Synchronization

Synchronization Primitives

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Objectives

- Upon completion of this unit you will be able to synchronize threads and processes in Windows
- We must
  - Describe the various Windows synchronization mechanisms
    - Critical Sections
    - Mutexes
    - Semaphores
    - Events
  - Differentiate synchronization object features
  - Understand how to select among them

Synchronization, hug?

- Why is process (P) or thread (T) synchronization required?
  - Without proper synchronization, you risk defects such as
    - Race conditions
    - Concurrent update to a resource
  - In general, a T (P) cannot proceed until certain conditions are satisfied
    - The boss thread cannot proceed until worker threads complete

The software's behaviour is dependent on the sequence or timing

Example: (LIFO) Stack

P_i / T_i

```
void push (int val) {
  if(top>=SIZE)
    return;
  stack[top] = val;
  top++;
  return;
}
```

P_j / T_j

```
void pop (int *val) {
  if(top<=0)
    return;
  top--;
  *val = stack[top];
  return;
}
```

- Functions **push** and **pop**
  - Modify the same array extreme
  - Variable **top** is shared and not atomically modified

It is possible to overwrite or lose a data item

top++ becomes:
register = top; register = register+1; top = register;
The same transformation is done for top--.
Which order?

Example: (LIFO) Stack

P_i / T_i

```
void push (int val) {
  if(top>=SIZE)
    return;
  stack[top] = val;
  top++;
  return;
}
```

register = top
register = register+1
top = register

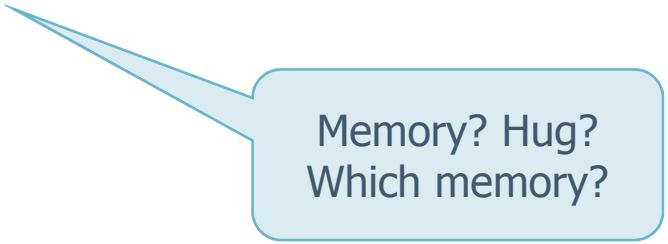
- There is one more latent problem in the previous code segment
  - Is the machine code always completely implemented/run or there may be some compiler (or CPU) optimization?

Volatile variables

- When a variable is modified, a thread may hold its value in a register
  - If the variable is not copied back to memory the change is not visible to other threads
  - The ANSI C **volatile** quantifier ensures that
    - The variable will be always fetched from memory before use
    - The variable will be always stored to memory after modification
  - Volatile variables must be declare as
    - **volative** DWORD var;

Volatile variables

- The **volatile** quantifier
  - Informs the compiler that the variable can change at any time
  - Tells the compiler the variable must be
    - Fetched from memory every time
    - Storere into **memory** after it is modified



  - Can negatively effect performance

Volatile variables

- Unfortunately, even if a variable is **volatile** a processor may hold its value into the **cache** memory
  - In multi-core architectures each core has its own cache (level 1 and level 2) memory
  - Each thead may copy the variable into its own cache before committing it into the main memory
  - There is no assurance that the new value (even if the object is volatile) will be visible to threads running on other cores

Volatile variables

- This behavior may alter the order in which different processor may modify it
  - To ensure that changes are visible by all processors we must use "memory barriers" (or "memory fences")
    - A memory barrier assures that the value is moved to main memory
    - A memory barrier assures cache coherence
  - All the following synchronization functions may act as memory barriers
    - Obviously there is a cost, as moving data between main and memory, cache memory, and cores is expensive (hundreds of cycles)

Thread Synchronization Objects

- A T (P) can wait for
  - The termination of one or more Ts (Ps)
    - The waiting thread can wait on the thread handle using
      - WaitForSingleObject if it awaits a single thread
      - WaitForMultipleObjects if it awaits more than one thread
  - Obtaining a file lock used for synchronizing file access
  - Reading from a pipe (or socket) that allows the thread to wait for another to write to the pipe (socket)

Already analized ...

Thread Synchronization Objects

- Specific events, using Ts (Ps) synchronization primitives such as
  - Interlocked functions
  - Critical Sections
  - Mutexes
  - Semaphores
  - Events

kernel objects
(they have HANDLEs).
They can be used for
inter-process synchronization

Interlocked Functions

- If we simply need to manipulate signed numbers, interlocked functions will suffice
  - Limited to increment or decrement variables
    - Can not directly solve general mutual exclusion problems
  - Operations take place in the user space
    - No kernel call
    - Easy to use
    - No deadlock risk
    - **Faster** than any other alternative
  - Variables need to be **volatile**

top++

stack[top]=val

Interlocked Functions

Signed volatile object

```
LONG InterlockedIncrement (LONG volatile *lpAddend);
LONG InterlockedIncrement64 (LONGLONG volatile *lpAddend);

LONG InterlockedDecrement (LONG volatile *lpAddend);
LONG InterlockedDecrement64 (LONGLONG volatile *lpAddend);
```

There are 32-bit and 64-bit versions of interlocked functions. 64-bit integer access is not atomic on 32-bit systems

- They increment (decrement) the volatile variable in an atomic way
  - Notice that the resulting value may be changed (by another T or P) before it is used

Interlocked Functions

- Other interlocked functions
  - InterlockedExchange
    - Stores a variable into another and return the original value
  - InterlockedExchangeAdd
    - Adds the second operand to the first
  - InterlockedCompareExchange
  - InterlockedAnd
  - InterlockedOr
  - InterlockedXor
  - InterlockedCompare64Exchange128

See Hart, end of Chapter 8

With 8, 16, 32 and 64-bit versions

Critical Sections

- Critical sections (**CSs**) can only be used to synchronize Ts within a (unique, single) process
  - They are not kernel objects
  - Thus, among the 4 synch objects, are often the most efficient one
    - "Fast mutexes"
  - Apply then to as many application scenarios as possible
- Critical section objects are
  - Initialized, not created
  - Deleted, not closed

Critical Sections

- 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

Critical Sections

```
CRITICAL_SECTION CriticalSection;
```

Object definition

Object initialization
InitializeCriticalSection (&CriticalSection);

```
VOID InitializeCriticalSection (
  LPCRITICAL_SECTION lpcsCriticalSection
);
```

Object deletion
DeleteCriticalSection (&CriticalSection);

```
VOID DeleteCriticalSection (
  LPCRITICAL_SECTION lpcsCriticalSection
);
```

Critical Sections

A thread can enter a CS more than once ("recursive")

Blocks a thread if another thread is in ("owns") the section

```
VOID EnterCriticalSection (
  LPCRITICAL_SECTION lpcsCriticalSection
);
```

Use this API to avoid blocking. TRUE is returned when the CS can be entered

```
BOOL TryEnterCriticalSection (
  LPCRITICAL_SECTION lpcsCriticalSection
);
```

The waiting thread unblocks when the "owning" thread executes LeaveCriticalSection

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

A thread must leave a CS once for every time it entered

Critical Sections and _finally

- Always be certain to leave a CS.
  - How can we make sure a thread leaves a critical section?
  - Use a _finally block
    - Even if someone later modifies your code
    - This technique also works with file locks and the other synchronization objects discussed next

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

Critical Sections

- CSs test the lock in user-space
  - Fast, there is no kernel call
  - Threads wait in kernel space
- Almost always faster than mutexes
  - Factors include number of threads, number of processors, and amount of thread contention

Critical Sections and Spin Locks

- When a CS is owned by a thread and another thread executes the CS the original thread
  - Enters the kernel
  - Blocks until the CS is released
- Even if CS are fast, the entire process may be quite time consuming

Critical Sections and Spin Locks

- Sometimes, it may be beneficial (faster) to use spin-lock variants
  - InitializeCriticalSectionAndSpinCount
  - SetCriticalSectionSpinCount
  - Etc.
- They should be used
  - On multi-core machines (only)
  - When there is high contention among Ts on the CS
  - The CS is hold for only few instructions

Example

This thread code section does not guarantee ME

```
CRITICAL_SECTION cs1, cs2;
volatile DWORD N = 0;
ICS (&cs1); ICS (&cs2);
                ...
DWORD ThreadFunc (...) {
  ECS (&cs1);
  N = N - 2;
  LCS (&cs1);

  ...

  ECS (&cs2);
  N = N + 2;
  LCS (&cs2);
}
```

ICS → InitializeCriticalSection

ECS → EnterCriticalSection

LCS → LeaveCriticalSection

How would you fix it?

Example

This thread code section can cause a deadlock

```
CRITICAL_SECTION cs1, cs2;
volatile DWORD N = 0, M = 0;
ICS (&cs1); ICS (&cs2);
                ...
DWORD ThreadFunc (...) {
  ECS (&cs1); ECS (&cs2);
  N = N - 2; M = M + 2;
  LCS (&cs1); LCS (&cs2);

  ...

  ECS (&cs2); ECS (&cs1);
  N = N + 2; M = M – 2;
  LCS (&cs2); LCS (&cs1);
}
```

ICS → InitializeCriticalSection

ECS → EnterCriticalSection

LCS → LeaveCriticalSection

How would you fix it?
HRU = Hierarchical Resource Usage

Mutexes

- Mutex (mutual exclusion) objects
  - 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
  - Mutexes are recursive
    - A thread can acquire a specific mutex several times without blocking but it must release the mutex the same number of times
    - This feature can be convenient, for example, with nested transactions

Mutexes

- A mutex can be checked (polled) to avoid blocking
- A mutex becomes "abandoned" if its owning thread terminates
  - Abandoned mutex are automatically signaled
  - This feature (not present with CSs) allow safer use of mutexes

- Mutex are
  - Created (with CreateMutex)
  - Waited for (with WaitForSingleObject or WaitForMultipleObjects)
  - Released (with ReleaseMutex)

Already introduced for other uses

Mutexes

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

- It returns a new mutex handle
  - A NULL value indicates a failure
- Parameters
  - lpsa
    - Security attributes (already describe in other API calls)
    - Usually NULL

Mutexes

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

- fInitialOwner is a flag
  - If it is TRUE, it gives the calling thread immediate ownership of the new mutex
  - It is ignored if the mutex already exists
- lpszMutexName is the mutex name
  - It points to a null-terminated pathname
  - Pathnames are case sensitive
  - Mutexes are unnamed if the parameter is NULL

Mutexes

```
BOOL ReleaseMutex (HANDLE hMutex);
```

- It frees a mutex that the calling thread owns
  - Fails if the T does not own it
- If a mutex is abandoned, a wait will return WAIT_ABANDONED_0
  - This is one of the possible return value for the API WaitForMultipleObjects

Mutex Naming

- A mutex can be named if it is to be used by more than one process
  - Mutexes, semaphores, events, memory mapped objects, waitable timers, all processes share the same name space
  - Pay attention to name collisions
    - Name objects carefully
- Don't name a mutex used in a single process

Mutexes

```
HANDLE OpenMutex(
  DWORD desiredAccess,
  BOOL inheritHandle,
  LPCTSTR lpszMutexName
);
```

Google the system call for more details

- It opens an exiting named mutex
  - It allow synch among Ts in different Ps
  - A CreateMutex in one P must precede an OpenMutex in another P
  - Alernatively, all Ps can use CreateMutex
    - CreateMutex will fail if one mutex has already been created

Mutex Naming

- Process interaction with a named mutex
  - Same name space as used for mem maps, …

```
PROCESS 1
...
H = CreateMutex (... “mutexName” ...);
```

```
PROCESS 2
...
H = OpenMutex (... “mutexName” ...);
```

Semaphores

- 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
    - **Not signaled** when the count is zero

Semaphores

- A semaphore must be
  - Created
  - Waited for
    - Ts (Ps) wait in the normal way, using one of the wait functions (WaitForsingleObject or WaitForMultipleObjects)
    - It is just possible to decrement the count by **one**
  - Released
    - When a waiting thread is released, the semaphore's count is incremented by one
    - It is possible to increment the counter by any value up to the maximum value
    - Any thread can release
      - Not restricted to the thread that "acquired" the semaphore

Semaphores

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

- It returns the semaphore handle
- Parameters
  - lpsa
    - Usually NULL for us
  - cSemInitial
    - Is the initial value for the semaphore

Semaphores

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

- cSemMax is the maximum value for the semaphore
  - It must be: `0 <= cSemInitial <= cSemMax`
- lpszSemName is the semaphore name
  - Often NULL

Semaphores

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

- A release operation can increase the counter by any value
  - Again notice that any wait decrease the counter by 1
- Parameters
  - hSemaphore is the semaphore handle

Semaphores

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

- cRealeaseCount is the increment value
  - It must be greater than zero
  - If it would cause the semaphore count to exceed the maximum, the call will return FALSE and the count will remain unchanged

Semaphores

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

- lpPreviousCount is the previous value of the counter
  - The pointer can be NULL if you do not need this value

Example

Notice again that there is no "atomic" wait for multiple semaphore units, but it is possible to release multiple units atomically.

```
WaitForSingleObject (hSem, INFINITE);
WaitForSingleObject (hSem, INFINITE);
...
ReleaseSemaphore (hSem, 2, &previousCount);
```

This is a potential deadlock in a thread function

- Solution
  - Treat the loop on WFSO as a critical section, guarded by a CS or Mutex object
  - A multiple wait semaphore can be created with an event, mutex, and counter

Events

- The additional capability offered by events is that they can release multiple threads from a wait simultaneously when a single event is signaled
- Events can be
  - Signalled using **PulseEvent** or **SetEvent**
  - Reset **automatically** or **manually**
  - This creates four combinations with very different behavior
    - Be careful, there are numerous subtle problems using events

Events

1 T is released

From 1 to n Ts are released

Ts are released in t=[0,∞]

	AutoReset	ManualReset
SetEvent	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.
PulseEvent	Exactly one thread is released, but only if a thread is currently waiting on the event.	All currently waiting threads are areleased. Then the event is automatically reset.

Ts are released in t=[0]

Events

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

- Create a new event object
- Parameters
  - lpsa often NULL
  - fManualReset
    - TRUE for manual-reset event
    - FALSE otherwise

Events

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

- fInitialState
  - The event is initially set to signaled if it is TRUE
- It is possible to use **OpenEvent** to open a named event, possibly created by another process

Events

```
BOOL SetEvent (HANDLE hEvent);

BOOL ResetEvent (HANDLE hEvent);
```

- For set evens
  - A thread signals an event with **SetEvent**
  - If no threads are waiting on the event, the event remains in the signaled state until some thread waits on it and it is immediately released
  - 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

Events

```
BOOL SetEvent (HANDLE hEvent);

BOOL ResetEvent (HANDLE hEvent);
```

- If the event is manually 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

Events

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

- For pulse event
  - **PulseEvent** allows you to release all threads currently waiting on a manual-reset event
    - The event is then automatically reset

Events

- 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

Comparison

	CS	Mutex	Semaphore	Event
Named, Securable Synchronization Object	No	Yes	Yes	Yes
Accessible from Multiple Ps	No	Yes	Yes	Yes
Synchronization	Enter	Wait	Wait	Wait
Release	Leave	Release or owner terminates	Any thread can release	Set or Pulse
Ownership	One T at a time; recursive	One T at a time; recursive.	N/A	N/A
Effect of Release	One waiting T can enter	One waiting T can gain ownership after last release	Multiple Ts can proceed, depending on release count	One or several waiting Ts will proceed after a Set or Pulse

Example: Using multiple primitives

- There are several problems in which two or more synchronization primitives have to be used together
- Example
  - Two processes want to work on a shared memory
    - They may use a memory mapped file
  - They need to synhronize their R/W activity
    - They may use a mutex
  - The writer (producer) need a strategy to let the reader (consumer) know when he has done
    - They may use an event

Events

```
hE=CreateEvent("done",…);
hMut=CreateMutex("m");
hFM=CreateFileMapping(…);
WaitForSingleObject(hMut);
ptr=MapViewOfFile(hFM,…);
... write to shared memory
SetEvent(hE);
UnmapViewOfFile(hFM);
ReleaseMutex(hMut);
CloseHandle(…);
```

P_i

P_j

```
hE=CreateEvent("done",…);
hMut=CreateMutex("m");
hFM=CreateFileMapping(…);
WaitForMultipleObjects(
  [hE, hMult] ,WAIT_ALL, …);
  ptr=MapViewOfFile(hFM,…);
  ... Read shared memory
  UnmapViewOfFile(hFM);
ReleaseMutex(hMut);
CloseHandle(…);
```