Upravljanje procesima

Zad 1. Jun 2005.

(b)(10) Dat je sledeći C API nekog operativnog sistema za koncept standardnog brojačkog semafora:

```
typedef unsigned int SemID;
SemID createSemaphore ();
void releaseSemaphore (SemID);
void initSemaphore(SemID, int value);
void wait(SemID);
void signal(SemID);
```

Korišćenjem ovih funkcija na jeziku C napisati globalne deklaracije i inicijalizacije, kao i kod tela dve uporedne niti A i B koje ciklično rade sledeće:

A: upisuje vrednost u deljene promenljive `x` i `y`, a zatim čeka da proces B upiše zbir `x` i `y` u promenljivu `z` čiju vrednost ispisuje na standardni izlaz;

B: čeka da proces A upiše vrednosti u deljene promenljive `x` i `y`, zatim ove dve vrednosti sabira i zbir upisuje u deljenu promenljivu `z`.

Rešenje:

(b)(10)

```
// Globalne deklaracije i inicijalizacije:
SemID xySem = createSemaphore();
SemID zSem = createSemaphore();
initSemaphore(xySem,0);
initSemaphore(zSem,0);

// Kod tela niti A:
while (1) {
  x = ...;
  y = ...;
  signal(xySem);
  wait(zSem);
  printf("%d\n", z);
}

// Kod tela niti B:
while (1) {
  wait(xySem);
  z = x+y;
  signal(zSem);
}
```

Zad 2. Septembar 2005.

(b)(10) Funkcija `wait()` iz C API nekog operativnog sistema vrši sinhronizaciju procesa roditelja i njegove dece: proces-roditelj koji izvrši poziv ove operacije blokira se sve dok svi njegovi potomci ne završe izvršavanje. Predložiti realizaciju ove sinhronizacije pomoću standardnih brojačkih semafora. Precizno navesti:

- koje podatke treba obezbediti u PCB-u; precizno navesti njihove deklaracije i objasniti njihovu namenu;
- navesti tačno gde, kada i kako se ovi podaci inicijalizuju i menjaju;
- pod takvim pretpostavkama, napisati kod operacije `wait()`.

Rešenje:

(b)(10) Potrebno je obezbediti sledeće strukture podataka i operacije:

- Sastavni deo PCB-a treba da bude jedan brojački semafor `waitChildren` inicijalizovan na 0 prilikom kreiranja niti, koji služi za sinhronizaciju roditelja i dece:

```
Semaphore waitChildren(0);
```

- Sastavni deo PCB-a treba da bude pokazivač na roditeljsku nit koji se inicijalizuje tako da ukazuje na roditelja koji kreira novu nit (to je tekuća nit, `running`) prilikom kreiranja te nove niti:

```
// Deklaracija:
Thread* myParent;
// Incijalizacija:
myParent = running;
```

- Sastavni deo PCB-a je celobrojni podatak `numOfChildren` koji čuva broj kreiranih potomaka. Inicijalno je postavljen na 0, uvećava se za jedan prilikom kreiranja potomka, npr. u operaciji `fork()`:

```
running->numOfChildren++
```

- Na samom kraju svog izvršavanja, svaka nit, ukoliko ima roditelja, signalizira roditelju svoj završetak:

```
if (myParent) myParent->waitChildren.signal();
```

- Operacija `wait()`:

```
void wait () {
   while(numOfChildren>0) {
          waitChildren.wait();
          numOfChildren--;
   }
}
```

(c)(5) U operativnim sistemima koji podržavaju i koncept procesa i koncept niti, režijsko vreme kreiranja procesa značajno je veće, i po nekoliko desetina puta, od vremena kreiranja niti. Šta po Vašem mišljenju najviše utiče na ovakav odnos? Dati što preciznije objašnjenje.

Rešenje:

(c)(5) Kreiranje procesa, za razliku od kreiranja niti, podrazumeva formiranje novog adresnog prostora, koji uključuje i sam program koji se izvršava. Ukoliko proces-dete dobija kopiju adresnog prostora roditelja, uključujući i program i podatke (kao kod `fork()`), onda to u najmanju ruku podrazumeva kopiranje velikog dela operativne memorije. Inače, to podrazumeva učitavanje programa i inicijalnih podataka sa diska, što predstavlja vremenski još zahtevniju operaciju. U svakom slučaju, bilo koja od ovih operacija je značajno trajnija od operacije kreiranja samog konteksta niti (koja se uglavnom svodi na formiranje konteksta izvršavanja).

Zad 3. Koristeći gotovu klasu Semaphore, u kojoj je implementiran standardni brojački semafor, implementirati na programskom jeziku Java, sistem opisan u zadatku 1 pod b).

```
public class SharedData {
     public int x;
     public int y;
     public int z;

     public Semaphore xySem = new Semaphore(0);
     public Semaphore zSem  = new Semaphore(0);
}

public class TestSemaphore {
     public static void main(String[] args){
          SharedData sd = new SharedData();

          NitA na = new NitA(sd);
          na.start();

          NitB nb = new NitB(sd);
          nb.start();
     }
}

public class NitA extends Thread {

     public NitA(SharedData data){
          sd = data;
     }

     protected SharedData sd;

     public void run(){
          int i = 0;
          while(true){
               i++;
               sd.x = i;
               sd.y = i*i;
               sd.xySem.signalS();
               sd.zSem.waitS();

               System.out.println( sd.x + " + " + sd.y +
                                   " = " + sd.z);

               if (i == 10)
                    System.exit(0);
          }
     }
```

```
}

public class NitB extends Thread{
     public NitB(SharedData data){
          sd = data;
     }

     protected SharedData sd;

     public void run(){
          while(true){
               sd.xySem.waitS();

               sd.z = sd.x + sd.y;

               sd.zSem.signalS();
          }
     }
}
```

Zad 4. April 2006.

Proces P treba da sačeka da sva tri procesa X, Y i Z ispune neki svoj uslov, u bilo kom redosledu. Napisati deo koda procesa P i bilo kog od druga tri procesa, uz potrebne deklaracije, koji obezbeđuju ovu uslovnu sinhronizaciju pomoću jednog standardnog brojačkog semafora.

Rešenje:

```
var sem : Semaphore = 0;
process P                process X
begin                    begin
  ...                      ...
  wait(sem);               signal(sem);
  wait(sem);               ...
  wait(sem);             end;
  ...
end;
```

Zad 5. Oktobar 2005.

(a)(10) U nekom operativnom sistemu broj procesa koji se može kreirati limitiran je parametrom `MaxProcs`, pa su PCB strukture unapred alocirane i složene u jedan niz (vidi deklaracije dole). Redovi spremnih i blokiranih procesa ne postoje kao zasebne liste, već se stanje procesa čuva kao informacija unutar samog PCB. Funkcija `yield()` vrši snimanje konteksta procesora u PCB koji je dat kao prvi argument i restauraciju konteksta iz PCB-a koji je dat kao drugi argument. Polazeći od dole datih deklaracija i korišćenjem postojeće funkcije `yield()`, napisati kod funkcije `dispatch()` koja vrši promenu konteksta, s tim da se procesor daje prvom spremnom procesu u nizu svih procesa.

```
const int MaxProcs; // Maximal number of processes
enum ProcessState { free, running, ready, blocked };
  // free: the PCB slot is free (has not been allocated to
a process yet)
struct PCB {
  ProcessState state;
  ...
};
PCB allProcesses[MaxProcs]; // Array of all processes
PCB* running; // Running process
void yield(PCB* current, PCB* next);
void dispatch();
```

Rešenje:

(a)(10)

```
void dispatch () {
  for (int i=0; i<MaxProcs; i++) {
    PCB* next = &allProcesses[i];
    if (next->state!=ready) continue;
    next->state = running;
    running->state = ready;
    PCB* current = running;
    running = next;
    yield(current,next);
    return;
  }
}
```

(b)(5) Šta je ozbiljan nedostatak algoritma raspoređivanja procesa opisanog i realizovanog u tački pod (a)?

Rešenje:

(b)(5) Ozbiljan nedostatak je što na ovaj način zapravo samo prva dva spremna procesa jedan drugom naizmenično predaju procesor. Naime, posmatrajmo prva dva spremna

procesa u nizu, *X* i *Y*, tim redom. Kada *X* pozove `dispatch()`, predaće procesor procesu *Y*, pošto je on prvi u spreman u nizu (sam *X* je u stanju `running`). Kada potom *Y* pozove `dispatch()`, prvi spreman proces biće proces *X*, itd. Na taj način svi ostali procesi neće dobiti procesor sve dok se *X* ili *Y* ne završi. Ova pojava naziva se *izgladnjivanje* (*starvation*).

(c)(10) Dva procesa *X* i *Y* "proizvode" cele brojeve uporedo, nezavisnim i promenljivim brzinama. Proces *Z* uzima po dva proizvedena broja, bez obzira koji proces je proizveo te brojeve, i njihov zbir ispisuje na standardni izlaz. Važno je obezbediti da proces *Z* uvek uzima samo "sveže" proizvedene brojeve, tj. nikada ne uzme više puta isti proizvedeni broj. Nije važno koji proces je proizveo brojeve – procese *X* i *Y* ne treba nepotrebno sinhronizovati niti uslovljavati njihovu naizmeničnost: ako je npr. proces *X* spreman da proizvede još jedan broj, a proces *Y* nije, onda će proces *X* proizvesti dva uzastopna broja koja *Z* sabira, i obratno. Korišćenjem deljenih promenljivih i klasičnih brojačkih semafora, napisati sve potrebne deklaracije i kod procesa *X*, *Y* i *Z*.

Rešenje:

(c)(10)

```
var
    i : 0..1 := 0;
    a : array [0..1] of integer;
    mutex : semaphore := 1;
    readyToWrite : semaphore := 2;
    readyToRead : semaphore := 0;

process X begin
  loop
    readyToWrite.wait;
    mutex.wait;
    a[i]:=...;
    i:=(i+1) mod 2;
    mutex.signal;
    readyToRead.signal;
  end loop
end;

process Y - the same as X
```

```
process Z begin
  loop
    readyToRead.wait;
    readyToRead.wait;
    writeln(a[0]+a[1]);
    readyToWrite.signal;
    readyToWrite.signal;
  end loop
end;
```

Zad 6. Implementirati kružni, ograničeni bafer, kapaciteta N, koristeći brojačke semafore.

```
BoundedBuffer:
    i,j : integer := 0;
    const n : integer := …;
    a : array [0..n-1] of Item;
    mutex : semaphore := 1;
    spaceAvailable : semaphore := n;
    itemAvailable : semaphore := 0;

void put(Item itm)
begin
    spaceAvailable.wait;
    mutex.wait;
    a[i] := itm;
    i:=(i+1) mod n;
    mutex.signal;
    itemAvailable.signal;
end;

Item get()
begin
    itemAvailable.wait;
    mutex.wait;
    Item r = a[j];
    j:=(j+1) mod n;
    mutex.signal;
    spaceAvailable.signal;
    return r;
end;
```

Zad 7. Realizovati primitive wait(sem) i signal(sem), za binarni semafor, koristeći uposleno čekanje, na asembleru procesora 8086. Kada se isplati ovakva implementacija? (optimistički pristup)

`wait(sem):`

```
sloop: mov ax, 0
       xchg ax, sem
       jz sloop
       // ili jz dispatch
```

`signal(sem):`

```
mov ax, 1
mov sem, ax
```

Napomena: Mašinska instrukcija *xchg* ATOMIČNO vrši zamenu vrednosti svojih argumenata.

Zad 8. Januar 2006.

(a)(10) Na jeziku C napisati program `multirun` za Unix koji se poziva na sledeći način:

```
multirun 3 some_program
```

Program `multirun` treba da pokrene više procesa, onoliko koliko je zadato prvim argumentom u komandnoj liniji (3 u datom primeru), nad istim programom zadatim drugim argumentom u komandnoj liniji (`some_program` u datom primeru), a zatim da se odmah završi, ne čekajući da njegovi potomci završe. Bilo kakva greška (npr. nedostatak parametra u komandnoj liniji ili nemogućnost kreiranja procesa) treba samo da prekine ovaj program.

Rešenje:

(a)(10)

```
#include <stdio.h>

int main (int argc, char* argv[]) {
  if (argc<3) exit(-1);
  int num = 0;
  sscanf(argv[1],"%d",&num); // gets num. of required proc.
  char* program = argv[2];
  for (int i=0; i<num; i++) {
    int pid = fork();
    if (pid<0) exit(-1); //error
    if (pid==0) execlp(program);
  }
  exit(0);
}
```

Napomena:

- ID procesa PID – jedinstveni **int**
- sistemski poziv za kreiranje procesa: **fork()**
- proces-dete dobija kopiju adresnog prostora roditelja i kompletan kontekst, pa nastavlja izvršavanje od istog mesta kao i roditelj
- **fork()** vraća 0 u procesu-detetu, a ID deteta (!=0) u roditelju
- sistemski poziv **execlp()** zamenjuje program pozivajućeg procesa drugim programom – učitava programski kod iz fajla u adresni prostor procesa pozivaoca i počinje njegovo izvršavanje.

(b)(10) U nekom operativnom sistemu postoji podrška za rešavanje konflikta pri konkurentnom izvršavanju kritičnih sekcija tehnikom optimističkog pristupa bez čekanja i zaključavanja (*wait-free, lock-free synchronization*). U tu svrhu postoji generička funkcija

```
int cmpAndWrite(void* oldArr1, void* oldArr2, void* newArr,
int arrSize);
```

koja ovo podržava oslanjajući se na odgovarajuću mašinsku instrukciju, tj. atomično radi sledeće: poredi niz bajtova na koji ukazuje `oldArr1` sa nizom bajtova na koji ukazuje `oldArr2` i ako su isti, prepisuje niz bajtova `newArr` u niz bajtova `oldArr1` i vraća 1, inače samo vraća 0; sva tri niza su iste dužine `arrSize` (u jedinicama `sizeof(char)`).
Data je sledeća struktura `Buffer` i njoj pridružena operacija `put()`:

```
#define N 10000L
struct Buffer {
  char buf[N];  // Bafer za znakove
  long unsigned int firstEmptySlot; // Indeks prvog
                                    // slobodnog mesta u buf
};
void put(Buffer* b, char* str);
```

Funkcija `put()` dati niz znakova `str` (završen sa `'\0'`) dodaje na već postojeći sadržaj bafera `buf` zadate strukture `b`, ali tako da se bafer ne „prelije“, tj. da se ne prekorači veličina bafera `buf`. Ova funkcija nije bezbedna za konkrurentni pristup jer ne rešava potencijalne konflikte (nema međusobno isključenje, nije *thread-safe*).
Korišćenjem tehnike optimističkog pristupa bez čekanja i zaključavanja, realizovati funkciju `putThreadSafe()` sa istim potpisom kao i `put()`, ali tako da se može pozivati iz konkurentnih niti bez konflikta (da bude *thread-safe*).

Rešenje:
(b)(10)
Optimistički pristup bez čekanja i zaključavanja (*optimistic concurrency control, wait-free synchronization, lock-free synchronization*):

1. Napravi kopiju strukture na koju ukazuje dati pokazivač, modifikuj kopiju
2. Atomično uradi: uporedi izvornu strukturu sa pročitanom, ako su iste zameni pokazivač da ukazuje na kopiju
3. Ako je neki drugi proces izmenio polaznu strukturu, ponovi sve

```
void putThreadSafe(Buffer* b, char* str) {
  static const int sz = sizeof(Buffer);
  Buffer bOld, bNew;  // Ove dve strukture su na steku,
                      // privatne za nit
  int ok=0;
  while (!ok) {
    cmpAndWrite(&bOld,&bOld,b,sz); // bOld=*b, atomično
    bNew=bOld; // Ne mora da bude atomično
    put(&bNew,str);
    ok = cmpAndWrite(b,&bOld,&bNew,sz)
  }
}
```

(c)(5) Neki procesor poseduje koprocesor koji izvršava aritmetičke operacije nad brojevima u pokretnom zarezu (*floating point arithmetics*). Jezgro ovog operativnog sistema napravljeno je tako da u kontekst izvršavanja procesa ne ulazi kontekst koprocesora, drugim rečima, registri koprocesora i njegov status izvršavanja nisu deo konteksta procesa. Za taj operativni sistem pravi se standardna C biblioteka za matematičke funkcije koje treba da iskoriste ovaj koprocesor. Šta treba obezbediti u implementaciji ove biblioteke? (Precizno objasniti šta i kako treba uraditi u implementaciji biblioteke.)

Rešenje:

(c)(5) Problem je što funkcije ove biblioteke nisu *thread-safe*, jer se kontekst njihovog izvršavanja (u koji ulazi kontekst koprocesora) ne čuva unutar konteksta procesa. Koprocesor se zato može posmatrati kao resurs koga žele da dele različiti procesi, ali nije predviđen za konkurentan pristup. Zbog toga su sve funkcije ove biblioteke koje koriste koprocesor kritične sekcije za koje treba obezbediti međusobno isključenje. To se može uraditi klasično, korišćenjem semafora. Treba uraditi sledeće:

- obezbediti jedan globalni, statički semafor za celu biblioteku, inicijalizovan na 1
- sve ulaze i izlaze iz funkcija ove biblioteke koje koriste koprocesor obezbediti odgovarajućim operacijama *wait* i *signal* na ovom semaforu
- voditi računa da ne postoje ugnežđeni ulazi u kritičnu sekciju od strane istog procesa; (Zašto? Šta bi se desilo ukoliko bi isti proces pozivao više puta operaciju *wait* nad ovim globalnim semaforom?) Ukoliko postoji potreba da jedna funkcija biblioteke poziva drugu, a obe se mogu pozivati iz korisničkog procesa, treba ih rekomponovati na način prikazan u tački b) ovog zadatka, tj. rastaviti biblioteku na dva sloja: napraviti jezgro funkcija koje nemaju međusobno isključenje i ne koriste semafor, ali rade konkretne operacije, a onda napraviti viši sloj funkcija koje se pozivaju iz korisničkih procesa, imaju međusobno isključenje i pozivaju samo funkcije iz jezgra – nižeg sloja biblioteke.

Zadatak 9.
Kreirano je više procesa istog tipa *P* koji imaju istu sledeću strukturu. U kritičnoj sekciji *A* nalaze se ugnežđene dve kritične sekcije *B* i *C*, s tim da se sekcije *B* i *C* izvršavaju jedna posle druge (nisu ugnežđene). Potrebno je obezbediti sledeću sinhronizaciju: u kritičnoj sekciji *A* može se u jednom trenutku nalaziti najviše *N* ovih procesa tipa *P*, dok se u sekciji *B*, odnosno *C* može nalaziti najviše jedan proces. (Svaka od sekcija *B* i *C* je međusobno isključiva, ali se *B* i *C* ne isključuju međusobno – jedan proces može biti u *B* dok je drugi u *C*.) Korišćenjem standardnih brojačkih semafora obezbediti ovu sinhronizaciju: prikazati strukturu tipa procesa *P*, uz odgovarajuće definicije i inicijalizacije potrebnih semafora.

Rešenje:

```
shared var mutexA : semaphore:=N;
           mutexB, mutexC : semaphore:=1;

type P  = process begin
  ...
  wait(mutexA);
    <critical section A>
    ...
    wait(mutexB);
      <critical section B>
    signal(mutexB);
    ...
    wait(mutexC);
      <critical section C>
    signal(mutexC);
    ...
  signal(mutexA);
  ...
end;
```

Zadatak 10.
U nekom operativnom sistemu svi sistemski pozivi izvršavaju se kao softverski prekid koji skače na prekidnu rutinu označenu kao `sys_call`, dok se sama identifikacija sistemskog poziva i njegovi parametri prenose kroz registre procesora. Jezgro tog operativnog sistema je višenitno – poseduje više internih kernel niti koje obavljaju različite poslove: izvršavaju uporedne I/O operacije, vrše druge interne poslove jezgra itd. Prema tome, sve potrebne radnje prilikom promene konteksta korisničkih procesa (osim samog čuvanja i restauracije konteksta procesora), kao što su smeštanje PCB korisničkog procesa koji je do tada bio tekući u odgovarajući red (spremnih ili blokiranih, u zavisnosti od situacije), izbor novog tekućeg procesa iz skupa spremnih, promena memorijskog konteksta, obrada samog konkretnog sistemskog poziva, itd. obavljaju se u kontekstu internih kernel niti. Prilikom obrade sistemskog poziva u prekidnoj rutini `sys_call`, prema tome, samo se oduzima procesor tekućem korisničkom procesu i dodeljuje se tekućoj kernel niti.

Na PCB tekućeg korisničkog procesa ukazuje globalni pokazivač `runningUserProcess`, a na PCB tekuće interne niti jezgra ukazuje globalni pokazivač `runningKernelThread`. I interne niti jezgra vrše promenu konteksta između sebe na isti način, pozivom softverskog prekida koji ima istu internu strukturu kao i rutina `sys_call`.

Posebna interna kernel nit zadužena je za obradu sistemskog poziva izdatog od strane korisničkog procesa. Ova nit najpre određuje o kom sistemskom pozivu se radi i preuzima njegove parametre, a onda poziva odgovarajuće operacije jezgra koje obavljaju zahtevanu sistemsku uslugu, recimo operacije `wait` ili `signal` na semaforu.

Po uzoru na implementaciju klase `Semaphore` u školskom jezgru, implementirati ovu klasu za potrebe opisanog jezgra. Međusobno isključenje koda operacija `wait` i `signal` obezbeđuju uobičajene operacije `lock` i `unlock`. Operacije `lock` i `unlock` su implementirane (ne treba ih implementirati).

Rešenje:

```
void Semaphore::wait () {
  lock(lck);
  if (--val<0) {
    blocked.put(runningUserProcess);
    runningUserProcess = UserProcessScheduler::get();
  }
  unlock(lck);
}

void Semaphore::signal () {
  lock(lck);
  if (val++<0)
    UserProcessScheduler::put(blocked.get());
  unlock(lck);
}
```

Pomoćne operacije `block()` i `unblock()` više nisu potrebne (izbacuju se). Ostatak definicije klase `Semaphore` ostaje isti.

Zad 11. Jun 2005.

(c)(5) Posmatraju se dva procesa koji jedan drugom šalju i primaju podatke korišćenjem dva kružna ograničena bafera A i B veličine po 4KB. Ukoliko je bafer pun, proces koji želi u njega da upiše podatak se blokira sve dok se u baferu ne pojavi mesto za upis podatka. Ova dva procesa izgledaju ovako:

```
Process 1:                          Process 2:
send N bytes to BuffA               while (take 4KB of data
                                              from BuffA)
while(take data from BuffB)           process data
  display data                        send 4KB result to BuffB
end                                 end
```

Za koju vrednost N > 4K ovi procesi uzrokuju mrtvo blokiranje (*deadlock*)? Pretpostaviti da komanda `take` čeka neko vreme da se podaci pojave u baferu (*timeout*). Ako se podaci jave u predviđenom roku, vraća vrednost `true`, inače `false`.

(d) Šta bi se desilo u tački c), da komanda `take` nema `timeout`? Koliko je tada N koje uzrokuje blokiranje?

Rešenje:

(c)(5) Odgovor: N > 12 K.
Obrazloženje: Problem nastaje kada se oba procesa beskonačno blokiraju zato što ne mogu da završe operaciju slanja (jer je čekanje kod operacije prijema vermenski ograničeno). Konkretno, mrtvo blokiranje nastaje u sledećem slučaju: bafer B je pun sa 4KB podataka, ali proces 1 nije stigao do tačke kada iz njega uzima podatke, jer želi da pošalje još podataka u bafer A; proces 2 je već uzeo 4KB podataka iz bafera A i procesirao ih, ali ne može da ih smesti u bafer B jer je ovaj pun; bafer A je pun sa 4KB podataka; proces 1 želi da pošalje bar još jedan podatak u bafer A, ali ne može, jer je ovaj pun. Dakle, proces 1 je već poslao 3 puta po 4KB podataka kroz bafer A, ali želi da pošalje bar još jedan pre nego što uzme rezultat iz bafera B.

(d) Bilo koje N će uzrokovati blokiranje.

Obrazloženje:

Za N < 4K ili 4K < N < 8K ili 8K < N < 12K -> proces 2 se beskonačno blokira jer ne može u poslednjoj iteraciji da prihvati svih 4KB podataka, tako da ne može da pošalje rezultat procesu 2 koji kada završi slanje ostaje beskonačno blokiran na prijemu.

Za N = 4K ili 8K ili 12K -> proces 2 se beskonačno blokira na prijemu podataka, jer proces 1, pošto je ispisao sve rezultate koje je primio od procesa 2, ostaje je blokiran čekajući rezultate od procesa 2 (bafer A je prazan).

Za N >12K -> isto kao pod (c)

Zad 12. Implementirati sistem klasa koji predstavlja generalizaciju prekida.

Rešenje:

- Klasa `InterruptHandler` predstavlja generalizaciju prekida i 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.
- 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 IVT: 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;
```

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.
- 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 binarni semafor koji će prekidna rutina da signalizira, i jedan proces koji će na tom semaforu 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,
    - bafer je tipično složena struktura koja zahteva međusobno isključenje, pa time i potencijalno blokiranje.
    - Binarni semafor 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:

  Semaphore 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();
}
```