**Upravljanje procesima**

1. a)(10) Precizno odgovoriti na sledeća pitanja:
2. Šta čini kontekst izvršavanja nula-adresnog procesora (stek-mašine)?

 Programski brojač (PC) i pokazivač steka (SP).

1. Neki OS prilikom promene konteksta procesa ne briše (invaliduje) TLB. Zašto?

Zato što TLB je „globalan“, sadrži deskriptore za više procesa (i preslikava i ID procesa), a ne samo za jedan.

1. Neki OS prilikom kreiranja procesa ne alocira prostor ni za jednu stranicu novog procesa. Zašto?

Zato što primenjuje straničenje na zahtev (*demand paging*) bez ijedne inicijalno učitane stranice za novi proces. Kada proces bude dobio procesor, prilikom izvršavanja njegove prve instrukcije, generisaće *page fault* i stranica će biti učitana.

1. (a)(10) Neki procesor čuva sve programski dostupne registre na steku prilikom obrade i hardverskog i softverskog prekida, a restaurira ih u instrukciji *Return From Interrupt*. Jezgro operativnog sistema čuva sve programski dostupne registre procesora u strukturi PCB prilikom promene konteksta.
2. Precizno objasniti zašto ovakva realizacija promene konteksta nije najefikasnija.

Navedeni pristup nije dobar jer bespotrebno kopira (već sačuvane) vrednosti registara u PCB strukturu u memoriji. To kopiranje podrazumeva izvršavanje niza instrukcija i mnogo pristupa memoriji (uključujući i dohvatanje tih instrukcija), a za tim nema potrebe, jer su te vrednosti već sačuvane na steku i to tokom same obrade prekida (u jednoj instrukciji, nema trošenja vremena na dohvatanje, dekodovanje i izvršavanje više instrukcija).

1. Koji pristup biste vi izabrali za čuvanje konteksta izvršavanja prilikom promene konteksta? Šta bi u tom slučaju sadržao PCB (od konteksta izvršavanja)?

Umesto navedenog pristupa, bolje je koristiti pristup u kome se kontekst izvršavanja čuva na steku, a u PCBu čuva samo pokazivač steka (SP). Takav pristup, zajedno sa kodom za promenu konteksta, prikazan je i na predavanjima i na vežbama.

*iii*) Na koji način treba vršiti sistemski poziv koji uključuje promenu konteksta?

Ovakve sistemske pozive treba prosto vršiti softverskim prekidom, jer on obezbeđuje čuvanje konteksta izvršavanja na steku. Broj prekida može da označava sistemsku uslugu, a parametri poziva se mogu prenositi kroz registre ili u posebnoj strukturi u memoriji.

1. (b)(5) Na raspolaganju je funkcija swap(int\*,int\*) koja, korišćenjem odgovarajuće mašinske instrukcije, atomično zamenjuje vrednosti dve memorijske reči (C/C++ tip int je uvek veličine mašinske reči) na adresama zadatim argumentima. Korišćenjem ove funkcije realizovati apstrakciju Mutex koja obezbeđuje međusobno isključenje svojim operacijama wait() i signal(), odnosno ima semantiku binarnog semafora, pri čemu koristi uposleno čekanje.

class Mutex {
public:
 Mutex();
 void wait();
 void signal();
private:
 int lock;
};

Mutex::Mutex() : lock(1) {}

void Mutex::wait () {
 for(int acquired=0; !acquired;) swap(&acquired,&lock);
}

void Mutex::signal () { lock=1; }

1. U kojim situacijama je optimistički pristup bez zaključavanja i čekanja na kritičnu sekciju ili deljeni resurs generalno efikasniji od pesimističkog sa zaključavanjem?

U situacijama malog opterećenja i male učestanosti konflikta na kritičnoj sekciji ili deljenom resursu.

1. (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.)

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; 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.
1. 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.

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

* (a)(10) Kernel operativnog sistema za neki multiprocesorski računar sa deljenom memorijom čuva informaciju o *running* procesima za sve procesore u strukturi running:

struct PCB;
const int NumOfProcessors = ...;
PCB\* running[NumOfProcessors];

1. Precizno objasniti šta je potrebno da izvršavanje koda Kernela za promenu konteksta na jednom od procesora dohvati PCB baš svog *running* procesa.

Potrebna je hardverska podrška za identifikaciju procesora na kome se dati kod izvršava. Ova identifikacija može biti obezbeđena hardverskom konfiguracijom (npr. ožičavanjem odgovarajućih ulaznih pinova procesora na samoj procesorskoj ploči ili drugačije), a dostupna softveru pomoću odgovarajuće instrukcije ili čitanjem nekog konfiguracionog registra. Pretpostavimo da je takva instrukcija ili pristup registru enkapsulirana u C funkciju koja vraća ID (broj) procesora na kome se izvršava ta instrukcija:

int getProcessorID();

1. Napisati deo koda Kernela koji na određeno mesto u *running* nizu upiše pokazivač na novi *running* proces newRunning. Pretpostaviti da je pokazivač newRunning na steku.

Tada traženi kod za pristup strukturi izgleda jednostavno:

running[getProcessorID()] = newRunning;

(b)(5) Na jeziku C napisati kod za program multirun.exe za Unix koji pokreće više procesa-potomaka, onoliko koliko je zadato argumentom u komandnoj liniji, a zatim se odmah završava, ne čekajući da se njegovi potomci završe. Njegovi potomci ne treba da kreiraju svoje potomke, već prosto da ispišu neku poruku na standardni izlaz i da se završe. Bilo kakva greška (npr. nedostatak parametra u komandnoj liniji ili nemogućnost kreiranja procesa) treba samo da prekine ovaj program.

#include <stdio.h>

int main (int argc, char\* argv[]) {
 if (argc<2) exit(-1);
 int num = 0;
 sscanf(argv[1],”%d”,&num);
 for (int i=0; i<num; i++) {
 int pid = fork();
 if (pid<0) exit(-1);
 if (pid==0) {
 printf(“I’m a child! I’m terminating…\n”);
exit(0);
 }
 }
 exit(0);
}

(c)(5) Uz pretpostavku da proces-dete nasleđuje isti standardni izlazni uređaj od roditelja i da operativni sistem obezbeđuje međusobno isključenje operacija izlaza na taj uređaj (svaki poziv funkcije printf jeste jedan sistemski poziv), odgovoriti na sledeća pitanja:

void main () {
 int i = 1;
 printf(“%d”,i++);
 fork();
 printf(“%d”,i++);
 fork();
 printf(“%d”,i++);
 fork();
}

*i*) Koliko se ukupno jedinica ispisuje? Odgovor: 1

*ii*) Koliko se ukupno dvojki ispisuje? Odgovor:2

*ii*) Koliko se ukupno trojki ispisuje? Odgovor: 4

(b)(5) Neki operativni sistem podržava niti koje se kreiraju sistemskim pozivom

TID thread\_create(void (\*body)(void\*),void\* arg); // Returns thread ID

Ovim pozivom kreira se nit nad funkcijom na koju ukazuje pokazivač dostavljen kao prvi parametar ovog poziva, pri čemu se toj funkciji kao argument dostavlja vrednost dostavljena kao drugi parametar ovog poziva. Pomoću ovog sistemskog poziva implementirati C++ klasu Thread koja realizuje koncept niti nad polimorfnom funkcijom Thread::run(), a pokreće pozivom funkcije Thread::start().

(b)(10)

class Thread {
public:
 void start();
protected:
 Thread() {}
 virtual void run() {}
private:
 TID myTID;
};

void threadWrapper (void\* t) {
 if (t) ((Thread\*)t)->run();
}

Thread::start() {
 myTID = thread\_start(threadWrapper,this);
}

(c)(5) Neki operativni sistem poseduje sledeće sistemske pozive:

char\* get\_program\_name() vraća naziv (fajla) programa koji tekući proces izvršava;

int create\_process(char\* program\_name) kreira proces nad programom (u fajlu) sa datim imenom i vraća PID (>0) kreiranog procesa (<0 ako je došlo do greške).

Pomoću ova dva sistemska poziva napravljena je sledeća bibliotečna funkcija:

int quasi\_fork () { return create\_process(get\_program\_name()); }

Precizno objasniti u čemu je osnovna razlika između semantike ovakve operacije quasi\_fork() i standardnog sistemskog poziva fork().

Odgovor:

Funkcija quasi\_fork() pravi proces-potomak nad istim programom nad kojim radi i proces-roditelj, ali ne sa identičnim tekućim kontekstom izvršavanja kao što to radi fork(). Poziv fork() kreira potpunog klona sa identičnim tekućim kontekstom, koga čini ista tekuća pozicija u izvršavanju i identičan početni sadržaj adresnog prostora kao kod roditelja, dok quasi\_clone() kreira potomka koji počinje izvršavanje istog programa ali od njegovog početka, i to nad inicijalnim sadržajem adresnog prostora. Zbog toga se, recimo, u potomku izvršavanje nastavlja povratkom na isto mesto odakle je fork() pozvan u roditelju, dok kod quasi\_fork() potomak počinje izvršavanje od svog početka.

* Korišćenjem mehanizma za sinhronizaciju opisanog u tački (a), realizovane su dve klase Ping i Pong izvedene iz klase Thread, kreiran po jedan objekat ove dve klase (pokazivači myPong i myPing su podaci-članovi ove dve klase, respektivno, i za ova dva objekta ukazuju jedan na drugog) i pokrenute niti nad njima. Tela niti izgledaju ovako:

void Ping::run () { void Pong::run () {
 while(1) { while(1) {
 printf(“Ping\n”); waitOn(myPing);
 myPong->signal(); printf(“Pong\n”);
 waitOn(myPong); myPing->signal();
 } }
} }

Namera je da se obezbedi da ova dva procesa naizmenično ispisuju reči „Ping“ i „Pong“, tako što najpre Ping obavi svoj zadatak (ispiše „Ping“), pa se onda suspenduje i preda kontrolu niti Pong, itd. u nedogled. Da li gore dati kod obezbeđuje ovakvo ponašanje? Ukoliko obezbeđuje, dokazati, ukoliko ne, precizno objasniti zašto.

Ne obezbeđuje. Moguć je, između ostalih, sledeći neželjeni scenario koji vodi do blokade oba procesa: Ping se izvršava i izvrši celo telo svoje petlje – izvrši i myPong‑>signal() koji nema efekta, a onda izvrši waitOn(myPong) i suspenduje se. Za sve to vreme Pong uopšte ne kreće u izvršavanje jer ne dobija procesor, iako je spreman. Kada Pong krene u izvršavanje, izvrši waitOn(myPing) i suspenduje se. Tako obe niti ostaju trajno blokirane.

c)(10) Napisati tela dve niti Ping i Pong opisane u tački (c) tako da se obezbedi željeno ponašanje, korišćenjem standardnih brojačkih semafora. Naglasiti koje su inicijalne vrednosti korišćenih semafora.

Semaphore ping(1), pong(0);

void Ping::run () { void Pong::run () {
 while(1) { while(1) {
 ping.wait(); pong.wait();
 printf(“Ping\n”); printf(“Pong\n”);
 pong.signal(); ping.signal();
 } }
} }

(c)(5) Data su tri uporedna procesa koja rade nad istim deljenim semaforom *S* čija je inicijalna vrednost 2. Koja je minimalna, a koja maksimalna moguća vrednost koju ovaj semafor može imati u nekom trenutku nekog izvršavanja ova tri procesa? (Napomena: ne zahteva se da se te dve vrednosti postignu u istom izvršavanju.)

Proces A: Proces B: Proces C:

wait(S) wait(S) signal(S)
signal(S)

Minimalna vrednost: 0 Maksimalna vrednost: 3

(b)(5) Neki interaktivni višeprocesni i višekorisnički sistem ne podržava raspodelu vremena (*time sharing*). Šta je njegov osnovni problem?

Osnovni problem je neujednačenost (moguća velika razlika) u vremenu odziva na korisničku akciju. Ukoliko je više procesa spremno (aktivirano na korisničku akciju) i treba da obradi korisničku akciju, oni procesi koji ranije dobiju procesor imaće kraće vreme odziva, dok oni koji kasnije dobiju imaju duže vreme odziva. Još gore, vreme odziva istog procesa na sukcesivne korisničke akcije potpuno je nedeterminisano i zavisi samo od toga kako se u svakom slučaju rasporede procesi, pa može da varira od veoma kratkog do veoma dugačkog, u slučaju većeg opterećenja sistema. Kod sistema sa raspodelom vremena nema takvih varijacija i vreme odziva se približno ravnomerno produžava pod većim opterećenjem (više procesa), ali ne varira toliko od slučaja do slučaja.

(c)(5) Precizno objasniti zbog čega je potreban mehanizam raspodele vremena (*time sharing*) kod interaktivnih višeprocesnih i višekorisničkih operativnih sistema.

Kod interaktivnih sistema, jedan od važnih faktora jeste vreme odziva sistema na korisničku akciju, koje ne sme biti predugo (za subjektivni osećaj korisnika). Kod višekorisničkih, višeprocesnih sistema, važno je da vreme odziva ne bude predugo ni za jednog korisnika. Ukoliko kod ovakvih sistema ne bi postojala raspodela vremena (*time sharing*), dešavalo bi se to da je vreme odziva na neke akcije relativno kratko (za one procese koji prvi dođu na red za izvršavanje obrade akcije), a za druge relativno dugo, jer se obrade akcija sekvencijalizuju i svaka izvršava u potpunosti, neprekidivo (vidi sliku). Ovaj efekat velike razlike u vremenu odziva je utoliko izraženiji ukoliko je aktivno više korisničkih procesa. Sa mehanizmom raspodele vremena ovo vreme postaje ujednačenije, približnije kod svih procesa, i ujednačeno raste sa porastom broja aktivnih procesa, što stvara utisak da sistem paralelno obrađuje zahteve svih korisnika i niko od njih ne trpi čekanje zbog obrade onih drugih, već svi ravnopravno dobijaju procesorsko vreme.

(c)(5) Precizno objasniti šta je efekat izvršavanjasledećeg programa, pod pretpostavkom da su svi sistemski pozivi izvršeni uspešno (bez greške), kao i da su svi ispisi svih procesa usmereni na isti standardni izlaz (ekran), uz regularno međusobno isključenje pojedinačnih operacija ispisa različitih procesa (preko poziva printf()):

#include <stdio.h>
int main () {
 for (int i=0; i<10; i++) {
 int pid = fork();
 if (pid!=0) {
 printf(“%d”,i);
exit(0);
 }
 }
}

Svaki proces koji je nastao izvršavanjem ovog programa najpre kreira potomka koji nastavlja od iste tačke i stanja (iste vrednosti promenljive i i istog mesta u izvršavanju petlje), onda ispiše svoju vrednost promenljive i i ugasi se, dok potomak nastavlja dalje (ne ispisujući tu istu vrednost promenljive i, već prelazeći odmah na sledeću iteraciju). Dakle, od jednog inicijalnog procesa-izvršavanja ovog programa, nastaće još 9 potomaka i svaki od njih će na (isti) standardni izlaz ispisati po jedan znak ‘0’ do ’9’. Međutim, zbog navedenog redosleda izvršavanja operacija – najpre se kreira potomak pa tek onda ispiše vrednost promenljive i – redosled ispisa ovih deset znakova je potpuno nedeterminisan (isključivo zavisi od načina promene konteksta i raspoređivanja) i može biti bilo koja permutacija ovih deset znakova, nikako samo i uvek baš niz 0123456789.

(c)(5) Nad sledećim programom pokrenut je jedan inicijalni proces. Koliko iznosi zbir svih brojeva koji se ispisuju na zajedničkom standardnom izlaznom uređaju nakon završetka svih procesa koji su kreirani od ovog inicijalnog procesa, uključujući i njega? Pretpostaviti da proces-dete nasleđuje standardni izlazni uređaj od roditelja, da se izlazne operacije pozivom printf() na isti standardni uređaj od strane više procesa međusobno isključuju, kao i da su svi sistemski pozivi izvršeni uspešno.

#include <stdio.h>
void main () {
 int i = 1;
 int pid = fork();
printf(“%d\n“,i++);
 if (pid==0) { fork(); printf(“%d\n“,i++); }
}

Odgovor: 6

(c)(5) Koliko se ukupno procesa kreira nakon pokretanja jednog inicijalnog procesa nad sledećim program koji se nalazi u fajlu p.exe?

void main () {
 for (int i=0; i<3; i++) {
 int pid = fork();
 if (pid<0) exit(-1);
 if (pid==0) execlp(“p.exe”);
 }
}

Neodređeno mnogo, pošto svaki kreirani proces-potomak pokreće isti program ispočetka i kreira tri svoja direktna potomka, tako da će se procesi kreirati sve dok sistem ne bude preopterećen i ne dođe do svojih granica kapaciteta, pa prestane da kreira nove procese.

(b)(5) Pretpostavka je da standardni sistemski poziv execlp() u slučaju bilo kakve greške ne vrši nikakvu akciju i samo vraća pozivaocu kod greške koji je uvek različit od 0. Da li i kada sledeći program ispisuje rečenicu „Ok. Started program...“? Precizno obrazložiti odgovor.

#include <stdio.h>
void main (int argc, char\* argv[]) {
 if (argc>1)
 if (execlp(argv[1])!=0)
 printf(„Cannot start program %s.\n“, argv[1]);
 else
 printf(„Ok. Started program %s...\n“, argv[1]);
}

Odgovor:

Ne, nikada. U slučaju uspeha, poziv execlp() učitava imenovani program i započinje njegovo izvršavanje ispočetka, u kontekstu istog procesa iz koga je izvršen i ovaj sistemski poziv, uz inicijalizovan adresni prostor. U tom slučaju izvršavanje se nikada ne vraća iz poziva execlp(), pa se u datom programu else grana nikada ne izvršava.

b) U nekom C++ okruženju niti su predstavljene klasom Thread i imaju semantiku istu kao u jeziku Java (kako je pokazano i na predavanjima). Kreirana je jedna nit osnovne klase Thread čija operacija run() ima podrazumevano ponašanje (ne radi ništa):

Thread\* myThr = new Thread();

Precizno objasniti u čemu je razlika između poziva:

myThr->start();

i

myThr->run();

Odgovor:

Poziv myThr->start() kreira novu nit kao nezavisan i uporedan tok izvršavanja od onoga u kome je ovaj poziv izvršen; funkcija run() se izvršava u kontekstu te druge, nove niti. Poziv myThr->run()izvršava funkciju run() u kontekstu iste, pozivajuće niti i ne kreira novu nit kao uporedni tok izvršavanja.

c)(5) Neka rekurzivna funkcijaf implementirana je na sledeći način sa namerom da obezbedi međusobno isključenje svojih poziva i tako bude bezbedna za pozive iz konkurentnih niti. Objasniti šta je problem sa ovom implementacijom.

Semaphore s(1);

void f (...) {
 s.wait();
 ...
 if (...) f(...);
 ...
 s.signal();
}

Odgovor:

) Problem je to što jedna nit koja pozove f i uđe u rekurziju ulazi u mrtvu blokadu (*deadlock*) sama sa sobom. Naime, kada se f pozove iz neke niti, izvršava se wait na semaforu, što zatvara ulaz u f za nove pozive. Kada ta nit rekurzivno pozove f, ponovo se izvršava wait na istom semaforu, što dovodi do blokade pozivajuće niti koja zauvek ostaje blokirana na kritičnoj sekciji (semaforu) koju je sama zatvorila.

c)(5) Na raspolaganju je sistemska operacija:

int cmp\_xchg(int\* slot, int oldVal, int newVal);

koja atomično radi sledeće: najpre poredi celobrojnu vrednost u lokaciji na koju ukazuje slot sa vrednošću oldVal; ako su te vrednosti jednake, u lokaciju slot upisuje vrednosti newVal i vraća 1; inače, vrednost u lokaciji slot ostaje nepromenjena i vraća se 0.

Na jeziku C napisati funkciju

int inc (int& sharedVar);

koja inkrementira deljenu celobrojnu promenljivu na koju ukazuje dati argument sharedVar korišćenjem optimističkog međusobnog isključenja. Ako je inkrementiranje uspelo, vraća se 1, a ukoliko je detektovan konflikt, vraća se 0. Ova funkcija može da se poziva i uporedo izvršava iz konkurentnih niti.

Rešenje:

int inc (int& sharedVar) {
 int myCopy = \*sharedVar;
 int newVal = myCopy+1;
 return cmp\_xchg(sharedVar,myCopy,newVal);
}

(b)(5) Dat je sledeći program koji koristi koncept niti (klasa Thread) koje se mogu kreirati nad datom globalnom funkcijom, pri čemu se ta funkcija poziva u kontekstu novokreirane uporedne niti, kao što je pokazano u školskom jezgru datom na predavanjima. Zamisao programera je da ovaj program na kraju ispiše zbir celih brojeva od 0 do N-1.

#include „kernel.h“
#include <stdio.h>
#define N 10
int a[N];
int sum = 0;
Semaphore end(0);
void writer () { for (int i=0; i<N; i++) a[i]=i; }
void adder () { for (int i=0; i<N; i++) sum+=a[i]; end.signal(); }

void main () {
 for (int i=0; i<N; i++) a[i]=0;
 Thread\* t1 = new Thread(writer); t1->start();
 Thread\* t2 = new Thread(adder); t2->start();
 end.wait();
 printf(„Sum: %d\n“,sum);
}

Precizno objasniti šta je problem sa ovim programom.

To što su dve uporedne niti, prva nad funkcijom writer, druga nad funkcijom adder, redom pokrenute baš u tom poretku, nikako ne znači da će one i započeti izvršavanje u tom poretku, a posebno ne da će prva završiti upis u neki element niza pre nego što druga pročita taj isti element. Ove dve niti teku uporedo i sasvim nezavisno, bez ikakve sinhronizacije. Zbog toga je način pristupa do deljenih promenljivih (elemenata niza a) potpuno neodređen i proizvoljan, što znači da nikako nije garantovano da će ovaj program ispisati željeni rezultat (može, ali ne mora uvek), jer je za to potrebno da nit nad funkcijom writer završi upis nekog elementa niza pre nego što nit nad funkcijom adder pristupa istom tom elementu niza. Problem je u *utrkivanju* ovih niti (engl. *race condition*) zbog nedostatka sinhronizacije.

c)(5) Koliko ukupno procesa nastane (uključujući i početni) kada se nad sledećim programom kreira proces, pod uslovom da su svi sistemski pozivi uspešni?

const unsigned int N = ...; // N>0
void main () {
 for (unsigned int i=0; i<N; i++)
 if (fork()!=0) break;
}

Odgovor:Ukupno nastane N+1 procesa (računajući i početni).

c)(5) Na jeziku C ili C++ napisati program cli za Unix koji ciklično radi sledeće. Najpre na standardni izlaz (npr. konzolu) ispisuje kurzor kao znak spremnosti za učitavanje (znak „>“), a potom učitava jednu liniju teksta sa standardnog ulaza, pri čemu ta linija predstavlja komandnu liniju u kojoj je prvi argument naziv programa, a ostali su parametri programa. Kada učita liniju, program cli pokreće novi proces nad programom zadatim u učitanoj komandnoj liniji, a kada se taj proces završi, ponovo ispisuje kurzor i prihvata novu komandnu liniju. Kada učita komandu „exit“, program cli se završava. Pretpostaviti da sistemski poziv execlp() prihvata jedan niz znakova u kome su i naziv programa i njegovi parametri.

Rešenje:

#include <iostream.h>
#include <string.h>

const int maxCLSize = 512; // command line maximal size
char cl[maxCLSize];

int main () {
 while (1) {
 cout<<”\n>”;
 cin.getline(cl,maxCLSize);
 if (strcmp(cl,”exit”)==0) break;
 int pid = fork();
 if (pid<0)
 cout<<”System error: Unable to create process.\n”;
 else if (pid==0) {
 if (execlp(cl)<0)
 cout<<”System error: Program ”<<cl<<” not found or unable to run.\n”;
 }
 else if (pid>0)
 wait(0); // Wait child
 }
 exit(0);
}

c)(5) Koliko ukupno procesa nastane (uključujući i početni) kada se nad sledećim programom kreira proces, pod uslovom da su svi sistemski pozivi uspešni?

const unsigned int N = ...; // N>0
void main () {
 for (unsigned int i=0; i<N; i++)
 if (fork()==0) break;
}

Ukupno nastane N+1 proces.

(c)(5) Potrebno je uraditi transformaciju i prenos neke jako velike količine ličnih podataka o građanima jedne države iz jedne baze podataka u drugu. Podaci vezani za svakog građanina su disjunktni, tj. podaci o jednom građaninu ne referenciraju podatke o drugim građanima. Za tu potrebu napravljen je program koji je u stanju da za dati ulazni podskup podataka iz izvorišne baze (specifikacija načina dolaska do tog podskupa u izvornoj bazi je ulazni parametar tog programa) uzima po N (N je još jedan parametar ovog progama) zapisa o građanima, transformiše ih, i da tako transformisane podatke upiše u jedan izlazni fajl, potom uzima N narednih zapisa i prebacuje ih u novi fajl itd, sve dok ne prebaci ceo svoj podskup. Sa druge strane, postoji program koji je u stanju da uzima jedan po jedan ulazni fajl iz zadatog direktorijuma i da podatke iz fajla učitava u odredišnu bazu. Precizno objasnite kako biste ceo ovaj posao transformacije i prenosa podataka organizovali što efikasnije.

ODGOVOR:

Može se kreirati M parova procesa koji se sastoje od jednog čitaoca izvorišne baze i jednog punioca odredišne baze. Ova dva procesa u paru razmenjuju fajlove preko zajednikog „bafera“, što može prosto biti jedan direktorijum. Dakle, čitalac prebacuje podatke iz izvorne baze u fajlove u tom direktorijumu, a punilac uzima iz tog direktorijuma jedan po jedan fajl i puni odredišnu bazu. Za svaki od tih parova definiše se podskup u izvornoj bazi koji je disjunktan sa ostalim podskupovima, tako da svih M podskupova pokrivaju ceo ulazni skup (tj. predstavljaju particionisanje ulaznog skupa), i direktorijum koji služi kao bafer, a onda se pokrene takvih M parova koji rade paralelno, na istom ili na različitim računarima.

(a)(10) U jezgru nekog multiprocesorskog operativnog sistema raspoređivanje niti (*threads*) vrši se iz jednog zajedničkog reda spremnih niti, implementiranog kao dvostruko ulančana lista. Raspoređivanje se vrši po FIFO principu, a red sadrži niti za sve procesore, tako da se na procesoru za koji se vrši izbor uzima prva spremna nit iz tog zajedničkog reda. Red spremnih niti implementiran je klasom Schedulerčija je definicija:

class Scheduler {
public:
 static void put (Thread\*);
 static Thread\* get();
private:
static Thread \*head, \*tail;
 static int lck; // Lock for mutual exclusion
};

Procesori su svi isti, 32-bitni, dvoadresni, sa RISC *load-store* arhitekturom. Registri opšte namene, označeni sa R0..R31 su 32-bitni. Postoji instrukcija SWAP R*i*,[R*j*] koja vrši atomičnu zamenu sadržaja datog registra R*i*i memorijske lokacije na adresi koja je u R*j*.

*i*)(5) Na asembleru datog procesora implementirati funkcije:

void lock (int\* ptrLock);
void unlock (int\* ptrLock);

koje obezbeđuju međusobno isključenje pristupa kritičnoj sekciji od strane programa koji se izvršavaju na više procesora, pri čemu je argument pokazivač na lokaciju koja služi kao „ključ“ (barijera) za pristup kritičnoj sekciji. Nije potrebno maskirati i demaskirati prekide, pošto se to radi pri ulasku u sistemski režim rada.

void lock (int\*) {
 asm {
 push r0
 push r1
 clr r0
 load r1,sp[-3] //sp points to the last used loc, stack grows upwards
wait: // busy waiting loop -- spin lock
 swap r0,[r1]
 and r0,r0
 jz wait
 pop r1
 pop r0
 }
}

void lock (int\*) {
 asm {
 push r0
 push r1
 load r0,#1
 load r1,sp[-3] //sp points to the last used loc, stack grows upwards
 store r0,[r1]
 pop r1
 pop r0
 }
}

*ii*)(5) Implementirati klasu Scheduler.

int Scheduler::lck=1;
Thread \*Scheduler::head=0, \*Scheduler::tail=0;

void Scheduler::put (Thread\* t) {
 if (t==0) return; // Exception!
 lock(&lck);
 if (tail) tail->next=t;
 t->prev=tail;
 t->next=0;
 tail=t;
 if (head==0) head=tail;
 unlock(&lck);
}

Trhead\* Scheduler::get () {
 lock(&lck);
 Thread\* t=head;
 if (head) head=head->next;
 if (head) head->prev=0;
 if (head==0) tail=0;
 if (t) t->prev=t->next=0;
 unlock(&lck);
 return t;
}

c)(5) Korišćenjem sistemskog poziva fork(), napisati C program koji, kada se nad njim kreira proces, napravi jednog svog potomka i završi se, ovaj novi proces napravi jednog svog potomka i završi se, itd., tako da ukupno bude kreirano N procesa (uključujući i početni, N>1), pod uslovom da su svi sistemski pozivi uspešni.

Rešenje:

const unsigned int N = ...; // N>1
void main () {
 for (unsigned int i=0; i<N-1; i++)
 if (fork()!=0) break;
}

Upravljanje memorijom

* Precizno objasniti šta su sličnosti, a šta razlike između dinamičkog učitavanja (*dynamic loading*) i preklopa (*overlays*).

Osnovna sličnost je što se kod oba pristupa učitavanje modula programa odlaže za vreme izvršavanja – modul se učitava kada mu se pristupi a on nije u memoriji. Druga sličnost je što je ceo mehanizam inkorporiran u biblioteke koje se statički povezuju u korisnički program i što prevodilac ima zadatak podele programa na module, a OS nema ulogu (osim same usluge učitavanja dela programa). Osnovna razlika je što se kod preklopa modul može učitati na mesto drugog modula (preklopiti ga), dok se kod dinamičkog učitavanja modul učitava na svoje mesto i time se deo memorije koji proces zauzima povećava

* Virtuelni adresni prostor računara je 16 GB, adresibilna jedinica je 32-bitna reč, a virtuelni adresni prostor je organizovan segmentno-stranično. Svaki proces može imati najviše 64 segmenta, a segment najviše 1024 stranice. Fizički adresni prostor je veličine 4 GB. Pristup segmentima se kontroliše pomoću tri bita zaštite koji se označavaju sa R (*read*), W (*write*) i E (*execute*). Adresa bloka na disku je veličine 24 bita. Tabele preslikavanja stranica (PMT) se smeštaju u poseban deo adresnog prostora kernela i indeksiraju se indeksom u nizu tih tabela kojih najviše može biti 4096. Koliko najmanje bita mora da ima deskriptor segmenta, a koliko deskriptor stranice?

Odgovor: Deskriptor segmenta mora imati najmanje 25 bita, a deskriptor stranice najmanje 25 bita.

Veličina virtuelnog adresnog prostora je 16GB/4B = 4G = 232 reči, pa je virtuelna adresa veličine 32 bita.

Veličina fizičkog adresnog prostora je 4GB/4B = 1G = 230 reči, pa je fizička adresa veličine 30 bita.

Struktura virtuelne adrese: *Segment*(6):*Page*(10):*Offset*(16)

Struktura fizičke adrese: *Frame*(14):*Offset*(16)

Struktura deskriptora segmenta: *PMTIndex*(12):*Limit*(10):*RWE*(3).

Struktura deskriptora stranice: *InMemory*(1):*Frame*(14)|*DiskBlk*(24)

* U nekom integrisanom okruženju za razvoj programa (*Integrated Development Environment*, IDE) za jezik C/C++ postoji pojam „projekta“ koji prosto sadrži spisak fajlova. Ovaj pojam ima samo sledeće značenje: kada se pokrene operacija „Build Project“ prevodilac prevodi sve fajlove (nezavisno od njihovog tipa) sa ovog spiska, a potom linker povezuje sve te prevedene fajlove. Precizno objasniti šta će se dogoditi ako se u jednom projektu (tj. u njegovom spisku fajlova) nađu i .h fajlovi i .cpp fajlovi jednog softverskog paketa: da li će okruženje prijaviti grešku i ako hoće, ko će prijaviti grešku – prevodilac ili linker i kakvu grešku će prijaviti? Pretpostavlja da je sav izvorni kod sintaksno i semantički ispravan.

Prevodilac neće prijaviti grešku, jer on nezavisno prevodi svaki fajl, a prilikom prevođenja .h fajlova neće pronaći nikakvu grešku, s obzirom da su u njima korektne deklaracije. Međutim, prevodilac će i za takve .h fajlove generisati .obj fajlove u čijim će se zaglavljima naći simboli koji se izvoze za sve deklaracije koje imaju eksterno vezivanje u tim fajlovima (verovatno većina njih). Isto će važiti i za .c/.cpp fajlove, koji po pravilu uključuju .h fajlove i zbog toga sadrže deklaracije istih simbola, pa će zbog toga linker prijaviti grešku tipa „višestruko definisani simbol“ jer će naići na iste simbole izvežene u zaglavljima više .obj fajlova koje povezuje.

* Neki računar koristi straničnu organizaciju memorije, bez mehanizma zamene stranica, pa su uvek sve stranice nekog procesa u memoriji. Podržana je i zaštita pristupa stranicama pomoću tri bita *rwe* u deskriptoru stranice. Virtuelni adresni prostor je veličine  16 MB, adresibilna jedinica je bajt, veličina stranice je 512 KB. Fizički adresni prostor je veličine 4 GB. Kolika je veličina tabele preslikavanja stranica PMT?

VA: Page(5):Offset(19)

PA: Frame(13):Offset(19).

Deskriptor stranice: RWE(3):Frame(13), pa zbog toga deskriptor (ulaz u PMT) zauzima 2B.

PMT ima 25 = 32 ulaza, pa zauzima 64 B.

* U računaru opisanom pod b) za potrebe operativnog sistema odvojen je prostor iste veličine kao i prostor za jedan proces. Koliko se korisničkih procesa može maksimalno kreirati? Koliki deo memorijskog prostora operativnog sistema zauzimaju tabele preslikavanja (PMT) za korisničke procese?

Ukupno ima 213 okvira, a svaki proces (kao i OS) zauzima 25 okvira, pa postoji prostor za 28 takvih, odnosno 255 procesa i OS.

* Precizno objasniti šta će se dogoditi ako se puna definicija C funkcije f (sa telom) napiše u fajlu-zaglavlju (.h) i taj fajl uključi (pomoću #include) u više .c fajlova koji se prevode i povezuju u isti program? Ko će prijaviti grešku (prevodilac ili linker), kakvu grešku i zašto?

Prevodilac neće prijaviti grešku, jer on nezavisno prevodi svaki .c fajl, a u njemu će videti definiciju funkcije f isto kao da je ona napisana u .c fajlu (prevodilac, posle pretprocesiranja, ne vidi razliku u kodu napisanom u .h i .c fajlu). Međutim, pošto je funkcija f definisana u svakom takvom fajlu, prevodilac će simbol f tretirati kao simbol sa „spoljnjim povezivanjem“, što znači da će svaki .obj fajl nastao iz ovakvog .c fajla imati u svom zaglavlju simbol f kao simbol koji se izvozi. Prilikom povezivanja ovih .obj fajlova, linker će prijaviti grešku tipa „višestruko definisani simbol“ jer će isti simbol f naći kao izvežen u zaglavljima više .obj fajlova koje povezuje.

* (a)(10) Zašto preklopi (*overlays*) ne mogu da se koriste ako program ima više niti koje obezbeđuje operativni sistem? Precizno objasniti.

Preklopi (*overlays*) se realizuju bez podrške i bilo kakve interakcije sa operativnim sistemom (osim same podrške dinamičkom učitavanju), tako što na ulasku u određeni potprogram koji je alociran u neki preklop, prevodilac generiše kod koji proverava da li je taj kod učitan u preklop i učitava ga ako nije. Ako program koristi niti koje obezbeđuje OS, sistem održavanja preklopa (u odgovornosti programa) i promena konteksta niti (u odgovornosti OS‑a) nemaju interakcije i „ne znaju jedan za drugog“. Zato se može dogoditi sledeći konfliktni scenario:

* + jedna nit uđe u izvršavanje nekog potprograma P koji pripada jednom preklopu, proveri i po potrebi učita kod za taj potprogram u preklop X
	+ OS izvrši promenu konteksta niti i pokrene drugu nit; ova druga nit ulazi u potprogram Q koji pripada drugom preklopu, a koji nije učitan
	+ sistem preklopa učitava preklop kome pripada Q na isto mesto X
	+ ponovo dođe do preuzimanja i prva nit nastavi izvršavanje od adrese na kojoj očekuje kod za potprogram P, ali se tu sada nalazi kod za Q.
* (a)(10) Neki C program sastoji se samo iz dva fajla sa datim sadržajem:

// a.c // b.c
float base = 2.0; float log(float);
float log(float); extern float base;
float ln(float);

float log(float x) { float f(float x) {
 return ln(x)/ln(base); return log(x\*base)/log(base);
} }

Odgovoriti na sledeća pitanja:

1) Koliko nerazrešenih adresnih polja instrukcija prevodilac ostavlja u fajlu a.obj? 2

2) Koliko nerazrešenih adresnih polja instrukcija prevodilac ostavlja u fajlu b.obj?

4 ili 3, u zavisnosti da li je optimizovao dohvatanje vrednosti base u neki registar

3) Koje simbole izvozi fajl a.obj? base, log

4) Koje simbole uvozi fajl a.obj? ln

5) Koje simbole izvozi fajl b.obj? f

6) Koje simbole uvozi fajl b.obj? base, log

7) Koliko grešaka prijavljuje linker prilikom povezivanja a.obj i b.obj u p.exe? 1

a)(10) U nekom sistemu sa straničnom organizacijom virtuelne memorije u datom trenutku raspored stranica tri procesa po okvirima u fizičkoj memoriji izgleda redom ovako (slova A, B i C označavaju procese kojima pripadaju stranice u okvirima, brojevi označavaju brojeve stranica tih procesa, K označava okvir koji pripada jezgru OS-a, a DLL označava okvir čiji je sadržaj DLL koga koriste sva tri procesa A, B i C):

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Okvir | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Sadržaj | K | B2 | C1 | A1 | C0 | DLL | B3 | K | K |

U virtuelnom adresnom prostoru procesa A dati DLL se nalazi na stranici broj 0, a u prostorima procesa B i C na stranici broj 4. Napisati kako izgledaju tabele preslikavanja stranica za sva tri procesa A, B i C (u tabelu upisati brojeve okvira i T ili F kao oznaku da li je stranica u memoriji ili ne, za prvih 6 stranica).

Odgovor:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Strana: | 0 | 1 | 2 | 3 | 4 | 5 |
| Proces A | T5 | T3 | F | F | F | F |
| Proces B | F | F | T1 | T6 | T5 | F |
| Proces C | T4 | T2 | F | F | T5 | F |

b)(5) Precizno objasniti zašto kompakcija slobodnih fragmenata kod kontinualne alokacije memorije nije moguća ukoliko ne postoji hardverska podrška za dinamičku relokaciju u vidu registra za relokaciju i dinamičkog preslikavanja adresa?

Kompakcija slobodnog prostora podrazumeva premeštanje kompletnog sadržaja memorijskog prostora nekog procesa sa jednog mesta u fizičkoj memoriji na drugo, dakle kopiranje tog sadržaja. U opštem slučaju, neke vrednosti u tom prostoru predstavljaju podatke koji ne menjaju svoju semantiku premeštanjem na drugo mesto, dok neke druge vrednosti zapravo predstavljaju adrese (pokazivače) na druge vrednosti (u strukturama podataka koje je korisnički program organizovao). Program koji vrši kompakciju i premeštanje u opštem slučaju nema načina da zna koje od ovih vrednosti predstavljaju jednu, a koje drugu kategoriju, ukoliko takve informacije nisu eksplicitno date na određeni način (a po pravilu nisu), pa zbog toga ne može da izvrši promene ovih drugih. Ako ne postoji hardverska podrška za dinamičku relokaciju i preslikavanje adresa, onda su te vrednosti apsolutne adrese u fizičkoj memoriji, pa je prilikom relokacije potrebna promena njihovih vrednosti, koja nije moguća, kako je objašnjeno.

c)(5) Kako i na osnovu čega onda *loader* vrši (statičku) relokaciju programa prilikom kreiranja procesa?

Odgovor:

*Loader* poseduje informacije koje su upisane u neku strukturu (npr. zaglavlje) prevedenog programa koji precizno navodi mesta (adresnih polja instrukcija i statički inicijalizovanih pokazivača) u programskom fajlu koja treba promeniti, odnosno postaviti na odgovarajuće apsolutne adrese prilikom kreiranja procesa. Treba uočiti da su te informacije vezane samo za statičku, inicijalnu strukturu programa (adresna polja instrukcija i statički inicijalizovani pokazivači). Problem zašto se isti princip ne može iskoristiti kod kompakcije jeste u tome što su mnogi pokazivači u programu koji se izvršava dinamički kreirani (dinamičke strukture, strukture na steku itd.) i ove informacije se po pravilu ne obezbeđuju od strane programa tokom izvršavanja.

a)(5) U nekom trenutku linker zaključuje da se simbol na koga je upravo naišao u zaglavlju nekog od ulaznih .obj fajlova već nalazi u njegovoj tabeli simbola. Da li će ovo prijaviti kao grešku?

Odgovor:

Zavisi od toga šta u tom trenutku linker zapravo radi i šta znači ta provera, odnosno da li se dati simbol na koga je naišao uvozi ili izvozi. Ako se radi o simbolu koji dati fajl izvozi, linker će prijaviti grešku, jer je simbol već definisan u nekom drugom fajlu. Ako se radi o simbolu koji neki fajl uvozi, onda je ovo regularna situacija.

c)(5) Virtuelna memorija nekog sistema organizovana je stranično. Virtuelni adresni prostor je veličine 16GB, odnosno 16M stranica, a adresibilna jedinica je 32-bitna reč. Fizički adresni prostor je veličine 64MB. Memorijski prostor odvojen za jezgro operativnog sistema zauzima 4K okvira. Koliko okvira je raspoloživo za korisničke procese ukoliko računar ima instaliran maksimalan kapacitet operativne memorije?

Virtuelni adresni prostor je veličine 16GB, odnosno 4GW = 232W (4 giga 32-bitnih reči), odakle sledi da je virtuelna adresa veličine 32 bita. Kako on ima 16M = 224 stranica, struktura virtuelne adrese je: Page(24):Offset(8).

Fizički adresni prostor je veličine 64MB, odnosno 16MW = 224W (16 mega 32-bitnih reči), pa je struktura fizičke adrese: Frame(16):Offset(8). Odatle sledi da fizički adresi prostor ima 216 = 64K okvira.

Kako jezgro zauzima 4K okvira, za korisničke procese ostaje 60K okvira.

a)(5) Da li je dozvoljeno da dva fajla sa izvornim kodom međusobno unakrsno referišu simbole koji su u njima definisani (npr. fajl a.c definiše x a referiše y, dok fajl b.c definiše ya referiše x)? Ako nije, objasniti zašto? Ako jeste, objasniti kako linker razrešava ovo unakrsno referisanje simbola definisanih u drugom fajlu.

Odgovor:

Dozvoljeno je. Linker ovo razrešava u dva prolaza: u prvom samo sakuplja informacije o svim simbolima koje moduli definišu i izvoze, dok tek u drugom, kada su sve adrese definisanih simbola poznate, razrešava reference na simbole. Zbog toga nije bitno kojim redosledom obrađuje fajlove, pa kružna referisanja nisu problem.

b)(5) Da li kod stranične organizacije virtuelne memorije ima smisla hardverski vršiti proveru prekoračenja granice opsega dozvoljenih adresa unutar stranice radi zaštite od nedozvoljenog pristupa fizičkim adresama koje pripadaju drugim procesima? Objasniti zašto. Ako nema, objasniti čemu bi takva kontrola prekoračenja opsega adresa unutar stranice mogla da služi.

Odgovor:

 Nema, jer se unutar jednog fizičkog okvira nalaze uvek samo adrese koje pripadaju virtuelnom prostoru jedne stranice jednog procesa – nije moguće da se delovi adresnog prostora više procesa nalaze unutar istog okvira u isto vreme. Ovakva zaštita ima smisla radi provere ispravnosti rada samog programa, u cilju sprečavanja prekoračenja opsega adresa koje program legalno može da koristi. Ovakvo prekoračenje može samo da znači da program neispravno radi, ali ne može da naudi tuđim adresnim prostorima.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 |  | 1 1 |  | 1 0 |
| 1 |  | 1 1 |  | 1 1 |
| 0 |  | 1 0 |  | 1 1 |
| 0 |  | 1 0 |  | 0 |

a)(5) Ako neki C/C++ program ne sadrži definiciju funkcije main(), linker će prijaviti grešku da simbol main nigde nije definisan. Precizno objasniti kako nastaje ta greška kada se funkcija main() nigde ne referiše u korisničkom kodu (niko je ne poziva)?

Odgovor:

 Kada se povezuje C/C++ program, lista .obj fajlova mora da sadrži i obavezni modul u kome je sam glavni program od koga počinje izvršavanje. Taj modul ne sadrži definiciju funkcije main() niti bilo kakav korisnički kod, već je standardni, generički modul za dati operativni sistem i C/C++ programe i sadrži kod koji od operativnog sistema preuzima argumente poziva programa, poziva funkciju main() koja mora biti definisana u korisničkom kodu, pri tom joj prosleđuje te argumente (kroz parametre argc i argv), a posle povratka iz ove funkcije poziva sistemski poziv za gašenje procesa (sa ili bez greške, u zavisnosti od toga šta je funkcija main() vratila). Prema tome, ovaj modul poziva funkciju main(), pa je zato i referiše kao spoljni simbol koga uvozi, što znači da će linker prijaviti grešku ako taj simbol nijedan drugi modul nije definisao.

c)(5) Vreme pristupa fizičkoj memoriji je 60 ns, vreme pristupa TLB-u je 10 ns, a procenat pogotka u TLB je 95%. Virtuelna memorija je organizovana stranično, sa tabelama preslikavanja stranica organizovanim u dva nivoa. Koliko je efektivno vreme pristupa operativnoj memoriji?

Odgovor: 76 ns.

Račun: *t*eff = 95%⋅(10ns + 60 ns) + 5%⋅(10ns + 3⋅60 ns).

a)(10) Neki program organizovan sa preklopima (*overlays*) ima zajednički (statički) deo veličine 80 KB u kome se nalaze *overlay driver*, globalni podaci i potprogrami koji su uvek u memoriji tokom izvršavanja, kao i tri preklopa, ovde označeni sa A, B i C:

* A u koji se učitavaju i preklapaju moduli A1 (45 KB), A2 (70 KB) i A3(20 KB);
* B u koji se učitavaju i preklapaju moduli B1 (115 KB) i B2 (250 KB);
* C u koji se učitavaju i preklapaju moduli C1 (90 KB), C2 (510 KB), C3 (200 KB) i C4 (20 KB).

Koliko ukupno memorije u toku izvršavanja zauzima ovaj program?

Obrazloženje-račun: 80KB + max(45,70,20)KB + max(115,250)KB + max(90,510,200,20)KB = 80KB + 70KB + 250KB + 510KB = 910KB.

b) Virtuelna memorija nekog sistema organizovana je stranično. Virtuelni adresni prostor je veličine 16 EB (eksabajt, kilo-mega-giga-tera-peta-eksa), fizički adresni prostor je veličine 1 TB (terabajt), adresibilna jedinica je bajt, a stranica je veličine 16 MB. Svaki ulaz tabele preslikavanja stranica (PMT), koja je organizovana u samo jednom nivou, sadrži samo broj okvira. Hardver za preslikavanje adresa poseduje podršku za proveru granice dela virtuelnog prostora koji proces koristi pomoću registra granice (*page-table length register*, PTLR).

*i*)(5) Kolika je veličina PMT procesa koji je deklarisao da koristi samo prvih 2 GB svog adresnog prostora?

 Račun: VA: Page(40):Offset(24); PA: Frame(16):Offset(24);

PMT treba da pokrije 2 GB = 231 B, što zauzima 231/224 = 27 = 128 stranica.

PMT ovog procesa ima 27 ulaza po 16 bita, odnosno 256 B.

1. Kolika jevrednost PTLR ovog procesa?

Odgovor: 27 - 1 = 127, pod pretpostavkom da je logika za kontrolu granice realizovana tako da PTLR čuva najveću dozvoljenu vrednost broja stranice u generisanoj virtuelnoj adresi.

a)(5) Da li je bitan redosled kojim linker prolazi kroz fajlove koji su mu na spisku za povezivanje? Obrazložiti.

 Redosled nije bitan, jer linker svakako radi u dva prolaza: u prvom prolazu prođe kroz sve .obj fajlove i formira tabelu simbola, pa tek u drugom prolazu razrešava adresna polja. Zbog toga fajlovi mogu unakrsno referisati simbole i mogu biti nabrojani u bilo kom poretku.

c)(5) Virtuelna memorija nekog sistema organizovana je stranično. Virtuelni adresni prostor je veličine 16EB (eksabajt, kilo-mega-giga-tera-peta-eksa), fizički adresni prostor je veličine 1TB (terabajt), adresibilna jedinica je bajt, a stranica je veličine 16MB. Kolika bi bila veličina cele PMT organizovane u samo jednom nivou, ako bi njen ulaz čuvao samo broj okvira?

Račun: VA: Page(40):Offset(24); PA: Frame(16):Offset(24); PMT bi imala 240 ulaza po 16 bita, odnosno 2 TB

a)(5) U jeziku C++ moguće je definisati više funkcija sa istim imenom, ali različitim tipovima argumenata (preklapanje imena funkcija, *function overloading*). Iako linker za C++ programe jeste isti kao i za C programe u kojima preklapanje imena funkcija nije moguće, i iako linker ne poznaje pojam tipova i argumenata iz izvornog jezika, linker neće prijaviti grešku ako povezuje dva fajla jednog C++ programa u kojima su definisane dve različite funkcije sa istim imenom. Precizno objasniti zašto i kako.

Prevodilac generiše naziv simbola koji predstavlja adresu definicije (koda) funkcije u nekom fajlu tako što u naziv enkoduje i informacije o tipovima argumenata. Na primer, odgovarajućim slovnim oznakama i delimiterima u naziv simbola uvrsti i tipove argumenata. Zbog toga, bez obzira što se funkcije u izvornom kodu zovu isto, linker uopšte ne vidi iste simbole u .obj fajlovima, već se oni razlikuju i linker ih tretira potpuno razdvojeno.

c)(5) **Neki računar podržava segmentno-stranični mehanizam virtuelne memorije, pri čemu je virtuelna adresa 16-bitna, fizički adresni prostor je veličine 8GB, a adresibilna jedinica je 16‑bitna reč. Stranica je veličine 512B. Maksimalan broj segmenata u virtuelnom adresnom prostoru je 4. Prikazati logičku strukturu virtuelne i fizičke adrese i navesti širinu svakog polja.**

**VA: Segment(2):Page(6):Offset(8); PA: Frame(24):Offset(8). //**zašto je Page(6) ?

a)(10) Kada prevodi kod potprograma u kome se pristupa statičkom podatku, da li prevodilac može da upotrebi apsolutni način adresiranja (unutar adresnog prostora procesa) u instrukciji pristupa tom podatku za slučaj da je taj statički podatak definisan:

(*i*)(5) van jedinice koju trenutno prevodi;

(*ii*)(5) u istoj jedinici koju trenutno prevodi, sa ciljem da se taj potprogram može deliti između više procesa sa virtuelnim adresnim prostorima kao deo deljene biblioteke.

Obrazložiti odgovore.

Odgovori:

*i*)(5) Da, s tim što će adresno polje te instrukcije razrešiti linker prilikom povezivanja u izvršni program.

*ii*)(5) Ne, pošto u adresnim prostorima različitih procesa u kojima će se pozivati dati potprogram taj statički podatak može da ima različite adrese; u ovom slučaju prevodilac koristi relativno adresiranje.

a)(10) Ako je prilikom pokretanja linkera zaboravljena da se navede biblioteka koju treba povezati sa ostatkom programa, kakvu vrstu greške očekujete od linkera? Obrazložiti.

Odgovor:

Grešku tipa „simbol koji je referisan nije nigde definisan“. Pošto je biblioteka potrebna, znači da je neki drugi modul programa referisao simbole iz te biblioteke, a simbol nije definisan pošto linker nije obradio tabele simbola te biblioteke, pa će doći do navedene greške.

b)(10) U nekom računarskom sistemu i virtuelna i fizička adresa su 32-bitne, a adresibilna jedinica je bajt. Jedna mašinska reč je veličine 32 bita (to je i veličina tipa int). Virtuelna memorija je organizovana stranično, a tabela preslikavanja stranica (PMT) je organizovana u dva nivoa. Struktura virtuelne adrese je sledeća: P1(8 bita):P2(8 bita):D(16 bita), gde je P1 broj ulaza u tabeli prvog, P2 drugog nivoa, a D pomeraj unutar stranice. Svaki ulaz u tabelama i prvog i drugog nivoa ima 32-bitni sadržaj koji se tumači na sledeći način:

|  |  |  |
| --- | --- | --- |
| Vrednost | U tabeli prvog nivoa | U tabeli drugog nivoa |
| = 0 | Tabela drugog nivoa nije alocirana | Stranica nije u memoriji |
| = -1 (sve jedinice) | Opseg adresa pokriven ovim ulazom nije dozvoljen za pristup | Stranica nije dozvoljena za pristup |
| ostalo | Fizička adresa tabele drugog nivoa | Fizička adresa okvira |

Sistemskim pozivom:

void mem\_reg(int start\_addr, int num\_of\_pages)

korisnički proces „registruje“ opseg virtuelne memorije počev od adrese date prvim argumentom (adresa mora biti poravnata na početak stranice) i veličine date drugim argumentom (izraženo u broju stranica) kao dozvoljen za pristup. Ovaj poziv ne alocira stranice u fizičkoj memoriji, već se stranice alociraju i dohvataju na zahtev, kada im se prvi put pristupi (*demand paging*). I tabele drugog nivoa alociraju se tek kada im se prvi put pristupa. Dat je sledeći program:

void main () {
 int start\_addr = 0x10800000;
mem\_reg(start\_addr,512);
 int\* p = (int\*)start\_addr;
 for (int i=0; i<0x400000; i++,p++) int dummy = \*p;
}

Ceo kod programa zauzima jednu, prvu stranicu virtuelnog prostora, a njegov stek još jednu, narednu stranicu. Jezgro operativnog sistema smešta tebele preslikavanja u posebno alocirane okvire memorijskog prostora i pakuje ih kompaktno, više njih u isti okvir, onoliko koliko može da ih stane. Koliko fizičke memorije (izražene u bajtovima) zauzima proces kreiran nad ovim programom na samom kraju svog izvršavanja, računajući i prostor alociran za same stranice virtuelnog prostora procesa, kao i onaj alociran za tabele preslikavanja?

Odgovor: 16516 KB = (256+2)⋅64 KB + 4 KB

Račun:

Stranica je veličine 216 B = 64 KB.

Tabela prvog i svaka tabela drugog nivoa ima po 28 = 256 ulaza, svaki veličine 4 B, pa zauzima po 1 KB.

Jedna tabela drugog nivoa pokriva 256 stranica, odnosno 28⋅216 = 224 adresa (tj. 16 MB).

Dati program pristupa sledećim delovima svog virtuelnog adresnog prostora:

- prvoj stranici sa kodom;

- drugoj stranici sa stekom (za smeštanje lokalnih automatskih podataka i traga poziva potprograma);

- podacima preko pokazivača p u sledećem delu adresnog prostora: počev od adrese 10800000h, pa sledećih 400000h⋅4 = 1000000h = 224 adresa; to je pristup do ukupno 28 = 256 susednih stranica koje čine gornju polovinu adresnog prostora koga pokriva ulaz 10h u tabeli prvog nivoa i donju polovinu adresnog prostora koga pokriva sledeći ulaz 11h tabele prvog nivoa.

Zbog toga su u memoriji alocirane sledeće strukture:

- Tabela prvog nivoa i tabele drugog nivoa koje odgovaraju ulazima 0, 10h i 11h tabele prvog nivoa; one ukupno zauzimaju 4⋅1 KB = 4 KB.

- Prve dve stranice (za kod i stek) i 256 stranica za podatke, što je ukupno 258 stranica, odnosno (256+2)⋅64 KB.

Odatle sledi gore dati rezultat.

b)(5) U nekom sistemu sa straničnom organizacijom virtuelne memorije hardver za preslikavanje adresa proverava da li je broj stranice u generisanoj virtuelnoj adresi veći od vrednosti posebnog registra dužine tabele stranica (*page-table length register*, PLTR) i generiše izuzetak ukoliko jeste. Ko, kada i koju vrednost upisuje u registar PLTR?

Vrednost u PLTR upisuje kod jezgra OS prilikom promene konteksta, uzimajući je iz strukture PCB procesa koji dobija procesor.

c)(5) U nekom sistemu sa straničnom organizacijom virtuelne memorije i virtuelna i fizička adresa su 64-bitne, adresibilna jedinica je bajt, a virtuelni adresni prostor procesa ima 256T (tera) stranica. Koliko fizičke memorije treba alocirati procesu da bi se u njega mogle smestiti 4 stranice?

Odgovor: 256KB

Račun:

256T = 28⋅240

Za adresiranje 256T stranica treba 8+40=48 bita. Zato virtuelna adresa ima strukturu: Page(48):Offset(16), pa je jedna stranica veličine 216B=64KB. Za smeštaj 4 stranice potrebno je 256KB.

a)(5) Da li kôd deljene biblioteke sa dinamičkim vezivanjem (DLL) može da koristi apsolutno adresiranje memorije, npr. kod skokova, u višeprocesnom sistemu sa virtuelnom memorijom? Obrazložiti.

Ne sme, pošto se taj kod može nalaziti na različitim virtuelnim adresama različitih procesa koji je koriste (dele). Ako bi to bio slučaj, takva jedna apsolutna virtuelna adresa unutar ovog koda bi se preslikavala u sasvim različite fizičke adrese različitih procesa. Umesto apsolutnog, kod DLL-a koristi relativno adresiranje memorije, ili pozive operacija iz drugih DLL‑ova.

b)(5) **U nekom sistemu virtuelni adresni prostor je veličine 4 GB, adresibilna jedinica je 32‑bitna reč, a fizička adresa je veličine 32 bita. Ceo virtuelni adresni prostor ima 64 K stranica, a tabela preslikavanja stranica (PMT) je u jednom nivou i u svakom ulazu čuva samo broj okvira u koji se stranica preslikava. Koliko 32-bitnih reči zauzima PMT?**

**Odgovor: 64K 32-bitnih reči.**

**Pošto virtuelni prostor ima 64K = 216 stranica, toliko ulaza ima i PMT.**

**Svaki ulaz je veličine jedne reči, pa PMT zauzima 64 K reči. //**zar nije 18bita jedan ulaz u PMT?

a)(10) Posmatra se prevodilac koji prevodi izvorni kod i generiše mašinski kod za deljenu biblioteku sa dinamičkim vezivanjem (DLL) u sistemu sa virtuelnom memorijom. Navesti koji načini adresiranja mogu da se koriste u mašinskom kodu koga generiše ovaj prevodilac za pristup do statičkog podatka u takvoj biblioteci (u tabelu upisati „Da“ ili „Ne“):

|  |  |
| --- | --- |
| Način adresiranja | Dozvoljen? |
| Apsolutno | Ne |
| Relativno u odnosu na SP | Ne |
| Relativno u odnosu na PC | Da |

a)(10) U nekom operativnom sistemu moguće je statičku biblioteku (.lib) koja ne uvozi, već samo izvozi simbole (i zato u sebi nema nerazrešenih adresa), „registrovati“ kao biblioteku sa dinamičkim vezivanjem (DLL). Precizno objasniti šta tačno radi sistem prilikom postupka „registracije“ ovakve biblioteke, a šta kada neki proces prvi put izvrši poziv za preslikavanje simbola iz date biblioteke u adresu u svom virtuelnom adresnom prostoru.

Odgovor:

Ovakva biblioteka ne uvozi, već samo izvozi simbole, sva adresiranja internih simbola su relativna, pa u sebi nema nerazrešenih adresnih polja instrukcija. Simbole koje izvozi, sa njihovim internim relativnim adresama, biblioteka ima popisane u svom zaglavlju. Prema tome, postupak registracije ovakve biblioteke kao DLL sastoji se u sledećem:

1. Sistem alocira prostor za smeštanje ove biblioteke i učitava je u memoriju.

2. Sistem zavodi podatak o učitanom DLL-u u svoju strukturu registrovanih DLL-ova (naziv biblioteke i adresa u memoriji).

3. Tabelu izveženih simbola biblioteke eventualno prepisuje u neku svoju internu strukturu za datu biblioteku ili je jednostavno ostavlja kao deo (zaglavlje) učitane biblioteke odakle će uzimati podatke za preslikavanje adresa.

Kada neki proces prvi put izvrši poziv za preslikavanje nekog simbola iz ove biblioteke u adresu, sistem radi sledeće:

1. U svojoj strukturi registrovanih DLL-ova pronalazi deskriptor traženog DLL-a (ako ga nema, vraća grešku) i pronalazi tamo njegovu adresu.

2. Iz njegovog zaglavlja ili sopstvene strukture pronalazi traženi simbol (ako ga nema vraća grešku) i njegovu relativnu adresu u odnosu na početak koda DLL-a, i sabiranjem ove dve adrese dobija apsolutnu adresu traženog simbola u memoriji.

3. Preslikava traženi DLL u adresni prostor pozivajućeg procesa (ažuriranjem njegove PMT) i inverzno preslikava dobijenu (fizičku) adresu traženog simbola u virtuelnu adresu u adresnom prostoru pozivajućeg procesa.

b) U nekom sistemu sa straničnom organizacijom virtuelne memorije preslikavanje virtuelnih u realne adrese vrši se i u kernel modu, odnosno i za izvršavanje koda kernela. Kako treba da izgleda preslikavanje adresa za kernel da bi on mogao da pristupi celom fizičkom adresnom prostoru i to tako da ga vidi „onakvog kakav jeste“, tj. da mu pristupa direktno navođenjem realnih fizičkih adresa, odnosno da preslikavanje za njega bude transparentno?

Preslikavanje treba da bude jedan-na-jedan, tako da redom stranice virtuelnog adresnog prostora kernela, počev od stranice 0, odgovaraju fizičkim okvirima u memoriji, počev od okvira 0 (stranica 0 -> okvir 0, stranica 1 -> okvir 1 itd. redom, onoliko stranica koliko okvira ima instalirana fizička memorija). Naravno, kernel vodi računa koje od tih stranica pripadaju njemu, koje korisničkim procesima, a koje su slobodne, što i jeste jedan od osnovnih zadataka modula za upravljanje memorijom.

c)(5) U nekom sistemu sa straničnom organizacijom virtuelne memorije i virtuelna i fizička adresa su 64-bitne, adresibilna jedinica je bajt, a neki proces je sistemu prijavio da koristi samo prvih 1G (giga) stranica, što iznosi 64TB (terabajta). Deskriptor stranice u PMT čuva samo broj okvira u fizičkoj memoriji. Sistem koristi registar ograničenja veličine PMT (*page table length register*, PTLR) da bi smanjio veličinu prostora potrebnog za smeštanje PMT samo na onaj deo koji proces zaista koristi. Koliko fizičke memorije treba alocirati za smeštanje PMT ovog procesa?

Odgovor:

Račun:

64TB = 26⋅240B = 216⋅230B = 216B⋅1G, pa je stranica veličine 216B = 64KB.

Zato virtuelna i fizička adresa imaju strukturu: Page/Frame(48):Offset(16), pa je deskriptor stranice veličine 48bita, odnosno 6B. Za smeštaj 1G deskriptora stranica potrebno je 6GB.

a)(5) U jezicima C i C++, globalna imena deklarisana kao static, kao npr. u sledećim deklaracijama:

static int x=1, y=2;
static void f(int i) {...}

imaju tzv. *interno vezivanje* (*internal linking*). Precizno objasniti šta to znači i kakve efekte ima na prevođenje i povezivanje.

Odgovor:

) Imena sa ovakvim deklaracijama i internim vezivanjem imaju vidljivost samo u tom fajlu i ne mogu se referisati iz drugih fajlova. Za njih prevodilac ne generiše deklaracije u zaglavlju .obj fajla kao za simbole koji se izvoze iz tog fajla, što znači da se ti simboli ne izvoze (kako je podrazumevano za globalna imena). Zbog toga njih linker jednostavno i ne vidi, pa ih ne uvodi u svoju tabelu simbola, ne razrešava referisanja i, uopšte, ni na koji način ih ne tretira.

b)(5) Precizno objasniti kakav problem i na koji način (pod kojim uslovima) može da nastane ako se iz memorije izbaci (*swap out*) proces koji čeka na završetak I/O operacije.

Odgovor:

Na primer, u slučaju kada I/O operaciju izvršava DMA, a kao bafer se koristi deo memorije koja pripada procesu, izbacivanje tog procesa može da dovede do sledećeg problema. Deo fizičke memorije koju je zauzimao bafer izbačenog procesa dodeli se drugom procesu, što znači da sadržaj tog dela memorije čita ili upisuje taj proces, a sa druge strane, isti deo fizičke memorije koristi DMA (bilo da čita ili da upisuje), što dovodi do potpuno nepredvidivih i nekorektnih efekata (onaj ko čita dobija potpuno neadekvatne podatke iz memorije).

a)(5) Precizno objasniti zašto je potrebno linkeru navesti da kao svoj proizvod treba da napravi statičku biblioteku (lib), a ne izvršni program (exe)? Precizno objasniti razlike između postupka i rezultata pravljenja ove dve vrste proizvoda povezivanja.

Odgovor:

Statička biblioteka sadrži zaglavlje sa spiskom simbola koje izvozi i uvozi, i samo telo biblioteke (prevedeni kod), drugim rečima, proizvod je istog oblika kao i proizvod prevođenja, dok izvršni fajl sadrži telo (kod) i zaglavlje u kome se nalazi adresa prve instrukcije koja treba da se izvrši. Zbog ove razlike u proizvodima, linkeru je potrebna informacija šta da napravi. Osim toga, u samom postupku razrešavanja simbola, prilikom pravljenja izvršnog fajla, postojanje nedefinisanog a referisanog simbola se nakon prvog prolaza prijavljuje kao greška. U slučaju pravljenja biblioteke, ovakav slučaj je dozvoljen.

b)(5) Navesti dva glavna nedostatka kontinualne alokacije memorije po statičkim particijama.

Odgovor:

Ograničeni stepen multiprogramiranja i slabo iskorišćenje zbog potencijalno velike interne fragmentacije.

c)(10) U nekom sistemu sa straničnom organizacijom virtuelne memorije u datom trenutku raspored stranica tri procesa po okvirima u fizičkoj memoriji izgleda redom ovako (oznake su složene redom po rastućem broju okvira, slova A, B i C označavaju procese kojima pripadaju stranice u tim okvirima, brojevi označavaju brojeve stranica tih procesa, K označava okvir koji pripada jezgru OS-a, a DLL označava okvir čiji je sadržaj DLL koga koriste sva tri procesa A, B i C):

K, B2, A1, C3, C0, DLL, B0, K, K

U virtuelnom adresnom prostoru procesa A dati DLL se nalazi na stranici broj 3, a u prostorima procesa B i C na stranici broj 4. Napisati kako izgledaju tabele preslikavanja stranica za sva tri procesa A, B i C (u tabelu upisati brojeve okvira i T ili F kao oznaku da li je stranica u memoriji ili ne, za prvih 6 stranica).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Strana: | 0 | 1 | 2 | 3 | 4 | 5 |
| Proces A | F | T 2 | F | T 5 | F | F |
| Proces B | T 6 | F | T 1 | F | T 5 | F |
| Proces C | T 4 | F | F | T 3 | T 5 | F |

a)(5) Pored svake tvrdnje dati odgovor da li je ona tačna ili netačna.

* Lakše je konstruisati program tako da koristi preklope nego isti taj program da koristi samo dinamičko učitavanje. **netacno**
* Program koji koristi preklope (*overlay*) u opštem slučaju zahteva manje memorije od istog programa kada koristi samo dinamičko učitavanje. **tacno**
* U svakom svom izvršavanju, program koji koristi preklope (*overlay*) u opštem slučaju zahteva manje memorije od istog takvog izvršavanja istog programa kada koristi samo dinamičko učitavanje. **Netacno**

b)(5) U nekom sistemu primenjuje se kontinualna alokacija memorije. Slobodni fragmenti memorije organizovani su u jednostruko ulančanu listu. Kako ova lista treba da bude uređena da bi *best fit* algoritam radio najefikasnije? Kako tada postupa ovaj algoritam?

Odgovor:

Lista treba da bude uređena po veličini slobodnog fragmenta, u neopadajućem poretku veličine. Tada *best fit* treba da pronađe prvi fragment u listi koji po veličini zadovoljava zahtev.

c)(10) U nekom sistemu sa straničnom organizacijom virtuelne memorije primenjuje se straničenje u dva nivoa. Stranica je veličine 512 bajtova. PMT i prvog i drugog nivoa su veličine po 32 ulaza, pri čemu vrednost 1 u svim bitima ulaza označava da deo virtuelne memorije koji pokriva taj ulaz uopšte nije korišćen (alociran) od strane procesa. PMT drugog nivoa se i ne alocira ako se u opsegu koga ona pokriva ne koristi ni jedna stranica. U PMT prvog nivoa nekog procesa postoji tačno 22 ulaza koji imaju vrednost 1 u svim bitima. Koliko svog virtuelnog adresnog prostora koristi ovaj proces? (Odgovor dati u vidu granica opsega u kome se može kretati ta vrednost.)

Odgovor: Ovaj proces koristi najmanje \_\_\_\_\_\_\_\_B, a najviše \_\_\_\_\_\_\_B svog virtuelnog adresnog prostora.

Obrazloženje:

Ovaj proces koristi najmanje 5 KB, a najviše 160 KB svog virtuelnog adresnog prostora.

Ovaj proces ima alocirano 10 ulaza u PMT prvog nivoa (32-22=10).

Prema tome, ovaj proces ima alocirano 10 PMT drugog nivoa i u svakoj od njih maksimalno po 32 stranice po 512 bajtova. Prema tome, koristi maksimalno 10x32x512B = 160KB.

Najmanje ovaj proces koristi samo po jedan ulaz (za po jednu stranicu) u PMT drugog nivoa, pa je donja granica 10x1x512B = 5KB.

a)(5) Počeli ste razvoj nekog svog velikog programa koji se sastoji od mnogo izvornih fajlova, preveli ste ih i prvi put pokrenuli linker koji je tada prijavio nekoliko desetina grešaka tipa „Simbol ... nije definisan“. Šta ćete prvo proveriti?

Odgovor:

Da li su u spisak za povezivanje uneti svi potrebni .obj i .lib fajlovi.

***############### do ovde sam stigao u ovoj oblasti ################***

b)(5) Neki program treba organizovati u preklope (*overlays*). Na slici je prikazan graf poziva potprograma u ovom programu nastao statičkom analizom (čvorovi predstavljaju potprograme, grana je usmerena od pozivaoca prema pozvanom potprogramu). Predložiti organizaciju ovog programa – navesti koji potprogrami bi bili u stacionarnom modulu koji se inicijalno učitava i uvek ostaje u memoriji, a koji u preklopima i kako se preklapaju.

strcmp

store

getItem

compare

sort

output

transform

main

extract

U stacinarnom modulu main i strcmp, a u jednom istom segmentu preklapaju se dva modula (preklopa): u prvom su extract, getItem, compare i store, a u drugom transform, output i sort.

a)(5) Navesti koje simbole će prevodilac zapisati u zaglavlju proizvedenog .obj fajla kao simbole koji se izvoze, nakon prevoda sledećeg C fajla:

extern int alfa;
extern unsigned beta;
unsigned beta;
void f(int,int);
void g ();
void h (float) {}
void g (void\*) {}

Odgovor: beta, h, g

b)(5) U nekom sistemu primenjuje se *best-fit* algoritam kontinualne alokacije memorije. Inicijalno je prostor veličine 256KB potpuno slobodan za alokaciju korisničkih procesa. Potom su različiti procesi zadavali sledeće zahteve (slovna oznaka označava proces koji je postavio zahtev, brojna oznaka označava veličinu alociranog prostora u KB, a minus označava gašenje procesa i oslobađanje njegove memorije):

A64, B16, C128, D32, A-, E8, F32, B-

Odgovoriti na sledeća pitanja koja se odnose na stanje memorije nakon ove sekvence zahteva:

1. Kolika je ukupna veličina trenutno slobodne memorije? Odgovor: *i*) 56KB
2. Koliko je ukupno slobodnih fragmenata? Odgovor: *ii*) 2
3. Kolika je veličina najmanjeg slobodnog fragmenta? Odgovor: *iii*) 8KB
4. Kolika je veličina najvećeg slobodnog fragmenta? Odgovor: *iv*) 48KB

c)(10) U nekom sistemu sa virtuelnom memorijom broj stranice u virtuelnoj adresi je veličine 48 bita. Da bi čuvanje PMT učinio izvodljivim, sistem koristi *hash* tabelu sa 64K ulaza za smešanje PMT svakog procesa. *Hash* funkcija je prosta *modulo*funkcija: ulaz u tabelu određuje se pomoću 16 najnižih bita broja stranice. U svakom ulazu *hash* tabele nalazi se 64‑bitna glava ulančane liste zapisa za alocirane stranice koje se preslikavaju u taj ulaz. Svaki zapis sadrži viših 32 bita broja stranice, broj okvira u koji je ta stranica preslikana (32 bita, vrednost 0 označava da stranica nije u memoriji) i pokazivač na sledeći zapis (64 bita, vrednost 0 označava kraj liste).

Neki proces je alocirao 256 najnižih i 256 najviših stranica svog virtuelnog adresnog prostora. Koliko prostora (u bajtovima) ukupno zauzima PMT ovog procesa?

Odgovor:

520KB.

Sama *hash* tabela zauzima uvek 64K=216 64-bitnih ulaza, što je 64K∙8B=512KB.

Dati proces je alocirao po 256 najnižih i 256 najviših stranica, pri čemu se po dve (jedna iz grupe najnižih i odgovarajuća iz grupe najviših) preslikavaju u isti ulaz (ukupno 256 najnižih ulaza) *hash* tabele. Dakle, svaki od 256 najnižih ulaza *hash* tabele predstavlja glavu liste sa po dva zapisa. Veličina svakog zapisa je 128 bita, odnosno 16B. Prema tome, svaka od 256 lista zauzima po 32B, što znači da liste zauzimaju 256∙32B=8KB.

Dakle, PMT ovog procesa zauzima ukupno 512KB+8KB=520KB.

a)(5) Putem veb pretraživača (engl. *web browser*) pregledate povremeno neku svoju veb aplikaciju koja je instalirana na nekom udaljenom serverskom računaru na kome se nalaze i druge aplikacije drugih korisnika koje izvršavaju njihovi procesi. Vi ste jedini posetilac te svoje aplikacije. Kada nakon duže pauze (od npr. nekoliko dana) posetite jednu određenu stranicu svoje aplikacije, primećujete sledeći fenomen. Kada prvi put zatražite tu svoju stranicu nakon pauze od nekoliko dana, vreme odziva je relativno dugo (npr. reda nekoliko sekundi). Kada odmah nakon toga osvežavate (ponovo zahtevate) istu tu stranicu, svaki naredni put vreme odziva je veoma kratko (reda dela sekunde). Pri tome, vaša aplikacija za svaki zahtev za tom stranicom radi uvek isti posao, pa u njoj nema razloga za tako drastično različito vreme odziva. Osim toga, proces koji izvršava vašu aplikaciju je odavno pokrenut i nije uopšte gašen, već je sve vreme aktivan kao serverski proces koji čeka zahteve sa veb klijenata. Kako biste objasnili ovaj fenomen velikih razlika u vremenu odziva?

Odgovor:

Operativni sistem na serverskom računaru je u međuvremenu izbacio (*swapped out*) ceo proces koji izvršava vašu aplikaciju, ili bar njegov veći deo, zbog njegove duže neaktivnosti, pošto je on bio suspendovan čekajući na zahteve sa klijenta i nije koristio svoju memoriju. Kada se nakon pauze ponovo postave zahtevi tom procesu, on se deblokira i ponovo pokrene izvršavanje, pa onda generiše mnogo straničnih grešaka, odnosno ponovo se učitava (*swap in*), što dugo traje. Kada se u kratkom periodu postave novi zahtevi sa klijenta, proces je u memoriji, pa se zahtevi izvršavaju mnogo brže.

b)(5) U nekom sistemu primenjuje se *first-fit* algoritam kontinualne alokacije memorije. Inicijalno je prostor veličine 256KB potpuno slobodan za alokaciju korisničkih procesa. Potom su različiti procesi zadavali sledeće zahteve (slovna oznaka označava proces koji je postavio zahtev, brojna oznaka označava veličinu alociranog prostora u KB, a minus označava gašenje procesa i oslobađanje njegove memorije):

A16, B64, C32, B-, D128, A-, E8, F32

Odgovoriti na sledeća pitanja koja se odnose na stanje memorije nakon ove sekvence zahteva:

1. Kolika je ukupna veličina trenutno slobodne memorije? Odgovor:56KB
2. Koliko je ukupno slobodnih fragmenata? Odgovor: 2
3. Kolika je veličina najmanjeg slobodnog fragmenta? Odgovor: 16KB

Kolika je veličina najvećeg slobodnog fragmenta? Odgovor: 40KB

a)(5) Posmatra se neki izvorni program *p* koji je podeljen na nekoliko modula i njegove tri izvršive verzije: prva dobijena statičkim povezivanjem modula, druga koja koristi dinamičko učitavanje modula i treća koja koristi preklapanje modula (*overlays*). Uz svaku dole navedenu tvrdnju navesti da li je tačna ili netačna:

*i*) Za svaki program *p*, svako izvršavanje njegove statički povezane varijante zauzima više memorije nego semantički ekvivalentno izvršavanje varijante sa dinamičkim učitavanjem.

Odgovor: Netačno.

*ii*) Za svaki program *p*, svako izvršavanje njegove varijante sa dinamičkim učitavanjem traje isto ili kraće nego semantički ekvivalentno izvršavanje varijante sa preklopima.

Odgovor: Tačno.

b)(5) U nekom operativnom sistemu virtuelna adresa je veličine 40 bita, adresibilna jedinica je bajt, a stranica je veličine 4KB. Operativni sistem preslikava prvih 256K stranica adresnog prostora svakog procesa u svoj (kernel) memorijski prostor, tako da taj deo virtuelnog adresnog prostora nije na raspolaganju samom procesu. Koliko virtuelne memorije jeste na raspolaganju za korišćenje svakom procesu?

Odgovor:1 TB – 1 GB = 1023 GB.

a)(5) U implementaciji nekog linkera postoji operacija pronalaženja datog simbola u tabeli simbola i vraćanje deskriptora tog simbola. Ova operacija se koristi na svim mestima na kojima je potrebno pristupiti tabeli simbola i u njoj pronaći dati simbol ili iz nje dohvatiti njegov deskriptor. Da li slučaj nepostojanja tražnog simbola u tabeli simbola treba tretirati kao izuzetak koji ova operacija treba da podigne, odnosno grešku koju treba prijaviti? Precizno obrazložti odgovor.

Odgovor:

Zavisi od konteksta (mesta i trenutka) poziva ove operacije. U prvom prolazu linkera, kada se prikupljaju definisani simboli iz .obj fajlova, ova operacija se poziva da bi se utvrdilo da li definisani simbol iz .obj fajla koji se obrađuje već postoji u tabeli simbola. U tom slučaju, nepostojanje tog simbola je regularna situacija, a ne izuzetak/greška koju treba prijaviti; naprotiv, postojanje takvog simbola jeste greška. U drugom prolazu, kada se razrešavaju upotrebe simbola, ova operacija se poziva da bi se dohvatio deskriptor simbola koji se koristi. U tom slučaju, nepostojanje tog simbola jeste izuzetak/greška koju treba prijaviti

b)(5) U nekom sistemu primenjuje se *best-fit* algoritam kontinualne alokacije memorije. Inicijalno je prostor veličine 512KB potpuno slobodan za alokaciju korisničkih procesa. Potom su različiti procesi zadavali sledeće zahteve (slovna oznaka označava proces koji je postavio zahtev, brojna oznaka označava veličinu alociranog prostora u KB, a minus označava gašenje procesa i oslobađanje njegove memorije):

A32, B128, C64, B-, D256, A-, E16, F64

Odgovoriti na sledeća pitanja koja se odnose na stanje memorije nakon ove sekvence zahteva:

1. Kolika je ukupna veličina trenutno slobodne memorije?
2. Koliko je ukupno slobodnih fragmenata?
3. Kolika je veličina najmanjeg slobodnog fragmenta?
4. Kolika je veličina najvećeg slobodnog fragmenta?

*i*) 112KB *ii*) 2 *iii*) 16KB *iv*) 96KB

Ulazno izlazni podsistem

* a)(5) Da li je potrebno baferisanje da bi se realizovao API koji:
1. za znakovni uređaj obezbeđuje blokovski pristup;

Da, jer se mora sakupiti ceo blok znakova u bafer koji se onda prenosi na uređaj ili sa uređaja.

1. za uređaj sa direktnim pristupom obezbeđuje sekvencijalni pristup (uz isti tip podataka koji se prenose)?

Ne, jer se uređaju sa direktnim pristupom uvek može pristupati i sekvencijano, pa nije potrebno čuvati podatke u baferu – oni se odmah mogu proslediti.

* b)(5) Navesti i precizno opisati mehanizam kojim se baferi i keševi za I/O operacije kernela u potpunosti štite od slučajnog ili malicioznog upisa od strane korisničkih procesa.

Mehanizam virtuelne memorije: okviri fizičke memorije u kojima se nalaze baferi i keševi I/O podsistema se ne preslikavaju u adresne prostore korisničkih procesa (nema ih u njihovim tabelama preslikavanja), pa im oni jednostavno nikako ne mogu pristupiti („ne vide“ ih).

* Predložiti najjednostavniji interfejs generičkog drajvera za sve blokovski orijentisane uređaje sa direktnim pristupom, pogodne da se na njima organizuje fajl sistem. (Interfejs treba da obezbedi učitavanje i upisivanje na nivou bloka, ne na nivou fajla. Koncept fajla i operacije sa njim treba da obezbedi fajl sistem koji koristi usluge ovog drajvera. Interfejs treba da bude dovoljan da se pomoću njega izgradi fajl sistem.)

typedef unsigned long BlockSize;
typedef unsigned long BlockNo;

enum IOStatus {Err,OK};

class DeviceDriver {
public:

 virtual BlockSize getBlockSize (); // Returns block size
 virtual BlockNo getNumOfBlocks(); // Returns number of blocks

 virtual IOStatus read (BlockNo, void\* buffer); // Reads a block to mem
 virtual IOStatus write(BlockNo, void\* buffer); // Writes a block to dev
};

* Ako kontroler uređaja poseduje sopstveni, hardverski keš i bafer koji obezbeđuju veoma brz odziv i kod operacija čitanja i kod operacija upisa bloka, samerljiv sa odzivom operativne memorije, kojom tehnikom programiranog ulaza-izlaza treba realizovati drajver ovakvog uređaja: prozivanjem (*polling*) ili korišćenjem prekida? Kratko obrazložiti odgovor.

Prozivanjem (*polling*), jer je u ovom slučaju efikasnije od obrade prekida (koja uključuje dodatne režije).

* Ako sistem koristi *spooling*, kakva je po svojoj prirodi operacija slanja niza znakova na štampač koju poziva korisnički proces, posmatrana „sa kraja na kraj“, tj. iz perspektive korisničkog procesa na operaciju koju zadaje i uređaj koji treba da je izvrši - asinhrona ili sinhrona (blokirajuća)?

Uvek asinhrona, jer proces zadaje operaciju i nastavlja svoje izvršavanje, ne čekajući da se uređaj završi zadatu operaciju. Operacija neće ni početi pre nego što dati proces „zatvori“ svoj posao sa uređajem (a posao može da uključuje više ovakvih operacija) i kada taj posao dođe na red za obradu od strane *spooler*-a. Zato pozivajući proces nikada i ne treba da čeka na završetak ove operacije.

* Povećanje bafera i blokova prenosa prilikom upisa na disk ima mnoge pozitivne efekte, poput ređih i efikasnijih operacija sa uređajima. Međutim, kakav je negativan uticaj povećanja bafera, u pogledu posledica u slučaju otkaza?

Veći baferi znače i ređe slanje podataka na disk, što znači vežu štetu u slučaju otkaza: više podataka duže vreme nije snimljeno na disk i ostaje trajno izgubljeno u slučaju otkaza računara.

a)(5) Drajver (*device driver*) nekog diska kešira blokove sa diska u operativnoj memoriji. Prokomentarisati šta su prednosti, a šta nedostaci sledećih tehnika upisa podataka na disk ovog drajvera:

* *asinhroni* upis ili *write-back*: blok iz keša se upisuje na disk odloženo, najkasnije onda kada se zamenjuje drugim blokom sa diska;
* *sinhroni* upis ili *write-through*: svaki upis u blok u kešu se odmah prosleđuje i na disk.

Odgovor:

Asinhroni (*write-back*): bolje performanse, jer se višestruki upisi u blok reflektuju samo u jedan odloženi upis na disk, ali povećanja osetljivost na otkaze, jer se prilikom otkaza sistema gube podaci koji nisu upisani na disk. Sinhroni (*write-through*) – obratno.

b)(5) Neki proces otvara datoteku sa najavom da će joj pristupati isključivo sekvencijalno. Koje tehnike biste predložili da bi se performanse pristupa ovoj datoteci poboljšale?

Odgovor:

Keširanje i dohvatanje unapred (*read-ahead*).

)Predložiti API, u smislu funkcija za sistemske pozive i njihovih deklaracija na jeziku C, za sledeće usluge nekog operativnog sistema vezanih za realno vreme:

a)(5) očitaj i vrati tekući datum i vreme;

c)(5) pokreni datu operaciju u zadato vreme.

Precizno objasniti način korišćenja predloženih sistemskih poziva (može i na primeru).

Odgovor:

a)

struct date {
 int day, month, year;
};
struct time {
 int hour, min, sec, millisec;
};
struct date\_time {
 date d; time t;
};

date\_time get\_date\_time ();

Primer korišćenja:

date\_time now = get\_date\_time();
printf(“Today is %d/%d/%d.\n”, now.d.month, now.d.day, now.d.year);
printf(“It is %d:%d:%d now.\n”, now.t.hour, now.t.min, now.t.sec);

a)(5) Koja tehnika omogućava da se upis na neki izlazni, znakovno orijentisani uređaj sa strane procesa vrši kao blokovski upis? Precizno obrazložiti odgovor.

Baferisanje, po principu „proizvođač-potrošač“. Korisnički proces upisuje u bafer blok podataka (kao „proizvođač“), dok poseban proces koji kontroliše uređaj (kao „potrošač“) iz bafera uzima znak po znak i neposredno ga šalje na uređaj.

b)(5) Koja tehnika omogućava da zahtev za štampanjem koji je dat pre isključenja računara bude obrađen nakon ponovnog uključenja računara? Precizno obrazložiti odgovor.

Odgovor:

*Spooling*. Zahtev je upisan u fajl koji se smešta na određeno mesto (u određeni sistemski direktorijum). Taj fajl ostaje na istom mestu i po isključenju računara. Po ponovnom uključenju računara, pokreće se *spooler* koji pronalazi fajlove na ovom mestu i jedan po jedan obrađuje slanjem na štampač.

b)(5) Posmatra se neki sekvencijalni, izlazni, blokovski orijentisani uređaj sa blokom veličine N i operacijom koja na uređaj izbacuje dati blok:

void output(const char\* block);

Korišćenjem ove operacije, realizovati operaciju

void putc(char);

koja izbacuje jedan znak na dati uređaj, uz baferisanje (znak po znak se upisuje u bafer, a kada se bafer napuni, izbacuje se ceo bafer kao blok).

Rešenje:

void putc(char c) {
 static char buffer[N];
 static int cursor=0;
 buffer[cursor++]=c;
 if (cursor==N) {
 output(buffer);
 cursor=0;
 }
}

Na nekom računaru sa multiprogramskim operativnim sistemom neki kontroler periferije nema vezan svoj signal završetka operacije na ulaze za prekid procesora, ali je poznato vreme završetka operacije zadate tom uređaju u najgorem slučaju (maksimalno vreme završetka operacije). Za obradu ulazno/izlaznih operacija zadatih tom uređaju brine se poseban proces tog operativnog sistema. Kakvu uslugu treba operativni sistem da obezbedi, a ovaj proces da koristi, da bi se izbegla tehnika kojom se može detektovati završetak operacije čitanjem statusnog registra i ispitivanjem bita završetka operacije (*polling*), tj. da bi se izbeglo uposleno čekanje (*busy waiting*)? Precizno opisati kako ovaj proces treba da koristi ovu uslugu.

Odgovor:

Potrebno je obezbediti sistemsku uslugu (poziv) „uspavljivanja“ (suspenzije) procesa na zadato vreme. Opisani proces bi, nakon pokretanja ulazno/izlazne operacije, pozvao ovu uslugu i suspendovao se na vreme zaveršetka ove operacije u najgorem slučaju. Kada se ponovo aktivira, proces može da ispita samo uspešnost završetka zadate operacije, odnosno postojanja eventualne greške, a ne i spremnost uređaja za novu operaciju.

a)(5) Umesto tehnike *spooling*, pristup uporednih procesa štampaču u nekom sistemu omogućuje se međusobnim isključenjem pomoću operacije rezervacije: svaki proces, pre nego što zatraži bilo koju operaciju sa štampačem, mora da ga rezerviše, a tu rezervaciju otpušta tek kada završi sa celim jednim poslom štampe. Šta je nedostatak ove tehnike?

Odgovor:

Problem je u veoma slaboj konkurentnosti: svi drugi procesi koji su došli do stanja u kome bi želeli da pokrenu svoj posao štampe moraju da čekaju dok proces koji je rezervisao štampač ne završi ceo svoj posao. Time su poslovi štampe od različitih procesa potpuno sekvencijalizovani, a mogu da traju relativno dugo. Zbog toga je odziv sistema veoma loš, a zadržavanje procesa u sistemu značajno produženo.

Fajl podsistem

* Kolika je maksimalna veličina fajla za fajl sistem opisan pod a) instaliran na particiji veličine 4TB (terabajta) i blokom veličine 1KB?

Na particiji ima 4TB/1KB = 22x240/210 = 232 blokova, pa se fizički blok adresira 32-bitnom adresom (brojem bloka).

Zbog toga je jedan ulaz u indeksu veličine 32 bita, odnosno 4B.

Kako se indeks smešta u jedan blok veličine 1KB, indeks sadrži 1KB/4B = 256 ulaza.

Svaki ulaz sadrži adresu jednog bloka datog fajla, pa fajl može da ima najviše 256 blokova, odnosno 256KB.

* Kolika je veličina FAT za fajl sistem opisan pod a) instaliran na particiji veličine 4TB (terabajta) i blokom veličine 1KB, ako je veličina tipa short int jedan bajt, a tipa long int 4 bajta? Napomena: FAT sadrži ulaze za sve fizičke blokove u particiji, čak i za one koji se ne koriste za smeštanje fajlova (npr. one za smeštanje same FAT).

Particija ima 4 TB / 1 KB = 22\*240 B / 210 B = 232 = 4 G blokova.

Jedan ulaz u FAT zauzima 5 bajtova.

Tako FAT zauzima 4 G \* 5 B = 20 GB.

* Neki fajl sistem koristi indeksirani pristup alokaciji blokova za fajlove, sa indeksima u jednom nivou. Indeks za fajl se smešta u poseban blok na koga ukazuje jedno polje u FCB (*file control block*); ceo taj blok koristi se samo za smeštanje indeksa fajla. Ovaj fajl sistem podržava particije maksimalne veličine 4G blokova.

a)(5) Kolika je maksimalna veličina fajla za taj fajl sistem instaliran na disku sa blokom veličine 1KB?

4G blokova se adresira 32-bitnom adresom (brojem bloka), jer je 4G = 232.

Zbog toga je jedan ulaz u indeksu veličine 32 bita, odnosno 4B.

Kako se indeks smešta u jedan blok veličine 1KB, indeks sadrži 1KB/4B = 256 ulaza.

Svaki ulaz sadrži adresu jednog bloka datog fajla, pa fajl može da ima najviše 256 blokova, odnosno 256KB.

b)(5) Ako se umesto indeksa u jednom nivou koristi indeks u dva nivoa, pri čemu se indeks prvog nivoa smešta u ceo zasebanblok na koga ukazuje polje u FCB, a indeksi drugog nivoa takođe zauzimaju cele blokove, koliko onda iznosi maksimalna veličina fajla na ovom disku?

I dalje jedan blok ima 256 ulaza, ali ceo indeks može da adresira 256\*256 blokova fajla, dakle ukupno 216 = 64K blokova, pa je maksimalna veličina fajla sada 64MB.

c)(5) Za oba slučaja pod a) i b) izračunati koeficijent iskorišćenja prostora na disku za smeštanje fajla maksimalne veličine, kao količnik prostora koga zauzimaju podaci u fajlu i ukupnog prostora koga zauzima fajl (podaci plus indeks, ne računajući FCB). Rezultate dati kao razlomke.

Za slučaj pod a): Korisni prostor za podatke je 256KB, indeks zauzima 1 blok, odnosno 1KB, pa je iskorišćenje 256/(256+1), odnosno približno 99.6%.

Za slučaj pod b): Korisni prostor za podatke je 64MB, indeks zauzima 257 blokova, odnosno 257KB, pa je iskorišćenje 65536/(65536+257), odnosno približno 99.6%.

* Veličina bloka na disku je 512B, a fajl sistem podržava adresiranje 4G blokova na disku. Fajlovi se alociraju indeksirano (*indexed allocation*) sa indeksima u jednom nivou i bez ulančavanja indeksnih blokova (indeks je samo jedan blok). Kolika je maksimalna veličina fajla koju podržava ovaj sistem?

Maksimalan broj blokova na disku: 4G = 232

Zbog toga jedan ulaz u indeksu zauzima 32 bita, odnosno 4B = 22B

Broj ulaza u indeksnom bloku: 29/22=27=128

Maksimalna veličina fajla je: 128 blocks \* 512B/block = 64KB.

* (a)(5) Da li je veličina fajla ograničena ako je način alokacije blokova za fajlove na disku:

1) ulančani Nije.

2) indeksirani Jeste.

* (b)(5) Navedite bar jedan razlog zašto bi neki operativni sistem zahtevao ili bar omogućavao da korisnički proces prilikom kreiranja fajla eksplicitno navede da će taj fajl biti isključivo ili dominantno korišćen tako što će se sekvencijalno čitati.

Ulančana alokacija blokova na disku za taj fajl i učitavanje unapred (*read-ahead*) su tehnike koje povećavaju efikasnost za ovakav pristup.

* (c)(5) Neki fajl sistem koristi indeksirani pristup alokacije blokova za fajlove na disku, sa kombinovanom tehnikom indeksiranja u jednom, dva i tri nivoa, kao kod UNIX *inode* strukture. Pretpostavljajući da disk ima uniformno srednje vreme pristupa do bilo kog bloka na disku, da li je vreme pristupa do različitih delova veoma velikih fajlova jednako? Ako jeste, precizno objasniti zašto jeste, a ako nije, objasniti kako se i zašto razlikuje.

Nije. Vreme pristupa raste za blokove bliže kraju veoma velikog fajla, jer se za pristup do njih mora prolaziti kroz višestepene indekse. Pristup do delova bližim početku fajla je brži jer prolazi samo kroz jednostepeni indeks.

b)(5) Kratko, ali precizno objasniti zbog čega se kod mnogih fajl sistema kombinuju pristupi sa indeksom u jednom nivou i u više nivoa, kao kod Unix inode strukture.

Odgovor:

Indeks u jednom nivou obezbeđuje brži (direktniji) pristup sadržaju fajla, jer zahteva manje pristupa medijumu (za čitanje samo jednog indeksnog bloka), dok indeks u više nivoa omogućava veće dozvoljene veličine fajlova. Zbog toga se kombinuju pristupi sa indeksima u jednom i više nivoa, tako da se za male fajlove dobija brz pristup, a za veće fajlove brži pristup blokovima koji su logički bliži početku fajla, dok se i dalje omogućava značajno veća maksimalna dozvoljena veličina fajla.

a)(5) Neki fajl sistem podržava implicitno zaključavanje fajla prilikom njegovog otvaranja. Postoje dve vrste ključa: deljeni (*shared*, S), koji se traži prilikom otvaranja fajla samo za čitanje (proces koji je tako otvorio fajl ima pravo samo da čita iz fajla) i ekskluzivni (*exclusive*, X), koji se traži prilikom otvaranja fajla i za upis (proces koji je otvorio fajl ima pravo upisa). Popuniti sledeću tabelu upisivanjem oznaka onih procesa čiji će zahtev za otvaranjem istog fajla biti ispunjen, za svaki od dva data slučaja. Procesi postavljaju zahteve redom navedenim u drugoj koloni, pri čemu oznaka npr. A-Rd označava da proces A postavlja zahtev za otvaranjem fajla za čitanje, a B-Wr da proces B postavlja zahtev za otvaranjem fajla za upis.

|  |  |  |
| --- | --- | --- |
| Slučaj  | Zahtevi za otvaranje fajla | Procesi koji su uspeli da otvore fajl |
| 1 | A-Rd, B-Wr, C-Rd, D-Rd, E-Wr |  |
| 2 | A-Wr, B-Rd, C-Wr, D-Rd, E-Rd |  |

|  |  |  |
| --- | --- | --- |
| Slučaj  | Zahtevi za otvaranje fajla | Procesi koji su uspeli da otvore fajl |
| 1 | A-Rd, B-Wr, C-Rd, D-Rd, E-Wr | A, C, D |
| 2 | A-Wr, B-Rd, C-Wr, D-Rd, E-Rd | A |

b)(10) Neki fajl sistem koristi indeksirani pristup alokaciji blokova za fajlove sa indeksima u dva nivoa. Prvi nivo indeksa smešten je u sam FCB (*file control block*) i ima 256 ulaza koji direktno referenciraju blokove sa podacima, a još jedan ulaz u FCB ukazuje na jedan blok sa indeksom drugog nivoa čiji ulazi takođe ukazuju na blokove sa podacima. Veličina ulaza u indeksnom bloku (reference na blok diska) je 8 bajtova, a veličina bloka na disku je 512KB. Kolika je maksimalna veličina fajla u ovom fajl sistemu?

Odgovor: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Račun i objašnjenje:

Račun:

Maksimalan broj blokova koje fajl može da zauzima je 256 + 512K/8 = 28+ 219/23 = 28 + 216 = 28(28+1).

Maksimalna veličina fajla je 28(28+1)⋅29KB = 27(28+1)⋅MB = 32GB + 128MB = 32.125GB

b)(5) Na nekom sistemu izvršene su sledeće komande iz komandne linije, datim redom:

|  |  |
| --- | --- |
| Komanda | Rezultat ispisan na terminalu nakon izvršene komande |
| dir | a.dir, b.txt, c.dir |
| cd a | /home/webmail/incoming/a |
| cd .. | /home/webmail/incoming |

Šta će biti ispisano nakon komande cd ../.. koja se izvrši odmah nakon ove sekvence?

/home

a)(5) Ovo je trag jedne sesije nekog korisnika u nekom operativnom sistemu (linije koje počinju sa > su komande korisnika, iza njih sledi odgovor sistema):

>dir
/usr/john/mydocs/a.txt 256kB 09/01/2007 rw-rw--w-
/usr/john/mydocs/b.txt 16kB 09/01/2007 rw----r--
> type a.txt
Command failed: (25064) Access denied
> type b.txt

Da li će ova poslednje izdata komanda biti uspešno izvršena? Obrazložiti.

Hoće. Komanda type a.txt nije uspela jer je sistem zabranio pristup (u ovom slučaju za čitanje) fajlu a.txt ovom korisniku. Iz flegova prava pristupa ovog fajla vidi se da je čitanje zabranjeno samo poslednjoj kategoriji korisnika, što znači da korisnik pripada baš toj (i nijednoj drugoj) kategoriji. Kako je pristup fajlu b.txt za čitanje dozvoljen ovoj kategoriji korisnika, komanda type b.txt će se izvršiti uspešno.

a)(10) Neki fajl sistem podržava organizaciju direktorijuma kao aciklični usmereni graf (DAG). Pri tome, isti fizički fajl ili direktorijum može biti referisan iz više direktorijuma, u svakom sa različitim imenom. Na slici je prikazana struktura trenutnog sadržaja jednog takvog sistema, pri čemu krugovi predstavljaju fizičke fajlove, a pravougaonici direktorijume, pri čemu su nazivi unutar pravougaonika identifikatori elemenata unutar tog direktorijuma.

**OS**

**PRV**

**Ispiti**

**Skripta**

**Ispiti**

**Literatura**

**Jan05**

**Feb05**

**Jan05**

**Jan06**

**OS1**

**PRV**

**Ispiti**

**OS**

**PRV**

Napisati sve moguće pune staze kojim se mogu identifikovati fajlovi označeni sa A, B i C.

A: /OS/Ispiti/Feb05, /Ispiti/OS/Feb05

B: /OS/Literatura/PRV, PRV/Skripta

C: /PRV/Ispiti/Jan06, / Ispiti/PRV/Jan06

b)(5) Precizno objasniti kako i zašto asinhroni upis podataka u fajl utiče na performanse fajl sistema. Objašnjenje potkrepiti primerom.

Kod asinhronog upisa podataka korisnički proces upisuje podatke samo u keš i nastavlja dalje svoje izvršavanje, ne čekajući da se duga operacija prenosa tih podataka na uređaj potpuno završi. Tu operaciju vrše posebne niti ili procesi uporedo, koristeći „pauze“ u radu korisničkih procesa, već u zavisnosti od načina raspoređivanja. Zbog toga je, u principu, odziv korisničkih procesa bolji. Jednostavan primer je interaktivna aplikacija u kojoj korisnik aktivira snimanje dokumenta nad kojim radi na disk, ali ne mora da čeka da se snimanje završi, već nastavlja svoj rad nad dokumentom; snimanje će se vršiti u pauzama između akcija korisnika i rada njegovog programa (procesa).

b)(5) Neki fajl sistem primenjuje FAT (*File Allocation Table*). Veličina FAT je 16K ulaza, a FAT „pokriva“ (referiše) sve one i samo one blokove na disku koji su namenjeni za smeštanje sadržaja fajlova. Veličina bloka koji se alocira na disku je 512KB. FCB (*File Control Block*) strukture su prealocirane na posebnom delu diska koji se ne koristi za smeštanje fajlova i ima ih ukupno 256K. Direktorijumi se smeštaju kao i fajlovi, a podržana je hijerarhijska struktura direktorijuma. Kolika je teorijski maksimalna veličina fajla?

Odgovor:8GB - 512KB (skoro 8GB).

Jedan fajl može zauzeti ceo prostor predviđen za smeštanje fajlova, naravno ukoliko je to jedini fajl (granični slučaj), jer je alokacija kod FAT ulančana i nema ograničenja u broju blokova na nivou jednog fajla kao kod npr. indeksiranja, osim ograničenja veličinom same FAT. Podatak o broju FCB-ova je ovde nebitan. Maksimalna veličina fajla jednaka je, dakle, veličini prostora određenog za smeštanje fajlova, a to je proizvod broja ulaza u FAT (16K) i veličine bloka (512KB). Ovo važi u slučaju da je taj fajl i jedini, što znači da je ili direktorijum (dakle prazan, pa ne može da zauzima toliki prostor), ili jedini fajl u jedinom direktorijumu, kada i taj direktorijum mora zauzimati bar jedan blok. Zbog toga je teorijski maksimalna veličina fajla:

16K ⋅ 512KB = 214⋅ 219B = 233B = 8GB, umanjeno za veličinu jednog bloka od 512KB.

a)(5) U cilju optimizacije vremena pristupa fajlovima, neki fajl sistem za isti fajl pokušava da alocira blokove na disku koji su međusobno što bliži. Koja struktura podataka za evidenciju slobodnih blokova je efikasnija u smislu brzine alokacije blokova prilikom proširenja postojećeg fajla – ulančana lista slobodnih blokova ili bit-vektor? Obrazložiti.

Bit-vektor, zato što omogućava direktan pristup. Kod alokacije novih blokova za postojeći fajl može se direktno (i efikasno) pristupiti delu vektora koji reprezentuje blokove koji su susedni blokovima koje taj fajl već zauzima i među njima pronaći slobodni. Ulančana lista, sa druge strane, zahteva sekvencijalni prolaz kroz listu slobodnih blokova, od početka pa do mesta koje je blizu blokovima koji pripadaju datom fajlu.

a)(5) U nekom fajl sistemu primenjuje se sistem provere prava pristupa koja se definišu preko tri bita (rwx) za korisnika koji je kreirao fajl (tzv. vlasnik, engl. *owner*) i za sve ostale korisnike. Dat je prikaz jedne interakcije korisnika i sistema preko komandne linije:

>dir -all
my.dir 02/06/2008 12:34 rw-r--
 a.txt 02/06/2008 12:40 rw-r--
 p.exe 02/06/2008 12:45 rwxr--
> run p.exe
Error: Access denied.
>delete a.txt

Da li će ova poslednja operacija brisanja fajla (del a.txt) biti uspešna? Obrazložiti.

Neće, već će sistem prijaviti grešku zbog nedozvoljenog pristupa. Naime, iz rezultata komande runp.exe vidi se da korisnik čija je interakcija prikazana nema prava izvršavanja ovog fajla, pa je on onaj na koga se odnose druga tri znaka rwx u listi prava pristupa. Bez obzira da li sistem operaciju brisanja fajla smatra operacijom izmene (pisanja) direktorijuma ili samog fajla, ni jedna od njih nije dozvoljena datom korisniku, pa neće biti dozvoljeno ni brisanje.

b)(10) Posmatraju se tri načina rukovanja slobodnim prostorom u nekom fajl sistemu:

1) Pomoću bit-vektora koji se smešta u tačno određene blokove na disku.

2) Pomoću ulančane liste, pri čemu se pokazivači za ulančavanje smeštaju neposredno u slobodne blokove.

3) Pomoću ulančane liste slobodnih blokova, ali uz korišćenje FAT (*File Allocation Table*).

Uporediti sledeće karakteristike ovih tehnika i odgovoriti koja je od navedenih tehnika efikasnija. Odgovore obrazložiti.

*i*)(4) Koja tehnika, 1 ili 2, je generalno efikasnija u smislu korišćenja prostora na disku i količine korisnog prostora za podatke?

*ii*)(3) Koja tehnika, 1 ili 2, je generalno efikasnija u smislu brzine alokacije nekoliko slobodnih blokova na disku?

*iii*)(3) Koja tehnika, 2 ili 3, je generalno efikasnija u smislu brzine alokacije nekoliko slobodnih blokova na disku?

*i*)(4) Tehnika 2 (ulančana lista), jer se kod nje strukture za evidenciju slobodnog prostora smeštaju u same slobodne blokove, dok se kod bit-vektora alocira poseban prostor na disku za smeštanje bit-vektora i time smanjuje koristan prostor za podatke.

*ii*)(3) Tenika 1 (bit-vektor), jer se pretraga i alokacija slobodnih blokova generalno svodi na učitavanje jednog ili nekoliko susednih blokova sa delom bit-vektora, a zatim na iterativnom postupku pristupa mašinskim rečima i ispitivanju bita u njima, što su efikasne mašinske operacije. Osim toga, ovi blokovi sa bit-vektorom se mogu keširati u posebnom kešu i zato biti stalno prisutni u operativnoj memoriji. Kod tehnike 2 se u svakoj iteraciji prolaska kroz listu pristupa novom bloku na disku.

*iii*)(3) Tehnika 3 (FAT). Iako se obe svode na prolazak kroz ulančanu listu, kod FAT se pristupa manjem fizičkom prostoru, odnosno učitavanju manjeg broja blokova sa samom FAT. Kod tehnike 2 se u svakoj iteraciji prolaska kroz listu pristupa novom bloku na disku. Osim toga, FAT se može u celini ili delimično keširati u posebnom kešu i zato biti stalno prisutna u memoriji.

a)(5) U nekom fajl sistemu podržano je zaključavanje fajlova od strane procesa koji ih otvaraju, uz postojanje koncepta *deljenog* (*shared*) i *ekskluzivnog* (*exclusive*) ključa. Proces fajl zaključava implicitno, prilikom njegovog otvaranja, kada u sistemskom pozivu deklariše da li taj fajl otvara samo za čitanje (*read-only*) ili i za izmene (*write-enabled*). Proces koji fajl otvara samo za čitanje zahteva i, ukoliko su zadovoljeni uslovi, dobija deljeni ključ; proces koji fajl otvara i za izmene zahteva i, ukoliko su zadovoljeni uslovi, dobija ekskluzivni ključ. Deljeni ključ se može dobiti ako i samo ako ni jedan drugi proces trenutno nema ekskluzivni ključ; ekskluzivni ključ se može dobiti ako i samo ako ni jedan drugi proces nema bilo kakav ključ nad istim fajlom. Ukoliko proces ne može da dobije traženi ključ, operacija otvaranja fajla završava se greškom.

U ovom sistemu izvršena je sledeća sekvenca sistemskih poziva; pozivi su označeni simbolički, gde Open znači otvaranje, a Close zatvaranje fajla koji je identifikovan prvim argumentom, dok drugi argument označava vrstu operacije (R-*read-only*, W-*write-enabled*); identifikator ispred dvotačke označava proces koji je izvršio taj poziv:

P1:Open(F1,W), P2:Open(F2,W), P3:Open(F3,R), P1:Close(F1)

Navesti koje od sledećih sekvenci koje se, svaka za sebe i nezavisno, posmatraju kao nastavak ove sekvence, izvršavaju bez greške, a koje sa greškom:

*i*) P2:Open(F3,R), P2:Open(F1,W)
*ii*) P1:Open(F3,R), P1:Open(F2,R)
*iii*) P2:Close(F2), P2:Open(F1,W), P1:Open(F3,R)

*i*) Bez greške. *ii*) Sa greškom. *iii*) Bez greške.

a)(5) U nekom operativnom sistemu postoji koncept tekućeg direktorijuma procesa, pri čemu proces može promeniti svoj tekući direktorijum odgovarajućim sistemskim pozivom, ali mu se dodeljuje podrazumevani tekući direktorijum prilikom pokretanja. Navesti bar dve različite mogućnosti – koji direktorijum se dodeljuje kao podrazumevani tekući direktorijum.

Neke mogućnosti:

- tekući direktorijum procesa-roditelja koji pokreće dati proces; specijalni slučaj ovoga je tekući direktorijum procesa koji izvršava interpreter komandne linije;

- direktorijum u kome se nalazi fajl programa koji izvršava dati proces;

- podrazumevani direktorijum pridružen korisniku koji je „vlasnik“ pokrenutog procesa (a koji je i „vlasnik“ roditeljskog procesa);

- zadaje se kao obavezni parametar pokretanja procesa.

a)(5) U operativnim sistemima uobičajen je protokol u kome se svaki proces pokreće „u ime“ nekog korisnika (koji je njegov „pokretač“ ili „vlasnik“), i to istog onog u čije ime je pokrenut i roditeljski proces koji je pokrenuo taj proces odgovarajućim sistemskim pozivom za kreiranje procesa, pri čemu se početni procesi (npr. školjka) pokreću u ime korisnika koji se prijavio na sistem implicitno po prijavljivanju (*log-in*) tog korisnika. Prava pristupa do fajlova određuju se na nivou korisnika i grupa korisnika. Precizno obrazložiti kakve bi posledice u smislu pristupa do fajlova imalo uvođenje mogućnosti da se u sistemskom pozivu za kreiranje procesa proces-dete kreira tako da se izvršava „u ime“ nekog drugog korisnika, a ne istog onog u čije ime se izvršava i proces-roditelj, pri čemu bi ta mogućnost bila dozvoljena svim procesima.

Odgovor:

To bi otvorilo veliku „sigurnosnu rupu“ (*security hole*) u sistemu provere prava pristupa do fajlova. Naime, sistem provere prava pristupa do fajla upoređuje pravo *procesa* da izvrši odgovarajući sistemski poziv za pristup do nekog fajla. Pri tome uzima u obzir onog korisnika „u čije ime“ se taj proces izvršava, odnosno korisnika koji je „vlasnik“ tog procesa. Na osnovu odgovarajućih pravila, proverava se da li je taj korisnik „vlasnik“ fajla ili pripada određenoj grupi korisnika itd. Ako bi se proces-dete kreirao tako da se izvršava u ime nekog drugog korisnika, onda bi na taj način neko slučajno ili zlonamerno mogao da kreira proces koji bi smeo da pristupi fajlovima do kojih inače korisnik u čije ime se izvršava roditeljski proces ne bi imao prava pristupa, što bi probilo zamišljeni sistem zaštite pristupa do fajlova.

a)(5) Neki fajl sistem podržava organizaciju direktorijuma u obliku usmerenog acikličkog grafa (DAG). Zbog ovakve organizacije neophodni su sistemski pozivi za dve različite operacije kopiranja datog fajla na odredišno mesto. Predložiti ova dva sistemska poziva, napisati njihove deklaracije na jeziku C i objasniti njihovo značenje. U oba poziva zadaju se identifikator fajla koji se kopira („ručka“) i naziv (puna putanja) koji će imati iskopirani fajl.

Odgovor:

Pravi potpuno novi fajl na odredištu sa iskopiranim sadržajem, tj. u odredišnom direktorijumu kreira ulaz sa zadatim nazivom koji ukazuje na novokreirani fajl kao objekat u sistemu (novi *inode*):

error\_code fileDeepCopy (FHANDLE src, char\* dstFileName);

Kopira samo referencu, tj. u odredišnom direktorijumu kreira ulaz sa zadatim nazivom koji ukazuje na isti fajl kao objekat u sistemu (isti *inode*):

error\_code fileShallowCopy (FHANDLE src, char\* dstFileName);

Reakcija oba sistemska poziva na izuzetne situacije može da bude ista. Izuzetne situacije su, na primer: odredišni direktorijum ne postoji, u odredišnom direktorijumu već postoji ulaz sa datim nazivom, pozivajući proces nema pravo upisa u odredišni direktorijum.

b)(10) U nekom fajl sistemu direktorijum se implementira kao običan fajl čiji je sadržaj organizovan korišćenjem *hash* tabele na sledeći način. U prvom delu sadržaja je sama *hash* tabela koja ima 4 ulaza (0..3). Svaki ulaz ima sadržaj koji predstavlja redni broj (počev od 1) zapisa u ulančanoj listi zapisa za taj ulaz. Ovi zapisi nalaze se u nastavku *hash* tabele u sadržaju. Vrednost 0 u ulazu označava praznu listu. U svakom zapisu u listama nalazi se naziv elementa direktorijuma (niz znakova), identifikator objekta u fajl sistemu kojim je predstavljen taj element (broj *inode* objekta) i pokazivač na sledeći zapis u listi zapisa za isti ulaz u *hash* tabeli. Ovaj pokazivač ima vrednost koja se tumači na isti način kao i u ulazu *hash* tabele (0 – *null*, >0 – redni broj). U nekom direktorijumu, koji je inicijalno bio prazan, kreirani su tim redom sledeći fajlovi (u trećoj koloni su date *hash* vrednosti njihovih naziva):

|  |  |  |
| --- | --- | --- |
| Naziv | *inode#* | *hash value* |
| alfa | 100 | 3 |
| beta | 200 | 0 |
| gama | 300 | 1 |
| delta | 400 | 3 |
| eta | 500 | 1 |

Popuniti sadržaj *hash* tabele i lista zapisa u njenom nastavku za posmatrani direktorijum

*Hash* tabela:

|  |  |
| --- | --- |
| Ulaz | Pokazivač na prvi zapis u listi |
| 0 | 2 |
| 1 | 3 |
| 2 | 0 |
| 3 | 1 |

Lista zapisa:

|  |  |  |  |
| --- | --- | --- | --- |
| Ulaz | Naziv | *inode*# | Sledeći |
| 1 | alfa | 100 | 4 |
| 2 | beta | 200 | 0 |
| 3 | gama | 300 | 5 |
| 4 | delta | 400 | 0 |
| 5 | eta | 500 | 0 |

a)(5) U opštem slučaju, da li ista vrednost identifikatora („ručke“) fajla (FHANDLE) koju su dva procesa dobila prilikom operacija otvaranja fajla koje su izvršili, znači da oni pristupaju istom fizičkom fajlu preko tih identifikatora? Precizno obrazložiti.

Odgovor:

Ne. Stvarna vrednost identifikatora (ručke) u mnogim sistemima ima samo lokalni opseg, unutar procesa koji je otvorio fajl, jer predstavlja identifikator ulaza u tabeli otvorenih fajlova samo tog procesa (npr. broj/indeks ulaza u toj tabeli). Zbog toga dva procesa mogu imati istu ovakvu vrednust, a da zapravo ti ulazi predstavljaju sasvim različite fajlove.

a)(5) U nekom fajl sistemu struktura direktorijuma ima oblik usmerenog acikličkog grafa (DAG), pri čemu se isti fajl (kao objekat u fajl sistemu) može referencirati iz više različitih direktorijuma pod različitim imenima. Na primer, sledeće staze predstavljaju isti fajl: /mydocs/letters/to\_john.doc i /correspondence/john/letter\_17\_09\_2009.doc. Gde se čuva ime fajla u ovom sistemu, da li kao svaki drugi atribut fajla u FCB ili negde drugde? Precizno obrazložiti odgovor.

Odgovori:

U ovom sistemu ime fajla nije atribut fajla kao objekta koji se čuva u FCB, jer ono nije jedinstveno. Ime fajla je zapravo identifikator fajla u samo jednom direktorijumu iz koga se taj fajl referencira, a kojih može biti više. Zbog toga se ime fajla čuva u samom direktorijumu, tj. u strukturi koja implementira direktorijum kao objekat fajl sistema, i u svakom direktorijumu u kome je fajl sadržan ono je potencijalno različito. Direktorijum preslikava to ime (kao identifikator) u fajl kao jedinstveni objekat sa identitetom (predstavljen strukturom FCB).

a)(5) Neki fajl sistem pruža sledeće operacije u svom API za tekstualne fajlove:

* FHANDLE open(char\* filename) Otvara fajl sa datim imenom.
* void close(FHANDLE) Zatvara dati fajl.
* int size(FHANDLE) Vraća trenutnu veličinu sadržaja fajla u znakovima.
* void append(FHANDLE,int) Proširuje sadržaj fajla za dati broj znakova na kraju.
* voidseek(FHANDLE,int) Postavlja kurzor datog fajla na datu poziciju (redni broj znaka počev od 0).
* void write(FHANDLE,char\*) Na poziciju kurzora datog fajla upisuje dati niz znakova, ne uključujući završni znak ‘\0’, i pomera kurzor iza upisanog niza znakova.

Napisati program koji na kraj postojećeg fajla sa imenom proba.txt upisuje sve što je uneseno preko standardnog ulaza, sve dok se na ulazu ne unese znak ’X’.Zanemariti sve potencijalne greške u ulazu/izlazu.

Rešenje:

#include <stdio.h>

void main () {
 FHANDLE output = open(„proba.txt“);
 int size = size(output);
 char c[2];
 c[1]=’\0’;
 while () {
 getc(&c[0]);
 if (c[0]==’X’) break;
 append(output,1);
 seek(output,size++);
 write(output,c);
 }
 close(output);
}

a)(5) Neki fajl sistem podržava direktorijume sa strukturom usmerenog acikličkog grafa (DAG). Posmatraju se dva različita pristupa realizaciji operacije brisanja fajla sa zadatim simboličkim imenom (stazom, referencom): brisanje fajla i svih drugih referenci prilikom bilo kog (tj. prvog) zahteva sa brisanjem (tj. preko bilo koje reference) i brisanje fajla tek pri zahtevu za brisanjem preko poslednje preostale reference (pre toga se brišu samo reference, ne i fajl). Koja varijanta je po vašem mišljenu jednostavnija za implementaciju i zbog čega?

Odgovor:

Jednostavinija je varijanta sa brisanjem fajla tek prilikom uklanjanja poslednje reference. Za to rešenje dovoljno je u FCB imati brojač referenci i ažurirati taj brojač prilikom svakog kreiranja ili brisanja reference, dok je dovoljna jednosmerna strukturna veza od referenci (u direktorijumima) prema fajlovima (tj. prema FCB), a ne i obratno. Za onu drugu varijantu potrebno je čuvati i veze u suprotnom smeru (od fajla prema referencama), kao i implementirati brisanje svih referenci prilikom brisanja fajla.

b)(10) Neki fajl sistem koristi indeksirani pristup alokaciji blokova za fajlove. U jedan indeksni blok može da stane *N* pokazivača na blokove na disku. Odrediti maksimalnu veličinu fajla (u blokovima) za svaku od sledećih varijanti organizacije indeksa:

*i*)(3) Jedan indeksni blok na koga ukazuje određeno polje u FCB.

Odgovor:*N* blokova

*ii*)(3) Ulančavanje indeksnih blokova, pri čemu na prvi indeksni blok ukazuje određeno polje u FCB, a poslednji ulaz u svakom indeksnom bloku je pokazivač na sledeći indeksni blok u lancu.

Odgovor:Nema logičkog ograničenja.

*iii*)(4) Indeksiranje u dva nivoa, s tim da na indeksni blok prvog nivoa ukazuje određeno polje u FCB, a svaki indeks prvog nivoa sadrži pokazivače na indeksne blokove drugog nivoa.

Odgovor:*N*2blokova.

a)(5) Neki fajl sistem podržava implicitno zaključavanje fajla prilikom njegovog otvaranja. Postoje dve vrste ključa: deljeni (*shared*, S), koji se traži prilikom otvaranja fajla samo za čitanje (proces koji je tako otvorio fajl ima pravo samo da čita iz fajla) i ekskluzivni (*exclusive*, X), koji se traži prilikom otvaranja fajla i za upis (proces koji je otvorio fajl ima pravo upisa). Popuniti sledeću tabelu upisivanjem oznaka onih procesa čiji će zahtev za otvaranjem istog fajla biti ispunjen, za svaki od dva data nezavisna slučaja. Procesi postavljaju zahteve redom navedenim u drugoj koloni, pri čemu oznaka npr. A-Rd označava da proces A postavlja zahtev za otvaranjem fajla za čitanje, a B-Wr da proces B postavlja zahtev za otvaranjem fajla za upis.

|  |  |  |
| --- | --- | --- |
| Slučaj  | Zahtevi za otvaranje fajla | Procesi koji su uspeli da otvore fajl |
| 1 | A-Rd, B-Rd, C-Wr, D-Rd, E-Wr | A, B, D |
| 2 | A-Wr, B-Wr, C-Rd, D-Rd, E-Rd | A |

a)(5) Neki fajl sistem podržava montiranje nekog direktorijuma fajl sistema sa nekog eksternog uređaja ili udaljenog računara u bilo koji prazan lokani direktorijum, i to na više mesta (u različite prazne lokalne direktorijume). U fajl sistemu eksternog uređaja X nalazi se direktorijum \first\second u kome su tri fajla: a.txt, b.txt i c.txt.Šta će ispisati poslednja naredba u sledećoj sekvenci naredbi (sve su izvršene uspešno; prvi argument komandi mount i copy je izvorište, drugi odredište):

mount X:\first\second \home\buzz
mount X:\first\second \home\foo
del \home\foo\a.txt
copy \home\buzz\b.txt \home\foo\d.txt
dir \home\buzz

Odgovor:

b.txt
c.txt
d.txt

a)(5) U nekom fajl sistemu struktura direktorijuma je aciklički usmereni graf (DAG). Postoje sledeće sistemske komande:

* copy src dst: kopira fajl do koga se može doći stazom src i registruje kopiju pod punim imenom (stazom) dst;
* alias src dst: registruje fajl do koga se može doći stazom src pod još jednim punim imenom (stazom) dst, bez kopiranja samog fajla.

Na donjoj slici prikazano je posmatrano početno stanje sistema; direktorijumi su prikazani kao pravougaonici sa ulazima u kojima su nazivi, a fajlovi su prikazani krugovima. Prikazati stanje ovog sistema nakon izvršavanja sledeće sekvence komandi:

copy /a/e /a/d/n
alias /a/d/n /c/i/p
copy /b /c/q
alias /b /r



Odgovor:

Sadržaji fajlova označenih sa 7 i 8 neposredno posle kopiranja su isti kao i sadržaj fajla označenog sa 3.



a)(5) Objasniti zašto se prava pristupa do fajla (prava na izvršenje određenih operacija nad fajlom) po pravilu čuvaju u tabeli otvorenih fajlova koja pripada kontekstu procesa, a ne u globalnoj (sistemskoj) tabeli otvorenih fajlova zajedničkoj za sve procese.

Odgovor:

* Po pravilu, opšta prava pristupa do fajla definišu se na nivou korisnika. Svaki proces se pokreće „u vlasništvu“ datog korisnika, pa time i nasleđuje prava pristupa do fajla podešena za tog korisnika. Zato se prava pristupa do fajla razlikuju od procesa do procesa i nisu globalna za dati fajl u celom sistemu.
* Bez obzira na opšta prava koja korisnik koji je kreirao proces ima do fajla, API za fajl sistem po pravilu omogućava da proces otvori fajl i da pri tome deklariše kako će ga koristiti (koje grupe operacija će koristiti). Ovo omogućava kontrolu grešaka u samom programu, tako što sistem zabranjuje one operacije koje nisu predviđene pri otvaranju fajla, smatrajući ih prekršajem zbog greške u programu. Zato se prava pristupa razlikuju od procesa do procesa, čak i za istog korisnika.

Jan2011

(c)(5) Šta je efekat pokretanja jednog inicijalnog procesa nad sledećim programom?

const unsigned int N = ...; // N>=1
void main () {
 for (unsigned int i=N; i>0; i--)
 if (fork()!=0) break;
}

Odgovor:

Dati kreirani proces napravi jednog svog potomka i završi se, ovaj novi proces napravi jednog svog potomka i završi se, itd., tako da ukupno biva kreirano N procesa (uključujući i početni, N>=1), pod uslovom da su svi sistemski pozivi uspešni.

b)(5) Neki računar podržava segmentno-straničnu organizaciju virtuelne memorije, pri čemu je virtuelna adresa 16-bitna, fizički adresni prostor je veličine 8GB, a adresibilna jedinica je 16‑bitna reč. Stranica je veličine 512B. Maksimalan broj segmenata u virtuelnom adresnom prostoru je 4. Prikazati logičku strukturu virtuelne i fizičke adrese i navesti širinu svakog polja.

Odgovor:

VA: Segment(2):Page(6):Offset(8); PA: Frame(24):Offset(8)

a) U nekom modularnom operativnom sistemu drajveri za raznovrsne ulazno/izlazne blokovski orijentisane uređaje registruju se kao moduli na sledeći način. Sistem održava jedinstvenu, centralizovanu tabelu registrovanih drajvera blockIOMap u kojoj svaki ulaz odgovara jednom registrovanom drajveru. Prilikom instalacije, svaki drajver u novi ulaz ove tabele upisuje adresu svoje tabele tipa BlockIOVectorTable sa adresama svojih funkcija zaduženih za odgovarajuće ulazno/izlazne operacije. Date su sledeće deklaracije:

// I/O request:
struct BlockIORequest {...};

// I/O operation:
typedef int (\*BlockIOOp) (BlockIORequest\*);

// I/O driver vector table:
struct BlockIOVectorTable {
 BlockIOOp open, close, read, write;
};

// I/O device driver map:
BlockIOVectorTable\* blockIOMap[MAXDEVICES];
extern int numOfRegisteredDrivers;
typedef int HANDLE;

a)(5) Implementirati sledeće funkcije sistema kojima se zadaju operacije uređaju datom prvim argumentom:

int open (HANDLE device, BlockIORequest\* req);
int close (HANDLE device, BlockIORequest\* req);
int read (HANDLE device, BlockIORequest\* req);
int write (HANDLE device, BlockIORequest\* req);

int open (HANDLE device, BlockIORequest\* req) {
 if (device>=numOfRegisteredDrivers) return -1;
 BlockIOVectorTable\* vt = blockIOMap[device];
 if (vt==0) return -1;
 BlockIOOp op = vt->open;
 if (op==0) return -1;
 return (\*op)(req);
}

Analogno ostale.

b)(5) Implementirati operaciju za registraciju drajvera koja vraća identifikator uređaja koji će se koristiti kao „ručka“ u daljim operacijama sa tim uređajem, a predstavlja broj ulaza u tabeli regostrovanih uređaja (-1 u slučaju greške):

HANDLE blockIODriverReg (BlockIOVectorTable\*);

HANDLE blockIODriverReg (BlockIOVectorTable\* vt) {
 if (numOfRegisteredDrivers>=MAXDEVICES) return -1;
 blockIOMap[numOfRegisteredDrivers]=vt;
 return numOfRegisteredDrivers++;
}

b)(5) Kolika je maksimalna moguća veličina fajla, ako je INDEXSIZE=1024, a tip BLKNO je 64-bitni neoznačeni ceo broj?

Odgovor: 8GB+8MB ~ 8GB

Feb2011

b)(10) Data su dva segmenta koda dva uporedna procesa. Namera je da izvršavanje svakog od ovih segmanata koda umanji odgovarajuću deljenu promenljivu a ili b za po 100, ali da se vrednosti ovih promenljivih uvek očuvaju tako da je njihov zbir veći ili jednak 0. Inicijalno nije obezbeđeno međusobno isključenje ovih segmenata.

begin begin
 if a+b>=100 then a:=a-100; if a+b>=100 then b:=b-100;
end; end;

*i*)(5) Precizno objasniti koja neregularna situacija može da nastane i kako.

 Može da dođe do neregularnog stanja u kome ne važi da je a+b>=0utrkivanjem (*race condition*) na sledeći način. Dva uporedna procesa izvršavaju svako svoj od ova dva segmenta koda. Oba uporedo pročitaju vrednosti promenljivih a i b za koje na početku važi da im je zbir veći ili jednak 100, ali je manji od 200. Zbog ispunjenja oba uslova, oba procesa uporedo izvrše umanjenje, svako svoje promenljive, za po 100. Na kraju, zbir ove dve promenljive iznosi manje od 0.

*ii*)(5) Pridružiti svakoj promenljivoj (a i b) po jedan standardni brojački semafor, a onda pomoću tih semafora sprečiti nastanak ove neregularne situacije, pri čemu treba paziti da ne nastane mrtva blokada.

 shared var sema, semb : semaphore(1);
begin begin
 sema.wait(); semb.wait(); sema.wait(); semb.wait();
 if a+b>=100 then a:=a-100; if a+b>=100 then b:=b-100;
 sema.signal(); semb.signal(); sema.signal(); semb.signal();
end; end;

(c)(5) Korišćenjem standardnih sistemskih poziva fork() i execlp() napisati program bore.txt koji, kada se nad njim pokrene proces, najpre izvršava svoju proceduru proc(), a zatim pokreće nov proces nad istim programom i ugasi se, ali tako da se taj program u procesu-detetu izvršava ispočetka. Voditi računa da izvršavanje ispočetka treba da ima početno stanje eventualnih statičkih podataka, propisno inicijalizovanih prilikom učitavanja programa, a koje procedura proc() potencijalno menja.

Rešenje:

void main () {
 proc();
 if (fork()!=0) return;
 else execlp(“bore.exe”);
}

a)(5) U nekom programu prijavljena je sledeća greška:

Error 1 error LNK2019: unresolved external symbol \_main referenced in function \_\_\_tmainCRTStartup   MSVCRTD.lib   MyProgram

Ko je prijavio ovu grešku: prevodilac, linker, ili izvršno okruženje? Precizno objasnite uzrok ove greške.

Odgovor:

Grešku je prijavio linker zato što nije pronašao simbol \_main ni u jednom .obj fajlu koje povezuje. Ovaj simbol referiše se u biblioteci koja se pominje u grešci, a koja zapravo predstavlja obavezni deo svakog povezanog i izvršivog programa u kojoj se nalazi kod za pokretanje programa (od koga počinje izvršavanje kada operativni sistem pokrene proces nad ovim programom) i poziv glavne funkcije main. Razlog greške je ili što nigde u izvornom programu nije definisana funkcija main, ili .obj fajl u kome se nalazi njen prevod nije u spisku fajlova koje linker treba da poveže.

b)(5) U nekom operativnom sistemu, kod kernela se izvršava u zaštićenom procesorskom režimu, ali i u tom režimu procesor preslikava virtuelne u fizičke adrese kroz PMT na koju ukazuje poseban registar procesora. Objasniti kako kernel može da pristupa celoj fizičkoj memoriji ovog procesora da bi mogao da održava svoje strukture podataka koje se mogu nalaziti u svim delovima fizičke memorije, pa i delovima memorije koji pripadaju korisničkim procesima (na primer, vidi zadatak pod c) za primer potrebe za ovakvim pristupom).

Odgovor:

Dok izvršava kod kernela, za preslikavanje se koristi PMT koja preslikava adresni prostor kernela 1-1 u fizički prostor, tako što stranicu broj *n* preslikava u okvir sa istim brojem *n*. Ovu PMT je inicijalizovao sam operativni sistem. Na ovaj način, kernel zapravo direktno vidi ceo fizički adresni prostor i pristupa mu kao da preslikavanja i nema.

b)(5) Koliko je blokova raspoloživo za podatke nekog fajla za koga je alocirano ukupno 3K blokova na disku (i za indeksne blokove i za blokove za podatke zajedno), ako je INDEXSIZE=1024?

Odgovor: 3⋅1023 = 3069 blokova.

Ukupan broj blokova alociran za fajl je tada:

*N* = 3⋅1023 + ⎡3⋅1023/1023⎤ = 3⋅1023 + 3 = 3⋅1024 = 3K.

Jul2011

**3.(3)** Šta je problem sledeće implementacije kritične sekcije uposlenim čekanjem?

process P1
begin
 loop
 while flag2 = true do null end; (\* Busy wait \*)
 flag1 := true;
<critical section> (\* Critical section \*)
 flag1 := false; (\* Exit protocol \*)
<non-critical section>
 end
end P1;

process P2
begin
 loop
 while flag1 = true do null end; (\* Busy wait \*)
 flag2 := true;
<critical section> (\* Critical section \*)
 flag2 := false; (\* Exit protocol \*)
<non-critical section>
 end
end P2;

Odgovor:

Nije obezbedjeno medjusobno iskljucenje.

Npr. flag1=false; flag2=false; P1 prodje while flag2=true i desi se promena konteksta zatim P2 prodje while flag1=false i nakon toga se tek postavljaju vrednosti flagova ali oba procesa ce se naci u kriticnoj sekciji.

**4.(3)** Na jeziku C++ implementirati klasu BoundedBuffer koja realizuje ograničeni bafer elemenata tipa Data kapaciteta N pomoću semafora.

Class Data;

Const int N = ...

Class BoundedBuffer{

 Semaphore mutex,itemAvailable,spaceAvailable;

 Int head,tail;

 Data\* buf[N];

Public:

BoundedBuffer();

Void append(Data\*);

Data\* take();

};

BoundedBuffer::BoundedBuffer() :mutex(1),itemAvailable(0),

spaceAvailable(N),head(0),tail(o)

{ buf = new Data[N]; }

Void BoundedBuffer::append(Data\* d){

spaceAvailable.wait();

mutex.wait();

buf[tail]=d;

tail=(tail+1)%N;

mutex.signal();

itemAvailable.signal();

}

Data\* BoundedBuffer::take(){

itemAvailable.wait();

mutex.wait();

Data\* d = buf[head];

Head = (head+1)%N;

Mutex.signal();

spaceAvailable.signal();

return d;

}

**5.(3)** Šta je osnovna razlika između tehnike dinamičkog učitavanja i tehnike preklopa (*overlays*)?

Odgovor:

Prilikom dinamickog ucitavanja, ukoliko odredjeni modul nije ucitan u memoriju loader ga ucitava u odredjenu lokaciju i postavlja njegov pokazivac u tabelu ucitanih potprograma dok prilikom preklopa pozvani modul se ucitava na isto mesto koje koristi neki drugi modul.

**6.(3)** Ukratko objasniti zašto je kod segmentne organizacije virtuelne memorije obavezna provera prekoračenja granice segmenta prilikom svakog adresiranja, a kod stanične organizacije ta provera ne postoji.

Odgovor:

**7.(3)** U nekom sistemu postoje sledeći sistemski pozivi:

int async\_write (char\* buffer);
void wait (int request\_id);

Operacija async\_write asinhrono zadaje operaciju izlaza datog niza znakova na neki izlazni uređaj i vraća interni sistemski identifikator tog zahteva (veći od 0), odnosno kod greške (manji od nula). Operacija wait suspenduje pozivajući proces sve dok operacija sa datim identifikatorom nije završena. Korišćenjem ovih sistemskih poziva, realizovati sinhroni izlaz:

int sync\_write (char\* buffer);

Rešenje:

int sync\_write(char\* buffer){

 int id = async\_write (char\* buffer);

if (id<0) return id;

wait(id);

return 0; // sta vraca prilikom uspesnog poziva

**8.(3)** Ukratko objasniti kako se u Unix fajl sistemu definišu prava pristupa do fajla.

Odgovor:

Uz pomoc tri bita *RWE* za vlasnika,grupu i ostale ova tri bita definisu prava pristupa.

**9.(3)** Navestirazlog zašto bi neki fajl sistem koristio klastere na disku različite veličine.

Odgovor:

Radi smanjenja interne fragmentacije.

**10.(3)** Ukratko objasniti šta je inkrementalni, a šta totalni bekap (*backup*) fajl sistema?

Odgovor:

Inkrementalni pamti samo promene od zadatog datuma i vremena a kompletni kopira sve.

Jun2011

**1.(3)** Dati kratke definicije multiprocesorskog sistema i distribuiranog sistema.

Multiprocesorski sistemi predstavljaju sistem sa više procesora sa deljenom memorijom;

Distribuirani sistemi predstavljajuskup procesora bez zajedničke memorije, povezanih komunikacionom mrežom. *(specijalizovani homogeni distribuirani sistemi, lokalne mreže, mreže šireg područja (Wide Area Network, WAN), Internet)*

**2.(3)** Koji je razlog deobe procesorskog vremena (*time sharing*) kod interaktivnih sistema? (Zaokružiti odgovor)

a) Brži odziv na svaku akciju svakog korisnika nego bez deobe vremena.

b) Ravnomerniji odziv na akcije više korisnika nego bez deobe vremena.

c) Mogućnost podrške više korisnika nego bez deobe vremena.

**3.(3)** Zašto je stek deo konteksta izvršavanja programa?

Zato sto se sve lokalne promenljive i „trag“ izvrsavanja programa nalaze na steku.

**4.(3)** Napisati deo koda koji obezbeđuje kritičnu sekciju pomoću brojačkih semafora, uz dozvolu konkurentnog izvršavanja kritične sekcije do N uporednih procesa.

shared var: sem(N)

begin

...

Wait(sem);

<critical section>

Signal(sem);

...

end

**5.(3)** Korišćenjem sistemskih poziva fork() i execlp() napisati C program koji kreira proces nad programom u fajlu čije je ime zadato argumentom komandne linije.

#include <stdio.h> // sta se desava sa rotideljskim procesom?

int main (int argc, char\* argv[]) {

 char\* program = argv[2];
 int pid = fork();
 if (pid<0) exit(-1);
 if (pid==0) execlp(program);
 }
 exit(0);
}

**6.(3)** Memorija nekog računara organizovana je stranično, sa stranicom veličine 4KB. Adresibilna jedinica je bajt, a virtuelna adresa je 32-bitna. Fizička adresa je veličine 32 bita. Ako je PMT organizovana u dva nivoa, s tim da su veličine polja za broj ulaza u tabele oba nivoa isti, kolika je veličina (u bajtovima) PMT prvog nivoa?

**7.(3)** Da li linker prilikom povezivanja sa ciljem dobijanja izvršnog programa (.exe) pravi razliku između fajlova tipa .obj i tipa .lib?

NE, .lib ima isto znacenje i oblik kao .obj sem sto je nastao prevodjenjem i povezivanjem vise izvornih fajlova.

**8.(3)** Kojom tehnikom se znakovno orijentisani ulazni uređaj može učiniti logički (virtuelno) blokovski orijentisanim?

Tehnikom baferisanja.

**2.(3)** Na asembleru nekog zamišljenog RISC procesora sa LOAD/STORE arhitekturom napisati program koji vrši učitavanje bloka podataka sa ulaznog uređaja tehnikom programiranog ulaza korišćenjem prekida. Samostalno usvojiti potrebne detaljne pretpostavke.

LOAD R1, #CNT
LOAD R2, #ADR
LOAD R0, #0
STORE [CTRL], #0...01b

wait: CMP R0, #1
JNZ wait
...

INTR: PUSH R3
LOAD R3, [R2]
STORE [DATA], R3
INC R2
DEC R1
JNZ back
LOAD R0, #1
STORE [CTRL], #0...00b
back: POP R3
RTI

**4.(3)** Korišćenjem standardnih brojačkih semafora u školskom jezgru, na jeziku C++ u potpunosti realizovati klasu za ograničeni bafer (*bounded buffer*).