Лабораторная работа № 3

Синхронизация процессов и потоков

# Введение

Проектирование и разработка многопоточных программ — одна из самых сложных областей программирования, но в то же время сильно востребованная на практике. С одной стороны, необходимо понимать и выстраивать логику одновременной и взаимосвязанной работы нескольких алгоритмов — это требует как теоретических знаний, так и опыта, сноровки. С другой стороны, логические ошибки в таких программах (если они есть) могут проявляться или нет практически случайным образом, что сильно затрудняет отладку.

Описание механизма потоков и процессов ОС имеется в лекционном курсе, а ниже представлены лишь краткие сведения о многопоточном программировании в ОС Windows.

# Теоретические сведения

Ключевыми сущностями многопоточного программирования являются состояния потока — работа или ожидание, примитивы синхронизации, а также понятие атомарности.

В грубом приближении сложность многопоточного программирования проистекает из того, что в любой точке программы её выполнение может прерваться и перейти к другому потоку или процессу. Упорядочить работу потоков можно, опираясь на действия, которые прерваны быть не могут — атомарные операции (см. ниже).

Из лекционного курса известно, что запущенный поток может не работать по двум причинам: внешней — когда очередь выполнения потока еще не наступила, и внутренней — когда управление не передается потоку, пока не наступило некое событие. В многопоточном программировании ожиданием называют второй случай.

Примитив синхронизации — это системный объект, к которому могут обращаться несколько потоков (или процессов), чтобы каждому определить, можно ли продолжать работу или следует перейти к ожиданию. Из лекционного курса известен пример — критическая секция: при попытке войти в неё (обращение) поток или продолжит работу, если КС свободна, или начнет ожидание её освобождения другим потоком.

## Атомарные операции

Атомарными (неделимыми) называются операции, выполнение которых не может быть прервано переключением потоков — такая операция гарантированно завершится до переключения. Атомарным может быть сложение, обмен переменных значениями и т. п. Важно понимать, что даже одно простое выражение на языке высокого уровня (C = A + B) с точки зрения процессора делится на несколько. Для приведенного примера (условно):

1. поместить значение A в регистр eax;
2. поместить значение B в регистр ebx;
3. сложить значения в eax и ebx, поместить результат в eax;
4. переместить значение из регистра eax в переменную C.

Windows API предоставляет большое количество [функций для различных атомарных операций](http://msdn.microsoft.com/en-us/library/windows/desktop/ms684122(v=vs.85).aspx): InterlockedAdd() — сложение, InterlockedExchange() — обмен значений и т. д. Важную роль играет функция InterlockedCompareExchange(), которая атомарно сравнивает значение в памяти с эталоном, и если они равны, заменяет значение в памяти на новое, — это и есть инструкция TSL (Test-and-Set-Lock), упомянутая в лекционном курсе, а в настоящее время её часто называют CAS (Compare-And-Swap), что то же самое.

Атомарные операции выполняются дольше обычных — примерно 50 тактов ЦП. Однако, не требуется и переход в режим ядра, что занимает тысячи тактов (единицы микросекунд).

Последовательность атомарных операций не является атомарной сама по себе. Например, если требуется поддерживать соотношение i == j, следующий код небезопасен:

1. DWORD i = 0, j = 0;
2. InterlockedIncrement(&i);
3. InterlockedIncrement(&j);

Между строками 2 и 3 может произойти переключение потоков, и окажется i = 1, а j = 0, пока управление не вернется потоку. Необходимо обеспечивать атомарное выполнение не только отдельных операций, но и целых участков кода. Задача решается примитивами синхронизации.

## Примитивы синхронизации Windows

### Критические области (critical section)

[Критическая область](http://msdn.microsoft.com/en-us/library/windows/desktop/ms682530(v=vs.85).aspx) (КО) — это примитив для синхронизации нескольких потоков одного процесса. Критической областью защищают участок кода, который не должен одновременно выполняться несколькими потоками.

Объект-критическую область необходимо инициализировать перед использованием один раз из произвольного потока вызовом функции InitializeCriticalSection().

Перед обращением к защищенному участку все потоки должны попытаться захватить критическую область (говорят: войти в неё) вызовом EnterCriticalSection(). Если данную КО уже захватил другой поток, текущий войдет в состояние ожидания, пока КО освободится вызовом LeaveCriticalSection() из владеющего КО потока. Обеим функциям передается переменная типа CRITICAL\_SECTION; при доступе к критической области все потоки должны использовать одну и ту же переменную.

Пример с двумя переменными можно было бы исправить следующим образом:

1. DWORD i = 0, j = 0;
2. CRITICAL\_SECTION cs;
3. InitializeCriticalSection(&cs);
4. EnterCriticalSection(&cs);
5. i = i + 1;
6. j = j + 1;
7. LeaveCriticalSection(&cs);

Переменные, объявленные в строках 1 и 2, являются общими для использующих их потоков. Вызов на строке 3 необходимо сделать один раз из любого потока. В строке 4 происходит вход в критическую область, а в строке 7 — выход из нее. Операции в строках 5 и 6 защищены КО, поэтому их более не требуется делать атомарными. Таким образом, код в строках 3—7 допустимо разместить в нескольких потоках — в любой момент времени только один поток будет выполнять эти инструкции.

Бывает полезна и функция TryEnterCriticalSection(): если критическая область свободна, поток входит в неё, если же КО занята, вместо блокировки потока функция просто возвращает FALSE.

### Функции ожидания

Одной из операций с любым примитивом синхронизации является блокировка потока до определенного события. Например, EnterCriticalSection() блокирует вызывавший поток, пока другой поток не вызовет LeaveCriticalSection(). Большая часть остальных примитивов синхронизации использует для той же цели единый набор [функций ожидания](https://msdn.microsoft.com/en-us/library/windows/desktop/ms687069.aspx).

Простейшей функцией ожидания является WaitForSingleObject(), которой передается дескриптор примитива синхронизации и время (в миллисекундах), которое поток готов оставаться заблокированным («ждать»). Если поток готов ждать сколь угодно долго, можно вместо числа миллисекунд передать специальное значение INFINITE; если поток не готов ждать, допустимо использовать значение 0 мс. При успешном завершении, то есть когда другой поток освободил примитив синхронизации, функция возвращает код WAIT\_OBJECT\_0. Это может произойти мгновенно, если в момент вызова примитив синхронизации не был захвачен другим потоком (как EnterCriticalSection()). В случае, если время ожидания истекло, функция возвращает код WAIT\_TIMEOUT.

Функция WaitForMultipleObjects() позволяет ожидать освобождения нескольких примитивов синхронизации одновременно или любого из нескольких. Функция Sleep() приостанавливает выполнение потока на заданное количество миллисекунд.

### Мьютексы (mutex)

[Мьютекс](http://msdn.microsoft.com/en-us/library/windows/desktop/ms684266(v=vs.85).aspx) — аналог критической области, позволяющий синхронизировать потоки в разных процессах. Критическая область идентифицировалась переменной типа CRITICAL\_SECTION, которая использовалась потоками совместно. У процессов не может быть общей переменной, поэтому мьютексам присваиваются имена при создании. Сначала поток одного из процессов создает мьютекс функцией CreateMutex(); потоки остальных процессов получают доступ к этому мьютексу функций OpenMutex() по известному имени мьютекса.

Захват мьютекса может выполняться функциями ожидания. Завершив работу с захваченным мьютексом, поток может освободить его функцией ReleaseMutex().

Прекращение доступа к мьютексу выполняется функцией CloseHandle(). На каждый вызов CreateMutex() или OpenMutex() должно приходиться по вызову CloseHandle() с тем же дескриптором. Мьютекс уничтожается, когда закрывается последний дескриптор, связанный с ним. Если мьютекс уничтожается во время ожидания одним из потоков, WaitForSignleObject() и аналоги возвращают код WAIT\_ABANDONED.

Термин «mutex» зачастую не переводится. Вне Windows API мьютексом может называться и критическая область (например, std::mutex в С++11).

### Семафоры (semaphore)

Примитив синхронизации «семафор» подробно рассмотрен в лекционном курсе. Семафор позволяет ограничить число потоков-пользователей общего ресурса.

[Семафоры Windows API](https://msdn.microsoft.com/en-us/library/windows/desktop/ms685129%28v=vs.85%29.aspx) создаются функцией CreateSemaphore(), которая позволяет указать максимальное значение счетчика семафора (число одновременных пользователей общего ресурса). Возможности получить текущее значение счетчика семафора не предоставляется. Если необходим доступ к семафору из разных процессов, можно задать имя семафора, а затем получать его дескриптор в других процессах функцией OpenSemaphore().

Захват семафора (действие «DOWN», см. лекционный курс) выполняется функциями ожидания. Освобождение семафора (действие «UP») производится функцией ReleaseSemaphore(). Прекращение доступа к семафорам и их уничтожение выполняется аналогично мьютексам.

### События (event)

[Примитив синхронизации «событие»](http://msdn.microsoft.com/en-us/library/windows/desktop/ms682655(v=vs.85).aspx) служит для передачи оповещений между потоками. Один или несколько потоков могут ожидать наступления события, а другой поток может через объект-событие оповестить ожидающих, что можно продолжить работу.

Создание событий и доступ к ним выполняется функциями CreateEvent() и OpenEvent() аналогично мьютексам. События могут быть сбрасываемыми автоматически (auto-reset) и вручную (manual-reset). В первом случае после оповещения пробуждается один из ожидавших наступления события потоков, а для прочих событие по-прежнему считается не наступившим. После оповещения о сбрасываемых вручную событиях они считаются наступившими до тех пор, пока не будет вызвана функция ResetEvent(); ожидавшие события потоки пробуждаются один за другим. Неизвестно, успеют ли все потоки, ожидавшие наступления события, начать работу, пока событие не будет сброшено, поэтому бывает удобнее функция PulseEvent(): после её вызова *для события, сбрасываемого вручную* пробуждаются все ожидавшие событие потоки, а затем событие вновь *автоматически* сбрасывается (считается не наступившим). В современных версиях ОС для тех же целей рекомендуется более надежный механизм условных переменных (condition variables), который в курсе не рассматривается.

Ожидание наступления событий выполняется функциями ожидания. Уничтожение событий и прекращение работы с ними производится так же, как и для мьютексов.

### Циклические блокировки (spinlock)

Синхронизацию возможно выполнить и без системных примитивов, пользуясь только атомарными переменными. Поток может ожидать в простом цикле, пока атомарный флаг не примет значения «истина», а другой поток может изменять значение флага (сигнализировать о событии). Состояние ожидающего потока при этом остается активным, то есть, он продолжает расходовать процессорное время. С другой стороны, если ожидание длится недолго, затраты могут оказаться меньше, чем на системный вызов для перевода потока в состояние ожидания, а затем на пробуждение потока. Можно организовать spinlock как вручную, так и [настроить критическую область](http://msdn.microsoft.com/en-us/library/windows/desktop/ms683476(v=vs.85).aspx), чтобы вместо немедленной блокировки первые несколько миллисекунд использовалась spinlock, и лишь затем поток начал ожидание.

## Некоторые аспекты многопоточного программирования

### Видимость изменений

Известно, что память ЭВМ имеет несколько уровней, среди них оперативная память (ОЗУ), кэш процессора и его регистры. Переменные хранятся в ОЗУ, часто используемые помещаются в кэш, а при операциях над ними — в регистры процессора. Таким образом, после изменения переменной новое значение не сразу оказывается в ОЗУ, а некоторое время может быть только в регистрах процессора. В многопоточном программировании это может привести к проблемам: если один поток изменяет переменную (не атомарно), другой может «не видеть» нового значения, пока оно не перенесено из регистра в ОЗУ. Компилятор может даже оптимизировать программу так, что переменная никогда не окажется в ОЗУ.

В C++ ключевым словом volatile можно указать компилятору, что переменную следует размещать в ОЗУ и считывать её значение всякий раз на случай, если она изменилась из других потоков: volatile int n. Отметим, что это не делает операции над n атомарными. Delphi не предоставляет подобного способа. Самым надежным и верным подходом является использование атомарных операций в обоих языках.

### Средства многопоточного программирования C++ и Delphi

В данной ЛР необходимо использовать для синхронизации системные вызовы ОС Windows, однако на практике используются более высокоуровневые и удобные средства.

#### C и C++

Богатый [набор средств](http://en.cppreference.com/w/cpp/thread) для многопоточного программирования появился в С11 и С++11, поэтому для их использования необходим современный компилятор.

Заголовочный файл <atomic> содержит классы и шаблоны классов для атомарных типов данных: std::atomic\_int, std::atomic\_double и т. п. Над переменными этих типов возможны все те же операции, что и над обычными (int, double и т. п.), но их реализация атомарна. Гарантированно без блокировок реализует инструкцию TSL (CAS) класс std::atomic\_flag.

Класс std::mutex из заголовочного файла <mutex> представляет критическую область. Средств синхронизации для межпроцессного взаимодействия в стандартной библиотеке языка C++ нет, но они имеются в популярной библиотеке Boost.Interprocess.

Класс потока std::thread доступен в заголовочном файле <thread>. Также в STL присутствуют условные переменные (аналог событий Windows API).

Благодаря гарантированному вызову деструкторов, C++ позволяет защититься от удержания примитивов синхронизации потоком при ошибках: специальному объекту-защитнику (guard) в конструкторе передается объект-примитив, который сразу же захватывается, а в своем деструкторе объект-защитник освобождает захваченный примитив. Поскольку деструктор объекта-защитника будет вызван всегда (при штатном или нештатном завершении потока), блокировка обязательно будет снята, и программа не «зависнет». Такую стратегию реализует класс std::lock\_guard.

#### Delphi

Библиотека VCL содержит [модуль SyncObjs](http://docwiki.embarcadero.com/Libraries/XE6/en/System.SyncObjs) с классами TCriticalSection, TMutex, TEvent, TSemaphore и т. п., инкапсулирующие одноименные примитивы ОС.

Важным классом является TThread, представляющий поток. Предлагается порождать классы-потомки TThread и записывать код потока в переопределенным методе Execute(). Специальный метод Synchronize() позволяет выполнить код в специальной критической области, глобальной для всех потоков Delphi, включая основной. Например, метод Synchronize() необходимо использовать для обращения к оконному интерфейсу из фоновых потоков.

# Задание на лабораторную работу

## Подготовка к лабораторной работе

1. Повторить лекционные материалы о процессах и потоках. Уяснить понятие процесса, потока и разницу между ними; суть задачи синхронизации потоков и способы её решения; проблему взаимоблокировки.
2. Изучить теоретическое введение ЛР № 5 по сборнику ЛР (с. 35—45).

Вариант № 1

Усовершенствовать программу для работы с проецируемыми в память файлами, написанную в ходе лабораторной работы № 2.

1. Добавить механизм оповещений об изменении общей области памяти через событие.
   1. Перед началом работы требуется запрашивать у пользователя имя события и либо подключаться к работе с данным событием, либо создавать его.
   2. При записи данных в спроецированную область памяти следует производить оповещение о наступлении события.
   3. К имеющимся действиям (чтение данных, запись данных, выход из программы) необходимо добавить четвертое: ожидать наступления события записи данных в спроецированную область памяти, после чего печатать её содержимое.

*Указание.* Каждое сообщение должны получать все экземпляры программы. Тип события и метод оповещения выберите самостоятельно.

1. Сделать ожидание изменений разделяемой области памяти фоновым.
   1. Перенести ожидание наступления события в фоновый поток. (В основном потоке оставить запрос команд у пользователя, кроме команды «ожидать изменения данных».)
   2. Вместо печати данных в фоновом потоке:
      1. считывать данные в специальный буфер;
      2. атомарно выставлять флаг, сигнализирующий об изменении данных.
   3. В основном потоке перед выводом приглашения пользователю проверять флаг, сигнализирующий о том, что данные менялись, пока пользователь вводил предыдущую команду. Если флаг взведен, печатать данные из буфера и атомарно сбрасывать флаг.

Указание. Очевидно, фоновый поток не должен реагировать на изменение разделяемых данных из основного потока той же программы. Однако оповещать другие экземпляры программы событием по-прежнему необходимо. Можно завести еще один флаг, которым основной поток сможет сигнализировать фоновому, что является источником события.

Вариант № 2

Спровоцировать и пронаблюдать проблемы, возникающие в многопоточных программах при отсутствии синхронизации потоков или неправильном её выполнении.

1. Создать программу из пяти потоков — основного, А, B, C и D:
   1. Основной поток запускает остальные четыре и ожидает их завершения функцией [WaitForMultipleObjects()](http://msdn.microsoft.com/en-us/library/windows/desktop/ms687025(v=vs.85).aspx).
   2. Поток A добавляет в список S числа 1, 2, 3 и т. д.
   3. Поток B извлекает из списка S последний элемент, возводит его в квадрат и помещает в список R. Если в списке S нет элементов, поток B ожидает одну секунду функцией Sleep().
   4. Поток C извлекает из списка S последний элемент, делит его на 3 и помещает в список R. Если в списке S нет элементов, поток C ожидает одну секунду.
   5. Поток D извлекает из списка R последний элемент и печатает его. Если в списке R нет элементов, поток D печатает сообщение об этом и ожидает одну секунду.

Синхронизацию потоков производить на данном этапе не нужно. Запустить программу несколько раз, пронаблюдать результаты и занести их в отчет.

*Примечание.* Стабильной работы программы не ожидается.

1. Обеспечить корректную синхронизацию потоков.
   1. Каждое обращение к спискам S и R из любого потока защитить критической областью (одна КО для списка S, другая — для списка R).
   2. Запустить программу несколько раз, пронаблюдать результаты, занести в отчет:
      1. результаты наблюдений;
      2. граф использования потоками A, B, C и D ресурсов R и S.

*Примечание.* Ожидается, что программа будет работать стабильно, не только не завершаясь аварийным образом, но и не «зависая».

1. Спровоцировать взаимоблокировку вследствие состязания.
   1. Изменить алгоритм потока B на следующий:
      1. вход в КО для списка S, вход в КО для списка R;
      2. извлечение элемента из S, вычисление, добавление элемента в R;
      3. выход из КО для списка R, выход из КО для списка S.
   2. Аналогично изменить и алгоритм потока C, но порядок входа и выхода из критической области сделать другим.
   3. Повторить пункт 2.2.

*Указание.* В целях наглядности можно между обращениями к КО в обоих потоках вставить задержки функцией Sleep(). Так можно проверить сценарии с разным порядком входа и выхода из КО.

*Примечание.* Ожидается «зависание» потоков A, B и С, что будет проявляться в постоянно пустом списке R.

1. Спровоцировать блокировку при неконтролируемом удержании КО.
   1. В версии программы, полученной в п. 2, внести изменение в алгоритм потока B: после захвата КО для списка R с вероятностью 0,1 следует произвести выход из потока функцией ExitThread().
   2. Запустить программу, пронаблюдать, занести в отчет и объяснить результаты.

Вариант № 3

Усовершенствовать программу для работы с почтовыми ящиками Windows, написанную в ходе лабораторной работы № 2.

1. Добавить механизм оповещений о новых сообщениях через семафор.
   1. После создания почтового ящика или подключения к нему запрашивать у пользователя имя семафора и либо подключаться к работе с данным семафором, либо создавать его.
   2. При добавлении сообщения в почтовый ящик (в программе-клиенте) выполнять действие «UP» над семафором.
   3. К имеющимся действиям в программе-сервере (чтение сообщения, проверка наличия сообщений, выход из программы) добавить четвертое: ожидать поступления новых сообщений. При этом программа:
      1. выполняет над семафором действие «DOWN»;
      2. считывает новое сообщение и отображает его на экране;
      3. проверяет, равен ли счетчик семафора нулю;
      4. если не счетчик семафора не равен нулю, переходит к шагу 1).

*Указание.* Проверка счетчика семафора на равенство нулю выполняется при любой попытке выполнить действие «DOWN».

1. Сделать ожидание новых сообщений фоновым.
   1. В программе-сервере создать фоновый поток, в который перенести действия пункта 1.3. (В основном потоке оставить запрос команд у пользователя, кроме команды «ожидать поступления новых сообщений».)
   2. Синхронизировать доступ потоков к консоли. Для этого завести критическую область, в которой выполняется ввод пользователем команды и её исполнение (в основном потоке) и печать поступившего сообщения (в фоновом потоке).

*Примечание.* Таким образом, если пользователь начал вводить команду, и в это время пришло сообщение, оно не должно быть напечатано, пока пользователь не введет команду, и она не будет выполнена.

*Указание.* Начало ввода пользователем можно определить функцией getch() из <conio.h>. Она возвращает символ, соответствующий нажатой клавише, сразу после нажатия. Следует ввести первый символ команды через getch(), затем войти в КО, ввести остаток команды и исполнить её, затем выйти из КО.

# Контрольные вопросы

1. В чем состоит проблематика многопоточного программирования, что такое синхронизация потоков и для чего она нужна?
2. Что такое атомарная операция? Приведите примеры функций Windows API, реализующие их, и объясните работу названных функций.
3. Что такое критические области (critical section) и какие функции и типы Windows API используются для работы с ними?
4. Что такое мьютексы (mutex) и чем они отличаются от критических областей? Какие функции и типы Windows API используются для получения доступа к мьютексу, его захвата и освобождения?
5. Каково назначение функций ожидания в Windows API? Приведите пример функции ожидания, объясните смысл всех её параметров и возможных значений результата.
6. Что такое семафоры (semaphore), для чего они могут использоваться, какие возможны операции и с помощью каких функций Window API они выполняются?
7. Что такое события (event) в Windows API и для чего они могут использоваться? Какие существуют типы событий и чем они отличаются друг от друга?
8. Какими функциями Windows API осуществляется работа с событиями? В каком случае событие, сбрасываемое вручную, всё-таки сбрасывается автоматически?
9. Что такое циклическая блокировка (spinlock), каковы её преимущества и недостатки по сравнению с полноценной блокировкой? Когда имеет смысл применять циклические блокировки и как это возможно в Windows API?
10. В чем заключается проблема видимости изменений при многопоточном программировании и какие средства для её преодоления существуют?
11. Что такое взаимоблокировка (deadlock)? Каковы условия её возникновения и можно ли функциями Windows API предупреждать взаимоблокировки?
12. Что такое состояние состязания, или гонки (race condition)? Какие эффекты при этом наблюдаются? Чем опасно состояние состязания и как его предотвратить?