

Sistemas Operativos

Comunicación y Sincronización - IV



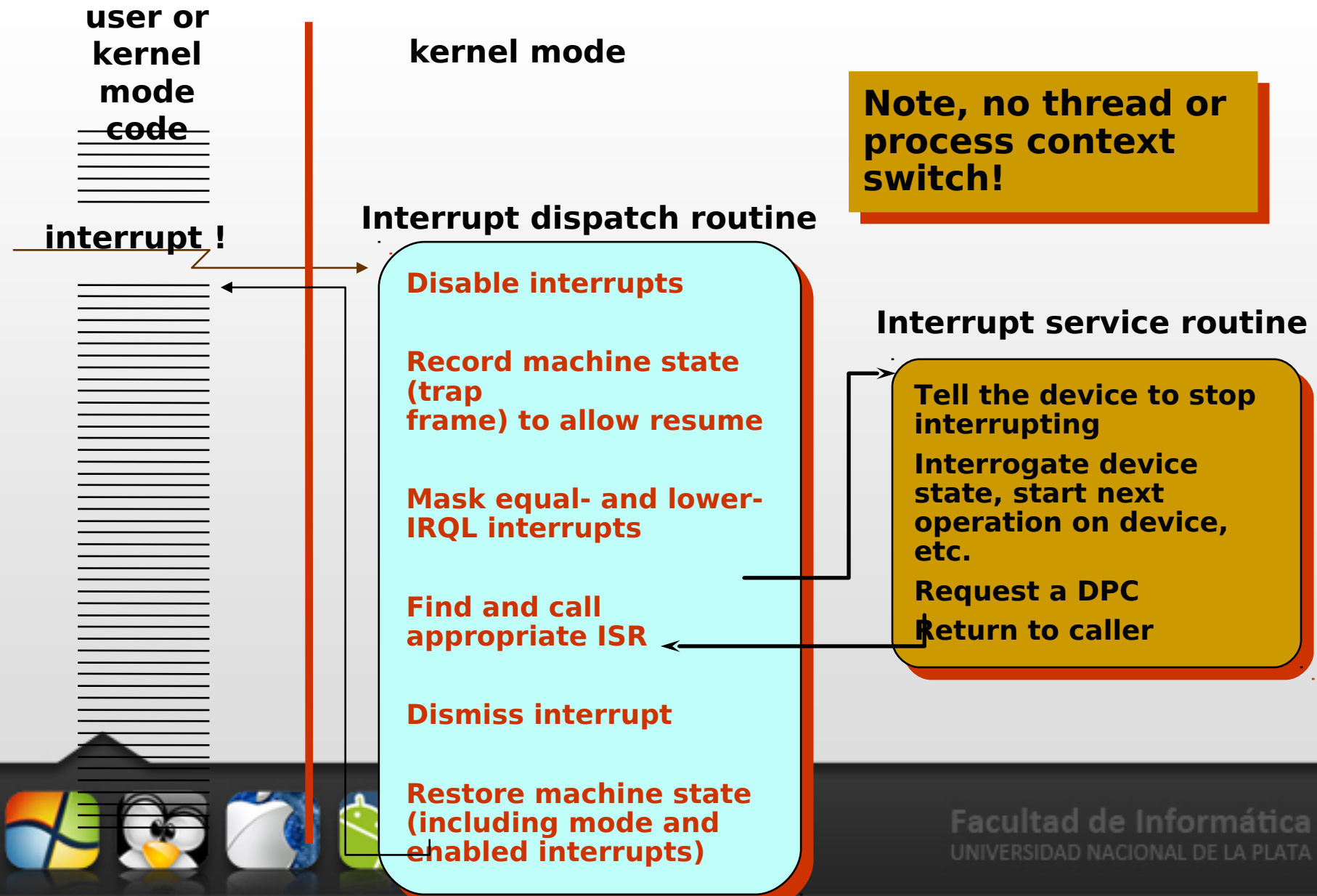
Sistemas Operativos

- ✓ Versión: Abril 2017
- ✓ Palabras Claves: Proceso, Comunicación, IPC, Windows, Sincronización, LPC, Memoria Compartida

Algunas diapositivas han sido extraídas de las ofrecidas para docentes desde el libro de Stallings (Sistemas Operativos) , el de Silberschatz (Operating Systems Concepts)



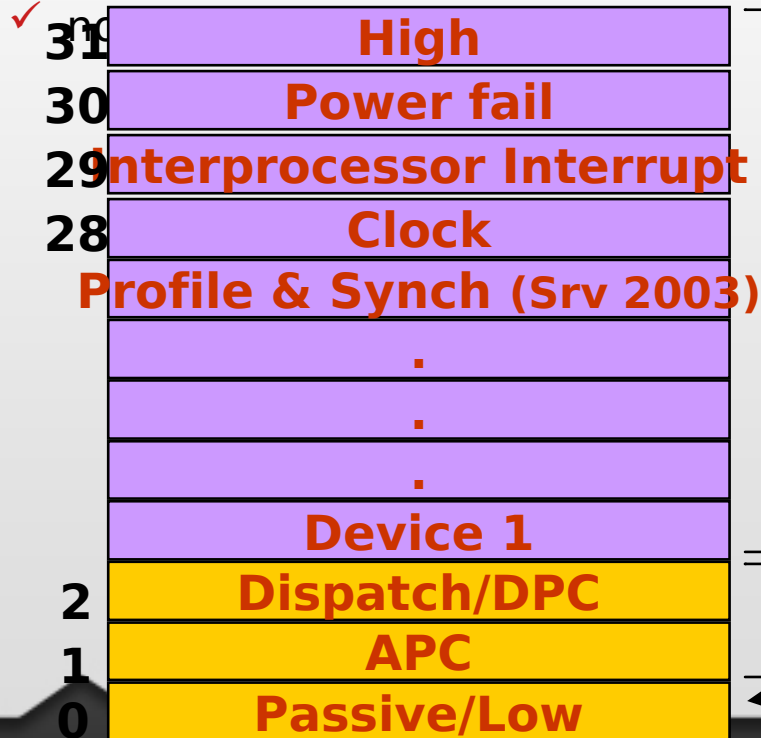
Interrupt Dispatching



Interrupt Precedence via IRQLs

✓ **IRQL = Interrupt Request Level**

- ✓ the “precedence” of the interrupt with respect to other interrupts
- ✓ Different interrupt sources have different IRQLs



- ✓ IRQL is also a state of the processor
- ✓ Servicing an interrupt raises processor IRQL to that interrupt's IRQL
 - ✓ this masks subsequent interrupts at equal and lower IRQLs
- ✓ User mode is limited to IRQL 0
- ✓ No waits or page faults at IRQL \geq DISPATCH_LEVEL

Hardware interrupts

Deferrable software interrupts

normal thread execution



Predefined IRQs

✓ **High**

- ✓ used when halting the system (via *KeBugCheck()*)

✓ **Power fail**

- ✓ originated in the NT design document, but has never been used

✓ **Inter-processor interrupt**

- ✓ used to request action from other processor (dispatching a thread, updating a processors TLB, system shutdown, system crash)

✓ **Clock**

- ✓ Used to update system's clock, allocation of CPU time to threads

✓ **Profile**

- ✓ Used for kernel profiling (see Kernel profiler – Kernprof.exe, Res Kit)



Predefined IRQs (contd.)

☑ **Device**

- ✓ Used to prioritize device interrupts

☑ **DPC/dispatch** and **APC**

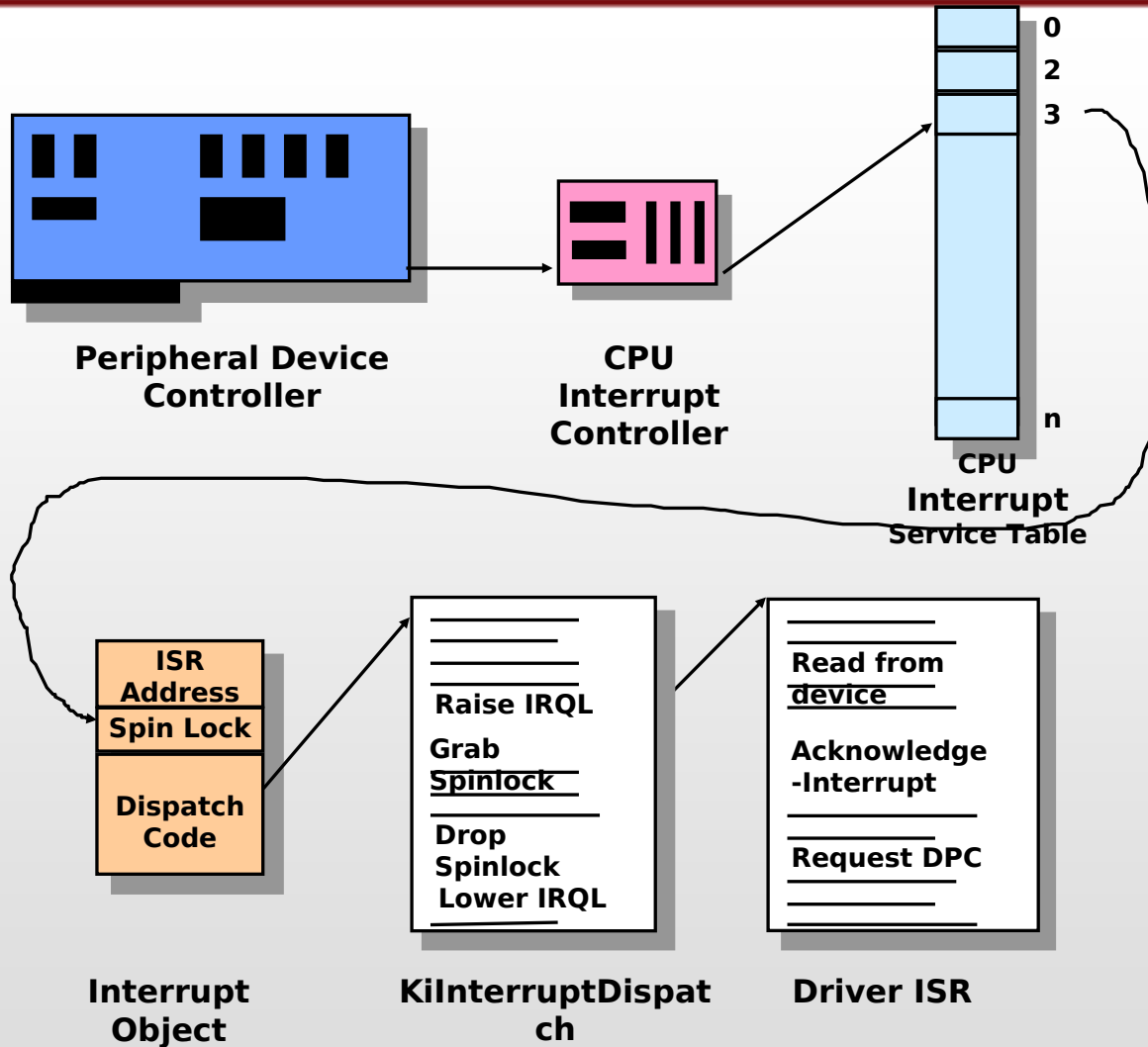
- ✓ Software interrupts that kernel and device drivers generate

☑ **Passive**

- ✓ No interrupt level at all, normal thread execution



Flow of Interrupts



Synchronization on SMP Systems

- ✓ Synchronization on MP systems use spinlocks to coordinate among the processors
- ✓ Spinlock acquisition and release routines implement a one-owner-at-a-time algorithm
 - ✓ A spinlock is either free, or is considered to be owned by a CPU
 - ✓ Analogous to using Windows API mutexes from user mode
- ✓ A spinlock is just a data cell in memory
 - ✓ Accessed with a test-and-modify operation that is atomic across all processors



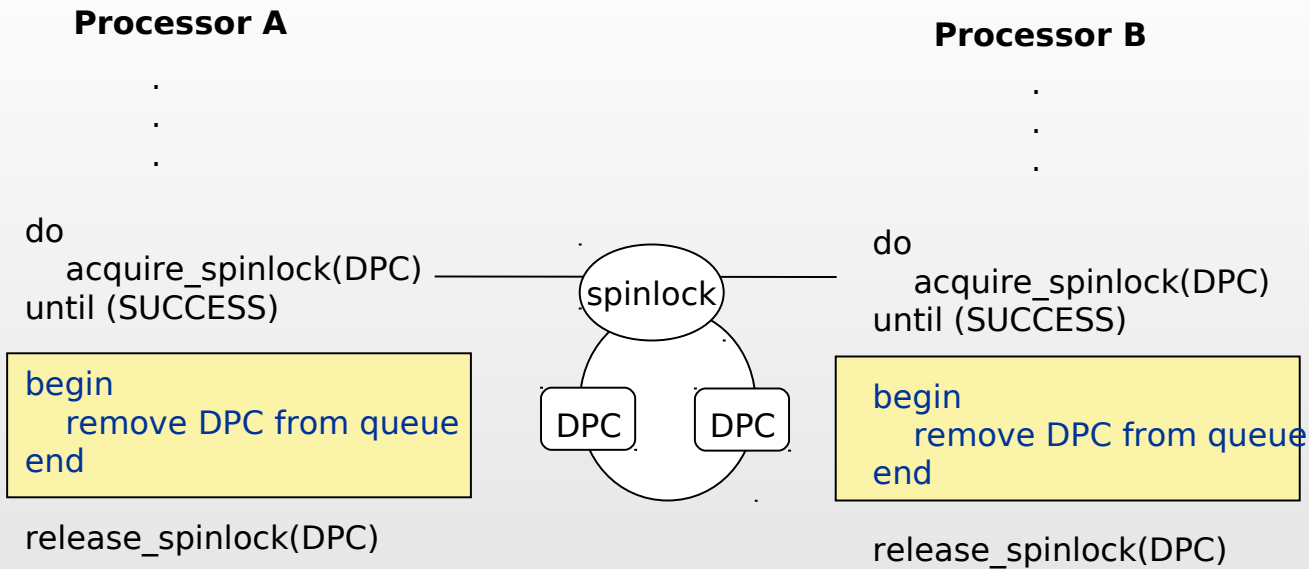
31

0

- ✓ KSPIN_LOCK is an opaque data type, typedef'd as a ULONG
- ✓ To implement synchronization, a single bit is sufficient



Kernel Synchronization



 Critical section

A spinlock is a locking primitive associated with a global data structure, such as the DPC queue



Queued Spinlocks

- ✓ **Problem:** Checking status of spinlock via test-and-set operation creates bus contention
- ✓ Queued spinlocks maintain queue of waiting processors
- ✓ First processor acquires lock; other processors wait on processor-local flag
 - ✓ Thus, busy-wait loop requires no access to the memory bus
- ✓ When releasing lock, the first processor's flag is modified
 - ✓ Exactly one processor is being signaled
 - ✓ Pre-determined wait order



Synchronizing Threads with Kernel Objects

```
DWORD WaitForSingleObject( HANDLE hObject, DWORD dwTimeout );
```

```
DWORD WaitForMultipleObjects( DWORD cObjects,  
    LPHANDLE lpHandles, BOOL bWaitAll,  
    DWORD dwTimeout );
```

The following kernel objects can be used to synchronize threads:

- ✓ Processes
- ✓ Threads
- ✓ Files
- ✓ Console input
- ✓ File change notifications
- ✓ Mutexes
- ✓ Events (auto-reset + manual-reset)
- ✓ Waitable timers



Wait Functions - Details

☑ WaitForSingleObject():

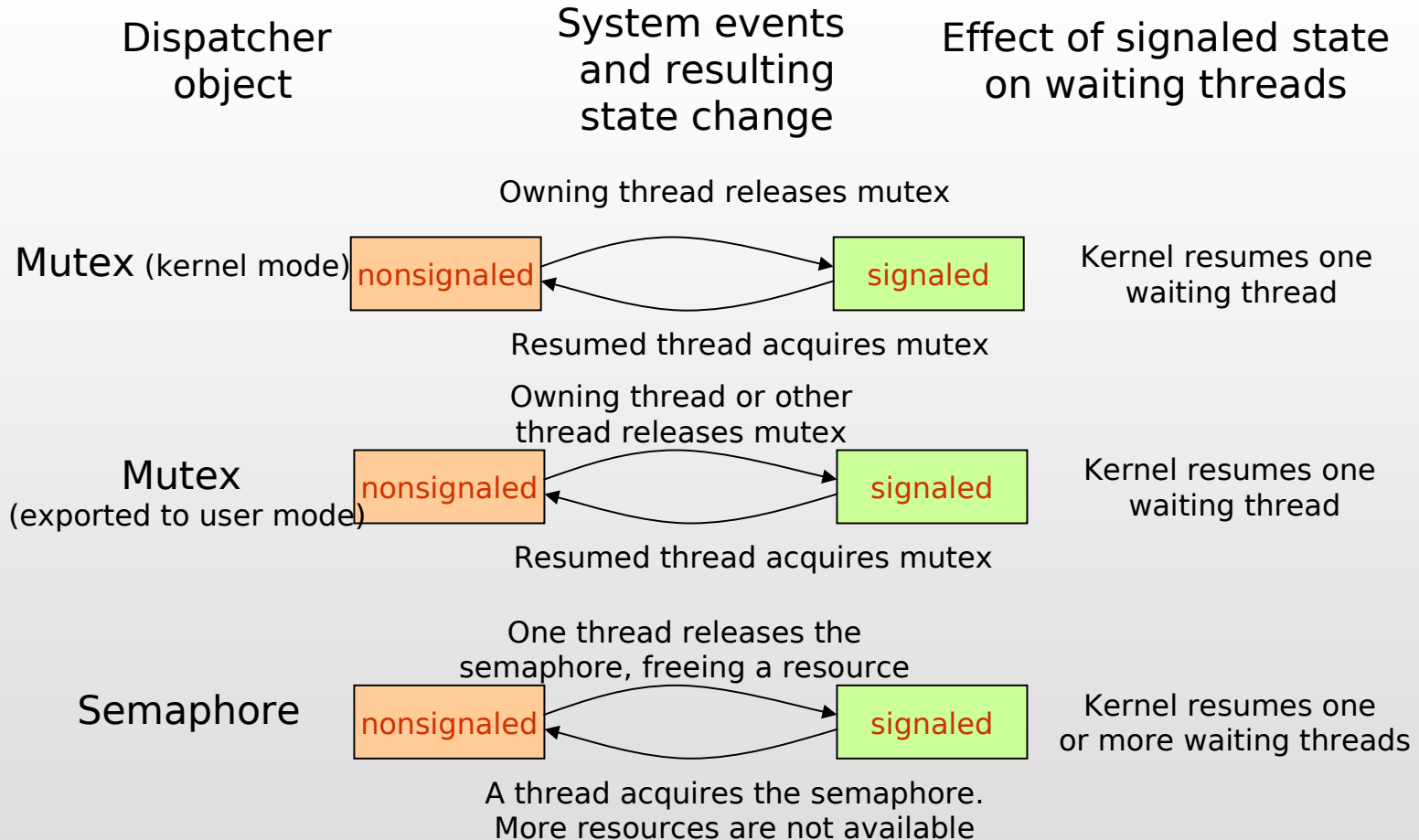
- ✓ hObject specifies kernel object
- ✓ dwTimeout specifies wait time in msec
 - ♦ dwTimeout == 0 - no wait, check whether object is signaled
 - ♦ dwTimeout == INFINITE - wait forever

☑ WaitForMultipleObjects():

- ✓ cObjects <= MAXIMUM_WAIT_OBJECTS (64)
- ✓ lpHandles - pointer to array identifying these objects
- ✓ bWaitAll - whether to wait for first signaled object or all objects
 - ♦ Function returns index of first signaled object





What signals an object?




What signals an object? (contd.)

Dispatcher object	System events and resulting state change	Effect of signaled state on waiting threads
-------------------	--	---

Event	<p>A thread sets the event</p>  <p>Kernel resumes one or more threads</p>	Kernel resumes one or more waiting threads
-------	---	--

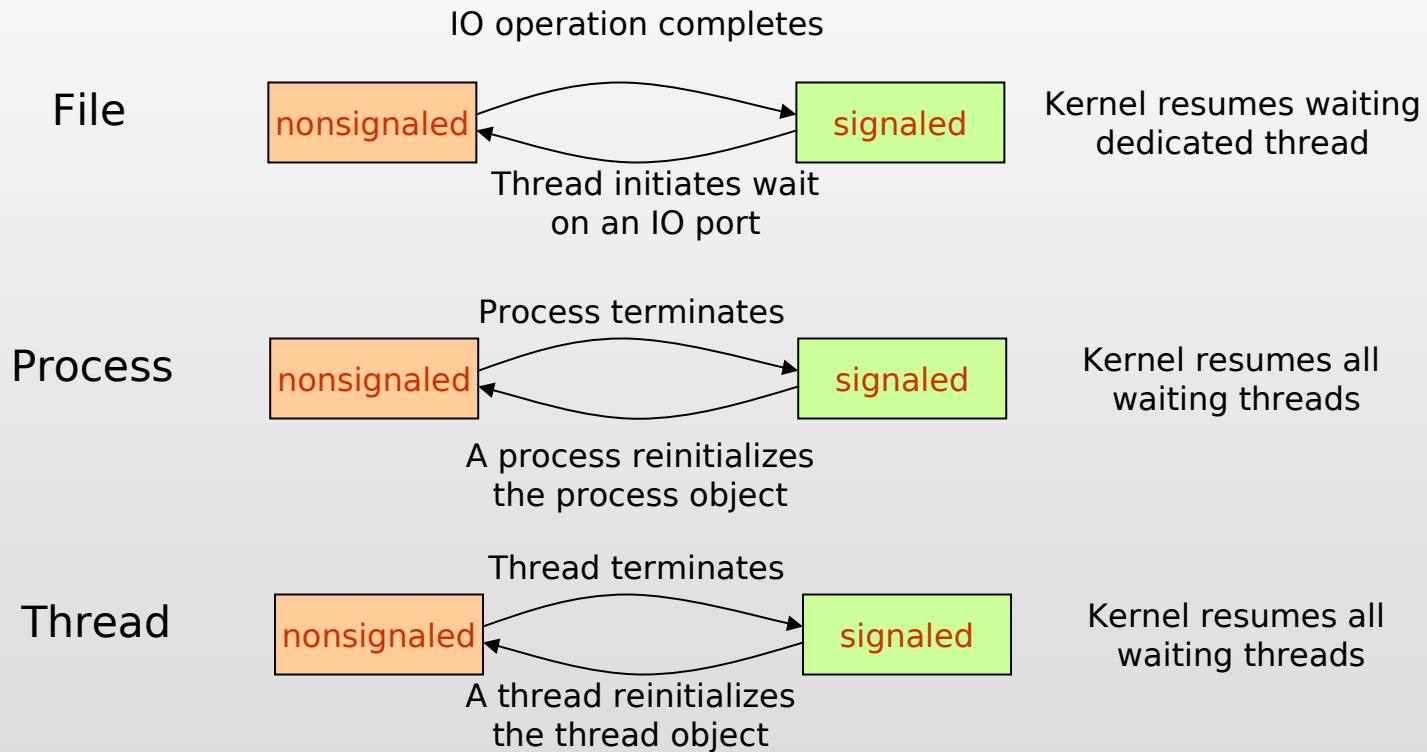
Event pair	<p>Dedicated thread sets one event in the event pair</p>  <p>Kernel resumes the other dedicated thread</p>	Kernel resumes waiting dedicated thread
------------	--	---

Timer	<p>Timer expires</p>  <p>A thread (re) initializes the timer</p>	Kernel resumes all waiting threads
-------	--	------------------------------------



What signals an object? (contd.)

Dispatcher object	System events and resulting state change	Effect of signaled state on waiting threads
-------------------	--	---



Mutexes

```
HANDLE CreateMutex( LPSECURITY_ATTRIBUTES lpsa,  
                   BOOL fInitialOwner, LPTSTR lpszMutexName );  
  
HANDLE OpenMutex( LPSECURITY_ATTRIBUTES lpsa,  
                 BOOL fInitialOwner, LPTSTR lpszMutexName );  
  
BOOL ReleaseMutex( HANDLE hMutex );
```

Mutexes work across processes

- ✓ First thread has to call CreateMutex()
- ✓ When sharing a mutex, second thread (process) calls CreateMutex() or OpenMutex()
- ✓ fInitialOwner == TRUE gives creator immediate ownership
- ✓ Threads acquire mutex ownership using WaitForSingleObject() or WaitForMultipleObjects()
- ✓ ReleaseMutex() gives up ownership
- ✓ CloseHandle() will free mutex object



Mutex Example

```
/* counter is global, shared by all threads */
volatile int done, counter = 0;
HANDLE mutex = CreateMutex( NULL, FALSE, NULL );

/* main loop in any of the threads, ret is local */
DWORD ret;
while (!done) {
    ret = WaitForSingleObject( mutex, INFINITE );
    if (ret == WAIT_OBJECT_0)
        counter += local_value;
    else /* mutex was abandoned */
        break; /* exit the loop */
    ReleaseMutex( mutex );
}
CloseHandle( mutex );
```



Comparison - POSIX mutexes

- ☑ POSIX pthreads specification supports mutexes
 - ✓ Synchronization among threads in same process
- ☑ Five basic functions:
 - ✓ pthread_mutex_init()
 - ✓ pthread_mutex_destroy()
 - ✓ pthread_mutex_lock()
 - ✓ pthread_mutex_unlock()
 - ✓ pthread_mutex_trylock()
- ☑ Comparison:
 - ✓ pthread_mutex_lock() will block - equivalent to WaitForSingleObject(hMutex);
 - ✓ pthread_mutex_trylock() is nonblocking (polling) - equivalent to WaitForSingleObject() with timeout == 0



Semaphores

```
HANDLE CreateSemaphore( LPSECURITY_ATTRIBUTE lpsa,  
                        LONG cSemInit, LONG cSemMax,  
                        LPTSTR lpszSemName );
```

```
HANDLE OpenSemaphore( LPSECURITY_ATTRIBUTE lpsa,  
                     LONG cSemInit, LONG cSemMax,  
                     LPTSTR lpszSemName );
```

```
HANDLE ReleaseSemaphore( HANDLE hSemaphore,  
                        LONG cReleaseCount, LPLONG lpPreviousCount );
```

- ☑ Semaphore objects are used for resource counting
 - ✓ A semaphore is signaled when count > 0
- ☑ Threads/processes use wait functions
 - ✓ Each wait function decreases semaphore count by 1
 - ✓ ReleaseSemaphore() may increment count by any value
 - ✓ ReleaseSemaphore() returns old semaphore count



Critical Sections

```
VOID InitializeCriticalSection( LPCRITICAL_SECTION sec );  
VOID DeleteCriticalSection( LPCRITICAL_SECTION sec );  
  
VOID EnterCriticalSection( LPCRITICAL_SECTION sec ); VOID  
LeaveCriticalSection( LPCRITICAL_SECTION sec );  
BOOL TryEnterCriticalSection ( LPCRITICAL_SECTION sec );
```

Only usable from within the same process

- ✓ Critical sections are initialized and deleted but do not have handles
- ✓ Only one thread at a time can be in a critical section
- ✓ A thread can enter a critical section multiple times - however, the number of Enter- and Leave-operations must match
- ✓ Leaving a critical section before entering it may cause deadlocks
- ✓ No way to test whether another thread is in a critical section



Critical Section Example

```
/* counter is global, shared by all threads */
volatile int counter = 0;
CRITICAL_SECTION crit;
InitializeCriticalSection ( &crit );

/* ... main loop in any of the threads */
while (!done) {
    _try {
        EnterCriticalSection ( &crit );
        counter += local_value;
        LeaveCriticalSection ( &crit );
    }
    _finally { LeaveCriticalSection ( &crit ); }
}
DeleteCriticalSection( &crit );
```

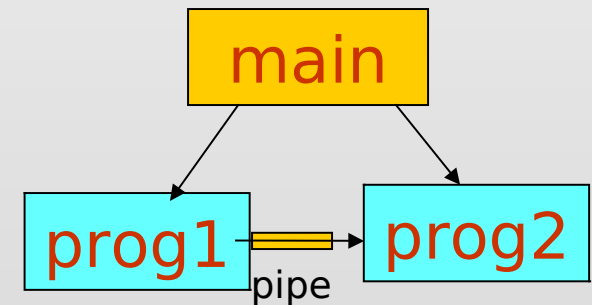


Anonymous pipes

```
BOOL CreatePipe( PHANDLE phRead,  
                PHANDLE phWrite,  
                LPSECURITY_ATTRIBUTES lpsa,  
                DWORD cbPipe )
```

Half-duplex character-based IPC

- ✓ cbPipe: pipe byte size; zero == default
- ✓ Read on pipe handle will block if pipe is empty
- ✓ Write operation to a full pipe will block
- ✓ Anonymous pipes are oneway



Named Pipes

- ☑ Message oriented:
 - ✓ Reading process can read varying-length messages precisely as sent by the writing process
- ☑ Bi-directional
 - ✓ Two processes can exchange messages over the same pipe
- ☑ Multiple, independent instances of a named pipe:
 - ✓ Several clients can communicate with a single server using the same instance
 - ✓ Server can respond to client using the same instance
- ☑ Pipe can be accessed over the network
 - ✓ location transparency
- ☑ Convenience and connection functions



Using Named Pipes

```
HANDLE CreateNamedPipe (LPCTSTR lpszPipeName,  
    DWORD fdwOpenMode, DWORD fdwPipMode  
    DWORD nMaxInstances, DWORD cbOutBuf,  
    DWORD cbInBuf, DWORD dwTimeOut,  
    LPSECURITY_ATTRIBUTES lpsa );
```

- ☑ lpszPipeName:
 - ✓ Not possible to create a pipe on remote machine (. - local machine)
- ☑ fdwOpenMode:
 - ✓ PIPE_ACCESS_DUPLEX, PIPE_ACCESS_INBOUND, PIPE_ACCESS_OUTBOUND
- ☑ fdwPipeMode:
 - ✓ PIPE_TYPE_BYTE or PIPE_TYPE_MESSAGE
 - ✓ PIPE_READMODE_BYTE or PIPE_READMODE_MESSAGE
 - ✓ PIPE_WAIT or PIPE_NOWAIT (will ReadFile block?)

Use same flag settings for all instances of a named pipe



Named Pipes (contd.)

- ☑ nMaxInstances:
 - ✓ Number of instances,
 - ✓ PIPE_UNLIMITED_INSTANCES: OS choice based on resources
- ☑ dwTimeOut
 - ✓ Default time-out period (in msec) for WaitNamedPipe()
- ☑ First CreateNamedPipe creates named pipe
 - ✓ Closing handle to last instance deletes named pipe
- ☑ Polling a pipe:
 - ✓ Nondestructive – is there a message waiting for ReadFile

```
BOOL PeekNamedPipe (HANDLE hPipe,  
                    LPVOID lpvBuffer, DWORD cbBuffer,  
                    LPDWORD lpcbRead, LPDWORD lpcbAvail,  
                    LPDWORD lpcbMessage);
```



Comparison with UNIX

- ☑ UNIX FIFOs are similar to a named pipe
 - ✓ FIFOs are half-duplex
 - ✓ FIFOs are limited to a single machine
 - ✓ FIFOs are still byte-oriented, so its easiest to use fixed-size records in client/server applications
 - ✓ Individual read/writes are atomic
- ☑ A server using FIFOs must use a separate FIFO for each client's response, although all clients can send requests via a single, well known FIFO
- ☑ Mkfifo() is the UNIX counterpart to CreateNamedPipe()
- ☑ Use sockets for networked client/server scenarios



Windows IPC - Mailslots

- ☑ Broadcast mechanism:
 - ✓ One-directional
 - ✓ Multiple writers/multiple readers (frequently: one-to-many comm.)
 - ✓ Message delivery is unreliable
 - ✓ Can be located over a network domain
 - ✓ Message lengths are limited (w2k: < 426 byte)
- ☑ Operations on the mailslot:
 - ✓ Each reader (server) creates mailslot with CreateMailslot()
 - ✓ Write-only client opens mailslot with CreateFile() and uses WriteFile() - open will fail if there are no waiting readers
 - ✓ Client's message can be read by all servers (readers)
- ☑ Client lookup: *\mailslot\mailslotname
 - ✓ Client will connect to every server in network domain



Locate a server via mailslot

Mailslot Servers

App client 0

```
hMS = CreateMailslot(  
    "\\.\mailslot\status");  
ReadFile(hMS, &ServStat);  
/* connect to server */
```

App client n

```
hMS = CreateMailslot(  
    "\\.\mailslot\status");  
ReadFile(hMS, &ServStat);  
/* connect to server */
```

Message is
sent periodically



Mailslot Client

App Server

```
While (...) {  
    Sleep(...);  
    hMS = CreateFile(  
        "\\.\mailslot\status");  
    ...  
    WriteFile(hMS, &StatInfo  
    }
```



Creating a mailslot

```
HANDLE CreateMailslot(LPCTSTR lpszName,  
    DWORD cbMaxMsg,  
    DWORD dwReadTimeout,  
    LPSECURITY_ATTRIBUTES lpsa);
```

- ☑ `lpszName` points to a name of the form
 - ✓ `\\.\mailslot\[path]name`
 - ✓ Name must be unique; mailslot is created locally
- ☑ `cbMaxMsg` is msg size in byte
- ☑ `dwReadTimeout`
 - ✓ Read operation will wait for so many msec
 - ✓ 0 – immediate return
 - ✓ `MAILSLOT_WAIT_FOREVER` – infinite wait

