez eksplicitne konverzije. To se obavlja "odsecanjem" članova izvedene klase koji nisu i članovi osnovne klase.

\* Virtuelni mehanizam se aktivira ako se objektu pristupa preko pokazivača:

```
class Base {
public:
    virtual void f();
    //...
};

class Derived : public Base {
public:
    virtual void f();
};

void g1(Base b) {
    b.f();
}

void g2(Base *pb) {
    pb->f();
}

void main () {
    Derived d;
    g1(d);                // poziva se Base::f
    g2(&d);                // poziva se Derived::f
    Base *pb=new Derived;
    pb->f();                // poziva se Derived::f
    Base b=d;
    b.f();                // poziva se Base::f
    delete pb;
    pb=&b;
    pb->f();                // poziva se Base::f
}
```

\* Postupak koji obezbeđuje da se funkcija koja se poziva određuje u vreme izvršavanja, a ne prevođenja, naziva se *dinamičko vezivanje* (engl. *dynamic binding*).

### ***Virtuelni destruktor***

\* Destruktor je specifična funkcija članica klase koja pretvara "živi" objekat u "običnu gomilu bitova u memoriji". Zbog toga destruktor može da bude virtuelna funkcija.

\* Virtuelni mehanizam obezbeđuje da se pozove odgovarajući destruktor (osnovne ili izvedene klase) kada se objektu pristupa posredno:

```
class Base {
public:
    virtual ~Base(); // destruktor je virtuelan
    //...
};

class Derived : public Base {
```

```
public:
    ~Derived(); // destruktor je virtuelan
    //...
};

void release (Base *pb) { delete pb; }

void main () {
    Base *pb=new Base;
    Derived *pd=new Derived;
    release(pb); // poziva se ~Base
    release(pd); // poziva se ~Derived
}
```

\* Kada klasa ima neku virtuelnu funkciju, verovatno i njen destruktor (ako ga ima) treba da bude virtuelan.

\* Unutar virtuelnog destruktora izvedene klase ne treba eksplicitno pozivati destruktor osnovne klase, jer se on uvek implicitno poziva. Definisanjem destruktora kao virtuelne funkcije obezbeđuje se da se dinamičkim vezivanjem tačno određuje koji će destruktor (osnovne ili izvedene klase) biti *prvo* pozvan; destruktor osnovne klase se *uvek* izvršava (ili kao jedini ili posle destruktora izvedene klase).

\* Konstruktor je funkcija koja od "obične gomile bitova u memoriji" pravi "živi" objekat. Pošto se konstruktor poziva pre nego što je objekat nastao, nema smisla da bude virtuelan, pa C++ to ne dozvoljava. Kada se definiše objekat, uvek se navodi i tip (klasa) kome pripada, pa je određen i konstruktor koji se poziva.

### ***Nizovi i izvedene klase***

\* Objekat izvedene klase je vrsta objekta osnovne klase. Međutim, niz objekata izvedene klase nije vrsta niza objekata osnovne klase. Uopšte, neka kolekcija objekata izvedene klase nije vrsta kolekcije objekata osnovne klase.

\* Na primer, iako je automobil vrsta vozila, parking za automobile nije i parking za sve vrste vozila, jer na parking za automobile ne mogu da stanu i kamioni (koji su takođe vozila). Ili, ako korisnik neke funkcije prosledi toj funkciji korpu banana (banana je vrsta voća), ne bi valjalo da mu ta funkcija vrati korpu u kojoj je jedna šljiva (koja je takođe vrsta voća), smatrajući da je korpa banana isto što i korpa bilo kakvog voća.

\* Ako se računa sa nasleđivanjem, u programu ne treba koristiti nizove objekata, već nizove pokazivača na objekte. Ako se formira niz objekata izvedene klase i on prenese kao niz objekata osnovne klase (što, po prethodno rečenom, semantički nije ispravno, ali je moguće), može doći do greške:

```
class Base {
public: int bi;
};

class Derived : public Base {
public: int di;
};

void f(Base *b) { cout<<b[2].bi; }

void main () {
    Derived d[5];
    d[2].bi=77;
    f(d);           // neće se ispisati 77
}
```

\* U prethodnom primeru, funkcija *f* smatra da je dobila niz objekata osnovne klase koji su kraći (nemaju sve članove) od objekata izvedene klase. Kada joj se prosledi niz objekata izvedene klase (koji su duži), funkcija nema načina da odredi da se niz sastoji samo od objekata izvedene klase. Rezultat je, u opštem slučaju, neodređen.

\* Pored navedene greške, fizički nije moguće direktno smeštati objekte izvedene klase u niz objekata osnovne klase. Objekti izvedene klase su duži, a za svaki element niza je odvojen samo prostor koji je dovoljan za smeštanje objekta osnovne klase.

\* Zbog svega što je rečeno, kolekcije (nizove) objekata treba kreirati kao nizove pokazivača na objekte:

```
void f(Base **b, int i) { cout<<b[i]->bi; }

void main () {
    Base b1,b2;
    Derived d1,d2,d3;
    Base *b[5]; // b se može konvertovati u tip
                // Base**
    b[0]=&d1; b[1]=&b1; b[2]=&d2; // konverzije Derived* u Base*
    b[3]=&d3; b[4]=&b2;
    d2.bi=77;
    f(b,2); // ispisaće se 77
}
```

\* Kako je objekat izvedene klase vrsta objekta osnovne klase, C++ dozvoljava implicitnu konverziju pokazivača *Derived\** u *Base\** (prethodni primer). Zbog logičkog pravila da niz objekata izvedene klase nije vrsta niza objekata osnovne klase, a kako se nizovi ispravno realizuju pomoću nizova pokazivača, C++ ne dozvoljava implicitnu konverziju pokazivača *Derived\*\** (u koji se može konvertovati tip niza pokazivača na objekte izvedene klase) u *Base\*\** (u koji se može konvertovati tip niza pokazivača na objekte osnovne klase). Za prethodni primer nije dozvoljeno:

```
void main () {
    Derived *d[5]; // d je tipa Derived**
    //...
    f(d,2); // nije dozvoljena konverzija Derived** u Base**
}
```

### Apstraktne klase

\* Čest je slučaj da neka osnovna klasa nema nijedan konkretan primerak (objekat), već samo predstavlja generalizaciju izvedenih klasa.

\* Na primer, svi izlazni, znakovno orijentisani uređaji računara imaju funkciju za ispis jednog znaka, ali se u osnovnoj klasi izlaznog uređaja ne može definisati način ispisa tog znaka, već je to specifično za svaki uređaj posebno. Ili, ako iz osnovne klase osoba izvedemo dve klase muškaraca i žena, onda klasa osoba ne može imati primerke, jer ne postoji osoba koja nije ni muškog ni ženskog pola.

\* Klasa koja nema instance (objekte), već su iz nje samo izvedene druge klase, naziva se *apstraktna klasa* (engl. *abstract class*).

\* U jeziku C++, apstraktna klasa sadrži bar jednu virtuelnu funkciju članicu koja je u njoj samo deklarirana, ali ne i definisana. Definicije te funkcije daće izvedene klase. Ovakva virtuelna funkcija naziva se čistom virtuelnom funkcijom. Njena deklaracija u osnovnoj klasi završava se sa =0:

```
class OCharDevice {
public:
    virtual int put (char) =0; // čista virtuelna funkcija
```

```
//...  
};
```

\* U jeziku C++ apstraktna klasa je klasa koja sadrži bar jednu čistu virtuelnu funkciju. Ovakva klasa ne može imati instance, već se iz nje izvode druge klase. Ako se u izvedenoj klasi ne navede definicija neke čiste virtuelne funkcije iz osnovne klase, i ova izvedena klasa je takođe apstraktna.

\* Pokazivači i reference na apstraktnu klasu mogu da se formiraju, ali oni ukazuju na objekte izvedenih konkretnih (neapstraktnih) klasa.

## Višestruko nasleđivanje

\* Nekad postoji potreba da izvedena klasa ima osobine više osnovnih klasa istovremeno. Tada se radi o *višestrukom nasleđivanju* (engl. *multiple inheritance*).

\* Na primer, motocikl sa prikolicom je vrsta motocikla, ali i vrsta vozila sa tri točka. Pri tom, motocikl nije vrsta vozila sa tri točka, niti je vozilo sa tri točka vrsta motocikla, već su ovo dve različite klase. Klasa motocikala sa prikolicom naleđuje obe ove klase.

\* Klasa se deklarise kao naslednik više klasa tako što se u zaglavlju deklaracije, iza znaka :, navode osnovne klase razdvojene zarezima. Ispred svake osnovne klase treba da stoji reč `public`. Na primer:

```
class Derived : public Base1, public Base2, public Base3 {  
//...  
};
```

\* Sva navedena pravila o nasleđenim članovima važe i ovde. Konstruktori svih osnovnih klasa pozivaju se pre poziva konstruktora članova izvedene klase i pre izvršavanja tela konstruktora izvedene klase. Konstruktori osnovnih klasa pozivaju se po redosledu deklarisanja tih klasa u zaglavlju izvedene klase. Destruktori osnovnih klasa izvršavaju se na kraju, posle izvršavanja tela destruktora izvedene klase i destruktora članova.

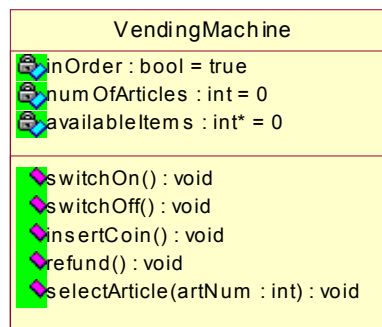
## **II      Osnove objektno          orijentisanog          modelovanja na          jeziku UML**

# Modelovanje strukture

- \* Klasa je osnovna jedinica strukturnog modela sistema.
- \* Klasa je veoma retko izolovana. Ona dobija smisao samo uz druge klase sa kojima je u relaciji. Osnovne relacije između klasa su: asocijacija, zavisnost i generalizacija/specijalizacija.
- \* Koncept *interfejsa* omogućava pravljenje fleksibilnih, labavo spregnutih softverskih komponentata koje se mogu zamenjivati.

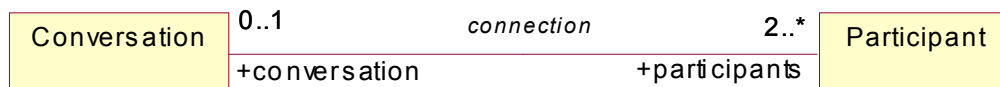
## Klasa, atributi i operacije

- \* Klasom se modeluje apstrakcija. Klasa je opis skupa objekata koji dele iste atribute, operacije, relacije i semantiku.
- \* Atribut je imenovano svojstvo entiteta. Njime se opisuje opseg vrednosti koje instance tog svojstva mogu da imaju.
- \* Operacija je implementacija usluge koja se može zatražiti od bilo kog objekta klase da bi se uticalo na ponašanje.
- \* Klasa se prikazuje pravougaonim simbolom u kome mogu postojati posebni odeljci za ime klase, attribute i operacije:



## Asocijacija

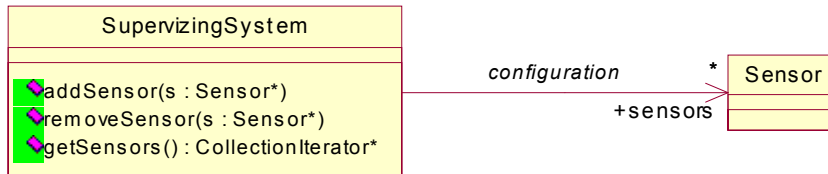
- \* Asocijacija (pridruživanje, engl. *association*) je relacija između klasa čiji su objekti na neki način strukturno povezani. Ta veza između objekata klasa tipično postoji određeno duže vreme, a ne samo tokom trajanja izvršavanja operacije jednog objekta koju poziva drugi objekat. Instanca asocijacije naziva se *vezom* (engl. *link*) i postoji između objekata datih klasa.



- \* Asocijacija se predstavlja punom linijom koja povezuje dve klase. Asocijacija može da ima ime koje opisuje njeno značenje. Svaka strana u asocijaciji ima svoju *ulogu* (engl. *role*) koja se može naznačiti na strani date klase.
- \* Na svakoj strani asocijacije može se definisati kardinalnost (multiplikativnost, engl. *multiplicity*) pomoću sledećih oznaka:

1      tačno 1

- \* proizvoljno mnogo (0 ili više)
- 1..\* 1 ili više
- 0..1 0 ili 1
- 3..7 zadati opseg
- i slično.
- \* Druga posebna karakteristika svake strane asocijacije je *navigabilnost* (engl. *navigability*): sposobnost da se sa te strane (od objekta sa jedne strane veze) dospe do druge strane (do objekta sa druge strane veze). Prema ovom svojstvu, asocijacija može biti simetrična (dvosmerna, bidirekciona) ili asimetrična (jednosmerna, unidirekciona).



- \* Asocijacija prikazana u prvom primeru se na ciljnom OO programskom jeziku na strani klase `Participant` može realizovati na sledeći način:

```

class Conversation;

class Participant {
public:
    //...

    // Funkcije za uspostavljanje, raskidanje i navigaciju
    // preko veza asocijacije:

    void setConversation (Conversation* c) { conversation = c; }
    Conversation* getConversation() { return conversation; }

private:
    Conversation* conversation;
};

```

- \* Asocijacija prikazana u drugom primeru se na ciljnom OO programskom jeziku na strani klase `SupervizingSystem` može realizovati na sledeći način:

```

class Sensor;

class SupervizingSystem {
public:
    //...

    // Funkcije za uspostavljanje, raskidanje i navigaciju
    // preko veza asocijacije:

    void addSensor (Sensor* s) { sensors.append(s); }
    void removeSensor (Sensor* s) { sensors.remove(s); }
    CollectionIterator* getSensors() { return sensors.getIterator(); }

private:
    Collection sensors;
};

```

- \* Primer ukoliko na strani klase B postoji mogućnost navigacije prema klasi A, i multiplikativnost na strani A je tačno 1:

```

class A;

class B {

```

```

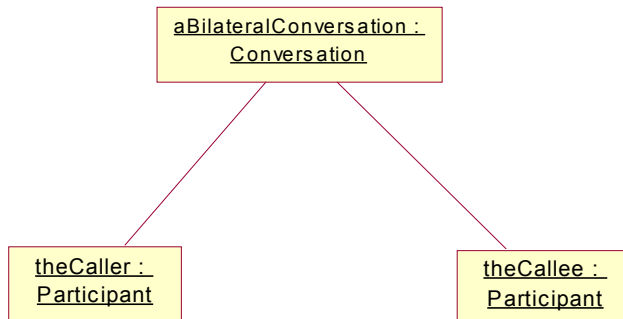
public:
    B (A* a) { myA=a; }
    //...
private:
    A* myA;
};

```

\* Kod navedene realizacije na jeziku C++, potrebno je obratiti pažnju na sledeće. U deklaraciji klase B nije potrebna potpuna definicija klase A, već samo prosta deklaracija `class A;`, jer je objekat B vezan za objekat klase A preko pokazivača. Samo u implementaciji neke složenije operacije koja pristupa članovima klase A potrebna je potpuna definicija klase A. Kako se implementacije ovih funkcija tipično nalaze u modulu B.cpp, samo ovaj modul zavisi od modula sa interfejsom klase A, dok modul B.h ne zavisi. Na ovaj način se značajno smanjuju zavisnosti između modula i vreme prevođenja.

\* Treba obratiti pažnju da pokazivač ili struktura pokazivača u implementaciji klase nije njen *atribut*, pa se ne modeluje atributom u UML modelu, već je to samo *podatak član* koji je manifestacija (posledica) navigabilne asocijacione uloge na drugoj strani asocijacije i generisan je (ručno ili automatski) u implementacionom kodu na ciljnom programskom jeziku kao posledica UML modela i asocijacije u njemu. Treba zato razlikovati dva razdvojena nivoa apstrakcije: model na jeziku UML (u kome postoje klase, atributi, asocijacije, uloge asocijacija, operacije itd.) i implementaciju (kod) na ciljnom programskom jeziku, u kome postoje klase, podaci članovi i funkcije članice, kao posledice (dobijene ručnim ili automatskim generisanjem koda) elemenata modela.

\* Ukoliko je struktura objekata i njihovih veza složena i ne može se direktno videti iz dijagrama klasa i njihovih relacija, jasniji prikaz te strukture može se dati pomoću *dijagrama objekata* (engl. *object diagram*). Na tom dijagramu prikazuju se objekti (kao instance klasa) i njihove veze (kao instance asocijacija).



\* Instancom (objektom) na dijagramu objekata se može modelovati konkretna stvar koja postoji u konceptualnom svetu i sa kojom se mogu raditi određene operacije. Međutim, instanca može prikazivati i apstraktnu stvar, npr. instancu apstraktne klase koja predstavlja bilo koju konkretnu instancu konkretne izvedene klase. Najčešće instanca u modelu predstavlja neki, bilo koji ugledni primerak, tj. prototip datog tipa.

\* Na jeziku UML instanca (objekat) se prikazuje kao pravougaonik u kome su upisani i podvučeni ime (neobavezno), dvotačka i tip objekta (neobavezno).

\* Skupina objekata iste vrste može se prikazati simbolički kao kolekcija anonimnih objekata (engl. *multiobjects*).





- \* Kada se prikazuje instanca, može se prikazati i stanje te instance (vrednosti njenih atributa). Taj prikaz se odnosi na jedan trenutak tokom izvršavanja. Dakle, ovim se prikazuje "snimak" stanja u jednom trenutku dinamičkog izvršavanja programa (statički način modelovanja dinamike).
- \* Vrednosti atributa (stanje) prikazuju se navođenjem liste naziva atributa i njihovih vrednosti (`atribut:tip=vrednost`).
- \* Može se navesti i eksplicitno stanje objekta, ukoliko je ponašanje njegove klase opisano mašinom stanja, navođenjem naziva stanja između srednjih zagrada [].

## Zavisnost

- \* Relacija zavisnosti (engl. *dependency*) postoji ako klasa A na neki način koristi usluge klase B (pristupa njenim članovima). To može biti npr. odnos klijent-server (klasa A poziva operacije klase B) ili odnos instancijalizacije (klasa A pravi objekte klase B).
- \* Za realizaciju ove relacije između klase A i B potrebno je da interfejs ili implementacija klase A "zna" za definiciju klase B. Zbog toga klasa A, kao element modela, zavisi od klase B u sledećem smislu: ako se B kao element modela na neki način promeni ili nestane iz modela, i A možda trpi posledice zbog te promene.
- \* Oznaka:



- \* Primer:



- \* Značenje relacije može da se navede kao stereotip relacije na dijagramu, npr. `<<call>>` ili `<<create>>`.
- \* Ako klasa `Client` koristi usluge klase `Supplier` tako što poziva operacije objekata ove klase (odnos klijent-server), onda ona tipično "vidi" ove objekte kao argumente svojih operacija. U ovom slučaju, za implementaciju na jeziku C++, interfejsu klase `Client` nije potrebna definicija klase `Supplier`, već samo njenoj implementaciji:

```

class Supplier;

class Client {
public:
    //...
    void aFunction (Supplier*);
};

// Implementacija:
void Client::aFunction (Supplier* s) {
    //...
    s->doSomething();
}
  
```

- \* Drugi slučaj zavisnosti stereotipa `<<call>>` je sa pristupom do globalno dostupnog objekta:

```

class Supplier;
  
```

```
class Client {
public:
    //...
    void aFunction ();
};
```

```
// Implementacija:
void Client::aFunction () {
    //...
    Supplier::Instance()->doSomething();
}
```

\* Ako klasa Client instancijalizuje klasu Supplier (zavisnost stereotipa <<create>>), onda je realizacija nalik na:

```
Supplier* Client::createSupplier (/*some_arguments*/) {
    return new Supplier(/*some_arguments*/);
}
```

## Generalizacija/Specijalizacija



\* Relacija generalizacije/specijalizacije (engl. *generalization/specialization*) predstavlja relaciju između klasa koja ima dve važne semantičke manifestacije:

- (a) *Nasleđivanje*: nasleđena (izvedena) klasa implicitno poseduje (nasleđuje) sve atribute, operacije i asocijacije osnovne klase (važi i tranzitivnost).
- (b) *Supstitucija* (engl. *substitution*): objekat (instancija) izvedene klase može se naći svugde gde se očekuje objekat osnovne klase (važi i tranzitivnost). Za strukturni aspekt sistema ovo pravilo ima sledeću bitnu manifestaciju: ako u nekoj asocijaciji učestvuje osnovna klasa, onda u nekoj vezi kao instanci te asocijacije mogu učestvovati objekti svih klasa izvedenih (neposredno ili posredno) iz te klase.

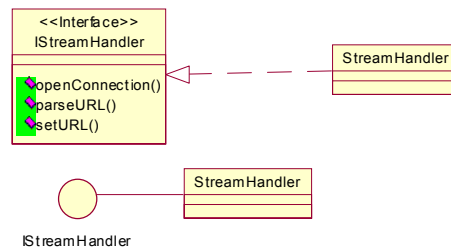
\* Realizacija:

```
class Derived : public Base //...
```

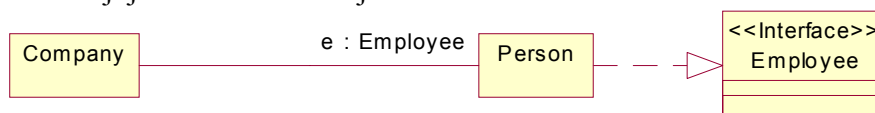
\* Zbog ovako definisane semantike osnovnih relacija između klasa (prvenstveno asocijacije i nasleđivanja), softverski sistemi (aplikacije) koji su modelovani objektno imaju jedno opšte svojstvo: njihova struktura u vreme izvršavanja može se apstraktno posmatrati kao *tipizirani graf*, jer se sastoji iz objekata (instanci klasa) povezanih vezama (instancama asocijacija). Dakle, objekti predstavljaju čvorove, a veze grane jednog grafa. Pri tom, objekti kao čvorovi grafa imaju svoje tipove (to su klase čije su ovo instance), kao i veze koje su instance odgovarajućih asocijacija. Na stranama svake veze koja je instanca neke asocijacije nalaze se instance onih klasa koje povezuje ta asocijacija, ili klasa izvedenih iz njih (uključujući i tranzitivnost nasleđivanja).

## Interfejsi

- *Interfejs* (engl. *interface*) je skup operacija koje definišu uslugu klase ili komponente pružene nekom korisniku (u nekom kontekstu).
- Operacije u interfejsu nemaju implementaciju. Konkretna klasa ili komponenta koje *implementiraju* interfejs obezbeđuje realizaciju ovih operacija. Takve realizacije operacija nazivaju se *metode* (engl. *method*).
- Interfejs ne sadrži atribute. Interfejs je samo specifikacija skupa (apstraktnih) operacija.
- Interfejs se u jeziku C++ koji ne podržava direktno ovaj koncept realizuje apstraktnom klasom koja ima samo apstraktne operacije (čisto virtuelne funkcije), bez atributa.
- Nazivi interfejsa obično počinju velikim slovom I.
- Interfejs se u jeziku UML može predstaviti svojim simbolom (krug sa nazivom interfejsa) ili kao klasa sa stereotipom <<interface>>.
- Klasa ili komponenta koja realizuje dati interfejs povezuje se relacijom *realizacije* (engl. *realize*) prema interfejsu. Ova relacija se prikazuje kao isprekidana linija sa zatvorenom strelicom na vrhu ukoliko je interfejs prikazan kao klasa sa stereotipom, ili kao obična puna linija ukoliko je interfejs prikazan ikonom.



- Interfejsom se može opisati ugovor koji nudi slučaj upotrebe ili komponenta. Interfejsom se tipično opisuju usluge koje data apstrakcija nudi u odgovarajućem kontekstu, ili koje neka apstrakcija traži od učesnika u kolaboraciji.
- Interfejs može učestvovati u relacijama kao i klasa. U relacijama se on može prikazivati kao ikona (krug) ili kao klasa sa stereotipom.
- Interfejs je sličan apstraktnoj klasi. Ipak, razlike postoje. Interfejs nema atribute, dok apstraktna klasa može da ih ima. Interfejs nema nijednu metodu (implementaciju operacije), dok apstraktna klasa može da ih ima. Interfejs može biti realizovan i klasom (logički koncept), ali i komponentom (fizički koncept).
- Klasa može da realizuje više interfejsa. Instanca takve klase podržava sve te interfejse, jer interfejs predstavlja ugovor koji mora ispuniti onaj ko mu podleže. Međutim, u datom kontekstu može biti značajan samo neki od interfejsa koje data klasa zadovoljava. Na primer, u nekoj asocijaciji, klasa sa jedne strane asocijacije može da zahteva od druge strane samo neki interfejs. To znači da ta druga strana igra određenu *ulogu* u asocijaciji.
- *Uloga* (engl. *role*) je "lice" koje data apstrakcija pokazuje datom okruženju. Uloga koju jedna apstrakcija igra u nekoj relaciji označava se navođenjem naziva interfejsa koji ta uloga nudi i koji je bitan za tu relaciju:



---

## Modelovanje ponašanja

---

- \* Ponašanje sistema se na jeziku može UML modelovati pomoću različitih koncepata:
  - operacije, koje se specifikuju na dijagramima klasa, a pojavljuju na mnogim drugim dijagramima
  - interakcije, koje se prikazuju na dijagramima interakcija
  - aktivnosti i akcije, koje se prikazuju na dijagramima aktivnosti
  - mašine stanja, koje se prikazuju na dijagramima prelaza stanja.

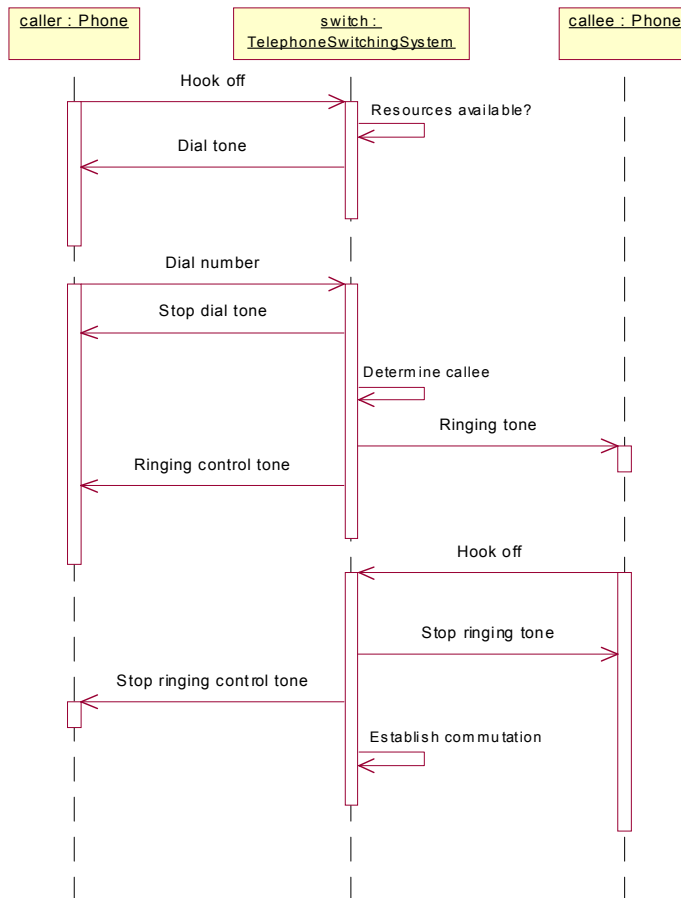
### Interakcije i dijagrami interakcije

\* Ponašanje sistema se veoma često realizuje interakcijama između objekata. *Interakcija* (engl. *interaction*) je ponašanje koje se sastoji od skupa poruka koje se razmenjuju između objekata u nekom kontekstu da bi se ispunila neka svrha. *Poruka* (engl. *message*) je specifikacija komunikacije između objekata koja prenosi informaciju sa očekivanjem da će se pokrenuti neka aktivnost.

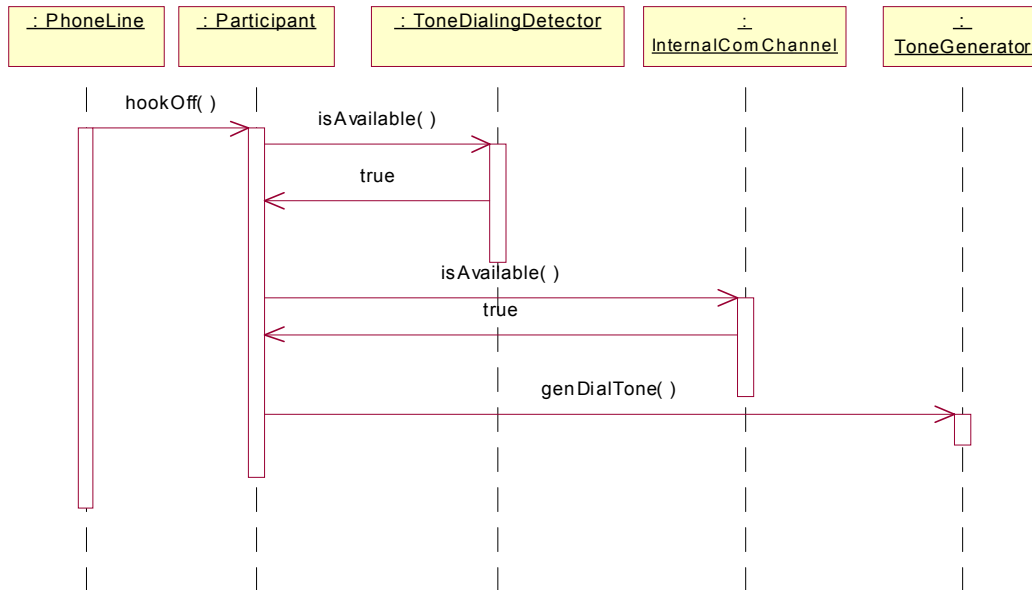
\* *Dijagram interakcije* (engl. *interaction diagram*) opisuje ponašanje i odnosi se na neki kontekst – deo sistema čije ponašanje opisuje. To može biti neka funkcionalnost sistema ili neka operacija klase.

\* Dijagramom interakcije mogu se prikazati interakcije na različitom nivou apstrakcije i granularnosti, u različitim fazama razvoja softvera:

- Za specifikaciju ili analizu zahteva, u ranim fazama analize i projektovanja sistema. U ovom slučaju dijagrami su neformalni, često i nekompletni, i predstavljaju samo grube skice interakcija koje ne uključuju konkretne softverske objekte niti njihove operacije, već češće samo konceptualne, apstraktne objekte iz domena problema i neformalne poruke između njih:

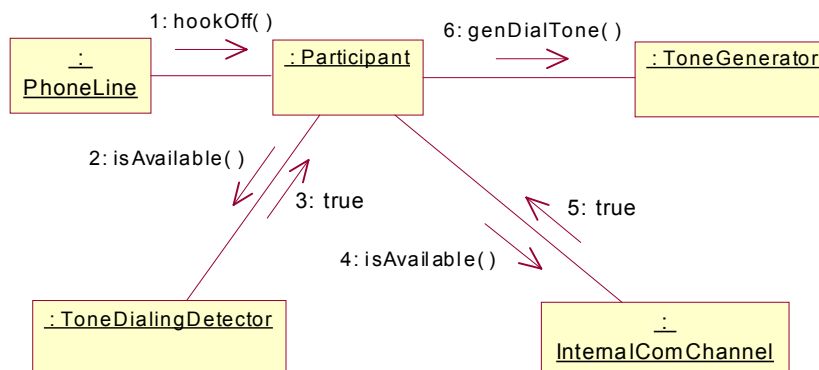


- Za detaljan dizajn, u kasnim fazama projektovanja. U ovom slučaju dijagrami sadrže instance klasa iz modela (kao na dijagramima objekata) i pozive njihovih operacija:



\* Postoje dve vrste dijagrama interakcije: dijagram kolaboracije i dijagram sekvence. Ovi dijagrami predstavljaju dva različita pogleda na semantički istu stvar (istu interakciju), samo što naglašavaju različite aspekte te interakcije.

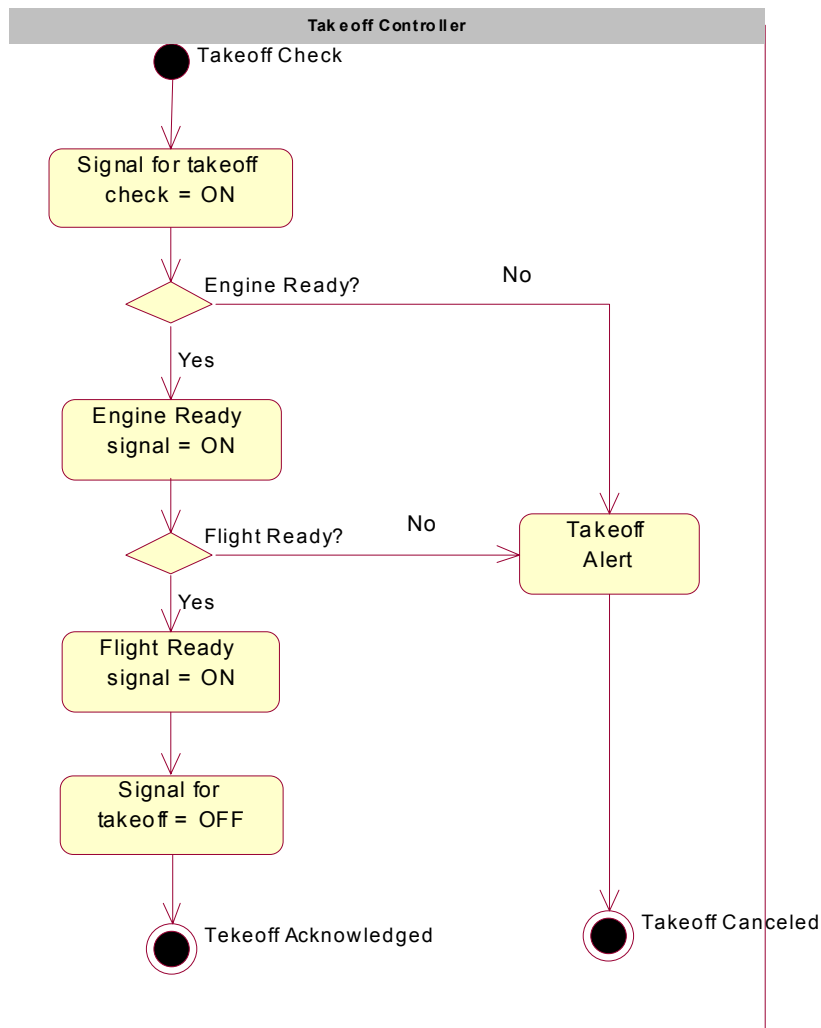
\* *Dijagram kolaboracije* (engl. *collaboration diagram*) je dijagram interakcije koji naglašava strukturnu organizaciju objekata koji razmenjuju poruke. Grafički, ovaj dijagram je skup čvorova koji predstavljaju objekte i skup linija koji ih povezuju i preko kojih idu poruke. Čvorovi predstavljaju objekte (kao instance klasa), a linije predstavljaju instance relacija između njihovih klasa (tipično asocijacija i zavisnosti):



\* *Dijagram sekvence* (engl. *sequence diagram*) je dijagram interakcije koji naglašava vremenski redosled poruka (vidi prve primere u ovom odeljku). Ovaj dijagram prikazuje objekte poredane po  $x$  osi, dok se poruke ređaju kao horizontalne linije po  $y$  osi, pri čemu vreme raste nadole.

## Aktivnosti i dijagrami aktivnosti

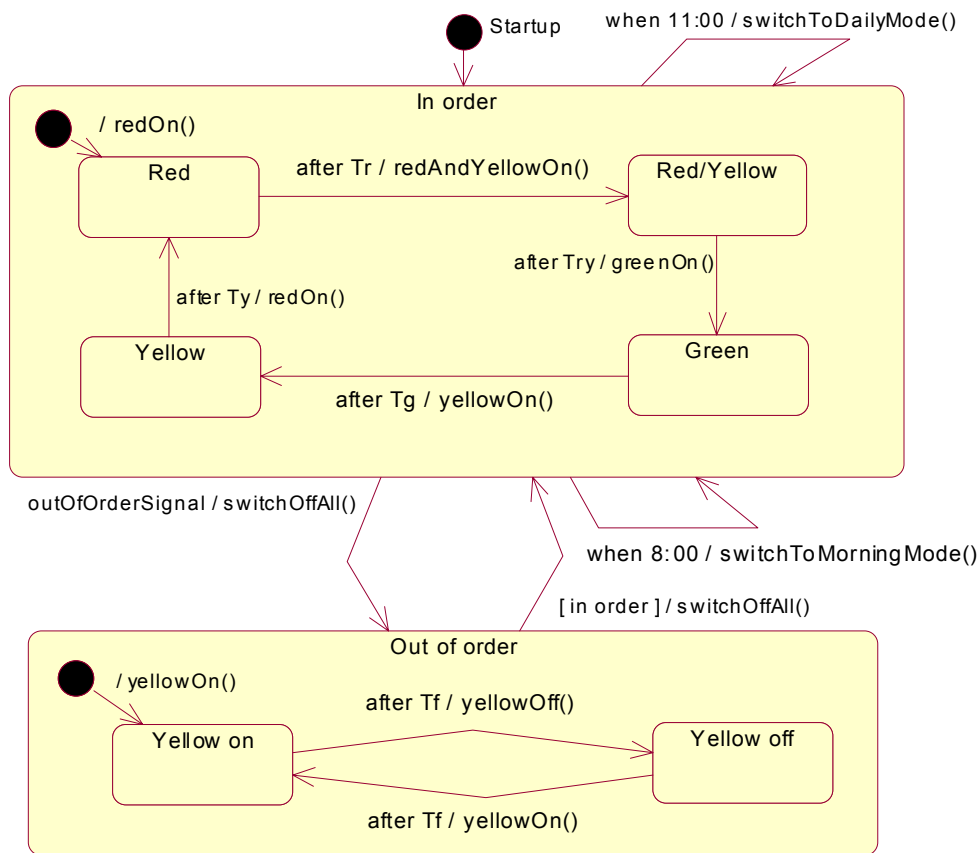
- Ponekad je ponašanje nekog dela sistema (npr. operacije ili interakcije) pogodno opisati dijagramom nalik na dijagram kontrole toka (engl. *control-flow chart*). Za ovakve potrebe, u jeziku UML postoje *dijagrami aktivnosti* (engl. *activity diagram*).



- Dijagram aktivnosti ima kontekst u kome opisuje neko ponašanje, dakle, može biti pridružen nekom elementu (operaciji, klasi i slično).
- *Dijagram aktivnosti* (engl. *activity diagram*) prikazuje tok od aktivnosti do aktivnosti. *Aktivnost* (engl. *activity*) je neatomično izvršavanje u okviru neke mašine stanja. Aktivnost se konačno svodi (dekomponuje) na *akcije* (engl. *action*).
- *Akcija* (engl. *action*) predstavlja atomično izračunavanje koje menja stanje sistema i/ili proizvodi rezultat. Akcija može biti poziv druge operacije, slanje signala nekom objektu, nastanak ili uništavanje nekog objekta, ili neko prosto izračunavanje, kao što je naredba ili izraz na ciljnom programskom jeziku.
- Prema tome, akcije predstavljaju atomično, elementarno izvršavanje, dok se aktivnosti mogu dalje dekomponovati. Aktivnosti se mogu predstavljati drugim dijagramima aktivnosti.
- Dijagram aktivnosti predstavlja skup čvorova (aktivnosti) i linija koje ih povezuju (tranzicije).
- Akcije i aktivnosti predstavljaju se stanjima – pravougaonicima sa zaobljenim temenima. U pravougaoniku stanja napisan je naziv aktivnosti ili akcije koja se izvršava.
- Početno stanje aktivnosti predstavljene dijagramom označava se punim krugom, a završno punim krugom sa opisanim koncentričnim krugom.

- Tranzicije se predstavljaju usmerenim linijama koje povezuju stanja. Tranzicija može biti uslovljena. Uslov se piše unutar srednjih zagrada [].
- Uslovno grananje predstavlja se malim romбом iz koga izlazi više tranzicija sa uslovima.
- Da bi se bolje prikazalo koji objekat izvršava koju aktivnost, dijagram se može podeliti vertikalnim "plivačkim stazama" (engl. *swimlanes*) koje predstavljaju objekte, sa nazivima objekata na vrhu.

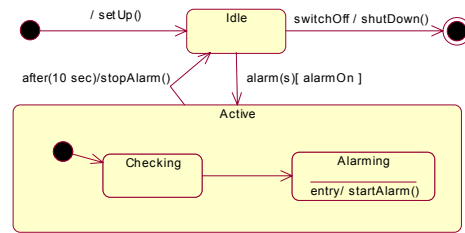
## Mašine stanja i dijagrami prelaza stanja



- Dinamički aspekt ponašanja nekog dela sistema (klase, podsistema itd.) može se modelovati *mašinom stanja* (engl. *state machine*), ukoliko to odgovara semantici tog ponašanja.
- Mašinom stanja modeluje se ponašanje (tipično klase) kod koje reakcija na spoljašnji događaj ne zavisi isključivo od tog događaja, nego i od predistorije događaja, odnosno od stanja u kome se objekat nalazi.
- Za razliku od dijagrama interakcije i aktivnosti kojima se modeluje ponašanje grupe instanci, mašinom stanja modeluje se ponašanje pojedinačne instance datog tipa.
- *Mašina stanja* (engl. *state machine*) je ponašanje koje definiše sekvencu stanja kroz koje objekat prolazi tokom svog životnog veka, kao odgovor na događaje.
- *Dijagramom stanja* (engl. *statechart diagram*) prikazuje se ponašanje definisano mašinom stanja pomoću stanja i prelaza.

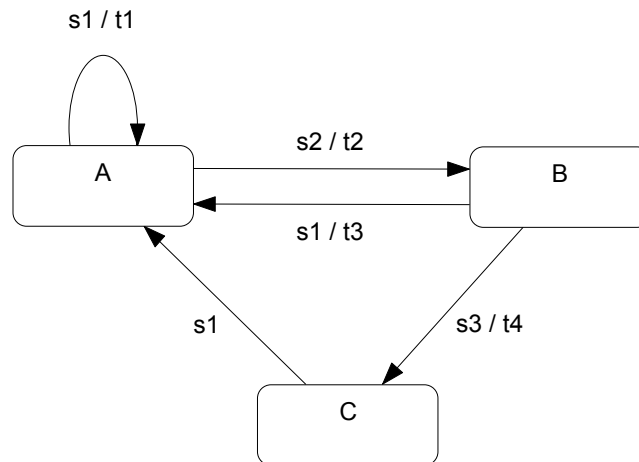


- *Stanje* (engl. *state*) je uslov ili situacija u životu objekta tokom kojeg objekat zadovoljava neki uslov, obavlja neku aktivnost ili čeka na događaj.
- Objekat menja svoje stanje na pojavu događaja. Tada se vrši *prelaz* u isto ili neko drugo stanje.
- *Prelaz* (ili *tranzicija*, engl. *transition*) je relacija između stanja koja definiše da će objekat, na pojavu odgovarajućeg događaja i pod određenim uslovom, izvršiti neke akcije i preći u neko drugo ili isto stanje.
- Stanje se predstavlja pravougaonikom sa zaobljenim uglovima. Prelaz se označava usmerenom linijom između stanja.
- Stanje ima sledeće delove:
  - Ime (engl. *name*): ime stanja.
  - Ulazne/izlazne akcije (engl. *entry/exit actions*): akcije koje se izvršavaju svaki put kada objekat ulazi, odnosno izlazi iz datog stanja.
  - Interni prelazi (engl. *internal transitions*): prelazi koji se obavljaju bez promene stanja.
  - Podstanja (engl. *substates*): ugnežđena stanja.
  - Odloženi događaji (engl. *deferred events*): događaji koji se ne obrađuju u tom stanju, ali se i ne odbacuju. Njihova obrada se odlaže i obaviće se kada objekat bude u stanju koje reaguje na te događaje.
- Početno i krajnje stanje mašine označavaju se kao kod dijagrama aktivnosti.
- Prelaz se opisuje sledećim delovima:
  - Izvorišno stanje (engl. *source state*).
  - Događaj (engl. *event*) koji pokreće prelaz, ukoliko je uslov zadovoljen. Ukoliko događaj nije zadat, stanje predstavlja aktivnost, a prelaz se vrši implicitno, kada se aktivnost završi.
  - Uslov (engl. *guard condition*) koji treba da bude zadovoljen da bi se prelaz izvršio. Ukoliko nijedna tranzicija za dato stanje i dati događaj nema zadovoljen uslov, događaj se odbacuje (ukoliko nije odložen).
  - Akcija (engl. *action*) koja se vrši prilikom prelaza. Ona može biti upućivanje događaja (signal ili poziv operacije) drugom ili istom objektu, ili neko drugo prosto izračunavanje.
  - Odredišno stanje (engl. *target state*) u koje se prelazi.
- Na prelazu se ovi elementi navode po sledećoj sintaksi: `event[condition]/action`. Svaki element je opcioni.
- Stanje može imati ugnežđena stanja. Ugnežđivanje može biti hijerarhijsko do proizvoljne dubine. Ugnežđena stanja mogu biti sekvencijalna ili konkurentna. Kod sekvencijalnih podstanja, objekat je uvek u jednom, najdublje ugnežđenom podstanju. Ukoliko dato podstanje ne reaguje na pristigli događaj, traži se, redom naviše, prvo okružujuće stanje koje reaguje. Kada tranzicija vodi u neko složeno stanje, onda se ulazi u najdublje ugnežđeno stanje po hijerarhiji, korišćenjem početnih stanja za ugnežđena stanja. Za primer sa donje slike: tranzicija `alarm(s)` vodi u stanje `Check`; iz stanja `Alarming` se po isteku 10 sec prelazi u stanje `Idle`.



### Implementacija mašina stanja

- Postoji mnogo načina implementacije mašina stanja, koji se razlikuju po mnogim parametrima, kao što su složenost ili efikasnost implementacije. Još neki parametri mogu da budu:
  - Kontrola toka. Mašina stanja može imati sopstvenu, nezavisnu nit kontrole toka. U tom slučaju mašina prima događaje najčešće preko nekog bafera, obrađuje ih jedan po jedan, a drugim mašinama poruke šalje sinhrono ili asinhrono. Sa druge strane, mašina stanja može biti i pasivan objekat, pri čemu se prelaz (obrada događaja) izvršava u kontekstu onoga ko je događaj poslao (pozivaoca).
  - Način prijema događaja. Događaji se mogu primati centralizovano, preko jedinstvene funkcije za prijem događaja, ili jedinstvenog bafera za događaje. U tom slučaju sadržaj događaja određuje prelaz mašine stanja. Sa druge strane, interfejs objekta može da sadrži više operacija i da svaka operacija predstavlja zapravo jedan događaj na osnovu koga se vrši prelaz.
- Ovde će biti prikazan samo jedan jednostavan način realizacije mašina stanja. Interfejs mašine sadrži sve one operacije koje predstavljaju događaje na koje mašina reaguje.
- Implementacija objekta-automata sadrži više podobjekata koji predstavljaju stanja mašine. Svi ovi podobjekti imaju zajednički interfejs, što znači da su njihove klase izvedene iz osnovne klase stanja date mašine (u primeru klasa *State*). Ovaj interfejs stanja sadrži sve operacije interfejsa samog automata, s tim da je njihovo podrazumevano ponašanje prazno. Izvedene klase konkretnih stanja redefinišu ponašanje za svaki događaj za koji postoji prelaz iz datog stanja. Objekat-automat sadrži pokazivač na tekuće stanje, kome se obraća preko zajedničkog interfejsa tako što poziva onu funkciju koja je pozvana spolja. Virtualni mehanizam obezbeđuje da se izvrši prelaz svojstven tekućem stanju. Posle prelaza, tekuće stanje vraća pokazivač na određeno, naredno tekuće stanje.
- Na ovaj način dobija se efekat da objekat-automat menja ponašanje u zavisnosti od tekućeg stanja (projektni obrazac *State*).
- Ograničenja ovog jednostavnog koncepta su da ne postoji ugnežđivanje stanja, *entry* i *exit* akcije se vrše uvek, čak i ako je prelaz u isto stanje, nema inicijalnih prelaza ni pamćenja istorije itd.
- Realizacija opisanog šablona biće prikazana na primeru sledeće mašine stanja:



Izvorni kod koji implementira ovu mašinu stanja izgleda ovako:

```

// Project: Real-Time Programming
// Subject: Finite State Machines (FSM)
// Module: FSM Example
// File: fsmexmpl.cpp
// Date: 23.11.1996.
// Author: Dragan Milicev
// Contents: State Design Pattern Example

#include <iostream.h>

/////////////////////////////////////////////////////////////////
// class State
/////////////////////////////////////////////////////////////////

class FSM;

class State {
public:

    State (FSM* fsm) : myFSM(fsm) {}

    virtual State* signal1 () { return this; }
    virtual State* signal2 () { return this; }
    virtual State* signal3 () { return this; }

    virtual void entry () {}
    virtual void exit () {}

protected:

    FSM* fsm () { return myFSM; }

private:

    FSM* myFSM;

};

/////////////////////////////////////////////////////////////////
// classes StateA, StateB, StateC
/////////////////////////////////////////////////////////////////

class StateA : public State {
public:

```

```
    StateA (FSM* fsm) : State(fsm) {}

    virtual State* signal1 ();
    virtual State* signal2 ();

    virtual void entry () { cout<<"Entry A\n"; }
    virtual void exit  () { cout<<"Exit  A\n"; }

};

class StateB : public State {
public:

    StateB (FSM* fsm) : State(fsm) {}

    virtual State* signal1 ();
    virtual State* signal3 ();

    virtual void entry () { cout<<"Entry B\n"; }
    virtual void exit  () { cout<<"Exit  B\n"; }

};

class StateC : public State {
public:

    StateC (FSM* fsm) : State(fsm) {}

    virtual State* signal1 ();

    virtual void entry () { cout<<"Entry C\n"; }
    virtual void exit  () { cout<<"Exit  C\n"; }

};

////////////////////////////////////
// class FSM
////////////////////////////////////

class FSM {
public:

    FSM ();

    void signal1 ();
    void signal2 ();
    void signal3 ();

protected:

    friend class StateA;
    friend class StateB;
    friend class StateC;
    void transition1 () { cout<<"Transition 1\n"; }
    void transition2 () { cout<<"Transition 2\n"; }
    void transition3 () { cout<<"Transition 3\n"; }
    void transition4 () { cout<<"Transition 4\n"; }

private:
```

```

    StateA stateA;
    StateB stateB;
    StateC stateC;

    State* currentState;

};

FSM::FSM () : stateA(this), stateB(this), stateC(this),
              currentState(&stateA) {
    currentState->entry();
}

void FSM::signal1 () {
    currentState->exit();
    currentState=currentState->signal1();
    currentState->entry();
}

void FSM::signal2 () {
    currentState->exit();
    currentState=currentState->signal2();
    currentState->entry();
}

void FSM::signal3 () {
    currentState->exit();
    currentState=currentState->signal3();
    currentState->entry();
}

////////////////////////////////////
// Implementation
////////////////////////////////////

State* StateA::signal1 () {
    fsm()->transition1();
    return this;
}

State* StateA::signal2 () {
    fsm()->transition2();
    return &(fsm()->stateB);
}

State* StateB::signal1 () {
    fsm()->transition3();
    return &(fsm()->stateA);
}

State* StateB::signal3 () {
    fsm()->transition4();
    return &(fsm()->stateC);
}

State* StateC::signal1 () {

```

```
    return &(fsm()->stateA);
}

////////////////////////////////////
// Test
////////////////////////////////////

void main () {
    cout<<"\n\n";
    FSM fsm; cout<<"\n";
    fsm.signal1(); cout<<"\n";
    fsm.signal2(); cout<<"\n";
    fsm.signal1(); cout<<"\n";
    fsm.signal3(); cout<<"\n";
    fsm.signal1(); cout<<"\n";
    fsm.signal2(); cout<<"\n";
    fsm.signal3(); cout<<"\n";
    fsm.signal2(); cout<<"\n";
    fsm.signal1(); cout<<"\n";
}
```

# Organizacija modela

- \* Model se hijerarhijski organizuje u *pakete*, koji sadrže različite elemente modela.
- \* Delovi modela prikazuju se na *dijagramima*.

## Paketi

- Specifikacija elemenata složenog sistema podrazumeva manipulisanje velikim brojem klasa, dijagrama i ostalih elemenata. Da bi se model bolje organizovao, elementi se grupišu u celine – pakete.
- *Paket* (engl. *package*) je opšti mehanizam jezika UML za grupisanje elemenata u celine.
- Paket može grupisati elemente bilo koje vrste. Ipak, u dobro organizovanom sistemu paketi sadrže elemente koji su srodni i kohezivni a slabije vezani sa elementima van paketa.
- Paket može *sadržati* elemente koji su u njegovom vlasništvu (kada se obriše paket, nestaju i elementi koje on sadrži), ali isto tako i *referisati* (koristiti) elemente iz drugih paketa.
- Paket predstavlja oblast važenja imena (engl. *namespace*). Elementi jedne vrste moraju imati jedinstveno ime unutar paketa koji ih sadrži.
- Paket može sadržati i druge pakete. Na ovaj način se sistem hijerarhijski organizuje.
- Paket predstavlja i jedinicu enkapsulacije: za svaki element paketa može se definisati njegova *vidljivost* (engl. *visibility*) izvan tog paketa. Vidljivost može biti *public*, *protected* ili *private* sa istim značenjem kao i za članove klasa (paketi se mogu nasledivati). Paket može referisati (koristiti) samo elemente drugih paketa do kojih ima pravo pristupa.
- Ako neki element paketa A treba da koristi neki od (njemu dostupnih) elemenata paketa B, onda je potrebno definisati relaciju zavisnosti od A ka B (A zavisi od B). Da bi elementi paketa A imali pristup do elemenata paketa B, ova relacija ima stereotip `<<import>>`. Ovo je jednosmerna dozvola pristupa do elemenata uključenog paketa.
- Ugnežđeni paket može da pristupi svim elementima do kojih može da pristupi paket koji ga sadrži.
- Paket se na jeziku UML prikazuje pravougaonikom sa "ručkom":



## Dijagrami

- *Dijagram* (engl. *diagram*) je grafička prezentacija skupa elemenata koji predstavlja samo jedan pogled na jedan deo modela.
- Dijagram ne nosi semantiku i ne sadrži elemente modela. Dijagram samo predstavlja prikaz nekih elemenata modela.
- Dijagrami treba da prikazuju samo po jedan isečak modela koji želite da naglasite. Dijagram ne treba da ima više od 5-10 elemenata i treba da bude jasan i pregledan.

# **III Uvod u sisteme za rad u realnom vremenu**



---

# Definicija sistema za rad u realnom vremenu

---

- Kako su vremenom računari postajali manji, brži, pouzdaniji i jeftiniji, tako su se širile i oblasti njihove primene. Jedna od najbrže razvijanih oblasti primene su one aplikacije koje nemaju kao svoj primarni cilj obradu informacija, nego je obrada informacija samo neophodni preduslov za ispunjenje njihove osnovne namene. Na primer, sistem za kontrolu aviona ima primarni zadatak da upravlja letom aviona, pri čemu koristi obradu podataka koja nije njegov primarni cilj.
- Ovakve aplikacije, u kojima softverski sistem služi za nadzor i upravljanje određenog većeg inženjerskog (hardverskog) okruženja i koji ispunjava svoj cilj obradom informacija, ali pri čemu obrada informacija jeste samo sredstvo, a ne primarni cilj, nazivaju se *ugrađeni* (engl. *embedded* sistemi).
- Postoji procena da od ukupne svetske proizvodnje mikroprocesora, 99% njih radi u ugrađenim sistemima.
- Ovakvi sistemi često spadaju u kategoriju *sistema za rad u realnom vremenu* (engl. *real-time system*, kratko RT sistem). Nekoliko definicija RT sistema:
  - RT sistem je sistem za koji je vreme za koje se proizvede odgovor značajno. To je najčešće zbog toga što ulaz predstavlja neku promenu u realnom okruženju, a izlaz treba da odgovara toj promeni. Kašnjenje od ulaza do izlaza mora da bude dovoljno malo da bi izlaz bio prihvatljiv.
  - RT sistem je bilo koja aktivnost ili sistem koji obrađuje informacije i koji treba da odgovori na spoljašnje pobude u konačnom i specifikovanom roku.
  - RT sistem je sistem koji reaguje na spoljašnje pobude (uključujući i protok vremena) u okviru vremenskih intervala koje diktira okruženje.
  - *RT sistem je sistem koji obrađuje informacije i čije korektno funkcionisanje ne zavisi samo od logičke korektnosti rezultata, nego i od njihove pravovremenosti.* Drugim rečima, rezultat isporučen neblagovremeno ili uopšte ne isporučen je isto tako loš kao i pogrešan rezultat.
- Treba primetiti da se u svim definicijama pominje *vremenska* odrednica funkcionisanja sistema, ali da se ona uvek posmatra relativno. Ne postoji apsolutna odrednica koja bi definisala koliko malo vreme odziva treba da bude. Ono treba samo da bude "dovoljno malo", posmatrano relativno za dati sistem. To može biti od nekoliko milisekundi (npr. sistem za vođenje projektila), desetina ili stotina milisekundi (npr. telefonska centrala), do čak nekoliko sekundi (npr. kontrola inernog industrijskog procesa).

## Podela i terminologija RT sistema

- Tradicionalna podela RT sistema je na sledeće tipove:
  - "Tvrdi" (engl. *hard*): RT sistemi za koje je apsolutni imperativ da odziv stigne u zadatom roku (engl. *deadline*). Prekoračenje roka (engl. *deadline miss*) ili neisporučenje rezultata može da dovede do katastrofalnih posledica na živote ljudi, materijalna sredstva ili okolinu. Primeri: sistem za kontrolu nuklearne elektrane, sistem za upravljanje letom aviona itd.

- "Meki" (engl. *soft*): RT sistemi kod kojih su rokovi važni, ali se povremeno mogu i prekoračiti, sve dok performanse sistema (propusnost i kašnjenje) statistički ulaze u zadate okvire. Primeri: telefonska centrala, sistem za prikupljanje podataka u industriji itd.
- Prema ovim definicijama, važne karakteristike RT sistema su sledeće:
  - Za *hard* sisteme, bitno je teorijski dokazati njihovu *izvodljivost* (engl. *feasibility*), tj. pokazati da se zadati rokovi neće prekoračiti ni u kom slučaju, pri zadatim uslovima i resursima. Ova analiza izvodljivosti najčešće podrazumeva analizu *rasporedivosti* (engl. *schedulability*) definisanih poslova na raspoložive procesne jedinice.
  - Za *soft* sisteme, bitno je teorijski, simlaciono ili praktično pokazati da su performanse (engl. *performance*) sistema zadovoljavajuće, tj. u zadatim granicama pod svim uslovima. To podrazumeva statističku analizu (npr. srednje vrednosti i disperzije) parametara performansi, kao što su kašnjenje (engl. *delay*) ili propusnost (engl. *throughput*).
- "Stvarnim" RT sistemom (engl. *real real-time*) se naziva *hard* RT sistem kod koga su vremenski rokovi apsolutno kratki (reda milisekundi).
- "Strogim" RT sistemom (engl. *firm real-time*) se naziva *soft* RT sistem kod koga je zakasneli odgovor beskoristan.
- Mnogi sistemi u praksi imaju više svojih komponenata koje spadaju u različite navedene kategorije. Često se za prekoračenje roka definiše odgovarajuća funkcija cene koju treba minimizovati pri realizaciji sistema.

## Primeri RT sistema

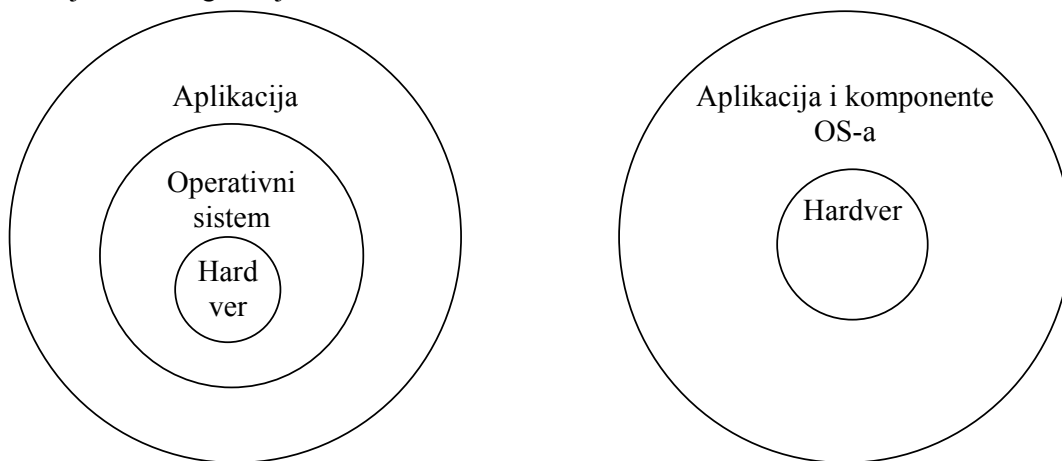
- Sistemi za kontrolu procesa (engl. *process control systems*): upravljanje cevovodima, namešavanje supstanci, praćenje sagorevanja, nadzor električne centrale itd.
- Sistemi za proizvodnju (engl. *manufacturing systems*): pokretna traka za sastavljanje delova, mašina za proizvodnju delova itd.
- Sistemi za komunikaciju, upravljanje i nadzor (engl. *communication, command, and control systems*, CCC): kontrola leta, upravljanje šinskim saobraćajem, upravljanje projektilima, avionski sistem, komunikacioni sistemi itd.
- Telekomunikacioni sistemi: telefonska centrala (javna, kućna, za mobilnu telefoniju), mobilni telefon, komunikacioni uređaji (*router, switch*, itd.) itd.
- Razni ugrađeni (engl. *embedded*) sistemi: medicinski sistemi i sl.
- Simulacioni sistemi: simulacija leta aviona, simulacija borbenih dejstava itd.

---

## Karakteristike RT sistema

---

- RT sistemi su najčešće vrlo veliki i kompleksni. Njihova funkcionalnost je složena, a implementacija može da varira od nekoliko stotina linija asemblerskog ili C koda, sve do desetak miliona linija koda nekog višeg programskog jezika. Teško je ili nemoguće da takav sistem razume, napravi ili održava jedna osoba.
- RT sistemi su najčešće konkurentni, jer to odgovara uporednom i kontinualnom dešavanju promena, procesa i događaja u realnom okruženju. Zbog toga i koncepti za RT programiranje treba da podrže konkurentnost, jer je tako lakše modelovati konkurentne prirodne procese nego pomoću sekencijalnih koncepata.
- RT sistemi često manipulišu racionalnim brojevima koji su numeričke aproksimacije veličina iz okruženja. Algoritmi ugrađeni u RT softver moraju da uzmu u obzir ograničenu tačnost ovih aproksimacija i mogućnost neregularnih operacija.
- RT softver neposredno interaguje sa hardverom, pa je zato neophodno da programski jezik omogućuje programski pristup do posebnih hardverskih uređaja.
- RT sistemi vrlo često moraju da budu ekstremno pouzdani i sigurni. Oni operišu u realnom okruženju pod različitim uticajima koji mogu da dovedu do otkaza ili neispravnog funkcionisanja, a koje može uzrokovati štetu ili ugrožavanje života i okoline.
- RT sistemima se postavljaju zahtevi za garantovanim vremenom odziva. Potrebno je zato imati sredstvo za pouzdanu predikciju najgoreg mogućeg vremena izvršavanja. Iako su performanse važne za *soft* sisteme, za *hard* sisteme je suštinski važna pouzdana predvidivost ponašanja u vremenu.
- Varijante konfiguracije softvera za RT sisteme:



## **IV Pouzdanost i tolerancija otkaza**

---

# Pouzdanost i tolerancija otkaza

---

- Zahtevi za pouzdanost i sigurnost RT sistema su obično mnogo strožiji nego za druge aplikacije. Na primer, ako dođe do greške u radu nekog programa za naučna izračunavanja, izvršavanje tog programa može jednostavno da se prekine. Sa druge strane, sistem koji kontroliše neki industrijski proces, npr. neku veliku peć, ne sme sebi da dozvoli da prestane da radi zbog greške. Umesto toga, on treba da nastavi da radi sa eventualno degradiranom funkcionalnošću, ili da preduzme skupu i komplikovanu, ali kontrolisanu operaciju gašenja peći. Ne treba ni naglašavati važnost optornosti na otkaze sistema koji mogu da ugroze živote u slučaju pada, npr. sistem za kontrolu leta ili nuklearne centrale.
- Izvori grešaka u izvršavanju programa su:
  - Neadekvatna specifikacija softvera
  - Greške u realizaciji softverskih komponenata
  - Otkazi procesorskih komponenata u sistemu
  - Tranzientni ili permanentni uticaji na komunikacioni podsistem.

## Pouzdanost, padovi i otkazi

- *Pouzdanost* (engl. *reliability*) je mera uspešnosti kojom se sistem pridržava autoritativne specifikacije svog ponašanja. Ta specifikacija bi, u idealnom slučaju, trebalo da bude kompletna, konzistentna, razumljiva i nedvosmislena. Važan deo te specifikacije su vremena odziva.
- *Pad* (engl. *failure*) je slučaj kada ponašanje sistema odstupa od navedene specifikacije.
- Padovi su eksterne manifestacije u ponašanju prouzrokovane internim *greškama* (engl. *error*) u sistemu. Mehanički ili algoritamski uzroci grešaka nazivaju se *otkazi* (engl. *fault*).
- Složeni sistemi se sastoje iz komponenata koje su opet složeni sistemi. Zbog toga se otkaz u jednom podsistemu manifestuje kao greška, ova kao pad tog podsistema, a to opet kao otkaz na višem nivou hijerarhije itd.
- Vrste otkaza su:
  - Tranzientni otkazi (engl. *transient faults*) su otkazi koji nastaju u nekom trenutku, postoje u sistemu i onda nestaju posle nekog vremena. Primer je otkaz hardverske komponente koja reaguje na prisutnost spoljašnjeg elektromagnetskog zračenja. Kada uzrok otkaza nestane, nestaje i greška.
  - Permanentni otkazi (engl. *permanent faults*) su otkazi koji nastaju u nekom trenutku i ostaju prisutni u sistemu sve dok se ne uklone popravkom sistema. Primeri: prekinuta žica ili greška u softveru.
  - Intermitentni otkazi (engl. *intermittent faults*) su tranzientni otkazi koji se dešavaju s vremena na vreme. Primer je hardverska komponenta koja je osetljiva na zagrevanje; ona radi neko vreme i kada se zagreje, isključi se, zatim se ohladi i ponovo uključi itd.

## Sprečavanje i tolerancija otkaza

- Dva osnovna pristupa povećanju pouzdanosti su sprečavanje i tolerancija otkaza.
- *Sprečavanje otkaza* (engl. *fault prevention*) pokušava da eliminiše svaku mogućnost pojave otkaza sistema i pre nego što on postane operativan.
- *Tolerancija otkaza* (engl. *fault tolerance*) omogućava da sistem nastavi da funkcioniše i u prisustvu otkaza.

### *Sprečavanje otkaza*

- Dva stepena sprečavanja otkaza: *izbegavanje otkaza* (engl. *fault avoidance*) i *otklanjanje otkaza* (engl. *fault removal*).
- Izbegavanje otkaza teži da ograniči unošenje otkaza tokom konstrukcije samog sistema pomoću:
  - korišćenja najpouzdanijih komponenata u okviru datih ograničenja u pogledu cene i performansi
  - korišćenje preciznih i pouzdanih tehnika za interkonekciju komponenata i sklapanje podsistema
  - pakovanje hardvera tako da bude zaštićen od spoljašnjih štetnih uticaja
  - rigorozna, ako ne i formalna specifikacija zahteva, upotreba dokazanih metoda projektovanja softvera i jezika koji podržavaju apstrakciju i modularnost
  - upotreba alata za softversko inženjerstvo kao pomoć za manipulisanje složenim softverskim komponentama.
- Uprkos svim ovim naporima, greške u dizajnu i hardvera i softvera uvek će biti moguće.
- Otklanjanje otkaza podrazumeva procedure za pronalaženje a zatim i uklanjanje grešaka u hardveru i softveru. Te procedure su, npr. pregled projekta, pregled koda (engl. *code review*), verifikacija programa i testiranje sistema.
- Testiranje sistema nikada ne može biti potpuno iscrpno da eliminiše sve greške zato što:
  - test nikada ne može da dokaže da grešaka nema, nego samo da ih ima
  - RT sisteme je često nemoguće testirati u realnim uslovima
  - većina testova se vrši na sistemu koji radi u simulacionom režimu, a teško je garantovati da je simulacija adekvatna realnim uslovima
  - greške uzrokovane pogrešnom specifikacijom zahteva se ne mogu manifestovati sve dok sistem ne postane operativan.
- I pored upotrebe svih tehnika softverske verifikacije i testiranja, hardverske komponente će otkazivati u radu, pa je pristup sprečavanja otkaza neuspešan jer:
  - učestanost ili trajanje popravki nije prihvatljivo, ili
  - sistem prosto nije fizički dostupan za pronalaženje i uklanjanje otkaza (npr. bespilotna svemirska letilica).
- Zbog svega toga, uspešnija alternativa je *tolerancija otkaza* (engl. *fault tolerance*).

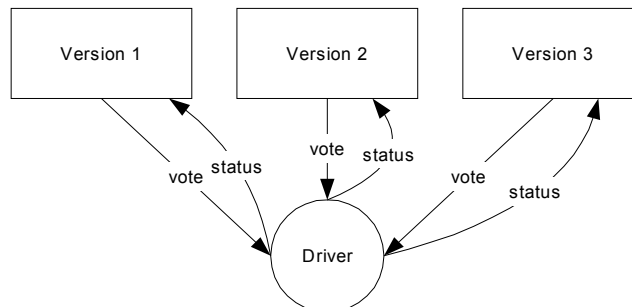
### *Tolerancija otkaza*

- *Tolerancija otkaza* (engl. *fault tolerance*) omogućava da sistem nastavi da funkcioniše i u prisustvu otkaza.
- Sve tehnike tolerancije otkaza zasnivaju se na ekstra elementima koji su uvedeni u sistem radi detekcije i oporavka od otkaza – *redundansa* (engl. *redundancy*). Ovi elementi ne bi bili potrebni u idealnom sistemu.

- Cilj je minimizovati redundansu a maksimizovati pouzdanost, pod zadatim ograničenjima u pogledu performansi i cene. Dodatno unete komponente obično negativno utiču na performanse i cenu sistema.
- Pristupi u hardverskoj redundansi: statička i dinamička redundansa.
- Statička redundansa podrazumeva da su u konstrukciju sistema uvedene dodatne hardverske komponente koje treba da sakriju efekte grešaka.
- Najpoznatiji primer statičke redundanse je tzv. Trostruko Modularna Redundansa (engl. *triple modular redundancy*, TMR), ili u opštem slučaju N-Modularna Redundansa: tri (ili N) identične komponente rade nezavisno i proizvode svaka svoj rezultat; posebna komponenta upoređuje rezultate i usvaja onaj većinski, ukoliko se rezultati razlikuju; rezultat koji odstupa se odbacuje. Ovaj pristup pretpostavlja da greška nije zajednička (npr. greška u dizajnu) nego je npr. tranzijentna. Uočavanje grešaka iz više od jedne komponente zahteva NMR.
- Dinamička redundansa se ugrađuje u komponentu tako da ukazuje da postoji greška u njenom rezultatu. Ova redundansa obezbeđuje samo detekciju greške, dok otklanjanje greške mora da obezbedi druga komponenta. Primeri: kontrolne sume (engl. *checksums*) pri prenosu podataka ili bitovi parnosti u memorijskim ćelijama.
- Softverska redundansa:
  - Statička: *N-Version* programiranje
  - Dinamička: detekcija greške, *oporavak od greške unazad* (engl. *backward error recovery*) i *oporavak od greške unapred* (engl. *forward error recovery*).

## N-Version Programiranje

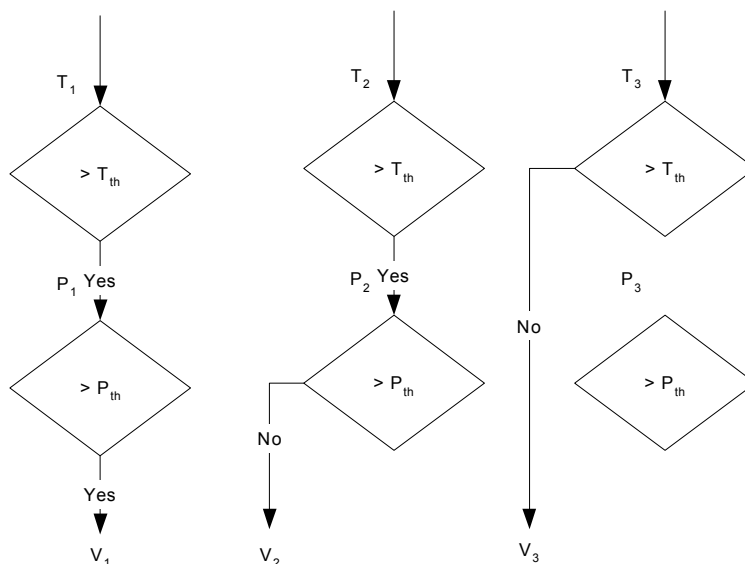
- Uspeh tehnika statičke redundanse u hardveru (TMR i NMR) inspirisalo je upotrebu sličnih pristupa i za rešavanje problema otkaza u softveru. Međutim, razlika je u tome što se softver ne haba i ne troši tokom upotrebe, pa je njegovo ponašanje nepromenjeno, već je osnovni uzrok njegovorg otkaza nastao tokom njegove realizacije, a ne eksploatacije.
- Zbog toga se osnovna ideja *N-version* programiranja zasnova na *nezavisnoj* realizaciji  $N$  (gde je  $N \geq 2$ ) funkcionalno ekvivalentnih verzija programa iz iste početne specifikacije. Nezavisna realizacija  $N$  programa podrazumeva da  $N$  pojedinaca ili grupa proizvodi  $N$  verzija softvera *bez* međusobnog uticaja (interakcije). Zbog toga se ovaj pristup naziva i *raznolikost dizajna* (engl. *design diversity*).
- Kada se napravi  $N$  verzija programa, ti programi se izvršavaju uporedo, sa istim ulaznim podacima, dok njihove izlazne rezultate upoređuje tzv. *driver* proces. U principu, rezultati bi trebalo da budu identični. Ukoliko postoji razlika, usvaja se većinski, pod uslovom da takav postoji.



- *N-version* programiranje se oslanja na pretpostavku da se program može u potpunosti, konzistentno i nedvosmisleno specifikovati, kao i da programi koji su razvijeni nezavisno

i otkazuju nezavisno. Drugim rečima, da ne postoje korelacije između grešaka u nezavisno razvijenim programima.

- Ovo obično podrazumeva realizaciju programa u različitim programskim jezicima, i/ili u različitim okruženjima (prevodioci, biblioteke, izvršna okruženja itd.).
- *Driver* proces ima zadatak da: (a) proziva svaku od verzija, (b) čeka na sve rezultate i (c) upoređuje rezultate i donosi odluku. Ukoliko postoje razlike u rezultatu, *driver* proces može da donese odluku da nastavi sve procese, da neki od njih prekine, ili da promeni jednu ili više manjinskih rezultata u većinski.
- Problem u odlučivanju može da bude poređenje rezultata različitih verzija. Naime, ukoliko su rezultati npr. celobrojni ili stringovi (nizovi znakova), rezultati mogu i treba da budu identični. Međutim, ukoliko su rezultati realni brojevi, njihove racionalne aproksimacije mogu da budu različite u različitim verzijama, a da ipak sve budu korektne.
- Zbog toga je u nekim slučajevima potrebno *neegzaktno glasanje* (engl. *inexact voting*). Jedno jednostavno rešenje je da se u takvim slučajevima izračunava srednja vrednost različitih rezultata i odstupanje svakog od te srednje vrednosti, pri čemu se definiše maksimalna granica tog odstupanja.
- Problem kod neegzaktog glasanja može da bude i tzv. *problem nekonzistentnog poređenja* (engl. *inconsistent comparison problem*), kada aplikacija treba da donosi odluke na osnovu zaokruženih rezultata. Na primer, sledeće tri verzije programa koji kontroliše temperaturu i pritisak u odnosu na granične zadate vrednosti, zbog tri približne vrednosti koje su sve različite, ali blizu praga, donose tri različite odluke, pri čemu se sve mogu smatrati ispravnim:



- Problemi mogu da nastanu i kod neegzaktog poređenja, jer se vrednosti mogu opet nalaziti blizu praga tolerancije. Slično, postoje problemi kod kojih postoji više tačnih rešenja (npr. rešenje kvadratne jednačine).
- Uspeh *N-version* programiranja jako zavisi od sledećih činilaca:
  - Inicijalna specifikacija. Većina grešaka u softveru jeste posledica neadekvatne specifikacije. Greška u specifikaciji će se manifestovati u svih  $N$  verzija implementacije.
  - Nezavisnost realizacije. Eksperimenti koji su imali zadatak da provere hipotezu o nezavisnosti otkaza nezavisno realizovanih programa dali su konfliktne rezultate. Osim toga, u slučaju kada je neki deo specifikacije složen, postoji velika



verovatnoća da će on dovesti do nerazumevanja zahteva. U tom slučaju će sve realizacije biti pogrešne. Ako se pri tom deo odnosi na ulazne podatke koji se retko pojavljuju, testiranje sistema će verovatno propustiti da uoči zajedničke greške u realizaciji.

- Adekvatan budžet. Dominantan trošak u razvoju RT sistema je razvoj softvera. Ukoliko se realizuje program u npr. tri verzije, postavlja se pitanje da li se trostruki trošak razvoja isplati i da li se on više isplati nego trostruko pouzdaniji razvoj jedne verzije (rigoroznije metode razvoja i testiranja).
- Zbog svega ovoga, iako *N-version* programiranje može značajno da doprinese povećanju pouzdanosti sistema, treba ga upotrebljavati pažljivo i samo u slučajevima visokih zahteva za pouzdanošću, i to u saradnji sa drugim tehnikama povećanja pouzdanosti.

## Dinamička softverska redundansa

- *N-version* programiranje je softverski ekvivalent *statičke*, maskirajuće redundanse, jer se otkaz komponente sakriva od spoljašnjosti i svaka verzija ima statičke relacije sa drugim verzijama i *driver* procesom.
- Kod *dinamičke* redundanse, redundantna komponenta se aktivira samo ukoliko dođe do otkaza u osnovnoj komponenti. Ovaj pristup ima sledeće faze:
  - *Detekcija greške* (engl. *error detection*): većina otkaza će se na kraju manifestovati kao greške. Nijedan pristup oporavku od otkaza ne može da se upotrebi ukoliko greška nije otkrivena.
  - *Izolacija i procena štete* (engl. *damage confinement and assessment*): kada se detektuje greška, potrebno je proceniti do koje mere je sistem ugrožen i oštećen. Kašnjenje od trenutka nastanka otkaza do trenutka detekcije greške znači da se pogrešna informacija možda proširila i na druge delove sistema.
  - *Oporavak od greške* (engl. *error recovery*): tehnike oporavka od greške treba da teže da transformišu sistem koji je oštećen u stanje u kome će sistem nastaviti normalno operisanje (možda sa degradiranim funkcionalnošću).
  - *Tretman otkaza* (engl. *fault treatment*): greška je samo simptom otkaza. Iako je šteta možda nadoknađena, otkaz možda još uvek postoji i može uzrokovati novu grešku u daljem radu sistema, sve dok se sistem ne popravi.

### Detekcija greške

- Tehnike detekcije greške se mogu klasifikovati u sledeće kategorije:
  - Detekcije u okruženju. U ovu kategoriju spadaju detekcije grešaka u okruženju u kome se program izvršava. To mogu biti greške koje detektuje hardver, kao što su greške tipa "*illegal instruction executed*" ili "*arithmetic overflow*" ili "*memory access violation*". Tu spadaju i greške koje uočava izvršno okruženje (engl. *runtime environment*), kao što su greške tipa "*value out of range*" ili "*array bound error*" ili "*null pointer dereferencing*".
  - Detekcije u aplikaciji. Ovde spadaju sledeće tehnike provere grešaka unutar same aplikacije:
    - Provere pomoću replika (engl. *replication checks*): *N-version* programiranje uočava greške poređenjem rezultata *N* nezavisnih replika.
    - Vremenske provere (engl. *timing checks*): (a) *watchdog timer* je proces koji aplikacija restartuje periodično, ukoliko ispravno radi; ukoliko dođe do greške, aplikacija "odluta" i ne restartuje ovaj proces, tako da njemu vreme

ističe i on generiše signal o nastanku greške. (b) kontrola prekoračenja rokova (engl. *deadline miss*); ukoliko se raspoređivanje radi u okruženju, ovo se može smatrati greškom detektovanom u okruženju.

- Reverzne provere (engl. *reversal checks*) su moguće u komponentama u kojima postoji jedan-na-jedan preslikavanje ulaza na izlaz; u tom slučaju se izlazni rezultat može proveriti inverznom funkcijom i poređenjem dobijenog argumenta sa ulaznim. Npr. komponenta koja izračunava koren broja može se proveriti prostim kvadriranjem izlaza (primetiti da je potrebno neegzaktno poređenje zbog zaokruživanja).
- Provere pomoću kodova (engl. *coding checks*) se koriste za kontrolu grešaka u podacima, npr. *parity*, *checksum*, CRC i sl.
- Provere razumnosti (engl. *reasonableness checks*) se oslanjaju na znanje o internoj konstrukciji sistema. One proveravaju stanje sistema i vrednosti izlaza u odnosu na to šta se sa njima treba raditi. Ove uslove provere razumnosti (engl. *assertions*) u programe ugrađuju sami programeri. To su logički izrazi nad programskim varijablama koji treba da budu zadovoljeni u određenim trenucima izvršavanja programa.
- Strukturne provere (engl. *structural checks*) se odnose na provere integriteta struktura podataka, npr. povezanosti listi. Oslanjaju se na tehnike kao što su brojanje referenci na objekte, redundantni pokazivači itd.
- Dinamičke provere razumnosti (engl. *dynamic reasonableness checks*) se oslanjaju na pretpostavku da se npr. dva rezultata dobijena sa nekog analognog ulaza u dva bliska vremenska trenutka ne mogu mnogo razlikovati zbog prirode ulaznog signala.

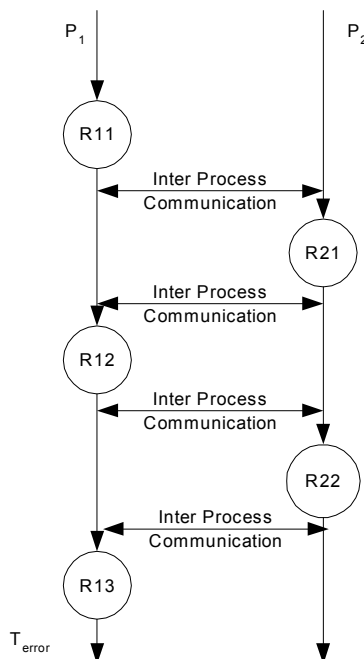
Treba primetiti da se mnoge od ovih provera mogu vršiti u hardveru i da se tada mogu smatrati greškama iz okruženja.

### ***Izolacija i procena štete***

- Tehnike procene štete su tesno povezane sa tehnikama izolacije greške, jer se od trenutka nastanka otkaza do trenutka detekcije greške pogrešna informacija možda proširila po drugim delovima sistema i okruženja.
- Izolacija štete se zasniva na strukturiranju sistema tako da se minimizuje šteta nastala zbog otkazane komponente.
- Jedna tehnika izolacije je *modularizacija* uz *enkapsulaciju*, pri čemu se sistem deli na module koji interaguju samo kroz jasno definisane i kontrolisane interfejse, dok su njihove implementacije sakrivene i nedostupne spolja. To doprinosi da se greška nastala u jednom modulu teže proširi i na druge. Ovo je *statički* pristup izolaciji.
- Dinamički pristup izolaciji podržavaju *atomične akcije* (engl. *atomic actions*). Atomična akcija je *izolovana* (engl. *isolated*) u smislu da za vreme njenog izvršavanja nema interakcija sa ostatkom sistema. Ona je i *nedeljiva* (engl. *indivisible*), u smislu da se ona u celini izvršava, sa svim efektima njenog celokupnog izvršavanja; ukoliko dođe do otkaza u toku njenog izvršavanja, efekti na sistem i okruženje ne postoje. One se koriste da prevedu sistem iz jednog u drugo konzistentno stanje.

### Oporavak od greške

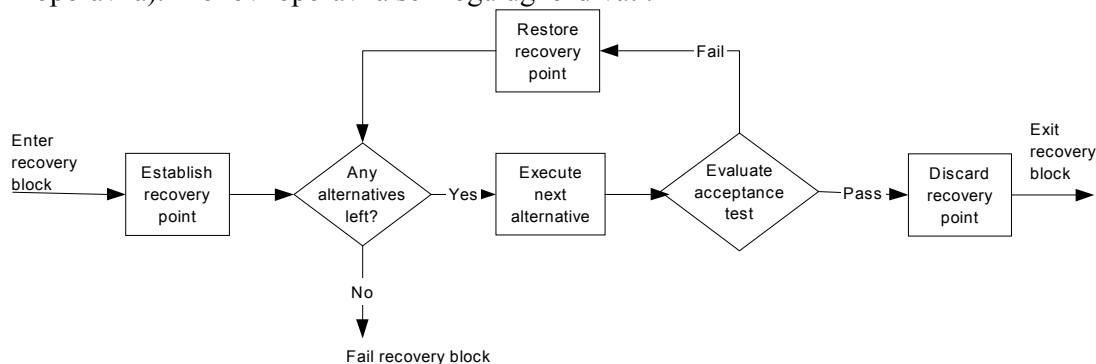
- Proces oporavka od greške je verovatno najvažniji element tolerancije otkaza. On mora da sistem sa greškom dovede u stanje normalnog operisanja, mada možda uz degradiranu funkcionalnost.
- Postoje dva pristupa oporavku od greške: *oporavak unapred* (engl. *forward error recovery*, FER) i *oporavak unazad* (engl. *backward error recovery*, BER).
- FER tehnike pokušavaju da nastave izvršavanje od stanja sa greškom, selektivno korigujući stanje sistema. To uključuje aktivnosti koje treba da svaki aspekt kontrolisanog okruženja koji je oštećen ili opasan učine sigurnim. Ove aktivnosti su sasvim specifične za dati sistem i zavise od preciznosti procene lokacije i uzroka greške. Na primer, korišćenje redundantnih pokazivača koji se koriste kao rezerva za oštećene, ili samokorigujućih kodova.
- BER tehnike se zasnivaju na vraćanju sistema u sigurno prethodno stanje pre nastanka greške i izvršavanje alternativne sekcije programa.
- Tačka restauracije sistema se naziva *tačka oporavka* (engl. *recovery point*) i predstavlja trenutak u kome se stanje sistema (koje se smatra konzistentnim i sigurnim) pamti radi kasnije eventualne restauracije. Kada dođe do greške, stanje sistema se restaurira na poslednju prođenu tačku oporavka (engl. *rollback*).
- Prednost ovakve tehnike je što ne zavisi od uzroka i lokacije greške, već se njen efekat jednostavno poništava. Zato se ova tehnika može koristiti i za oporavak od grešaka koje se ne mogu predvideti, kao što su greške u implementaciji.
- Međutim, BER tehnike imaju i značajne nedostatke:
  - Ne može se poništiti efekat promene okruženja (npr. jednom ispaljeni projektil ne može se zaustaviti).
  - Implementacija pamćenja stanja može da bude veoma složena i zahtevna po pitanju vremena u toku izvršavanja.
  - Kod konkurentnih procesa koji međusobno komuniciraju moguć je tzv. *domino efekat* (engl. *domino effect*): u primeru na slici, ukoliko greška u naznačenom trenutku nastane u procesu P1, samo on se razmotava do tačke oporavka R13; međutim, ukoliko greška nastane u procesu P2, zbog potrebe poništavanja efekta međusobne razmene informacija i uticaja na drugi proces, nastaje domino efekat jer se oba procesa moraju razmotavati unazad do jako daleke tačke; u najgorem slučaju, svi procesi koji interaguju se moraju potpuno poništiti, što može biti jako skupo.



- Zbog svega što je rečeno, obe tehnike FER i BER imaju svoje prednosti i nedostatke. Najčešće se one upotrebljavaju kombinovano. FER tehnike su krajnje zavisne od aplikacije i neće biti dalje razmatrane.

## Blokovi oporavka

- *Blok oporavka* (engl. *recovery block*) predstavlja jezičku podršku za BER.
- Blok oporavka je jezički konstrukt koji je *blok* kao u standardnom programskom jeziku, osim što se njegov ulazak implicitno smatra *tačkom oporavka* (engl. *recovery point*), a njegov izlaz je *test prihvatljivosti* (engl. *acceptance test*).
- Test prihvatljivosti se koristi da bi se proverilo da li je sistem u prihvatljivom stanju posle izvršavanja bloka, odnosno njegovog *primarnog modula*.
- Ako test prihvatljivosti nije zadovoljen, program se restaurira do tačke oporavka i izvršava se alternativni modul, itd. Ako nijedan alternativni modul ne dovede do prolaska testa, oporavak se preduzima na višem nivou (u eventualnom okružujućem bloku oporavka). Blokovi oporavka se mogu ugnežđivati.



- Moguća sintaksa ovog konstrukta:

```

ensure <acceptance test>
by
  <primary module>
else by
  <alternative module>
else by
  <alternative module>
...
else by
  <alternative module>
else error;

```

- Nijedan komercijalni i rašireni jezik ne podržava direktno ovaj konstrukt, ali se on može realizovati postojećim konstruktima i mehanizmima (npr. izuzecima). Ključni problem u realizaciji je pamćenje i restauriranje tačke oporavka.
- Na primer, rešavanje diferencijalnih jednačina može se obaviti tzv. eksplicitnim Kutta metodom, koji je brz ali neprecizan za određene jednačine, ali i implicitnim Kutta metodom, koji je skuplji ali uspešno rešava složene jednačine. Sledeći primer obezbeđuje da se efikasniji metod upotrebljava sve dok daje zadovoljavajuće rezultate, a da se u složenim situacijama koristi složeniji i pouzdaniji metod; osim toga, ovaj pristup može i da toleriše potencijalne greške u implementaciji eksplicitne metode, ukoliko je test prihvatljivosti dovoljno fleksibilan:

```

ensure RoundingErrorWithinAcceptableTolerance
by
  ExplicitKuttaMethod
else by
  ImplicitKuttaMethod
else error;

```

- Test prihvatljivosti obezbeđuje detekciju greške koja omogućuje iskorišćenje redundanse u softveru. Dizajn testa prihvatljivosti je ključan za efikasnost ove tehnike. Potrebno je pronaći balans između ugrađivanja razumljivih testova prihvatljivosti u program i troškova njegovog izvršavanja, pri čemu se čuva pouzdanost.
- Treba primetiti da se ovde radi o testu *prihvatljivosti*, a ne *korektnosti*, što znači da komponenta može da radi zadovoljavajuće i sa degradiranom funkcionalnošću.
- Sve ranije navedene tehnike detekcije grešaka mogu se koristiti za formiranje testova prihvatljivosti. Treba obratiti pažnju da pogrešno implementiran test prihvatljivosti može da dovede do situacije u kojoj se greške provlače nezapaženo.

### ***Poređenje između N-version programiranja i blokova oporavka***

- *N-version* programiranje (NVP) se oslanja na statičku redundansu u kojoj *N* verzija softvera radi istovremeno. Blokovi oporavka (RB) se oslanjaju na dinamičku redundansu u kojoj se alternativni moduli aktiviraju samo ukoliko se test prihvatljivosti ne prođe.
- Obe varijante uključuju posebne troškove razvoja, jer obe zahtevaju realizaciju *N* različitih komponentata. NVP još zahteva realizaciju *driver* procesa. RB zahteva realizaciju testa prihvatljivosti.
- Tokom izvršavanja, NVP generalno zahteva *N* puta veću količinu resursa nego jedna varijanta. Iako RB zahtevaju samo jedan skup resursa, potrebni su dodatni resursi i vreme izvršavanja za uspostavljanje i restauraciju tačaka oporavka.
- Obe tehnike iskorišćavaju raznovrsnost izrade da bi tolerisale nepredvidive greške. Međutim, nijedna od njih ne može da se bori protiv grešaka zbog neispravne specifikacije zahteva.

- NVP koristi glasanje za detekciju greške, dok RB koristi test prihvatljivosti. Tamo gde je egzaktno ili neegzaktno glasanje moguće, glasanje je generalno jeftinije nego izvršavanje testa. Međutim, u slučaju kada ono nije moguće, testovi su bolje rešenje.
- RB ne mogu poništiti efekat na okruženje, ali se mogu strukturirati tako da se promene okruženja ne rade u okviru blokova oporavka. NVP nema taj problem jer se smatra da su verzije modula izolovane i promene sredine se vrše samo po donošenju odluke.

---

# Izuzeci i njihova obrada

---

## Dinamička redundansa i izuzeci

- *Izuzetak* (engl. *exception*) se može definisati kao pojava greške.
- Greška se generalno definiše kao manifestacija devijacije ponašanja od specifikacije. Greška može biti predvidiva (npr. vrednost izvan zadatog opsega) ili nepredvidiva (npr. greška u implementaciji softverske komponente). Šta se smatra greškom je često relativno pitanje.
- Ukazivanje pozivaocu neke operacije na izuzetak koji je u njoj nastao naziva se *podizanje* (engl. *raising*) ili *signaliziranje* (engl. *signalling*) ili *bacanje* (engl. *throwing*) izuzetka.
- Odgovor pozivaoca na podignut izuzetak naziva se *obrada* (engl. *handling*) izuzetka.
- Obrada izuzetka je FER mehanizam, jer ne postoji implicitni povratak sistema u sigurno stanje. Umesto toga, kontrola se vraća pozivaocu koji treba da preduzme proceduru za oporavak. Međutim, mehanizam obrade izuzetaka može se iskoristiti za implementaciju BER mehanizama.
- Mehanizam obrade izuzetaka može se koristiti za:
  - rešavanje abnormalnih uslova koji se pojavljuju u okruženju;
  - tolerisanje grešaka u dizajnu softvera;
  - opštu detekciju i oporavak od grešaka.
- Mehanizam obrade izuzetaka u nekom programskom jeziku treba da zadovolji sledeće opšte uslove:
  - Kao i ostali mehanizmi jezika, i on treba da bude jednostavan za razumevanje i upotrebu.
  - Kod za obradu izuzetka ne sme da bude rogovatan tako da ugrozi razumevanje osnovnog toka bez pojave grešaka.
  - Mehanizam treba da bude implementiran tako da ne uzima posebne resurse i vreme u toku izvršavanja programa ukoliko izuzetaka nema, već samo u slučaju njihove pojave.
  - Mehanizam treba da podjednako tretira izuzetke nastale u okruženju (npr. od strane hardvera) i u samom programu (npr. zbog ne zadovoljavanja uslova korektnosti).

## Obrada izuzetaka bez posebne jezičke podrške

- Ukoliko sam jezik ne podržava obradu izuzetaka, onda se neregularne situacije mogu rešavati pomoću posebnih povratnih vrednosti iz potprograma, kao što je to uobičajeno u proceduralnim jezicima:

```
if (functionCall(someParams)==OK) {  
    // Normal operation  
} else {  
    // Error handling code  
}
```

- Iako je ovaj mehanizam jednostavan, on ne zadovoljava ostale zahteve: kod je nejasan, jer ne razdvaja jasno neregularnu od regularne staze, uključuje režijske troškove u svakom slučaju i nije jasno kako se obrađuju izuzeci iz okruženja.
- U nastavku će se tretirati moderni koncepti i mehanizmi obrade izuzetaka u programskim jezicima, i to najviše u objektno orijentisanim jezicima C++ i Java.

## Izuzeci i njihova reprezentacija

- Kao što je ranije naglašeno, postoje dva izvora izuzetaka: okruženje programa (hardver ili izvršno okruženje) i sam program (aplikacija).
- Izuzetak se može podići *sinhrono*, kao neposredni rezultat dela koda koji je pokušao neku neregularnu operaciju, ili *asinhrono*, što znači nezavisno od operacije koja se trenutno izvršava.
- Prema tome, postoje četiri kategorije izuzetaka:

- Detektovani od strane okruženja i podignuti sinhrono. Na primer, izuzeci tipa deljenja sa nulom ili prekoračenje granica niza.

Za C++, generisanje ovakvih izuzetaka nije garantovano, već zavisi od implementacije. Izvršno okruženje ne generiše ovakve izuzetke (nema nikakve provere u vreme izvršavanja), pa je sama aplikacija odgovorna za detekciju ovakvih grešaka.

Java dosledno generiše sve tipove ovakvih izuzetaka.

Na primer, sledeći kod na jeziku C++ može (ali ne mora) da podigne hardverski izuzetak koji nije uhvaćen od strane programa, dok se u jeziku Java podiže određeni izuzetak:

```
int a[N];
...
for (int i = 0; i<=N; i++)
    ...a[i]...
```

Slično, eksplicitna konverzija pokazivača na osnovnu klasu u pokazivač na izvedenu klasu u jeziku C++ ne generiše izuzetak ukoliko se iza pokazivača ne krije objekat željenog tipa, već će program imati nezgodnu grešku. Java generiše izuzetak u ovom slučaju u vreme izvršavanja. Na primer, sledeći kod na jeziku C++ neće podići izuzetak, već će uzrokovati nezgodnu grešku, dok će ekvivalentan kod na jeziku Java podići izuzetak:

```
Base* pb = new Base;
...
Derived* pd = (Derived*)pb;
```

- Detektovani od strane aplikacije i podignuti sinhrono. Na primer, nezadovoljenje uslova definisanih u samom programu. C++ i Java nemaju ugrađene mehanizme za implicitnu proveru uslova, već se oni jednostavno eksplicitno programiraju. Na primer:

```
if (! some_assertion) throw anException;
```

- Detektovani od strane okruženja i podignuti asinhrono. Na primer, pad napona ili signal o neispravnosti nekog nadzornog sistema.
- Detektovani od strane aplikacije i podignuti asinhrono. Na primer, jedan proces može da obrađuje neku grešku i time ugrozi drugi proces koji će zbog toga zakasniti i probiti svoj rok ili završiti neregularno.



- Asinhroni izuzeci se obično nazivaju asinhronim signalima (engl. *asynchronous signal*) i pojavljuju se u kontekstu konkurentnog programiranja.
- Za sinhronu izuzetku, bitan element mehanizma je kako se oni deklariraju. Postoje dva najčešća pristupa:
  - Izuzetak je eksplicitno deklarirana konstanta (jezik Ada).
  - Izuzetak je objekat nekog tipa (C++ i Java).
- U jeziku C++, izuzetak može biti instanca bilo kog tipa, i ugrađenog i korisničkog (objekat klase). U trenutku podizanja izuzetka, kreira se jedan privremeni, bezimni objekat (smešta se u statički alociranu memoriju) i inicijalizuje izrazom iza naredbe `throw`. U trenutku pokretanja bloka koji obrađuje izuzetak (engl. *handler*) iza naredbe `catch`, formira se argument tog bloka kao automatski lokalni objekat tog bloka koji se inicijalizuje navedenim privremenim objektom. Semantika inicijalizacije u oba slučaja je ista kao i kod prenosa argumenata u funkciju. Na primer, izuzetak se može podići ovako:

```
if (...) throw HighTemperatureException(...);
```

ili ovako:

```
HighTemperatureException* preparedException = new HighTemperatureException;
...
if (...) throw preparedException;
```

ili ovako:

```
if (...) throw new HighTemperatureException(...);
```

a onda uhvatiti ovako:

```
catch(HighTemperatureException e) {
    ...
}
```

odnosno ovako:

```
catch(HighTemperatureException* e) {
    ...
    delete e;
}
```

- U jeziku Java, izuzetak je instanca klase direktno ili indirektno izvedene iz klase `Throwable`. Njegov životni vek je isti kao i životni vek bilo kog drugog objekta klase (nastaje eksplicitno i uništava se implicitno). Od izvora izuzetka do onoga ko ga hvata prosleđuje se, zapravo, samo referenca na taj objekat. Na primer, izuzetak se može podići ovako:

```
HighTemperatureException preparedException = new HighTemperatureException;
...
if (...) throw preparedException;
```

Ili ovako:

```
if (...) throw new HighTemperatureException(...);
```

a uhvatiti ovako:

```
catch(HighTemperatureException e) {
    ...
}
```

- U jeziku C++ deklaracija funkcije može da uključuje i spisak tipova izuzetaka koje ta funkcija može da podigne, bilo direktno (iz svog koda), bilo indirektno (iz ugnežđenih poziva drugih funkcija, tranzitivno). Ukoliko ovaj spisak postoji, funkcija ne može podići izuzetak nijednog drugog tipa osim onih sa spiska, ni direktno ni indirektno. Ukoliko taj

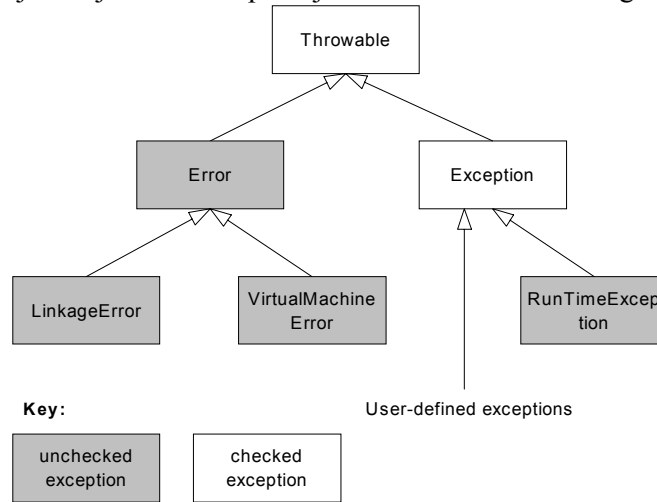
spisak ne postoji, funkcija može podići izuzetak bilo kog tipa (zbog kompatibilnosti sa jezikom C i starijim verzijama jezika C++). Na primer:

```
void checkTemperature () throw (HighTemperatureException*);
```

- U jeziku Java postoji sličan princip, osim što ukoliko funkcija ne sadrži spisak izuzetaka, onda ona ne može podići nijedan. Ovaj pristup je značajno restriktivniji, ali time i pouzdaniji. Na primer:

```
public void checkTemperature () throws HighTemperatureException {...}
```

- U jeziku C++ ne postoji predefinisana hijerarhija klasa ili tipova za izuzetke, već oni mogu biti bilo kog tipa.
- U jeziku Java, hijerarhija izuzetaka počinje klasom `Throwable` i izgleda kao na datoj slici.



- U jeziku Java, izuzeci koji se ne proveravaju (engl. *unchecked*) ne moraju da se navode u listi izuzetaka koje funkcija podiže, oni se uvek podrazumevaju. Ove izuzetke podiže okruženje (npr. linker ili virtuelna mašina, ili su to greške u izvršavanju, npr. neregularna konverzija reference ili prekoračenje granica niza). Ostali tipovi izuzetaka se moraju deklarirati u deklaraciji funkcije ukoliko ih ona direktno ili indirektno podiže (engl. *checked*). Korisnički definisani tipovi izuzetaka realizuju se kao klase izvedene iz klase `Exception`.

## Obrada izuzetka

- U nekim programskim jezicima kod za obradu izuzetka (engl. *exception handler*) može se pridružiti pojedinačnoj naredbi. Međutim, u većini blok-strukturiranih jezika, kod za obradu izuzetaka je vezan za blok.
- U jezicima C++ i Java, blok koji može da izazove izuzetak koji treba uhvatiti deklarira se kao `try` blok:

```
try {
    // Neki kod koji može podići izuzetak
} catch (ExceptionType e) {
    // Exception Handler za izuzetke tipa ExceptionType
}
```

- Prilikom izvršavanja, najpre se izvršava blok iza `try`. Ukoliko se tokom njegovog izvršavanja ne podigne izuzetak (ni u ugnežđenim pozivima), ceo konstrukt se završava. Ukoliko dođe do izuzetka koji je tipa navedenog u argumentu `catch` naredbe (uz pravilo objektne supstitucije, tj. prihvataju se i izvedeni tipovi), izvršava se kod za obradu u bloku iza `catch`. Ukoliko tip iza `catch` ne prihvata tip podignutog izuzetka, ovaj izuzetak se prosleđuje u okružujući kontekst poziva (prvi dinamički okružujući `try` blok unutar iste funkcije ili pozivaoca).
- Oblast obrade izuzetka i njena granularnost zapravo određuju preciznost određivanja lokacije izvora greške. Ukoliko je potrebna finija granularnost, u sam objekat izuzetka mogu se ugraditi dodatne informacije kao parametri izuzetka, ili sam tip izuzetka može nosi dodatnu informaciju.
- Na primer, potrebno je očitati tri senzora za temperaturu, pritisak i protok, pri čemu kod svakog čitanja može nastupiti izuzetak zbog prekoračenja opsega dozvoljenih vrednosti; zatim treba preduzeti odgovarajuće akcije. Naivan pristup bi bio sledeći:

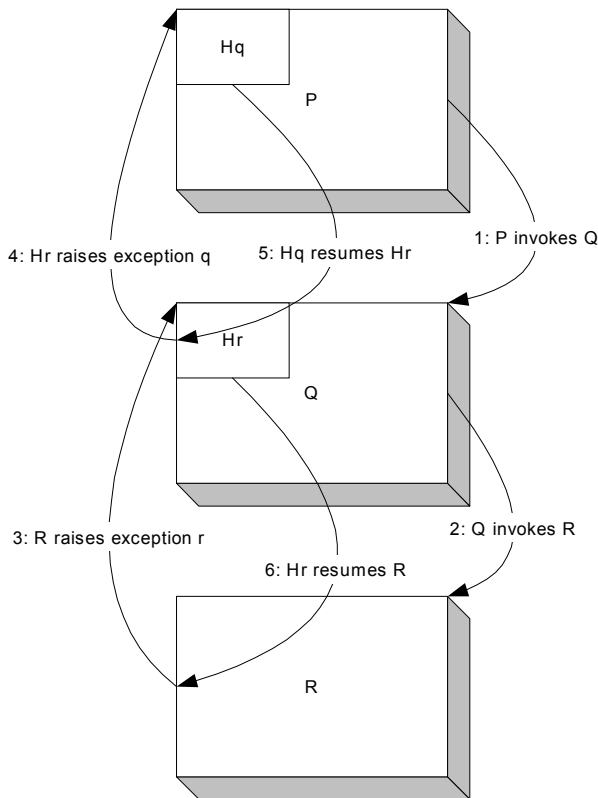
```
try {  
    // Read temperature sensor  
} catch (Overflow ovf) {  
    // Handler for temperature overflow  
}  
try {  
    // Read pressure sensor  
} catch (Overflow ovf) {  
    // Handler for pressure overflow  
}  
try {  
    // Read flow sensor  
} catch (Overflow ovf) {  
    // Handler for flow overflow  
}  
// Now adjust the temperature, pressure, and flow  
// according to the requirements
```

- Međutim, ovakav pristup je nepregledan i zamoran. Loše je što je kod za obradu izuzetka nepregledno učešljan u sam osnovni regularni tok. Bolji pristup je da se informacija o vrsti izuzetka ugradi u njegov tip ili u njegove attribute. Na primer:

```
try {  
    // Read temperature sensor  
    // Read pressure sensor  
    // Read flow sensor  
}  
catch (TemperatureOverflow ovf) {  
    // Handler for temperature overflow  
}  
catch (PressureOverflow ovf) {  
    // Handler for pressure overflow  
}  
catch (FlowOverflow ovf) {  
    // Handler for flow overflow  
}  
// Now adjust the temperature, pressure, and flow  
// according to the requirements
```

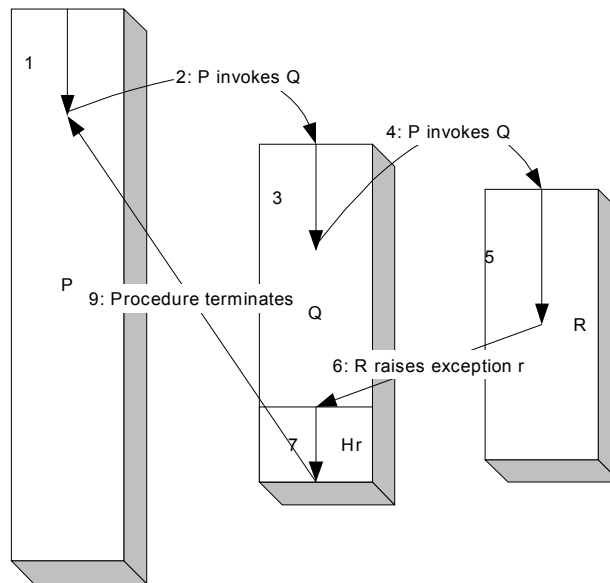
## Propagacija izuzetka

- Pitanje je kako tretirati situaciju kada bloku u kome može nastati izuzetak nije pridružen blok za njegovu obradu. U jezicima postoji nekoliko pristupa:
  - Tretirati to kao grešku u programu koja se može otkriti u vreme prevođenja. Ovo je dosta restriktivan pristup, jer vrlo često nije moguće obraditi izuzetak tamo gde nastaje; na primer, kod izuzetaka koji nastaju zbog neregularnih argumenata procedure.
  - Propagirati izuzetak u vreme izvršavanja u okružujuću proceduru (pozivaoca), tj. redom naviše, sve dok se ne nađe pozivalac koji ima kod za obradu izuzetka.
- Osnovno pitanje kod obrade izuzetaka je da li podizač izuzetka treba da nastavi svoje izvršavanje posle obrade izuzetka ili ne. Postoje dva generalna pristupa: *model povratka* (engl. *resumption model*) i *model terminacije* (engl. *termination model*).
- Kod *resumption* modela, podizač izuzetka nastavlja svoje izvršavanje posle obrade izuzetka, od mesta na kome je nastao izuzetak.
- Na primer, neka postoje tri procedure, *P*, *Q* i *R* koje se redom pozivaju kao na donjoj slici. Neka *R* podiže izuzetak *r* za koji nema kod za obradu. Takav kod za obradu se traži u pozivajućoj proceduri *Q* koja ima kod za obradu *Hr*. Pokreće se izvršavanje tog koda, ali neka i on podiže izuzetak *q* za koji se kod za obradu pronalazi u proceduri *P*. Posle završetka ove obrade, izvršavanje se vraća na mesto nastanka izuzetka u *Hr* koji se završava, a zatim se nastavlja izvršavanje procedure *R* od mesta na kome je nastao izuzetak.



- Ovaj mehanizam se može najlakše razumeti ako se kod za obradu izuzetka posmatra kao procedura koja se implicitno poziva pri nastanku izuzetka i koja se dinamički traži u pozivajućim procedurama.

- Problem sa ovim pristupom je što je često nemoguće popraviti otkaze u okruženju tako da se nastavi izvršavanje. Problem je restaurirati kontekst izvora izuzetka posle završetka obrade izuzetka. Zbog toga je implementacija strogo *resumption* modela veoma složena. Kompromis postoji u vidu *retry* modela, kod koga se ceo blok izvršava ispočetka.
- *Termination* model podrazumeva da se kontrola ne vraća na mesto nastanka izuzetka posle njegove obrade, nego se blok ili procedura koja sadrži kod za obradu završava a kontrola vraća pozivaocu. Kada je kod za obradu unutar bloka, kontrola se posle obrade nastavlja od prve naredbe iza tog bloka. Primer je pokazan na donjoj slici:



- Jezici C++ i Java podržavaju *termination* model na potpuno isti način. Jedina specifičnost jezika C++ je da se pri prenosu kontrole na kod za obradu pozivaoca "razmotava stek", tj. propisno uništavaju (pozivom destruktora) svi automatski objekti klasa kreirani od trenutka ulaska u blok čiji je kod za obradu pronađen, do trenutka nastanka izuzetka. U jeziku Java za ovo nema potrebe, jer ne postoje automatski objekti klasa.
- Moguć je i hibridni model, kod koga kod za obradu izuzetka može da odluči da li je greška popravljiva ili ne i da li će se izvršavanje nastaviti na mestu nastanka izuzetka ili ne.

## Vežbe

### 4.1

Posmatra se program koji sekvencijalno obrađuje podatke iz neke ulazne datoteke. Diskutovati da li nailazak na kraj datoteke treba smatrati izuzetkom.

### 4.2

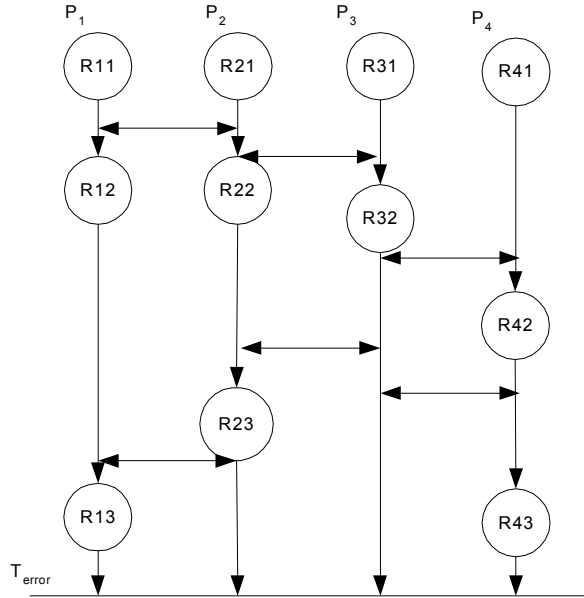
Korišćenjem blokova oporavka realizovati pouzdan program koji sortira niz celih brojeva.

### 4.3

Uraditi isti zadatak korišćenjem *N-version* programiranja.

## 4.4

Na slici je šematski prikazano izvršavanje četiri procesa  $P_1$  do  $P_4$  sa svojim tačkama oporavka  $R_{ij}$  i međusobnom komunikacijom. Objasniti šta se dešava u slučaju nastanka greške u naznačenom trenutku  $T_{error}$ , ako je greška nastala u jednom od ta četiri procesa. Odgovor dati za svaki pojedinačni slučaj od ta četiri.



## 4.5

Realizovati strukturu FIFO reda čekanja kod koga operacija uzimanja elementa iz praznog reda podiže izuzetak. Ilustrovati primerom korišćenje takve strukture.

## 4.6

Korišćenjem mehanizma obrade izuzetaka, pokazati kako se blokovi oporavka mogu implementirati u jeziku C++.

## 4.7

Dat je interfejs klase koja omogućava čitanje znakova sa terminala, a poseduje i operaciju za odbacivanje svih preostalih znakova na tekućoj liniji. Operacije ove klase podižu izuzetak tipa `IOError`.

```
class CharacterIO {
public:
    char get    () throw (IOError);
    void flush () throw (IOError);
};
```

Klasa `Look` sadrži funkciju `read()` koja pretražuje tekuću ulaznu liniju i traži sledeće znakove interpunkcije: zarez, tačku i tačku-zarez. Ova funkcija će vratiti prvi sledeći znak interpunkcije na koji naiđe ili podići izuzetak tipa `IllegalPunctuation` ukoliko naiđe na znak koji nije alfa-numerički. Ako se tokom učitavanja znakova sa ulazne linije dogodi izuzetak tipa `IOError`, on će biti prosleđen pozivaocu funkcije `read()`. Kada vrati regularan znak interpunkcije, ova funkcija treba da odbaci preostale znakove na liniji.

Realizovati klasu `Look` korišćenjem klase `CharacterIO`. U klasi `Look` zatim realizovati i funkciju `getPunctuation()` koja će uvek vratiti sledeći znak interpunkcije, bez obzira na postojanje neregularnih znakova i grešaka na ulazu. Pretpostaviti da je ulazni tok neograničen, da uvek sadrži neki znak interpunkcije i da se greške na ulazu dešavaju slučajno.

#### 4.8

Posmatra se neki sistem za kontrolu procesa u kome se gas zagreva u nekoj posudi. Posuda je okružena hladnjakom koji smanjuje njegovu temperaturu odvođenjem toplote preko tečnosti za hlađenje. Postoji takođe i slavina čijim se otvaranjem gas ispušta u atmosferu. Interfejs klase koja upravlja ovim procesom dat je u nastavku. Zbog sigurnosnih razloga, klasa prepoznaje nekoliko neregularnih uslova koji se korisniku dojavljuju putem izuzetaka. Izuzetak tipa `HeaterStuckOn` signalizira da grejač nije moguće isključiti jer se prekidač zaglavio. Izuzetak tipa `TemperatureStillRising` signalizira da hladnjak nije u stanju da snizi temperaturu gasa povećanjem protoka tečnosti za hlađenje. Konačno, izuzetak tipa `ValveStuck` signalizira da se slavina zaglavila i da je nije moguće otvoriti. Operacija `panic()` podiže znak za uzbunu.

```
class TemperatureControl {
public:
    void heaterOn ();
    void heaterOff () throw (HeaterStuckOn);
    void increaseCoolant () throw (TemperatureStillRising);
    void openValve () throw (ValveStuck);
    void panic();
};
```

Korišćenjem ove klase napisati funkciju koja, kada se pozove, pokušava da isključi grejač. Ukoliko se on zaglavio, treba povećati protok hladnjaka. Ukoliko temperatura i dalje raste, potrebno je otvoriti slavinu za izbacivanje gasa. Ukoliko ni to ne uspe, treba dići znak za uzbunu.

# **V Osnove konkurentnog programiranja**



---

# Konkurentnost i procesi

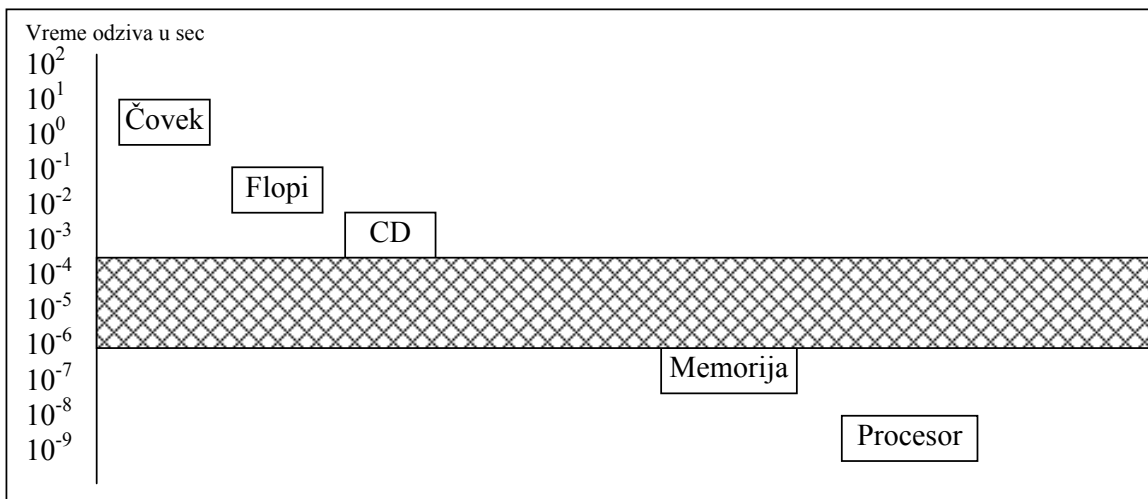
---

## Konkurentno programiranje

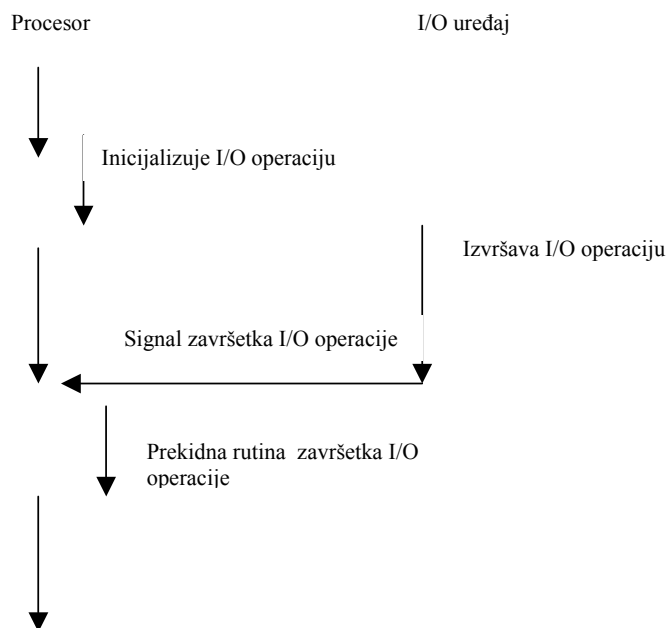
- Praktično svi RT sistemi su inherentno *uporedni* (*konkurentni*, engl. *concurrent*). To znači da oni uporedno kontrolišu različite komponente sistema, kao i da spoljašnje pobude i događaji stižu uporedo u vremenu.
- Da bi program za konkurentni sistem bio lakši za projektovanje i razumevanje, potrebno je i da programski jezik bude *konkurentan*. Konkurentni jezici poseduju veću moć izražavanja i lakoću upotrebe nego sekvencijalni jezici, jer programeru nude konstrukte kojima se direktno može modelovati konkurentnost sistema.
- *Konkurentno programiranje* (engl. *concurrent programming*) je naziv za programsku notaciju i tehnike za izražavanje potencijalnog paralelizma dešavanja i izvršavanja, kao i za rešavanje problema sinhronizacije i komunikacije koji zbog toga nastaju.
- Implementacija paralelizma u hardveru i softveru jeste pitanje koje je principijelno nezavisno od konkurentnog programiranja.
- Konkurentno programiranje je paradigma koja omogućava da se potencijalni paralelizam izrazi na apstraktan način, ne razmatrajući pitanje stvarne implementacije paralelizma.
- Polazna osnova koja važi u konkurentnom programiranju je, dakle, da se ništa ne pretpostavlja o stvarnoj implementaciji paralelizma, tj. da se ne pretpostavlja da se konkurentni procesi zaista izvršavaju fizički paralelno u vremenu ili ne, i na koji se način oni sekvencijalizuju ili prepliću, ukoliko se ne izvršavaju fizički paralelno (nego se izvršavaju npr. na samo jednom procesoru).

### *Zašto je potrebno konkurentno programiranje?*

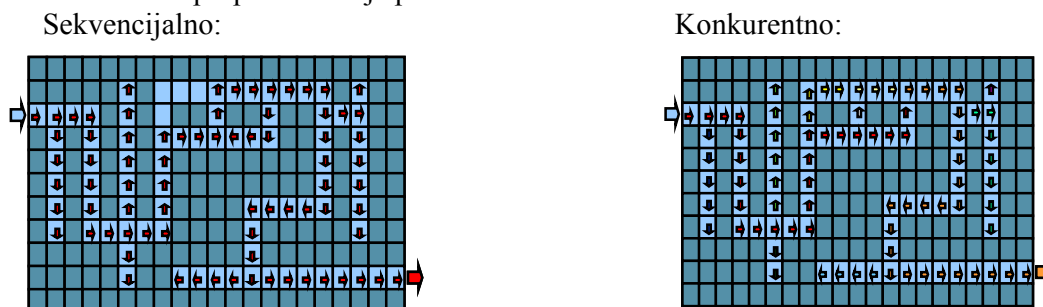
- Konkurentno programiranje obezbeđuje način da se:
  - iskoristi procesorsko radno vreme, kako bi procesor mogao da radi neki koristan posao dok čeka da okolni sistemi, koji imaju daleko veće vreme odziva, izvrše svoje zadatke:



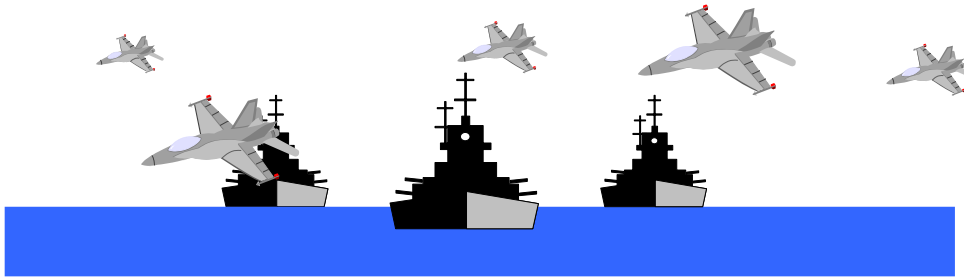
- iskoristi paralelizam u radu procesora i ulazno/izlaznih uređaja:



- izrazi potencijalni paralelizam, tako da zadatak može da rešava više procesora; npr. pronalaženje prolaza kroz lavirint:



- modeluje paralelizam u realnom svetu:



- Alternativa konkurentnom programiranju je upotreba sekvencijalnih programskih tehnika. Međutim, takav pristup ima niz slabosti:
  - Programer mora da konstruiše sistem tako da on uključuje ciklično izvršavanje programske sekvence, kako bi obradio različite uporedne aktivnosti.
  - To komplikuje ionako složen programerski zadatak i nameće programeru baratanje programskim strukturama koje su irelevantne za kontrolu aktivnosti.
  - Rezultujući programi su nepregledni i teški za razumevanje i održavanje.
  - Dekompozicija problema je složeniya.
  - Teže je postići paralelno izvršavanje programa na više procesora.
  - Ugradnja koda za obradu grešaka je problematičnija.

## Pojam procesa

- Konkurentni program je skup autonomnih sekvencijalnih procesa koji se izvršavaju (logički) paralelno.
- *Proces* (engl. *process*) je sekvenca akcija koja se izvršava uporedno sa ostalim procesima. Proces ima svoj *tok kontrole* (engl. *thread of control*).
- Proces predstavlja deo programskog koda zajedno sa strukturama podataka koje omogućuju uporedno (konkurentno) izvršavanje tog programskog koda sa ostalim procesima. Koncept procesa omogućuje izvršavanje dela programskog koda tako da su svi podaci koji su definisani kao lokalni za taj deo programskog koda zapravo lokalni za jedno izvršavanje tog koda, i da se njihove instance razlikuju od instanci istih podataka istih delova tog koda, ali različitih procesa. Ova lokalnost podataka procesa pridruženih jednom izvršavanju datog koda opisuje se kao izvršavanje datog dela koda u *kontekstu* nekog procesa.
- Stvarna implementacija (tj. izvršavanje) skupa procesa obično ima jedan od sledeća tri oblika:
  1. *Multiprogramiranje* (engl. *multiprogramming*): izvršavanje procesa se multipleksira na jednom procesoru.
  2. *Multiprocesiranje* (engl. *multiprocessing*): izvršavanje procesa se multipleksira na više procesora koji imaju zajedničku memoriju (tzv. *multiprocesorski sistem*).
  3. *Distribuirano procesiranje* (engl. *distributed processing*): izvršavanje procesa se multipleksira na više procesora koji nemaju zajedničku memoriju, nego komuniciraju preko komunikacionih veza (tzv. *distribuirani sistem*).
- U terminologiji konkurentnog programiranja razlikuju se obično dve vrste procesa:
  1. Proces na nivou operativnog sistema (engl. *process*). Ovakvi procesi nazivaju se ponekad "teškim" (engl. *heavy-weight*) procesima. Ovakav proces kreira se nad celim programom, ili ponekad nad delom programa. Pri tome svaki proces ima sopstvene (lokalne) instance svih vrsta podataka u programu: statičkih (globalnih), automatskih (lokalnih za potprograme) i dinamičkih.

2. Proces u okviru jednog programa. Ovakvi procesi nazivaju se "lakim" (engl. *light-weight*) ili *nitima* (engl. *thread*). Niti se kreiraju nad delovima jednog programa, najčešće kao tok izvršavanja koji polazi od jednog potprograma. Svi dalji ugneždeni pozivi ostalih potprograma izvršavaju se u kontekstu date niti. To znači da sve niti unutar jednog programa dele statičke (globalne) i dinamičke podatke. Ono što ih razlikuje je lokalnost automatskih podataka: *svaka nit poseduje svoj kontrolni stek* na kome se kreiraju automatski objekti (alokacioni blokovi potprograma). Kaže se zato da sve niti poseduju *zajednički adresni prostor*, ali različite *tokove kontrole*. Niti međusobno sarađuju bez ikakve intervencije operativnog sistema.

- Termin *proces* upotrebljava se u oba značenja, kao opšti pojam i kao teški proces, u zavisnosti od konteksta.
- Termin *zadatak* (engl. *task*) se upotrebljava u različitim značenjima, u nekom kontekstu označava pojam procesa kao opšti pojam, u nekim operativnim sistemima označava teški proces, a u nekim jezicima označava nit.
- Postoji dugogodišnja debata o tome da li programski jezik treba da uključi koncepte konkurentnog programiranja ili da to ostavi operativnom sistemu:
  - Ukoliko jezik uključuje konkurentnost, onda multipleksiranje izvršavanja procesa obavlja "virtuelna mašina" koja je sastavni deo izvršnog okruženja programa (engl. *runtime environment*, *runtime support system*) koga je proizveo prevodilac. Na primer, jezici Ada i Java imaju ugrađenu konkurentnost.
  - Ukoliko jezik ne podržava konkurentnost, onda se ona može dograditi posebnim delom koda, ili osloniti na usluge operativnog sistema (sistemske pozivi). Jezici C i C++ nemaju ugrađenu konkurentnost. Ona se može obezbediti:
    - Izgradnjom sopstvenog izvršnog okruženja, kao što će to biti izvedeno u ovom kursu.
    - Oslanjanjem na usluge operativnog sistema preko nekog programskog interfejsa (engl. *application programming interface*, API) za sistemske pozive iz aplikativnog programa. Primer jednog standardnog API-a za jezik C za konkurentno programiranje na operativnim sistemima koji to podržavaju jeste POSIX (*Portable Operating System based on Unix*).
- Raspoređivanje izvršavanja procesa na procesoru (engl. *scheduling*) svakako utiče na vremensko ponašanje programa. Međutim, *logičko ponašanje dobro konstruisanog programa ne sme da zavisi od implementacije raspoređivanja u izvršnom okruženju*.
- Drugim rečima, posmatrano sa strane programa, podrazumeva se da izvršno okruženje raspoređuje procese nedeterministički. Nikakva pretpostavka o determinisanom raspoređivanju se ne sme uzeti u obzir ukoliko se želi korektan, prenosiv program. Sva neophodna međuzavisnost između procesa mora se rešiti logikom samog programa i to:
  - sinhronizacijom između procesa (engl. *process synchronization*)
  - komunikacijom između procesa (engl. *interprocess communication*, IPC).

## Predstavljanje procesa

### *Korutine*

- *Korutina* (engl. *coroutine*) je nalik potprogramu, samo što se dozvoljava da kontrola toka eksplicitno pređe sa jedne korutine na drugu naredbom `resume` koja direktno imenuje korutinu na koju se prebacuje kontrola toka.

- Kada korutina izvrši `resume`, privremeno se prekida njeno izvršavanje, a nastavlja se izvršavanje imenovane korutine od mesta gde je ona prekinuta sa `resume`.
- Programski jezik Modula-2 podržava koncept korutine.
- Primer:

```
coroutine A (...)      coroutine B (...)      coroutine C (...)
begin
  ...
  resume B;
  ...
  resume B;
  ...
  resume C;
  ...
end;

coroutine B (...)
begin
  ...
  resume C;
  ...
  resume A;
  ...
end;

coroutine C (...)
begin
  ...
  resume A;
  ...
  resume B;
  ...
end;
```

- Za korutine nema potrebe za izvršnim okruženjem, jer one same eksplicitno međusobno uređuju redosled izvršavanja. Semantika korutina eksplicitno određuje da u jednom trenutku može da se izvršava samo jedna korutina. Zbog toga one ne predstavljaju istinsko konkurentno izvršavanje, već su samo njegova preteča.

### ***Fork/Join***

- Ovaj pristup ne obezbeđuje jasno razlikovanje i eksplicitno deklarisanje procesa, već jednostavno podrazumeva postojanje dve vrste naredbi:
  - *Fork* (na engleskom "viljuška") označava da imenovani potprogram startuje konkurentno izvršavanje od trenutka izvršavanja naredbe *fork*, uporedo sa tokom kojim nastavlja sekvenca u kojoj se nalazi *fork*;
  - *Join* (na engleskom "spoj") označava da tok kontrole u kome se nalazi *join* mora da sačeka završetak toka kontrole koji je imenovan u naredbi *join*.
- Ovaj koncept zastupljen je u operativnom sistemu Unix i svim njegovim varijantama, pa je podržan i u POSIX-u, s tim da *fork* kod njih označava započinjanje konkurentnog izvršavanje nove instance procesa koji je isti kao proces u kome je *fork*. Programski jezik Mesa takođe podržava ovaj koncept.
- Primer:

```
function F ...;
begin
  ...
end F;

procedure P;
begin
  ...
  c := fork F;
  ...
  j := join c;
  ...
end;
```

- Primer za POSIX: koliko procesa kreira sledeći deo koda?

```
for (i=0; i<10; i++) {
  pid[i] = fork();
}
wait ... // Equivalent to "join"
```

- Iako fleksibilan, ovaj koncept nije posebno zgodan jer ne podržava strukturirano i jasno deklarisanje procesa.

***Cobegin***

- Konstrukt *cobegin* (ili *parbegin* ili *par*) predstavlja strukturirani način da se označi konkurentno izvršavanje skupa naredbi.
- Struktura konstrukta *cobegin* je ista kao i struktura složene naredbe (bloka) u strukturiranim programskim jezicima, samo što se skup naredbi u bloku ne izvršavaju sekvencijalno kao kod tradicionalnih jezika, već konkurentno. Izvršavanje konstrukta *cobegin* se završava kada se završi izvršavanje svih njegovih komponenata.
- Primer:

```
cobegin
  S1;
  S2;
  ...
  Sn
coend;
```

- Programski jezici Concurrent Pascal i occam2 podržavaju ovaj konstrukt.

***Eksplicitno deklarisanje procesa na jeziku Ada***

- Jezik Ada poseduje pojam procesa koji se eksplicitno deklarise kao programski modul nalik potprogramu, samo što je njegovo izvršavanje konkurentno.
- U jeziku Ada se proces naziva *task* i definiše slično potprogramu. Važno je uočiti da se time ne definiše *kada* će se proces izvršavati:

```
task Process;           // Task declaration

task body Process is    // Task definition
begin
  ...
end;
```

- Jezik Ada podržava i statičko i dinamičko definisanje procesa. Drugim rečima, jedan *task* može predstavljati i samo jednu jedinu instancu procesa u vreme izvršavanja (prethodni primer), ali i tip, tj. skup instanci koje se mogu kreirati dinamički, u toku izvršavanja:

```
task type P;
...
ptrP : access P; // A reference/pointer to the task type P
...
ptrP := new P;   // Creation/activation of the process of the type P
```

***Konkurentno izvršavanje na jeziku Java***

- Jezik Java podržava niti (engl. *thread*) na potpuno objektno orijentisani način. Skup niti koje se mogu kreirati u toku izvršavanja deklarise se kao klasa izvedena iz bibliotečne klase *Thread*. Kod (telo) niti definiše se kao polimorfna operacija *run()* ove klase.
- U toku izvršavanja programa, mogu se kreirati objekti ove klase na uobičajeni način. Međutim, izvršavanje niti se mora eksplicitno startovati pozivom operacije *start()*.
- Klasa čiji objekti predstavljaju procese, tj. imaju sopstvenu nit toka kontrole, naziva se *aktivna klasa* (engl. *active class*). Prema UML notaciji, aktivna klasa se označava kao i obična klasa, samo što je okvirna linija zadebljana.
- Primer:

```
public class UserInterface
{
```

```

    public int newSetting (int Dim) { ... }
    ...
}
public class Arm
{
    public void move(int dim, int pos) { ... }
}
...
UserInterface ui = new UserInterface();
Arm robot = new Arm();
...
public class Control extends Thread
{
    private int dim;
    public Control(int dimension)
    {
        super();
        dim = dimension;
    }
    public void run()
    {
        int position = 0;
        int setting;
        while(true)
        {
            robot.move(dim, position);
            setting = ui.newSetting(dim);
            position = position + setting;
        }
    }
}
...
final int xPlane = 0; // final indicates a constant
final int yPlane = 1;
final int zPlane = 2;

Control c1 = new Control(xPlane);
Control c2 = new Control(yPlane);
Control c3 = new Control(zPlane);
c1.start();
c2.start();
c3.start();

```

- Nit u Javi završava izvršavanje kada se dogodi nešto od sledećeg:
  - kada se završi operacija `run()`, bilo normalno, bilo zbog neobrađenog izuzetka;
  - kada se pozove operacija `stop()` klase `Thread` za taj objekat; ovoj operaciji se može proslediti referenca na objekat tipa `Throwable`, koji će biti podignut kao izuzetak u određenoj niti; operacija `stop()` nije sigurna, jer bezuslovno oslobađa sve objekte koje je nit zaključala; ovu operaciju zato u principu ne treba pozivati.

## Interakcija između procesa

- Osnovni problem povezan sa konkurentnim programiranjem leži u interakciji između procesa. Procesi su retko nezavisni. Korektno ponašanje konkurentnog programa zavisi od interakcije između procesa, koja može biti:

- *sinhronizacija* (engl. *synchronization*) predstavlja zadovoljavanje ograničenja u pogledu preplitanja akcija različitih procesa (npr. neka akcija jednog procesa mora da se dogodi pre neke akcije drugog procesa i sl.); ovaj termin može da se odnosi i na uži smisao simultanog dovođenja više procesa u predefinisano stanje;
  - *komunikacija* (engl. *communication*) predstavlja razmenu informacija između procesa.
- Pojmovi sinhronizacije i komunikacije su međusobno povezani, jer neki oblici komunikacije podrazumevaju prethodnu sinhronizaciju, dok se sinhronizacija može smatrati komunikacijom bez razmene sadržaja.
- Međuprocesna komunikacija se obično zasniva na jednom od dva modela:
  - *deljena promenljiva* (engl. *shared variable*) je objekat kome može pristupiti više procesa; komunikacija se tako obavlja razmenom informacija preko deljene promenljive;
  - *razmena poruka* (engl. *message passing*) podrazumeva eksplicitnu razmenu informacija između procesa u vidu poruka koje putuju od jednog do drugog procesa preko nekog agenta.
- Izbor modela komunikacije je stvar konstrukcije programskog jezika ili operativnog sistema. On ne implicira nikakav poseban metod implementacije. Naime, deljene promenljive je jednostavno implementirati ukoliko procesori koji izvršavaju uporedne procese imaju zajedničku memoriju. Međutim, deljene promenljive se mogu, doduše nešto teže, implementirati i na distribuiranom sistemu. Slično, razmena poruka se može implementirati i na multiprocesorskim i na distribuiranim sistemima.
- Osim toga, aplikacija iste funkcionalnosti se u principu može programirati korišćenjem bilo kog od ova dva modela, s tim da je za neke probleme neki model pogodniji.
- Ova dva modela detaljnije su obrađena u naredne dve glave.

## Implementacija niti

- U ovom kursu biće prikazana realizacija jednog jezgra (engl. *kernel*) višeprocesnog sistema sa nitima (engl. *multithreaded kernel*) na jeziku C++, koji može da služi kao izvršno okruženje za konkurentni korisnički program. Ova realizacija biće nazivana "školsko Jezgro".
- Kod ovog sistema aplikativni sloj softvera treba da se poveže sa kodom Jezgra da bi se dobio kompletan izvršni program koji ne zahteva nikakvu softversku podlogu (nije mu potreban operativni sistem). Ovo je pogodno za ugrađene sisteme. Prema tome, veza između višeg sloja softvera i Jezgra je na nivou izvornog koda i zajedničkog povezivanja, a ne kao kod složenih operativnih sistema, gde se sistemski pozivi rešavaju u vreme izvršavanja, najčešće preko softverskih prekida.
- Na nivou aplikativnog sloja softvera, želja je da se postigne semantika analogna onoj na jeziku Java: nit je aktivan objekat koji poseduje sopstveni tok kontrole (sopstveni stek poziva).
- Nit se može kreirati nad nekom globalnom funkcijom. Pri tome se svi ugnežđeni pozivi, zajedno sa svojim automatskim objektima, odvijaju u sopstvenom kontekstu te niti. Na primer, korisnički program može da izgleda ovako:

```
#include "kernel.h" // uključivanje deklaracija Jezgra
#include <iostream.h>

void threadBody () {
    for (int i=0; i<3; i++) {
```



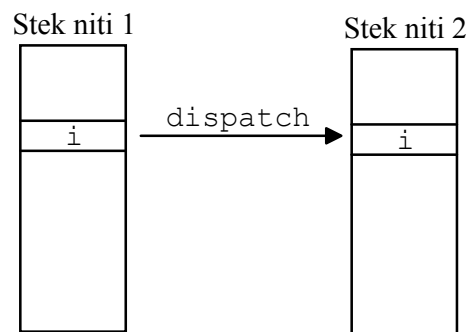
```

        cout<<i<<"\n";
        dispatch();
    }
}

void userMain () {
    Thread* t1=new Thread(threadBody);
    Thread* t2=new Thread(threadBody);
    t1->start();
    t2->start();
    dispatch();
}

```

- Funkcija `threadBody()` predstavlja telo (programski kod) niti. Funkcija `dispatch()` predstavlja eksplicitni zahtev za preuzimanje (dodelu procesora drugoj niti), slično kao kod koncepta korutina, samo što se ne imenuje nit koja preuzima izvršavanje.
- Funkcija `userMain()` predstavlja početnu nit aplikativnog, korisničkog dela programa. Funkcija `main()` nalazi se u nadležnosti Jezgra, pa korisniku nije dostupna. Jezgro inicijalno kreira jednu nit nad obaveznom funkcijom `userMain()`.
- U ovom primeru obe niti imaju isti kod, ali svaka poseduje svoj stek poziva, na kome se kreira automatski objekat `i`. Kada dođe do preuzimanja u funkciji `dispatch()`, Jezgro obezbeđuje pamćenje konteksta tekuće niti i povratak konteksta niti koja je izabrana za tekuću, što znači da se dalje izvršavanje odvija na steku nove tekuće niti. Ovo prikazuje sledeća slika:



### Promena konteksta

- Promena konteksta (engl. *context switch*) podrazumeva da procesor napušta kontekst izvršavanja jednog procesa i prelazi u kontekst izvršavanja drugog procesa.
- Tipično se operativni sistemi konstruišu tako da postoje dva režima rada, koja su obično podržana i od strane procesora: sistemski i korisnički. U sistemskom režimu dozvoljeno je izvršavanje raznih sistemskih operacija, kao što je pristup do nekih područja memorije koji su zaštićeni od korisničkih programa. Osim toga, kada postoji mogućnost da se pojavi prekid kao posledica nekog spoljašnjeg događaja na koji sistem treba da reaguje, potrebno je da se sistemski delovi programa izvršavaju neprekidivo, bez promene konteksta, kako ne bi došlo do poremećaja sistemskih delova podataka. U realizaciji ovog Jezgra naznačena su mesta prelaska u sistemski i korisnički režim. Prelaz na sistemski režim obavlja funkcija `lock()`, a na korisnički funkcija `unlock()`. Sve kritične sistemske sekcije uokvirene su u par poziva ovih funkcija. Njihova realizacija je zavisna od platforme i za sada je prazna:

```
void lock    () {} // Switch to kernel mode
void unlock () {} // Switch to user mode
```

- Kada dolazi do promene konteksta, u najjednostavnijem slučaju eksplicitnog pomoću funkcije `dispatch()`, Jezgro treba da uradi sledeće:
  1. Sačuva kontekst niti koja je bila tekuća (koja se izvršavala, engl. *running*).
  2. Smesti nit koja je bila tekuća u red niti koje su spremne za izvršavanje (engl. *ready*).
  3. Izabere nit koja će sledeća biti tekuća iz reda niti koje su spremne.
  4. Povrati kontekst novoizabrane niti i nastavi izvršavanje.
- Čuvanje konteksta niti znači sledeće: vrednosti svih relevantnih registara procesora čuvaju se u nekoj strukturi podataka da bi se kasnije mogle povratiti. Ova struktura naziva se najčešće PCB (engl. *process control block*). Povratak konteksta znači smeštanje sačuvanih vrednosti registara iz PCB u same registre procesora.
- U registre spada pokazivač adrese naredne instrukcije PC (engl. *program counter*), koji čuva informaciju o lokalnosti toka izvršavanja niti (vezano za instrukcijski kod), i pokazivač steka (engl. *stack pointer*, SP), koji čuva informaciju vezanu za lokalnost podataka niti. Kada se u PC i SP povrate vrednosti sačuvane u PCB-u, dalje izvršavanje nastaviće od mesta gde je to izvršavanje prekinuto, tj. gde ukazuje sačuvani PC, i koristiće upravo stek na koji ukazuje SP, čime se postiže svojstvo konkurentnosti niti: lokalnost automatskih podataka, odnosno sopstveni tok kontrole.
- U standardnoj biblioteci jezika C (pa time i C++) definisane su dve funkcije koje obezbeđuju koncept korutina. Ove funkcije "sakrivaju" neposredno baratanje samim registrima procesora, pa se njihovim korišćenjem može dobiti potpuno prenosiv program.
- Deklaracije ovih funkcija nalaze se u `<setjmp.h>` i izgledaju ovako:

```
int  setjmp (jmp_buf context);
void longjmp (jmp_buf context, int value);
```

- Tip `jmp_buf` deklarisan je u istom zaglavlju i predstavlja zapravo PCB. To je struktura koja čuva sve relevantne, programski dostupne registre procesora čije su vrednosti bitne za kontekst izvršavanja C programa prevedenog pomoću datog prevodioca na datom procesoru.
- Funkcija `setjmp()` vrši smeštanje vrednosti registara u strukturu `jmp_buf`. Pri tom smeštanju ova funkcija vraća rezultat 0. Funkcija `longjmp()` vrši povratak konteksta sačuvanog u `jmp_buf`, što znači da izvršavanje vraća na poziciju steka koja je sačuvana pomoću odgovarajućeg `setjmp()`. Pri tome se izvršavanje nastavlja sa onog mesta gde je pozvana `setjmp()`, s tim da sada `setjmp()` vraća onu vrednost koju je dostavljena pozivu `longjmp()` (to mora biti vrednost različita od 0).
- Prema tome, pri čuvanju konteksta, `setjmp()` vraća 0. Kada se kontekst povrati iz `longjmp()`, dobija se efekat da odgovarajući `setjmp()` vraća vrednost različitu od 0. Veoma je važno da se pazi na sledeće: od trenutka čuvanja konteksta pomoću `setjmp()`, do trenutka povratka pomoću `longjmp()`, izvršavanje u kome je `setjmp()` *ne sme* da se vrati iz funkcije koja neposredno okružuje poziv `setjmp()`, jer bi se time stek narušio, pa povratak pomoću `longjmp()` dovodi do kraha sistema.
- Tipična upotreba ovih funkcija za potrebe realizacije korutina može da bude ovakva:

```
if (setjmp(runningThread->context)==0) {
    // Sačuvan je kontekst.
    // Može da se pređe na neki drugi,
    // i da se njegov kontekst povrati sa:
    longjmp(runningThread->context,1);
} else {
```

```
// Ovde je povraćen kontekst onoga koji je sačuvan u setjmp()
}
```

- U realizaciji Jezgra ovi pozivi su "upakovani" u OO okvire. Nit je predstavljena klasom `Thread` koja poseduje atribut tipa `jmp_buf` (kontekst). Funkcija članica `resume()` vrši povratak konteksta jednostavnim pozivom `longjmp()`. Funkcija članica `setContext()` čuva kontekst pozivom `setjmp()`. Kako se iz ove funkcije ne sme vratiti pre povratka konteksta, ova funkcija je samo logički okvir i mora biti prava *inline* funkcija, kako prevodilac ne bi generisao kod za poziv i povratak iz ove funkcije `setContext()`:

```
// WARNING: This function MUST be truely inline!
inline int Thread::setContext () {
    return setjmp(myContext);
}
```

```
void Thread::resume () {
    longjmp(myContext,1);
}
```

- Klasa `Scheduler` realizuje raspoređivanje. U njoj se nalazi red spremnih niti (engl. *ready*), kao i protokol raspoređivanja. Funkcija `get()` ove klase vraća nit koja je na redu za izvršavanje, a funkcija `put()` stavlja novu spremnu nit u red.
- Klasa `Scheduler` poseduje samo jedan jedini objekat u sistemu (engl. *Singleton*). Ovaj jedini objekat sakriven je unutar klase kao statički objekat. Otkrivena je samo statička funkcija `Instance()` koja vraća pokazivač na ovaj objekat. Konstruktor ove klase sakriven je od prisupa spolja. Na ovaj način korisnici klase `Scheduler` ne mogu kreirati objekte ove klase, već je to u nadležnosti same te klase, čime se garantuje jedinstvenost objekta. Osim toga, korisnici ove klase ne moraju da znaju ime tog jedinog objekta, već im je dovoljan interfejs same klase i pristup do statičke funkcije `Instance()`. Ovakav projektni šablon (engl. *design pattern*) naziva se *Singleton*.
- Najzad, statička funkcija `Thread::dispatch()` koju jednostavno poziva globalna funkcija `dispatch()` izgleda jednostavno:

```
void Thread::dispatch () {
    lock ();
    if (runningThread->setContext()==0) {

        // Context switch:
        Scheduler::Instance()->put(runningThread);
        runningThread = Scheduler::Instance()->get();
        runningThread->resume();

    } else {
        unlock ();
        return;
    }
}
```

- Treba primetiti sledeće: deo funkcije `dispatch()` iza poziva `setContext()`, a pre poziva `resume()`, radi i dalje na steku prethodno tekuće niti (pozivi funkcija klase `Scheduler`). Tek od poziva `resume()` prelazi se na stek nove tekuće niti. Ovo nije nikakav problem, jer taj deo predstavlja "đubre" na steku iznad granice koja je zapamćena u `setContext()`. Prilikom povratka konteksta prethodne niti, izvršavanje će se nastaviti od zapamćene granice steka, ispod ovog "đubreta".

### Raspoređivanje

- Kao što je opisano, klasa `Scheduler` realizuje apstrakciju koja obavlja skladištenje spremnih niti, kao i raspoređivanje. Pod raspoređivanjem se smatra izbor one niti koja je na redu za izvršavanje. Ovo obavlja funkcija članica `get()`. Funkcija `put()` smešta novu nit u red spremnih.
- U ovoj realizaciji obezbeđen je samo jednostavan kružni (engl. *round-robin*) raspoređivač, korišćenjem realizovanog reda čekanja.
- Klasa `Scheduler` je realizovana kao *Singleton*, što znači da ima samo jedan objekat. Ovaj objekat je zapravo lokalni statički objekat funkcije `Instance()`:

```
Scheduler* Scheduler::Instance () {  
    static Scheduler instance;  
    return &instance;  
}
```

### Kreiranje niti

- Nit je predstavljena klasom `Thread`. Kao što je pokazano, korisnik kreira nit kreiranjem objekta ove klase. U tradicionalnom pristupu nit se kreira nad nekom globalnom funkcijom programa. Međutim, ovaj pristup nije dovoljno fleksibilan. Naime, često je potpuno beskorisno kreirati više niti nad istom funkcijom ako one ne mogu da se međusobno razlikuju, npr. pomoću argumenata pozvane funkcije. Zbog toga se u ovakvim tradicionalnim sistemima često omogućuje da korisnička funkcija nad kojom se kreira nit dobije neki argument prilikom kreiranja niti. Ipak, broj i tipovi ovih argumenata su fiksni, definisanim samim sistemom, pa ovakav pristup nije u duhu jezika C++.
- U realizaciji ovog Jezgra, pored navedenog tradicionalnog pristupa, omogućen je i OO pristup u kome se nit može definisati kao aktivan objekat. Taj objekat je objekat neke klase izvedene iz klase `Thread` koju definiše korisnik. Nit se kreira nad polimorfnom operacijom `run()` klase `Thread` koju korisnik može da redefiniše u izvedenoj klasi. Na ovaj način svaki aktivni objekat iste klase poseduje sopstvene atribute, pa na taj način mogu da se razlikuju aktivni objekti iste klase (niti nad istom funkcijom). Suština je zapravo u tome da jedini (doduše skriveni) argument funkcije `run()` nad kojom se kreira nit jeste pokazivač `this`, koji ukazuje na čitavu strukturu proizvoljnih atributa objekta.
- Prema tome, interfejs klase `Thread` prema korisnicima izgleda ovako:

```
class Thread {  
public:  
  
    Thread ();  
    Thread (void (*body)());  
    void start ();  
  
protected:  
  
    virtual void run () {}  
  
};
```

- Konstruktor bez argumenata kreira nit nad polimorfnom operacijom `run()`. Drugi konstruktor kreira nit nad globalnom funkcijom na koju ukazuje pokazivač-argument. Funkcija `run()` ima podrazumevano prazno telo, tako da se i ne mora redefinisati, pa klasa `Thread` nije apstraktna.

- Funkcija `start()` služi za eksplicitno pokretanje niti. Implicitno pokretanje moglo je da se obezbedi tako što se nit pokreće odmah po kreiranju, što bi se realizovalo unutar konstruktora osnovne klase `Thread`. Međutim, ovakav pristup nije dobar, jer se konstruktor osnovne klase izvršava pre konstruktora izvedene klase i njenih članova, pa se može dogoditi da novokreirana nit počne izvršavanje pre nego što je kompletan objekat izvedene klase kreiran. Kako nit izvršava redefinisanu funkciju `run()`, a unutar ove funkcije može da se pristupa članovima, moglo bi da dođe do konflikta.
- Treba primetiti da se konstruktor klase `Thread`, odnosno kreiranje nove niti, izvršava u kontekstu one niti koja poziva taj konstruktor, odnosno u kontekstu niti koja kreira novu nit.
- Prilikom kreiranja nove niti ključne i kritične su dve stvari: 1) kreirati novi stek za novu nit i 2) kreirati početni kontekst te niti, kako bi ona mogla da se pokrene kada dođe na red.
- Kreiranje novog steka vrši se prostom alokacijom niza bajtova u slobodnoj memoriji, unutar konstruktora klase `Thread`:

```
Thread::Thread ()
: myStack(new char[StackSize]), //...
```

- Obezbeđenje početnog konteksta je mnogo teži problem. Najvažnije je obezbediti trenutak "cepanja" steka: početak izvršavanja nove niti na njenom novokreiranom steku. Ova radnja se može izvršiti direktnim smeštanjem vrednosti u SP. Pri tom je veoma važno sledeće. Prvo, ta radnja se ne može obaviti unutar neke funkcije, jer se promenom vrednosti SP više iz te funkcije ne bi moglo vratiti. Zato je ova radnja u programu realizovana pomoću makroa (jednostavne tekstualne zamene), da bi ipak obezbedila lokalnost i fleksibilnost. Drugo, kod procesora i8086 SP se sastoji iz dva registra (SS i SP), pa se ova radnja vrši pomoću dve asemblerske instrukcije. Prilikom ove radnje vrednost koja se smešta u SP ne može biti automatski podatak, jer se on uzima sa steka čiji se položaj menja jer se menja i (jedan deo registra) SP. Zato su ove vrednosti statičke. Ovaj deo programa je ujedno i jedini mašinski zavisani deo Jezgra i izgleda ovako:

```
#define splitStack(p) \
static unsigned int sss, ssp; \ // FP_SEG() vraća segmentni, a FP_OFF() \
sss=FP_SEG(p); ssp=FP_OFF(p); \ // ofsetni deo pokazivača; \
asm { \ // neposredno ugrađivanje asemblerskih \
mov ss,sss; \ // instrukcija u kod; \
mov sp,ssp; \ \
mov bp,sp; \ // ovo nije neophodno; \
add bp,8 \ // ovo nije neophodno; \
}
```

- Početni kontekst nije lako obezbediti na mašinski nezavisan način. U ovoj realizaciji to je urađeno na sledeći način. Kada se kreira, nit se označi kao "započinjuća" atributom `isBeginning`. Kada dobije procesor unutar funkcije `resume()`, nit najpre ispituje da li započinje rad. Ako tek započinje rad (što se dešava samo pri prvom dobijanju procesora), poziva se globalna funkcija `wrapper()` koja predstavlja "omotač" korisničke niti:

```
void Thread::resume () {
if (isBeginning) {
isBeginning=0;
wrapper();
} else
longjmp(myContext,1);
}
```

- Prema tome, prvi poziv `resume()` i poziv `wrapper()` funkcije dešava se opet na steku prethodno tekuće niti, što ostavlja malo "đubre" na ovom steku, ali iznad granice zapamćene unutar `dispatch()`.
- Unutar statičke funkcije `wrapper()` vrši se konačno "cepanje" steka, odnosno prelazak na stek novokreirane niti:

```
void Thread::wrapper () {
    void* p=runningThread->getStackPointer(); // vrati svoj SP
    splitStack(p);                          // cepanje steka

    unlock ();
    runningThread->run();                     // korisnička nit
    lock ();

    runningThread->markOver();                // nit je gotova,
    runningThread = Scheduler::Instance()->get(); // predaje se procesor
    runningThread->resume();
}
```

- Takođe je jako važno obratiti pažnju na to da ne sme da se izvrši povratak iz funkcije `wrapper()`, jer se unutar nje prešlo na novi stek, pa na steku ne postoji povratna adresa. Zbog toga se iz ove funkcije nikad i ne vraća, već se po završetku korisničke funkcije `run()` eksplicitno predaje procesor drugoj niti.
- Zbog ovakve logike, neophodno je da u sistemu uvek postoji bar jedna spremna nit. Uopšte, u sistemima se to najčešće rešava kreiranjem jednog "praznog", besposlenog (engl. *idle*) procesa, ili nekog procesa koji vodi računa o sistemskim resursima i koji se nikad ne može blokirati, pa je uvek u redu spremnih (tzv. demonski procesi, engl. *daemon process*). U ovoj realizaciji to će biti nit koja briše gotove niti, opisana u narednom odeljku.
- Na ovaj način, startovanje niti predstavlja samo njeno upisivanje u listu spremnih, posle označavanja kao "započinjuće":

```
void Thread::start () {
    //...
    fork();
}

void Thread::fork () {
    lock();
    Scheduler::Instance()->put(this);
    unlock();
}
```

### ***Ukidanje niti***

- Ukidanje niti je sledeći veći problem u konstrukciji Jezgra. Gledano sa strane korisnika, jedan mogući pristup je da se omogućí eksplicitno ukidanje kreirane niti pomoću njenog destruktora. Pri tome se poziv destruktora opet izvršava u kontekstu onoga ko uništava nit. Za to vreme sama nit može da bude završena ili još uvek aktivna. Zbog toga je potrebno obezbediti odgovarajuću sinhronizaciju između ova dva procesa, što komplikuje realizaciju. Osim toga, ovakav pristup nosi i neke druge probleme, pa je on ovde odbačen, iako je opštiji i fleksibilniji.
- U ovoj realizaciji opredeljenje je da niti budu zapravo aktivni objekti, koji se eksplicitno kreiraju, a implicitno uništavaju. To znači da se nit kreira u kontekstu neke druge niti, a da

zatim živi sve dok se ne završi funkcija `run()`. Tada se nit "sama" implicitno briše, tačnije njeno brisanje obezbeđuje Jezgro.

- Brisanje same niti ne sme da se izvrši unutar funkcije `wrapper()`, po završetku funkcije `run()`, jer bi to značilo "sečenje grane na kojoj se sedi": brisanje niti znači i dealokaciju steka na kome se izvršava sama funkcija `wrapper()`.
- Zbog ovoga je primenjen sledeći postupak: kada se nit završi, funkcija `wrapper()` samo označi nit kao "završenu" atributom `isOver`. Poseban aktivni objekat (nit) klase `ThreadCollector` vrši brisanje niti koje su označene kao završene. Ovaj objekat je nit kao i svaka druga, pa ona ne može doći do procesora sve dok se ne završi funkcija `wrapper()`, jer završni deo ove funkcije izvršava u sistemskom režimu.
- Klasa `ThreadCollector` je takođe *Singleton*. Kada se pokrene, svaka nit se "prijavi" u kolekciju ovog objekta, što je obezbeđeno unutar konstruktora klase `Thread`. Kada dobije procesor, ovaj aktivni objekat prolazi kroz svoju kolekciju i jednostavno briše sve niti koje su označene kao završene. Prema tome, ova klasa je zadužena tačno za brisanje niti:

```
void Thread::start () {
    ThreadCollector::Instance()->put(this);
    fork();
}
```

```
class ThreadCollector : public Thread {
public:
```

```
    static ThreadCollector* Instance ();
```

```
    void put (Thread*);
    int  count ();
```

```
protected:
```

```
    virtual void run ();
```

```
private:
```

```
    ThreadCollector ();
```

```
    Collection rep;
```

```
    static ThreadCollector* instance;
```

```
};
```

```
void ThreadCollector::run () {
    while (1) {

        int i=0;
        CollectionIterator* it = rep.getIterator();

        for (i=0,it->reset(); !it->isDone(); it->next(),i++) {
            Thread* cur = (Thread*)it->currentItem();
            if (cur->isOver) {
                rep.remove(i);
                delete cur;
                it->reset(); i=0;
                dispatch();
            }
        }
    }
}
```

```
    }  
}  
  
if (count()==1)  
    longjmp(mainContext,1); // return to main  
  
dispatch();  
}  
}
```

### ***Pokretanje i gašenje programa***

- Poslednji veći problem pri konstrukciji Jezgra jeste obezbeđenje ispravnog pokretanja programa i povratka iz programa. Problem povratka ne postoji kod ugrađenih (engl. *embedded*) sistema jer oni rade neprekidno i ne oslanjaju se na operativni sistem. U okruženju operativnog sistema kao što je PC DOS/Windows, ovaj problem treba rešiti jer je želja da se ovo Jezgro koristi za eksperimentisanje na PC računaru.
- Program se pokreće pozivom funkcije `main()` od strane operativnog sistema, na steku koji je odvojen od strane prevodioca i sistema. Ovaj stek nazivaćemo glavnim. Jezgro će unutar funkcije `main()` kreirati nit klase `ThreadCollector` (ugrađeni proces) i nit nad korisničkom funkcijom `userMain()`. Zatim će zapamtiti kontekst glavnog programa, kako bi po završetku svih korisničkih niti taj kontekst mogao da se povрати i program regularno završi:

```
void main () {  
  
    ThreadCollector::Instance()->start();  
  
    Thread::runningThread = new Thread(userMain);  
    ThreadCollector::Instance()->put(Thread::runningThread);  
  
    if (setjmp(mainContext)==0) {  
        unlock();  
        Thread::runningThread->resume();  
    } else {  
        ThreadCollector::destroy();  
        return;  
    }  
}
```

- Treba još obezbediti "hvatanje" trenutka kada su sve korisničke niti završene. To najbolje može da uradi sam `ThreadCollector`: onog trenutka kada on sadrži samo jednu jedinu evidentiranu nit u sistemu (to je on sam), sve ostale niti su završene. (On evidentira sve aktivne niti, a ne samo spremne.) Tada treba izvršiti povratak na glavni kontekst:

```
void ThreadCollector::run () {  
    //...  
    if (count()==1)  
        longjmp(mainContext,1); // return to main  
    //...  
}
```

### ***Realizacija***

- Zaglavlje `kernel.h` služi samo da uključi sva zaglavlja koja predstavljaju interfejs prema korisniku. Tako korisnik može jednostavno da uključi samo ovo zaglavlje u svoj kod da bi dobio deklaracije Jezgra.



- Prilikom prevođenja u bilo kom prevodiocu treba obratiti pažnju na sledeće opcije prevodioca:
  1. Funkcije deklarisanе kao *inline* moraju tako i da se prevode. U Borland C++ prevodiocu treba da bude isključena opcija Options\Compiler\C++ options\Out-of-line inline functions. Kritična je zapravo samo funkcija Thread::setContext().
  2. Program ne sme biti preveden kao *overlay* aplikacija. U Borland C++ prevodiocu treba izabrati opciju Options\Application\DOS Standard.
  3. Memorijski model treba da bude takav da su svi pokazivači tipa *far*. U Borland C++ prevodiocu treba izabrati opciju Options\Compiler\Code generation\Compact ili Large ili Huge.
  4. Mora da bude isključena opcija provere ograničenja steka. U Borland C++ prevodiocu treba da bude isključena opcija Options\Compiler\Entry/Exit code\Test stack overflow.
- Sledi kompletan izvorni kod opisanog dela Jezgra.

• Datoteka kernel.h:

```
// Project: Real-Time Programming
// Subject: Multithreaded Kernel
// Module: Kernel
// File: kernel.h
// Created: November 1996
// Revised: August 2003
// Author: Dragan Milicev
// Contents: Kernel Interface
```

```
#include "thread.h"
#include "semaphor.h"
#include "msgque.h"
#include "timer.h"
```

```
inline void dispatch () { Thread::dispatch(); }
```

\* Datoteka krnl.h:

```
// Project: Real-Time Programming
// Subject: Multithreaded Kernel
// Module: Kernel
// File: krnl.h
// Created: November 1996
// Revised: August 2003
// Author: Dragan Milicev
// Contents: Kernel module interface
//      Helper functions:
//                  lock
//                  unlock

#ifndef _KRNL_
#define _KRNL_

#include <setjmp.h>

void lock ();    // Switch to kernel mode
void unlock (); // Switch to user mode

extern void userMain(); // User's main function

extern jmp_buf mainContext; // Context of the main thread

#endif
```

\* Datoteka thread.h:

```
// Project:   Real-Time Programming
// Subject:   Multithreaded Kernel
// Module:    Thread
// File:      thread.h
// Created:   November 1996
// Revised:   August 2003
// Author:    Dragan Milicev
// Contents:  Threading and context switching
//           Class: Thread

#ifndef _THREAD_
#define _THREAD_

#include "krnl.h"
#include "collect.h"
#include "recycle.h"

/////////////////////////////////////////////////////////////////
// class Thread
/////////////////////////////////////////////////////////////////

class Thread : public Object {
public:

    Thread ();
    Thread (void (*body) ());

    void start ();
    static void dispatch ();

    static Thread* running ();

    CollectionElement* getCEForScheduler ();
    CollectionElement* getCEForCollector ();
    CollectionElement* getCEForSemaphore ();

protected:

    virtual void run ();

    void markOver ();

    friend class ThreadCollector;
    virtual ~Thread ();

    friend class Semaphore;

    inline int setContext ();
    void resume ();
    char* getStackPointer () const;

    static void wrapper ();
    void fork();

private:

    void (*myBody) ();
    char* myStack;

    jmp_buf myContext;

    int isBeginning;
    int isOver;
```

```
friend void main ();
static Thread* runningThread;

CollectionElement ceForScheduler;
CollectionElement ceForCollector;
CollectionElement ceForSemaphore;

RECYCLE_DEC(Thread)

};

// WARNING: This function MUST be truely inline!
inline int Thread::setContext () {
    return setjmp(myContext);
}

inline void Thread::markOver () {
    isOver=1;
}

inline void Thread::run () {
    if (myBody!=0) myBody();
}

inline Thread* Thread::running () {
    return runningThread;
}

inline Thread::~~Thread () {
    delete [] myStack;
}

inline CollectionElement* Thread::getCEForScheduler () {
    return &ceForScheduler;
}

inline CollectionElement* Thread::getCEForCollector () {
    return &ceForCollector;
}

inline CollectionElement* Thread::getCEForSemaphore () {
    return &ceForSemaphore;
}

#endif

*      Datoteka schedul.h:
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Scheduler
// File:     schedul.h
// Created:  November 1996
```

```
// Revised: August 2003
// Author: Dragan Milicev
// Contents:
//      Class: Scheduler

#ifndef _SCHEDUL_
#define _SCHEDUL_

////////////////////////////////////
// class Scheduler
////////////////////////////////////

class Thread;

class Scheduler {
public:

    static Scheduler* Instance ();

    virtual void put (Thread*) = 0;
    virtual Thread* get () = 0;

protected:
    Scheduler () {}
};

#endif
```

**\* Datoteka thrcol.h:**

```
// Project: Real-Time Programming
// Subject: Multithreaded Kernel
// Module: Thread Collector
// File: thrcol.h
// Created: November 1996
// Revised: August 2003
// Author: Dragan Milicev
// Contents: Thread Collector responsible for thread deletion
//      Class: ThreadCollector

#ifndef _THRCOL_
#define _THRCOL_

#include "collect.h"
#include "thread.h"

////////////////////////////////////
// class ThreadCollector
////////////////////////////////////

class ThreadCollector : public Thread {
public:

    static ThreadCollector* Instance ();

    void put (Thread*);
    int count ();

protected:

    friend void main ();
    static void create ();
    static void destroy ();
```

```

    virtual void run ();

private:

    ThreadCollector () {}
    ~ThreadCollector () {}

    Collection rep;

    static ThreadCollector* instance;

};

inline void ThreadCollector::put (Thread* t) {
    if (t) rep.append(t->getCEForCollector());
}

inline int ThreadCollector::count () {
    return rep.size();
}

#endif

```

\*      **Datoteka kernel.cpp:**

```

// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Kernel
// File:     kernel.cpp
// Created:  November 1996
// Revised:  August 2003
// Author:   Dragan Milicev
// Contents: Kernel main module
//           Helper functions: lock
//                               unlock
//           Functions:         main

#include "krnl.h"
#include "thread.h"
#include "schedul.h"
#include "thrcol.h"

////////////////////////////////////
// Helper functions lock () and unlock ()
////////////////////////////////////

void lock () {}    // Switch to Kernel mode
void unlock () {} // Switch to User mode

////////////////////////////////////
// Function: main ()
////////////////////////////////////

jmp_buf mainContext; // Context of the main thread

void main () {

```

```
ThreadCollector::create();
ThreadCollector::Instance()->start();

Thread::runningThread = new Thread(userMain);
ThreadCollector::Instance()->put(Thread::running());

if (setjmp(mainContext)==0) {
    unlock();
    Thread::running()->resume();
} else {
    ThreadCollector::destroy();
    return;
}
}
```

\*      Datoteka thread.cpp:

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Thread
// File:     thread.cpp
// Created:  November 1996
// Revised:  August 2003
// Author:   Dragan Milicev
// Contents: Threading and context switching
//          Class: Thread

#include <dos.h>
#include "thread.h"
#include "thrcol.h"
#include "schedul.h"

////////////////////////////////////
// class Thread
////////////////////////////////////

const int StackSize = 4096;

RECYCLE_DEF(Thread);

Thread::Thread ()
: RECYCLE_CON(Thread), myBody(0), myStack(new char[StackSize]),
  isBeginning(1), isOver(0),
  ceForScheduler(this), ceForCollector(this), ceForSemaphore(this) {}

Thread::Thread (void (*body)())
: RECYCLE_CON(Thread), myBody(body), myStack(new char[StackSize]),
  isBeginning(1), isOver(0),
  ceForScheduler(this), ceForCollector(this), ceForSemaphore(this) {}

void Thread::resume () {
    if (isBeginning) {
        isBeginning=0;
        wrapper();
    } else
        longjmp(myContext,1);
}

Thread* Thread::runningThread = 0;
```

```

void Thread::start () {
    ThreadCollector::Instance()->put(this);
    fork();
}

void Thread::dispatch () {
    lock ();
    if (runningThread && runningThread->setContext()==0) {

        Scheduler::Instance()->put(runningThread);
        runningThread = (Thread*)Scheduler::Instance()->get();
        // Context switch:
        runningThread->resume();

    } else {
        unlock ();
        return;
    }
}

/////////////////////////////////////////////////////////////////
// Warning: Hardware/OS Dependent!
/////////////////////////////////////////////////////////////////

char* Thread::getStackPointer () const {
    // WARNING: Hardware\OS dependent!
    // PC Stack grows downwards:
    return myStack+StackSize-10;
}

// Borland C++: Compact, Large, or Huge memory Model needed!
#if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)
    #error Compact, Large, or Huge memory model needed
#endif

#define splitStack(p) \
    static unsigned int sss, ssp; \
    sss=FP_SEG(p); ssp=FP_OFF(p); \
    asm { \
        mov ss,sss; \
        mov sp,ssp; \
        mov bp,sp; \
        add bp,8 \
    }

/////////////////////////////////////////////////////////////////
// Enf of Dependencies
/////////////////////////////////////////////////////////////////

void Thread::fork () {
    lock();
    Scheduler::Instance()->put(this);
    unlock();
}

void Thread::wrapper () {

```

```
void* p=runningThread->getStackPointer();
splitStack(p);

unlock ();
runningThread->run();
lock ();

    runningThread->markOver();
    runningThread=(Thread*)Scheduler::Instance()->get();
    runningThread->resume();
}

*      Datoteka schedul.cpp:

// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Scheduler
// File:     schedul.cpp
// Created:  November 1996
// Revised:  August 2003
// Author:   Dragan Milicev
// Contents:
//          Classes: Scheduler
//                  RoundRobinScheduler

#define _RoundRobinScheduler

#include "schedul.h"
#include "queue.h"
#include "thread.h"

////////////////////////////////////
// class RoundRobinScheduler
////////////////////////////////////

class RoundRobinScheduler : public Scheduler {
public:

    virtual void    put (Thread* t) { if (t) rep.put(t->getCEForScheduler()); }
    virtual Thread* get ()          { return (Thread*)rep.get(); }

private:
    Queue rep;
};

////////////////////////////////////
// class Scheduler
////////////////////////////////////

Scheduler* Scheduler::Instance () {
    #ifdef _RoundRobinScheduler
        static RoundRobinScheduler instance;
    #endif
    return &instance;
}

*      Datoteka thrcol.cpp:
```



```
// Project: Real-Time Programming
// Subject: Multithreaded Kernel
// Module: Thread Collector
// File: thrcol.cpp
// Created: November 1996
// Revised: August 2003
// Author: Dragan Milicev
// Contents: Thread Collector responsible for thread deletion
// Class: ThreadCollector

#include "thrcol.h"

////////////////////////////////////
// class ThreadCollector
////////////////////////////////////

ThreadCollector* ThreadCollector::instance = 0;

ThreadCollector* ThreadCollector::Instance () {
    if (instance==0) create();
    return instance;
}

void ThreadCollector::create () {
    instance = new ThreadCollector;
}

void ThreadCollector::destroy () {
    delete instance;
}

void ThreadCollector::run () {
    while (1) {

        int i=0;
        CollectionIterator* it = rep.getIterator();

        for (i=0,it->reset(); !it->isDone(); it->next(),i++) {
            Thread* cur = (Thread*)it->currentItem();
            if (cur->isOver) {
                rep.remove(i);
                delete cur;
                it->reset(); i=0;
                dispatch();
            }
        }

        if (count()==1)
            longjmp(mainContext,1); // return to main

        dispatch();
    }
}
```

## Vežbe

### 5.1

Dat je sledeći kod:

```
class Calculate : public Thread {
public:
    virtual void run () {
        // A long calculation ...
    }
    ...
}

Calculate* aCalculation = new Calculate;
```

Objasniti u čemu je razlika između:

```
aCalculation->run();
i
aCalculation->start();
```

### 5.2

Korišćenjem školskog Jezgra realizovati funkciju `runConcurrently()` koja kao parametar prima niz pokazivača na globalne funkcije bez argumenata i pokreće po jednu nit nad svakom od tih funkcija.

### 5.3

Korišćenjem školskog Jezgra napraviti niz niti, pri čemu je svaka nit parametrizovana svojom pozicijom u tom nizu.

### 5.4

U prikazanoj realizaciji postoji problem ukoliko kreator niti ima referencu (pokazivač) na tu nit, jer se po završetku niti ona implicitno uništava, pa referenca ostaje viseća (engl. *dangling reference*). Rešiti ovaj problem tako da se:

- može ispitati da li je izvršavanje niti gotovo ili ne, pozivom operacije `isDone()`;
- nit ne uništava implicitno, već eksplicitno, pozivom operacije `destroy()`, ali samo pod uslovom da je njeno izvršavanje gotovo.

### 5.5

Pokazati kako se korišćenjem niti iz modifikovanog školskog Jezgra u prethodnom zadatku može realizovati konstrukt `cobegin`.

### 5.6

Komentarizovati kako se propagira izuzetak podignut u nekoj niti konkurentnog programa realizovanog u školskom Jezgru. Šta se dešava ukoliko se izuzetak ne obradi u korisničkom kodu jedne niti? Rešiti ovaj problem odgovarajućom modifikacijom Jezgra.

### 5.7

Komentarisati da li i kako bi se mogla realizovati sledeća semantika propagacije izuzetaka u programu realizovanom pomoću datog školskog Jezgra: ukoliko izuzetak nije obrađen u datoj niti, on se propagira u nit koja je kreirala tu nit ("roditeljska" nit).

### 5.8

Potrebno je da klasa `x` ima operaciju `do()` čija implementacija treba da ima sledeću semantiku:

```
void X::do(Y* y, int i) {  
    int j = this->helper(i);  
    y->do(j);  
}
```

Korišćenjem školskog Jezgra realizovati klasu `x`, ali tako da se svaki poziv operacije `X::doSomething()` izvršava u sopstvenom kontekstu, a ne u kontekstu pozivaoca.

### 5.9

Korišćenjem niti iz školskog Jezgra, skicirati strukturu programa koji kontroliše parking za vozila. Parking ima jedan ulaz i jedan izlaz, kao i znak da na parkingu više nema slobodnih mesta.

---

# Sinhronizacija i komunikacija pomoću deljene promenljive

---

## Međusobno isključenje i uslovna sinhronizacija

### *Međusobno isključenje*

- Iako se deljena promenljiva čini kao jednostavan, pravolinijski koncept za razmenu informacija između procesa, njeno neograničeno korišćenje od strane uporednih procesa je nepouzđano zbog mogućnosti višestrukog upisa. Na primer, neka dva procesa pristupaju deljenoj promenljivoj  $x$  tako što uporedo izvršavaju naredbu:

```
x := x+1
```

Na većini procesora ova naredba biće izvršena kao sekvenca tri mašinske instrukcije koje nisu *nedeljive* (engl. *indivisible*), tj. *atomične* (engl. *atomic*):

- učitaj vrednost  $x$  iz memorije u registar procesora
- uvećaj vrednost registra za jedan
- upiši vrednost iz registra u memoriju (promenljivu  $x$ ).

Kako ove instrukcije nisu nedeljive, različita preplitanja njihovog izvršavanja u kontekstu dva uporedna procesa može da da različite rezultate, od kojih su neki nekorektni. Na primer, ukoliko je prethodna vrednost  $x$  bila 0, konačna vrednost za  $x$  može biti i 1 (tako što oba procesa učitaju vrednost 0, a zatim oba upišu vrednost 1) ili 2 (korektno).

- Posmatrajmo još jedan primer. Policijski helikopter prati kriminalca-begunca i navodi policijski automobil koji ga prati. Neka deljena struktura podataka koja čuva koordinate begunca izgleda ovako:

```
type Coord = record {  
  x : integer;  
  y : integer;  
};
```

```
var sharedCoord : Coord;
```

Neka proces `Helicopter` prati koordinate begunca i upisuje ih u deljenu strukturu na sledeći način:

```
process Helicopter  
var nextCoord : Coord;  
begin  
  loop  
    computeNextCoord(nextCoord);  
    sharedCoord := nextCoord;  
  end;  
end;
```

Neka proces `PoliceCar` predstavlja policijski automobil koji prema koordinatama zadatim iz helikoptera prati begunca na sledeći način:

```
process PoliceCar  
begin
```

```

loop
  moveToCoord(sharedCoord);
end;
end;

```

Kako se može očekivati da se operacija upisa podataka u strukturu (naredba `sharedCoord:=nextCoord`) implementira pomoću (bar) dve mašinske instrukcije koje nisu nedeljive, može se dogoditi da begunac pobegne, iako ga helikopter uspešno prati, zbog sledećeg preplitanja izvršenih instrukcija dva procesa:

Proces <code>Helicopter</code> upisuje u <code>sharedCoord</code> :	Vrednost u <code>sharedCoord</code> :	Proces <code>PoliceCar</code> čita iz <code>sharedCoord</code> :
	0,0	
<code>x:=1</code>	1,0	
<code>y:=1</code>	1,1	
	1,1	<code>x=1</code>
	1,1	<code>y=1</code>
<code>x:=2</code>	2,1	
<code>y:=2</code>	2,2	
	2,2	<code>x=2</code>
	2,2	<code>y=2</code>
<code>x:=3</code>	3,2	
	3,2	<code>x=3</code>
	3,2	<code>y=2</code>
<code>y:=3</code>	3,3	

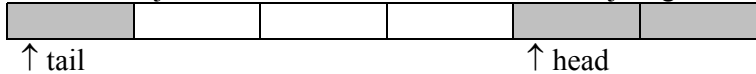
- Prema tome, neki delovi koda procesa koji pristupaju deljenim promenljivim moraju da se izvršavaju nedeljivo (atomično) u odnosu na druge takve delove drugih procesa.
- Deo koda (sekvenca naredbi) procesa koji se mora izvršavati nedeljivo (engl. *indivisible*) u odnosu na druge takve delove koda drugih procesa naziva se *kritična sekcija* (engl. *critical section*).
- Sinhronizacija koja je neophodna da bi se obezbedila atomičnost izvršavanja kritičnih sekcija naziva se *međusobno isključenje* (engl. *mutual exclusion*).
- Pretpostavlja se da je atomičnost operacije dodele vrednosti skalarnoj promenljivoj obezbeđena na nivou upisa u memoriju (instrukcija upisa vrednosti u skalarnu promenljivu je atomična). Na primer, ukoliko jedan proces izvršava naredbu `x:=1` a drugi naredbu `x:=2`, onda će vrednost `x` biti ili 1 ili 2, a ne neka druga vrednost. Naravno, ukoliko se vrši upis u neskalarnu promenljivu, atomičnost u opštem slučaju nije obezbeđena, osim na nivou jedne mašinske reči.
- Problem međusobnog isključenja je prvi opisao Dijkska 1965. godine. Ovaj problem je od izuzetnog teorijskog i praktičnog značaja za konkurentno programiranje.

### Uslovna sinhronizacija

- Međusobno isključenje nije jedina vrsta sinhronizacije od interesa. Druga je *uslovna sinhronizacija* (engl. *condition synchronization*). Uslovna sinhronizacija je potrebna kada jedan proces želi da izvrši akciju koja ima smisla ili je sigurna samo ako je neki drugi proces preduzeo neku svoju akciju ili se nalazi u nekom definisanom stanju.
- Najpoznatiji, školski primer je problem *ograničenog bafera* (engl. *bounded buffer*). Dva procesa razmenjuju podatke preko bafera koji je ograničenog kapaciteta. Prvi proces "proizvodi" podatke i upisuje ih u bafer; on se naziva *proizvođačem* (engl. *producer*).

Drugi proces uzima (čita) podatke iz bafera i konzumira ih na neki način; on se naziva *potrošačem* (engl. *consumer*).

- Ovakva indirektna komunikacija između procesa obezbeđuje njihovu nezavisnije izvršavanje koje dozvoljava male fluktuacije u brzini kojom proizvode, odnosno konzumiraju podatke. Na primer, ukoliko je u nekom periodu potrošač nešto sporiji, proizvođač može puniti bafer proizvedenim podacima; slično, u nekom drugom trenutku potrošač može biti nešto brži i trošiti zalihu podataka iz bafera. Ovakav baferisani sistem se često naziva još i sistem *proizvođač-potrošač* (engl. *producer-consumer*).
- Ukoliko je bafer ograničenog kapaciteta, što je često slučaj zbog ograničenosti resursa u sistemu, onda je potrebna sledeća uslovna sinhronizacija između procesa:
  - proizvođač ne sme da stavi podatak u bafer ukoliko je bafer pun;
  - potrošač ne može da uzme podatak iz bafera ukoliko je bafer prazan;
  - ukoliko su moguće simultane operacije stavljanja i uzimanja podatka, onda se mora obezbediti i međusobno isključenje, kako ne bi došlo do korupcije internih struktura koje pamte poziciju "prvog" i "poslednjeg" stavljenog elementa; ukoliko je bafer realizovan kao kružni, situacija izgleda kao na sledećoj slici:



## Uposleno čekanje

- Jedan jednostavan način za implementaciju sinhronizacije je da procesi postavljaju i proveravaju indikatore (engl. *flags*). Proces koji signalizira ispunjenje uslova postavlja indikator; proces koji čeka da uslov bude ispunjen, proverava indikator u petlji:

```
process P1; (* Waiting process *)
begin
  ...
  while flag = false do
    null
  end;
  ...
end P1;

process P2; (* Signalling process *)
begin
  ...
  flag := true;
  ...
end P2;
```

- Ovakva realizacija sinhronizacije, gde proces koji čeka na ispunjenje uslova izvršava petlju sve dok indikator ne bude postavljen, naziva se *uposlano čekanje* (engl. *busy waiting*).
- Algoritam uposlenog čekanja je jednostavan za uslovnu sinhronizaciju. Međutim, međusobno isključenje nije jednostavno sasvim korektno realizovati uposlenim čekanjem.
- Jedan (neispravan) pristup rešavanju međusobnog isključenja pomoću uposlenog čekanja je sledeći: proces najavljuje svoju želju da uđe u kritičnu sekciju, a potom ispituje da li je drugi proces već ušao u kritičnu sekciju:

```
process P1
begin
  loop
    flag1 := true; (* Announce intent to enter *)
    while flag2 = true do (* Busy wait if the other process is in *)
```

```

        null
    end;
    <critical section>          (* Critical section *)
    flag1 := false;            (* Exit protocol *)
    <non-critical section>
end
end P1;

process P2
begin
    loop
        flag2 := true;          (* Announce intent to enter *)
        while flag1 = true do    (* Busy wait if the other process is in *)
            null
        end;
        <critical section>      (* Critical section *)
        flag2 := false;         (* Exit protocol *)
        <non-critical section>
    end
end P2;

```

Ovo rešenje ima problem jer se može desiti sledeći scenario: jedan proces najavi svoj ulazak u kritičnu sekciju postavljajući svoj indikator, onda to uradi i drugi proces, a zatim oba procesa večno ostaju u petljama čekajući na obaranje indikatora onog drugog procesa. Dakle, nijedan proces neće moći da uđe u kritičnu sekciju, iako se oba procesa izvršavaju. Ovakvo neispravno stanje sistema naziva se *živo blokiranje* (engl. *livelock*).

- Drugi pristup može da bude promena redosleda najave ulaska u sekciju i postavljanja indikatora:

```

process P1
begin
    loop
        while flag2 = true do    (* Busy wait if the other process is in *)
            null
        end;
        flag1 := true;
        <critical section>        (* Critical section *)
        flag1 := false;          (* Exit protocol *)
        <non-critical section>
    end
end P1;

process P2
begin
    loop
        while flag1 = true do    (* Busy wait if the other process is in *)
            null
        end;
        flag2 := true;
        <critical section>        (* Critical section *)
        flag2 := false;          (* Exit protocol *)
        <non-critical section>
    end
end P2;

```

Ovakvo rešenje ne obezbeđuje međusobno isključenje, jer se može dogoditi sledeći scenario: oba indikatora su *false*, oba procesa ispituju tuđ indikator i pronalaze da je on *false*, pa zatim oba postavljaju svoj indikator i ulaze u kritičnu sekciju. Problem je što se operacije ispitivanja tuđeg indikatora i postavljanja svog ne obavljaju atomično.

- Sledeće rešenje uvodi promenljivu *turn* koja ukazuje na koga je došao red da uđe u sekciju:

```
process P1
begin
  loop
    while turn = 2 do
      null
    end;
    <critical section>
    turn := 2;
    <non-critical section>
  end
end P1;
```

```
process P2
begin
  loop
    while turn = 1 do
      null
    end;
    <critical section>
    turn := 1;
    <non-critical section>
  end
end P2;
```

Ovo rešenje obezbeđuje međusobno isključenje i nema problem živog blokiranja, ali ima problem zato što nasilno uvodi nepotreban redosled izvršavanja procesa, jer se procesi ovde uvek naizmenično smenjuju u kritičnoj sekciji. To je neprihvatljivo u opštem slučaju autonomnih procesa.

- Konačno, sledeće rešenje nema ovaj problem nasilnog uslovljavanja redosleda procesa, kao ni živog blokiranja, a obezbeđuje međusobno isključenje (Peterson 1981.). Ukoliko oba procesa žele da uđu u kritičnu sekciju, onda dozvolu dobija onaj na kome je red (definisan promenljivom *turn*); ukoliko samo jedan proces želi da uđe u sekciju, on to može da uradi:

```
process P1
begin
  loop
    flag1 := true;
    turn := 2;
    while flag2 = true and turn = 2 do
      null
    end;
    <critical section>
    flag1 := false;
    <non-critical section>
  end
end P1;
```

```
process P2
begin
  loop
    flag2 := true;
    turn := 1;
    while flag1 = true and turn = 1 do
      null
    end;
    <critical section>
    flag2 := false;
    <non-critical section>
  end
end P1;
```



- Osnovni problem algoritama uposlenog čekanja je njihova neefikasnost: proces troši procesorsko vreme na nekoristan rad, čekajući u petlji da uslov bude ispunjen. Čak i na multiprocesorskim sistemima sa deljenom memorijom, oni mogu da uzrokuju intenzivan saobraćaj u sistemu (na memorijskoj magistrali). Osim toga, algoritmi za međusobno isključenje više procesa postaju znatno komplikovaniji za realizaciju i razumevanje. Zbog toga se realni sistemi retko oslanjaju samo na uposleno čekanje.

## Semafori

- Semafori predstavljaju jednostavan mehanizam i koncept za programiranje međusobnog isključenja i uslovne sinhronizacije. Predložio ih je Dijkstra 1968.
- Semafor je celobrojna nenegativna promenljiva sa kojom se, osim inicijalizacije, mogu izvršiti još samo dve operacije:
  - `wait(S)`: (Dijkstra je originalno zvao P) Ako je vrednost semafora `s` veća od nule, ta vrednost se umanjuje za jedan; u suprotnom, proces mora da čeka sve dok `S` ne postane veće od nule, a tada se vrednost takođe umanjuje za jedan.
  - `signal(S)`: (Dijkstra je originalno zvao V) Vrednost semafora se uvećava za jedan.
- Važno je uočiti da su operacije `wait` i `signal` *atomične* (nedeljive). Prema tome, dva procesa koja uporedo izvršavaju neku od ovih operacija međusobno ne interaguju.

### Implementacija

- Kada je vrednost semafora nula, proces koji je izvršio operaciju `wait()` treba da čeka da neki drugi proces izvrši operaciju `signal()`. Iako je ovo čekanje moguće implementirati uposlenim čekanjem, takva implementacija je, kao što je objašnjeno, neefikasna. Praktično sve sinhronizacione primitive, a ne samo semafori, oslanjaju se na neki vid *suspenzije* (engl. *suspension*) izvršavanja. Suspenzija se ponekad naziva i *blokiranje* (engl. *blocking*).
- Ovaj pristup se sastoji u sledećem: kada proces izvršava operaciju `wait()`, kontrolu preuzima kod koji pripada izvršnom okruženju. Ako proces treba suspendovati, onda izvršno okruženje ne smešta kontekst procesa (tj. njegov PCB) u listu spremnih procesa, već u posebnu listu pridruženu svakom semaforu (lista procesa suspendovanih na semaforu). Na taj način suspendovani proces ne može dobiti procesor sve dok ga sistem ponovo ne vrati u listu spremnih, pa zbog toga on ne troši procesorsko vreme dok čeka, kao kod uposlenog čekanja.
- Implementacija semafora zato može da bude sledeća: semafor sadrži red procesa koji čekaju na semaforu i jednu celobrojnu promenljivu `val` koja ima sledeće značenje:
  - 1) `val > 0`: još `val` procesa može da izvrši operaciju `wait` a da se ne blokira;
  - 2) `val = 0`: nema blokiranih na semaforu, ali će se proces koji naredni izvrši `wait` blokirati;
  - 3) `val < 0`: ima `-val` blokiranih procesa, a `wait` izaziva blokiranje.
    - Algoritam operacije `wait` je sledeći:

```

procedure wait(S)
  val:=val-1;
  if val<0 then
    begin
      suspend the running process by putting it into the suspended queue of S
      take another process from the ready queue and switch the context to it
    end
  end

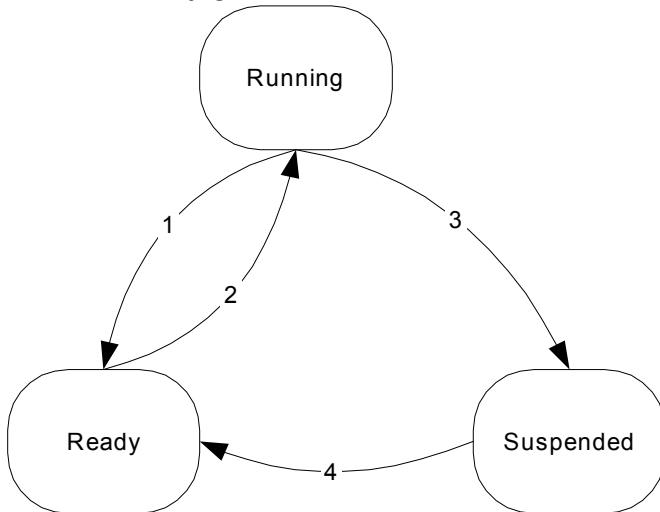
```

```
end
end;
```

- Algoritam operacije `signal` je sledeći:

```
procedure signal(S)
  val:=val+1;
  if val<=0 then
    begin
      take one process from the suspended queue of S
      and deblock it by putting it into the ready queue
    end
  end
end;
```

- Treba primetiti da ovi algoritmi ne definišu redosled po kome se procesi ređaju u listi suspendovanih (ili *blokiranih*, engl. *blocked*) na semaforu. Obično implementacije podrazumevaju FIFO redosled, mada programer treba da smatra da je taj redosled nedeterminisan.
- Prema tome, osnovna stanja kroz koje proces prolazi tokom svog života prikazana su na sledećem dijagramu:



gde prelazi označavaju sledeće situacije:

1. Proces gubi procesor i prelazi u stanje *Ready*, ali ne zbog suspenzije, nego zato što je to sam eksplicitno tražio (operacijom `dispatch()`), ili se u sistemu pojavio novi spremni proces koji treba da preuzme procesor, ukoliko je sistem sa *preuzimanjem* (engl. *preemptive*).
  2. Izabrani proces iz liste spremnih dobija procesor.
  3. Proces gubi procesor i postaje suspendovan zato što se blokira na sinhronizacionoj primitivi (npr. semaforu, po izvršavanju operacije `wait()`).
  4. Proces prelazi iz stanja suspenzije u listu spremnih, jer je tekući (*running*) proces izvršio operaciju `signal()` na semaforu.
- Problem može da predstavlja obezbeđenje nedeljivosti operacija `wait` i `signal`, kao i drugih sličnih atomičnih primitiva. Mogući su različiti slučajevi:
    - Ukoliko izvršno okruženje ima potpunu kontrolu nad sistemom, atomičnost je jednostavno obezbediti, prosto tako što se sistem programira tako da ne vrši promenu konteksta kada neki proces izvršava ove operacije.
    - Međutim, ovo je retko slučaj, jer često postoje spoljašnji događaji, signalizirani prekidima koji dolaze asinhrono, pa mogu prekinuti izvršavanje ovih operacija. U tom slučaju, izvršno okruženje prosto može da zabrani (maskira) prekide tokom izvršavanja sistemskih primitiva.

- Ovakav pristup je adekvatan za jednogprocesorske sisteme. Međutim, kod višepprocesorskih sistema takav pristup nije moguć jer više procesora može paralelno izvršavati operacije `wait` ili `signal` na istom semaforu. U tom slučaju, hardver mora podržavati neki mehanizam za "zaključavanje" samog semafora i obezbeđenje izolovanog pristupa do semafora. Primeri dva takva mehanizma jesu:
  - Hardver obezbeđuje nedeljivu instrukciju *test-and-set* nad bitom u memoriji koja vraća vrednost tog bita a potom ga postavlja na 1. Ova dva koraka su nedeljiva na hardverskom nivou, jer hardver obezbeđuje međusobno isključenje pristupa odgovarajućoj memorijskoj lokaciji (npr. "zaključavanjem" magistrale). Inicijalna vrednost *lock* bita pridruženog semaforu je 0. Ako dva procesa pokušaju da pristupe *lock* bitu istog semafora, samo jedan će pročitati vrednost 0 i odmah je postaviti na 1, pa će mu pristup do semafora biti dozvoljen. Drugi proces će pročitati vrednost 1, pa će mu pristup do semafora biti zabranjen, sve dok prvi proces ne postavi vrednost 0 u *lock* bit po završetku svoje primitive.
  - Sličan mehanizam obezbeđuje operacija *swap* koja atomično zamenjuje vrednost *lock* bita u memoriji i operanda instrukcije. Preostala logika je potpuno ista kao prethodna.

Treba primetiti da u ovom slučaju jedan proces izvršava uposlano čekanje na *lock* bitu. Međutim, to uposlano čekanje je veoma kratko i traje samo dok jedan proces ne završi operaciju `wait` ili `signal` na semaforu, što je ne samo kratkog, nego i sasvim predvidivog trajanja. Treba uočiti razliku između tog uposlenog čekanja i onog na visokom nivou, kada proces uposlano čeka na pristup do kritične sekcije, što može biti nepredvidivog trajanja i znatno duže. Atomičnost se ne može dobiti "ni iz čega", već se ipak mora nekako podržati i na hardverskom nivou.

### ***Međusobno isključenje i uslovna sinhronizacija pomoću semafora***

- Međusobno isključenje je jednostavno obezbediti pomoću semafora:

```
var mutex : Semaphore(1); // Initially equal to 1

process P1;
  loop
    wait(mutex);
    <critical section>
    signal(mutex);
    <non-critical section>
  end
end P1;

process P2;
  loop
    wait(mutex);
    <critical section>
    signal(mutex);
    <non-critical section>
  end
end P2;
```

- Treba primetiti da je kod koji okružuje kritičnu sekciju jednostavan i uvek isti, bez obzira koliko procesa želi da uđe u kritičnu sekciju. Osim toga, inicijalna vrednost semafora predstavlja zapravo maksimalan broj procesa koji može istovremeno ući u kritičnu sekciju, pa u opštem slučaju može biti i veći od 1.
- Uslovnu sinhronizaciju je takođe jednostavno obezbediti semaforima:

```
var sync : Semaphore(0); // Initially 0

process P1; // Waiting process
...
wait(sync);
...
end P1;

process P2; // Signalling process
...
signal(sync);
...
end P2;
```

- **Primer realizacije ograničenog bafera pomoću semafora:**

```
const int N = ...; // Capacity of the buffer
class Data;

class BoundedBuffer {
public:

    BoundedBuffer ();

    void append (Data*);
    Data* take ();

private:
    Semaphore mutex;
    Semaphore spaceAvailable, itemAvailable;

    Data* buffer[N];
    int head, tail;
};

BoundedBuffer::BoundedBuffer () :
    mutex(1), spaceAvailable(N), itemAvailable(0),
    head(0), tail(0) {}

void BoundedBuffer::append (Data* d) {
    spaceAvailable.wait();
    mutex.wait();
    buffer[tail] = d;
    tail = (tail+1)%N;
    mutex.signal();
    itemAvailable.signal();
}

Data* BoundedBuffer::take () {
    itemAvailable.wait();
    mutex.wait();
    Data* d = buffer[head];
    head = (head+1)%N;
    mutex.signal();
    spaceAvailable.signal();
    return d;
}

class Producer : public Thread {
public:
    Producer (BoundedBuffer* bb) : myBuffer(bb) {...}

protected:
```

```

    virtual void run ();

    Data* produce(); // Produce an item

private:
    BoundedBuffer* myBuffer;
};

void Producer::run () {
    while (1) {
        Data* d = produce();
        myBuffer->append(d);
    }
}

class Consumer : public Thread {
public:
    Consumer (BoundedBuffer* bb) : myBuffer(bb) {...}

protected:
    virtual void run ();

    void consume(Data*); // Consume an item

private:
    BoundedBuffer* myBuffer;
};

void Consumer::run () {
    while (1) {
        Data* d = myBuffer->take();
        consume(d);
    }
}

```

- Potencijalni problem koji može da se pojavi pri nekorektnoj upotrebi semafora je *kružno blokiranje* (engl. *deadlock*): stanje sistema u kome je nekoliko procesa suspendovano (blokirano) međusobnim uslovljavanjem. Na primer:

```

process P1;
    wait(S1);
    wait(S2);
    ...
    signal(S2);
    signal(S1);
end P1;

process P2;
    wait(S2);
    wait(S1);
    ...
    signal(S1);
    signal(S2);
end P2;

```

- Iako su semafori jednostavan i moćan koncept, oni nisu sasvim pogodni za programiranje složenih sistema. Samo jedna greška u uparivanju operacija `wait` i `signal` kod kritične sekcije ili uslovne sinhronizacije dovodi do potpuno nekorektnog ponašanja programa. Semafori su važni iz istorijskih razloga i zbog toga što predstavljaju jednostavnu, elementarnu sinhronizacionu primitivu, pomoću koje je moguće realizovati mnoge druge.

Zato se nijedan složeniji sistem ne zasniva isključivo na semaforima, nego su potrebni strukturirani konstrukti.

### ***Binarni semafori***

- Opisani semafor se naziva *brojačkim* (engl. *counting*) ili *n-arnim*. Za mnoge primene (npr. za međusobno isključenje) dovoljno je da semafor ima najveću vrednost 1. Ukoliko negativne vrednosti nisu potrebne (npr. nije potrebno znati koliko ima procesa koji čekaju na semaforu ili samo jedan proces može čekati), onda je dovoljna samo binarna vrednost semafora (0 i 1). Takvi semafori nazivaju se *binarnim* (engl. *binary*).
- U nekim sistemima se koncept binarnog semafora naziva *događajem* (engl. *event*), u smislu da binarna vrednost semafora označava da se neki događaj ili desio, ili nije desio.
- Operacija `wait` suspenduje proces, ukoliko vrednost događaja nije 1, a postavlja vrednost događaja na 0, ako je njegova vrednost bila 1. Operacija `signal` deblokira proces koji je suspendovan, ako ga ima, odnosno postavlja vrednost događaja na 1, ako suspendovanog procesa nema.
- Na događaj po pravilu čeka samo jedan proces, pa je semantika događaja nedefinisana ako postoji više suspendovanih procesa. Zato se u nekim sistemima događaj proglašava kao vlasništvo nekog procesa, i jedino taj proces može izvršiti operaciju `wait`, dok operaciju `signal` može vršiti bilo koji proces.
- U mnogim sistemima postoje složene operacije čekanja na više događaja, po kriterijumu "i" i "ili".

### **Uslovni kritični regioni**

- *Uslovni kritični region* (engl. *conditional critical region*) je deo koda za koji se garantuje međusobno isključenje. (Ovo treba razlikovati od kritične sekcije za koju *treba* obezbediti međusobno isključenje, ali koje ne mora biti obezbeđeno u slučaju greške.)
- Deljene promenljive koje treba obezbediti od konkurentnog pristupa grupišu se i proglašavaju *resursima* (engl. *resource*), a koncept kritičnog regiona obezbeđuje da više procesa ne može konkurentno pristupati nekom resursu.
- Uslovna sinhronizacija se obezbeđuje pomoću tzv. *čuvara* (engl. *guard*), koji predstavljaju logičke izraze kao uslove za ulazak u region. Kada proces ulazi u kritični region, izračunava se *guard* (uz međusobno isključenje); ako je rezultat `true`, proces može da uđe u region, inače se suspenduje. Kao i za semafore, ne podrazumeva se nikakav determinisani redosled pristupa regionu ukoliko više procesa čeka da uđe u isti region.
- Primer: realizacija ograničenog bafera pomoću uslovnih kritičnih regiona:

```
program buffer;
  type BufferTp is record
    slots      : array(1..N) of character;
    size       : integer range 0..N;
    head, tail : integer range 1..N;
  end record;

  buffer : BufferTp;
  resource buf : buffer;

  process producer is separate;
  process consumer is separate;
end.
```

```

process producer;
  loop
    region buf when buffer.size < N do
      -- place char in buffer etc
    end region
  end loop;
end producer

process consumer;
  loop
    region buf when buffer.size > 0 do
      -- take char from buffer etc
    end region
  end loop;
end consumer

```

- Jedan problem kritičnih regiona su performanse, jer svaki proces koji čeka na *guard* mora ponovo da izračuna *guard* izraz svaki put kada neki proces napusti region. To znači da suspendovani proces mora da postane izvršan da bi izračunao izraz, iako je rezultat izraza možda ponovo *false*.
- Drugi, osnovni problem kritičnih regiona je što je kod za pristup deljenim resursima rasut po programu i ne obezbeđuje enkapsulaciju resursa.

## Monitori

- Koncept monitora teži da reši probleme uslovnih kritičnih regiona tako što enkapsulira podatke (resurse) i procedure (kritične sekcije) koje nad njima operišu u jedinstvenu strukturu. Monitori takođe koriste oblik uslovne sinhronizacije koji se može efikasnije implementirati.
- Kritični regionu se pišu kao procedure i zajedno sa podacima se grupišu u jedinstveni modul pod nazivom *monitor* (engl. *monitor*). Podaci koji pripadaju tom modulu su sakriveni i nedostupni spolja. Samo su procedure dostupne za pozive spolja, one predstavljaju interfejs modula.
- Procedure se podrazumevano izvršavaju međusobno isključeno, pa nije potrebna nikakva eksplicitna sinhronizacija u tom cilju. Drugim rečima, međusobno isključenje procedura je implicitno garantovano semantikom monitora.
- Koncept monitora je nastao kao unapređenje koncepta uslovnih kritičnih regiona. Na razvoju koncepta monitora radili su Dijkstra (1968), Brinch-Hansen (1973) i Hoare (1974). Monitori u svom izvornom obliku postoje u jezicima Modula 1, Concurrent Pascal i Mesa. Naprednije varijante monitora postoje i u jezicima Ada i Java.
- Primer monitora koji realizuje ograničeni bafer:

```

monitor buffer;
  export append, take;

  var ... (* Declaration of necessary variables *)

  procedure append (i : integer);
    ...
  end;

  procedure take (var i : integer);
    ...
  end;

```

```
begin
  ... (* Initialization of monitor variables *)
end;
```

### **Uslovna sinhronizacija u monitoru**

- Iako monitor implicitno obezbeđuje međusobno isključenje, potrebna je i uslovna sinhronizacija (npr. kod ograničenog bafera). Iako se za tu svrhu mogu koristiti semafori, postoje i jednostavnije sinhronizacione primitive vezane za monitore, ali je njihova semantika različita za različite koncepte i jezike.
- Kod monitora koje je predložio Hoare (1974), sinhronizaciona primitiva se naziva *uslovna promenljiva* (engl. *condition variable*). Uslovna promenljiva je član monitora. Nad njom se mogu vršiti dve operacije sa sledećom semantikom:
  - *wait*: proces koji je izvršio *wait* se (bezuslovno) suspenduje (blokira) i smešta u red čekanja pridružen ovoj uslovnoj promenljivoj; proces potom oslobađa svoj ekskluzivni pristup do monitora i time dozvoljava da drugi proces uđe u monitor;
  - *signal*: kada neki proces izvrši ovu operaciju, sa reda blokiranih procesa na ovoj uslovnoj promenljivoj oslobađa se (deblokira) jedan proces, ako takvog ima; ako takvog procesa nema, onda operacija *signal* nema nikakvog efekta.
- Primer ograničenog bafera sa uslovnom sinhronizacijom:

```
monitor buffer;
  export append, take;
  var
    buf : array[0..size-1] of integer;
    top, base : 0..size-1;
    numberInBuffer : integer;
    spaceAvailable, itemAvailable : condition;

  procedure append (i : integer);
  begin
    if numberInBuffer = size then
      wait(spaceAvailable);
    end if;
    buf[top] := i;
    numberInBuffer := numberInBuffer+1;
    top := (top+1) mod size;
    signal(itemAvailable);
  end append;

  procedure take (var i : integer);
  begin
    if numberInBuffer = 0 then
      wait(itemAvailable);
    end if;
    i := buf[base];
    base := (base+1) mod size;
    numberInBuffer := numberInBuffer-1;
    signal(spaceAvailable);
  end take;

begin (* Initialization *)
  numberInBuffer := 0;
  top := 0; base := 0
end;
```

- Treba obratiti pažnju na razlike između operacija *wait* i *signal* na semaforu i na uslovnoj promenljivoj:



- Operacija `wait` na uslovnoj promenljivoj uvek blokira proces, za razliku od operacije `wait` na semaforu.
  - Operacija `signal` na uslovnoj promenljivoj nema efekta na tu promenljivu ukoliko na njoj nema blokiranih procesa, za razliku od operacije `signal` na semaforu.
- Pitanje je šta se dešava kada se operacijom `signal` deblokira neki proces: tada postoje dva procesa koja konkurišu za pristup monitoru (onaj koji je izvršio `signal` i onaj koji je deblokiran), pri čemu ne smeju oba nastaviti izvršavanje? Postoje različite varijante definisane semantike operacije `signal` koje ovo rešavaju:
  - Operacija `signal` je dozvoljena samo ako je poslednja akcija procesa pre napuštanja monitora (kao u primeru ograničenog bafera).
  - Operacija `signal` ima sporedni efekat izlaska procesa iz procedure monitora (implicitni `return`); drugim rečima, proces nasilno napušta monitor.
  - Operacija `signal` koja deblokira drugi proces implicitno blokira proces koji je izvršio `signal`, tako da on može da nastavi izvršavanje tek kada monitor ostane slobodan. Procesi koji su blokirani na ovaj način imaju prednost u odnosu na druge procese koji tek žele da uđu u monitor.
  - Operacija `signal` koja deblokira drugi proces ne blokira proces koji je izvršio `signal`, ali deblokirani proces može da nastavi izvršavanje tek kada proces koji je izvršio `signal` napusti monitor.

### **Problemi vezani za monitore**

- Jedan od osnovnih problema vezanih za koncept monitora jeste pitanje kako razrešiti situaciju kada se proces koji je napravio ugneždeni poziv operacije drugog monitora iz operacije jednog monitora suspenduje unutar tog drugog monitora? Zbog semantike `wait` operacije, pristup drugom monitoru biće oslobođen, ali neće biti oslobođen pristup monitoru iz koga je napravljen ugneždeni poziv. Tako će procesi koji pokušavaju da uđu u taj monitor biti blokirani, što smanjuje konkurentnost.
- Najčešći pristup ovom problemu je da se spoljašnji monitori drže zaključanim (Java, POSIX, Mesa). Drugi pristup je da se potpuno zabrani ugnežđivanje poziva operacija monitora (Modula-1). Treći pristup je da se obezbede konstrukti kojima bi se definisalo koji monitori se oslobađaju u slučaju blokiranja na uslovnoj promenljivoj u ugnežđenom pozivu.
- Iako su monitori dobar koncept visokog nivoa apstrakcije, koji uspešno obezbeđuje enkapsulaciju i međusobno isključenje, uslovna sinhronizacija se i dalje obavlja primitivama niskog nivoa apstrakcije. Zbog toga svi nedostaci semafora važe i za uslovne promenljive.

### **Zaštićeni objekti u jeziku Ada**

- Osnovni nedostatak izvornog koncepta monitora jeste upotreba uslovnih promenljivih. Naprednija varijanta monitora u jeziku Ada kombinuje prednosti monitora i kritičnih regiona, tako što se umesto uslovnih promenljivih za sinhronizaciju koriste čuvari (engl. *guards*).
- Ovakva varijanta monitora u jeziku Ada naziva se *zaštićenim objektom* (engl. *protected object*), dok se čuvari nazivaju *barijerama* (engl. *barrier*).
- Zaštićeni objekat u jeziku Ada obezbeđuje enkapsulaciju podataka i potprograma koji nad tim podacima operišu. Interfejs zaštićenog objekta specificira koji su potprogrami dostupni spolja. Ti potprogrami nazivaju se *zaštićenim procedurama i funkcijama*.

- Zaštićene procedure mogu da čitaju ili upisuju vrednosti u zaštićene podatke, ali međusobno isključivo, tako da najviše jedan proces može u datom trenutku pristupiti zaštićenoj proceduri.
- Sa druge strane, zaštićene funkcije dozvoljavaju samo čitanje zaštićenih podataka. Dozvoljava se da više procesa simultano izvršava zaštićene funkcije.
- Pozivi zaštićenih procedura i funkcija su međusobno isključivi. Ovaj princip naziva se *više čitalaca-jedan pisac* (engl. *multiple readers-single writer*).
- Zaštićeni objekat može da se definiše ili kao pojedinačna instanca, ili kao tip koji se može instancirati u vreme izvršavanja, slično kao i procesi u jeziku Ada.
- Primer:

```
protected type SharedData (initial : DataItem) is
  function read return DataItem;
  procedure write (newValue : in DataItem);
private
  theData : DataItem := initial;
end SharedData;
```

```
protected body SharedData is
  function read return DataItem is
    begin
      return theData;
    end read;

  procedure write (newValue : in DataItem) is
    begin
      theData := newValue;
    end write;
end SharedData;
```

- Posebnu vrstu zaštićenih procedura predstavljaju *ulazi* (engl. *entry*). To su procedure kojima su pridružene tzv. *barijere* (engl. *barrier*), koje predstavljaju Bulove izraze koji moraju da daju rezultat `True` da bi ulaz bio otvoren za pristup.
- Ako je rezultat barijere `False` kada proces poziva ulaz, pozivajući proces se suspenduje sve dok barijera ne promeni rezultat i dok drugi procesi ne napuste objekat. Ovime se obezbeđuje uslovna sinhronizacija.
- Prema tome, zaštićeni objekti predstavljaju kombinaciju klasičnih monitora i uslovnih kritičnih regiona.
- Primer ograničenog bafera:

```
bufferSize : constant Integer :=10;
type Index is mod bufferSize;
subtype Count is Natural range 0..bufferSize;
type Buffer is array (Index) of DataItem;

protected type BoundedBuffer is
  entry get (item : out DataItem);
  entry put (item : in DataItem);
private
  first : Index := Index'First;
  last : Index := Index'Last;
  num : Count := 0;
  buf : Buffer;
end BoundedBuffer;

protected body BoundedBuffer is
  entry get (item : out DataItem) when num /= 0 is
```

```

begin
    item := buf(first);
    first := first + 1;
    num := num - 1;
end get;

entry put (item : in DataItem) when num /= bufferSize is
begin
    last := last + 1;
    buf(last) := item;
    num := num + 1;
end put;
end BoundedBuffer;

```

- Barijere se izračunavaju kada:
  1. proces poziva neki zaštićeni ulaz i pridružena barijera referencira neku promenljivu ili atribut koji se možda promenio od kada je barijera poslednji put izračunavana;
  2. proces napušta zaštićenu proceduru ili ulaz, a postoje procesi koji čekaju na ulazima čije barijere referenciraju promenljive ili attribute koji su se možda promenili od kada je barijera poslednji put izračunavana.

Treba primetiti da se barijera ne izračunava kada neki proces napušta zaštićenu funkciju, jer ona ne može da promeni vrednost promenljivih.

- Zaštićeni objekti u jeziku Ada su veoma nalik objektnim konceptima, ali su samo objektno bazirani, a ne objektno orijentisani, jer ne podržavaju nasleđivanje.

### *Sinhronizovane operacije u jeziku Java*

- Jezik Java podržava koncept monitora u objektnom duhu. U jeziku Java, svaka klasa je implicitno, bilo direktno ili indirektno, izvedena iz ugrađene klase `Object`. Svakom objektu u jeziku Java pridružen je *ključ* (engl. *lock*), kome se ne može pristupiti direktno, ali na koji utiču:
  - operacije klase specifikovane kao `synchronized`
  - sinhronizacija na nivou bloka.
- Pristup do operacije označene kao `synchronized` je omogućen procesu (niti) samo ukoliko on može da dobije ključ pridružen objektu čiju operaciju poziva. Kada jedan proces (nit) dobije ključ, ostali to ne mogu. Na ovaj način se obezbeđuje međusobno isključenje pristupa operacijama označenim kao `synchronized`. Pristup do operacija koje nisu označene kao `synchronized` je uvek omogućen. Na primer:

```

class SharedInteger {

    private int myData;

    public SharedInteger (int initialValue) {
        myData = initialValue;
    };

    public synchronized int read () {
        return myData;
    };

    public synchronized void write (int newValue) {
        myData = newValue;
    };

    public synchronized void incrementBy (int by) {
        myData += by;
    };
}

```

```
};  
}
```

- Sinhronizacija na nivou bloka može da se ostvari zahtevanjem ključa za određeni objekat, naredbom `synchronized` koja kao parametar prima referencu na zahtevani objekat. Na primer, operacija označena kao `synchronized` se implicitno implementira na sledeći način:

```
public int read () {  
    synchronized(this) {  
        return myData;  
    }  
}
```

- Sinhronizaciju na nivou bloka treba upotrebljavati krajnje pažljivo i ograničeno, jer inače ona dovodi do nepreglednog koda, pošto se sinhronizacija na nivou objekta ne može razumeti samo posmatranjem jedne klase, već na tu sinhronizaciju utiče sav kod po kome je rasuta sinhronizacija na nivou bloka.
- Ovakvo zaključavanje ne utiče na statičke podatke članove klase. Međutim, zaključavanje statičkih podataka članova može se postići na sledeći način. Naime, u jeziku Java, za svaku klasu u programu postoji odgovarajući objekat ugrađene klase `Class`. Zaključavanjem ovog objekta, zapravo se zaključavaju statički podaci članovi:

```
synchronized(this.getClass()) {...}
```

- Za uslovnu sinhronizaciju služe sledeće operacije klase `Object` (iz koje su implicitno izvedene sve klase, direktno ili indirektno):

```
public void wait          () throws IllegalMonitorStateException;  
public void notify        () throws IllegalMonitorStateException;  
public void notifyAll     () throws IllegalMonitorStateException;
```

- Ove operacije smeju da se pozivaju samo iz operacija koje drže ključ nad objektom; u suprotnom, podiže se izuzetak tipa `IllegalMonitorStateException`.
- Operacija `wait()` bezuslovno blokira pozivajuću nit i oslobađa ključ nad objektom. Ukoliko je poziv napravljen iz ugnežđenog monitora, oslobađa se ključ samo za taj unutrašnji monitor.
- Operacija `notify()` deblokira jednu nit blokiranu sa `wait()`. Osnovna verzija jezika Java ne definiše koja je to nit od blokiranih, ali Real-Time Java to definiše. Operacija `notify()` ne oslobađa ključ koji pozivajuća nit ima nad objektom, pa deblokirana nit mora da čeka da dobije ključ pre nego što nastavi izvršavanje.
- Slično, operacija `notifyAll()` deblokira sve blokirane niti. Ukoliko blokiranih niti nema, ove dve operacije nemaju efekta.
- Najvažnija razlika između opisanog mehanizma i mehanizma uslovnih promenljivih je da deblokirani proces ne može da računa da je uslov na koji je on čekao ispunjen, pošto su svi blokirani procesi eventualno deblokirani, nezavisno od svog uslova. Međutim, za mnoge slučajeve ovo nije problem, pošto su uslovi međusobno isključivi. Na primer, kod problema ograničenog bafera, ukoliko neki proces čeka na jedan uslov (npr. da se u baferu pojavi element), onda sigurno nema procesa koji čekaju na suprotan uslov (da se u baferu pojavi slobodno mesto). U suprotnom, ukoliko se koristi `notifyAll()`, onda svi procesi moraju da ponovno izračunavaju svoje uslove kada se probude. Primer ograničenog bafera:

```
public class BoundedBuffer {  
  
    private int buffer[];  
    private int first = 0;
```

```

private int last = 0;
private int numberInBuffer = 0;
private int size;

public BoundedBuffer (int length) {
    size = length;
    buffer = new int[size];
};

public synchronized void put (int item) throws InterruptedException {
    while (numberInBuffer == size) wait();
    last = (last + 1) % size ;
    numberInBuffer++;
    buffer[last] = item;
    notifyAll();
};

public synchronized int get () throws InterruptedException {
    while (numberInBuffer == 0) wait();
    first = (first + 1) % size ;
    numberInBuffer--;
    notifyAll();
    return buffer[first];
};
}

```

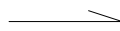
## Klasifikacija poziva operacija

- Posmatrano sa strane pozivaoca, poziv operacije, ulaza, ili nekog drugog servisa klijenta može biti:

- *sinhron* (engl. *synchronous*), što znači da pozivalac čeka dok se pozvana operacija (servis) ne završi, pa tek onda nastavlja dalje svoje izvršavanje; na jeziku UML, ovakav poziv se na dijagramima interakcije označava običnom strelicom:



- *asinhron* (engl. *asynchronous*), što znači da pozivalac ne čeka na završetak pozvane operacije (servisa), već odmah po upućenom pozivu nastavlja svoje izvršavanje; na jeziku UML, ovakav poziv se na dijagramima interakcije označava polu-strelicom:



- U slučaju sinhronog poziva, pozvana operacija može blokirati klijentski proces. Na jeziku UML, potencijalno blokirajući poziv se na dijagramima interakcije označava precrtanom strelicom:



## Implementacija sinhronizacionih primitiva

- U nastavku je opisana realizacija sinhronizacionih primitiva (semafor i događaj) u školskom Jezgru. Takođe je prikazana i realizacija monitora u jeziku C++.

### Semafor

- Klasom `Semaphore` realizovan je koncept standardnog Dijkstra semafora.

- Operacija `signalWait(s1,s2)` izvršava neprekidivu sekvencu operacija `s1->signal()` i `s2->wait()`. Ova operacija je pogodna za realizaciju uslovnih promenljivih.
- Izvorni kod klase `Semaphore`:

```
class Semaphore {
public:
    Semaphore (int initValue=1) : val(initValue) {}
    ~Semaphore ();

    void wait    ();
    void signal ();

    friend void signalWait (Semaphore* s, Semaphore* w);

    int  value () { return val; };

protected:

    void block ();
    void deblock ();

    int val;

private:

    Queue blocked;

};

Semaphore::~~Semaphore () {
    lock();
    for (IteratorCollection* it=blocked->getIterator();
        !it->isDone(); it->next()) {
        Thread* t = (Thread*)it->currentItem();
        Scheduler::Instance()->put(t);
    }
    unlock();
}

void Semaphore::block () {
    if (Thread::runningThread->setContext()==0) {
        // Blocking:
        blocked->put(Thread::runningThread->getCEForSemaphore());
        Thread::runningThread = Scheduler::Instance()->get();
        Thread::runningThread->resume();           // context switch
    } else return;
}

void Semaphore::deblock () {
    // Deblocking:
    Thread* t = (Thread*)blocked->get();
    Scheduler::Instance()->put(t);
}

void Semaphore::wait () {
    lock();
    if (--val<0)
        block();
}
```

```

    unlock();
}

void Semaphore::signal () {
    lock();
    if (val++<0)
        deblock();
    unlock();
}

void signalWait (Semaphore* s, Semaphore* w) {
    lock();
    if (s && s->val++<0) s->deblock();
    if (w && --w->val<0) w->block();
    unlock();
}

```

### ***Događaj***

- Događaj je ovde definisan kao binarni semafor. Izvorni kod za klasu Event izgleda ovako:

```

class Event : public Semaphore {
public:
    Event ();

    void wait    ();
    void signal ();
};

Event::Event () : Semaphore(0) {}

void Event::wait () {
    lock();
    if (--val<0)
        block();
    unlock();
}

void Event::signal () {
    lock();
    if (++val<=0)
        deblock();
    else
        val=1;
    unlock();
}

```

### ***Monitor***

- Na jeziku C++, međusobno isključenje neke operacije (funkcije članice) može da se obezbedi na jednostavan način pomoću semafora:

```

class Monitor {
public:
    Monitor () : sem(1) {}
    void criticalSection ();
};

```

```
private:
    Semaphore sem;
};
```

```
void Monitor::criticalSection () {
    sem.wait();
    //... telo kritične sekcije
    sem.signal();
}
```

- Međutim, opisano rešenje ne garantuje ispravan rad u svim slučajevima. Na primer, ako funkcija vraća rezultat nekog izraza iza naredbe `return`, ne može se tačno kontrolisati trenutak oslobađanja kritične sekcije, odnosno poziva operacije *signal*. Drugi, teži slučaj je izlaz i potprograma u slučaju izuzetka. Na primer:

```
int Monitor::criticalSection () {
    sem.wait();
    return f()+2/x; // gde pozvati signal()?
}
```

- Opisani problem se jednostavno rešava na sledeći način. Potrebno je unutar funkcije koja predstavlja kritičnu sekciju, na samom početku, definisati lokalni automatski objekat koji će u svom konstruktoru imati poziv operacije *wait*, a u destrukturu poziv operacije *signal*. Semantika jezika C++ obezbeđuje da se destruktork ovog objekta uvek poziva tačno na izlasku iz funkcije, pri svakom načinu izlaska (izraz iza `return` ili izuzetak).
- Jednostavna klasa `Mutex` obezbeđuje ovakvu semantiku:

```
class Mutex {
public:

    Mutex (Semaphore* s) : sem(s) { if (sem) sem->wait(); }
    ~Mutex () { if (sem) sem->signal(); }

private:
    Semaphore *sem;
};
```

- Upotreba ove klase je takođe veoma jednostavna: ime samog lokalnog objekta nije uopšte bitno, jer se on i ne koristi eksplicitno.

```
void Monitor::criticalSection () {
    Mutex dummy(&sem);
    //... telo kritične sekcije
}
```

### ***Ograničeni bafer***

- Jedna realizacija ograničenog bafera u školskom Jezgru može da izgleda kao što je prikazano u nastavku.
- Treba primetiti sledeće: operacija oslobađanja kritične sekcije (`mutex.signal()`) i blokiranja na semaforu za čekanje na prazan prostor (`notFull.wait()`) moraju da budu nedeljive, inače bi moglo da se dogodi da između ove dve operacije neki proces uzme poruku iz bafera, a prvi proces se blokira na semaforu `notFull` bez razloga. Isto važi i u operaciji *receive*. Zbog toga je upotrebljena neprekidiva sekvenca `signalWait()`.



- Pomoćna operacija `receive()` koja vraća `int` je neblokirajuća: ako je bafer prazan, ona vraća 0, inače smešta jedan element u argument i vraća 1. Kompletan kod izgleda ovako:

```
class MsgQueue {
public:

    MsgQueue (): mutex(1), notEmpty(0), notFull(0) {}
    ~MsgQueue () { mutex.wait(); }

    void      send      (CollectionElement*);
    Object*   receive   ();           // blocking
    int       receive   (Object**);   // nonblocking
    void      clear     ();

    Object*   first     ();
    int       isEmpty   ();
    int       isFull    ();
    int       size      ();

private:

    Queue rep;
    Semaphore mutex, notEmpty, notFull;

};

void MsgQueue::send (CollectionElement* ce) {
    Mutex dummy(&mutex);
    if (rep.isFull()) {
        signalWait(&mutex,&notFull);
        mutex.wait();
    }
    rep.put(ce);
    if (notEmpty.value()<0) notEmpty.signal();
}

Object* MsgQueue::receive () {
    Mutex dummy(&mutex);
    if (rep.isEmpty()) {
        signalWait(&mutex,&notEmpty);
        mutex.wait();
    }
    Object* temp=rep.get();
    if (notFull.value()<0) notFull.signal();
    return temp;
}

int MsgQueue::receive (Object** t) {
    Mutex dummy(&mutex);
    if (rep.isEmpty()) return 0;
    *t=rep.get();
    if (notFull.value()<0) notFull.signal();
    return 1;
}

void MsgQueue::clear () {
    Mutex dummy(&mutex);
    rep.clear();
}
```

```
}

Object* MsgQueue::first () {
    Mutex dummy(&mutex);
    return rep.first();
}

int MsgQueue::isEmpty () {
    Mutex dummy(&mutex);
    return rep.isEmpty();
}

int MsgQueue::isFull () {
    Mutex dummy(&mutex);
    return rep.isFull();
}

int MsgQueue::size () {
    Mutex dummy(&mutex);
    return rep.size();
}
```

## Vežbe

### 6.1

U školskom Jezgru treba realizovati događaj sa mogućnošću složenog logičkog uslova čekanja. Događaj treba da bude vlasništvo neke niti, a ne nezavisan objekat. Samo nit koja sadrži dati događaj može čekati na tom događaju. Jedna nit sadrži više događaja (konstantan broj  $N$ ). Osim proste operacije čekanja na događaju, može se zadati i složeni uslov čekanja na događajima: operacija `waitAnd(e1,e2)` blokira nit sve dok se ne pojavi signal na oba događaja  $e1$  i  $e2$ .

### 6.2

Korišćenjem školskog Jezgra, realizovati ograničeni bafer (engl. *bounded buffer*) koji će konkurentni aktivni klijenti koristiti kao čuvani (engl. *guarded*) jedinstveni (engl. *singleton*) objekat. Proizvođači (engl. *producers*), odnosno potrošači (engl. *consumers*) mogu da stave, odnosno izvade nekoliko elemenata u toku svog jednog pristupa baferu, ali u jednoj istoj nedeljivoj transakciji. Broj elemenata koji se stavlja, odnosno uzima u jednoj transakciji određuje sam proizvođač, odnosno potrošač sopstvenim algoritmom, pa zato bafer ima samo operacije za stavljanje i uzimanje po jednog elementa. Treba obezbediti i uobičajenu potrebnu sinhronizaciju u slučaju punog, odnosno praznog bafera. Prikazati klase za bafer, proizvođač i potrošač.

### 6.3

Projektuje se konkurentni sistem za modelovanje jednog klijent/server sistema. Server treba modelovati jedinstvenim (*Singleton*) sinhronizovanim objektom (monitorom). Klijenti su aktivni objekti koji ciklično obavljaju svoje aktivnosti. Pre nego što u jednom ciklusu neki klijent započne svoju aktivnost, dužan je da od servera traži dozvolu u obliku "žetona" (*token*). Kada dobije žeton, klijent započinje aktivnost. Po završetku aktivnosti, klijent vraća

žeton serveru. Server vodi računa da u jednom trenutku ne može biti izdato više od N žetona: ukoliko klijent traži žeton, a ne može da ga dobije jer je već izdato N žetona, klijent se blokira. Prikazati rešenje korišćenjem:

- klasičnih monitora i uslovnih promenljivih
- zaštićenih objekata u jeziku Ada
- koncepata iz jezika Java
- školskog Jezgra.

#### 6.4

Potrebno je projektovati sistem za kontrolu saobraćaja u nekom jednosmernom tunelu. U cilju bezbednosti, u tunelu se ne sme nalaziti više od približno N vozila u datom trenutku. Semafor na ulazu u tunnel kontroliše priliv vozila, dok detektori vozila na ulazu i izlazu iz tunela prate tok vozila. Potrebno je napisati dva procesa i monitor kojima se kontroliše tok saobraćaja u tunelu. Prvi proces nadzire ulazni, a drugi izlazni detektor saobraćaja. Monitor kontroliše semafor na ulazu u tunnel. Pretpostavlja se da su realizovane sledeće globalne funkcije:

```
int carsExited (); // Vraća broj vozila koja su napustila tunnel
                    // od trenutka poslednjeg poziva ove funkcije

int carsEntered (); // Vraća broj vozila koja su ušla u tunnel
                    // od trenutka poslednjeg poziva ove funkcije

void setLights (Color); // Postavlja svetlo semafora na zadatu boju:
                        // enum Color {Red, Green};

void delay10Seconds (); // Blokira pozivajući proces na 10 sekundi
```

Program treba da očitava senzore (preko funkcija `carsExited()` i `carsEntered()`) svakih 10 sekundi, sve dok tunnel ne postane prazan ili pun. Kada tunnel postane pun (i na semaforu se upali crveno svetlo), proces koji nadgleda ulaz u tunnel ne treba više kontinualno da poziva funkciju `carsEntered()`. Slično, kada tunnel postane prazan, proces koji nadgleda izlaz iz tunela ne treba više kontinualno da poziva funkciju `carsExited()`. Prikazati rešenje korišćenjem:

- klasičnih monitora i uslovnih promenljivih
- zaštićenih objekata u jeziku Ada
- koncepata iz jezika Java
- školskog Jezgra.

#### 6.5

Posmatra se sistem od tri procesa koji predstavljaju pušače i jednog procesa koji predstavlja agenta. Svaki pušač ciklično zavija cigaretu i puši je. Za zavijanje cigarete potrebna su tri sastojka: duvan, papir i šibica. Jedan pušač ima samo duvan, drugi papir, a treći šibice. Agent ima neograničene zalihe sva tri sastojka. Agent postavlja na sto dva sastojka izabrana slučajno. Pušač koji poseduje treći potreban sastojak može tada da uzme ova dva, zavije cigaretu i puši. Kada je taj pušač popužio svoju cigaretu, on javlja agentu da može da postavi nova dva sastojka, a ciklus se potom ponavlja. Realizovati deljeni objekat koji sinhronizuje agenta i tri pušača. Prikazati rešenje korišćenjem:

- klasičnih monitora i uslovnih promenljivih
- zaštićenih objekata u jeziku Ada
- koncepata iz jezika Java
- školskog Jezgra.

## 6.6

*Multicast* je konstrukt koji omogućava da jedan proces pošalje istu poruku (podatak) grupi procesa koji čekaju na poruku. Data je sledeća specifikacija apstrakcije *Multicast*:

```
class Multicast {  
public:  
    void send (Data* d);  
    Data* receive ();  
};
```

Proces-primatelj izražava svoju želju da primi podatak pozivajući operaciju `receive()`. Ovaj poziv je blokirajući. Proces-pošiljalac šalje poruku pozivom operacije `send()`. Svi procesi koji su trenutno blokirani čekajući da prime poruku oslobađaju se kada se pozove `send()`, a podatak koji je poslat im se prosleđuje. Kada se završi operacija `send()`, svi naredni pozivi operacije `receive()` blokiraju procese do narednog slanja podatka. Realizovati apstrakciju *Multicast* korišćenjem:

- klasičnih monitora i uslovnih promenljivih
- zaštićenih objekata u jeziku Ada
- koncepata iz jezika Java
- školskog Jezgra.

## 6.7

*Broadcast* je sličan konceptu *multicast*, osim što se poruka šalje *svim* učesnicima u sistemu koji mogu da prime poruku. Specifikacija apstrakcije *Broadcast* izgleda isto kao u prethodnom zadatku, ali je razlika u tome što se pošiljalac blokira prilikom slanja poruke sve dok *svi* procesi-primaci (a ima ih tačno  $N$  u sistemu) ne prime poruku. Ako se u međuvremenu pošalje nova poruka, ona se stavlja u red čekanja. Realizovati apstrakciju *Broadcast* korišćenjem:

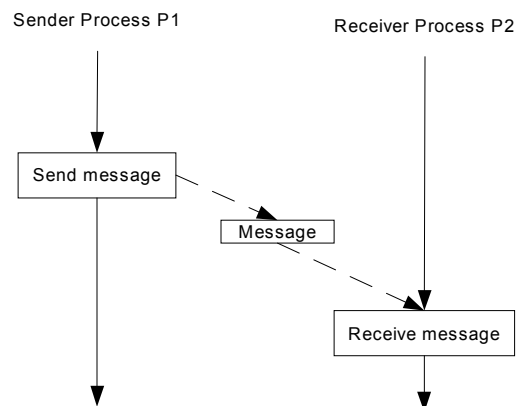
- klasičnih monitora i uslovnih promenljivih
- zaštićenih objekata u jeziku Ada
- koncepata iz jezika Java
- školskog Jezgra.

# Sinhronizacija i komunikacija pomoću poruka

- Alternativa konceptu deljene promenljive jeste *prosleđivanje poruka* (engl. *message passing*). Ideja je zasnovana na upotrebi jedinstvenog konstrukta i za sinhronizaciju i za komunikaciju između procesa.
- Osnova celokupnog pristupa leži u tome da neki proces može da pošalje, a neki proces da primi poruku, uz eventualnu međusobnu sinhronizaciju procesa prilikom slanja i prijema. Međutim, ovako jednostavna osnova je u različitim jezicima realizovana na mnogo različitih načina, koji variraju u odnosu na:
  - model sinhronizacije procesa
  - način imenovanja procesa
  - strukturu poruke.
- Jezik Java ne podržava razmenu poruka neposrednim jezičkim konstruktima. Jezik Ada ima razvijene konstrukte za razmenu poruka, ali će oni ovde biti prikazani samo ukratko. Navedeni elementi biće objašnjeni u načelu, jedan po jedan.

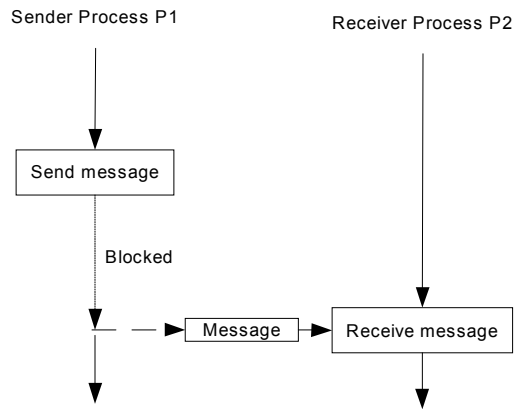
## Sinhronizacija procesa

- Kod komunikacije pomoću poruka postoji implicitna sinhronizacija u smislu da primalac ne može da primi poruku pre nego što je ona poslata. Iako ovo deluje očigledno, treba to uporediti sa komunikacijom pomoću deljene promenljive, gde primalac može da pročita vrednost promenljive ne znajući da li je vrednost upisana od strane pošiljaoca. Kod komunikacije porukama, ukoliko primalac izvršava bezuslovni, blokirajući prijem poruke, on će biti suspendovan dok poruka ne stigne.
- Modeli sinhronizacije variraju u odnosu na to kakva je semantika operacije slanja i mogu se generalno klasifikovati u sledeće kategorije:
  - *Asinhrono*, ili *bez čekanja* (engl., *asynchronous, no-wait*): pošiljalac nastavlja svoje izvršavanje odmah posle slanja poruke, bez čekanja da poruka bude primljena. Ovakav model podržavaju neki programski jezici (npr. CONIC) i POSIX. Analogija: slanje pisama običnom poštom.



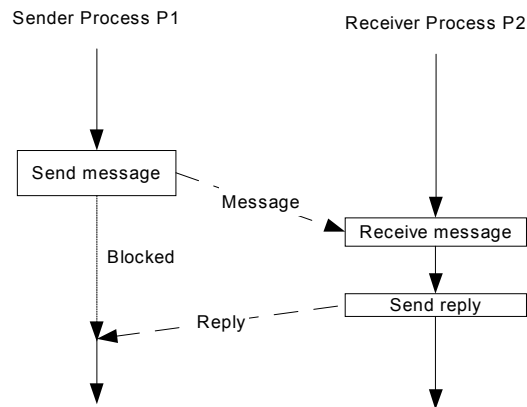
Ovakav pristup zahteva odgovarajuće bafere za prihvatanje poruka i amortizaciju različitih brzina slanja i prijema poruka. Problem je što je prostor za bafere generalno ograničen, pa je pitanje šta se dešava u slučaju punih bafera.

- *Sinhrono*, ili *randevu* (engl. *synchronous, rendez-vous*): pošiljalac se blokira sve dok poruka nije primljena i tek tada nastavlja svoje izvršavanje. Neki jezici podržavaju ovaj koncept (npr. CSP i occam2). Analogija: telefonski poziv.



Ovakav model je pogodan jer ne zahteva bafere za smeštanje poruka.

- *Udaljeni poziv* ili *prošireni randevu* (engl. *remote invocation, extended rendez-vous*): pošiljalac nastavlja izvršavanje tek kada je primalac obradio poruku i poslao odgovor. Ovakav koncept podržavaju razni jezici (npr. Ada, SR, CONIC) i neki operativni sistemi.



- Očigledno je da postoje veze između ovih modela slanja. Na primer, sinhrona komunikacija se može realizovati pomoću asinhronne komunikacije:

Sender process P1:	Receiver process P2:
<code>async_send(message);</code>	<code>receive(message);</code>
<code>receive(acknowledgement);</code>	<code>async_send(acknowledgement);</code>

Slično, udaljeni poziv se može realizovati pomoću sinhronne komunikacije:

Sender process P1:	Receiver process P2:
<code>sync_send(message);</code>	<code>receive(message);</code>
<code>receive(reply);</code>	<code>...</code>
	<code>construct reply;</code>
	<code>...</code>
	<code>sync_send(reply);</code>

- Kako je, prema tome, asinhrono slanje zapravo elementarni konstrukt pomoću koga se mogu realizovati i ostali, može se pomisliti da je on dovoljan i jedini potreban u programskim jezicima. Međutim, upotreba samo ovog modela ima mnogo slabosti:
  - potrebni su potencijalno neograničeni baferi za prihvatanje poruka koje su poslate a još nisu primljene;
  - pošto asinhrona komunikacija uzrokuje "bajate" poruke, najčešće se komunikacija programira tako da se zahteva i potvrda o prijemu, što se onda svodi na sinhronu komunikaciju;
  - zbog toga je potrebno više komunikacije i programi postaju glomazniji i nepregledniji;
  - teže je dokazati korektnost programa.
- U slučaju da jezik podržava sinhronu komunikaciju, asinhrona komunikacija se može jednostavno realizovati konstrukcijom bafera poruka, poput ograničenog bafera (vidi implementaciju klase `MsgBuffer` iz prethodnog poglavlja). Naravno, ovakva implementacija na visokom nivou utiče na degradaciju performansi.

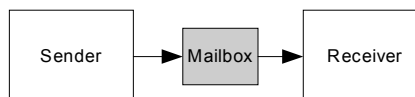
## Imenovanje procesa i struktura poruke

- Imenovanje procesa uključuje dva aspekta:
  - direkciju ili indirekciju u imenovanju
  - simetriju ili asimetriju u imenovanju.
- Direktno imenovanje podrazumeva da pošiljalac neposredno imenuje primaoca kome se poruka šalje:

```
send <message> to <process identification>;
```

- Indirektno imenovanje podrazumeva da između pošiljaoca i primaoca postoji neki međumedium koga pošiljalac imenuje. U zavisnosti od modela, taj medijum se naziva *kanal* (engl. *channel*), *poštansko sanduče* (engl. *mailbox*), *veza* (engl. *link*), *cevovod* (engl. *pipe*), ili *vrata* (engl. *port*). Prema tome, pošiljalac ne imenuje direktno primaoca, već posrednika:

```
send <message> to <mailbox>;
```



Treba primetiti da i u slučaju indirektnog imenovanja slanje može biti sinhrono (pošiljalac čeka da poruka bude primljena).

- Prednost direktnog imenovanja jeste jednostavnost. Prednost indirektnog imenovanja jeste slabija povezanost modula, bolja enkapsulacija i veća fleksibilnost. Medijum za prosleđivanje poruka zapravo predstavlja interfejs između enkapsuliranih modula; promena jednog učesnika u komunikaciji ne zahteva promenu drugog učesnika, već samo prevezivanje na medijum.
- Imenovanje je simetrično ukoliko i pošiljalac i primalac imenuju drugu stranu, makar i indirektno:

```
send <message> to <process identification>;
receive <message> from <process identification>;
```

ili:

```
send <message> to <mailbox>;
receive <message> from <mailbox>;
```

- Imenovanje je asimetrično ukoliko primalac ne imenuje izvor poruke, već prihvata poruku iz bilo kog izvora:

```
receive <message>;
```

- Asimetrično imenovanje odgovara paradigmi klijent/server, gde server obrađuje zahteve od bilo kog klijenta, ne znajući za klijenta.
- Kod indirektnog imenovanja, među-medijum može da dozvoljava sledeće veze između pošiljalaca i primalaca:
  - više-u-jedan: više pošiljalaca može da šalje, a samo jedan primalac da prima;
  - više-u-više: više pošiljalaca može da šalje i više primalaca da prima;
  - jedan-u-jedan: jedan pošiljalac može da šalje i samo jedan primalac da prima;
  - jedan-u-više: jedan pošiljalac može da šalje, a više primalaca da prima.
- Programski jezici koji podržavaju komunikaciju pomoću poruka uglavnom dozvoljavaju da poruka bude objekat bilo kog tipa koji jezik podržava.
- Ukoliko se poruka (objekat) šalje preko mreže na udaljeni procesor, onda je potrebno izvršiti serijalizaciju objekta izvornog tipa u niz bajtova na mestu slanja (engl. *marshalling*) i deserijalizaciju na mestu prijema (engl. *unmarshalling*). Problem mogu da predstavljaju različite implementacije formata tipova na različitim procesorima.
- Operativni sistemi koji podržavaju komunikaciju isključivo dozvoljavaju slanje prostih nizova bajtova.
- Naredni odeljci prikazaće nekoliko koncepata komunikacije pomoću poruka u postojećim jezicima Ada i ROOM (*Real-Time Object-Oriented Modeling*).

## Randevu u jeziku Ada

- Ada podržava prošireni randevu između procesa po principu klijent/server. Serverski proces deklariše svoje servise koje nudi klijentima kao javne *ulaze* (engl. *entry*) u specifikaciji procesa. Svaki ulaz definisan je nazivom i parametrima, koji mogu biti ulazni i izlazni (povratni rezultati). Na primer:

```
task type TelephoneOperator is
  entry directoryEnquiry (person:in Name; addr:in Address; num:out Number);
  -- Other services are possible
end TelephoneOperator;
```

- Klijentski proces poziva servis serverskog procesa u notaciji običnog poziva procedure. Na primer:

```
anOperator : TelephoneOperator;

-- Client task:
task type Subscriber;
task body Subscriber is
begin
  ...
  loop
    ...
    anOperator.directoryEnquiry("John Smith","11 Main Street",johnsNumber);
    ...
  end loop;
  ...
end Subscriber;
```



- Serverski proces navodi mesto svoje spremnosti da prihvati poziv servisa pomoću naredbe `accept`. Na primer:

```
task body TelephoneOperator is
begin
  ...
  loop
    -- Prepare to accept next enquiry
    accept directoryEnquiry (p:in Name; addr:in Address; num:out Number) do
      -- Look up telephone number using p and addr
      num := ... ; -- Prepare the reply
    end directoryEnquiry;
  end loop;
  ...
end TelephoneOperator;
```

- Semantika randevua u jeziku Ada je sledeća. Oba procesa moraju biti spremna da uđu u randevu (i klijent koji izvršava poziv ulaza, i server koji treba da izvrši `accept`). Ukoliko neki proces nije spreman za randevu, onaj drugi mora da čeka. Kada su oba procesa spremna za randevu, ulazni parametri se prosleđuju od klijenta ka serveru. Zatim server nastavlja izvršavanje tela naredbe `accept`. Na kraju izvršavanja naredbe `accept`, izlazni parametri se vraćaju pozivaocu, a zatim oba procesa nastavljaju svoje izvršavanje konkurentno.
- U slučaju da server želi da prihvati *bilo koji* poziv ulaza od strane proizvoljnih klijenata u nekom trenutku, moguće je upotrebiti naredbu `select`. Ova naredba omogućuje izbor bilo kog postojećeg poziva za randevu; ukoliko takvih poziva u datom trenutku ima više, jedan od njih se bira nedeterministički. Na primer:

```
task type TelephoneOperator is
  entry directoryEnquiry (p:Name; a:Address; n : out Number);
  entry directoryEnquiry (p:Name; pc:PostalCode; n:out Number);
  entry reportFault (n:Number);
end TelephoneOperator;

task body TelephoneOperator is
begin
  ...
  loop
    select
      accept directoryEnquiry(p:in Name;addr:in Address;num:out Number) do
        -- Look up telephone number using p and addr
        num := ... ; -- Prepare the reply
      end directoryEnquiry;
    or
      accept directoryEnquiry(p:in Name;pc:in PostalCode;num:out Number) do
        -- Look up telephone number using p and pc
        num := ... ; -- Prepare the reply
      end directoryEnquiry;
    or
      accept reportFault (n:in Number) do
        -- Handle the fault with the number n
      end reportFault;
    end select;
  end loop;
  ...
end TelephoneOperator;
```

- Naredba `accept` može biti uslovljena *čuvarem* (engl. *guard*) koji predstavlja Bulov izraz. Randevu na toj naredbi `accept` može biti ostvaren samo ukoliko je rezultat ovog izraza `True`. Na primer:

```
task body TelephoneOperator is
begin
  ...
  loop
    select
      accept directoryEnquiry(p:in Name;addr:in Address;num:out Number) do
        -- Look up telephone number using p and addr
        num := ... ; -- Prepare the reply
      end directoryEnquiry;
    or
      accept directoryEnquiry(p:in Name;pc:in PostalCode;num:out Number) do
        -- Look up telephone number using p and pc
        num := ... ; -- Prepare the reply
      end directoryEnquiry;
    or
      when workersAvailable =>
        accept reportFault (n:in Number) do
          -- Handle the fault with the number n
        end reportFault;
    end select;
  end loop;
  ...
end TelephoneOperator;
```

- Alternativa u naredbi `select` može biti i naredba `delay`. Ona se izabira ukoliko se randevu ne uspostavi na nekoj od `accept` naredbi u definisanom roku. Na primer, sledeći proces periodično očitava neki senzor svakih 10 sekundi, ali može i da primi zahtev za promenu periode:

```
task SensorMonitor is
  entry newPeriod (p : Duration);
end SensorMonitor;

task body SensorMonitor is
  currentPeriod : Duration := 10.0;
  nextCycle : Time := Clock + currentPeriod;
begin
  loop
    -- Read sensor value etc.
    select
      accept newPeriod (p : Duration) do
        currentPeriod := p;
      end newPeriod;
      nextCycle := Clock + currentPeriod;
    or
      delay until nextCycle;
      nextCycle := nextCycle + currentPeriod;
    end select;
  end loop;
end SensorMonitor;
```

## Komunikacija u metodi ROOM

- ROOM (*Real-Time Object-Oriented Modeling*, Selić et al. 1994) je jezik i metoda za modelovanje RT sistema pokretanih događajima (engl. *event-driven*). To je vizuelni, formalni jezik koji u potpunosti podržava OO koncepte i koji omogućava neposredno izvršavanje visoko apstraktnih modela.
- Osnovna jedinica organizacije modela je *aktor* (engl. *actor*). Aktor je specifikacija klase aktivnih objekata čija se implementacija može definisati na sledeće načine:
  - hijerarhijskom dekompozicijom pomoću ugrađenih aktora;
  - specifikacijom ponašanja pomoću mašine stanja.
- Koncept aktora u potpunosti podržava enkapsulaciju, ne samo tako što klijent ne može pristupiti implementaciji servera, nego i obratno: aktor ne vidi direktno svoje okruženje, već mu može pristupiti samo preko svog sopstvenog interfejsa.
- Interfejs aktora čine *portovi* (engl. *port*). Port referiše *protokol* (engl. *protocol*). Protokol predstavlja klasu koja definiše:
  - smer* toka poruke (ulaz u aktor ili izlaz iz aktora);
  - signal* koji predstavlja identifikaciju poruke i koji pobuđuje mašinu stanja aktora kao događaj;
  - podatke koji predstavljaju parametre signala odnosno poruke.

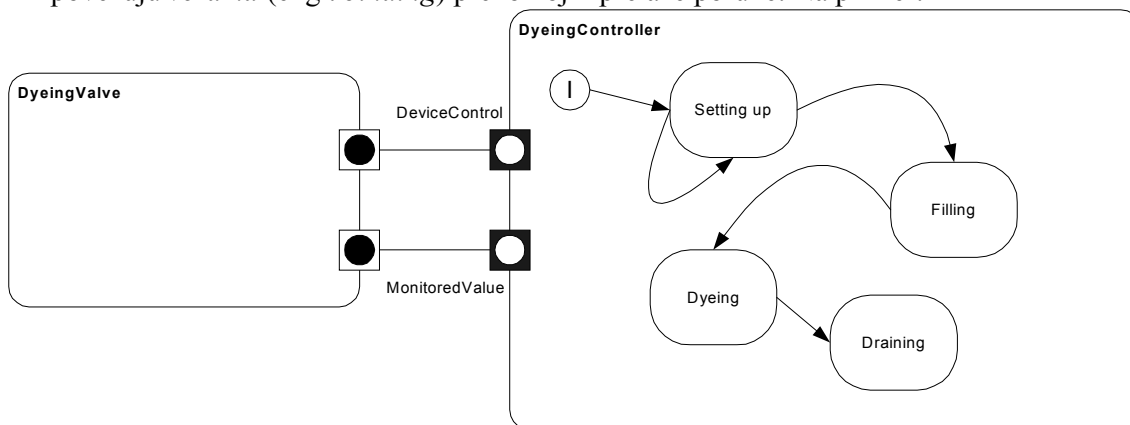
Na primer:

```
protocol class DeviceControl:
```

```
  in: {deviceStatus}
```

```
  out: {deviceCommand}
```

- Svaki port u interfejsu aktora referiše neki protokol, bilo u osnovnom obliku, ili kao *konjugovani* protokol, u kome su smerovi signala obrnuti. Portovi različitih aktora se povezuju *vezama* (engl. *binding*) preko kojih prelaze poruke. Na primer:



- Imenovanje prilikom slanja je indirektno i asimetrično, jer:
  - aktor šalje poruke samo na svoje portove koji primaju odgovarajuće poruke u izlaznom smeru; prilikom slanja se imenuje port na koji se šalje poruka; takve poruke šalju se posredno, preko veza, do portova drugih aktora primalaca;
  - aktor prima poruke sa svojih portova, bez obzira odakle one stižu (serverski koncept); te poruke pobuđuju mašinu stanja kojom je implementiran aktor, ili se prosleđuju ugrađenim aktorima.
- Komunikacija u metodi ROOM je u osnovi asinhrona, ali postoji i sinhroni modalitet slanja poruke.
- Sama poruka je objekat u ciljnom programskom jeziku, koji može biti npr. C++.

## Vežbe

### 7.1

Korišćenjem koncepata procesa i randevua u jeziku Ada, implementirati sistem koji se sastoji od proizvođača, potrošača i ograničenog bafera.

### 7.2

Posmatra se sistem od tri procesa koji predstavljaju pušače i jednog procesa koji predstavlja agenta. Svaki pušač ciklično zavija cigaretu i puši je. Za zavijanje cigarete potrebna su tri sastojka: duvan, papir i šibica. Jedan pušač ima samo duvan, drugi papir, a treći šibice. Agent ima neograničene zalihe sva tri sastojka. Agent postavlja na sto dva sastojka izabrana slučajno. Pušač koji poseduje treći potreban sastojak može tada da uzme ova dva, zavije cigaretu i puši. Kada je taj pušač popuštao svoju cigaretu, on javlja agentu da može da postavi nova dva sastojka, a ciklus se potom ponavlja. Realizovati procese pušača i agenta korišćenjem koncepata procesa i randevua u jeziku Ada.

### 7.3

Korišćenjem koncepata mašine stanja iz jezika UML i koncepata aktora i komunikacije iz metode ROOM, realizovati zahteve iz prethodnog zadatka.

### 7.4

Korišćenjem realizovanih koncepata iz školskog Jezgra, koncipirati podršku za randevu u jeziku C++.

---

# Kontrola resursa

---

- Veliki deo logike konkurentnih programa leži u međusobnom nadmetanju procesa za dobijanje deljenih resursa programa, kao što su deljeni podaci, baferi, memorijski prostor, eksterni uređaji, datoteke i sl. Iako procesi ne moraju međusobno da komuniciraju po pitanju svojih aktivnosti, ipak često moraju posredno da komuniciraju i da se sinhronizuju kako bi se obezbedila koordinacija pristupa do deljenih resursa.
- Kao što je do sada pokazano, implementacija deljenih resursa obično podrazumeva neki agent za kontrolu pristupa resursu. Ukoliko je taj agent pasivan (nema svoj tok kontrole), on se naziva *zaštićenim* (engl. *protected*) ili *sinhronizovanim* (engl. *synchronized*) objektom. Ako je taj agent aktivan (ima svoj tok kontrole), on se obično naziva *serverom* (engl. *server*).
- Ovo poglavlje izlaže najpre nekoliko čestih modela kontrole resursa koji se koriste u konkurentnom programiranju, a zatim diskutuje probleme koji mogu da nastupe kod nadmetanja za deljene resurse.

## Modeli za pristup deljenim resursima

- Jedan od standardnih problema kontrole konkurentnosti jeste problem ograničenog bafera koji je do sada bio detaljno analiziran. Međutim, pored njega, još nekoliko problema se često koristi u teoriji i praksi konkurentnog programiranja za analizu koncepata konkurentnih jezika i konkurentnih algoritama. Ovde će biti razmatrana dva takva problema, problem *čitalaca i pisaca* (engl. *readers-writers*) i problem *filozofa koji večeraju* (engl. *dining philosophers*).

### Čitaoci i pisci

- Koncept monitora obezbeđuje međusobno isključenje pristupa konkurentnih procesa do deljenog resursa, uz eventualnu uslovnu sinhronizaciju. Međutim, koncept potpunog međusobnog isključenja kod monitora ponekad predstavlja suviše restriktivnu politiku koja smanjuje konkurentnost programa.
- Naime, veoma često se operacije nad deljenim resursom mogu svrstati u operacije koje:
  - samo čitaju deljene podatke, odnosno ne menjaju stanje resursa (operacije čitanja)
  - upisuju u deljene podatke, tj. menjaju stanje resursa (operacije upisa).
- Koncept monitora ne dozvoljava nikakvu konkurentnost ovih operacija. Međutim, konkurentnost se može povećati ukoliko se dozvoli da:
  - proizvoljno mnogo procesa izvršava operacije čitanja (tzv. *čitaoci*, engl. *readers*)
  - najviše jedan proces izvršava operaciju upisa (tzv. *pisac*, engl. *writer*), međusobno isključivo sa drugim piscima, ali i sa čitaocima.
- Na taj način, deljenom resursu u datom trenutku može pristupati ili samo jedan pisac, ili više čitalaca, ali ne istovremeno i jedni i drugi. Zato se ovaj koncept naziva *više čitalaca-jedan pisac* (engl. *multiple readers-single writer*).
- Koncept zaštićenog objekta u jeziku Ada inherentno podržava više čitalaca i jednog pisca. Međutim, ukoliko taj koncept nije direktno podržan u jeziku, on se mora realizovati pomoću drugih koncepata.

- Postoje različite varijante ove šeme koje se razlikuju u pogledu prioriteta koji se daje procesima koji čekaju na pristup resursu. Ovde je realizovana sledeća varijanta: prioritet imaju pisci koji čekaju, tj. čim postoji pisac koji čeka, svi novi čitaoci biće blokirani sve dok svi pisci ne završe.
- Implementacija opisane varijante korišćenjem monitora je da monitor poseduje četiri operacije: `startRead`, `stopRead`, `startWrite` i `stopWrite`. Čitaoci i pisci moraju da budu strukturirani na sledeći način:

Reader:	Writer:
<code>startRead();</code>	<code>startWrite();</code>
<code>... // Read data structure</code>	<code>... // Write data structure</code>
<code>stopRead();</code>	<code>stopWrite();</code>

- Na jeziku Java, monitor može da izgleda ovako:

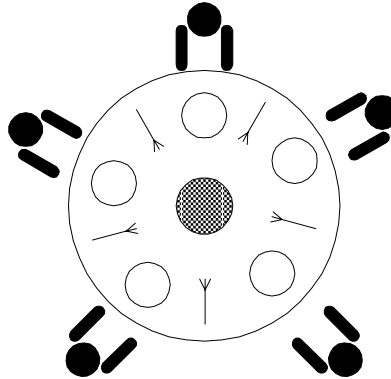
```
public class ReadersWriters {  
  
    private int readers = 0;  
    private int waitingWriters = 0;  
    private boolean writing = false;  
  
    public synchronized void startWrite () throws InterruptedException {  
        while (readers>0 || writing) {  
            waitingWriters++;  
            wait();  
            waitingWriters--;  
        }  
        writing = true;  
    }  
  
    public synchronized void stopWrite () {  
        writing = false;  
        notifyAll();  
    }  
  
    public synchronized void startRead () throws InterruptedException {  
        while (writing || waitingWriters>0) wait();  
        readers++;  
    }  
  
    public synchronized void stopRead () {  
        reader--;  
        if (readers==0) notifyAll();  
    }  
}
```

- Problem sa ovim rešenjem je što se pri svakoj značajnoj promeni (pisac ili poslednji čitalac završava) svi blokirani procesi deblokiraju, pa svi moraju ponovo da izračunaju svoje uslove. Iako se svi deblokiraju, mnogi od njih će ponovo biti blokirani, pa je rešenje neefikasno.

### ***Filozofi koji večeraju***

- Jedan od najstarijih problema koji se najčešće koriste za proveru izražajnosti i upotrebljivosti koncepta konkurentnog programiranja, kao i korektnosti konkurentnih algoritama, jeste problem *filozofa koji večeraju* (engl. *dining philosophers*). Predložio ga je Dijsktra.
- Pet filozofa sedi za okruglim stolom na kom se nalazi posuda sa špagetama, jedu i razmišljaju. Svaki filozof ima svoj tanjir, a između svaka dva susedna tanjira stoji po

jedna viljuška. Pretpostavlja se da su svakom filozofu, da bi se poslužio, potrebne dve viljuške, kao i da može da koristi samo one koje se nalaze levo i desno od njegovog tanjira. Ako je jedna od njih zauzeta, on mora da čeka. Svaki filozof ciklično jede, pa razmišlja. Kad završi sa jelom, filozof spušta obe viljuške na sto i nastavlja da razmišlja. Posle nekog vremena, filozof ogladni i ponovo pokušava da jede. Potrebno je definisati protokol (pravila ponašanja, algoritam) koji će obezbediti ovakvo ponašanje filozofa i pristup do viljušaka. U principu, filozofe predstavljaju procesi, a viljuške su deljeni resursi.



## Problemi nadmetanja za deljene resurse

- Neispravna logika konkurentnih programa može da dovede do različitih *pogrešnih stanja* programa (engl. *error condition*), od kojih su najvažniji:
  - *utrkivanje* (engl. *race condition*)
  - *izgladnjivanje* (engl. *starvation*) ili *neograničeno odlaganje* (engl. *indefinite postponement*)
  - *živo blokiranje* (engl. *livelock*)
  - *mrtvo (kružno) blokiranje* (engl. *deadlock*).
- Da bi bio logički korektan, konkurentan program ne sme da dozvoli mogućnost za nastajanje nekog od ovih pogrešnih stanja.

### Utrkivanje

- Pretpostavimo da okruženje ili operativni sistem podržavaju dve jednostavne primitive:
  - `suspend`: bezuslovno suspenduje (blokira) pozivajući proces;
  - `resume`: bezuslovno deblokira imenovani proces, ukoliko je on suspendovan.
- Pretpostavimo da sinhronizaciju između dva procesa P1 (koji čeka na signal) i P2 (koji postavlja signal) treba obaviti pomoću ovih primitiva i deljene promenljive `flag` tipa Boolean na sledeći način:

```
flag : Boolean := false;
```

```
Process P1:
```

```
...
if not flag then suspend;
flag := false;
...
```

```
Process P2:
```

```
...
flag := true;
P1.resume;
...
```

- Problem ovog rešenja je što može da se dogodi sledeći scenario: proces P1 ispita vrednost promenljive `flag` (koja je `false`), a odmah potom izvršno okruženje ili operativni sistem izvrši preuzimanje i dodeli procesor procesu P2. Proces P2 postavi `flag` na `true` i izvrši `resume` procesa P1 (bez efekta, jer P1 još nije suspendovan). Kada P1 nastavi izvršavanje, on će biti nekorektno suspendovan. Tako ova dva procesa ispadaju iz sinhronizacije.
- Ovakav neispravan uslov naziva se *utrkivanje* (engl. *race condition*) i posledica je toga što se odluka o promeni stanja procesa (suspenciji) donosi na osnovu ispitivanja vrednosti deljene promenljive, pri čemu ta dva koraka nisu nedeljiva, pa može doći do preuzimanja, tj. "utrkivanja" od strane drugog procesa koji prisutpa istoj deljenoj promenljivoj.
- Sličan problem bi postojao u implementaciji ograničenog bafera u školskom Jezgru, ukoliko operacije `signal` i `wait` koje obezbeđuju uslovnu sinhronizaciju ne bi bile nedeljive:

```
void MsgQueue::send (CollectionElement* ce) {
    Mutex dummy(&mutex);
    if (rep.isFull()) {
        mutex.signal();           // Race condition!
        notFull.wait();
        mutex.wait();
    }
    rep.put(ce);
    if (notEmpty.value() < 0) notEmpty.signal();
}
```

```
Object* MsgQueue::receive () {
    Mutex dummy(&mutex);
    if (rep.isEmpty()) {
        mutex.signal();           // Race condition!
        notEmpty.wait();
        mutex.wait();
    }
    Object* temp=rep.get();
    if (notFull.value() < 0) notFull.signal();
    return temp;
}
```

### Mrtvo blokiranje

- Posmatrajmo sledeći algoritam po kome postupa svaki filozof u primeru filozofa koji večeraju. Pretpostavlja se da su viljuške resursi za koje je obezbeđena ekskluzivnost pristupa, tako da se proces filozofa blokira ukoliko je viljuška zauzeta:

```
task type Philosopher
loop
    think;
    take left fork;
    take right fork;
    eat;
    release left fork;
    release right fork;
end;
end;
```

- Ovaj algoritam može biti implementiran korišćenjem semafora na sledeći način:

```
Semaphore forks[5];

class Philosopher : public Thread {
public:
```



```

    Philosopher (int orderNumber)
        : myNum(orderNumber), left(myNum-1), right((myNum+1)%5) {
            if (myNum==0) left=4;
        }

protected:
    virtual void run ();

    void eat();
    void think();

private:
    int myNum, left, right;
};

void Philosopher::run () {
    while (1) {
        think();
        forks[left].wait();
        forks[right].wait();
        eat();
        forks[left].signal();
        forks[right].signal();
    }
}

...
Philosopher p0(0), p1(1), p2(2), p3(3), p4(4);
p0.start(); p1.start(); p2.start(); p3.start(); p4.start();

```

- Problem ovog rešenja je što može nastati sledeći scenario: svi filozofi uporedo uzmu po jednu viljušku, svaki svoju levu, a onda se blokiraju prilikom pristupa do svoje desne viljuške, jer su sve viljuške zauzete. Tako svi procesi ostaju trajno kružno blokirani.
- Ovakav neregularan uslov koji nastaje tako što se grupa procesa koji konkurišu za deljene resurse međusobno kružno blokiraju, naziva se *mrtvo* (ili *kružno*) *blokiranje* (engl. *deadlock*).
- U opštem slučaju, mrtvo blokiranje nastaje tako što se grupa procesa nadmeće za ograničene resurse, pri čemu proces *P1* drži ekskluzivan pristup do resursa *R1* i pri tom čeka blokiran da se oslobodi resurs *R2*, proces *P2* drži ekskluzivan pristup do resursa *R2* i pri tom čeka blokiran da se oslobodi resurs *R3*, itd., proces *Pn* drži ekskluzivan pristup do resursa *Rn* i pri tom čeka blokiran da se oslobodi resurs *R1*. Tako procesi ostaju neograničeno suspendovani u cikličnom lancu blokiranja.
- Mrtvo blokiranje je jedan od najtežih problema koji mogu da se pojave u konkurentnim programima.
- Postoje četiri neophodna uslova za nastanak mrtvog blokiranja:
  - *međusobno isključenje* (engl. *mutual exclusion*): samo jedan proces može koristiti resurs u jednom trenutku; drugim rečima, resursi nisu deljivi ili je pristup do njih ograničen;
  - *držanje i čekanje* (engl. *hold and wait*): moraju postojati procesi koji drže zauzete resurse i pri tom čekaju na druge;
  - *nema preuzimanja resursa* (engl. *no preemption*): resurs može biti samo dobrovoljno oslobođen od strane procesa koji ga je zauzeo;
  - *kružno čekanje* (engl. *circular wait*): mora postojati cikličan lanac procesa tako da svaki proces u lancu drži resurs koga traži naredni proces u lancu.
- Da bi program bio pouzdan, on mora da bude zaštićen od mrtvog blokiranja (engl. *deadlock-free*). Postoji nekoliko pristupa za rešavanje ovog problema:

- sprečavanje mrtvog blokiranja (engl. *deadlock prevention*);
- izbegavanje mrtvog blokiranja (engl. *deadlock avoidance*);
- detekcija i oporavak od mrtvog blokiranja (engl. *deadlock detection and recovery*).

### *Sprečavanje mrtvog blokiranja*

- Sprečavanje mrtvog blokiranja uglavnom se svodi na eliminisanje bar jednog od neophodnih uslova za njegovo nastajanje:
  - *Međusobno isključenje*. Ako su resursi deljivi, onda oni ne mogu da izazovu mrtvo blokiranje. Nažalost, resursi veoma retko dozvoljavaju konkurentan pristup.
  - *Držanje i čekanje*. Jedan jednostavan način za sprečavanje mrtvog blokiranja je da procesi traže sve potrebne resurse pre svog izvršavanja ili u trenutku kada ne drže druge resurse zauzetim. Nažalost, ovakav pristup je veoma neefikasan i retko izvodljiv.
  - *Nema preuzimanja*. Mrtvo blokiranje se može sprečiti ukoliko se uslov da resursi ne mogu biti preuzeti relaksira. Postoji nekoliko pristupa za to: proces oslobađa sve zauzete resurse ukoliko pokuša i ne uspe da zauzme novi resurs, ili preuzima zauzeti resurs od drugog procesa koji je blokiran čekajući na drugi resurs. Nedostatak ovog pristupa je što se prilikom preotimanja resursa stanje tog resursa mora sačuvati, što je često neizvodljivo.
  - *Kružno čekanje*. Da bi se izbeglo kružno čekanje, moguće je uvesti linearno uređenje svih resursa. Pri tom se svakom resursu  $R_i$  dodeljuje redni broj  $F(R_i)$  prema tom uređenju. Tada se procesu koji je zauzeo resurs  $R_i$  dozvoljava da zauzme novi resurs  $R_j$  samo ukoliko je  $F(R_i) < F(R_j)$ . Alternativno, proces može da zauzme novi resurs  $R_j$  samo ukoliko prethodno oslobodi sve resurse  $R_i$  za koje je  $F(R_i) < F(R_j)$ . Funkciju  $F$  treba definisati prema načinu korišćenja resursa. Lako je pokazati da se u ovom slučaju ne može dogoditi kružno čekanje.
- Drugi način za izbegavanje kružnog blokiranja je da se konkurentni program formalno analizira i dokaže da je kružno blokiranje nemoguće. Takva formalna analiza se često svodi na ispitivanje skupa svih diskretnih stanja u koja program može da uđe, pri čemu se za stanja posmatraju diskretni trenuci sinhronizacije. Nažalost, ovo je često veoma teško izvesti. Takva analiza dosta zavisi od modela konkurentnosti.

### *Izbegavanje mrtvog blokiranja*

- Ukoliko je poznat obrazac po kome procesi koriste resurse, moguće je formirati algoritam koji će dozvoliti postojanje sva četiri uslova za nastanak mrtvog blokiranja, ali koji će obezbediti da do takvog blokiranja ne dođe.
- Algoritmi za izbegavanje mrtvog blokiranja dinamički (u toku izvršavanja programa) analiziraju stanje zauzetosti resursa i preduzimaju akcije da do blokiranja ne dođe. Pri tome, nije dovoljno samo uvideti da naredno stanje sistema predstavlja mrtvo blokiranje, jer u tom slučaju nije moguće preduzeti odgovarajuću akciju. Potrebno je odrediti da li sistem u narednom stanju ostaje u tzv. *bezbednom stanju* (engl. *safe state*).
- Stanje sistema u nekom trenutku izvršavanja, u smislu alokacije resursa, definiše se kao skup sledećih podataka:
  - broj raspoloživih (slobodnih) resursa
  - broj alociranih resursa za svaki proces
  - broj zahtevanih resursa za svaki proces.

- Sistem je u *bezbednom stanju* ukoliko se može pronaći način, tj. redosled alokacije resursa od strane procesa, takav da se svakom procesu dodeli još onoliko resursa koliko on zahteva i pri tome ipak izbegne mrtvo blokiranje.
- Na primer, neka u sistemu postoji 12 jedinica resursa i tri procesa:  $P_0$ , koji maksimalno zahteva 10 jedinica resursa,  $P_1$  koji zahteva 4 jedinice i  $P_2$  koji zahteva 9 jedinica. Neka u datom trenutku  $T_i$  proces  $P_0$  zauzima 5,  $P_1$  zauzima 2, a  $P_2$  zauzima 2 jedinice resursa:

Proces	Zauzeo	Traži
$P_0$	5	10
$P_1$	2	4
$P_2$	2	9
Ukupno zauzeto: 9		
Slobodnih: 3		

Ovo stanje je bezbedno, jer postoji sekvenca izvršavanja  $\langle P_1, P_0, P_2 \rangle$  koja omogućava da svi procesi završe svoje izvršavanje.

Ako u trenutku  $T_{i+1}$  proces  $P_2$  zahteva i dobija još jednu jedinicu, stanje postaje:

Proces	Zauzeo	Traži
$P_0$	5	10
$P_1$	2	4
$P_2$	3	9
Ukupno zauzeto: 10		
Slobodnih: 2		

Ovo stanje više nije bezbedno, jer sa slobodnim resursima samo  $P_1$  može da završi, ostavljajući  $P_0$  sa 5 i  $P_2$  sa 3 zauzeta resursa, ali sa samo 4 slobodna resursa. Ni  $P_0$  ni  $P_2$  ne može da završi, pa sistem može da uđe u mrtvu blokadu.

Prema tome, ukoliko sistem ima ugrađen algoritam za izbegavanje blokiranja, on će odbiti zahtev procesa  $P_2$  u trenutku  $T_{i+1}$  i blokirati ga.

- Stanje mrtvog blokiranja je, očigledno, nebezbedno stanje. Treba, međutim, primetiti da sistem koji je u nebezbednom stanju može, ali *ne mora* da uđe u mrtvu blokadu. Važno je, naravno, da sistem koji je u bezbednom stanju *sigurno neće* ući u mrtvu blokadu.
- U slučaju postojanja više tipova resursa, algoritmi izbegavanja mrtvog blokiranja postaju složeniji.

#### Detekcija i oporavak od mrtvog blokiranja

- U većini konkurentnih sistema opšte namene, način korišćenja resursa od strane procesa nije unapred poznat ili je cena izbegavanja mrtvog blokiranja prevelika. U takvim sistemima može se pribеći *detekciji i oporavku* od mrtvog blokiranja.
- Detekcija stanja mrtve blokade se uglavnom svodi na kreiranje *grafa alokacije resursa* od strane procesa. Formiranje takvog grafa zahteva informaciju o tome koji je proces zauzeo koji resurs ili je blokiran tražeći neki resurs. Ukoliko u takvom grafu postoji petlja, u sistemu postoji mrtva blokada. Zbog toga se ovaj graf može koristiti i za izbegavanje blokade, tako što se prilikom alokacije resursa ispituje da li se u grafu zatvara neka petlja.
- Oporavak od ovakvog stanja je mnogo teži problem. Oporavak se može izvesti deljenjem nekog zauzetog resursa, ukidanjem nekog procesa, ili preuzimanjem nekog zauzetog resursa. Međutim, svaki od ovih pristupa ima svoje ozbiljne nedostatke.
- U svakom slučaju, bilo koji pristup za izbegavanje ili rešavanje mrtvog blokiranja je veoma skup i često neprihvatljiv za RT sisteme. Zbog toga se često pribegava jednostavnom pristupu čekanja na zauzeti resurs sa *vremenskim ograničenjem* (engl. *time-out*): kada traži neki resurs koji je zauzet, proces se blokira, ali samo na ograničeno

vreme; ukoliko se u tom roku resurs ne oslobodi, proces se deblokira (smatrajući da je došlo do eventualne mrtve blokade) i preduzima neku alternativnu akciju.

### Živo blokiranje

- Posmatrajmo algoritam rada filozofa koji pokušava da izbegne mrtvo blokiranje na sledeći način:

```
task type Filosofpher
loop
  think;
  loop
    take_left_fork;
    if can_take_right_fork then
      take_right_fork;
      exit loop;
    else
      release_left_fork;
    end if;
  end;
  eat;
  release_left_fork;
  release_right_fork;
end;
end;
```

- Problem ovog rešenja je što može nastati sledeći scenario. Svi filozofi istovremeno uzmu svoju levu viljušku. Zatim svi zaključe da ne mogu da uzmu svoju desnu viljušku, jer je ona zauzeta, pa spuštaju svoju levu viljušku. Teorijski, ovaj postupak se može beskonačno ponavljati, što znači da se svi procesi izvršavaju, tj. ne postoji mrtvo blokiranje, ali nijedan proces ne nastavlja dalje svoj koristan rad.
- Ovakva neregularna situacija u konkurentnom programu, kod koje se grupa procesa izvršava, ali nijedan ne može da napreduje jer u petlji čeka na neki uslov, naziva se *živo blokiranje* (engl. *livelock*).
- Treba razlikovati živo od mrtvog blokiranja. Iako se u oba slučaja procesi nalaze "zaglavljani" čekajući na ispunjenje nekog uslova, kod mrtvog blokiranja su oni suspendovani, dok se kod živog izvršavaju, tj. uposlano čekaju. Obe situacije su neispravna stanja u kojima program ne može da napreduje u korisnom smeru, tj. nije obezbeđena njegova *živost* (engl. *liveness*). Živost konkurentnog programa znači da nešto što treba da se desi, da se konačno i desi.
- Isti problem postoji kod ranije pokazane varijante međusobnog isključenja pomoću uposlenog čekanja:

```
process P1
begin
  loop
    flag1 := true; (* Announce intent to enter *)
    while flag2 = true do (* Busy wait if the other process is in *)
      null
    end;
    <critical section> (* Critical section *)
    flag1 := false; (* Exit protocol *)
    <non-critical section>
  end
end P1;

process P2
begin
  loop
```

```

    flag2 := true;                (* Announce intent to enter *)
    while flag1 = true do        (* Busy wait if the other process is in *)
        null
    end;
    <critical section>           (* Critical section *)
    flag2 := false;              (* Exit protocol *)
    <non-critical section>
end
end P2;
```

### Izgladnjivanje

- Posmatrajmo algoritam rada filozofa koji pokušava da izbegne mrtvo i živo blokiranje na sledeći način:

```

task type Filosofpher
loop
    think;
    take_both_forks;
    eat;
    release_both_forks;
end;
end;
```

Podrazumeva se da je operacija uzimanja obe viljuške atomična.

- Problem ovog rešenja je što može nastati sledeći scenario. Posmatrajmo filozofa X. Neka je njegov levi sused označen sa L, a desni sa D. U jednom trenutku L može da uzme obe svoje viljuške, što sprečava filozofa X da uzme svoju levu viljušku. Pre nego što L spusti svoje viljuške, D može da uzme svoje, što opet sprečava filozofa X da počne da jede. Teorijski, ovaj postupak se može beskonačno ponavljati, što znači da filozof X nikako ne uspeva da zauzme svoje viljuške (resurse) i počne da jede, jer njegovi susedi naizmenično uzimaju njegovu levu, odnosno desnu viljušku.
- Ovakva neregularna situacija u konkurentnom programu, kod koje jedan proces ne može da dođe do željenog resursa jer ga drugi procesi neprekidno pretiču i zauzimaju te resurse, naziva se *izgladnjivanje* (engl. *starvation*), ili *neograničeno odlaganje* (engl. *indefinite postponement*), ili *lockout*.
- Prema tome, živost programa može da bude ugrožena neregularnim situacijama kao što su mrtvo blokiranje, živo blokiranje i izgladnjivanje.
- Isti problem izgladnjivanja postoji kod protokola sistema čitalaca i pisaca gde čitaoci imaju prioritet nad piscima koji čekaju, jer u slučaju da su neki čitaoci aktivni, moguće je da novi čitaoci stalno pristižu, tako da pisac ostaje da čeka neograničeno.

### Jedno rešenje problema filozofa

- Jedno moguće rešenje problema filozofa koji večeraju, a koje ne poseduje probleme mrtvog i živog blokiranja, kao ni izgladnjivanja, jeste sledeće. Svaki filozof uzima najpre svoju levu, pa onda svoju desnu viljušku. Da bi se sprečilo mrtvo blokiranje, uzimanje leve viljuške dozvoljava se samo prvoj četvorici koji to pokušaju; ukoliko i peti filozof u nekom trenutku želi da uzme svoju levu viljušku, on se blokira.
- Ovo rešenje se jednostavno implementira pomoću semafora, pri čemu pet semafora predstavlja viljuške, a još jedan semafor, koji ima inicijalnu vrednost 4, služi za izbegavanje mrtvog blokiranja i sprečavanje petog filozofa da uzme svoju viljušku:

```

Semaphore forks[5]; // Initially set to 1
Semaphore deadlockPrevention(4);
```

```
...  
void Philosopher::run () {  
    while (1) {  
        think();  
        deadlockPrevention.wait();  
        forks[left].wait();  
        forks[right].wait();  
        eat();  
        forks[left].signal();  
        forks[right].signal();  
        deadlockPrevention.signal();  
    }  
}
```

## Vežbe

### 8.1

Navesti još neke varijante prioritiranja čitalaca i pisaca osim opisane. Modifikovati datu implementaciju monitora tako da podrži te varijante. Prodiskutovati eventualne probleme koji mogu da nastanu u tim varijantama (eventualna kružna blokiranja i izgladnjivanje).

### 8.2

Implementirati opisanu varijantu čitalaca i pisaca korišćenjem standardnih monitora i uslovnih promenljivih `okToRead` i `okToWrite`.

### 8.3

Realizovati najjednostavniju varijantu čitalaca i pisaca korišćenjem školskog Jezgra.

### 8.4

Prikazati realizaciju sistema čitalaca i pisaca na jeziku Java, pri čemu se prioritet daje čitaocima, a piscima se garantuje pristup u FIFO (*First-In-First-Out*) redosledu.

### 8.5

Precizno formulisati postupak formiranja grafa alokacije resursa u cilju detekcije mrtvog blokiranja.

### 8.6

U cilju izbegavanja mrtvog blokiranja, primenjuje se tehnika čekanja sa vremenskim ograničenjem (engl. *timeout*) na zauzeti resurs. Kako procesu koji zahteva resurs dojaviti da resurs nije zauzet, već da je isteklo vreme čekanja? Prikazati kako izgleda deklaracija operacije zauzimanja resursa i deo koda procesa koji zahteva taj resurs na jeziku Java ili C++. Odgovor prikazati na primeru proizvođača koji periodično proizvodi podatke i šalje ih u bafer, pri čemu u slučaju isteka vremena čekanja na smeštanje u bafer proizvođač povećava svoju periodu rada. (Pretpostaviti da funkcija `delay(int)` blokira pozivajući proces na vreme zadato argumentom.)

**8.7**

Posmatra se sistem sa pet procesa  $P_1, P_2, \dots, P_5$  i sedam tipova resursa  $R_1, R_2, \dots, R_7$ . Postoji po jedna instanca resursa 2, 5 i 7, a po dve resursa 1, 3, 4 i 6. Proces 1 je zauzeo jednu instancu  $R_1$  i zahteva jednu instancu  $R_7$ . Proces 2 je zauzeo po jednu instancu  $R_1, R_2$  i  $R_3$  i zahteva jednu instancu  $R_5$ . Proces 3 je zauzeo po jednu instancu  $R_3$  i  $R_4$  i zahteva jednu instancu  $R_1$ . Proces 4 je zauzeo po jednu instancu  $R_4$  i  $R_5$  i zahteva jednu instancu  $R_2$ . Proces 5 je zauzeo jednu instancu  $R_7$ . Da li je ovaj sistem u mrtvoj blokadi?

**8.8**

Neki sistem je u stanju prikazanom u tabeli. Da li je ovaj sistem u bezbednom ili nebezbednom stanju?

Proces	Zauzeo	Traži
$P_0$	2	12
$P_1$	4	10
$P_2$	2	5
$P_3$	0	5
$P_4$	2	4
$P_5$	1	2
$P_6$	5	13
Slobodnih: 1		

**8.9**

Implementirati prikazani algoritam filozofa koji poseduje problem živog blokiranja, korišćenjem proizvoljnih koncepata za sinhronizaciju.

**8.10**

Implementirati prikazani algoritam filozofa koji poseduje problem izgladnjivanja, korišćenjem proizvoljnih koncepata za sinhronizaciju.

# **VI Specifičnosti RT programiranja**



---

# Realno vreme

---

- Pojam *realno vreme* (engl. *real time*) odnosi se na protok fizičkog vremena, nezavisno od rada računarskog sistema ili relativnog napredovanja konkurentnih procesa.
- Za RT sisteme je veoma važno da programski jezik ili okruženje za programiranje obezbeđuje usluge (engl. *facilities*) vezane za realno vreme. Te usluge se tipično grade nad konceptima konkurentnog programiranja.
- Usluge vezane za realno vreme tipično uključuju sledeće aspekte:
  - predstavu o protoku vremena, kao što je pristup časovniku realnog vremena (tj. informacija o apsolutnom datumu i vremenu), merenje proteklog vremena, kašnjenje procesa za zadato vreme, programiranje vremenskih kontrola (engl. *timeout*), itd.;
  - predstavljanje vremenskih zahteva, npr. zadavanje krajnjih rokova ili periode procesa;
  - zadovoljavanje vremenskih zahteva.

## Časovnik realnog vremena

- Prvi servis vezan za realno vreme odnosi se na dobijanje informacije o *apsolutnom realnom vremenu* (engl. *absolute real time*), tj. o datumu i vremenu u realnom svetu.
- Protok realnog vremena može da se prati na sledeće načine:
  - izvršno okruženje pobuđuje periodični hardverski prekid, a okruženje "odbrojava" te prekide; okruženje obezbeđuje aplikaciji informaciju u odgovarajućem formatu, preko programskog interfejsa ili koncepta direktno ugrađenog u jezik;
  - postoji izdvojeni hardverski uređaj (časovnik) koji obezbeđuje dovoljno tačnu aproksimaciju protoka vremena; aplikacija pristupa tom časovniku kao što inače pristupa uređajima.
- Poseban problem predstavlja sinhronizacija lokalnih časovnika realnog vremena u distribuiranim sistemima.
- Mnogi programski jezici, a među njima i Ada, Java i C/C++, nemaju neposredne jezičke konstrukte za očitavanje realnog vremena, već obezbeđuju standardne bibliotečne servise.

### Časovnik u jeziku Ada

- Pristup časovniku realnog vremena u jeziku Ada omogućen je preko standardnog paketa `Calendar`. Ovaj paket realizuje apstraktni tip podataka `Time`, koji predstavlja vremenski trenutak u realnom vremenu, funkciju `Clock`, koja vraća trenutno vreme, kao i niz operacija za konverziju tipa `Time` u ljudski čitljive informacije (godina, mesec, dan itd.):

```
package Ada.Calendar is
  type Time is private;
  subtype Year_Number is Integer range 1901..2099;
  subtype Month_Number is Integer range 1..12;
  subtype Day_Number is Integer range 1..31;
  subtype Day_Duration is Duration range 0.0..86_400.0;

  function Clock return Time;
```

```

function Year(Date:Time) return Year_Number;
function Month(Date:Time) return Month_Number;
function Day(Date:Time) return Day_Number;
function Seconds(Date:Time) return Day_Duration;

procedure Split(Date:in Time; Year:out Year_Number;
    Month:out Month_Number; Day:out Day_Number;
    Seconds:out Day_Duration);
function Time_Of(Year:Year_Number; Month:Month_Number;
    Day:Day_Number; Seconds:Day_Duration := 0.0) return Time;

function "+"(Left:Time; Right:Duration) return Time;
function "+"(Left:Duration; Right:Time) return Time;
function "-"(Left:Time; Right:Duration) return Time;
function "-"(Left:Time; Right:Time) return Duration;
function "<"(Left,Right:Time) return Boolean;
function "<=" (Left,Right:Time) return Boolean;
function ">"(Left,Right:Time) return Boolean;
function ">=" (Left,Right:Time) return Boolean;

Time_Error:exception;
-- Time_Error may be raised by Time_Of,
-- Split, Year, "+" and "-"
private
    implementation-dependent
end Ada.Calendar;

```

- Tip `Duration` je predefinisani racionalni tip u fiksnom zarezu koji predstavlja interval vremena izražen u sekundama. Njegova tačnost i opseg zavisi su od implementacije, ali njegov opseg mora biti najmanje od  $-86400.00$  do  $86400.00$ , što pokriva broj sekundi u danu, a granularnost mora biti bar 20 milisekundi.
- Opcioni paket `Real_Time` u aneksu jezika koji se naziva Real-Time Ada obezbeđuje sličan pristup satu realnog vremena, ali sa finijom granularnošću. Konstanta `Time_Unit` predstavlja najmanji interval vremena koji se može predstaviti tipom `Time`. Vrednost `Tick` ne sme biti veća od jedne milisekunde. Opseg tipa `Time`, koji predstavlja vreme od početka izvršavanja programa, mora biti najmanje 50 godina:

```

package Ada.Real_Time is
    type Time is private;
    Time_First: constant Time;
    Time_Last: constant Time;
    Time_Unit: constant := implementation_defined_real_number;

    type Time_Span is private;
    Time_Span_First: constant Time_Span;
    Time_Span_Last: constant Time_Span;
    Time_Span_Zero: constant Time_Span;
    Time_Span_Unit: constant Time_Span;

    Tick: constant Time_Span;
    function Clock return Time;

    function "+" (Left: Time; Right: Time_Span) return Time;
    function "+" (Left: Time_Span; Right: Time) return Time;
    -- similarly for "-", "<", etc.

    function To_Duration(TS: Time_Span) return Duration;
    function To_Time_Span(D: Duration) return Time_Span;
    function Nanoseconds (NS: Integer) return Time_Span;
    function Microseconds (US: Integer) return Time_Span;
    function Milliseconds (MS: Integer) return Time_Span;

```

```

type Seconds_Count is range implementation-defined;
procedure Split(T : in Time; SC: out Seconds_Count; TS : out Time_Span);
function Time_Of(SC: Seconds_Count; TS: Time_Span) return Time;

private
  -- not specified by the language
end Ada.Real_Time;

```

### Časovnik u jeziku Java

- Standardna Java podržava pristup časovniku realnog vremena na sličan način kao i Ada. U paketu `java.lang` postoji klasa `System`, čija statička operacija `currentTimeMillis()` vraća broj milisekundi proteklih od ponoći, 1. januara 1970. po Griniču. Klasa `Date` obezbeđuje apstraktni tip podataka za datum i vreme, uz operacije za konverziju.
- RT Java još pruža i usluge vezane za časovnik realnog vremena i vremenske tipove visoke rezolucije. Apstraktna klasa `HighResolutionTime` predstavlja generalizaciju takvih tipova:

```

public abstract class HighResolutionTime implements java.lang.Comparable {
    ...
    public boolean equals(HighResolutionTime time);

    public final long getMilliseconds();
    public final int getNanoseconds();
    public void set(HighResolutionTime time);
    public void set(long millis);
    public void set(long millis, int nanos);
}

```

- Tri izvedene klase jesu `AbsoluteTime`, `RelativeTime` i `RationalTime`. Klasa `AbsoluteTime` predstavlja realno vreme izraženo relativno u odnosu na 1. januar 1970. Klasa `RelativeTime` predstavlja interval vremena (kao `Duration` u jeziku Ada). Klasa `RationalTime` predstavlja relativno vreme (interval), koje se odnosi na periodu dešavanja nekih događaja (npr. periodičnih procesa):

```

public class AbsoluteTime extends HighResolutionTime {
    // Various constructor methods including:
    public AbsoluteTime(AbsoluteTime t);
    public AbsoluteTime(long millis, int nanos);

    public AbsoluteTime add(long millis, int nanos);
    public final AbsoluteTime add(RelativeTime time);
    ...
    public final RelativeTime subtract(AbsoluteTime time);
    public final AbsoluteTime subtract(RelativeTime time);
}

public class RelativeTime extends HighResolutionTime {
    // Various constructor methods including:
    public RelativeTime(long millis, int nanos);
    public RelativeTime(RelativeTime time);
    ...
    public RelativeTime add(long millis, int nanos);
    public final RelativeTime add(RelativeTime time);
    public void addInterarrivalTo(AbsoluteTime destination);
    public final RelativeTime subtract(RelativeTime time);
    ...
}

```

- Apstraktna klasa `Clock` predstavlja generalizaciju časovnika koji se mogu kreirati u programu. Jezik dozvoljava formiranje proizvoljno mnogo vrsta časovnika, npr. jedan koji meri proteklo vreme od početka izvršavanja programa. Pri tom, uvek postoji jedan časovnik realnog vremena kome se pristupa statičkom operacijom `getRealTimeClock()`:

```
public abstract class Clock {
    public Clock();

    public static Clock getRealtimeClock();

    public abstract RelativeTime getResolution();
    public abstract AbsoluteTime getTime();
    public abstract void getTime(AbsoluteTime time);
    public abstract void setResolution(RelativeTime resolution);
}
```

## Merenje proteklog vremena

- Merenje proteklog vremena, npr. vremena izvršavanja nekog dela koda, ili vremena proteklog između dva događaja, ili vremena čekanja na neki događaj, predstavlja sledeći važan element vezan za vreme u RT programima.
- Ukoliko jezik ili okruženje podržava pristup časovniku realnog vremena, informacija o proteklom vremenu dobija se jednostavno. Na primer, u jeziku Ada, to se može uraditi na sledeći način:

```
declare
    oldTime, newTime : Time;
    interval : Duration;
begin
    oldTime := Clock;
    -- Other computations
    newTime := Clock;
    interval := newTime - oldTime;
end;
```

ili:

```
declare
    use Ada.Real_Time;
    start, finish : Time;
    interval : Time_Span := To_Time_Span(1.7);
begin
    start := Clock;
    -- sequence of statements
    finish := Clock;
    if finish-start > interval then
        raise Time_Error; -- a user-defined exception
    end if;
end;
```

- Sličan pristup je moguć i u jeziku Java:

```
{
    AbsoluteTime oldTime, newTime;
    RelativeTime interval;
    Clock clock = Clock.getRealtimeClock();

    oldTime = clock.getTime();
    // other computations
    newTime = clock.getTime();
}
```

```
interval = newTime.subtract(oldTime);
}
```

### ***Merenje proteklog vremena u školskom Jezgru***

- Merenje proteklog vremena u školskom Jezgru podržano je apstrakcijom `Timer`. Ova apstrakcija predstavlja vremenski brojač kome se može zadati početna vrednost i koji odbrojava po otkucajima sata realnog vremena. Brojač se može zaustaviti (operacija `stop()`) pri čemu vraća proteklo vreme:

```
typedef Time ...;
const Time maxTimeInterval = ...;

class Timer {
public:

    Timer ();

    void start    (Time period=maxTimeInterval);
    Time stop    ();
    void restart (Time=0);

    Time elapsed ();
    Time remained();

};
```

- Funkcija `restart()` ponovo pokreće brojač za novozadatiim vremenom, ili sa prethodno zadatim vremenom, ako se novo vreme ne zada. Funkcije `elapsed()` i `remained()` vraćaju proteklo, odnosno preostalo vreme.
- Ovakav pristup omogućava kreiranje proizvoljno mnogo objekata klase `Timer` koji će nazavisno i uporedo moći da mere svoje intervale, npr. na sledeći način:

```
Timer* timer = new Timer();

timer->start();
// Other computations
timer->stop();
Time t = timer->elapsed();
delete timer;
```

## **Vremenske kontrole**

- Jedan od najčešćih vremenskih zahteva u RT programima jeste vremenska ograničenost čekanja na neki događaj. Na primer, neki proces treba periodično, svakih 10 sekundi, da očitava vrednost sa senzora za temperaturu, pri čemu se nemogućnost očitavanja vrednosti u roku od jedne sekunde smatra neregularnom situacijom (otkazom). Ili, pouzdano slanje neke poruke preko mreže zahteva čekanje na povratnu potvrdu o prijemu poruke; ukoliko ta potvrda ne stigne u određenom roku, smatra se da prenos nije uspeo.
- U opštem slučaju, *vremenska kontrola* (engl. *timeout*) je ograničenje vremena za koje je neki proces spreman da čeka na neki događaj ili uspostavljanje komunikacije.
- Slično ograničenje može da postoji i u pogledu ograničenosti trajanja izvršavanja nekog dela koda. Ukoliko to izvršavanje traje duže od predviđenog, može se smatrati da je nastala neregularna situacija, ili to izvršavanje prosto treba prekinuti.

- Zbog stalne mogućnosti otkaza, u RT sistemima je neophodno da sinhronizacioni i komunikacioni konstrukti podržavaju vremenske kontrole, tj. čekanje (suspenciju, blokiranje) ograničenog trajanja.

### ***Deljene promenljive i vremenske kontrole***

- Kao što je ranije izneseno, komunikacija i sinhronizacija pomoću deljene promenljive uključuje:
  - međusobno isključenje
  - uslovnu sinhronizaciju.
- Bez obzira kakav se konstrukt koristi, međusobno isključenje podrazumeva potencijalno blokiranje procesa koji želi da uđe u zauzetu kritičnu sekciju. Vreme tog čekanja u opštem slučaju zavisi od i ograničeno je vremenom potrebnim da se kod kritične sekcije izvrši od strane drugih procesa. Zbog toga nije uvek potrebno obezbediti vremensku kontrolu pridruženu čekanju na ulaz u kritičnu sekciju. Međutim, to je sasvim moguće, a ponekad i primereno, npr. kod nekih tehnika izbegavanja mrtvog blokiranja.
- Sa druge strane, uslovna sinhronizacija može da blokira proces na neodređeno vreme. Na primer, proizvođač koji čeka na slobodno mesto u ograničenom baferu može dugo da ostane suspendovan ukoliko je proces potrošača otkazao ili nije u stanju da preuzme element iz bafera duže vreme. Zbog toga je u ovakvim slučajevima potrebno obezbediti vremenski ograničeno čekanje.
- Ovakve vremenske kontrole treba da budu moguće za sve prikazane koncepte uslovne sinhronizacije: semafore, uslovne kritične regione, uslovne promenljive u monitorima i ulaze u zaštićene objekte.
- Na primer, POSIX podržava operaciju `wait()` na semaforu uz zadavanje vremenske kontrole:

```
if (sem_timedwait(&sem, &timeout) < 0) {  
    if (errno == ETIMEDOUT) {  
        /* timeout occurred */  
    }  
    else { /* some other error */ }  
} else {  
    /* semaphore locked */  
};
```

- Jezik Ada tretira pozive ulaza u zaštićene objekte na isti način kao i randevu, pa je njima moguće pridružiti vremenske kontrole kao što će to biti opisano u nastavku.
- U jeziku Java, poziv operacije `wait()` može da bude sa zadatom vremenskom kontrolom, i to sa milisekundnom ili nanosekundnom granularnošću.

### ***Komunikacija porukama i vremenske kontrole***

- Kod sinhronne komunikacije, pošiljalac se potencijalno blokira dok poruka ne bude primljena, pa je ovo čekanje potrebno ograničiti vremenskom kontrolom.
- I kod sinhronne i kod asinhronne komunikacije, primalac se blokira dok ne dobije poruku, pa je i ovo čekanje potrebno vremenski kontrolisati.
- U jeziku Ada čekanje na prijem poruke, tj. uspostavljanje randevua moguće je vremenski kontrolisati pomoću naredbe `delay` u jednoj grani nedeterminističke `select` naredbe. Na primer, sledeći kod prikazuje primer kontrolera koji prima pozive drugih procesa, pri čemu se odsustvo poziva u roku od 10 sekundi posebno tretira:

```

task Controller is
  entry call(T : Temperature);
end Controller;

task body Controller is
  -- declarations
begin
  loop
    select
      accept call(t : Temperature) do
        newTemp := t;
      end Call;
    or
      delay 10.0;
      -- action for timeout
    end select;
    -- other actions
  end loop;
end Controller;

```

- Moguće je specifikovati i apsolutno vreme ograničenja prijema poruke:

```

task TicketAgent is
  entry registration(...);
end TicketAgent;

task body TicketAgent is
  -- declarations
  shopOpen : Boolean := True;
begin
  while shopOpen loop
    select
      accept registration(...) do
        -- log details
      end registration;
    or
      delay until closingTime;
      shopOpen := False;
    end select;
    -- process registrations
  end loop;
end TicketAgent;

```

- U jeziku Ada moguće je vremenski kontrolisati uspostavu komunikacije i na strani pozivaoca, kako za pozive ulaza u procese (randevu), tako i za pozive ulaza u zaštićene objekte. Na primer:

```

loop
  -- get new temperature T
  select
    Controller.call(T);
  or
    delay 0.5;
    null; -- other operations are possible here
  end select;
end loop;

```

- Ovo je poseban oblik naredbe `select`, koji ne može imati više od jedne alternative poziva. Vremenska kontrola predstavlja rok za uspostavljanje komunikacije, a ne za njen završetak.
- U slučaju da pozivalac želi da uspostavi komunikaciju samo ukoliko je pozvani odmah spreman da je prihvati, onda je poziv sledeći:

```
select
    Controller.call(T);
else
    -- other operations are possible here
end select;
```

- Vremenske kontrole mogu da budu pridružene i aktivnostima koje treba prekinuti ukoliko traju duže od zadatog vremena. Oblik je tada:

```
select
    delay 0.1;
then abort
    -- action
end select;
```

- Sledeći primer pokazuje kako se može vršiti neko izračunavanje sa postepenim povećanjem tačnosti rezultata, sve dok ograničeno vreme ne istekne. Prikazani proces ima obavezni deo koji izračunava neki početni, približni rezultat, a onda ulazi u iterativno popravljjanje rezultata sve dok vremensko ograničenje ne istekne:

```
declare
    preciseResult : Boolean;
begin
    completionTime := ...;
    -- Compulsory part
    results.write(...); -- Call a procedure in an external protected object
    select
        delay until completionTime;
        preciseResult := False;
    then abort
        while canBeImproved loop
            -- Improve result
            results.write(...);
        end loop;
        preciseResult := True;
    end select;
end;
```

### *Vremenske kontrole u školskom Jezgru*

- Isti apstrakcija `Timer` iz školskog Jezgra može se koristiti i za vremenske kontrole. Apstrakcija `Timer` predstavlja vremenski brojač kome se zadaje početna vrednost i koji odbrojava po otkucajima sata realnog vremena.
- Ako je potrebno vršiti vremensku kontrolu, onda korisnička klasa treba da implementira interfejs (tj. bude izvedena iz jednostavne apstraktne klase) `Timeable` koja poseduje čistu virtuelnu funkciju `timeout()`. Ovu funkciju korisnik može da redefiniše, a poziva je `Timer` kada zadato vreme istekne. Opisani interfejsi izgledaju ovako:

```
class Timeable {
public:
    virtual void timeout () = 0;
};

class Timer {
public:

    Timer (Time period=maxTimeInterval, Timeable*=0);
    ~Timer ();

    void start (Time period=maxTimeInterval);
```



```

Time stop    ();
void restart (Time=0);

Time elapsed () const;
Time remained() const;

};

```

- Drugi argument konstruktora predstavlja pokazivač na objekat kome treba poslati poruku `timeout()` kada zadato vreme istekne. Ako se ovaj pokazivač ne zada, brojač neće poslati ovu poruku pri isteku vremena.

## Kašnjenje procesa

- Osim opisanih usluga, potrebno je ponekad i da proces zahteva svoje "uspavljivanje", odnosno kašnjenje nastavka svog izvršavanja za određeno vreme, bilo relativno u odnosu na tekući trenutak, bilo do određenog apsolutnog trenutka.
- U jeziku Ada, konstrukt `delay` dozvoljava specifikaciju ovakvog kašnjenja:

```
delay 10.0;  -- Relative delay for 10 seconds
```

ili:

```

start := Clock;
first_action;
delay until start+10.0;  -- Absolute delay
second_action;

```

- Važno je naglasiti da konstrukt `delay` garantuje jedino da proces neće nastaviti izvršavanje *pre* zadatog vremena. Koliko će stvarno kašnjenje biti, naravno zavisi od ostalih procesa koji konkurišu za procesor. Pored toga, na preciznost kašnjenja utiče i granularnost časovnika realnog vremena. Sve u svemu, faktori koji utiču na stvarno kašnjenje jesu:
  - vreme kašnjenja definisano programom, tj. `delay` konstruktom;
  - razlika granularnosti između časovnika i `delay` specifikacije;
  - vreme za koje su prekidi zabranjeni;
  - vreme za koje je proces spreman, ali se ne izvršava, jer se izvršavaju drugi procesi.
- Nepreciznost buđenja procesa, koja zavisi od navedenih faktora, naziva se *lokalno plivanje* (engl. *local drift*) i ne može se eliminisati. Međutim, kod implementacije periodičnih procesa, važno je eliminisati akumulaciju ovih grešaka, tj. *kumulativno plivanje* (engl. *cumulative drift*), pri kome se lokalne greške mogu superponirati.
- Na primer, sledeća implementacija periodičnog procesa pati i od lokalnog i od kumulativnog plivanja:

```
task T;
```

```

task body T is
begin
  loop
    Action;
    delay 5.0;
  end loop;
end T;

```

- Bolje rešenje, koje eliminiše kumulativno plivanje (ali i dalje poseduje lokalno plivanje koje se ne može eliminisati) jeste sledeće:

```
task body T is
  interval : constant Duration := 5.0;
  nextTime : Time;
begin
  nextTime := Clock + interval;
  loop
    Action;
    delay until nextTime;
    nextTime := nextTime + interval;
  end loop;
end T;
```

## Specifikacija vremenskih zahteva

- RT sistemima se postavljaju veoma raznovrsni vremenski zahtevi. Na žalost, postojeća inženjerska praksa uglavnom primenjuje *ad-hoc* metode za ispunjavanje tih zahteva. To znači da se sistem najpre konstruiše tako da bude logički ispravan, a zatim se koristi i testira na vremenske zahteve. Ukoliko ti vremenski zahtevi nisu zadovoljeni, vrše se fina podešavanja i ispravke sistema. Zbog toga konačni sistem može da postane zamršen i nepregledan.
- Zato su potrebne strožije metode za specifikaciju i zadovoljavanje vremenskih ograničenja. Istraživanja u tom domenu uglavnom su išla u dva pravca:
  - Upotreba formalnih jezika sa preciznom semantikom koja uključuje i vremenske karakteristike koje se mogu analizirati.
  - Utvrđivanje performansi realizovanog sistema i dokazivanje izvodljivosti, odnosno rasporedivosti zahtevanog opterećenja na postojeće resurse (procesore i sl.).
- Verifikacija RT sistema tako uključuje dve faze:
  - Verifikacija specifikacije zahteva: pod pretpostavkom da je na raspolaganju proizvoljno brz procesor, treba proveriti da li su vremenski zahtevi koherentni i konzistentni, tj. da li je uopšte moguće ispuniti ih. Ovaj pristup podrazumeva upotrebu formalnih metoda dokazivanja vremenske korektnosti uz pomoć odgovarajućih alata.
  - Verifikacija implementacije: korišćenjem konačnih resursa (potencijalno nepouzdatih), da li se vremenski zahtevi mogu ispuniti?
- Jedan koncept koji olakšava specifikaciju različitih vremenskih zahteva u RT aplikacijama jeste koncept *temporalnih opsega* (engl. *temporal scope*). Temporalni opseg je skup naredbi u programu, najčešće neki konstrukt u postojećem sekvencijalnom ili konkurentnom jeziku (npr. blok ili proces), za koji se mogu specifikovati sledeća vremenska ograničenja:
  - *rok* (engl. *deadline*): trenutak do kog se izvršavanje opsega mora završiti;
  - *minimalno kašnjenje* (engl. *minimum delay*): minimalno vreme koje mora proteći pre nego što započne izvršavanje opsega;
  - *maksimalno kašnjenje* (engl. *maximum delay*): maksimalno vreme koje može proteći pre nego što započne izvršavanje opsega;
  - *maksimalno vreme izvršavanja* (engl. *maximum execution time*): maksimalno procesorsko vreme koje opseg sme uzeti tokom svog izvršavanja;
  - *maksimalno proteklo vreme* (engl. *maximum elapse time*): maksimalno vreme koje može proteći od trenutka započinjanja do trenutka završetka izvršavanja opsega.
- Sam temporalni opseg može biti definisan kao:

- *periodični* (engl. *periodic*); takvi su tipično poslovi koji periodično uzimaju odbirke ili izvršavaju kontrolne petlje i imaju svoje vremenske rokove koji moraju biti zadovoljeni;
- *aperiodični* (engl. *aperiodic*); takvi su obično poslovi koji nastaju kao posledica asinhronih spoljašnjih *događaja* (engl. *event*) i koji obično imaju definisano *vreme odziva* (engl. *response time*).
- U opštem slučaju, može se smatrati da aperiodični događaji stižu slučajno, prema nekoj slučajnoj raspodeli. Takva raspodela, teorijski, dozvoljava nalete događaja u određenom periodu, i to u proizvoljnoj gustini. Drugim rečima, verovatnoća da dva događaja stignu u proizvoljno malom vremenskom razmaku je uvek veća od nule. Međutim, ovakav teorijski slučaj najčešće ne odgovara praksi, a osim toga i ne dozvoljava analizu izvodljivosti sistema u najgorem slučaju.
- Zbog toga se uvek uvodi pretpostavka o *minimalnom mogućem vremenskom razmaku* između asinhronih događaja (i njima pridruženih poslova, odnosno temporalnih opsega). Takvi aperiodični opsezi i poslovi, koji imaju definisan minimalni (najgori) vremenski razmak, nazivaju se *sporadičnim* (engl. *sporadic*).
- U praksi, specifikacija temporalnih opsega, koji se najčešće vezuju za procese, svodi se na specifikaciju sledećih vremenskih ograničenja:
  - pokretanje periodičnih procesa sa odgovarajućom frekvencijom;
  - završavanje svih procesa do njihovog roka.
- Tako se problem zadovoljenja vremenskih zahteva svodi na problem raspoređivanja procesa tako da zadovolje svoje rokove (engl. *deadline scheduling*):
  - ukoliko su rokovi strogi i ne smeju se propustiti, sistem se naziva *hard RT sistemom*; najčešće se pod tim podrazumeva da sistem mora da reaguje odgovarajućom akcijom oporavka od otkaza ukoliko je neki rok prekoračen;
  - ukoliko sistem toleriše povremeno prekoračenje rokova, naziva se *soft RT sistemom*.

### ***Periodični procesi***

- Opšta šema periodičnog procesa izgleda ovako:

```
process P;
...
begin
  loop
    IDLE
    start of temporal scope
    ...
    end of temporal scope
  end;
end;
```

- Vremenska ograničenja definišu minimalno i maksimalno vreme za `IDLE` deo, kao i vremenski rok (engl. *deadline*) završetka temporalnog opsega.
- Maksimalno vreme završetka `IDLE` dela, tj. maksimalno vreme do početka temporalnog opsega ima smisla kod procesa koji uzimaju odbirke sa nekog spoljašnjeg senzora i posle toga proizvode neki izlaz ili prosto baferišu pročitane vrednosti. U tom slučaju je bitno ograničiti maksimalno vreme započinjanja temporalnog opsega, odnosno završetka `IDLE` dela, kako bi očitana vrednost imala smisla. To očitavanje se dešava na početku temporalnog opsega, dok se ostatak (obrada ili baferisanje odbirka) može odraditi u nastavku, sa kasnijim rokom završetka.

- Vremenski rok završetka temporalnog opsega (engl. *deadline*) se može izraziti kao apsolutno vreme, vreme izvršavanja od trenutka započinjanja temporalnog opsega, ili vreme proteklo od započinjanja temporalnog opsega.
- Vremenski rok ima ulogu da obezbedi da proces može da započne svoju narednu iteraciju petlje na vreme, u narednoj periodu, ili da obezbedi pravovremenost nekog izlaznog signala koji se generiše na kraju temporalnog opsega.
- U jeziku Ada, periodični proces može da se konstruiše na sledeći način:

```
task body Periodic_T is
  Release_Interval : Duration := ...; -- or
  Release_Interval : Time_Span := Milliseconds(...);
begin
  -- read clock and calculate the next
  -- release time (Next_Release)
  loop
    -- sample data (for example) or
    -- calculate and send a control signal
    delay until Next_Release;
    Next_Release := Next_Release + Release_Interval;
  end loop;
end Periodic_T;
```

- U jeziku RT Java, periodični proces može da se konstruiše na sledeći način:

```
public class Periodic extends RealtimeThread {
  public Periodic(PriorityParameters pp, PeriodicParameters p) { ... };

  public void run() {
    while(true) {
      // Code to be run each period
      ...
      waitForNextPeriod();
    }
  }
}

PeriodicParameters p = new PeriodicParameters(
  new AbsoluteTime(...),      // Start time
  new RelativeTime(10,0),     // Period in milliseconds, nanoseconds
  new RelativeTime(1,0),      // Maximum execution time in ms,ns
  new RelativeTime(5,0),      // Deadline in ms,ns
  null,                       // Overrun handler
  null                        // Miss handler
);
PriorityParameters pp = new PriorityParameters(...);
Periodic myThread = new Periodic(pp,p); // Create thread
myThread.start(); // and release it
```

- Periodični procesi u školskom Jezgru mogu se lako konstruisati pomoću koncepta niti (klasa Thread) i vremenskog brojača (klasa Timer). Videti Vežbe.

### *Sporadični procesi*

- Sporadični proces se pokreće na asinhroni događaj, pa mu je opšti oblik sledeći:

```
process Sporadic;
...
begin
  loop
    wait for event/interrupt
```

```

    start of temporal scope
    ...
    end of temporal scope
end;
end;

```

- Minimalno vreme razmaka između dve pojave sporadičnog događaja može se smatrati periodom odgovarajućeg sporadičnog procesa u najgorem slučaju, pa se ta veličina može koristiti u analizi rasporedivosti.
- U jeziku Ada, sporadični proces tipično koristi zaštićeni objekat koji je vezan za spoljašnji prekid i koji pokreće proces:

```

protected Sporadic_Controller is
  procedure Interrupt; -- mapped onto interrupt
  entry Wait_For_Next_Interrupt;
private
  Call_Outstanding : boolean := false;
end Sporadic_Controller;

protected Sporadic_Controller is
  procedure Interrupt is
    begin
      Call_Outstanding := True;
    end Interrupt;
  entry Wait_For_Next_Interrupt when Call_Outstanding is
    begin
      Call_Outstanding := False;
    end Wait_For_Next_Interrupt;
end Sporadic_Controller;

task body Sporadic_T is
begin
  loop
    Sporadic_Controller.Wait_For_Next_Interrupt;
    -- action
  end loop;
end Sporadic_T;

```

- U jeziku RT Java, sporadični procesi se konstruišu slično kao i periodični, osim što proces čeka na sledeći događaj umesto na sledeći period. Međutim, događaji koji pokreću sporadične niti u jeziku RT Java još uvek nisu dobro definisani.
- U školskom Jezgru sporadični procesi pobuđuju se *prekidom* (engl. *interrupt*). Prekidi predstavljaju važan elemenat RT sistema.
- Klasa `InterruptHandler` predstavlja generalizaciju prekida. Njen interfejs izgleda ovako:

```

typedef unsigned int IntNo; // Interrupt Number

class InterruptHandler : public Thread {
protected:

  InterruptHandler (IntNo num, void (*intHandler)());

  virtual int handle () { return 0; }
  void interruptHandler ();

};

```

- Korisnik iz ove apstraktne klase treba da izvede sopstvenu klasu za svaku vrstu prekida koji se koristi. Korisnička klasa treba da bude *Singleton*, a prekidna rutina definiše se kao statička funkcija te klase (jer ne može imati argumente). Korisnička prekidna rutina treba

samo da pozove funkciju jedinog objekta `InterruptHandler::interruptHandler()`. Dalje, korisnik treba da redefiniše virtuelnu funkciju `handle()`. Ovu funkciju će pozvati sporadični proces kada se dogodi prekid, pa u njoj korisnik može da navede proizvoljan kod. Treba primetiti da se taj kod izvršava svaki put kada se dogodi prekid, pa on ne treba da sadrži petlju, niti čekanje na prekid.

- Osim navedene uloge, klasa `InterruptHandler` obezbeđuje i implicitnu inicijalizaciju interapt vektor tabele: konstruktor ove klase zahteva broj prekida i pokazivač na prekidnu rutinu. Na ovaj način ne može da se dogodi da programer zaboravi inicijalizaciju, a ta inicijalizacija je lokalizovana, pa su zavisnosti od platforme svedene na minimum.
- Primer upotrebe:

```
// Timer interrupt entry:  
const int TimerIntNo = 0;
```

```
class TimerInterrupt : public InterruptHandler {  
protected:  
  
    TimerInterrupt () : InterruptHandler(TimerIntNo,timerInterrupt) {}  
  
    static void timerInterrupt () { instance->interruptHandler(); }  
  
    virtual int handle () {  
        ... // User-defined code for one release of the sporadic process  
    }  
  
private:  
  
    static TimerInterrupt* instance;  
  
};
```

```
TimerInterrupt* TimerInterrupt::instance = new TimerInterrupt;
```

## Kontrola zadovoljenja vremenskih zahteva

- Uključivanje vremenskih ograničenja u RT programe podrazumeva i mogućnost njihovog narušavanja, što se može smatrati otkazom u RT programu.
- *Soft* RT sistemi ponekad treba da budu svesni prekoračenja rokova, iako to mogu da smatraju svojim normalnim tokom. Međutim, *hard* RT sistemi moraju da preduzimaju procedure oporavka od otkaza ukoliko dođe do prekoračenja nekog roka.
- Iako analiza rasporedivosti projektovanog sistema može da "dokaže" da rokovi neće biti prekoračeni, ipak je potrebno u programe ugraditi ovakve procedure oporavka zato što:
  - procena vremena izvršavanja procesa u najgorem slučaju (engl. *worst-case execution time*, WCET) nije bila precizna;
  - pretpostavke koje su uzete pri analizi rasporedivosti nisu tačne;
  - analiza rasporedivosti nije tačno izvedena zbog greške;
  - algoritam raspoređivanja ne može da podnese dato opterećenje, iako je ono teorijski rasporedivo;
  - sistem funkcioniše izvan opsega za koji je projektovan.
- Zbog toga je u RT sistemima potrebno detektovati sledeće otkaze u pogledu vremenskih ograničenja:
  - prekoračenje roka (engl. *overrun of deadline*, *deadline miss*)

- prekoračenje vremena izvršavanja u najgorem slučaju (engl. *overrun of WCET*)
- pojavu sporadičnih događaja frekventnije nego što je predviđeno
- istek vremenskih kontrola (engl. *timeout*).
- Naravno, pojava poslednja tri navedena otkaza ne znači obavezno da će neki rok biti prekoračen. Na primer, prekoračenje WCET jednog procesa može biti konpenzovano time što će se neki drugi sporadični proces pojaviti ređe nego što je predviđeno u najgorem slučaju. Zbog toga procedure izolacije i procene štete treba da odluče koju akciju treba preduzeti u slučaju ovakvih otkaza.
- U slučaju detekcije nekog od vremenskih prekoračenja (tj. otkaza), mogu se preduzimati ranije opisane tehnike oporavka od otkaza (BER ili FER).

### Detekcija prekoračenja roka

- Kao što je već pokazano, u jeziku Ada, detekcija prekoračenja roka periodičnog procesa može se obaviti pomoću `select-then-abort` konstrukta (slično važi i za sporadične procese):

```
task body PeriodicTask is
  nextRelease : Time;
  nextDeadline : Time;
  releaseInterval : constant Time_Span := Milliseconds(...);
  deadline : constant Time_Span := Milliseconds(...);
begin
  -- Read clock and calculate nextRelease and nextDeadline
  loop
    select
      delay until nextDeadline;
      -- Deadline overrun detected here; perform recovery
    then abort
      -- Code of application
    end select;
    delay until nextRelease;
    nextRelease := nextRelease + releaseInterval;
    nextDeadline := nextRelease + deadline;
  end loop;
end PeriodicTask;
```

- Jedan od potencijalnih problema kod ovog pristupa jeste što se proces koji je prekoračio svoj rok prekida, a nastavlja se izvršavanje koda za oporavak. Drugačiji pristup bi bio, recimo, da proces nastavi svoje izvršavanje pod drugih uslovima, npr. sa višim prioritetom.
- U jeziku RT Java, izvršno okruženje (Java virtuelna mašina) će asinhrono signalizirati prekoračenje roka procesa i biće pozvan kod definisan u klasi izvedenoj iz klase `AsyncEventHandler`. Objekat ovakve klase zadaje se kao parametar `overrunHandler` konstruktora `PeriodicParameters` (vidi gore). Sporadični procesi u jeziku RT Java nemaju eksplicitni vremenski rok, pa se smatraju *soft* procesima.

### Detekcija prekoračenja WCET

- Prekoračenje roka može biti uzrokovano greškom u sasvim drugom delu programa, tj. u drugom procesu. Na primer, ako neki proces visokog prioriteta prekorači svoje vreme izvršavanja u najgorem slučaju (WCET), možda on uopšte neće prekoračiti svoj rok, ali će uzrokovati da drugi procesi nižeg prioriteta prekorače svoje rokove. Iako se opisani mehanizmi mogu koristiti za obradu ovakvih otkaza u tim procesima, često je bolje i lakše otkaz obraditi tamo gde je greška i nastala, tj. u procesu koji je prekoračio svoj WCET.

- Ukoliko izvršavanje jednog procesa nije prekidano, onda se detekcija prekoračenja WCET može obaviti na isti način kao i detekcija prekoračenja roka. Međutim, to je veoma retko slučaj u konkurentnim programima.
- Jezik Java dozvoljava kontrolu prekoračenja WCET za objekte tipa `RealTimeThread` na isti način kao i za prekoračenje roka. Izvršno okruženje (Java virtuelna mašina) će asinhrono signalizirati prekoračenje WCET procesa i biće pozvan kod definisan u klasi izvedenoj iz klase `AsyncEventHandler`. Objekat ovakve klase zadaje se kao parametar `costHandler` konstruktora `PeriodicParameters` (vidi gore).

### ***Prekoračenje učestanosti sporadičnih događaja***

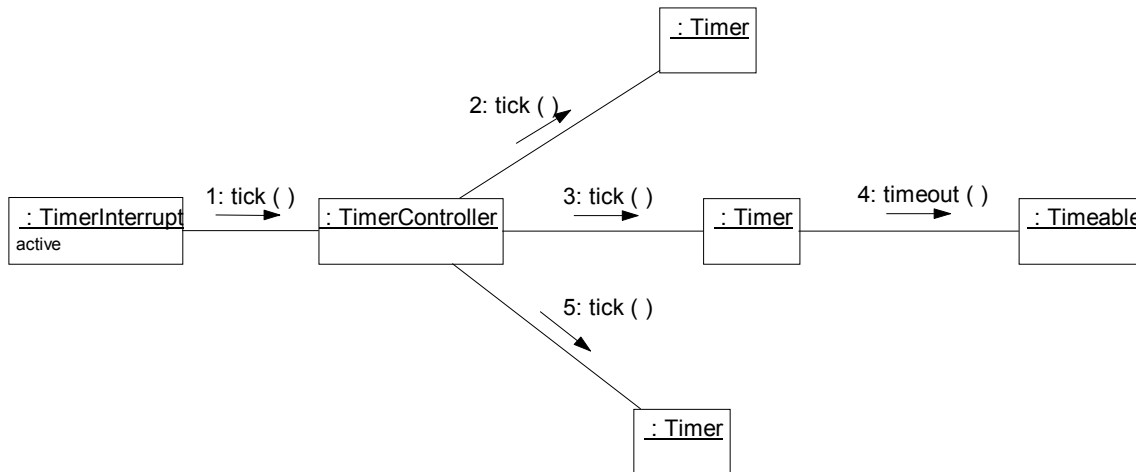
- Pojava sporadičnih događaja češće nego što je predviđeno može ozbiljno ugroziti poštovanje rokova u *hard* RT sistemima. Zbog toga je potrebno ovu pojavu ili sprečiti, ili na nju reagovati.
- Jedan pristup sprečavanju prečestih pojava sporadičnih događaja jeste kontrola učestanosti hardverskih prekida uticajem na same registre hardverskih uređaja koji generišu te prekide. Drugi pristup vezan je za raspoređivanje i ovde neće biti detaljnije prikazivan.

## **Implementacija u školskom Jezgru**

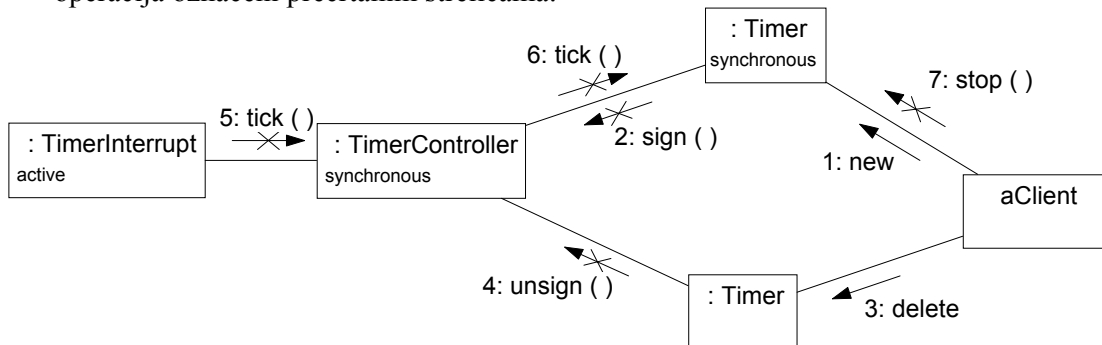
### ***Mehanizam merenja vremena***

- Mehanizam merenja vremena u školskom Jezgru može se jednostavno realizovati na sledeći način. Hardver mora da obezbedi (što tipično postoji u svakom računaru) brojač (sat) realnog vremena koji periodično generiše prekid sa zadatim brojem. Ovaj prekid kontrolisaće objekat klase `TimerInterrupt`. Ovaj objekat pri svakom otkucaju sata realnog vremena, odnosno po pozivu prekidne rutine, prosleđuje poruku `tick()` jednom centralizovanom *Singleton* objektu tipa `TimerController`, koji sadrži spisak svih kreiranih objekata tipa `Timer` u sistemu. Ovaj kontroler će proslediti poruku `tick()` svim brojačima.
- Svaki brojač tipa `Timer` se prilikom kreiranja prijavljuje u spisak kontrolera (operacija `sign()`), što se obezbeđuje unutar konstruktora klase `Timer`. Analogno, prilikom ukidanja, brojač se odjavljuje (operacija `unsign()`), što obezbeđuje destruktor klase `Timer`.
- Vremenski brojač poseduje atribut `isRunning` koji pokazuje da li je brojač pokrenut (odbrojava) ili ne. Kada primi poruku `tick()`, brojač će odbrojati samo ako je ovaj indikator jednak 1, inače jednostavno vraća kontrolu pozivaocu. Ako je prilikom odbrojavanja brojač stigao do 0, šalje se poruka `timeout()` povezanom objektu tipa `Timeable`.
- Opisani mehanizam prikazan je na sledećem dijagramu interakcije:

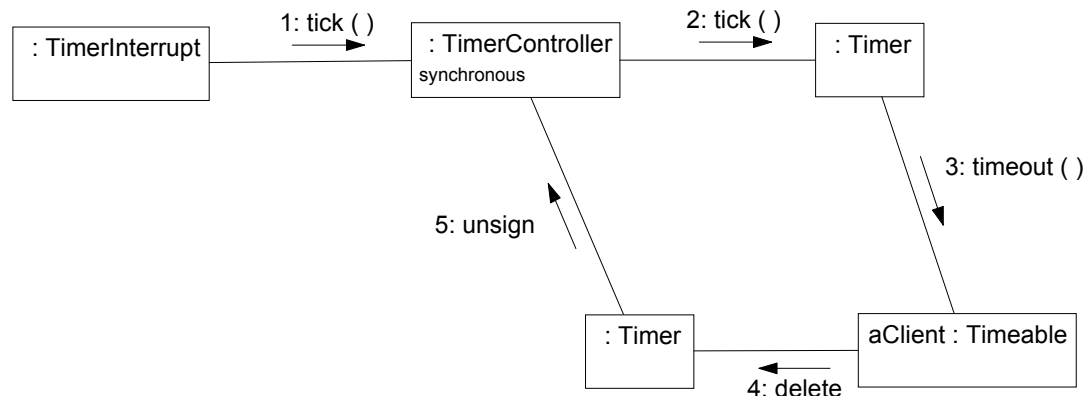




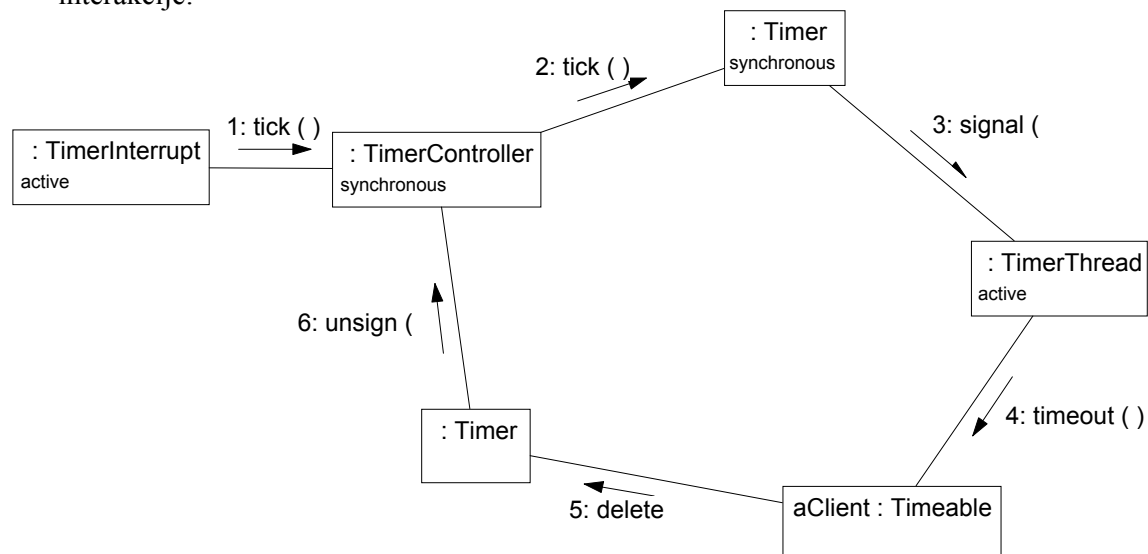
- Kako do objekta `TimerController` stižu konkurentne poruke sa dve strane, od objekta `InterruptHandler` poruka `tick()` i od objekata `Timer` poruke `sign()` i `unsign()`, ovaj objekat mora da bude sinhronizovan (monitor). Slično važi i za objekte klase `Timer`. Ovo je prikazano na sledećem dijagramu scenarija, pri čemu su blokirajući pozivi isključivih operacija označeni precrtanim strelicama:

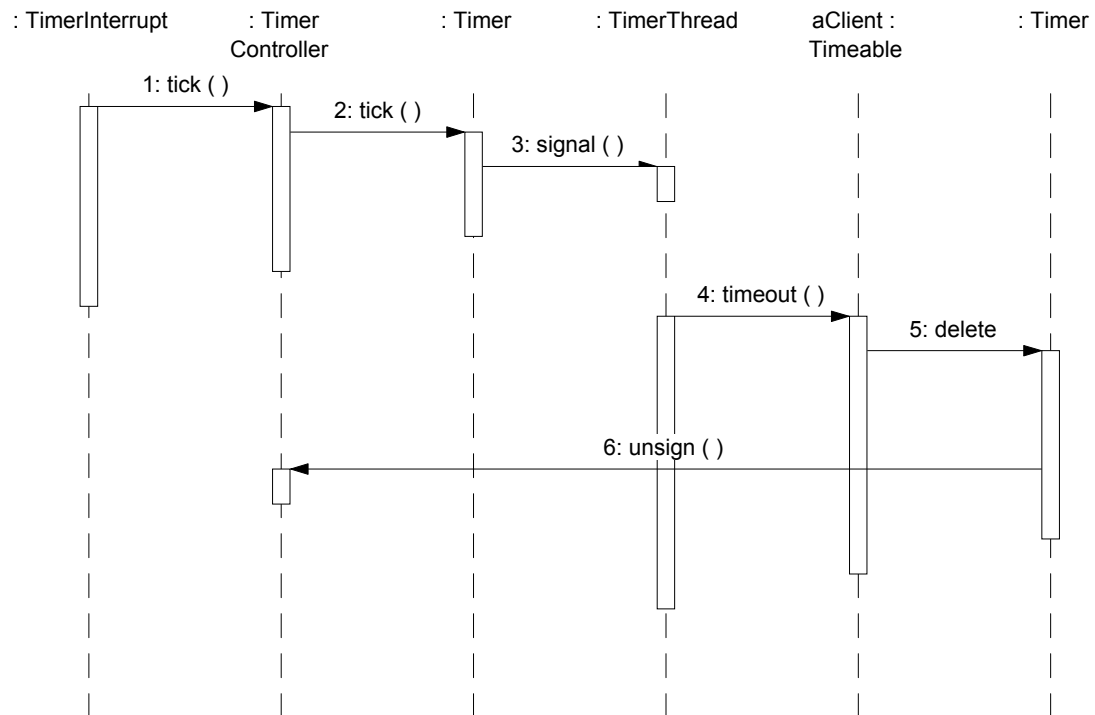


- Međutim, ovakav mehanizam dovodi do sledećeg problema: može se dogoditi da se unutar istog toka kontrole (niti) koji potiče od objekta `TimerInterrupt`, pozove `TimerController::tick()`, čime se ovaj objekat zaključava za nove pozive svojih operacija, zatim odatle pozove `Timer::tick()`, brojač dobrojava do nule, poziva se `Timeable::timeot()`, a odatle neka korisnička funkcija. Unutar ove korisničke funkcije može se, u opštem slučaju, kreirati ili brisati neki `Timer`, sve u kontekstu iste niti, čime se dolazi do poziva operacija objekta `TimerController`, koji je ostao zaključan. Na taj način dolazi do kružnog blokiranja (engl. *deadlock*) i to jedne niti same sa sobom.
- Čak i ako se ovaj problem zanemari, ostaje problem eventualno predugog zadržavanja unutar konteksta niti koja ažurira brojače, jer se ne može kontrolisati koliko traje izvršavanje korisničke operacije `timeout()`. Time se neodređeno zadržava mehanizam ažuriranja brojača, pa se gubi smisao samog merenja vremena. Opisani problemi prikazani su na sledećem dijagramu scenarija:

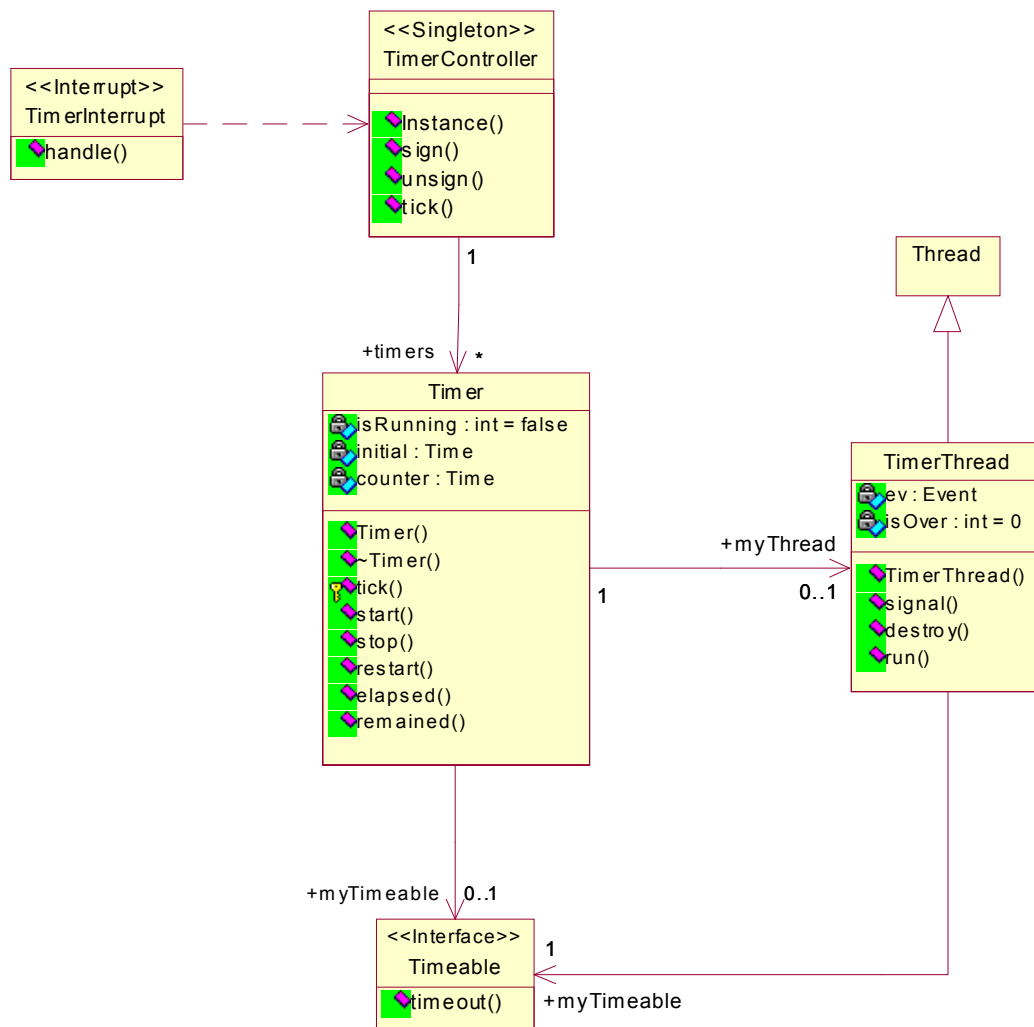


- Problem se može rešiti na sledeći način: potrebno je na nekom mestu prekinuti kontrolu toka i razdvojiti kontekste uvođenjem niti kojoj će biti signaliziran događaj. Ovde je to učinjeno tako što objekat `Timer`, ukoliko poseduje pridružen objekat `Timeable`, poseduje i jedan aktivni objekat `TimerThread` koji poseduje nezavisan tok kontrole u kome se obavlja poziv operacije `timeout()`. Objekat `Timer` će, kada vreme istekne, samo signalizirati događaj pridružen objektu `TimerThread` i vratiti kontrolu objektu `TimerController`. `TimerThread` će, kada primi signal, obaviti poziv operacije `timeout()`. Na ovaj način se navedeni problemi eliminišu, jer se sada korisnička funkcija izvršava u kontekstu sopstvene niti. Mehanizam je prikazan na sledećim dijagramima interakcije:





- Dijagram opisanih klasa izgleda ovako:



- Kompletan kod za ovaj podsistem dat je u nastavku.

```
// Project: Real-Time Programming
// Subject: Multithreaded Kernel
// Module: Timer
// File: timer.h
// Created: November 1996
// Revised: August 2003
// Author: Dragan Milicev
// Contents: Timers
//      Types:      Time
//      Classes:    Timer
//      Interfaces: Timeable
```

```
#ifndef _TIMER_
#define _TIMER_
```

```
#include "collect.h"
#include "semaphor.h"
```

```
////////////////////////////////////
// type Time
// constant maxTimeInterval
```

```

////////////////////////////////////
typedef unsigned long int Time;
const Time maxTimeInterval = ~0;

////////////////////////////////////
// interface Timeable
////////////////////////////////////

class Timeable {
public:
    virtual void timeout () = 0;
};

////////////////////////////////////
// class Timer
////////////////////////////////////

class TimerThread;
class Semaphore;

class Timer : public Object {
public:

    Timer (Time period=maxTimeInterval, Timeable* toNotify=0);
    ~Timer ();

    void start    (Time period=maxTimeInterval) { restart(period); }
    Time stop    ();
    void restart (Time=0);

    Time elapsed () { return initial-counter; }
    Time remained() { return counter; }

protected:

    friend class TimerController;
    CollectionElement* getCEForController () { return &ceForController; }
    void tick ();

private:

    Timeable* myTimeable;
    TimerThread* myThread;

    Time counter;
    Time initial;

    int isRunning;

    Semaphore mutex;
    CollectionElement ceForController;

    RECYCLE_DEC(Timer)
};

#endif

// Project:  Real-Time Programming

```

```
// Subject: Multithreaded Kernel
// Module: Timer
// File: timer.cpp
// Created: November 1996
// Revised: August 2003
// Author: Dragan Milicev
// Contents: Timers
//          Classes:
//              Timer
//              TimerThread
//              TimerController
//              TimerInterrupt

#include "timer.h"
#include "semaphor.h"

////////////////////////////////////
// class TimerThread
////////////////////////////////////

class TimerThread : public Thread {
public:

    TimerThread (Timeable*);

    void signal ();
    void destroy ();

protected:

    virtual void run ();

private:

    Event ev;
    Timeable* myTimeable;
    int isOver;

    RECYCLE_DEC(TimerThread)

};

RECYCLE_DEF(TimerThread);

TimerThread::TimerThread (Timeable* t) : myTimeable(t), isOver(0),
RECYCLE_CON(TimerThread) {}

void TimerThread::signal () {
    ev.signal();
}

void TimerThread::destroy () {
    isOver=1;
    ev.signal();
}

void TimerThread::run () {
    while (1) {
        ev.wait();
```

[illegible]

```
// Timer interrupt entry:
const int TimerIntNo = 0;

class TimerInterrupt : public InterruptHandler {
protected:

    TimerInterrupt () : InterruptHandler(TimerIntNo,timerInterrupt) {}

    static void timerInterrupt () { instance->interruptHandler(); }
    virtual int handle () { TimerController::Instance()->tick(); return 1; }

private:

    static TimerInterrupt* instance;

};

TimerInterrupt* TimerInterrupt::instance = new TimerInterrupt;

////////////////////////////////////
// class Timer
////////////////////////////////////

RECYCLE_DEF(Timer);

Timer::Timer (Time t, Timeable* tmb1) : RECYCLE_CON(Timer),
myTimeable(tmb1), myThread(0),
        counter(t), initial(t), isRunning(0),
        mutex(1), ceForController(this) {
    if (myTimeable!=0) {
        myThread=new TimerThread(myTimeable);
        myThread->start();
    }
    TimerController::Instance()->sign(this);
}

Timer::~~Timer () {
    mutex.wait();
    TimerController::Instance()->unsign(this);
    if (myThread!=0) myThread->destroy();
}

Time Timer::stop () {
    Mutex dummy(&mutex);
    isRunning=0;
    return initial-counter;
}

void Timer::restart (Time t) {
    Mutex dummy(&mutex);
    if (t!=0)
        counter=initial=t;
    else
        counter=initial;
}
```



```
    isRunning=1;
}

void Timer::tick () {
    Mutex dummy(&mutex);
    if (!isRunning) return;
    if (--counter==0) {
        isRunning=0;
        if (myThread!=0) myThread->signal();
    }
}
```

### ***Obrada prekida***

- Posao koji se obavlja kao posledica prekida logički nikako ne pripada niti koja je prekinuta, jer se u opštem slučaju i ne zna koja je nit prekinuta: prekid je za softver signal nekog asinhronog spoljašnjeg događaja. Zato posao koji se obavlja kao posledica prekida treba da ima sopstveni kontekst, tj. da se pridruži sporadičnom procesu, kao što je ranije rečeno.
- Osim toga, ne bi valjalo dopustiti da se u prekidnoj rutini, koja se izvršava u kontekstu niti koja je prekinuta, poziva neka operacija koja može da blokira pozivajuću nit.
- Drugo, značajno je da se u prekidnoj rutini vodi računa kako dolazi do preuzimanja, ako je to potrebno.
- Treće, u svakom slučaju, prekidna rutina treba da završi svoje izvršavanje što je moguće kraće, kako ne bi zadržavala ostale prekide.
- Prema tome, opasno je u prekidnoj rutini pozivati bilo kakve operacije drugih objekata, jer one potencijalno nose opasnost od navedenih problema. Ovaj problem rešava se ako se na suštinu prekida posmatra na sledeći način.
- Prekid zapravo predstavlja obaveštenje (asinhroni signal) softveru da se neki događaj dogodio. Pri tome, signal o tom događaju ne nosi nikakve druge informacije, jer prekidne rutine nemaju argumente. Sve što softver može da sazna o događaju svodi se na softversko čitanje podataka (eventualno nekih registara hardvera). Prema tome, prekid je *asinhroni signal događaja*.
- Navedeni problemi rešavaju se tako što se obezbedi jedan događaj koji će prekidna rutina da signalizira, i jedan proces koji će na taj događaj da čeka. Na ovaj način su konteksti prekinutog procesa (i sa njim i prekidne rutine) i sporadičnog procesa koji se prekidom aktivira potpuno razdvojeni, prekidna rutina je kratka jer samo obavlja signal događaja, a prekidni proces može da obavlja proizvoljne operacije posla koji se vrši kao posledica prekida.
- Ukoliko operativni sistem treba da odmah odgovori na prekid, onda operacija signaliziranja događaja iz prekidne rutine treba da bude sa preuzimanjem (engl. *preemptive*), pri čemu treba voditi računa kako se to preuzimanje vrši na konkretnoj platformi (maskiranje prekida, pamćenje konteksta u prekidnoj rutini i slično).
- Treba primetiti da eventualno slanje poruke unutar prekidne rutine u neki bafer ne dolazi u obzir, jer je bafer tipično složena struktura koja zahteva međusobno isključenje, pa time i potencijalno blokiranje. Događaj, kako je opisano, predstavlja pravi koncept za ovaj problem, jer je njegova operacija *signal* potpuno "bezazlena" (u svakom slučaju neblokirajuća).
- Kod za opisano rešenje dato je u nastavku:

```
typedef unsigned int IntNo; // Interrupt Number
```

```
class InterruptHandler : public Thread {
protected:

    InterruptHandler (IntNo num, void (*intHandler)());

    virtual void run ();

    virtual int handle () { return 0; }
    void interruptHandler ();

private:

    Event ev;

};

void initIVT (IntNo, void (*)() ) {
    // Init IVT entry with the given vector
}

InterruptHandler::InterruptHandler (IntNo num, void (*intHandler)()) {
    // Init IVT entry num by intHandler vector:
    initIVT(num,intHandler);

    // Start the thread:
    start();
}

void InterruptHandler::run () {
    for(;;) {
        ev.wait();
        if (handle()==0) return;
    }
}

void InterruptHandler::interruptHandler () {
    ev.signal();
}
```

## Vežbe

### 9.1

Projektuje se optimizovani podsistem za merenje vremena u nekom Real-Time operativnom sistemu. Podsistem se zasniva na konceptu vremenskog brojača realizovanog klasom `Timer` poput onog u postojećem školskom Jezgru, ali u kome vremenski brojači služe samo za kontrolu isteka zadatog vremenskog intervala (*timeout*) i u kome je implementacija mehanizma praćenja isteka tog intervala drugačija. Vremenski brojači, kao objekti klase `Timer`, uvezani su u jednostruko ulančanu listu, uređenu neopadajuće prema trenutku isteka intervala koje brojači mere. Pri tom, prvi objekat u listi (onaj kome vreme najpre ističe), u svom atributu čuva relativno vreme u odnosu na sadašnji trenutak za koje dolazi do isteka intervala koji meri, a ostali objekti klase `Timer` u tom svom atributu čuvaju samo relativno vreme u odnosu na prethodni brojač u listi (može biti i 0 ukoliko dva susedna brojača ističu u

istom trenutku). Na primer, ukoliko brojači u listi imaju vrednosti ovog atributa redom: 1, 0, 0, 2, 5, onda to znači da prva tri brojača ističu za 1, sledeći za 3, a poslednji za 8 jedinica od sadašnjeg trenutka. Prema tome, prilikom smeštanja novog brojača u listu, on se umeće na odgovarajuću poziciju, a njegov atribut se podešava u odnosu na prethodnike, kao što je opisano. U svakom trenutku otkucaja sata realnog vremena, ažurira se samo prvi brojač u listi. Ukoliko pri tome prvi brojač u listi padne na nulu, vremenski istek se signalizira onim brojačima sa početka liste koji u svom atributu imaju nulu. Realizovati na jeziku C++ klasu `Timer` prema opisanom mehanizmu. Ne koristiti gotovu strukturu podataka za ulančanu listu, već tu strukturu inkorporirati u klasu `Timer`. Realizovati i dve ključne operacije ove klase:

- (a) `Timer::start(Time t)`: pokreće brojač sa zadatim vremenom isteka u odnosu na sadašnji trenutak.
- (b) `Timer::tick()`: statička funkcija klase koju spolja poziva prekidna rutina sata realnog vremena na svaki otkucaj (ne treba realizovati ovu prekidnu rutinu). Ova operacija treba da pokrene operaciju isteka vremena `timeout()` za one brojače koji su istekli.

Prilikom kreiranja, brojač se ne umeće u navedenu listu; tek prilikom startovanja se to radi. Nije potrebno realizovati nijednu drugu pomoćnu operaciju (npr. restart, očitavanje proteklog vremena itd.), već samo opisane operacije koje služe za kontrolu isteka zadatog intervala. Takođe nije potrebno voditi računa o cepanju konteksta prilikom poziva operacije `timeout()` pridruženog objekta koji se krije iza interfejsa `Timeable`.

## 9.2

Potrebno je proširiti školsko Jezgro tako da podržava detekciju prekoračenja roka i prekoračenja WCET. Prodiskutovati:

- (a) Kako bi se u aplikativnom (korisničkom) kodu specifikovala ova ograničenja i koje su modifikacije Jezgra potrebne za prihvatanje ovih specifikacija?
- (b) Kako modifikovati Jezgro da bi se obezbedila detekcija ovih prekoračenja? Kako dojaviti ovaj otkaz korisničkoj aplikaciji?
- (c) Prikazati primerom način korišćenja predloženih koncepata.

## 9.3

Pomoću raspoloživih koncepata školskog Jezgra i korišćenjem apstrakcije `Timer`, potrebno je realizovati opisani konstrukt `delay` koji postoji u mnogim operativnim sistemima i programskim jezicima. Korisnički program može pozvati na bilo kom mestu operaciju `delay(Time)` koja suspenduje tekuću nit (u čijem se kontekstu ova operacija izvršava) na vreme dato argumentom. Posle isteka datog vremena, sistem sam (implicitno) deblokira datu nit. Data nit se može deblokirati i iz druge niti, eksplicitnim pozivom operacije `Thread::wakeUp()` date suspendovane niti. Navesti precizno koje izmene treba učiniti i gde u postojećem Jezgru i dati realizaciju operacija `delay()` i `wakeUp()`.

Prikazati upotrebu ovog koncepta na primeru niti koje kontrolišu deset svetiljki koje se pale i gase naizmenično, svaka sa svojom periodom ugašenog i upaljenog svetla.

## 9.4

Korišćenjem školskog Jezgra, realizovati klasu `TimedSemaphore` koja obezbeđuje logiku standardnog semafora, ali uz vremenski ograničeno čekanje.

### 9.5

Korišćenjem klase `TimedSemaphore` iz prethodnog zadatka, prikazati realizaciju ograničenog bafera u školskom Jezgru, pri čemu su vremenske kontrole pridružene i čekanju na ulaz u kritičnu sekciju i čekanju na uslovnu sinhronizaciju. Ilustrovati upotrebu ovakvog bafera primerom proizvođača i potrošača.

### 9.6

Korišćenjem školskog Jezgra, realizovati klasu `TimeoutActivity` čiji je interfejs prema izvedenim klasama i klijentima dat. Ova klasa predviđena je za izvođenje, tako da aktivna korisnička izvedena klasa može da pokrene neku aktivnost sa zadatom vremenskom kontrolom (*timeout*). Kada pokrene datu aktivnost, korisnička klasa poziva funkciju `waitWithTimeout()` sa zadatim vremenom. Pozivajuća korisnička nit se tada blokira. Kraj aktivnosti neki klijent zadaje spolja pozivom funkcije `endActivity()`. Tada se korisnička nit deblokira. Korisnička klasa može posle toga ispitati poreklo deblokade pozivom funkcije `status()` koja vraća EOA u slučaju okončanja aktivnosti pre isteka vremenske kontrole, odnosno TO u slučaju isteka roka pre okončanja aktivnosti. Predvideti da se kraj aktivnosti može signalizirati i posle isteka roka.

```
class TimeoutActivity {
public:

    void endActivity ();

protected:

    TimeoutActivity ();
    enum Status { NULL, EOA, TO };

    void waitWithTimeout (Time timeout);
    Status status ();

};
```

Prikazati upotrebu ovog koncepta na sledećem primeru. Program treba da posle slučajnog intervala vremena u opsegu  $[0..T1]$  (slučajan broj u opsegu  $[0..1]$  dobija se pozivom bibliotečne funkcije `rnd()`) ispiše korisniku poruku "Brzo pritisni taster!" Posle toga program očekuje da korisnik pritisne taster (pritisak na taster generiše prekid koji treba u programu obraditi) u roku od  $T2$  jedinica vremena. Ukoliko korisnik pritisne taster u tom roku, program ispisuje poruku "Ala si brz, svaka čast!" i ponavlja isti postupak posle slučajnog vremena. Ukoliko ne pritisne, program ispisuje poruku "Mnogo si, brate, kilav!" i ponavlja postupak. Pretpostaviti da je na raspolaganju bibliotečna funkcija `delay(Time t)` kojom se pozivajuća nit blokira na vreme  $t$ .

### 9.7

Korišćenjem školskog jezgra, realizovati apstraktnu klasu `MultipleTimeouts` koja je namenjena za nasleđivanje i čiji je interfejs prema izvedenim klasama dat u nastavku. Klasa je predviđena da podrži višestruku kontrolu do  $N$  kontrolnih intervala vremena (*timeout*). Iz koda izvedene klase može se startovati pojedinačno merenje kontrolnog vremena pozivom operacije `startTiming()`, čiji parametar  $i$  ukazuje na to koji od  $N$  intervala se startuje ( $0 \leq i < N$ ), a parametar `time` zadaje kontrolni vremenski interval. Merenje  $i$ -tog intervala može se zaustaviti pozivom operacije `stopTiming()`. Kada  $i$ -ti interval istekne, sistem treba da pozove apstraktnu operaciju `timeout()` čiji parametar ukazuje na to da je istekao interval

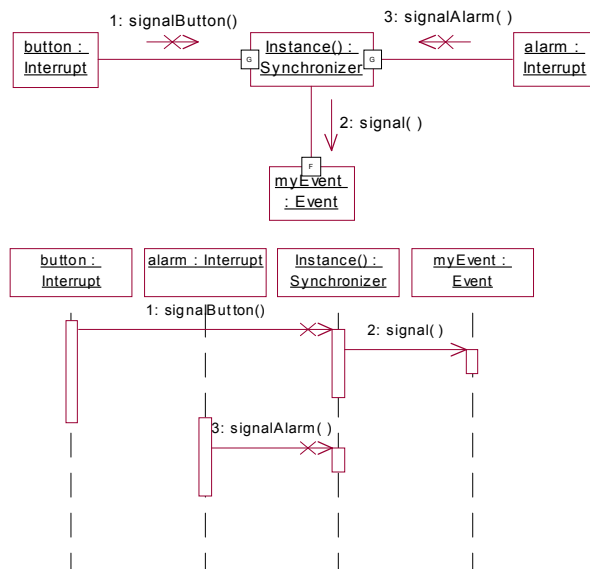
- i. Obezbediti da se eventualni istovremeni signali isteka više intervala sekvencijalizuju, tj. međusobno isključe.

```
const int N = ...;
```

```
class MultipleTimeouts {
protected:
    void startTiming (int i, Time time);
    void stopTiming (int i);
    virtual void timeout (int i) = 0;
};
```

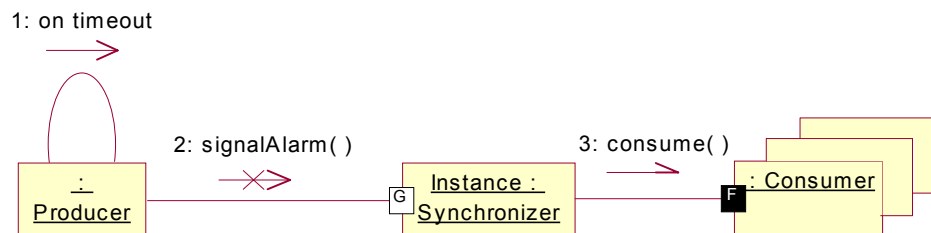
## 9.8

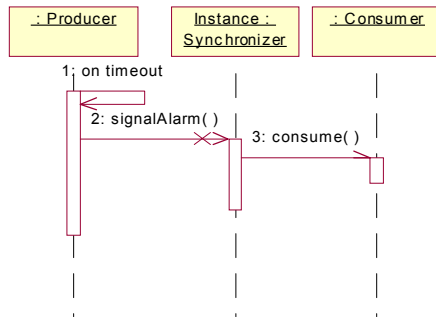
Korišćenjem školskog Jezgra, realizovati klase i njihove operacije prema datim dijagramima interakcija (dati dijagram kolaboracija i sekvence opisuju isti scenario). Klasa `Synchronizer` je sinhronizovana (monitor). Poziv `signalAlarm()` obavlja se u kontekstu nezavisnom od poziva `signalButton()`.



## 9.9

Na slikama su dati dijagrami interakcija koji prikazuju scenario izvršavanja operacije klase `Producer` koja se periodično izvršava. Napisati C++ kod za klase čije se instance pojavljuju na dijagramima, uz korišćenje školskog Jezgra.





### 9.10

Korišćenjem školskog Jezgra, potrebno je koncept *periodičnog posla* realizovati klasom `PeriodicTask`. Korisnik može da definiše neki periodični posao tako što definiše svoju klasu izvedenu iz klase `PeriodicTask` i redefiniše njenu virtuelnu funkciju `step()` u kojoj navodi kod za posao koji treba da se uradi u svakoj periodi. Konstruktoru klase `PeriodicTask` zadaje se perioda posla tipa `Time`, a funkcijom `start()` ove klase pokreće se posao. Korišćenjem ovog koncepta dati kompletan program kojim se kontroliše  $n$  sijalica koje trepću periodično, svaka sa svojom periodom (periode su zadate u nekom nizu), pri čemu su poluperiode ugašenog i upaljenog svetla jednake. Sijalica se pali i gasi pozivima funkcije `lightOnOff(int lighNo, int onOff)`.

### 9.11

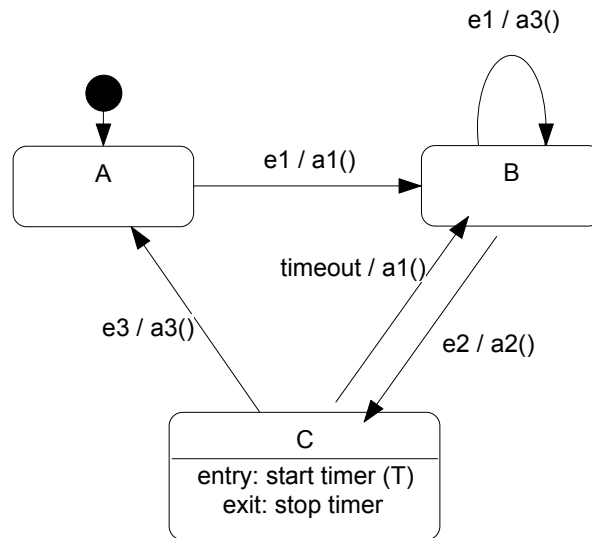
Svaki od više objekata klase `Client` periodično, sa svojom periodom koja mu se zadaje pri kreiranju, upućuje asinhronu poruku `accept()` sa sadržajem tipa `int` objektu klase `Server`. Ovaj objekat obrađuje pristigle poruke tako što njihov sadržaj ispisuje redom na ekran. Sadržaji poruka su brojevi koji označavaju redni broj poruke svakog klijenta. Realizovati klase `Client` i `Server` korišćenjem školskog Jezgra. Dati definicije klasa, a u glavnom programu konfigurisati sistem tako da poseguje  $N$  objekata klase `Client` i pokrenuti rad ovih objekata.

### 9.12

Klasa `Poller` je *Singleton*. Njen jedini objekat je zadužen da periodično proziva sve prijavljene objekte klase `DataSource` tako što poziva njihovu abstraktnu operaciju `poll()`. Perioda prozivanja treba da bude konstanta u programu koja se lako definiše i menja. Klasa `DataSource` je abstraktna. Konkretno izvedene klase treba da definišu funkciju `poll()`. Svaki objekat ove klase prilikom kreiranja prijavljuje se objektu funkcijom `sign(DataSource*)` i odjavljuje prilikom gašenja funkcijom `unsign(DataSource*)`. Nacrtati UML dijagram klasa i dijagram interakcije za ovaj sistem, a potom realizovati klase `Poller` i `DataSource` korišćenjem školskog Jezgra.

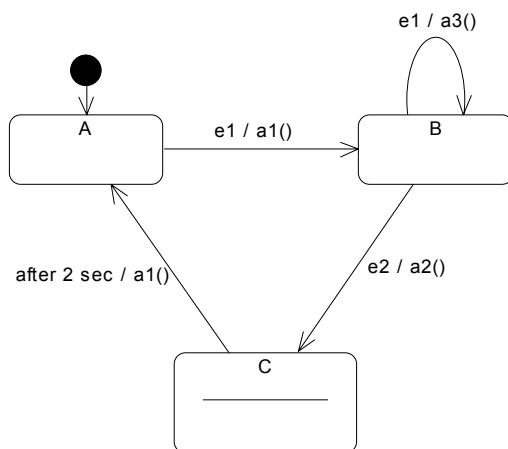
### 9.13

Korišćenjem školskog Jezgra, realizovati klasu `Actor` čije je ponašanje definisano konačnim automatom prikazanim na slici. Objekat klase `Actor` sadrži jedan vremenski brojač (*timer*) koji kontroliše vreme zadržavanja u stanju C. Na istek vremena  $T$  provedenog u stanju C vrši se prikazani prelaz u stanje B (vidi sliku).



### 9.14

Na slici je prikazan UML dijagram prelaza stanja za klasu  $x$ . Klasa  $x$  je aktivna, *Singleton*, a događaji  $e1$  i  $e2$  su konstante tipa `Event` i šalju se asinhrono. Implementirati klasu  $x$  na jeziku C++, uz korišćenje školskog Jezgra (pretpostaviti da je jedinica vremena 10 ms).



### 9.15

Potrebno je napisati program za kontrolu semafora za vozila sa tri svetla (crveno, žuto, zeleno). Semafor menja stanja na uobičajeni način: crveno, crveno-žuto, zeleno, žuto i tako ciklično. Vremena trajanja ovih stanja su TR, TRY, TG i TY, respektivno. Kada se semafor pokvari, treba da trepće žuto svetlo, sa jednakim poluperiodama upaljenog i ugašenog svetla jednakim TF; tada semafor trepće sve dok ne dođe serviser, otkloni kvar i ne resetuje semafor (semafor startuje od svog početnog stanja - start periode crvenog svetla).

Svetla se kontrolišu pozivom funkcije `light(Light which, int onOff)`; argument `which` (tipa `enum Light {R,G,Y}`) ukazuje na svetlo, a argument `onOff` da li svetlo treba upaliti ili ugasiti. Neispravnost semafora uzrokuje generisanje prekida. Osim toga, poseban prekid dolazi kada se iz kontrolnog centra vrši sinhronizacija svih semafora; tada semafor treba da pređe u unapred definisano stanje (koje se zadaje konfigurisanjem semafora).

Prikazati dijagram stanja semafora, a zatim realizovati sve potrebne klase za upravljanje semaforom po datom dijagramu stanja, korišćenjem školskog Jezgra.

### 9.16

Mnogi digitalni uređaji, npr. muzički aparati ili mobilni telefoni, zasnivaju svoju interakciju sa korisnikom prilikom podešavanja funkcionalnosti na sledećem principu. Ako korisnik kratko pritisne neki taster, onda taj taster ima jednu funkciju. Ako pak korisnik nešto duže drži pritisnut isti taster (recimo više od 2 sec), onda taster ima drugačiju funkciju. Korišćenjem školskog Jezgra, napisati na jeziku C++ klasu čiji je interfejs prema izvedenoj klasi prikazan dole. Ova klasa služi kao osnovna apstraktna klasa iz koje se može izvesti neka konkretna klasa koja definiše dve funkcije nekog tastera. Konkretna izvedena klasa treba da bude *Singleton*, vezana za jedan taster. Svaki pritisak ili otpuštanje tastera generiše prekid; informacija da li se radi o pritisku ili otpuštanju tastera može se dobiti očitavanjem zadatog registra (vrednost !=0 znači pritisak, a ==0 znači otpuštanje). Broj prekida kome se pridružuje izvedena klasa i adresa porta sa koga se čita navedena informacija parametri su konstruktora osnovne klase. Granično vreme koje određuje funkciju pritiska tastera određeno je takođe parametrom konstruktora, a može se i posebno postaviti odgovarajućom funkcijom `setTime()` ili očitati funkcijom `getTime()`. Konkretna izvedena klasa može da redefiniše dve virtuelne funkcije `onShortPress()` i `onLongPress()` koje se pozivaju ako je detektovan kratki, odnosno duži pritisak, respektivno.

```
class DblFunKey {
protected:

    DblFunKey (IntNo interruptNo, int portAddr, Time borderTime = 0);

    void setTime (const Time& borderTime);
    Time getTime ();

    virtual void onShortPress () {};
    virtual void onLongPress () {};
};
```

### 9.17

Korišćenjem školskog Jezgra, potrebno je realizovati podsistem kojim se može izmeriti prosečna perioda pojave prekida na ulazu `IntN`. Uslugu ovog merenja korisnik može da dobije pozivom funkcije:

```
Time getIntPeriod (Time timeInterval, Time maxPeriod);
```

Ova funkcija startuje merenje broja prekida koji se dogode za zadato vreme `timeInterval`. Argumentom `maxPeriod` se zadaje maksimalno očekivano rastojanje između pojave dva uzastopna prekida. Funkcija vraća količnik proteklog vremena merenja `timeInterval` i broja prekida koji su se dogodili. Ukoliko se desi da u roku od `maxPeriod` posle jednog prekida vremena ne stigne novi prekid, ova funkcija vraća 0. Korisnički proces se blokira unutar ove funkcije sve do završetka merenja.

### 9.18

Korišćenjem školskog Jezgra, realizovati klasu `InterruptTimer` koja je zadužena da meri vremenski interval između dva susedna prekida i da maksimalnu zabeleženu vrednost tog intervala čuva u svom atributu. Pretpostavlja se da maksimalna dužina tog intervala može biti



znatno veća od maksimalne dužine intervala koji može da meri objekat sistemske klase `Timer` pre nego što generiše signal *timeout*.

### 9.19

U sistemu postoje spoljni događaji koji se registruju prekidima. Za svaki događaj potrebno je izvršiti odgovarajući posao predstavljen prekidnim, sporadičnim procesom. Posmatrano na velikom intervalu vremena, prosečan vremenski razmak između događaja dovoljno je veliki da sistem uspe da odradi prekidni proces. Međutim, na kraćem vremenskom intervalu može se desiti da vremensko rastojanje između nekoliko događaja bude dosta kraće od vremena potrebnog da se obavi prekidni proces. Sistem treba u svakom slučaju da za svaki događaj koji se desio obavi po jedan ciklus prekidnog procesa (sekvencijalno jedan po jedan ciklus, za svaki događaj), predstavljen jednim pozivom funkcije `handle()` koju obezbeđuje korisnik. Za ovakve potrebe, treba realizovati klasu `InterruptHandler` koja će biti nalik na onu koja je data u školskom Jezgru, ali modifikovanu tako da zadovolji navedene zahteve. Osim toga, potrebno je da ova klasa obezbedi i zaštitu od preopterećenja sistema na sledeći način. Ukoliko se registruje da je broj trenutno neobrađenih događaja prešao odgovarajuću zadatu kritičnu granicu, treba da aktivira odgovarajući proces koji korisnik treba da obezbedi za tu namenu. Ovaj proces može, recimo, da ukloni spoljašnje uzroke prečestog nastajanja događaja. Realizovati ovako modifikovanu klasu `InterruptHandler` korišćenjem školskog Jezgra.

### 9.20

Neki prekid u sistemu stiže sporadično. Kada stigne prekid, potrebno je posle 10 jedinica vremena signalizirati semafor koji je pridružen tom prekidu. Ukoliko prekid ne stigne u roku od 50 jedinica vremena od prethodnog prekida, potrebno je uključiti sijalicu koja trepće periodično. Kada stigne novi prekid, potrebno je isključiti treptanje. Sijalica se pali ili gasi pozivom funkcije `lightOnOff(int onOff)`. Nacrtati sve relevantne UML dijagrame koji opisuju ovaj sistem, a potom ga realizovati korišćenjem školskog Jezgra.

### 9.21

Tri tastera su vezana tako da svaki pritisak na bilo koji od njih generiše isti prekid procesoru. U svakom trenutku, u bitima 2..0 registra koji je vezan na 8-bitni port TAST nalazi se stanje tri tastera (1-pritisnut, 0-otpušten). Ovaj sistem koristi se za suđenje boks-meča, kod kojeg tri sudije prate meč i broje udarce koje zadaje jedan bokser. Kada se dogodi udarac, svaki sudija pritiska svoj taster ukoliko je primetio taj udarac i smatra da ga treba odbrojati. Kako sudije mogu da pritiskaju tastere ili istovremeno (teoretski), ili sa vremenskom zadržkom, ali i tako da jedan još uvek drži taster dok je drugi izvršio pritisak, sistem treba da se ponaša na sledeći način. U periodu od 500 ms od prvog pritiska tastera, svaki pritisak jednog sudije smatra se za isti registrovani udarac. Dakle, eventualni ponovni pritisci ili duže držanje tastera jednog sudije u datom periodu se ignorišu. Posle isteka datog perioda od 500 ms, novi pritisak nekog tastera smatra se početkom novog perioda u kome se prate tastere (smatra se da sudija ne može da registruje dva različita udarca u razmaku manjem od 500 ms). Pošto se može dogoditi da neki sudija greškom pritisne taster, ili da neki sudija ne primeti udarac, odluka se donosi većinskom logikom: ako i samo ako je u navedenom periodu od 500 ms registrovano da su dvojica ili trojica sudija pritisnuli svoje tastere, udarac se broji. Treba obezbediti i kontrolu trajanja runde od 3 min: pritisci na tastere se uzimaju u obzir samo tokom trajanja runde. Korišćenjem školskog Jezgra realizovati ovaj sistem na jeziku C++.

### 9.22

Svetlosna dioda (LED) se pali i gasi pozivom funkcije `ledOnOff(int onOrOff)`, a na računar je vezan i taster koji generiše prekid procesoru kada se pritisne. Korišćenjem školskog Jezgra, potrebno je realizovati sistem za merenje motoričke sposobnosti korisnika. Sistem treba da, kada se pokrene pozivom operacije `start()`, upali svetlosnu diodu i drži je upaljenu 10 jedinica vremena. U toku tog perioda, dok je dioda upaljena, korisnik treba da što brže i češće pritisne i otpušta taster, a sistem treba da broji koliko puta je taster pritisnut. Kada se dioda ugasi, svi dalji pritisci na taster se ignorišu. Nakon 5 jedinica vremena od gašenja diode, ukoliko je broj pritisaka veći od zadatog broja  $N$ , sistem treba da upali diodu još jednom i drži je upaljenu 5 jedinica vremena, kao znak da je korisnikova motorika zadovoljavajuća. Ukoliko je broj pritisaka manji ili jednak  $N$ , diodu više ne treba paliti.

### 9.23

Svetlosna dioda (LED) se pali i gasi pozivom funkcije `ledOnOff(int onOrOff)`, sirena se pali i gasi pozivom funkcije `alarmOnOff(int onOrOff)`, a na računar je vezan i taster koji generiše prekid procesoru kada se pritisne. Pomoću ovakvog sistema kontroliše se budnost mašinovođe. Sistem treba na svakih 5 min da upali diodu. Ukoliko mašinovođa pritisne taster u roku od 5 s od uključenja diode, sistem se "povlači" i ponavlja postupak za 5 min. Ukoliko mašinovođa ne pritisne taster u roku od 5 s, sistem treba da uključi sirenu, koja treba da radi sve do pritiska na taster. Voditi računa da se eventualni pritisak na taster pre nego što je dioda uključena ignoriše. Ponašanje ovog sistema modelovati mašinom stanja i nacrtati taj model, a zatim realizovati ovaj sistem korišćenjem školskog Jezgra. Pretpostaviti da je jedinica vremena u Jezgru 100 ms.

### 9.24

Na ulaz računara vezan je A/D konvertor koji dati analogni signal konvertuje u digitalnu vrednost periodično, sa dovoljno velikom frekvencijom da se sa strane softvera konvertovana digitalna vrednost može smatrati kontinualno dostupnom. Konvertovana digitalna vrednost očitava se funkcijom `readValue():double`. Ovaj signal treba modulirati primenom adaptivne delta modulacije na sledeći način. Vrednost na ulazu treba očitavati (uzimati odbirke) u vremenskim intervalima čija je veličina obrnuto srazmerna veličini promene signala (veća promena – češće odabiranje). Preciznije, ako je  $\Delta A$  vrednost promene signala (razlika poslednje i preposlednje učitane vrednosti), onda narednu vrednost treba očitati kroz  $\Delta T = T_k * (A_k / \Delta A)$ , gde su  $T_k$  i  $A_k$  odgovarajuće konstante. Modulirana vrednost  $\Delta A$  predstavlja razliku poslednje i preposlednje učitane vrednosti. Tako modulirane vrednosti  $\Delta A$  treba slati periodično, sa periodom  $T_s$ , na izlaz računara. Jedna vrednost šalje se na izlaz funkcijom `sendValue(double)`. Uz korišćenje školskog Jezgra, napisati program na jeziku C++ koji obavlja opisanu modulaciju.

---

# Raspoređivanje

---

## Osnovne strategije raspoređivanja

### *Pojam raspoređivanja, rasporedivosti, prioriteta i preuzimanja*

- Kao što je do sada razmatrano, u konkurentnom programu nije potrebno definisati precizan redosled po kome se procesi izvršavaju. Lokalna ograničenja u redosledu izvršavanja, kao što su međusobno isključenje ili uslovna sinhronizacija, obezbeđuju se odgovarajućim sinhronizacionim primitivama. Međutim, izvršavanje konkurentnog programa i dalje podrazumeva značajnu dozu nedeterminizma u redosledu izvršavanja. Na primer, pet nezavisnih procesa se na jednom procesoru bez preuzimanja mogu izvršiti na  $5!=120$  načina. Ako je program korektan, onda će on dati isti izlaz bez obzira na detalje implementacije izvršnog okruženja koje određuje redosled izvršavanja.
- Sa druge strane, iako logičko ponašanje, tj. izlaz korektnog konkurentnog programa ne zavisi od redosleda izvršavanja, njegovo vremensko ponašanje značajno zavisi. Na primer, ukoliko jedan od pet procesa ima tesan rok, onda će verovatno samo redosledi u kome se taj proces izvršava među prvima biti vremenski korektni u smislu poštovanja vremenskih ograničenja, dok će ostali biti nekorektni. Zbog toga je kod RT sistema važno ograničiti nedeterminizam izvršavanja konkurentnih programa.
- Taj postupak naziva se *raspoređivanjem* (engl. *scheduling*). U opštem slučaju, raspoređivanje uključuje dva elementa:
  - Algoritam za definisanje redosleda korišćenja ograničenih sistemskih resursa (uglavnom procesorskog vremena) od strane konkurentnog programa.
  - Način za predviđanje ponašanja sistema u najgorem slučaju pri primeni datog algoritma raspoređivanja. Uz takvo predviđanje se onda može potvrditi da će vremenski zahtevi biti ispunjeni.
- Skup procesa je *rasporediv* (engl. *schedulable*) na datom ograničenom skupu resursa, ukoliko se može definisati raspored izvršavanja tih procesa koji zadovoljava sva postavljena vremenska ograničenja, u prvom redu zadovoljenje vremenskih rokova (engl. *deadline*).
- Rasporedivost je ključni problem kod *hard* RT sistema i predstavlja centralnu temu ovog poglavlja.
- Raspoređivanje može biti *statičko* (engl. *static*), što znači da se redosled definiše pre izvršavanja programa, ili *dinamičko* (engl. *dynamic*), što znači da se raspoređivanje vrši u vreme izvršavanja. Ovde će biti razmatrano samo statičko raspoređivanje.
- Ovde će uglavnom biti razmatrani postupci raspoređivanja *bazirani na prioritetima* (engl. *priority-based*) koji se i najčešće primenjuju. To znači da se procesima dodeljuju prioriteti i da se u svakom trenutku izvršava proces najvišeg prioriteta, osim ako je on suspendovan.
- U RT sistemima, prioriteti procesa su posledice njihovih vremenskih karakteristika, odnosno vremenskih ograničenja, a ne njihovog značaja za ispravnu funkcionalnost sistema ili njegov integritet.

- Ovde će biti razmatrani postupci raspoređivanja *sa preuzimanjem* (engl. *preemptive*). To znači da će sistem vršiti preuzimanje svaki put kada se u sistemu pojavi proces višeg prioriteta od onog koji se izvršava, npr. kao posledica deblokiranja procesa zbog poziva sinhronizacione primitive ili asinhronog spoljašnjeg događaja.
- Preuzimanje (engl. *preemption*), tj. dodela procesora drugom procesu može da se dogodi u sledećim slučajevima:
  - Kada proces eksplicitno traži preuzimanje, tj. "dobrovoljno" se odriče procesora, npr. pozivom funkcije `dispatch()` u školskom Jezgru ili naredbe `delay`.
  - Kada se proces suspenduje na nekom sinhronizacionom elementu, npr. semaforu.
  - Kada izvršno okruženje ili operativni sistem dobije kontrolu u nekom, bilo kom sistemskom pozivu. To može biti neblokirajuća operacija nekog sinhronizacionog elementa (npr. *signal* semafora), ili operacija koja je potencijalno blokirajuća (npr. *wait* semafora), nit se ne blokira jer nisu zadovoljeni uslovi za to, ali okruženje ipak implicitno vrši preuzimanje.
  - Kada dođe vreme za aktivaciju nekog periodičnog procesa ili istekne vreme čekanja nekog suspendovanog procesa.
  - Kada se dogodi neki spoljašnji asinhroni događaj koji se manifestuje npr. kao signal zahteva za prekid. Kao posledica tog događaja može se aktivirati neki sporadični proces višeg prioriteta, pa se vrši preuzimanje.
  - Kada istekne vreme dodeljeno datom procesu, npr. ako postoji ograničenje WCET ili mehanizam raspodele vremena (engl. *time sharing*).

Prva tri slučaja preuzimanja su *sinhrona*, jer se dešavaju kao posledica operacije koju je izvršio sam proces koji je prekinut. Druga tri slučaja preuzimanja su *asinhrona*, jer se dešavaju potpuno nezavisno od operacije koju tekući proces izvršava.

### ***Jednostavni model procesa***

- Proizvoljno složen konkurentni program nije jednostavno analizirati u smislu predviđanja njegovog ponašanja u najgorem slučaju. Zbog toga je neophodno uvesti određena pojednostavljenja, odnosno ograničenja i pretpostavke u vezi sa strukturom RT programa.
- Ovde se uvodi jedan jednostavan model RT programa koga koriste standardne opšte tehnike raspoređivanja, a koji je i često direktno primenjiv na *hard* RT sisteme. Taj model podrazumeva sledeća ograničenja:
  - Program se sastoji iz konačnog i fiksnog skupa procesa.
  - Svi procesi su periodični, sa unapred poznatom periodom.
  - Procesni su međusobno potpuno nezavisni.
  - Sva režijska vremena, kao što je vreme promene konteksta, se zanemaruju.
  - Svi procesi imaju rok (engl. *deadline*) jednak svojoj periodi; to znači da svaki periodični proces mora da se završi pre početka naredne periode.
  - Svi procesi imaju poznato i fiksno vreme izvršavanja u najgorem slučaju (WCET).
- Jedna posledica nezavisnosti procesa je da se može pretpostaviti da će u jednom trenutku svi periodični procesi biti aktivirani, tj. spremni za izvršavanje (engl. *released*). Taj trenutak predstavlja maksimalno opterećenje procesora i naziva se *kritični trenutak* (engl. *critical instant*).
- Notacija koja se ovde koristi je sledeća:

*B*      Vreme blokiranja u najgorem slučaju (ukoliko postoji), (engl. *worst-case blocking time*, WCBT)

*C*      Vreme izvršavanja u najgorem slučaju (engl. *worst-case execution time*, WCET)

*D*      Vremenski rok završetka (engl. *deadline*)

*I*      Vreme interferencije procesa

- $N$  Broj procesa u sistemu  
 $P$  Prioritet dodeljen procesu  
 $R$  Vreme odziva procesa u najgorem slučaju (engl. *worst-case response time*, WCRT)  
 $T$  Minimalno vreme između dve susedne aktivacije periodičnog procesa (period procesa)  
 $U$  Iskorišćenje (engl. *utilization*) procesa, jednako  $C/T$   
 $a-z$  Indeksi za označavanje procesa

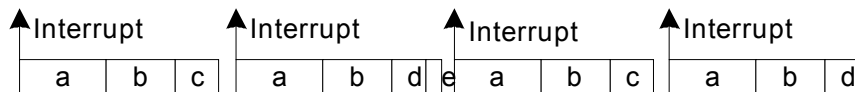
- Ovaj model biće neznatno relaksiran i uopšten na kraju ovog poglavlja.

### Ciklično izvršavanje

- Jedan krajnje jednostavan, ali veoma često korišćen način raspoređivanja u *hard* RT sistemima, jeste tzv. *ciklično izvršavanje* (engl. *cyclic executive*).
- Iako se sistem konstruiše kao konkurentan program, njegov fiksni skup isključivo periodičnih procesa se raspoređuje tako da se ti procesi izvršavaju ciklično, kao najobičnije procedure. Te procedure se preslikavaju na tzv. *mali cikluse* (engl. *minor cycle*) koji opet sačinjavaju tzv. *veliki ciklus* (engl. *major cycle*). Na primer, mali ciklus može da ima periodu od 25 ms, a četiri mala ciklusa mogu da čine veliki ciklus sa periodom od 100 ms. Tokom izvršavanja, takt (npr. prekid sata realnog vremena) nailazi svake male periode.
- Na primer, neka se sistem sastoji od sledećih periodičnih procesa sa zadatim periodama i vremenom izvršavanja (WCET):

Proces	$T$ [ms]	$C$ [ms]
$a$	25	10
$b$	25	8
$c$	50	5
$d$	50	4
$e$	100	2

Jedno moguće ciklično izvršavanje koje zadovoljava vremenske zahteve je sledeće:



Kod koji obezbeđuje ovakvo izvršavanje je:

```

loop
  wait_for_interrupt;
  procedure_a;
  procedure_b;
  procedure_c;
  wait_for_interrupt;
  procedure_a;
  procedure_b;
  procedure_d;
  procedure_e;
  wait_for_interrupt;
  procedure_a;
  procedure_b;
  procedure_c;
  wait_for_interrupt;
  procedure_a;
  procedure_b;
  procedure_d;
end loop;

```

- Neke važne osnovne karakteristike ovog pristupa jesu:

- Raspored je potpuno determinisan.
- U vreme izvršavanja zapravo ne postoje stvarni procesi; svaki mali ciklus je samo obična sekvenca poziva procedura.
- Procedure dele zajednički adresni prostor preko koga mogu da razmenjuju podatke. Ti podaci ne moraju da budu zaštićeni, jer zapravo nema konkurentnog pristupa.
- Sve periode "procesa" moraju da budu umnošci malog ciklusa.
- Poslednja navedena karakteristika je ujedno i osnovni nedostatak ovog pristupa. Ostali nedostaci su:
  - Teško je uklopiti sporadične procese u raspored.
  - Teško je uklopiti procese sa dugim periodama u raspored.
  - Teško je konstruisati, a potom i održavati ciklus.
  - Proces sa velikim vremenom izvršavanja mora da se podeli na komade fiksne i manje veličine, što smanjuje preglednost programa i otežava njegovo održavanje.
- Iako je ovo jednostavan i dosta korišćen pristup, njegovi nedostaci ukazuju na potrebu za opštijim pristupima koji ne moraju biti tako deterministički, ali i dalje moraju obezbediti predvidivost.

### ***Raspoređivanje procesa***

- Ciklično izvršavanje nije dovoljno opšte i ne podržava neposredno koncept procesa. Raspoređivanje treba zato direktno da podrži koncept procesa i da u svakom trenutku preuzimanja obezbedi odluku o tome koji će proces biti sledeći izvršavan.
- Pretpostavljajući nezavisnost procesa u jednostavnom modelu (bez interakcije procesa), proces se može naći u jednom u sledećih stanja: izvršava se (engl. *running*), spreman je za izvršavanje (engl. *runnable* ili *ready*), suspendovan je jer čeka na vremenski događaj (za periodične procese), ili je suspendovan jer čeka na ne-vremenski događaj (za sporadične procese).
- U teoriji i praksi RT sistema postoji veliki broj pristupa raspoređivanju. Ovde će biti razmatrana samo dva najznačajnija i najčešće primenjivana:
  - *Raspoređivanje na osnovu fiksnih prioriteta* (engl. *Fixed-Priority Scheduling*, FPS). Svaki proces ima svoj fiksni, statički (pre izvršavanja) određen prioritet. Za izvršavanje se uvek izabira onaj spreman proces koji ima najviši prioritet. U RT sistemima, prioriteti procesa su posledice njihovih vremenskih karakteristika, odnosno vremenskih ograničenja, a ne njihovog značaja za ispravnu funkcionalnost sistema ili njegov integritet. FPS je najčešće primenjivani pristup.
  - *Najkraći-rok-prvi* (engl. *Earliest Deadline First*, EDF). U trenutku preuzimanja, za izvršavanje se izabira onaj spreman proces koji je ima najskorije apsolutno vreme svog vremenskog roka. Iako se obično vremenski rok određuje statički i to kao relativno vreme (npr. 25 ms od trenutka aktivacije procesa, tj. početka njegove periode), apsolutno vreme roka se izračunava dinamički, u vreme izvršavanja.
- Kod FPS, proces višeg prioriteta može biti aktiviran (jer mu je došlo vreme za izvršavanje) tokom izvršavanja procesa nižeg prioriteta. Kod *preemptive* sistema doći će do preuzimanja u tom trenutku. Kod sistema koji nisu *preemptive*, proces nižeg prioriteta biće završen, pa će tek onda doći do promene konteksta. *Preemptive* sistemi su zbog toga reaktivniji, pa su time i poželjniji.
- Postoje i međuvarijante, kod kojih proces nižeg prioriteta nastavlja svoje izvršavanje, ali ne obavezno do svog završetka, već do isteka nekog ograničenog vremena, kada se dešava preuzimanje. Ovakav pristup naziva se *odloženo preuzimanje* (engl. *deferred preemption*).

- EDF šema takođe može biti sa ili bez preuzimanja.

### ***FPS i RMPO***

- Postoji jedan veoma jednostavan, ali *optimalan* način za dodelu prioriteta periodičnim procesima po FPS šemi, tzv. *dodela prioriteta monotono po učestanostima* (engl. *Rate-Monotonic Priority Ordering*, RMPO): periodičnim procesima se dodeljuju jedinstveni prioriteti, monotono uređeni prema učestanosti procesa, tako da proces sa kraćom periodom ima viši prioritet.
- Drugim rečima, za svaka dva procesa  $i$  i  $j$  važi:  $T_i < T_j \Rightarrow P_i > P_j$ . U ovom poglavlju se prioritetom smatra veličina iz skupa za koji je definisana relacija totalnog uređenja, npr. ceo broj, pri čemu veća vrednost znači viši prioritet.
- Na primer, za dati skup procesa, raspored prioriteta bi bio sledeći:

<i>Proces</i>	<i>Period, T</i>	<i>Prioritet, P</i>
<i>a</i>	25	5
<i>b</i>	60	3
<i>c</i>	42	4
<i>d</i>	105	1
<i>e</i>	75	2

- Ovaj raspored je optimalan u smislu koji iskazuje sledeća teorema:

*Teorema:* (O optimalnosti RMPO) Ako je dati skup periodičnih procesa rasporediv pomoću neke (bilo koje) *preemptive* FPS šeme, onda je on sigurno rasporediv i pomoću RMPO šeme.

Dokaz ove teoreme biće dat kasnije.

## **Testovi rasporedivosti**

### ***Test rasporedivosti za FPS zasnovan na iskorišćenju***

- Postoji jedan veoma jednostavan kriterijum provere rasporedivosti skupa periodičnih procesa pomoću neke (bilo koje) FPS šeme, zasnovan samo na ispitivanju iskorišćenja procesa. Iako on nije egzaktni, jer predstavlja samo dovoljan, ali ne i potreban uslov rasporedivosti, veoma je popularan zbog svoje jednostavnosti. Ispitivanje ovog kriterijuma složenosti je  $O(N)$ , što je značajno bolje od ispitivanja svih mogućih kombinacija raspoređivanja, što je *NP-kompletni* problem.
- Ovaj kriterijum definiše sledeća teorema.

*Teorema:* (FPS test rasporedivosti, Liu & Layland, 1973) Ako je sledeći uslov ispunjen, onda je dati skup periodičnih procesa (za koje važi  $D = T$ ) rasporediv pomoću FPS šeme:

$$\sum_{i=1}^N \left( \frac{C_i}{T_i} \right) \leq N(2^{1/N} - 1).$$

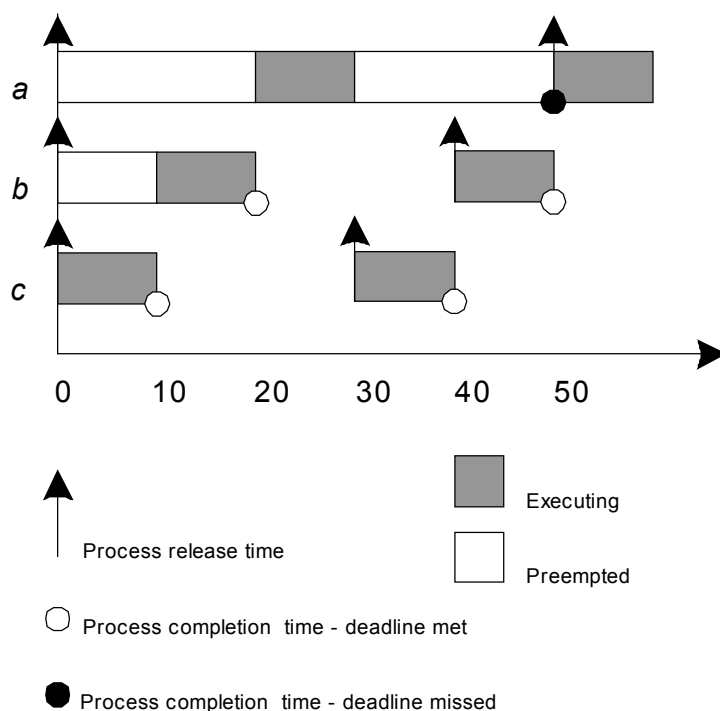
- Treba uočiti da suma sa leve strane nejednakosti zapravo predstavlja sumu svih iskorišćenja procesa, odnosno ukupno iskorišćenje procesora  $U$ .
- Izraz sa desne strane nejednakosti predstavlja graničnu vrednost iskorišćenja procesora za  $N$  procesa preko koje ovaj uslov više ne važi. Ova granična vrednost prikazana je u donjoj tabeli i teži ka 69.3% kada  $N$  teži beskonačnosti. Zbog toga svaki skup procesa koji ima

ukupno iskorišćenje manje od 69.3% *sigurno jeste* rasporediv pomoću *preemptive* RMPO šeme.

$N$	Granično $U$
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%

- Ovaj uslov je *dovoljan* za rasporedivost, što znači da će svi procesi sigurno ispoštovati svoje rokove ukoliko je uslov zadovoljen. Međutim, on nije i *neophodan*: ukoliko uslov nije zadovoljen, može, ali *ne mora* da se desi da neki proces prekorači svoj rok u vreme izvršavanja. Zbog toga ovaj test nije sasvim tačan, ali je siguran ukoliko je ispunjen.
- Primer 1 pokazuje skup od tri procesa koji imaju ukupno iskorišćenje 0.82 (ili 82%). Kako je ova vrednost iznad granične za tri procesa (0.78), ovaj test ukazuje da se dati skup procesa možda ne može rasporediti. Na slici je prikazana *vremenska osa* (engl. *time-line*) koja ilustruje ponašanje ovih procesa u vreme izvršavanja, ukoliko su sva tri počela u trenutku 0. U trenutku 50 je proces *a* iskoristio samo 10 jedinica vremena, a potrebno mu je 12, pa je on prekoračio svoj prvi rok.

Proces	$T$	$C$	$P$	$U$
<i>a</i>	50	12	1	0.24
<i>b</i>	40	10	2	0.25
<i>c</i>	30	10	3	0.33



- Primer 2 pokazuje skup od tri procesa čije ukupno iskorišćenje iznosi 0.775, što je ispod granične vrednosti, pa je garantovano da su ovi procesi rasporedivi.

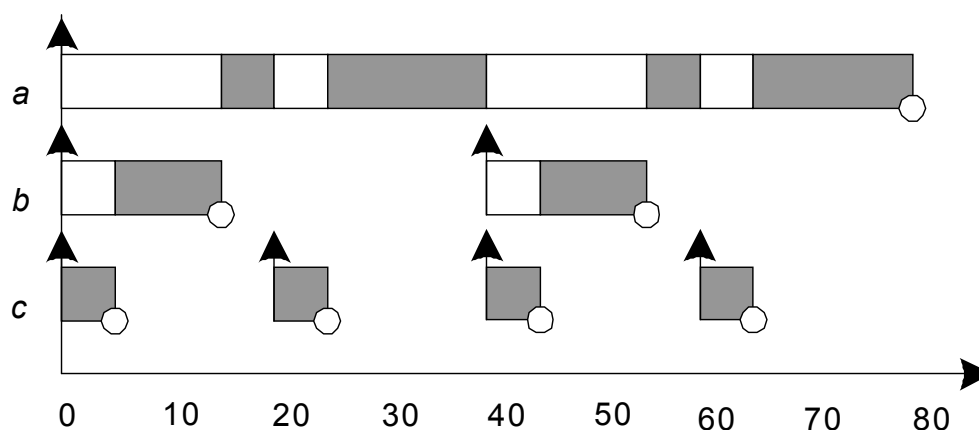
Proces	$T$	$C$	$P$	$U$
<i>a</i>	80	32	1	0.4



<i>b</i>	40	5	2	0.125
<i>c</i>	16	4	3	0.25

- Crtanje vremenskih osa može da posluži kao test provere rasporedivosti. Međutim, postavlja se pitanje koliko dugo mora da se konstruiše raspored u vremenu da bi bilo garantovano da se u budućnosti neće dogoditi iznenađenje i neki proces prekoračiti svoj rok? Za procese koji počinju u istom trenutku (kritični trenutak) može se pokazati da je dovoljno posmatrati vreme do isteka prve najduže periode procesa (Liu & Layland, 1973). To znači da je dovoljno pokazati da svi procesi zadovoljavaju svoj prvi rok, pa je sigurno da će zadovoljiti i svaki naredni.
- Primer 3 pokazuje skup od tri procesa čije ukupno iskorišćenje iznosi 100%, pa očigledno ne prolazi ovaj test rasporedivosti. Međutim, na vremenskoj osi se može pokazati da svi procesi zadovoljavaju svoj prvi rok, sve do trenutka 80, pa je ovaj skup ipak rasporediv.

<i>Proces</i>	<i>T</i>	<i>C</i>	<i>P</i>	<i>U</i>
<i>a</i>	80	40	1	0.5
<i>b</i>	40	10	2	0.25
<i>c</i>	20	5	3	0.25



### Test rasporedivosti za EDF zasnovan na iskorišćenju

- U istom antologijskom radu, Liu i Layland (1973) su formulisali i test rasporedivosti za EDF zasnovan na iskorišćenju:

*Teorema:* (EDF test rasporedivosti, Liu & Layland, 1973) Ako i samo ako je sledeći uslov ispunjen, onda je dati skup periodičnih procesa (za koje važi  $D = T$ ) rasporediv pomoću EDF šeme:

$$\sum_{i=1}^N \left( \frac{C_i}{T_i} \right) \leq 1.$$

- Kao što se vidi, ovaj test je najpre znatno jednostavniji nego za FPS. Osim toga, on je sasvim tačan, jer predstavlja *potreban i dovoljan* uslov rasporedivosti (očigledno je potreban, jer je jasno da skup procesa sa ukupnim iskorišćenjem većim od 1 nije rasporediv).

- Prema tome, EDF šema se u ovom smislu čini superiornijom u odnosu na FPS, jer dozvoljava znatno veća iskorišćenja procesora. Drugim rečima, EDF može rasporediti svaki skup procesa koji može rasporediti FPS, ali obrnuto ne važi.
- I pored ovih prednosti EDF šeme, FPS se ipak češće primenjuje iz sledećih razloga:
  - FPS je znatno lakše implementirati jer je prioritet (kao atribut procesa) statičan i izračunava se pre izvršavanja, a ne tokom izvršavanja kao kod EDF. EDF zato ima veće režijske troškove tokom izvršavanja.
  - Lakše je inkorporirati procese bez vremenskog roka u FPS nego u EDF, jer davanje proizvoljnog roka takvom procesu nije prirodno.
  - Vremenski rok ponekad nije jedini parametar od važnosti. Opet je lakše inkorporirati druge parametre raspoređivanja u pojam prioriteta (FPS) nego u pojam vremenskog roka (EDF).
  - Tokom perioda preopterećenja (engl. *overload*) koji su uzrokovani otkazima, ponašanje FPS je predvidljivije (procesni nižeg prioriteta će najpre prekoračiti svoje rokove) nego ponašanje EDF. EDF je nepredvidljiv u tim slučajevima i može da dovede do domino efekta kada veliki broj procesa prekoračuje svoje rokove.
  - Test rasporedivosti ponekad vodi do pogrešnog zaključka, jer je kod FPS on samo dovoljan, a ne i potreban, dok je za EDF on i potreban i dovoljan. Zbog toga se i za FPS mogu postizati veća iskorišćenja u nekim slučajevima.

### **Test rasporedivosti za FPS zasnovan na vremenu odziva**

- Test zasnovan na iskorišćenju ima dva bitna nedostatka: nije uvek tačan i nije primenjiv na opštije modele procesa. Opštiji pristup je zasnovan na analizi *vremena odziva* procesa (engl. *response-time analysis*, RTA).
- Ovaj pristup ima dve faze. U prvoj fazi se analitičkim putem izračunava vreme odziva u najgorem slučaju za svaki proces. Vreme odziva je vreme koje protekne od trenutka aktiviranja (engl. *release time*) periodičnog procesa (trenutak početka periode) ili sporadičnog procesa (spoljašnji događaj), do trenutka završetka njegovog izvršavanja. U drugoj fazi se jednostavno proverava da li je to vreme odziva u najgorem slučaju kraće od zadatog vremenskog roka ( $R \leq D$ ).
- Kod FPS raspoređivanja, vreme odziva najprioritetnijeg procesa jednako je njegovom vremenu izvršavanja ( $R = C$ ). Svi ostali procesi će trpeti *interferenciju* (engl. *interference*) procesa višeg prioriteta. Interferencija predstavlja vreme provedeno u izvršavanju procesa višeg prioriteta dok je posmatrani proces spreman za izvršavanje. U opštem slučaju, za proces  $i$  tako važi:

$$R_i = C_i + I_i,$$

gde je  $I_i$  maksimalna interferencija koju trpi proces  $i$  u bilo kom vremenskom intervalu  $[t, t+R_i)$ . Maksimalna interferencija očigledno nastupa kada se svi procesi višeg prioriteta aktiviraju istovremeno sa procesom  $i$ . Bez gubitka opštosti, ovaj kritični trenutak može se označiti kao trenutak 0.

- Posmatrajmo neki proces  $j$  koji je višeg prioriteta od procesa  $i$ . Tokom intervala  $[t, t+R_i)$ , proces  $j$  biće aktiviran bar jednom (jer njegova perioda sigurno nije veća od  $T_i$ ), a možda i više puta. Jednostavan izraz za broj njegovih aktivacija je:

$$\text{Number\_of\_releases}_j = \left\lceil \frac{R_i}{T_j} \right\rceil$$

- Na primer, ako je  $R_i$  jednako 15 a  $T_j$  jednako 6, onda će proces  $j$  biti aktiviran 3 puta (u trenucima 0, 6 i 12).

- Svaka aktivacija procesa  $j$  uzrokuje interferenciju dužine  $C_j$ . Zbog toga je:

$$\text{Maximum\_Interference}_j = \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

- Kako svaki proces višeg prioriteta uzrokuje interferenciju, onda je ukupna interferencija:

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

gde je  $hp(i)$  skup svih procesa višeg prioriteta od procesa  $i$ .

- Konačna jednačina izgleda ovako:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

- Iako je ova formula tačna, vrednost interferencije se ne zna, jer se ne zna ni  $R_i$ . Ova jednačina ima  $R_i$  na obe svoje strane, ali ju je teško rešiti zbog funkcije gornjeg zaokruživanja. U opštem slučaju, može postojati mnogo vrednosti  $R_i$  koje zadovoljavaju ovu jednačinu. Najmanja takva vrednost predstavlja vreme odziva procesa u najgorem slučaju.
- Najjednostavniji način za rešavanje navedene jednačine jeste formiranje rekurentne relacije:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

- Niz ovako dobijenih vrednosti  $\{w_i^0, w_i^1, w_i^2, \dots, w_i^n, \dots\}$  je monotono neopadajući. Kada se postigne da je  $w_i^{n+1} = w_i^n$ , rešenje jednačine je nađeno. Ako je  $w_i^0 < R_i$  (za  $w_i^0$  se može uzeti  $C_i$ ), onda je  $w_i^n$  najmanje rešenje, a time i tražena vrednost. Ako jednačina nema rešenja, onda će niz  $w$  nastaviti da raste. To se dešava za proces niskog prioriteta ukoliko je ukupno iskorišćenje veće od 100%. Ako se u nekom trenutku dobije vrednost veća od  $T_i$ , može se smatrati da proces neće ispoštovati svoj rok.
- Odavde se može formulisati algoritam za izračunavanje  $R_i$ :

```

for each process do
  W_old := C_i;
  loop
    Calculate W_new from W_old using the given equation;
    if W_new = W_old then
      R_i := W_new;
      exit {Value found};
    end if;
    if W_new > T_i then
      exit {Value not found};
    endif;
    W_old := W_new;
  end loop;
end for;

```

- Ukoliko ovaj algoritam da rešenje, ono će biti manje od  $T_i$ , pa time i od  $D_i$ , što znači da će proces zadovoljiti svoj rok ( $D_i = T_i$ ).
- Primer:

Proces	$T$	$C$	$P$
$a$	7	3	3
$b$	12	3	2
$c$	20	5	1

Proces  $a$ :  $R_a = 3$

Proces  $b$ :

$$w_b^0 = 3$$

$$w_b^1 = 3 + \left\lceil \frac{3}{7} \right\rceil 3 = 6$$

$$w_b^2 = 3 + \left\lceil \frac{6}{7} \right\rceil 3 = 6$$

$$R_b = 6$$

Proces  $c$ :

$$w_c^0 = 5$$

$$w_c^1 = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 3 = 11$$

$$w_c^2 = 5 + \left\lceil \frac{11}{7} \right\rceil 3 + \left\lceil \frac{11}{12} \right\rceil 3 = 14$$

$$w_c^3 = 5 + \left\lceil \frac{14}{7} \right\rceil 3 + \left\lceil \frac{14}{12} \right\rceil 3 = 17$$

$$w_c^4 = 5 + \left\lceil \frac{17}{7} \right\rceil 3 + \left\lceil \frac{17}{12} \right\rceil 3 = 20$$

$$w_c^5 = 5 + \left\lceil \frac{20}{7} \right\rceil 3 + \left\lceil \frac{20}{12} \right\rceil 3 = 20$$

$$R_c = 20$$

Kako za sve procese u najgorem slučaju važi  $R \leq T = D$ , svi procesi će zadovoljiti svoje rokove. Proces  $c$  će taj rok zadovoljiti "tesno".

- Za primer koji nije prošao test zasnovan na iskorišćenju, ovaj test daje pozitivan ishod:

Proces	$T$	$C$	$P$	$U$	$R$
$a$	80	40	1	0.5	80
$b$	40	10	2	0.25	15
$c$	20	5	3	0.25	5

- Zbog toga je ovaj test superiorniji, ali i složeniji od testa za FPS zasnovanog na iskorišćenju. On je, naime, i potreban i dovoljan: ako skup procesa prođe ovaj test, onda će svi procesi zadovoljiti svoje rokove; ako skup procesa ne prođe ovaj test, neki proces će prekoračiti svoj rok, osim ako procena WCET ( $C$ ) nije bila previše pesimistička i neprecizna.

### Procena WCET

- Sve opisane šeme raspoređivanja, kao i testovi rasporedivosti, pretpostavljaju da je unapred poznato vreme izvršavanja u najgorem slučaju (WCET) svakog procesa.
- Procena WCET se može izvesti na dva načina, merenjem i analizom koda. Problem sa merenjem je što se ne može uvek biti siguran da je u probnom izvršavanju postignut baš najgori slučaj. Problem sa analizom koda je što je potrebno poznavati model procesora (uključujući keš memoriju, protočnu obradu—engl. *pipeline*, predikciju grananja, čekanje na pristup memoriji itd.).

- Većina tehnika analize koda uključuje sledeće aktivnosti. Najpre se kod procesa dekomponuje na delove sekvencijalnog koda (bez skokova) koji se nazivaju *bazičnim blokovima* (engl. *basic block*). Povezivanjem bazičnih blokova (kao čvorova) granama koje predstavljaju tok kontrole programa formira se usmereni graf izvršavanja. Zatim se analizira mašinski kod pridružen svakom bazičnom bloku i model procesora, i pokušava da predvidi najduže vreme izvršavanja bazičnog bloka. Kada su vremena izvršavanja bazičnih blokova poznata, vrši se redukcija grafa tako što se više čvorova zamenjuje jednim sa najdužim vremenom; na primer, dva bloka koji čine *if-then-else* konstrukt zamenjuju se onim koji ima duže vreme izvršavanja.
- Naprednije tehnike analize koriste informaciju o semantici programa da bi dale precizniju procenu. Na primer, posmatrajmo sledeći deo koda:

```
for i in 1..10 loop
  if cond then
    -- Basic block of cost 100
  else
    -- Basic block of cost 10
  end if;
end loop;
```

Bez ikakve posebne dodatne informacije, procena WCET za ovaj kod bila bi  $10 \times 100 +$  vreme potrebno za režiju petlje, neka je to npr. 1005 vremenskih jedinica. Međutim, moguće je zaključiti (pomoću statičke analize koda) da uslov `cond` može biti tačan samo u tri iteracije. Tako se dobija mnogo manje pesimistična i preciznija procena od 375 jedinica.

- Očigledno je da kod koji se želi analizirati na WCET mora biti ograničen; na primer, petlje i rekurzije moraju biti ograničene.
- Poseban problem kod procene WCET je postojanje modernih elemenata u arhitekturi procesora i računara, kao što je keš memorija, protočna obrada, prediktor grananja itd. Ako se uticaj ovih elemenata zanemari, dobijaju se suviše pesimistične i neprecizne procene, dok je uticaj ovih elemenata veoma teško precizno modelovati.
- Zbog toga se u praksi uglavnom upotrebljavaju kombinovane tehnike analize (uglavnom pesimistične, koje zanemaruju uticaj optimizacija u hardveru i softveru) i detaljnog testiranja.

## Opštiji model procesa

### *Sporadični procesi*

- Da bi se jednostavni model procesa proširio tako da uključuje i sporadične procese, veličina  $T$  se za sporadične procese uzima tako da predstavlja njihovo minimalno vreme između susednih pojavljivanja. Na primer, za neki sporadični proces za koji se zna da se ne može pojaviti više od jednom u svakih 20 ms, uzima se da je  $T = 20$  ms u opisanim modelima. Iako sporadičan proces u stvarnosti može da se pojavi znatno ređe, analiza vremena odziva će dati rezultat za najgori slučaj.
- Drugi element vezan za sporadične procese jeste definicija pojma vremenskog roka  $D$ . U jednostavnom modelu procesa uzima se da je  $D = T$ . Međutim, ovo je za sporadične procese često neprihvatljivo, jer oni često predstavljaju obradu otkaza ili nekih vanrednih situacija. Oni se možda izvršavaju retko, ali su tada veoma hitni. Zato model mora da dozvoli postojanje  $D < T$ .

- Test rasporedivosti zasnovan na vremenu odziva, međutim, sasvim dobro funkcioniše i za  $D < T$ , sve dok je uslov zaustavljanja  $w_i^{n+1} < D_i$ . On takođe radi i za bilo koji redosled prioriteta, jer  $hp(i)$  uvek predstavlja skup procesa višeg prioriteta od procesa  $i$ .

### ***Hard i soft procesi***

- Na žalost, veoma često je vreme izvršavanja sporadičnih procesa u najgorem slučaju mnogo veće od prosečnog vremena izvršavanja. Prekidi često dolaze u naletima, a nenormalno očitavanje senzora može da dovede do značajnog vanrednog izračunavanja. Zbog svega toga, proveru rasporedivosti koja uzima u obzir vrednosti za najgori slučaj može da dovede do znatno slabijeg iskorišćenja procesora u stvarnom izvršavanju sistema.
- Kao opšti princip za projektovanje RT sistema se zato primenjuju sledeća pravila:
  - Svi procesi moraju da budu rasporedivi uzimajući u obzir *prosečne* vrednosti  $C$  i  $T$  svih procesa.
  - Svi *hard* procesi moraju biti rasporedivi uzimajući u obzir vrednosti  $C$  i  $T$  svih procesa u *najgorem slučaju*.
- Posledica prvog pravila je da se mogu pojaviti situacije u kojima neće biti moguće ispoštovati sve vremenske rokove. Ova situacija naziva se *tranzijentno preopterećenje* (engl. *transient overload*).
- Drugo pravilo, međutim, obezbeđuje da nijedan *hard* proces neće prekoračiti svoj rok. Ukoliko ovo pravilo ukaže na izuzetno malo iskorišćenje procesora u regularnim situacijama, treba preduzeti mere za smanjenje WCET.

### ***Aperiodični procesi***

- Jedan jednostavan pristup za raspoređivanje aperiodičnih procesa (koji nemaju definisano minimalno vreme između susednih pojava) po FPS šemi jeste da se oni izvršavaju sa prioritetom ispod svih *hard* procesa. To obezbeđuje da *hard* procesi ne mogu biti ugroženi od strane aperiodičnih procesa. Iako je ovaj pristup siguran, on može da ugrozi izvršavanje *soft* procesa jer oni mogu često da prekoračuju rokove zbog pojave aperiodičnih procesa.
- Da bi se ovaj problem rešio, može se primeniti tehnika *servera*. Server je koncept koji čuva resurse za *hard* procese, ali i obezbeđuje da se *soft* procesi izvršavaju što je pre moguće. POSIX podržava jednu varijantu servera.
- Jedan pristup za realizaciju servera je sledeći. Analiza rasporedivosti uzima u obzir i jedan poseban proces, tzv. server, koji ima definisan  $T_s$  i  $C_s$  tako svi *hard* procesi ostanu rasporedivi čak i ako se server izvršava sa periodom  $T_s$  i vremenom izvršavanja  $C_s$ .  $C_s$  predstavlja "kapacitet" dodeljen aperiodičnim procesima. U vreme izvršavanja, kada se pojavi aperiodični proces i pod uslovom da ima preostalog kapaciteta, on se izvršava sve dok se ne završi ili dok ne potroši kapacitet. Kapacitet se dopunjuje svakih  $T_s$  jedinica vremena.

### ***Sistem procesa sa $D < T$***

- Kao što je ranije pokazano, za sistem procesa sa  $D = T$  i FPS šemom raspoređivanja, RMPO je optimalan. Sa druge strane, u slučaju postojanja sporadičnih procesa, potrebno je dopustiti i slučaj da je  $D < T$ .
- Za sistem procesa sa  $D < T$  postoji sličan pristup koji je takođe optimalan – *monotono uređenje prioriteta prema vremenskom roku* (engl. *Deadline-Monotonic Priority*

*Ordering*, DMPO). Kod ove šeme, fiksni, statički prioriteti dodeljuju se procesima prema njihovom vremenskom roku, tako da važi:  $D_i < D_j \Rightarrow P_i > P_j$ .

- Na primer, sledeća tabela prikazuje skup procesa sa  $D < T$ , kojima su pridruženi prioriteti prema DMPO šemi i za koje je izvršena analiza vremena odziva u najgorem slučaju (primetiti da bi DMPO dodelio prioritete drugačije):

<i>Proces</i>	<i>Period, T</i>	<i>Rok, D</i>	<i>WCET, C</i>	<i>Prioritet, P</i>	<i>Vreme odziva, R</i>
<i>a</i>	20	5	3	4	3
<i>b</i>	15	7	3	3	6
<i>c</i>	10	10	4	2	10
<i>d</i>	20	20	3	1	20

- Treba naglasiti da test rasporedivosti baziran na iskorišćenju za EDF šemu (ukupno iskorišćenje manje od jedan) ne važi za sistem sa  $D < T$ . Međutim, i dalje važi da je EDF efikasnija šema u smislu da svaki sistem koji je rasporediv po FPS šemi jeste rasporediv i po EDF šemi, tako da se uslov za rasporedivost po FPS šemi može smatrati dovoljnim i za rasporedivost po EDF šemi.
- Optimalnost DMPO šeme iskazuje sledeća teorema:

*Teorema:* (O optimalnosti DMPO, Leung & Whitehead, 1982) Ako je dati skup periodičnih procesa rasporediv pomoću neke (bilo koje) *preemptive* FPS šeme, onda je on sigurno rasporediv i pomoću DMPO šeme.

*Dokaz:* Dokaz će pokazati sledeće: ukoliko je neki skup procesa  $Q$  rasporediv po nekoj šemi  $W$ , onda se ta šema (tj. raspodela prioriteta procesima)  $W$  može transformisati tako da se procesima preraspodele prioriteti prema DMPO, a da se rasporedivost ne naruši (tj. nijedan proces ne prekorači svoj rok). Svaki korak u transformaciji rasporedele prioriteta iz  $W$  u DMPO će očuvati rasporedivost.

Neka su  $i$  i  $j$  dva procesa iz  $Q$  sa susednim prioritetima tako da u  $W$  važi  $P_i > P_j$  i  $D_i > D_j$ . Definišimo šemu  $W'$  koja je ista kao i  $W$ , osim što su prioriteti procesa  $i$  i  $j$  međusobno zamenjeni. Posmatrajmo rasporedivost  $Q$  po šemi  $W'$ :

- Svi procesi sa prioritetima višim od  $P_i$  neće biti nikako ugroženi ovom promenom nižih prioriteta.
- Svi procesi sa prioritetima nižim od  $P_j$  neće biti nikako ugroženi ovom promenom, jer će i dalje trpeti istu interferenciju od strane procesa viših prioriteta.
- Proces  $j$ , koji je bio rasporediv po šemi  $W$ , sada ima viši prioritet, pa će trpeti manje interferencije i zato će sigurno biti rasporediv po šemi  $W'$ .

Ostaje da se pokaže da je proces  $i$ , kome je smanjen prioritet, još uvek rasporediv.

Prema šemi  $W$  važi:  $R_j < D_j$ ,  $D_j < D_i$ ,  $D_i \leq T_i$ , pa zato proces  $i$  samo jednom interferira tokom izvršavanja procesa  $j$ . Kada se prioriteti ova dva procesa zamene u  $W'$ , novo vreme odziva procesa  $i$  postaje jednako starom vremenu odziva procesa  $j$ . Ovo važi zato što se po obe šeme ukupno  $C_i + C_j$  vremena troši na izvršavanje ova dva procesa sa istom interferencijom od strane procesa viših prioriteta. Proces  $j$  je aktiviran samo jednom tokom  $R_j$ , tako da interferira samo jednom tokom izvršavanja  $i$  po šemi  $W'$ . Odatle sledi:

$$R'_i = R_j \leq D_j < D_i.$$

Odatle sledi da je proces  $i$  rasporediv po šemi  $W'$ .

Na isti način se šema  $W'$  može transformisati u neku novu šemu  $W''$  zamenom prioriteta druga dva procesa čiji prioriteti nisu u skladu sa DMPO itd., sve dok se ne postigne redosled prema DMPO. Konačno, doći će se do šeme DMPO, pri čemu će svi procesi i dalje biti rasporedivi. Prema tome, DMPO je optimalna.

- Treba primetiti da se isti ovaj dokaz može iskoristiti za specijalni slučaj procesa sa  $D = T$ , tj. kao dokaz optimalnosti šeme RMPO.

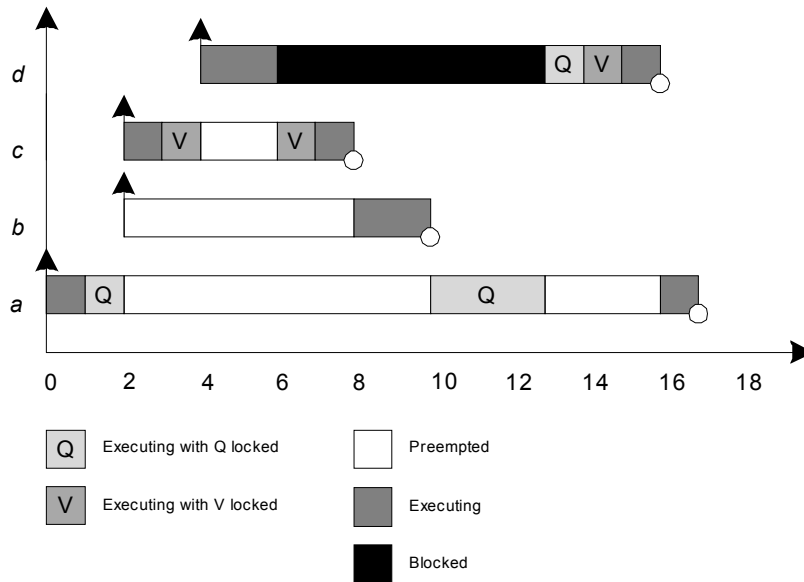
**Interakcija procesa i blokiranje**

- Jedno od najvećih pojednostavljenja u jednostavnom modelu procesa bila je pretpostavka da su procesi nezavisni, tj. da ne interaguju. Ova pretpostavka očigledno nije realna, jer praktično svaka stvarna aplikacija zahteva interakciju između procesa.
- Kao što je ranije pokazano, procesi interaguju ili pomoću neke forme zaštićenih deljenih podataka (koristeći, na primer, semafore, kritične regione, monitore ili zaštićene objekte), ili direktnom razmenom poruka (npr. pomoću randevua).
- Svi ovi koncepti omogućuju da neki proces bude suspendovan sve dok se ne ispuni neki uslov (npr. čekanje na semaforu ili uslovnoj promenljivoj, na ulazu u kritični region, monitor ili zaštićeni objekat, ili čekanje saučesnika za randevu). Zbog toga analiza rasporedivosti mora da uzme u obzir i vreme suspenzije (čekanja na takav događaj) u najgorem slučaju.
- Analiza u narednom izlaganju odnosi se isključivo na FPS šemu.
- Ako je proces suspendovan čekajući da neki drugi proces nižeg prioriteta završi neko izračunavanje, npr. oslobodi deljeni resurs, onda prioriteti procesa u neku ruku gube svoj smisao.
- Ovakva situacija, kada je proces suspendovan čekajući da neki drugi proces nižeg prioriteta završi svoje izračunavanje ili oslobodi resurs, naziva se *inverzija prioriteta* (engl. *priority inversion*). Za takav proces višeg prioriteta koji čeka na proces nižeg prioriteta kaže se u ovom kontekstu da je *blokiran* (engl. *blocked*).
- U idealnom slučaju, inverzija prioriteta ne bi smela da se desi. U realnim slučajevima je nju nemoguće izbeći, ali je cilj da se ona minimizuje i da bude predvidiva, kako bi mogla da se izvrši analiza rasporedivosti.
- Kao ilustraciju inverzije prioriteta, posmatrajmo primer četiri procesa ( $a$ ,  $b$ ,  $c$  i  $d$ ) kojima su prioriteti dodeljeni npr. prema DMPO šemi, kao što je prikazano u donjoj tabeli. Pretpostavimo da procesi  $d$  i  $a$  dele kritičnu sekciju (resurs), zaštićenu međusobnim isključenjem, označenu sa  $Q$ , a procesi  $d$  i  $c$  dele kritičnu sekciju  $V$ . U datoj tabeli simbol  $E$  označava nezavisno izračunavanje dužine jedne jedinice vremena, a simboli  $Q$  i  $V$  označavaju izvršavanje u kritičnoj sekciji  $Q$  odnosno  $V$ , dužine jedne jedinice vremena.

Proces	Prioritet	Sekvenca izvršavanja	Vreme aktivacije
$a$	1	EQQQQE	0
$b$	2	EE	2
$c$	3	EVVE	2
$d$	4	EEQVE	4

Na sledećoj slici pokazan je redosled izvršavanja ovih procesa prema FPS šemi:

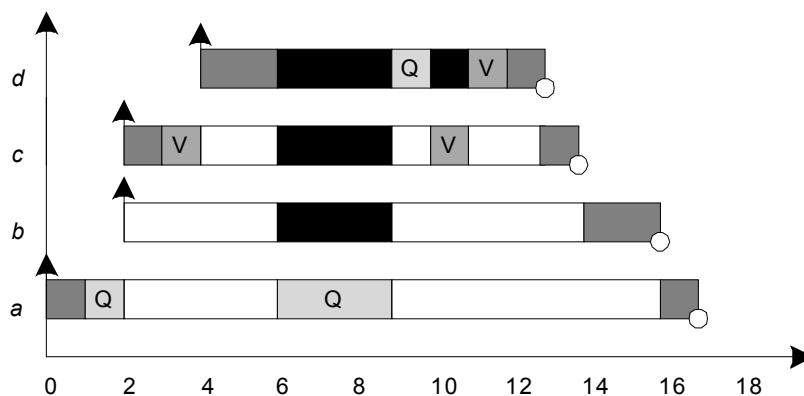




Kao što se vidi, proces  $d$  biva suspendovan u trenutku kada pokušava da pristupi resursu  $Q$ , jer je taj resurs već zauzet od strane procesa  $a$ . Zbog toga proces  $d$  ostaje blokiran sve dok proces  $a$  ne oslobodi resurs  $Q$ . Međutim, proces  $a$  je najnižeg prioriteta i zbog toga se ne izvršava sve dok procesi  $b$  i  $c$  ne završe, pa celo izvršavanje ozbiljno pati zbog inverzije prioriteta. Prema ovom izvršavanju,  $d$  završava u trenutku 16, pa ima vreme odziva jednako 12;  $c$  ima vreme odziva jednako 6,  $b$  jednako 8, a  $a$  jednako 17.

Proces  $d$  tako trpi blokadu ne samo od procesa  $a$ , nego i od procesa  $c$  i  $b$ . Blokada od strane procesa  $a$  je neizbežna, jer ona treba da obezbedi međusobno isključenje pristupa deljenom resursu  $Q$ . Međutim, blokada od strane drugih procesa se može i treba izbeći, jer ona ugrožava rasporedivost sistema.

- Jedan pristup za smanjenje vremena blokiranja jeste *nasleđivanje prioriteta* (engl. *priority inheritance*). Kod ove šeme prioriteta procesa nisu više fiksni, već se menjaju u vreme izvršavanja na sledeći način. Ukoliko je proces  $p$  suspendovan čekajući na pristup do resursa koga je zaključao proces  $q$  nižeg prioriteta, onda se procesu  $q$  dodeljuje prioritet procesa  $p$ . Kaže se tada da  $q$  nasleđuje prioritet od  $p$ . Tako je prioritet procesa u vreme izvršavanja jednak maksimumu njegovog podrazumevanog prioriteta, koji mu je statički dodeljen pomoću neke šeme (npr. DMPO), i prioriteta svih procesa koji su od njega zavisni u datom trenutku (čekaju ga).
- Prema ovoj šemi, izvršavanje procesa iz prethodnog primera izgleda ovako:



U trenutku kada se  $d$  aktivira i pokuša da pristupi resursu  $Q$ , prioritet procesa  $a$  koji je zauzeo  $Q$  se povećava na vrednost 4. Zato  $a$  oslobađa resurs  $Q$  procesu  $d$  znatno ranije, koji zbog toga takođe završava ranije. Treba primetiti da proces  $d$  sada trpi i drugu blokadu, ali mu je vreme odziva sada kraće i iznosi 9.

- U opštem slučaju, nasleđivanje prioriteta nije ograničeno na samo jedan korak, već je tranzitivno. Na primer, ako proces  $d$  čeka na proces  $c$ , a  $c$  opet čeka na proces  $b$ , onda će procesu  $b$  biti dodeljen prioritet procesa  $d$ . Zato će u vreme izvršavanja prioriteta procesa biti često menjani, pa o tome treba voditi računa pri implementaciji raspoređivača. Ranije prikazana implementacija reda sa prioritetima upravo odgovara ovim potrebama.
- Iako je nasleđivanje prioriteta od ogromnog značaja za RT sisteme, nije ga uvek jednostavno implementirati jer koncepti za sinhronizaciju i komunikaciju u konkurentnom jeziku nisu uvek pogodni za to. Na primer, izvršavanje operacije *wait* na semaforu, koja blokira proces, ne mora uvek da garantuje da će semafor osloboditi baš onaj proces koji je poslednji prošao kroz semafor.
- Kod šeme sa nasleđivanjem prioriteta postoji gornja granica broja blokada koje neki proces može da trpi. Ukoliko proces prolazi kroz  $m$  kritičnih sekcija, onda je maksimalni broj blokada koje mogu da mu se dogode u najgorem slučaju jednak  $m$ . Ukoliko postoji samo  $n < m$  procesa nižeg prioriteta, onda je taj broj jednak  $n$ .
- Ukupno maksimalno vreme  $B_i$  koje neki proces  $i$  može da provede blokiran od strane procesa nižih prioriteta, u slučaju nasleđivanja prioriteta, dato je sledećom formulom:

$$B_i = \sum_{k=1}^K usage(k,i)C(k)$$

gde je  $K$  ukupan broj resursa (kritičnih sekcija), a  $usage$  je 0/1 funkcija:  $usage(k,i)=1$  ako resurs  $k$  koristi bar jedan proces nižeg prioriteta od procesa  $i$  i bar jedan proces višeg ili istog prioriteta od prioriteta procesa  $i$ ; inače  $usage$  daje 0.  $C(k)$  je vreme izvršavanja kritične sekcije za resurs  $k$  u najgorem slučaju.

- Kada je poznato vreme blokade u najgorem slučaju za svaki proces, onda je njegovo vreme odziva jednako:

$$R_i = C_i + B_i + I_i,$$

odnosno može se dobiti prema formuli:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_j}{T_j} \right\rceil C_j$$

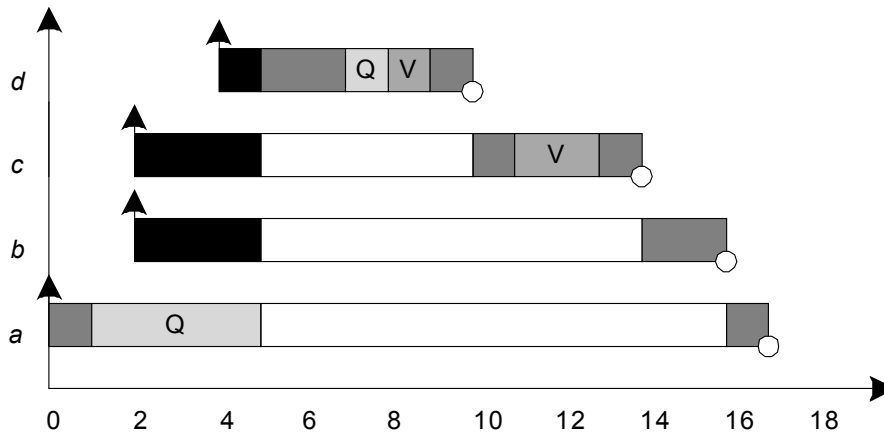
koja se opet izračunava na isti iterativan način kao i ranije.

- Treba naglasiti da ova analiza rasporedivosti sada može da bude pesimistična, tj. da uslov više nije potreban i dovoljan. Naime, da li će proces zaista trpeti blokadu za najgori slučaj zavisi od toga kako se procesi aktiviraju, tj. kako su fazno pomereni. Na primer, ako su svi procesi periodični i sa istom periodom, onda neće biti inverzije prioriteta.

### Protokoli prioriteta sa gornjom granicom

- Jedan pristup za značajno smanjenje vremena blokiranja predstavljaju *protokoli prioriteta sa gornjom granicom* (engl. *priority ceiling protocols*) (Sha et al., 1990).
- Jedan od nekoliko takvih protokola jeste tzv. *protokol prioriteta sa neposrednom gornjom granicom* (engl. *Immediate Ceiling Priority Protocol*, ICPP). On je definisan na sledeći način:
  - Svaki proces ima svoj statički dodeljen podrazumevani prioritet (dodeljen npr. DMPO šemom).

- Svaki resurs ima svoju statički dodeljenu *gornju vrednost prioriteta*, tzv. *plafon-vrednost* (engl. *ceiling value*), koja je jednaka maksimumu podrazumevanih prioriteta svih procesa koji koriste taj resurs.
- U vreme izvršavanja, proces ima svoj dinamički prioritet koji je u svakom trenutku jednak maksimumu vrednosti njegovog statičkog prioriteta i plafon-vrednosti svih resursa koje u tom trenutku on drži zauzete.
- Procesu se tako u vreme izvršavanja prioritet potencijalno menja čim zauzme neki resurs i postavlja se na plafon-vrednost tog resursa, ukoliko je prioritet tog procesa bio niži. Prilikom oslobađanja resursa, prioritet procesa se vraća na prethodnu dinamičku vrednost.
- Za isti prethodni primer, redosled izvršavanja procesa prema ICPP šemi izgleda ovako:



Proces *a* koji je zaključao resurs *Q* izvršava se sa prioritetom 4, što je plafon-vrednost ovog resursa (jer njega koristi i proces *d*). Zbog toga procesi *b*, *c* i *d* ne mogu da započnu svoje izvršavanje sve dok *a* ne oslobodi resurs *Q*, kada mu se prioritet vraća na podrazumevanu vrednost. Dalje procesi nastavljaju izvršavanje prema svojim prioritetima. Tako je vreme odziva procesa *d* sada samo 6 jedinica. Treba primetiti da se blokiranje procesa dešava samo na početku njegovog izvršavanja.

- ICPP podržavaju i POSIX (gde se naziva *Priority Protect Protocol*) i RT Java (gde se naziva *Priority Ceiling Emulation*).
- Važno svojstvo ovog protokola u opštem slučaju jeste da će proces trpeti blokadu eventualno samo na početku svog izvršavanja. Kada proces započne svoje izvršavanje, svi resursi koje on traži biće raspoloživi. Zaista, ukoliko bi neki resurs bio zauzet, onda bi on bio zauzet od strane procesa koji trenutno ima viši ili jednak prioritet od posmatranog (jer prema ICPP taj prioritet sigurno nije manji od plafon-vrednosti resursa, a ta vrednost opet nije manja od prioriteta posmatranog procesa), pa bi izvršavanje posmatranog procesa bilo odloženo.
- Posledica ovog svojstva na jednoprocorskim sistemima jeste da zapravo sam *protokol raspoređivanja obezbeđuje međusobno isključenje* pristupa resursima, pa to nije potrebno obezbeđivati na nivou sinhronizacionih primitiva. Kod ICPP, ako jedan proces zauzima neki resurs, onda će se taj proces izvršavati najmanje sa plafon-vrednošću prioriteta tog resursa, koja nije manja od prioriteta bilo kog drugog procesa koji koristi isti resurs. Tako nijedan drugi proces koji koristi isti resurs neće dobiti priliku za izvršavanjem od strane samog raspoređivača, pa je implicitno obezbeđeno međusobno isključenje.
- Takođe na jednoprocorskim sistemima, još jedno važno svojstvo jeste da ovaj protokol *sigurno sprečava mrtvo blokiranje*, jer proces sigurno dobija resurse koje traži posle

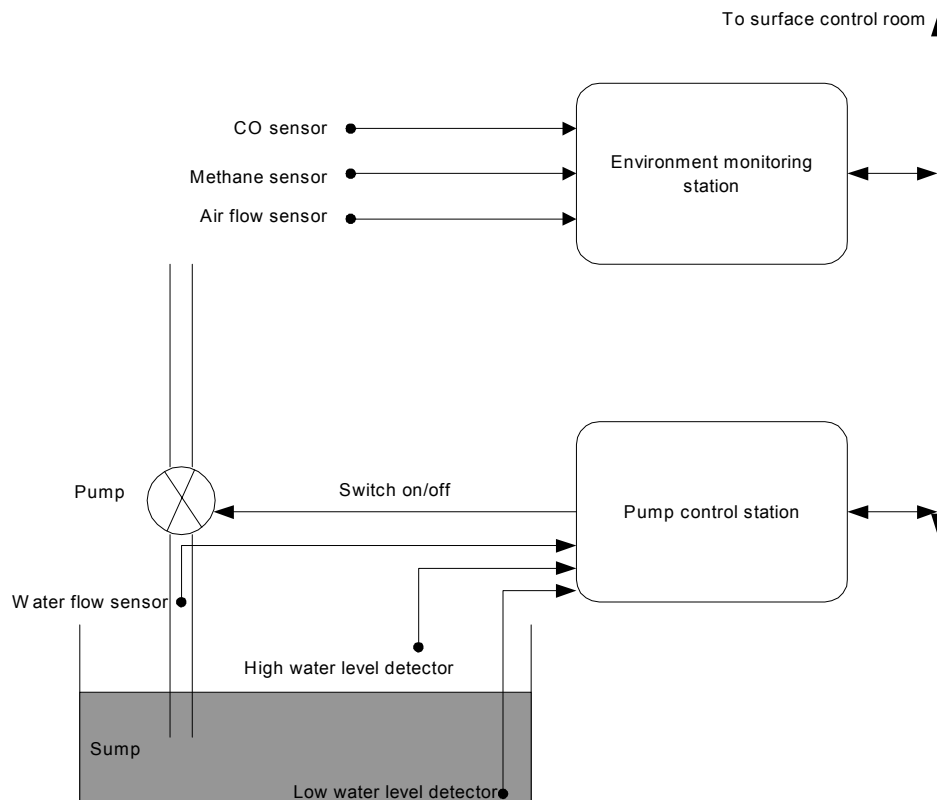
početka svog izvršavanja. Zbog toga je primena ovog protokola jedan način za prevenciju mrtvog blokiranja u RT sistemima.

- Prema tome, važna svojstva protokola sa gornjom granicom prioriteta na jednoprocorskim sistemima jesu:
  - Proces visokog prioriteta može biti blokiran samo jednom i to na početku svog izvršavanja.
  - Mrtvo blokiranje je sprečeno.
  - Transzitivno blokiranje je sprečeno.
  - Implicitno je obezbeđeno međusobno isključenje pristupa resursima.
- Zbog toga je vreme blokiranja procesa u najgorem slučaju dato sledećim izrazom:

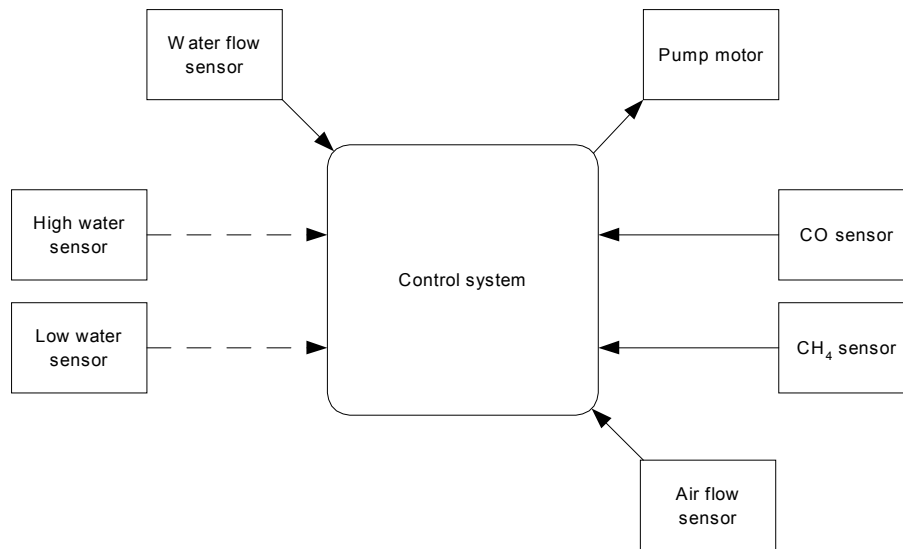
$$B_i = \max_{k=1}^K usage(k,i)C(k).$$

## Projektovanje prema vremenskim zahtevima

- Način na koji se sistem projektuje da bi ispunio postavljene vremenske zahteve biće ilustrovan na primeru softvera za upravljanje jednostavnog sistema pumpi za isušivanje rudnika. Pretpostavlja se da se sistem implementira na jednom procesoru.
- Sistem se koristi da bi na površinu ispumpavao podzemnu vodu koja se prilikom iskopavanja u rudniku sakuplja u cisterni. Osnovni sigurnosni zahtev je da zbog opasnosti od eksplozije pumpa ne sme da radi kada je nivo metana u rudniku iznad kritične granice. Šematski prikaz ovog sistema dat je na sledećoj slici:



- Relacije između upravljačkog sistema i spoljnih uređaja prikazane su na sledećoj slici. Samo senzori za visok i nizak nivo vode generišu prekide. Svi ostali uređaji moraju da budu kontrolisani prozivanjem (engl. *polling*).



### Funkcionalni zahtevi

- Kontroler pumpe treba da nadzire nivo vode u rezervoaru. Kada voda dostigne visok nivo, ili kada to operater zahteva, pumpu treba uključiti sve dok nivo vode ne dostigne donju granicu. U tom trenutku, ili kada operater to zahteva, pumpu treba isključiti. Stvarni tok vode kroz pumpu može se detektovati odgovarajućim senzorom, kako bi se proverilo stvarno stanje pumpe. Pumpa sme da radi samo ukoliko je nivo metana ispod kritične granice.
- Sistem nadzire i okolinu kako bi detektovao nivo prisustva metana i ugljen-monoksida u vazduhu, kao i to da li postoji dovoljan protok vazduha kroz ventilacioni sistem rudnika. Ukoliko nivo nekog od gasova ili protok vazduha dostigne kritičnu vrednost, potrebno je uključiti alarm.
- Operater na površini rudnika nadgleda sistema preko konzole. Operater se obaveštava o svim kritičnim događajima u sistemu. Pored toga, svi događaji i akcije u sistemu beleže se u arhivu, kako bi mogli da budu pregledani na zahtev.

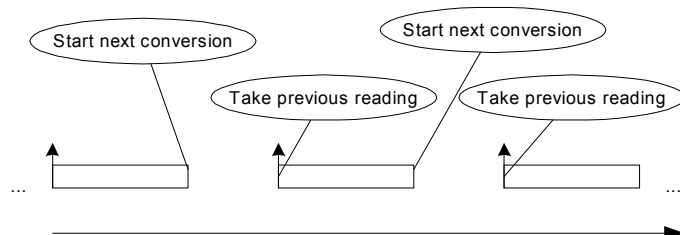
### Vremenski zahtevi

- Uređaji koji očitavaju neku fizičku veličinu iz okruženja pomoću senzora često funkcionišu po sledećem principu (podsetiti se načina funkcionisanja A/D konvertora). Procesor mora najpre da zada uzimanje odbirka sa analognog senzora i početak njegove konverzije u digitalni oblik. To zadavanje obično se izvodi upisom neke vrednosti u upravljački (kontrolni) registar uređaja (npr. A/D konvertora). Posle toga, uređaj vrši potrebno očitavanje i konverziju, što zahteva određeno vreme. Kada završi sa konverzijom i pripremi podatak za očitavanje u svom registru za podatke, uređaj može ili generisati prekid procesoru, ili samo postaviti odgovarajući indikator u svom statusnom registru. U svakom slučaju, taj podatak za očitavanje može biti spreman tek posle određenog vremena, a o spremnosti tog podatka procesor (tj. aplikacija) može biti obaveštena na jedan od sledeća tri načina:
  - Stalnim očitavanjem statusnog registra i ispitivanjem indikatora (engl. *polling*), tj. uposlenim čekanjem (engl. *busy waiting*) da taj indikator postane postavljen.
  - Prekidom koji generiše postavljeni indikator (engl. *interrupt*).

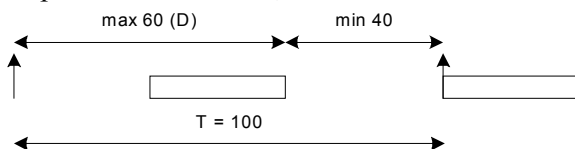
- Puštanjem da protekne odgovarajuće vreme za koje će ova vrednost sigurno biti raspoloživa, odnosno za koje će konverzija biti završena.

U slučaju da konverzija nije uspeła ili je došlo do bilo kakve greške, podaci o grešci se mogu očitati iz statusnog registra uređaja.

- U ovom primeru biće korišćen drugi pristup za detektore nivoa vode u rezervoaru, odnosno treći pristup za sve ostale senzore, jer uposlano čekanje nije dovoljno efikasno rešenje (nepotrebno troši procesorsko vreme).
- Senzore koji će biti prozivani kontrolisaće periodični procesi. Maksimalne periode očitavanja senzora mogu biti definisane odgovarajućim propisima. U ovom primeru pretpostavićemo da su periode očitavanja svih senzora iste i da iznose 100 ms. U slučaju praćenja nivoa prisustva metana, zahtevi mogu biti nešto strožiji, zbog inercije pumpe i potrebe da se osigura da ona bude isključena u slučaju kritičnog nivoa metana, kao što će biti opisano u nastavku.
- Pretpostavlja se da će ovi senzori sigurno završiti svoje očitavanje i konverziju za 40 ms od trenutka zadavanja konverzije. To očitavanje zadavaće se periodično, sa navedenom periodom od 100 ms. Za to periodično očitavanje biće zaduženi periodični procesi u projektovanoj RT aplikaciji. Međutim, bilo bi neefikasno da ti periodični procesi u istoj aktivaciji i zadaju konverziju i očitavaju vrednost, jer bi to zahtevalo nepotrebnu pauzu tokom istog izvršavanja procesa od 40 ms.
- Zato se u ovakvim situacijama primenjuje tehnika tzv. *pomeranja periode* (engl. *period displacement*). Ona se sastoji u tome da se na samom kraju jednog izvršavanja (aktivacije) periodičnog procesa koji proziva senzor zapravo zada očitavanje i pokrene konverzija (upisom odgovarajuće vrednosti u upravljački registar uređaja), a na početku narednog izvršavanja konvertovana vrednost očita iz registra podataka. Na taj način se rad konvertora paralelizuje sa radom procesora na drugim procesima u sistemu, pa se procesorsko vreme ne troši na čekanje konverzije. Ova tehnika prikazana je na sledećoj slici:

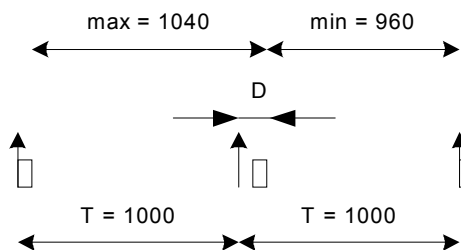


- Prema tome, sa jedne strane mora biti ispoštovana navedena perioda očitavanja senzora od 100 ms. Sa druge strane, međutim, mora biti ispoštovano vreme konverzije od 40 ms, što znači da najmanji razmak između kraja jednog izvršavanja procesa i početka drugog mora biti 40 ms. Ovo se može ispuniti zadavanjem sledećih parametara periodičnog procesa: perioda  $T = 100$  ms, vremenski rok  $D = 60$  ms:



- Proces koji kontroliše protok vode kroz pumpu takođe je periodičan i ima dve uloge. Prvo, kada je pumpa uključena, on proverava da li zaista ima protoka vode. Drugo, kada je pumpa isključena, on proverava da li je protok zaustavljen, kao potvrdu da je pumpa zaista isključena. Zbog inertnosti fizičkog sistema (pokretanja i zaustavljanja pumpe i protoka), ovom procesu se daje perioda od 1 sekunde, pri čemu će on koristiti dva

uzastopna očitavanja da bi utvrdio stvarno stanje pumpe. Da bi se obezbedilo da su dva uzastopna očitavanja zaista na razmaku od oko 1 sekunde, ovom procesu se zadaje tesan rok od 40 ms. Na taj način dva susedna očitavanja mogu da budu na razmaku od 960 ms do 1040 ms:



- Pretpostavlja se dalje da su procesi za nadzor nivoa vode pokretani događajem (engl. *event-driven*), tj. sporadični i da sistem treba da na njih reaguje u roku od 200 ms. Fizika sistema ukazuje da će proći bar 6 sekundi između dva ovakva događaja.
- Da bi se izbegla eksplozija, postoji vremenski rok do koga pumpa mora biti isključena od trenutka kada je nivo metana dostigao kritičnu vrednost. Ovaj rok je u relaciji sa periodom očitavanja senzora i vremenskim rokom procesa koji vrši to očitavanje. Ukoliko se koristi direktno očitavanje, tj. ukoliko se u svakoj aktivaciji procesa uzima odbirak bez čekanja na konverziju, onda je relacija sledeća:

$$T + D < T_c,$$

gde je:

$T$  perioda procesa koji očitava senzor;

$D$  vremenski rok procesa koji očitava senzor;

$T_c$  vreme koje se sme dozvoliti od trenutka kada nivo metana postane kritičan do gašenja pumpe.

Ovakva relacija je diktirana najgorim slučajem koji izgleda ovako. U jednom očitavanju vrednost može biti malo ispod kritične, pa proces neće zaustaviti pumpu. Međutim, odmah posle tog trenutka, nivo metana može postati kritičan i dalje rasti sve do narednog očitavanja kada će proces detektovati opasan nivo i preduzeti akciju isključenja pumpe. Od prvog do drugog trenutka može proteći u najgorem slučaju  $T + D$  vremena, kao što je pokazano na sledećem dijagramu:



Međutim, ukoliko se za očitavanje senzora koristi tehnika pomeranja periode, onda na ovo vreme treba dodati još jedno  $T$ , jer se u prvom očitavanju koje se obavlja na kraju izvršavanja  $i-2$  može očitati vrednost nešto ispod granične, odmah posle toga ona može postati kritična, a proces će to očitati tek u izvršavanju  $i$ . Zato relacija postaje:

$$2T + D < T_c$$

Ova dva parametra  $T$  i  $D$  mogu se birati i podešavati prema drugim uslovima i jedan prema drugom. U ovom primeru uzeto je da je  $T_c$  jednako 200 ms, što se može zadovoljiti sa  $T = 80$  ms i  $D = 30$  ms. Treba primetiti da se time zadovoljava i uslov da je vreme konverzije jednako 40 ms, jer je sa ovim parametrima vreme između uzimanja odbirka i očitavanja vrednosti najmanje  $80 \text{ ms} - 30 \text{ ms} = 50 \text{ ms}$ .

- Sve ukupno, parametri identifikovanih periodičnih i sporadičnih procesa u sistemu dati su u sledećoj tabeli (vremena su u milisekundama):

Proces	Periodičan/Sporadičan	"Period", $T$	Vremenski rok, $D$
CH <sub>4</sub> senzor	P	80	30

CO senzor	P	100	60
Protok vazduha	P	100	100
Protok vode	P	1000	40
Detektori nivoa vode	S	6000	200

- Dalji tok projektovanja i implementacije sistema bio bi sledeći. Najpre bi se prema funkcionalnim zahtevima konstruisao model ovog sistema korišćenjem nekog jezika za modelovanje (npr. UML). Zatim bi se iz tog modela napravila implementacija na ciljnom programskom jeziku (npr. Ada, RT Java ili C++), uz korišćenje odgovarajućih koncepata konkurentnog programiranja.
- Dobijeni program bi se zatim najpre funkcionalno testirao. Zatim bi se analizom ili merenjem odredilo maksimalno vreme izvršavanja (WCET,  $C$ ) i maksimalno vreme blokiranja ( $B$ ) svakog procesa u najgorem slučaju, sve za dati procesor koji je izabran za realizaciju.
- Na kraju, izvršila bi se analiza rasporedivosti ovih procesa na datom procesoru. Ukoliko analiza pokaže da je sistem rasporediv, vršilo bi se i njegovo testiranje pod realnim uslovima ili u simulacionom okruženju. Ukoliko analiza pokaže da sistem nije rasporediv, onda se mora pristupiti rešavanju problema bilo promenom vremenskih zahteva sistema, bilo promenom implementacije (smanjenjem  $C$  i  $B$ ), bilo izborom bržeg procesora.
- U ovom primeru nisu razmatrana pitanja tolerancije otkaza, ali bi se u realnom slučaju i taj element morao uzeti u obzir.
- Ostavlja se čitaocu da do kraja sprovede navedeni postupak i realizuje opisani program.

## Vežbe

### 10.1

Implementirati FPS raspoređivanje u školskom Jezgru. Izvršiti potrebne modifikacije u programskom interfejsu (engl. *Application Programming Interface*, API) prema korisničkom kodu koje su potrebne za zadavanje prioriteta niti, ali tako da kod koji je pravljen za prethodnu verziju Jezgra bude prevodiv i na novoj verziji.

### 10.2

Implementirati EDF raspoređivanje u školskom Jezgru. Izvršiti potrebne modifikacije u programskom interfejsu (engl. *Application Programming Interface*, API) prema korisničkom kodu koje su potrebne za zadavanje vremenskih rokova niti, ali tako da kod koji je pravljen za prethodnu verziju Jezgra bude prevodiv i na novoj verziji.

### 10.3

Implementirati raspoređivanje sa nasleđivanjem prioriteta (engl. *priority inheritance*) u školskom Jezgru. Izvršiti potrebne modifikacije u programskom interfejsu (engl. *Application Programming Interface*, API) prema korisničkom kodu koje su potrebne za zadavanje prioriteta niti, ali tako da kod koji je pravljen za prethodnu verziju Jezgra bude prevodiv i na novoj verziji. Kao jedinu primitivu za pristup do deljenih resursa predvideti realizovanu klasu `Mutex` koju treba modifikovati na odgovarajući način.



## 10.4

Implementirati ICPP raspoređivanje u školskom Jezgru. Izvršiti potrebne modifikacije u programskom interfejsu (engl. *Application Programming Interface*, API) prema korisničkom kodu koje su potrebne za zadavanje prioriteta niti, ali tako da kod koji je pravljen za prethodnu verziju Jezgra bude prevodiv i na novoj verziji. Kao jedinu primitivu za pristup do deljenih resursa predvideti realizovanu klasu `Mutex` koju treba modifikovati na odgovarajući način.

## 10.5

Tri procesa sa  $D = T$  imaju sledeće karakteristike:

Proces	$T$	$C$
$a$	3	1
$b$	6	2
$c$	18	5

- Pokazati kako se može konstruisati ciklično izvršavanje ovih procesa.
- Pokazati kako izgleda raspoređivanje ovih procesa prema FPS šemi, uz prioritete dodeljene prema RMPO šemi, počev od kritičnog trenutka.

## 10.6

Tri procesa sa  $D = T$  imaju sledeće karakteristike:

Proces	$T$	$C$
$a$	100	30
$b$	5	1
$c$	25	5

Pretpostavimo da  $a$  ima najveći značaj za sistem prema svojim funkcionalnostima, pa da zatim sledi  $b$  i potom  $c$ .

- Pokazati kako izgleda raspoređivanje ovih procesa po FPS šemi počev od kritičnog trenutka, pod uslovom da su prioritati dodeljeni prema značaju procesa.
- Koje je iskorišćenje procesa  $a$ ,  $b$  i  $c$ ?
- Kako treba raspoređivati procese da svi vremenski rokovi budu ispoštovani?

## 10.7

U sistem procesa iz prethodnog zadatka dodat je još jedan proces  $d$  čiji otkaz neće ugroziti sigurnost sistema i koji ima period 50 i vreme izvršavanja koje varira od 5 do 25 jedinica vremena. Diskutovati kako ovaj proces treba uključiti u prethodni sistem procesa.

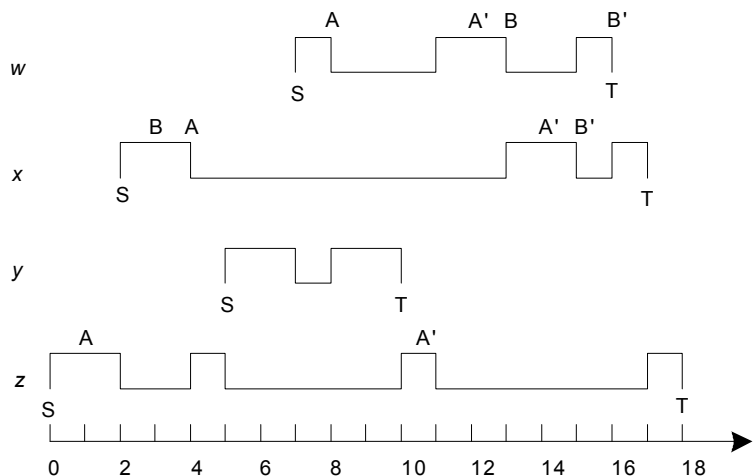
## 10.8

Dat je skup od četiri procesa sa sledećim karakteristikama:

Proces	Prioritet	Vreme aktivacije	WCET	Semafori
$w$	10	7	4	$A, B$
$x$	8	2	5	$A, B$
$y$	6	5	4	-
$z$	4	0	5	$A$

Ova četiri procesa dele dva resursa koji su zaštićeni pomoću semafora  $A$  i  $B$ . Simbol  $A$  (odnosno  $B$ ) na datom dijagramu označava izvršavanje operacije *wait* na semaforu  $A$  (odnosno  $B$ ), a simbol  $A'$  (odnosno  $B'$ ) označava izvršavanje operacije *signal* na semaforu  $A$

(odnosno  $B$ ). Dijagram prikazuje istoriju izvršavanja ova četiri procesa po FPS šemi. Svaki proces počinje u trenutku označenom sa  $S$ , a završava u trenutku označenom sa  $T$ . Na primer, proces  $x$  startuje u trenutku 2, izvršava uspešnu operaciju *wait* na semaforu  $B$  u trenutku 3, ali neuspešnu operaciju *wait* na semaforu  $A$  u trenutku 4 (jer je  $z$  već zaključao  $A$ ). U trenutku 13  $x$  se ponovo izvršava (sada zaključava  $A$ ), oslobađa  $A$  u trenutku 14, a  $B$  u trenutku 15. Tada biva preuzet od strane procesa  $w$ , ali se ponovo izvršava u trenutku 16 i konačno završava u trenutku 17.



- Nacrtati dijagram koji prikazuje izvršavanje ovih procesa uz raspoređivanje sa nasleđivanjem prioriteta.
- Nacrtati dijagram koji prikazuje izvršavanje ovih procesa uz raspoređivanje po ICPP šemi.

### 10.9

Da li je sledeći skup procesa sa  $D = T$  rasporediv po FPS šemi?

Proces	$T$	$C$
$a$	50	10
$b$	40	10
$c$	30	9

- Rasporedivost ispitati pomoću testa zasnovanog na iskorišćenju.
- Rasporedivost ispitati pomoću testa zasnovanog na vremenu odziva.

### 10.10

Sledeći skup procesa ne prolazi test rasporedivosti po FPS šemi zasnovan na iskorišćenju, ali je ipak rasporediv po FPS šemi. Objasniti kako.

Proces	$T$	$C$
$a$	75	35
$b$	40	10
$c$	20	5

### 10.11

U nekom RT sistemu kritične sigurnosti (engl. *safety-critical system*), skup procesa može da se koristi za nadzor ključnih događaja iz okruženja. Tipično postoji vremensko ograničenje

između trenutka pojave događaja i odgovarajuće reakcije sistema (izlaza). Pokazati kako se mogu koristiti periodični procesi za nadzor ovih događaja.

### 10.12

Posmatra se skup događaja zajedno sa vremenom izvršavanja potrebnim za reakciju na događaje. Ako se za nadzor svakog događaja koristi zaseban periodičan proces i ti procesi raspoređuju po FPS šemi, pokazati kako se može obezbediti da svi vremenski zahtevi budu ispunjeni.

<i>Događaj</i>	<i>Ograničenje vremena reakcije</i>	<i>Vreme izvršavanja</i>
<i>A</i>	36	2
<i>B</i>	24	1
<i>C</i>	10	1
<i>D</i>	48	4
<i>E</i>	12	1