System Programming in Windows

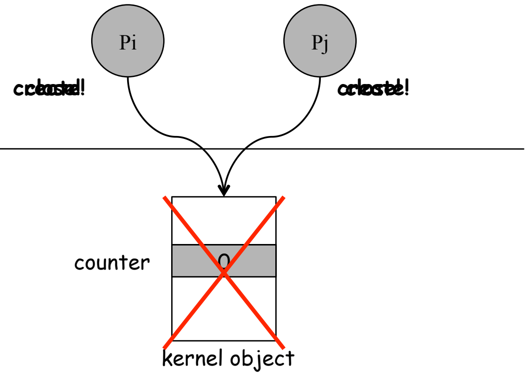
- Naming in Windows: Kernel Objects and Kernel Object Handles
- Processes, Jobs, Threads
- Synchronization

Kernel Objects

- Whenever you want to access a kernel entity (file, process, semaphore, etc.) you request a **kernel object**.
  - Access token object
  - File object
  - File mapping object
  - Job object
  - Mutex object
  - Pipe object
  - Process object
  - Semaphore object
  - Thread object
  - Waitable timer object

Object Lifetime and Garbage Collection

- Objects can be accessed from multiple processes.
- **Counters** keep track of that.



Creating Kernel Objects

```
HANDLE CreateThread(
  PSECURITY_ATTRIBUTES psa,
  size_t dwStackSize,
  LPTHREAD_START_ROUTINE pfnStartAddress,
  PVOID  pvParam,
  DWORD  dwCreationFlags,
  PDWORD pdwThreadId);
```

```
HANDLE CreateFile(
  PCTSTR pszFileName,
  DWORD  dwDesiredAccess,
  DWORD  dwShareMode,
  PSECURITY_ATTRIBUTES psa,
  DWORD  dwCreationDisposition,
  HANDLE hTemplateFile);
```

```
HANDLE CreateFileMapping(
  HANDLE hFile,
  PSECURITY_ATTRIBUTES psa,
  DWORD  flProtect,
  DWORD  dwMaximumSizeHigh,
  DWORD  dwMaximumSizeLow,
  PCTSTR pszName);
```

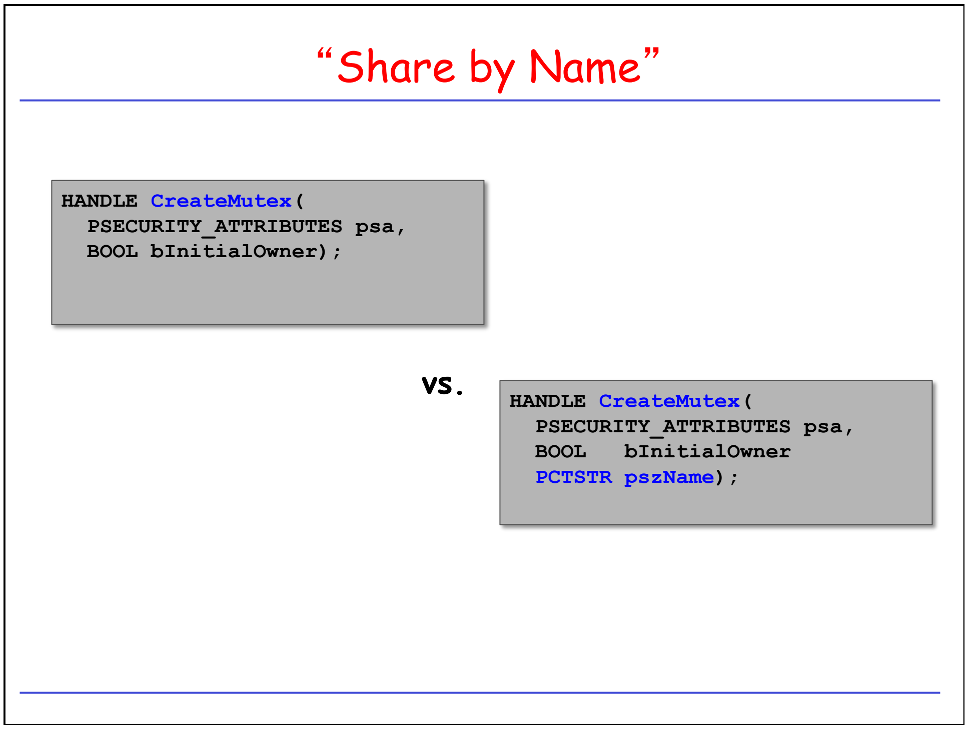
Closing Kernel Objects

```
// Indicate to the system that you
// are done manipulating the object.

BOOL CloseHandle(HANDLE hObject);
```

Note: Application may **leak** objects, but when **process** terminates, handles are closed.

Sharing Kernel Objects

- Q: How do we share pipes, semaphores, etc. across processes?
- “Share by Handle Inheritance”
- “Share by Name”
- “Share by Handle Duplication”

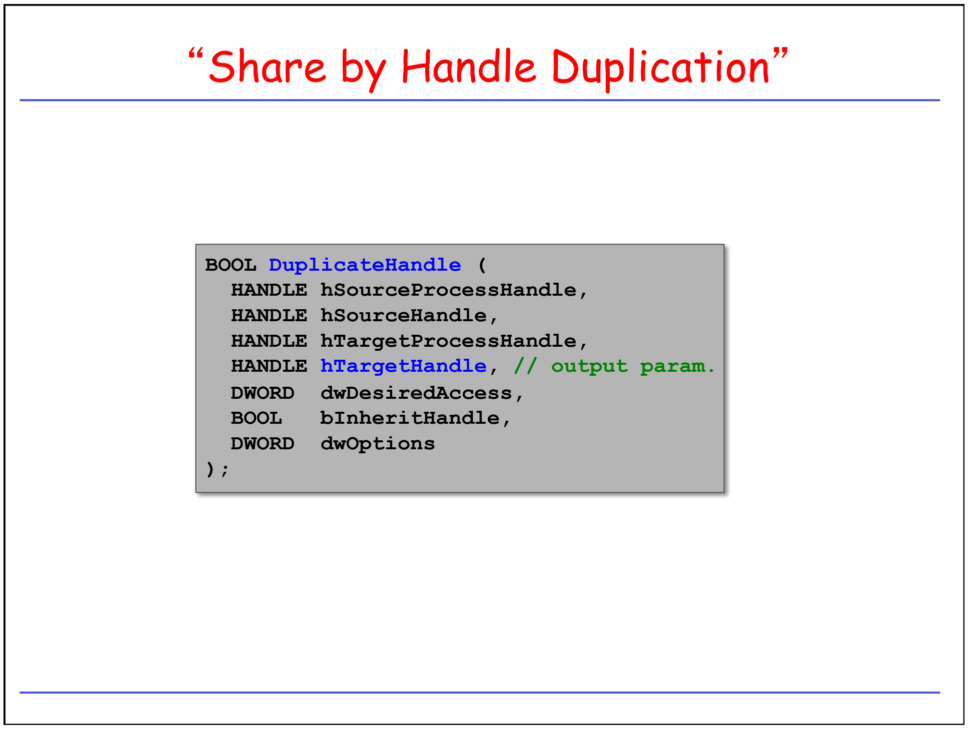


“Share by Name”

```
HANDLE CreateMutex(
  PSECURITY_ATTRIBUTES psa,
  BOOL bInitialOwner);
```

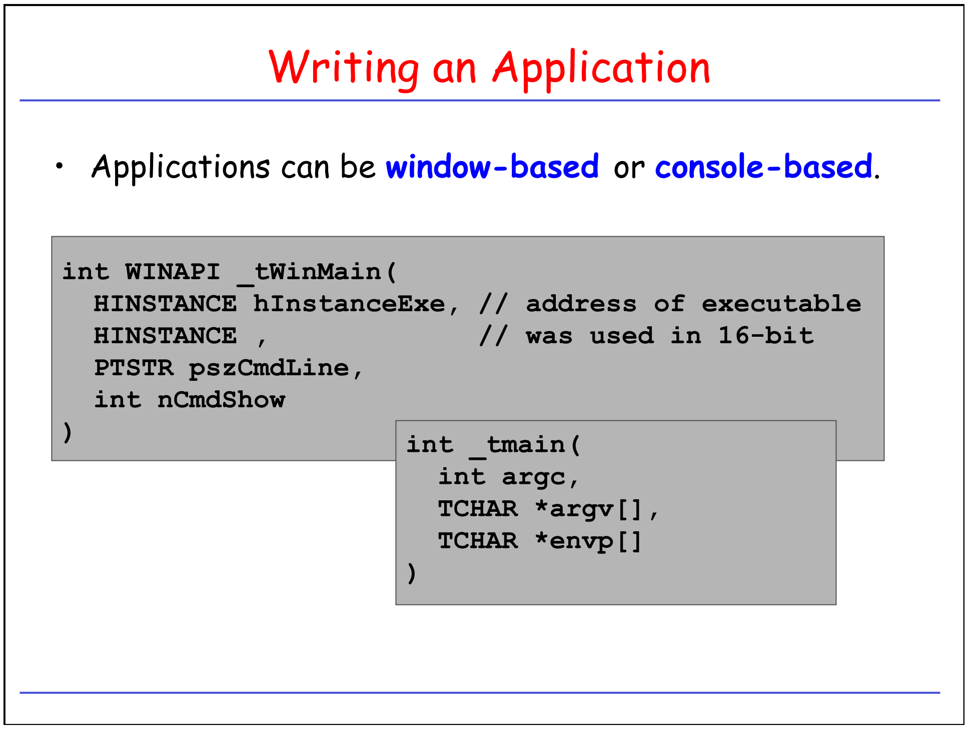
vs.

```
HANDLE CreateMutex(
  PSECURITY_ATTRIBUTES psa,
  BOOL    bInitialOwner
  PCTSTR pszName);
```



“Share by Handle Duplication”

```
BOOL DuplicateHandle (
  HANDLE hSourceProcessHandle,
  HANDLE hSourceHandle,
  HANDLE hTargetProcessHandle,
  HANDLE hTargetHandle, // output param.
  DWORD  dwDesiredAccess,
  BOOL   bInheritHandle,
  DWORD  dwOptions
);
```

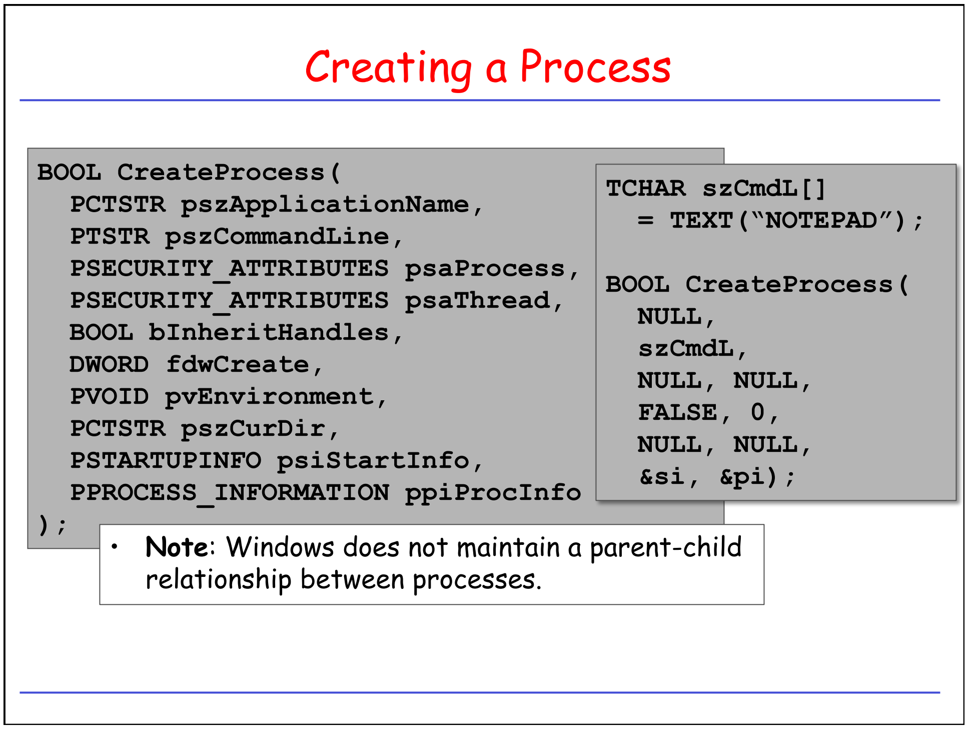


Writing an Application

- Applications can be **window-based** or **console-based**.

```
int WINAPI _tWinMain(
  HINSTANCE hInstanceExe, // address of executable
  HINSTANCE ,             // was used in 16-bit
  PTSTR pszCmdLine,
  int nCmdShow
)
```

```
int _tmain(
  int argc,
  TCHAR *argv[],
  TCHAR *envp[]
)
```

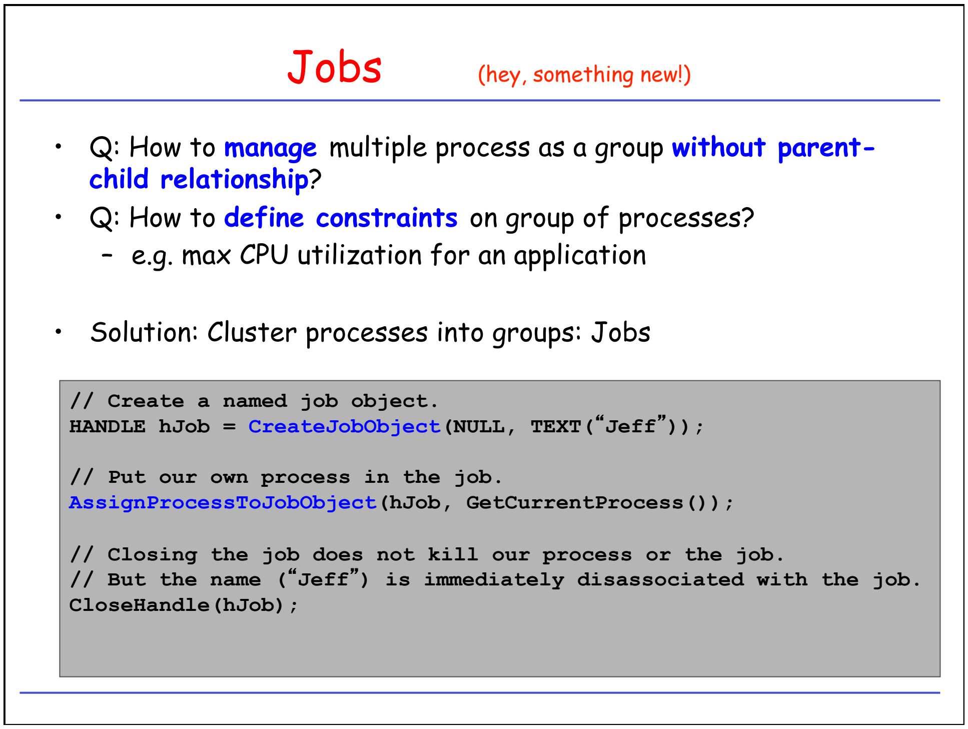


Creating a Process

```
BOOL CreateProcess(
  PCTSTR pszApplicationName,
  PTSTR pszCommandLine,
  PSECURITY_ATTRIBUTES psaProcess,
  PSECURITY_ATTRIBUTES psaThread,
  BOOL bInheritHandles,
  DWORD fdwCreate,
  PVOID pvEnvironment,
  PCTSTR pszCurDir,
  PSTARTUPINFO psiStartInfo,
  PPROCESS_INFORMATION ppiProcInfo
);
```

```
TCHAR szCmdL[]
  = TEXT(“NOTEPAD”);

BOOL CreateProcess(
  NULL,
  szCmdL,
  NULL, NULL,
  FALSE, 0,
  NULL, NULL,
  &si, &pi);
```

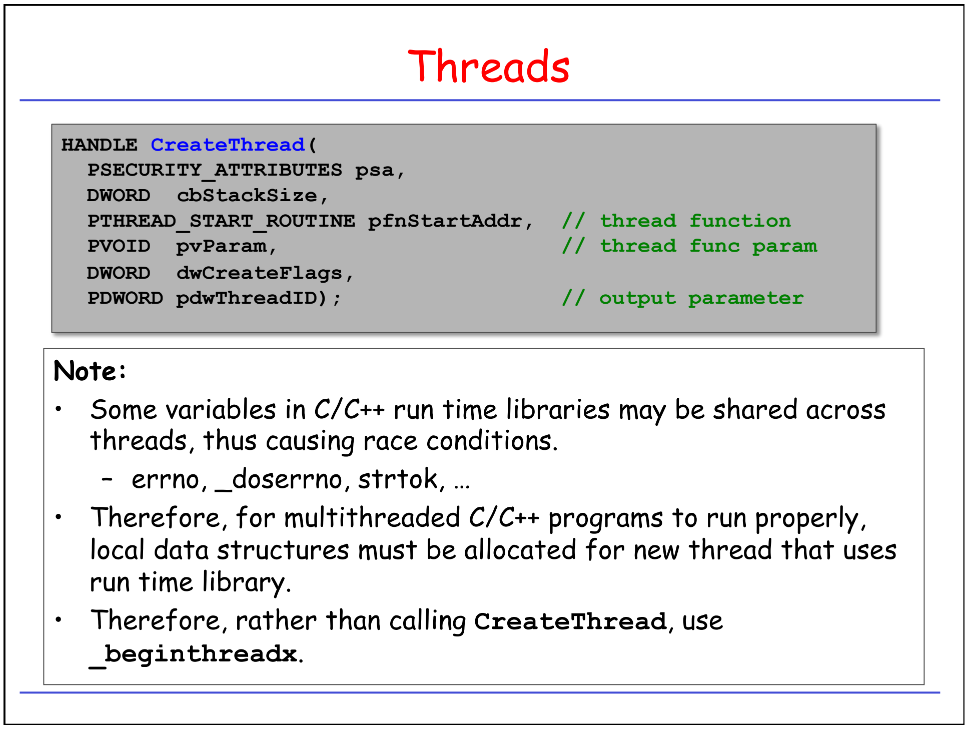
- **Note**: Windows does not maintain a parent-child relationship between processes.



Jobs (hey, something new!)

- Q: How to **manage** multiple process as a group **without parent-child relationship**?
- Q: How to **define constraints** on group of processes?
  - e.g. max CPU utilization for an application
- Solution: Cluster processes into groups: Jobs

```
// Create a named job object.
HANDLE hJob = CreateJobObject(NULL, TEXT("Jeff"));

// Put our own process in the job.
AssignProcessToJobObject(hJob, GetCurrentProcess());

// Closing the job does not kill our process or the job.
// But the name ("Jeff") is immediately disassociated with the job.
CloseHandle(hJob);
```

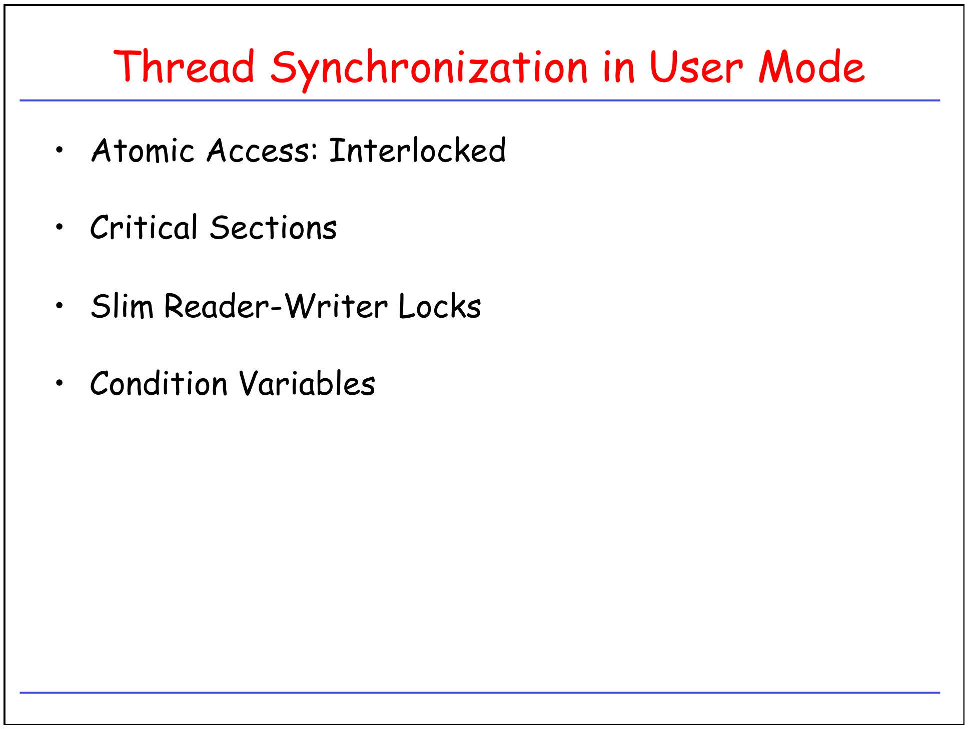


Threads

```
HANDLE CreateThread(
  PSECURITY_ATTRIBUTES psa,
  DWORD   cbStackSize,
  PTHREAD_START_ROUTINE pfnStartAddr,   // thread function
  PVOID   pvParam,                      // thread func param
  DWORD   dwCreateFlags,
  PDWORD pdwThreadID);                  // output parameter
```

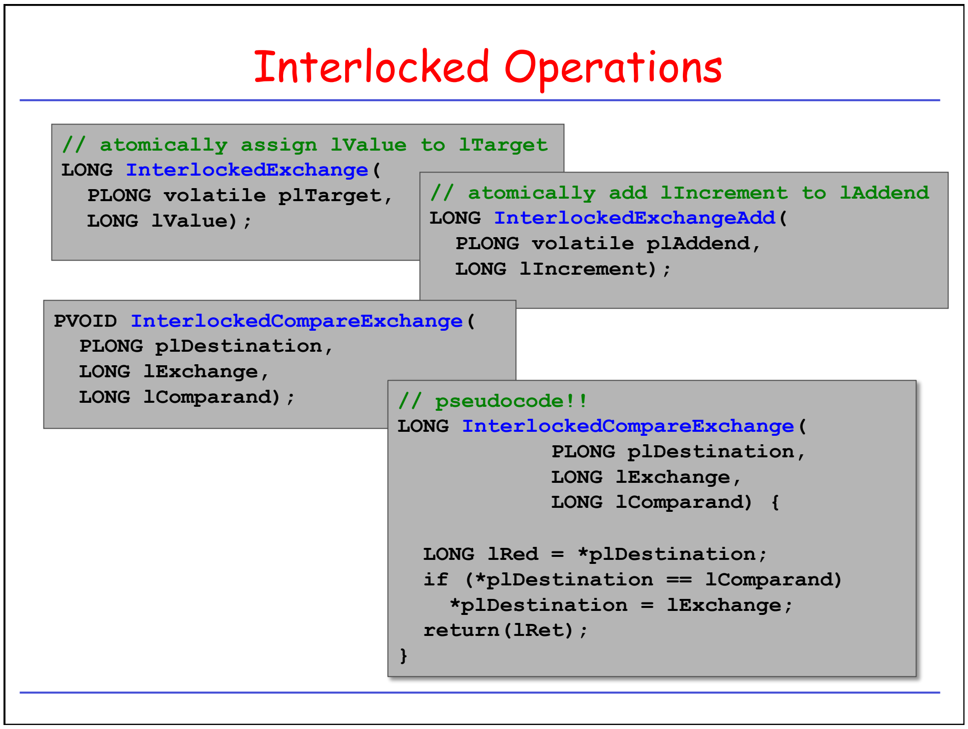
Note:

- Some variables in C/C++ run time libraries may be shared across threads, thus causing race conditions.
  - errno, _doserrno, strtok, …
- Therefore, for multithreaded C/C++ programs to run properly, local data structures must be allocated for new thread that uses run time library.
- Therefore, rather than calling `CreateThread`, use `_beginthreadx`.



Thread Synchronization in User Mode

- Atomic Access: Interlocked
- Critical Sections
- Slim Reader-Writer Locks
- Condition Variables



Interlocked Operations

```
// atomically assign lValue to lTarget
LONG InterlockedExchange(
  PLONG volatile plTarget,
  LONG lValue);
```

```
// atomically add lIncrement to lAddend
LONG InterlockedExchangeAdd(
  PLONG volatile plAddend,
  LONG lIncrement);
```

```
PVOID InterlockedCompareExchange(
  PLONG plDestination,
  LONG lExchange,
  LONG lComparand);
```

```
// pseudocode!!
LONG InterlockedCompareExchange(
            PLONG plDestination,
            LONG lExchange,
            LONG lComparand) {

  LONG lRed = *plDestination;
  if (*plDestination == lComparand)
    *plDestination = lExchange;
  return(lRet);
}
```

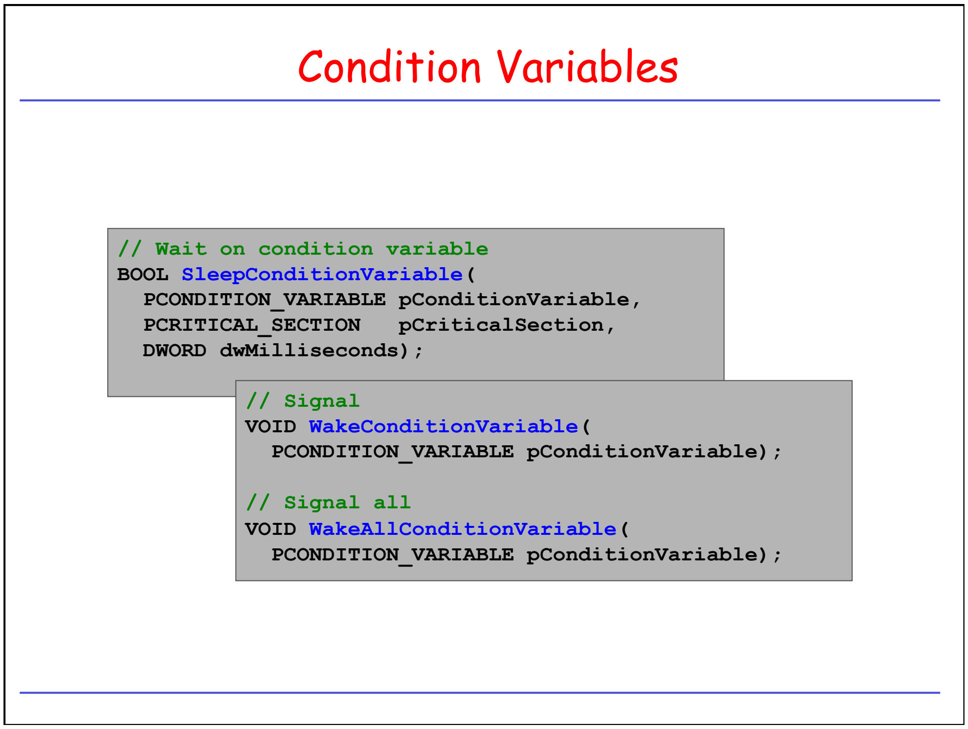
Interlocked Operations

```
// Global variable
long g_x = 0;

DWORD WINAPI ThreadFunc1(PVOID pvParam) {
  g_x++;
  return(0);
}

DWORD WINAPI ThreadFunc1(PVOID pvParam) {
  g_x++;
  return(0);
}
```

```
DWORD WINAPI ThreadFunc1(PVOID pvParam) {
  InterlockedExchangeAdd(&g_x, 1);
  return(0);
}

DWORD WINAPI ThreadFunc1(PVOID pvParam) {
  InterlockedExchangeAdd(&g_x, 1);
  return(0);
}
```

Critical Sections

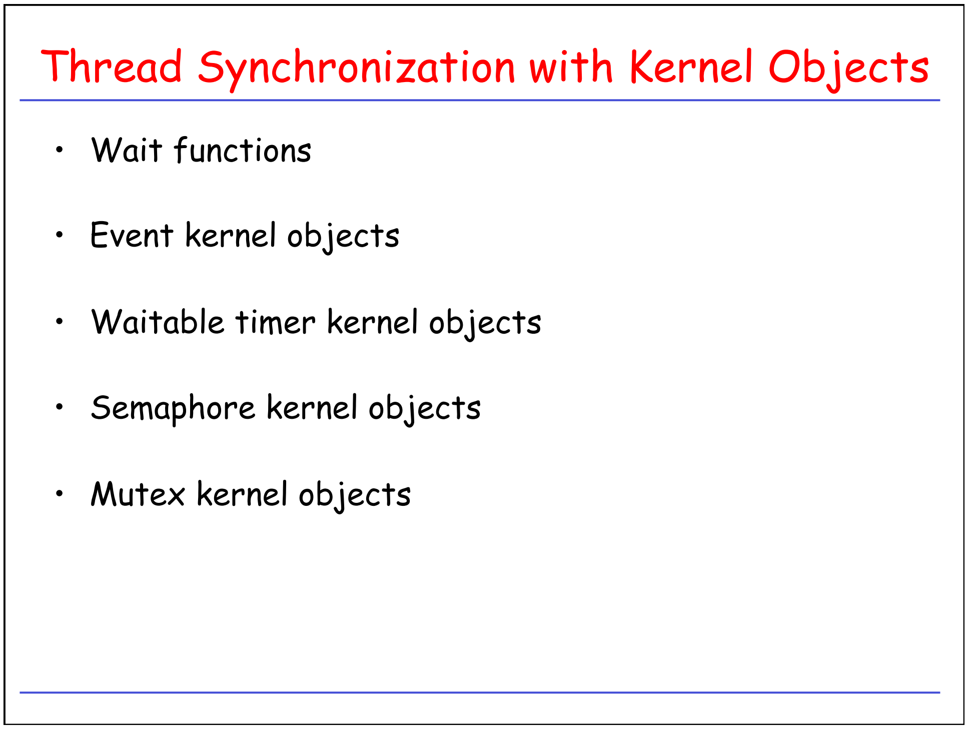
```
// EXAMPLE

int g_nSum = 0;
CRITICAL_SECTION g_cs;

DWORD WINAPI FirstThread(PVOID pvParam) {
  EnterCriticalSection(&g_cs);
  g_nSum = 0;
  for(int n = 1; n <= 10; n++) {
    g_nSum += n;
  }
  LeaveCriticalSection(&g_cs);
  return(g_nSum);
}
```



Condition Variables

```
// Wait on condition variable
BOOL SleepConditionVariable(
  PCONDITION_VARIABLE pConditionVariable,
  PCRITICAL_SECTION   pCriticalSection,
  DWORD dwMilliseconds);
```

```
// Signal
VOID WakeConditionVariable(
  PCONDITION_VARIABLE pConditionVariable);

// Signal all
VOID WakeAllConditionVariable(
  PCONDITION_VARIABLE pConditionVariable);
```



Thread Synchronization with Kernel Objects

- Wait functions
- Event kernel objects
- Waitable timer kernel objects
- Semaphore kernel objects
- Mutex kernel objects

Thread Synchronization with Kernel Object

- Most kernel objects (events, waitable timer, threads, jobs, processes, semaphores, mutexes) can be in **signaled** or **non-signaled** mode.

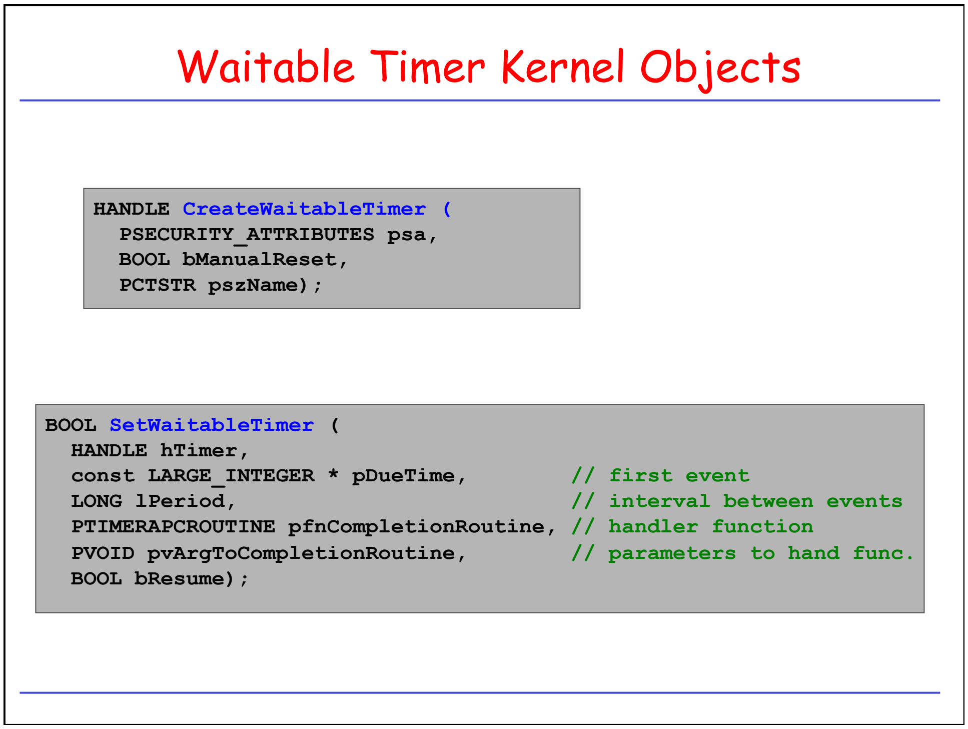
```
// Calling thread waits until object becomes signaled.
DWORD WaitForSingleObject(
  HANDLE hObject,        // kernel object that is sig/non-sig
  DWORD dwMilliseconds   // time-out
);
```

Event Kernel Objects

```
HANDLE CreateEvent (
  PSECURITY_ATTRIBUTES psa,
  BOOL bManualReset,
  BOOL bInitialState,
  PCTSTR pszName);
```

```
// Change event to signaled state
BOOL SetEvent(HANDLE hEvent);

// Change event to non-signaled state
BOOL ResetEvent(HANDLE hEvent);
```

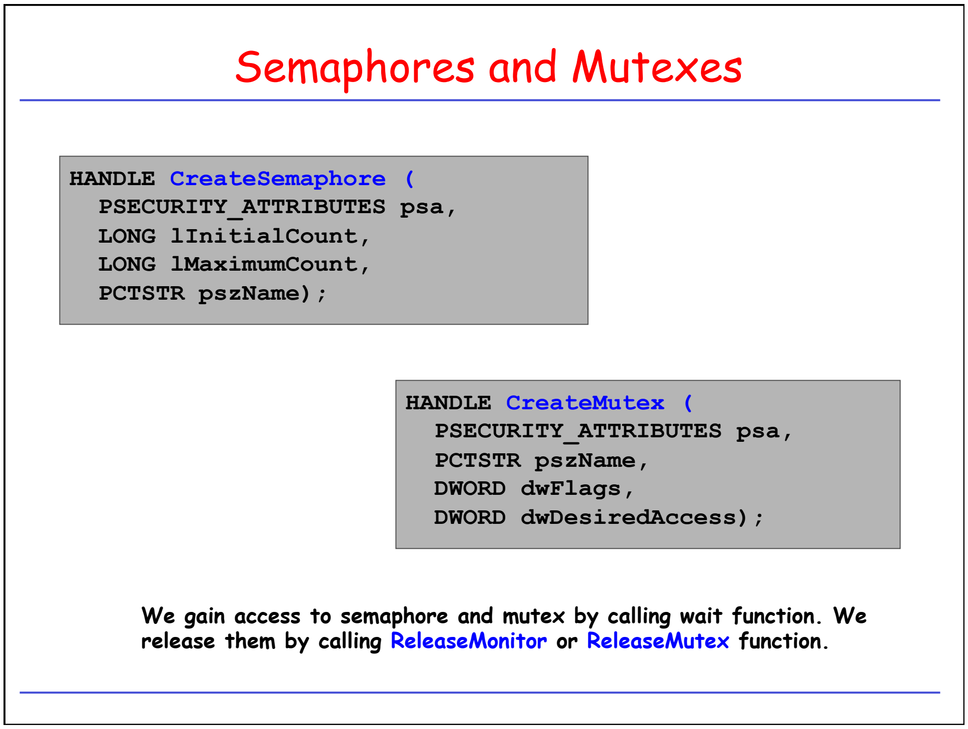
Recall: We wait with WaitForSingleEvent(…).



Waitable Timer Kernel Objects

```
HANDLE CreateWaitableTimer (
  PSECURITY_ATTRIBUTES psa,
  BOOL bManualReset,
  PCTSTR pszName);
```

```
BOOL SetWaitableTimer (
  HANDLE hTimer,
  const LARGE_INTEGER * pDueTime,         // first event
  LONG lPeriod,                           // interval between events
  PTIMERAPCROUTINE pfnCompletionRoutine,  // handler function
  PVOID pvArgToCompletionRoutine,         // parameters to hand func.
  BOOL bResume);
```

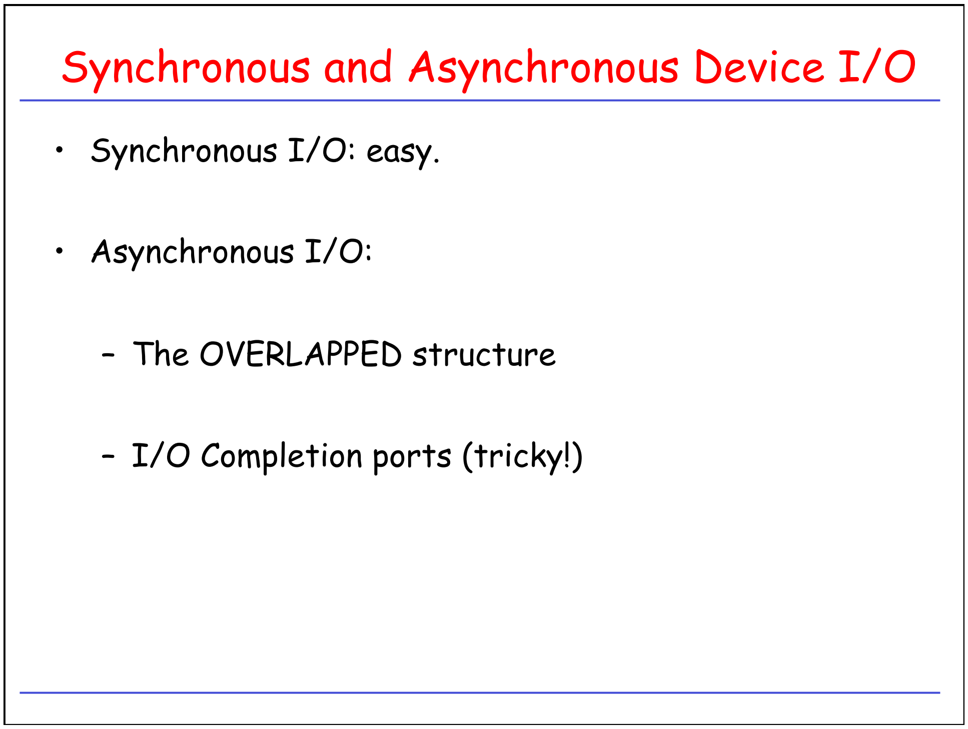


Semaphores and Mutexes

```
HANDLE CreateSemaphore (
  PSECURITY_ATTRIBUTES psa,
  LONG lInitialCount,
  LONG lMaximumCount,
  PCTSTR pszName);
```

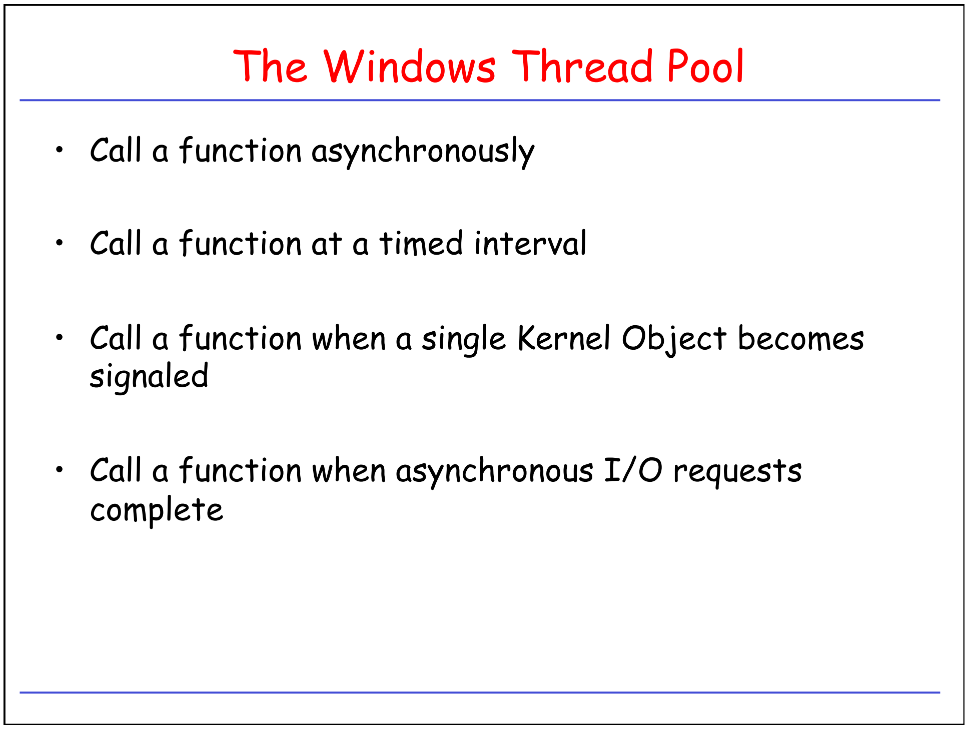
```
HANDLE CreateMutex (
  PSECURITY_ATTRIBUTES psa,
  PCTSTR pszName,
  DWORD dwFlags,
  DWORD dwDesiredAccess);
```

We gain access to semaphore and mutex by calling wait function. We release them by calling ReleaseMonitor or ReleaseMutex function.



Synchronous and Asynchronous Device I/O

- Synchronous I/O: easy.
- Asynchronous I/O:
  - The OVERLAPPED structure
  - I/O Completion ports (tricky!)



The Windows Thread Pool

- Call a function asynchronously
- Call a function at a timed interval
- Call a function when a single Kernel Object becomes signaled
- Call a function when asynchronous I/O requests complete

Other Topics…

- Fibers
- Virtual Memory
- Memory-Mapped Files
- Dynamically Linked Libraries
- … and that's about it!

THANK YOU!