Механизмы синхронизациии

Синхронизация потоков и процессов в Windows NT

Концепция **взаимного исключения** - один из краеугольных камней построения любой ОС, которая означает гарантию того, что с некоторым ресурсом в конкретный момент времени может работать только один поток. Взаимное исключение необходимо в том случае, когда ресурс принципиально не может быть разделен либо его разделение приведет к непредсказуемому результату. Блоки кода, которые производят операции над ресурсом, доступ к которому не должен быть одновременным, называются **критическими секциями**.

Задача взаимного исключения становится еще более сложной в SMP-системах, когда за один и тот же ресурс могут конкурировать потоки, исполняющиеся на разных логических процессорах. Потоки операционной системы также могут работать параллельно; таким образом, внутри нее самой имеются структуры данных, которые нуждаются в защите от одновременного доступа. Более того, некоторые механизмы операционной системы предъявляют особые требования к организации критических секций. Например, планировщик потоков не имеет права полагаться на механизмы взаимного исключения, которые могут вызвать необходимость в передиспетчеризации. Аналогично менеджер памяти не должен использовать способы, потенциально генерирующие исключение отсутствия страницы. Вдобавок ко всему в системе постоянно происходят прерывания, приводящие к вызову процессором их обработчиков, которые также могут попытаться поработать с разделяемыми данными. В однопроцессорных системах подобные проблемы относительно успешно могут быть разрешены путем блокировки прерываний во время обработки ядром разделяемых ресурсов, однако для многопроцессорных систем такая схема не является приемлемой.

В связи с этим Windows NT реализует две группы примитивов синхронизации: так называемые **High IRQL** и **Low IRQL**. Первые из них отличаются тем, что нацелены на использование непосредственно ядром ОС и драйверами в вышеописанных проблемных ситуациях. Вторые предоставляют более эффективный и/или удобный интерфейс для приложений пользовательского уровня и подсистем ядра, не выдвигающих особых требований к организации взаимного исключения.

Примитивы синхронизации High IRQL

Поскольку задачей для данных примитивов стоит обеспечить синхронизацию в жестких условиях внутри самого ядра ОС, то единственное, на что они могут полагаться - это поддержка со стороны оборудования.

Атомарные операции

Атомарные операции представляют собой действия, которые гарантировано выполнятся процессором как единое неделимое целое, зачастую являясь единственной ассемблерной инструкцией. Эта инструкция на время своего выполнения блокирует шину памяти для того, чтобы к ней имел эксклюзивный доступ только процессор,

исполняющий данное действие. (Однако, в современных системах возможны более оптимальные решения, исключающие полную блокировку шины и полагающиеся целиком на механизм когерентности кэшей). Современные процессоры поддерживают достаточно широкий класс подобных операций; для примера здесь приведены только наиболее популярные из функций, опирающихся на этот механизм.

InterlockedIncrement[,B,16,64]

https://github.com/reactos/reactos/blob/master/sdk/include/crt/mingw32/intrin_x86.h#L294

Выполняет атомарное увеличение значения переменной на единицу.

InterlockedDecrement[,B,16,64]

https://github.com/reactos/reactos/blob/master/sdk/include/crt/mingw32/intrin_x86.h#L554

Аналогично предыдущей, но значение уменьшается на единицу.

InterlockedExchange[,B,16,64]

https://github.com/reactos/reactos/blob/master/sdk/include/crt/mingw32/intrin_x86.h#L175

Меняет местами значения двух переменных.

InterlockedCompareExchange[,B,16,64]

https://github.com/reactos/reactos/blob/master/sdk/include/crt/mingw32/intrin_x86.h#L147

Сравнивает одно из значений с эталоном и, в зависимости от исхода сравнения, обменивает его с другим.

Спинлоки (spinlocks)

Спинлоки - самый простой из возможных механизм синхронизации, сконструированный поверх атомарной операции обмена двух значений. При захвате спинлока процесс циклически пытается поменять местами значение двух переменных: локальной со значением **0** и некоторой глобальной. Как только в результате этой операции локальная переменная стала равна **1** - спинлок захвачен, можно выполнять действия, относящиеся к критической секции. Когда выполнение критической секции завершено, в глобальную переменную обратно помещается **1**, сигнализируя, что спинлок освобожден.

Пример псевдокода приведен ниже. Внимание: это всего лишь **иллюстрационный пример**, дающий общее представление о происходящем! Не надо использовать данный код в реальных задачах!

```
extern int globalLock; // Где-то описанная глобальная переменная
int local = 0; // Локальная переменная
while (!local)
```

```
    InterlockedExchange(local, globalLock);

// Отлично, local = 1; мы успешно захватили спинлок

// Критическая секция...

InterlockedExchange(local, globalLock);
```

Нетрудно заметить, что спинлок организует **занятое ожидание** (отсюда и название - "spinlock", "вращающаяся блокировка"). Иными словами, это далеко не самый эффективный механизм синхронизации, в прямом смысле пожирающий ресурсы процессора и заваливающий шину памяти безумным количеством обращений. Однако, он является единственно возможным в условиях, когда необходима синхронизация между несколькими процессорами. На практике обычно в цикл захвата вставляются специальные ассемблерные инструкции, которые "притормаживают" его выполнение.

KeAcquireSpinLock (обращается к KxAcquireSpinLock)

https://github.com/reactos/reactos/blob/master/hal/halx86/generic/spinlock.c#L53 - KeAcquireSpinLock

https://github.com/reactos/reactos/blob/master/ntoskrnl/include/internal/spinlock.h#L52 - KxAcquireSpinLock

Захватывает спинлок, одновременно поднимая уровень прерываний (запрещая их, по сути) для текущего ядра.

KeReleaseSpinLock (обращается к KxReleaseSpinLock)

https://github.com/reactos/reactos/blob/master/hal/halx86/generic/spinlock.c#L68 - KeReleaseSpinLock

https://github.com/reactos/reactos/blob/master/ntoskrnl/include/internal/spinlock.h#L89 - KxReleaseSpinLock

Освобождает спинлок, одновременно восстанавливая уровень прерываний, который был до захвата спинлока.

Спинлоки с очередями

Спинлок с очередью похож на спинлок без очереди, только с очередью. Идея в том, что при неудачной попытке захвата спинлока поток кладет свой идентификатор в некоторую очередь, которая связана с этим спинлоком. Поток же, который владеет спинлоком, не освобождает его напрямую, а **передает** заблокированный спинлок тому потоку, который находится на верхушке очереди. Таким образом, от схемы "кто успел, тот и съел" классического спинлока происходит переход к FIFO. Кроме того, теперь поток, пытающийся захватить спинлок, ожидает уже установки локального (для процессора) флага и не мучает шину памяти безумным количеством запросов.

Стоит заметить, что этот функционал в ReactOS реализован не полностью в соответствии с официальной документацией Microsoft.

KeAcquireQueuedSpinLock

https://github.com/reactos/reactos/blob/master/hal/halx86/generic/spinlock.c#L81

Захватывает спинлок.

KeReleaseQueuedSpinLock

https://github.com/reactos/reactos/blob/master/hal/halx86/generic/spinlock.c#L153

Освобождает спинлок.

Исполнительные атомарные операции

Функции данной группы предоставляются исполнительной подсистемой NT и представляют собой обертки поверх спинлоков для работы со сложными структурами данных, гарантирующие их целостность.

ExInterlockedPopEntryList

https://github.com/reactos/reactos/blob/master/ntoskrnl/ex/interlocked.c#L201

Достает элемент из стека (односвязного списка).

ExInterlockedPushEntryList

https://github.com/reactos/reactos/blob/master/ntoskrnl/ex/interlocked.c#L227

Кладет элемент в стек (односвязный список).

ExInterlockedInsertHeadList

https://github.com/reactos/reactos/blob/master/ntoskrnl/ex/interlocked.c#L114

Вставляет элемент в начало двусвязного списка.

ExInterlockedRemoveHeadList

https://github.com/reactos/reactos/blob/master/ntoskrnl/ex/interlocked.c#L166

Удаляет элемент из начала двусвязного списка.

Примитивы синхронизации Low IRQL

Данные примитивы, по сравнению с методами группы High IRQL, предлагают более удобные для программиста, но и более сложные по внутреннему устройству механизмы обеспечения взаимного исключения.

Мютексы (Mutex)

Мютекс - это объект, предназначенный для организации взаимного исключения (отсюда и его название - MUTual EXclusion, взаимное исключение). После того, как один из потоков захватил мютекс, никакой другой поток до освобождения мютекса

заблокировавшим его потоком не может его получить и при попытке это сделать блокируется.

CreateMutex (NtCreateMutant)

https://github.com/reactos/reactos/blob/master/ntoskrnl/ex/mutant.c#L83

Создает или открывает именованный или неименованный мютекс.

OpenMutex (NtOpenMutant)

https://github.com/reactos/reactos/blob/master/ntoskrnl/ex/mutant.c#L166

Позволяет нескольким потокам использовать один общий мютекс. Мютекс должен быть создан ранее с помощью CreateMutex.

ReleaseMutex (KeReleaseMutant)

https://github.com/reactos/reactos/blob/master/ntoskrnl/ke/mutex.c#L98

Освобождает ресурсы, выделенные потоку под мютекс. Если поток не владеет этим мютексом, возвращает ошибку.

Критические секции

Критическая секция в WinAPI (не путать с одноименным понятием в теории) - по сути, тот же самый мютекс, но может быть использован только потоками внутри одного и того же процесса. Кроме того, в отличие от всех остальных синхронизационных примитивов, критические секции реализованы в пространстве пользователя (управление ядру передается только при блокировке процесса), что положительным образом сказывается на производительности (отсутствует необходимость переключения в режим ядра и обратно).

CRITICAL_SECTION

https://github.com/reactos/reactos/blob/master/sdk/include/psdk/winbase.h#L858

Структура, описывающая критическую секцию.

InitializeCriticalSection (RtlInitializeCriticalSection)

https://github.com/reactos/reactos/blob/master/sdk/lib/rtl/critical.c#L543

Инициализирует ресурсы, связанные с критической секцией

EnterCriticalSection (RtlEnterCriticalSection)

https://github.com/reactos/reactos/blob/master/sdk/lib/rtl/critical.c#L485

Блокирует процесс до тех пор, пока он не войдет в критическую секцию.

TryEnterCriticalSection (RtlTryEnterCriticalSection)

https://github.com/reactos/reactos/blob/master/sdk/lib/rtl/critical.c#L760

Пытается войти в критическую секцию. Если это не удается, сразу возвращает `FALSE`, не блокируя поток.

LeaveCriticalSection (RtlLeaveCriticalSection)

https://github.com/reactos/reactos/blob/master/sdk/lib/rtl/critical.c#L695

Освобождает критическую секцию, делая ее доступной для входа другому потоку.

DeleteCriticalSection (RtlDeleteCriticalSection)

https://github.com/reactos/reactos/blob/master/sdk/lib/rtl/critical.c#L395

Очищает ресурсы, связанные с критической секцией.

События (Events)

Объекты-события - это примитивы синхронизации, которые спроектированы для того, чтобы удобно передавать между потоками сигналы о наступлении (внезапно) некоторого события. Событие имеет два возможных состояния: сигнализированное и несигнализированное. События могут быть двух типов:

- Сбрасываемые вручную, чье состояние остается сигнализированным до тех пор, пока оно явно не будет сброшено
- Сбрасываемые автоматически, когда сброс происходит сразу после сообщения одному из ожидающих потоков о переходе события в сигнализированное состояние

CreateEvent (NtCreateEvent)

https://github.com/reactos/reactos/blob/master/ntoskrnl/ex/event.c#L100

Создает именованное или неименованное событие.

OpenEvent (NtOpenEvent)

https://github.com/reactos/reactos/blob/master/ntoskrnl/ex/event.c#L185

Позволяет получить доступ к именованному событию внутри другого процесса.

SetEvent (KeSetEvent)

https://github.com/reactos/reactos/blob/master/ntoskrnl/ke/eventobj.c#L159

Устанавливает событие в "сигнализированное" состояние.

ResetEvent (KeResetEvent)

https://github.com/reactos/reactos/blob/master/ntoskrnl/ke/eventobj.c#L133

Устанавливает событие в "несигнализированное" состояние.

Семафоры (Semaphore)

Семафоры в NT по сути представляют собой реализацию классической модели Дейкстры, за тем исключением, что они могут защищать более одной единицы ресурса, и, кроме того, процесс может запрашивать и освобождать также более одной единицы ресурса за раз.

CreateSemaphore (NtCreateSemaphore)

https://github.com/reactos/reactos/blob/master/ntoskrnl/ex/sem.c#L69

Создает именованный или неименованный семафор.

OpenSemaphore (NtOpenSemaphore)

https://github.com/reactos/reactos/blob/master/ntoskrnl/ex/sem.c#L161

Позволяет получить доступ к именованному семафору из другого процесса.

ReleaseSemaphore (KeReleaseSemaphore)

https://github.com/reactos/reactos/blob/master/ntoskrnl/ke/semphobj.c#L54

Освобождает семафор, "давая дорогу" другим потокам.

Ожидание и захват объекта

Внимательный человек заметит, что для событий, семафоров и мютексов (да на самом деле много чего еще) были определены операции освобождения, а операции захвата - нет. Это не ошибка. Операции захвата для данных типов объектов (на самом деле еще и для некоторых других, не рассматриваемых здесь) выполнены в виде отдельных функций WinAPI, рассмотренных ниже.

WaitForSingleObject (KeWaitForSingleObject)

https://github.com/reactos/reactos/blob/master/ntoskrnl/ke/wait.c#L416

Организует блокирующий захват семафора, мютекса или ожидание наступления некоторого события, позволяя, кроме того, указать таймаут ожидания.

WaitForMultipleObjects (KeWaitForMultipleObjects)

https://github.com/reactos/reactos/blob/master/ntoskrnl/ke/wait.c#L586

Выполняет те же операции, что и `WaitForSingleObject`, но сразу для множества потенциально разнородных объектов. Кроме таймаута ожидания можно указать его режим: ждать выполнения действия над любым объектом из множества или над всеми.

MsgWaitForMultipleObjects

https://github.com/reactos/reactos/blob/master/win32ss/user/user32/windows/message.c#L3134

Функция `WaitForMultipleObjects` имеет одну неприятную особенность: она блокирует выполнение вызвавшего ее потока полностью - в том числе, прекращается обработка этим потоком сообщений от ОС Windows, таких, как `WM_PAINT`, `WM_KEYPRESS` и тому подобных. Если поток создавал какие-либо окна, то внешне подобная ситуация будет выглядеть как глухое "зависание" окна приложения. Для решения этой проблемы и предназначена функция `MsgWaitForMultipleObjects`. Она позволяет указать, для каких типов событий необходимо прервать блокирующее ожидание.

Взаимное исключение и синхронизация: сходство и различия

Очень часто можно столкнуться с тем, что эти термины употребляются в схожем смысле или даже как взаимозаменяемые понятия. Еще больше путаницы вносит сюда WinAPI, которое представляет эти механизмы единообразно. Однако **это не одно и то же**:

- Взаимное исключение направлено на предотвращение одновременного использования ресурса несколькими потоками. Из описанных здесь механизмов для этого предназначены спинлоки, мютексы и критические секции.
- Синхронизация, исходя из своего названия, позволяет некоторым образом обеспечить выполнение тех или иных действий в разных потоках в некотором нам необходимом порядке или некоторой требуемой последовательности. Для этого подходят события и семафоры.

Использование примитивов синхронизации для организации взаимного исключения и наоборот встречается сплошь и рядом, но **является логической ошибкой**!

На стороне C#: блокировки и мониторы

Рассматриваемые ниже классы содержатся в пространстве имён `System.Threading`.

Класс Mutex

https://msdn.microsoft.com/ru-ru/library/system.threading.mutex(v=vs.110).aspx

Традиционно для .NET, данный класс есть суть обёртка над системным мютексом. И, как и системный, может быть двух видов:

- Неименованный - доступен в пределах AppDomain (домена приложения), ибо завязан на память процесса.
- Именованный - для межпроцессной синхронизации. Идентификатор - строка. Сами процессы при этом могут быть не только из платформы .NET.

Не забывайте вызвать `Dispose()` перед удалением мютекса! Это нужно для освобождения системных (неуправляемых) ресурсов, связанных с ним.

WaitOne()

Метод блокирует текущий поток до тех пор, пока не получит сигнал того, что мютекс свободен. Есть перегрузки, ограничивающие время ожидания - могут также использоваться для простой проверки занятости мютекса.

ReleaseMutex()

Освобождает ранее заблокированный текущим потоком мютекс, позволяя другому ожидающему потоку захватить его. Недопустимо "забыть" вызвать эту функцию после `WaitOne()`.

Класс Semaphore

https://msdn.microsoft.com/ru-ru/library/system.threading.semaphore(v=vs.110).aspx

И вновь класс-обёртка над системным семафором. Точно так же поддерживаются и неименованные, и именованные экземпляры.

И вновь не забывайте вызвать `Dispose()` перед удалением экземпляра этого класса! Зачем это нужно - см. выше.

WaitOne()

Блокирует поток в ожидании сигнала, если счётчик семафора равен нулю. Поддерживает перегрузки, аналогичные таковым для этого же метода класса `Mutex`. При успешной блокировке, очевидно, уменьшает счётчик семафора на 1.

Release()

Увеличивает счётчик семафора на 1. Обратите внимание, что счётчик можно программно увеличить на бОльшее, чем максимальное, значение - это вызовет соответствующее исключение. Благодаря природе семафоров, этот метод можно вызывать без предварительного вызова `WaitOne()`, что логично - семафор не принадлежит блокирующему потоку.

Критическая секция

Класс Monitor

https://msdn.microsoft.com/ru-ru/library/system.threading.monitor(v=vs.110).aspx

Низкоуровневый класс, состоящий исключительно из статических методов, реализующих механизм критической секции. *Напрямую практически не используется*. Для определения критической секции используется отдельный объект, передающийся в методы `Enter()` и `Exit()`:

- При входе в критическую секцию (метод `Enter()`), переданный объект устанавливается в состояние блокировки. После этого любой повторный вызов этой функции с тем же объектом-параметром будет заблокирован - вызвавший метод поток будет добавлен в очередь ожидания критической секции.
- При вызове `Exit()` переданный объект разлокируется, чем вызывает просмотр очереди ожидания, и если она не пуста, то случайным образом выбранный поток будет допущен в критическую секцию, а объект вновь будет заблокирован до вызова указанной функции данным потоком.

В качестве объекта блокировки могут выступать только ссылочные типы, кроме строк. Естественно, что этот объект должен быть один для всех потоков, для которых предназначена критическая секция.

Конструкция lock()

Сахарок вида

```
lock (obj)
{
    // здесь код
}
```

транслируется в

```
bool lockWasTaken = false;
var temp = obj;
try
{
    Monitor.Enter(temp, ref lockWasTaken);
    // здесь код
}
finally
{
    if (lockWasTaken)
    {
        Monitor.Exit(temp);
    }
}
```

Здесь важно знать, что obj должен быть един для всех потоков, его использующих. Обычно в качестве такого объекта используют экземпляр класса `object` и не парятся, но в принципе здесь может быть любой объект, *кроме* `this` - с ним есть нюансы, потенциально приводящие к dead lock'у.

Атомарные операции

Класс Interlocked

https://msdn.microsoft.com/ru-ru/library/system.threading.interlocked(v=vs.110).aspx

Угадайте с одного раза: что оно оборачивает? Предоставляет статические методы - атомарные операции для чисел, операции обмена и сравнения. Эти реализации, разумеется, эффективнее "наивных".

События

Класс EventWaitHandle

https://msdn.microsoft.com/ru-ru/library/csc3y90t(v=vs.110).aspx

ВНЕЗАПНО, вновь обёртка поверх WinAPI'шных `CreateEvent()` и т.п. И вновь поддерживает как именованные, так и неименованные события. Может быть создан в двух режимах - с ручным или с автоматическим сбросом сигнала.

В очередной раз не забывайте вызвать `Dispose()` перед удалением экземпляра этого класса!

Set()

Устанавливает сигнальное состояние события.

Reset()

Устанавливает несигнальное состояние. Не нужно при автоматическом сбросе сигнала.

WaitOne()

Ожидает сигнального состояния события. Точно так же есть перегрузки с фиксированным временем ожидания.

Класс ManualResetEvent

Событие с ручным сбросом сигнала. Эквивалент `EventWaitHandle` в режиме `EventResetMode.ManualReset`. Реализован только в неименованном варианте.

Класс AutoResetEvent

Событие с автоматическим сбросом сигнала. Эквивалент `EventWaitHandle` в режиме `EventResetMode.AutoReset`. Реализован только в неименованном варианте.

Further reading

Конечно, сейчас вы вряд ли "сходу" найдёте современное приложение на .NET, использующее "сырые" потоки. Всё потому, что разработано множество более высокоуровневых компонентов, которые оперируют не потоками, а задачами.

Начать можно хотя бы с Г. Шилдта *"C# 4.0 Полное руководство"*, где описана *Task Parallel Library*, и продолжить изучением более современных редакций платформы .NET и "фишек" свежих версий C# а-ля `async/await` (с 5.0, т.е. ~2012 год).

На стороне C++: мютексы и блокировки

C++, являясь кроссплатформенным языком, учитывает ограничения множества популярных платформ, и специфицирует только те возможности, которые доступны на большинстве из них. Так, стандарт C++ не включает в себя, например, семафоры.

Атомарные операции: std::atomic

`std::atomic` предоставляет шаблон, позволяющий создать новый тип данных, операции над которым гарантированно будут атомарными. Стандартная библиотека представляет большое количество уже объявленных типов-оберток поверх интегральных типов, таких, как `std::atomic_bool`, `std::atomic_int` и тому подобное, однако можно объявить и свои - единственным требованием в этом случае является

тривиальная копируемость объектов подлежащего класса (упрощенно говоря, класс не должен иметь виртуальных функций и виртуальных базовых классов).

Мютексы

C++ предлагает большое количество разнообразных абстракций над стандартными мютексами, доступными в операционной системе.

std::mutex

Очень простая обертка. Имеет методы `lock()`, `unlock()`, `try_lock()` и все.

std::timed_mutex

Аналогичен обычному `std::mutex`, но имеет два дополнительных метода `try_lock_for()` и `try_lock_until()`, позволяющие отказываться от попытки блокировки по истечению таймаута.

std::recursive_mutex

В отличие от обычного мютекса, **рекурсивный** позволяет одному и тому же потоку блокировать его несколько раз подряд без вызова `unlock()`. Подобные операции с обычным `std::mutex` могут привести к непредсказуемым последствиям.

std::recursive_timed_mutex

Объединяет в себе черты `std::timed_mutex` и `std::recursive_mutex`.

std::shared_mutex

В противовес обычному мютексу, `std::shared_mutex` предоставляет контроль не за одним, а за двумя уровнями доступа: разделяемый (shared) и эксклюзивный (exclusive). Он был спроектирован специально для решения задачи читателей и писателей. Кроме обычных методов `lock()`, `unlock()`, `try_lock()`, выполняющих операции над эксклюзивной блокировкой, класс расширен методами `lock_shared()`, `unlock_shared()`, `try_lock_shared()`, выполняющих действия над неэксклюзивной частью блокировки. Кому отдавать приоритет - читателям или писателям - определяется специальным описанным в стандарте алгоритмом, характеризуемым как "честный".

Блокировки

Блокировки - это такой true-C++ way управления мютексами. Они абстрагируют многие их детали, позволяя удобнее и безопаснее с ними обращаться.

std::lock_guard<>

Обертка, позволяющая управлять мютексом в RAII-стиле. Мютекс сразу блокируется, а при выходе из блока, в котором объявлен `lock_guard` - автоматически разблокируется.

std::unique_lock<>

Обертка, позволяющая делать с мютексом много замечательных вещей: отложенная блокировка, таймауты блокирования, рекурсивное блокирование, передача

блокировки другому потоку, одновременная блокировка нескольких мютексов с избежанием тупика и многое другое.

std::shared_lock<>

Аналогичен `std::unique_lock`, но предназначен специально для работы с `std::shared_mutex` и его собратьями в разделяемом режиме. Для эксклюзивной блокировки `std::shared_mutex` можно использовать все тот же `std::unique_lock`.