МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Кафедра прикладной математики

Вл. Пономарев

# ПРАКТИКУМ

по операционным системам

Часть 2. Процессы и потоки Windows

Учебно-методическое пособие

2018 г.

УДК 681.3.06
П 56

Вл. Пономарев. Практикум по операционным системам. Часть 2. Процессы и потоки Windows. Учебно-методическое пособие. Озерск: ОТИ НИЯУ МИФИ, 2018. — 63 с.

В пособии подробно излагается, как выполнять практические работы по дисциплине «Операционные системы». Работы второй части включают изучение процессов и потоков Windows.

В качестве основного материала при выполнении практических работ пособие предназначено для студентов, обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника» и специальности 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения».

Рецензенты:
1.
2.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

---

Содержание

Общие цели занятий

В ходе практических работ предлагается изучить основы управления процессами и потоками в операционной среде Windows. В этой части работ рассматриваются следующие темы:

- ввод-вывод;
- создание процессов;
- создание потоков;
- приоритеты процессов и потоков;
- синхронизация процессов и потоков;
- выполнение домашнего задания.

Программирование ведется в среде Microsoft® Visual Studio®, язык программирования C/C++.

На выполнение и защиту работ этой части предположительно отводится 16-18 академических часов. Предполагается, что перед выполнением работы студент изучает работу, используемые структуры данных и функции. Это позволит сэкономить время, отведенное на выполнение работы.

Каждая выполненная работа должна быть защищена.

Для защиты знаний и навыков, полученных в ходе выполнения практических работ студент должен иметь тетрадь (12-18 листов). Для каждой выполненной работы в тетрадь записывается отчет.

Отчет по работе начинается с заголовка:

1. Фамилия Имя Отчество.
2. Группа.
3. Дата начала выполнения работы.
4. Код работы.
5. Название работы.
6. Цели работы.
7. Задачи работы.

При необходимости при выполнении работы в отчет записываются контрольные значения. Во время защиты тетрадь используется для вопросов преподавателя и ответов студента.

1. Работа OS-203. Создание процессов

Цели:
- изучение управления процессами.
Задачи:
- изучение основной функции приложения Win32;
- изучение функции создания процесса;
- создание и запуск процессов.
Опорные документы:
[3, с.48]

1.1. Рабочее пространство

Скачаем с сайта преподавателя архив для выполнения работы 203.
Извлечем из архива каталог process в корневой каталог диска C:.
Откроем проект. Убедимся, что целевой платформой является x86 или win32, и в том, что проект компилируется и запускается.
В проекте два основных модуля. Модуль process.h содержит глобальные переменные и вспомогательные функции. Модуль process.cpp содержит функции для выполнения работы. В начале этого модуля укажите сведения об организации, о себе, о проекте, дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Фамилия Имя Отчество
// OS-203. Процессы
// 01.09.2000
```

1.2. Параметры основной функции

Сначала изучим параметры основной функции WinMain.
Для этого в функции onCreate напишем строчку кода:

```
void onCreate(HINSTANCE hInstance, . . . {
    int foo = 0;
}
```

Установим на ней точку остановки и запустим проект F5.
Первый параметр hInstance типа HINSTANCE — это базовый адрес программы в виртуальном адресном пространстве процесса. Некоторая начальная часть этого пространства отводится для процессов MS-DOS, которые не могут использовать более 32 Мбайт памяти. Запишем в отчет шестнадцатеричное и десятичное значения этого адреса.
Второй параметр hPrevInstance типа HINSTANCE — это дескриптор предыдущей копии данной программы, если она уже запущена в момент запуска новой копии. Этот параметр сейчас равен нулю, и фактически он не используется. Для защиты программы от повторного запуска следует применить какой-нибудь синхронизирующий объект, например файл или мьютекс.

Третий параметр — это хвост командной строки.

Чтобы его увидеть, нужно иметь параметры в командной строке.

В проекте эти параметры задаются в свойствах проекта.

Откроем свойства проекта, раздел Debugging, название параметра Command Arguments, впишем три параметра:

```
first  second  456
```

Запустим программу F5 и после того, как управление придет в функцию onCreate, убедимся, что третий параметр содержит все заданные значения. Остановим выполнение Shift+F5.

Зачем нужны параметры? Они управляют поведением программы, переключая ее в некоторый режим, задают спецификации рабочих файлов, выходных файлов и т.п.

Четвертый параметр — это команда, управляющая состоянием окна после запуска. Этот параметр может принимать 11 различных значений, но основными являются:

SW_SHOW — показать окно в текущей позиции и текущего размера,

SW_HIDE — не показывать окно, активировать другое,

SW_SHOWNORMAL — показать окно в текущей позиции и текущего размера, даже если оно было свернуто или развернуто,

SW_SHOWMINIMIZED — свернуть окно,

SW_SHOWMAXIMIZED — развернуть окно.

Префикс SW_ означает Show Window.

Сейчас попробуем изменить этот параметр.

Функция onCreate, модуль process.cpp:

```
void onCreate(HINSTANCE hInstance, . . . {
    int foo = 0;
    nCmdShow = SW_SHOWMAXIMIZED;
    . . .
}
```

Запустим программу Ctrl+F5. Окно развернется на весь экран.

Заменим константу на SW_SHOWMINIMIZED. Запустим программу, попробуем найти окно. Оно свернуто и находится на панели задач.

Удалим строку, изменяющую параметр nCmdShow.

### 1.3. Путь к исполняемому файлу программы

Получим спецификацию исполняемого файла программы. Это можно сделать при помощи функции GetModuleFileName.

Параметры этой функции:

- модуль, тип HMODULE, используем дескриптор hInstance,
- буфер для приема пути,
- размер буфера.

В модуле process.cpp объявим константу MAXPATH, задающую максимальный размер буфера спецификации 260.

Затем создадим глобальную переменную szAppPath — буфер для приема спецификации.

Это массив типа TCHAR и размером MAXPATH.

Вызываем функцию в функции onCreate:

```
void onCreate(HINSTANCE hInstance, . . . {
    int foo = 0;
    // получим имя исполняемого модуля
    GetModuleFileName(дескриптор, буфер, размер_буфера);
}
```

Введем в окно Watch переменную с именем "`@err, hr`". Она будет показывать ошибки, если они возникнут при выполнении программы.

Исполняем программу шаг за шагом, пока не будем выполнена функция GetModuleFileName. Убеждаемся, что переменная szAppPath получила значение C:\process\Debug\process.exe.

Зачем это нужно? Программы очень часто работают с файлами, ведь файлы — это хранилища различных данных. Обычно файлы располагаются там же, где находится программа. Тогда нужно вычислить это расположение, например, отрезав от полученной нами спецификации имя файла программы process.exe.

### 1.4. Параметры командной строки

Рассмотрим, как передаются и как получаются параметры командной строки. У нас параметры заданы в настройках проекта. Они не влияют на выполнение непосредственно файла. Чтобы увидеть параметры, будем выводить их в окна сообщений.

Есть два способа получения параметров. Первый — заменить основную функцию на такую, которая передает параметры в виде массива. Это сложно, используем второй способ — получить параметры в массив при помощи функций CommandLineToArgvW и GetCommandLineW:

```
void onCreate(HINSTANCE hInstance, . . . {
    int foo = 0;
    // указатель на строку Unicode
    LPWSTR *szArglist;
    // счетчик параметров
    int cArgs, i = 0;
    szArgList = CommandLineToArgvW(GetCommandLineW(), & cArgs);
    for (; i < cArgs; i++) {
        MessageBoxW(0, szArgList[i], L"Параметр", MB_OK);
    }
    . . .
}
```

Запустим программу, и убедимся, что выводятся 4 значения, первое значение командной строки — это сама команда.

Когда программа запускается из командной строки, параметры передаются в самой командной строке. Откроем FAR, перейдем в каталог C:\process\Debug. Введем команду:

```
process  1  2  3
```

Опять увидим 4 сообщения.

Теперь передадим параметры для запуска программы через проводник. Для этого откроем проводник, найдем исполняемый файл process.exe, выберем в контекстном меню «Копировать». Далее перейдем на Рабочий стол, и выберем в контекстном меню «Вставить ярлык». Когда ярлык появится, в его контекстном меню выберем «Свойства».

В поле «Объект» впишем параметры:

```
C:\process\Debug\process.exe  1  2  3
```

Закроем свойства и дважды щелкнем на ярлык.

### 1.5. Переменные окружения

Получить какую-либо переменную окружения можно при помощи функции GetEnvironmentVariable. Параметры функции:

- название переменной окружения,
- буфер для приема переменной окружения,
- размер буфера.

Создадим массив szEnviron аналогично массиву szAppPath.

Вызываем функцию GetEnvironmentVariable в функции onCreate:

```
void onCreate(HINSTANCE hInstance, . . . {
    . . .
    // переменная окружения
    GetEnvironmentVariable("ComSpec", szEnviron, MAXPATH);
}
```

Исполняем программу, пока функция не будет вызвана. Убедимся, что путь к файлу командной строки получен. Запишем его в отчет. Аналогичным образом получим путь к каталогу временных файлов, переменная окружения TEMP. Запишем этот путь в отчет.

### 1.6. Текущий каталог

Чтобы получить текущий каталог и текущий диск, используем функцию GetCurrentDirectory. Вызываем эту функцию в функции onCreate:

```
void onCreate(HINSTANCE hInstance, . . . {
    . . .
    // текущий диск и каталог
    GetCurrentDirectory(MAXPATH, szEnviron);
}
```

Выполняем программу при помощи F10, пока не будет вызвана эта функция. Запишем в отчет полученный путь. Остановим программу.

### 1.7. Версия операционной системы

Получим версию операционной системы.

Это можно сделать при помощи функции GetVersionEx:

```
void onCreate(HINSTANCE hInstance, . . . {
    . . .
    // версия операционной системы
    OSVERSIONINFO vi;
    memset(&vi, 0, sizeof(vi));
    vi.dwOSVersionInfoSize = sizeof(vi);
    GetVersionEx(&vi);
}
```

Выполняем программу, пока не будет вызвана эта функция. Исследуем содержимое структуры vi. Запишем в отчет полную версию ОС, например, 5.1.2600, платформа 2. Найдем в MSDN, что означает платформа 2, и запишем значение в отчет. Остановим программу.

Версию ОС возвращает также функция GetVersion:

```
// версия операционной системы
DWORD V = GetVersion();
```

Выполняем программу, пока не будет вызвана эта функция. Запишем в отчет полученное в V число. Переведем это число в шестнадцатеричную форму. Старшие три шестнадцатеричных цифры переведем в десятичную форму. Что получилось? Остановим программу.

### 1.8. Создание дочернего процесса

Переходим к созданию дочернего процесса.

Процесс создается функцией CreateProcess.

Параметры функции:

1) имя исполняемого файла (используется редко),
2) командная строка (обычно используется этот параметр),
3) атрибуты безопасности процесса,
4) атрибуты безопасности потока,
5) наследование описателей родительского процесса,
6) флаги создания,
7) указывает на блок окружения процесса,
8) задает рабочий каталог процесса,
9) указатель на структуру STARTUPINFO,
10) указатель на структуру PROCESS_INFORMATION.

В качестве примера будем создавать приложение Блокнот, которому соответствует файл программы notepad.exe.

Сначала создадим текстовый файл. Откроем FAR, перейдем в каталог C:\process, нажмем Shift+F4, введем название файла a.txt, Enter, введем в файл какую-нибудь строку, Escape, Enter. В результате этих действий в каталоге C:\process должен появиться текстовый файл a.txt.

Код функции onCreate больше не понадобится, закомментируем его.

Дальнейшие действия будем выполнять в функции onStart.

Для запуска процесса нужны две структуры данных, объявляем их в начале модуля process.cpp:

```
// информация для запуска
STARTUPINFO si = { sizeof(STARTUPINFO) };
// информация о процессе
PROCESS_INFORMATION pi = { 0 };
```

Переходим в функцию onStart.

Сначала описываем команду для запуска:

```
void onStart() {
    // команда для запуска
    TCHAR cmd[] = "Notepad c:\\process\\a.txt";
}
```

За названием программы следует пробел и параметр командной строки, указывающий имя открываемого файла.

Теперь описываем создание процесса, учитывая, что передаем только команду и структуры STARTUPINFO и PROCESS_INFORMATION:

```
void onStart() {
    // команда для запуска
    TCHAR cmd[] = "Notepad c:\\process\\a.txt";
    // создаем дочерний процесс
    CreateProcess(0, cmd, 0, 0, 0, 0, 0, 0, &si, &pi);
    int foo = 0;
}
```

Последняя строка нужна для точки остановки.

Установим ее и запустим программу F5.

Программа Блокнот должна открыться с файлом a.txt.

Введем в окно Watch переменную pi. Она содержит:

- дескриптор процесса hProcess,
- дескриптор потока hThread,
- идентификатор процесса dwProcessId,
- идентификатор потока dwThreadId.

Откроем диспетчер задач Ctrl+Alt+Del. Убедимся, что идентификатор dwProcessId соответствует PID приложения Блокнот в диспетчере задач.

Продолжим выполнение программы F5, закроем приложение process. Блокнот продолжает работать. Закроем его.

Теперь перейдем в функцию onStop и закроем дескрипторы, чтобы не было утечек памяти, перед этим остановим дочернее приложение:

```
void onStop() {
    // остановим дочернее приложение
    TerminateProcess(pi.hProcess, 0);
    // закрываем дескрипторы
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
}
```

Если теперь запустить приложение process при помощи Ctrl+F5 и затем закрыть его, Блокнот также должен закрыться.

Можно иначе открыть файл блокнотом. Для этого укажем рабочий каталог блокнота и относительную спецификацию файла в команде:

```
void onStart() {
    // команда для запуска
    TCHAR cmd[] = "Notepad a.txt";
    // создаем дочерний процесс
    CreateProcess(0, cmd, 0, 0, 0, 0, 0, "c:\\process", &si, &pi);
    int foo = 0;
}
```

Запускаем программу, убеждаемся, что текстовый файл открывается.

Мы изучили, как порождается дочерний процесс, и как ему передаются параметры через аргументы командной строки.

Код функции onStart заключим в комментарий.

## 1.9. Передача переменных и блока окружения

Извлечем из архива каталог consproc в корневой каталог диска C:.

Откроем проект. Убедимся, что целевой платформой является x86 или win32, и в том, что проект компилируется и запускается. В результате в каталоге C:\consproc\Debug\ должен появиться файл consproc.exe. Этот файл содержит программу, которая выводит переменную окружения testvar.

Сначала добавим в родительский блок окружения переменную окружения testvar со значением 123. Функция onStart:

```
void onStart() {
    // создаем переменную окружения в родительском блоке
    SetEnvironmentVariableA("testvar", "123");
    // создаем процесс consproc
    CreateProcess(0, "c:\\consproc\\Debug\\consproc",
        0, 0, 0, 0, 0, 0, &si, &pi);
}
```

Заметим, что если запускаемый файл имеет расширение .exe, то его можно не указывать. Запускаем программу F5, убеждаемся, что дочерний процесс находит переменную окружения со значением "123".

Заключим код функции onStart в комментарий.

Теперь создадим новый блок окружения из родительского. Получим указатель на блок окружения родительского процесса, заменим его новым блоком окружения, и передадим указатель на блок дочернему процессу:

```
void onStart() {
    // указатель на блок окружения
    char * env = (char*)GetEnvironmentStringsA();
    // новый блок окружения
    strcpy(env, "testvar=456\0abc=xyz\0\0");
    // создаем процесс consproc
    CreateProcess(0, "c:\\consproc\\Debug\\consproc",
        0, 0, 0, 0, env, 0, &si, &pi);
}
```

Запускаем программу F5, убеждаемся, что дочерний процесс находит переменную окружения со значением "456". Обратим внимание, что каждая переменная окружения заканчивается нулем, и в конце блока окружения должен быть еще один ноль. Это стандартный формат блока.

## 1.10. Параметра запуска процесса

Извлечем из архива каталог winsproc в корневой каталог диска C:.

Откроем проект. Убедимся, что целевой платформой является x86 или win32, и в том, что проект компилируется и запускается. В результате в каталоге C:\winsproc\Debug\ должен появиться файл winsproc.exe.

Заключим код функции onStart в комментарий.

При помощи структуры STARTUPINFO задаются параметры запуска процесса. Например, можно задать размеры и положение окна процесса, если только основное окно порождаемого процесса создается при помощи констант CW_USEDEFAULT. Любой задаваемый параметр действует только в том случае, если его действие указано в поле dwFlags.

Мы зададим размеры и положение окна программы winsproc:

```
void onStart() {
    // параметры запуска
    si.dwX = 300;
    si.dwY = 200;
    si.dwXSize = 400;
    si.dwYSize = 400;
    si.dwFlags = STARTF_USESIZE | STARTF_USEPOSITION;
    // создаем процесс winsproc
    CreateProcess(0, "c:\\winsproc\\Debug\\winsproc",
       0, 0, 0, 0, 0, 0, &si, &pi);
}
```

## 1.11. Флаги создания процесса

Заслуживающим внимания является флаг CREATE_SUSPENDED. При этом выполнение основного потока дочернего приложения приостанавливается до того момента, когда будет вызвана функция ResumeThread.

Эту часть работы выполните самостоятельно.

## 1.12. Контрольные вопросы и упражнения

1. Опишите параметры функции WinMain.
2. Как передаются и извлекаются параметры командной строки?
3. Как запустить и остановить дочерний процесс?
4. Как дочернему процессу передать переменную окружения?
5. Какой формат имеет блок окружения?
6. Какую информацию содержит структура PROCESSINFO?
7. Какую информацию содержит структура STARTUPINFO?
8. Создайте пакетный файл dir.bat с командой "dir". Запустите этот файл функцией CreateProcess, указав первым параметром "cmd" (командный процессор), а вторым параметром — имя пакетного файла "dir.bat".
9. Запустите программу Microsoft® Word® и передайте ей документ.

2. Работа OS-204. Создание потоков

Цели:
- изучение управления потоками.
Задачи:
- создание и запуск потоков;
- получение сведений о потоках:
- использование общих структур данных.
Опорные документы:
[3, с.131]

2.1. Рабочее пространство

Скачаем с сайта преподавателя архив для выполнения работы 204.
Извлечем из архива каталог threads в корневой каталог диска C:.
Откроем проект. Убедимся, что целевой платформой является x86 или win32, и в том, что проект компилируется и запускается.

Проект представляет собой консольное приложение threads, и содержит два модуля. Модуль threads.h вспомогательный, он не используется в работе. Модуль threads.cpp является рабочим модулем, здесь выполняется исследование потоков. В начале этого модуля укажите сведения об организации, о себе, о проекте, дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Фамилия Имя Отчество
// OS-204. Потоки
// 01.09.2000

#include "threads.h"

// действия по клавише 1
void on1() {
}

// действия по клавише 2
void on2() {
}

// действия при старте
void onStart() {
}

// вычисляет число в структуре FILETIME
__int64 FT2I64(PFILETIME p) {
    return Int64ShllMod32(p->dwHighDateTime, 32) | p->dwLowDateTime;
}

// точка входа
void mainf() {
}
```

## 2.2. Программный цикл

Сначала нужно организовать программный цикл:

```
void mainf() {
    printf("-- program starts\n");
    onStart();
    int user = 0;
    printf("1 - start\n2 - stop\nEsc - end\n");
    while (1) {
        user = _getch();
        if (user == 27) break;
        if (user == '1') {
            printf("%c\n", user);
            on1();
        } else if (user == '2')
            printf("%c\n", user);
            on2();
        }
    }
    on2();
    printf("\n-- program ends\n");
}
```

Запустим программу, убедимся, что она реагирует на клавиши 1, 2 и Escape. Примерный вид вывода в консоль следующий:

```
-- program starts
1 - start
2 - stop
Esc - end
1
2

-- program ends
```

## 2.3. Данные о текущем процессе и потоке

Каждый процесс имеет первичный, главный поток. При программировании на C/C++ код потока соответствует коду функции main. Другие, вторичные потоки создаются программистом по мере необходимости.

В функции onStart получим идентификаторы процесса и потока:

```
void onStart() {
    DWORD pid = GetCurrentProcessId();
    DWORD tid = GetCurrentThreadId();
    printf("ProcessID = %d\n", pid);
    printf("ThreadID = %d\n", tid);
}
```

Запустим программу. Откроем диспетчер задач, выберем вкладку «Подробнее», в контекстном меню названия столбца таблицы установим флажки «Потоки» и «Базовый приоритет» (если Windows 10). Убедимся, что PID (process identifier) процесса threads совпадает с тем, что выводит программа, и что в процессе threads есть ровно один поток.

Запишем в отчет выведенные значения. Остановим программу.

## 2.4. Создание потока

Для многих программ дополнительные потоки не нужны. Но иногда без них не обойтись. В случае если нужно выполнить длительные вычисления, лучше использовать поток, который этим займется. Особенно это касается интерактивных программ, которые взаимодействуют с пользователем посредством графического интерфейса.

Например, в описании операционной системы Android явно говорится, что не следует загружать основной поток вычислениями, так как это снижает интерактивность потока пользователя.

Часто потоки выполняют фоновые задачи.

Например, в Microsoft® Word® процесс разбивки на страницы выполняется фоновым потоком. Это заметно во время открытия документа. Сначала в статусной строке отображается статистическая информация, потом начинает работать фоновый поток и заметно, как в статусной строке перечисляются страницы.

Наконец, потоки особенно важны для распараллеливания вычислений, особенно в программах, имеющих циклы однотипных вычислений.

Каждому потоку соответствует программная функция, которая описывает код потока, и слово «поток» обозначает поток команд. Например, главному потоку в программе на C/C++ соответствует функция main.

Разным потокам могут соответствовать разные функции, или одна и та же функция. Иначе говоря, в программе может быть несколько потоков, выполняющих одинаковые действия.

Функция потока, за исключением функции главного потока, должна быть описана в соответствии со следующим прототипом:

```
DWORD WINAPI название-функции(PVOID);
```

Функция потока должна возвращать код завершения типа DWORD. По принятому соглашению, значение 0 говорит об успешном завершении, а другие значения регламентируются только для главного потока.

Функции потока могут быть переданы какие угодно параметры через указатель типа PVOID (void*), а если параметров много, их упаковывают в структуру данных.

Поток создает функция CreateThread. Параметры этой функции:

1) атрибуты безопасности,
2) размер стека потока,
3) указатель на функцию потока,
4) указатель на параметры функции потока,
5) флаги создания потока,
6) возвращаемый идентификатор потока.

Описываем функцию потока в соответствии с прототипом в начале модуля threads.cpp. Для завершения потока будет использоваться глобальная переменная stop_sleep:

```
// признак завершения потока
int stop_sleep = 0;
// функция потока sleep
DWORD WINAPI SleepThread(PVOID) {
    DWORD tid = GetCurrentThreadId();
    printf("---- thread %d starts\n", tid);
    while (!stop_sleep) {
        // 0 - отдаем квант другому потоку
        // не 0 - спим столько мс
        Sleep(0);
    }
    printf("---- thread %d ends\n", tid);
    return 0;
}
```

Создание потока описываем в функции on1:

```
void on1() {
    stop_sleep = 0;
    DWORD dwSleepID;
    HANDLE hSleep = CreateThread(0, 0, SleepThread, 0, 0, &dwSleepID);
}
```

Поток остановит функция on2:

```
void on2() {
    stop_sleep = 1;
}
```

Запускаем программу и вводим 1.

Убеждаемся, что поток работает.

Открываем диспетчер задач и убеждаемся, что у процесса threads два работающих потока.

Вводим еще раз 1. Убеждаемся, что запустился еще один поток, а в диспетчере задач число потоков процесса увеличилось.

Вводим 2. Оба потока завершают работу. Вводим Escape.

Можно поэкспериментировать с этими действиями, чтобы убедиться, что потоки завершают работу в случайном порядке. Попытайтесь объяснить, почему в одном случае первым завершает работу первый из созданных потоков, а в другом случае — второй.

## 2.5. Завершение работы потока

Поток нормально завершается, когда выполнение функции потока завершилось. Чтобы видеть момент окончания работы программы, добавим в функцию mainf несколько строк кода:

```
void mainf() {
    . . .
    printf("\n-- program ends\n");
    printf("Press any key to exit.");
    _getch();
    _putch(10);
    _putch(13);
}
```

Закомментируем код функции on2.

Запустим программу.

Введем 1, чтобы создать один поток.

Введем Escape.

Программа завершает работу, а поток — нет, он прерывается. Иначе в консоль было бы выведено сообщение. Поток нормальным образом завершает свою работу, если функция потока завершилась. Наш поток завершается, когда переменная sleep_stop принимает истинное значение.

Может ли поток узнать, что его прерывают? Для определения этого момента есть способ, мы его рассмотрим.

В нашем случае завершить потоки в случае, если программа завершает работу, легко, — достаточно убрать комментарий строки кода функции on2 и добавить вызов функции Sleep:

```
void on2() {
    stop_sleep = 1;
    Sleep(0);
}
```

Основной поток отдает свой квант для того, чтобы потоки программы успели завершиться раньше, чем основной поток выведет свое сообщение «program ends». Если потоков много, одного кванта может и не хватить.

Запустим программу, создадим два потока и введем Escape.

Убедимся, что потоки нормально завершаются.

Сейчас нам нужно ограничить бесконтрольное создание потоков.

В начале модуля опишем константу максимального числа потоков, массив дескрипторов и счетчик потоков:

```
// максимальное число потоков
#define MAX_THREAD 3
// дескрипторы потоков
HANDLE thread[MAX_THREAD];
// счетчик потоков
int tcount = 0;
```

Теперь каждый создаваемый поток будет управляемым, нам известен его дескриптор. Переписываем функцию on1:

```
void on1() {
    if (tcount == MAX_THREAD) return;
    stop_sleep = 0;
    thread[tcount++] = CreateThread(0, 0, SleepThread, 0, 0, 0);
}
```

Запустим программу и убедимся, что создается максимум 3 потока.

Теперь мы можем по завершении программы прервать потоки функцией TerminateThread. Это плохая идея, так как потоки прерываются в работе. Эта функция вызывается для каждого потока автоматически, когда программа завершается. Недостаток такого завершения — возможны утечки памяти, так как не вся память будет освобождена.

Интересно все же узнать, какой код завершения потока возвращается в случае его принудительного завершения. Следующий код принудительно завершает потоки и устанавливает им код завершения, а мы проверяем его:

```
void on2() {
    //stop_sleep = 1;
    //Sleep(0);
    for (int i = 0; i < tcount; i++) {
        // принудительное завершение потока i
        TerminateThread(thread[i], 1);
        DWORD exitcode;
        // код завершения потока i
        GetExitCodeThread(thread[i], &exitcode);
        printf("exitcode = %d\n", exitcode);
    }
}
```

Запустим программу, создадим 3 потока и введем Escape.

Изучим вывод завершения программы. Он может оказаться разным. В случае принудительного завершения потока он возвращает код 0x103, соответствующий константе STILL_ACTIVE (еще работаю). Десятичное значение этой константы равно 259. В одних случаях код завершения будет равен 1, а в других — 259. Следует помнить, что завершение потока таким образом имеет неприятные последствия, в основном, утечку памяти.

## 2.6. Ожидание завершения потоков

Желательно, чтобы программа завершала работу, когда ее вторичные потоки завершат свои действия. Это достигается ожиданием завершения потоков в конце программы. При этом используются дескрипторы потоков, так как они являются сигнальными объектами. Сигнальный объект может находиться в двух состояниях — сигнальном и несигнальном.

Определяет сигнальное состояние объекта одна из двух функций — WaitForSingleObject и WaitForMultipleObjects. Первая функция ждет сигнального состояния одного объекта, вторая — нескольких. Эти функции останавливают выполнение потока, переводя его в состояние блокировки.

Используем вторую функцию для ожидания завершения потоков, при этом нам нужно вернуть завершение нормальным образом:

```
void on2() {
    stop_sleep = 1;
    //Sleep(0);
    // ждем завершения всех вторичных потоков
    WaitForMultipleObjects(tcount, thread, TRUE, INFINITE);
    /* for (int i = 0; i < tcount; i++) {
    } */
}
```

Первый параметр функции WaitForMultipleObjects — число объектов, второй — массив дескрипторов, третий — признак, указывающий, ждать сигнального состояния любого одного или всех объектов, TRUE — ждать всех, четвертый — таймаут ожидания, INFINITE — ждать бесконечно.

В работе программы мало что изменится, но это правильный подход к ее завершению. На практике после этой функции следует разбор причины, по которой эта функция завершилась, это отдельная тема другой работы.

## 2.7. Сигнализация функции потока о завершении

Мы можем сигнализировать функции потока, что функция должна завершить свою работу. При этом функция потока предпринимает необходимые действия для нормального завершения своих действий. Для этого можно использовать специальный сигнальный объект, существующий в API Windows — объект Event (событие). Это простейший объект, который только и может, что находиться в сигнальном или несигнальном состоянии. Мы также используем тот факт, что функции потоков процесса находятся в памяти процесса, и имеют доступ к любой глобальной переменной.

Описываем дескриптор объекта Event в начале модуля:

```
// дескриптор объекта Event
HANDLE StopEvent;
```

В функции onStart создаем событие в несигнальном состоянии с ручным сбросом:

```
void onStart() {
    . . .
    StopEvent = CreateEvent(0, 1, 0, 0);
}
```

Параметры функции CreateEvent: 1) атрибуты безопасности, 2) тип события, TRUE — сбрасываемый вручную, 3) начальное состояние, FALSE — несигнальное, 4) имя объекта, 0 — безымянный.

В функции on2 комментируем переменную stop_sleep, переводим событие в сигнальное состояние, проверяем коды завершения потоков, и закрываем все дескрипторы:

```
void on2() {
    //stop_sleep = 1;
    //Sleep(0);
    SetEvent(StopEvent);
    // ждем завершения всех вторичных потоков
    WaitForMultipleObjects(tcount, thread, TRUE, INFINITE);
    for (int i = 0; i < tcount; i++) {
        DWORD exitcode;
        // код завершения потока i
        GetExitCodeThread(thread[i], &exitcode);
        printf("exitcode = %d\n", exitcode);
        CloseHandle(thread[i]);
    }
    CloseHandle(StopEvent);
}
```

В функции потока определяем состояние события StopEvent при помощи функции WaitForSingleObject и нулевом времени ожидания:

```
DWORD WINAPI SleepThread(PVOID) {
    . . .
    while (!stop_sleep) {
        . . .
        Sleep(0);
        // проверяем состояние сигнального объекта
        if (WaitForSingleObject(StopEvent, 0) == WAIT_OBJECT_0) {
            break;
        }
    }
    printf("---- thread %d ends\n", tid);
    return 0;
}
```

Выполнение функции WaitForSingleObject при нулевом времени ожидании завершается немедленно, и используется для определения состояния объекта. Эта функция возвращает значение константы WAIT_OBJECT_0, если ожидаемый объект находится в сигнальном состоянии.

Запускаем программу и убеждаемся, что все потоки завершаются нормальным образом, возвращая нулевые значения.

## 2.8. Передача параметров потоку

Мы передадим потоку всего один параметр — его порядковый номер.

Сначала изменим функцию потока:

```
DWORD WINAPI SleepThread(PVOID id) {
    DWORD tid = GetCurrentThreadId();
    printf("---- thread %d starts with numebr %d\n", tid, (int)id);
    . . .
    printf("---- thread %d ends with numebr %d\n", tid, (int)id);
}
```

Затем передадим номер создаваемого потока в функции on1:

```
void on1() {
    if (tcount == MAX_THREAD) return;
    stop_sleep = 0;
    int num = tcount + 1;
    thread[tcount++] = CreateThread(0, 0, SleepThread, (PVOID)num, ...
}
```

Запускаем программу и убеждаемся, что потоки выводят свой номер при старте и при завершении.

Потоки различаются тем, что у каждого потока есть своя память для стека и своя локальная память, называемая TLS (Thread Local Storage). Эта память используется в основном при разработке модулей DLL, и на самом деле сохранить что-то в стеке потока гораздо проще. Но иногда нужно связать данные с объектом, а не с потоком. Это сложная тема, поэтому мы приведем только пример использования.

Чтобы сохранить номер потока в TLS, процесс должен создать слот для данных. Слот — это ячейка массива указателей, выделяемого для TLS. Некоторые ячейки заняты, и нужно получить индекс свободной ячейки.

Объявляем этот индекс:

```
// слот TLS
DWORD TLSIndex;
```

В функции onStart получаем свободный слот для потоков:

```
void onStart() {
    . . .
    // запрашиваем свободный слот
    TLSIndex = TlsAlloc();
}
```

В функции потока нужно выделить память для данных и записать в нее значение параметра, указатель на выделенную память записать в слот, по завершении потока освободить память:

```
DWORD WINAPI SleepThread(PVOID id) {
    DWORD tid = GetCurrentThreadId();
    // память 4 байта из кучи
    LPVOID data = LocalAlloc(LPTR, 4);
    // записываем в память значение id
    *(int*)data = (int)id;
    // сохраняем указатель на память в локальном слоте потока
    TlsSetValue(TLSIndex, data);
    printf("---- . . . \n", tid, *(int*)TlsGetValue(TLSIndex));
    . . .
    printf("---- . . . \n", tid, *(int*)TlsGetValue(TLSIndex));
    // освобождаем выделенную из кучи память
    LocalFree(data);
    return 0;
}
```

Запускаем программу, и убеждаемся, что номер потока сохраняется.

Память TLS нужно освобождать по завершении работы потоков:

```
void on2() {
    . . .
    CloseHandle(StopEvent);
    // освобождаем слот
    TlsFree(TLSIndex);
}
```

## 2.9. Времена выполнения потока

Функция GetThreadTimes возвращает времена начала и завершения работы потока, нахождения в режиме ядра, и нахождения в режиме пользователя. Возвращаемые значения — это структуры данных FILETIME. Это 64-битное число, содержащее число 100-наносекунд, отсчитанное от даты 01.01.1601, или просто 100-наносекунды (мне вот интересно, кто запускал поток в 1601 году?).

В функции on2 получим временные характеристики потока:

```
void on2() {
    . . .
    for (int i = 0; i < tcount; i++) {
        DWORD exitcode;
        // код завершения потока i
        GetExitCodeThread(thread[i], &exitcode);
        printf("exitcode = %d\n", exitcode);
        // временные характеристики потока
        FILETIME tstart, tstop, tkernel, tuser;
        GetThreadTimes(thread[i], &tstart, &tstop, &tkernel, &tuser);
        printf("kernel time %d\n", (FT2I64(&tkernel) / 10000));
        printf("user time %d\n", (FT2I64(&tuser) / 10000));
        CloseHandle(thread[i]);
    }
    CloseHandle(StopEvent);
    // освобождаем слот
    TlsFree(TLSIndex);
}
```

Первое выводимое время — это время, затраченное потоком на выполнение кода ОС, второе — это время, затраченное потоком на выполнение кода приложения. Время переведено в миллисекунды.

Запустите программу, создайте и остановите поток. Оцените времена.

Запишите времена в отчет.

## 2.10. Контрольные вопросы и упражнения

1. Как соотносятся процессы и потоки?
2. Что называется первичным и вторичным потоком?
3. Для какой цели создаются потоки?
4. Опишите прототип функции потока.
5. В чем разница между дескриптором и идентификатором потока?
6. Как получить дескриптор и идентификатор потока?
7. Какие значения возвращает поток по завершении?
8. Опишите параметры функции CreateThread.
9. Чем различаются два одинаковых потока?
10. Когда поток завершается нормально?
11. Как потоку передаются параметры?
12. Что такое TLS?
13. Какие времена возвращает функция GetThreadTimes, в чем они измеряются?

3. Работа OS-205. Приоритеты потоков

Цели:
- изучение системы приоритетов Windows.
Задачи:
- изучение базовых приоритетов процесса;
- изучение относительных приоритетов потока;
- приостановка и возобновление выполнения потоков.
Опорные документы:
[3, с.156]

3.1. Рабочее пространство

Скачаем с сайта преподавателя архив для выполнения работы 205.
Извлечем из архива каталог priority в корневой каталог диска C:.
Откроем проект. Убедимся, что целевой платформой является x86 или win32, и в том, что проект компилируется и запускается.
Проект представляет собой оконное приложение priority (рисунок 1).

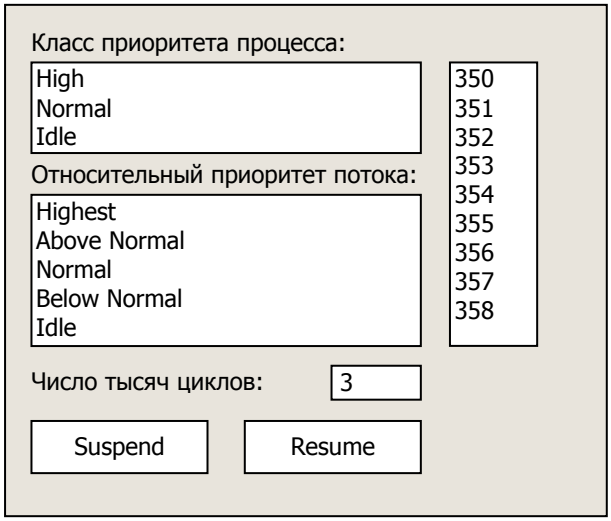


Рисунок 1

В списках «Класс приоритета процесса» и «Относительный приоритет потока» выбираются соответствующие приоритеты. Кнопки «Resume» и «Suspend» предназначены для приостановки и возобновления выполнения потока. В списке, расположенном в правой части окна, отображается активность потока. Кроме того, можно задать задержку активности потока при помощи поля «Число тысяч циклов», которые поток выполняет перед тем, как выполнить основную работу — подсчет итераций.
Шаблон проекта содержит только показанный на рисунке интерфейс.
Функциональность программы требует реализации.
В начале модуля priority.h укажите сведения об организации, о себе, о проекте, дату начала над работой.

## 3.2. Приоритеты

Приоритеты назначаются для того, чтобы дать преимущество более важным потокам, например, системным. Чтобы убедиться в этом, откроем диспетчер задач и выберем вкладку «Подробнее». Если диспетчер задач не отображает приоритеты процессов, в контекстном меню названия столбца таблицы включим флажок «Базовый приоритет». Рассмотрим выполняющиеся процессы, убедимся, что большинство потоков выполняется с нормальным (обычным) приоритетом. Однако должны найтись несколько процессов, приоритет которых выше остальных. Скорее всего, это системные процессы.

В Windows выполняются не процессы, а потоки, и именно потоки конкурируют друг с другом за обладание процессорным временем. Поэтому приоритеты в Windows на самом деле закрепляются за потоками.

Приоритет выражается некоторым числом. Число может быть целым или вещественным (дробным). Каким числом обозначается более высокий приоритет, а каким — более низкий, зависит от конкретной операционной системы. В Windows приоритет — это целое число от 0 до 31, всего 32 приоритета. Низший приоритет нулевой, высший — 31. Шкала приоритетов поделена на две части — процессы и потоки нормального приоритета и процессы и потоки приоритета реального времени. Если приоритет потока выше 15, поток становится потоком реального времени. Эти приоритеты в данной работе не рассматриваются. Для процессов и потоков нормального приоритета выделяются приоритеты от 1 до 15 включительно. Приоритет 0 не назначается никакому потоку, он закреплен за специальным системным потоком, который начинает работать, если в системе не окажется других активных потоков.

При программировании приоритеты задаются не числами, а специальными константами. Числа приведены для ориентировки, а разработчики операционной системы могут изменить конкретные числа. Тем не менее, фактически приоритеты таковы. Поток с обычным приоритетом получает приоритет, равный восьми.

Приоритет в Windows складывается из базового приоритета процесса и относительного приоритета потока. Базовый приоритет процесса нужен для того, чтобы дать преимущество одним процессам над другими. Обычно этого достаточно, чтобы повысить или понизить приоритет потоков процесса. Приоритет потока задается относительно базового приоритета процесса. Если базовый приоритет процесса повысился, автоматически повышается приоритет его потоков, потому что приоритет потока прибавляется к базовому приоритету процесса, или вычитается из него, поэтому он и называется относительным. Фактически изменение приоритета потока выражается как плюс или минус одна или две единицы по отношению к базовому приоритету процесса.

В Windows предусмотрено 6 базовых приоритетов и 7 относительных.

Для назначения базовых приоритетов процессам в модуле winbase.h определены следующие константы:

```
#define NORMAL_PRIORITY_CLASS       0x00000020
#define IDLE_PRIORITY_CLASS         0x00000040
#define HIGH_PRIORITY_CLASS         0x00000080
#define REALTIME_PRIORITY_CLASS     0x00000100
```

Как видим, фактически для обычного разработчика оставлено четыре базовых приоритета из шести: нормальный, ленивый, высокий и приоритет реального времени. Поэтому в интерфейсе программы предусмотрено всего три приоритета: нормальный, ленивый и высокий. Заметим, что правильно название базового приоритета — класс приоритета процесса.

Для относительных приоритетов определены следующие константы:

```
#define THREAD_PRIORITY_LOWEST          THREAD_BASE_PRIORITY_MIN
#define THREAD_PRIORITY_BELOW_NORMAL    (THREAD_PRIORITY_LOWEST+1)
#define THREAD_PRIORITY_NORMAL          0
#define THREAD_PRIORITY_HIGHEST         THREAD_BASE_PRIORITY_MAX
#define THREAD_PRIORITY_ABOVE_NORMAL    (THREAD_PRIORITY_HIGHEST-1)
#define THREAD_PRIORITY_TIME_CRITICAL   THREAD_BASE_PRIORITY_LOWRT
#define THREAD_PRIORITY_IDLE            THREAD_BASE_PRIORITY_IDLE
```

а в файле winnt.h определены константы:

```
#define THREAD_BASE_PRIORITY_LOWRT  15
#define THREAD_BASE_PRIORITY_MAX    2
#define THREAD_BASE_PRIORITY_MIN    -2
#define THREAD_BASE_PRIORITY_IDLE   -15
```

Таким образом, обычный разработчик может выбрать любой из семи относительных приоритетов: низкий, ниже нормального, нормальный, высокий, выше нормального, критический и ленивый. В данной работе используются только шесть относительных приоритетов (не используется критический относительный приоритет).

### 3.3. Базовый приоритет процесса

Теперь приступим непосредственно к установке базового приоритета нашего процесса. Указанные выше константы приоритетов прикреплены к элементам списка. Модуль priority.cpp, функция CreateControls:

```
// классы приоритетов
SendMessage(hWndClass, LB_SETITEMDATA, 0, HIGH_PRIORITY_CLASS);
SendMessage(hWndClass, LB_SETITEMDATA, 1, NORMAL_PRIORITY_CLASS);
SendMessage(hWndClass, LB_SETITEMDATA, 2, IDLE_PRIORITY_CLASS);
```

Эти константы извлекаются в оконной процедуре WndProc:

```
    case LBN_SELCHANGE:
        if ((HWND)lParam == hWndClass) {
            index = SendMessage(hWndClass, LB_GETCURSEL, 0, 0);
            pr = SendMessage(hWndClass, LB_GETITEMDATA, index, 0);
            ProcessPriority(pr);
            return 0;
```

и далее передаются функции ProcessPriority модуля priority.h, в котором мы пишем необходимый код.

Функция onStart модуля priority.h. Получим дескриптор процесса:

```
void onStart() {
    // дескриптор процесса
    MainHandle = GetCurrentProcess();
}
```

Функция ProcessPriority:

```
// класс приоритета процесса
void ProcessPriority(int process_class) {
    int foo = 0;
}
```

Впишем строку int foo, установим на ней точку остановки, запустим программу и выберем в списке классов приоритета «High». Убедимся, что параметр process_class получает значение 128. Выберем в списке «Normal» и убедимся, что process_class равно 32. Выберем в списке «Low» и убедимся, что process_class равно 64. Если все так, вместо строки int foo вызываем функцию для установки приоритета:

```
void ProcessPriority(int process_class) {
    SetPriorityClass(MainHandle, process_class);
}
```

Точка остановки на вызове функции SetPriorityClass.

Запустим программу, выберем в списке базовых приоритетов «High».

Программа остановится.

Введем в окно Watch переменную "`@err,hr`", выполним вызов функции SetPriorityClass при помощи F10 и убедимся, что переменная "`@err,hr`" имеет значение "S_OK", а не "Неверно заданы параметры". Если это так, то приоритет процесса задан. Удалим точку остановки.

Откроем диспетчер задач, найдем процесс priority, и убедимся, что приоритет процесса изменился на высокий. Выбирая в списке разные приоритеты, убедимся, что в диспетчере задач приоритеты также меняются.

### 3.4. Рабочий поток

Для продолжения исследования нам нужен поток. Основная задача потока — генерирование нового числа и запись его в правый список.

Модуль priority.h. Задаем необходимые константы и переменные:

```
// буфер текста
#define MAXBUF 32
char buf[MAXBUF] = {0};
// множитель задержки
#define FACTOR 1000
// рабочий счетчик
int count = 0;
// завершение потока
int stop = 0;
```

Далее описываем функцию потока:

```
// рабочий поток
DWORD WINAPI WorkThread(PVOID) {
    int i, j, k, n, maxc = MAXC;
    while (1) {
        if (stop) break;
        sprintf(buf, "%d", count++);
        // добавим в список строку k
        k = SendMessage(hWndWorks, LB_ADDSTRING, 0, (LPARAM)buf);
        // выделим строку k
        SendMessage(hWndWorks, LB_SETCURSEL, k, 0);
        if (stop) break;
    }
    return 0;
}
```

Как видим, это бесконечный цикл, завершающийся, если переменная stop получит ненулевое значение. В цикле генерируется новое число count, преобразуется в строку, строка добавляется в список и выделяется.

Этот поток можно создать и запустить.

Создание потока описываем в функции onStart:

```
void onStart() {
    // дескриптор процесса
    MainHandle = GetCurrentProcess();
    // рабочий поток
    WorkHandle = CreateThread(0, 0, WorkThread, 0, 0, 0);
}
```

В функции onStop описываем завершение потока:

```
void onStop() {
    // остановим поток
    stop = 1;
    // освободим дескриптор
    CloseHandle(WorkHandle);
}
```

Запустим программу. Убедимся, что рабочий список заполняется.

Откроем диспетчер задач, убедимся, что в процессе priority два потока. Остановим программу Escape.

Теперь нужно проверить количество элементов в списке и счетчик:

```
DWORD WINAPI WorkThread(PVOID) {
    int i, j, k, n, maxc = MAXC;
    while (1) {
        . . .
        SendMessage(hWndWorks, LB_SETCURSEL, k, 0);
        if (k > 9) {
            // удалим первую строку списка
            SendMessage(hWndWorks, LB_DELETESTRING, 0, 0);
        }
        if (count > 100000) count = 0;
        if (stop) break;
    }
    return 0;
}
```

### 3.5. Задержка основной работы потока

Рабочий список заполняется очень активно. Чтобы приостановить основную работу, организуем в функции потока цикл в цикле, каждый цикл имеет maxc×FACTOR итераций. Переменная maxc получит значение, введенное в поле «Число тысяч циклов», а изначально это значение задано константой MAXC. Редактируем функцию рабочего потока:

```
DWORD WINAPI WorkThread(PVOID) {
    int i, j, k, n, maxc = MAXC;
    while (1) {
        if (stop) break;
        // текст поля
        GetWindowText(hWndDelay, buf, MAXBUF);
        if (strlen(buf) > 0) {
            // преобразуем в число
            maxc = atoi(buf);
            if (maxc < 1) maxc = 1;
            if (maxc > 9) maxc = 9;
        }
        // число итераций
        n = maxc * FACTOR;
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                if (stop) return 0;
            }
        }
        sprintf(buf, "%d", count++);
        . . .
        if (stop) break;
    }
    return 0;
}
```

### 3.6. Относительный приоритет потока

Функция ThreadPriority:

```
void ThreadPriority(int thread_relative) {
    SetThreadPriority(WorkHandle, thread_relative);
}
```

Точка остановки на функции SetThreadPriority. Запустим программу, выберем относительный приоритет Below Normal. Убедимся, что параметр thread_relative имеет значение –1. Выполним функцию SetThreadPriority, нажав F10. Убедимся, что переменная "**@err, hr**" в поле Watch имеет значение "S_OK", что означает, что относительный приоритет установлен.

Остановим программу.

### 3.7. Приостановка и возобновление выполнения потока

Для приостановки и возобновления выполнения потока в программе есть две соответствующие кнопки, Suspend и Resume.

Программируем действия по нажатию кнопок:

```
// кнопка Suspend нажата
void SuspendClicked() {
    SuspendThread(WorkHandle);
}
// кнопка Resume нажата
void ResumeClicked() {
    ResumeThread(WorkHandle);
}
```

Здесь все просто. Функция SuspendThread увеличивает счетчик простоев потока, а функция ResumeThread уменьшает его. Если счетчик простоев потока больше нуля, поток приостанавливается.

Запустим программу. Нажмем кнопку Suspend. Поток остановится. Нажмем Suspend еще раз. Ничего не произойдет. Нажмем Resume два раза, чтобы поток снова начал выполняться.

Последнее приготовление заключается в том, чтобы изменить условия запуска рабочего потока:

```
void onStart() {
    // дескриптор процесса
    MainHandle = GetCurrentProcess();
    // рабочий поток
    WorkHandle = CreateThread(0, 0, WorkThread, 0, CREATE_SUSPENDED,...
}
```

При старте программы поток будет приостановлен.

Программу нужно построить, например, запустить и остановить.

Полигон для исследований готов, и проект можно закрыть.

### 3.8. Исследование приоритетов

Откроем FAR. Перейдем в каталог C:\priority\Debug. Нажмем Shift+F4 и создадим пакетный файл 2.bat для запуска двух копий программы:

```
start priority.exe
start priority.exe
•
```

Точка • показывает конец пакетного файла.

Закроем редактор Escape, Enter (Enter сохранит файл).

Будем выполнять несколько тестов. Для каждого теста в отчет записываем номер теста, условия теста и примерное время его выполнения. Это время будет зависеть от того, насколько быстро вы запустите поток второй программы. После запуска программ их нужно расположить в ряд, при этом левую программу будем называть первой, а правую второй, и запускать будем сначала первую программу, затем вторую.

Важно, чтобы не было открыто лишних окон. Мы должны наблюдать работу при одинаковых условиях, когда ни одна из программ не обладает фокусом. Чтобы убрать фокус с программ, нужно щелкать в Рабочий стол.

После запуска программ нужно установить число тысяч циклов в максимальное значение 9, так проще будет наблюдать счетчики.

### 3.8.1. Тест 1

Запускаем пакетный файл 2.bat.
Устанавливаем число тысяч циклов 9.
Устанавливаем приоритеты теста.
Условия теста:
- программа 1 имеет базовый приоритет Idle,
- программа 2 имеет базовый приоритет High.

Записываем время начала теста, после чего нажимаем кнопку Resume программы 1, затем кнопку Resume программы 2.
Ждем, когда счетчик программы 2 обгонит счетчик программы 1.
Записываем время окончания теста, останавливаем программы.
Желательно, чтобы разница значений счетчиков программ находилась в пределах 8-10 единиц, иначе ждать придется долго. Вероятно, сначала нужно потренироваться запускать потоки.

### 3.8.2. Тест 2

Запускаем пакетный файл 2.bat.
Устанавливаем число тысяч циклов 9.
Устанавливаем приоритеты теста.
Условия теста:
- программа 1 имеет относительный приоритет Idle,
- программа 2 имеет относительный приоритет Highest.

Записываем время начала теста, после чего нажимаем кнопку Resume программы 1, затем кнопку Resume программы 2.
Ждем, когда счетчик программы 2 обгонит счетчик программы 1.
Записываем время окончания теста, останавливаем программы.

### 3.8.3. Тест 3

Запускаем пакетный файл 2.bat.
Устанавливаем число тысяч циклов 9.
Устанавливаем приоритеты теста.
Условия теста:
- программа 1 имеет базовый приоритет Idle,
- программа 1 имеет относительный приоритет Idle,
- программа 2 имеет базовый приоритет High.
- программа 2 имеет относительный приоритет Highest.

Записываем время начала теста, после чего нажимаем кнопку Resume программы 1, затем кнопку Resume программы 2.
Ждем, когда счетчик программы 2 обгонит счетчик программы 1.
Записываем время окончания теста, останавливаем программы.

### 3.8.4. Тест 4

Создадим копию файла 2.bat с именем 3.bat.

Добавим в файле 3.bat запуск третьей копии программы:

```
start priority.exe
start priority.exe
start priority.exe
•
```

Запускаем пакетный файл 3.bat.

Устанавливаем число тысяч циклов 9.

Устанавливаем приоритеты теста.

Условия теста:

- программа 1 имеет относительный приоритет Idle,
- программа 2 имеет приоритеты по умолчанию,
- программа 3 имеет базовый приоритет High,
- программа 3 имеет относительный приоритет Highest.

Записываем время начала теста, нажимаем кнопку Resume программы 1, затем кнопку Resume программы 2 и кнопку Resume программы 3.

Записываем счетчик программы 1.

Ждем, когда счетчик программы 3 обгонит счетчик программы 2 и записываем время. Устанавливаем базовый приоритет программы 1 High, относительный приоритет Highest. Записываем счетчик программы 2. Ждем, когда счетчик программы 1 обгонит счетчик программы 2. Записываем время, останавливаем программы.

В оставшееся время можно поиграть с тремя программами, наблюдая, как они соперничают друг с другом, когда меняются приоритеты.

## 3.9. Контрольные вопросы и упражнения

1. Что называется приоритетом?
2. Для чего нужны приоритеты?
3. Каковы числовые значения приоритетов в Windows?
4. Каков диапазон числовых значений нормальных приоритетов?
5. Из чего складывается приоритет потока?
6. Почему приоритет потока называется относительным?
7. Назовите классы приоритетов процесса.
8. Назовите относительные приоритеты потока.
9. Как работают функции SuspendThread и ResumeThread?

## 4. Работа OS-206. Синхронизация потоков

Цели:
- изучение механизмов синхронизации Windows.

Задачи:
- исследование параллельного доступа к разделяемым ресурсам;
- синхронизация при помощи спин-блокировки;
- синхронизация при помощи Interlocked-функций;
- синхронизация при помощи критических секций.

Опорные документы:
[1, с.144], [2, с.128], [3, с.187]

### 4.1. Тестирующее приложение

Скачаем с сайта преподавателя архив для выполнения работы 206.

Извлечем из архива каталог crisect в корневой каталог диска C:.

Откроем проект. Убедимся, что целевой платформой является x86 или win32, и в том, что проект компилируется и запускается.

Все действия происходят в модуле crisect.h.

Сначала приводим сведения об организации, о себе, о проекте, дату начала работы:

```
// crisect.h
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Фамилия Имя Отчество
// OS-206. Синхронизация потоков
// 01.09.2000

// число циклов
#define MAXW 1
// число потоков
#define MAXT 2
// дескрипторы потоков
// разделяемый ресурс
// блокирующая переменная
// функция потока
// основная функция
void mainf() {
}
```

Описываем необходимые переменные.

Дескрипторы потоков — массив размером MAXT типа HANDLE.

Разделяемый ресурс — переменная shared типа int.

Блокирующая переменная — переменная locks типа int.

Далее описываем функцию потока:

```
DWORD WINAPI Thread(PVOID id) {
    int c = 0, i = 0, r = 0, n = (int)id;
    return c;
}
```

Переменная c — это счетчик обращений к разделяемой переменной. Этот счетчик является также кодом завершения потока.

Переменная i — итератор цикла, который выполняет поток.

Переменная r — считанное значение разделяемой переменной.

Переменная n — номер потока, передаваемый при его создании.

Цель первого исследования — выяснить, как происходит взаимодействие потоков при параллельной работе и одновременном (попеременном) обращении к разделяемой глобальной переменной. Для этого создадим некоторое количество потоков, целью которых будет увеличить значение разделяемой переменной на единицу некоторое количество раз.

В функции потока описываем цикл по переменной i с количеством итерация MAXW. В цикле подсчитываем обращение к разделяемому ресурсу (которым является переменная share), модифицируем ресурс, выводим считанное значение в консоль для наблюдения, примерно так:

```
    for (i = 0; i < MAXV; i++) {
        // подсчитываем обращение
        c++;
        // считываем значение ресурса
        r = share
        // выводим считанное значение
        printf("%d\tby\t%d\tshare\n", r, n);
        // модифицируем ресурс
        share = share + 1;
    }
```

Теперь в основной функции нужно сформировать цикл для создания потоков, обеспечить ожидание завершения потоков, просуммировать коды завершения потоков, и вывести в консоль текущее значение разделяемого ресурса и число обращений к нему:

```
// основная фунция
void mainf() {
    int i = 0, count = 0, j = 0;
}
```

Переменная i — это переменная цикла, переменная count — счетчик обращений, переменная j — номер, передаваемый потоку.

Сначала убедимся, что количество потоков не превышает 64, так как это максимальное количество сигнальных объектов, которые могут быть использованы в функции WaitForMultipleObjects (это число может зависеть от версии операционной системы):

```
void mainf() {
    int i = 0, count = 0, j = 0;
    int t = MAXT > 64 ? 64 : MAXT;
}
```

Переменная t принимает либо значение MAXT, либо 64. Значение этой переменной используется как число создаваемых потоков.

Далее формируем цикл для создания потоков:

```
void mainf() {
    . . .
    // создаем t потоков
    for (i = 0; i < t; i++) {
        j = i + 1;
        thread[i] = CreateThread(0, 0, Thread, (PVOID)j, 0, 0);
    }
}
```

Далее в функции mainf нужно дождаться их завершения:

```
void mainf() {
    . . .
    // некоторое время спим
    Sleep(1);
    // ждем завершения потоков
    WaitForMultipleObjects(MAXT, thread, 1, INFINITE);
}
```

Теперь формируем цикл по потокам, считываем коды завершения, суммируем их, закрываем дескрипторы:

```
void mainf() {
    . . .
    for (i = 0; i < MAXT; i++) {
        DWORD exitcode = 0;
        // код завершения
        GetExitCodeThread(thread[i], &exitcode);
        // сумма обращений
        count += exitcode;
        // освобождаем дескриптор
        CloseHandle(thread[i]);
    }
}
```

Наконец, выводим значения интересующих нас переменных:

```
void mainf() {
    . . .
    // разделяемый ресурс 1
    printf("share = %d\n", share);
    // разделяемый ресурс 2
    printf("locks = %d\n", locks);
    // число обращений
    printf("count = %d\n", count);
}
```

Полигон для исследований готов.

### 4.2. Тест 1

Условия теста: число циклов 1, число потоков 2.

Построим проект.

Открываем FAR. Переходим в каталог C:\crisect\Debug. В нем находится исполняемый файл crisect.exe. Открываем указанный каталог в правой половине FAR при помощи Alt+F2 или Ctrl+F2, а левую половину закрываем при помощи Ctrl+F1. Курсор установим на исполняемый файл.

Нажимаем Enter. Программа выполнится, а в левой части окна выведется результат. Запускаем программу несколько раз. Нужно добиться следующего результата:

```
0   by  1   share
0   by  2   share
share = 2
locks = 0
count = 2
```

Видим, что оба потока увеличили ресурс на единицу, так как окончательное значение ресурса равно двум, но считывали в итерации при этом нулевое значение. Попробуйте объяснить, почему так произошло.

Условия теста, результат теста и объяснение запишем в отчет.

Рассмотрим, как происходит работа потока на уровне ассемблера. Откроем свойства проекта, C/C++, Output Files, Assembler output, выбираем Assembly, Machine Code and Source. Построим проект.

Открываем FAR, переходим в каталог C:\crisect\crisect\Debug, открываем файл crisect.cod (приведена часть кода, примерно 111 строка):

```
; 24   :          // модифицируем ресурс
; 25   :          share = share + 1;
  0006d a1 00 00 00 00   mov     eax, DWORD PTR ?share@@3HA ; share
  00072 83 c0 01         add     eax, 1
  00075 a3 00 00 00 00   mov     DWORD PTR ?share@@3HA, eax ; share
```

Рассмотрим выполнение оператора share = share + 1, приведенное в сгенерированном коде. Сначала значение переменной share записывается в регистр процессора eax, к регистру прибавляется единица, значение в регистре записывается обратно в переменную. В этой части кода есть потенциальная опасность, ведущая к неправильному взаимодействию параллельно работающих потоков.

Как известно, потоки выполняются попеременно, для работы им выделяются кванты времени. Рассмотрим, как могут взаимодействовать два наших потока. Пусть сначала квант получает поток 1. Он выполняет часть кода функции потока до инструкции, включая ее:

```
  0006d a1 00 00 00 00   mov     eax, DWORD PTR ?share@@3HA ; share
```

Поток 1 успел прочитать текущее значение переменной share.

В этот момент происходит смена потоков, и управление переходит к потоку 2, который выполняет код функции потока от начала до конца. При этом поток 2 устанавливает значение переменной share, равное 1.

Управление возвращается к потоку 1. Он завершает выполнение кода функции потока:

```
  00072 83 c0 01         add     eax, 1
  00075 a3 00 00 00 00   mov     DWORD PTR ?share@@3HA, eax ; share
```

Поскольку в регистре eax этого потока было считано значение 0, поток 1 завершает выполнение, устанавливая значение 1 переменной share.

Таким образом, правильная работа потоков не гарантируется, и программист обязан:

- понимать, в каком случае возможна ситуация неправильного взаимодействия потоков,

- знать, когда и какие средства можно использовать для предотвращения подобной ситуации.

Приведенный случай не возник в результате теста 1, но это по чистой случайности, в тесте 2 мы должны смоделировать эту ситуацию. Однако описанный случай должен помочь понять, почему в тесте 1 потоки считали одно и то же значение переменной share.

### 4.3. Тест 2

Условия теста: число циклов 1, число потоков 2.

Внесем небольшое изменение в код функции потока:

```
// модифицируем ресурс
share = r + 1;
```

То есть, заменим строку share = share + 1 приведенной выше строкой. Таким образом мы разнесли во времени момент считывания значения переменной share и момент ее записи. Это справедливо, так как потоки обычно выполняют множество действий, связанных со значением разделяемого ресурса, занимающих определенное время.

Построим проект.

Открываем FAR. Переходим в каталог C:\crisect\Debug. Курсор установим на исполняемый файл. Нажимаем Enter. Программа выполнится, а в левой части окна выведется результат. Запускаем программу несколько раз. Нужно добиться следующего результата:

```
0   by  1   share
0   by  2   share
share = 1
locks = 0
count = 2
```

Как видим, действительно произошло нарушение правильного взаимодействия, так как выполнение кода в промежутке между считыванием переменной и ее записью было прервано операционной системой. Запишем в отчет условия и результат теста, объяснение смоделированной ситуации.

### 4.4. Тест 3

Условия теста: число циклов 1, число потоков 25.

Условия теста запишем в отчет.

Построим проект.

Открываем FAR. Переходим в каталог C:\crisect\Debug.

Запускаем программу 10 раз, каждый раз записывая в отчет полученное значение переменной share.

### 4.5. Тест 4

Условия теста: число циклов 25, число потоков 2.

Условия теста запишем в отчет.

Построим проект.

Открываем FAR. Переходим в каталог C:\crisect\Debug.

Запускаем программу 10 раз, каждый раз записывая в отчет полученное значение переменной share.

### 4.6. Критическая секция потока и синхронизация

Синхронизация — это приостановка выполнения потока до наступления некоторого события. Синхронизация позволяет сформировать правильную последовательность выполнения потоков.

В нашем случае нужно обеспечить нахождение в критической секции только одного потока. Критическая секция потока — та часть кода функции потока, которая обращается к разделяемому ресурсу, от момента его считывания, до момента его изменения, включая сами эти моменты.

Для повышения производительности системы потоки не должны занимать критическую секцию надолго, иначе другие потоки, ожидающие входа в нее, будут простаивать. Кроме того, потоки должны освобождать критическую секцию, сигнализируя другим потокам, что критическая секция доступна.

Не следует путать критическую секцию потока с критической секцией операционной системы. В первом случае это часть кода функции потока, а во втором — механизм, используемый для синхронизации.

Синхронизация должна выполняться для каждого отдельного разделяемого ресурса. Если поток обращается к нескольким разделяемым ресурсам, он имеет несколько критических секций.

Для взаимного исключения нахождения нескольких потоков в одной и той же критической секции можно использовать несколько механизмов. В первую очередь эти механизмы делятся на синхронизацию потоков одного процесса, и синхронизацию потоков разных процессов. Первый случай проще, так как все потоки одного процесса находятся в одной памяти.

Для синхронизации потоков одного процесса можно использовать:

- запрещение прерываний;
- блокирующую переменную;
- критическую секцию операционной системы.

### 4.7. Спин-блокировка

Спин-блокировка — это прием, позволяющий ограничить вход в критическую секцию потока при помощи блокирующей переменной.

Пусть есть разделяемый ресурс share.

Пусть есть блокирующая переменная locks.

Тогда, для того, чтобы получить доступ к ресурсу share, будем проверять, какое значение имеет блокирующая переменная. Если она истинна, то будем считать, что вход в критическую секцию закрыт, а если ложна, то будем считать вход открытым. Этот прием называется спин-блокировкой потому, что в ожидании входа поток бесполезно «вращается» в цикле проверки значения блокиратора.

Модифицируем поток следующим образом:

```
DWORD WINAPI Thread1(PVOID id) {
    int c = 0, i = 0, r = 0, n = (int)id;
    for (i = 0; i < MAXW; i++) {
        // подсчитываем обращение
        c++;
        // сбрасываем счетчик
        r = 0;
        // ожидаем входа (спин-блокировка)
        while (locks) r++;
        // закрываем критическую секцию
        locks = 1;
        // число циклов ожидания
        printf("%d\tby\t%d\tlocks\n", r, n);
        // считываем значение ресурса
        r = share;
        // выводим считанное значение
        printf("%d\tby\t%d\tshare\n", r, n);
        // модифицируем ресурс
        share = r + 1;
        // открываем критическую секцию
        locks = 0;
    }
    return c;
}
```

Полигон готов.

Построим проект.

### 4.8. Тест 5

Условия теста: число циклов 1, число потоков 2.

Запишем в отчет «спин-блокировка» и условия теста.

Открываем FAR. Переходим в каталог C:\crisect\Debug.

Запускаем программу 10 раз, каждый раз записывая в отчет полученное значение переменной locks для потоков 1 и 2 (не финальное значение).

### 4.9. Тест 6

Условия теста: число циклов 1, число потоков 25.

Запишем в отчет «спин-блокировка» и условия теста.

Открываем FAR. Переходим в каталог C:\crisect\Debug.

Запускаем программу 10 раз, каждый раз записывая в отчет максимальное полученное значение переменной locks для потока 1 или 2 (не финальное значение).

### 4.10. Тест 7

Условия теста: число циклов 25, число потоков 25.

Запишем в отчет «спин-блокировка» и условия теста.

Открываем FAR. Переходим в каталог C:\crisect\Debug.

Запускаем программу столько раз, сколько нужно для того, чтобы финальные значения share и count не совпали. Запишем в отчет, сколько раз пришлось запустить программу.

### 4.11. Interlocked-функции

В результате предыдущих тестов мы должны убедиться, что блокирующая переменная не является надежным механизмом синхронизации.

Поэтому обращаемся к справочникам и находим, что существуют так называемые Interlocked-функции. Это функции, которые управляют целочисленной переменной в режиме монопольного доступа. При этом гарантируется, что если один поток изменяет переменную, то никакой другой поток не сможет этого сделать.

Модифицируем функцию потока следующим образом:

```
DWORD WINAPI Thread1(PVOID id) {
    int c = 0, i = 0, r = 0, n = (int)id;
    for (i = 0; i < MAXW; i++) {
        // подсчитываем обращение
        c++;
        // сбрасываем счетчик
        r = 0;
        // ожидаем входа (спин-блокировка)
        while (InterlockedCompareExchange((LONG*)&locks, 1, 0) == 1);
        // число циклов ожидания
        printf("%d\tby\t%d\tlocks\n", r, n);
        // считываем значение ресурса
        r = share;
        // выводим считанное значение
        printf("%d\tby\t%d\tshare\n", r, n);
        // модифицируем ресурс
        share = r + 1;
        // открываем критическую секцию
        InterlockedCompareExchange((LONG*)&locks, 0, 1);
    }
    return c;
}
```

Кроме того, нужно объявить переменную locks как используемую для разделяемого доступа:

```
// блокирующая переменная
volatile int locks = 0;
```

Функция InterlockedCompareExchange проверяет и изменяет значение блокирующей переменной следующим образом. Если значение переменной равно третьему параметру, то устанавливается значение второго параметра. Функция возвращает начальное значение переменной.

Первый раз функция ожидает, когда переменная будет равна нулю, выполняя спин-блокировку. Второй раз функция меняет значение 1 на 0.

Построим проект.

### 4.12. Тест 8

Условия теста: число циклов 25, число потоков 25.

Запишем в отчет «спин-блокировка Interlocked» и условия теста.

Открываем FAR. Переходим в каталог C:\crisect\Debug.

Запускаем программу несколько раз, убеждаемся, что значения переменных share и count всегда совпадают. Запишем в отчет результат и примерное время выполнения теста.

### 4.13. Критическая секция ОС

Если бы задача потока состояла бы только в том, чтобы увеличить счетчик share, то можно было бы использовать Interlocked-функцию, применяя ее непосредственно к переменной share.

Проблема Interlocked-функции в том, что она может изменять значение только целочисленной переменной, в то время как разделяемый ресурс редко оказывается такой структурой. Кроме этого, спин-блокировка впустую растрачивает процессорное время.

Чаще всего для синхронизации потоков одного процесса используют критическую секцию ОС. Это структура данных CRITICAL_SECTION, управляют которой следующие функции:

InitializeCriticalSection — выполняет инициализацию структуры,

EnterCriticalSection — выполняет вход в критическую секцию кода,

LeaveCriticalSection — освобождает критическую секцию кода,

DeleteCriticalSection — удаляет критическую секцию.

Функция EnterCriticalSection пытается занять критическую секцию, выполняя некоторое время спин-блокировку в надежде, что критическая секция освободится в ближайшее время. Связано это с тем, что критическая секция переводит поток в режим блокировки, а переход в режим ядра занимает примерно 1000 тактов процессора, и столько же для выхода оттуда. Если удастся занять критическую секцию во время спин-блокировки, производительность получится выше.

Есть также функция TryAndEnterCriticalSection, которая пытается войти в критическую секцию. Если критическая секция занята, функция немедленно возвращается, и поток может заниматься другими полезными делами до следующего обращения к функции.

Объявим критические секции в начале модуля:

```
// критические секции
CRITICAL_SECTION cs1; // для share
CRITICAL_SECTION cs2; // для locks
```

В начале функции mainf инициализируем их:

```
void mainf() {
    // инициализируем критические секции
    InitializeCriticalSection(&cs1);
    InitializeCriticalSection(&cs2);
    . . .
}
```

В конце функции mainf удаляем их:

```
void mainf() {
    . . .
    // удаляем критические секции
    DeleteCriticalSection(&cs1);
    DeleteCriticalSection(&cs2);
}
```

Модифицируем функцию потока следующим образом:

```
DWORD WINAPI Thread1(PVOID id) {
    int c = 0, i = 0, r = 0, n = (int)id;
    for (i = 0; i < MAXW; i++) {
        // подсчитываем обращение
        c++;
        // входим в критическую секцию share
        EnterCriticalSection(&cs1);
        // считываем значение ресурса
        r = share;
        // обрабатываем ресурс
        Sleep(0);
        // модифицируем ресурс
        share = r + 1;
        // выходим из критической секции share
        LeaveCriticalSection(&cs1);
        // выводим считанное значение
        printf("%d\tby\t%d\tshare\n", r, n);
        // работаем
        Sleep(1);
        // входим в критическую секцию locks
        EnterCriticalSection(&cs2);
        // считываем значение ресурса
        r = locks;
        // обрабатываем ресурс
        Sleep(0);
        // модифицируем ресурс
        locks = r + 1;
        // выходим из критической секции locks
        LeaveCriticalSection(&cs2);
        // выводим считанное значение
        printf("%d\tby\t%d\tshare\n", r, n);
    }
```

Мы добавили вызовы функции Sleep(0), которые отдают квант какому-нибудь другому потоку. Это имитирует работу потока.

Заметим, что для каждого отдельного ресурса мы используем свою критическую секцию. В данном случае можно было бы обойтись и одной критической секцией, но в общем случае так получится не всегда.

### 4.14. Тест 9

Условия теста: число циклов 25, число потоков 25.

Запишем в отчет «CS» и условия теста.

Открываем FAR. Переходим в каталог C:\crisect\Debug.

Запускаем программу несколько раз, убеждаемся, что значения переменных share, locks и count всегда совпадают. Запишем в отчет результат и примерное время выполнения теста.

Следует обратить внимание на то, в каком порядке потоки получают доступ к разделяемым ресурсам.

### 4.15. Тест 10

Условия теста: число циклов 1, число потоков 64.

Запишем в отчет «CS» и условия теста.

Открываем FAR. Переходим в каталог C:\crisect\Debug.

Запускаем программу несколько раз, убеждаемся, что значения переменных share, locks и count всегда совпадают. Запишем в отчет результат и примерное время выполнения теста.

Закомментируйте вызовы функции Sleep и повторите тест.

### 4.16. Контрольные вопросы и упражнения

1. Что такое синхронизация?
2. Что называется критической секцией программы?
3. Назовите три условия, при которых параллельно выполняющиеся потоки будут правильно взаимодействовать.
4. Как классифицируются механизмы синхронизации.
5. Назовите механизмы синхронизации потоков одного процесса.
6. Что такое Interlocked-функция?
7. Как работает функция InterlockedCompareExchange?
8. Что называется блокирующей переменной?
9. Как работает блокирующая переменная?
10. Что называется спин-блокировкой? В чем ее недостаток?
11. Перечислите порядок применения функций для управления критическими секциями ОС.
12. Как работает функция EnterCriticalSection?
13. Как работает функция TryAndEnterCriticalSection?

## 5. Работа OS-207. Задача о философах

Цели:
- синхронизация потоков.

Задачи:
- моделирование ситуации deadlock;
- моделирование ситуации life-lock;
- моделирование правильного взаимодействия.

Опорные документы:
[2, с.150]

### 5.1. Шаблон проекта

Скачаем с сайта преподавателя архив для выполнения работы 207.

Извлечем из архива каталог philmeal в корневой каталог диска C:.

Откроем проект. Убедимся, что целевой платформой является x86 или win32, и в том, что проект компилируется и запускается.

Эта работа выполняется самостоятельно как домашнее задание.

Все действия выполняются в модуле philmeal.h. Сначала приведите в нем сведения об организации, о себе, о проекте, дату начала работы.

### 5.2. Задача об обедающих философах

Задача была предложена в 1965 году Э. Дейкстрой как экзаменационное упражнение для студентов. Суть задачи следующая.

Есть пять философов, которые обдумывают свои философские проблемы. Некоторое время обдумав, они идут в столовую, чтобы покушать.

В столовой есть круглый стол, вокруг него 5 стульев, на столе 5 чашек и между чашками 5 вилок (вилка по-английски fork). В центре стола есть чаша с едой, которая никогда не опустеет. Еда — очень длинные спагетти, есть которые можно *только двумя вилками одновременно*.

Чтобы покушать, философ берет сначала левую вилку, затем правую вилку, кушает, кладет сначала правую вилку, затем левую вилку.

Все философы сели кушать одновременно.

Вариант 1. Если после того, как взята левая вилка, правая вилка занята, философ ждет ее освобождения. Философы умирают в ожидании правых вилок, ситуация называется deadlock (голодная смерть).

Вариант 2. Если после того, как взята левая вилка, правая вилка занята, философ кладет левую вилку обратно. Философы умирают от истощения от физической работы, вариант называется life-lock.

Вариант 3. Если философ пришел покушать, то он покушает, возможно, после некоторого ожидания.

Следует реализовать все три варианта программы.

Отчет о домашней работе должен описание тех механизмов, которые были использованы в разных вариантах для достижения цели, и не должен содержать кода программы.

6. Работа OS-208. Синхронизация процессов

Цели:
- изучение механизмов синхронизации Windows.
Задачи:
- синхронизация потоков при помощи мьютекса;

Опорные документы:
[1, с.155], [2, с.128, с.139], [3, с.229]

6.1. Шаблон проекта

Скачаем с сайта преподавателя архив для выполнения работы 208.

Извлечем из архива каталог mutex в корневой каталог диска C:.

Откроем проект. Убедимся, что целевой платформой является x86 или win32, и в том, что проект компилируется и запускается.

Все действия происходят в модуле mutex.h.

Сначала приводим сведения об организации, о себе, о проекте, дату начала работы:

```
// crisect.h
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Фамилия Имя Отчество
// OS-208. Синхронизация процессов
// 01.09.2000
```

6.2. Тестирующее приложение

В этой работе будем синхронизировать выполнение потоков разных процессов, и называть это синхронизацией процессов.

Цель первой части работы — обеспечить синхронизацию доступа к разделяемому ресурсу из разных процессов при помощи мьютекса.

Разделяемым ресурсом служит текстовый файл. Конкурирующие процессы записывают в него свои строки, но странным образом, по одному символу за одну итерацию. Каждому процессу присваивается свой номер *n* от нуля до 9, и процесс пытается записать строку из MAXW символов, соответствующих номеру. Если номер процесса 0, процесс записывает строку вида "000", завершая запись переводом строки. Если *m* процессов запишут свои строки, файл должен содержать *m* строк длиной MAXW каждая. Посмотрим, получится ли процессам это осуществить без синхронизации.

Файл будет располагаться в каталоге C:\mutex\Debug. Чтобы при отладке этот каталог был рабочим, нужно установить его в свойствах проекта: Debugging, установим Command Arguments равным 1 (номер процесса), а Working Directory равным ..\Debug.

Нам нужна функция, которая либо создаст новый файл, либо откроет его, если файл был создан другим процессом. Модуль mutex.h.

Описываем эту функцию перед функцией mainf:

```
// открывает файл для добавления
HANDLE open(const char * file) {
    HANDLE file = 0;
    file = CreateFile(file, FILE_APPEND_DATA, FILE_SHARE_WRITE, 0,
        OPEN_EXISTING, 0, 0);
    if (file != INVALID_HANDLE_VALUE) return file;
    file = CreateFile(file, FILE_APPEND_DATA, FILE_SHARE_WRITE, 0,
        CREATE_NEW, 0, 0);
    return file;
}
```

Сначала функция пытается файл открыть. Если это не удается, функция пытается файл создать. Параметром функции является путь к файлу.

Описываем этот путь, вспомогательный буфер, и константу числа итераций в начале модуля:

```
char path[32] = "share.txt";
char buff[32] = "";
// число итераций
#define MAXW 3
```

Переходим к функции mainf (первичный поток процесса):

```
void mainf(int n) {
    int i = 0;
    HANDLE file = 0, mutex = 0;
    DWORD written = 0, err = 0;
}
```

Переменная i — итератор цикла, переменная file — дескриптор файла, переменная mutex — дескриптор мьютекса, переменная written — число записанных байт, переменная err — код, возвращаемый GetLastError.

Далее в mainf формируем цикл записи строк:

```
    for (i = 1; i <= MAXW; i++) {
        HANDLE file = open((const char *)path);
        if (file == INVALID_HANDLE_VALUE) {
            err = GetLastError();
            sprintf(buff, "Process #%d failed due to %d\n", n, err);
            MessageBoxA(0, buff, "Error open file", MB_OK);
            return;
        }
        if (i == MAXW) {
            sprintf(buff, "%d\r\n", n);
            WriteFile(file, buff, 3, &written, 0);
        } else {
            sprintf(buff, "%d", n);
            WriteFile(file, buff, 1, &written, 0);
        }
        Sleep(1);
        FlushFileBuffers(file);
        CloseHandle(file);
    }
```

В цикле сначала открываем файл и анализируем, удалось его открыть, или нет. Если не удалось, выводим соответствующее сообщение.

Затем записываем в файл одну цифру, а если строка последняя, то еще и символ конца строки \r\n. В конце итерации сбрасываем файловый буфер в файл на диске и закрываем файл.

Полигон для первого тестирования готов.

Построим проект.

Открываем FAR. Переходим в каталог C:\mutex\Debug. Там находится исполняемый файл программы mutex.exe. Shift+F4, вводим название нового файла start.bat, вводим следующий текст:

```
del share.txt
start mutex.exe 0
start mutex.exe 2
start mutex.exe 4
•
```

Точка • показывает конец файла.

Нажимаем Escape, Enter (Enter сохранит файл).

Таким образом, будем тестировать три процесса.

### 6.3. Тест 1

Условия теста: процессов 3, итераций 3.

Запишем в отчет условия теста.

Переходим в FAR, каталог C:\mutex\Debug.

Запускаем пакет start.bat.

По окончании открываем файл share.txt при помощи F4 и записываем в отчет получившуюся последовательность символов. Обратите внимание на то, сколько раз в файл записан конец строки. Для этого закроем файл Escape, откроем для просмотра F3, и если отображается не дамп, то дополнительно F4. Коды символов конца строки равны 0D и 0A.

Имеет смысл экспериментировать со временем сна и его положением.

### 6.4. Синхронизация при помощи мьютекса

Мьютекс предназначен для синхронизации потоков разных процессов. Нам он как раз подходит. Его можно использовать и для синхронизации потоков одного процесса, но мьютекс намного «тяжеловеснее» критической секции, поэтому в этом качестве его не используют.

Работа мьютекса основана на принципе обладания. Мьютекс занят, если некоторый поток владеет им. Фактически в объект ядра мьютекса записывает идентификатор обладающего им потока. Кроме этого, объект мьютекса содержит счетчик рекурсии, в который записывается количество захватов мьютекса. Мьютекс создает функция CreateMutex, освобождает функция ReleaseMutex, а захват выполняется при помощи Wait-функции, или во время создания указанием значения TRUE вторым параметром.

Переходим к функции mainf и описываем создание мьютекса. Чтобы несколько потоков могли получить доступ к одному и тому же мьютексу,

последний должен быть поименованным. Тогда во время создания мьютекса несколькими потоками первый из них создает мьютекс, другие потоки узнают о том, что мьютекс создан, через возвращаемое значение функции GetLastError, равное ERROR_ALREADY_EXISTS.

Описываем процесс создания мьютекса:

```
void mainf(int n) {
    int i = 0;
    HANDLE file = 0, mutex = 0;
    DWORD written = 0, err = 0;
    // создаем мьютекс
    mutex = CreateMutex(0, 0, "sharefile");
    if (mutex == 0) {
        err = GetLastError();
        sprintf(buff, "Mutex error in process #%d due to %d", n, err);
    } else {
        err = GetLastError();
        if (err == ERROR_ALREADY_EXISTS) {
            sprintf(buff, "Mutex exists in process #%d", n);
        } else {
            sprintf(buff, "Mutex created in process #%d", n);
        }
    }
    MessageBoxA(0, buff, "Mutex creation", MB_OK);
    . . .
}
```

Запускаем программу, убеждаемся, что мьютекс создается.

Далее мьютекс нужно захватить (перед циклом):

```
    // захватываем мьютекс
    WaitForSingleObject(mutex, INFINITE);
```

Перед завершением функции мьютекс нужно освободить:

```
void mainf(int n) {
    . . .
    for (i = 1; i <= MAXW; i++) {
        . . .
    }
    // освобождаем мьютекс
    ReleaseMutex(mutex);
}
```

В конце цикла записи в файл добавляем две строки:

```
void mainf(int n) {
    . . .
    for (i = 1; i <= MAXW; i++) {
        . . .
        sprintf(buff, "Iteration %d in process #%d", i, n);
        MessageBoxA(0, buff, "Iteration", MB_OK);
    }
    // освобождаем мьютекс
    ReleaseMutex(mutex);
}
```

Полигон для второго теста готов. Построим проект.

### 6.5. Тест 2

Условия теста: процессов 3, итераций 3.
Запишем в отчет «мьютекс» и условия теста.
Переходим в FAR, каталог C:\mutex\Debug.
Запускаем пакет start.bat.
Записываем в отчет все возникшие сообщения.

Заметим, что окна сообщений перекрывают друг друга так, что первое сообщение является последним после закрытия предыдущих окон. Записывайте в отчет окна в том порядке, в котором они являются активными. Выждите несколько секунд прежде, чем закрывать окна сообщений.

По окончании открываем файл share.txt при помощи F4 и записываем в отчет получившуюся последовательность символов. Сравниваем записи сообщений с записями в файле share.txt.

### 6.6. Контрольные вопросы и упражнения

1. Чем различается синхронизация потоков и процессов?
2. Опишите объект мьютекс.
3. Опишите принцип работы мьютекса.
4. Опишите параметры функции CreateMutex.
5. Сравните мьютекс с критической секцией ОС.

## 7. Работа OS-209. Синхронизация ресурсов

Цели:
- изучение механизмов синхронизации Windows.

Задачи:
- синхронизация потоков при помощи мьютекса;
- синхронизация ресурсов при помощи семафора;
- взаимодействие мьютексов и семафоров.

Опорные документы:
[1, с.148], [2, с.128, с.137], [3, с.227]

### 7.1. Шаблон проекта

Скачаем с сайта преподавателя архив для выполнения работы 209.
Извлечем из архива каталог wr в корневой каталог диска C:.
Откроем каталог wr. В каталоге содержатся:
- проект компонента «читатель» reader,
- проект компонента «писатель» writer,
- включаемый файл queue.h, задающий число ресурсов,
- пакетные файлы для запуска тестов test*x*.bat,
- текстовые файлы, имитирующие ресурсы,
- пакетный файл clean.bat для очистки текстовых файлов.

### 7.2. Задача о писателях и читателях

Это классическая задача о взаимодействии параллельных потоков при ограниченном числе ресурсов. Есть несколько процессов, называемых «писателями», и несколько процессов, называемых «читателями». Хорошей интерпретацией является модель печати документов. В этом случае в качестве писателей выступают приложения, которые хотят напечатать свои документы, а в качестве читателей выступают принтеры, которые могут распечатать эти документы. Как правило, число писателей велико, а читателей много меньше. Кроме того, в системе может быть очередь, либо одна для всех принтеров, либо для каждого принтера своя. Размер очереди ограничен объемом выделенной памяти, поэтому возникает проблема синхронизации ресурсов — нельзя поставить в очередь документ, если очередь полна, с другой стороны — нельзя печатать документ, если очередь пуста.

Писатели могут посылать свои документы в очередь целиком или по частям, а принтеры должны печатать документы так, чтобы страницы документов не перепутывались.

Для моделирования задачи мы разрабатываем два компонента — проект writer моделирует писателя, а проект reader моделирует читателя. На первом этапе будем считать очередь неограниченной, и проведем несколько тестов без синхронизации потоков.

Затем синхронизируем потоки писателей и читателей, с тем, чтобы записываемая и читаемая информация не пропадала и не дублировалась, после чего перейдем к синхронизации ресурсов для того, чтобы ограничить память, выделяемую для очереди.

## 7.3. Моделируемая система

Мы планируем запустить четырех писателей и двух читателей.

Писатели записывают в файл que.txt свои сообщения в произвольном порядке, так, как будут распределены их потоки операционной системой.

Очередь сообщений que.txt будет очищаться читателями, поэтому мы будем дублировать сообщения в контрольном файле bak.txt, а для наблюдения дублировать сообщения в консоли. У каждого процесса будет своя консоль (cmd.exe), которая будет оставаться на экране, пока мы не закроем ее при помощи клавиши Escape (или любой другой).

Сообщения будут иметь строгий формат, состоящий ровно из восьми символов, и это для нашей системы важно. Текст сообщения состоит из двух чисел, например, "01-001". Первые две цифры указывают номер процесса писателя, следующие три цифры указывают номер итерации, или номер сообщения. За этими шестью знаками должны следовать символы возврата каретки, десятичный код 13, и перевода строки, код 10.

Писатели записывают свои сообщения в конец очереди, то есть в конец файла que.txt, а читатели считывают сообщения по одному из начала очереди, то есть из начала файла que.txt, после чего «печатают» их, записывая модифицированные сообщения в контрольный файл out.txt. Модификация сообщения заключается в том, что читатель в начале сообщения приписывает номер своего процесса (две цифры) и стрелку " => ".

Записав сообщение, писатель «обдумывает» следующее сообщение в течение некоторого количества миллисекунд, заданного ему параметром delay функции mainf. Читатель, получив сообщение, «обрабатывает» его в течение некоторого количества миллисекунд, заданного ему аналогичным параметром.

Писатель записывает количество сообщений, заданное ему параметром attempts функции mainf. Читатель выполняет некоторое количество циклов чтения, заданное ему аналогичным параметром. Мы не можем поставить читателя в бесконечное чтение, как это должно было бы быть в реальной системе, для наших тестов мы задаем читателям число циклов чтения заведомо (много) больше, чем число генерируемых сообщений.

## 7.4. Проект писателя

Начнем с компонента, моделирующего писателя.

Откроем проект writer. Убедимся, что целевой платформой является x86 или win32, и в том, что проект компилируется и запускается.

Все действия происходят в модуле writer.h. Сначала укажем сведения об организации, о себе, о проекте и дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Студент Имя Отчество
// OS-209. Синхронизация ресурсов
// Компонент "Писатель"
// 01.09.2000
#pragma once
// длина очереди
#include "..\..\maxque.h"

// n        - номер процесса
// attempts - число циклов записи
// delay    - задержка Sleep
void mainf(int n, int attempts, int delay) {
}
```

Заметим, что функция mainf вызывается с параметрами, определяющими характеристики процесса. Эти параметры нужно задать в параметрах проекта. Откроем параметры проекта, раздел Debugging, параметр Command Arguments, запишем в него три числа через пробел: 1 4 1. Эти числа означают номер процесса 1, число сообщений 4, задержка 1 мс.

Описываем в начале модуля буфер и пути к файлам:

```
// буфер сообщения для записи в очередь
__declspec(align(64)) char quebuff[16] = "";
// путь к файлу очереди
char quepath[] = "c:\\wr\\que.txt";
// путь к контрольному файлу
char bakpath[] = "c:\\wr\\bak.txt";
```

Мы выравниваем границу буфера для ускорения работы с ним. В некоторых случаях такое выравнивание является обязательным, а граница может быть кратной, например, размеру страницы памяти (4 Кбайт).

Далее нам нужна функция для открытия файла в режиме общего доступа, для разделяемого чтения и записи. Разделяемый доступ означает, что файл может быть использован разными процессами для записи и (или) для чтения. На самом деле писателю не нужно разделяемое чтение, но читателю нужно, и нельзя один и тот же файл открыть для писателя с разделяемым доступом для записи, а для читателя открыть его с разделяемым доступом для чтения, у кого-то случится нарушение доступа, ошибка 80:

```
// открывает файл в режиме общего доступа
HANDLE open(LPCSTR path) {
    return CreateFile(path, GENERIC_WRITE | GENERIC_READ,
        FILE_SHARE_WRITE | FILE_SHARE_READ, 0,
        OPEN_EXISTING, 0, 0);
}
```

Заметим, что файл должен существовать. Предполагается, что файлы имитируют ресурсы, которые существуют все время, а не иногда, как файлы. Поэтому наличие файлов в каталоге wr является обязательным.

Теперь нам нужна функция, которая выполнит запись буфера с сообщением в указанный файл:

```
// записывает сообщение
void writen(HANDLE file, LPCSTR msg) {
    DWORD wb = 0;
    // указатель в конец файла
    SetFilePointer(file, 0, 0, FILE_END);
    // добавляем сообщение
    WriteFile(file, (LPVOID)msg, strlen(msg), &wb, 0);
    // завершаем
    FlushFileBuffers(file);
}
```

Первый параметр функции — это дескриптор открытого файла, второй параметр — буфер с сообщением. Сначала нужно установить текущую позицию в файле на его конец при помощи функции SetFilePointer. Затем записать сообщение при помощи функции WriteFile, и принудительно сбросить файловый буфер операционной системы в файл на диске при помощи функции FlushFileBuffers. Переменная wb (written bytes) показывает количество записанных байтов.

Вспомогательные функции готовы, переходим к моделированию алгоритма писателя. Сначала открываем необходимые файлы, затем формируем цикл из attempts итераций, по завершении цикла закрываем файлы:

```
void mainf(int n, int attempts, int delay) {
    // открываем файлы
    HANDLE bakfile = open(bakpath);
    if (bakfile == INVALID_HANDLE_VALUE) {
        printf("bak-fails=%d\n", GetLastError());
        return;
    }
    HANDLE quefile = open(quepath);
    if (quefile == INVALID_HANDLE_VALUE) {
        printf("que-fails=%d\n", GetLastError());
        return;
    }
    // цикл записи сообщений в очередь
    for (int i = 0; i < attempts; i++) {
    }
    // закрываем файлы
    CloseHandle(quefile);
    CloseHandle(bakfile);
}
```

В цикле:

- формируем сообщение в буфере quebuff функцией sprintf,
- записываем сообщение в файл очереди функцией writen,
- записываем сообщение в контрольный файл функцией writen,
- записываем сообщение в консоль функцией printf, и
- отправляем писателя «подумать» функцией Sleep.

Порядок действий не особо важен, но вывод в консоль должен происходить после записи в очередь:

```
    for (int i = 0; i < attempts; i++) {
        // формируем сообщение
        sprintf(quebuff, "%02d-%03d\r\n", n, (i + 1));
        // сообщение в que-файл
        writen(quefile, quebuff);
        // контрольное сообщение в bak-файл
        writen(bakfile, quebuff);
        // контрольное сообщение в консоль
        printf(quebuff);
        // работаем над сообщением
        Sleep(delay);
    }
```

Проект писателя для первых тестов готов. Рекомендуется сохранить его в отдельном каталоге своего носителя информации.

Запустим программу.

Откроем FAR. Перейдем в каталог C:\wr. Убедимся, что файл que.txt сформирован, и его размер равен в точности 4×8 = 32 байта.

Если это не так, вероятно, в программе есть ошибка.

### 7.5. Проект читателя

Откроем проект reader. Убедимся, что целевой платформой является x86 или win32, и в том, что проект компилируется и запускается.

Все действия происходят в модуле reader.h. Сначала укажем сведения об организации, о себе, о проекте и дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Студент Имя Отчество
// OS-209. Синхронизация ресурсов
// Компонент "Читатель"
// 01.09.2000
#pragma once
// длина очереди
#include "..\..\maxque.h"
// длина записи
#define RECW 8

// n        - номер процесса
// attempts - число циклов записи
// delay    - задержка Sleep
void mainf(int n, int attempts, int delay) {
}
```

Функция mainf в этом проекте имеет такие же параметры, как и в проекте writer, поэтому нужно установить в настройках проекта Command Arguments в значение 1 40 1 (номер процесса 1, число циклов чтения 40, задержка процесса 1мс).

Описываем необходимые структуры данных.

Они включают в себя буфер для перезаписи файла que.txt, буфер для прочитанной записи из файла que.txt, а также спецификации файлов que.txt и out.txt:

```
// буфер для перезаписи файла
__declspec(align(64)) char temp[16] = "";
// буфер прочитанного сообщения
__declspec(align(64)) char quebuff[16] = "";
// путь к файлу очереди
char quepath[] = "c:\\wr\\que.txt";
// путь к файлу результата
char outpath[] = "c:\\wr\\out.txt";
```

Скопируем из проекта writer две первых функции, open и writen, и вставим их в проект reader перед функцией mainf.

Для чтения первой записи и ее удаления из файла очереди que.txt описываем функцию read1 перед функцией mainf:

```
// считывает первое сообщение в quebuff
// удаляет первое сообщение из файла
int read1(HANDLE file, int n) {
    return 1;
}
```

Первый параметр — это дескриптор открытого файла que.txt, второй параметр — номер процесса, который нужен для формирования результирующего сообщения. В функции описываем необходимые переменные:

```
int read1(HANDLE file, int n) {
    DWORD rb = 0, wb = 0;
    DWORD len = GetFileSize(file, 0);
    if (len == 0) {
        return 0;
    }
    return 1;
}
```

Переменная rb (read bytes) принимает количество прочитанных байтов функцией ReadFile, переменная wb принимает количество записанных байтов функцией WriteFile, переменная len — размер файла. Если размер равен нулю, функция завершается, файл пуст.

Далее нужно сформировать в буфере quebuff идентификатор читателя и стрелку, затем приписать к буферу первую запись файла:

```
    char * p = quebuff;
    // идентификатор читателя
    p += sprintf(p, "%02d => ", n);
    // указатель на начало
    SetFilePointer(file, 0, 0, FILE_BEGIN);
    // читаем первую запись
    ReadFile(file, (LPVOID)p, RECW, &rb, 0);
```

Удалить первую запись файла в принципе несложно, если есть буфер подходящего размера, но это проблема. Поэтому будем перезаписывать запись за записью, помещая каждую последующую запись файла на место предыдущей записи.

Для этой цели нам нужно два указателя позиции в файле. Указатель rp — это положение в файле для считывания записи, указатель wp — положение в файле для записи:

```
    // новая длина файла
    len -= RECW;
    // указатели чтения и записи
    DWORD rp = RECW, wp = 0;
```

Далее формируем цикл перезаписи, перемещая указатели и укорачивая длину файла:

```
    while (len > 0) {
        // следующую запись
        SetFilePointer(file, rp, 0, FILE_BEGIN);
        ReadFile(file, (LPVOID)temp, RECW, &rb, 0);
        // запишем вперед
        SetFilePointer(file, wp, 0, FILE_BEGIN);
        WriteFile(file, (LPVOID)temp, RECW, &wb, 0);
        // смещаем указатели
        rp += RECW;
        wp += RECW;
        // остаток файла
        len -= RECW;
    }
```

По завершении цикла перезаписи усекаем файл:

```
    // усекаем файл
    SetFilePointer(file, wp, 0, FILE_BEGIN);
    SetEndOfFile(file);
    // завершаем
    FlushFileBuffers(file);
```

Остается описать последовательность действий читателя в mainf.

Сначала отправляем читателя спать 10 миллисекунд. Далее открываем файлы que.txt и out.txt, и формируем цикл чтения:

```
void mainf(int n, int attempts, int delay) {
    Sleep(10);
    // открываем файлы
    HANDLE quefile = open(quepath);
    if (quefile == INVALID_HANDLE_VALUE) {
        printf("que-fails=%d\n", GetLastError());
        return;
    }
    HANDLE outfile = open(outpath);
    if (outfile == INVALID_HANDLE_VALUE) {
        printf("out-fails=%d\n", GetLastError());
        return;
    }
    // цикл чтения сообщений из очереди
    for (int i = 0, m = 1; i < attempts; i++, m++) {
    }
    CloseHandle(outfile);
    CloseHandle(quefile);
}
```

В цикле чтения:

- считываем первое сообщение при помощи read1,

- проверяем результат: если ноль, отправляем процесс отдохнуть 1 мс, после чего переходим к следующей итерации,

- выводим сообщение в файл out.txt,
- выводим сообщение в консоль для контроля:

```
// считываем первое сообщение, если есть
int result = read1(quefile, n);
// проверяем результат
if (result == 0) {
    // пусто
    printf("empty file %03d\n", m);
    Sleep(1);
    continue;
}
// обрабатываем сообщение
Sleep(delay);
// сообщение в out-файл
writen(outfile, quebuff);
// контрольное сообщение в консоль
printf(quebuff);
```

Проект читателя для первых тестов готов. Рекомендуется сохранить его в отдельном каталоге своего носителя информации.

Приступаем к отладке. Точка остановки на строке:

```
DWORD rp = RECW, wp = 0;
```

В окне Watch вводим переменные `@err,hr`, quebuff и temp.

Запустим программу F5.

Убедимся, что буфер quebuff содержит `01 => 01-001`.

Точка остановки на строке в конце цикла перезаписи:

```
// остаток файла
len -= RECW;
```

Нажмем F5. Убедимся, что буфер temp содержит `01-002`.

Нажмем F5. Убедимся, что буфер temp содержит `01-003`.

Нажмем F5. Убедимся, что буфер temp содержит `01-004`.

Уберем точки остановки, нажмем F5 для завершения программы.

Переходим в FAR. Убедимся, файл que.txt имеет длину 0, а файлы bak.txt и out.txt содержат все записи, от первой до четвертой.

Переходим в окно консоли, проверяем сообщения читателя, находим контрольные сообщения прочитанных записей.

Если все правильно, полигон для первых тестов готов.

### 7.6. Тесты 1-3

Первая серия тестов выполняется при помощи пакетных файлов test1, test2 и test3. Для каждого теста в отчет записываются: номер теста, условия теста, прочитанные в пакетном файле, результаты. Результаты для первых двух тестов включают в себя: число строк в файлах bak.txt и out.txt, есть ли нарушения записей в файле out.txt, число пропущенных записей в обоих файлах, число дублированных записей в обоих файлах. Кроме того, нужно сопоставить порядок записей в файлах bak и out, совпадает или нет.

Кроме того, нужно оценить, как распределяется нагрузка между читателями в виде количества записей, прочитанных читателями.

Для третьего теста результаты включают в себя только число строк в обоих файлах и примерное распределение нагрузки читателей.

Нарушения записей в файле out.txt имеют вид "`01 => 01 =>`". Они могут появляться и не появляться. Попробуйте выполнить тесты несколько раз, нарушения записей могут появиться. Выполняя тест несколько раз, в отчет запишите результат какого-нибудь одного теста.

Просматривайте и анализируйте сообщения в консолях процессов.

## 7.7. Синхронизация процессов

Переходим к синхронизации процессов. В проекте писателя writer создадим пару мьютексов, и применим их для синхронизации доступа к файлам que и bak. Создаем мьютексы после открытия файлов в mainf:

```
// создаем мьютексы
HANDLE bakmutex = CreateMutex(0, 0, "bakfile");
if (bakmutex == 0) {
    printf("bakmutex error due to %d", GetLastError());
}
HANDLE quemutex = CreateMutex(0, 0, "quefile");
if (quemutex == 0) {
    printf("quemutex error due to %d", GetLastError());
}
```

Синхронизируем доступ к файлам в цикле записи:

```
    // формируем сообщение
    sprintf(quebuff, "%02d-%03d\r\n", n, (i + 1));
    // захватываем мьютекс que
    WaitForSingleObject(quemutex, INFINITE);
    // сообщение в que-файл
    writen(quefile, quebuff);
    // освобождаем мьютекс que
    ReleaseMutex(quemutex);
    // захватываем мьютекс bak
    WaitForSingleObject(bakmutex, INFINITE);
    // контрольное сообщение в bak-файл
    writen(bakfile, quebuff);
    // освобождаем мьютекс bak
    ReleaseMutex(bakmutex);
    // контрольное сообщение в консоль
    printf(quebuff);
    // работаем над сообщением
    Sleep(delay);
```

Переходим в проект читателя reader. Создаем два мьютекса. Первый мьютекс имеет имя не bakfile, а outfile, не перепутайте, это важно, а дескриптор мьютекса называется outmutex, а не bakmutex. После создания мьютексов применяем их для синхронизации доступа к файлам que и out, по аналогии с проектом писателя.

По завершении построим оба проекта.

### 7.8. Тесты 4-6

Вторая серия тестов выполняется при помощи пакетных файлов test1, test2 и test3, только номера тестов записываем как 4, 5 и 6. Оцениваем распределение нагрузки, наличие всех записей в файлах bak и out, совпадение порядка записей в этих файлах.

### 7.9. Синхронизация ресурсов

Наша последняя задача — ограничить число ресурсов, то есть число записей в файле очереди que. Максимальное количество записей задано константой MAXQ, определенной в файле maxque.h.

Для целей ограничения доступа к ресурсам используют механизм синхронизации операционной системы «семафор». Например, семафор может ограничивать число пользователей системы, или число открываемых соединений при доступе к серверу.

У семафора есть счетчик ресурсов, и он «знает» их количество. Если счетчик ресурсов больше нуля, семафор открыт, иначе закрыт. Создается семафор функцией CreateSemaphore, второй параметр которой — начальное значение счетчика ресурсов, третий параметр — число ресурсов, четвертый — имя семафора. Первый параметр, атрибуты безопасности, принимаем равным нулю.

Вход в семафор выполняется при помощи Wait-функции, так как же, как и вход в мьютекс. При этом счетчик ресурсов семафора уменьшается на единицу. Поэтому в семафор можно войти ограниченное число раз, если он не освобождается. Освобождается семафор функцией ReleaseSemaphore, подобной функции освобождения мьютекса, но она отличается по своему действию от ReleaseMutex. Во-первых, в функции нужно указать число возвращаемых ресурсов, например, один. Во-вторых, функция увеличивает счетчик ресурсов, но не бесконечно, как счетчик рекурсии мьютекса, а до максимального значения числа ресурсов.

Нам нужно два семафора. Один будет считать число свободных ресурсов, назовем его empty, а второй — число занятых ресурсов, назовем его full. Писатель может войти в семафор empty, если есть свободный ресурс. Записав сообщение, он должен выйти из семафора full, увеличивая таким образом число занятых ресурсов.

Читатель поступает наоборот. Он входит в семафор full, если есть занятый ресурс, а прочитав запись, должен выйти из семафора empty, указывая таким образом, что ресурс освободился.

Начальное значение счетчика ресурсов семафора empty равно MAXQ, а начальное значение счетчика ресурсов семафора full равно нулю.

Начинаем с проекта писателя writer.

Сначала в функции mainf создаем два семафора, располагая код после создания мьютексов:

```
// создаем семафоры
HANDLE empty = CreateSemaphore(0, MAXQ, MAXQ, "empty");
if (empty == 0) {
    printf("empty semaphore error due to %d", GetLastError());
}
HANDLE full = CreateSemaphore(0, 0, MAXQ, "full");
if (full == 0) {
    printf("full semaphore error due to %d", GetLastError());
}
```

Входим в семафор empty сразу после формирования сообщения в цикле записи сообщений. Время ожидания входа зададим как бесконечное:

```
    // формируем сообщение
    sprintf(quebuff, "%02d-%03d\r\n", n, (i + 1));
    // входим в семафор empty
    DWORD wait = WaitForSingleObject(empty, INFINITE);
```

Выходим из семафора full после освобождения мьютекса quefile:

```
    . . .
    // освобождаем мьютекс que
    ReleaseMutex(quemutex);
    // выходим из семафора full
    ReleaseSemaphore(full, 1, 0);
    // захватываем мьютекс bak
    WaitForSingleObject(bakmutex, INFINITE);
    . . .
```

Проект писателя готов. Построим его.

Скопируем создание семафоров.

Переходим к проекту читателя reader.

Вставим создание семафоров в проект reader, в функцию mainf, после создания мьютексов.

Входим в семафор full в начале цикла чтения сообщений:

```
for (int i = 0, m = 1; i < attempts; i++, m++) {
    // входим в семафор full
    DWORD wait = WaitForSingleObject(full, 100);
```

Обратим внимание, что мы задаем время ожидания входа 100 мс.

Если мы выберем бесконечное время ожидания, то при пустом файле процесс читателя зависнет навсегда. Теперь мы должны проверить, почему завершилась Wait-функция: семафор открыт (или открылся) или же истекло время ожидания при закрытом семафоре:

```
    // разбираем причину входа
    if (wait == WAIT_TIMEOUT) {
        // очередь пуста
        printf("empty queue %03d\n", m);
        continue;
    }
```

Если причина входа в Wait-функцию — истекло время ожидания, выводим в консоль контрольное сообщение.

Выходим из семафора empty после освобождения мьютекса quefile:

```
// освобождаем мьютекс que
ReleaseMutex(quemutex);
// выходим из семафора empty
ReleaseSemaphore(empty, 1, 0);
// проверяем результат
if (result == 0) {
```

Проект читателя готов. Построим его.

Приступаем к отладке.

Перейдем в FAR. Запустим пакетный файл clean.bat.

Проект писателя writer. Точка остановки на строке цикла записи:

```
Sleep(delay);
```

Запустим проект F5. Нажмем F5 два раза.

Убедимся, что в файл que записано три сообщения и его размер 24.

Нажмем F5. Убедимся, что ожидает входа в семафор empty, который закрылся, — максимальное число ресурсов достигнуто.

Проект читателя reader. Точка остановки на строке цикла чтения:

```
ReleaseSemaphore(empty, 1, 0);
```

Запустим проект F5. После прихода в точку остановки нажмем F10, освобождая семафор empty.

Проект писателя writer. Убедимся, что писатель вошел в семафор empty. Нажмем F5, чтобы писатель завершил свою работу.

Проект читателя reader. Уберем точку остановки, нажмем F5, чтобы читатель завершил свою работу.

### 7.10. Тесты 7-9

Третья серия тестов выполняется при помощи пакетных файлов test4, test5 и test6, номера тестов 7, 8 и 9. Для каждого теста записываем в отчет номер теста, условия теста, результаты. В качестве результата оцениваем, как распределилась нагрузка читателей (число прочитанных ими записей), какие читатели каких писателей обслужили. Кроме того, следует отметить, в каком порядке сформированы записи по номерам писателей.

### 7.11. Тесты 10-12

Проект писателя writer. Изменим условия входа в семафор empty:

```
DWORD wait = WaitForSingleObject(empty, 100);
// разбираем причину входа
if (wait == WAIT_TIMEOUT) {
    // очередь заполнена
    printf("full queue %03d\n", (i + 1));
    i--;
    continue;
}
```

Определив, что истекло время таймаута, выводим в консоль сообщение, что очередь заполнена.

При этом мы уменьшаем счетчик цикла, переменную *i*, чтобы число записей в очередь не уменьшилось, и переходим к следующей итерации.

Построим проект.

Выполняем тесты 10, 11 и 12 при помощи пакетных файлов test4, test5 и test6. Результаты тестов 10 и 11 записываем в отчет:

- число сообщений «full queue» отдельно по каждому из писателей,
- число сообщений «empty queue», появившихся в начале цикла, а не в конце, отдельно по каждому из читателей.

В тесте 12 никаких результатов не требуется.

## 7.12. Тест 13

Проект читателя reader.

Изменим взаимное расположение входа в семафор и в мьютекс:

```
for (int i = 0, m = 1; i < attempts; i++, m++) {
    // захватываем мьютекс que
    WaitForSingleObject(quemutex, INFINITE);
    // входим в семафор full
    DWORD wait = WaitForSingleObject(full, 100);
```

Мы перенесли строчки, описывающие вход в мьютекс quefile.

Построим проект. Запустим пакетный файл test4.

По завершении работы программ посмотрим на консоли читателей. Один из читателей закончил работу, а второй читатель завис. Закроем консоль того читателя, который завершил работу. Убедимся, что другой читатель завершает свою работу. Запишем в отчет результат. Проанализируем работу читателей и приведем в отчете объяснение результата.

Данный тест показывает, что порядок входа в семафор и мьютекс имеет значение, и нарушать его нельзя. Сначала входим в семафор, потом в мьютекс. Выходить из мьютекса и из семафора можно в произвольном порядке, если это не нарушает правильное взаимодействие с другими синхронизирующими объектами.

## 7.13. Контрольные вопросы и упражнения

1. Опишите задачу о писателях и читателях.
2. В чем заключается синхронизация ресурсов?
3. Опишите семафор.
4. Опишите функцию создания семафора.
5. Опишите функцию освобождения семафора.
6. Опишите взаимодействие семафоров empty и full.

8. Литература

1. Сетевые операционные системы / В. Г. Олифер, Н. А. Олифер. — СПб.: Питер, 2002. — 544 с. :ил.

2. Таненбаум Э. Современные операционные системы. 2-е изд. — СПб.: Питер, 2002. — 1040 с.: ил.

3. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Пер. с англ. — 4-е изд. — СПб.: Питер; Издательско-торговый дом «Русская Редакция», 2001. — 753 с.: ил.

---

Пономарев Владимир Вадимович
Практикум по операционным системам
Учебно-методическое пособие

Отпечатано с готового оригинал-макета
Издательство ОТИ НИЯУ МИФИ, 2019
Тираж 11 экз.