

**Session 8**  
**Windows Thread Synchronization –**  
**Session I**  
**WSP4, Chapters 8, 10**

# ***OBJECTIVES***

**Upon completion of this Session, you will be able to:**

- ◆ **Describe the various Windows synchronization mechanisms**
- ◆ **Differentiate synchronization object features and how to select between them**
- ◆ **Use synchronization in Windows applications**

# ***AGENDA***

- Part I      Need for Synchronization**
- Part II     Thread Synchronization Objects**
- Part III    CRITICAL\_SECTIONs**
- Part IV     Deadlock Example**
- Part V     Interlocked Functions**
- Part VI     Mutexes for Mutual Exclusion**
- Part VII    Events**
- Part VIII   Semaphores**
- Part IX     Synchronization Object Summary**
- Part X     Lab/Demo Exercise 8-1**
- Part XI     Condition Variable Model**
- Part XII    Hints: Designing, Debugging, and Testing**
- Part XIII   Lab/Demo Exercise 8-2**

# ***IMPORTANT APIs IN THIS SESSION***

**Initialize/Delete/Enter/Leave/TryCriticalSection**

**InterlockedIncrement/Decrement/Exchange/...**

**CreateMutex/Event/Semaphore**

**OpenMutex/Event/Semaphore**

**ReleaseMutex/Semaphore, Pulse/SetResetEvent**

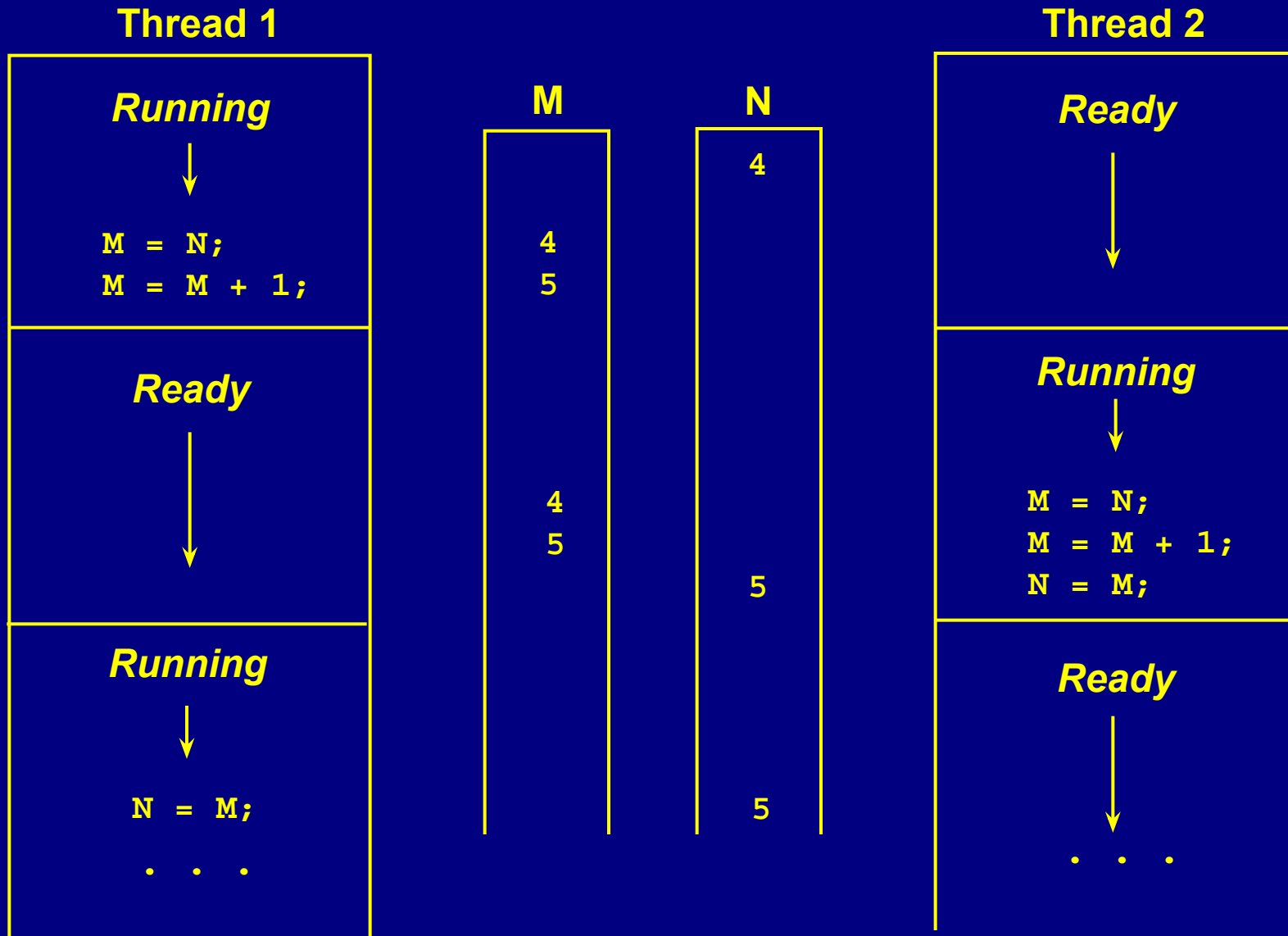
**SignalObjectAndWait**

# ***Part I - Need for Synchronization***

**Why is thread synchronization required? Examples:**

- ◆ **Boss thread cannot proceed until workers complete**
- ◆ **Worker cannot proceed until its environment is initialized**
- ◆ **A thread cannot proceed until certain conditions are satisfied. Ex:**
  - **A free buffer is available**
  - **A buffer is filled with data to be processed.**
- ◆ **Threads must not update the same variable concurrently**
  - **Read-modify-write**
- ◆ **Without proper synchronization, you risk defects such as:**
  - **Race conditions**
  - **Concurrent update to a single resource**

# A SYNCHRONIZATION PROBLEM



# ***Part II - Thread Synchronization Objects***

## **Known Windows mechanism to synchronize threads:**

- ◆ **A thread can wait for another to terminate (using `return`) by waiting on the thread handle using `WaitForSingleObject` or `WaitForMultipleObjects`**
  - **A process can wait for another process to terminate (`return`) in the same way**

## **Other common methods (not covered here):**

- ◆ **Reading from a pipe or socket that allows one process or thread to wait for another to write to the pipe (socket)**
- ◆ **File locks are specifically for synchronizing file access**

# ***Synchronization Objects***

**Windows (pre-Vista) provides four other objects for thread and process synchronization**

**Three are kernel objects (they have HANDLES)**

- ◆ **Events**
- ◆ **Semaphores**
- ◆ **Mutexes**

**Many inherently complex problems – beware:**

- ◆ **Deadlocks**
- ◆ **Race conditions**
- ◆ **Missed signals**
- ◆ **Many more**



# ***Critical Section Objects***

## **Critical sections**

- ◆ **Fourth object type can only synchronize threads within a process**
- ◆ **Often the most efficient choice**
  - **Apply to many application scenarios**
  - **“Fast mutexes”**
  - **Not kernel objects**
- ◆ **Critical section objects are initialized, not created**
  - **Deleted, not closed**
- ◆ **Threads enter and leave critical sections**
- ◆ **Only 1 thread at a time can be in a critical code section**
- ◆ **There is no handle — there is a `CRITICAL_SECTION` type**

# ***Part III - CRITICAL\_SECTIONS***

```
VOID InitializeCriticalSection (  
    LPCRITICAL_SECTION lpCriticalSection)
```

```
VOID DeleteCriticalSection (  
    LPCRITICAL_SECTION lpCriticalSection)
```

```
VOID EnterCriticalSection (  
    LPCRITICAL_SECTION lpCriticalSection)
```

# ***CRITICAL\_SECTION Management***

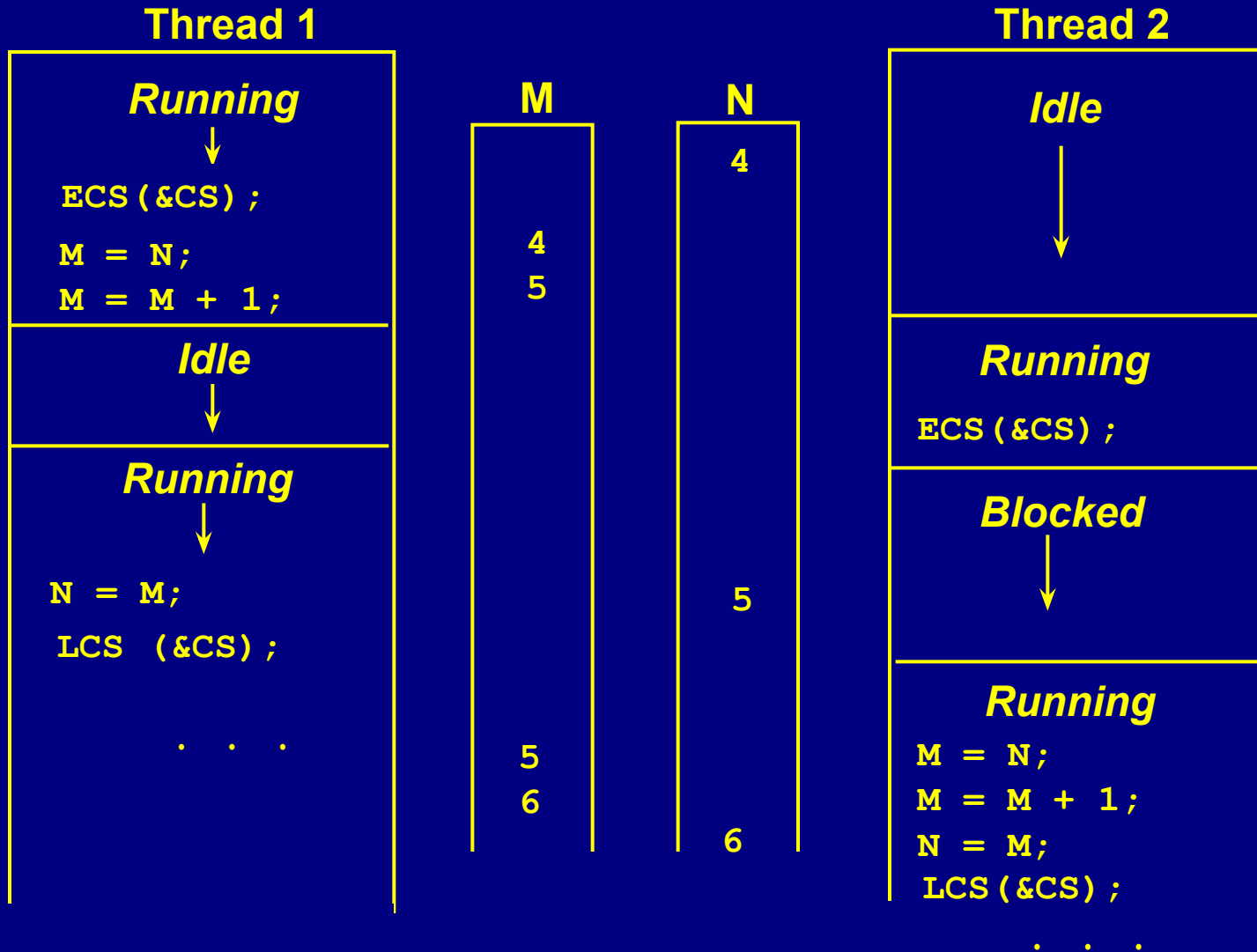
```
VOID LeaveCriticalSection (  
    LPCRITICAL_SECTION lpcsCriticalSection)
```

```
BOOL TryCriticalSection (  
    LPCRITICAL_SECTION lpcsCriticalSection)
```

# ***CRITICAL\_SECTION Usage***

- ◆ **EnterCriticalSection** blocks a thread if another thread is in (“owns”) the section
  - Use **TryCriticalSection** to avoid blocking
  - A thread can enter a CS more than once (“recursive”)
- ◆ The waiting thread unblocks when the “owning” thread executes **LeaveCriticalSection**
  - A thread must leave a CS once for every time it entered
- ◆ **Common usage: allow threads to access global variables**
  - **Session 9: Cache issues**

# SYNCHRONIZATION CSs



# **CRITICAL SECTIONS AND finally**

**Here is method to assure that you leave a critical section**

- ◆ **Even if someone later modifies your code**
- ◆ **This technique also works with file locks and the other synchronization objects discussed next**

## finally (2 of 2)

```
CRITICAL_SECTION cs;  
...  
InitializeCriticalSection (&cs);  
...  
EnterCriticalSection (&cs);  
_try { ... }  
_finally { LeaveCriticalSection (&cs); }
```

# ***CRITICAL\_SECTION Comments***

**CRITICAL\_SECTIONS test the lock in user-space**

- ◆ **Fast – no kernel call**
- ◆ **But wait in kernel space**

**Almost always faster than Mutexes**

- ◆ **Factors include number of threads, number of processors, and amount of thread contention**

**Performance can sometimes be “tuned”**

- ◆ **Adjust the “spin count” – more later**
- ◆ **CSs operate using polling and the equivalent of interlocked functions**



# ***Part IV - Deadlock Example***

**Here is a program defect that could cause a deadlock**

- ◆ ***Some function calls are abbreviated for brevity***
- ◆ **Deadlocks are specific kind of *race condition***
- ◆ **Always enter CSs in the same order; leave in reverse order**
- ◆ **To avoid deadlocks, create a lock hierarchy**

**You can generalize this example to multiple CSs**

# Deadlock Example

```
CRITICAL_SECTION csM, csN;
volatile DWORD M = 0, N = 0;
InitializeCriticalSection (&csM); ICS (&csN);
...
DWORD ThreadFunc (...)
{
    ECS (&csM); ECS (&csN);
    M = ++N; N = M - 2;
    LCS (&csN); LCS (&csM);
    ... How would you fix it?
    ECS (&csN); ECS (&csM);
    M = N--; N = M + 2;
    LCS (&csN); LCS (&csM);
}
```

# ***Part V - Interlocked Functions***

**For simple manipulation of signed 32-bit numbers, you can use the *interlocked* functions**

- ◆ **There are also 64-bit versions**
  - **64-bit integer access is not atomic on 32-bit systems**
- ◆ **All operations are atomic**
- ◆ **Operations take place in user space**
  - **No kernel call, but, a memory barrier (mistake previously)**
- ◆ **Easy, fast, no deadlock risk**
- ◆ **Limited to increment, decrement, exchange, exchange/add, and compare/exchange**
- ◆ **Can not directly solve general mutual exclusion problems**

# *Interlocked Functions*

`LONG InterlockedIncrement(LONG volatile *lpAddend)`

`LONG InterlockedDecrement(LONG volatile *lpAddend)`

- ◆ **Return the resulting value**
  - Which might change before you can use the value

`LONG InterlockedExchangeAdd(InterlockedDecrement(  
LONG volatile *lpAddend, LONG Increment)`

- ◆ **Return value is the old value of \*lpAddend**

`LONG InterlockedExchange(InterlockedDecrement(  
LONG volatile *lpTarget, LONG Value)`

- ◆ **Return value is the old value of \*lpTarget**

# Interlocked CompareExchange

LONG InterlockedCompareExchange (

LONG volatile \*lpDest, LONG Exch,  
LONG Comparand)

- ◆ Operands must be 4-byte aligned
- ◆ Returns initial value of \*lpDest
- ◆ \*lpdest = (\*lpDest == Comparand) ?  
Exch : \*lpDest

# ***Other Interlocked Functions***

**InterlockedExchangePointer**

**InterlockedAnd, InterlockedOr, InterlockedXor**

- ◆ 8, 16, 32, and 64-bit versions

**InterlockedIncrement64, InterlockedDecrement64**

**InterlockedCompare64Exchange128**

- ◆ Compares 64-bit objects, exchanges 128-bit

# *Part VI - Mutexes (1 of 6)*

- ◆ **Mutexes can be named and have HANDLES**
  - They are kernel objects
- ◆ **They can be used for interprocess synchronization**
- ◆ **They are owned by a thread rather than a process**
- ◆ **Threads gain mutex ownership by waiting on mutex handle**
  - With `WaitForSingleObject` or `WaitForMultipleObjects`
- ◆ **Threads release ownership with `ReleaseMutex`**

# Mutexes (2 of 6)

- ◆ **Recursive**: A thread can acquire a specific mutex several times but must release the mutex the same number of times
  - Can be convenient, for example, with nested transactions
- ◆ You can poll a mutex to avoid blocking
- ◆ A mutex becomes “abandoned” if its owning thread terminates
- ◆ Events and semaphores are the other kernel objects
  - Very similar life cycle and usage



# Mutexes (3 of 6)

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpsa,  
    BOOL fInitialOwner,  
    LPCTSTR lpszMutexName)
```

- ◆ The `fInitialOwner` flag, if `TRUE`, gives the calling thread immediate ownership of the new mutex
  - It is overridden if the *named* mutex already exists
- ◆ `lpszMutexName` points to a null-terminated pathname
  - Pathnames are case sensitive
  - Mutexes are unnamed if the parameter is `NULL`

# Mutexes (4 of 6)

**BOOL ReleaseMutex(HANDLE hMutex)**

- ◆ **ReleaseMutex** frees a mutex that the calling thread owns
  - Fails if the thread does not own it
- ◆ **If a mutex is abandoned, a wait will return `WAIT_ABANDONED_0`**
  - This is the base value on a wait multiple
- ◆ **OpenMutex** opens an existing named mutex
  - Allows threads in different processes to synchronize

# Mutexes (5 of 6)

## Mutex naming:

- ◆ Name a mutex that is to be used by more than one process
  - Mutexes, semaphores, & events share the same name space
  - Memory mapping objects also use this name space
  - Also waitable timers
  - And, all processes share this name space
  - **Alert: Name collisions – name carefully**
- ◆ Normal: Don't name a mutex used in a single process

# Mutexes (6 of 6)

## Process interaction with a named mutex

Same name space as used for mem maps, ...

### Process 1

```
h = CreateMutex ("MName");
```

### Process 2

```
h = OpenMutex ("MName");
```



# ***Part VII - EVENTS (1 of 6)***

- ◆ **Events can release multiple threads from a wait simultaneously when a single event is signaled**
- ◆ ***A manual-reset* event can signal several threads simultaneously and must be reset by the thread**
- ◆ ***An auto-reset* event signals a single thread, and the event is reset automatically**
- ◆ **Signal an event with either `PulseEvent` or `SetEvent`**
- ◆ **Four combinations with very different behavior**
  - **Be careful! There are numerous subtle problems**
- ◆ **Recommendation: Only use with `SignalObjectAndWait()`**
  - **(Much) more on this later**

# ***EVENTS (2 of 6)***

```
HANDLE CreateEvent(  
    LPSECURITY_ATTRIBUTES lpsa,  
    BOOL fManualReset, BOOL fInitialState,  
    LPTCSTR lpszEventName)
```

- ◆ **Manual-reset event: set fManualReset to TRUE**
- ◆ **Event is initially set to signaled if fInitialState is TRUE**
- ◆ **Open a named event with OpenEvent**

# ***EVENTS (3 of 6)***

**The three functions for controlling events are:**

**BOOL SetEvent(HANDLE hEvent)**

**BOOL ResetEvent(HANDLE hEvent)**

**BOOL PulseEvent(HANDLE hEvent)**

# ***EVENTS (4 of 6)***

- ◆ **A thread signals an event with `SetEvent`**
- ◆ **If the event is auto-reset, a single waiting thread (possibly one of many) will be released**
  - **The event automatically returns to the non-signaled state**
- ◆ **If no threads are waiting on the event, it remains in the signaled state until some thread waits on it and is immediately released**



# ***EVENTS (5 of 6)***

- ◆ **If the event is manual-reset, the event remains signaled until some thread calls `ResetEvent` for that event**
  - **During this time, all waiting threads are released**
  - **It is possible that other threads will wait, and be released, before the reset**
- ◆ **`PulseEvent` allows you to release all threads currently waiting on a manual-reset event**
  - **The event is then automatically reset**

# ***EVENTS (6 of 6)***

- ◆ **When using `WaitForMultipleEvents`, wait for all events to become signaled**
  - **A waiting thread will be released only when all events are simultaneously in the signaled state**
  - **Some signaled events might be released before the thread is released**

# Event Notes

## Behavior depends on manual or auto reset, Pulse or Set Event

- ◆ All 4 forms are useful

	<i>AutoReset</i>	<i>ManualReset</i>
<b>SetEvent</b>	Exactly one thread is released. If none are currently waiting on the event, the next thread to wait will be released.	All currently waiting threads released. The event remains signaled until reset by some thread.
<b>PulseEvent</b>	Exactly one thread is released, but only if a thread is currently waiting on the event.	All currently waiting threads released, and the event is then reset.

# ***Part VIII - SEMAPHORES (1 of 4)***

- ◆ **A semaphore combines event and mutex behavior**
  - **Can be emulated with one of each and a counter**
- ◆ **Semaphores maintain a count**
  - **No ownership concept**
- ◆ **The semaphore object is *signaled* when the count is greater than zero, and the object is *not signaled* when the count is zero**
- ◆ **With care, you can achieve mutual exclusion with a semaphore**

# ***SEMAPHORES (2 of 4)***

- ◆ **Threads or processes wait in the normal way, using one of the wait functions**
- ◆ **When a waiting thread is released, the semaphore's count is incremented by one**
  - **Any thread can release**
  - **Not restricted to the thread that “acquired” the semaphore**
  - **Consider a producer/consumer model to see why**

# SEMAPHORES (3 of 4)

```
HANDLE CreateSemaphore(  
    LPSECURITY_ATTRIBUTES lpsa,  
    LONG cSemInitial, LONG cSemMax,  
    LPCTSTR lpszSemName)
```

- ◆ **cSemMax** is the maximum value for the semaphore
- ◆ **Must be one or greater**
- ◆  $0 \leq cSemInitial \leq cSemMax$  is the initial value
- ◆ **You can only decrement the count by one with any given wait operation, but you can release a semaphore and increment its count by any value up to the maximum value**

# ***SEMAPHORES (4 of 4)***

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

- ◆ **You can find the count preceding the release, but the pointer can be `NULL` if you do not need this value**
- ◆ **The release count must be greater than zero, but if it would cause the semaphore count to exceed the maximum, the call will return `FALSE` and the count will remain unchanged**
- ◆ **There is also an `OpenSemaphore` function**

# ***A SEMAPHORE DEADLOCK DEFECT***

**There is no “atomic” wait for multiple semaphore units**

- ◆ **But you can release multiple units atomically!**
- ◆ **Here is a potential deadlock in a thread function**

```
for (i = 0; i < NumUnits; i++)  
    WaitForSingleObject (hSem, INFINITE);
```

- ◆ **Solution: Treat the loop as a critical section, guarded by a `CRITICAL_SECTION` or mutex**
- ◆ **Or, a multiple wait semaphore can be created with an event, mutex, and counter – this is an optional lab**



***Part IX - Windows***  
***SYNCHRONIZATION OBJECTS***

**Summary**

# CRITICAL SECTION

<b>Named, Securable Synchronization Object</b>	<b>No</b>
<b>Accessible from Multiple Processes</b>	<b>No</b>
<b>Synchronization</b>	<b>Enter</b>
<b>Release</b>	<b>Leave</b>
<b>Ownership</b>	<b>One thread at a time. Recursive</b>
<b>Effect of Release</b>	<b>One waiting thread can enter</b>

# MUTEX

<b>Named, Securable Synchronization Object</b>	<b>Yes</b>
<b>Accessible from Multiple Processes</b>	<b>Yes</b>
<b>Synchronization</b>	<b>Wait</b>
<b>Release</b>	<b>Release or owner terminates</b>
<b>Ownership</b>	<b>One thread at a time. Recursive</b>
<b>Effect of Release</b>	<b>One waiting thread can gain ownership after last release</b>

# SEMAPHORE

<b>Named, Securable Synchronization Object</b>	<b>Yes</b>
<b>Accessible from Multiple Processes</b>	<b>Yes</b>
<b>Synchronization</b>	<b>Wait</b>
<b>Release</b>	<b>Any thread can release</b>
<b>Ownership</b>	<b>N/A – Many threads at a time, up to the maximum count</b>
<b>Effect of Release</b>	<b>Multiple threads can proceed, depending on release count</b>

# EVENT

<b>Named, Securable Synchronization Object</b>	<b>Yes</b>
<b>Accessible from Multiple Processes</b>	<b>Yes</b>
<b>Synchronization</b>	<b>Wait</b>
<b>Release</b>	<b>Set, Pulse</b>
<b>Ownership</b>	<b>N/A – Any thread can Set or Pulse an event</b>
<b>Effect of Release</b>	<b>One or several waiting threads will proceed after a Set or Pulse - Caution</b>

# ***Part X - Lab/Demo Exercise 8-1***

**Create two working threads: A producer and a consumer**

- ◆ **The producer periodically creates a message**
  - **The message is checksummed**
- ◆ **The consumer prompts the user for one of two commands**
  - **Consume the most recent message – wait if necessary**
  - **Stop**
- ◆ **Once the system is stopped, print summary statistics**
- ◆ **Start with `eventPC_x.c`. A correct solution is provided**
- ◆ **`simplePC.c` is similar, but simpler. It also uses a CS**
- ◆ **Note how the mutex/CS guards the data object**
- ◆ **Note how the event signals a change in the data object**

# ***Part XI - Condition Variable Model***

## ***Using Models (or Patterns)***

**Use well-understood and familiar techniques and models**

- ◆ **Aid development, understanding and maintenance**
- ◆ **“Boss/worker” and “work crew” models**
- ◆ **Critical section essential for mutexes**
- ◆ **Even defects have models (deadlocks)**
  - **“Anti-pattern”**
- ◆ **Helps to understand and control event behavior**

# ***Events and Mutexes Together (1 of 2)***

## ***Similar to POSIX Pthreads***

### **Illustrated with a message producer/consumer**

- ◆ **The mutex and event are both associated with the message block data structure**
- ◆ **The mutex defines the critical section for accessing the message data structure object**
  - **Assures the object's invariants**
- ◆ **The event is used to signal that there is a new message**
  - **Signals that the object has changed to a specified state**



# ***Events and Mutexes Together (2 of 2)***

- ◆ **One thread (producer) locks the data structure**
  - **Changes the object's state by creating a new message**
  - **Sets or pulses the event – new message**
- ◆ **One or more threads (consumers) wait on the event for the object to reach the desired state**
  - **The wait must occur outside the critical section**
- ◆ **A consumer thread can also lock the mutex**
  - **And test the object's state**

# ***The Condition Variable Model (1 of 4)***

## **Several key elements:**

- ◆ **Data structure of type `STATE_TYPE`**
  - **Contains all the data such as messages, checksums, etc.**
- ◆ **A mutex and one or more events associated with the data structure**
- ◆ **One or more Boolean functions to evaluate the “condition variable predicates”**
  - **For example, “a new message is ready”**
  - **An event is associated with each condition variable predicate**

# The Condition Variable Model (2 of 4)

```
typedef struct _state_t {
    HANDLE Guard; /* Mutex to protect the object */
    HANDLE CvpSet;
    /* Autoreset Event for "signal" model */
    . . . other condition variables
    /* State structure with counts, etc. */
    struct STATE_VAR_TYPE StateVar;
} STATE_TYPE State;


. . .
/* Initialize State, creating mutex and event */
. . .
```

# The Condition Variable Model (3 of 4)

```
/* This is the "Signal" CV model */
/* PRODUCER thread that modifies State */
WaitForSingleObject(State.Guard, INFINITE);
/* Change state so that the CV predicate holds */
. . . State.StateVar.xyz = . . . ;
SetEvent (State.CvpSet); /* Signal one consumer */
ReleaseMutex (State.Guard);
/* End of the interesting part of the producer */
. . .
```

# The Condition Variable Model (4 of 4)

```
/* CONSUMER thread waits for a particular state */
WaitForSingleObject(State.Guard, INFINITE);
while (!cvp(&State)) {
    ReleaseMutex(State.Guard);
    WaitForSingleObject(State.CvpSet, Timeout);
    WaitForSingleObject(State.Guard, INFINITE);
}
. . .
ReleaseMutex(State.Guard);
/* End of the interesting part of the consumer */
```



**Alert**

# ***Condition Variable Model Comments***

## **Three essential steps of the loop in the consumer**

- ◆ **Unlock the mutex**
- ◆ **Wait on the event**
- ◆ **Lock the mutex again**

**Note: Pthreads (UNIX) combines these into a single function.  
Windows does this in NT 6.x (Next session)**

**Windows `SignalObjectAndWait` performs the first two steps atomically – NT 5.x (there's still legacy code that avoids SOAW)**

- **No timeout and no missed signal**

# Using SignalObjectAndWait()

```
/* CONSUMER thread waits for NEXT state change */
/* Use SignalObjectAndWait() */
WaitForSingleObject (State.Guard, INFINITE);
do {
    SignalObjectAndWait(State.Guard,
                       State.CvpSet, INFINITE);
    WaitForSingleObject (State.Guard, INFINITE);
} while (!cvp(&State));
/* Thread now owns the mutex and cvp(&State) holds */
/* Take appropriate action, perhaps modifying State */
. . .
ReleaseMutex (State.Guard);
/* End of the interesting part of the consumer */
```

# CV Model Variation

In producer/consumer code, ONE consumer is released

- *Signal CV model*
- ◆ Auto-reset event, `SetEvent`

Or, there may be only one message available and multiple consuming threads – *broadcast CV model*

- ◆ Event should be manual-reset
- ◆ Producer should call `PulseEvent`
  - Assure exactly one thread is released
  - Careful: Risk of missed signal, even with SOAW
  - You need the timeout – consumer thread could preempt the thread and the signal could be missed
- ◆ NT 6.x has a real condition variable (at last)
- ◆ CV model is basic in POSIX Pthreads



# Broadcast CV Model Consumer

```
/* CONSUMER thread waits for NEXT state change */
/* Event is manual-reset */
WaitForSingleObject (State.Guard, INFINITE);
do {
    SignalObjectAndWait (State.Guard,
                        State.CvpSet, Timeout);
    WaitForSingleObject (State.Guard, INFINITE);
} while (!cvp(&State));
/* Thread owns the mutex and cvp(&State) holds */
. . .
ReleaseMutex (State.Guard);
/* End of the interesting part of the consumer */
```

# ***Part XII - Multithreading: Designing, Debugging, Testing Hints***

## **Avoiding Incorrect Code (1 of 3)**

- ◆ **Pay attention to design, implementation, and use of familiar programming models**
- ◆ **The best debugging technique is not to create bugs in the first place – Easy to say, harder to do**
- ◆ **Many serious defects will elude the most extensive and expensive testing**
- ◆ **Debuggers change timing behavior**
  - **Masking the race conditions you wish to expose**

# ***Avoiding Incorrect Code (2 of 3)***

- ◆ **Avoid relying on “thread inertia”**
- ◆ **Never bet on a thread race**
- ◆ **Scheduling is not the same as synchronization**
- ◆ **Sequence races can occur**
  - **Even when you use mutexes to protect shared data**
- ◆ **Cooperate to avoid deadlocks**
- ◆ **Never share events between predicates**

# ***Avoiding Incorrect Code (3 of 3)***

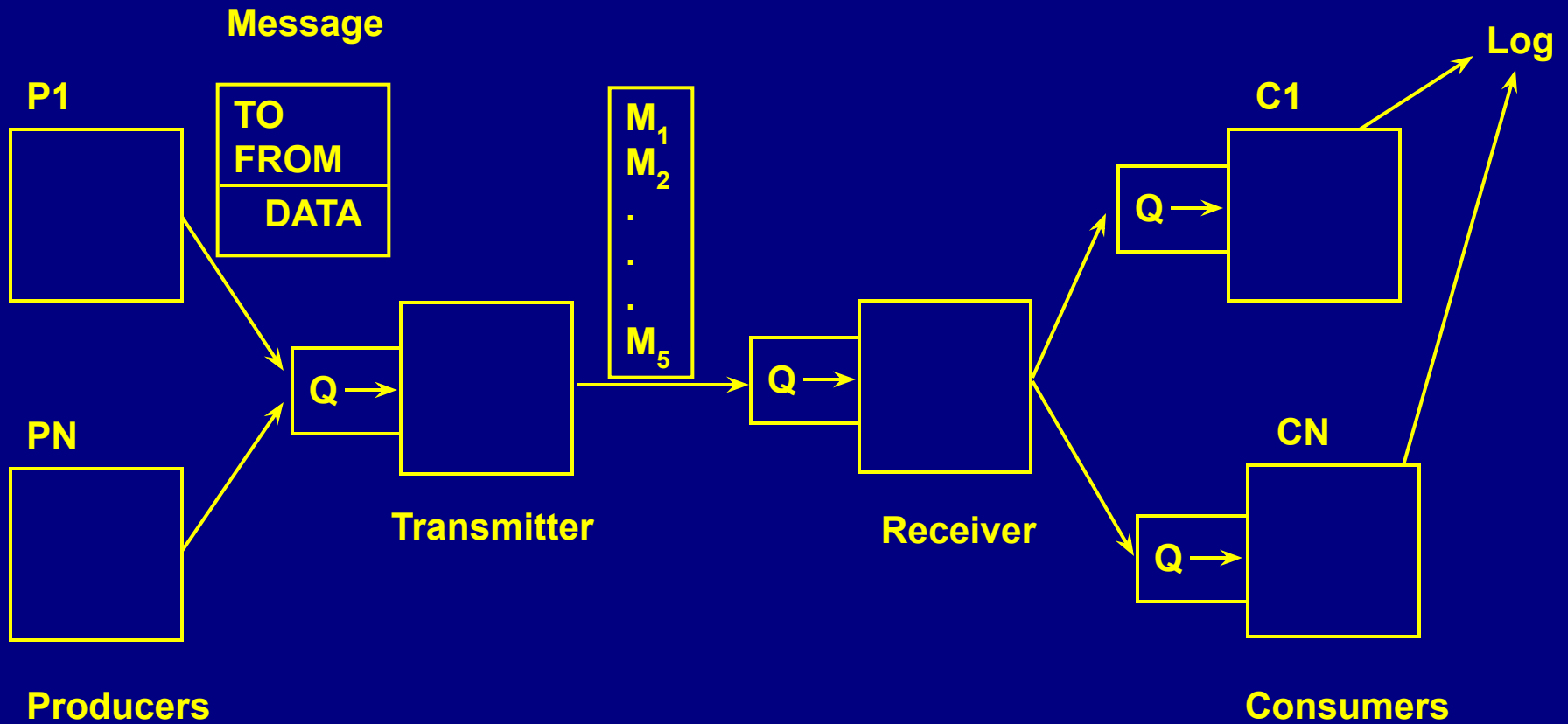
- ◆ **Beware of sharing stacks and related memory corrupters**
- ◆ **Use the condition variable model properly**
- ◆ **Understand your invariants and condition variable predicates**
- ◆ **Keep it simple**
- ◆ **Test on multiple systems (single and multiprocessor)**
- ◆ **Testing is necessary but not sufficient**
- ◆ **Be humble**

# Part XIII - Lab 8-2

## Debug and test the ThreeStage pipeline implementation

- ◆ Two libraries are required: `Messages.c` and `QueueObj.c`
- ◆ The Compete directory has numerous alternative implementations of `QueueObj.c`. As an alternative to fixing the defective version, time and compare the alternative solutions
- ◆ Also, check out the variants: `QueueObjCS.c`, etc.
- ◆ Add thread cancelation (see `ThreeStageCancel`, `QueueObjCancel`) – *discuss & Supplemental Session 9*
- ◆ Idea: Use a real signature on the messages – performance impact?
- ◆ Alternative: Port the Pthreads implementation
  - There is an open source library, but don't use it

# Lab 8-2: Multistage Pipeline



# *Harder Exercises*

1. **MultiSem – Atomic multiple wait semaphore**
  - The solution must also work between processes
  - This will require memory mapping
  - There are x, xx, and xxx versions
  
2. **compMP**
  - Windows CMD utility to compare files
  - Find first 8 (arbitrary) different bytes & report in order
  - Requires “speculative processing”
  - My solution is much faster than the CMD version and it scales well with processor count
  - x, xx, and xxx versions