SEMAPHORES

- **Semaphores** are a form of **Interprocess Communications** (**IPC**) facility.
- These **kernel objects** are used for **resource counting**. That is, a resource **counter** is used to check the availability of a particular resource.
- The **counter** is **decremented** when a thread (or process) is **granted** the semaphore and **incremented** when the semaphore is **released**.
- Semaphores perform this **increment** and **decrement** operation **atomically**.
- That is, when a request is made for that resource, the system will ensure that the resource is available and decrements the counter while stopping other threads from acting on the counter (requesting the resource).
- Once the counter has been decremented, the system then allows other threads to request that resource.
- Several threads can affect a semaphore's resource count and therefore, unlike a critical section or mutex, a **semaphore** is **not** considered to be **owned** by a thread.
- This implies that we can have one thread waiting for a semaphore object (decrement counter) and another release the semaphore (increment counter).
- A semaphore is said to **signaled** when its resource count is **greater than zero**, and is **non-signaled** when the resource count is **equal to 0**.
- Each time a thread calls **WaitForSingleObject** with the semaphore handle, the system will first the resource count to see if it is greater than zero.
- If it is **greater than zero**, the system **decrements** the resource **count** and **wakes** the thread.
- If the **count is zero**, the system **suspends** the **thread** until another thread releases the semaphore by incrementing the resource counter.
- Consider an example where we have 3 serial ports on machine, and each port is assigned to one thread at a time. Therefore more than three threads can use the ports at any given time.
- We can now create a semaphore to monitor the ports with an initial counter value set to 3 (i.e., 3 ports).
- Once three threads have done a **WaitForSingleObject** on that semaphore, its resource counter will be zero and therefore any other thread requesting access to the port will be suspended.
- The first step is to create a semaphore as follows:

  HANDLE CreateSemaphore (LPSECURITY_ATTRIBUTES lpsa, LONG lSemInitial, LONG lSemMax, LPCTSTR lpszSemName);

- The **lpsa** parameter is a pointer to a **SECURITY_ATTRIBUTES** structure that specifies the security attributes for the semaphore object.
- The **lSemInitial** parameter is used to specify an **initial resource count** for the semaphore. In our serial port example we would set it to 3 at the start of the application to indicate all three ports are available.
- The **lSemMax** is used to set the **maximum resource count** for the semaphore object. In our serial port example we would set it to 3 to represent the three serial ports.
- Note that if we set **lSemMax** to 1, the semaphore essentially becomes a mutex, allowing only one thread access to the resource at any time.

- The **lpszSemName** parameter is used to assign a character string (**name**) to the semaphore object.
- The string name can then be used by other processes to get the semaphore handle by calling:

HANDLE OpenSemaphore (DWORD fdwAccess, BOOL fInherit, LPCTSTR lpszSemName);

- The syntax is similar to the **OpenMutex** function discussed earlier.
- A thread can **release** the semaphore by calling:

BOOL ReleaseSemaphore (HANDLE hSemaphore, LONG cReleaseCount, LPLONG lplPreviousCount);

- The **ReleaseSemaphore** function **increases** the **count** of the specified semaphore object by a **specified amount** (i.e., increment the resource count by a specified amount).
- This function is similar to the **ReleaseMutex** function, except for two differences.
- First, any thread can call this function at any time because threads cannot own semaphore objects.
- Second, the **cReleaseCount** parameter can be used to increment the resource count of the semaphore by more than one.
- The **hSemaphore** parameter Identifies the semaphore object.
- The **lplPreviousCount** parameter points to a 32-bit variable to receive the previous count for the semaphore. This parameter can be NULL if the previous count is not required.
- Note that in Win32 there is no way to get the count of a semaphore without altering it.
- The **cReleaseCount** specifies the amount by which the semaphore object's current count is to be increased.
- Consider an application which copies data from one serial port to another.
- The application will have to acquire two serial ports by acquiring the semaphore for that serial port twice by calling **WaitForSingleObject** twice.
- Later on it can release both ports with just one **ReleaseSemaphore** call with **cReleaseCount** set to 2.

- The following code fragment illustrates this:

  // Get two serial ports.

  WaitForSingleObject (hSemSerialPort, INFINITE);

  WaitForSingleObject (hSemSerialPort, INFINITE);

  .....

  .....

  // Use the serial ports to do the data transfer

  .....

  .....

  // Release the serial ports so other applications may use them

  ReleaseSemaphore (hSemSerialPort, 2, NULL);

- The syntax for the **WaitForSingleObject** call syntax is as follows:

  DWORD WaitForSingleObject (HANDLE hObject, DWORD dwTimeout);

- This function call **waits** on a semaphore (or any other object specified by the handle) and upon **successful** completion, it will **decrement** the **counter** associated with the semaphore object.
- The **hObject** parameter identifies the object.
- The **dwTimeout** parameter specifies the time-out interval, in milliseconds, how long the function will wait.
- If **dwTimeout** is **INFINITE**, the function's time-out interval never elapses.
- The code given illustrates the use of semaphores in our thread synchronization example.