Операционные системы

Межпроцессное взаимодействие

Межпроцессное взаимодействие

Синхронизация потоков с использованием объектов ядра Windows 2000+

Синхронизация потоков

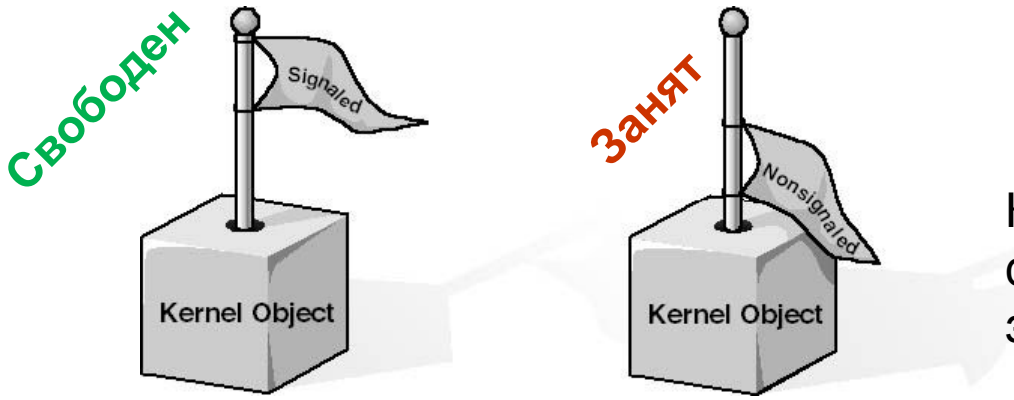
- Простейшей формой коммуникации потоков является синхронизация (synchronization).
- **Синхронизация** означает способность потока добровольно приостанавливать свое исполнение и ожидать, пока не завершится выполнение некоторой операции другим потоком.
- Все операционные системы, поддерживающие многозадачность или мультипроцессорную обработку, должны предоставлять потокам способ ожидания того, что другой поток что–либо сделает. Пример – асинхронный ввод-вывод.

Объекты синхронизации и их состояния

- Объекты синхронизации (synchronization objects) – это объекты ядра, при помощи которых поток синхронизирует свое выполнение.
- В любой момент времени синхронизационный объект находится в одном из двух состояний: свободен (signaled state) или занят.
- Правила, по которым объект переходит в свободное или занятое состояние, зависят от типа этого объекта.

Объекты синхронизации MS Windows 2000+ и их состояния

- процессы
- потоки
- задания
- файлы
- консольный ввод
- уведомления об изменении файлов
- события
- ожидаемые таймеры
- семафоры
- мьютексы

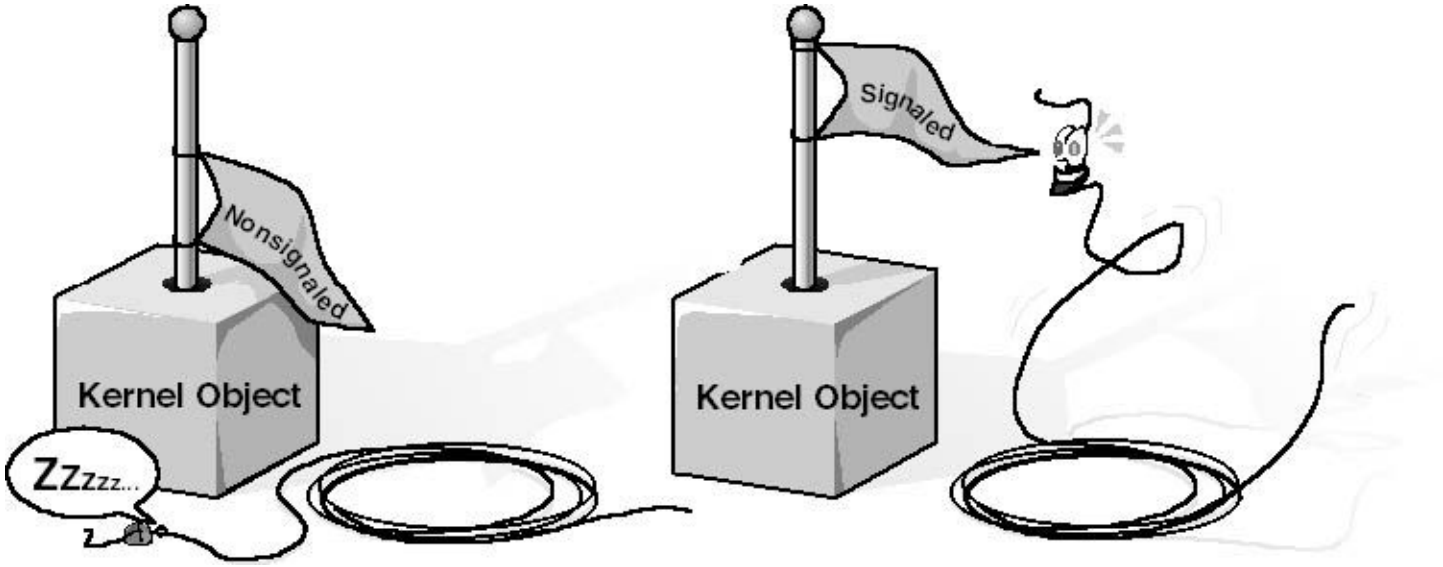


Когда объект свободен, флажок поднят, а когда он занят, флажок опущен.

Примеры функционирования объектов синхронизации

- Объект-поток находится в состоянии «занят» все время существования, но устанавливается системой в состояние «свободен», когда его выполнение завершается.
- Аналогично, ядро устанавливает процесс в состояние «свободен», когда завершился его последний поток.
- В противоположность этому, объект – таймер «срабатывает» через заданное время (по истечении этого времени ядро устанавливает объект – таймер в состояние «свободен»).

Спящие потоки



- Потоки спят, пока ожидаемые ими объекты заняты (флажок опущен).
- Как только объект освободился (флажок поднят), спящий поток замечает это, просыпается просыпается и возобновляет выполнение.

Функции ожидания

- Wait-функции позволяют потоку в любой момент приостановиться и ждать освобождения одного или нескольких объектов ядра.

```
DWORD WaitForSingleObject(
    HANDLE hObject,
    DWORD dwMilliseconds
);
```

```
DWORD WaitForMultipleObjects(
    DWOHD dwCount,
    CONST HANDLE* phObjects,
    BOOL fWaitAll,
    DWORD dwMilliseconds
);
```

Функция одиночного ожидания

- Первый параметр, *hObject,* идентифицирует объект ядра, поддерживающий синхронизацию.
- Второй параметр, *dwMilliseconds,* указывает, сколько времени (в миллисекундах) поток готов ждать освобождения объекта.
- Представленный ниже вызов сообщает системе, что поток будет ждать до тех пор, пока не завершится процесс, идентифицируемый дескриптором *hProcess.*

```
WaitForSingleObject (hProcess, INFINITE);
```

Функция множественного ожидания

- Функция *WaitForMultipleObjects* позволяет ждать освобождения сразу нескольких объектов или какого-то одного из списка объектов.
- Параметр *dwCount* определяет количество интересующих Вас объектов ядра Его значение должно быть в пределах от 1 до MAXIMUM_WAIT_OBJECTS (в заголовочных файлах Windows оно определено как 64).
- Параметр *phObject* – это указатель на массив дескрипторов объектов синхронизации.
- Параметр *fWaitAll* – флаг, который указывает режим ожидания всех заданных объектов ядра (TRUE), либо режим ожидания освобождения любого первого из них (FALSE).

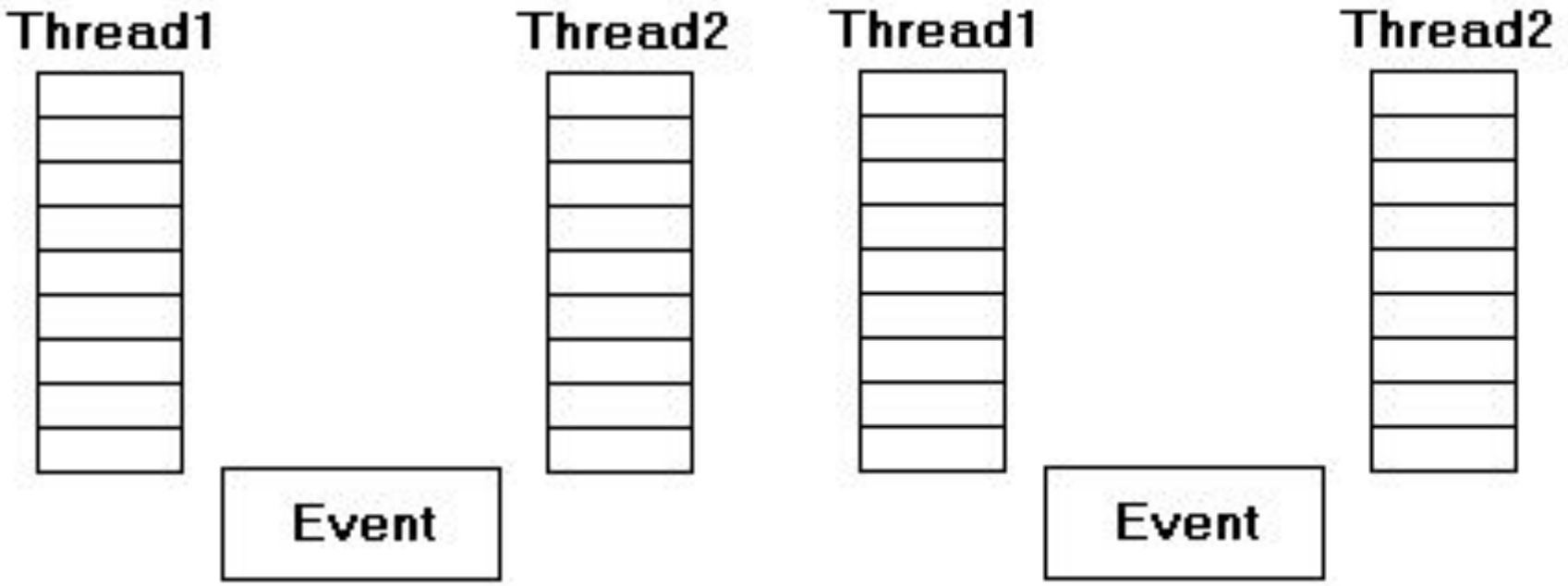
Специализированные объекты синхронизации

- Событие
- Таймер ожидания
- Семафор
- Мьютекс

События

- Событие – самая примитивная разновидность объектов синхронизации, которая просто уведомляют об окончании какой-либо операции.
- События содержат счетчик числа пользователей (как и все объекты ядра) и две булевы переменные: одна сообщает тип данного объекта-события, другая – его состояние (свободен или занят).
- Объекты-события бывают двух типов: со сбросом вручную (manual-reset events) и с автосбросом (auto-reset events).
- События с «ручным» сбросом нужно применять, если событие ждут несколько потоков. Только этот тип события позволяет выполнить синхронизацию нескольких потоков от одного события.

Сравнение работы события с ручным сбросом и автосбросом



событие с ручным сбросом

событие с автоматическим сбросом

Применение «событий»

- Объекты-события обычно используют в том случае, когда какой-то поток выполняет инициализацию, а затем сигнализирует другому потоку, что тот может продолжить работу.
- Инициализирующий поток переводит объект «событие» в занятое состояние и приступает к своим операциям.
- Закончив, он сбрасывает событие в свободное состояние.
- Тогда другой поток, который ждал перехода события в свободное состояние, пробуждается и вновь становится планируемым.

Создание объекта типа «событие»

```
HANDLE CreateEvent (
   LPSECURITY_ATTRIBUTES lpEventAttributes, // атрибуты
                                            // защиты
   BOOL bManualReset, // тип сброса
   BOOL bInitialState,    // начальное состояние
   LPCTSTR lpName      // имя объекта
);
```

- Параметр *bManualReset* сообщает системе, хотите Вы создать событие со сбросом вручную (TRUE) или с автосбросом (FALSE).
- Параметр *bInitialState* определяет начальное состояние события – свободное (TRUE) или занятое (FALSE).

Совместное использование объекта «события» процессами

- Если объект «событие» успешно создан, то *CreateEvent ()* возвращает дескриптор объекта специфичный для конкретного процесса.
- Потоки из других процессов могут получить доступ к этому объекту:
  1) наследованием описателя с применением функции *DuplicateHandle ()* (универсальный способ)*;*
  2) вызовом *OpenEvent ()* с передачей в параметре *lpName* имени, совпадающего с указанным в аналогичном параметре функции *CreateEvent ().*

Открытие объекта типа «событие»

```
HANDLE OpenEvent(
    DWORD dwDesiredAccess, // режим доступа
    BOOL bInheritHandle, // флаг наследования
    LPCTSTR lpName // имя обьекта
);
```

- Функция *OpenEvent ()* возвращает дескриптор существующего именованного объекта события.
- Вызывающий процесс может использовать возвращаемый данной функцией дескриптор в качестве аргумента любой функции, использующей дескриптор объекта события, при условии соблюдения ограничений, налагаемых значением аргумента *dwDesiredAccess*.

Режимы доступа к «событиям»

- EVENT_ALL_ACCESS – полный доступ (разрешены все возможные виды доступа)
- EVENT_MODIFY_STATE – обеспечивает возможность использования дескриптора объекта события в функциях *SetEvent ()* и *ResetEvent ()*, изменяющих состояние объекта события
- SYNCHRONIZE – позволяет использовать дескриптор объекта события в любой из функций ожидания

Управление «событиями»

- Перевод события в свободное состояние:

  BOOL SetEvent (HANDLE hEvent);

- Перевод события в занятое состояние:

  BOOL ResetEvent (HANDLE hEvent);

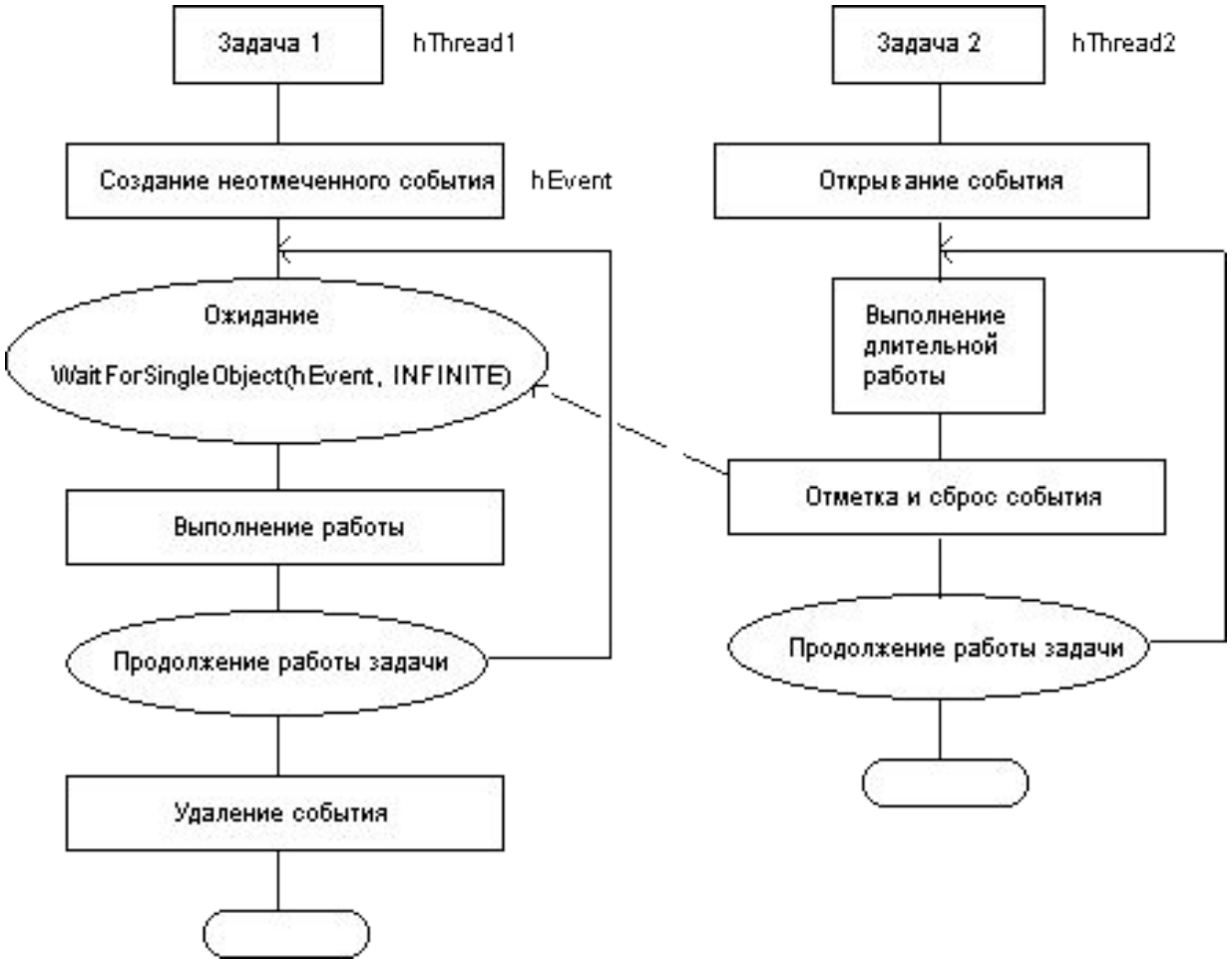
- Освобождение события и перевод его обратно в занятое состояние:

  BOOL PulseEvent (HANDLE hEvent);

Особенности *PulseEvent*

- Функция *PulseEvent ()* устанавливает событие и тут же переводит его обратно в сброшенное состояние, это равнозначно последовательному вызову *SetEvent ()* и *ResetEvent ()*.
- Если *PulseEvent ()* вызывается для события со сбросом вручную, то все потоки, ожидающие этот объект, получают управление.
- При вызове *PulseEvent ()* для события с автосбросом пробуждается только один из ждущих потоков. А если ни один из потоков не ждет объект-событие, вызов функции не дает никакого эффекта.

Пример использования «события» для синхронизации потоков



Таймеры ожидания

- Таймеры ожидания (waitable timers) – это объекты ядра, которые самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени.
- Объекты «таймеры ожидания» всегда создаются в занятом состоянии («0»).

Создание и открытие таймера ожидания

```
HANDLE CreateWaitableTimer (
   LPSECURITY_ATTRIBUTES lpMutexAttributes,
                  // атрибуты защиты
   BOOL bManualReset,// тип сброса таймера, TRUE – ручной
   LPCTSTR lpName      // имя объекта
);

HANDLE OpenWaitableTimer (
   DWORD fdwAccess, // режим доступа
   BOOL fInherit,          // флаг наследования
   LPCTSTR lpName      // имя объекта
);
```

Режимы доступа к таймеру

- TIMER_ALL_ACCESS – полный доступ (разрешены все возможные виды доступа)
- TIMER_MODIFY_STATE – определяет возможность изменения состояния таймера функциями *SetWaitableTimer ()* и *CancelWaitableTimer ()*
- SYNCHRONIZE – позволяет использовать дескриптор объекта таймера в любой из функций ожидания

Управление таймером ожидания

```
BOOL SetWaitableTimer(
   HANDLE hTimer,
   const LARGE_INTEGER *pDueTime,
   LONG lPeriod,
   LPTIMERAPCROUTINE pfnCompletionRoutine,
   LPVOID pvArgToCompletionRoutine,
   BOOl fResume
);

BOOL CancelWaitableTimer (HANDLE hTimer);
```

Запуск таймера ожидания

- Параметры *pDueTime* и *lPeriod* функции SetWaitableTimer () задают соответственно, время когда таймер должен сработать в первый раз, и интервал, с которым должны происходить дальнейшие срабатывания.
- Если таймер должен сработать только 1 раз, то достаточно передать 0 в параметре *lPeriod*.
- При необходимости Вы можете не выполнять синхронизацию с объектом таймера с помощью *Wait*-функций, а сразу вызвать некоторую функцию. Адрес этой функции и аргументы для ее вызова указываются в параметрах *pfnCompletionRoutine* и *pvArgToCompletionRoutine*.
- Параметр *fResume* полезен на компьютерах с поддержкой режима сна. Если этот параметр был установлен в состояние *TRUE*, то когда компьютер выйдет из режима сна, то пробудятся потоки, ожидавшие этот таймер.

Отмена действия таймера

- Для отмены действия таймера ожидания следует использовать функцию *CancelWaitableTimer ()*, после ее применения таймер не сработает до следующего вызова *SetWaitableTimer ()*.
- Для переустановки текущих параметров таймера ожидания нет необходимости вызывать *CancelWaitableTimer ()*, каждый вызов *SetWaitableTimer ()* автоматически отменяет предыдущие настройки перед установкой новых.

Создание объекта типа «семафор»

```
HANDLE CreateSemaphore(
   LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
                        // атрибуты защиты
   LONG lInitialCount,     // начальное значение
                        // счетчика семафора
   LONG lMaximumCount,    //  максимальное значение
                           // счетчика
   LPCTSTR lpName              // имя объекта
);
```

Открытие объекта типа «семафор»

```
HANDLE OpenSemaphore(
   DWORD fdwAccess,    // режим доступа
   BOOL fInherit,      // флаг наследования
   LPCTSTR lpName   // имя объекта
);
```

Режимы доступа к семафору

- SEMAPHORE_ALL_ACCESS – полный доступ (разрешены все возможные виды доступа)
- SEMAPHORE_MODIFY_STATE – определяет возможность изменения значение счетчика семафора функцией *ReleaseSemaphore ()*
- SYNCHRONIZE – позволяет использовать дескриптор объекта семафора в любой из функций ожидания

Захват семафора

- Для прохождения через семафор (захвата семафора) необходимо использовать функции *WaitForSingleObject ()* или *WaitForMultipleObject ()*.
- Если *Wait*-функция определяет, что счетчик семафора равен «0» (семафор занят), то система переводит вызывающий поток в состояние ожидания. Когда другой поток увеличит значение этого счетчика, система вспомнит о ждущем потоке и снова начнет выделять ему процессорное время (а он, захватив ресурс, уменьшит значение счетчика на 1).

Освобождение семафора

- Для увеличения значения счетчика семафора приложение должно использовать функцию *ReleaseSemaphore ()*.
- Функция *ReleaseSemaphore ()* увеличивает значение счетчика семафора, дескриптор которого передается ей через параметр *hSemaphore*, на значение, указанное в параметре *lReleaseCount*.
- Предыдущее значение счетчика, которое было до использования функции *ReleaseSemaphore ()*, записывается в переменную типа LONG. Адрес этой переменной передается функции через параметр *lplPreviousCount*.

Функция *ReleaseSemaphore*

```
BOOL ReleaseSemaphore(
    HANDLE hSemaphore, // дескриптор семафора LONG
    lReleaseCount, // значение инкремента LPLONG
    lplPreviousCount // адрес переменной для //
    записи предыдущего // значения
    семафора
);
```

Определение текущего состояния семафора

- Заметим, что в операционной системе Windows не предусмотрено средств, с помощью которых можно было бы определить текущее значение семафора, не изменяя его.
- Нельзя передавать 0 параметр инкремента в функцию *ReleaseSemaphore ()*.
- Можно только узнать открыт или закрыт семафор, для этого следует воспользоваться *Wait*-функцией с нулевым тайм-аутом ожидания.

Создание и открытие мьютекса

```
HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,
                    // атрибуты защиты
    BOOL bInitialOwner, // начальное состояние
    LPCTSTR lpName  // имя объекта
);

HANDLE OpenMutex(
    DWORD fdwAccess,      // режим доступа
    BOOL fInherit, // флаг наследования
    LPCTSTR lpName  // имя объекта
);
```

Режимы доступа к мьютексу

- MUTEX _ALL_ACCESS – полный доступ (разрешены все возможные виды доступа)
- MUTEX _MODIFY_STATE – определяет возможность изменения значение счетчика мьютекса функцией *ReleaseMutex ()*
- SYNCHRONIZE – позволяет использовать дескриптор объекта мьютекса в любой из функций ожидания

Управление мьютексом

- Для захвата мьютекса необходимо использовать одну из *Wait*-функций.
- Для освобождения мьютекса используется функция *ReleaseMutex ()*.

```
BOOL ReleaseMutex (HANDLE hMutex);
```

Межпроцессное взаимодействие

Критические секции в Win32 API

Критические секции

- В составе API ОС Windows имеются специальные и эффективные функции для организации входа в критическую секцию (Critical Section) и выхода из нее потоков одного процесса в режиме пользователя.
- Критическая секция позволяет добиться решения задачи взаимного исключения – в случае невозможности входа в критический участок поток переходит в состояние ожидания. Впоследствии, когда такая возможность появится, поток будет «разбужен» и сможет сделать попытку входа в критическую секцию.

Функции Win32 API для работы с критическими секциями

- Для работы с критическими секциями используют функции:
  - InitializeCriticalSection – создание КС;
  - DeleteCriticalSection – освобождение ресурсов КС;
  - EnterCriticalSection – вход в КС;
  - LeaveCriticalSection – освобождение КС;
  - TryEnterCriticalSection – проверка КС.
- Эти функции имеют в качестве параметра предварительно проинициализированную структуру типа CRITICAL_SECTION.

Структура типа CRITICAL_SECTION

```
typedef struct _RTL_CRITICAL_SECTION
{
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo; // исп. ОС
    LONG LockCount;              // счетчик блокировок
    LONG RecursionCount;         // счетчик повторного захвата
    HANDLE OwningThread;         // ID потока,
                                 // владеющего секцией
    HANDLE LockSemaphore;        // дескриптор события
    ULONG_PTR SpinCount;         // кол-во холостых циклов
                                 // перед вызовом ядра
}
RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

Основные поля структуры CRITICAL_SECTION

- Поле **LockCount** увеличивается на единицу при каждом вызове *EnterCriticalSection()* и уменьшается при каждом вызове *LeaveCriticalSection()*.
- Поле **OwningThread** содержит ID потока-владельца или «0» для незанятых секций.
- Поле **RecursionCount** хранит количество повторных входов в критическую секцию одним и тем же потоком-владельцем. Для освобождения секции необходимо вызывать функцию *LeaveCriticalSection ()* столько же раз, сколько раз была вызвана функция *EnterCriticalSection ()*.
- Поле **LockSemaphore** содержит дескриптор объекта синхронизации типа «событие», функционирующий в режиме автосброса, и реализующий конкурентный доступ к секции со стороны нескольких потоков.

Вход в свободную секцию

- Если при попытке входа в критическую секцию, LockCount уже равен 0, т.е. секция свободна:
  - поле LockCount увеличивается на 1;
  - в поле OwningThread записывает ID текущего потока, который может быть получен с помощью функции *GetCurrentThreadId ()*.
  - текущий поток продолжает выполнение – функция *EnterCriticalSection()* немедленно возвращает управление.

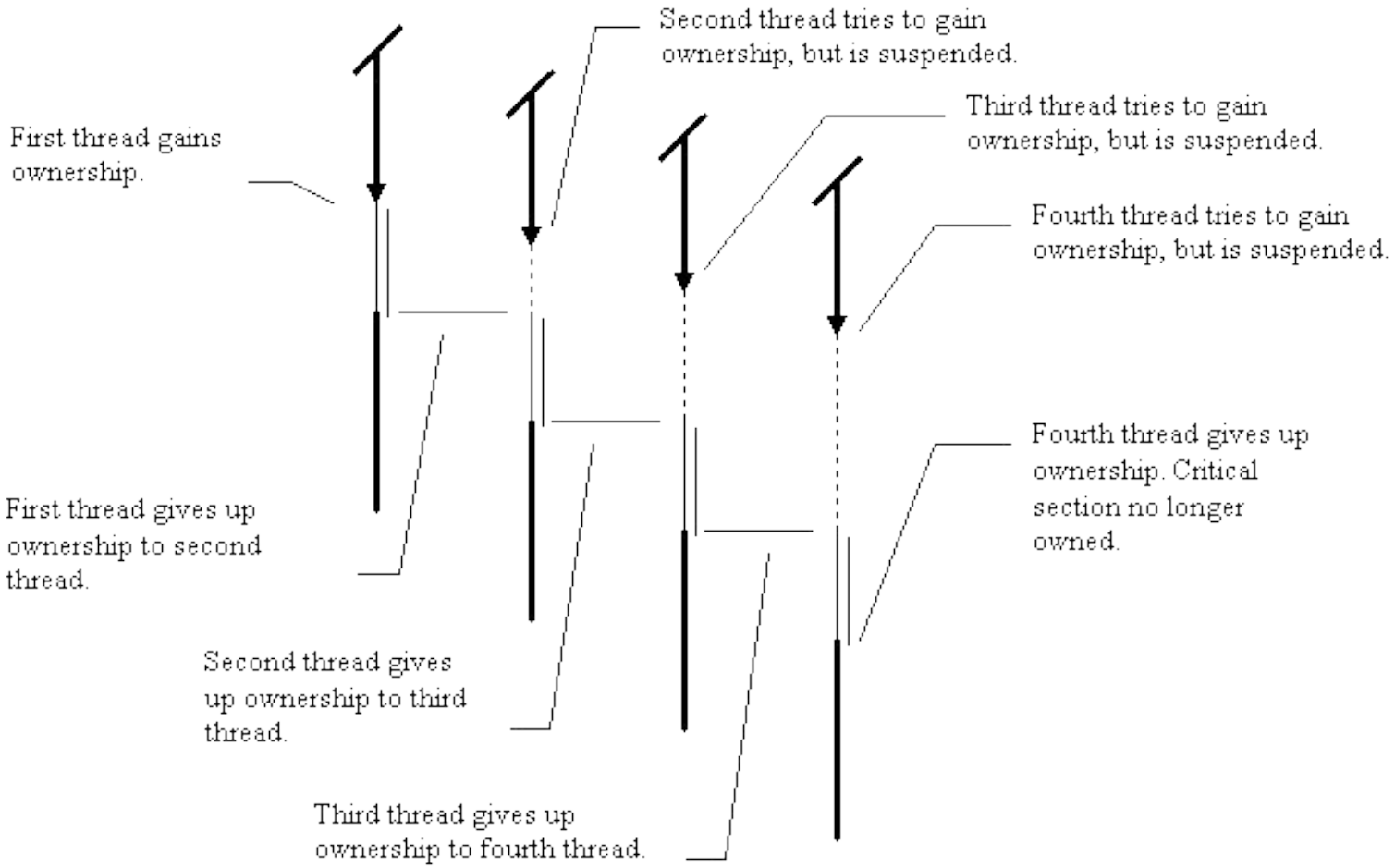
Вход в занятую секцию

- При попытке входа в занятую критическую секцию, (LockCount > 0) проверяется поле OwningThread:
  - если OwningThread совпадает с ID текущего потока, то имеет место рекурсивный захват, и RecursionCount просто увеличивается на единицу, а *EnterCriticalSection()* возвращает управление немедленно.
  - иначе текущий поток проверяет поле LockSemaphore. Если поле LockSemaphore равно 0, то поток создает событие LockSemaphore в сброшенном состоянии. Далее поток вызывает *WaitForSingleObject (LockSemaphore)* и ожидает пока поток, захвативший критическую секцию, не вызовет *LeaveCriticalSection()* количество раз равное значению поля RecursionCount.

Освобождение секции

- Поток-владелец при вызове *LeaveCriticalSection()* уменьшает поле RecursionCount на единицу и проверяет его.
- Если значение этого поля стало равным 0, а LockCount > 0, то это значит, что есть как минимум один поток, ожидающий готовности события LockSemaphore.
- В этом случае поток-владелец вызывает *SetEvent (LockSemaphore)*, что дает возможность пробудиться первому по очереди из ожидающих потоков и выполнить вход в уже свободную критическую секцию.

Иллюстрация использования критической секции



Критические секции в многопроцессорных системах

- Поле **SpinCount** используется только многопроцессорными системами.
- В однопроцессорных системах, если критическая секция занята другим потоком, можно только переключить управление на него и подождать наступления события.
- В многопроцессорных системах есть альтернатива: прогнать некоторое количество раз холостой цикл, проверяя каждый раз, не освободилась ли наша критическая секция. Если за SpinCount раз это не получилось, то выполняется переход к ожиданию. Это гораздо эффективнее, чем переключение на планировщик ядра и обратно.

Пример использования критической секции

```
CRITICAL_SECTION cs;
DWORD WINAPI SecondThread()
{
   InitializeCriticalSection(&cs); EnterCriticalSection(&cs);
   //…критический участок кода
   LeaveCriticalSection(&cs);
}
```

Использование нескольких критических секций

- В том случае, когда процесс работает с двумя ресурсами, доступ к которым должен выполняться последовательно, он может создать несколько критических секций.
- Однако в этом случае все потоки должны использовать одинаковую последовательность входа в эти критические секции и выхода из них, иначе возможны взаимные блокировки потоков.

Опасный код

Поток 1.

```
EnterCriticalSection(&cs1);
EnterCriticalSection(&cs2);

//…критический участок кода

LeaveCriticalSection(&cs2);
LeaveCriticalSection(&cs1);
```

Поток 2.

```
EnterCriticalSection(&cs2);
EnterCriticalSection(&cs1);

//…критический участок кода

LeaveCriticalSection(&cs1);
LeaveCriticalSection(&cs2);
```

Правила хорошего тона

- Критические секции должны быть короткими.
- Критических секций не должно быть много.
- Критическая секция не должна быть одна на весь код.

Реализация критических секций

- Функции *EnterCriticalSection ()* и *LeaveCriticalSection ()* реализованы на основе атомарных Interlocked-функций, поэтому они более производительны, чем мьютексы!!!

Сравнение инструментов синхронизации Windows 2000+

	CRITICAL_ SECTION	Мьютекс	Семафор	Событие
Именованный защищаемый объект	Нет	Да	Да	Да
Доступность из нескольких процессов	Нет	Да	Да	Да
Синхронизация	Вхождение	Ожидание	Ожидание	Ожидание
Освобождение	Выход	Мьютекс может быть освобожден или оставлен без контроля.	Освобождается любым потоком.	Функции SetEvent, PulseEvent.
Права владения	В каждый момент времени иметь права владельца может только один поток. Владеющий поток может осуществлять вхождение несколько раз, не блокируя свое выполнение.	В каждый момент времени иметь права владельца может только один поток. Владеющий поток может выполнять функцию ожидания несколько раз, не блокируя свое выполнение.	Понятие владения неприменимо. Доступ разрешен одновременно нескольким потокам, число которых ограничено максимальным значением счетчика.	Понятие владения неприменимо. Функции SetEvent и PulseEvent могут быть вызваны любым потоком.
Результат освобождения	Разрешается вхождение одного потока из числа ожидающих.	Вслед за последним освобождением права владения разрешается приобрести одному потоку из числа ожидающих.	Продолжать выполнение могут несколько потоков, число которых определяется текущим значением счетчика.	После вызова функций *SetEvent* или *PulseEvent* продолжать выполнение будет один или несколько ожидающих потоков.

Межпроцессное взаимодействие

Атомарные операции и lockless программирование

Lockless программирование

- Lockless программирование – разработка неблокирующих многопоточных приложений.
- Отказ от использования блокирующих примитивов типа мьютексов и даже критических секций для доступа к разделяемым данным.
- Достоинство – повышенная производительность многопоточных приложений на многоядерных процессорах.

Атомарные операции как lockless-инструмент

- Простейшим способом lockless-программирования является активное использованием атомарных операций при конкурентном доступе нескольких потоков к общим переменным.
- В достаточно частых случаях необходимо обеспечить конкурентный доступ к какой-либо целочисленной переменной, являющейся счетчиком. Тогда бывает достаточно просто обеспечить атомарность выполнения операций увеличения, уменьшения или изменения значения переменной.

Виды атомарных операций

- Все инструкции вида Операция Регистр-Регистр можно считать атомарными так как регистры за пределами вычислительного процессорного ядра не видны.
- Загрузка данных из памяти по выровненному адресу в регистр общего назначения.
- Сохранение данных из регистра общего назначения в память по выровненному адресу.
- Специальные операции для атомарной работы (например, **cmpxchg**).
- Многие команды вида Чтение-Модификация-Запись могут быть сделаны искусственно атомарными с помощью операции блокировки шины (префикс **lock**).

Реализация атомарных операций в Windows 2000+

- Для увеличения значения целочисленных переменных –InterlockedIncrement, InterlockedIncrement64.
- Для уменьшения значения целочисленных переменных – InterlockedDecrement, InterlockedDecrement64.
- Для изменения значений целочисленных переменных – InterlockedExchange, InterlockedExchange64, InterlockedExchangeAdd, InterlockedExchangePointer.
- Для изменения значений целочисленных переменных со сравнением – InterlockedCompareExchange, InterlockedCompareExchangePointer.

Пример кода с использованием атомарной операции

```
static DWORD array [100];
…
for (int i = 0; i < 100; i++)
  InterlockedIncrement(array+i);
```

Эффект переупорядочивания

- Поток 1:

```
result = 7;
flag = TRUE;
```

- Поток 2:

```
if (flag) show(result);
```

- Какое значение будет передано в функцию show?

- Значение result попадающее в show() совершенно не обязательно будет равно 7.
- Всему виной – переупорядочивание (reordering).

Переупорядочивание и модель памяти

- Чтение или запись не всегда будет происходить в том порядке, который указан в вашем коде.
- Переупорядочивать операции **могут компилятор, исполняемая среда и процессор**.
- Говоря о переупорядочивании, мы приходим к термину модели памяти (memory consistency model или просто memory model).

«Сильная» и «слабая» модели памяти

- Модель памяти, в которой нет переупорядочиваний чтения и записи будет считаться сильной (strong), а модель памяти, где возможны любые переупорядочивания чтения и записи принято считать слабой (weak).
- При этом слабые модели обеспечивают наибольшую производительность за счет возможных оптимизаций, но и порождают большое количество проблем при программировании.

Сводная таблица для некоторых архитектур

Переупорядо-чивание	Архитектура					
	x86	em64t	amd64	ia64	xbox360	CLR2.0+
Чтение после чтения	Нет	Нет	Нет	Да	Да	Да
Чтение после записи (запись перед чтением)	Нет	Нет	Да	Да	Да	Да
Запись после записи	Нет	Нет	Нет	Да	Да	Нет
Запись после чтения (чтение перед записью)	Да	Да	Да	Да	Да	Да

Барьеры памяти и оптимизации

- Для борьбы с переупорядочиванием применяются так называемые барьеры памяти (memory barrier или memory fence).
- В случае барьера для компилятора применяют еще и термин барьер оптимизации (optimization barrier).
- Барьеры могут быть как явными, так и не явными (с точки зрения программиста).
- Кроме того барьеры могут быть полными, двухсторонними и односторонними.

Полные барьеры

- Полный барьер (full fence) предотвращает любые переупорядочивания операций чтения или записи через него.
- Можно говорить о том, что все операции чтения и записи до полного барьера будут завершены, по отношению к операциям, расположенным после барьера.
- Данный барьер предлагает использование инструкции mfence для x86/x64 архитектуры.

Двухсторонние барьеры

- Двусторонние барьеры (Store fence и Load fence) предотвращают переупорядочивание лишь одного вида операций.
- Барьер записи (store fence) не позволяет переупорядочивать через себя операции записи.
- Барьер чтения (load fence) не позволяет переупорядочивать через себя операции чтения соответственно.
- На x86/x64 архитектурах данные барьеры реализованы в инструкциях lfence и sfence.

Односторонние барьеры

- Односторонние барьеры обычно реализуют одну из двух семантик – write release (store release) или read acquire (load acquire).
- Write release семантика предотвращает любое переупорядочивание чтения и записи через барьер до барьера (запрет ↓), но не предотвращает переупорядочивания после него.
- Read acquire семантика предотвращает переупорядочивание чтения и записи через барьер после барьера (запрет ↑), но не предотвращает переупорядочивание до него.

Управление переупорядочиванием в MS VC

- MS VC для предотвращения переупорядочивания инструкций со стороны компилятора предлагает использовать *_ReadBarrier()*, *_WriteBarrier()*, *_ReadWriteBarrier()*.
- Эти функции не предотвращают переупорядочивание на уровне процессора, они являются лишь инструментом более тонкого контроля над оптимизациями со стороны компилятора.
- Для предотвращения переупорядочивания инструкций со стороны процессора предлагает макрос *MemoryBarrier()*, который является полным барьером и предотвращает переупорядочивание как чтения, так и записи. Исходный код этого макроса можно увидеть в MSDN.

Неявные барьеры в MS VC 2005

- В случае MS VC 2005+ ключевое слово volatile приобретает значение, выходящее за рамки C++ стандарта.
- Запись в volatile переменную всегда реализует семантику одностороннего барьера write release.
- Чтение из volatile переменной всегда реализует семантику одностороннего барьера read acquire.
- Причем оба эти неявных барьера реализуются как для компилятора, так и для процессора.

http://msdn.microsoft.com/en-us/library/windows/desktop/ee418650%28v=vs.85%29.aspx

Пример решения проблемы переупорядочивания

- Поток 1:

```
result = 7; // store
_WriteBarrier();
flag = TRUE; // store
```

- Поток 2:

```
// load, load
if (flag) show(result);
```

- На x86 архитектуре в нашем случае (запись после записи) не надо бояться переупорядочивания, поэтому единственным и достаточным в коде будет барьер для компилятора *_WriteBarrier()*.
- При использовании MS VC 2005+, объявление переменной flag как volatile позволит отказаться от явного использования барьера.

Производительность lockless приложений

- *MemoryBarrier ()* занимает 20-90 циклов.
- *InterlockedIncrement ()* занимает 36-90 циклов.
- Вхождение или освобождение критической секции может занимать 40-100 циклов.
- Захват или освобождение мьютекса может занимать 750-2500 циклов.