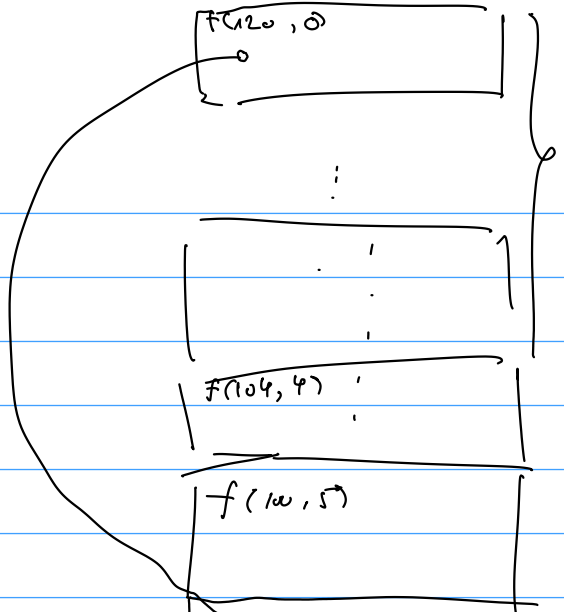


3. (10 poena)

U nekom 32-bitnom RISC procesoru svi registri su 32-bitni, adrese su 32-bitne, a adresibilna jedinica je bajt. Prevodilac za jezik C za taj procesor povratnu vrednost funkcije prenosi kroz registar r0. Data je jedna nekorektna implementacija standardnih funkcija setjmp() i longjmp() za taj procesor i taj prevodilac:

```
int setjmp (jmp_buf buf) {
  asm {
    clr r0; // r0:=0
    push r1;
    load r1,-2*4[sp]; // r1:=buf
    store sp,0*4[r1];
    store psw,1*4[r1];
    store r2,2*4[r1];
    store r3,3*4[r1];
    ...
    store r31,31*4[r1];
    store pc,32*4[r1];
    pop r1
  }
}

void longjmp (jmp_buf buf, int val) {
  asm {
    load r0,-2*4[sp]; // r0:=val
    and  r0,r0,r0;    // fix r0 if it is zero
    jnz  continue
    load r0,#1-
continue:
    load r1,-1*4[sp]; // r1:=buf
    load sp,0*4[r1];
    load psw,1*4[r1];
    load r2,2*4[r1];
    load r3,3*4[r1];
    ...
    load r31,31*4[r1];
    load pc,32*4[r1];
  }
}
```

Posmatrati upotrebu ovih funkcija, na primer kao u operaciji dispatch() školskog jezgra i precizno objasniti šta je problem sa ovom implementacijom.

3. (10 poena)

Korišćenjem standardnih bibliotečnih funkcija `setjmp()` i `longjmp()`, realizovati operaciju

`yield(jmp_buf old, jmp_buf new);`

koja čuva čuva kontekst niti čiji je `jmp_buf` dat kao prvi argument, oduzima joj procesor i restaurira kontekst niti čiji je `jmp_buf` dat kao drugi argument, kojoj predaje procesor.

Koristeći ovu operaciju `yield()`, realizovati operaciju `dispatch()` koja ima isti efekat kao i ona data u školskom jezgru.

Rešenje:

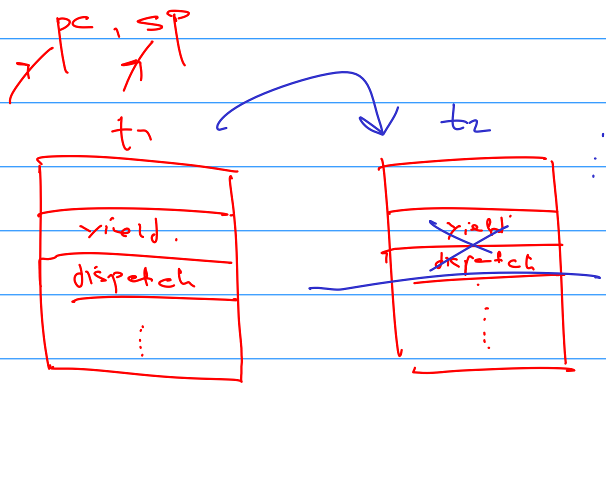
```
void Thread::dispatch() {
    Thread * next = Scheduler::get();
    Scheduler::put(this);
    yield(this->context, next->context);
}

static void Thread::yield(jmp_buf old, jmp_buf new) {
    int val = setjmp(old);
    if (val == 0) {
        longjmp(new, 1);
    }
}
```

ra — return address.

call — псеудо инстр.   ret — псеудо инстр.

jalr ra, lab_f.   jr ra   pc := ra.

ra := pc + 4, pc := lab_f



3. (10 poena)

U nekom sistemu realizovana je operacija

```
void yield (jmp_buf old, jmp_buf new);
```

koja čuva kontekst niti čiji je `jmp_buf` dat kao prvi argument, oduzima joj procesor i restaurira kontekst niti čiji je `jmp_buf` dat kao drugi argument, kojoj predaje procesor.

Koristeći ovu operaciju `yield()`, realizovati operacije:

- `Thread::suspend()`: (statička) suspenduje (blokira) izvršavanje pozivajuće niti sve dok je neka druga nit ne „probudi" pomoću `resume()`;
- `Thread::resume()`: (nestatička) „budi" (deblokira) nit za koju je pozvana; vrši i promenu konteksta predajući procesor niti koja je na redu za izvršavanje.

Pretpostaviti da je lista spremnih procesa uvek neprazna prilikom suspenzije niti i da je nit za koju se poziva `resume` sigurno suspendovana (ignorisati mogućnost greške). Klase `Thread` i `Scheduler` su u preostalim delovima implementirane kao u školskom jezgru.

Rešenje:

this

Thread * t   t -> context

```
class WorkerThread : public Thread {          setjmp( ...
   :
                                              this -> suspend();
   void run() {
      while (true) {
         cout << "hello" << endl;
         Thread::suspend();
      }
   }
};
:
worker -> resume();


void Thread::resume() {
      Scheduler::put(this);
```

Scheduler::put , Scheduler::get

static Thread * Thread::running;

```
static void Thread::suspend() {
    Thread *old = Thread::running;
    Thed *new = Scheduler::get();
    Thred::running = new;
    yield(old->context, new->context);
}
```

```
void Thread::resume() {
    Scheduler::put(this);
    Thred *new = Scheduler::get();
    Thred *old = Thred::running;
    Scheduler::put(old);
    Thread::running = new
    yield(old, new);
}
```

t_1 t_2

{ t_1->suspend(); }

2. (10 poena)

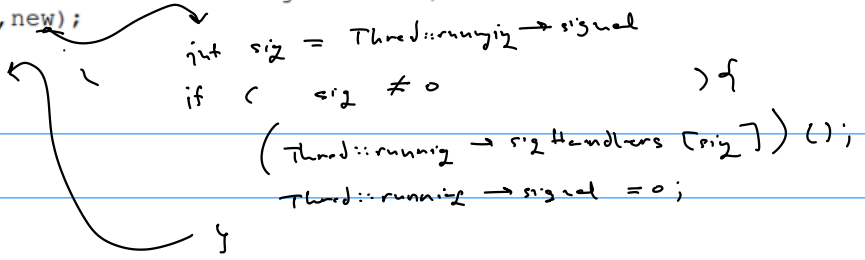
Školsko jezgro proširuje se konceptom tzv. *asinhronih signala*, koji podržavaju mnogi operativni sistemi, sa sledećim značenjem.

Kernel može „poslati“ *signal* nekoj korisničkoj niti. Signal je celobrojna konstanta u opsegu `1..SIGS-1`. Čim ta nit ponovo dobije procesor, umesto da odmah nastavi izvršavanje tamo gde je bila prekinuta, odnosno izgubila procesor, najpre će skočiti u proceduru za obradu tog signala, tzv. *signal handler*, pa kada se iz te procedure vrati, nastaviće dalje izvršavanje tamo gde je ono bilo prekinuto.

Na procedure za obradu signala pokazuju pokazivači u tabeli koja se nalazi u PCB svake niti, poput vektor tabele, u nizu `Thread::sigHandlers[SIGS]`. Za svaku nit, signalu *n* odgovara procedura za obradu na koju ukazuje pokazivač u ulazu `Thread::sigHandlers[n]` (ulaz 0 se ne koristi). Kada „šalje“ signal nekoj niti, kernel u PCB te niti postavi vrednost signala *n* u polje `Thread::signal`; vrednost različita od 0 u ovom polju ukazuje na postojanje poslatog signala, dok 0 znači da signala nema.

Dole je data implementacija sistemskog poziva `dispatch()` u školskom jezgru. Modifikovati ovu implementaciju tako da podrži obradu poslatog signala datoj niti. Smatrati da su na isti ovakav način realizovani i svi ostali sistemski pozivi i preuzimanja procesora.

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

void 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();
}
```



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 pozivajuća 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 {
        ⋮
public:
    Thread * parent;
    int child_count = 0;
    void created (Thread *parent) {
        this -> parent = parent;
    }
    void completed() {
        parent -> child_count -= 1;
    }
}.

static void Thread:: wait() {
    while (Thread:: running -> child_count ≠ 0) {
        .dispatch();
    }
}.
```

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 (nestatičku funkciju članicu):

```
int Thread::join (int* status=nullptr);
```

kojim pozivajuća nit čeka (suspenduje se ako je potrebno) dok se ne završi nit dete na koje ukazuje `this` i potom vraća 0; ako `this` ukazuje na nit koja nije dete pozivajuće niti, pozivalac ne čeka ništa i ova funkcija vraća -1. Ukoliko je parametar `status` različit od *null*, u dati izlazni parametar treba upisati povratni status završene niti deteta.

Za te potrebe treba implementirati i sledeće nestatičke funkcije članice:

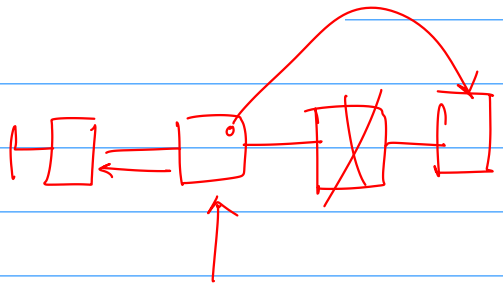
- `void Thread::created()`: poziva je jezgro interno za datu novokreiranu nit (`this`), kada je ta nit kreirana;
- `void Thread::completed(int status)`: poziva je jezgro za datu nit (`this`), kada se ta nit završila; celobrojni argument je dostavila završena nit kao svoj povratni status.

Pretpostaviti da se objekti klase `Thread` nikada ne uništavaju (ili barem ne uništavaju dok se ne završe i unište roditelji). Ukoliko proširujete klasu `Thread` novim članovima, precizno navedite kako.

```
Inline void yield (Thread* oldThr, Thread* newThr) {
  if (setjmp(oldThr->context)==0) longjmp(newThr->context,1);
}

void Thread::dispatch () {
  lock();
  Thread* oldThr = Thread::running;
  Scheduler::put(oldThr);
  Thread* newThr = Thread::running = Scheduler::get();
  yield(oldThr,newThr);
  unlock();
}
```

Rešenje:

```
class Thread {
  .
  .
  .
Thread * parent;
void created () {
        parent = Thread::running;
}

Bool is_completed = false;
int status;

void completed(int status) {
        is_completed = true;
        this -> status = status;
}
```



```
tx
{
}
ty->join();
```

```
int join(int *status=nullptr){
    if (this->parent == Thread::running()){
        return -1;
    }

    while(!this->is_completed){
        dispatch();
    }

    if (status != nullptr){
        *status = this->status;
    }

    return 0;
}
```

2. (10 poena)

U nekom asimetričnom multiprocesorskom operativnom sistemu jedan od procesora posvećen je samo obavljanju ulazno-izlaznih operacija koje su zahtevali korisnički procesi. Operacije sa svakim pojedinčanim ulazno-izlaznim uređajem obavlja po jedna interna kernel nit predstavljena objektom klase `IOThread`. Ove niti izvršavaju se samo na ovom posvećenom procesoru i nemaju nikakve veze sa ostalim nitima kernela niti korisničkim procesima (osim preuzimanja zahteva koje su oni postavili); na ovom procesoru izvršavaju se samo ove niti.

Svaka od tih niti ima sledeći generički oblik: ona uzme jedan zahtev za ulazno-izlaznom operacijom iz reda zahteva postavljenih za taj uređaj, pokrene operaciju sa uređajem na način specifičan za taj uređaj, a onda se suspenduje pozivom operacije `IOThread::suspend` dok uređaj ne signalizira spremnost za novu operaciju spoljašnjim prekidom. Prekidne rutine svih tih uređaja ne vrše promenu konteksta (preotimanje procesora), već samo postavljaju polje `IOThread::isReady` svoje niti na 1, čime ta nit ponovno postaje spremna i može da zada novu operaciju. Zbog svega ovoga nije potrebno raditi nikakvo maskiranje prekida niti međusobno isključenje sa ostalim procesorima.

Sve ove niti predstavljene su objektima klase `IOThread` u statičkom nizu `IOThread::allThreads`, a njihov broj je konstantan i iznosi `IOThread::NumberOfThreads`. Polje `IOThread::running` je pokazivač na nit koja se trenutno izvršava na ovom posvećenom procesoru. Na raspolaganju je funkcija:

```
void IOThread::yield (IOThread* oldThread, IOThread* newThread);
```

koja čuva kontekst tekuće niti u za to predviđeno polje objekta na koga ukazuje prvi argument i restaurira kontekst niti iz polja objekta na koga ukazuje drugi argument.

Implementirati operaciju `IOThread::suspend`. Za izvršavanje je dovoljno uzeti prvu (ili bilo koju drugu) spremnu nit iz niza `IOThread::allThreads`. Obratiti pažnju na to da je moguće da nijedna nit nije spremna za izvršavanje, jer su sve suspendovane i čekaju na završetak svojih operacija i prekide od svojih uređaja koji će ih ponovo učiniti spremnim; u tom slučaju procesor treba da uposleno čeka dok neka nit ne postane spremna.

Rešenje:

```
static void IOThread::suspend() {
    IOThread::running -> isReady = false;
    IOThread * new = nullptr;
    while ( new == nullptr ) {
        for ( int i = 0; i < IOThread::NumberOfThreads; i++) {
            if ( IOThread::allThreads[i].isReady ) {
                new = IOThread::allThreads[i]
                break;
            }
        }
    }
    IOThread * old = IOThread::running;
    IOThread::running = new;
    yield (old, new);
}
```

Busy wait