Concurrency (II)
--- Synchronization

Road Map For This Lecture

- Synchronization in Windows & Linux
- High-IRQL Synchronization (spin locks)
- Low-IRQL Synchronization (dispatcher objects)
- Windows APIs for synchronization

Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems (by raising or lowering IRQLs).
- Uses *spinlocks* on multiprocessor systems.
- Provides *dispatcher objects* which may act as mutexes and semaphores.
- Dispatcher objects may also provide *events*. An event acts much like a condition variable.

Linux Synchronization

- Kernel *disables interrupts* for synchronizing access to global data on uniprocessor systems.
- Uses *spinlocks* for multiprocessor synchronization.
- Uses *semaphores* and *readers-writers* locks when longer sections of code need access to data.
- Implements POSIX synchronization primitives to support multitasking, multithreading (including real-time threads), and multiprocessing.

High-IRQL Synchronization

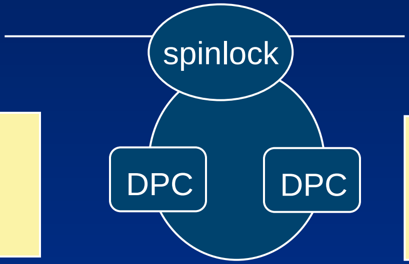
- Synchronization on MP systems use spinlocks to coordinate among the processors
- Spinlock acquisition and release routines implement a one-owner-at-a-time algorithm
  - A spinlock is either free, or is considered to be owned by a CPU
  - Analogous to using Windows API mutexes from user mode
- A spinlock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

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

Using a spinlock

Processor A

```
.
.
.

do
    acquire_spinlock(DPC)
until (SUCCESS)

begin
    remove DPC from queue
end

release_spinlock(DPC)
```



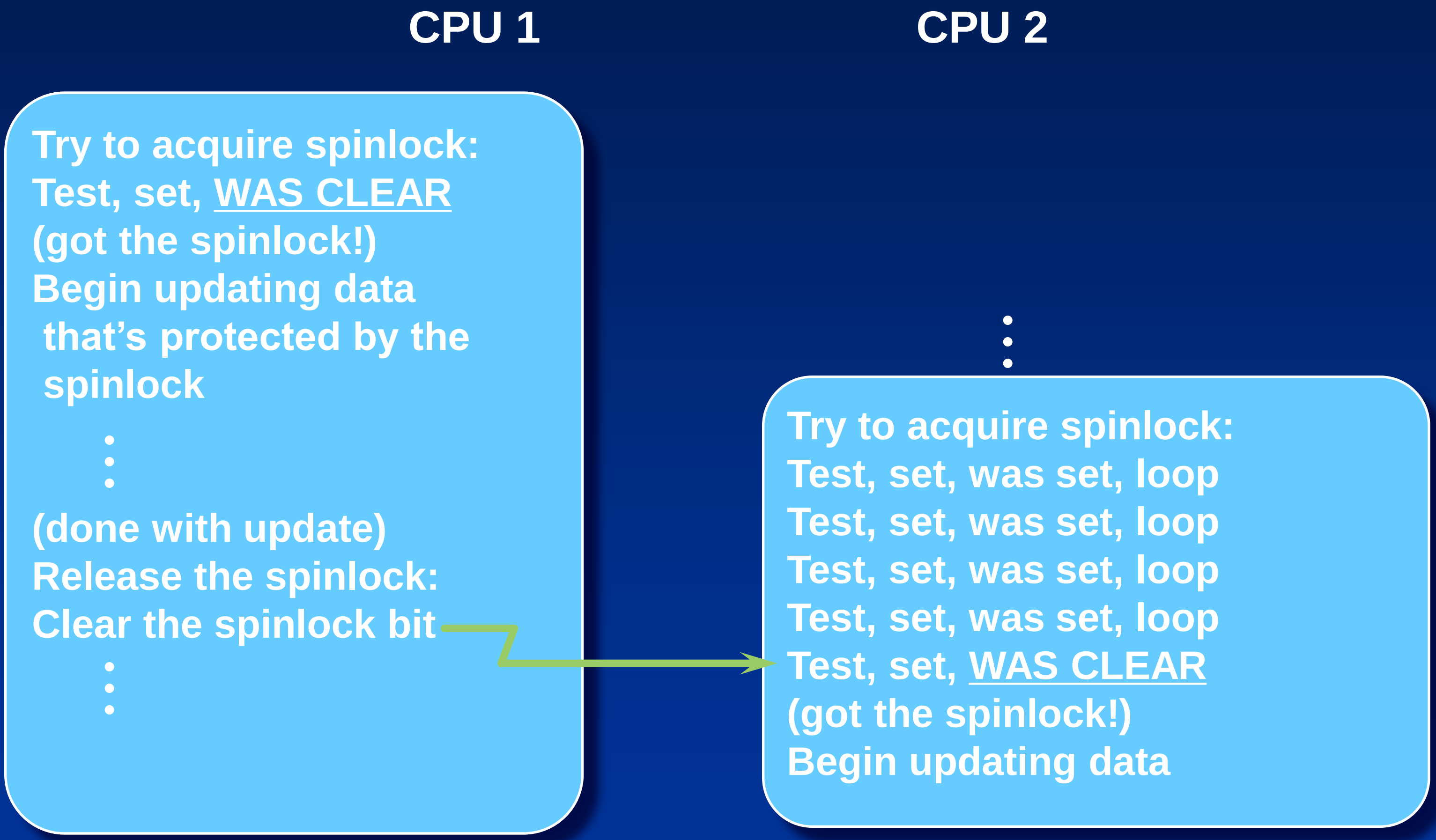
Processor B

```
.
.
.

do
    acquire_spinlock(DPC)
until (SUCCESS)

begin
    remove DPC from queue
end

release_spinlock(DPC)
```

🟨 Critical section

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

Spinlocks in Action



CPU 1

Try to acquire spinlock:
Test, set, WAS CLEAR
(got the spinlock!)
Begin updating data that's protected by the spinlock

⋮

(done with update)
Release the spinlock:
Clear the spinlock bit

⋮

CPU 2

⋮

Try to acquire spinlock:
Test, set, was set, loop
Test, set, was set, loop
Test, set, was set, loop
Test, set, was set, loop
Test, set, WAS CLEAR
(got the spinlock!)
Begin updating data

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

SMP Scalability Improvements

- Windows 2000: queued spinlocks
  - !qlocks in Kernel Debugger
- XP/2003:
  - Minimized lock contention for hot locks (PFN or Page Frame Database) lock
  - Some locks completely eliminated
    - Charging nonpaged/paged pool quotas, allocating and mapping system page table entries, charging commitment of pages, allocating/mapping physical memory through AWE functions
  - New, more efficient locking mechanism (*pushlocks*)
    - Doesn't use spinlocks when no contention
    - Smaller size than mutex or semaphore (4 bytes on 32-bit systems)
    - Used for object manager and address windowing extensions (AWE) related locks
- Server 2003:
  - More spinlocks eliminated (context swap, system space, commit)
  - Further reduction of use of spinlocks & length they are held
  - Scheduling database now per-CPU
    - Allows thread state transitions in parallel

Low-IRQL Synchronization

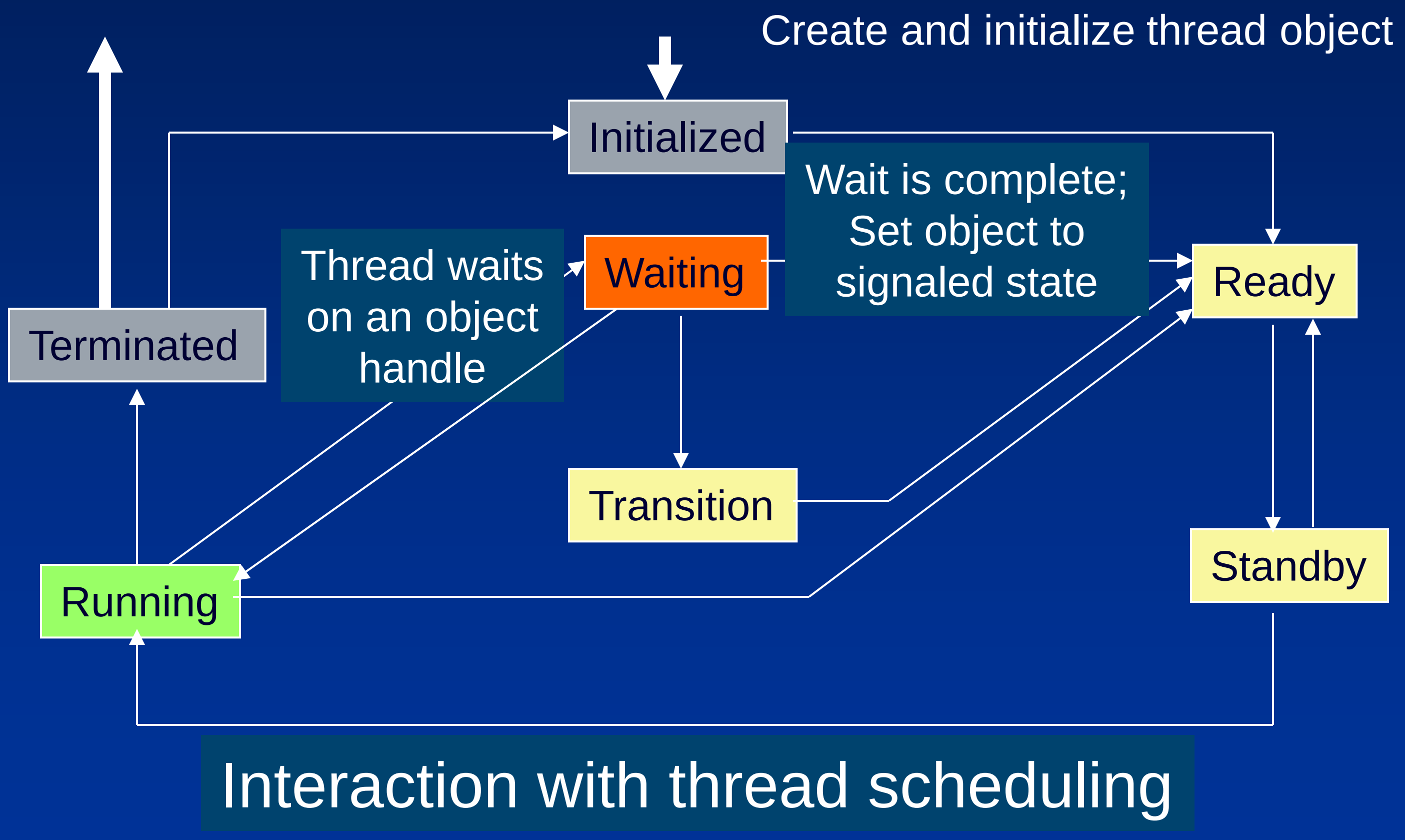
- Kernel mode:
  - **Kernel dispatcher objects**
  - Fast mutexes and guarded mutexes
  - Executive resources
  - Pushlocks
- User mode:
  - Condition variables
  - Slim read-write locks
  - Run once initialization
  - Critical sections

Waiting

- Flexible wait calls
  - Wait for one or multiple objects in one call
  - Wait for multiple can wait for “any” one or “all” at once
    - “All”: all objects must be in the signalled state concurrently to resolve the wait
  - All wait calls include optional timeout argument
  - Waiting threads consume no CPU time
- Waitable objects include:
  - Events (may be auto-reset or manual reset; may be set or “pulsed”)
  - Mutexes (“mutual exclusion”, one-at-a-time)
  - Semaphores (n-at-a-time)
  - Timers
  - Processes and Threads (signaled upon exit or terminate)
  - Directories (change notification)
- No guaranteed ordering of wait resolution
  - If multiple threads are waiting for an object, and only one thread is released (e.g. it’s a mutex or auto-reset event), which thread gets released is unpredictable
  - Typical order of wait resolution is FIFO; however APC delivery may change this order

Executive Synchronization

- Waiting on Dispatcher Objects – outside the kernel



Create and initialize thread object

Initialized

Wait is complete; Set object to signaled state

Thread waits on an object handle

Waiting

Ready

Terminated

Transition

Standby

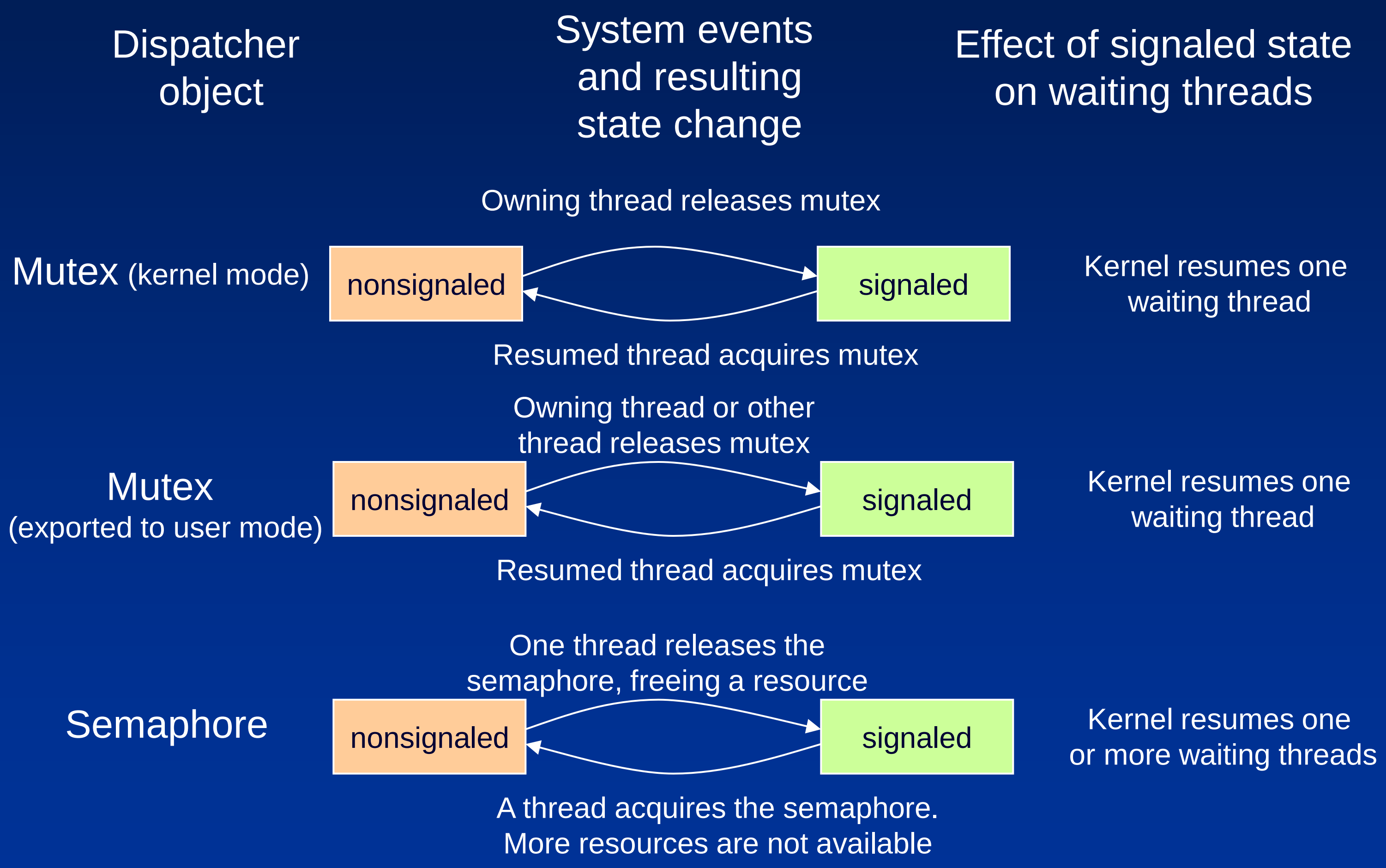
Running

Interaction with thread scheduling

Interactions between Synchronization and Thread Dispatching

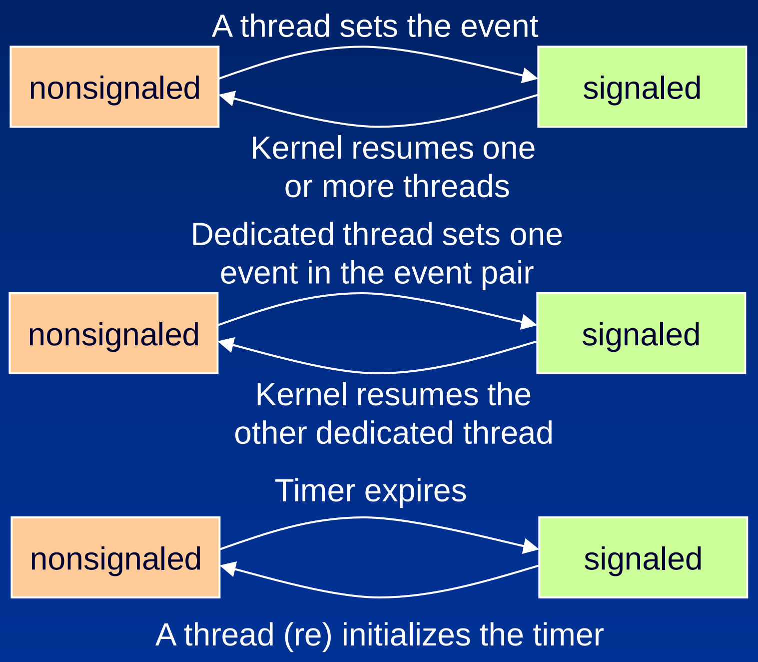
1. User mode thread waits on an event object‘s handle
2. Kernel changes thread‘s scheduling state from ready to waiting and adds thread to wait-list
3. Another thread sets the event
4. Kernel wakes up waiting threads; variable priority threads get priority boost
5. Dispatcher re-schedules new thread – it may preempt running thread if it has lower priority and issues software interrupt to initiate context switch
6. If no processor can be preempted, the dispatcher places the ready thread in the dispatcher ready queue to be scheduled later

What signals an object?



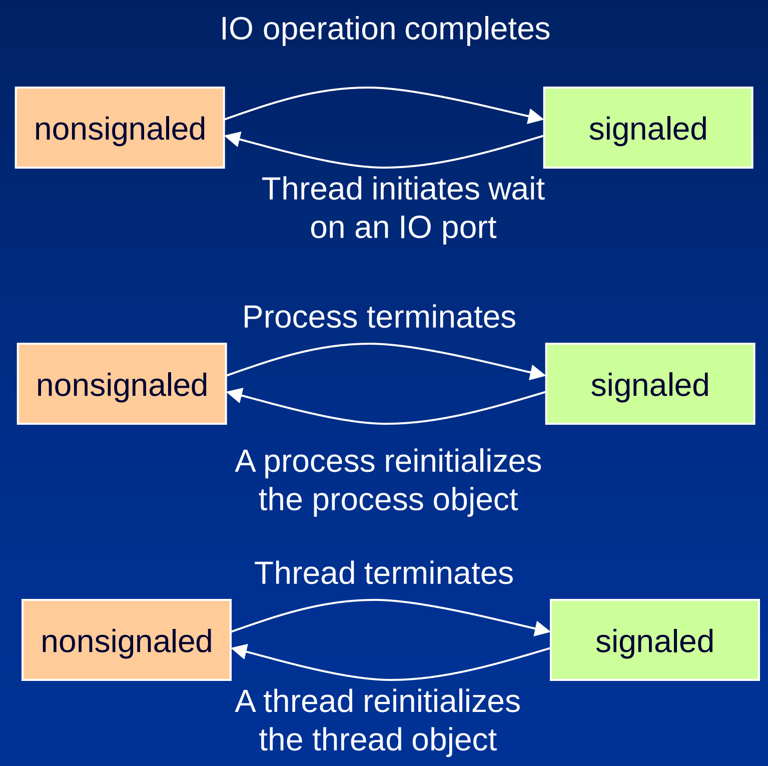
Dispatcher object	System events and resulting state change	Effect of signaled state on waiting threads
Mutex (kernel mode)	Owning thread releases mutex nonsignaled signaled Resumed thread acquires mutex	Kernel resumes one waiting thread
Mutex (exported to user mode)	Owning thread or other thread releases mutex nonsignaled signaled Resumed thread acquires mutex	Kernel resumes one waiting thread
Semaphore	One thread releases the semaphore, freeing a resource nonsignaled signaled A thread acquires the semaphore. More resources are not available	Kernel resumes one or more waiting threads

What signals an object? (contd.)



Dispatcher object	System events and resulting state change	Effect of signaled state on waiting threads
Event	A thread sets the event nonsignaled signaled Kernel resumes one or more threads	Kernel resumes one or more waiting threads
Event pair	Dedicated thread sets one event in the event pair nonsignaled signaled Kernel resumes the other dedicated thread	Kernel resumes waiting dedicated thread
Timer	Timer expires nonsignaled signaled A thread (re) initializes the timer	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

File

IO operation completes

nonsignaled

signaled

Thread initiates wait on an IO port

Kernel resumes waiting dedicated thread

Process

Process terminates

nonsignaled

signaled

A process reinitializes the process object

Kernel resumes all waiting threads

Thread

Thread terminates

nonsignaled

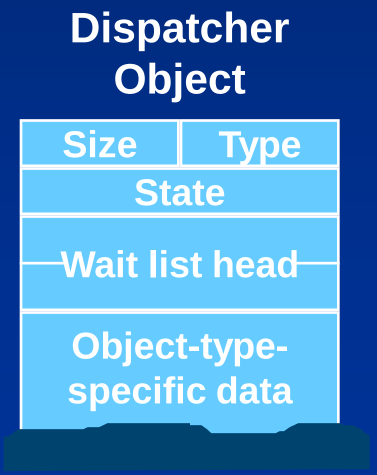
signaled

A thread reinitializes the thread object

Kernel resumes all waiting threads

Wait Internals 1:
Dispatcher Objects

- Any kernel object you can wait for is a “dispatcher object”
  - some exclusively for synchronization
    - e.g. events, mutexes (“mutants”), semaphores, queues, timers
  - others can be waited for as a side effect of their prime function
    - e.g. processes, threads, file objects
  - non-waitable kernel objects are called “control objects”
- All dispatcher objects have a common header
- All dispatcher objects are in one of two states
  - “signaled” vs. “nonsignaled”
  - when signalled, a wait on the object is satisfied
  - different object types differ in terms of what changes their state
  - wait and unwait implementation is common to all types of dispatcher objects



Dispatcher Object

Size

Type

State

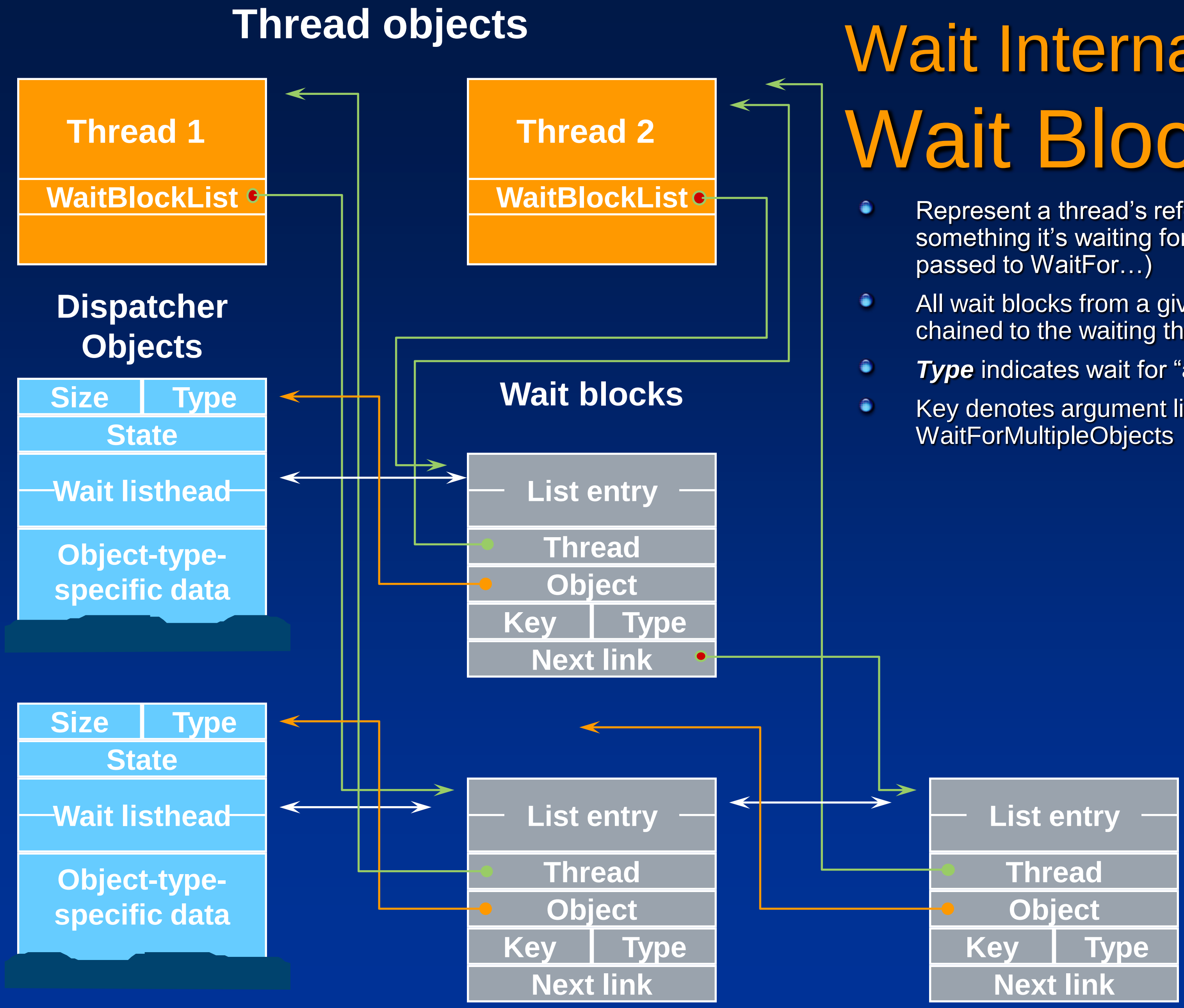
Wait list head

Object-type-specific data

(see \ntddk\inc\ddk\ntddk.h)

Wait Internals 2: Wait Blocks

- Represent a thread's reference to something it's waiting for (one per handle passed to WaitFor…)
- All wait blocks from a given wait call are chained to the waiting thread
- ***Type*** indicates wait for "any" or "all"
- Key denotes argument list position for WaitForMultipleObjects



Thread objects

Thread 1
WaitBlockList

Thread 2
WaitBlockList

Dispatcher Objects

Size | **Type**
State
Wait listhead
Object-type-specific data

Size | **Type**
State
Wait listhead
Object-type-specific data

Wait blocks

List entry
Thread
Object
Key | **Type**
Next link

List entry
Thread
Object
Key | **Type**
Next link

List entry
Thread
Object
Key | **Type**
Next link

Windows APIs for Synchronization

- Windows API constructs for synchronization and interprocess communication
- Synchronization
  - Critical sections
  - Mutexes
  - Semaphores
  - Event objects
- Synchronization through inter-process communication
  - Anonymous pipes
  - Named pipes
  - Mailslots

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 );
```

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
- Jobs
- Files
- Console input
- File change notifications
- Mutexes
- Semaphors
- 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
- Side effects:
  - Mutexes, auto-reset events and waitable timers will be reset to non-signaled state after completing wait functions

Mutexes

```
HANDLE CreateMutex( LPSECURITY_ATTRIBUTE lpsa,
            BOOL fInitialOwner, LPTSTR lpszMutexName );

HANDLE OpenMutex( LPSECURITY_ATTRIBUTE 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

Events

```
HANDLE CreateEvent( LPSECURITY_ATTRIBUTE lpsa,
            BOOL fManualReset, BOOL fInititalState
            LPTSTR lpszEventName );

BOOL SetEvent( HANDLE hEvent );
BOOL ResetEvent( HANDLE hEvent );
BOOL PulseEvent( HANDLE hEvent );
```

- Multiple threads can be released when a single event is signaled (barrier synchronization)
  - Manual-reset event can signal several thread simultaneously; must be reset manually
  - SetEvent sets the event object to be *signaled*
  - ResetEvent sets of the event object to be *unsignaled*
  - PulseEvent() will release all threads waiting on a manual-reset event and automatically reset the event
  - Auto-reset event signals a single thread; event is reset automatically
  - fInitialState == TRUE - create event in signaled state

Comparison - POSIX condition variables

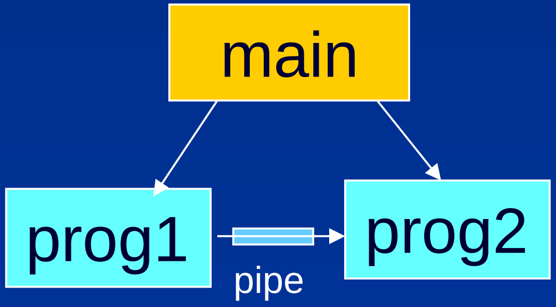
- pthread’s condition variables are comparable to events
  - pthread_cond_init()
  - pthread_cond_destroy()
- Wait functions:
  - pthread_cond_wait()
  - pthread_cond_timedwait()
- Signaling:
  - pthread_cond_signal() - one thread
  - pthread_cond_broadcast() - all waiting threads
- No exact equivalent to manual-reset events

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 one-way (half-duplex)



I/O Redirection using an Anonymous Pipe

```
/* Create default size anonymous pipe, handles are inheritable. */
if (!CreatePipe (&hReadPipe, &hWritePipe, &PipeSA, 0)) {
      fprintf(stderr, "Anon pipe create failed\n"); exit(1);
}
/* Set output handle to pipe handle, create first processes. */
StartInfoCh1.hStdInput  = GetStdHandle (STD_INPUT_HANDLE);
StartInfoCh1.hStdError  = GetStdHandle (STD_ERROR_HANDLE);
StartInfoCh1.hStdOutput = hWritePipe;
StartInfoCh1.dwFlags = STARTF_USESTDHANDLES;

if (!CreateProcess (NULL, (LPTSTR)Command1, NULL, NULL, TRUE,
              0, NULL, NULL, &StartInfoCh1, &ProcInfo1)) {
      fprintf(stderr, "CreateProc1 failed\n"); exit(2);
}
CloseHandle (hWritePipe);
```

Pipe example (contd.)

```
/* Repeat (symmetrically) for the second process. */
StartInfoCh2.hStdInput  = hReadPipe;
StartInfoCh2.hStdError  = GetStdHandle (STD_ERROR_HANDLE);
StartInfoCh2.hStdOutput = GetStdHandle (STD_OUTPUT_HANDLE);
StartInfoCh2.dwFlags = STARTF_USESTDHANDLES;

if (!CreateProcess (NULL, (LPTSTR)targv, NULL, NULL,TRUE,/* Inherit
handles. */
              0, NULL, NULL, &StartInfoCh2, &ProcInfo2)) {
     fprintf(stderr, "CreateProc2 failed\n"); exit(3);
}
CloseHandle (hReadPipe);

/* Wait for both processes to complete. */
WaitForSingleObject (ProcInfo1.hProcess, INFINITE);
WaitForSingleObject (ProcInfo2.hProcess, INFINITE);
CloseHandle (ProcInfo1.hThread); CloseHandle (ProcInfo1.hProcess);
CloseHandle (ProcInfo2.hThread); CloseHandle (ProcInfo2.hProcess);
return 0;
```

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: \\.\pipe\[path]pipename
  - 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);
```

Named Pipe Client Connections

- CreateFile with named pipe name:
  - \\.\pipe\[path]pipename
  - \\servername\pipe\[path]pipename
  - First method gives better performance (local server)
- Status Functions:
  - GetNamedPipeHandleState
  - SetNamedPipeHandleState
  - GetNamedPipeInfo

Convenience Functions

- WriteFile / ReadFile sequence:

```
BOOL TransactNamedPipe( HANDLE hNamedPipe,
            LPVOID lpvWriteBuf, DWORD cbWriteBuf,
            LPVOID lpvReadBuf, DWORD cbReadBuf,
            LPDOWRD lpcbRead, LPOVERLAPPED lpa);
```

- CreateFile / WriteFile / ReadFile / CloseHandle:
  - dwTimeOut: NMPWAIT_NOWAIT, NMPWAIT_WIAT_FOREVER, NMPWAIT_USE_DEFAULT_WAIT

```
BOOL CallNamedPipe( LPCTSTR lpszPipeName,
            LPVOID lpvWriteBuf, DWORD cbWriteBuf,
            LPVOID lpvReadBuf, DWORD cbReadBuf,
            LPDWORD lpcbRead, DWORD dwTimeOut);
```

Server: eliminate the polling loop

```
BOOL ConnectNamedPipe (HANDLE hNamedPipe,
          LPOVERLAPPED lpo );
```

- lpo == NULL:
  - Call will return as soon as there is a client connection
  - Returns false if client connected between CreateNamed Pipe call and ConnectNamedPipe()
- Use DisconnectNamedPipe to free the handle for connection from another client
- WaitNamedPipe():
  - Client may wait for server's named pipe name (string)
- Security rights for named pipes:
  - GENERIC_READ, GENERIC_WRITE, SYNCHRONIZE

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

Client Example using Named Pipe

```
WaitNamedPipe (ServerPipeName, NMPWAIT_WAIT_FOREVER);
hNamedPipe = CreateFile (ServerPipeName, GENERIC_READ | GENERIC_WRITE,
        0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
if (hNamedPipe == INVALID_HANDLE_VALUE) {
        fptinf(stderr, Failure to locate server.\n"); exit(3);
}
        /* Write the request. */
WriteFile (hNamedPipe, &Request, MAX_RQRS_LEN, &nWrite, NULL);

        /* Read each response and send it to std out. */
while (ReadFile (hNamedPipe, Response.Record, MAX_RQRS_LEN, &nRead, NULL))
        printf ("%s", Response.Record);

CloseHandle (hNamedPipe);
return 0;
```

Server Example Using a Named Pipe

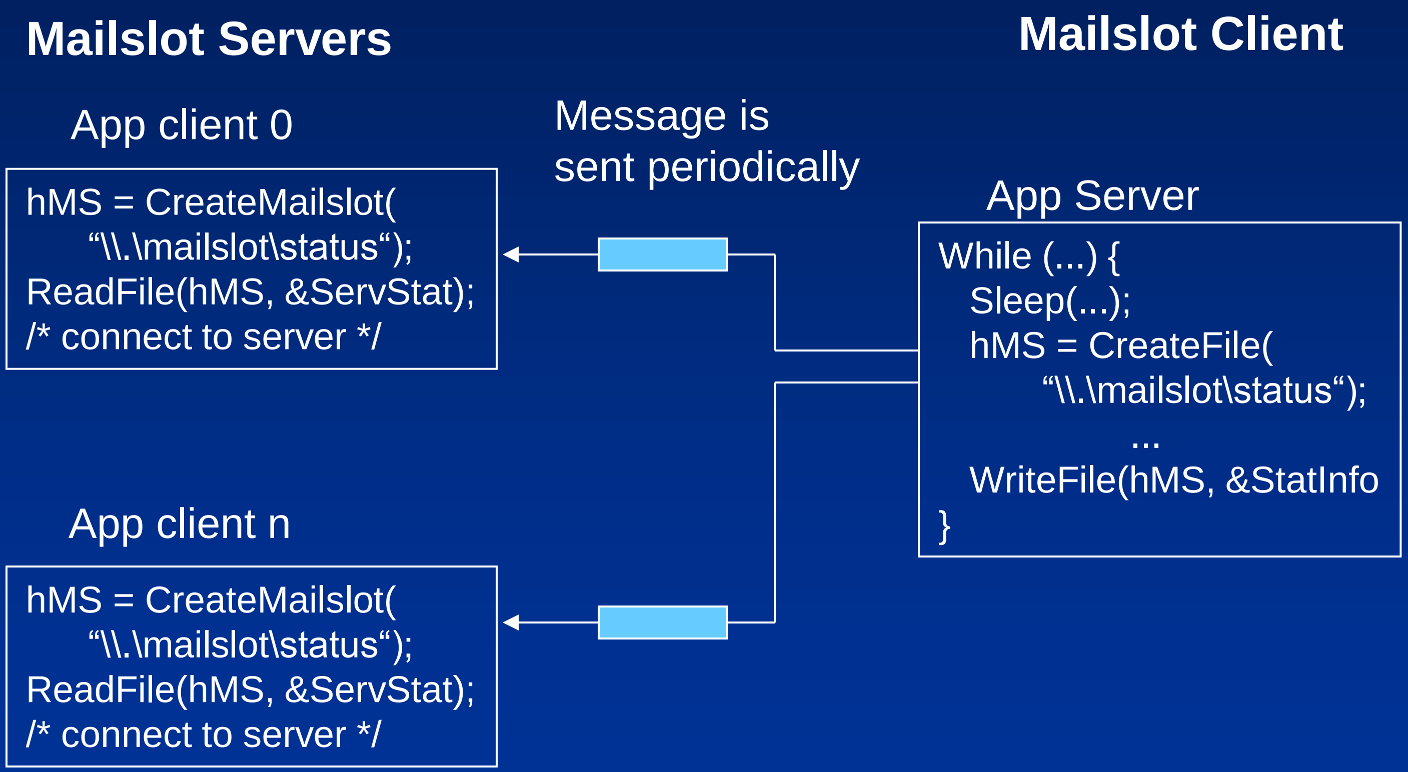
```
hNamedPipe = CreateNamedPipe (SERVER_PIPE_NAME, PIPE_ACCESS_DUPLEX,
            PIPE_READMODE_MESSAGE | PIPE_TYPE_MESSAGE | PIPE_WAIT,
            1, 0, 0, CS_TIMEOUT, pNPSA);
while (!Done) {
            printf ("Server is awaiting next request.\n");
            if (!ConnectNamedPipe (hNamedPipe, NULL)
                        || !ReadFile (hNamedPipe, &Request, RQ_SIZE, &nXfer, NULL)) {
                        fprintf(stderr, “Connect or Read Named Pipe error\n”); exit(4);
            }
            printf( “Request is: %s\n", Request.Record);
            /* Send the file, one line at a time, to the client. */
            fp = fopen (File, "r");
            while ((fgets (Response.Record, MAX_RQRS_LEN, fp) != NULL))
                        WriteFile (hNamedPipe, &Response.Record,
                                    (strlen(Response.Record) + 1) * TSIZE, &nXfer, NULL);
            fclose (fp);
            DisconnectNamedPipe (hNamedPipe);
}    /* End of server operation. */
```

Windows IPC - Mailslots

Mailslots bear some nasty implementation details; they are almost never used

- 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 (< 424 bytes)
- 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

Opening a mailslot

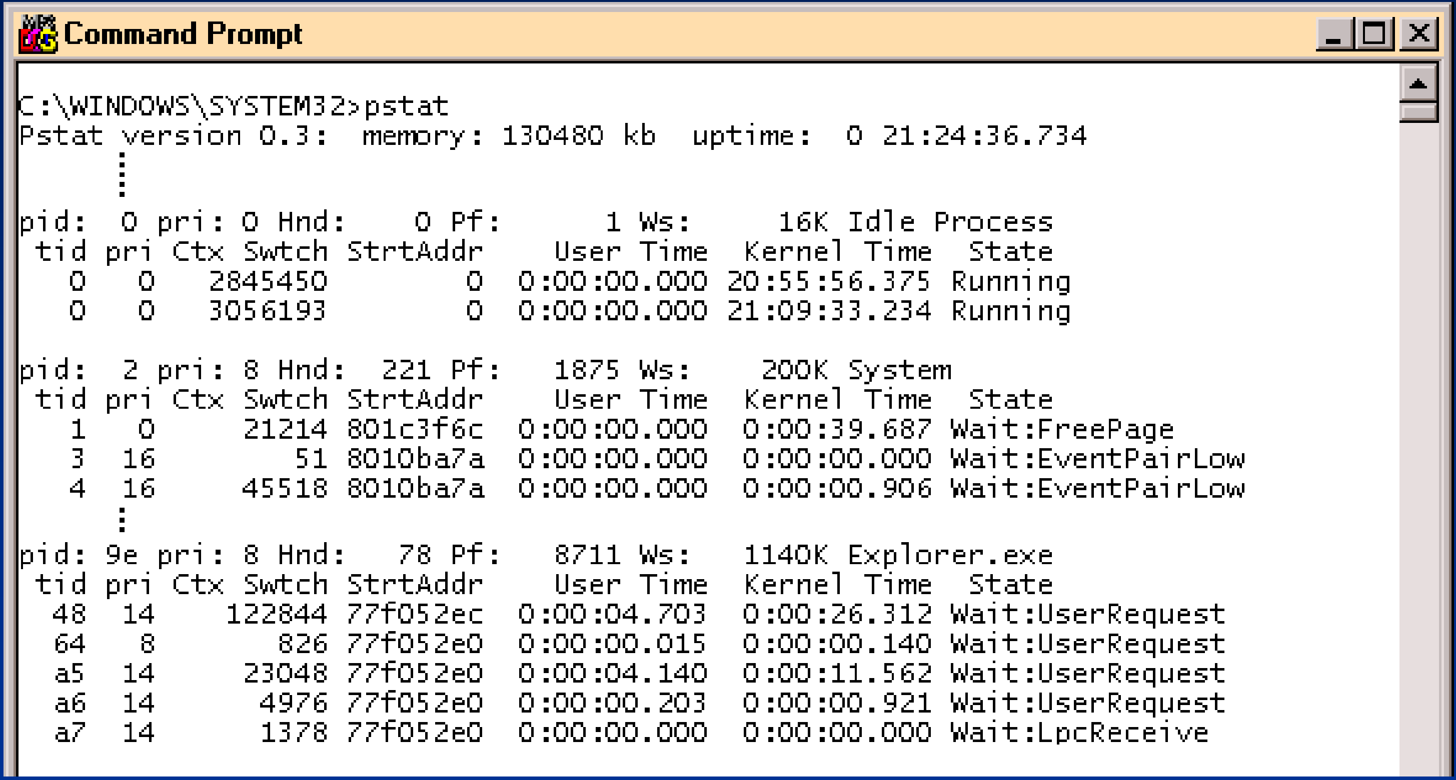
- CreateFile with the following names:
  - \\.\mailslot\[path]name - retrieve handle for local mailslot
  - \\host\mailslot\[path]name - retrieve handle for mailslot on specified host
  - \\domain\mailslot\[path]name - returns handle representing all mailslots on machines in the domain
  - *\mailslot\[path]name - returns handle representing mailslots on machines in the system‘s primary domain: max mesg. len: 400 bytes
  - Client must specifiy FILE_SHARE_READ flag
- GetMailslotInfo() and SetMailslotInfo() are similar to their named pipe counterparts

Lab: Viewing Global Queued Spinlocks

```
kd> !qlocks  Key: O = Owner,1-n = Waitorder, blank = notowned/waiting, C = Corrupt
                     Processor Number
        LockName 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
KE-Dispatcher O
KE-ContextSwap
MM-PFN
MM-SystemSpace
CC-Vacb

CC– Master
```

Lab: Looking at Waiting Threads

- For waiting threads, user-mode utilities only display the wait reason
- Example: pstat



```
C:\WINDOWS\SYSTEM32>pstat
Pstat version 0.3:  memory: 130480 kb  uptime:  0 21:24:36.734
      ⋮
pid:  0 pri: 0 Hnd:    0 Pf:      1 Ws:     16K Idle Process
 tid pri Ctx Swtch StrtAddr    User Time  Kernel Time   State
   0   0   2845450        0  0:00:00.000 20:55:56.375 Running
   0   0   3056193        0  0:00:00.000 21:09:33.234 Running

pid:  2 pri: 8 Hnd:  221 Pf:   1875 Ws:    200K System
 tid pri Ctx Swtch StrtAddr    User Time  Kernel Time   State
   1   0     21214 801c3f6c  0:00:00.000  0:00:39.687 Wait:FreePage
   3  16        51 8010ba7a  0:00:00.000  0:00:00.000 Wait:EventPairLow
   4  16     45518 8010ba7a  0:00:00.000  0:00:00.906 Wait:EventPairLow
      ⋮
pid: 9e pri: 8 Hnd:   78 Pf:   8711 Ws:   1140K Explorer.exe
 tid pri Ctx Swtch StrtAddr    User Time  Kernel Time   State
  48  14    122844 77f052ec  0:00:04.703  0:00:26.312 Wait:UserRequest
  64   8       826 77f052e0  0:00:00.015  0:00:00.140 Wait:UserRequest
  a5  14     23048 77f052e0  0:00:04.140  0:00:11.562 Wait:UserRequest
  a6  14      4976 77f052e0  0:00:00.203  0:00:00.921 Wait:UserRequest
  a7  14      1378 77f052e0  0:00:00.000  0:00:00.000 Wait:LpcReceive
```

- To find out <u>what</u> a thread is waiting on, must use kernel debugger

Further Reading

- Mark E. Russinovich, *et al*. Windows Internals, 5th Edition, Microsoft Press, 2009.
  - Synchronization (from pp.170-198)
  - Named Pipes and Mailslots (from pp. 1021)
- Ben-Ari, M., Principles of Concurrent Programming, Prentice Hall, 1982
- Lamport, L., The Mutual Exclusion Problem, Journal of the ACM, April 1986
- Abraham Silberschatz, Peter B. Galvin, Operating System Concepts, John Wiley & Sons, 6th Ed., 2003;
  - Chapter 7 - Process Synchronization
  - Chapter 8 - Deadlocks
- Jeffrey Richter, Programming Applications for Microsoft Windows, 4th Edition, Microsoft Press, September 1999.
  - Chapter 10 - Thread Synchronization
  - Critical Sections, Mutexes, Semaphores, Events (from pp. 315)
- Johnson M. Hart, Win32 System Programming: A Windows® 2000 Application Developer's Guide, 2nd Edition, Addison-Wesley, 2000.

Source Code References

- Windows Research Kernel sources
  - \base\ntos\ke – primitive kernel support
    - eventobj.c - Event object
    - mutntobj.c – Mutex object
    - semphobj.c – Semaphore object
    - timerobj.c, timersup.c – Timers
    - wait.c, waitsup.c – Wait support
  - \base\ntos\ex – executive object (layered on kernel support)
    - Event.c – Event object
    - Mutant.c – Mutex object
    - Semphore.c – Semaphore object
    - Timer.c – Timer object
  - \base\ntos\inc\ke.h, ex.h – structure/type definitions