Синхронизация процессов и потоков

Цель работы

1. Знать основные понятия и проблематику многопоточности и программирования в многопоточной среде.
2. Уметь использовать системные примитивы синхронизации Windows для обеспечения корректной работы многопоточных программ.

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, понятие состязания, атомарности и потокобезопасности, а также примитивы синхронизации.

Из лекционного курса известно, что запущенный поток может не работать по двум причинам: внешней — когда потоку не выделено процессорное время (выполняется другой поток), и внутренней — когда управление не передается

потоку, пока не наступило некое событие, то есть поток ожидает события. В многопоточном программировании ожиданием называют второй случай.

Состязанием (data race condition) называют конкурентный доступ двух и более потоков к одним и тем же данным. Без специальных мер это может привести данные в несогласованное состояние. Рассмотрим пример: добавление элемента в динамический массив. Это действие состоит из двух операций: записи нового элемента в свободную ячейку (ее индекс определяется количеством имеющихся элементов) и увеличения переменной с количеством элементов. Пусть поток А поместил новое значение в свободную ячейку и был прерван, не успев увеличить количество элементов. Поток Б использует необновленное количество элементов, чтобы записать свое новое значение в «свободную» ячейку (на самом деле, в ней уже размещен элемент потока А, но поток Б не оповещен об этом). Далее оба потока увеличат счетчик элементов по одному разу (не важно, в каком порядке). Получится, что в список попадет только значение из потока Б, а переменная с количеством элементов не будет отражать реальный размер массива. Состояние массива стало несогласованным, некорректным; такая программа будет работать неправильно. В многоядерных системах состязания возникают и без прерывания потоков: если два ядра также конкурентно работают со массивом, их операции над переменными могут выполняться вперемежку по времени. Про описанную операцию говорят, что она не является *потокобезопасной (non thread-safe)*.

Для ликвидации состязания требуется упорядочить работу потоков с массивом: пока один поток не закончил добавление, другой поток не должен начинать операцию, то есть должен ожидать своей очереди. Это позволяют примитивы синхронизации — системные объекты, к которым могут обращаться несколько потоков, чтобы каждому потоку определить, можно ли продолжать работу или следует перейти к ожиданию. Из лекционного курса известен пример — критическая область: при попытке войти в неё поток или продолжит работу, если критическая область свободна, или начнет ожидание её освобождения другим потоком. Часть примитивов синхронизации позволяет упорядочить работу потоков из разных процессов.

Атомарные операции

Аппаратную основу для примитивов синхронизации составляют атомарные инструкции процессора. Рассмотрим сначала конкретный пример: операцию `test-and-set-lock` (TSL) из лекционного курса. Она оперирует общей для потоков переменной `lock`, переменной `expected` (ожидаемое значение `lock`) и значением `desired` (желаемое новое значение `lock`). Если `lock` равна `expected`, значение `lock` меняется на `desired`, и операция считается успешной; иначе в `expected` записывается реальное значение `lock`. Действие TSL можно представить как в виде функции на C (на самом деле, TSL — одна инструкция процессора):

```
1 bool
2 test_and_set_lock(int* lock, int* expected, int desired) {
3     if (*lock == *expected) {
4         *lock = desired;
5         return true;
6     }
7     *expected = *lock;
8     return false;
9 }
```

Ключевым свойством TSL является атомарность: изменение переменной `lock` из другого потока не может произойти во время работы TSL (если другой поток тоже выполняет TSL, они выполняются последовательно).

При помощи TSL можно организовать критическую область так:

```
1 int expected = 0;
2 while (!test_and_set_lock(&lock, &expected, 1)) {
3     expected = 0;
4     wait();
5 }
6 array[length] = 42;
7 length++;
8 lock = 0;
```

Критическая область считается свободной, если в переменной `lock` — нуль. В строках 1–5 поток пытается войти в критическую область, пытаясь атомарно заменить значение `lock` на 1. Когда это удастся, строки 6 и 7 добавят элемент в массив. В строке 8 критическая область освобождается (будем считать, что присваивание атомарно — на архитектуре `x86_64` это так).

Если критическую область захватить не удалось (`test_and_set_lock()` вернула `false`), поток ожидает какое-то время (строка 4), затем пробует

захватить критическую область еще раз (продолжает цикл). В зависимости от реализации ожидания различают обычные и циклические блокировки (spinlock). При обычной блокировке `wait()` переводит поток в состояние ожидания, то есть отдает управление ОС, что более эффективно для системы (она может использовать остаток кванта времени потока), но сам поток не сможет еще раз попытаться захватить критическую область, пока не получит очередной квант времени от ОС. Переход в режим ядра занимает единицы микросекунд (10^{-6} с), а квант времени порядка 20 мс (10^{-3} с); TSL длится десятки наносекунд (10^{-9} с). Если код в критической области выполняется быстро, то есть ожидание недолгое, нерационально отдавать управление ОС. При циклической блокировке `wait()` просто задерживает поток на короткое время или ничего не делает, то есть следующая попытка захватить критическую область наступит максимально быстро. Однако во время ожидания поток будет раходовать процессорное время впустую (т. н. состояние busy-wait).

Windows API предоставляет большое количество [функций для различных атомарных операций](#): `InterlockedAdd()` — сложение, `InterlockedExchange()` — обмен значений и т. д. Операцию TSL выполняет функция `InterlockedCompareExchange()`. В настоящее время она используется не только для блокировок (locks), поэтому ее чаще называют compare-and-swap (CAS).

Примитивы синхронизации Windows

Критические области (critical section)

[Критическая область \(КО\)](#) — это примитив для синхронизации нескольких потоков одного процесса. Критической областью защищают участок кода, который не должен одновременно выполняться несколькими потоками. В Windows API критической областью называют не участок кода, а переменную типа `CRITICAL_SECTION`, которая используется для защиты этого участка.

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.

Синхронизация добавления элемента в массив с помощью критических областей Window API выполняется следующим образом:

```
1 CRITICAL_SECTION cs;
2 InitializeCriticalSection(&cs);
3 EnterCriticalSection(&cs);
4 array[length] = 42;
5 length++;
6 LeaveCriticalSection(&cs);
```

Переменная `cs` является общей всех потоков. Вызов на строке 2 необходимо сделать один раз из любого потока. В строке 3 происходит вход в критическую область, а в строке 6 — выход из нее. Операции в строках 4 и 5 защищены КО.

Бывает полезна и функция `TryEnterCriticalSection()`: если критическая область свободна, поток входит в неё, если же КО занята, вместо блокировки потока функция просто возвращает `FALSE`. Это можно использовать, чтобы выполнить операцию любым, но только одним из потоков.

Можно [настроить критическую область](#), чтобы вместо немедленной блокировки первые несколько миллисекунд она работала как циклическая блокировка, и лишь затем поток начал ожидание.

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](#).

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 — 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, WaitForSingleObject() и аналоги возвращают код WAIT_ABANDONED.

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

Семафоры (semaphore)

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

[Семафоры Windows API](#) создаются функцией `CreateSemaphore()`, которая позволяет указать начальное и максимальное значение счетчика семафора (число одновременных пользователей общего ресурса). 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()`, с помощью которой можно также определить значение счетчика. Прекращение доступа к семафорам и их уничтожение выполняется аналогично мьютексам.

События (event)

[Примитив синхронизации «событие»](#) 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.

Многопоточное программирование в C++ и Delphi

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

С и C++

Богатый [набор средств](#) для многопоточного программирования появился в C11 и C++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](#) 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 переопределенном 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

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.
 - 1.1. Перед началом работы требуется запрашивать у пользователя имя события и либо подключаться к работе с данным событием, либо создавать его.
 - 1.2. При записи данных в спроецированную область памяти следует производить оповещение о наступлении события.

1.3. 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.

2. Сделать ожидание изменений разделяемой области памяти фоновым.

2.1. Перенести ожидание наступления события в фоновый поток. (В основном потоке оставить запрос команд у пользователя, кроме команды «ожидать изменения данных».)

2.2. Вместо печати данных в фоновом потоке:

1) считывать данные в специальный буфер;

2) атомарно выставлять флаг, сигнализирующий об изменении данных.

2.3. 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, A, B, C и D:

- 1.1. Основной поток запускает остальные четыре и ожидает их завершения функцией `WaitForMultipleObjects()`.
- 1.2. Поток А добавляет в список S числа 1, 2, 3 и т. д.
- 1.3. Поток В извлекает из списка S последний элемент, возводит его в квадрат и помещает в список R. Если в списке S нет элементов, поток В ожидает одну секунду функцией `Sleep()`.
- 1.4. Поток С извлекает из списка S последний элемент, делит его на 3 и помещает в список R. Если в списке S нет элементов, поток С ожидает одну секунду.
- 1.5. 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.

2. Обеспечить корректную синхронизацию потоков.

- 2.1. Каждое обращение к спискам S и R из любого потока защитить критической областью (одна КО для списка S, другая — для списка R).
- 2.2. Запустить программу несколько раз, пронаблюдать результаты, занести в отчет:
 - 1) результаты наблюдений;
 - 2) граф использования потоками А, В, С и D ресурсов R и S.

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

3. Спровоцировать взаимоблокировку вследствие состязания.

- 3.1. Изменить алгоритм потока В 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.
- 3.2. Аналогично изменить и алгоритм потока C, но порядок входа и выхода из критической области сделать другим.
 - 3.3. 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 C, что будет проявляться в постоянно пустом списке R.

4. Спровоцировать блокировку при неконтролируемом удержании КО.
 - 4.1. 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()`.
 - 4.2. Запустить программу, 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.
 - 1.1. После создания почтового ящика или подключения к нему запрашивать у пользователя имя семафора и либо подключаться к работе с данным семафором, либо создавать его.

Указание. Можно выбрать небольшой предел значения счетчик, например, 16.
 - 1.2. При добавлении сообщения в почтовый ящик (в программе-клиенте) выполнять действие «UP» над семафором.
 - 1.3. 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».

2. Сделать ожидание новых сообщений фоновым.

2.1. 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».)

2.2. 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. Что такое взаимоблокировка (deadlock)? Каковы условия её возникновения и можно ли функциями Windows API предупреждать взаимоблокировки?
11. Что такое состояние состязания, или гонки (race condition)? Какие эффекты при этом наблюдаются? Чем опасно состояние состязания и как его предотвратить?