OS1
Projekat

2018/2019.

Uvod

Dobro, vama bi trebalo da sada bude u potpunosti jasno otprilike šta treba da napravite. Ja da sam student i da sam slušao sebe kako držim vežbe, meni bi bilo jasno. Ideja je da, ovako…

Prva napomena: Kada budete razvijali projekat radite sa onom virtuelnom mašinom koja stoji na sajtu. Zašto ? Zato što ćete tu istu virtuelnu mašinu da dobijete kad dođete da branite projekat.

Da ne bude „E kući mi je radilo a ovde sada neće, nešto je do vas.“ Nije do nas nego do vašeg projekta. Tako da skinete onu virtuelnu mašinu i tu razvijate projekat. Morate da koristite onaj kompajler koji smo tamo zadali. Zašto morate da koristite taj kompajler ? Ideja je sledeća: Vi kad budete napisali vaš kod, i kad ga prevedete sa onim kompajlerom, on za vas napravi jedan .exe fajl koji može da se pokrene. Ali unutar tog .exe fajla on ugradi kod koji ste vi napisali, ali i ugradi nešto što se zove emulator, vi možete to da shvatite kao simulator, ugradi simulator onog procesora 8086. I onda se ceo taj .exe fajl u stvari izvršava na sledeći način: simulira se rad 8086 i po tom simulatoru trči kod koji ste vi napisali. I zbog toga morate da radite sa onim kompajlerom.

Šta je osnovna ideja ? Osnovna ideja je da u projektu napravite koncept niti, dve vrste tehnika sinhronizacije, jedna uz pomoć semafora, jedna uz pomoć event-a, i to je u suštini prvi deo za 20 poena. Onaj deo koji nosi od 20 do 30 poena se odnosi na signale. Ajmo sad da vidimo dalje šta ima ovde kroz sam tekst.

Izvršavanje vašeg programa mora da krene od main funkcije. Pošto pišete ovo u cpp-u, vi ćete dakle imati main funkciju. To će biti ulazna tačka vašeg programa. Ovde sad ima onih standardnih parametara, nebitno. I izvršavanje samih test programa je spakovano u posebnu funkciju koja se zove `userMain`.

```
int userMain(int argc, char* argv[]);
```

Prima iste parametre kao i ovaj main. Unutar ove `userMain` funkcije, to je ono što ćete vi da dobijete od nas, vi morate samo da je pozovete, a unutar `userMain` funkcije biće napisani testovi. Takođe, kada vi budete testirali sa vašim nekim testovima, napravite `userMain`, i onda unutar, a ovo `userMain` biće neka globalna funkcija, i onda unutar nje pišete vaše testove.

```
int main(int argc, char* argv[])
{
        init();
        userMain(...);
        restore();
}
```

Ono što je bitno ovde u ovako jednoj prostoj strukturi jeste sledeća stvar:

Pre nego što pozovete `userMain` vi treba da namestite da se ovde (`init()`) obavi potpuna inicijalizacija vašeg kernela. Šta znači potpuna inicijalizacija ? Sigurno ćete morati da promenite prekidnu rutinu tajmera, možda će vam trebati još neka prekidna rutina koju ćete morati da promenite. Sve te stvari treba da uradite sa ovim pozivom `init`. **Kako ćete ovo da izvedete ? Možete da imate posebnu klasu koja će da se zove `System`, i koja će u svom konstruktoru da uradi svu neophodnu inicijalizaciju.** Pored promena tih prekidnih rutina, gotovo sigurno biće vam potrebna gomila nekih lista ili nizova gde ćete čuvati informacije o nitima koje postoje u sistemu, o semaforima koji postoje u sistemu, itd… Dakle, svu tu inicijalizaciju treba da uradite pre poziva `userMain`.

Onda se izvršava `userMain`. Tu sad neko piše testove, nešto se dešava, da vidimo kako radi to što ste napisali. Kad se završi `userMain`, ono što treba da uradite jeste da vratite sistem u stanje kakvo je bilo. Dakle, ovde (`restore()`) treba da dođe neki poziv kojim ćete vi, sve one prekidne rutine koje ste čačkali, da vratite na stare prekidne rutine, da oslobodite svu memoriju koju ste koristili za te strukture, liste i nizove, gde ste čuvali informacije o nitima i semaforima. I tek nakon toga se završava `main`. **Ovo mora savršeno dobro da radi**. Kad budete testirali, vi pokrenete iz konzole vaš .exe fajl koji ste dobili prevođenjem. Kad se završi izvršavanje, unutar iste konzole, konzola mora da vam ostane živa, i mora da se ponovo bez problema pokrene isti program. Ako vam to ne radi, ili ako vam pukne nakon završetka, znači verovatno da ovaj `restore` niste dobro uradili. Nije dobro rešenje ako vi morate da ubijete konzolu pa tek onda da pokrenete novo izvršavanje. Dakle konzola jedna jedina mora da ostane živa za pokretanje iznova projekta. I zato je bitno da ovaj `restore` bude dobro napisan.

Dakle, ulazna tačka jeste `main` funkcija, ali sama srž se nalazi unutar `userMain`. Tu će biti testovi. Ono što ovde morate da radite jeste neki `init`, ovo shvatite kao podizanje sistema, a ovaj `restore` shvatite kao gašenje.

Pitanje: Šta bi se tipično radilo unutar `init` i `restore` ?
Odgovor: Preusmeravanje svih prekidnih rutina, znači tajmer, možda vam treba još neka, pravljenje globalnih struktura gde ćete da čuvate informacije o thread-ovima, o semaforima, možda o event-ovima koje imate u celom sistemu. Pravljenje PCB-a za `main`, sistemskog PCB-a, pošto i `main` se izvršava kao jedan thread, u nekom trenutku će i `main` da izgubi procesor pa morate da zapamtite dokle je on stigao. To su neke stvari koje mi padaju na pamet sad. Kada budete krenuli da implementirate videćete šta vam još nedostaje.

Dakle, ideja je da nakon završetka `init`-a imate sve spremno da se ovo izvršava bez problema. A `restore` sve što je čačnuo `init` mora da vrati. I to je cela filozofija. Dobro, to sad imate napisano i ovde kroz ovaj tekst, možda ne baš ovako ali to ćete već iščitati. I prvi deo koji treba da napravite, a malo je krupniji, jeste koncept niti.

Koncept niti

Kako da krenete da razvijate projekat ? Uzmete onaj četvrti zadatak sa vežbi 2 čini mi se, i da li je isto četvrti zadatak na laboratorijskoj vežbi, to je onaj zadatak gde imate dve niti A i B i gde imate prekidnu rutinu od tajmera, koja je izmenjena da vrši promenu konteksta. Tamo je bila ideja da mi imamo funkciju A i funkciju B, i onda na osnovu tih funkcija u stvari napravimo dve niti koje izvršavaju telo tih funkcija. E, ovde treba da napravite klasu, i ta klasa ima potpis koji jako liči na potpis one klase Thread iz Jave.

```
// File: thread.h
#ifndef _thread_h_
#define _thread_h_

typedef unsigned long StackSize;
const StackSize defaultStackSize = 4096;
typedef unsigned int Time; // time, x 55ms
const Time defaultTimeSlice = 2; // default = 2*55ms
typedef int ID;

class PCB; // Kernel's implementation of a user's thread

class Thread {
public:

        void start();
        void waitToComplete();
        virtual ~Thread();

        ID getId();
        static ID getRunningId();
        static Thread * getThreadById(ID id);

protected:
        friend class PCB;
        Thread (StackSize stackSize = defaultStackSize, Time timeSlice =
defaultTimeSlice);
        virtual void run() {}

private:
        PCB* myPCB;
};

void dispatch ();

#endif
```

Prva najbitnija stvar jeste ovo `virtual void run()`. Definisana je kao virtuelna zato što će ova Thread služiti kao osnovna klasa za izvođenje thread-ova koji će biti pravljeni ovde u `userMain`-u. Nakon tog `run`-a, dakle sva logika thread-a se smešta unutar `run` metode. **Ko**

će da piše run metodu ? Pa onaj ko bude override-ovao, ko bude izvodio iz ove Thread klase. Pored toga imate metodu `start()`. Metodom `start` vi treba samo da kažete da ta nit od tog trenutka postaje konkurent da dobije procesor na izvršavanje. Odnosno, šta treba `start()` da uradi? Treba samo da uzme PCB te niti i da stavi u scheduler. Od trenutka kad se PCB nalazi u scheduler-u, kad dođe do promene konteksta onom metodom `get()` kojom vadimo nešto iz scheduler-a, možda dohvatimo baš taj PCB i krenuće izvršavanje. Dakle, ovo `start()` samo treba da ubaci PCB u scheduler. Pored toga što treba da ubaci PCB u sam scheduler, treba da promenite i stanje niti. Dakle, ta nit, od trenutka kad se ubaci u scheduler, ona prelazi u stanje koje se zove ready, zato što konkuriše da dobije procesor.

Imate ovde i destruktor. Unutar ovog destruktora treba da pobrišete sve pomoćne strukture koje ste koristili za implementaciju vaše niti.

Imate ovde `getId()`. Treba da napravite mehanizam da svaka nit u sistemu ima jedinstven ID. Ovim `getId()` dohvatate ID neke niti.

Imate dve statičke metode. `getRunningId()` vraća ID od trenutne running niti. I imate `getThreadById(ID Id)`, gde vi prosledite ID a dobijete pokazivač na thread sa tim ID-em. Što odmah znači da ovde morate da imate neki globalni niz, ili neku listu još bolje, gde ćete čuvati informaciju o svim nitima koje je neko napravio u vašem sistemu. **Kad ćete da ubacujete nešto u tu listu ? Svaki put kad se izvrši konstruktor Thread-a.** Unutar konstruktora Thread-a uzmete taj this pointer i čuvate u neku globalnu listu Thread-a. Odnosno, nećete verovatno imati globalnu listu Thread-ova, već ćete imati globalnu listu PCB-a. PCB je nešto što nam pruža internu implementaciju niti i to je nešto što je sakriveno od korisnika. Korisnik jedino vidi ove public metode iz klase Thread. Ono što je protected i ono što je private se ne vidi. Dakle, svu implementaciju vezanu za samu nit treba da strpate unutar klase PCB.

Šta će morati klasa PCB da ima ? Moraće sigurno da ima stek. Moraće da ima status, u kom je stanju trenutna nit, da li je blokirana, da li je running, da li je ready, da li je finished, da li je završila svoje izvršavanje. I ko zna šta još. **Dakle sve što treba da čuvate vezano za nit to trpate u klasu PCB**.

Pitanje: Da li može na onom mestu umesto liste ili niza da se koristi šablonska mapa ?
Odgovor: To morate da otkucate i da probate da prevedete sa onim kompajlerom. Ako on može da prevede lepo šablon, onda koristite.

Važna napomena: Ne smete ništa da promenite od potpisa ovog zaglavlja. Dakle ovo uzmete, prekopirate, i napravite thread.h. I ne smete ništa da menjate. Kad kažem ne smete ništa da menjate, od ovog potpisa ne smete ništa da brišete i ne smete ništa, recimo sad ovo se zove `getRunningId()`, vama se to ne sviđa pa promenite da se drugačije zove. Onda vam neće raditi oni testovi, pošto se mi oslanjamo na ovaj interfejs. Ako vam bude trebalo nešto da dodate, smete da dodate. U nekom trenutku će možda stvarno da vam nešto fali, možda neko

prijateljstvo sa nekom drugom klasom, to smete da dodajete. **Možete da proširujete ovo ali ne smete da suzite ovaj interfejs.** I nikako da menjate nazive ovih stvari.

Ovde sad ima kako izgleda taj konstruktor `Thread`-a. On prima dva parametra: jedan je `stackSize`, i ovde je definisana neka konstanta kolika je podrazumevana veličina steka. I imate `timeSlice`. Sad ako se setite onih zadataka, ovaj `timeSlice` u stvari predstavlja broj u koliko onih inkrementa od 55ms će data nit u cugu da se izvršava. Dakle ako kažem ovde `timeSlice=3`, to znači da ta nit kad dobije procesor ona će u cugu da se izvršava 3*55ms. Kad prođe to vreme, o tome vodite računa u prekidnoj rutini tajmera, onda treba da vršite promenu konteksta.

Dobro, rekao sam vam za PCB, dakle interna implementacija niti, sve što vam treba za nit stavljate u klasu PCB.

Ono što nisam ovde još objasnio jeste ova metoda `waitToComplete()`. Metoda `waitToComplete()`, ako imate recimo ovako neki scenario, imam `Thread t1` i imam `Thread t2`. I sad oni se nekako izvršavaju, i u nekom trenutku `t1` pozove `t2.waitToComplete()`. To znači da se ova nit `t1`, pošto je ona pozvala za nit `t2` `waitToComplete()`, zablokira na ovom mestu. `t1` se zablokira na ovom mestu i čeka da nit `t2` završi svoje izvršavanje.

Pitanje: To je `join` u stvari ?
Odgovor: To je `join`, tako je.

E, kad ova završi izvršavanje, onda nekako mora da vidi, dakle `t2` kad završi svoje izvršavanje mora da vidi koje su sve niti čekale da ona završi svoje izvršavanje. U ovom ovde konkretnom slučaju ona će videti da je nit `t1` čekala da se ona završi, i mora da napravite neki mehanizam kojim ćete sada ovu nit `t1` da oslobodite da nastavi izvršavanje, pošto se `t2` završila. **Što znači da klasa `Thread` mora da ima sigurno i jednu listu gde će da čuva pokazivače na PCB klasu ili na `Thread`, kako god hoćete, svih onih niti koje čekaju da se ta nit završi.** To sad samo treba da napravite nekako.

Šta je još bitno a vezano je za ovaj `waitToComplete()` ? Veliki izvor grešaka bude baš ovde, a neke greške utvrdite tek kad dođete na odbranu projekta, tako da budete pažljivi kad budete pravili ovaj `waitToComplete()`. **Gde treba još da se pozove `waitToComplete()` ? U desktruktoru**. Zašto ? Zamislite sledeću situaciju: Da ova t1 ovde nije zvala ovo `waitToComplete()`, nego se izvršavala, ima pointer na `t2`, i kaže `delete t2`. Šta može da se desi ? Može da se desi da se ovo delete zove u trenutku dok `t2` još uvek nije završila svoje izvršavanje. I trebate da sprečite takvu situaciju, ne smete da dozvolite da se obriše nit. **Jer vi ako obrišete nit, dealociraće se njen stek, i onda ova kad treba da dovrši svoje izvršavanje, taj stek više neće postojati.** I doći će do pucanja programa. **I ono što treba da uradite jeste da prvi poziv unutar svakog destruktora, odnosno unutar desktruktora `Thread`-a, bude upravo ova metoda `waitToComplete()`**. I sad kad ovaj pozove `delete t2`, ući će u destruktor, ali će

u destruktoru da naleti na metodu `waitToComplete()`, i ako nit `t2` nije završila on se tu zablokira, i ne može da obriše nit, sve dok se ova nit ne završi. Tek kad se ova nit završi, ona ga isto odblokira sa ovim `waitToComplete()`, i onda ovaj može da obriše tu nit. Dakle, unutar destruktora prva metoda koja mora da se pozove jeste ova `waitToComplete()`. Videćemo posle kako da znate da li je nit završila svoje izvršavanje. Ovako po logici stvari, koji je to trenutak? Kad smo sigurni da je ona završila koju svoju metodu? Metodu **`run()`**. I sad samo treba da nađemo mesto gde ćemo to da logujemo.

Kako da znam kad se ovo `run()` završilo ? Meni ovo `run()` piše korisnik, on neće na kraju da mi kaže `state=finished`. Nego mi moramo negde u samom kodu kernela da nađemo taj trenutak gde smo sigurni da se ovo završilo. Videćete posle kako, imate pomoć ovde.

Unutar ovog header fajla za `Thread` imate još funkciju `dispatch()`, koja je definisana kao globalna funkcija. I ona služi za eksplicitnu promenu konteksta. Dakle nit može da pozove negde `dispatch()`, to znači da se ona u tom trenutku odriče procesora i da neka druga nit može da koristi procesor. **Kako da napravite `dispatch()` ? Pa to ima u onom četvrtom zadatku. Stavite onaj flag `contextOnDemand` na 1 i onda pozovete `timer()`. I kad odete u prekidnu rutinu `timer()` tamo se vidi da je taj flag setovan i izvršiće se promena konteksta.**

Ovde sad ima primer recimo kako neko ko bude dobio vaš kernel može da ga iskoristi. I ovde sad neki producer napravljen tako što je izveden iz klase `Thread`. Pošto taj producer treba nešto da radi smisleno unutar `run()` metode, da puni neki bafer, ovde je redefinisana `run()` metoda. I onda negde u `userMain`-u može da se napravi producer i može da se startuje. Ovo je samo primer kako se koristi ona vaša klasa `Thread`.

Ovde imate opis šta treba da radi `waitToComplete()`, to je ono što sam pričao.

Promena konteksta i preuzimanje

E, ovde sad ima pet nekih pobrojanih stvari kada se sve vrši promena konteksta. Dakle, kako jedna nit, na koji način u stvari, jedna nit može sve da izgubi procesor.

1. Na eksplicitni zahtev niti (sinhrono preuzimanje) pozivom funkcije dispatch() koja vrši eksplicitno preuzimanje.

2. Na pojavu prekida (asinhrono preuzimanje), ali samo uz korišćenje koncepta događaja koji će biti opisan naknadno.

3. Zbog operacije na nekoj sinhronizacionoj primitivi, tj. na semaforu ili događaju, kako je opisano kasnije (sinhrono preuzimanje).

4. Po isteku vremenskog intervala dodeljenog niti za izvršavanje (engl. time slice). Vremenski interval koji se dodeljuje procesu pri svakom povratku konteksta zadaje se prilikom kreiranja niti, kao parametar (umnožak od 55ms). Vrednost 0 ovog parametra označava da vreme izvršavanja date niti nije ograničeno, već će se ona izvršavati (kada dobije procesor) sve dok ne dođe do preuzimanja iz nekog drugog razloga.

5. Eksplicitnim pozivom operacije waitToComplete().

Kaže na eksplicitni zahtev niti pozivom funkcije `dispatch()`. To je dakle kao ovo u kodu napišem `dispatch()` i onda se tog trenutka gubi procesor.

Drugi kaže na pojavu prekida, ali samo uz korišćenje koncepta događaja, koji će biti opisan naknadno. Videćemo posle šta rade događaji pa će vam to biti tada jasno.

Tri, kaže zbog operacije na nekoj sinhronizacionoj primitivi, tj. na semaforu ili događaju. Može da se desi da nit ne može da prođe `wait()` na nekom semaforu, zato što je onaj brojač negativan. Pošto ne može da pređe `wait()` vi ne smete da dozvolite da ta nit nastavi izvršavanje. Morate da promenite njeno stanje u blocked, da je blokirana, i ne smete da je vratite u scheduler. Unutar scheduler-a treba da držite samo niti čije je stanje ready. **Niti koje nisu u stanju ready ne smete da ubacujete u scheduler.** Pošto vi morate te blokirane niti nekako da čuvate, gde ćete da ih čuvate ? Pa ako se zablokirala na semaforu, svaki semafor mora da ima listu blokiranih niti. I onda je samo ubacite u tu listu. Kad se pozove signal, vi onda iz te liste odaberete jednu nit koju ćete da pustite i izbacujete je iz te liste, i stavljate je u scheduler. Videćemo posle kod semafora.

Četiri, po isteku vremenskog intervala dodeljenog niti za izvršavanje. To je onaj `timeSlice` koji se prosleđuje kao parametar. Vi tamo kažete `timeSlice=5`, to znači da će ta nit imati 5*55ms u jednom naletu da se izvršava. Kad prođe to vreme, onda nit treba da pređe u stanje ready iz stanja running i da odaberete novu nit koja će da se izvršava. Kako ćete da merite to

vreme ? Imate tamo na labu, u onom četvrtom zadatku, je to odrađeno. Imate neku globalnu promenljivu, svaki put kad dođe do promene konteksta vi u tu globalnu promenljivu upišete `timeSlice` nove running niti. Svaki put kad se desi tajmer vi tu globalnu promenljivu smanjite za 1, kad dođe na nulu to znači da je `timeSlice` istekao.

Peto, kaže eksplicitnim pozivom operacije `waitToComplete()`. Dakle, recimo ova `t1` kad je pozvala `t2.waitToComplete()`, ako `t2` nije završila izvršavanje ona mora da se blokira. Pošto mora da se blokira, mora procesor da prepusti nekoj drugoj niti.

Imaćete jednu globalnu funkciju čiji je potpis samo `void tick()`. To `void tick()` treba samo da pozovete unutar prekidne rutine od tajmera. Čemu služi ovo `tick()` ? Pa služi nama ako pišemo neke testove da možemo da merimo koliko je vremena proteklo. I ovo `tick()` treba samo da vam se izvrši u onim situacijama kad je stvarno došao hardverski prekid od tajmera, ne ako je neko pozvao tajmer kao običnu funkciju, već stvarno kad je okinuo hardverski tajmer. Dakle u takvim situacijama treba da pozovete samo ovo još `tick()` unutar prekidne rutine na nekom mestu.

Je l' jasno otprilike kako treba da napravite Thread ? Kako da krenete ovo ? Uzmete sad onaj četvrti zadatak i tamo imate one funkcije A i B. I vi sad kažete neće meni Thread da bude A i B, nego ću ja da napravim klasu Thread. Pa ću onda iz klase Thread da izvedem dve klase, klasu A i klasu B, a unutar `run()` metode ću da napišem tela onih funkcija A i B. Šta treba da uradite u konstruktoru Thread-a ? Ako se setite tamo u onom četvrtom zadatku su bile neke metode koje su se zvale `createThread()` valjda, i ona je pravila onaj početni kontekst. Sećate se pakovali smo na stek šta treba. E sad celu tu logiku samo bupnete u konstruktor Thread-a. Odnosno, celu tu logiku pošto je interna implementacija u stvari PCB, celu tu logiku `createThread()` bupnete u konstruktor PCB-a. I onda probate da ono pokrenete bez išta da menjate samo da ubacite klasu Thread, da napravite instance A i B, i da probate da pokrenete.

Ono što je jako bitno jeste na internetu možete da nađete gomilu ovih projekata. Neke stvari prosto ne mogu da se napišu drugačije. Skloni ste tome da prepisujete. To je okej. Vi ako prepišete nešto, jako je bitno da to što budete prepisali razumete. Ako ga razumete onda možete da ga prepakujete malo na vaš neki način. Nemojte da se desi sledeća situacija: da nađete gotov projekat, mislim našli ste vi već verovatno gotove projekte, da uzmete sve to da otkucate, da menjate malo one neke stvari što se razlikuju u odnosu na prethodne godine, i da tek onda prvi put probate da testirate. To verovatno neće da vam radi. Zato je najbolji način da svaki put kad napravite neku smislenu celinu, makar to bila i jedna metoda unutar klase Thread, napišete takve testove da testirate samo tu metodu. I tako deo po deo, čim nešto napravite napišete neke testove da budete sigurni da vam to radi. Ako budete bupnuli ceo kod odjednom, otkucali sve i probali da testirate, ko zna koliko će vam trebati vremena da nađete sve greške. **Dakle jako bitno je da svaki deo testirate čim ga napravite.**

```
// File: schedule.h
class PCB;

class Scheduler
{
public:
        static void put (PCB*);
        static PCB* get ();
};
```

Evo ga ovaj Scheduler, imate da skinete tamo na sajtu predmeta header fajl, samo ubacite tamo gde će biti ostali vaši header fajlovi, i imate onaj Aplicat.lib čini mi se da se zove, njega prekopirate u root projekta, i onda možete da koristite klasu Scheduler. Šta je bitno ovde ? **Nemojte, ni slučajno, da se oslanjate na to koji algoritam koristi ovaj raspoređivač. Dakle jedino pravo ispravno rešenje kako smete da ga koristite je da zovete `put` i `get`, bez o tome da razmišljate kakav je algoritam raspoređivanja.** Ili još gore, da probate da vadite iz Scheduler-a sve dok ne dobijete neku nit koja vam treba zbog nekog testa da bi vam test prošao. To nikako ne radite.

Dve rečenice, koje ću da pročitam od a do š, da čujete samo:

Student treba da bude svestan da je svaka ovakva pojava po pravilu uzrokovana nedostatkom u samoj realizaciji, a ne nekim drugim spoljnim faktorima. Ovo se odnosi na greške koje mogu da vam se nađu. Sve ovakve nedostatke studenti treba sami da otkriju i razreše, bilo čija pomoć može samo da olakša rešavanje, ali nikako i da reši problem. Ne zbog toga što neko drugi, uključujući i nastavnika i asistenta, ne želi da pomogne, već što jednostavno to nije u stanju. Ja nekad kad sam bio mlad, držao sam te konsultacije pa dođe čovek i kaže meni „Ovo ne radi, puca“. I onda se desi da ja to debagujem jedno tri četiri sata, i da nađem možda jednu grešku, ili nijednu grešku. Prosto, neke stvari se jako teško uočavaju, možda je ovo najveći projekat do sada koji radite, ali nema tu puno koda. Ako naletite na neki problem, prvo pitate kolege sa kojima se družite, i one sa kojima se ne družite, da li je neko naišao na takav problem. U 99% slučajeva verovatno je neko već imao problem koji vi imate, pa može da vam kaže kako ga je rešio i onda da ga vi, na sličan ili isti način, rešite. Ako ne možete da rešite, možete nama da se javite, ali ne ulazimo u debagovanje koda. Možemo da vam pričamo idejno kako treba nešto da pravite, i vi nama da kažete kako ste napravili, pa možda tako da vam pomognemo da nađete grešku. Bio je jedan kolega koji mi je slao, bogami jedno godinu dana, kako će on da odustane od ovog predmeta, kako ne može više, kako ne zna više šta da radi, dolazio dva puta da debagujemo projekat. I ja mu svaki put rekao „Pa kolega, krenite ispočetka.“ A njega je verovatno mrzelo, i onda je verovatno jednom seo i krenuo ispočetka, uradio ga je i odbranio. Dakle samo nemojte da odustajete. A i nemojte da šaljete toliko mejlova.

Imate sad ovde neke posebne napomene koje se odnose na samu promenu konteksta. Od tih posebnih napomena ja ću da vam skrenem pažnju na dve neke stvari. Prva stvar jeste da, vi ćete najčešće vaš projekat da debagujete tako što ćete da radite neki ispis. Možete da koristite

printf, možete da koristite cout, šta god hoćete. Ono što je bitno jeste da treba da obezbedite da se ti ispisi urade atomično. Problem može da nastane što recimo cout, ako mu date neki string, ne stigne da izvrši sav taj upis u neki bafer, koji nakon toga ispisuje u konzolu. Može da ga prekine neka druga nit, i da vam ispis pukne na pola. I onda će sve to nešto da se poremeti. **Iz tog razloga je najbolja praksa ako radite print debug, jeste da pre nego što pozovete cout ili printf, uradite lock, tako što ćete da zabranite prekide, ili da dozvolite prekide a zabranite preuzimanje, pa onda radite cout, i nakon toga zovete unlock, gde dozvoljavate prekide i dozvoljavate promenu konteksta. Dakle bitno je da se ispis uradi atomično.**

Ono što je bitno, projekat koji budete predali, ne sme da sadrži nikakav dodatni ispis. Dakle, sve što budete koristili za testiranje morate da pobrišete pre predaje.

Napomena: Možete da imate neke male kratke komentare. Ima ljudi koji prepišu tuđe projekte, pa pošto vas tamo neko naravno ispituje malo o projektu da vidi da li nešto znate, oni nemaju pojma šta su predali pa onda napišu komentare sa strane da bi mogli kao tamo da čitaju šta im to radi. Nemojte to da radite.

Kad budete predali projekat, radi se provera sličnosti koda. Nemojte da se plašite toga, zato što postoji tolerancija, jer prosto znamo da neke stvari se pišu na slične ili gotovo iste načine, pa postoji neki prag koji tolerišemo. Ali radovi koji budu slični nekim drugim radovima, takvi studenti onda moraju da podlegnu disciplinskoj komisiji. I mi onda imamo neoboriv dokaz da ste vi prepisivali zato što softver vrati tačno kojim radovima je sličan vaš rad, tačno u liniju. Vodite računa da se ne porede radovi samo iz tekuće generacije, nego u bazi su verovatno radovi od kad postoji ovaj projekat. Tako da šta god što nađete to verovatno i mi imamo. Taj softver vrati nama s leve strane jedan rad, s desne strane drugi rad. I tačno linije uparuje koje su iste. Ako je recimo ulančavanje u listu, ja tu ne mogu da vam kažem prepisivali ste, uzeli ste Laslovu zbirku i prepisali ste jednostruko ulančanu listu. E, a recimo problem je ako ja otvorim prekidnu rutinu tajmer i vidim tamo da se svaka naredba gađa sa svakom naredbom. Pa sad, oni probaju da pređu sistem, pa ne znam ovaj ga nazove timeslice a ovaj ga timeSlice (camel case). Možete da ga nazovete i Pera i Žika, on poredi naredbe, tako da bićete uhvaćeni.

Da se vratimo na ono što vam treba da bi uopšte počeli da radite projekat. Vidite ovde da se unutar Thread-a metoda `run()` nalazi unutar protected sekcije. Ako se setite kako smo radili onaj četvrti zadatak, tamo smo morali na stek da stavimo početnu adresu onih funkcija A i B odakle kreće izvršavanje. Ovde moramo da imamo mehanizam da na stek, kao adresu odakle treba da počne izvršavanje niti, podmetnemo adresu `run()` metode. Gde će da se nalazi ova `run()` metoda ? Pitaj boga. Ja kad prevedem kod, i kad se moje klase učitaju u memoriju, to će biti negde razbacano i neću moći nikako da dohvatim adresu od ovog run-a, odnosno od bilo kog run-a bilo koje klase. I onda je ovde primenjena sledeća ideja, recimo ovakav deo koda morate da imate svi, možda može na neki drugi način da se izvede ali ovo vam je nešto najlakše i nešto što vam je dato u projektu.

```
void PCB::wrapper()
{
        PCB::running->myThread->run();
...
```

Unutar PCB-a, dakle PCB vam je ona interna implementacija niti, treba da napravite jednu statičku metodu, koja će da se zove wrapper ili kako god, i ta statička metoda će unutar sebe imati sledeću stvar: `PCB::running->myThread->run();` Running će biti globalni pokazivač na PCB čiji Thread se trenutno izvršava, onda ćete da dohvatite Thread, i onda ćete da pozovete run. E sad, kako ovo može da vam pomogne ? Pošto je ovo statička metoda, možete da je definišete kao javnu statičku metodu unutar klase PCB-a. Pošto je statička metoda, nema parametara, pa ne morate ništa preko steka da joj prosleđujete, a pošto je statička javna možete da dohvatite adresu wrapper-a. I onda adresu ovog wrapper-a podmetnete na stek niti kad se bude pravila. Kad krene izvršavanje, šta će da se desi ? Skupiće se adresa wrapper-a, doći ću u ovu statičku metodu, u tom trenutku running će da pokazuje baš na tu nit koja treba da se izvršava, tako da će se preko running-a doći do myThread-a a onda će se preko myThread-a doći do run metode. Nemojte da vas ovo buni, ovo running je pokazivač na PCB, a unutar PCB-a svaki PCB ima i pokazivač na svoj Thread. I onda za svaku nit na steku, kao prvu adresu odakle treba da krene izvršavanje, podmećete ovaj wrapper.

Pitanje: Da li će biti testova koji će imati neke, da kažem greške, na primer da nit sama za sebe pozove `waitToComplete()` ili tako nešto?
Odgovor: Morate da pokrpite sve te situacije. Da li će neko namerno da piše takve testove ? Možda. U javnom testu verovatno nema takvih stvari. Ali kad dođete na modifikaciju, ako postoji test za modifikaciju možda bude tako neka stvar. Zato sam rekao da tek tamo ustanovite šta vam sve ne radi. Dozvoljeno je da promenite taj kod koji ste doneli kad dođete na odbranu. Dakle nemojte da vas hvata panika ako se setite da ste nešto loše uradili, možete tamo da promenite na licu mesta.

Pitanje: Šta treba da se desi ako nit sama za sebe pozove `waitToComplete()` ?
Odgovor: Ignorišeš takav poziv. Ne smeš to da dozvoliš. Taj `waitToComplete()` će imati jedan if koji će da ima jedno četiri pet uslova da li sme uopšte da se pozove. A to će da vam se javi sve u svoje vreme kako budete razvijali projekat.

Je l' vam jasno ovo za wrapper kako treba da podmetnete početnu adresu ? Kako da znam da se nit završila ? Koje je to mesto ? Evo, sledeća naredba ispod run-a mi kaže da se nit završila. Dakle, ovde ako hoću da znam kad se nit završila, to je mesto ispod ovog poziva run. Za one ljude koji budu radili signale, tamo ima neka dva signala, jedno se javlja roditelju da se nit završila, a drugi signal se šalje svakoj niti kad se završi. Gde je mesto gde treba da pozovete ta dva signala ? Mislim da su signali jedan i dva. To je ovde.

Koncept semafora

```
// File: semaphor.h
#ifndef _semaphor_h_
#define _semaphor_h_

typedef unsigned int Time;

class KernelSem;

class Semaphore
{
public:
        Semaphore (int init=1);
        virtual ~Semaphore ();

        virtual int wait (Time maxTimeToWait);
        virtual int signal(int n=0);

        int val () const;  // Returns the current value of the semaphore

private:
        KernelSem* myImpl; };

#endif
```

Ajmo da vidimo koncept semafora. Ovi semafori su, odnosno ova klasa Semaphore koju treba da napravite je potpuno identična, ima sličan potpis kao klasa Semaphore koju ste videli na predavanjima, samo su malo izmenjene metode `wait()` i `signal()`. Metoda `wait()` i metoda `signal()` videćete da primaju neke parametre. Metoda `wait()` prima jedan parametar `maxTimeToWait` i to predstavlja vreme koliko nit treba da čeka na tom semaforu. Dakle, ako ja ovde prosledim broj 5, to znači da ta nit treba samo da bude blokirana 5 timeSlice-ova (5*55ms, isto su oni inkrementi), i nakon toga treba da prođe kroz taj semafor. Takvo je ponašanje ako se prosledi nešto što je veće od 0. Ako se prosledi 0, onda se `wait()` ponaša na standardan način. Dakle, gledate onaj brojač value kakav je, i onda na osnovu toga ili zablokirate nit ili je pustite da prođe. Kako ćete da implementirate ovakav `wait()`? Morate svaki put kad vam se desi prekid od tajmera da imate jedan niz ili listu svih semafora u sistemu, i za svaki semafor da proveravate da li postoje niti na tom semaforu koje čekaju neko određeno vreme. I svaki put kad dođe tajmer vi treba da označite da je jedan inkrement vremena protekao, i onda tako imate evidenciju o tome da je vreme isteklo, i onda pustite nit da nastavi. Pošto ovaj semafor mora nekako te blokirane niti negde da čuva, on će verovatno imati jednu listu blokiranih niti na ovom semaforu, ili možda čak i dve liste. **Jedna lista da služi za one koje su se blokirale zato što je value negativno, a druga lista da čuvate sve one niti koje su se zablokirale zato što su pozvale `wait()` sa nekim vremenom.** Sve niti koje su blokirane ne stavljate u Scheduler, i njihovo stanje jeste neko koje će da se zove blocked. U trenutku kad nit

treba da se odblokira, stanje joj menjate u ready, izbacujete je iz tih listi sa semafora, i vraćate u scheduler. Ko može da odblokira nit ? Pa može `signal()`, ili da je ovo vreme isteklo tamo u tajmeru. Ako tamo u tajmeru ustanovite da je vreme isteklo, onda nit izbacujete iz odgovarajuće liste i vraćate je u scheduler. Ovaj header fajl ne smete da menjate. Evo još jedna stvar recimo kad upoređujemo kod, svi ćete imati iste header fajlove, i on će tamo da prikaže sličnost ali nećemo zbog toga da vas zovemo na odgovornost. Isto kao što Thread internu implementaciju ima kroz PCB strukturu, kernelSem će biti klasa koja će da predstavlja internu implementaciju semafora. Kako u stvari treba da izgledaju sve ove metode `wait()`, `signal()` i šta ti ja znam šta već ovde ima kod semafora ? Recimo `signal()` u semaforu, to će biti samo wrapper oko poziva koji će reći ovako `myImpl->signal();` tako isto izgleda i `wait()` i sve ostalo. **Dakle kad neku metodu odavde pozivate, ona treba da pozove odgovarajuću metodu iz kernelSem**.

Napomena: Vezana za ograničenje kod onog prevodioca. To su vam verovatno rekli na labu, možda sam vam i ja rekao. **Naziv header fajlova i klasa ne sme da vam ima više od 8 slova, odnosno .cpp i .h fajlovi ne smeju da imaju više od 8 slova (bez ekstenzije).** Ako ima više, ovaj prevodilac može da pravi problem.

Kako radi `signal()` ? Signal isto ima malo promenjeno ponašanje. On prima jedan parametar, i taj parametar ako je 0 onda se ponaša na standardan način. Ali ako je n>0, to znači da ja treba toliko niti da oslobodim sa tog semafora u tom trenutku. **Dakle ako prosledim 2, treba dve niti da oslobodim sa tog semafora.** Šta da radim ako nemam dve niti ? Ništa, probao sam da oslobodim, nemam koga da oslobodim. Ali ono što je bitno jeste da se value na semaforu uvek uvećava za ovo n, ako je n>0. **Dakle ako je n = 2 i nisam imao šta da oslobodim, ja ću value da povećam za 2**. Povratna vrednost u situaciji ako sam imao niti koje mogu da oslobodim, ova povratna vrednost predstavlja broj oslobođenih niti, ovaj int koji `signal()` vraća. **Ako neko prosledi negativnu vrednost, signal treba da bude bez dejstva, value ne dirate, i samo vratite tu negativnu vrednost kao rezultat**. To vam je objašnjeno kroz dva pasusa o signal-u i o wait-u. I tako napravite semafor.

Pitanje: Šta radimo u graničnim slučajevima, kada neko hoće da overflow-uje ponašanje, na primer da je value MAX_INT i da neko uradi `signal(5)` ?
Odgovor: Ne morate da vodite računa o tome. Da li će neko da natera ovo value da postane MAX_INT ? Neće.

Pre nego što krenemo na događaje, događaji su nešto što je možda ovde najkritičnije da se napravi u ovom prvom delu, sad znate za semafor. Može da se desi sledeća situacija: da su sve niti koje imate u sistemu u stanju blocked, i onda nemate šta da izvršavate, a onaj main tamo ne možete da završite. **U situaciji kada nemate šta da izvršavate, vi morate da obezbedite da ipak nešto trči po tom procesoru. I to što će da se izvršava u situaciji kad nijedna korisnička nit nije u stanju ready jeste idle nit**. Dakle, u onom delu, ako se setite ovde da sam napisao `init()`, u tom delu treba da napravite i tu idle nit, koja će u stvari biti kao klasa Thread, samo što će njena

`run()` metoda da radi nešto bezveze i ne sme nikad da se završi. Dakle morate da je zavrtite u neku beskonačnu petlju. **Ono što morate da vodite računa jeste `waitToComplete()` za idle. Ako je ona u beskonačnoj petlji, neće nikad da se završi**, pa to morate nekako da sredite.

Kako da prepoznate situaciju kad nijednu nit ne možete više da izvršavate jer su sve blokirane ? **Ako vi pitate Scheduler da vam vrati jednu nit za izvršavanje, i Scheduler vam vrati nullptr**, to znači da u Scheduler-u nema nijedna nit koja je u ready stanju, a u Scheduler smo rekli da stavljamo samo niti koje su u ready stanju. E, **to je situacija kad onda treba da pustite da se izvršava idle nit**. Koliko dugo treba da se izvršava ta idle nit ? Pa svaki put joj dajete timeSlice 1. I onda će ona da se izvršava do narednog prekida od tajmera. Kad se desi prekid, onda će opet da se proverava Scheduler. Ako Scheduler sad ima neku nit koja je u ready stanju, onda će ta nit da krene da se izvršava, a idle nit ostaje po strani. **I idle nit nemojte da ubacujete u Scheduler**.

Koncept događaja

```
// File: event.h
#ifndef _event_h_
#define _event_h_

typedef unsigned char IVTNo;
class KernelEv;

class Event
{
public:
        Event (IVTNo ivtNo);
        ~Event ();

        void wait  ();

protected:
        friend class KernelEv;
        void signal(); // can call KernelEv

private:
        KernelEv* myImpl;
};
#endif
```

I u ovom prvom delu za 20 poena ostao je još koncept događaja. Ja ću sad da probam da vam skiciram otprilike šta treba da napravite. Dakle, prvo ideja za koncept događaja jeste sledeća: da vi nekako za neki prekid vežete prekidnu rutinu koja će nad objektom klase Event da pozove `signal()`. Svaki put kad se desi prekid treba da se pozove `signal()`. Kako izgleda da klasa

Event ? Ona ima potpis koji jako liči na semafor, u suštini Event se i ponaša kao binarni semafor. Ovde imamo konstruktor, obratite pažnju, taj konstruktor jedino prima broj prekida, odnosno broj ulaza unutar IV tabele za koji želimo da vežemo ovaj Event. Ima destruktor, ima metodu `wait()`, i ovde ima metodu `signal()`. Ova metoda `signal()` je stavljena u protected zato što nju spolja ne bi trebalo niko da može da pozove, jer taj `signal()` isključivo treba da se zove iz prekidne rutine. I naravno, kao što je i Thread, kao što je i semafor imao internu implementaciju, i ovde je ta interna implementacija sakrivena u klasi KernelEvent, a ovo su samo omotači.

Napomena jedna kod Event-a koja stoji u tekstu zadatka kaže sledeće: `wait()` na Event-u može da pozove samo nit koja je napravila taj događaj. **To znači da Event ima u stvari vlasnika, a vlasnik je ona nit koja je napravila Event**. Kako da uhvatite ko je vlasnik ? **U trenutku kad se izvršavao konstruktor Event-a vidite koja nit je running, i taj running je u stvari vlasnik Event-a**. Ako neka druga nit pozove `wait()`, a ona nije vlasnik, `wait()` treba da bude bez dejstva. Dakle ništa da se ne desi, samo vlasnik može da se blokira na Event-u.

I sad ovde imate opisano kako treba da izgleda struktura toga što treba da se napravi. Dakle, morate da imate klasu Event, i morate da imate klasu KernelEvent. E, ono što još morate da napravite jeste klasa koja se zove IVTEntry. I ta klasa IVTEntry mora da ima neki konstruktor. I unutar tog konstruktora ono što treba da uradite jeste sledeće: taj konstruktor će da primi broj prekida, odnosno ulaz unutar IV tabele. I ono što treba da uradi taj konstruktor jeste sledeće: da pročita adresu prekidne rutine koja se nalazi na ovom broju unutar IV tabele, da to sačuva negde. Gde će da čuva ? Pa u objektu klase IVTEntry. Dakle, da sačuva staru prekidnu rutinu. I na mesto gde je bila ta stara prekidna rutina da podmetne novu prekidnu rutinu. Nova prekidna rutina treba da zove `signal()` nad ovim KernelEvent-om. To znači da ovaj objekat IVTEntry, on mora da ima i pokazivač na nešto što se zove KernelEvent. Dakle ovaj odavde, svaki objekat mora da ima pokazivač na KernelEvent. I kad se desi prekid, ta nova prekidna rutina samo treba da pozove `signal()` nad ovim ovde. Šta to znači ? To znači da morate da obezbedite neki mehanizam kako ćete ovaj pointer koji pokazuje na KernelEvent da dodelite. Kako ćete da postavite njegovu vrednost da on pokazuje baš na određeni KernelEvent ? **Ova klasa IVTEntry, u okviru svog konstruktora, treba da sačuva staru prekidnu rutinu i da podmetne novu prekidnu rutinu u određeni ulaz**. I to je to. Kako sad da ja dobijem tu novu prekidnu rutinu ? I ovo je sad nešto što svi prepišete, a ima ljudi koji ne razumeju šta to radi. Treba da napravite jedan makro, koji se zove PREPARE_ENTRY. Ja neću sad da vam ga napišem, napisaću samo strukturu njegovu, šta on treba da uradi. To ćete da nađete, ima ga koliko hoćete. On prima neki broj, taj broj jeste broj ulaza unutar IV tabele, i prima neki flag. Ono što on treba da uradi jeste sledeća stvar: prvo što mora da uradi jeste da napravi deklaraciju prekidne rutine. Kako da napravi deklaraciju ?

```
#define PREPARE_ENTRY(No, flag)
        1. deklaraciju prekidne rutine
                void interrupt intr();
        2. jedan objekat IVTEntry
                IVTEntry ivte(No);
        3. void interrupt intr()
                {
                        ...
                        ivte.signal();
                }
```

Pa kažemo `void`, ključna reč `interrupt`, pa onda kažemo kako se zove ta prekidna rutina, recimo `intr()`, to nema parametre, i ;. Druga stvar koju on mora da uradi jeste da mora da kreira jedan objekat klase IVTEntry. Kako će da kreira objekat klase IVTEntry ? Pa ovde može da napravi statički objekat, da kaže IVTEntry, i sad da se to zove recimo ivte, i da pozove ovde konstruktor. Onaj konstruktor tamo prima samo broj ulaza. To je druga stvar koju on treba da uradi. I treća stvar koju ovaj makro mora da uradi jeste sledeće. Prvo ovo da vas pitam. Zašto sam prvo morao da napišem ovu deklaraciju intr, pa tek onda da pravim objekat ? Zato što će unutar ovog konstruktora ovaj da traži adresu ovog. Pa da bi znao gde se ovo nalazi, jer smo rekli da on mora tu adresu da podmetne u određeni broj ulaza. E, nakon toga, kad ovaj napravi objekat IVTEntry, moramo da napišemo definiciju sad ovog interrupt-a. I onda ovde kažemo `void interrupt intr()`, i sad ovde ide telo te prekidne rutine. Šta treba da uradi ta prekidna rutina ? Ona samo treba da kaže `ivte.signal();`. `ivte` će ovde recimo imati metodu `signal()`, i taj `signal()` šta treba da uradi ? On treba da kaže, pošto ima ovaj pokazivač na KernelEvent, on treba da kaže `KernelEvent->signal();`. Ili recimo da ovaj ovde ima neku metodu kojom dohvata pokazivač na KernelEvent i onda da pozove `signal()` nad KernelEvent-om. Kako god hoćete. Ovde mora da se izvrši poziv `signal()`-a nad KernelEvent-om unutar prekidne rutine. I čemu služi još ovaj flag ? Ovaj flag vam govori, ako je 1, to znači da treba da pozovete i staru prekidnu rutinu. I onda još u telu ove prekidne rutine morate da imate jedan if kojim ćete da pitate kakav je flag. Ako je flag 1 onda treba da zovete i staru prekidnu rutinu. Gde sam sačuvao staru prekidnu rutinu ? IVTEntry će imati jedno polje koje će da predstavlja adresu stare prekidne rutine. I rekli smo da se to polje postavlja unutar konstruktora. Odnosno kad se bude ovo pravilo, ja ću u nekom polju da sačuvam adresu stare prekidne rutine. I onda ovde treba da pozovem samo staru prekidnu rutinu ako mi je ovaj flag setovan.

Ispričaću još jednom ovu strukturu. Dakle, vama je samo nezgodno što IVTEntry mora nekako da zna za KernelEvent, to morate da se snađete kako ćete to da uradite, i ovaj PREPARE_ENTRY bi trebao da ima ovakvu strukturu. Šta je problem sa ovakvim rešenjem, ako je ovako makro napisan ? Šta se desi ako neko pozove dva puta ovaj makro ? Pukne. Ja sam ovde nazvao ivtr i nazvao sam ivte. Ako mi neko pozove ovaj makro, sa različitim parametrima dva puta, dobiću dve prekidne rutine koje se zovu ivtr i dva objekta koji se zovu ivte. Vi sad kad budete prepisivali ovaj makro, vi dobro naučite šta rade dve tarabe, ##, koje se pojavljuju ovde negde. To treba da

radi makro, imamo IVTEntry, još jednom ponavljam, **IVTEntry mora da sačuva staru prekidnu rutinu, da ima pokazivač na KernelEvent, i da ima metodu `signal()`, i unutar te metode `signal()` treba da pozove `signal()` nad KernelEvent-om**.

Kako se koristi ovo ? Kako neko ko će da testira da li vam rade Event-i, kako će ovo iskoristi ? Ovako… Prvo, kad neko hoće da koristi Event-e na ovakav način napravljene, on mora prvo da pozove PREPARE_ENTRY. Dakle, kako se koristi, prvo se zove negde PREPARE_ENTRY, pa kažem recimo ulaz 252, i kažem 1, a nakon toga mora da napravi objekat klase Event.

```
PREPARE_ENTRY(252,1);
Event ev(252);
```

Klasa Event je ova klasa koja je opisana ovim header-om, i ovo se zove recimo ev, i prosledimo isti taj broj ulaza. I sad ovde je samo ostalo, dakle ovo je kako se koristi, ovde sad samo treba još da uočimo jednu stvar. **Kad se bude pravio Event, mora da se unutar tog konstruktora napravi i KernelEvent**. Pošto se Event pravi nakon PREPARE_ENTRY-a, to znači da će ovaj objekat IVTEntry već postojati. I ja sad moram da nađem način kako da ovaj pointer ovde postavim da pokazuje na KernelEvent, ako se on pravi tek nakon PREPARE_ENTRY-a, nakon ove klase IVTEntry. **Ovde definišemo pokazivač, koji kad se izvrši konstruktor on ostaje null ili šta već, a onda kad se ovaj Event pravi, ovaj ovde treba da ima metodu kojom se setuje pokazivač**. Kako ćemo samo da znamo gde je ovaj IVTEntry ? Gde je taj objekat ? Ono što vam piše u projektu jeste sledeće: **morate da imate jedan globalan niz IVTEntry-a od 256 elemenata. I brojevi ulaza koji će da se koriste za preusmeravanje ovih prekidnih rutina jesu od 0 do 255. I vi sad kad pravite ovde Event sa brojem 252, vi onda iz ovog globalnog niza izvučete pokazivač na IVTEntry**. IVTEntry, ovaj ovde, kad se bude izvršavao njegov konstruktor, on treba `this` podmetne na određenu lokaciju u ovom globalnom nizu. I pošto ovaj globalni niz sadrži sad sve pokazivače na IVTEntry, onda ovaj kad pravi Event nema problema, može da dođe do IVTEntry-a, koji je napravljen pozivom makroa PREPARE_ENTRY.

Dobro, ovo možda deluje konfuzno. Ovo sam pričao i prošle godine, ništa se nije promenilo od prošle godine pa možete da gledate onaj snimak.

KRAJ DELA ZA 20 POENA

Signali

Šta je ideja kod signala ? Ovo nije teško da napravite i moj vam je savet da probate da uzmete i tih 10 poena. Sećate se da je tamo postojalo zaglavlje klase Thread. Ko bude radio za 30 poena on treba ovo zaglavlje klase Thread da proširi na ovakav način. Da doda ovih 8 metoda.

```
typedef void (*SignalHandler)();

typedef unsigned SignalId;

class Thread
{
        ...
        void signal(SignalId signal);
        void registerHandler(SignalId signal, SignalHandler handler);
        void unregisterAllHandlers(SignalID id);
        void swap(SignalId id, SignalHandler hand1, SignalHandler hand2);

        void blockSignal(SignalId signal);
        static void blockSignalGlobally(SignalId signal);
        void unblockSignal(SignalId signal);
        static void unblockSignalGlobally(SignalId signal);
}
```

Dakle sve ovo što ima Thread, plus ovo. To je za ljude koji rade za 30 poena. Savet moj, nemojte, pošto su potpuno razdvojeni, dakle uradite prvo za 20 poena, a ovo za dodatnih 10 je nadogradnja one verzije od 20 poena. Tako da prvo uradite za 20, testirate da vam to za 20 radi, i onda krenete da dodajete ove stvari za 30.

Šta je ideja ? Ideja je sledeća. Nit t1 može da kaže niti t2 signal(), dakle ovo je sad metoda klase Thread, i može da prosledi neki broj, recimo 5. To znači sledeće. Da je nit t1, pošto se ona ovde izvršavala, poslala signal broj 5 niti t2. Nit t2 treba da odreaguje na taj signal tako što će da pozove neku funkciju. Funkciju koju treba da pozove ovde smo nazvali SignalHandler, i to je funkcija ili funkcije koje su vezane za signal broj 5. Dakle ova nit t2, kad se pozove signal(5), ona mora da zapamti prvo da je dobila signal broj 5, neki flag da podigne na 1. U trenutku kad bude proveravala koji su signali stigli, ona će videti da je stigao signal broj 5, i treba da pozove sve funkcije koje su vezane na signal 5. Odnosno sve handler-e tog signala. Ovde smo rekli da svaka ova nit može da ima maksimalno 16 signala, sa brojevima od 0 do 15. Za svaki taj signal može biti vezana nijedna, jedna ili više SignalHandler funkcija. Što znači, recimo na signalu 5, ako su vezane neke dve funkcije, Thread 2 kad bude obrađivao svoje signale, moraće za signal 5 da pozove obe te dve handler funkcije. Ako ima jednu pozvaće samo jednu. U kom redosledu se pozivaju te handler funkcije ? **Onako kako su dodavane signalu**. Dakle signal 5, prvo smo mu dodali jednu funkciju, pa smo mu dodali još jednu funkciju, i treba u tom redosledu i da se pozovu kad se izvrši signal(5).

Kako sad ja postavljam te handler funkcije ? U klasi Thread postoji registerHandler metoda koja prima broj signala, to je ovo SignalId, i prima nešto što se zove SignalHandler. To je gore definisano kao pokazivač na funkciju koja vraća void i ne prima parametre. Što znači da vi za svaku klasu morate da imate za svaki signal unutar te klase listu njenih handler-a. Kako ćete to strukturno da organizujete to je na vama, to je ono što morate da imate. Na takav neki način ga čuvate. Kad se desi signal, onda odete u tu listu nađete handler-e za signal broj 5, i pozovete sve handler-e za taj odgovarajući signal. Ovo možete da shvatite idejno kao da jedna nit ima mogućnost da pošalje prekid drugoj niti, a da su signalHandler-i u stvari prekidne rutine koje su vezane za te brojeve signala. Pored toga što možete da nakačite handler na neki signal, imate i metodu `unregisterAllHandlers()` koja briše sve handler-e za određeni signal. Dakle mogao sam na nekom signalu da narokam 5 handler-a, kad pozovem ovu metodu treba da očistim tu listu. Dakle da nema više nijednog handler-a.

Sledeća metoda je metoda `swap()`, koja menja redosled pozivanja dva handler-a. Dakle ako je neka nit imala recimo za signal broj 5, imala je `f1()` pa `f2()` pa `f3()`, ova swap metoda ako joj se prosledi 2 i 3 treba da zameni mesta pozivanja ova dva handler-a. Šta se prosleđuju kao parametri ? Prosleđuje se broj signala, i prosleđuju se pokazivači na te dve funkcije, odnosno na ta dva handler-a.

Pitanje: Kako treba da proveravamo da li je signal stigao ?
Odgovor: To treba malo da mućnete glavom. Imate još dva minuta pa ću ja da vam kažem gde možete.

Ovo nije teško, samo ima da se sedi i da se malo čuka.

Pitanje: Za swap, ako prođe nešto neregularno ?
Odgovor: Bez dejstva.

Postoji još opcija, kao što kod prekida možete da zablokirate odnosno da maskirate prekide, ovde možete da blokirate signale, i da kažete sledeću stvar: postoji globalno blokiranje i postoji selektivno blokiranje samo na nivou jednog Thread-a. Šta znači da je signal blokiran ? To znači da nit t1 može da pozove `signal(5)`, ali ako je taj signal 5 blokiran kod niti t2, nit t2 treba samo da zapamti da je taj signal stigao, ali ne sme da pozove njene handler-e. **Kad će da pozove njene handler-e ? U trenutku kad taj signal postane odblokiran.** Dakle treba da se zapamti da je stigao signal, ali ne smete da pozovete njihove handler-e. To je ideja sa blokiranjem. I sad ovde još imate globalno blokiranje, kažete da za sve niti, za celu klasu Thread, je blokiran signal 5. A možete da kažete i da samo za nit t2 je blokiran signal 5. Ponašanje je isto, samo skup nad kojim se blokiranje radi se razlikuje.

U kom trenutku ćete pozivati i obrađivati sve signale koji su stigli ? Idealno bi bilo ako bi mogli tako da napravite, ali ne znam kako to da izvedete, nisam razmišljao mnogo o tome, možete da probate čim on pozove signal(5), da se ovom nekako oduzme procesor, pa da se ishendluju ti, handler te funkcije, pa da se vrati ovom procesor. To je jedan od načina. Drugi način kako

možete da uradite, pošto je ovde samo, ovim nećete ništa mnogo da dobijete. Ovaj način što sam sad ispričao bio bi dobar da imate više od jednog core-a, da imate dva procesora, jer bi to značilo da stvarno mogu u paraleli da trče dve niti. Pošto ovde imate samo jedan procesor, vi možete te signale da obrađujete na sledeći način. Kad odaberete tamo u tajmeru, gde ćete verovatno vršiti promenu konteksta, kad odaberete nit t2 da je ona ta koja treba da krene sad izvršavanje, pre nego što izađete iz tajmera vi ste odabrali ovu nit t2, i tad uradite proveru, da li dok je ona bila u Scheduler-u, da li su njoj pristigli neki signali i koji, i ako jesu, vi pre nego što izađete iz tajmer prekidne rutine pozovete sve handler-e. Onda se handler-i izvrše, završi se hendlovanje svih signala i onda se nastavlja izvršavanje niti t2. Dakle kad je odaberete ponovo za izvršavanje to je trenutak kad možete da proveravate koji su signali stigli.

Treba da napravite za signale 0, 1 i 2 neko podrazumevano ponašanje. To podrazumevano ponašanje znači da vi treba da napišete u stvari 3 handler-a koje ćete da vežete *…upad tetkice…*

Kaže ovako: postoji podrazumevana funkcija za obradu signala sa rednim brojem 0, koja nasilno prekida nit i oslobađa sve resurse koje je ona zauzela. Dakle mogu da ubijem neku nit. Ova recimo nit t1 može da kaže `t2.signal(0)`, to znači da se ubija nit t2. Vi morate da napišete signal koji će to da radi. Odnosno, da napišete handler koji ćete da vežete za signal 0. Gde ćete da radite to vezivanje i to postavljanje ? U onoj `init()` metodi koja je bila ovde na prvom času. Dakle pre nego što krene `userMain()` morate sve to da postavite.

Kaže: podrazumevanu funkciju koja obrađuje signal za redni broj 0 treba implementirati. Dakle, to morate vi da napišete kao deo projektnog zadatka. Obezbediti da se signali, za redne brojeve 1 i 2, šalju od strane sistema. Dakle, signali 1 i 2 se šalju od strane sistema, što bi značilo da treba da onemogućite da korisnik nekoj niti pošalje signal 1 ili signal 2.

Signal sa rednim brojem 1 se prosleđuje niti kada se nit koju je ona napravila završi. To znači da dete prosleđuje roditelju signal sa rednim brojem 1 kad se završi. Šta je dete šta je roditelj ? Ako se izvršava t1, i ovde pozovem `new t3` pa napravim t3, ova sintaksa ne postoji, lupam, ali da shvatite samo šta je poenta. Nije baš kao fork jer ovo nastavlja da se izvršava ali je napravila t3. E onda je t1 roditelj t3, i kad se t3 završi, ona mora da pošalje `signal(1)` niti t1. Dakle javlja roditelju da se završila.

Signal za redni broj 2 se prosleđuje niti kada se ona završi. Dakle svaka nit kad se završi, njoj treba da se pošalje još signal broj 2. Gde je mesto gde ćete da šaljete ta dva signala ? **Wrapper ispod poziva run metode.**