Конкурентно и дистрибуирано програмирање - скрипта за колоквијумски део

Иван Јаневски

Ова скрипта је отвореног кода (GNU Free Documentation License) чији се изворишни код написан у LaTeX-у може наћи на следећем линку
`https://github.com/yanevskiv/notes/tree/master/2022/kdp`

Садржај

1 Spinlock алгоритми

1.1 Увод

Шта су нити?
Нити (енг. Threads) су механизам који омогућава конкурентно или паралелно извршавање неких делова кода програма.

Конкурентно извршавање подразумева једну извршну јединицу (језгро или процесор) коју нити временски деле (т.ј. извшавају се по принципу ”час једна, час друга, ...”).

Паралелно извршавање подразумева више извршних јединица (језгара или процесора) где се нити физички извршавају у исто време. На ком језгру или процесору ће се одредјена нит извршавати и колико јединица времена одлучује `scheduler` оперативног система (preemptive multitasking). Уколико не постоји `scheduler` већ нити саме одлучују која ће се извршавати следеће (cooperative multitasking) такве нити се називају **корутинама** (енг. Coroutines).

Пример креирања две POSIX нити где једна нит исписује црвено слово “A” док друга исписује зелено слово “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;
}
```

Шта је атомичност?
Израз (т.ј. низ операција) је атомичан уколико се извршавање операција не може прекинути на пола пута т.ј. или се све операције изврше одједном, или се не изврши ни једна.

Шта је критична референца?
У неком изразу који једна нит извршава, **критична референца** је референца на променљиву у изразу чија се вредност мења од стране неке друге нити.

Шта је At-Most-Once Property (AMOP)?
Исказ доделе вредности $x = e$ задовољава **At-Most-Once Property** уколико је једна од следеће две ставке тачна:

1. Израз e садржи **највише једну** критичну референцу, и x се не чита од стране других нити.
2. Израз e не садржи **ни једну** критичну референцу, а x се може читати од стране других нити.

Уколико додела вредности задовољава AMOP, у погледу свих нити ће изгледати као да се додела извршава атомично.

У следећем примеру, обе доделе вредности $x = e_1$ и $y = e_2$ **задовољавају** AMOP јер e_1 и e_2 не садрже ни једну критичну референцу (задовољавају ставку 2):

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

У следећем примеру, обе доделе вредности $x = e_1$ и $y = e_2$ **задовољавају** AMOP. У $x = e_1$, e_1 садржи критичну референцу на y, али се x не чита од стране других нити (задовољава ставку 1). У $x = e_2$, e_2 не садржи ни једну критичну референцу (задовољава ставку 2).

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

У следећем примеру, $x = e_1$ и $y = e_2$ **не задовољавају** AMOP. У $x = e_1$, e_1 садржи критичну референцу на y, али x се чита од стране нити `b()` (не задовољава ставку 2.). У $y = e_2$, e_2 садржи критичну референцу на x, али y се чита од стране нити `a()` (не задовољава ставку 2.).

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

Шта је критична секција?
Критична секција је део кода који само једна нит сме да извршава у неком тренутку и може се имплементирати помоћу `mutual exclusion lock`-а (`mutex`-а).

Шта је lock?
Брава (енг. Lock) је механизам који омогућава некој нити да ограничи приступ неком делу кода или ресурсу осталим нитима. На пример, нит може закључати фајл са екслузивним правом приступа за упис у фајл (`exclusive lock`), или са дељеним правом приступа за читање из фајла (`shared lock`).

Шта је spinlock?
Spinlock је брава имплементирана помоћу упосленог чекања т.ј. петље које не ради ништа.

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

Шта је deadlock (livelock)?
Две или више нити се могу наћи у deadlock-у уколико блокирају једна друге тако да ни једна не може да настави са извршавањем. Livelock је сличан deadlock-у с тим што нити стално мењају уступају предност некој другој нити и на тај начин такође ни једна нит не наставља са извршавањем.

Шта је изгладњивање (starvation)?

Изгладњивање нити се јавља када нит предуго чека на неком услову. На пример, код Readers/Writers проблема се може јавити изгладњивање писаца уколико читаоци стално надолазе.

Шта је `volatile`?
`volatile` је кључна реч у језицима C и C++ која наговештава преводиоцу да не прави оптимизације са објектном назначеном са `volatile`. На пример, уколико желимо да радимо упослено чекање тако што ћемо имати једну `for` петљу која не ради ништа:

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

Преводилац ће оптимизовати код тако што уопште неће генерисати код за петљу:

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

Док уколико откоментаришемо `volatile`, код за петљу се генерише:

```
        .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
```

Правило је да све дељене променљиве треба означити са модификатором `volatile`.

Напомена: У језику C++ **није довољно** користити `volatile` за коректну синхронизацију помоћу spinlock-а већ за имплементацију неопходно користити конструкте из заглавља `<atomic>`, тако да све кодове наведене у наставку треба разумети само као псеудо код.

Шта је `skip()`?
Позвиом `skip()`, нит се одриче процесорског времена. Ради оптимизације, `skip()` је потребно позвати приликом упосленог чекања уколико услов није испуњен.

На пример, уколико је `scheduler` доделио процесор некој нити у времену од 150 ms, а она установи у року од 7 μs да услов за наставак није испуњен, нема потребе да се нит врти у петљи наредних 149,993 ms уколико постој нека друга нит која би се могла извршавати.

`skip()` је могуће имплементирати као `sched_yield()` или `usleep(0)`.

1.2 Test-and-set алгоритам

Код **Test-and-set** алгоритма постоји глобална брава `lock` која је иницијално креирана као откључана (`false`). Нит која жели да удје у критичну секцију атомично закључава глобалну браву и дохвата стару вредност браве. Атомично дохватање старе вредности глоблане и постављање на вредност "закључано" (`true`) се врши позивом специјалне инструкције `test_and_set(v)`. Уколико је стара вредност браве била "откључано", упослено чекање се прекида улази се у критичну секцију, с тим што је вредност браве сада постављена на "закључано". Уколико је стара вредност била "закључано", у браву се опет уписује вредност "закључано" и упослено чекање се наставља. Стални поновни упис вредности `true` у променљиву `lock` успорава рад система, јер врши честу инвалидацију кеширане вредности `lock` и често закључавање `cache` линије за упис. Овај проблем решава следећи алгоритам, **Test-and-test-and-set**. Када нит у критичној секцији заврши са критичном секцијом, вредност глобалне браве поставља на "откључано".

```
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 алгоритам

Код **Test-and-test-and-set** алгоритма вредност глобалне браве се најпре проверава из кеш меморије. Уколико је вредност браве "закључано", нит врши упослено чекање у унутрашњој нити где се вредност глобалне браве `lock` дохвата из кеш меморије. Уколико нит која је у критичној секцији напусти критичну секцију позивом `lock = false;`, све нити које су биле на упосленом чекању у унутрашњој петљи ће пробати да закључају браву и уђу у критичну секцију са `while (test_and_set(&lock))`. Једна од нити ће успети да уђе у критичну секцију, док ће се остале нити вратити на упослено чекање у унутрашњој петљи.

```
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 Питерсонов (Tie-Breaker) алгоритам за N процеса

Код **Питерсоновог (Tie-Breaker) алгоритма за N процеса**, нит која жели да уђе у критичну секцију најпре мора да прође кроз $N-1$ нивоа. Нит остаје блокирана у нивоу уколико постоји нека нит која је у једнаком или вишем нивоу (`inNivo[pId] >= inNivo[id]`) и уколико је дата нит последња која је ушла у дати ниво (`lastId[nivo] == id`). Нити се полако "филтрирају" тако што у сваком нивоу једна нит која се утркује мора да остане "заробљена" у том нивоу. Тако се у првом нивоу се може наћи највише N нити, у другом нивоу може се наћи највише $N-1$ нити, у трећем нивоу се може наћи највише $N-2$ нити, итд. У последњем нивоу се може наћи само једна нит и она улази у критичну секцију. Када нит заврши са критичном секцијом, ниво дате нити се ресетује на 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 алгоритам

Код **Ticket** алгоритма, нит која жели да удје у критичну секцију најпре атомично дохвата глобални број `ticket` у локалну промењиву `number` и инкрементира глобални `ticket` за вредност 1. Глобални `ticket` се атомично дохвата и инкрементира позивом специјалне инструкције `fetch_and_add(x, v)`. Нит потом чека док глобална вредност `next` ("следећи") не постане баш вредност претходно дохватаног броја `number`. Када нит заврши са критичном секцијом, следећој нити се допушта улазак у критичну секцију са `next += 1` (није неопходно да се вредност `next` инкрементира специјалном инструкцијом `fetch_and_add(x, v)` јер се критична секција `/*CRITICAL SECTION*/` наставља и на секцију `/*UNLOCK*/`). Тицкет алгоритам гарантује "фер" (FIFO) улазак у критичну секцију.

```
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 Андерсонов (ABQL) алгоритам

Код **Андерсоновог ABQL (Array-Based Queuing Lock)** алгоритма, свака нит чека на свој слот у неком низу чија је величина једнака броју нити. Први слот је означен као откључан, тако да прва нит која наиђе може да продје у критичну секцију, док су остали слотови закључани. Нит атомично дохвата `slot` у свој приватни `mySlot` и инкрементира глобални `slot`. Глобални `slot` се атомично дохвата и инкрементира позивом специјалне инструкције `fetch_and_add(x, v)`. Када нит заврши са ктритичном секцијом, свој слот ресетује, а нит заблокирану на следећем слоту пропушта. Као и Ticket алгоритам, Андерсонов алгоритам гарантује "фер" (FIFO) редослед уласка у критичну секцију.

```
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 алгоритам

Код **Lamport's bakery** алгоритма, свака нит има свој број `turn[id]` који се поставља на вредност већу од `turn`-а свих осталих нити. Нит најпре изражава жељу да уђе у критичну секцију постављањем вредности `turn[id] = 1` (вредност `turn[id] = 0` значи да нит уопште не жели још да уђе у критичну секцију), а потом гледа `turn`-ове осталих нити, тражи највећи `turn[j]` у низу, и ажурира свој `turn[id]` на вредност за 1 већу од надјеног максимума. Тражење максимума **не мора** да се обавља атомично, тако да може се догодити да више нити имају исту вредност својих `turn`-ова (`turn[j] == turn[id]`), и у том случају предност има нит са мањим `id`-ем. Нит са најмањим не-нултим `turn`-ом улази у критичну секцију. Када нит заврши са критичном секцијом нит свој `turn[id]` ресетује на вредност 0 што означава нитима које чекају да уђу у критичну секцију да је критична секција сада слободна.

```
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 алгоритам

Код **CLH (Craig, Landin, and Hagerste)** алгоритма постоји глобална брава постоји глобална брава `clh_lock` која је иницијално креирана као откључана. Нит која жели да удје у критичну секцију креира нову **закључану** браву и атомично је замењује са глобалном бравом `clh_lock` дохватајући стару вредност глобалне браве у `old_lock`. Стара вредност глобалне браве се дохвата и поставља на нову вредност позивом специјалне инструкције `get_and_set(x, v)`. Нит потом чека док стара брава (коју је неко други напарвио и треба да откључа) не постане откључана, улази у критичну секцију, и потом откључава браву `new_lock`, пропуштајући нит чија је то брава `old_lock`. Прва нит која наидје ће ући у критичну секцију док се остале нити блокирају на последњу браву која је замењена. Овим се нити виртуелно уланчавају и пропуштају по оном редоследу ком су пристигле. Као и Ticket и Андерсонов алгоритам, CLH алгоритам гарантује "фер" (FIFO) редослед уласка у критичну секцију.

```
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 Семафори

2.1 Увод

Шта је семафор?
Семафори (енг. Semaphore) су механизам који омогућава синхронизацију или међусобно искључивање између две или више нити. Семафор има свој интерну вредност (`int value`), ред блокираних нити на семафору (`queue<Thread*>`), и подржава следеће две операције:

- `void wait(s)` - Уколико је вредност семафора нула, нит се додаје у ред блокираних на семафору. Уколико је вредност семафора већа од нула, вредност семафора се декрементира за 1.
- `void signal(s)` - Уколико постоје заблокиране нити у реду блокираних, једна нит из реда блокираних се деблокира. Уколико је ред заблокираних празан, вредност семафора се инкрементира за 1.

Вредност овако дефинисаног семафора не може да оде испод нуле. Такође, у реду заблокираних не могу да постоје нити уколико је вредност семафора већа од нуле.

Поштен семафор је онај који позивом операције `signal(s)` деблокира нити по FIFO редоследу (нит која је раније позвала `wait(s)` ће се раније деблокирати).
Непоштен семафор је онај који позивом операције `signal(s)` деблокира нити по насумичном редоследу.

У овој скрипти (и на КДП-у) се подразумевају **непоштени семафори** - тако да је за имплементацију FIFO редоследа неопходно нпр. користити технику приватних семафора са `queue<int>`.

Пример коришћења POSIX семафора:

```
#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

Задатак Процес је покренуо две нити са бесконачним петљама, где прва нит исписује на конзоулу знак “a” док друга нит исписује знак “b” (aaaaaaaaabbbbbbbbbbaaaaaaaabbbb...). Користећи семафоре, потребно је синхронизовати нити тако да се на конзоли наизменично исписују знакови “a” и “b”. (abababababababab...).

Шта је расподељени бинарни семафор?
Расподељени бинарни семафор је бинарни семафор (семафор чија се вредност креће између 0 или 1) чији су позиви операције `wait(s)` расподељени унутар кода једне нити, а позиви операције `signal(s)` расподељени унутар кода друге нити. Овиме једна нит може да контролише ток друге нити. Наиме, нит која позива `signal(s)` контролише извршавање нити која позива `wait(s)`.

Решење:

```
#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

Задатак Процес је покренуо две нити, где прва нит попуњава дељени низ знакова знаком ‘a’, док друга нит попуњава низ знакова знаком ‘b’. Ове две нити ремете рад једна другој, тиме што се утркују (race condition) и “газе” податке друге нити својим подацима. Користећи семафоре, потребно је заштити низ знакова тако да једна нит не може да започне упис док друга нит не потпуно заврши.

Шта је mutex?
Mutex је ресурс који само једна нит може да “држи” у неком тренутку. Када нека нит “узме” mutex (“закључа” га) све остале нити које покушају исто ће се блокирати док нит која држи mutex не врати (“откључа”) исти. Mutex служи за имплементацију **критичне секције** у коду. Критична секција је део кода који искључиво једна нит може да извршава у неком тренутку. Пример коришћења mutex-а помоћу POSIX нити:

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

Mutex се лако имплементира помоћу бинарног семафора (семафори нису једини начин да се направи mutex!). Вредност семафора у сврси mutex-а се иницијализира на вредност `s=1`, а потом `wait(s)` означава “заузеће” mutex-а, док `signal(s)` означава “ослобађање” истог. нпр. помоћу POSIX семафора:

```
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);
```

Решење

```
#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

Задатак. Процес има један дељени ограничен кружни бафер величине B и покренуо је две нити: једну нит `producer()` која креира податке и смешта их у бафер, и једну нит `consumer()` која дохвата податке из бафера и користи их. Уколико `producer` ради брже од `consumer`-а, наступиће прекорачење бафера (buffer overflow), док уколико `consumer` ради брже од `producer`-а, `consumer` ће почети да дохвата невалидне или непостојаће податке. Користећи семафоре, потребно је обезбедити да се ове нежељене ситуације не дешавају.

Напомена за решење:
Mutex увек треба да стоји најуже могуће уз податке које штити, а синхронизација увек треба да буде око mutex-а. Дакле следећи код за `consumer`-а је грешка и узрокује deadlock:

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

Решење:

```
#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

Задатак Процес је покренуо N нити које треба да заједнички одраде неки посао `work1()`, и треба предју на посао `work2()` тек када свих N нити заврше посао `work1()`. Слично, потребно је да свих N заврше посао `work2()` пре него што се предје назад на посао `work1()`. Послови се морају обављати са ексклузивним правом приступа. Имплементирати овакво понашање нити помоћу семафора.

Шта је баријера?
Баријера (енг. Barrier) је тачка у коду где је потребно да дође одређен број нити пре него што нити заглављене на баријери не наставе са својим извршавањем. Уколико нека нит дође до баријере, број нити на баријери ће се повећати за један. Укилико је у том случају и даље недовољан број нити на баријери, дата нит ће се блокирати. У супротном, дата нит ће деблокирати све нити раније заблокиране на баријери и наставити даље са извшавањем. Пример коришћења баријере са POSIX нитима:

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

1. начин

Нит која уђе кроз прва врата `door1` позивом `wait(door1)` добија ексклузивно право приступа и може да обави посао `work1()`. Када нит заврши посао `work1()` инкрементира `counter` и, пре него што дође до других врата (т.j. `wait(door2)`), проверава која врата треба отворити следеће: Уколико је `count < N`, пушта се још једна нит кроз `door1` позивом `signal(door1)`. Уколико је `count == N`, цоунтер се ресетује на 0 како би се могао користити у другој секцији, и једна нит се пушта кроз друга врата позивом `signal(door2)` у другу секцију, То може бити управо та нит која је позвала `signal(door2)` и тек треба да дође до `wait(door2)`, или нека од $N-1$ нити већ заблокиране на `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. начин.

Можемо направити структуру `struct Barrier` која имплементира баријеру помоћу семафора, и онда инстанцирати две баријере: `bar1` и `bar2`. За разлику од "врата" у претходном примеру (`door1` и `door2`) која су уједно нам обезбеђивала, и синхронизацију, и медјусобно искључивање, баријере нам омогућавају само синхронизацију, а медјусобно искључивање обезбедјујемо сами помоћу **mutex**-а.

```
#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

Задатак Процес је покренуо N нити које треба приступе критичној секцији по приоритетном редоследу. Наиме, уколико постоје више нити које истовремено желе да уђу у критичну секцију, прва улази она која има најмање посла. Потребно је помоћу семафора имплементирати методу `sjn_wait(id, prio)` (нижа вредност аргумента `prio` значи виши приоритет) која се позива пре уласка у критичну секцију, и методу `sjn_signal()` која се позива након изласка из критичне секције.

Шта је техника приватних семафора?
Приватни семафор неке нити је семафор иницијализован на вредност 0 који нит може да користи да се вештачки заблокира и сачека док је нека друга нит не деблокира. Пре блокирања на приватном семафору неопходно је ослободити `mutex` уколико се држи (иначе настаје deadlock!) и неопходно је на неки начин запамтити `id` нити (или показивач на приватни семафор) како би се касније тај приватни семафор сигнализирао, и тиме заблокирана нит деблокирала.

Шта је техника прослеђивања штафетне палице (passing the baton)?
То је техника синхронизације где се - уместо пуштања mutex-а - сигнализира нека блокирана нит (нпр. блокирана на приватном семафору). Тиме деблокирана нит "добија" mutex од нити која је држала mutex и деблокирана нит може користити податке које су заштићене под mutex-ом. Нит потом може да деблокира неку другу нит и на исти начин јој достави mutex, а може и једноставно да ослободи mutex. Уколико не постоји ни једна нит којој mutex може да се проследи, неопходно је ослободити mutex. Mutex представља штафетну палицу која се прослеђује између нити.

1. начин - са прослеђивањем штафетне палице
Ово решење личи на имплементацију `Signal-and-Wait` монитора.

```
#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. начин - без прослеђивања штафетне палице

Ово решење личи на имплементацију `Signal-and-Continue` монитора.

```
#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

Задатак Процес је покренуо једну нит која симулира понашање успаваног берберина и N нити које симулирају понашање муштерија које долазе код берберина да се шишају. Берберин спава док му не дође муштерија у берберницу. Муштерија када дође у берберницу треба да пробуди берберина и седне у ред чекања док је берберин не позове на шишање. Када берберин заврши са шишањем муштерије, муштерја плаћа берберину и одлази. Уколико у берберници нема места за седење, муштерија одлази и враћа се након неког времена. Помоћу семафора имплементирати методе `void barber()` и `void customer(int id)` које респективно опонашају берберина и муштерије.

Решење

```
#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

Задатак Процес има један отворен дељени фајл и покренуо је NR нити читалаца које извршавају функцију `reader(int id)` и NW нити писаца које извршавају функцију `writer(int id)`. Нити читаоци могу истовремено читати из фајла, док нити писци морају имати ексклузивно право приступа. Не сме се догодити да читалац и писац приступају фајлу у исто време, и не сме се догодити да два писца уписују у фајл у исто време.

Шта је rwlock?
Readers/Writers lock је брава која штити неки ресурс, и која може да се закључа на два одвојена начина: за екслузивно право приступа (`exclusive (writer) lock`) или за дељено право прситупа (`shared (reader) lock`). У овом примеру ћемо имплементирати `rw_lock(...)` помоћу семафора и техником прослеђивања штафетне палице, и на тај начин решити проблем читаоца и писаца.

Пример rwlock-a са `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);
```

Пример rwlock-a са POSIX нитима:

```
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);
```

Полурешење (изгладњивање писаца)

```
#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;
}
```

Решење (FIFO улазак)

```
#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

Задатак Процес има дељену структуру `table` и покренуо је једну нит агента `agent()` и три нити пушача `smoker(MATCHES)`, `smoker(PAPER)` и `smoker(TOBACCO)`. Пушачи имају бесконачне залихе једне врсте предмета, али су им потребене и друге две врсте да би пушили. Агент на сто насумично ставља нека два предмета и чека док их неки пушач не узме. Пушач који види да може да пуши са та два предмета узима их са стола и пуши. Када пушач заврши пушење, агенту сигнализира да може опет да ставља предмете на сто. Имплементирати понашање пушача `void smoker(int item)` и агента `agent()` помоћу семафора.

Решење

```
#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. додатно решење у случају да `agent()` не ради сигнализацију

Ако агент никако не сигнализира пушаче, пушачи морају да извршавају `busy wait` и повремено проверавају да ли се на столу налазе предмети који су потребни. С обзиром да више нити могу истовремено да приступају столу, неопходан је 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. додатно решење - у случају да `agent()` сигнализира један семафор

Рецимо да агент када стави два предмета на сто сигнализира семафор оног предмета који **није** ставио на сто. Пушач тада треба да чека на семафор предмета који му **не треба** јер се тада налазе на столу налазе предмети који му требају. С обзиром да у овом решењу само једна нит може приступати столу у неком тренутку, mutex није потребан али није грешка искористи га.

```
#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. додатно решење - у случају да `agent()` сигнализира два семафора

Рецимо да агент када стави два предмета на сто сигнализира два семафора и то баш оних предмета које је и ставио на сто. У том случају, да би сви сигнали били ухваћени, сваки пушач и даље мора да чека на семафор једног предмета и тој оног предмета који им **не треба**. Два пушача који ухвате сигнале од агента који треба да се договоре да пробуде оног трећег јер је трећи пушач тај који може да узме предмете са стола и пуши. Mutex није неопходан за `table[]` али је неопходан за `agree[]` који пушачи користе да се договарају на поменут начин.

```
#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. додатно решење - у случају да `agent()` сигнализира три семафора.

Рецимо да агент сигнализира семафоре свих предмета када год стави нешто на сто. У том случају, пушач треба да чека на један од семафора и само провери да ли су предмети који су њему потребни на столу.

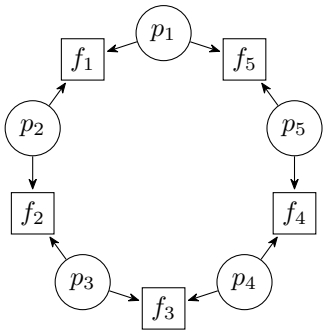
```
#include <thread.h>
#include <sem.h>
#include <stdlib.h>
#define ITEM_COUNT  3
#define THIRD_ITEM(first, second) (1 - (((first) - 1) + ((second) -1)))
#define OTHER_ITEM_1(item) ((item) == 0 ? ITEM_COUNT - 1 : (item) - 1)
#define OTHER_ITEM_2(item) (((item) + 1) % ITEM_COUNT)
enum {
    MATCHES = 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;
}
```

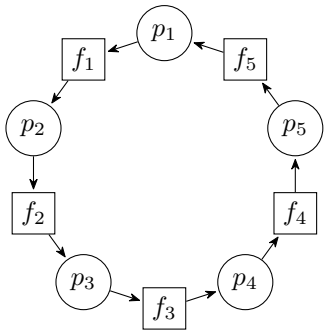
2.10 Dining philosophers

Задатак Процес је покренуо 5 нити које симулирају понашање филозофа за столом. Филозоф неко време мисли, узима леву и десну виљушку на столу, једе ручак на тањиру, враћа виљушке, и наставља да мисли. Филозоф не може да узме леву виљушку уколико јој је тренутно филозоф лево користи, и слично не мозе да узме десну виљушку уколико јој је тренутно филозоф десно користи. Уколико свих пет филозофа у исто време покушају да узму леву виљушку, ни један филозоф неће моћи да узме десну виљушку, и тада наступа деадлоцк. Имплементирати понашање филозофа помоћу семафора тако да се овај деадлоцк не може догодити.

Шта је граф алокације ресурса?
Граф алокације ресурса (енг. Resource allocation graph (RAG)) је усмерен граф у коме чворови представљају нити или ресурсе, а гране повезују нити и ресурсе. Уколико је грана усмерена од чвора нити ка чвору ресурса (нит → ресурс), дата нит може у неком тренутку заузети тај ресурс. Уколико је нит усмерена од чвора ресурса ка чвору нити (нит ← ресурс), дата нит је заузела ресурс и тренутно га држи. Уколико се у графу појави **петља**, настаје deadlock. Пример графа алокације ресурса:



(а) Граф филозофа



(б) Deadlock филозофа

1. начин
Можемо обрнути редослед узимања виљушака једног филозофа (или парних филозофа) и тиме отклонити могућност појаве петље у графу алокације ресурса.

```
#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. начин

Можемо направити упослено чекање и заузети виљушке када постану слободне (**Напомена:** Упослено чекање је неефикасно и када су на располагању семафори, обично је лоше правити упослено чекање уколико оно није неопходно)

```
#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. начин

Можемо имати семафор `ticket` иницијализован на вредност $N-1$, уколико је N број филозофа. Тако последњи филозоф не може да захтева виљушке уколико $N-1$ филозофа пре захтевало виљшке, па се не може јавити петља.

```
#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. начин

Можемо користити приватне семафоре филозофа на који ће се филозоф блокирати уколико не може да узме виљушке. Филозоф мора да најави да није успео да узме виљушке (преко `hungry[]`) како би филозоф који ослобађа виљушке знао да ли треба да покуша да деблокира филозофа лево или десно поред себе.

```
#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 Региони

3.1 Увод

Шта је условни критични регион?
Условни критични региони пружају могућност за међусобно искључивање између нити коришћењем исказа `region (s) {...}` , као и могућност да нит унутар региона привремено напусти критични регион док се не испуни неки неопходан услов помоћу `await(...)` исказа.

Критични региони су везани за неку структуру *s* над којом може да се позове `region (s) { ... }` исказ. То је овој скрипти било која структура која субкласира виртуелну класу `Region`. Унутар себе, структура `Region` садржи mutex који се закључава приликом сваког уласка у регион и откључава приликом сваког изласка из региона.

Након субкласирања структуре `Region`, у структури се могу додати додатни подаци чије вредности, по правилу, **није дозвољено** читати или модификовати ван регионског исказа. Такође, није смислено позивати `await(...)` ван регионског исказа.

У језику Јава, `synchronized (this) {...}` представља пример региона над објектром `this`. Исказ `await(...)` се може имплементирати као `while (! ...)` петља која позива `Object.wait()`. Уколико услов постане испуњен, некако се мора позвати `notify()` што може урадити нпр. нит која успешно излази из регионског блока.

```
// 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
```

А следећи код је пример имплементације региона помоћу POSIX нити:

```
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

Задатак Процес је покренуо две нити са бесконачним петљама, где прва нит исписује на конзоулу знак “a” док друга нит исписује знак “b” (aaaaaaaaabbbbbbbbbbaaaaaaaabbbb...). Користећи условне критичне регионе, потребно је синхронизовати нити тако да се на конзоли наизменично исписују знакови “a” и “b”. (abababababababab...).

Решење

```
#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

Задатак Процес је покренуо две нити, где обе раде неки посао `work()` који се мора обављати са ексклузивним правом приступа (нпр. упис у фајл). Обезбедити овај вид синхронизације помоћу условних критичних региона.

1. начин

Можемо унутар региона имати промењљиву `lock` коју ћемо сетовати на `true` када улазимо у критичну или на `false` уколико излазимо из критичне секције. Нећемо улазити у критичну секцију уколико је она тренутно закључана.

```
#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. начин

С обзиром да критични регион и сам већ имплементира критичну секцију, можемо једноставно ставити `work()` унутар регионског блока.

```
#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

Задатак. Процес има један дељени ограничен кружни бафер величине B и покренуо је две нити: једну нит `producer()` која креира податке и смешта их у бафер, и једну нит `consumer()` која дохвата податке из бафера и користи их. Уколико `producer` ради брже од `consumer`-а, наступиће прекорачење бафера (buffer overflow), док уколико `consumer` ради брже од `producer`-а, `consumer` ће почети да дохвата невалидне или непостојаће податке. Користећи условне критичне регионе, потребно је обезбедити да се ове нежељене ситуације не дешавају.

Решење

```
#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

Задатак Процес је покренуо N нити које треба да заједнички одраде неки посао `work1()` и треба предју на посао `work2()` тек када свих N нити заврше посао `work1()`. Слично, потребно је да свих N заврше посао `work2()` пре него што се предје назад на посао `work1()`. Послови се морају обављати са екскључивним правом приступа. Имплементирати овакво понашање помоћу условних критичних региона.

Решење

```
#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

Задатак Процес је покренуо N нити које треба приступе критичној секцији по приоритетном редоследу. Наиме, уколико постоје више нити које истовремено желе да уђу у критичну секцију, прва улази она која има најмање посла. Имплементирати овакво понашање нити помоћу условних критичних региона.

Решење

Уколико је критична секција заузета, можемо ставити себе у приоритетни ред и сачекати док следећи у реду не буде баш наш `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

Задатак Процес је покренуо једну нит која симулира понашање успаваног берберина и N нити које симулирају понашање муштерија које долазе код берберина да се шишају. Берберин спава док му не дође муштерија у берберницу. Муштерија када дође у берберницу треба да пробуди берберина и седне у ред чекања док је берберин не позове на шишање. Када берберин заврши са шишањем муштерије, муштерја плаћа берберину и одлази. Уколико у берберници нема места за седење, муштерија одлази и враћа се након неког времена. Помоћу условних критичних региона, имплементирати функције `void barber()` и `void customer(int id)` које опонашају берберина и муштерије респективно.

Решење

```
#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

Задатак Процес има један отворен дељени фајл и покренуо је NR нити читалаца које извршавају функцију `reader(int id)` и NW нити писаца које извршавају функцију `writer(int id)`. Нити читаоци могу истовремено читати из фајла, док нити писци морају имати ексклузивно право приступа фајлу. Не сме се догодити да читалац и писац приступају фајлу у исто време, и не сме догодити да два писца уписују у фајл у исто време.

Полурешење (изгладњивање писаца)

```
#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;
}
```

Решење (FIFO улазак)

```
#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

Задатак Процес има дељену структуру `table` и покренуо је једну нит агента `agent()` и три нити пушача `smoker(MATCHES)`, `smoker(PAPER)` и `smoker(TOBACCO)`. Пушачи имају бесконачне залихе једне врсте предмета, али су им потребене и друге две врсте да би пушили. Агент на сто насумично ставља нека два предмета и чека док их неки пушач не узме. Пушач који види да може да пуши са та два предмета узима их са стола и пуши. Када пушач заврши пушење, агенту сигнализира да може опет да ставља предмете на сто. Имплементирати понашање пушача `void smoker(int item)` и агента `agent()` помоћу условних критичних региона.

Решење

```
#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

Задатак Процес је покренуо 5 нити које симулирају понашање филозофа за столом. Филозоф неко време мисли, узима леву и десну виљушку на столу, једе ручак на тањиру, враћа виљушке, и наставља да мисли. Филозоф не може да узме леву виљушку уколико јој је тренутно филозоф лево користи, и слично не мозе да узме десну виљушку уколико јој је тренутно филозоф десно користи. Уколико свих пет филозофа у исто време покушају да узму леву виљушку, ни један филозоф неће моћи да узме десну виљушку, и тада наступа деадлоцк. Имплементирати понашање филозофа помоћу условних критичних региона тако да се овај деадлоцк не може догодити.

Решење

Најприродније решење филозофа са условним критичним регионима јесте атомично узети обе виљушке када обе виљушке постану слободне.

```
#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 Монитори

4.1 Увод

Шта је монитор?
Монитор је конструкт који је налик класи (у овој скрипти је то било која класа која субкласира апстраткну класу `Monitor`) који омогућава да се код за синхронизацију и медјусобно искључивање апстрахује ван логике коју извршава нит. Код семафора и региона, код за медјусобно искључивање и синхронизацију је био измешан са кодом за логику. Код монитора, нити искључиво позивају **мониторске процедуре** и не врше сами синхронизацију, већ монитор обавља то монитор за њих. Монитор садржи интерни `mutex`, ред чекања за улазак у монитор, и може садржати условне и променљиве `cond`.

Шта су мониторске процедуре?
Мониторске процедуре су методе монитора у које се улази синхроно т.ј. закључавањем монитора приликом уласка, и откључавањем монитора приликом изласка. Када нека нит удје у мониторску процедуру њеним позивом, ни једна друга нит не може ући ни у једну мониторску процедуру док првобитна нит не напусти монитор долажењем до краја мониторске процедуре, позивом `cond.wait()`, или позивом `cond.signal()` код `Signal-and-Wait` дисциплине.

Са семафорима, мониторска процедура се може имплемнтирати позивом `wait(mutex)` на почетку процедуре, и `signal(mutex)` на крају процедуре. Са регионима, мониторска процедура се може имплементирати као метода која обавија свој код са `region (this) {...}`. У овој скрипти, свака мониторска процедура је означена блоком `monitor {...}` што има исто значење као `region (this) {...}`, или `synchronized (this) {...}` у језику Јава.

Пример мониторских процедура у језику Јава:

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

Шта су условне променљиве (`cond`)?
Условне променљиве омогућавају привремено напуштање монитора и суспендовање нити унутар мониторске процедуре док се не испуни неки услов. Услове променљиве садрже (приоритетни) ред блокираних нити (`priority_queue<int, Thread*>`) и подржавају следећих 6 (или 7) операција:

1. `void wait()` - Нит која позове ову операцију излази монитора и додаје се у ред блокираних на условној променљивој. Деблокирање ће се вршити по FIFO редоследу блокирања.

2. `void wait(int rank)` - Нит која позове ову операцију излази монитора и додаје се у ред блокираних на условној променљивој. Деблокирање ће се вршити по приоритетном редоследу блокирања (мања вредност значи већи приоритет).

3. `void signal()` - Уколико нема блокираних нити на условној променљивој, ова операција не

ради ништа. Уколико има блокирних нити, једна нит се деблокира по редоследу наведеном горе. Која нит добија контекст зависи од дисциплине монитора (`SW` или `SC`).

4. `void signalAll()` - Све нити блокиране на условној променљивој се деблокирају. Уколико је ред блокираних празан, ова операција не ради ништа. **Напомена:** Ова операција постоји само код монитора са `Signal-and-Continue` дисциплином!

5. `bool queue()` - Одговор да ли постоје нити блокиране на условној променљивој.

6. `bool empty()` - Одговор да ли је ред блокираних празан (супротна вредност од `queue()`).

7. `int minrank()` - Приоритет нити која би била деблокирана позивом `signal()`.

Шта је разлика између условних променљивих и семафора?
Осим што условне променљиве имају више операција од семафора, за разлику од семафора условне променљиве немају "вредстност", па се стога понашање операција `wait()` и `signal()` разликују:

- Позив `cond.wait()` **увек** блокира позивајућу нит, док позив `wait(sem)` може да блокира нит у зависности од вредности семафора (наиме, ако је вредност семафора нула).
- Уколико је ред блокираних нити **празан**, `cond.signal()` не ради ништа, док `signal(sem)` инкрементира вредност семафора.
- Уколико је ред блокираних нити **непразан**:
  Код `SW` дисциплине, `cond.signal()` деблокира нит по FIFO (или приоритетном) редоследу и **предаје** контекст деблокираној нити, док `signal(sem)` деблокира нит по насумичном редоследу и **не предаје** контекст деблокираној нити.

  Код `SC` дисциплине, ни семафор ни условна променљива **не предају** контекст деблокираној нити, али условна променљива и даље деблокира нити по FIFO (или приоритетном) редоследу, док семафор - који је подразумевано непоштен - деблокира нити по насумичном редоследу.

Корисна чињеница јесте да, по дефиницији, условна променљива деблокира нити по FIFO (или приоритетном) редоследу, за разлику од семафора који су подразумевано непоштени (не-FIFO).

Шта је дисциплина монитора?
Дисциплина монитора дефинише понашање операције `cond.signal()`. Постоје три дисциплине монтиора, од којих је трећа специјални случај друге:

1. `Signal-and-Continue (SC)`: Када нит унутар мониторске процедуре нит позове `cond.signal()`, нит која је позвала операцију наставља даље са извршавањем мониторске процедуре, а сигнализирана нит се ставља у ред чекања за улазак у монитор (`entry queue`).

2. `Signal-and-Wait (SW)`: Када нит унутар мониторске процедуре позове `cond.signal()` нит која је позвала операцију одлази у ред чекања за улазак у монитор, а сигнализирана нит одмах улази у монитор.

3. `Signal-and-Urgent-Wait (SUW)`: Када нит унутар мониторске процедуре позове `cond.signal()` нит која је позвала операцију одлази у посебан ред "`urgent queue`", а сигнализирана нит одмах улази у монитор. Када сигнализирана нит напусти монитор, следећа нит која улази у монитор је она из `urgent queue`-а уместо `entry queue`-а.

Шта је мониторска инваријанта?
Мониторска инваријанта (енг. Invariant) је булов израз који је иницијално истинит (`true`) и који монитор (коришћењем условних променљивих) покушава да очува истинитим при свакој измени унутрашњих променљивих. На пример, мониторска инваријанта за монитор који имплементира ограничен бафер код Producer/Consumer-а је `(count >= 0 && count < B)`.

Како се имплементира монитор са `Signal-and-Continue` дисциплином помоћу семафора?

```
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;
        }
    };
};
```

Како се имплементира монитор са Signal-and-Wait дисциплином помоћу семафора?

```
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

Задатак Процес је покренуо две нити са бесконачним петљама, где прва нит исписује на конзоулу знак “a” док друга нит исписује знак “b” (aaaaaaaaabbbbbbbbbbaaaaaaaabbbb...). Користећи семафоре, потребно је синхронизовати нити тако да се на конзоли наизменично исписују знакови “a” и “b”. (abababababababab...).

Решење са Signal-and-Wait дисциплином

```
#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;
}
```

Решење са Signal-and-Continue дисциплином

```
#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

Задатак Процес је покренуо две нити, где обе раде неки посао `work()` који се мора обављати са ексклузивним правом приступа (нпр. упис у фајл). Обезбедити овај вид синхронизације помоћу монитора.

Решење са `Signal-and-Wait` дисциплином

```
#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;
}
```

Решење са Signal-and-Continue дисциплином

```
#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

Задатак. Процес има један дељени ограничен кружни бафер величине B и покренуо је две нити: једну нит `producer()` која креира податке и смешта их у бафер, и једну нит `consumer()` која дохвата податке из бафера и користи их. Уколико `producer` ради брже од `consumer`-а, наступиће прекорачење бафера (buffer overflow), док уколико `consumer` ради брже од `producer`-а, `consumer` ће почети да дохвата невалидне или непостојаће податке. Користећи мониторе, потребно је обезбедити да се ове нежељене ситуације не дешавају.

Решење са `Signal-and-Wait` дисциплином

```
#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;
}
```

Решење са Signal-and-Continue дисциплином

```
#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

Задатак Процес је покренуо N нити које треба да заједнички одраде неки посао `work1()`, и треба предју на посао `work2()` тек када свих N нити заврше посао `work1()`. Слично, потребно је да свих N заврше посао `work2()` пре него што се предје назад на посао `work1()`. Послови се морају обављати са ексклузивним правом приступа. Имплементирати овакво понашање нити помоћу монитора.

Решење са Signal-and-Wait дисциплином

```
#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;
}
```

Решење са Signal-and-Continue дисциплином

```
#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

Задатак Процес је покренуо N нити које треба приступе критичној секцији по приоритетном редоследу. Наиме, уколико постоје више нити које истовремено желе да уђу у критичну секцију, прва улази она која има најмање посла. Имплементирати овакво понашање нити помоћу монитора.

Решење са Signal-and-Wait дисциплином

```
#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;
}
```

Решење са Signal-and-Continue дисциплином

```
#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

Задатак Процес је покренуо једну нит која симулира понашање успаваног берберина и N нити које симулирају понашање муштерија које долазе код берберина да се шишају. Берберин спава док му не дође муштерија у берберницу. Муштерија када дође у берберницу треба да пробуди берберина и седне у ред чекања док је берберин не позове на шишање. Када берберин заврши са шишањем муштерије, муштерја плаћа берберину и одлази. Уколико у берберници нема места за седење, муштерија одлази и враћа се након неког времена. Помоћу монитора, имплементирати функције `void barber()` и `void customer(int id)` које опонашају берберина и муштерије респективно.

Решење са `Signal-and-Wait` дисциплином и са `Signal-and-Continue` дисциплином

```
#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

Задатак Процес има један отворен дељени фајл и покренуо је NR нити читалаца које извршавају функцију `reader(int id)` и NW нити писаца које извршавају функцију `writer(int id)`. Нити читаоци могу истовремено читати из фајла, док нити писци морају имати ексклузивно право приступа фајлу. Не сме се догодити да читалац и писац приступају фајлу у исто време, и не сме догодити да два писца уписују у фајл у исто време.

Решење са Signal-and-Wait дисциплином

```
#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;
}
```

Решење са Signal-and-Continue дисциплином

```
#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

Задатак Процес има дељену структуру `table` и покренуо је једну нит агента `agent()` и три нити пушача `smoker(MATCHES)`, `smoker(PAPER)` и `smoker(TOBACCO)`. Пушачи имају бесконачне залихе једне врсте предмета, али су им потребене и друге две врсте да би пушили. Агент на сто насумично ставља нека два предмета и чека док их неки пушач не узме. Пушач који види да може да пуши са та два предмета узима их са стола и пуши. Када пушач заврши пушење, агенту сигнализира да може опет да ставља предмете на сто. Имплементирати понашање пушача `void smoker(int item)` и агента `agent()` помоћу монитора.

Решење са Signal-and-Wait и са Signal-and-Continue дисциплином

```
#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

Задатак Процес је покренуо 5 нити које симулирају понашање филозофа за столом. Филозоф неко време мисли, узима леву и десну виљушку на столу, једе ручак на тањиру, враћа виљушке, и наставља да мисли. Филозоф не може да узме леву виљушку уколико јој је тренутно филозоф лево користи, и слично не мозе да узме десну виљушку уколико јој је тренутно филозоф десно користи. Уколико свих пет филозофа у исто време покушају да узму леву виљушку, ни један филозоф неће моћи да узме десну виљушку, и тада наступа деадлоцк. Имплементирати понашање филозофа помоћу монитора тако да се овај деадлоцк не може догодити.

1. начин

Решење где обрћемо узимања виљушака редослед парним филозофа.

Решење са Signal-and-Wait дисциплином

```
#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;
}
```

Решење са Signal-and-Continue дисциплином

```
#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. начин

Решење где имамо "ticket" иницијализован на $N-1$, где је N број филозофа.

Решење са `Signal-and-Wait` дисциплином

```
#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;
}
```

Решење са Signal-and-Continue дисциплином

```
#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. начин

Решење где узимамо обе виљушке у исто време и враћамо обе виљушке у исто време, притом будећи филозофа лево или десно уколико они раније нису успели.

Решење са `Signal-and-Wait` дисциплином

```
#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;
}
```

Решење са `Signal-and-Continue` дисциплином

```
#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;
}
```