Synchronization

Synchronization (Part A)

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Objectives

- To synchronize threads (and processes) in Windows we must understand
  - The various Windows synchronization mechanisms
    - Volatile Variables
    - Interlocked functions
    - Critical Sections
    - Mutexes
    - Semaphores
    - **Events**
  - How to differentiate synchronization object features
  - How to select among them

File locking, WFSO and WFMO are simple form of synchronization

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

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;

i++;
→
register = i
register++
i = register

Interlocked functions **need** volatile variables

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
    - **Stored** into memory after it is modified
  - This has 2 implications
    - Can negatively effect performance
    - Memory? Hug? Which memory?

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 fences" (or "memory barriers")
    - A memory fence assures that the value is moved to main memory
    - A memory fence assures cache coherence
  - All the following synchronization functions may act as memory fences
    - Obviously there is a cost, as moving data between main and memory, cache memory, and cores is expensive (hundreds of cycles)

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**

i++, i--

i= j*k+23
vet[i]=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... (vi);
... use variable vi ...
```

Interlocked Functions

See Hart, end of Chapter 8

- 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

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 synch objects, are often the most efficient one
    - "Fast mutexes"
  - Apply them 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 **try** and **_finally** block
    - Even if someone later modifies your code
    - This technique also works with file locks and the other synchronization objects discussed next

See C++ section for further details

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

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

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

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 **WFSO** or **WFMO**)
  - Released (with ReleaseMutex)

Already introduced with thread essentials

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

- 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

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

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

See next section

- 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

- cSemMax is the maximum value for the semaphore
  - It must be
    - 0 <= cSemInitial <= cSemMax
- lpszSemName is the semaphore name
  - Often NULL, we manipulate it using its handle

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

Semaphores

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

- A release operation can **increase** the counter by **any** value
  - Notice that any wait decrease the counter by **one only**
- Parameters
  - hSemaphore is the semaphore handle

Semaphores

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

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

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 (e.g., ECS & LCS) or a mutex
  - A multiple wait semaphore can be created with an event, mutex, and counter

Exercise

- Write a C application able to manage bank accounts with the following specs
  - All bank accounts, with their current balaces, are defined in an ACCOUNT binary file
  - All operations done on these bank accounts are defined in an OPERATION binary file

ACCOUNT

id, bank account number, last anf first name, balance

```
1 100000 Romano Antonio 1250
2 150000 Fabrizi Aldo 2245
3 200000 Verdi Giacomo 11115
4 250000 Rossi Luigi 13630
```

OPERATION

id, bank account number, …, bank withdrawal or deposit

```
1 100000 Romano Antonio +50
3 200000 Verdi Giacomo +115
1 100000 Romano Antonio +250
1 100000 Romano Antonio -55
3 200000 Verdi Giacomo -1015
```

Exercise

Report 4 solutions:
lock, critical section,
mutexes, semaphores

- The application
  - Receives N parameters on the command line
    - The first parameter is the name of an ACCOUNT file
    - All other parameters indicate the name of OPERATION files
  - Opens the ACCOUNT file, and then run one thread for each OPERATION file
  - Each thread reads one OPERATION file and it performs on the ACCOUNT file the set of operations specified in that file
  - When all OPERATION files have been managed the program must display the final balance for all bank accounts in the ACCOUNT file

Solution

- The presented implementation
  - Includes 4 different solutions, each one adopting a different synchronization mechanism
    - Set the corresponding flag to true (1) to enable the corresponding solution

```
#define FL 1  // File Locking
#define CS 0  // Critical Sections
#define MT 0  // Mutexes
#define SE 0  // Semaphores
```

  - Includes two main data structures
    - The first one to read from file

```
#typedef struct files {
   ...
};
```

Solution

- The second one as a thread parameter

```
typedef struct threads {
  LPTSTR nameAccount;
  HANDLE hAccount;
  LPTSTR nameOperation;
} threads_t;
```

File Locking: Threads share filename and have different file handles

Other synch strategies: Threads share the same file handle

Local OPERATION file name

➢ The main program

- Open the ACCOUNT file
- Create all threads
- Initialize synch primitives
- Wait for all threads

Solution

- Each thread function
  - If file locking is used, open the "unique" ACCOUNT file
  - Open its "personal" OPERATION file
  - Cycle through the following opeartions
    - Read the next operation from the OPERATION file
    - Protect the correct record within the ACCOUNT file
    - Apdate balance (critical section)
    - Unprotect that record