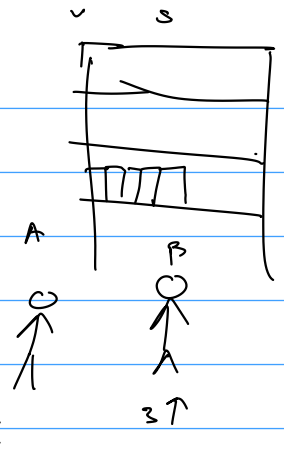



```
Book Shelf {
    int bc;

    void take(int k) {
        bc -= k;
    }

    void put(int k) {
        bc += k;
    }
}
```

```
bc = 5;
```

[6]

t_A t_B

s.take(2) s.put(3)

take:

```
load R5, #bc
sub R5, R5, R1
store R5, #bc
```

put:

```
load R5, #bc
add R5, R5, R1
store R5, #bc.
```

[1 процесор]

bc := 5

t_A и t_B се изпълняват конкурентно.

t_A:
```
load R5, #bc      (5)
sub R5, R5, R1    (3)
```

t_B:
```
load R5, #bc      (5)
add R5, R5, R1    (8)
store R5, #bc
```

bc = 8

t_A:
```
store R5, #bc.
```

[bc = 3]

[Няколко процесора]

t_A и t_B се изпълняват паралелно.

t_A [CORE0]
```
load R5, #bc.     (5)
sub R5, R5, R1    (3)
store R5, #bc
```
bc := 3

t_B [CORE1]
```
load R5, #bc      (5)
add R5, R5, R1    (8)
store R5, #bc.
```
[bc = 8]

```
bool busy = false;

f() {
    while (busy);
    busy = true;
    < critical section >
    busy = false;
}
```

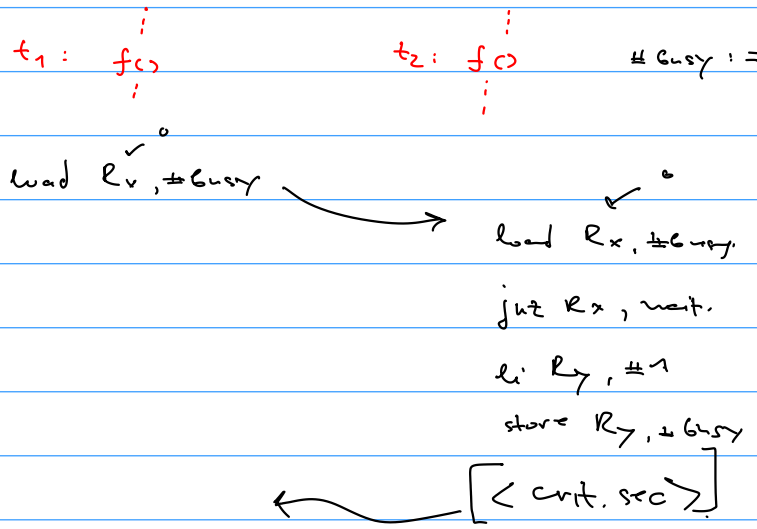
false ↦ 0

true ↦ 1

#busy ← memorija lokacija sec.

```
wait: load Rx, #busy
      jnz Rx, wait
      li Ry, #1
      store Ry, #busy
```

#busy := false.




```
jnz Rx, wait
li Ry, #1
stor Ry, #busy
[< crit. sec >]
```

hardware

turn = 1; {1, 2}

PROCESS 1:

```
while (turn == 2);
< CRIT >
turn = 2;
```

turn≠2 (T)

PROCESS 2:

```
while (turn == 1);
< CRIT >
turn = 1;
```

Петерсон

turn = 1.

flag1 = false, flag2 = false;

P1:

- flag1 = true;
- turn := 2

while (flag2 && turn == 2);

<CRIT>

flag1 = false.

flag2 = true

turn = 1

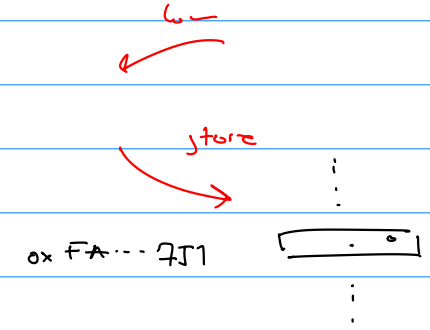
while (flag1 and turn = 1);

<crit>

flag2 = false;

n — Пекарски

Хардвер test_and_set




```
int test_and_set (int *x){
    int ret = *x;
    *x = 1;
    return ret;
}
```

једна инструкција.

[int lock = 0;]

0 — крит. сек. је доступна

1 — крит. с. је заузета.

```
f() {
    while (test_and_set(&lock));
    <CRIT>
    lock = 0;
}
```

lock 1/0

t1: ;

f() — CR

t2:

f();

t3:

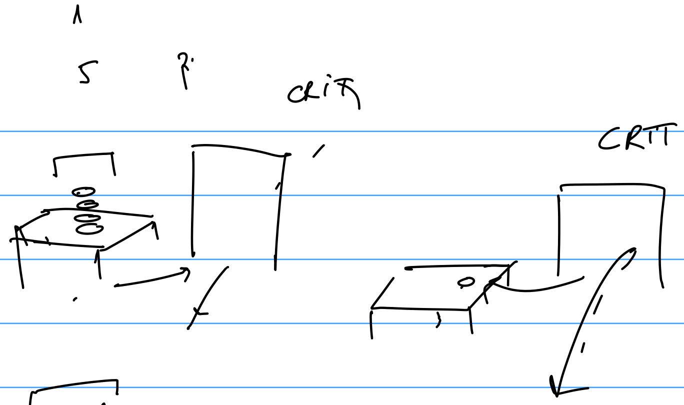
f()

Семафор је интерфејс

int val

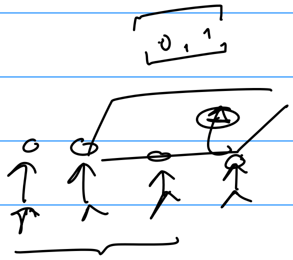
wait

signal.



val -= 1;

val += 1;

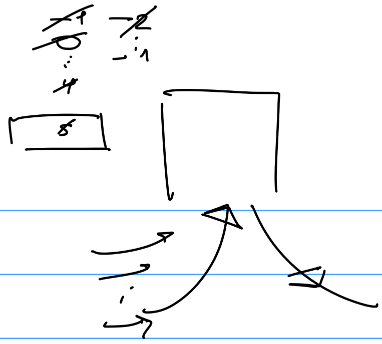


S сем

једном јр.

```
class Semaphore {
    int val;
    Semaphore (int val) : val(val) {}
    Queue blocked;      // ← ред нити које су блокиране на сем.
    void wait() {
        if (val > 0) {
            val -= 1;
        } else {
            blocked.put(Thread::running);          \
            Thread *new = Scheduler::get();         |
            Thread *old = Thread::running;          |  block();
            running = new;                          |
            yield(old->ctx, new->ctx);              /
        }
    }.

    void signal() {
        if (blocked.empty()) {
            val += 1;
        } else {
            Scheduler::put(blocked.get());   // unblock();
        }
    }.
```

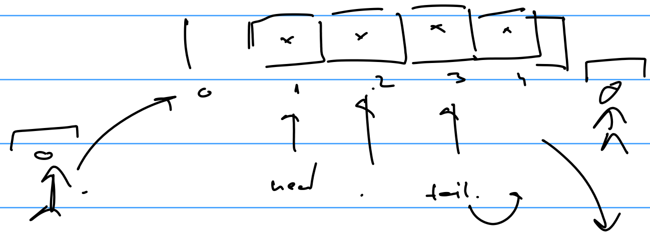


```
class Semaphor {
    int val;

    void wait() {
        val -= 1;
        if (val < 0)    block();
    }

    void signal() {
        val += 1;
        if (val <= 0)   unblock();
    }
```

pthreads




```
class Shelf {
    Box boxes[SIZE];
    int head = 0, tail = 0.
    void put(Box box) {
        tail = (tail + 1) % SIZE;
        boxes[tail] = box;
    }
    Box take() {

    }
}
```

чака да [illegible] има св. място

чака да [illegible] нещо

```
Semaphore boxes(0);
Semaphore slots(SIZE);
Semaphore mutex(1);

void put(Box box) {
    slots.wait();
    mutex.wait();
    tail = (tail+1) % SIZE;
    boxes[tail] = box;
    mutex.signal();
    boxes.signal();
}

Box take() {
    boxes.wait();
    mutex.wait();
    Box box = boxes[head];
    head = (head+1) % SIZE
    mutex.signal();
    slots.signal();
    return box;
}
```

---

2. (10 poena)

Data je sledeća implementacija operacije `Semaphore::lock()` koja obezbeđuje zaključavanje semafora (međusobno isključenje operacija na semaforu) u kodu školskog jezgra za višeprocesorski sistem. Operacija `swap()` implementirana je pomoću odgovarajuće atomične instrukcije procesora. Objasniti zašto ova implementacija operacije `lock()` nije dobra, a onda je popraviti.

```
extern void swap (int*, int*);

void Semaphore::lock() {
  int zero = 0;
  mask_interrupts();
  while (!this->lck) {};
  swap(&zero,&(this->lck));
}
```

lck < 0 zaključano
1 otključano

lck je 0

P1 P2

```
lock() {
    int zero = 0;
    while (zero == 0) {
        swap(&zero, &(this->lck));
    }
}
```

1. (10 poena)

Neki multiprocesorski operativni sistem sa preotimanjem (*preemptive*) dozvoljava preuzimanje tokom izvršavanja koda kernela, osim u kritičnim sekcijama, kada se pristupa deljenim strukturama podataka jezgra kojima mogu da pristupaju različiti procesori. Svaku takvu deljenu strukturu „štiti“ jedna celobrojna promenljiva koja služi za međusobno isključenje pristupa sa različitih procesora pomoću tehnike *spin lock*, dok se zabrana preuzimanja na datom procesoru obezbeđuje maskiranjem prekida. Međusobno isključenje pristupa kritičnoj sekciji, odnosno deljenoj strukturi, obavlja se pozivima operacija čiji je argument pokazivač na celobrojnu promenljivu koja „štiti“ datu deljenu strukturu:

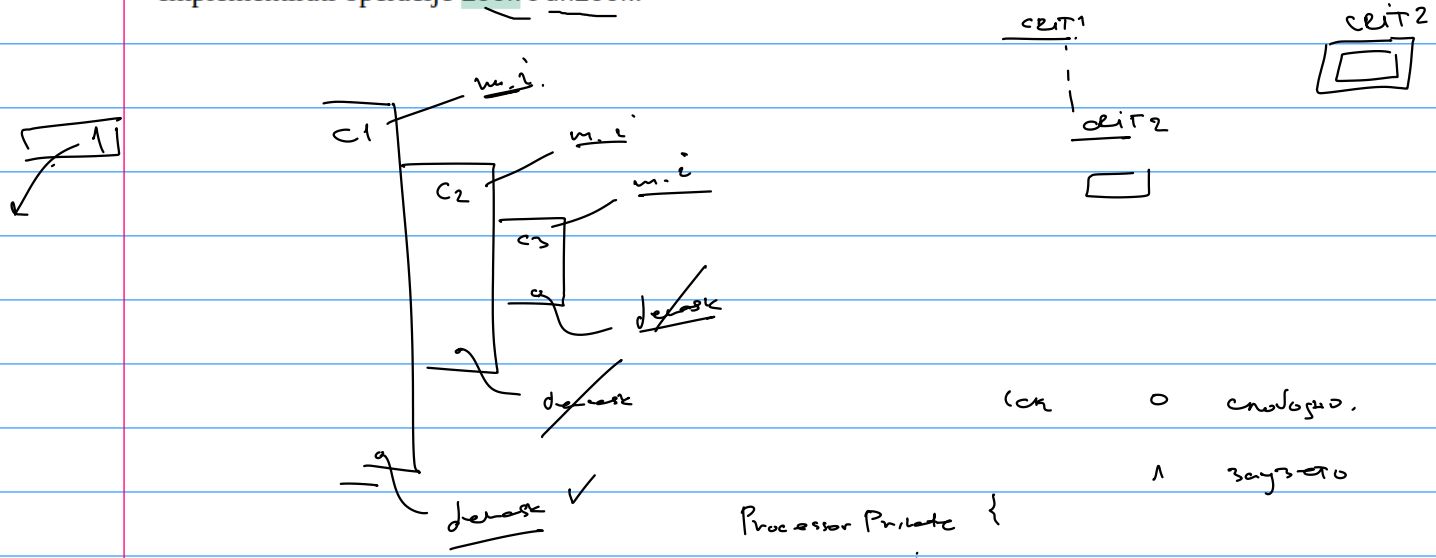
```
void lock (short* lck);
void unlock (short* lck);
```

na ulazu, odnosno izlazu iz kritične sekcije. Pošto postoji potreba da kernel kod istovremeno pristupa različitim deljenim strukturama, kritične sekcije se mogu ugnježdavati. Zato demaskiranje prekida treba uraditi tek kada se izađe iz krajnje spoljašnje kritične sekcije (a ne pri svakom pozivu `unlock`).

Podaci koji su svojstveni svakom procesoru (svaki procesor ima svoju instancu ovih podataka) grupisani su u strukturu tipa `ProcessorPrivate`. (Ovu strukturu možete proširivati po potrebi.) Na raspolaganju su i sledeće funkcije ili makroi:

- `disable_interrupts()`: onemogućava spoljašnje prekide na ovom procesoru;
- `enable_interrupts()`: dozvoljava spoljašnje prekide na ovom procesoru;
- `test_and_set(short* lck)`: „obavija“ instrukciju procesora tipa *test-and-set*;
- `this_processor()`: vraća celobrojni identifikator tekućeg procesora (na kome se kod izvršava);
- `ProcessorPrivate processor_private[NumOfProcessors]`: niz struktura sa „privatnim“ podacima svakog procesora.

Implementirati operacije `lock` i `unlock`.




```
ProcessorPrivate {
    int lock_depth = 0;
}

void lock (short *lck) {
    disable_intr();
    while (test_and_set(lck));
    processor_private[this_processor()].lock_depth += 1
}

void unlock (short *lck) {
    *lck = 1;
    processor_private[this_processor()].lock_depth -= 1;
    if (processor_private[this_processor()].lock_depth == 0) {
        enable_intr();
    }
}
```

2. (10 poena)

Izmeniti datu implementaciju operacije *wait* na semaforu u školskom jezru tako da, pre nego što odmah blokira pozivajuću nit ukoliko je semafor zatvoren, najpre pokuša da uposleno sačeka, ali ograničeno, ponavljajući petlju čekanja najviše `SemWaitLimit` puta. Ostatak klase `Semaphore` se ne menja.

```
void Semaphore::wait () {
  lock(lck);
  if (--val<0)
    block();
  unlock(lck);
}
```

Rešenje:

```
void wait() {
    for(int i=0; i<SemWaitLimit && val<=0; i++);
    lock(lck);
    val -= 1;
    if (val<0) block();
    unlock(lck);
}
```

val = 0 +n

pth---

2. (10 poena)

U nekom višeprocesorskom sistemu ne postoji stanje suspendovanih (blokiranih) niti, već su sve aktivne niti uvek spremne, dok se operacije čekanja na semaforu i drugim sinhronizacionim primitivama realizuju uposlenim čekanjem. Jezgro sistema povremeno (na prekid od tajmera) jednostavno preotima neki procesor od tekuće niti i predaje ga nekoj drugoj aktivnoj niti. U sistemu su implementirane operacije

```
void lock (int lck);
void unlock (int lck);
```

koje realizuju međusobno isključenje nad deljenom strukturom podataka zaštićenom celobrojnim ključem `lck` maskiranjem prekida i mehanizmom *spin-lock* za višeprocesorski pristup.

Realizovati klasu `Semaphore` koja apstrahuje standardni brojački semafor, sa uposlenim čekanjem.

Rešenje:

0.

```
class Semaphor {
    int val;
    Semaphore(int val) : val(val) {}
    int lck = 0;
    void signal() {
        lock((lck));
        val += 1;
        unlock((lck));
    }
```

void wait() {

Cool done = false;

while (! done) {

lock (lck);

if (val > 0) {

val -= 1;

done = true;

}

unlock (lck);

}

}

```
void Semaphore::wait () {
  int done = 0;
  while (!done) {
    lock(lck);
    if (v>0) v--, done=1;
    unlock(lck);
  }
}
```

2. (10 poena)

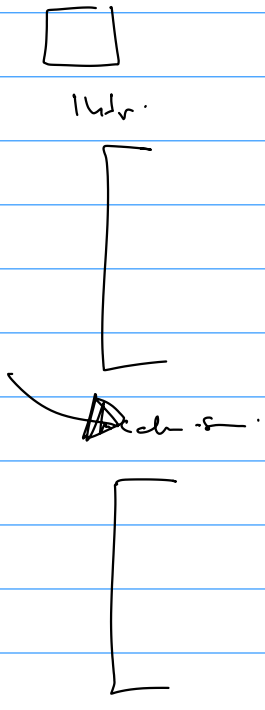
U nekom operativnom sistemu svi sistemski pozivi izvršavaju se kao softverski prekid koji skače na prekidnu rutinu označenu kao `sys_call`, dok se sama identifikacija sistemskog poziva i njegovi parametri prenose kroz registre procesora. Jezgro tog operativnog sistema je višenitno – poseduje više internih kernel niti koje obavljaju različite poslove: izvršavaju uporedne I/O operacije, vrše druge interne poslove jezgra itd. Prilikom obrade sistemskog poziva `sys_call`, prema tome, prekidna rutina samo oduzima procesor tekućem korisničkom procesu uz čuvanje njegovog konteksta i dodeljuje procesor tekućoj kernel niti.

Jedna od tih niti zadužena je za samu obradu zahteva koju je korisnički proces postavio u sistemskom pozivu. Ona preuzima parametre poziva iz sačuvanih registara i, na osnovu vrste sistemskog poziva, poziva odgovarajuću proceduru kernela za taj sistemski poziv.

Ovde se posmatraju sistemski pozivi za operacije na binarnim semaforima (događajima). Unutar kernela, binarni semafori su realizovani klasom `Event`, poput klase `Semaphore` date u školskom jezgru. Realizovati operacije `wait` i `signal` ove klase, koje poziva gore pomenuta kernel nit kada obrađuje te sistemske pozive. Na raspolaganju su operacije `lock()` i `unlock()` koje obezbeđuju međusobno isključenje unutar jezgra, kao i klasa `Scheduler` koja implementira raspoređivanje, kao u školskom jezgru. Na korisnički proces koji je izvršio sistemski poziv ukazuje pokazivač `runningUserThread` unutar kernela.

Rešenje:

val = 1, val > 1



```
class Event {
    int val = 1;
    Queue blocked;
    void wait() {
        lock();
        if (val == 0) {
            blocked.put(runningUserThd);
            runningUserThd = Scheduler::get();
        } else {
            val -= 1;
        }
        unlock();
    }
    void signal() {
        lock();
        if (val == 0) val = 1;
        unlock();
    }
```

1. (10 poena)

Modifikovati operaciju `wait` klase `Semaphore` u školskom jezgru tako da ima izmenjenu deklaraciju datu dole sa sledećim dodatnim mogućnostima i izmenjenim ponašanjem:

- ukoliko je vrednost argumenta `toBlock` različita od 0, operacija se ponaša na standardan način i vraća 1 ako se pozivajuća nit blokirala (suspendovala), a 0 ako nije;
- ukoliko je vrednost argumenta `toBlock` jednaka 0, ako je vrednost semafora nula (ili manja od 0), pozivajuća nit se neće blokirati, vrednost semafora se neće promeniti, a operacija će odmah vratiti -1; inače, ukoliko je vrednost semafora veća od nule, operacija se ponaša na standardan način (i vraća 0 pošto se nit nije blokirala).

```
int Semaphore::wait (int toBlock);
```

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

Semaphore :: wait ( int toBlock ) {
        if (toBlock) {
                lock();
                val -= 1;
                if( val < 0) {
                        Thread* old = Thread::running;
                        blocked.put(old);
                        Thread* new = Scheduler::get();
                        Thread::running = new;
                        unlock();
                        yield();
                        return 1;
                } else {
                        unlock();
                        return 0;
                }
        } else {
                lock();
                if( val > 0 ) {
                        val -= 1;
```

```
            unlock();
            return 0;
        } else {
            unlock();
            return -1;
        }
    }
}
```