Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

ОТЧЁТ

к лабораторной работе №4

на тему

**ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ (ПОТОКОВ): ВЗАИМНОЕ**

**ИСКЛЮЧЕНИЕ И СИНХРОНИЗАЦИЯ**

Студент: Ющук И.А.

Преподаватель: Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Формулировка задачи 3](#_Toc177507035)

[2 Описание функций программы 4](#_Toc177507036)

[2.1 Создание и запуск процессов «обедающих философов» 4](#_Toc177507037)

[2.2 Алгоритм действий «обедающих философов» 4](#_Toc177507038)

[Заключение 6](#_Toc177507040)

[Список использованных источников 7](#_Toc177507041)

[Приложение А (обязательное) Исходный код программы 8](#_Toc177507042)

1 ФОРМУЛИРОВКА ЗАДАЧИ

Реализация модели взаимодействия конкурирующих параллельных процессов (потоков) «обедающие философы» с возможностью ее параметризации и наглядного (не обязательно графического) представления результатов. Проверка и демонстрация различных подходов к решению задачи обеспечения корректного взаимодействия. Изменяемые параметры модели: количество «философов»; выбор логики разрешения конфликта; интенсивность обращений к критическому ресурсу, длительность использования ресурса и т.п. (характеристики случайных величин при моделировании); величина тайм-аутов и др. Результаты моделирования: соотношение времени активности/блокировки для отдельных «философов» и модели в целом; соотношение успешных/неуспешных обращений; общая эффективность (пропускная способность); фиксация возникновения (и разрешения, если это предусмотрено) тупиков и др.

2 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Согласно формулировке задачи, были спроектированы следующие функции программы:

– функция создания и запуска потоков «обедающих философов»;

– функция, имитирующая алгоритм действий «обедающих философов».

## **2.1** **Создание и запуск процессов «обедающих философов»**

При запуске программы создаются потоки «философы» и семафоры, которые играют роль вилок. После чего потоки философы начинают работать. [1,2] (рисунок 2.1).

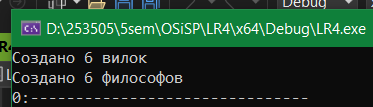
****

Рисунок 2.1 – Результат создания потоков «философов» и семафоров вилок

## **2.2** **Алгоритм действий «обедающих философов»**

Отдельная переменная в коде отвечает за алгоритм действий «философов». Например, базовый метод в котором философы пытаются завладеть двумя вилками и ждет освобождения вилки если у него не получается. [3,4] (рисунок 2.2).

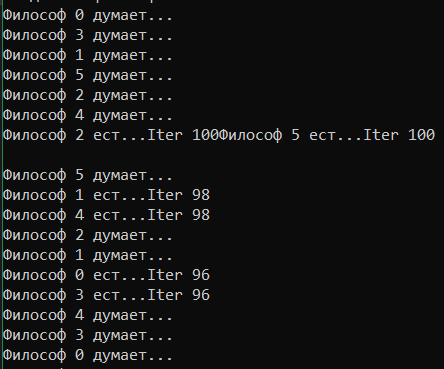


Рисунок 2.2 – Результат вывода данных о состоянии каждого потока

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была изучена и реализована классическая модель взаимодействия конкурирующих параллельных процессов под названием «Обедающие философы». Основной целью работы было изучение методов взаимного исключения и синхронизации, а также проверка различных подходов к обеспечению корректного взаимодействия процессов при доступе к общим ресурсам.

В результате реализации модели были исследованы ключевые аспекты проблемы взаимного исключения и возможности предотвращения взаимных блокировок (deadlocks). В частности, мы рассмотрели несколько подходов к синхронизации процессов: использование семафоров, мьютексов и различных алгоритмов распределения ресурсов, таких как решение с порядком использования и механизмом предотвращения тупиков. Каждый из методов показал различную эффективность в зависимости от параметров модели, таких как количество философов, интенсивность запросов к ресурсам, а также продолжительность владения ресурсами.

Были исследованы и продемонстрированы механизмы регулирования доступа философов к критическим ресурсам (вилкам), что позволило оценить соотношение времени активности и блокировки процессов. Моделирование показало, что распределение ресурсов играет важную роль в общей эффективности системы: чем более сбалансирована логика разрешения конфликтов, тем выше пропускная способность модели. Кроме того, использование случайных величин для моделирования интенсивности и длительности обращений к ресурсам способствовало изучению поведения системы в различных сценариях, что позволило сделать выводы о надежности и устойчивости реализованных алгоритмов.

Проведенная работа также позволила выявить ситуации взаимных блокировок и протестировать различные методы их разрешения. Например, введение тайм-аутов и переупорядочивание запросов в определенных случаях позволило избежать тупиков и обеспечить равномерное распределение ресурсов между процессами. Это особенно важно для систем, где требуется поддержание высокой производительности и минимизация времени простоя.

На основании проведенного анализа было установлено, что выбор конкретного алгоритма синхронизации влияет не только на эффективность использования ресурсов, но и на устойчивость модели к нагрузкам. В рамках лабораторной работы удалось провести параметризацию модели, изменяя количество философов и характеристики использования ресурсов, что дало возможность наблюдать, как меняются показатели успешных и неуспешных обращений, а также общая пропускная способность системы. Таким образом, реализация модели «Обедающие философы» позволила глубже понять принципы системного программирования, связанные с синхронизацией и взаимным исключением, и дала практический опыт применения различных подходов к решению задач параллельного программирования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] CreateThread function [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread

[2] CreateSemaphore function [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createsemaphorea

[3] ReleaseSemaphore function [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-releasesemaphore

[4] WaitForSingleObject function [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject

ПРИЛОЖЕНИЕ А  
(обязательное)  
Исходный код программы

#include <windows.h>

#include <iostream>

using namespace std;

const int NUM\_PHILOSOPHERS = 6;

int iter = 100;

HANDLE mutex; // Мьютекс для синхронизации

HANDLE forkControl; // Общий семафор для контроля захвата вилок

const DWORD TIMEOUT = 100; // Таймаут ожидания вилки в миллисекундах

bool show = true;

int ATTITUDE\_MODEL = 0;

// 0 - естественный

// 1 - классичческий

// 2 - запрет на действия пока 1 захватывает вилки

// 3 - запрет на захват пока 1 захватывает вилки

// 4 - избыточность

// 5 - тайм аут

int\* NUMBER\_OF\_MEALS = new int[NUM\_PHILOSOPHERS] {0};

HANDLE semaphores[NUM\_PHILOSOPHERS]; // Семафоры для философов

int THINKING\_TIME = 100;

int EATING\_TIME = 2;

HANDLE\* forks = new HANDLE[NUM\_PHILOSOPHERS]; // Семафоры для вилок

HANDLE philosophers[NUM\_PHILOSOPHERS]; // Потоки философов

DWORD WINAPI Philosopher(LPVOID lpParam)

{

int id = (int)(intptr\_t)lpParam;

int leftFork = id;

int rightFork = (id + 1) % NUM\_PHILOSOPHERS;

while (iter > 0) {

// Философ думает

if(show)

{

cout << "Философ " << id << " думает...\n";

}

Sleep(THINKING\_TIME);

if (ATTITUDE\_MODEL == 0)

{

// Философ пытается взять вилки

WaitForSingleObject(forks[leftFork], INFINITE); // Захват левой вилки

WaitForSingleObject(forks[rightFork], INFINITE); // Захват правой вилки

}

else if(ATTITUDE\_MODEL == 1)

{ // Попытка захватить семафор (право на еду)

WaitForSingleObject(semaphores[id], INFINITE); // Захват семафора философа

if (show)

{

cout << "Философ " << id << " захватил :" << id << "\n";

}

if (id == 0) {

// Философ 0: проверяем соседа 1 и 4

WaitForSingleObject(semaphores[1], INFINITE);

if (show)

{

cout << "Философ " << id << " захватил :" << 1 << "\n";

}

WaitForSingleObject(semaphores[NUM\_PHILOSOPHERS - 1], INFINITE);

if (show)

{

cout << "Философ " << id << " захватил :" << NUM\_PHILOSOPHERS - 1 << "\n";

}

}

else {

// Другие философы: проверяем соседей

WaitForSingleObject(semaphores[id - 1], INFINITE);

if (show)

{

cout << "Философ " << id << " захватил :" << id - 1 << "\n";

}

WaitForSingleObject(semaphores[(id + 1) % NUM\_PHILOSOPHERS], INFINITE);

if (show)

{

cout << "Философ " << id << " захватил :" << (id + 1) % NUM\_PHILOSOPHERS << "\n";

}

}

}

else if (ATTITUDE\_MODEL == 2)

{

WaitForSingleObject(mutex, INFINITE); // Захват мьютекса

//cout << "Философ " << id << " захватил мьютекс...\n";

WaitForSingleObject(forks[leftFork], INFINITE); // Захват левой вилки

WaitForSingleObject(forks[rightFork], INFINITE); // Захват правой вилки

}

else if (ATTITUDE\_MODEL == 3)

{

WaitForSingleObject(forkControl, INFINITE); // Захват мьютекса

WaitForSingleObject(forks[leftFork], INFINITE); // Захват левой вилки

WaitForSingleObject(forks[rightFork], INFINITE); // Захват правой вилки

// Освобождение семафора для других философов

ReleaseSemaphore(forkControl, 1, NULL);

}

else if (ATTITUDE\_MODEL == 4)

{

// Философ пытается взять вилки

WaitForSingleObject(forks[id\*2], INFINITE); // Захват левой вилки

WaitForSingleObject(forks[id\*2+1], INFINITE); // Захват правой вилки

}

else if (ATTITUDE\_MODEL == 5)

{

DWORD result;

// Захват левой вилки с таймаутом

result = WaitForSingleObject(forks[leftFork], TIMEOUT);

if (result == WAIT\_OBJECT\_0) { // Успешно захвачена левая вилка

// Попытка захватить правую вилку с таймаутом

result = WaitForSingleObject(forks[rightFork], TIMEOUT);

if (result == WAIT\_OBJECT\_0) { // Успешно захвачена правая вилка

//// Философ ест

//cout << "Философ " << id << " ест...\n";

//mealCount[id]++;

}

else {

// Не удалось захватить правую вилку, освобождаем левую

ReleaseSemaphore(forks[leftFork], 1, NULL);

// cout << "Философ " << id << " освободил левую вилку...\n";

}

}

}

// Философ ест

if (show)

{

cout << "Философ " << id << " ест..." << "Iter " << iter << '\n';

}

NUMBER\_OF\_MEALS[id]++;

Sleep(EATING\_TIME);

if (ATTITUDE\_MODEL == 0)

{

// Освобождаем вилки

ReleaseSemaphore(forks[leftFork], 1, NULL);

ReleaseSemaphore(forks[rightFork], 1, NULL);

}

else if (ATTITUDE\_MODEL == 1)

{

if (id == 0) {

// Философ 0: проверяем соседа 1 и 4

ReleaseSemaphore(semaphores[1], 1, NULL);

if (show)

{

cout << "Философ " << id << " отпустил :" << 1 << "\n";

}

ReleaseSemaphore(semaphores[NUM\_PHILOSOPHERS - 1], 1, NULL);

if (show)

{

cout << "Философ " << id << " отпустил :" << NUM\_PHILOSOPHERS - 1 << "\n";

}

}

else {

// Другие философы: проверяем соседей

ReleaseSemaphore(semaphores[id - 1], 1, NULL);

if (show)

{

cout << "Философ " << id << " отпустил :" << id - 1 << "\n";

}ReleaseSemaphore(semaphores[(id + 1) % NUM\_PHILOSOPHERS], 1, NULL);

if (show)

{

cout << "Философ " << id << " отпустил :" << (id + 1) % NUM\_PHILOSOPHERS << "\n";

}

}

// Освобождение своего семафора

ReleaseSemaphore(semaphores[id], 1, NULL);

if (show)

{

cout << "Философ " << id << " отпустил :" << id << "\n";

}

}

else if (ATTITUDE\_MODEL == 2)

{

// Освобождение мьютекса

ReleaseMutex(mutex);

ReleaseSemaphore(forks[leftFork], 1, NULL);

ReleaseSemaphore(forks[rightFork], 1, NULL);

}

else if (ATTITUDE\_MODEL == 3)

{

ReleaseSemaphore(forks[leftFork], 1, NULL);

ReleaseSemaphore(forks[rightFork], 1, NULL);

}

else if (ATTITUDE\_MODEL == 4)

{

ReleaseSemaphore(forks[id \* 2], 1, NULL);

ReleaseSemaphore(forks[id \* 2 + 1], 1, NULL);

}

else if (ATTITUDE\_MODEL == 5)

{

// Освобождение вилок

ReleaseSemaphore(forks[leftFork], 1, NULL);

ReleaseSemaphore(forks[rightFork], 1, NULL);

}

iter--;

}

return 0;

}

int main() {

setlocale(LC\_ALL, "russian");

// Инициализация семафоров для вилок

for (int n = 0; n < 100; n++)

{

iter = 100;

NUMBER\_OF\_MEALS = new int[NUM\_PHILOSOPHERS] {0};

for (int i = 0; i < NUM\_PHILOSOPHERS; ++i)

{

forks[i] = CreateSemaphore(NULL, 1, 1, NULL); // Создаем семафор с начальным и максимальным значением 1

}

cout << "Создано " << NUM\_PHILOSOPHERS << " вилок\n";

if (ATTITUDE\_MODEL == 4)

{

//Добавляем вилки

forks = new HANDLE[NUM\_PHILOSOPHERS \* 2];

for (int i = 0; i < NUM\_PHILOSOPHERS\*2; ++i)

{

forks[i] = CreateSemaphore(NULL, 1, 1, NULL); // Создаем семафор с начальным и максимальным значением 1

}

}

if (ATTITUDE\_MODEL == 0)

{

}

else if (ATTITUDE\_MODEL == 1)

{

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

semaphores[i] = CreateSemaphore(NULL, 1, 1, NULL); // Создаем семафор для каждого философа

}

}

else if (ATTITUDE\_MODEL == 2)

{// Создание мьютекса

mutex = CreateMutex(NULL, FALSE, NULL);

}

else if (ATTITUDE\_MODEL == 3)

{// Создание мьютекса

mutex = CreateMutex(NULL, FALSE, NULL);

}

// Создание общего семафора для контроля захвата вилок

forkControl = CreateSemaphore(NULL, 1, 1, NULL);

// Создание потоков философов

if (ATTITUDE\_MODEL == 1)

{

for (int i = 0; i < NUM\_PHILOSOPHERS; i+=2)

{

philosophers[i] = CreateThread(NULL, 0, Philosopher, (LPVOID)(intptr\_t)i, 0, NULL);

}

Sleep(100);

for (int i = 1; i < NUM\_PHILOSOPHERS; i += 2)

{

philosophers[i] = CreateThread(NULL, 0, Philosopher, (LPVOID)(intptr\_t)i, 0, NULL);

}

}

else

{

cout << "Создано " << NUM\_PHILOSOPHERS << " философов\n";

for (int i = 0; i < NUM\_PHILOSOPHERS; ++i)

{

philosophers[i] = CreateThread(NULL, 0, Philosopher, (LPVOID)(intptr\_t)i, 0, NULL);

}

}

// Ожидание завершения потоков (они работают бесконечно в этом примере)

WaitForMultipleObjects(NUM\_PHILOSOPHERS, philosophers, TRUE, INFINITE);

// Освобождение ресурсов (этот код в примере недостижим, но должен быть для завершения)

for (int i = 0; i < NUM\_PHILOSOPHERS; ++i)

{

CloseHandle(forks[i]);

CloseHandle(philosophers[i]);

}

cout << n << ":-------------------------------\n";

for (int i = 0; i < NUM\_PHILOSOPHERS; ++i)

{

cout << "id: " << i << " meal num: " << NUMBER\_OF\_MEALS[i] << '\n';

}

cout << "---------------------------------\n";

}

return 0;

}