Multi-core Architecture and Programming

Yang Quansheng(杨全胜)
http://www.njyangqs.com/

School of Computer Science & Engineering
Southeast University

Programming with Windows Threads

- Content
  - Windows threads library
  - Using the Windows thread API
  - Synchronization

Windows threads library

- Win32 API is the interface between the kernel and applications. It is a package of system functions that can be called by application
- MFC is Microsoft Foundation Classes, it pack the Win32 API as classes.
- .NET Framework include Common Language Runtime (CLR) and Framework Class Library (FCL). System.Threading name space in .NET Foundation Classes offer the class and interfaces to support the thread.

Programming with Windows Threads

- Content
  - Windows threads library
  - Using the Windows thread API
  - Synchronization

Using the Windows thread API

- Win32* HANDLE type
  - Each Windows object is referenced by HANDLE type variables
    - Pointer to kernel objects
    - Thread, process, file, event, mutex, semaphore, etc.
  - Object creation functions return HANDLE
  - Object controlled through its handle
    - Don't manipulate objects directly

Using the Windows thread API

- **Win32* Thread Creation**

```
static HANDLE CreateThread(
          LPSECURITY_ATTRIBUTES lpsa,
          DWORD dwStackSize,
          LPTHREAD_START_ROUTINE pfnThreadProc,
          void* pvParam,
          DWORD dwCreationFlags,
          DWORD* pdwThreadId ) ;
```

lpsa: **The security attributes for the new thread.**

dwStackSize: **The stack size for the new thread.**

pfnThreadProc **: The thread procedure of the new thread.**

pvParam **: The parameter to be passed to the thread procedure.**

dwCreationFlags **: The creation flags (0 or CREATE_SUSPENDED).**

pdwThreadId **:[out] Address of the DWORD variable that, on success, receives the thread ID of the newly created thread.**

Using the Windows thread API

- LPTHREAD_START_ROUTINE
  - CreateThread() expects pointer to global function
    - Returns DWORD
    - Calling convention WINAPI
    - Single LPVOID (void *) parameter

```
DWORD WINAPI MyThreadStart(LPVOID p);
```

  - Thread begins execution of function

Using the Windows thread API

- Using Explicit Threads
  - Identify portions of code to thread
  - Encapsulate code into function
    - ☞ If code is already a function, a driver function may need to be written to coordinate work of multiple threads
  - Add CreateThread call to assign thread(s) to execute function

Using the Windows thread API

- Manage the threads
  - Set the priority
    - Thread PRI = process PRI + thread relative PRI
    - *Bool SetThreadPriority (HANDL E hPriority , int nPriority)*
  - Suspend and resume the thread
    - If suspend count in thread = 0, thread executed. If it > 0, scheduler will not schedule the thread.
    - *DWORD SuspendThread(HANDLE hThread);*
    - *DWORD ResumeThread(HANDLE hThread);*

Using the Windows thread API

- Manage the threads
  - Waiting for thread
    - *WaitForSingleObject*
    - *WaitForMultipleObjects*
  - Exit thread
    - ExitThread(DWORD dwExitCode)
    - TerminateThread(HANDLE hThread, DWORD dwExitCode)
  - Destroying Threads
    - Frees OS resources
      - Clean-up if done with thread before program completes
    - BOOL CloseHandle(HANDLE hObject);

Using the Windows thread API

- Example: Thread Create

```
#include <stdio.h>
#include <windows.h>

DWORD WINAPI   helloFunc(LPVOID arg) {
                    printf(“Hello Thread”);
                    return 0;
}

Main() {
            HANDLE  hThread =
                             CreateThread(NULL, 0, helloFunc, NULL, 0, NULL);
}
```

What Happens?

Using the Windows thread API

- Example Explained
  - Main thread is process
  - When process goes, all thread goes
  - Need some method of waiting for a thread to finish

Using the Windows thread API

- Waiting for Windows* Thread

```
#include <stdio.h>
#include <windows.h>
BOOL  threadDone = FALSE;

DWORD WINAPI   helloFunc(LPVOID arg) {
                    printf(“Hello Thread”);
                    threadDone = TRUE;
                    return 0;
}

Main() {
            HANDLE  hThread =
                             CreateThread(NULL, 0, helloFunc, NULL, 0, NULL);
      While (!threadDone);   // wasted cycles
}
```

Not a good idea!

Using the Windows thread API

- Waiting for a thread
  - Wait for one object (thread)

```
DWORD WaitForSingleObject(
                          HANDLE hHandle,
                           DWORD dwMilliseconds );
```

  - Call thread waits (blocks) until
    - Time expires
      - Return code use to indicate this
    - Thread exits (handle is signaled)
      - Use INFINITE to wait until thread termination
  - Dose not use CPU cycles

Using the Windows thread API

- Waiting for Many threads
  - Wait for up to 64 objects (threads)

```
DWORD WaitForMultipleObjects(
                          DWORD nCount,
                          const HANDLE* lpHandles,   // array
                          BOOL bWaitAll,   // wait for one or all
                          DWORD dwMilliseconds );
```

- ☞ Wait for all: bWaitAll =TRUE
- ☞ Wait for any: bWaitAll = FALSE
- ☞ Return value is first array index found

Using the Windows thread API

- Notes on WaitFor* Funtions
  - Handle as parameter
  - Used for different types of objects
  - Kernel objects have two states
    - Signaled
    - Non-signaled
  - Behavior is defined by object referred to by handle
    - Thread: signaled means terminated

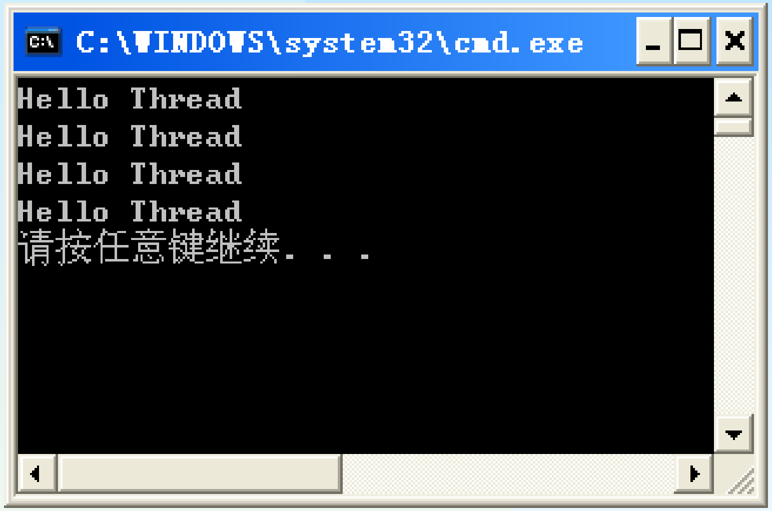
Using the Windows thread API

- Example: Multiple Thread

```
#include <stdio.h>
#include <windows.h>
const int numThread = 4;
DWORD WINAPI   helloFunc(LPVOID arg) {
                    printf(“Hello Thread”);
                    return 0;
}
Main() {
            HANDLE  hThread[numThread];
            for (int i = 0; i < numThread; i++)
               hThread[i] =
                     CreateThread(NULL, 0, helloFunc, NULL, 0, NULL);
      WaitForMultipleObjects(numThreads, hThread,TRUE,INFINITE);
}
```

Using the Windows thread API

- How to print the thread number?



```
Hello Thread
Hello Thread
Hello Thread
Hello Thread
请按任意键继续. . .
```

=>

```
Hello from Thread #0

Hello from Thread #1

Hello from Thread #2

Hello from Thread #3
```

Using the Windows thread API

- Is that right? Why or why not?

```
#include <stdio.h>
#include <windows.h>
const int numThread = 4;
DWORD WINAPI helloFunc(LPVOID pArg) {
        int* p = (int *)pArg;
        int mynum = *p;
        printf("Hello Thread %d\n",mynum);
        return 0;
}
main() {
        HANDLE hThread[numThreads];
        for (int i = 0; i < numThreads; i++)
          hThread[i] =
                CreateThread(NULL, 0, helloFunc, &i, 0, NULL );
}
```

Using the Windows thread API

- Hello threads Timeline

Time	*main*	*Thread 0*	*Thread 1*
T_0	i = 0	---	----
T_1	create(&i)	---	---
T_2	i++ (i == 1)	launch	---
T_3	create(&i)	p = pArg	---

T_4	i++ (i == 2)	myNum = *p
		myNum = 2
			launch

T_5	wait	print(2)	p = pArg

T_6	wait	exit	myNum = *p
			myNum = 2

Programming with Windows Threads

- Content
  - Windows threads library
  - Using the Windows thread API
  - Synchronization

Synchronization

- Race Conditions
  - Concurrent access of same variable by multiple threads
    - Read/Write conflict
    - Write/Write conflict
  - Most common error in concurrent programs
  - May not be apparent at all times

Synchronization

- How to Avoid Data Races
  - Scope variables to be local to threads
    - Variables declared within threaded functions
    - Allocate on thread's stack
    - TLS (Thread Local Storage)
  - Control shared access with critical regions
    - Mutual exclusion and synchronization
    - Lock, semaphore, event, critical section, mutex…

Synchronization

- Solution – “local” Storage

```
DWORD WINAPI helloFunc(LPVOID pArg)
{
        int mynum = *((int *)pArg);
        printf("Hello Thread %d\n",mynum);
        return 0;
}

// from main
        int tNum[numThreads];
        for (int i = 0; i < numThreads; i++){
                tNum[i] = i;
                hThread[i] =
                  CreateThread(NULL, 0, helloFunc, &tNum[i], 0, NULL );
        }
        WaitForMultipleObjects(numThreads, hThread, TRUE, INFINITE);
```

Synchronization

- Win32* Mutexes
  - Kernel object reference by handle
  - “Signaled” when available
  - Operations:
    - CreateMutex(…) // create new
    - WaitForSingleObject // wait & lock
    - ReleaseMutex(…) // unlock
  - Available between processes

Synchronization

- Win32* Critical Section
  - Lightweight, intra-process only mutex
  - Most useful and most used
  - New type
    - CRITICAL_SECTION cs;
  - Create and destroy operations
    - InitializeCriticalSection(&cs)
    - DeleteCriticalSection(&cs);

Synchronization

- Win32* Critical Section

```
CRITICAL_SECTION cs ;
```

Attempt to enter protected code

```
EnterCriticalSection(&cs)
```

- ☞ Blocks if another thread is in critical section
- ☞ Returns when no thread is in critical section

Upon exit of critical section

```
LeaveCriticalSection(&cs)
```

- ☞ *Must* be from obtaining thread

Synchronization

- Example: Computing Pi

```
static long num_steps=1000000;
double step, pi;

void main()
{  int i;
   double x, sum = 0.0;

   step = 1.0/(double) num_steps;
   for (i=0; i< num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0 + x*x);
   }
   pi = step * sum;
   printf("Pi = %12.9f\n",pi);
}
```

Parallelize the numerical integration code using Win32 Threads

How can the loop iterations be divided among the threads?

What variables can be local?

What variables need to be visible to all threads?

Synchronization

- Win32* Semaphores
  - Synchronization object that keeps a count
    - Represents the number of available resources
    - Formalized by Edsger Dijkstra (1968)
  - Two operations on semaphores
    - Wait [P(s)]: Thread waits until s > 0, then s = s-1
    - Post [V(s)]: s = s + 1
  - Semaphore is in signaled state if count > 0

Synchronization

Win32* Semaphore Creation

```
HANDLE CreateSemaphore(
        LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
        LONG lInitialCount,   // Initial count for the semaphore object
        LONG lMaximumCount, // Maximum value for count
        LPCTSTR lpName );  // text name for object
```

- Value of IMaximumCount must be greater than zero.
- Value of lInitialCount must be
  - Greater than or equal to zero
  - Less than or equal to IMaximumCount
  - Cannot go outside of range

Synchronization

- Wait and Post Operations
  - Use WaitForSingleObject to wait on semaphore
    - ☞ If count is == 0, thread waits
    - ☞ Decrement count by 1 when count > 0
  - Increment semaphore (Post operation)

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

    - ☞ Increase semaphore count by cReleaseCount
    - ☞ Returns the previous count through lpPreviousCount

Synchronization

- Semaphore Uses
  - Control access to data structures of limited size
    - Queues, stacks, deques
    - Use count to enumerate available elements
  - Control access to finite number of resourse
    - File descriptors, tape drives…
  - Throttle number of active threads within a region
  - Binary semaphore [0,1] can act as mutex

Synchronization

- Semaphore Cautions
  - No ownership of semaphore
  - Any thread can release a semaphore, not just the last thread that waits
    - Use good programming practice to avoid this
  - No concept of abandoned semaphore
    - If thread terminates before post, semaphore increment may be lost
    - Deadlock

Synchronization

- Example: Semaphore as Mutex
  - Main thread opens input file, waits for thread termination
  - Thread will
    - Read line from input file
    - Count all five_letter words in line

Synchronization

- Example: Main

```
HANDLE hSem1, hSem2;
FILE *fd;
int fiveLetterCount = 0;
```

```
main()
{ HANDLE hThread[NUMTHREAD];
  hSem1 = CreateSemaphore(NULL,1,1,NULL); // Binary semphore
  hSem2 = CreateSemaphore(NULL,1,1,NULL); // Binary semphore
  fd = fopen(“InFile”, “r”);   // open file for read
  for (int i = 0; i < NUMTHREAD; i++)
     hThread[i] = CreateThread(NULL, 0,CountFives,NULL,0,NULL);
  WaitForMultipleObjects(NUMTHREAD, hThread, TRUE, INFINITE);
  fclose(fd);
  printf(“Numbers of five letter words is %d\n”, fiveLetterCount);
}
```

Synchronization

- Example: Semaphores

```
DWORD WINAPI CountFives(LPVOID arg) {
    BOOL bDone = FALSE;
    char  inLine[132]; int lCount = 0;
    while (!bDone)
    {
        WaitForSingleObject(hSem1, INFINITE); // access to input
          bDone = (GetNextLine(fd, inLine) == EOF);
        ReleaseSemaphore(hSem1, 1, NULL);
        if (!bDone)
          if (lCount = GetFiveLetterWordCount(inLine)) {
            WaitForSingleObject(hSem2, INFINITE); // update global
              fiveLetterCount += lCount;
            ReleaseSemaphore(hSem2, 1, NULL);
         }
     }
}
```

Synchronization

- Win32* Events
  - Used to signal other threads that some event has occurred
    - Data is available, message is ready
  - Threads wait for signal with `WaitFor*` function
  - Two kinds of events
    - Auto-reset
    - Manual-reset

Synchronization

- Types of Events

Auto-reset

- Event stays signaled until any one thread waits and is released
  - If no threads waiting, state stays signaled
  - Once thread is released, state reset to non-signaled

Manual-reset

- Event remains signaled until reset by API call
  - Multiple threads can wait and be released
  - Threads not originally waiting may start wait and be released

Caution: Be careful when using `WaitForMultipleObjects` to wait for ALL events

Synchronization

```
HANDLE CreateEvent(
        LPSECURITY_ATTRIBUTES lpEventAttributes,
        BOOL, bManualReset
        BOOL bInitialState,
        LPCTSTR lpName );
```

- Set *bManualReset* to TRUE for manual-reset event; FALSE for Auto reset event.
- Set *bInitialState* to TRUE for event to begin in signaled state; FALSE to begin unsignaled.

Synchronization

- Event set and Reset
  - Set an event to signaled state

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

  - Reset manual-reset event

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

  - Pulse event

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