**синхронизация** - механизм упорядочивания выполнения программных блоков двух или более потоков

# ЗАДАНИЕ 1

bts байт который мы указали нулем

BTR байт который мы указываем единицей

# ЗАДАНИЕ 2

**Critical section** – механизм синхронизации нескольких потоков одного процесса, НЕ ЯВЛЯЕТСЯ объектом ядра OS.

**Критическая секция (Critical Section)** это участок кода, в котором поток (thread) получает доступ к ресурсу (например переменная), который доступен из других потоков. Объект критическая секция обеспечивает синхронизацию. Этим объектом может владеть только один поток, что и обеспечивает синхронизацию.

**CRTITICAL\_SECTION csFlag**; - объявление критической секции **InitializeCriticalSection**(LPCRITICAL\_SECTION lpCriticalSection); - инициализациия критической секции

**EnterCriticalSection**(LPCRITICAL\_SECTION lpCriticalSection); - функция входа в критическую секция

**LeaveCriticalSection**(LPCRITICAL\_SECTION lpCriticalSection); - функция выхода из критической секции

**DeleteCriticalSection**(LPCRITICAL\_SECTION lpCriticalSection); - удаление(деинициализация) критической секции

# ЗАДАНИЕ 3

**Mutex** – механизм синхронизации нескольких потоков разных процессов, является объектом ядра OS.

**Мьютекс ( «взаимное исключение»)** — это базовый механизм синхронизации. Он предназначен для организации взаимоисключающего доступа к общим данным для нескольких потоков с использованием барьеров памяти (для простоты можно считать мьютекс дверью, ведущей к общим данным).

**Создание CreateMutex():**

**CreateMutex(**

**LPSECURITY\_ATTRIBUTES lpMutexAttributes, // атрибут безопастности**

**BOOL bInitialOwner, // флаг начального владельца**

**LPCTSTR lpName // имя объекта);**

**Освобождения объекта ReleaseMutex():**

**BOOL ReleaseMutex(**

**HANDLE hMutex // дескриптор mutex);**

# ЗАДАНИЕ 4

**Семафор** объект синхронизации позволяет ограничить доступ потоков к объекту синхронизации на основании их количества.

Сначала семафор инициализируется и ему передается количество потоков, которые к нему могут обратиться.

Дальше при каждом обращении к ресурсу его счетчик уменьшается. Когда счетчик уменьшиться до 0 к ресурсу обратиться больше нельзя. При отсоединении потока от семафора его счетчик увеличивается, что позволяет другим потокам обратиться к нему.

Сигнальному состоянию соответствует значение счетчика больше нуля. Когда счетчик равен нулю, семафор считается не установленным

**Создание CreateSemaphore ():**

**HANDLE CreateSemaphore(**

**LPSECURITY\_ATTRIBUTES lpSemaphoreAttributes, // атрибут доступа**

**LONG lInitialCount,// инициализированное начальное состояние счетчика**

**LONG lMaximumCount, // максимальное количество обращений**

**LPCTSTR lpName // имя объекта);**

**Освобождение счетчика ReleaseSemaphore():**

**BOOL ReleaseSemaphore(**

**HANDLE hSemaphore, // дескриптор семафора**

**LONG lReleaseCount, // на сколько изменять счетчик**

**LPLONG lpPreviousCount // предыдущее значение);**

# ЗАДАНИЕ 5

Объекты-события используются для уведомления ожидающих нитей о наступлении какого-либо события. Различают два вида событий - с ручным и автоматическим сбросом. Ручной сброс осуществляется функцией ResetEvent. События с ручным сбросом используются для уведомления сразу нескольких нитей. При использовании события с автосбросом уведомление получит и продолжит свое выполнение только одна ожидающая нить, остальные будут ожидать дальше.

**Создается CreateEvent()**

**HANDLE CreateEvent(**

**LPSECURITY\_ATTRIBUTES lpEventAttributes, // атрибут защиты**

**BOOL bManualReset, // тип сброса TRUE - ручной**

**BOOL bInitialState,// начальное состояние TRUE - сигнальное**

**LPCTSTR lpName // имя обьекта);**

**Открывает существующий именованный объект события OpenEvent():**

**HANDLE OpenEventA(**

**DWORD dwDesiredAccess, // Доступ к объекту события**

**BOOL bInheritHandle, // Если это значение TRUE , процессы, созданные этим процессом, унаследуют дескриптор. В противном случае процессы не наследуют этот дескриптор.**

**LPCSTR lpName // Название открываемого события );**

**Изменение состояния на сигнальное SetEvent():**

**BOOL SetEvent(**

**HANDLE hEvent // дескриптор события);**

**Изменение состояния на невыделенное ResetEvent():**

**BOOL ResetEvent(**

**HANDLE hEvent // дескриптор события);**

**Устанавливает указанный объект события в сигнальное состояние, а затем сбрасывает его в несигнальное состояние после освобождения соответствующего количества ожидающих потоков PulseEvent():**

**BOOL PulseEvent(**

**HANDLE hEvent //Дескриптор объекта события);**

# ЗАДАНИЕ 6 LINUX

Функция **pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr)** предназначена для инициализации мьютекса. где первый аргумент – указатель на мьютекс, а второй – аттрибуты мьютекса.

После создания мьютекса он может быть захвачен с помощью функции

**int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);**

После этого участок кода становится недоступным остальным потокам – их выполнение блокируется до тех пор, пока мьютекс не будет освобождён. Освобождение должен провести поток, заблокировавший мьютекс, вызовом

**int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);**

После использования мьютекса его необходимо уничтожить с помощью функции

**int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);**