Elektrotehnički fakultet u Beogradu
Katedra za računarsku tehniku i informatiku

Predmet: Operativni sistemi 1
Nastavnik: prof. dr Dragan Milićev
Odsek: Softversko inženjerstvo
Kolokvijum: Prvi, mart 2024.
Datum: 20. 3. 2024.

Prvi kolokvijum iz Operativnih sistema 1

Kandidat: _____

Broj indeksa: _____ *E-mail:*_____

Kolokvijum traje 90 minuta. Dozvoljeno je korišćenje literature.

Zadatak 1 _____/10 *Zadatak 3* _____/10
Zadatak 2 _____/10

Ukupno: _____/30 = _____% = _____/15

Napomena: Ukoliko u zadatku nešto nije dovoljno precizno definisano, student treba da uvede razumnu pretpostavku, da je uokviri (da bi se lakše prepoznala prilikom ocenjivanja) i da nastavi da izgrađuje preostali deo svog odgovora na temeljima uvedene pretpostavke. Ocenjivanje unutar potpitanja je po sistemu "sve ili ništa", odnosno nema parcijalnih poena. Kod pitanja koja imaju ponuđene odgovore treba **samo zaokružiti** jedan odgovor. Na ostala pitanja odgovarati **čitko, kratko i precizno**.

1. (10 poena)

Dat je sadržaj dva asemblerska fajla (`a.asm` i `b.asm`) za 32-bitni procesor picoRISC. Direktivom `export` asembleru se nalaže da navedene simbole (definisane dalje u tom fajlu) „izveze“ kao simbole dostupne za spoljno vezivanje, dok se direktivom `import` „uvoze“ navedeni simboli definisani u nekom drugom fajlu (ovom direktivom ti simboli su deklarisani u datom fajlu, ali sa nepoznatom vrednošću). Nakon prevođenja (asembliranja), linkeru se zadaje da spoji dva fajla dobijena prevođenjem fajlova `a.asm` i `b.asm`, tim redom. Linker slaže segmente u virtuelni adresni prostor redom kojim na njih nailazi počev od virtuelne adrese 0, s tim da svaki nov segment počinje od početka nove stranice veličine 4 KB (adresibilna jedinica je bajt).

a.asm

```
export main, a
import max

seg data
    a dd 1, 2, 3, 4, 5
endseg

seg text
    main:   load r1,a
            push r1
            load r1,a+4
            push r1
            call max
            pop  r1
            pop  r1
            ret
endseg
```

b.asm

```
export max

seg text
    max:    load r1,[sp+8]
            load r2,[sp+12]
            cmp  r1,r2
            jle  L0001
            load r0,r1
            ret
    L0001:  load r0,r2
            ret
endseg
```

a)(3) Koje virtuelne adrese dodeljuje linker simbolima `main`, `a` i `max` tokom povezivanja (napisati ih heksadecimalno)? Obrazložiti odgovor.

b)(3) Ukoliko instrukcija `call` na vrh steka stavlja samo PC, koje tri 32-bitne vrednosti se nalaze na vrhu steka prilikom ulaska u potprogram `max` pri izvršavanju procesa pokrenutog nad navedenim povezanim programom? Vrednosti navesti heksadecimalno u poretku kojim su stavljene na stek – poslednju onu na vrhu steka.

c)(4) Ukoliko se primenjuje stranična organizacija memorije, kolika je interna fragmentacija za sadržaj ova dva fajla (koliko je neiskorišćene memorije koja je alocirana za logičke segmente u ovim fajlovima)? Obrazložiti odgovor.

Rešenje:

2. (10 poena)

Neki sistem primenjuje kontinualnu alokaciju memorije, ali u jedinicama koje su blokovi memorije određene konstantne veličine (alokacija se uvek radi u delovima veličine zaokružene na cele blokove i poravnato na blokove). Za evidenciju slobodnih i zauzetih blokova memorije sistem koristi posebnu strukturu zapisanu u nizu `memMap` veličine `NumOfBlocks` (ukupan broj blokova dostupnih za alokaciju). Svaki element ovog niza odgovara jednom bloku memorije koja se evidentira ovim nizom: elementi ovog niza redom, jedan na jedan odgovaraju blokovima memorije koji se alociraju. Za određeni slobodan fragment memorije koji počinje u bloku sa rednim brojem `b` i koji je veličine `k` susednih blokova, element `memMap[b]` ima vrednost `k`, dok narednih `k-1` elemenata ovog niza ima vrednost 0. Slično, za zauzet segment memorije koji počinje u bloku sa rednim brojem `b` i ima veličinu `k` susednih blokova, element `memMap[b]` ima vrednost `-k` (negativna vrednost), dok narednih `k-1` elemenata ima vrednost 0. Inicijalno je element sa indeksom 0 ovog niza jednak `NumOfBlocks`, dok su svi ostali elementi niza jednaki 0 (cela memorija je slobodna).

Implementirati funkciju `alloc` koja treba da alocira deo memorije zadate veličine `size` susednih blokova po *first fit* algoritmu i vrati redni broj prvog bloka koji je alociran, a u slučaju da slobodnog prostora nema, vraća -1.

```
const int NumOfBlocks = ...;
int memMap[NumOfBlocks] = {NumOfBlocks};
int alloc (int size);
```

Rešenje:

3. (10 poena)

Neki sistem sa straničnom organizacijom memorije i alokacijom stranica na zahtev (*demand paging*) omogućava logičko deljenje (logičkih) segmenata virtuelne memorije, ali tako da najviše dva procesa mogu da dele jedan segment. U svakom od tih procesa deljeni logički segment može da zauzima različite pozicije u virtuelnom adresnom prostoru (ali iste veličine). Segment jednog procesa opisan je objektom klase `SegDesc`. PMT procesa kom pripada dati segment dat je u polju `pmt`, dok je broj prve stranice tog segmenta dat u polju `startPage`. Ukoliko je segment deljen, segment drugog procesa sa kojim je dati segment deljen dat je u polju `sharedSeg` (ima vrednost *null* ako segment nije deljen).

Data polimorfna operacija `SegDesc::initPage` po potrebi učitava, odnosno inicijalizuje datu stranicu u dati okvir prethodno alociran za tu stranicu (ona ne menja PMT), u zavisnosti od vrste segmenta; ova funkcija vraća status izvršenja (0 za uspeh). PMT je predstavljen klasom `PMT`, čija funkcija `getFrame` vraća broj okvira za datu stranicu (0 ako stranica nije alocirana), a funkcija `setFrame` postavlja broj okvira za datu stranicu u PMT.

Implementirati funkciju `SegDesc::allocPage` koja se poziva prilikom obrade stranične greške i koja treba da obezbedi da data stranica za dati PMT procesa koji je izazvao straničnu grešku bude alocirana, učitana u memoriju i postavljen njen ulaz u PMT; uzeti u obzir slučaj kada je stranica deljena i već alocirana od strane procesa sa kojim je deljena. Provera da data stranica pripada datom segmentu je već obavljena. Pretpostavlja se da su polja za prava pristupa u PMT uvek inicijalizovana za sve stranice alociranih segmenata, čak i ako stranice nisu alocirane. Ova operacija treba da vraća status izvršenja (0 za uspeh).

```
typedef Frame uint16;
typedef Page uint32;
Frame getFreeFrame ();

class SegDesc {
public:
  int allocPage (Page page);
  ...
protected:
  virtual int initPage (Page, Frame);
  ...
private:
  Page startPage;
  PMT* pmt;
  SegDesc* sharedSeg = nullptr;
  ...
};

class PMT {
public:
  Frame getFrame (Page) const;
  void  setFrame (Page, Frame);
  ...
};
```

Rešenje:

Prvi kolokvijum iz Operativnih sistema 1
Odsek za softversko inženjerstvo
Mart 2024.

1. (10 poena)

a)(3) Sva tri logička segmenta su vrlo mala, očigledno manja od jedne stranice (videti odgovor pod c). Sva tri navedena simbola se preslikavaju u relativne pomeraje 0 svojih segmenta, tako da se simboli preslikavaju u sledeće virtuelne adrese:
`a`: 0, `main`: 0x1000 (4K), `max`: 0x2000 (8K)

b)(3) Pri pozivu potprograma `max`, pozivalac (`main`) na vrh steka stavlja sadržaj prva dva 32-bitna elementa niza `a`, tj. 1 i 2, a potom vrednost PC-a tokom izvršavanja instrukcije `call`. Tako se na vrhu steka nalaze redom vrednosti: 1, 2, 0x1020 (adresa prve instrukcije iza instrukcije `call` je 4K+32, jer instrukcije zaključno sa instrukcijom `call` zauzimaju 8 32-bitnih reči).

c)(4) Prvi segment (`data`) koristi pet 32-bitnih reči (20 bajtova). Drugi segment (`text`) koristi 11 32-bitnih reči (44 bajta). Treći segment takođe koristi 11 reči (44 bajta). Ukupno je iskorišćeno 108 bajtova od tri cele stranice (12KB). Interna fragmentacija uključuje sav ovaj neiskorišćen prostor veličine 12KB - 108B.

2. (10 poena)

```
int alloc (int size) {
  if (size<=0 || size>NumOfBlocks) return -1; // Exception
  for (int i=0; i<NumOfBlocks; i++)
    if (memMap[i]>=size) {
      if (memMap[i]>size) memMap[i+size] = memMap[i]-size;
      memMap[i] = -size;
      return i;
    };
  return -1;  // No free mem
}
```

Moguće je učiniti ovu implementaciju efikasnijom, tako da preskače sve blokove koji pripadaju istom zauzetom ili slobodnom segmentu čim detektuje prvi – ostavlja se čitaocu.

3. (10 poena)

```
int SegDesc::allocPage (Page page) {
  Frame frame = 0;
  if (this->sharedSeg) {
    Page otherPage = page - this->startPage + this->sharedSeg->startPage;
    frame = this->sharedSeg->pmt->getFrame(otherPage);
  }
  int status = 0;
  if (!frame) {
    frame = getFreeFrame();
    status = this->initPage(page,frame);
    if (!status) return status;
  }
  this->pmt->setFrame(page,frame);
  return status;
}
```

Elektrotehnički fakultet u Beogradu
Katedra za računarsku tehniku i informatiku

Predmet: Operativni sistemi 1
Nastavnik: prof. dr Dragan Milićev
Odsek: Računarska tehnika i informatika, Softversko inženjerstvo
Kolokvijum: Drugi, avgust 2023.
Datum: 27. 8. 2023.

Drugi kolokvijum iz Operativnih sistema 1

Kandidat: _____

Broj indeksa: _____ *E-mail:*_____

Kolokvijum traje 90 minuta. Dozvoljeno je korišćenje literature.

Zadatak 1 _____/10 *Zadatak 3* _____/10
Zadatak 2 _____/10

Ukupno: _____/30 = _____% = _____/15

Napomena: Ukoliko u zadatku nešto nije dovoljno precizno definisano, student treba da uvede razumnu pretpostavku, da je uokviri (da bi se lakše prepoznala prilikom ocenjivanja) i da nastavi da izgrađuje preostali deo svog odgovora na temeljima uvedene pretpostavke. Ocenjivanje unutar potpitanja je po sistemu "sve ili ništa", odnosno nema parcijalnih poena. Kod pitanja koja imaju ponuđene odgovore treba **samo zaokružiti** jedan odgovor. Na ostala pitanja odgovarati **čitko, kratko i precizno**.

1. (10 poena)

a)(7) Korišćenjem Unix sistemskih poziva *fork, exit* i *wait/waitpid* implementirati funkciju `max` koja pronalazi maksimum celobrojnih vrednosti `val` sadržanih u čvorovima binarnog stabla predstavljenih strukturom `Node`, tako što proces koji poziva ovu funkciju sa argumentom koji ukazuje na koren datog stabla obrađuje taj koren i njegovo levo podstablo, a za obradu desnog podstabla kreira poseban proces koja teče uporedo sa obradom levog podstabla, i tako dalje rekurzivno za svako podstablo na isti opisani način. Ignorisati greške.

b)(3) Ako se ova funkcija pozove za koren potpunog balansiranog binarnog stabla sa *n* nivoa i 2^n - 1čvorova, koliko ukupno procesa obrađuje ovo stablo, uključujući i početni koji poziva ovu funkciju za koren ovog stabla? Precizno obrazložiti odgovor.

```
struct Node {
  Node *left, *right;
  int val;
};

int max (Node* nd);
```

Rešenje:

2. (10 poena)

a)(7) Korišćenjem standardnih funkcija `setjmp` i `longjmp` u školskom jezgru implementirati sledeće dve sistemske usluge (precizno navesti sva eventualna potrebna proširenja i napisati implementacije ovih funkcija):

- `void Thread::suspend()`: statička funkcija članica klase `Thread` koja bezuslovno suspenduje pozivajuću (tekuću) nit.
- `void Thread::resume()`: nestatička funkcija članica klase `Thread` koja reaktivira iz suspenzije nit za koju je pozvana, ukoliko je ta nit suspendovana sa `suspend()`; u suprotnom nema nikakvog efekta.

b)(3) Opisane usluge iskorišćene su za uslovnu sinhronizaciju dve niti na sledeći način:

- nit na čiji objekat klase `Thread` ukazuje pokazivač `t` i koja čeka na uslov izvršava:

  ```
  if (!condition) Thread::suspend();
  ```

- nit koja signalizira ispunjenje uslova izvršava:

  ```
  t->resume();
  ```

Da li je navedena upotreba ovih usluga ispravna? Precizno obrazložiti odgovor.

Rešenje:

3. (10 poena)

Klasa `Data` predstavlja neku strukturu podataka i poseduje konstruktor kopije. Objekti ove klase mogu se praviti operatorom `new` i brisati operatorom `delete`. Pokazivač `sharedData` je deljen između uporednih niti i ukazuje na deljeni objekat ove klase. Klasa `OptimisticCCtrl` implementira optimistički pristup kontroli konkurentnosti nad objektom na koga ukazuje deljeni pokazivač i namenjena je da se koristi na dole dat način. Svaki pokušaj transakcije izmene deljenog objekta mora da se započne pozivom operacije `startTrans` kojoj se dostavlja adresa pokazivača na deljeni objekat klase `Data`. Ova operacija vraća pokazivač na kopiju objekta nad kojim transakcija može da radi izmene (upis). Na kraju treba pozvati operaciju `commit` koja vraća `true` ako je uspela, `false` ako je detektovan konflikt; u ovom drugom slučaju transakcija mora da se pokuša iznova sve dok konačno ne uspe.

```
Data* sharedData = ...
OptimisticCCtrl* ctrl = new OptimisticCCtrl();

bool committed = false;
do {
  Data* myCopy = ctrl->startTrans(&sharedData);
  myCopy->write(...); // Write to *myCopy
  committed = ctrl->commit();
while (!committed);
```

Implementirati klasu `OptimisticCCtrl` čiji je interfejs dat dole. Na raspolaganju je sistemska funkcija `cmp_and_swap` koja radi atomičnu proveru jednakosti vrednosti pokazivača `*shared` i `read` i ako su oni isti, u `*shared` upisuje vrednost `copy`.

```
class OptimisticCCtrl {
public:
  OptimisticCCtrl ();

  Data* startTrans (Data** shared);
  bool  commit ();

};

bool cmp_and_swap(void** shared, void* read, void* copy);
```

R
e
š
e
n
j

Drugi kolokvijum iz Operativnih sistema 1
Avgust 2023.

1. (10 poena)

a)(7)

```
int max (Node* nd) {
  int m = nd->val, m1;
  if (nd->right)
    if (fork()==0)
      exit(max(nd->right));
  if (nd->left) {
    m1 = max(nd->left);
    if (m1>m) m = m1;
  }
  if (nd->right) {
    wait(&m1);
    if (m1>m) m = m1;
  }
  return m;
};
```

b)(3) Prvi nivo (koren stabla) obrađuje jedan, inicijalni proces. Drugi nivo, osim njega, obrađuje još jedan proces, njegovo dete. Svaki nivo obrađuju svi procesi koji obrađuju prethodni nivo i još toliko novih procesa njihove dece, odnosno dvostruko više (nijedan se ne gasi). Tako n-ti nivo obrađuju svi kreirani procesi, njih 2^{n-1}.

2. (10 poena)

a)(7) U klasu `Thread` dodaje se privatni podatak član `isSuspended` tipa `bool` inicijalizovan na `false`.

```
void Thread::suspend () {
  lock();
  runningThread->isSuspended = true;
  if (setjmp(runningThread->context)==0) {
    runningThread = Scheduler::get();
    longjmp(runningThread->context,1);
  }
  unlock();
}

void Thread::resume () {
  lock();
  if (this->isSuspended) {
    this->isSuspended = false;
    Scheduler::put(this);
  }
  unlock();
}
```

b)(3) Opisano rešenje ima problem utrkivanja (*race condition*) koje se može dogoditi na sledeći način: uslov nije ispunjen; nit koja treba da čeka na uslov ispituje uslov i zaključuje da treba da se suspenduje, ali pre nego što to uradi nit koja signalizira uslov uradi *resume* bez efekta, pa se nakon toga prva nit bespotrebno suspenduje potencijalno neograničeno.

3. (10 poena)

```
class OptimisticCCtrl {
public:
```

```
  OptimisticCCtrl () {}

  Data* startTrans (Data** original);
  bool  commit ();
private:
  Data **shared = 0, *read = 0, *copy = 0;
};

Data* OptimisticCCtrl::startTrans (Data** original) {
  shared = original;
  read = *original;
  return copy = new Data(*read);
}

bool OptimisticCCtrl::commit () {
  bool ret = cmp_and_swap(shared,read,copy);
  if (!ret) delete copy;
  return ret;
}
```

Elektrotehnički fakultet u Beogradu
Katedra za računarsku tehniku i informatiku

Predmet: Operativni sistemi 1
Nastavnik: prof. dr Dragan Milićev
Odsek: Softversko inženjerstvo, Računarska tehnika i informatika
Kolokvijum: Drugi, maj 2023.
Datum: 7. 5. 2023.

Drugi kolokvijum iz Operativnih sistema 1

Kandidat: _____

Broj indeksa: _____ *E-mail:*_____

Kolokvijum traje 90 minuta. Dozvoljeno je korišćenje literature.

Zadatak 1 _____/10 *Zadatak 3* _____/10
Zadatak 2 _____/10

Ukupno: _____/30 = _____% = _____/15

Napomena: Ukoliko u zadatku nešto nije dovoljno precizno definisano, student treba da uvede razumnu pretpostavku, da je uokviri (da bi se lakše prepoznala prilikom ocenjivanja) i da nastavi da izgrađuje preostali deo svog odgovora na temeljima uvedene pretpostavke. Ocenjivanje unutar potpitanja je po sistemu "sve ili ništa", odnosno nema parcijalnih poena. Kod pitanja koja imaju ponuđene odgovore treba **samo zaokružiti** jedan odgovor. Na ostala pitanja odgovarati **čitko, kratko i precizno**.

1. (10 poena)

U nekom sistemu nalik sistemu Unix postoji sistemski poziv `signal` čiji je potpis dat dole.[1] Ovim pozivom proces može da preusmeri obradu signala datog prvim parametrom na rutinu (*signal handler*) datu drugim parametrom. Ukoliko uspe, ovaj sistemski poziv vraća pokazivač na rutinu koja je bila postavljena za dati signal pre ove promene, dok u slučaju neuspeha vraća `SIG_ERR`. Proces dete nasleđuje rutine za obradu signala od procesa roditelja, ali može nezavisno da ih redefiniše.

Koristeći sistemske pozive *fork, exit, signal* i *kill*, kao i funkcije *printf* i *getchar*, napisati C program koji, kada se nad njim pokrene (roditeljski) proces, pokrene jedan proces dete, a onda taj proces dete gasi slanjem signala `SIGTERM`. Proces dete izvršava praznu petlju sve dok ne stigne ovaj signal. Kada primi signal za gašenje `SIGTERM`, proces dete treba da postavi pitanje korisniku ispisom na standardni izlaz rečenice „Da li ste sigurni da želite da prekinete izvršavanje? (D/N)“. Ukoliko se na standardni ulaz procesa deteta unose znak 'D' ili 'd', proces dete treba da se ugasi, a u suprotnom da nastavi da izvršava praznu petlju i tako dalje. Da bi se izbeglo utrkivanje u slučaju da proces roditelj pošalje signal pre nego što je proces dete preusmerio rutinu za njegovu obradu, proces roditelj treba najpre da preusmeri svoju rutinu za ovaj signal, potom pokrene proces dete koje će to naslediti, a onda sebi preusmeru obradu ovog signala na staru rutinu. Sve greške koje onemogućavaju dalje funkcionisanje kako je opisano obraditi ispisom poruke o grešci i završetkom procesa.

```
typedef void (*SIGHANDLER) (int);
SIGHANDLER signal (int signal, SIGHANDLER newHandler);
```

Rešenje:

[1]Ovo je stari i prevaziđen Unix sistemski poziv. Noviji oblik je nešto složeniji i robusniji, ali principijelno sličan.

2. (10 poena)

Neki procesor pri obradi prekida, sistemskog poziva i izuzetka prelazi na sistemski stek. Taj stek alociran je u delu memorije koju koristi kernel, a na vrh tog steka ukazuje poseban registar SSP procesora koji je dostupan samo u privilegovanom režimu.

Prilikom obrade ovih situacija procesor ništa ne stavlja na stek, već zatečenu, neizmenjenu vrednost registra PC kog je koristio prekinuti proces sačuva u poseban, za to namenjen registar SPC (procesor nema PSW), a vrednosti registara SP i SSP međusobno zameni (*swap*). Za pristup steku procesor tako uvek koristi SP. Prilikom povratka iz prekidne rutine instrukcijom `iret` procesor radi inverznu operaciju. Procesor je RISC, sa *load/store* arhitekturom i ima 32 registra opšte namene (R0..R31).

Kernel je višenitni, a svakom toku kontrole (procesu ili niti), uključujući i niti kernela, pridružen je poseban stek koji se koristi u sistemskom režimu. Promenu konteksta u kernelu radi data operacija `yield`. Kontekst procesora kernel čuva na steku, dok se informacija o vrhu steka čuva u polju `PCB::sp` čiji je pomeraj u odnosu na početak strukture PCB u asembleru označen istoimenom simboličkom konstantom. U kodu kernela postoji statički pokazivač `oldRunning` koji ukazuje na PCB tekućeg procesa, kao i pokazivač `newRunning` koji ukazuje na PCB procesa koji je izabran za izvršavanje.

Napisati kod funkcije `initContext` koju koristi kernel kada inicijalizuje procesorski kontekst za proces sa datim PCB-om tako da ovaj proces može dobiti procesor funkcijom `yield`. Na vrh već alociranog steka procesa (prvu praznu lokaciju) koji se koristi u neprivilegovanom režimu ukazuje `usrStk`, dok na vrh već alociranog steka procesa koji se koristi u kernelu ukazuje `krnlStk`, a početna adresa programa je `startAddr`. Stek raste ka nižim adresama. Svi pokazivači su 32-bitni, a tip `uint32` predstavlja 32-bitni ceo neoznačen broj.

```
void yield () {
  asm {
  ; Save the current context
  push      spc ; save regs on the process stack
  push      ssp
  push      r0
  push      r1
  ...
  push      r31
  load      r0, oldRunning ; r0 now points to the running PCB
  store     sp, [r0+PCB::sp] ; save SP

  ; Restore the new context
  load      r0, newRunning
  load      sp, [r0+PCB::sp] ; restore SP
  pop       r31 ; restore regs
  ...
  pop       r0
  pop       ssp
  pop       spc
  }
}
```

```
void initContext (PCB* pcb, void* usrStk, void* krnlStk, void* startAddr);
```

Rešenje:

3. (10 poena)

U školskom jezgru promenu konteksta obavlja operacija `yield` data dole. Korišćenjem ove funkcije za promenu konteksta implementirati binarni semafor za međusobno isključenje (*mutex*) klasom `Mutex` sa sledećim ograničenjima:

- Operaciju `wait` ne sme da pozove nit koja je zaključala *mutex*.
- Operaciju `signal` sme da pozove samo nit koja je zaključala *mutex*.

Ukoliko neko od navedenih ograničenja nije zadovoljeno treba baciti celobrojni izuzetak `ERR_MUTEX`.

```
void yield () {
  Thread* oldt = Thread::running;
  Thread* newt = Thread::running = Scheduler::get();
  if (setjmp(oldt->context)==0) longjmp(newt->context,1);
}
```

R
e
š
e
n
j

Drugi kolokvijum iz Operativnih sistema 1
Maj 2023.

1. (10 poena)

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define handle_error(msg) \
  do { printf("Error: %s\n",msg); exit(-1); } while(0)

typedef void (*SIGHANDLER) (int);

volatile int complete = 0;

void handler (int) { complete = 1; }

int main () {
  SIGHANDLER oldh = signal(SIGTERM,handler);
  if (oldh==SIG_ERR) handle_error("Cannot set the signal handler.");

  pid_t pid = fork();

  if (pid<0) handle_error("Cannot create the child process.");

  else if (pid==0)
    while (1) {
      while (!complete);
      printf("Are your sure you want to exit? (y/n)");
      int c = getchar();
      if (c=='Y' || c=='y') exit(0);
      complete = 0;
    };

  else {
    signal(SIGTERM,oldh);
    kill(pid,SIGTERM);
  }
}
```

2. (10 poena)

```
void initContext (PCB* pcb, void* usrStk, void* krnlStk, void* startAddr) {
  uint32* kst = (uint32*)krnlStk;
  *(kst-0) = (uint32)startAddr;
  *(kst-1) = (uint32)usrStk;
  pcb->sp = kst-34;
}
```

3. (10 poena)

```
class Mutex {
public:
  Mutex () : holder(0) {}

  void wait ();
  void signal ();

private:
  Queue blocked;
  Thread* holder;
};

void Mutex::wait () {
  lock();
  if (Thread::running == holder) { unlock(); throw ERR_MUTEX; }
  if (holder==0)
    Scheduler::put(holder = Thread::running);
  else {
    blocked.put(Thread::running);
  }
  yield();
  unlock();
}

void Mutex::signal () {
  lock();
  if (Thread::running != holder) { unlock(); throw ERR_MUTEX; }
  Thread* t = blocked.get();
  holder = t;
  if (t) Scheduler::put(t);
  unlock();
}
```

Elektrotehnički fakultet u Beogradu
Katedra za računarsku tehniku i informatiku

Predmet: Operativni sistemi 1
Nastavnik: prof. dr Dragan Milićev
Odsek: Softversko inženjerstvo, Računarska tehnika i informatika
Kolokvijum: Drugi, jun 2022.
Datum: 12. 6. 2022.

Drugi kolokvijum iz Operativnih sistema 1

Kandidat: _____

Broj indeksa: _____ *E-mail:*_____

Kolokvijum traje 1,5 sat. Dozvoljeno je korišćenje literature.

Zadatak 1 _____/10 *Zadatak 3* _____/10
Zadatak 2 _____/10

Ukupno: _____/30 = _____% = _____/15

Napomena: Ukoliko u zadatku nešto nije dovoljno precizno definisano, student treba da uvede razumnu pretpostavku, da je uokviri (da bi se lakše prepoznala prilikom ocenjivanja) i da nastavi da izgrađuje preostali deo svog odgovora na temeljima uvedene pretpostavke. Ocenjivanje unutar potpitanja je po sistemu "sve ili ništa", odnosno nema parcijalnih poena. Kod pitanja koja imaju ponuđene odgovore treba **samo zaokružiti** jedan odgovor. Na ostala pitanja odgovarati **čitko, kratko i precizno**.

1. (10 poena)

Koristeći samo sistemske pozive *fork*, *wait/waitpid* i *exit*, kao i funkciju *printf*, napisati C program koji pronalazi maksimalnu vrednost svih elemenata ogromne matrice dimenzije *M* puta *N* elemenata tipa `double` tako što uporedo pronalazi maksimum u svakoj vrsti matrice (maksimum svake vrste pronalazi u po jednom od pokrenutih procesa-dece), a onda pronalazi maksimum tih maksimuma. U slučaju greške, ovaj program treba da ispiše poruku o grešci i vrati status -1, a u slučaju uspeha treba da ispiše pronađeni maksimum i vrati status 0. Pretpostaviti da je matrica već nekako inicijalizovana.

```
const int M = ..., N = ...;
extern double mat[M][N];
```

Rešenje:

2. (10 poena)

U školsko jezgro dodaje je se koncept *uslova* (engl. *condition*) za uslovnu sinhronizaciju, podržan klasom `Condition` čiji je interfejs dat dole. Uslov može biti ispunjen ili neispunjen; inicijalna vrednost zadaje se konstruktorom. Niti koje smeju da nastave izvršavanje iza neke tačke samo ako je uslov ispunjen treba da pozovu operaciju `wait`, koja ih suspenduje ako uslov nije ispunjen. Bilo koja nit koja ispuni uslov to objavljuje pozivom operacije `set`; sve niti koje čekaju na taj uslov tada nastavljaju izvršavanje. Kada neka nit promeni uslov tako da on više nije ispunjen, treba da pozove operaciju `clear`. Implementirati klasu `Condition`.

```
class Condition {
public:
  Condition (bool init = false);
  void set ();
  void clear ();
  void wait ();
};
```

Rešenje:

3. (10 poena)

Više uporednih niti-pisaca upisuje izračunate celobrojne dvodimenzionalne koordinate (x, y) na koje treba pomeriti robota u deljeni objekat klase `SharedCoord` čiji je interfejs dat dole; svaka ovakva nit nezavisno upisuje svoj par izračunatih koordinata pozivom operacije `write` ove klase. Jedna nit-čitalac periodično očitava par koordinata iz tog deljenog objekta i pomera robota na očitane koordinate; ova nit to radi pozivom operacije `read` ove klase, nezavisno od pisaca, svojim tempom, tako da svaki put čita poslednje upisane koordinate (može pročitati više puta isti par koordinata ili neke izračunate koordinate i preskočiti).

Implementirati klasu `SharedCoord` uz neophodnu sinhronizaciju korišćenjem najmanjeg broja semafora školskog jezgra.

```
class SharedCoord {
public:
  SharedCoord ();
  void read (int& x, int& y);
  void write (int x, int y);
};
```

Rešenje:

Drugi kolokvijum iz Operativnih sistema 1
Jun 2022.

1. (10 poena)

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

#define handle_error(msg) do { \
  printf(msg); \
  exit(-1); \
} while (0)

const int M = ..., N = ...;
extern double mat[M][N];

int max (int i) {
  double m = mat[i][0];
  int mj = 0;
  for (int j=1; j<N; j++)
    if (mat[i][j]>m)
      m = mat[i][j], mj = j;
  return mj;
}

pid_t pids[M];

int main () {

  for (int i=0; i<M; i++) {
    pid_t pid = pids[i] = fork();
    if (pid<0) handle_error("Error: Cannot create a child process.\n");
    if (pid==0)
      exit(max(i));
  }

  double max;
  int mj;
  if (waitpid(pids[0],&mj)<0)
    handle_error("Error waiting a child process.\n");
  max = mat[0][mj];

  for (int i=1; i<M; i++) {
    if (waitpid(pids[i],&mj)<0)
      handle_error("Error waiting a child process.\n");
    if (mat[i][mj]>max) max = mat[i][mj];
  }
  printf("Max: %f\n",max);
  exit(0);
}
```

2. (10 poena)

```
class Condition {
public:
  Condition (bool init = false) : cond(init) {}

  void set ();
  void clear () { lock(); cond = false; unlock(); }
  void wait ();

private:
  bool cond;
  Queue blocked;
};

void Condition::wait () {
  lock();
  if (!cond)
    if (setjmp(Thread::runningThread->context)==0) {
      blocked.put(Thread::runningThread);
      Thread::runningThread = Scheduler::get();
      longjmp(Thread::runningThread->context,1);
    }
  unlock();
}


void Condition::set () {
  lock();
  cond = true;
  for (Thread* t = blocked.get(); t; t = blocked.get())
    Scheduler::put(t);
  unlock();
}
```

3. (10 poena)

```
#include "kernel.h"

class SharedCoord {
public:
  SharedCoord ();

  void read (int& x, int& y);
  void write (int x, int y);

private:
  Semaphore mutex;
  int x, y;
};

inline SharedCoord () : mutex(1) {}

inline void SharedCoord::read (int& x_, int& y_) {
  mutex.wait();
  x_ = this->x;
  y_ = this->y;
  mutex.signal();
}

inline void SharedCoord::write (int x_, int y_) {
  mutex.wait();
  this->x = x_;
  this->y = y_;
  mutex.signal();
}
```

Elektrotehnički fakultet u Beogradu
Katedra za računarsku tehniku i informatiku

Predmet: Operativni sistemi 1
Nastavnik: prof. dr Dragan Milićev
Odsek: Softversko inženjerstvo, Računarska tehnika i informatika
Kolokvijum: Drugi, maj 2022.
Datum: 8. 5. 2022.

Drugi kolokvijum iz Operativnih sistema 1

Kandidat: _____

Broj indeksa: _____ *E-mail:*_____

Kolokvijum traje 1,5 sat. Dozvoljeno je korišćenje literature.

Zadatak 1 _____/10 *Zadatak 3* _____/10
Zadatak 2 _____/10

Ukupno: _____/30 = _____% = _____/15

Napomena: Ukoliko u zadatku nešto nije dovoljno precizno definisano, student treba da uvede razumnu pretpostavku, da je uokviri (da bi se lakše prepoznala prilikom ocenjivanja) i da nastavi da izgrađuje preostali deo svog odgovora na temeljima uvedene pretpostavke. Ocenjivanje unutar potpitanja je po sistemu "sve ili ništa", odnosno nema parcijalnih poena. Kod pitanja koja imaju ponuđene odgovore treba **samo zaokružiti** jedan odgovor. Na ostala pitanja odgovarati **čitko, kratko i precizno**.

1. (10 poena)

Koristeći samo sistemske pozive *fork*, *wait* i *exit*, kao i funkciju *printf*, napisati C program koji pronalazi maksimalnu vrednost svih elemenata ogromne celobrojne matrice dimenzije *M* puta *N* tako što uporedo pronalazi maksimum u svakoj vrsti matrice (maksimum svake vrste pronalazi u po jednom od pokrenutih procesa-dece), a onda pronalazi maksimum tih maksimuma. U slučaju greške, ovaj program treba da ispiše poruku o grešci i vrati status -1, a u slučaju uspeha treba da ispiše pronađeni maksimum i vrati status 0. Pretpostaviti da je matrica već nekako inicijalizovana.

```
const int M = ..., N = ...;
extern int mat[M][N];
```

Rešenje:

2. (10 poena)

Neki procesor pri obradi prekida, sistemskog poziva i izuzetka prelazi na sistemski stek. Taj stek alociran je u delu memorije koju koristi kernel, a na vrh tog steka ukazuje poseban registar SSP procesora koji je dostupan samo u privilegovanom režimu.

Prilikom obrade ovih situacija, procesor ništa ne stavlja na stek, već zatečene, neizmenjene vrednosti registara PC i PSW koje je koristio prekinuti proces sačuva u posebne, za to namenjene registre, SPC i SPSW, respektivno, a vrednosti registara SP i SSP međusobno zameni (*swap*). Prilikom povratka iz prekidne rutine instrukcijom `iret` procesor radi inverznu operaciju. Procesor je RISC, sa *load/store* arhitekturom i ima 32 registra opšte namene (R0..R31).

Kernel je višenitni, a svakom toku kontrole (procesu ili niti), uključujući i niti kernela, pridružen je poseban stek koji se koristi u sistemskom režimu. Prilikom promene konteksta, kontekst procesora treba sačuvati na tom steku, dok informaciju o vrhu steka treba čuvati u polju PCB čiji je pomeraj u odnosu na početak strukture PCB označen simboličkom konstantom `offsSP`.

U kodu kernela postoji statički pokazivač `oldRunning` koji ukazuje na PCB tekućeg procesa, kao i pokazivač `newRunning` koji ukazuje na PCB procesa koji je izabran za izvršavanje.

Napisati kod funkcije `yield` koju koristi kernel kada želi da promeni kontekst (prebaci se sa izvršavanja jednog toka kontrole, `oldRunning`, na drugi, `newRunning`), na bilo kom mestu gde se za to odluči.

Rešenje:

3. (10 poena)

Više procesa proizvođača i potrošača međusobno razmenjuju znakove (`char`) preko ograničenog bafera tako što svi ti procesi dele jedan zajednički logički segment memorije veličine `sizeof(bbuffer)` koji su alocirali sistemskim pozivom *mmap* kao *bss* segment (inicijalizovan nulama pri alokaciji). Svaki od tih procesa adresu tog segmenta konvertuje u pokazivač na tip `struct bbuffer` i dalje radi operacije sa ograničenim baferom pozivajući sledeće operacije iz biblioteke `bbuf` čije su deklaracije (`bbuf.h`) date dole:

- `bbuf_init`: svaki proces koji želi da koristi bafer mora najpre da pozove ovu operaciju kako bi ona otvorila potrebne semafore; ukoliko neki od sistemskih poziva vezanih za semafore nije uspeo, sve one već otvorene semafore treba osloboditi i vratiti negativnu vrednost (greška); ukoliko je inicijalizacija uspela, treba vratiti 0;
- `bbuf_close`: svaki proces koji koristi bafer treba da pozove ovu operaciju kada završi sa korišćenjem bafera;
- `bbuf_append`, `bbuf_take`: operacije stavljanja i uzimanja elementa (`char`) u bafer.

Sve ove operacije podrazumevaju da je argument ispravan pokazivač (ne treba ga proveravati). Sve operacije osim `bbuf_init` takođe podrazumevaju da je inicijalizacija pre toga uspešno završena – proces ih ne sme pozivati ako nije, pa ne treba proveravati ispravnost stanja bafera.

Implementirati biblioteku `bbuf` (`bbuf.cc`): dati definiciju strukture `bbuffer` i implementirati sve ove operacije korišćenjem POSIX semafora.

```
struct bbuffer;
int  bbuf_init (struct bbuffer*);
void bbuf_append (struct bbuffer*, char);
char bbuf_take (struct bbuffer*);
void bbuf_close (struct bbuffer*);
```

Rešenje:

Drugi kolokvijum iz Operativnih sistema 1
Maj 2022.

1. (10 poena)

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

const int M = ..., N = ...;
extern int mat[M][N];

int max (int i) {
  int m = mat[i][0];
  for (int j=1; j<N; j++)
    if (mat[i][j]>m) m = mat[i][j];
  return m;
}


int main () {

  for (int i=0; i<M; i++) {
    pid_t pid = fork();
    if (pid<0) {
      printf("Error: Cannot create a child process.\n");
      exit(-1);
    }
    if (pid==0)
      exit(max(i));
  }

  int max;
  wait(&max);
  for (int i=1; i<M; i++) {
    int m;
    wait(&m);
    if (m>max) max = m;
  }
  printf("Max: %d\n",max);
  exit(0);
}
```

2. (10 poena)

```
void yield () {
  asm {
  ; Save the current context
  push      spc ; save regs on the process stack
  push      ssp
  push      spsw
  push      r0
  push      r1
  ...
  push      r31
  load      r0, oldRrunning ; r0 now points to the running PCB
  store     sp, [r0+offsSP] ; save SP

  ; Restore the new context
  load      r0, newRunning
  load      sp, [r0+offsSP] ; restore SP
  pop       r31 ; restore regs
  ...
  pop       r0
  pop       spsw
  pop       ssp
  pop       spc
  }
}
```

3. (10 poena)

```
#include <fcntl.h>
  #include <sys/stat.h>
  #include <semaphore.h>

const size_t BBSIZE = ...;

static sem_t *mutex, *space_available, *item_available;

struct bbuffer {
  char buf[BBSIZE];
  int head, tail;
};


int  bbuf_init (struct bbuffer* bb) {
  mutex = sem_open("/bounded_buf_mx",O_CREAT,O_RDWR,1);
  space_available = sem_open("/bounded_buf_sa",O_CREAT,O_RDWR,BBSIZE);
  item_available = sem_open("/bounded_buf_ia",O_CREAT,O_RDWR,0);
  if (!mutex || !space_available || !item_available) {
    if (mutex) sem_close(mutex);
    if (space_available) sem_close(space_available);
    if (item_available) sem_close(item_available);
    return -1;
  }
  return 0;
}


void  bbuf_close (struct bbuffer* bb) {
  if (mutex) sem_close(mutex);
  if (space_available) sem_close(space_available);
  if (item_available) sem_close(item_available);
}


void  bbuf_append (struct bbuffer* bb, char c) {
  sem_wait(space_available);
  sem_wait(mutex);
  bb->buf[bb->tail] = c;
  bb->tail = (bb->tail+1)%BBSIZE;
  sem_post(mutex);
  sem_post(item_available);
}


char bbuf_take (struct bbuffer* bb) {
  sem_wait(item_available);
  sem_wait(mutex);
  char c = bb->buf[bb->head];
  bb->head = (bb->head+1)%BBSIZE;
  sem_post(mutex);
  sem_post(space_available);
  return c;
}
```

```
  return &instance;
}
```

Kreiranje niti

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

```
class Thread {
public:

  Thread ();
  Thread (void (*body)());
  void start ();

protected:

  virtual void run () {}

};
```

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

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

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

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

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

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

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

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

    unlock ();
    runningThread->run();                      // korisnička nit
    lock ();

    runningThread->markOver();                      // nit je gotova,
    runningThread = Scheduler::Instance()->get(); // predaje se procesor
    runningThread->resume();
}
```

- Takođe je jako važno obratiti pažnju na to da ne sme da se izvrši povratak iz funkcije `wrapper()`, jer se unutar nje prešlo na novi stek, pa na steku ne postoji povratna adresa. Zbog toga se iz ove funkcije nikad i ne vraća, već se po završetku korisničke funkcije `run()` eksplicitno predaje procesor drugoj niti.
- Zbog ovakve logike, neophodno je da u sistemu uvek postoji bar jedna spremna nit. Uopšte, u sistemima se to najčešće rešava kreiranjem jednog "praznog", besposlenog (engl. *idle*) procesa, ili nekog procesa koji vodi računa o sistemskim resursima i koji se nikad ne može blokirati, pa je uvek u redu spremnih (tzv. demonski procesi, engl. *deamon process*). U ovoj realizaciji to će biti nit koja briše gotove niti, opisana u narednom odeljku.
- Na ovaj način, startovanje niti predstavlja samo njeno upisivanje u listu spremnih, posle označavanja kao "započinjuće":

```
void Thread::start () {
  //...
  fork();
}


void Thread::fork () {
  lock();
  Scheduler::Instance()->put(this);
  unlock();
}
```

Ukidanje niti

- Ukidanje niti je sledeći veći problem u konstrukciji Jezgra. Gledano sa strane korisnika, jedan mogući pristup je da se omogući eksplicitno ukidanje kreirane niti pomoću njenog destruktora. Pri tome se poziv destruktora opet izvršava u kontekstu onoga ko uništava nit. Za to vreme sama nit može da bude završena ili još uvek aktivna. Zbog toga je potrebno obezbediti odgovarajuću sinhronizaciju između ova dva procesa, što komplikuje realizaciju. Osim toga, ovakav pristup nosi i neke druge probleme, pa je on ovde odbačen, iako je opštiji i fleksibilniji.
- U ovoj realizaciji opredeljenje je da niti budu zapravo aktivni objekti, koji se eksplicitno kreiraju, a implicino uništavaju. To znači da se nit kreira u kontekstu neke druge niti, a da zatim živi sve dok se ne završi funkcija `run()`. Tada se nit "sama" implicitno briše, tačnije njeno brisanje obezbeđuje Jezgro.
- Brisanje same niti ne sme da se izvrši unutar funkcije `wrapper()`, po završetku funkcije `run()`, jer bi to značilo "sečenje grane na kojoj se sedi": brisanje niti znači i dealokaciju steka na kome se izvršava sama funkcija `wrapper()`.
- Zbog ovoga je primenjen sledeći postupak: kada se nit završi, funkcija `wrapper()` samo označi nit kao "završenu" atributom `isOver`. Poseban aktivni objekat (nit) klase `ThreadCollector` vrši brisanje niti koje su označene kao završene. Ovaj objekat je nit kao i svaka druga, pa ona ne može doći do procesora sve dok se ne završi funkcija `wrapper()`, jer završni deo ove funkcije izvršava u sistemskom režimu.
- Klasa `ThreadCollector` je takođe *Singleton*. Kada se pokrene, svaka nit se "prijavi" u kolekciju ovog objekta, što je obezbeđeno unutar konstruktora klase `Thread`. Kada dobije procesor, ovaj aktivni objekat prolazi kroz svoju kolekciju i jednostavno briše sve niti koje su označene kao završene. Prema tome, ova klasa je zadužena tačno za brisanje niti:

```
void Thread::start () {
  ThreadCollector::Instance()->put(this);
  fork();
}



class ThreadCollector : public Thread {
public:

  static ThreadCollector* Instance ();

  void put (Thread*);
  int  count ();

protected:

  virtual void run ();

private:

  ThreadCollector ();

  Collection rep;

  static ThreadCollector* instance;

};


void ThreadCollector::run () {
  while (1) {

    int i=0;
    CollectionIterator* it = rep.getIterator();

    for (i=0,it->reset(); !it->isDone(); it->next(),i++) {
      Thread* cur = (Thread*)it->currentItem();
      if (cur->isOver) {
            rep.remove(i);
            delete cur;
            it->reset(); i=0;
            dispatch();
      }
    }

    if (count()==1)
      longjmp(mainContext,1);  // return to main

    dispatch();
  }
}
```

Pokretanje i gašenje programa

- Poslednji veći problem pri konstrukciji Jezgra jeste obezbeđenje ispravnog pokretanja programa i povratka iz programa. Problem povratka ne postoji kod ugrađenih (engl. *embedded*) sistema jer oni rade neprekidno i ne oslanjaju se na operativni sistem. U okruženju operativnog sistema kao što je PC DOS/Windows, ovaj problem treba rešiti jer je želja da se ovo Jezgro koristi za eksperimentisanje na PC računaru.

- Program se pokreće pozivom funkcije `main()` od strane operativnog sistema, na steku koji je odvojen od strane prevodioca i sistema. Ovaj stek nazivaćemo glavnim. Jezgro će unutar funkcije `main()` kreirati nit klase `ThreadCollector` (ugrađeni proces) i nit nad korisničkom funkcijom `userMain()`. Zatim će zapamtiti kontekst glavnog programa, kako bi po završetku svih korisničkih niti taj kontekst mogao da se povrati i program regularno završi:

```
void main () {

  ThreadCollector::Instance()->start();

  Thread::runningThread = new Thread(userMain);
  ThreadCollector::Instance()->put(Thread::runningThread);

  if (setjmp(mainContext)==0) {
    unlock();
    Thread::runningThread->resume();

  } else {
    ThreadCollector::destroy();
    return;
  }
}
```

- Treba još obezbediti "hvatanje" trenutka kada su sve korisničke niti završene. To najbolje može da uradi sam `ThreadCollector`: onog trenutka kada on sadrži samo jednu jedinu evidentiranu nit u sistemu (to je on sam), sve ostale niti su završene. (On evidentira sve aktivne niti, a ne samo spremne.) Tada treba izvršiti povratak na glavni kontekst:

```
void ThreadCollector::run () {
  //...
    if (count()==1)
      longjmp(mainContext,1);  // return to main
  //...
}
```

Realizacija

- Zaglavlje `kernel.h` služi samo da uključi sva zaglavlja koja predstavljaju interfejs prema korisniku. Tako korisnik može jednostavno da uključi samo ovo zaglavlje u svoj kod da bi dobio deklaracije Jezgra.
- Prilikom prevođenja u bilo kom prevodiocu treba obratiti pažnju na sledeće opcije prevodioca:

1. Funkcije deklarisane kao *inline* moraju tako i da se prevode. U Borland C++ prevodiocu treba da bude isključena opcija Options\Compiler\C++ options\Out-of-line inline functions. Kritična je zapravo samo funkcija `Thread::setContext()`.
2. Program ne sme biti preveden kao *overlay* aplikacija. U Borland C++ prevodiocu treba izabrati opciju Options\Application\DOS Standard.
3. Memorijski model treba da bude takav da su svi pokazivači tipa *far*. U Borland C++ prevodiocu treba izabrati opciju Options\Compiler\Code generation\Compact ili Large ili Huge.
4. Mora da bude isključena opcija provere ograničenja steka. U Borland C++ prevodiocu treba da bude isključena opcija Options\Compiler\Entry/Exit code\Test stack overflow.

- Sledi kompletan izvorni kod opisanog dela Jezgra.
- Datoteka `kernel.h`:

```
#include "recycle.h"

//////////////////////////////////////////////////////////////////////
// class Thread
//////////////////////////////////////////////////////////////////////

class Thread : public Object {
public:

  Thread ();
  Thread (void (*body)());

  void start ();
  static void dispatch ();

  static Thread* running ();

  CollectionElement* getCEForScheduler ();
  CollectionElement* getCEForCollector ();
  CollectionElement* getCEForSemaphore ();

protected:

  virtual void run ();

  void markOver ();

  friend class ThreadCollector;
  virtual ~Thread ();

  friend class Semaphore;

  inline int setContext ();
  void resume ();
  char* getStackPointer () const;

  static void wrapper ();
  void fork();

private:

  void (*myBody)();
  char* myStack;

  jmp_buf myContext;

  int isBeginning;
  int isOver;

  friend void main ();
  static Thread* runningThread;

  CollectionElement ceForScheduler;
  CollectionElement ceForCollector;
  CollectionElement ceForSemaphore;

  RECYCLE_DEC(Thread)

};
```

```
void Thread::start () {
  ThreadCollector::Instance()->put(this);
  fork();
}


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

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

  } else {
    unlock ();
    return;
  }
}


////////////////////////////////////////////////////////////////////
// Warning: Hardware/OS Dependent!
////////////////////////////////////////////////////////////////////

char* Thread::getStackPointer () const {
  // WARNING: Hardware\OS dependent!
  // PC Stack grows downwards:
  return myStack+StackSize-10;
}


// Borland C++: Compact, Large, or Huge memory Model needed!
#if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)
  #error Compact, Large, or Huge memory model needed
#endif


#define splitStack(p)            \
  static unsigned int sss, ssp;  \
  sss=FP_SEG(p); ssp=FP_OFF(p);  \
  asm {                          \
    mov ss,sss;                  \
    mov sp,ssp;                  \
    mov bp,sp;                   \
    add bp,8                     \
  }


////////////////////////////////////////////////////////////////////
// Enf of Dependencies
////////////////////////////////////////////////////////////////////


void Thread::fork () {
  lock();
  Scheduler::Instance()->put(this);
  unlock();
```