**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ **«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ ИМ. В. Г. ШУХОВА»**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

**ЛАБОРАТОРНАЯ РАБОТА №3**

**Дисциплина: Операционные системы**

**Тема: Синхронизация потоков**

Выполнил: ст. группы ВТ-31

Подкопаев Антон Валерьевич

Проверил: ст. пр. ПОВТиАС

Михелев Владимир Михайлович

**Белгород 2019**

**Цель работы**: Получение практических навыков по использованию Win32 API для синхронизации процессов и потоков.

**Дескрипторы и идентификаторы потоков**

Тема о дескрипторах и идентификаторах потоков аналогична теме об аналогичных характеристиках процессов.

Вы знаете, что функция CreateProcess возвращает идентификатор и деск­риптор первого (и только первого) потока, выполняющегося во вновь созданном процессе. Функция CreateThread тоже возвращает идентификатор потока, но область действия которого - вся система.

Поток может использовать функцию GetCurrentThreadId, чтобы получить собственный ID. Функция GetWindowThreadProcessId возвращает идентификатор того потока, который создал конкретное окно (как иден­тификатор процесса, которому принадлежит данный поток).

Согласно документации Win32 API не предлагает способа для по­лучения дескриптора потока по его идентификатору. Если бы дескрипторы можно было находить, таким образом, то процесс, которому принадлежат по­токи, завершался бы неудачей, так как другой процесс смог бы выполнять не­санкционированные операции с его потоками, например, приостанавливать поток, возобновлять его действие, изменять приоритет или завершать его работу. Запрашивать дескриптор следует у процесса, создавшего данный поток, или у самого потока.

Наконец, поток может вызывать функцию GetCurrentThread для получе­ния собственного псевдодескриптора. Как и в случае псевдодескрипторов процес­сов, псевдодескриптор потока может использоваться только для вызова процесса и не может наследоваться. Можно использовать функцию DuplicateHandle для получения настоящего дескриптора потока по его псевдодескриптору так же, как это делается в процессах.

**Приоритет потоков**

Термин многозадачность(multitasking) или мультипрограммирование(multiprogramming), обозначает возможностьуправлять несколькими процессами (или несколькими потоками) на базе одного процессора. Многопроцессорной обработкой(multiprocessing) называется управление некоторым числом процессов или потоков на нескольких процессорах.

Компьютер может одновременно выполнять команды двух разных процессов только в многопроцессорной среде. Однако даже на одном процессоре с помощью переключения задач (task switching) можно создать впечатление, что одновремен­но выполняются команды нескольких процессов.

В старой 16-разрядной Windows существовал только один поток. Более того, в данной системе был реализован метод кооперативной (cooperative) многозадач­ности, который состоит в том, что каждое приложение само отвечает за высво­бождение единственного системного потока, после чего могут выполняться дру­гие приложения. Если программа выполняла задачи, требующие значительного времени, такие как форматирование гибкого диска, все другие загруженные при­ложения должны были ждать. А если программа, написанная с ошибками, входила в бесконечный цикл, то вся система становилась непригодной для использова­ния («зависала»), требовала перезагрузки с выключением питания.

**Уровни приоритета потоков**

Win32 значительно отличается от Win16. Во-первых, она является многопоточной (multithreaded), что определяет ее многозадачность. Во-вторых, в ней реа­лизована модель вытесняющей (preemptive) многозадачности, в которой операционная система решает, когда каждый поток получает процессорное время, выделяемое квантами времени (time slices), и сколько именно времени выделяет­ся. Временной интервал в Windows называется квантом.

Продолжительность кванта времени зависит от аппаратуры и может факти­чески меняться от потока к потоку. Например, базовое значение в Windows 95 составляет 20 мс, для Windows NT Workstation (на базе процессора Pentium) -30 мс, для Windows NT Server - 180 мс.

Давайте выясним, каким образом Windows выделяет кванты времени потокам в системе. Эти процедуры в Windows 9x и Windows NT довольно похожи, но не идентичны.

Каждый поток в системе имеет *уровень приоритета* (priority level), который представляет собой число в диапазоне от 0 до 31. Ниже перечислено то самое ос­новное, что нужно знать:

* если существуют какие-либо потоки с приоритетом 31, которые требуют процессорного времени, то есть не находятся в состоянии ожидания (not idle), операционная система перебирает эти потоки (независимо от того, каким процессам они принадлежат), выделяя по очереди каждому из них кванты времени. Потокам с более низким приоритетом кванты времени со всем не выделяются, поэтому они не выполняются. Если нет активных по­токов с приоритетом 31, операционная система ищет активные потоки с уровнем приоритета 30 и т.д. Не забудьте, однако, что потоки довольно часто простаивают. Тот факт, что приложение загружено, не означает, что все его потоки активны. Поэтому у потоков с более низким приоритетом все же есть возможность работать. Более того, если пользователь нажимает клавишу, относящуюся к процессу, потоки которого простаивают, операционная сис­тема временно выделяет процессорное время соответствующему потоку, чтобы он мог обработать нажатие клавиши;
* приоритет 0 зарезервирован исключительно за специальным системным по­током, который называется потоком нулевой страницы (zero page thread). Он освобождает незадействованные области памяти. Существует также по­ток idle, который работает с уровнем приоритета 0, опрашивая систему в поисках какой-нибудь работы;
* если поток с произвольным приоритетом выполняется в тот момент, когда потоку с большим приоритетом потребовалось процессорное время (напри­мер, он получает сообщение о том, что пользователь щелкнул мышью), опе­рационная система немедленно вытесняет поток с меньшим приоритетом и отдает процессорное время потоку с большим. Таким образом, поток может не успеть завершить выделенный ему квант времени;
* для того чтобы перейти с одного потока на другой, система осуществляет переключение контекста (context switch). Это процедура, сохраняющая состояние процессора (регистров и стека) и загрузки соответствующих значе­ний другого потока.

**Назначение приоритета потоку**

Назначение потоку приоритета происходит в два этапа. Во-первых, каждому процессу в момент создания присваивается класс приоритета. Узнать класс при­оритета можно с помощью функции *GetPriorityClass*, а изменить - с помо­щью функции *SetPriorityClass*. Ниже приведены имена классов при­оритета процессов, уровни приоритета и константы, которые используются с этими вышеупомянутыми функциями (как и с функцией *CreateProcess*).

|  |  |  |
| --- | --- | --- |
| Имя класса приоритета | Уровень приоритета класса | Символьная константа |
| *Idle* | 4 | *IDLE\_PRIORITY\_CLASS=&H40* |
| *Normal* | 8 | *NORMAL\_PRIORITY\_CLASS=&H20* |
| *High* | 13 | *HIGH\_ PRIORITY\_CLASS=&H80* |
| *Realtime* | 24 | *REALTIME\_ PRIORITY\_CLASS=&H100* |

Большинство процессов должно получать класс уровня приоритета Normal (обычный). Однако некоторым приложениям, таким как приложения мониторинга системы, возможно, более уместно назначать приоритет Idle (ожидания). Назначения приоритета *Realtime* (реального времени) обычно следует избегать, по­тому что в этом случае потоки изначально получают приоритет более высокий, чем системные потоки, такие как потоки ввода от клавиатуры и мыши, очистки кэша и обработки нажатия клавиш Ctrl+Alt+Del. Такой приоритет может быть подходящим для краткосрочных, критичных к времени выполнения процессов, которые относятся к взаимодействию с аппаратурой.

**Повышение приоритета потока и квант изменений приоритета**

Диапазон приоритета от 1 до 15 известен как диапазон динамического при­оритета (*dynamic priority*), а диапазон от 16 до 31 - как диапазон приоритета реального времени (*realtime priority).*

В Windows NT приоритет потока, находящийся в динамическом диапазоне, может временно повышаться операционной системой в различные моменты вре­мени. Соответственно, нижний уровень приоритета потока (установленный про­граммистом с помощью API функции) называется уровнем его базового приоритета (*base priority).* API функция Windows NT *SetProcessPriorityBoost* может использоваться для разрешения или запрещения временных изменений приоритета (priority boosting). Правда, она не поддерживается в Windows 9х.

Бывают также случаи, когда кванты времени, выделяемые потоку, временно увеличиваются.

Стремясь плавно выполнять операции, Windows будет повышать приоритет потока или увеличивать продолжительность его кванта времени при следующих условиях:

* если поток принадлежит приоритетному процессу (*foreground process*), то есть процессу, окно которого активно и имеет фокус ввода;
* если поток первым вошел в состояние ожидания;
* если поток выходит из состояния ожидания;
* если поток совсем не получает процессорного времени.

**Состояния потоков**

Потоки могут находиться в одном из нескольких состояний:

* *Ready* (готов) – находящийся в пуле (pool) потоков, ожидающих выполнения;
* *Running* (выполнение) - выполняющийся на процессоре;
* *Waiting* (ожидание), также называется idle или suspended, приоста­новленный - в состоянии ожидания, которое завершается тем, что поток начинает выполняться (состояние Running) или переходит в состояние *Ready*;
* *Terminated* (завершение) - завершено выполнение всех команд потока. Впоследствии его можно удалить. Если поток не удален, система может вновь установить его в исходное состояние для последующего использования.

**Синхронизация потоков**

Выполняющимся потокам часто необходимо каким-то образом взаимодей­ствовать. Например, если несколько потоков пытаются получить доступ к неко­торым глобальным данным, то каждому потоку нужно предохранять данные от изменения другим потоком. Иногда одному потоку нужно получить информацию о том, когда другой поток завершит выполнение задачи. Такое взаимодействие возможно между потоками как одного, так и разных процессов.

Синхронизация потоков (*thread synchronization*) - это обобщенный термин, относящийся к процессу взаимодействия и взаимосвязи потоков. Учтите, что син­хронизация потоков требует привлечения в качестве посредника самой операци­онной системы. Потоки не могут взаимодействовать друг с другом без ее участия.

В Win32 существует несколько методов синхронизации потоков. Бывает, что в конкретной ситуаций один метод более предпочтителен, чем другой. Давайте вкратце ознакомимся с этими методами.

**Критические секции**

Один из методов синхронизации потоков состоит в использовании критичес­ких секций (critical sections). Это единственный метод синхронизации потоков, который не требует привлечения ядра Windows. (Критическая секция не является объектом ядра). Однако этот метод может использоваться только для синхронизации потоков одного процесса.

Критическая секция — это некоторый участок кода, который в каждый момент времени может выполняться только одним из потоков. Если код, используемый для инициа­лизации массива, поместить в критическую секцию, то другие потоки не смогут войти в этот участок кода до тех пор, пока первый поток не завершит его выполнение.

До использования критической секции необходимо инициализировать ее с помощью процедуры Win32 API InitializeCriticalSection(), которая определяется (в Delphi) следующим образом:

procedure InitializeCriticalSection(var IpCriticalSection: TRTLCriticalSection); stdcall;

Параметр IpCriticalSection представляет собой запись типа TRTLCriticalSection, ко­торая передается по ссылке. Точное определение записи TRTLCriticalSection не имеет большого значения, поскольку вам вряд ли понадобится когда-либо заглядывать в ее содер­жимое. От вас требуется лишь передать неинициализированную запись в параметр IpCtitical Section, и эта запись будет тут же заполнена процедурой.

После заполнения записи в программе можно создать критическую секцию, поместив не­который участок ее текста между вызовами функций EnterCriticalSection() и LeaveCriticalSection(). Эти процедуры определяются следующим образом:

procedure EnterCriticalSection(var IpCriticalSection: TRTLCriticalSection); stdcall;

procedure LeaveCriticalSection(var IpCriticalSection: TRTLCriticalSection); stdcall;

Параметр IpCriticalSection, который передается этим процеду­рам, является не чем иным, как записью, созданной процедурой InitializeCriticalSection().

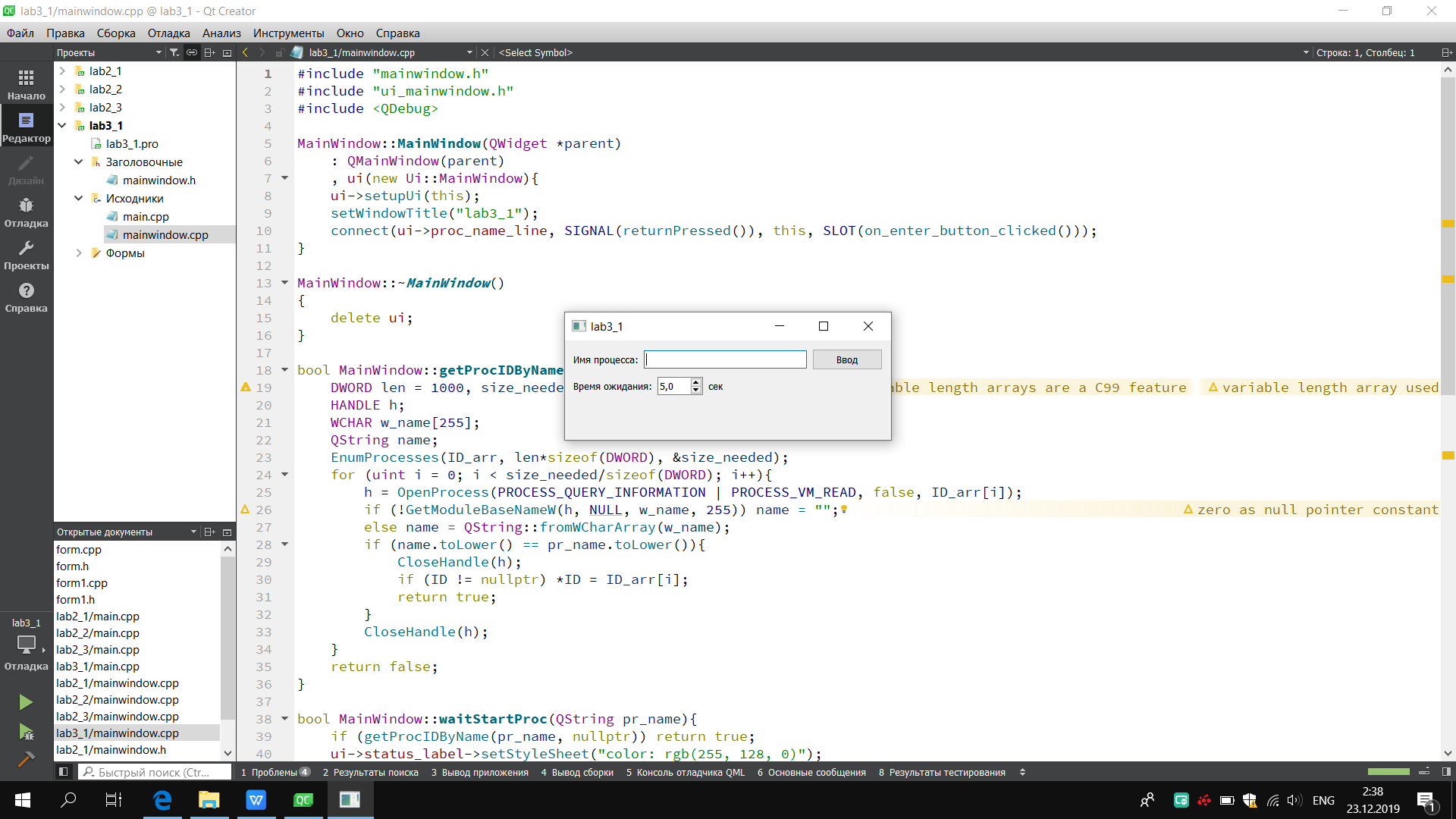
Функция *EnterCriticalSection* проверяет, не выполняет ли уже какой-нибудь другой поток критическую секцию своей программы, связанную с данным объектом критической секции. Если нет, поток получает разрешение на выполне­ние своего критического кода, точнее, ему не запрещают это делать. Если да, то поток, обратившийся с запросом, переводится в состояние ожидания, а о запросе делается запись. Так как нужно создавать записи, объект «критическая секция» представляет собой структуру данных.

Когда функция *LeaveCriticalSection* вызывается потоком, который вла­деет в текущий момент разрешением на выполнение своей критической секции кода, связанной с данным объектом «критическая секция», система может прове­рить, нет ли в очереди другого потока, ожидающего освобождения этого объекта. Затем система может вынести ждущий поток из состояния ожидания, и он про­должит свою работу (в выделенные ему кванты времени).

По окончании работы с записью TRTLCriticalSection необходимо освободить ее, вы­звав процедуру DeleteCriticalSection(), которая определяется следующим образом:

procedure DeleteCriticalSection(var IpCriticalSection: TRTLCriticalSection); stdcall;

**3-1**

****

mainwindow.h

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <windows.h>

#include <psapi.h>

#include <QTime>

QT\_BEGIN\_NAMESPACE

namespace Ui { class MainWindow; }

QT\_END\_NAMESPACE

class MainWindow : public QMainWindow

{

Q\_OBJECT

public:

MainWindow(QWidget \*parent = nullptr);

~MainWindow();

bool getProcIDByName(QString pr\_name, DWORD \*ID);

bool waitStartProc(QString pr\_name);

void waitEndProc(QString pr\_name);

private slots:

void on\_enter\_button\_clicked();

private:

Ui::MainWindow \*ui;

};

#endif // MAINWINDOW\_H

mainwindow.cpp

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include <QDebug>

MainWindow::**MainWindow**(QWidget \*parent)

: QMainWindow(parent)

, ui(new Ui::MainWindow){

ui->setupUi(this);

setWindowTitle("lab3\_1");

connect(ui->proc\_name\_line, SIGNAL(returnPressed()), this, SLOT(on\_enter\_button\_clicked()));

}

MainWindow::~***MainWindow***()

{

delete ui;

}

bool MainWindow::**getProcIDByName**(QString pr\_name, DWORD \*ID){

DWORD len = 1000, size\_needed, ID\_arr[len];

HANDLE h;

WCHAR w\_name[255];

QString name;

EnumProcesses(ID\_arr, len\*sizeof(DWORD), &size\_needed);

for (uint i = 0; i < size\_needed/sizeof(DWORD); i++){

h = OpenProcess(PROCESS\_QUERY\_INFORMATION | PROCESS\_VM\_READ, false, ID\_arr[i]);

if (!GetModuleBaseNameW(h, NULL, w\_name, 255)) name = "";

else name = QString::fromWCharArray(w\_name);

if (name.toLower() == pr\_name.toLower()){

CloseHandle(h);

if (ID != nullptr) \*ID = ID\_arr[i];

return true;

}

CloseHandle(h);

}

return false;

}

bool MainWindow::**waitStartProc**(QString pr\_name){

if (getProcIDByName(pr\_name, nullptr)) return true;

ui->status\_label->setStyleSheet("color: rgb(255, 128, 0)");

ui->status\_label->setText("Процесс " + pr\_name + " не запущен");

repaint();

return false;

}

void MainWindow::**waitEndProc**(QString pr\_name){

if (!waitStartProc(pr\_name)) return;

DWORD waiting\_time = 1000\*ui->time\_spin->value();

ui->status\_label->setStyleSheet("color: rgb(100, 100, 200)");

ui->status\_label->setText("Ожидание завершения процесса " + pr\_name);

repaint();

DWORD pr\_ID;

getProcIDByName(pr\_name, &pr\_ID);

HANDLE h = OpenProcess(PROCESS\_QUERY\_INFORMATION | PROCESS\_VM\_READ | SYNCHRONIZE, false, pr\_ID);

QTime t; t.start();

DWORD r\_value = WaitForSingleObject(h, waiting\_time);

double time = (t.elapsed()/100)/10.0;

CloseHandle(h);

switch (r\_value) {

case WAIT\_OBJECT\_0:

ui->status\_label->setStyleSheet("color: rgb(100, 150, 100)");

ui->status\_label->setText("Процесс " + pr\_name + " завершён");

ui->time\_label->setText("Время ожидания: " + QString::number(time) + " сек.");

break;

case WAIT\_TIMEOUT:

ui->status\_label->setStyleSheet("color: rgb(255, 128, 0)");

ui->status\_label->setText("Истекло время ожидания завершения процесса " + pr\_name);

ui->time\_label->setText("Время ожидания: " + QString::number(waiting\_time/1000.0) + " сек.");

break;

case WAIT\_FAILED:

ui->status\_label->setStyleSheet("color: rgb(255, 100, 100)");

ui->status\_label->setText("Произошла ошибка");

}

repaint();

}

void MainWindow::**on\_enter\_button\_clicked**(){

ui->status\_label->clear();

ui->time\_label->clear();

waitEndProc(ui->proc\_name\_line->text());

}

main.cpp

#include "mainwindow.h"

#include <QApplication>

int main(int argc, char \*argv[])

{

QApplication a(*argc*, argv);

MainWindow w;

w.show();

return a.exec();

}