Konkurentno i distribuirano programiranje - skripta za kolokvijumski deo

Ivan Janevski

Sadržaj

1 Spinlock algoritmi

1.1 Uvod

Šta su niti?
Niti (eng. Threads) su mehanizam koji omogućava konkurentno ili paralelno izvršavanje nekih delova koda programa.

Konkurentno izvršavanje podrazumeva jednu izvršnu jedinicu (jezgro ili procesor) koju niti vremenski dele (t.j. izvšavaju se po principu "čas jedna, čas druga, ...").

Paralelno izvršavanje podrazumeva više izvršnih jedinica (jezgara ili procesora) gde se niti fizički izvršavaju u isto vreme. Na kom jezgru ili procesoru će se odredjena nit izvršavati i koliko jedinica vremena odlučuje `scheduler` operativnog sistema (preemptive multitasking). Ukoliko ne postoji `scheduler` već niti same odlučuju koja će se izvršavati sledeće (cooperative multitasking) takve niti se nazivaju **korutinama** (eng. Coroutines).

Primer kreiranja dve POSIX niti gde jedna nit ispisuje crveno slovo “A” dok druga ispisuje zeleno slovo “B”:

```
// Compile and run: gcc -pthread main.c && ./a.out
#include <stdio.h>
#include <pthread.h>
#define RED "\x1b[31m"
#define GREEN "\x1b[32m"
#define NOCOLOR "\x1b[0m"

void *a(void *data) {
    for ( ; ; ) {
        printf(RED "A" NOCOLOR);
    }
}
void *b(void *data) {
    for ( ; ; ) {
        printf(GREEN "B" NOCOLOR);
    }
}
int main() {
    pthread_t ta, tb;
    pthread_create(&ta, NULL, a, NULL);
    pthread_create(&tb, NULL, b, NULL);
    pthread_join(ta, NULL);
    pthread_join(tb, NULL);
    return 0;
}
```

Šta je atomičnost?
Izraz (t.j. niz operacija) je atomičan ukoliko se izvršavanje operacija ne može prekinuti na pola puta t.j. ili se sve operacije izvrše odjednom, ili se ne izvrši ni jedna.

Šta je kritična referenca?
U nekom izrazu koji jedna nit izvršava, **kritična referenca** je referenca na promenljivu u izrazu čija se vrednost menja od strane neke druge niti.

Šta je At-Most-Once Property (AMOP)?

Iskaz dodele vrednosti $x = e$ zadovoljava **At-Most-Once Property** ukoliko je jedna od sledeće dve stavke tačna:

1. Izraz e sadrži **najviše jednu** kritičnu referencu, i x se ne čita od strane drugih niti.
2. Izraz e ne sadrži **ni jednu** kritičnu referencu, a x se može čitati od strane drugih niti.

Ukoliko dodela vrednosti zadovoljava AMOP, u pogledu svih niti će izgledati kao da se dodela izvršava atomično.

U sledećem primeru, obe dodele vrednosti $x = e_1$ i $y = e_2$ **zadovoljavaju** AMOP jer e_1 i e_2 ne sadrže ni jednu kritičnu referencu (zadovoljavaju stavku 2):

```
int x = 0, y = 0;
void a() {
    x = x + 1;
}
void b() {
    y = y + 1;
}
```

U sledećem primeru, obe dodele vrednosti $x = e_1$ i $y = e_2$ **zadovoljavaju** AMOP. U $x = e_1$, e_1 sadrži kritičnu referencu na y, ali se x ne čita od strane drugih niti (zadovoljava stavku 1). U $x = e_2$, e_2 ne sadrži ni jednu kritičnu referencu (zadovoljava stavku 2).

```
int x = 0, y = 0;
void a() {
    x = y + 1;
}
void b() {
    y = y + 1;
}
```

U sledećem primeru, $x = e_1$ i $y = e_2$ **ne zadovoljavaju** AMOP. U $x = e_1$, e_1 sadrži kritičnu referencu na y, ali x se čita od strane niti `b()` (ne zadovoljava stavku 2.). U $y = e_2$, e_2 sadrži kritičnu referencu na x, ali y se čita od strane niti `a()` (ne zadovoljava stavku 2.).

```
int x = 0, y = 0;
void a() {
    x = y + 1;
}
void b() {
    y = x + 1;
}
```

Šta je kritična sekcija?

Kritična sekcija je deo koda koji samo jedna nit sme da izvršava u nekom trenutku i može se implementirati pomoću `mutual exclusion lock`-a (`mutex`-a).

Šta je lock?

Brava (eng. Lock) je mehanizam koji omogućava nekoj niti da ograniči pristup nekom delu koda ili resursu ostalim nitima. Na primer, nit može zaključati fajl sa ekskluzivnim pravom pristupa za upis u fajl (`exclusive lock`), ili sa deljenim pravom pristupa za čitanje iz fajla (`shared lock`).

Šta je spinlock?

Spinlock je brava implementirana pomoću uposlenog čekanja t.j. petlje koje ne radi ništa.

```
while (lock)
    skip();
```

Šta je deadlock (livelock)?

Dve ili više niti se mogu naći u deadlock-u ukoliko blokiraju jedna druge tako da ni jedna ne može da nastavi sa izvršavanjem. Livelock je sličan deadlock-u s tim što niti stalno menjaju ustupaju prednost nekoj drugoj niti i na taj način takođe ni jedna nit ne nastavlja sa izvršavanjem.

Šta je izgladnjivanje (starvation)?

Izgladnjivanje niti se javlja kada nit predugo čeka na nekom uslovu. Na primer, kod Readers/Writers problema se može javiti izgladnjivanje pisaca ukoliko čitaoci stalno nadolaze.

Šta je `volatile`?
`volatile` je ključna reč u jezicima C i C++ koja nagoveštava prevodiocu da ne pravi optimizacije sa objektnom naznačenom sa `volatile`. Na primer, ukoliko želimo da radimo uposleno čekanje tako što ćemo imati jednu `for` petlju koja ne radi ništa:

```
// gcc -masm=intel -S -O2 main.c && cat ./main.s
int main() {
    /* volatile */ int i;
    for (i = 0; i < 100000; i++);
    return 0;
}
```

Prevodilac će optimizovati kod tako što uopšte neće generisati kod za petlju:

```
        .globl  main
        .type   main, @function
main:
        xor     eax, eax
        ret
```

Dok ukoliko otkomentarišemo `volatile`, kod za petlju se generiše:

```
        .globl  main
        .type   main, @function
main:
        mov     DWORD PTR -4[rsp], 0
        mov     eax, DWORD PTR -4[rsp]
        cmp     eax, 99999
        jg      .L2
        .p2align 4,,10
        .p2align 3
.L3:
        mov     eax, DWORD PTR -4[rsp]
        add     eax, 1
        mov     DWORD PTR -4[rsp], eax
        mov     eax, DWORD PTR -4[rsp]
        cmp     eax, 99999
        jle     .L3
.L2:
        xor     eax, eax
        ret
```

Pravilo je da sve deljene promenljive treba označiti sa modifikatorom `volatile`.

Napomena: U jeziku C++ **nije dovoljno** koristiti `volatile` za korektnu sinhronizaciju pomoću spinlock-a već za implementaciju neophodno koristiti konstrukte iz zaglavlja `<atomic>`, tako da sve kodove navedene u nastavku treba razumeti samo kao pseudo kod.

Šta je `skip()`?
Pozviom `skip()`, nit se odriče procesorskog vremena. Radi optimizacije, `skip()` je potrebno pozvati prilikom uposlenog čekanja ukoliko uslov nije ispunjen.

Na primer, ukoliko je `scheduler` dodelio procesor nekoj niti u vremenu od 150 ms, a ona ustanovi u roku od 7 μs da uslov za nastavak nije ispunjen, nema potrebe da se nit vrti u petlji narednih 149,993 ms ukoliko postoj neka druga nit koja bi se mogla izvršavati.

`skip()` je moguće implementirati kao `sched_yield()` ili `usleep(0)`.

1.2 Test-and-set algoritam

Kod **Test-and-set** algoritma postoji globalna brava `lock` koja je inicijalno kreirana kao otključana (`false`). Nit koja želi da udje u kritičnu sekciju atomično zaključava globalnu bravu i dohvata staru vrednost brave. Atomično dohvatanje stare vrednosti globlane i postavljanje na vrednost "zaključano" (`true`) se vrši pozivom specijalne instrukcije `test_and_set(v)`. Ukoliko je stara vrednost brave bila "otključano", uposleno čekanje se prekida ulazi se u kritičnu sekciju, s tim što je vrednost brave sada postavljena na "zaključano". Ukoliko je stara vrednost bila "zaključano", u bravu se opet upisuje vrednost "zaključano" i uposleno čekanje se nastavlja. Stalni ponovni upis vrednosti `true` u promenljivu `lock` usporava rad sistema, jer vrši čestu invalidaciju keširane vrednosti `lock` i često zaključavanje `cache` linije za upis. Ovaj problem rešava sledeći algoritam, **Test-and-test-and-set**. Kada nit u kritičnoj sekciji završi sa kritičnom sekcijom, vrednost globalne brave postavlja na "otključano".

```
volatile bool lock = false;

void thread(int id)
{
    while (true) {
        /* LOCK */
        while (test_and_set(&lock))
            skip();
        /* CRITICAL SECTION */
        work();
        /* UNLOCK */
        lock = false;
        /* NON-CRITICAL SECTION */
        usleep(50);
    }
}
```

1.3 Test-and-Test-and-Set algoritam

Kod **Test-and-test-and-set** algoritma vrednost globalne brave se najpre proverava iz keš memorije. Ukoliko je vrednost brave "zaključano", nit vrši uposleno čekanje u unutrašnjoj niti gde se vrednost globalne brave `lock` dohvata iz keš memorije. Ukoliko nit koja je u kritičnoj sekciji napusti kritičnu sekciju pozivom `lock = false;`, sve niti koje su bile na uposlenom čekanju u unutrašnjoj petlji će probati da zaključaju bravu i udju u kritičnu sekciju sa `while (test_and_set(&lock))`. Jedna od niti će uspeti da udje u kritičnu sekciju, dok će se ostale niti vratiti na uposleno čekanje u unutrašnjoj petlji.

```
volatile bool lock = false;

void thread(int id)
{
    while (true) {
        /* LOCK */
        do {
            while (lock)
                skip();
        } while (test_and_set(&lock));
        /* CRITICAL SECTION */
        work();
        /* UNLOCK */
        lock = false;
        /* NON-CRITICAL SECTION */
        usleep(50);
    }
}
```

1.4 Pitersonov (Tie-Breaker) algoritam za N procesa

Kod **Pitersonovog (Tie-Breaker) algoritma za N procesa**, nit koja želi da udje u kritičnu sekciju najpre mora da prodje kroz $N-1$ nivoa. Nit ostaje blokirana u nivou ukoliko postoji neka nit koja je u jednakom ili višem nivou (`inNivo[pId] >= inNivo[id]`) i ukoliko je data nit poslednja koja je ušla u dati nivo (`lastId[nivo] == id`). Niti se polako "filtriraju" tako što u svakom nivou jedna nit koja se utrkuje mora da ostane "zarobljena" u tom nivou. Tako se u prvom nivou se može naći najviše N niti, u drugom nivou može se naći najviše $N-1$ niti, u trećem nivou se može naći najviše $N-2$ niti, itd. U poslednjem nivou se može naći samo jedna nit i ona ulazi u kritičnu sekciju. Kada nit završi sa kritičnom sekcijom, nivo date niti se resetuje na 0.

```
volatile int inNivo[N] = { 0 }; // inNivo[processId] = nivo
volatile int lastId[N] = { 0 }; // lastId[nivo] = processId

void thread(int id)
{
    while (true) {
        /* LOCK */
        for (int nivo = 0; nivo < N - 1; nivo++) {
            inNivo[id] = nivo;
            lastId[nivo] = id;
            for (int pId = 0; pId < N; pId++) {
                if (pId == id)
                    continue;
                while (inNivo[pId] >= inNivo[id] && lastId[nivo] == id)
                    skip();
            }
        }
        /* CRITICAL SECTION */
        work();
        /* UNLOCK */
        inNivo[id] = 0;
        /* NON-CRITICAL SECTION */
        usleep(50);
    }
}
```

1.5 Ticket algoritam

Kod **Ticket** algoritma, nit koja želi da udje u kritičnu sekciju najpre atomično dohvata globalni broj `ticket` u lokalnu promenljivu `number` i inkrementira globalni `ticket` za vrednost 1. Globalni `ticket` se atomično dohvata i inkrementira pozivom specijalne instrukcije `fetch_and_add(x, v)`. Nit potom čeka dok globalna vrednost `next` ("sledeći") ne postane baš vrednost prethodno dohvatanog broja `number`. Kada nit završi sa kritičnom sekcijom, sledećoj niti se dopušta ulazak u kritičnu sekciju sa `next += 1` (nije neophodno da se vrednost `next` inkrementira specijalnom instrukcijom `fetch_and_add(x, v)` jer se kritična sekcija `/*CRITICAL SECTION*/` nastavlja i na sekciju `/*UNLOCK*/`). Ticket algoritam garantuje "fer" (FIFO) ulazak u kritičnu sekciju.

```
volatile int ticket = 0;
volatile int next = 0;

void thread(int id)
{
    while (true) {
        /* LOCK */
        int number = fetch_and_add(&ticket, 1);
        while (next != number)
            skip();
        /* CRITICAL SECTION */
        work();
        /* UNLOCK */
        next += 1;
        /* NON-CRITICAL SECTION */
        usleep(50);
    }
}
```

1.6 Andersonov (ABQL) algoritam

Kod **Andersonovog ABQL (Array-Based Queuing Lock)** algoritma, svaka nit čeka na svoj slot u nekom nizu čija je veličina jednaka broju niti. Prvi slot je označen kao otključan, tako da prva nit koja naiđe može da prodje u kritičnu sekciju, dok su ostali slotovi zaključani. Nit atomično dohvata `slot` u svoj privatni `mySlot` i inkrementira globalni `slot`. Globalni `slot` se atomično dohvata i inkrementira pozivom specijalne instrukcije `fetch_and_add(x, v)`. Kada nit završi sa ktritičnom sekcijom, svoj slot resetuje, a nit zablokiranu na sledećem slotu propušta. Kao i Ticket algoritam, Andersonov algoritam garantuje "fer" (FIFO) redosled ulaska u kritičnu sekciju.

```
volatile bool flag[N] = { true };
volatile int slot = 0;

void thread(int id)
{
    while (true) {
        /* LOCK */
        int mySlot = fetch_and_add(&slot, 1) % N;
        while (flag[mySlot] == false)
            skip();
        /* CRITICAL SECTION */
        work();
        /* UNLOCK  */
        flag[mySlot] = false;
        flag[(mySlot + 1) % N] = true;
        /* NON-CRITICAL SECTION */
        usleep(50);
    }
}
```

1.7 Bakery algoritam

Kod **Lamport's bakery** algoritma, svaka nit ima svoj broj `turn[id]` koji se postavlja na vrednost veću od `turn`-a svih ostalih niti. Nit najpre izražava želju da udje u kritičnu sekciju postavljanjem vrednosti `turn[id] = 1` (vrednost `turn[id] = 0` znači da nit uopšte ne želi još da udje u kritičnu sekciju), a potom gleda `turn`-ove ostalih niti, traži najveći `turn[j]` u nizu, i ažurira svoj `turn[id]` na vrednost za 1 veću od nadjenog maksimuma. Traženje maksimuma **ne mora** da se obavlja atomično, tako da može se dogoditi da više niti imaju istu vrednost svojih `turn`-ova (`turn[j] == turn[id]`), i u tom slučaju prednost ima nit sa manjim `id`-em. Nit sa najmanjim ne-nultim `turn`-om ulazi u kritičnu sekciju. Kada nit završi sa kritičnom sekcijom nit svoj `turn[id]` resetuje na vrednost 0 što označava nitima koje čekaju da udju u kritičnu sekciju da je kritična sekcija sada slobodna.

```
volatile int turn[N] = { 0 };

void thread(int id)
{
    while (true) {
        /* LOCK */
        turn[id] = 1;
        int max = 0;
        for (int j = 0; j < N; j++) {
            if (j == id)
                continue;
            if (turn[j] > max)
                max = turn[j];
        }
        turn[id] = max + 1;
        for (int j = 0; j < N; j++) {
            if (j == id)
                continue;
            while (turn[j] != 0 && (turn[id] > turn[j] || turn[j] == turn[id] && id > j))
                skip();
        }
        /* CRITICAL SECTION */
        work();
        /* UNLOCK */
        turn[id] = 0;
        /* NON-CRITICAL SECTION */
        usleep(50);
    }
}
```

1.8 CLH algoritam

Kod **CLH (Craig, Landin, and Hagerste)** algoritma postoji globalna brava postoji globalna brava `clh_lock` koja je inicijalno kreirana kao otključana. Nit koja želi da udje u kritičnu sekciju kreira novu **zaključanu** bravu i atomično je zamenjuje sa globalnom bravom `clh_lock` dohvatajući staru vrednost globalne brave u `old_lock`. Stara vrednost globalne brave se dohvata i postavlja na novu vrednost pozivom specijalne instrukcije `get_and_set(x, v)`. Nit potom čeka dok stara brava (koju je neko drugi naparvio i treba da otključa) ne postane otključana, ulazi u kritičnu sekciju, i potom otključava bravu `new_lock`, propuštajući nit čija je to brava `old_lock`. Prva nit koja naidje će ući u kritičnu sekciju dok se ostale niti blokiraju na poslednju bravu koja je zamenjena. Ovim se niti virtuelno ulančavaju i propuštaju po onom redosledu kom su pristigle. Kao i Ticket i Andersonov algoritam, CLH algoritam garantuje "fer" (FIFO) redosled ulaska u kritičnu sekciju.

```
typedef bool Lock;

volatile Lock *clh_lock = new Lock { false };

void thread(int id)
{
    while (true) {
        /* LOCK */
        volatile Lock *new_lock = new Lock { true };
        volatile Lock *old_lock = get_and_set(&clh_lock, new_lock);
        while (*old_lock == true)
            skip();
        delete old_lock;
        /* CRITICAL SECTION */
        work();
        /* UNLOCK */
        *new_lock = false;
        /* NON-CRITICAL SECTION */
        usleep(50);
    }
}
```

2 Semafori

2.1 Uvod

Šta je semafor?
Semafori (eng. Semaphore) su mehanizam koji omogućava sinhronizaciju ili međusobno isključivanje izmedju dve ili više niti. Semafor ima svoj internu vrednost (`int value`), red blokiranih niti na semaforu (`queue<Thread*>`), i podržava sledeće dve operacije:

- `void wait(s)` - Ukoliko je vrednost semafora nula, nit se dodaje u red blokiranih na semaforu. Ukoliko je vrednost semafora veća od nula, vrednost semafora se dekrementira za 1.
- `void signal(s)` - Ukoliko postoje zablokirane niti u redu blokiranih, jedna nit iz reda blokiranih se deblokira. Ukoliko je red zablokiranih prazan, vrednost semafora se inkrementira za 1.

Vrednost ovako definisanog semafora ne može da ode ispod nule. Takođe, u redu zablokiranih ne mogu da postoje niti ukoliko je vrednost semafora veća od nule.

Pošten semafor je onaj koji pozivom operacije `signal(s)` deblokira niti po FIFO redosledu (nit koja je ranije pozvala `wait(s)` će se ranije deblokirati).
Nepošten semafor je onaj koji pozivom operacije `signal(s)` deblokira niti po nasumičnom redosledu.

U ovoj skripti (i na KDP-u) se podrazumevaju **nepošteni semafori** - tako da je za implementaciju FIFO redosleda neophodno npr. koristiti tehniku privatnih semafora sa `queue<int>`.

Primer korišćenja POSIX semafora:

```
#include <semaphore.h>
sem_t mutex;
sem_init(&mutex, 0, 1);
...
sem_wait(&mutex) // wait
/* CRITICAL SECTION */
sem_post(&mutex); // sginal
...
sem_destroy(&mutex);
```

2.2 Simple synchronization

Zadatak Proces je pokrenuo dve niti sa beskonačnim petljama, gde prva nit ispisuje na konzoulu znak “a” dok druga nit ispisuje znak “b” (aaaaaaaaabbbbbbbbbbbaaaaaaaabbbb...). Koristeći semafore, potrebno je sinhronizovati niti tako da se na konzoli naizmenično ispisuju znakovi “a” i “b”. (abababababababab...).

Šta je raspodeljeni binarni semafor?
Raspodeljeni binarni semafor je binarni semafor (semafor čija se vrednost kreće izmedju 0 ili 1) čiji su pozivi operacije `wait(s)` raspodeljeni unutar koda jedne niti, a pozivi operacije `signal(s)` raspodeljeni unutar koda druge niti. Ovime jedna nit može da kontroliše tok druge niti. Naime, nit koja poziva `signal(s)` kontroliše izvršavanje niti koja poziva `wait(s)`.

Rešenje:

```
#include <thread.h>
#include <sem.h>

Sem sa = 1, sb = 0; // dva raspodeljena binarna semafora

void a()
{
    while (true) {
        wait(sa);
        print("a");
        signal(sb);
    }
}

void b()
{
    while (true) {
        wait(sb);
        print("b");
        signal(sa);
    }
}


int main() {
    Thread ta = createThread(a);
    Thread tb = createThread(b);
    join(ta);
    join(tb);
    return 0;
}
```

2.3 Mutual exclusion

Zadatak Proces je pokrenuo dve niti, gde prva nit popunjava deljeni niz znakova znakom ‘a’, dok druga nit popunjava niz znakova znakom ‘b’. Ove dve niti remete rad jedna drugoj, time što se utrkuju (race condition) i “gaze” podatke druge niti svojim podacima. Koristeći semafore, potrebno je zaštiti niz znakova tako da jedna nit ne može da započne upis dok druga nit ne potpuno završi.

Šta je mutex?
Mutex je resurs koji samo jedna nit može da “drži” u nekom trenutku. Kada neka nit “uzme” mutex (“zaključa” ga) sve ostale niti koje pokušaju isto će se blokirati dok nit koja drži mutex ne vrati (“otključa”) isti. Mutex služi za implementaciju **kritične sekcije** u kodu. Kritična sekcija je deo koda koji isključivo jedna nit može da izvršava u nekom trenutku. Primer korišćenja mutex-a pomoću POSIX niti:

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex);
...
pthread_mutex_lock(&mutex);
// CRITICAL SECTION
pthread_mutex_unlock(&mutex);
...
pthread_mutex_destroy(&mutex);
```

Mutex se lako implementira pomoću binarnog semafora (semafori nisu jedini način da se napravi mutex!). Vrednost semafora u svrsi mutex-a se inicijalizira na vrednost `s=1`, a potom `wait(s)` označava “zauzeće” mutex-a, dok `signal(s)` označava “oslobađanje” istog. npr. pomoću POSIX semafora:

```
sem_t mutex;
sem_init(&mutex, 0, 1); // init semaphore with value 1
...
sem_wait(&mutex); // lock
// CRITICAL SECTION
sem_post(&mutex); // unlock
...
sem_destroy(&mutex);
```

Rešenje

```
#include <thread.h>
#include <sem.h>
Sem mutex = 1;
char buffer[BUFSIZ];

void a() {
    while (true) {
        wait(mutex);
        memset(buffer, 'a', BUFSIZ);
        signal(mutex);
        usleep(50);
    }
}

void b() {
    while (true) {
        wait(mutex);
        memset(buffer, 'b', BUFSIZ)
        signal(mutex);
        usleep(50);
    }
}

int main() {
    Thread ta = createThread(a);
    Thread tb = createThread(b);
    join(ta);
    join(tb);
    return 0;
}
```

2.4 Producer and consumer

Zadatak. Proces ima jedan deljeni ograničen kružni bafer veličine B i pokrenuo je dve niti: jednu nit `producer()` koja kreira podatke i smešta ih u bafer, i jednu nit `consumer()` koja dohvata podatke iz bafera i koristi ih. Ukoliko `producer` radi brže od `consumer`-a, nastupiće prekoračenje bafera (buffer overflow), dok ukoliko `consumer` radi brže od `producer`-a, `consumer` će početi da dohvata nevalidne ili nepostojaće podatke. Koristeći semafore, potrebno je obezbediti da se ove neželjene situacije ne dešavaju.

Napomena za rešenje:
Mutex uvek treba da stoji najuže moguće uz podatke koje štiti, a sinhronizacija uvek treba da bude oko mutex-a. Dakle sledeći kod za `consumer`-a je greška i uzrokuje deadlock:

```
void consumer() {
    wait(mutex);
    wait(empty);
    ...
    wait(full);
    signal(mutex);
}
```

Rešenje:

```
#include <thread.h>
#include <sem.h>
#define B 10
int buffer[B];
int head = 0;
int tail = 0;
int count = 0;
Sem mutex = 1;
Sem full = B;
Sem empty = 0;

void producer() {
    while (true) {
        // Produce data...
        int data = rand() % 1000;
        // Put data into buffer
        wait(full);
        wait(mutex);
        buffer[head] = data;
        head = (head + 1) % B;
        count += 1;
        signal(mutex);
        signal(empty);
    }
}
void consumer() {
    while (true) {
        // Fetch data from buffer
        wait(empty);
        wait(mutex);
        int data  = buffer[tail];
        tail = (tail + 1) % B;
        count -= 1;
        signal(mutex);
        signal(full);
        // Consume data...
        print("%d ", data);
    }
}

int main() {
    Thread ta = createThread(producer);
    Thread tb = createThread(consumer);
    join(ta);
    join(tb);
    return 0;
}
```

2.5 Barrier synchronization

Zadatak Proces je pokrenuo N niti koje treba da zajednički odrade neki posao `work1()`, i treba predju na posao `work2()` tek kada svih N niti završe posao `work1()`. Slično, potrebno je da svih N završe posao `work2()` pre nego što se predje nazad na posao `work1()`. Poslovi se moraju obavljati sa ekskluzivnim pravom pristupa. Implementirati ovakvo ponašanje niti pomoću semafora.

Šta je barijera?
Barijera (eng. Barrier) je tačka u kodu gde je potrebno da dođe određen broj niti pre nego što niti zaglavljene na barijeri ne nastave sa svojim izvršavanjem. Ukoliko neka nit dođe do barijere, broj niti na barijeri će se povećati za jedan. Ukiliko je u tom slučaju i dalje nedovoljan broj niti na barijeri, data nit će se blokirati. U suprotnom, data nit će deblokirati sve niti ranije zablokirane na barijeri i nastaviti dalje sa izvšavanjem. Primer korišćenja barijere sa POSIX nitima:

```
pthread_barrier_t bar;
pthread_barrier_init(&bar, NULL, 5);
...
pthread_barrier_wait(&bar);
...
pthread_barrier_destroy(&bar);
```

1. način
Nit koja uđe kroz prva vrata `door1` pozivom `wait(door1)` dobija ekskluzivno pravo pristupa i može da obavi posao `work1()`. Kada nit završi posao `work1()` inkrementira `counter` i, pre nego što dođe do drugih vrata (t.j. `wait(door2)`), proverava koja vrata treba otvoriti sledeće: Ukoliko je `count < N`, pušta se još jedna nit kroz `door1` pozivom `signal(door1)`. Ukoliko je `count == N`, counter se resetuje na 0 kako bi se mogao koristiti u drugoj sekciji, i jedna nit se pušta kroz druga vrata pozivom `signal(door2)` u drugu sekciju, To može biti upravo ta nit koja je pozvala `signal(door2)` i tek treba da dođe do `wait(door2)`, ili neka od $N-1$ niti već zablokirane na `wait(door2)`.

```
#include <thread.h>
#include <sem.h>
#define N 5

Sem door1 = 1, door2 = 0;
int count = 0;

void a(int id)
{
    while (true) {
        wait(door1);
        /* SECTION 1 */
        work1();
        count += 1;
        if (count == N) {
            count = 0;
            signal(door2);
        } else {
            signal(door1);
        }
        wait(door2);
        /* SECTION 2 */
        work2();
        count += 1;
        if (count == N) {
            count = 0;
            signal(door1);
        } else {
            signal(door2);
        }
    }
}

int main() {
    Thread t[N];
    for (int i = 0; i < N; i++)
        t[i] = createThread(a, i);
    for (int i = 0; i < N; i++)
```

```
        join(t[i]);
    return 0;
}
```

2. način.

Možemo napraviti strukturu `struct Barrier` koja implementira barijeru pomoću semafora, i onda instancirati dve barijere: `bar1` i `bar2`. Za razliku od "vrata" u prethodnom primeru (`door1` i `door2`) koja su ujedno nam obezbeđivala, i sinhronizaciju, i medjusobno isključivanje, barijere nam omogućavaju samo sinhronizaciju, a medjusobno isključivanje obezbedjujemo sami pomoću **mutex**-a.

```
#include <thread.h>
#include <sem.h>
#define N 5

struct Barrier {
    void wait() {
        wait(m_mutex);
        m_count += 1;
        if (m_count == N) {
            m_count = 0;
            for (int i = 0; i < N; i++)
                signal(m_barrier);
        }
        signal(m_mutex);
        wait(m_barrier);
    }
private:
    int m_count = 0;
    Sem m_mutex = 1;
    Sem m_barrier = 0;
} bar1, bar2;

Sem mutex = 1;
void a(int id)
{
    while (true) {
        /* SECTION 1 */
        wait(mutex);
        work1();
        signal(mutex);
        bar1.wait();

        /* SECTION 2 */
        wait(mutex);
        work2();
        signal(mutex);
        bar2.wait();
    }
}

int main() {
    Thread t[N];
    for (int i = 0; i < N; i++)
        t[i] = createThread(a, i);
    for (int i = 0; i < N; i++)
        join(t[i]);
    return 0;
}
```

2.6 Shortest job next

Zadatak Proces je pokrenuo N niti koje treba pristupe kritičnoj sekciji po prioritetnom redosledu. Naime, ukoliko postoje više niti koje istovremeno žele da udju u kritičnu sekciju, prva ulazi ona koja ima najmanje posla. Potrebno je pomoću semafora implementirati metodu `sjn_wait(id, prio)` (niža vrednost argumenta `prio` znači viši prioritet) koja se poziva pre ulaska u kritičnu sekciju, i metodu `sjn_signal()` koja se poziva nakon izlaska iz kritične sekcije.

Šta je tehnika privatnih semafora?
Privatni semafor neke niti je semafor inicijalizovan na vrednost 0 koji nit može da koristi da se veštački zablokira i sačeka dok je neka druga nit ne deblokira. Pre blokiranja na privatnom semaforu neophodno je osloboditi `mutex` ukoliko se drži (inače nastaje deadlock!) i neophodno je na neki način zapamtiti `id` niti (ili pokazivač na privatni semafor) kako bi se kasnije taj privatni semafor signalizirao, i time zablokirana nit deblokirala.

Šta je tehnika prosledjivanja štafetne palice (passing the baton)?
To je tehnika sinhronizacije gde se - umesto puštanja mutex-a - signalizira neka blokirana nit (npr. blokirana na privatnom semaforu). Time deblokirana nit "dobija" mutex od niti koja je držala mutex i deblokirana nit može koristiti podatke koje su zaštićene pod mutex-om. Nit potom može da deblokira neku drugu nit i na isti način joj dostavi mutex, a može i jednostavno da oslobodi mutex. Ukoliko ne postoji ni jedna nit kojoj mutex može da se prosledi, neophodno je osloboditi mutex. Mutex predstavlja štafetnu palicu koja se prosleđuje između niti.

1. način - sa prosleđivanjem štafetne palice

Ovo rešenje liči na implementaciju `Signal-and-Wait` monitora.

```
#include <thread.h>
#include <sem.h>
#include <queue>
using namespace std;
#define N 5

typedef pair<int, int> PAIR;

Sem mutex = 1; // mutex za zastitu isWorking i qDelay
Sem privs[N] = { 0, 0, 0, 0, 0 }; // privatni semafori svih 5 niti
priority_queue<PAIR, vector<PAIR>, greater<PAIR>> qDelay;
bool isWorking = false;

void sjn_wait(int id, int prio)
{
    wait(mutex);          // uzimamo mutex da bi zastitili isWorking i qDelay
    if (isWorking) {      // ukoliko je neko u krit. sekciji blokiracemo se
        PAIR p(prio, id); // snimamo ID i prioritet.
        qDelay.push(p);   // stavljamo se u red
        signal(mutex);    // pustamo mutex
        wait(privs[id]);  // blokiramo se na privatnom semaforu
        // mutex cemo kasnije dobiti nazad od sjn_signal()
    }
    isWorking = true;  // postavljamo da smo u kriticnoj sekciji.
    signal(mutex);     // pustamo mutex (bilo da smo ga uzeli sami ili ga dobili nazad)
}

void sjn_signal()
{
    wait(mutex);            // uzimamo mutex
    isWorking = false;      // postavljamo da vise nismo u krit. sekciji.
    if (! qDelay.empty()) { // ako ima nekog zablokiranog, deblokiramo ga
        PAIR p = qDelay.top();
        int dPrio = p.first;
        int dId = p.second;
        qDelay.pop();
        // signaliziramo samo privatni semafor a *ne* i mutex
        // jer hocemo da mutex dobije onaj koga signaliziramo
        signal(privs[id]);
    } else {
```

```
        // posto u ovom ovom nema nikog koga treba da budimo
        // nemamo kome da mutex prosledimo, pa ga samo pustamo.
        signal(mutex);
    }
}

void a(int id)
{
    int prio = rand() % 1000; // nasumicni prioritet
    while (true) {
        sjn_wait(id, prio);
        work();
        sjn_signal();
    }
}


int main() {
    Thread ta[N];
    for (int i = 0; i < N; i++)
        ta[i] = createThread(a, i);
    for (int i = 0; i < N; i++)
        join(ta[i]);
    return 0;
}
```

2. način - bez prosleđivanja štafetne palice

Ovo rešenje liči na implementaciju `Signal-and-Continue` monitora.

```
#include <thread.h>
#include <sem.h>
#include <queue>
using namespace std;
#define N 5

typedef pair<int, int> PAIR;

Sem mutex = 1;
Sem privs[N] = { 0, 0, 0, 0, 0 };
priority_queue<PAIR, vector<PAIR>, greater<PAIR>> qDelay;
bool isWorking = false;

void sjn_wait(int id, int prio)
{
    wait(mutex);           // uzimamo mutex da bi zastitili isWorking i qDelay
    if (isWorking) {       // ukoliko je neko u krit. sekciji, blokiracemo se
        PAIR p(prio, id); // snimamo ID i prioritet.
        qDelay.push(p);    // stavljamo se u red
        signal(mutex);     // pustamo mutex
        wait(privs[id]);   // blokiramo se na privatnom semaforu
        wait(mutex)        // mutex moramo sami da uzmemo nazad
    }
    isWorking = true;      // postavljamo da smo u krit. sekciji
    signal(mutex);         // pustamo mutex
}


void sjn_signal()
{
    wait(mutex);               // uzimamo mutex
    isWorking = false;         // postavljamo da vise nismo u krit. sekciji
    if (! qDelay.empty()) {    // ako ima nekog zablokiranog
        PAIR p = qDelay.top();
        int dPrio = p.first;
        int dId = p.second;
        qDelay.pop();
        signal(privs[id]);     // deblokiramo zablokiranog
    }
    // pustamo mutex bilo da smo blokirali nekog ili ne
    // ako smo deblokirali nekog, on ce sam uzeti mutex
    signal(mutex);
}

void a(int id)
```

```
{
    int prio = rand() % 1000; // nasumicni prioritet
    while (true) {
        sjn_wait(id, prio);
        work();
        sjn_signal();
    }
}


int main() {
    Thread ta[N];
    for (int i = 0; i < N; i++)
        ta[i] = createThread(a, i);
    for (int i = 0; i < N; i++)
        join(ta[i]);
    return 0;
}
```

2.7 Sleeping barber

Zadatak Proces je pokrenuo jednu nit koja simulira ponašanje uspavanog berberina i N niti koje simuliraju ponašanje mušterija koje dolaze kod berberina da se šišaju. Berberin spava dok mu ne dodje mušterija u berbernicu. Mušterija kada dodje u berbernicu treba da probudi berberina i sedne u red čekanja dok je berberin ne pozove na šišanje. Kada berberin završi sa šišanjem mušterije, mušterja plaća berberinu i odlazi. Ukoliko u berbernici nema mesta za sedenje, mušterija odlazi i vraća se nakon nekog vremena. Pomoću semafora implementirati metode `void barber()` i `void customer(int id)` koje respektivno oponašaju berberina i mušterije.

Rešenje

```
#include <thread.h>
#include <sem.h>
#include <queue>
#define MAX_SEATS 5
#define N 10
Sem privb = 0; // privatni semafor berberina
Sem privc[N];  // privatni semafori musterija
Sem mutex = 1; // mutex koji stiti qSeats
std::queue<int> qSeats;
extern void cut_hair(int custId);

void barber() {
    while (true) {
        wait(privb);
        wait(mutex);
        int custId = qSeats.front();
        qSeats.pop();
        signal(mutex);
        signal(privc[custId]);
        cut_hair(custId);
    }
}
void customer(int id) {
    while (true) {
        wait(mutex);
        if (qSeats.size() < MAX_SEATS)  {
            qSeats.push(id);
            signal(mutex);
            wait(privc[id]);
            cut_hair(id);
        } else {
            signal(mutex);
        }
        usleep(100);
    }
}
int main() {
    for (int i = 0; i < N; i++)
        init(privc[i], 0);
    Thread tb = createThread(barber);
    Thread tc[N];
    for (int i = 0; i < N; i++)
        tc[i] = createThread(customer, i);
    join(tb);
    join(tb);
    return 0;
}
```

2.8 Readers and writers

Zadatak Proces ima jedan otvoren deljeni fajl i pokrenuo je NR niti čitalaca koje izvršavaju funkciju `reader(int id)` i NW niti pisaca koje izvršavaju funkciju `writer(int id)`. Niti čitaoci mogu istovremeno čitati iz fajla, dok niti pisci moraju imati ekskluzivno pravo pristupa. Ne sme se dogoditi da čitalac i pisac pristupaju fajlu u isto vreme, i ne sme dogoditi da dva pisca upisuju u fajl u isto vreme.

Šta je rwlock?
Readers/Writers lock je brava koja štiti neki resurs, i koja može da se zaključa na dva odvojena načina: za ekskluzivno pravo pristupa (`exclusive (writer) lock`) ili za deljeno pravo prsitupa (`shared (reader) lock`). U ovom primeru ćemo implementirati `rw_lock(...)` pomoću semafora i tehnikom prosleđivanja štafetne palice, i na taj način rešiti problem čitaoca i pisaca.

Primer rwlock-a sa `flock()`:

```
int fd = open("file.txt", O_CREAT | O_RDWR);
...
void writer() {
    flock(fd, LOCK_EX);
    write();
    flock(fd, LOCK_UN);
}
void reader() {
    flock(fd, LOCK_SH);
    read();
    flock(fd, LOCK_UN);
}
...
close(fd);
```

Primer rwlock-a sa POSIX nitima:

```
pthread_rwlock_t rw;
pthread_rwlock_init(&rw, NULL);
...
void writer() {
    pthread_rwlock_wrlock(&rw);
    write();
    pthread_rwlock_unlock(&rw);
}
void reader() {
    pthread_rwlock_rdlock(&rw);
    read();
    pthread_rwlock_unlock(&rw);
}
...
pthread_rwlock_destroy(&rw);
```

Polurešenje (izgladnjivanje pisaca)

```
#include <thread.h>
#include <sem.h>
#include <queue>
#define NR 10
#define NW 5
#define LOCK true
#define UNLOCK false
enum {
    READER,
    WRITER,
};

int wCount = 0; // nw
int rCount = 0; // nr
int wDelay = 0; // dw
int rDelay = 0; // dr
Sem mutex = 1;  // e
Sem wSem = 0;   // w
Sem rSem = 0;   // r
```

```
void rw_lock(int type, int lock) {
    // WAIT()
    wait(mutex);
    if (lock) {
        if (type == WRITER) {
            // DELAY WRITER
            if (wCount > 0 || rCount > 0) {
                wDelay += 1;
                signal(mutex);
                wait(wSem);
            }
            wCount += 1;
        } else if (type == READER) {
            // DELAY READER
            if (wCount > 0) {
                rDelay += 1;
                signal(mutex);
                wait(rSem);
            }
            rCount += 1;
        }
    } else {
        if (type == WRITER) {
            wCount -= 1;
        } else if (type == READER) {
            rCount -= 1;
        }

    }
    // SIGNAL()
    if (wDelay > 0 && wCount == 0 && rCount == 0) {
        wDelay -= 1;
        signal(wSem);
    } else if (rDelay > 0 && wCount == 0) {
        rDelay -= 1;
        signal(rSem);
    } else {
        signal(mutex);
    }
}

FILE *fShared = ...;
void reader(int id) {
    while (true) {
        rw_lock(READER, LOCK);
        read(fShared);
        rw_lock(READER, UNLOCK);
        usleep(10);
    }
}
void writer(int id) {
    while (true) {
        rw_lock(WRITER, LOCK);
        write(fShared);
        rw_lock(WRITER, UNLOCK);
        usleep(10);
    }
}
int main() {
    Thread tr[NR], tw[NW];
    for (int i = 0; i < NR; i++)
        tr[i] = createThread(reader, i);
    for (int i = 0; i < NW; i++)
        tw[i] = createThread(writer, NR + i);
    for (int i = 0; i < NR; i++)
        join(tr[i]);
    for (int i = 0; i < NW; i++)
        join(tw[i]);
    return 0;
}
```

Rešenje (FIFO ulazak)

```
#include <thread.h>
#include <sem.h>
#include <queue>
#include <stdio.h>
using namespace std;
#define NR 10
#define NW 5
#define LOCK true
#define UNLOCK false
enum {
    WRITER = 0,
    READER = 1
};

Sem mutex = 1;
int count[2] = {0, 0};
typedef pair<int, Sem&> PAIR;
queue<PAIR> qWait;

void rw_lock(int type, int lock)
{
    // WAIT()
    wait(mutex);
    if (lock) {
        // DELAY()
        int shouldDelay = ! qWait.empty()
            || type == READER && count[WRITER] > 0
            || type == WRITER && (count[WRITER] > 0 || count[READER] > 0);
        if (shouldDelay) {
            Sem sem = 0; // privatni semafor
            qWait.push(PAIR(type, sem));
            signal(mutex);
            wait(sem);
        }
        count[type] += 1;
    } else {
        count[type] -= 1;
    }
    // SIGNAL()
    if (! qWait.empty()) {
        PAIR p = qWait.front();
        int pType = p.first;
        Sem& pSem = p.second;
        int shouldUnblock =
            pType == READER && count[WRITER] == 0
            || pType == WRITER && count[WRITER] == 0 && count[READER] == 0;
        if (shouldUnblock) {
            qWait.pop();
            signal(pSem);
        } else {
            signal(mutex);
        }
    } else {
        signal(mutex);
    }
}

void reader(int id) {
    while (true) {
        rw_lock(READER, LOCK);
        read();
        rw_lock(READER, UNLOCK);
        usleep(10);
    }
}

void writer(int id) {
    while (true) {
        rw_lock(WRITER, LOCK);
        write();
        rw_lock(WRITER, UNLOCK);
        usleep(10);
```

```
    }
}

int main() {
    Thread tr[NR];
    Thread tw[NW];
    for (int i = 0; i < NR; i++)
        tr[i] = createThread(reader, i);
    for (int i = 0; i < NW; i++)
        tw[i] = createThread(writer, i);
    for (int i = 0; i < NR; i++)
        join(tr[i]);
    for (int i = 0; i < NW; i++)
        join(tw[i]);
    return 0;
}
```

2.9 Cigarette smokers problem

Zadatak Proces ima deljenu strukturu `table` i pokrenuo je jednu nit agenta `agent()` i tri niti pušača `smoker(MATCHES)`, `smoker(PAPER)` i `smoker(TOBACCO)`. Pušači imaju beskonačne zalihe jedne vrste predmeta, ali su im potrebene i druge dve vrste da bi pušili. Agent na sto nasumično stavlja neka dva predmeta i čeka dok ih neki pušač ne uzme. Pušač koji vidi da može da puši sa ta dva predmeta uzima ih sa stola i puši. Kada pušač završi pušenje, agentu signalizira da može opet da stavlja predmete na sto. Implementirati ponašanje pušača `void smoker(int item)` i agenta `agent()` pomoću semafora.

Rešenje

```
#include <thread.h>
#include <sem.h>
#include <stdlib.h>
enum {
    MATCHES = 0,
    TOBACCO = 1,
    PAPER   = 2
};
struct Table {
    bool matches = false;
    bool paper   = false;
    bool tobacco = false;
} table;
Sem matches = 0;
Sem tobacco = 0;
Sem paper   = 0;
Sem ok      = 0;

void agent() {
    while (true) {
        // produce items...
        int item = rand() % 3;
        // put items on the table and wait for smoker
        switch (item) {
            case MATCHES: {
                table.paper = true;
                table.tobacco = true;
                signal(matches);
                wait(ok);
            } break;
            case PAPER: {
                table.matches = true;
                table.tobacco = true;
                signal(paper);
                wait(ok);
            } break;
            case TOBACCO: {
                table.matches = true;
                table.paper = true;
                signal(tobacco);
                wait(ok);
            } break;
        }
    }
}
void smoker(int id) {
    while (true) {
        switch (id) {
            case MATCHES: {
                wait(matches);
                table.paper = false;
                table.tobacco = false;
                // create cigarette and smoke...
                // signal agent
                signal(ok);
            } break;
            case PAPER: {
                wait(paper);
                table.matches = false;
```

```
                table.tobacco = false;
                // create cigarette and smoke...
                // signal agent
                signal(ok);
            } break;
            case TOBACCO: {
                wait(tobacco);
                table.matches = false;
                table.paper = false;
                // create cigarette and smoke...
                // signal agent
                signal(ok);
            } break;
        }
    }

}
int main() {
    Thread ta = createThread(agent);
    Thread ts1 = createThread(smoker, MATCHES);
    Thread ts2 = createThread(smoker, PAPER);
    Thread ts3 = createThread(smoker, TOBACCO);
    join(ta);
    join(ts1);
    join(ts2);
    join(ts3);
    return 0;
}
```

1. dodatno rešenje u slučaju da `agent()` ne radi signalizaciju

Ako agent nikako ne signalizira pušače, pušači moraju da izvšavaju `busy wait` i povremeno proveravaju da li se na stolu nalaze predmeti koji su potrebni. S obzirom da više niti mogu istovremeno da pristupaju stolu, neophodan je mutex.

```
#include <thread.h>
#include <sem.h>
#include <stdlib.h>
#define ITEM_COUNT  3
#define THIRD_ITEM(first, second) (1 - (((first) - 1) + ((second) -1)))
#define OTHER_ITEM_1(item) ((item) == 0 ? ITEM_COUNT - 1 : (item) - 1)
#define OTHER_ITEM_2(item) (((item) + 1) % ITEM_COUNT)
enum {
    MATCHES = 0,
    TOBACCO = 1,
    PAPER   = 2
};
Sem mutex = 1;
Sem ok = 0;
bool table[ITEM_COUNT] = { false, false, false };

void agent() {
    while (true) {
        // produce two items...
        int item = rand() % 3;
        // put items on the table and wait for smoker
        wait(mutex);
        table[OTHER_ITEM_1(item)] = true;
        table[OTHER_ITEM_2(item)] = true;
        signal(mutex);
        wait(ok);
    }
}
void smoker(int item) {
    int item1 = OTHER_ITEM_1(item);
    int item2 = OTHER_ITEM_2(item);
    while (true) {
        // wait for items (busy wait)
        while (true) {
            wait(mutex);
            if (table[item1] && table[item2]) {
                table[item1] = false;
                table[item2] = false;
                signal(mutex);
                break;
            } else {
                signal(mutex);
                usleep(50);
            }
        }
        // create cigarette and smoke...
        // signal agent
        signal(ok);
    }
}
int main() {
    Thread ta = createThread(agent);
    Thread ts1 = createThread(smoker, MATCHES);
    Thread ts2 = createThread(smoker, PAPER);
    Thread ts3 = createThread(smoker, TOBACCO);
    join(ta);
    join(ts1);
    join(ts2);
    join(ts3);
    return 0;
}
```

2. dodatno rešenje - u slučaju da `agent()` signalizira jedan semafor

Recimo da agent kada stavi dva predmeta na sto signalizira semafor onog predmeta koji **nije** stavio na sto. Pušač tada treba da čeka na semafor predmeta koji mu **ne treba** jer se tada nalaze na stolu nalaze predmeti koji mu trebaju. S obzirom da u ovom rešenju samo jedna nit može pristupati stolu u nekom trenutku, mutex nije potreban ali nije greška iskoristi ga.

```
#include <thread.h>
#include <sem.h>
#include <stdlib.h>
#define ITEM_COUNT 3
#define OTHER_ITEM_1(item) (((item) + 1) % ITEM_COUNT)
#define OTHER_ITEM_2(item) ((item) == 0 ? (ITEM_COUNT - 1) : ((item) - 1))
enum {
    MATCHES = 0,
    TOBACCO = 1,
    PAPER   = 2
};
Sem sem[ITEM_COUNT] = { 0, 0, 0 };
Sem ok = 0;
// Sem mutex = 1;
bool table[ITEM_COUNT] = { false, false, false };

void agent() {
    while (true) {
        // produce two items...
        int item = rand() % 3;
        // put them on the table and wait for smoker
        //wait(mutex);
        table[OTHER_ITEM_1(item)] = true;
        table[OTHER_ITEM_2(item)] = true;
        //signal(mutex);
        signal(sem[item]);
        wait(ok);
    }
}
void smoker(int item) {
    int item1 = OTHER_ITEM_1(item);
    int item2 = OTHER_ITEM_2(item);
    while (true) {
        // wait for items
        wait(sem[item]);
        //wait(mutex);
        table[item1] = false;
        table[item2] = false;
        //signal(mutex);
        // create cigarette and smoke ...
        // signal agent
        signal(ok);
    }

}
int main() {
    Thread ta = createThread(agent);
    Thread ts1 = createThread(smoker, MATCHES);
    Thread ts2 = createThread(smoker, PAPER);
    Thread ts3 = createThread(smoker, TOBACCO);
    join(ta);
    join(ts1);
    join(ts2);
    join(ts3);
    return 0;
}
```

3. dodatno rešenje - u slučaju da `agent()` signalizira dva semafora

Recimo da agent kada stavi dva predmeta na sto signalizira dva semafora i to baš onih predmeta koje je i stavio na sto. U tom slučaju, da bi svi signali bili uhvaćeni, svaki pušač i dalje mora da čeka na semafor jednog predmeta i toj onog predmeta koji im **ne treba**. Dva pušača koji uhvate signale od agenta koji treba da se dogovore da probude onog trećeg jer je treći pušač taj koji može da uzme predmete sa stola i puši. Mutex nije neophodan za `table[]` ali je neophodan za `agree[]` koji pušači koriste da se dogovaraju na pomenut način.

```
#include <thread.h>
#include <sem.h>
#include <stdlib.h>
#define ITEM_COUNT 3
#define OTHER_ITEM_1(item) (((item) + 1) % ITEM_COUNT)
#define OTHER_ITEM_2(item) ((item) == 0 ? (ITEM_COUNT - 1) : ((item) - 1))
enum {
    MATCHES = 0,
    TOBACCO = 1,
    PAPER   = 2
};
Sem sem[] = { 0, 0, 0 };
Sem ok = 0;
Sem mutex = 1;
bool table[] = { false, false, false };
bool agree[] = { false, false, false };

void agent() {
    while (true) {
        // produce two items...
        int item = rand() % 3;
        // put items on the table and wait for smoker
        table[OTHER_ITEM_1(item)] = true;
        table[OTHER_ITEM_2(item)] = true;
        signal(sem[OTHER_ITEM_1(item)]);
        signal(sem[OTHER_ITEM_2(item)]);
        wait(ok);
    }
}
void smoker(int id) {
    while (true) {
        //wait for itmes
        while (true) {
            wait(sem[id]);
            wait(mutex);
            agree[id] = true;
            if (agree[PAPER] && agree[MATCHES] && agree[TOBACCO]) {
                agree[PAPER] = false;
                agree[MATCHES] = false;
                agree[TOBACCO] = false;
                signal(mutex);
                break;
            } else if (agree[id] && agree[OTHER_ITEM_1(id)]) {
                signal(sem[OTHER_ITEM_2(id)]);
            } else if (agree[id] && agree[OTHER_ITEM_2(id)]) {
                signal(sem[OTHER_ITEM_1(id)]);
            }
            signal(mutex);
        }
        table[OTHER_ITEM_1(id)] = false;
        table[OTHER_ITEM_2(id)] = false;
        // create cigarette and smoke...
        // signal agent
        signal(ok);
    }

}
int main() {
    Thread ta = createThread(agent);
    Thread ts1 = createThread(smoker, MATCHES);
    Thread ts2 = createThread(smoker, PAPER);
    Thread ts3 = createThread(smoker, TOBACCO);
    join(ta);
    join(ts1);
```

```
    join(ts2);
    join(ts3);
    return 0;
}
```

4. dodatno rešenje - u slučaju da `agent()` signalizira tri semafora.

Recimo da agent signalizira semafore svih predmeta kada god stavi nešto na sto. U tom slučaju, pušač treba da čeka na jedan od semafora i samo proveri da li su predmeti koji su njemu potrebni na stolu.

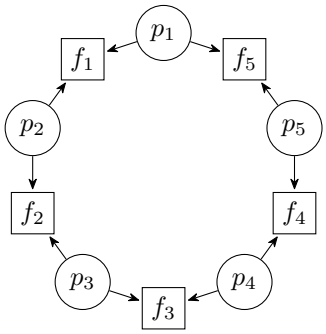
```
#include <thread.h>
#include <sem.h>
#include <stdlib.h>
#define ITEM_COUNT  3
#define THIRD_ITEM(first, second) (1 - (((first) - 1) + ((second) -1)))
#define OTHER_ITEM_1(item) ((item) == 0 ? ITEM_COUNT - 1 : (item) - 1)
#define OTHER_ITEM_2(item) (((item) + 1) % ITEM_COUNT)
enum {
    MATCHES = 0,
    TOBACCO = 1,
    PAPER   = 2
};
Sem mutex = 1;
Sem ok = 0;
Sem sem[ITEM_COUNT] = { 0, 0, 0 };
bool table[ITEM_COUNT] = { false, false, false };
void agent() {
    while (true) {
        // produce two items
        int item = rand() % 3;
        // put them on table and wait for smoker
        wait(mutex);
        table[OTHER_ITEM_1(item)] = true;
        table[OTHER_ITEM_2(item)] = true;
        signal(mutex);
        signal(sem[MATCHES]);
        signal(sem[TOBACCO]);
        signal(sem[PAPER]);
        wait(ok);
    }
}
void smoker(int item) {
    int item1 = OTHER_ITEM_1(item);
    int item2 = OTHER_ITEM_2(item);
    while (true) {
        // wait for items
        bool spin = true;
        while (spin) {
            wait(sem[item]);
            wait(mutex);
            if (table[item1] && table[item2]) {
                table[item1] = false;
                table[item2] = false;
                spin = false;
            }
            signal(mutex);
        }
        // create cigarette and smoke...
        signal(ok);
    }

}
int main() {
    Thread ta = createThread(agent);
    Thread ts1 = createThread(smoker, MATCHES);
    Thread ts2 = createThread(smoker, PAPER);
    Thread ts3 = createThread(smoker, TOBACCO);
    join(ta);
    join(ts1);
    join(ts2);
    join(ts3);
    return 0;
}
```

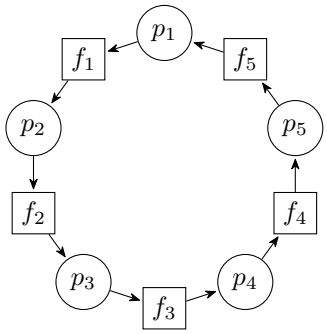
2.10 Dining philosophers

Zadatak Proces je pokrenuo 5 niti koje simuliraju ponašanje filozofa za stolom. Filozof neko vreme misli, uzima levu i desnu viljušku na stolu, jede ručak na tanjiru, vraća viljuške, i nastavlja da misli. Filozof ne može da uzme levu viljušku ukoliko ju je trenutno filozof levo koristi, i slično ne moze da uzme desnu viljušku ukoliko ju je trenutno filozof desno koristi. Ukoliko svih pet filozofa u isto vreme pokušaju da uzmu levu viljušku, ni jedan filozof neće moći da uzme desnu viljušku, i tada nastupa deadlock. Implementirati ponašanje filozofa pomoću semafora tako da se ovaj deadlock ne može dogoditi.

Šta je graf alokacije resursa?
Graf alokacije resursa (eng. Resource allocation graph (RAG)) je usmeren graf u kome čvorovi predstavljaju niti ili resurse, a grane povezuju niti i resurse. Ukoliko je grana usmerena od čvora niti ka čvoru resursa (nit → resurs), data nit može u nekom trenutku zauzeti taj resurs. Ukoliko je nit usmerena od čvora resursa ka čvoru niti (nit ← resurs), data nit je zauzela resurs i trenutno ga drži. Ukoliko se u grafu pojavi **petlja**, nastaje deadlock. Primer grafa alokacije resursa:



(a) Graf filozofa



(b) Deadlock filozofa

1. način

Možemo obrnuti redosled uzimanja viljušaka jednog filozofa (ili parnih filozofa) i time otkloniti mogućnost pojave petlje u grafu alokacije resursa.

```
#include <thread.h>
#include <sem.h>
#define N 5
Sem forks[N] = { 1, 1, 1, 1, 1 };
void phil(int id) {
    int right = (id + 1) % N;
    int left = (id == 0) ? (N - 1) : (id - 1); // alternative: ‘int left = id;‘
    while (true) {
        // Think
        think();
        // Acquire forks
        if (id % 2 == 0) { // alternative: ‘if (id == 0)‘
            wait(forks[left]);
            wait(forks[right]);
        } else {
            wait(forks[right]);
            wait(forks[left]);
        }
        // Eat
        eat();
        // Release forks
        signal(forks[left]);
        signal(forks[right]);
    }
}
int main() {
    Thread tp[N];
    for (int i = 0; i < N; i++)
        tp[i] = createThread(phil, i);
    for (int i = 0; i < N; i++)
        join(tp[i]);
    return 0;
```

```
}
```

2. način

Možemo napraviti uposleno čekanje i zauzeti viljuške kada postanu slobodne (**Napomena:** Uposleno čekanje je neefikasno i kada su na raspolaganju semafori, obično je loše praviti uposleno čekanje ukoliko ono nije neophodno)

```
#include <thread.h>
#include <sem.h>
#define N 5
Sem mutex = 1;
bool avail[] = { true, true, true, true, true };

void phil(int id) {
    int right = (id + 1) % N;
    int left = (id == 0) ? (N - 1) : (id - 1);
    while (true) {
        // Think
        think();
        // Acquire forks
        while (true) {
            wait(mutex);
            if (avail[left] && avail[right])  {
                avail[left] = false;
                avail[right] = false;
                signal(mutex);
                break;
            } else {
                signal(mutex);
                usleep(50);
            }
        }
        // Eat
        eat();
        // Release forks
        wait(mutex);
        avail[left] = true;
        avail[right] = true;
        signal(mutex);
    }
}
int main() {
    Thread tp[N];
    for (int i = 0; i < N; i++)
        tp[i] = createThread(phil, i);
    for (int i = 0; i < N; i++)
        join(tp[i]);
    return 0;
}
```

3. način

Možemo imati semafor `ticket` inicijalizovan na vrednost $N-1$, ukoliko je N broj filozofa. Tako poslednji filozof ne može da zahteva viljuške ukoliko $N-1$ filozofa pre zahtevalo viljške, pa se ne može javiti petlja.

```
#include <thread.h>
#include <sem.h>
#define N 5

Sem sFork[N] = { 1, 1, 1, 1, 1 };
Sem ticket = 4;

int phil(int id)
{
    int left = id;
    int right = (id + 1) % N;
    while (true) {
        // Think
        think();
        // Acquire forks
        wait(ticket);
        wait(sFork[left]);
        wait(sFork[right]);
        // Eat
        eat();
```

```
        // Release forks
        signal(sFork[left]);
        signal(sFork[right]);
        signal(ticket);
    }
}
int main() {
    Thread tp[N];
    for (int i = 0; i < N; i++)
        tp[i] = createThread(phil, i);
    for (int i = 0; i < N; i++)
        join(tp[i]);
    return 0;
}
```

4. način

Možemo koristiti privatne semafore filozofa na koji će se filozof blokirati ukoliko ne može da uzme viljuške. Filozof mora da najavi da nije uspeo da uzme viljuške (preko `hungry[]`) kako bi filozof koji oslobađa viljuške znao da li treba da pokuša da deblokira filozofa levo ili desno pored sebe.

```
#include <thread.h>
#include <sem.h>
#define N 5
#define LEFT(id) (((id) + 1) %  N)
#define RIGHT(id) (((id) == 0) ? (N - 1) : ((id) - 1))
#define CAN_EAT(id) (available[LEFT(id)] && available[RIGHT(id)])
Sem mutex = 1;
Sem privs[] = { 0, 0, 0, 0, 0 };
bool available[] = { true, true, true, true, true };
bool hungry[] = { false, false, false, false, false };

void phil(int id)
{
    int left = LEFT(id);
    int right = RIGHT(id);
    while (true) {
        // Think
        think();
        // Acquire forks
        wait(mutex);
        if (! CAN_EAT(id)) {
            hungry[id] = true;
            signal(mutex);
            wait(privs[id]);
        }
        hungry[id] = false;
        available[left] = false;
        available[right] = false;
        signal(mutex);
        // Eat
        eat();
        // Release forks
        wait(mutex);
        available[left] = true;
        available[right] = true;
        if (hungry[left] && CAN_EAT(left)) {
            signal(privs[left]);
        } else if (hungry[right] && CAN_EAT(right)) {
            signal(privs[right]);
        } else {
            signal(mutex);
        }
    }
}
int main() {
    Thread tp[N];
    for (int i = 0; i < N; i++)
        tp[i] = createThread(phil, i);
    for (int i = 0; i < N; i++)
        join(tp[i]);
    return 0;
}
```

3 Regioni

3.1 Uvod

Šta je uslovni kritični region?
Uslovni kritični regioni pružaju mogućnost za medjusobno isključivanje izmedju niti korišćenjem iskaza `region (s) {...}` , kao i mogućnost da nit unutar regiona privremeno napusti kritični region dok se ne ispuni neki neophodan uslov pomoću `await(...)` iskaza.

Kritični regioni su vezani za neku strukturu s nad kojom može da se pozove `region (s) { ... }` iskaz. To je ovoj skripti bilo koja struktura koja subklasira virtuelnu klasu `Region`. Unutar sebe, struktura `Region` sadrži mutex koji se zaključava prilikom svakog ulaska u region i otključava prilikom svakog izlaska iz regiona.

Nakon subklasiranja strukture `Region`, u strukturi se mogu dodati dodatni podaci čije vrednosti, po pravilu, **nije dozvoljeno** čitati ili modifikovati van regionskog iskaza. Takodje, nije smisleno pozivati `await(...)` van regionskog iskaza.

U jeziku Java, `synchronized (this) {...}` predstavlja primer regiona nad objektrom `this`. Iskaz `await(...)` se može implementirati kao `while (! ...)` petlja koja poziva `Object.wait()`. Ukoliko uslov postane ispunjen, nekako se mora pozvati `notify()` što može uraditi npr. nit koja uspešno izlazi iz regionskog bloka.

```
// region (lock) {
//      ...
//      await(condition)
//      ...
// }
synchronized(lock) { // enter region
    ...
    while (! condition) // await(condition)
        lock.wait();
    ...
    lock.notifyAll(); // make awaited threads wake up recheck their condition
} // exit region
```

A sledeći kod je primer implementacije regiona pomoću POSIX niti:

```
pthread_mutex_t m_mutex;
pthread_cond_t m_cond;
pthread_mutex_init(&m_mutex, NULL);
pthread_cond_init(&m_cond, NULL)
...
pthread_mutex_lock(&m_mutex); // enter region
...
while (! condition) // await(condition)
    pthread_cond_wait(&m_cond, &m_mutex);
...
pthread_cond_broadcast(&m_cond); // make awaited threads wake up recheck their condition
pthread_mutex_unlock(&m_mutex); // exit region
...
pthread_cond_destroy(&m_cond);
pthread_mutex_destroy(&m_mutex);
```

3.2 Simple synchronization

Zadatak Proces je pokrenuo dve niti sa beskonačnim petljama, gde prva nit ispisuje na konzoulu znak “a” dok druga nit ispisuje znak “b” (aaaaaaaaabbbbbbbbbbaaaaaaaabbbb...). Koristeći uslovne kritične regione, potrebno je sinhronizovati niti tako da se na konzoli naizmenično ispisuju znakovi “a” i “b”. (abababababababab...).

Rešenje

```
#include <thread.h>
#include <region.h>
#define N 10
struct AB : Region {
    int c = 'a';
} ab;

void a() {
    while (true) {
        region (ab) {
            await(ab.c == 'a');
        }
        print("A");
        region (ab) {
            ab.c = 'b';
        }
    }
}
void b() {
    while (true) {
        region (ab) {
            await(ab.c == 'b');
        }
        print("B");
        region (ab) {
            ab.c = 'a';
        }
    }
}
int main()
{
    Thread ta = createThread(a);
    Thread tb = createThread(b);
    join(ta);
    join(tb);
    return 0;
}
```

3.3 Mutual exclusion

Zadatak Proces je pokrenuo dve niti, gde obe rade neki posao `work()` koji se mora obavljati sa ekskluzivnim pravom pristupa (npr. upis u fajl). Obezbediti ovaj vid sinhronizacije pomoću uslovnih kritičnih regiona.

1. način

Možemo unutar regiona imati promenljivu `lock` koju ćemo setovati na `true` kada ulazimo u kritičnu ili na `false` ukoliko izlazimo iz kritične sekcije. Nećemo ulaziti u kritičnu sekciju ukoliko je ona trenutno zaključana.

```
#include <thread.h>
#include <region.h>
struct CritcalSection : Region {
    int lock = false;
} cs;

void a() {
    while (true) {
        region (cs) {
            await(cs.lock == false);
            cs.lock = true;
        }
        work();
        region (cs) {
            cs.lock = false;
        }
        usleep(10);
    }
}

void b() {
    while (true) {
        region (cs) {
            await(cs.lock == false);
            cs.lock = true;
        }
        work();
        region (cs) {
            cs.lock = false;
        }
        usleep(10);
    }
}
int main() {
    Thread ta = createThread(a);
    Thread tb = createThread(b);
    join(ta);
    join(tb);
    return 0;
}
```

2. način

S obzirom da kritični region i sam već implementira kritičnu sekciju, možemo jednostavno staviti `work()` unutar regionskog bloka.

```
#include <thread.h>
#include <region.h>
struct CritcalSection : Region {
    // nothing
} cs;

void a() {
    while (true) {
        region (cs) {
            work();
        }
        usleep(10);
    }
}
```

```
void b() {
    while (true) {
        region (cs) {
            work();
        }
        usleep(10);
    }
}
int main() {
    Thread ta = createThread(a);
    Thread tb = createThread(b);
    join(ta);
    join(tb);
    return 0;
}
```

3.4 Producer and consumer

Zadatak. Proces ima jedan deljeni ograničen kružni bafer veličine B i pokrenuo je dve niti: jednu nit `producer()` koja kreira podatke i smešta ih u bafer, i jednu nit `consumer()` koja dohvata podatke iz bafera i koristi ih. Ukoliko `producer` radi brže od `consumer`-a, nastupiće prekoračenje bafera (buffer overflow), dok ukoliko `consumer` radi brže od `producer`-a, `consumer` će početi da dohvata nevalidne ili nepostojaće podatke. Koristeći uslovne kritične regione, potrebno je obezbediti da se ove neželjene situacije ne dešavaju.

Rešenje

```
#include <thread.h>
#include <region.h>
#include <stdlib.h>
#define B 10
struct Buffer : Region {
    int data[B];
    int head = 0;
    int tail = 0;
    int count = 0;
} buffer;

void producer() {
    while (true) {
        /* Producing data... */
        int data = rand() % 1000;
        /* Put data into buffer */
        region (buffer) {
            await(buffer.count < B);
            buffer.data[buffer.head] = data;
            buffer.head = (buffer.head + 1) % B;
            buffer.count += 1;
        }
    }
}

void consumer() {
    while (true) {
        /* Get data from buffer */
        int data;
        region (buffer) {
            await(buffer.count > 0);
            data = buffer.data[buffer.tail];
            buffer.tail = (buffer.tail + 1) % B;
            buffer.count -= 1;
        }
        /* Consuming data... */
        print("%d", data);
    }
}
int main() {
    Thread tp = createThread(producer);
    Thread tc = createThread(consumer);
    join(tp);
    join(tc);
    return 0;
}
```

3.5 Barrier synchronization

Zadatak Proces je pokrenuo N niti koje treba da zajednički odrade neki posao `work1()` i treba predju na posao `work2()` tek kada svih N niti završe posao `work1()`. Slično, potrebno je da svih N završe posao `work2()` pre nego što se predje nazad na posao `work1()`. Poslovi se moraju obavljati sa ekskluzivnim pravom pristupa. Implementirati ovakvo ponašanje pomoću uslovnih kritičnih regiona.

Rešenje

```
#include <thread.h>
#include <region.h>
#include <stdlib.h>
#define N 10
struct Doors : Region {
    int count1 = 0;
    int count2 = 0;
} doors;

void a(int id) {
    while (true) {
        region (doors) {
            work1();
            doors.count1 += 1;
            if (doors.count1 == N) {
                doors.count2 = 0;
            }
            await(doors.count1 == N);
        }

        region (doors) {
            work2();
            doors.count2 += 1;
            if (doors.count2 == N) {
                doors.count1 = 0;
            }
            await(doors.count2 == N);
        }
    }
}

int main() {
    Thread ta[N];
    for (int i = 0; i < N; i++)
        ta[i] = createThread(a, i);
    for (int i = 0; i < N; i++)
        join(ta[i]);
    return 0;
}
```

3.6 Shortest job next

Zadatak Proces je pokrenuo N niti koje treba pristupe kritičnoj sekciji po prioritetnom redosledu. Naime, ukoliko postoje više niti koje istovremeno žele da udju u kritičnu sekciju, prva ulazi ona koja ima najmanje posla. Implementirati ovakvo ponašanje niti pomoću uslovnih kritičnih regiona.

Rešenje

Ukoliko je kritična sekcija zauzeta, možemo staviti sebe u prioritetni red i sačekati dok sledeći u redu ne bude baš naš `id`.

```
#include <thread.h>
#include <region.h>
#include <stdlib.h>
#include <queue>
using namespace std;
#define N 10

typedef pair<int, int> PAIR;

struct SJN : Region {
    bool locked = false;
    priority_queue<PAIR, vector<PAIR>, greater<PAIR>> qEnter;
} sjn;

void a(int id)
{
    int prio = rand() % 1000; // nasumicni prioritet
    while (true) {
        region (sjn) {
            sjn.qEnter.push(PAIR(prio, id));
            await(sjn.locked == false && sjn.qEnter.top().second == id);
            sjn.qEnter.pop();
            sjn.locked = true;
        }
        work();
        region (sjn) {
            sjn.locked = false;
        }
    }
}


int main()
{
    Thread tp[N];
    for (int i = 0; i < N; i++)
        tp[i] = createThread(a, i);
    for (int i = 0; i < N; i++)
        join(tp[i]);
    return 0;
}
```

3.7 Sleeping barber

Zadatak Proces je pokrenuo jednu nit koja simulira ponašanje uspavanog berberina i N niti koje simuliraju ponašanje mušterija koje dolaze kod berberina da se šišaju. Berberin spava dok mu ne dodje mušterija u berbernicu. Mušterija kada dodje u berbernicu treba da probudi berberina i sedne u red čekanja dok je berberin ne pozove na šišanje. Kada berberin završi sa šišanjem mušterije, mušterja plaća berberinu i odlazi. Ukoliko u berbernici nema mesta za sedenje, mušterija odlazi i vraća se nakon nekog vremena. Pomoću uslovnih kritičnih regiona, implementirati funkcije `void barber()` i `void customer(int id)` koje oponašaju berberina i mušterije respektivno.

Rešenje

```
#include <thread.h>
#include <region.h>
#include <queue>
using namespace std;
#define MAX_SEATS 5
#define N 10

struct BarberShop : Region {
    int custId = -1;
    bool paid = false;
    queue<int> qSeats;
} shop;

extern void cut_hair(int custId);

void barber() {
    int custId = -1;
    while (true) {
        region (shop) {
            await(shop.qSeats.size() > 0);
            shop.custId = shop.qSeats.front();
            shop.qSeats.pop();
        }
        /* cut hair */
        region (shop) {
            await(shop.paid);
            shop.paid = false;
            shop.custId = -1;
        }
    }
}

void customer(int id) {
    bool isCutting;
    while (true) {
        isCutting = false;
        region (shop) {
            if (shop.qSeats.size() < MAX_SEATS)  {
                isCutting = true;
                shop.qSeats.push(id);
                await(shop.custId == id);
            }
        }
        if (isCutting) {
            /* cut hair */
            region (shop) {
                shop.paid = true;
            }
        }
        usleep(100);
    }
}

int main() {
    Thread tb = createThread(barber);
    Thread tc[N];
    for (int i = 0; i < N; i++)
```

```
        tc[i] = createThread(customer, i);
    join(tb);
    for (int i = 0; i < N; i++)
        join(tc[i]);
    return 0;
}
```

3.8 Readers and writers

Zadatak Proces ima jedan otvoren deljeni fajl i pokrenuo je NR niti čitalaca koje izvršavaju funkciju `reader(int id)` i NW niti pisaca koje izvršavaju funkciju `writer(int id)`. Niti čitaoci mogu istovremeno čitati iz fajla, dok niti pisci moraju imati ekskluzivno pravo pristupa fajlu. Ne sme se dogoditi da čitalac i pisac pristupaju fajlu u isto vreme, i ne sme dogoditi da dva pisca upisuju u fajl u isto vreme.

Polurešenje (izgladnjivanje pisaca)

```
#include <thread.h>
#include <region.h>
#include <stdlib.h>
#define NR 10
#define NW 5
struct RW : Region {
    int wCount = 0;
    int rCount = 0;
} rw;
FILE* fShared = ...;

void reader(int id)  {
    while (true) {
        region (rw) {
            await(rw.wCount == 0);
            rw.rCount += 1;
        }
        read(fShared);
        region (rw) {
            rw.rCount -= 1;
        }
    }
}
void writer(int id)  {
    while (true) {
        region (rw) {
            await(rw.wCount == 0 && rw.rCount == 0);
            rw.wCount += 1;
        }
        write(fShared);
        region (rw) {
            rw.wCount -= 1;
        }
    }
}
int main() {
    Thread tr[NR], tw[NW];
    for (int i = 0; i < NR; i++)
        tr[i] = createThread(reader, i);
    for (int i = 0; i < NW; i++)
        tw[i] = createThread(writer, i);
    for (int i = 0; i < NR; i++)
        join(tr[i]);
    for (int i = 0; i < NW; i++)
        join(tw[i]);
    return 0;
}
```

Rešenje (FIFO ulazak)

```
#include <thread.h>
#include <region.h>
#include <queue>
#define NR 10
#define NW 5
enum {
    READER,
    WRITER
};
struct RW : Region {
    int count[2] = { 0, 0 };
    std::queue<int> delay;
} rw;

FILE *fShared = ...;
void reader(int id) {
    while (true) {
        region (rw) {
            rw.delay.push(id);
            await(rw.delay.front() == id && rw.count[WRITER] == 0);
            rw.delay.pop();
            rw.count[READER] += 1;
        }
        read(fShared);
        region (rw) {
            rw.count[READER] -= 1;
        }
    }
}

void writer(int id) {
    while (true) {
        region (rw) {
            rw.delay.push(id);
            await(rw.delay.front() == id && rw.count[WRITER] == 0 && rw.count[READER] ==
    0);
            rw.delay.pop();
            rw.count[WRITER] += 1;
        }
        write(fShared);
        region (rw) {
            rw.count[WRITER] -= 1;
        }
    }
}

int main() {
    Thread tr[NR], tw[NW];
    for (int i = 0; i < NR; i++)
        tr[i] = createThread(reader, i);
    for (int i = 0; i < NW; i++)
        tw[i] = createThread(writer, NR + i);
    for (int i = 0; i < NR; i++)
        join(tr[i]);
    for (int i = 0; i < NW; i++)
        join(tw[i]);
    return 0;
}
```

3.9 Cigarette smokers

Zadatak Proces ima deljenu strukturu `table` i pokrenuo je jednu nit agenta `agent()` i tri niti pušača `smoker(MATCHES)`, `smoker(PAPER)` i `smoker(TOBACCO)`. Pušači imaju beskonačne zalihe jedne vrste predmeta, ali su im potrebene i druge dve vrste da bi pušili. Agent na sto nasumično stavlja neka dva predmeta i čeka dok ih neki pušač ne uzme. Pušač koji vidi da može da puši sa ta dva predmeta uzima ih sa stola i puši. Kada pušač završi pušenje, agentu signalizira da može opet da stavlja predmete na sto. Implementirati ponašanje pušača `void smoker(int item)` i agenta `agent()` pomoću uslovnih kritičnih regiona.

Rešenje

```
#include <thread.h>
#include <region.h>
#include <stdlib.h>

enum {
    MATCHES = 0,
    TOBACCO = 1,
    PAPER   = 2
};

struct Table : Region {
    bool matches = false;
    bool paper   = false;
    bool tobacco = false;
    bool ok      = false;
} table;

void agent() {
    while (true) {
        // produce items...
        int item = rand() % 3;
        // put items on the table
        switch (item) {
            case MATCHES: {
                region (table) {
                    table.paper = true;
                    table.tobacco = true;
                    await(table.ok);
                    table.ok = false;
                }
            } break;
            case PAPER: {
                region (table) {
                    table.matches = true;
                    table.tobacco = true;
                    await(table.ok);
                    table.ok = false;
                }
            } break;
            case TOBACCO: {
                region (table) {
                    table.matches = true;
                    table.paper = true;
                    await(table.ok);
                    table.ok = false;
                }
            } break;
        }
    }
}

void smoker(int id) {
    while (true) {
        switch (id) {
            case MATCHES: {
                region (table) {
                    await(table.paper && table.tobacco);
```

```
                    table.paper = false;
                    table.tobacco = false;
                }
                // create a cigarette with the three items
                // smoke...
                region (table) {
                    table.ok = true;
                }
            } break;
            case PAPER: {
                region (table) {
                    await(table.matches && table.tobacco);
                    table.matches = false;
                    table.tobacco = false;
                }
                // create a cigarette with the three items
                // smoke...
                region (table) {
                    table.ok = true;
                }
            } break;
            case TOBACCO: {
                region (table) {
                    await(table.matches && table.paper);
                    table.matches = false;
                    table.paper = false;
                }
                // create a cigarette with the three items
                // smoke...
                region (table) {
                    table.ok = true;
                }
            } break;
        }
    }

}


int main() {
    Thread ta = createThread(agent);
    Thread ts1 = createThread(smoker, MATCHES);
    Thread ts2 = createThread(smoker, PAPER);
    Thread ts3 = createThread(smoker, TOBACCO);
    join(ta);
    join(ts1);
    join(ts2);
    join(ts3);
    return 0;
}
```

3.10 Dining philosophers

Zadatak Proces je pokrenuo 5 niti koje simuliraju ponašanje filozofa za stolom. Filozof neko vreme misli, uzima levu i desnu viljušku na stolu, jede ručak na tanjiru, vraća viljuške, i nastavlja da misli. Filozof ne može da uzme levu viljušku ukoliko ju je trenutno filozof levo koristi, i slično ne moze da uzme desnu viljušku ukoliko ju je trenutno filozof desno koristi. Ukoliko svih pet filozofa u isto vreme pokušaju da uzmu levu viljušku, ni jedan filozof neće moći da uzme desnu viljušku, i tada nastupa deadlock. Implementirati ponašanje filozofa pomoću uslovnih kritičnih regiona tako da se ovaj deadlock ne može dogoditi.

Rešenje

Najprirodnije rešenje filozofa sa uslovnim kritičnim regionima jeste atomično uzeti obe viljuške kada obe viljuške postanu slobodne.

```
#include <thread.h>
#include <region.h>
#define N 5

struct Table : Region {
    bool fork[N] = { false };
} t;

void phil(int id) {
    int left = id == 0 ? N - 1 : id - 1;
    int right = (id + 1) % N;
    while (true) {
        // Think
        think();
        // Acquire forks
        region (t) {
            await(t.fork[left] == false && t.fork[right] == false);
            t.fork[left] = true;
            t.fork[right] = true;
        }
        // Eat
        eat();
        // Release forks
        region (t) {
            t.fork[left] = false;
            t.fork[right] = false;
        }
    }
}

int main() {
    Thread tp[N];
    for (int i = 0; i < N; i++)
        tp[i] = createThread(phil, i);
    for (int i = 0; i < N; i++)
        join(tp[i]);
    return 0;
}
```

4 Monitori

4.1 Uvod

Šta je monitor?
Monitor je konstrukt koji je nalik klasi (u ovoj skripti je to bilo koja klasa koja subklasira apstratknu klasu `Monitor`) koji omogućava da se kod za sinhronizaciju i medjusobno isključivanje apstrahuje van logike koju izvršava nit. Kod semafora i regiona, kod za medjusobno isključivanje i sinhronizaciju je bio izmešan sa kodom za logiku. Kod monitora, niti isključivo pozivaju **monitorske procedure** i ne vrše sami sinhronizaciju, već monitor obavlja to monitor za njih. Monitor sadrži interni `mutex`, red čekanja za ulazak u monitor, i može sadržati uslovne i promenljive `cond`.

Šta su monitorske procedure?
Monitorske procedure su metode monitora u koje se ulazi sinhrono t.j. zaključavanjem monitora prilikom ulaska, i otključavanjem monitora prilikom izlaska. Kada neka nit udje u monitorsku proceduru njenim pozivom, ni jedna druga nit ne može ući ni u jednu monitorsku proceduru dok prvobitna nit ne napusti monitor dolaženjem do kraja monitorske procedure, pozivom `cond.wait()`, ili pozivom `cond.signal()` kod `Signal-and-Wait` discipline.

Sa semaforima, monitorska procedura se može implemntirati pozivom `wait(mutex)` na početku procedure, i `signal(mutex)` na kraju procedure. Sa regionima, monitorska procedura se može implementirati kao metoda koja obavija svoj kod sa `region (this) {...}`. U ovoj skripti, svaka monitorska procedura je označena blokom `monitor {...}` što ima isto značenje kao `region (this) {...}`, ili `synchronized (this) {...}` u jeziku Java.

Primer monitorskih procedura u jeziku Java:

```
public class Buffer {
    public synchronized void put(int data) {
        ...
    }
    public synchronized int get() {
        ...
    }
}
```

Šta su uslovne promenljive (`cond`)?
Uslovne promenljive omogućavaju privremeno napuštanje monitora i suspendovanje niti unutar monitorske procedure dok se ne ispuni neki uslov. Uslove promenljive sadrže (prioritetni) red blokiranih niti (`priority_queue<int, Thread*>`) i podržavaju sledećih 6 (ili 7) operacija:

1. `void wait()` - Nit koja pozove ovu operaciju izlazi monitora i dodaje se u red blokiranih na uslovnoj promenljivoj. Deblokiranje će se vršiti po FIFO redosledu blokiranja.
2. `void wait(int rank)` - Nit koja pozove ovu operaciju izlazi monitora i dodaje se u red blokiranih na uslovnoj promenljivoj. Deblokiranje će se vršiti po prioritetnom redosledu blokiranja (manja vrednost znači veći prioritet).
3. `void signal()` - Ukoliko nema blokiranih niti na uslovnoj promenljivoj, ova operacija ne radi ništa. Ukoliko ima blokirnih niti, jedna nit se deblokira po redosledu navedenom gore. Koja nit dobija kontekst zavisi od discipline monitora (`SW` ili `SC`).

4. `void signalAll()` - Sve niti blokirane na uslovnoj promenljivoj se deblokiraju. Ukoliko je red blokiranih prazan, ova operacija ne radi ništa. **Napomena:** Ova operacija postoji samo kod monitora sa `Signal-and-Continue` disciplinom!

5. `bool queue()` - Odgovor da li postoje niti blokirane na uslovnoj promenljivoj.

6. `bool empty()` - Odgovor da li je red blokiranih prazan (suprotna vrednost od `queue()`).

7. `int minrank()` - Prioritet niti koja bi bila deblokirana pozivom `signal()`.

Šta je razlika između uslovnih promenljivih i semafora?
Osim što uslovne promenljive imaju više operacija od semafora, za razliku od semafora uslovne promenljive nemaju "vredstnost", pa se stoga ponašanje operacija `wait()` i `signal()` razlikuju:

- Poziv `cond.wait()` **uvek** blokira pozivajuću nit, dok poziv `wait(sem)` može da blokira nit u zavisnosti od vrednosti semafora (naime, ako je vrednost semafora nula).
- Ukoliko je red blokiranih niti **prazan**, `cond.signal()` ne radi ništa, dok `signal(sem)` inkrementira vrednost semafora.
- Ukoliko je red blokiranih niti **neprazan**:
  Kod `SW` discipline, `cond.signal()` deblokira nit po FIFO (ili prioritetnom) redosledu i **predaje** kontekst deblokiranoj niti, dok `signal(sem)` deblokira nit po nasumičnom redosledu i **ne predaje** kontekst deblokiranoj niti.

  Kod `SC` discipline, ni semafor ni uslovna promenljiva **ne predaju** kontekst deblokiranoj niti, ali uslovna promenljiva i dalje deblokira niti po FIFO (ili prioritetnom) redosledu, dok semafor - koji je podrazumevano nepošten - deblokira niti po nasumičnom redosledu.

Korisna činjenica jeste da, po definiciji, uslovna promenljiva deblokira niti po FIFO (ili prioritetnom) redosledu, za razliku od semafora koji su podrazumevano nepošteni (ne-FIFO).

Šta je disciplina monitora?
Disciplina monitora definiše ponašanje operacije `cond.signal()`. Postoje tri discipline montiora, od kojih je treća specijalni slučaj druge:

1. `Signal-and-Continue (SC)`: Kada nit unutar monitorske procedure nit pozove `cond.signal()`, nit koja je pozvala operaciju nastavlja dalje sa izvršavanjem monitorske procedure, a signalizirana nit se stavlja u red čekanja za ulazak u monitor (`entry queue`).

2. `Signal-and-Wait (SW)`: Kada nit unutar monitorske procedure pozove `cond.signal()` nit koja je pozvala operaciju odlazi u red čekanja za ulazak u monitor, a signalizirana nit odmah ulazi u monitor.

3. `Signal-and-Urgent-Wait (SUW)`: Kada nit unutar monitorske procedure pozove `cond.signal()` nit koja je pozvala operaciju odlazi u poseban red "`urgent queue`", a signalizirana nit odmah ulazi u monitor. Kada signalizirana nit napusti monitor, sledeća nit koja ulazi u monitor je ona iz `urgent queue`-a umesto `entry queue`-a.

Šta je monitorska invarijanta?
Monitorska invarianta (eng. Invariant) je bulov izraz koji je inicijalno istinit (`true`) i koji monitor (korišćenjem uslovnih promenljivih) pokušava da očuva istinitim pri svakoj izmeni unutrašnjih promenljivih. Na primer, monitorska invarijanta za monitor koji implementira ograničen bafer kod Producer/Consumer-a je `(count >= 0 && count < B)`.

Kako se implementira monitor sa `Signal-and-Continue` disciplinom pomoću semafora?

```
class SC_Monitor {
    Sem mutex = 1;
    void enter() {
        wait(mutex);
    }
    void leave() {
        signal(mutex);
```

```
    }
    class cond {
        priority_queue<Sem&, int> qCond;
        void wait(int rank = 0) {
            Sem privs = 0; // private semaphore
            qCond.push(privs, rank);
            signal(mutex);
            wait(privs);
            wait(mutex);
        }
        void signal() {
            if (! qCond.empty()) {
                signal(qCond.pop());
            }
        }
        void signalAll() {
            while (! qCond.empty()) {
                signal(qCond.pop());
            }
        }
        bool queue() {
            return qCond.size() > 0;
        }
        bool empty() {
            return qCond.empty();
        }
        int minrank() {
            return qCond.peek().rank;
        }
    };
};
```

Kako se implementira monitor sa `Signal-and-Wait` disciplinom pomoću semafora?

```
class SW_Monitor {
    Sem mutex = 1;
    queue<Sem&> qWait;
    void enter() {
        wait(mutex);
    }
    void leave() {
        if (! qWait.empty()) {
            signal(qWait.pop());
        } else {
            signal(m_mutex);
        }
    }
    class cond {
        priority_queue<Sem&, int> qCond;
        void wait(int rank = 0) {
            Sem privs = 0; // private semaphore
            qCond.push(privs, rank);
            if (! qWait.empty()) {
                signal(qWait.pop());
            } else {
                signal(mutex);
            }
            wait(privs);
        }
        void signal() {
            if (! qCond.empty()) {
                Sem privs = 0; // private semaphore
                qWait.push(privs);
                signal(qCond.pop());
                wait(privs);
            }
        }
        bool queue() {
            return qCond.size() > 0;
        }
        bool empty() {
            return qCond.empty();
        }
```

```
        int minrank() {
            return qCond.peek().rank;
        }
    };
};
```

4.2 Simple synchronization

Zadatak Proces je pokrenuo dve niti sa beskonačnim petljama, gde prva nit ispisuje na konzoulu znak “a” dok druga nit ispisuje znak “b” (aaaaaaaaabbbbbbbbbbbaaaaaaaabbbb...). Koristeći semafore, potrebno je sinhronizovati niti tako da se na konzoli naizmenično ispisuju znakovi “a” i “b”. (abababababababab...).

Rešenje sa `Signal-and-Wait` disciplinom

```
#include <thread.h>
#include <stdlib.h>
#include <monitor.h>
#define B 10

class AB : SW_Monitor {
    char ch = 'a';
    cond condA;
    cond condB;
public:
    AB() : condA(this), condB(this) {
        // empty
    }
    void wait_a() {
        monitor {
            if (ch != 'a')
                condA.wait();
        }
    }
    void wait_b() {
        monitor {
            if (ch != 'b')
                condB.wait();
        }
    }
    void signal_a() {
        monitor {
            ch = 'a';
            condA.signal();
        }
    }
    void signal_b() {
        monitor {
            ch = 'b';
            condB.signal();
        }
    }
} ab;


void a() {
    while (true) {
        ab.wait_a();
        print("A");
        ab.signal_b();
        usleep(10);
    }
}

void b() {
    while (true) {
        ab.wait_b();
        print("B");
        ab.signal_a();
        usleep(10);
    }
}

int main() {
    Thread ta = createThread(a);
    Thread tb = createThread(b);
```

```
    join(ta);
    join(tb);
    return 0;
}
```

Rešenje sa Signal-and-Continue disciplinom

```
#include <thread.h>
#include <stdlib.h>
#include <monitor.h>
#define B 10

class AB : SC_Monitor {
    char ch = 'a';
    cond condA;
    cond condB;
public:
    AB() : condA(this), condB(this) {
        // empty
    }
    void wait_a() {
        monitor {
            while (ch != 'a')
                condA.wait();
        }
    }
    void wait_b() {
        monitor {
            while (ch != 'b')
                condB.wait();
        }
    }
    void signal_a() {
        monitor {
            ch = 'a';
            condA.signal();
        }
    }
    void signal_b() {
        monitor {
            ch = 'b';
            condB.signal();
        }
    }
} ab;


void a() {
    while (true) {
        ab.wait_a();
        print("A");
        ab.signal_b();
        usleep(10);
    }
}

void b() {
    while (true) {
        ab.wait_b();
        print("B");
        ab.signal_a();
        usleep(10);
    }
}

int main() {
    Thread ta = createThread(a);
    Thread tb = createThread(b);
    join(ta);
    join(tb);
    return 0;
}
```

4.3 Mutual exclusion

Zadatak Proces je pokrenuo dve niti, gde obe rade neki posao `work()` koji se mora obavljati sa ekskluzivnim pravom pristupa (npr. upis u fajl). Obezbediti ovaj vid sinhronizacije pomoću monitora.

Rešenje sa `Signal-and-Wait` disciplinom

```
#include <thread.h>
#include <monitor.h>
class CriticalSection : SW_Monitor {
    bool m_locked = false;
    cond m_delay;
public:
    CriticalSection() : m_delay(this) {

    }
    void enter_section() {
        monitor {
            if (m_locked)
                m_delay.wait();
            m_locked = true;
        }
    }

    void exit_section() {
        monitor {
            m_locked = false;
            m_delay.signal();
        }
    }

} cs;

char buffer[BUFSIZ];
void a() {
    while (true) {
        cs.enter_section();
        memset(buffer, 'a', BUFSIZ);
        cs.exit_section();
        usleep(100);
    }
}

void b() {
    while (true) {
        cs.enter_section();
        memset(buffer, 'b', BUFSIZ);
        cs.exit_section();
        usleep(10);
    }
}

int main() {
    Thread ta = createThread(a);
    Thread tb = createThread(b);
    join(ta);
    join(tb);
    return 0;
}
```

Rešenje sa Signal-and-Continue disciplinom

```
#include <thread.h>
#include <monitor.h>

class CriticalSection : SC_Monitor {
    bool m_locked = false;
    cond m_delay;
public:
    CriticalSection() : m_delay(this) {

    }
    void enter_section() {
        monitor {
            while (m_locked)
                m_delay.wait();
            m_locked = true;
        }
    }

    void exit_section() {
        monitor {
            m_locked = false;
            m_delay.signal();
        }
    }

} cs;

char buffer[BUFSIZ];
void a() {
    while (true) {
        cs.enter_section();
        memset(buffer, 'a', BUFSIZ);
        cs.exit_section();
    }
}

void b() {
    while (true) {
        cs.enter_section();
        memset(buffer, 'b', BUFSIZ);
        cs.exit_section();
    }
}

int main() {
    Thread ta = createThread(a);
    Thread tb = createThread(b);
    join(ta);
    join(tb);
    return 0;
}
```

4.4 Producer and consumer

Zadatak. Proces ima jedan deljeni ograničen kružni bafer veličine B i pokrenuo je dve niti: jednu nit `producer()` koja kreira podatke i smešta ih u bafer, i jednu nit `consumer()` koja dohvata podatke iz bafera i koristi ih. Ukoliko `producer` radi brže od `consumer`-a, nastupiće prekoračenje bafera (buffer overflow), dok ukoliko `consumer` radi brže od `producer`-a, `consumer` će početi da dohvata nevalidne ili nepostojaće podatke. Koristeći monitore, potrebno je obezbediti da se ove neželjene situacije ne dešavaju.

Rešenje sa `Signal-and-Wait` disciplinom

```
#include <thread.h>
#include <stdlib.h>
#include <monitor.h>
#define B 10

class Buffer_Monitor : SW_Monitor {
    int m_data[B];
    int m_head = 0;
    int m_tail = 0;
    int m_count = 0;
    cond m_full;
    cond m_empty;
public:
    Buffer_Monitor () : m_full(this), m_empty(this) {
        // empty
    }
    void put_data(int data) {
        monitor {
            if (m_count == B)
                m_full.wait();
            m_data[m_head] = data;
            m_head = (m_head + 1) % B;
            m_count += 1;
            m_empty.signal();
        }
    }
    int get_data() {
        monitor {
            if (m_count == 0)
                m_empty.wait();
            int result = m_data[m_tail];
            m_tail = (m_tail + 1) % B;
            m_count -= 1;
            m_full.signal();
            return result;
        }
    }
} buffer;

void producer() {
    while (true) {
        // Produce data ...
        int data = rand() % 1000;
        // Put data into buffer
        buffer.put_data(data);
    }
}

void consumer() {
    while (true) {
        // Get data from buffer
        int data = buffer.get_data();
        // Consume data ...
        print("%d ", data);
    }
}

int main() {
    Thread tp = createThread(producer);
    Thread tc = createThread(consumer);
```

```
    join(tp);
    join(tc);
    return 0;
}
```

Rešenje sa Signal-and-Continue disciplinom

```
#include <thread.h>
#include <stdlib.h>
#include <monitor.h>
#define B 10

class Buffer_Monitor : SC_Monitor {
    int m_data[B];
    int m_head = 0;
    int m_tail = 0;
    int m_count = 0;
    cond m_full;
    cond m_empty;
public:
    Buffer_Monitor () : m_full(this), m_empty(this) {
        // empty
    }
    void put_data(int data) {
        monitor {
            while (m_count == B)
                m_full.wait();
            m_data[m_head] = data;
            m_head = (m_head + 1) % B;
            m_count += 1;
            m_empty.signal();
        }
    }
    int get_data() {
        monitor {
            while (m_count == 0)
                m_empty.wait();
            int result = m_data[m_tail];
            m_tail = (m_tail + 1) % B;
            m_count -= 1;
            m_full.signal();
            return result;
        }
    }
} buffer;


void producer() {
    while (true) {
        // Produce data ...
        int data = rand() % 1000;
        // Put data into buffer
        buffer.put_data(data);
    }
}

void consumer() {
    while (true) {
        // Get data from buffer
        int data = buffer.get_data();
        // Consume data ...
        print("%d ", data);
    }
}

int main() {
    Thread tp = createThread(producer);
    Thread tc = createThread(consumer);
    join(tp);
    join(tc);
    return 0;
}
```

4.5 Barrier synchronization

Zadatak Proces je pokrenuo N niti koje treba da zajednički odrade neki posao `work1()`, i treba predju na posao `work2()` tek kada svih N niti završe posao `work1()`. Slično, potrebno je da svih N završe posao `work2()` pre nego što se predje nazad na posao `work1()`. Poslovi se moraju obavljati sa ekskluzivnim pravom pristupa. Implementirati ovakvo ponašanje niti pomoću monitora.

Rešenje sa `Signal-and-Wait` disciplinom

```
#include <thread.h>
#include <stdlib.h>
#include <monitor.h>
#define N 5
class CriticalSection;
class CS : public CriticalSection {
    ...
} cs;

class Barrier : SW_Monitor {
    int m_count = 0;
    cond m_barrier;
public:
    Barrier() : m_barrier(this) {
        // empty
    }
    void wait_barrier() {
        monitor {
            m_count += 1;
            if (m_count == N) {
                m_count = 0;
                m_barrier.signal();
            } else {
                m_barrier.wait();
                m_barrier.signal();
            }
        }
    }
} bar1, bar2;

void a(int id) {
    while (true) {
        cs.start_section();
        work1();
        cs.end_section();
        bar1.wait_barrier();

        cs.start_section();
        work2();
        cs.end_section();
        bar2.wait_barrier();
    }
}
int main() {
    Thread ta[N];
    for (int i = 0; i < N; i++)
        ta[i] = createThread(a, i);
    for (int i = 0; i < N; i++)
        join(ta[i]);
    return 0;
}
```

Rešenje sa `Signal-and-Continue` disciplinom

```
#include <thread.h>
#include <stdlib.h>
#include <monitor.h>
#define N 5
class CriticalSection;
class CS : public CriticalSection {
    ...
} cs;
```

```
class Barrier : SC_Monitor {
    int m_count = 0;
    cond m_barrier;
public:
    Barrier() : m_barrier(this) {
        // empty
    }
    void wait_barrier() {
        monitor {
            m_count += 1;
            if (m_count == B) {
                m_count = 0;
                m_barrier.signalAll();
            } else {
                m_barrier.wait();
            }
        }
    }
} bar1, bar2;

void a(int id) {
    while (true) {
        cs.start_section();
        work1();
        cs.end_section();
        bar1.wait_barrier();

        cs.start_section();
        work2();
        cs.end_section();
        bar2.wait_barrier();
    }
}
int main() {
    Thread ta[N];
    for (int i = 0; i < N; i++)
        ta[i] = createThread(a, i);
    for (int i = 0; i < N; i++)
        join(ta[i]);
    return 0;
}
```

4.6 Shortest job next

Zadatak Proces je pokrenuo N niti koje treba pristupe kritičnoj sekciji po prioritetnom redosledu. Naime, ukoliko postoje više niti koje istovremeno žele da udju u kritičnu sekciju, prva ulazi ona koja ima najmanje posla. Implementirati ovakvo ponašanje niti pomoću monitora.

Rešenje sa `Signal-and-Wait` disciplinom

```
#include <thread.h>
#include <stdlib.h>
#include <monitor.h>
#define N 5

class SJN : SW_Monitor {
    bool m_locked = false;
    cond m_delay;
public:
    SJN() : m_delay(this) {
        // empty
    }
    void enter_section(int prio) {
        monitor {
            if (m_locked)
                m_delay.wait(prio);
            m_locked = true;
        }
    }
    void exit_section() {
        monitor {
            m_locked = false;
            m_delay.signal();
        }
    }
} sjn;


void a(int id) {
    int prio = rand() % 1000; // nasumicni prioritet
    while (true) {
        sjn.enter_section(prio);
        work();
        sjn.exit_section();
    }
}

int main() {
    Thread ta[N];
    for (int i = 0; i < N; i++)
        ta[i] = createThread(a, i);
    for (int i = 0; i < N; i++)
        join(ta[i]);
    return 0;
}
```

Rešenje sa `Signal-and-Continue` disciplinom

```
#include <thread.h>
#include <stdlib.h>
#include <monitor.h>
#define N 5

class SJN : SC_Monitor {
    bool m_locked = false;
    cond m_delay;
public:
    SJN() : m_delay(this) {
        // empty
    }
    void enter_section(int prio) {
        monitor {
            if (m_locked || m_delay.queue())
```

```
                m_delay.wait(prio);
            m_locked = true;
        }
    }
    void exit_section() {
        monitor {
            m_locked = false;
            m_delay.signal();
        }
    }
} sjn;

void a(int id) {
    int prio = rand() % 1000; // nasumicni prioritet
    while (true) {
        sjn.enter_section(prio);
        work();
        sjn.exit_section();
    }
}

int main() {
    Thread ta[N];
    for (int i = 0; i < N; i++)
        ta[i] = createThread(a, i);
    for (int i = 0; i < N; i++)
        join(ta[i]);
    return 0;
}
```

4.7 Sleeping barber

Zadatak Proces je pokrenuo jednu nit koja simulira ponašanje uspavanog berberina i N niti koje simuliraju ponašanje mušterija koje dolaze kod berberina da se šišaju. Berberin spava dok mu ne dodje mušterija u berbernicu. Mušterija kada dodje u berbernicu treba da probudi berberina i sedne u red čekanja dok je berberin ne pozove na šišanje. Kada berberin završi sa šišanjem mušterije, mušterja plaća berberinu i odlazi. Ukoliko u berbernici nema mesta za sedenje, mušterija odlazi i vraća se nakon nekog vremena. Pomoću monitora, implementirati funkcije `void barber()` i `void customer(int id)` koje oponašaju berberina i mušterije respektivno.

Rešenje sa `Signal-and-Wait` disciplinom i sa `Signal-and-Continue` disciplinom

```
#include <thread.h>
#include <monitor.h>
#include <queue>
#define MAX_SEATS 5
#define SHOP_PRICE 400
#define N 10

class BarberShop : SW_Monitor /* SC_Monitor */ {
    std::queue<int> m_seats;
    int  m_money = 0;
    bool m_paid = false;
    cond m_waitForCustomer;
    cond m_waitForBarber;
    cond m_waitForPayment;
public:
    BarberShop() :
        m_waitForCustomer(this),
        m_waitForBarber(this),
        m_waitForPayment(this)
    {
        // empty
    }
    int barber_wait() {
        monitor {
            if (m_seats.empty())
                m_waitForCustomer.wait();
            int custId = m_seats.front();
            m_seats.pop();
            m_waitForBarber.signal();
            return custId;
        }
    }

    int barber_done(int id) {
        monitor {
            if (! m_paid)
                m_waitForPayment.wait();
            int payment = m_money;
            m_paid = false;
            m_money = 0;
            return payment;
        }
    }

    bool enter_shop(int id) {
        monitor {
            if (m_seats.size() < MAX_SEATS) {
                m_seats.push(id);
                m_waitForCustomer.signal();
                m_waitForBarber.wait();
                return true;
            }
            return false;
        }
    }

    void exit_shop(int money) {
```

```
        monitor {
            m_paid = true;
            m_money += money;
            m_waitForPayment.signal();
        }
    }
} shop;

void barber() {
    int money = 0;
    while (true) {
        int custId = shop.barber_wait();
        /* cut hair */
        money += shop.barber_done(custId);
    }
}

void customer(int id) {
    while (true) {
        if (shop.enter_shop(id)) {
            /* cut hair */
            shop.exit_shop(SHOP_PRICE);
        }
        usleep(50);
    }
}

int main() {
    Thread tb = createThread(barber);
    Thread tc[N];
    for (int i = 0; i < N; i++)
        tc[i] = createThread(customer, i);
    join(tb);
    join(tb);
    return 0;
}
```

4.8 Readers and writers

Zadatak Proces ima jedan otvoren deljeni fajl i pokrenuo je NR niti čitalaca koje izvršavaju funkciju `reader(int id)` i NW niti pisaca koje izvršavaju funkciju `writer(int id)`. Niti čitaoci mogu istovremeno čitati iz fajla, dok niti pisci moraju imati ekskluzivno pravo pristupa fajlu. Ne sme se dogoditi da čitalac i pisac pristupaju fajlu u isto vreme, i ne sme dogoditi da dva pisca upisuju u fajl u isto vreme.

Rešenje sa `Signal-and-Wait` disciplinom

```
#include <thread.h>
#include <monitor.h>
#include <queue>
#define NR 10
#define NW 5
enum {
    READER = 0,
    WRITER = 1
};
class RW : SW_Monitor {
    int m_readerCount = 0;
    int m_writerCount = 0;
    std::queue<int> m_delayType;
    cond m_delay;
public:
    RW() : m_delay(this) {
        // empty
    }
    void lock_reading() {
        monitor {
            if (m_writerCount > 0) {
                m_delayType.push(READER);
                m_delay.wait();
            }
            m_readerCount += 1;
            if (m_delay.queue() && m_delayType.front() == READER)  {
                m_delayType.pop();
                m_delay.signal();
            }
        }
    }
    void lock_writing() {
        monitor {
            if (m_writerCount > 0 || m_readerCount > 0) {
                m_delayType.push(WRITER);
                m_delay.wait();
            }
            m_writerCount += 1;
        }
    }
    void unlock() {
        monitor {
            if (m_writerCount > 0) {
                m_writerCount -= 1;
            } else if (m_readerCount > 0) {
                m_readerCount -= 1;
            }
            if (m_delay.queue()) {
                int shouldUndelay = (
                    (m_delayType.front() == READER && m_writerCount == 0)
                 || (m_delayType.front() == WRITER && m_writerCount == 0 && m_readerCount
    ));
                if (shouldUndelay) {
                    m_delayType.pop();
                    m_delay.signal();
                }
            }
        }
    }
} rw;
```

```
FILE* fShared = ...;
void reader(int id) {
    while (true) {
        rw.lock_reading();
        read(fShared, ...);
        rw.unlock();
    }
}

void writer(int id) {
    while (true) {
        rw.lock_writing();
        write(fShared, ...);
        rw.unlock();
    }
}

int main() {
    Thread tw[NW], tr[NR];
    for (int i = 0; i < NR; i++)
        tr[i] = createThread(reader, i);
    for (int i = 0; i < NW; i++)
        tw[i] = createThread(writer, i);
    for (int i = 0; i < NR; i++)
        join(tr[i]);
    for (int i = 0; i < NW; i++)
        join(tw[i]);
    return 0;
}
```

Rešenje sa `Signal-and-Continue` disciplinom

```
#include <thread.h>
#include <monitor.h>
#include <queue>
#define NR 10
#define NW 5
enum {
    READER = 0,
    WRITER = 1
};
class RW : SC_Monitor {
    int m_readerCount = 0;
    int m_writerCount = 0;
    std::queue<int> m_delayType;
    cond m_delay;
public:
    RW() : m_delay(this) {

    }
    void lock_reading() {
        monitor {
            if (m_writerCount > 0 || m_delay.queue()) {
                m_delayType.push(READER);
                m_delay.wait();
            } else {
                m_readerCount += 1;
            }
        }
    }
    void lock_writing() {
        monitor {
            if (m_writerCount > 0 || m_readerCount > 0 || m_delay.queue()) {
                m_delayType.push(WRITER);
                m_delay.wait();
            } else {
                m_writerCount += 1;
            }
        }
    }
    void unlock() {
        monitor {
            if (m_writerCount > 0) {
                m_writerCount -= 1;
```

```
            } else if (m_readerCount > 0) {
                m_readerCount -= 1;
            }
            if (m_delay.queue()) {
                if (m_delayType.front() == READER && m_writerCount == 0)  {
                    while (m_delayType.front() == READER)  {
                        m_readerCount += 1;
                        m_delayType.pop();
                        m_delay.signal();
                    }
                } else if (m_delayType.front() == WRITER && m_readerCount == 0 &&
    m_writerCount == 0) {
                    m_writerCount += 1;
                    m_delayType.pop();
                    m_delay.signal();
                }
            }
        }
    }
} rw;

FILE* fShared = ...;
void reader(int id) {
    while (true) {
        rw.lock_reading();
        read(fShared, ...);
        rw.unlock();
    }
}

void writer(int id) {
    while (true) {
        rw.lock_writing();
        write(fShared, ...);
        rw.unlock();
    }
}

int main() {
    Thread tw[NW], tr[NR];
    for (int i = 0; i < NR; i++)
        tr[i] = createThread(reader, i);
    for (int i = 0; i < NW; i++)
        tw[i] = createThread(writer, i);
    for (int i = 0; i < NR; i++)
        join(tr[i]);
    for (int i = 0; i < NW; i++)
        join(tw[i]);
    return 0;
}
```

4.9 Cigarette smokers

Zadatak Proces ima deljenu strukturu `table` i pokrenuo je jednu nit agenta `agent()` i tri niti pušača `smoker(MATCHES)`, `smoker(PAPER)` i `smoker(TOBACCO)`. Pušači imaju beskonačne zalihe jedne vrste predmeta, ali su im potrebene i druge dve vrste da bi pušili. Agent na sto nasumično stavlja neka dva predmeta i čeka dok ih neki pušač ne uzme. Pušač koji vidi da može da puši sa ta dva predmeta uzima ih sa stola i puši. Kada pušač završi pušenje, agentu signalizira da može opet da stavlja predmete na sto. Implementirati ponašanje pušača `void smoker(int item)` i agenta `agent()` pomoću monitora.

Rešenje sa `Signal-and-Wait` i sa `Signal-and-Continue` disciplinom

```
#include <thread.h>
#include <monitor.h>
#include <cstdlib>
#define ITEM_COUNT 3
#define OTHER_ITEM_1(item) ((item) == 0 ? ITEM_COUNT - 1 : ((item) - 1))
#define OTHER_ITEM_2(item) (((item) + 1) % ITEM_COUNT)
#define THIRD_ITEM(item1, item2) (1 - (((item1) - 1) + ((item2) - 1)))
enum {
    PAPER = 0,
    MATCHES = 1,
    TOBACCO = 2
};
class Table : SW_Monitor {
    bool m_items[ITEM_COUNT] = { false, false, false };
    cond m_waitForItems[ITEM_COUNT];
    bool m_ok = false;
    cond m_waitForOk;
public:
    Table() : m_waitForItems((cond[]){this, this, this}), m_waitForOk(this) {
        // empty
    }
    void put_items(int item1, int item2) {
        monitor {
            int third = THIRD_ITEM(item1, item2);
            m_items[item1] = true;
            m_items[item2] = true;
            m_waitForItems[third].signal();
        }
    }
    void get_items(int item1, int item2) {
        monitor {
            int third = THIRD_ITEM(item1, item2);
            if (! m_items[item1] || ! m_items[item2])
                m_waitForItems[third].wait();
            m_items[item1] = false;
            m_items[item2] = false;
        }
    }
    void wait_ok() {
        monitor {
            if (! m_ok)
                m_waitForOk.wait();
            m_ok = false;
        }
    }
    void signal_ok() {
        monitor {
            m_ok = true;
            m_waitForOk.signal();
        }
    }
} table;

void agent() {
    while (true) {
        int notItem = rand() % ITEM_COUNT;
        switch (notItem) {
            case MATCHES: {
                // produce items...
```

```
                // put items on the table
                table.put_items(PAPER, TOBACCO);
                table.wait_ok();
            } break;
            case PAPER: {
                // produce items...
                // put items on the table
                table.put_items(MATCHES, TOBACCO);
                table.wait_ok();
            } break;
            case TOBACCO: {
                // produce items...
                // put items on the table
                table.put_items(MATCHES, PAPER);
                table.wait_ok();
            } break;
        }
    }
}

void smoker(int hasItem) {
    while (true) {
        switch (hasItem) {
            case MATCHES: {
                table.get_items(PAPER, TOBACCO);
                // create a cigarette with the three items
                // smoke...
                table.signal_ok();
            } break;
            case PAPER: {
                table.get_items(MATCHES, TOBACCO);
                // create a cigarette with the three items
                // smoke...
                table.signal_ok();
            } break;
            case TOBACCO: {
                table.get_items(PAPER, MATCHES);
                // create a cigarette with the three items
                // smoke...
                table.signal_ok();
            } break;
        }
    }
}
int main() {
    Thread ta = createThread(agent);
    Thread ts1 = createThread(smoker, MATCHES);
    Thread ts2 = createThread(smoker, PAPER);
    Thread ts3 = createThread(smoker, TOBACCO);
    join(ta);
    join(ts1);
    join(ts2);
    join(ts3);
    return 0;
}
```

4.10 Dining philosophers

Zadatak Proces je pokrenuo 5 niti koje simuliraju ponašanje filozofa za stolom. Filozof neko vreme misli, uzima levu i desnu viljušku na stolu, jede ručak na tanjiru, vraća viljuške, i nastavlja da misli. Filozof ne može da uzme levu viljušku ukoliko ju je trenutno filozof levo koristi, i slično ne moze da uzme desnu viljušku ukoliko ju je trenutno filozof desno koristi. Ukoliko svih pet filozofa u isto vreme pokušaju da uzmu levu viljušku, ni jedan filozof neće moći da uzme desnu viljušku, i tada nastupa deadlock. Implementirati ponašanje filozofa pomoću monitora tako da se ovaj deadlock ne može dogoditi.

1. način

Rešenje gde obrćemo uzimanja viljušaka redosled parnim filozofa.

Rešenje sa `Signal-and-Wait` disciplinom

```
#include <thread.h>
#include <monitor.h>
#define N 5
#define LEFT(id) ((id) == 0 ? N - 1 : (id) - 1)
#define RIGHT(id) (((id) + 1) % N)

class Table : SW_Monitor {
    bool m_forkFree[N] = { true, true, true, true, true };
    cond m_waitForFork[N];
public:
    Table() : m_waitForFork {this, this, this, this, this} {

    }
    void acquire_fork(int which) {
        monitor {
            if (! m_forkFree[which])
                m_waitForFork[which].wait();
            m_forkFree[which] = false;
        }
    }

    void release_fork(int which) {
        monitor {
            m_forkFree[which] = true;
            m_waitForFork[which].signal();
        }
    }
} table;

void phil(int id) {
    int left = LEFT(id);
    int right = RIGHT(id);
    while (true) {
        // Think
        think();
        // Acquire forks
        if (id % 2 == 0) {
            table.acquire_fork(left);
            table.acquire_fork(right);
        } else {
            table.acquire_fork(right);
            table.acquire_fork(left);
        }
        // Eat
        eat();
        // Release forks
        table.release_fork(left);
        table.release_fork(right);
    }
}

int main() {
    Thread tp[N];
    for (int i = 0; i < N; i++)
        tp[i] = createThread(phil, i);
```

```
    for (int i = 0; i < N; i++)
        join(tp[i]);
    return 0;
}
```

Rešenje sa Signal-and-Continue disciplinom

```
#include <thread.h>
#include <monitor.h>
#define N 5
#define LEFT(id) ((id) == 0 ? N - 1 : (id) - 1)
#define RIGHT(id) (((id) + 1) % N)

class Table : SC_Monitor {
    bool m_forkFree[N] = { true, true, true, true, true };
    bool m_needFork[N] = { false, false, false, false, false};
    int  m_ticket = N - 1;
    cond m_waitForFork[N];
    cond m_waitForTicket;
public:
    Table()
        : m_waitForFork {this, this, this, this, this}
        , m_waitForTicket {this} {

    }
    void acquire_fork(int which) {
        monitor {
            if (! m_forkFree[which]) {
                m_needFork[which] = true;
                m_waitForFork[which].wait();
            } else {
                m_forkFree[which] = false;
            }
        }
    }
    void release_fork(int which) {
        monitor {
            m_forkFree[which] = true;
            if (m_needFork[which]) {
                m_forkFree[which] = false;
                m_needFork[which] = false;
                m_waitForFork[which].signal();
            }
        }
    }
    void acquire_ticket() {
        monitor {
            while (m_ticket == 0)
                m_waitForTicket.wait();
            m_ticket -= 1;
        }
    }
    void release_ticket() {
        monitor {
            m_ticket += 1;
            m_waitForTicket.signal();
        }
    }
} table;

void phil(int id) {
    int left = LEFT(id);
    int right = RIGHT(id);
    while (true) {
        // Think
        think();
        // Acquire forks
        table.acquire_ticket();
        table.acquire_fork(left);
        table.acquire_fork(right);
        // Eat
        eat();
        // Release forks
        table.release_fork(left);
```

```
        table.release_fork(right);
        table.release_ticket();
    }
}

int main() {
    Thread tp[N];
    for (int i = 0; i < N; i++)
        tp[i] = createThread(phil, i);
    for (int i = 0; i < N; i++)
        join(tp[i]);
    return 0;
}
```

2. način

Rešenje gde imamo "ticket" inicijalizovan na $N-1$, gde je N broj filozofa.

Rešenje sa `Signal-and-Wait` disciplinom

```
#include <thread.h>
#include <monitor.h>
#define N 5
#define LEFT(id) ((id) == 0 ? N - 1 : (id) - 1)
#define RIGHT(id) (((id) + 1) % N)

class Table : SW_Monitor {
    bool m_forkFree[N] = { true, true, true, true, true };
    bool m_needFork[N] = { false, false, false, false, false};
    int  m_ticket = N - 1;
    cond m_waitForFork[N];
    cond m_waitForTicket;
public:
    Table()
        : m_waitForFork {this, this, this, this, this}
        , m_waitForTicket {this} {

    }
    void acquire_fork(int which) {
        monitor {
            if (! m_forkFree[which])
                m_waitForFork[which].wait();
            m_forkFree[which] = false;
        }
    }
    void release_fork(int which) {
        monitor {
            m_forkFree[which] = true;
            m_waitForFork[which].signal();
        }
    }
    void acquire_ticket() {
        monitor {
            if (m_ticket == 0)
                m_waitForTicket.wait();
            m_ticket -= 1;
        }
    }
    void release_ticket() {
        monitor {
            m_ticket += 1;
            m_waitForTicket.signal();
        }
    }
} table;

void phil(int id) {
    int left = LEFT(id);
    int right = RIGHT(id);
    while (true) {
        // Think
        think();
        // Acquire forks
        table.acquire_ticket();
        table.acquire_fork(left);
        table.acquire_fork(right);
        // Eat
        eat();
        // Release forks
        table.release_fork(left);
        table.release_fork(right);
        table.release_ticket();
    }
}

int main() {
    Thread tp[N];
    for (int i = 0; i < N; i++)
```

```
        tp[i] = createThread(phil, i);
    for (int i = 0; i < N; i++)
        join(tp[i]);
    return 0;
}
```

Rešenje sa Signal-and-Continue disciplinom

```
#include <thread.h>
#include <monitor.h>
#define N 5
#define LEFT(id) ((id) == 0 ? N - 1 : (id) - 1)
#define RIGHT(id) (((id) + 1) % N)

class Table : SC_Monitor {
    bool m_forkFree[N] = { true, true, true, true, true };
    bool m_needFork[N] = { false, false, false, false, false};
    int  m_ticket = N - 1;
    cond m_waitForFork[N];
    cond m_waitForTicket;
public:
    Table()
        : m_waitForFork {this, this, this, this, this}
        , m_waitForTicket {this} {

    }
    void acquire_fork(int which) {
        monitor {
            if (! m_forkFree[which]) {
                m_needFork[which] = true;
                m_waitForFork[which].wait();
            } else {
                m_forkFree[which] = false;
            }
        }
    }
    void release_fork(int which) {
        monitor {
            m_forkFree[which] = true;
            if (m_needFork[which]) {
                m_forkFree[which] = false;
                m_needFork[which] = false;
                m_waitForFork[which].signal();
            }
        }
    }
    void acquire_ticket() {
        monitor {
            while (m_ticket == 0)
                m_waitForTicket.wait();
            m_ticket -= 1;
        }
    }
    void release_ticket() {
        monitor {
            m_ticket += 1;
            m_waitForTicket.signal();
        }
    }
} table;

void phil(int id) {
    int left = LEFT(id);
    int right = RIGHT(id);
    while (true) {
        // Think
        think();
        // Acquire forks
        table.acquire_ticket();
        table.acquire_fork(left);
        table.acquire_fork(right);
        // Eat
        eat();
        // Release forks
```

```
        table.release_fork(left);
        table.release_fork(right);
        table.release_ticket();
    }
}

int main() {
    Thread tp[N];
    for (int i = 0; i < N; i++)
        tp[i] = createThread(phil, i);
    for (int i = 0; i < N; i++)
        join(tp[i]);
    return 0;
}
```

3. način

Rešenje gde uzimamo obe viljuške u isto vreme i vraćamo obe viljuške u isto vreme, pritom budeći filozofa levo ili desno ukoliko oni ranije nisu uspeli.

Rešenje sa `Signal-and-Wait` disciplinom

```
#include <thread.h>
#include <monitor.h>
#define N 5
#define LEFT(id) ((id) == 0 ? N - 1 : (id) - 1)
#define RIGHT(id) (((id) + 1) % N)

class Table : SW_Monitor {
    bool m_forkFree[N] = { true, true, true, true, true };
    bool m_hungry[N] = { false, false, false, false, false };
    cond m_waitForForks[N];
public:
    Table() : m_waitForForks {this, this, this, this, this} {

    }
    void acquire_forks(int left, int right) {
        monitor {
            int id = RIGHT(left); // same as LEFT(right)
            if (! (m_forkFree[left] && m_forkFree[right])) {
                m_hungry[id] = true;
                m_waitForForks[id].wait();
            }
            m_hungry[id] = false;
            m_forkFree[left] = false;
            m_forkFree[right] = false;
        }
    }

    void release_forks(int left, int right) {
        monitor {
            m_forkFree[left] = true;
            m_forkFree[right] = true;
            if (m_hungry[left] && m_forkFree[LEFT(left)])  {
                m_waitForForks[left].signal();
            } else if (m_hungry[right] && m_forkFree[RIGHT(right)]) {
                m_waitForForks[right].signal();
            }
        }
    }
} table;

void phil(int id) {
    int left = LEFT(id);
    int right = RIGHT(id);
    while (true) {
        // Think
        think();
        // Acquire forks
        table.acquire_forks(left, right);
        // Eat
        eat();
        // Release forks
        table.release_forks(left, right);
    }
}

int main() {
    Thread tp[N];
    for (int i = 0; i < N; i++)
        tp[i] = createThread(phil, i);
    for (int i = 0; i < N; i++)
        join(tp[i]);
    return 0;
}
```

Rešenje sa `Signal-and-Continue` disciplinom

```
#include <thread.h>
#include <monitor.h>
```

```
#define N 5
#define LEFT(id) ((id) == 0 ? N - 1 : (id) - 1)
#define RIGHT(id) (((id) + 1) % N)

class Table : SC_Monitor {
    bool m_forkFree[N] = { true, true, true, true, true };
    bool m_hungry[N] = { false, false, false, false, false };
    cond m_waitForForks[N];
public:
    Table() : m_waitForForks {this, this, this, this, this} {

    }
    void acquire_forks(int left, int right) {
        monitor {
            int id = RIGHT(left); // same as LEFT(right)
            if (! (m_forkFree[left] && m_forkFree[right])) {
                m_hungry[id] = true;
                m_waitForForks[id].wait();
            } else {
                m_forkFree[left] = false;
                m_forkFree[right] = false;
            }
        }
    }

    void release_forks(int left, int right) {
        monitor {
            m_forkFree[left] = true;
            m_forkFree[right] = true;
            if (m_hungry[left] && m_forkFree[LEFT(left)])  {
                m_hungry[left] = false;
                m_forkFree[left] = false;
                m_forkFree[LEFT(left)] = false;
                m_waitForForks[left].signal();
            } else if (m_hungry[right] && m_forkFree[RIGHT(right)]) {
                m_hungry[right] = false;
                m_forkFree[right] = false;
                m_forkFree[RIGHT(right)] = false;
                m_waitForForks[right].signal();
            }
        }
    }
} table;

void phil(int id) {
    int left = LEFT(id);
    int right = RIGHT(id);
    while (true) {
        // Think
        think();
        // Acquire forks
        table.acquire_forks(left, right);
        // Eat
        eat();
        // Release forks
        table.release_forks(left, right);
    }
}

int main() {
    Thread tp[N];
    for (int i = 0; i < N; i++)
        tp[i] = createThread(phil, i);
    for (int i = 0; i < N; i++)
        join(tp[i]);
    return 0;
}
```