Министерство образования и науки Украины

Днепропетровский национальный университет железного

транспорта имени академика В.Лазаряна

**Лабораторная работа №2**

по дисциплине: «Операционные системы»

Выполнил: ст. группы ПЗ1611

Галушка Александр  
Принял: Андрющенко В. А.

г.Днепр

2018 год

**Тема: Об'єкти ядра. Очікування потоків і процесів. Робота з бар'єрами.Варіанти індивідуальних завдань.**

1. **Завдання**

Розробити програми, які моделюють рішення задачі синхронізації з використанням заданих системних ресурсів. Під час запуску процесів передавати параметри через командний рядок та через змінні оточення програми.

1. **Варіанти індивідуальних завдань**

Задача синхронізації:

1. Філософи, що обідають.

Об’єкти ядра для синхронізації:

1. Семафор;

Приналежність потоків:

1. Одному процесу;
2. **Теоретичні відомості**

Операційні системи Windows функціонують в багатозадачному режимі, тобто в їх середовищі одночасно виконуються декілька прикладних програм. Однак це не значить, що на одному процесорі паралельно виконуються машинні команди цих програм. Операційна система створює такі умови для кожної програми, щоб надати їй всі ресурси комп’ютера в виділений момент часу. Програмний код, якому необхідно "виділити процесор" в операційній системі Windows називається Потоком (Thread). Щоб одночасно функціонувало декілька Потоків, операційна система виділяє кожному із них певну частину часу, яка називається квантом. Тривалість кванту залежить від конкретної реалізації. Таким чином, вся система працює по принципу каруселі: операційна система почергово, на один квант часу відправляє процесору кожний Потік для його виконання. Після обробки останнього Потоку система знову запускає перший. Перед передачею процесору іншого Потоку, система зберігає для попереднього Потоку стан всіх регістрів процесора. Ця інформація називається контекстом Потоку. Повертаючись до цього Потоку знову, система завантажує його контекст в процесор, для того щоб той продовжив виконувати перерваний минулого разу потік.

Всі потоки та їх дані розміщуються в конкретному Процесі (Process). З точки зору прикладної програми Процес – це адресний простір, в якому розміщуються код та дані всіх EXE та DLL модулів. Саме в ньому знаходяться області пам’яті, які динамічно розподіляються для стеків потоку та інших потреб. При запуску на виконання EXE-програми система створює для неї Процес, в контексті якого запускає первинний Потік. Цей Потік може запустити на виконання ще дочірні Потоки, які будуть виконуватися паралельно. Коли всі Потоки знищуються, Процес теж знищується.

Процеси та Потоки є ресурсами операційної системи, які також являються її об’єктами ядра, якими можна оперувати. Кожний об’єкт ядра має свій унікальний ідентифікатор в системі. Однак в контексті кожного Процесу об’єктами ядра користуються через дескриптори (handle). Таким чином в кожному Процесі один і той самий об’єкт ядра буде мати однаковий ідентифікатор але різні дескриптори.

Потоки усипляють себе до звільнення якого-небудь синхронізуючого об'єкту за допомогою двох функцій:

DWORD WaitForSingleObject (HANDLE hObject, DWORD dwTimeOut);

DWORD WaitForMultipleObjects(DWORD nCount,

CONST HANDLE\* lpHandles, BOOL bWaitAll, DWORD dwTimeOut);

Функція WaitForSingleObject припиняє потік до тих пір, поки або заданий параметром hObject синхронізуючий об'єкт не звільниться, або доки не витече інтервал часу, що задається параметром dwTimeOut. Якщо вказаний об'єкт протягом заданого інтервалу не перейде у вільний стан, то система знов активізує потік і він продовжить своє виконання. Також можна вказати константу INFINITE, яка означає що потік буде очікувати необмежену кількість часу.

Функція повертає наступні значення.

|  |  |
| --- | --- |
| Код повернення / значення | Опис |
| WAIT\_ABANDONED  0x00000080L | Зазначений об'єкт є об'єктом mutex, який не був випущений потоком, якому належав об'єкт мьютекса, до того, як завершився спадний потік. Власність на об'єкт мьютекса надається викликає потоку, а стан мьютекса - несиметричним.  Якщо м'ютекс захищає постійну інформацію про стан, ви повинні перевірити його на узгодженість. |
| WAIT\_OBJECT\_0  0x00000000L | Вказується стан зазначеного об'єкта. |
| WAIT\_TIMEOUT  0x00000102L | Інтервал тайм-ауту закінчився, і стан об'єкта не має значення. |
| WAIT\_FAILED  (DWORD) 0xFFFFFFFF | Функція не виконано. Щоб отримати розширену інформацію про помилку, викличте GetLastError |

Функція WaitForMultipleObjects затримує потік і залежно від значення прапора bWaitAll чекає однієї з наступних подій:

звільнення хоч би одного синхронізуючого об'єкту із заданого списку; звільнення всіх вказаних об'єктів; закінчення заданого інтервалу часу.

NCount - Кількість ручок об'єктів у масиві, вказаному lpHandles. Максимальна кількість ручок об'єкта - MAXIMUM\_WAIT\_OBJECTS. Цей параметр не може бути нульовим.

LpHandles - Масив об'єктів обробляє. Список типів об'єктів, для яких можна вказати рукоятки, див. У розділі «Зауваження». Масив може містити ручки для об'єктів різних типів. Вона може не містити декілька копій однієї і тієї ж ручки.

Якщо одна з цих ручок закрита, тоді як очікування все ще очікує на розгляд, поведінка функції не визначена.

Ручки повинні мати право доступу SYNCHRONIZE. Для отримання додаткової інформації див. Розділ Стандартний доступ.

BWaitAll - Якщо цей параметр істинний, функція повертається, коли сигналізується стан усіх об'єктів у масиві lpHandles. Якщо FALSE, функція повертається, коли стан будь-якого з об'єктів встановлений для сигналізації. В останньому випадку повернення значення вказує об'єкт, стан якого викликав повернення функції.

DwMilliseconds -Інтервал тайм-ауту, в мілісекундах. Якщо вказано значення відмінного від нуля, функція очікує, доки не буде зазначено вказаних об'єктів або пройде інтервал. Якщо dwMilliseconds дорівнює нулю, функція не входить в стан очікування, якщо вказані об'єкти не сигналізуються; Він завжди повертається негайно. Якщо dwMilliseconds INFINITE, функція повернеться лише тоді, коли вказані об'єкти будуть сигнальними.

Функція повертає наступні значення

|  |  |
| --- | --- |
| Код / значення повернення | Опис |
| **WAIT\_OBJECT\_0** to (**WAIT\_OBJECT\_0** + nCount– 1) | Якщо bWaitAll TRUE, то повернення значення вказує на те, що сигналізується стан усіх вказаних об'єктів.  Якщо bWaitAll FALSE, то повернення значення мінус WAIT\_OBJECT\_0 вказує індекс масиву lpHandles об'єкта, який задовольняє очікування. Якщо під час дзвінка було надіслано більше одного об'єкта, це індекс масиву сигнального об'єкта з найменшим значенням індексу всіх сигнальних об'єктів. |
| **WAIT\_ABANDONED\_0** to (**WAIT\_ABANDONED\_0** + nCount– 1) | Якщо bWaitAll істинно, то повернення значення вказує на те, що стан усіх вказаних об'єктів є сигнальним, і принаймні один з об'єктів є покинутим об'єктом mutex.  Якщо bWaitAll - FALSE, то повернення значення мінус WAIT\_ABANDONED\_0 вказує індекс масиву lpHandles залишеного об'єкта mutex, який задовольняє очікування. Власність об'єкта mutex надається викликовій лінії, а mutex встановлено як несигнальований.  Якщо мьютекс захищає стійку інформацію про стан, вам слід перевірити його на предмет послідовності. |
| Код / значення повернення | Опис |
| **WAIT\_TIMEOUT**  0x00000102L | Інтервал тайм-ауту минув, і умови, зазначені параметром bWaitAll, не виконуються. |
| **WAIT\_FAILED**  (**DWORD**)0xFFFFFFFF | Функція не виконана. Щоб отримати розширену інформацію про помилку, зателефонуйте GetLastError. |

**Концепция семафоров**

Семафор представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: ***P*** (от датского слова proberen — проверять) и ***V*** (от verhogen — увеличивать). Классическое определение этих операций выглядит следующим образом:

|  |  |
| --- | --- |
| ***P(S):*** | ***пока S == 0 процесс блокируется;  S = S – 1;*** |
| ***V(S):*** | ***S = S + 1;*** |

Эта запись означает следующее: при выполнении операции ***P*** над семафором ***S*** сначала проверяется его значение. Если оно больше **0**, то из ***S*** вычитается**1**. Если оно меньше или равно**0**, то процесс блокируется до тех пор, пока ***S*** не станет больше **0**, после чего из ***S***вычитается **1**. При выполнении операции ***V*** над семафором***S*** к его значению просто прибавляется **1**.

Подобные переменные-семафоры могут быть с успехом применены для решения различных задач организации взаимодействия процессов. В ряде языков программирования они были непосредственно введены в синтаксис языка (например, в ALGOL-68), в других случаях применяются через использование системных вызовов. Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций***P*** и ***V***, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции ***P*** заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

**Створення семафора**

CreateSemaphore(атрибут доступа, ініціалізація початкового стану лічильника, Максимальна кількість звернень, ім’я об’єкта);

Після створення семафору він починає самостійно працювати до завершення своєї роботи.

**Рішення завдання виробників-споживачів.**

Однією з типових задач, що вимагають організації взаємодії процесів, є завдання producer-consumer (виробник-споживач). Нехай два процеси обмінюються інформацією через буфер обмеженого розміру. Виробник закладає інформацію в буфер, а споживач отримує її звідти. Грубо кажучи, на цьому рівні діяльність споживача і виробника можна описати таким чином.

Producer:

while(1) {

produce\_item;

put\_item;

}

Consumer:

while(1) {

get\_item;

consume\_item;

}

Якщо буфер забитий, то виробник повинен чекати, поки в ньому з'явиться місце, щоб покласти туди нову порцію інформації. Якщо буфер порожній, то споживач повинен чекати нового повідомлення. Як можна реалізувати ці умови за допомогою семафорів?

Візьмемо три семафора empty, full і mutex. Семафор full будемо використовувати для гарантії того, що споживач буде чекати, поки в буфері з'явиться інформація. Семафор empty будемо використовувати для організації очікування виробника при заповненому буфері, а семафор mutex - для організації взаємовиключення на критичних ділянках, якими є дії put\_item і get\_item (операції покласти інформацію і взяти інформацію не можуть перетинатися, так як тоді виникне небезпека спотворення інформації). Тоді рішення задачі виглядає так:

Semaphore mutex = 1;

Semaphore empty = N, где N – емкость буфера;

Semaphore full = 0;

Producer:

while(1) {

produce\_item;

P(empty);

P(mutex);

put\_item;

V(mutex);

V(full);

}

Consumer:

while(1) {

P(full);

P(mutex);

put\_item;

V(mutex);

V(empty);

consume\_item;

}

Легко переконатися, що це дійсно коректне рішення поставленого завдання. Попутно зауважимо, що семафори використовувалися тут для досягнення двох цілей: організації взаємовиключення на критичній ділянці і синхронізації швидкості роботи процесів.

**Текст програми**

// oslr2.cpp : Этот файл содержит функцию "main". Здесь начинается и заканчивается выполнение программы.

//

#include "pch.h"

#include <iostream>

#include <thread>

#include <Windows.h>

#include <process.h>

#include <vector>

#include <string>

#include <ctime>

#define SIZE 5

using namespace std;

HANDLE Semaphore;

class Fork {

public:

bool isUsing = false;

};

Fork \* forks;

class Philosopher {

bool isHunger;

string philosopherName;

int number;

int time\_ = 0;

public:

Philosopher(string name, int number) {

this->philosopherName = name;

this->number = number;

Sleep(253);

srand(time(NULL));

time\_ = rand() % 4000 + 1000;

}

private: void GetFork()

{

time\_ = rand() % 8000 + 2000;

cout << "Философ " << philosopherName << " ждет вилку" << "\ ("<< time\_ << "мс)\n";

int first = number;

int second = (number + 1) % 4;

WaitForSingleObject(Semaphore, INFINITE);

if (forks[first].isUsing || forks[second].isUsing) {

ReleaseSemaphore(Semaphore, 1, NULL);

return;

}

Sleep(rand() % 3000 + 1000);

forks[first].isUsing = true;

forks[second].isUsing = true;

cout << "Философ " << philosopherName << " обедает" << " ("<< time\_ << "мс)\n";

cout << "Вилки " << (first + 1) << " и " << (second + 1) << " заняты " << " ("<< time\_ <<"мс)\n";

ReleaseSemaphore(Semaphore, 1, NULL);

Sleep(rand() % 3000 + 1000);

forks[first].isUsing = false;

forks[second].isUsing = false;

cout << "Вилки " << (first + 1) << " и " << (second + 1) << " свободны " << " (" << time\_ << "мс)\n";

}

public: void Start()

{

while (true)

{

Sleep(time\_);

ChangeStatus();

if (isHunger)

GetFork();

}

}

private: void ChangeStatus()

{

isHunger = !isHunger;

if (!isHunger)

cout << "Философ " << philosopherName << " размышляет." << " ("<< time\_ <<"мс)\n";

}

};

Philosopher \*p1, \*p2, \*p3, \*p4, \*p5;

void test1(void \*) {

p1->Start();

}

void test2(void \*) {

p2->Start();

}

void test3(void \*) {

p3->Start();

}

void test4(void \*) {

p4->Start();

}

void test5(void \*) {

p5->Start();

}

vector <HANDLE> threads; vector <Philosopher> philosophers;

int main()

{

setlocale(LC\_ALL, "Russian");

Philosopher Sokrat(string("Sokrat"), 0);

Philosopher Platon(string("Platon"), 1);

Philosopher Kant(string("Kant"), 2);

Philosopher Marks(string("Marks"), 3);

Philosopher Archimed(string("Archimed"), 4);

p1 = &Sokrat;

p2 = &Platon;

p3 = &Kant;

p4 = &Marks;

p5 = &Archimed;

forks = new Fork[5];

Semaphore = CreateSemaphore(NULL, 1, 5, NULL);

threads.push\_back((HANDLE)\_beginthread(test1, 1024, NULL));

threads.push\_back((HANDLE)\_beginthread(test2, 1024, NULL));

threads.push\_back((HANDLE)\_beginthread(test3, 1024, NULL));

threads.push\_back((HANDLE)\_beginthread(test4, 1024, NULL));

threads.push\_back((HANDLE)\_beginthread(test5, 1024, NULL));

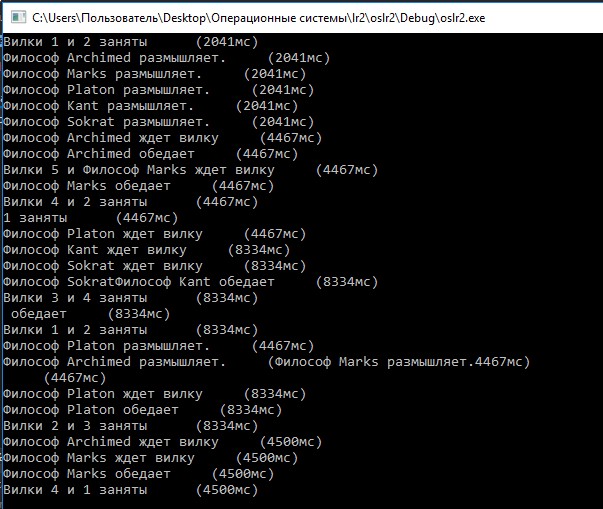
Sleep(1000);

//CloseHandle(Semaphore);

system("pause");

}

**Результати виконання програми**



**Висновок**

В процесі виконання цієї лабораторної роботи я ознайомився з поняттями процесу та потоку, а також засобами синхронізації потоків на мові С++, розробив програму яка вирішує задачу про філософів що обідають.