Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий Фізико-технічний інститут

ОПЕРАЦІЙНІ СИСТЕМИ

Комп'ютерний практикум
Робота №8

Виконав:
студент групи ФІ-12
Завалій Олександр
Перевірив:
Кірієнко О.В.

Київ-2023

Робота №8.
Засоби синхронізації потоків

Мета:

Оволодіння практичними навичками розроблення багатопотокових програм з підтримкою засобів синхронізаці.

Варіант №5

Зміст індивідуального завдання:

1. Розминка. Стандартна задача виробник-споживач.
   Задача була розглянута на лекції. Також детально розглянута в рекомендованих книжках [1, 5]. Розробіть програму, що демонструє рішення цієї задачі за допомогою семафорів. Для цього напишіть:

   - функції виробника і споживача (наприклад, як на лекції, або як у Шеховцові, але так, щоби працювало);
   - функції створення і споживання об'єктів (рекомендується "створювати" рядки тексту шляхом зчитування їх з файлу, хоча можливі й інші варіанти за вибором викладача або за вашою фантазією, наприклад розрахунки геш-функцій sha2 з рядків рандомних символів, а "споживати" їх шляхом роздрукування на екрані з додатковою інформацією такою як ідентифікатор потоку і мітка часу, причому і там, і там для моделювання складного характеру реального життя виробників і споживачів можна додавати рандомні затримки);
   - функцію main(), що створює потоки-виробники і потоки-споживачі, при цьому треба передбачити введення з клавіатури або як параметри командного рядка кількості записів у буфері, кількості виробників і кількості споживачів для досліджень їх роботи;
   - обов'язково передбачити коректне завершення усього цього господарства.

   Продемонструвати викладачеві як воно працює (не менше двох виробників і двох споживачів) і код, що ви написали.

2. Продовження розминки. Теж саме, але не на семафорах, а на м'ютексі і умовних змінних.
   Модифікуйте програму п. 1 так, щоби використовувати м'ютекс і умовну змінну.

3. Продовження розминки для тих, хто шукає пригод. Взаємне блокування.
   Модифікуйте програму п. 1 так, щоби викликати взаємне блокування. Для цього поміняйте місцями семафори. Переконайтесь у факті взаємного блокування і отримайте задоволення.

4. Індивідуальне завдання.
   Варіант 2 в. Філософи, що обідають 3
   Все тотожнє варіанту 2 б, за винятком типу очікування. Якщо філософ не може взяти виделку (першу або другу), він має покласти на стіл першу виделку, якщо він її вже захопив, і заснути. Реалізувати можна за допомогою додаткових м'ютексів і умовних змінних. Відповідно, коли філософ звільнює виделки, він має сигналізувати про це. Чи часто філософи падають в обморок? А якщо час його сну додавати до часу "голодування"?

Task I

Розминка. Стандартна задача виробник-споживач.
Задача була розглянута на лекції. Також детально розглянута в рекомендованих книжках [1, 5]. Розробіть програму, що демонструє рішення цієї задачі за допомогою семафорів.

```
alex@Oleksandr:~/labs/lab_8$ cat semaphore.cpp
#include <iostream>
#include <thread>
#include <chrono>
#include <semaphore.h>
using namespace std;

const int BUFFER_SIZE = 10;
sem_t empt, full;
int buffer[BUFFER_SIZE];
int in = 0, out = 0;

void produce(int id, int num_items) {
    for (int i = 0; i < num_items; i++) {
        sem_wait(&empt);
        buffer[in] = i;
        cout << "Producer " << id << " produced item " << i << endl;
        in = (in + 1) % BUFFER_SIZE;
        sem_post(&full);
        this_thread::sleep_for(chrono::milliseconds(500));
    }
}
void consume(int id, int num_items) {
    for (int i = 0; i < num_items; i++) {
        sem_wait(&full);
        int item = buffer[out];
        cout << "Consumer " << id << " consumed item " << item << endl;
        out = (out + 1) % BUFFER_SIZE;
        sem_post(&empt);
        this_thread::sleep_for(chrono::milliseconds(1000));
    }
}
int main() {
    sem_init(&empt, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    thread prod1(produce, 1, 5);
    thread prod2(produce, 2, 5);
    thread cons1(consume, 1, 5);
    thread cons2(consume, 2, 5);

    cout << "Processes were created!" << endl;

    prod1.join();
    prod2.join();
    cons1.join();
    cons2.join();

    sem_destroy(&empt);
    sem_destroy(&full);

    return 0;
alex@Oleksandr:~/labs/lab_8$
```

```
alex@Oleksandr:~/labs/lab_8$ g++ -pthread -o semaphore semaphore.cpp
alex@Oleksandr:~/labs/lab_8$ ./semaphore
Producer 1 produced item 0
Processes were created!
Consumer 1 consumed item 0
Producer 2 produced item 0
Consumer 2 consumed item 0
Producer 2 produced item 1
Producer 1 produced item 1
Consumer 2 consumed item 1
Consumer 1 consumed item 1
Producer 2 produced item 2
Producer 1 produced item 2
Producer 2 produced item 3
Producer 1 produced item 3
Consumer 2 consumed item 2
Consumer 1 consumed item 2
Producer 2 produced item 4
Producer 1 produced item 4
Consumer 2 consumed item 3
Consumer 1 consumed item 3
Consumer 2 consumed item 4
Consumer 1 consumed item 4
alex@Oleksandr:~/labs/lab_8$
```

Task II

Продовження розминки. Теж саме, але не на семафорах, а на м'ютексі і умовних змінних.
Модифікуйте програму п. 1 так, щоби використовувати м'ютекс і умовну змінну.

```
alex@Oleksandr:~/labs/lab_8$ cat mutex.cpp
#include <iostream>
#include <thread>
#include <chrono>
#include <mutex>
#include <condition_variable>
#include <unistd.h>

using namespace std;

const int BUFFER_SIZE = 10;
int buffer[BUFFER_SIZE];
int in = 0, out = 0;

mutex mtx;
condition_variable cvEmpty, cvFull;

void produce(int id, int num_items) {
    for (int i = 0; i < num_items; i++) {
        unique_lock<mutex> lock(mtx);
        cvEmpty.wait(lock, [] { return ((in + 1) % BUFFER_SIZE) != out; });
        buffer[in] = i;
        cout << "Producer " << id << " produced item " << i << endl;
        in = (in + 1) % BUFFER_SIZE;
        lock.unlock();
        cvFull.notify_all();
        this_thread::sleep_for(chrono::milliseconds(500));
        cout << "------------------------------------------------------------\n";
        sleep(2);
    }
}

void consume(int id, int num_items) {
    for (int i = 0; i < num_items; i++) {
        unique_lock<mutex> lock(mtx);
        cvFull.wait(lock, [] { return in != out; });
        int item = buffer[out];
        cout << "Consumer " << id << " consumed item " << item << endl;
        out = (out + 1) % BUFFER_SIZE;
        lock.unlock();
        cvEmpty.notify_all();
        this_thread::sleep_for(chrono::milliseconds(1000));
    }
}

int main() {
    thread prod1(produce, 1, 5);
    thread prod2(produce, 2, 5);
    thread cons1(consume, 1, 5);
    thread cons2(consume, 2, 5);

    cout << "Processes were created!" << endl;

    prod1.join();
    prod2.join();
    cons1.join();
    cons2.join();

    return 0;
alex@Oleksandr:~/labs/lab_8$
```

```
alex@Oleksandr:~/labs/lab_8$ g++ -pthread -o mutex mutex.cpp
alex@Oleksandr:~/labs/lab_8$ ./mutex
Processes were created!
Producer 2 produced item 0
Producer 1 produced item 0
Consumer 2 consumed item 0
Consumer 1 consumed item 0
------------------------------------------------------------
------------------------------------------------------------
Producer 2 produced item 1
Producer 1 produced item 1
Consumer 1 consumed item 1
Consumer 2 consumed item 1
------------------------------------------------------------
------------------------------------------------------------
Producer 2 produced item 2
Consumer 1 consumed item 2
Producer 1 produced item 2
Consumer 2 consumed item 2
------------------------------------------------------------
------------------------------------------------------------
Producer 2 produced item 3
Consumer 1 consumed item 3
Producer 1 produced item 3
Consumer 2 consumed item 3
------------------------------------------------------------
------------------------------------------------------------
Producer 2 produced item 4
Consumer 1 consumed item 4
Producer 1 produced item 4
Consumer 2 consumed item 4
------------------------------------------------------------
------------------------------------------------------------
alex@Oleksandr:~/labs/lab_8$
```

Task III

Продовження розминки для тих, хто шукає пригод. Взаємне блокування. Модифікуйте програму п. 1 так, щоби викликати взаємне блокування. Для цього поміняйте місцями семафори. Переконайтесь у факті взаємного блокування і отримайте задоволення.

```
alex@Oleksandr:~/labs/lab_8$ cat mblocking.cpp
#include <iostream>
#include <thread>
#include <chrono>
#include <semaphore.h>

using namespace std;

const int BUFFER_SIZE = 10;
sem_t empt, full;
int buffer[BUFFER_SIZE];
int in = 0, out = 0;

void produce(int id, int num_items) {
    cout << "Producer\n";
    for (int i = 0; i < num_items; i++) {
        sem_wait(&full);
        sem_wait(&empt);
        buffer[in] = i;
        cout << "Producer " << id << " produced item " << i << endl;
        in = (in + 1) % BUFFER_SIZE;
        sem_post(&empt);
        sem_post(&full);
        this_thread::sleep_for(chrono::milliseconds(500));
    }
}

void consume(int id, int num_items) {
    cout << "Consumer\n";
    for (int i = 0; i < num_items; i++) {
        sem_wait(&empt);
        sem_wait(&full);
        int item = buffer[out];
        cout << "Consumer " << id << " consumed item " << item << endl;
        out = (out + 1) % BUFFER_SIZE;
        sem_post(&full);
        sem_post(&empt);
        this_thread::sleep_for(chrono::milliseconds(1000));
    }
}

int main() {
    sem_init(&empt, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    thread prod1(produce, 1, 5);
    thread prod2(produce, 2, 5);
    thread cons1(consume, 1, 5);
    thread cons2(consume, 2, 5);

    cout << "Processes were created!" << endl;

    prod1.join();
    prod2.join();
    cons1.join();
    cons2.join();

    sem_destroy(&empt);
    sem_destroy(&full);

    return 0;
}alex@Oleksandr:~/labs/lab_8$
```

```
alex@Oleksandr:~/labs/lab_8$ g++ -pthread -o mblocking mblocking.cpp
alex@Oleksandr:~/labs/lab_8$ ./mblocking
Processes were created!
Producer
Producer
Consumer
Consumer
```

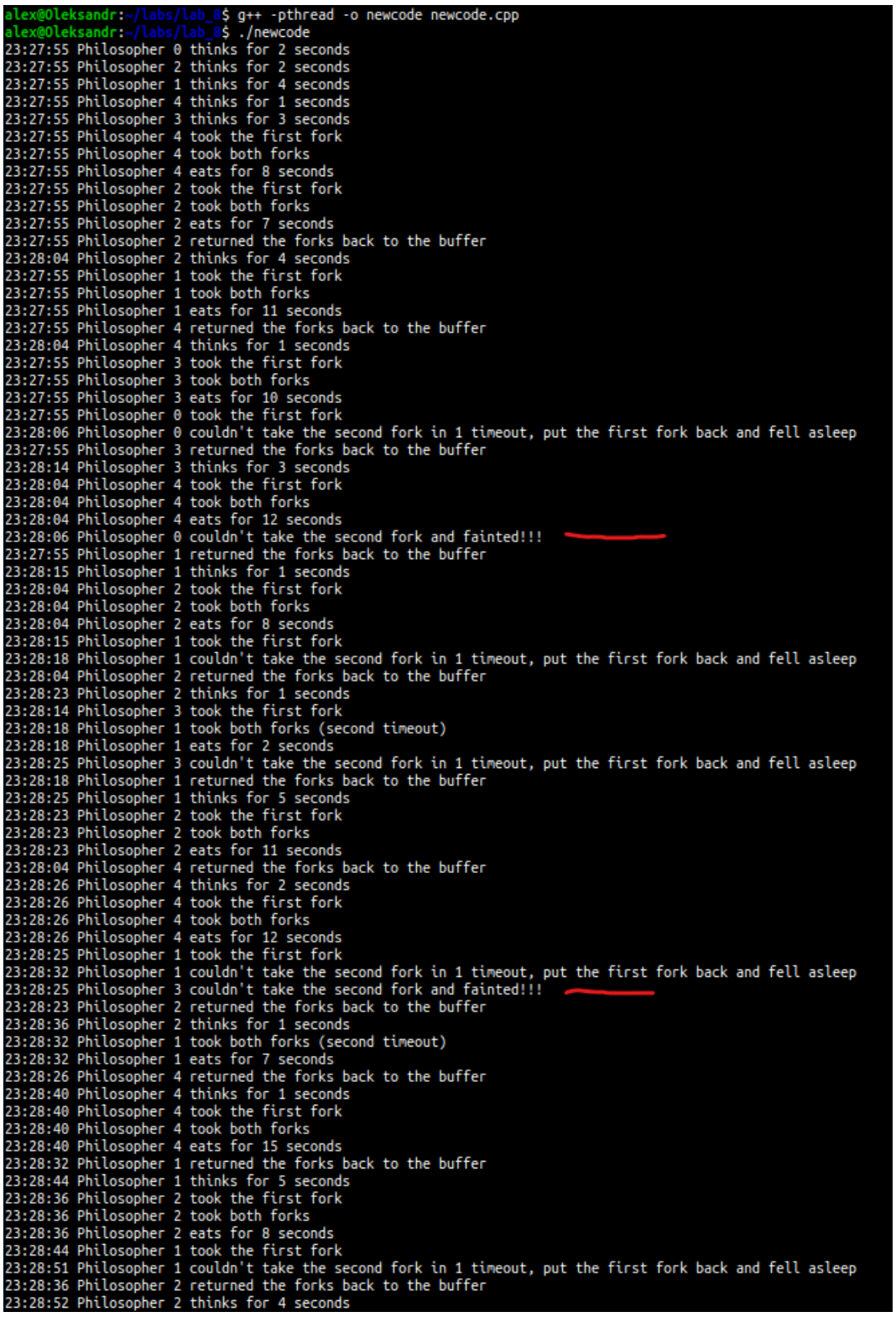
Task IV

Індивідуальне завдання.
Варіант 2 в. Філософи, що обідають 3
Все тотожнє варіанту 2 б, за винятком типу очікування. Якщо філософ не може взяти виделку (першу або другу), він має покласти на стіл першу виделку, якщо він її вже захопив, і заснути. Реалізувати можна за допомогою додаткових м'ютексів і умовних змінних. Відповідно, коли філософ звільнює виделки, він має сигналізувати про це. Чи часто філософи падають в обморок? А якщо час його сну додавати до часу "голодування"?

```
alex@Oleksandr:~/labs/lab_8$ cat newcode.cpp
#include <iostream>
#include <pthread.h>
#include <unistd.h>
#include <cstdlib>
#include <ctime>
#include <semaphore.h>

using namespace std;

const int NUM_PHILOSOPHERS = 5;
const int NUM_SPAGHETTI = 5;

sem_t forks_buffer;
sem_t forks[NUM_PHILOSOPHERS];

int spaghetti[NUM_PHILOSOPHERS];

void *philosopher(void *arg) {
    int id = *((int *)arg);

    while (spaghetti[id] > 0) {
        time_t currtime = time(nullptr);
        tm* timecurr = localtime(&currtime);
        char tbuffer[80];
        strftime(tbuffer, sizeof(tbuffer), "%H:%M:%S", timecurr);

        // A philosopher thinks
        int thinking_time = rand() % 5 + 1;
        cout << tbuffer << " Philosopher " << id << " thinks for " << thinking_time << " seconds" << endl;
        sleep(thinking_time);

        // The philosopher tries to take two forks
        sem_wait(&forks_buffer);
        sem_wait(&forks[id]);
        cout << tbuffer << " Philosopher " << id << " took the first fork" << endl;
```

```
        struct timespec timeout;
        clock_gettime(CLOCK_REALTIME, &timeout);
        timeout.tv_sec += 2;

        int result = sem_timedwait(&forks[(id + 1) % NUM_PHILOSOPHERS], &timeout);

        if (result == 0) {
            // The philosopher took both forks
            cout << tbuffer << " Philosopher " << id << " took both forks" << endl;

            // The philosopher eats
            int eating_time = rand() % 15 + 1;
            cout << tbuffer << " Philosopher " << id << " eats for " << eating_time << " seconds" << endl;
            sleep(eating_time);

            // The philosopher turns the forks back
            sem_post(&forks_buffer);
            sem_post(&forks[id]);
            sem_post(&forks[(id + 1) % NUM_PHILOSOPHERS]);
            cout << tbuffer << " Philosopher " << id << " returned the forks back to the buffer" << endl;

            // Reducing the amount of spaghetti
            spaghetti[id]--;
        } else {
            time_t currtime = time(nullptr);
            tm* timecurr = localtime(&currtime);
            char tbuffer[80];
            strftime(tbuffer, sizeof(tbuffer), "%H:%M:%S", timecurr);

            // The philosopher failed to take the second fork in the first timeout
            cout << tbuffer << " Philosopher " << id << " couldn't take the second fork in 1 timeout, put the first fork back and fell asleep" << endl;
            sem_post(&forks_buffer);
            sem_post(&forks[id]);
            sleep(5);

            clock_gettime(CLOCK_REALTIME, &timeout);
            timeout.tv_sec += 3;

            result = sem_timedwait(&forks[(id + 1) % NUM_PHILOSOPHERS], &timeout);

            if (result == 0) {
                // The philosopher took both forks
                cout << tbuffer << " Philosopher " << id << " took both forks (second timeout)" << endl;

                // The philosopher eats
                int eating_time = rand() % 15 + 1;
                cout << tbuffer << " Philosopher " << id << " eats for " << eating_time << " seconds" << endl;
                sleep(eating_time);

                // The philosopher turns the forks back
                sem_post(&forks_buffer);
                sem_post(&forks[id]);
                sem_post(&forks[(id + 1) % NUM_PHILOSOPHERS]);
                cout << tbuffer << " Philosopher " << id << " returned the forks back to the buffer" << endl;

                // Reducing the amount of spaghetti
                spaghetti[id]--;
            } else {
                // The philosopher failed to take the second fork in the second timeout
                cout << tbuffer << " Philosopher " << id << " couldn't take the second fork and fainted!!!" << endl;
                sem_post(&forks_buffer);
                sem_post(&forks[id]);
                pthread_exit(nullptr);
            }
        }
    }
    time_t currtime = time(nullptr);
    tm* timecurr = localtime(&currtime);
    char tbuffer[80];
    strftime(tbuffer, sizeof(tbuffer), "%H:%M:%S", timecurr);

    cout << tbuffer << " Philosopher " << id << " finished work" << endl;

    pthread_exit(nullptr);
}
```

```
int main() {
    srand(time(nullptr));
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_ids[NUM_PHILOSOPHERS];
    sem_init(&forks_buffer, 0, NUM_PHILOSOPHERS - 1);
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_init(&forks[i], 0, 1);
    }
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        spaghetti[i] = NUM_SPAGHETTI;
    }
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopher_ids[i] = i;
        pthread_create(&philosophers[i], nullptr, philosopher, &philosopher_ids[i]);
    }
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], nullptr);
    }
    sem_destroy(&forks_buffer);
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_destroy(&forks[i]);
    }
    return 0;
}
alex@Oleksandr:~/labs/lab_8$
```



На скріншоті видно, що на п'ять філософів двоє впали в обморок. Також перший процес міг завершитись три рази, але йому пощастило більше. Тобто філософи часто падають в обморок, оскільки веделок всього п'ять. Отже виходить одночасно їсти може лише двоє, а інші в цей час думають. Також це буде залежить від часу за який філософи їдять. Оскільки це число випадкове, то відповідно два філософи можуть їсти довше, аніж інші три думають. Якщо час сну додавати до часу "голодування то вийде що деякі філософи, в теорії, можуть весь час "голодувати". Адже веделки заберає той потік, що перший отримує доступ до буферу. Тому якщо потоків багато, а веделок мало, можемо виникнути ситуації коли потоки будуть чекати доки інші завершать своє виконання і тільки після цього отримають доступ.

Висновки

Оволодіння практичними навичками розроблення багатопотокових програм з підтримкою засобів синхронізації набирує актуальності. Оскільки у сучасному світі, де швидкість і ефективність програмного забезпечення мають ключове значення, уміння розробляти багатопотокові програми знаходить широке застосування.

Потокам та процесам часто потрібно взаємодіяти один з одним, наприклад, один процес може передавати дані іншому процесу, або декілька процесів можуть обробляти дані із загального файлу. У всіх цих випадках виникає проблема синхронізації процесів, яка може вирішуватися припиненням і активізацією процесів, організацією черг, блокуванням і звільненням ресурсів.

Якщо розбиратись детальніше, то механізмами синхронізації є засоби операційної системи, які допомагають розв'язувати основне завдання синхронізації — забезпечувати координацію потоків, які працюють зі спільно використовуваними даними. Якщо такі засоби — це мінімальні блоки для побудови багатопотокових програм, їх називають синхронізаційними примітивами.

Їх поділяють на такі основні категорії:

1. універсальні, низького рівня, які можна використовувати різними способами (семафори);

2. прості, низького рівня, кожен з яких пристосований до розв'язання тільки однієї задачі (м'ютекси та умовні змінні);

3. універсальні високого рівня, виражені через прості; до цієї групи належить концепція монітора, яка може бути виражена через м'ютекси та умовні змінні;

4. високого рівня, пристосовані до розв'язання конкретної синхронізаційної задачі (блокування читання-записування і бар'єри).