**Лекция 9. Многопоточное программирование**

Все многозадачные ОС используют концепцию процесса. Процесс – абстракция, описывающая выполняющуюся программу, которая представляет собой единицу работы и заявку на потребление системных ресурсов. В различных ОС принято по разному называть выполняемую работу. В ОС windows вычислительный процесс организуется в рамках потока, который формируется с разу после запуска процесса, лишь поток способен потреблять такой ресурс как процессорное время. В рамках unix подобных систем существует понятие родительский процесс и дочерний. Родительский – процесс, запускаемый при запуске приложения, он способен потреблять все системные ресурсы: память, процессорное время, внешние устройства и тд. Дочерние процессы порождаются родительским процессом для организации различных вычислений. Понятие потока в unix подобных системах не существует.

Для того чтобы иметь возможность организации многопоточного программирование архитектура ОС должна удовлетворять:

1. Чередование нескольких процессов с целью повышения степени использования системных ресурсов и обеспечение разумного времени отклика.
2. Распределение ресурсов между процессами в соответсвии с заданной стратегией. Т.е. определенным функциям или приложениям может быть предоставлен более высокий приоритет, в тоже время ОС должна разрешать вопросы взаимоблокировки, когда процесс А блокирует процесс В, а процесс В блокирует А.
3. Необходима поддержка обмена информации между процессами, а так же возможность организации нового пользовательского процесса.

В рамках unix каждый процесс может находится в одном из 3-х состояний:

1. Состояние выполнения. Когда поток использует процессорное время для решения своей задачи.
2. Состояние готовности. Когда поток имеет доступ ко все ресурсам и ожидает только выделение процессорного времени.
3. Блокировка. Когда поток не может выполняется в связи с отсутствием доступа к необходимым системным ресурсам, кроме процессорного времени.

Т.е. из состояние выполнения поток может перейти в состояние блокировки. Как только системные ресурсы будут выделены поток переходит в состояние готовности, а далее в состояние выполнения.

В многопоточной системе традиционная концепция процесса разделена на 2 части. Одна из них связана с потреблением ресурсов, а другая с выполнением команд. Таким образом любой процесс может содержать множество параллельно работающих потоков. Потоки используются как средство распараллеливание вычислений. Важной частью ОС является подсистема управления процессами. Эта система распределяет процессорное время между потоками процессов, существующих в системе. Она занимается созданием и уничтожением потоков, а так же диспетчеризацией выделяемых системных ресурсов. Для организации данной работы в памяти поддерживается специальная информационная структура, которая записывает какие ресурсы выделены каждому процессу. Ряд ресурсов выделяется процессу в единоличное пользование, часть ресурсов выделяется в совместно пользование с другими процессами. Ресурсы могут выделяется процессу статически в момент создания и динамически – по запросу во время работы программы. Ресурсы могут быть прикреплены к времени работы процесса или на заданный промежуток. Процесс – контейнер для набора ресурсов используемыми его потоками. Процесс в ОС windows включает в себя:

1. Закрытое виртуальное адресное пространство. Т.е. диапазон адресов виртуальной памяти, которым может пользоваться данный пользовательский процесс.
2. Исполняемая программа. Начальный код и данные проектируемые на ВАП.
3. Таблицу открытых описателей системных ресурсов, к которым относятся доступные процессу объекты синхронизации, коммутационные порты, файлы и другое.
4. Процесс содержит контекст защиты, который называется маркером доступа. Маркер доступа идентифицируется пользователем и определяет группу безопасности и привилегии процесса. Маркер доступа есть у процесса и у его потоков.
5. Каждый процесс имеет идентификатор.
6. Каждый процесс имеет минимум один поток. При этом под потоком понимается объект ядра ОС, получающий процессорное время для работы. Без потока программа процесса не может быть выполнена.

Поток включает в себя следующие элементы:

1. Содержимое набора регистров процессора отражающих текущее состояние процессора.
2. Два стека. Один в пользовательском режиме, а другой в режиме ядра.
3. Закрытую область памяти.
4. Уникальный идентификатор потока. (идентификатор клиента)

Говорят что поток существует в контексте создавшего его потока.

**Создание и работа с потоками**

Каждый поток начинает свое выполнение с входной функции. Входная функция имеет следующий прототип.

Функция потока может выполнять любые задачи. Как только она заканчивает свое выполнение – поток автоматически завершается. При завершении потока система выполняет следующие действия:

1. Останавливает поток
2. Освобождает стек, удаляя из нее стекфрейм
3. Декремент счетчика пользователей для объекта ядра потока. Как только счетчик обнуляется – система удаляет поток. Таким образом объекта ядра может существовать чуть больше потока.

Данный механизм необходим, чтобы части программы могли получать доступ к информации даже если он завершен (например код завершения потока). Функция потока всегда должна возвращать какое-нибудь значение, которое будет использоваться как код завершения.

При разработке потоковой функции желательно обходится локальными переменными, либо параметрами. Использование глобальных переменных приводит к усложнению алгоритма из-за введения синхронизации.

Создание потока происходит путем ручного или автоматического вызова CreateThread(). Вызов этой функции создает объект ядра и возвращает его дескриптор. При выполнении данной функции система выделяет память под стек нового потока из адресного пространства процесса, инициализирует структуру данных потока и передет управление входной потоковой функции. Новый поток имеет доступ ко всем описателям процесса и к адресному пространству процесса, в контексте которого создан поток. Это позволяет потокам взаимодействовать друг с другом через общие объекты.

*HANDLE CreateThread(*

*Psa // указатель а структуру атрибутов безопасности. По умолчанию 0.*

*cbStack // Размер стека. Если 0, то по умолчанию.*

*pfnStartAddr // Указатель на потоковую функцию*

*pvParam // Параметр потоковой функции*

*tdwCreate // Дополнительный параметр потока Create\_SIJPENDED или 0*

*pdwThreadID // Идентификатор потока. Указатель на переменную, которая содержит идентификатор. Может быть NULL, тогда автоматически ОС присваивает идентификатор*

*)*

**Завершение потока**

Поток может завершиться:

* Поток самоуничтожается с помощью вызова **ExitThread**. Так делать не хорошо
* Функция потока возвращает управление при помощи **return**
* Один из потоков данного или стороннего процесса вызывает **TerminateThread.** *Так тоже делать не хорошо*
* Завершает процесс, содержащий данный поток. *Тоже плохо*. (ctrl+alt+del)

Функция потока должна спроектирована так, чтобы поток завершался только тогда, когда вернет управление. Единственный нормальный способ, который все делает хорошо. При этом:

* Любые объекты уничтожаются деструкторами, который автоматически вызываются
* Система корректно освобождает память, которую занимал стек
* Система устанавливает код завершения данного потока
* Декрементируется счетчик потоков данного ядра

С каждым потоком связан контекст, доступ к некоторым полям доступен при помощи **WinAPI.** В случае переключения контекста, система сохраняет контекст в стеке ядра. Контекст потока описывается в структуре контекст, которая определена в заголовочном файле с **Windows NT.**

Необходимо определять совместимость потоков стандартных библиотек и потоков.

В случае использование библиотек времени выполнения для запуска и завершения потоков используется beginThread и endthread. BeginThread создает поток и возвращает идентификатор. Endthread тормозит поток и передает системе необходимую информацию для ее завершения.

Основная сложность при синхронизации потоков это определение точки доступа к ресурсам и обеспечение отсутсвие взаимоблокировок (deadlock’ов)

Работу любого потока можно приостановить следующими способами:

* **Sleep(dwmiliseconds)**
* **WaitForSingleObject(hHandle, dwMilliseconds)**
* **WaitForMutipleObjects()**

dwMiliseconds = {0.N.INFINITE);

Для выполнения синхронизации используются такие объекты:

* **Mutex.** Объект ядра, который создается с помощью функции **createMutex()** и может находится в одном из двух состояний. Используется для защиты единичного ресурса от одновременного обращения к нему из разных потоков. В каждый момент времени объектом mutex владеет только один поток. Что исключает одновременный доступ из нескольких потоков в защищаемому ресурсу. Как только поток освобождает mutex. Он может быть захвачен другим потоком ожидающий в очереди. **ReleaseMutex** освобождает. Mutex является объектом ядра, что обеспечивает доступ к нему любому потоку процесса. Для того, чтобы проверить возможность захвата ресурса охраняемого mutex используется **WaitForSingleObject()**. У mutex есть набор флагов, которые задаются при создании. Второй флаг определяет как наличие владельца, если false, то владеют все.
* **CriticalSections.** Синхронизация в пользовательском режиме. Это не объект ядра, а кусок кода принадлежащий одному потоку, который должен выполняется без переключения контекстов. Критическая секция должна быть создана и доступна в пределах одного процесса. **EnterCriticalSection(&cs)**. **LeaveCriticalSection(&cs).** Все что находится между ними выполняется без переключения контекста.
* **Event.** Событие – объект синхронизации состояние которого может быть установлено в сигнальное путем вызова соотвествующей функции. Событие имеет 2 состояния:
  + **Установленное.**
  + **Сброшенное.**

Существуют 2 типа событий:

* + С ручным сбросом. Объект с сигнальным состоянием которого сохраняется до ручного вызова функции сброса. Как только состояние объекта установлено в сигнальное, все потоки находящиеся в цикле ожидания данного объекта продолжают свое выполнение, т.е. освобождаются.
  + С автоматическим сбросом. Объект сигнальное состояние которого сохраняется до тех пор, пока не будет освобожден поток ожидающий данного события, после чего система автоматически устанавливает не сигнальное состояние события. Если в системе отсутствуют потоки ожидающий события, то объект остается в сигнальном состоянии.

Допустим есть несколько потоков, которые ожидают событие. Если авто сброс, то оно позволяет работать только одному потоку. Все остальные потоки остаются в режиме ожидания. В случае использования события с ручным сбросом. Все потоки ожидающие данное событие могут быть запущены и работать до тех пор, пока какой-либо из потоков не запустит функцию сброса.

События удобно использовать в случаях, когда необходимо передавать информацию о возникновении какого-либо события между потоками. Например, при асинхронных операциях ввода-вывода. Поток в любой момент времени может создать события с помощью функции **CreateEvent**(). Создающий поток всегда устанавливает событие в начальное состояние, тут же может быть присвоено имя. Другие потоки могут обратиться к событию по его идентификатору, либо по имени. Любой поток для установки сигнального состояния события может использовать функции: **setEvent** и **PulseEvent.** В случае PulseEvent событие устанавливается в сигнальное состояние, а потом в не сигнальное, после освобождения заданного количества освобождающих поток. Если их нет, то pulseEvent автоматически сбрасывает событие. Функция **setEvent** применяется только в случае события с ручным сбросом. Сброс события осуществляется с помощью события **ResetEvent()**.

* **Semaphore.** Семафор – множественный мьютекс. Создается с помощью функции CreateSemaphore() и позволяет одному или нескольким потокам получить доступ к критическому ресурсу. Для ограничения количества потоков в семафоре есть счетчик. Семафор открыт если счетчик больше 0 и закрыт если равен 0. Для работы семафора определены две симофорных операции: **wait** и **release**. Wait выполняется всякий раз, когда поток желает получить доступ к критическому ресурсу и зайти за семафор. При прохождении за семафор декрементируется симофорный счетчик доступа. При попытке следующего потока попасть к следующему ресурсу анализируется счетчик доступа и если он равен 0, то поток останавливается, переходит в состояние блокировки. Когда какой-нибудь из потоков завершает работу с ресурсом, он должен выполнить release. Release предполагает инкремент счетчика доступа, что позволяет зайти за семафор другому потоку.