THREAD SYNCHRONIZATION

- **Win32**-based operating systems provide several **synchronization objects** which can be used to synchronize several **threads** with each other.
- In general, a thread will synchronize itself with another thread by putting itself to sleep.
- Before putting itself to sleep, a thread will tell the **OS** which "**special event**" has to occur in order for it to resume **execution**.
- The OS will watch for that event on the threads behalf and when it does occur, it will schedule a **CPU slice** for the thread.
- Without synchronization objects, threads are forced to synchronize themselves with other threads by using techniques such as continuously **polling** a **shared** (global) **variable(s)**.
- We will look at three main ones: **Critical Sections**, **Mutexes**, and **Semaphores**.
- Most synchronization objects are known as **kernel objects**, which means that they are managed by the **operating system kernel** and manipulated using **handles**.
- The exception is **Critical Sections** which are not kernel objects.

Critical Sections

- These are the **simplest** to use.
- A critical section is simply a small section of code that requires **exclusive access** to some **shared data** before it can execute.

- Once a thread enters a critical section, no other thread may execute the code in the critical section until the first thread has left the critical section.
- Before the code in the critical sections can be executed, the OS will check a global record which indicated whether or not some other thread is in the critical section.
- This may sound like suspiciously similar to the polling a global technique. The difference however is that it is the OS which manipulates the global and not the program.
- Thus, critical sections allow only one thread at a time to gain access to a region of data.
- There are **four functions** which are used for creating and manipulating critical sections.

- The first step is to **define** a critical section object using a **global variable** of type **CRITICAL_SECTION**:

  CRITICAL_SECTION cs;

- The next step is to initialize the critical section object using one of the threads:

  InitializeCriticalSection (&cs);

- This will **create** a critical section object **cs**.
- A thread can now enter the critical section by calling:

  EnterCriticalSection (&cs);

- If the call is successful, (i.e., the function call returns), that thread is said to "**own**" the critical section.
- No two threads can own a critical section at the same time. If one thread already owns the critical section, the next thread to call **EnterCriticalSection** will be **suspended** in the function call (it will not return).
- The suspended function will return only when the first thread leaves the critical section by calling:

  LeaveCriticalSection (&cs);

- At this point, the suspended function will own the critical section and the function call will return.
- Note that it is possible to have more than one thread suspended on a critical section.
- When a thread releases a critical section, the system will wake up the first thread suspended on that critical section and gives it ownership of the critical section object. The rest of the blocked threads continue to sleep.
- Also, it is quite legal for a single thread to call **EnterCriticalSection** multiple times. Before another thread can own a critical section, the thread currently owning it must call **LeaveCriticalSection** multiple times until the critical section object **reference count** goes to **zero**.
- When a program no longer requires a critical section, it deletes it by calling:

  DeleteCriticalSection (&cs);

- Deleting a critical section object releases all system resources used by the object.
- The code given illustrates the use of critical sections.

MUTEXES (Mutual Exclusion)

- The main limitation with **critical sections** is that they can only be used to synchronize threads within the **same process**.
- **Mutexes** are very much like critical sections except that they can be used to synchronize threads across **multiple processes**.
- Only one thread can own a mutex at a time.
- The state of a mutex object is **signaled** when it is not owned by any thread.
- If a thread owns the mutex, then the mutex is said to be **non-signaled**.
- The first step is to **create** the **mutex** as follows:

HANDLE CreateMutex (LPSECURITY_ATTRIBUTES lpMutexAttributes, BOOL bInitialOwner, LPCTSTR lpName);

lpMutexAttributes

Points to a **SECURITY_ATTRIBUTES** structure that defines the security attributes for the mutex object.

bInitialOwner

Specifies whether or not the mutex is owned by the calling thread. This parameter can be used to send a mutex immediately into a signaled state.

lpName

Used to give the mutex object a name. An unnamed mutex can be created as:

hMutex = CreateMutex (NULL, FALSE, NULL);

- Threads can now simply use **hMutex** (process relative handle) to manipulate the object.

- Another method for obtaining a **mutex handle** is using the **OpenMutex** function:

HANDLE OpenMutex (DWORD fdwAccess, BOOL fInherit, LPCTSTR lpszMutexName);

fdwAccess

Specifies the requested access to the mutex object:

MUTEX_ALL_ACCESS: Specifies all possible access flags for the mutex object.

SYNCHRONIZE: Windows NT: Enables use of the mutex handle in any of the wait functions to acquire ownership of the mutex, or in the ReleaseMutex function to release ownership.

fInherit

Specifies whether any child process created by this process should inherit this handle to this mutex object.

lpszMutexName

Zero-terminated string name of the mutex object.

- Now instead of calling **EnterCriticalSection**, we use the **WaitForSingleObject** function:

DWORD WaitForSingleObject (HANDLE hObject, DWORD dwTimeout);

hObject

The handle to the mutex.

dwTimeout

Specifies the time-out interval, in milliseconds the function should wait before

returning.

- This function only **returns** when the mutex becomes **signaled** (it can now own the mutex) , or if the **time** specified **elapses**.
- Therefore, if we specify the **INFINITE** flag for the time-out, the routine will return only when the **mutex** is **signaled**, that is, the thread immediately grabs ownership of the mutex, which places it back into the **nonsignaled** state.

- A related function is the **WaitForMultipleObjects** function which returns either when any **one** or when **all** of the **specified objects** are in the **signaled** state, or when the **time-out** interval **elapses**.

DWORD WaitForMultipleObjects (DWORD cObjects, CONST HANDLE * lphObjects, BOOL fWaitAll, DWORD dwTimeout);

cObjects

Specifies the number of object handles in the array pointed to by **lphObjects**.

lphObjects

Specifies the an array of object handles.

fWaitAll

The wait type.

If **TRUE**, the function returns when all objects in the **lphObjects** array are **signaled** at the **same time**.

If **FALSE**, the function returns when any **one** of the objects is **signaled**. In the latter case, the return value indicates the object whose state caused the function to return.

dwTimeout

Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the conditions specified by the **fWaitAll** parameter are not satisfied.

If **dwTimeout** is **zero**, the function tests the states of the specified objects and returns **immediately**.

If **dwTimeout** is **INFINITE**, the function's time-out interval **never elapses**.

- When a thread is finished with the mutex, it must release the object using **ReleaseMutex**:

  BOOL ReleaseMutex (HANDLE hMutex);

  hMutex

  The handle to the mutex object.

- The **ReleaseMutex** function is analogous to the **LeaveCriticalSection** function.
- It changes the mutex from the **non-signaled** state to the **signaled** state.
- When the program does not need the mutex at all, it is closed using the **CloseHandle** function:

  BOOL CloseHandle (HANDLE hObject);