2. (10 poena)

U školskom jezgru promena konteksta implementirana je korišćenjem date funkcije `yield()`, u sistemskom pozivu `dispatch()` i na svim ostalim mestima na sličan način kao što je dato.

Potrebno je implementirati sistemski poziv (statičku operaciju) `Thread::wait()` kojim pozivajuća nit čeka (suspenduje se ako je potrebno) dok se ne završe sve niti-deca koje je ova pozivajuča nit do tada kreirala. Za te potrebe treba implementirati i sledeće nestatičke funkcije-članice:

- `void Thread::created(Thread* parent)`: poziva je jezgro interno za datu novokreiranu nit (`this`), kada je ta nit kreirana, sa argumentom `parent` koji ukazuje na roditeljsku nit u čijem kontekstu je ova nova nit-dete kreirana;
- `void Thread::completed()`: poziva je jezgro za datu nit (`this`), kada se ta nit završila.

Ukoliko proširujete klasu `Thread` novim članovima, precizno navedite kako.

```
void yield (jmp_buf old, jmp_buf new) {
  if (setjmp(old)==0) longjmp(new,1);
}

void Thread::dispatch () {
  lock();
  jmp_buf old = Thread::running->context;
  Scheduler::put(Thread::running);
  Thread::running = Scheduler::get();
  jmp_buf new = Thread::running->context;
  yield(old,new);
  unlock();
}
```

```
class Thread {
    Thread * parent;
    int child_count = 0;
    Bool waiting = false;
    void created ( Thread * parent) {
        this → parent = parent;
        parent → child_count ++;
    }
    static void wait() {
        if ( running → child_count == 0) return;
        running → waiting = true;
        Thread * old = running
        Thread * new = Scheduler::get();
        running = new;
        yield( old, new);
    }

    void completed() {
        parent → child_count --;
```

Thread::running

```
if (parent->waiting && parent->child_count == 0) {
    parent->waiting = false;
    Scheduler::put(parent);
}
}
```

2. (10 poena)

Školsko jezgro proširuje se konceptom tzv. *tačke spajanja* ili *susreta* (engl. *join*, *rendez-vous*), kao jednostavne sinhronizacione primitive sa sledećim značenjem.

Program može kreirati objekat klase `Join`, čiji je interfejs dat dole, zadajući mu kao parametre konstruktora (pokazivače na) dve niti koje će se sinhronizovati (susretati, „spajati“) na ovom objektu. Iz konteksta ove dve niti (i samo njih) može se pozvati operacija `wait` ovog objekta. Tom operacijom se ove dve niti „spajaju“ (sinhronizuju, susreću), odnosno međusobno sačekuju, tako da ni jedna od njih neće nastaviti svoje izvšavanje dalje ako ona druga nije stigla do iste te tačke spajanja (tj. pozvala ovu operaciju `wait` na istom ovom objektu klase `Join`). Drugim rečima, niti nastavljaju izvršavanje iza poziva `wait` samo kada su obe stigle do te tačke.

Dole je data implementacija sistemskog poziva `dispatch()` u školskom jezgru, kao primer kako su implementirana sva mesta promene konteksta. Implementirati u celini klasu `Join`, bez ikakve izmene ili dopune klase `Thread`. Operacija `wait` treba da vrati sledeće:

- -1 u slučaju da je ovu operaciju pozvala neka druga nit, osim one dve koje su definisane konstruktorom (ili bilo kakve druge greške);
- 0 u slučaju da je pozivajuća nit prva stigla do tačke spajanja, a 1 u suprotnom.

```
void dispatch () {
  lock();
  jmp_buf old = Thread::running->context;
  Scheduler::put(Thread::running);
  Thread::running = Scheduler::get();
  jmp_buf new = Thread::running->context;
  if (setjmp(old)==0) longjmp(new,1);
  unlock();
}

class Join {
public:
  Join (Thread* thread1, Thread* thread2);
  int wait ();
};
```

```
class Join {
public:
    Thread *t1, *t2;
    Join (Thread *t1, Thread *t2) : t1(t1), t2(t2) {}
    bool t1_waiting = false;
    bool t2_waiting = false;
    int wait() {
        if (running != t1 && running != t2) {
            return -1;
        }
```

```
if (running == t1) {
    if (t2_waiting) {
        Scheduler::put(t2);
        return 1;
    } else {
        t1_waiting = true;
        Thread * old = running;
        Thread * new = Scheduler::get();
        running = new;
        if (setjmp(old->context) == 0) {
            longjmp(new->context, 1);
        }
        return 0;
    }
} else {        running == t2.
    if (t1_waiting) {
        Scheduler::put(t1);
        return 1;
    } else {
        t2_waiting = true;
        Thread * old = running;
        Thread * new = Scheduler::get();
        running = new;
        if (setjmp(old->context) == 0) {
            longjmp(new->context, 1);
        }
        return 0;
    }
}
```

2. (10 poena)

U školskom jezgru promena konteksta implementirana je korišćenjem date funkcije `yield()`, u sistemskom pozivu `dispatch()` i na svim ostalim mestima na sličan način kao što je dato.

Potrebno je implementirati sistemski poziv (statičku operaciju):

```
void Thread::wait(Thread* forChild=0);
```

kojim pozivajuća nit čeka (suspenduje se ako je potrebno) dok se ne završi nit-dete na koje ukazuje argument, odnosno sve niti-deca koje je ova pozivajuća nit do tada kreirala, ako je ovaj argument *null*. Za te potrebe treba implementirati i sledeće nestatičke funkcije-članice:

- `void Thread::created(Thread* parent)`: poziva je jezgro interno za datu novokreiranu nit (`this`), kada je ta nit kreirana, sa argumentom `parent` koji ukazuje na roditeljsku nit u čijem kontekstu je ova nova nit-dete kreirana;
- `void Thread::completed()`: poziva je jezgro za datu nit (`this`), kada se ta nit završila.

Ukoliko proširujete klasu `Thread` novim članovima, precizno navedite kako.

```
void yield (jmp_buf old, jmp_buf new) {
  if (setjmp(old)==0) longjmp(new,1);
}

void Thread::dispatch () {
  lock();
  jmp_buf old = Thread::running->context;
  Scheduler::put(Thread::running);
  Thread::running = Scheduler::get();
  jmp_buf new = Thread::running->context;
  yield(old,new);
  unlock();
}
```

Rešenje:

```
class Thread {
    Thread * parent;
    bool waiting = false;
    int child_count = 0;
    Thread * waiting_for = nullptr;
    bool completed = false;

    void created (Thread *parent) {
        this -> parent = parent;
        parent -> child_count += 1;
    }

    static void wait ( Thread *for ) {
        if ( for == nullptr ) {
            if ( running -> child_count == 0 ) return;
        running -> waiting = true;
        Thread * old = running;
        Thread * new = Scheduler::get();
```

```
            running = new;
            yield(old->context, new->context);
        } else {
            if (for->completed || for->parent != running) return;
            running->waiting = true;
            running->waiting_for = for;
            Thread *old = running;
            Thread *new = Scheduler::get();
            running = new;
            yield(old->context, new->context);
        }
    }

    void completed() {
        this->completed = true;
        this->parent->child_count -= 1;
        if (this->parent->waiting) {
            if (this->parent->waiting_for == this) {
                this->parent->waiting = false;
                this->parent->waiting_for = nullptr;
                Scheduler::put(this->parent);
            }
            if (this->parent->waiting_for == nullptr &&
                this->parent->child_count == 0) {
                this->parent->waiting = false;
                this->parent->waiting_for = nullptr;
                Scheduler::put(this->parent);
            }
        }
    }
};
```

2. (10 poena)

Raspoređivanje i promena konteksta u školskom jezgru implementirani su na sledeći način:

- Svi objekti klase `Thread` smešteni su u statički vektor `Thread::allThreads` tipa `Thread[NumOfThreads]`.
- Nestatički podatak član `Thread::isRunnable` govori o tome da li je odgovarajuća nit spremna za izvršavanje ili se baš ona izvršava, ili nije (jer je suspendovana).
- Statički podatak član `Thread::running` tipa `int` ukazuje na onaj element niza `allThreads` koji predstavlja nit koja se trenutno izvršava (tekuća nit).
- Za ivršavanje se bira prva sledeća spremna nit u nizu `allThreads` iza one tekuće, i tako u krug.
- Nestatički podatak član `Thread::context` tipa `jmp_buf` čuva procesorski kontekst niti.
- Implementirana je funkcija `yield(jmp_buf oldC, jmp_buf newC)` koja čuva kontekst procesora u prvi argument i restaurira kontekst iz drugog argumenta.

Korišćenjem date funkcije `yield`, implementirati funkciju `dispatch` koja treba da preda procesor niti koja je data kao argument, pod uslovom da je taj argument dat i da je ta nit spremna. Ako je data nit spremna, funkcija treba da vrati 0; u suprotnom, tekuća nit treba da nastavi izvršavanje, a ova funkcija da vrati -1. Ako nije zadata nit, ova funkcija treba da preda procesor sledećoj spremnoj niti i da vrati 0.

```
int dispatch (Thread* newT = 0);
```

```
int dispatch (Thread * new) {
    if (new == nullptr) {
        int new_idx = running;
        for (int i = 1; i < NumberOfThreads; i++) {
            if (allThreads[(running + i) % NumberOfThreads].isRunnable) {
                new_idx = (running + i) % NumberOfThreads;
                break;
            }
        }
        if (new_idx == running) return -1;
        int old_idx = running;
        running = new_idx;
        yield (allThreads[old_idx].context, allThreads[new_idx].context);
        return 0;
    } else {                                   new ≠ nullptr
        if (!new->isRunnable) return -1;
        int new_idx = new - allThreads;
        int old_idx = running;
        running = new_idx;
        yield (allThreads[old_idx].context, allThreads[new_idx].context);
        return 0;
    }
}
```

1. (10 poena)

Dostupni su sledeći delovi školskog jezgra, kao i funkcije iz standardne biblioteke:

- `Thread::running`: statički član – pokazivač na objekat klase `Thread` koji predstavlja tekuću nit;
- `Thread::context`: nestatički član tipa `jmp_buf` u kom se čuva procesorski kontekst niti;
- `const size_t STACK_SIZE`: veličina prostora za stek niti (u jednicama `sizeof(char)`);
- `Thread::stack`: pokazivač tipa `void*` koji ukazuje na adresu početka dela memorije u kom je alociran stek niti;
- `jmp_buf::sp`: polje za sačuvanu vrednost registra SP; stek raste ka nižim adresama, a SP ukazuje na prvu slobodnu lokaciju veličine `sizeof(int)`;
- `Scheduler::put(Thread*)`: statička članica kojom se u red spremnih stavlja data nit;
- `void* malloc(size_t sz)`: standardna funkcija koja alocira prostor veličine `sz` (u jedinicama `sizeof(char)`); funkcija `free(void*)` oslobađa ovako alociran prostor;
- `memcpy(vodi* dst, const void* src, size_t size)`: kopira memorijski sadržaj veličine `size` sa mesta na koje ukazuje `src` na mesto na koje ukazuje `dst`;
- `setjmp()`, `longjmp()`: standardne bibliotečne funkcije.

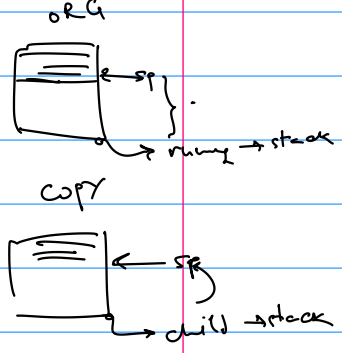
Pomoću ovih elemenata implementirati sistemski poziv školskog jezgra – funkciju:

```
Thread* t_fork();
```

kojom se kreira nova nit kao „klon“ pozivajuće, roditeljske niti, sa istim početnim kontekstom, ali sa sopstvenom kontrolom toka, po uzoru na standardni sistemski poziv `fork()` za procese. U slučaju uspeha, u kontekstu roditeljske niti ova funkcija treba da vrati pokazivač na objekat niti deteta, a u kontekstu niti deteta treba da vrati 0; u slučaju greške, treba da podigne izuzetak tipa `ThreadCreationException`.

Rešenje:

```
static Thread * Thread::t_fork() {
    Thread *child = new Thread();
    if (child == nullptr) {
        throw ThreadCreationException;
    }
    free(child->stack);
    child->stack = malloc(running->STACK_SIZE * sizeof(char));
    if (child->stack == null) {
        delete child;
        throw ThreadCreationException;
    }
    memcpy(child->stack, running->stack, running->STACK_SIZE * sizeof(char));
    if (setjmp(child->context) == 0) {
        child->context->sp =
        child->stack + (running->context->sp - running->stack);
```



```
        Scheduler::put(child);
        return child;
    } else {
        return null;
    }
}
```

2. (10 poena)

Školsko jezgro proširuje se nestatičkom funkcijom `Thread::join()` koju sme da pozove samo roditeljska nit date niti da bi sačekala da se data nit-dete završi; ukoliko ovu operaciju pozove neka druga nit koja nije roditelj date niti, ova funkcija vraća grešku (-1). Kada jezgro pravi novu nit, poziva funkciju `Thread::wrapper` koja obavlja sve potrebne radnje pre poziva funkcije `Thread::run` (u kontekstu napravljene niti) i nakon povratka iz nje:

```
void Thread::wrapper (Thread* toRun) {
  ...
  unlock();
  toRun->run();
  lock();
  ...
}
```

Precizno navesti sve izmene i dopune koje je potrebno napraviti u klasi `Thread` i implementirati operaciju `Thread::join`. Uzeti u obzir to da nit-roditelj može pozvati `join` više puta, pri čemu se samo pri prvom pozivu eventualno može zaustaviti dok nit-dete ne završi, svi pozivi nakon toga su neblokirajući. Problem gašenja niti i brisanja objekta klase `Thread`, kao i sinhronizacije potrebne za to ne treba rešavati u ovom zadatku.

Rešenje:

```
class Thread {
    Thread *parent;
    Thread *waiting_for = nullptr;
    bool completed = false;

static void wrapper(Thread *toRun) {
    lock();
    toRun->parent = running;
    unlock();
    toRun->run();
    lock();
    toRun->completed = true;
    if (toRun->parent->waiting_for == toRun) {
        toRun->parent->waiting_for = nullptr;
        Scheduler::put(toRun->parent);
    }
}
```

```
int join() {
    if (this->parent != running) return -1;
    if (!this->completed) {
        Thread *old = running;
        Thread *new = Scheduler::get();
        running = new;
        if (setjmp(old->context) == 0) {
            longjmp(new->context, 1);
        }
        return 0;
    }
    return -1;
}
```

2. (10 poena)

U implementaciji jezgra nekog jednoprocesorskog *time-sharing* operativnog sistema, radi pojednostavljenja celog mehanizma promene konteksta, primenjeno je sledeće neobično rešenje. Promena konteksta vrši se isključivo kao posledica prekida, na samo jednom mestu u kodu koji se izvršava na prekid. Prekidi dolaze od raznih uređaja, u najmanju ruku od vremenskog brojača, jer on u svakom slučaju generiše prekid zbog toga što je tekućoj niti isteklo dodeljeno procesorsko vreme. Zbog toga tekuća nit nikada ne gubi procesor sinhrono, čak ni kada poziva blokirajuću operaciju (sistemski poziv). Umesto toga, ukoliko je potrebno da se nit suspenduje (blokira) u nekom blokirajućem pozivu, nit se samo „označi" suspendovanom i nastavlja sa izvršavanjem (uposlenim čekanjem) sve dok ne stigne sledeći prekid. Kada takav prekid stigne, prekidna rutina vrši samu promenu konteksta.
Na primer, implementacija operacije `suspend`, koja suspenduje pozivajući proces, i `resume`, koja ponovo deblokira dati proces, izgledaju ovako:

```
void suspend () {
  running->status = suspended; // Mark as suspended
  while (running->status==suspended); // and then busy-wait
  // until it is preempted, suspended, and then resumed later
}

void resume (int pid) {
  processes[pid].status = ready; // Mark as ready
}
```

Procesor je RISC sa *load*-store arhitekturom, ima 32 registra opšte namene i SP. Prilikom prekida na steku čuva samo PC i PSW. Tajmer se restartuje upisom odgovarajuće vrednosti u registar koji se nalazi na adresi simbolički označenoj sa `Timer`. PCB procesa je dat strukturom definisanom dole, a svi procesi zapisani su u nizu `processes`. U asembleru, simboličko ime polja strukture ima vrednost pomeraja (engl. *offset*) tog polja od početka strukture.

```
enum ProcessStatus { unused, initiating, terminating, ready, suspended };
typedef unsigned long Time;
typedef unsigned Register;

struct PCB {
  ProcessStatus status;  // Process status
  Time timeSlice;        // Time slice for time sharing
  Register savedSP;      // Saved stack pointer
  ...
}

const unsigned long NumOfProcesses = ...;
PCB processes[NumOfProcesses];
PCB* running;  // Running process
```

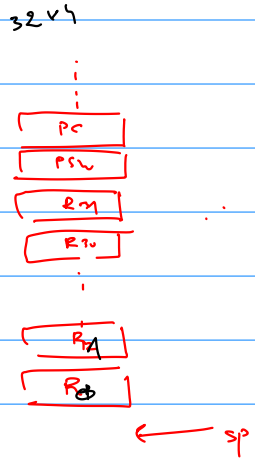
running → saved SP

a)(5) Na asembleru datog procesora napisati kod prekidne rutine koja vrši promenu konteksta. Ova prekidna rutina, pored čuvanja i restauracije konteksta na steku procesa, treba da pozove potprogram `scheduler` koji će u pokazivač `running` smestiti vrednost koja ukazuje na novoizabrani tekući proces, i da tajmer restartuje sa vremenskim kvantumom dodeljenim tom procesu. Pretpostavlja se da uvek postoji barem jedan spreman proces.

b)(5) Na jeziku C napisati potprogram `scheduler` koji bira sledeći proces za izvršavanje. Spremne procese treba da bira redom, u krug, a ne svaki put od početka niza `processes`.

interrupt:

```
addi sp, sp, -128
store R0, [sp + 1.4]
store R1, [sp + 2.4]
   ⋮
store R31, [sp + 32.4]
load R0, running
addi R0, R0, savedSP
store sp, [R0]
call scheduler
```



```
load R0, running.
addi R1 R0, saved PC
load sp, [R0]
addi R1, R0, time slice
load R1, [R1]          R1 := time slice.
store R1, Timer.
load R0, [sp + 1.4]
load R1, [sp + 2.4]
        :
load R31, [sp + 32.4]
iret.
```

```
void scheduler () {
    int running_idx = running - processes;
    running -> status = suspended;
    int new_idx;
    for ( int i = 0; i < Number of Processes; i++ ) {
        if ( processes [(running_idx + i) % hop ]. status == ready) {
            new_idx = (running_idx + i) % hop
            break;
        }
    }.
    running -> processes [new_idx]
}.
```

```
void scheduler () {
  do {
    running = processes + (running - processes + 1) % NUM_OF_PROCESSES;
  } while (running->status!=ready);
}
```

PCB

3. (10 poena)

Neki troadresni RISC procesor sa *load/store* arhitekturom, poput onog opisanog na predavanjima, poseduje 32 registra opšte namene, označenih sa *R*0..*R*31, statusnu reč PSW i pokazivač steka SP, koji su dostupni instrukcijama koje se izvršavaju u korisničkom režimu rada procesora, kao i dva posebna registra *Rx* i *Rp* koji su dostupni samo u privilegovanom (sistemskom) režimu rada procesora. Registar *Rx* se može koristiti kao i bilo koji registar *R*0..*R*31, i operativni sistem ga može koristiti proizvoljno za sopstvene potrebe (npr. prilikom promene konteksta). Registar *Rp* se može samo čitati, jer je ožičen tako da njegova vrednost predstavlja jedinstveni identifikator svakog pojedinačnog procesora u multiprocesorskom sistemu. Svi registri su 32-bitni, a adresibilna jedinica je bajt.

Za multiprocesorski sistem sa ovim procesorom pravi se operativni sistem. U kernelu tog sistema postoje sledeće definisane konstante i strukture podataka:

```
const int NumOfProcessors = ... // Number of processors
struct PCB;  // Process Control Block
PCB* runningProcesses[NumOfProcessors]; // Running processes
```

U strukturi PCB postoje polja za čuvanje vrednosti svih registara *R*0..*R*31, PSW i SP. Pomeraji ovih polja u odnosu na početak strukture PCB simbolički označavaju sa `offs_r0` itd. Niz `runningProcesses`, u svakom svom elementu *n*, sadrži pokazivač na PCB onog procesa koji se trenutno izvršava na procesoru broj *n* (*n*=0..*NumOfProcesses*-1). Na raspolaganju je operacija `schedule()` (bez argumenata), koja vrši odabir narednog procesa za izvršavanje na procesoru na kome se izvršava i koja upisuje adresu PCB tog procesa u odgovarajući element niza `runningProcesses`.

a)(7) Na asembleru datog procesora napisati operaciju `dispatch()` (bez argumenata) koja čuva kontekst tekućeg izvršavanja i restaurira kontekst izvršavanja procesa kome treba dati procesor. U asembleru datog procesora može se koristiti identifikator statički alociranog podatka iz C programa, pri čemu se takva upotreba prevodi u konstantu sa vrednošću adrese tog podatka. Ova operacija se izvršava u kodu kernela, u sistemskom režimu. U ovu operaciju ulazi se iz prekidne rutine koja obrađuje sistemski poziv, pa su prekidi već maskirani (ne treba ih maskirati i demaskirati).

b)(3) Da li je u ovu operaciju neophodno ubaciti kod za međusobno isključenje konkurentnog izvršavanja od strane različitih procesora tehnikom uposlenog čekanja (*spin lock*)? Obrazložiti.

Rešenje:

dispatch:

```
push  R0
push  R1
multi R0, Rp, 4
load  R1, runningProcesses
add   Rx, R1, R0        Rx := running.
pop   R1
pop   R0
store sp, [Rx + offs_sp]
store psw, [Rx + offs_psw]
store R0, [Rx + offs_r0]
  :
store R31, [Rx + offs_r31]
call schedule
```

Rx1 = Rx + 4·Rp

```
muli  R0, Rp, 4
load  R1, running Process
add   Rx, R1, R0          Rx := running.

load  sp, [Rx + offs_sp].
ld    psw, [Rx + offs_psw]
ld    R0, [Rx + offs_r0]
      :
ld    Rn, [Rx + offs_r31].

ret.
```

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).

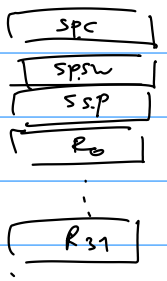
Kernel je višenitni, a svakom toku kontrole (procesu ili niti), uključujući i niti kernela, pridružen je poseban stek koji se koristi u sistemskom režimu. Prilikom promene konteksta, kontekst procesora treba sačuvati na tom steku, dok informaciju o vrhu steka treba čuvati u polju PCB čiji je pomeraj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

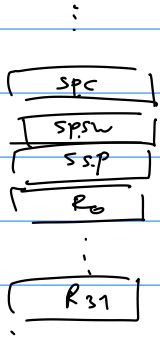
U kodu kernela postoji statički pokazivač `oldRunning` koji ukazuje na PCB tekućeg procesa, kao i pokazivač `newRunning` koji ukazuje na PCB procesa koji je izabran za izvršavanje.

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:


```
yield:  push  spc
        push  spsw
        push  ssp
        push  R0
              :
        push  R31
        store sp, offs_sp [oldRunning]
        load  sp, offs_sp [newRunning].
        pop   R31
              :
        pop   R0
```





pop SSP
pop SPSW
pop SPC
ret

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:

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