Operating Systems: Lecture 4

Basic Multithreaded Programming

Jinwoo Kim

jwkim@jjay.cuny.edu

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Overview

- What is multithreaded programming?
- Why do you need it?
- Basics of multithreaded programming and example codes

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Basic Concept

- The Operating System that you use was probably written with the idea of threads, or processes in mind
  - You can run programs of your choosing
- A process is a running program, and a program is a set of instructions that the machine can understand
- You can run more than one process at a time
  - When you open up separate programs, they run as processes in their own separate memory space
  - Each program running has a chunk of memory to which it and only it can use
  - That's how many programs can run without changing variables and states of other running programs

Basic Concept (Continued)

- Imagine for a second what it would be like to run a program in the same memory space as another running process

- This process running inside of another processes' memory space is called a “Thread”
  - A thread is a path of execution
  - A process requires at least one thread but it may contain more than one threads
  - If the process is closed, all the threads in the process are killed automatically

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Benefits of Threads

- Efficient
  - The multithreaded application uses CPU 100% effectively
- Economical
  - When we create a process, it will take memory space
  - Multithreaded application shares the same process memory space
  - Every thread contains stack
  - So the thread takes up less memory usage compared to a process

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

How does a Thread work?

- The Operating System has a scheduler for each thread (process) that is currently running
- It divides up time slices for each of them which are executed in the order that the Operating System seems fit
- It simply runs each one in some arbitrary order for a set number of milliseconds and then switches between them constantly

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Is it fast?

- Of course! One way to think about it is like this: The more processes that your program has running, the more time that your program can get from the system

- The switches from one thread to another (or from one process to another) happens so quickly that the entire system seems to be doing many different things at once!

Is there a lot of overhead involved?

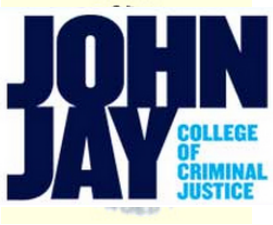
- Not much
  - Compared to multiple processes
- We will see from example code
  - A simple application that creates 3 threads and runs them simultaneously

Example code

```
// First, always include <windows.h> for all the Windows specific thread
information
 #include <windows.h>
 #include <iostream.h>
 #define MAX_THREADS 3

// Prototypes are good and handy, but not necessary in this example.
 // These three functions are run by each of our three threads
 // Please note how the functions are declared:
 // In Windows, thread functions MUST be declared like this:
 // DWORD WINAPI <name>(LPVOID)
 // In short,
 // Return value *must* be DWORD WINAPI
 // And the parameter must be LPVOID

 DWORD WINAPI genericThreadFunc1(LPVOID);
 DWORD WINAPI printString(LPVOID);
 DWORD WINAPI printNumber(LPVOID);
```



Example code (Continued)

```
 // We need an array of Handles to threads
HANDLE hThreads[MAX_THREADS];

// ...an array of thread id's
DWORD  id[MAX_THREADS];

// And a waiter (which I'll explain later)
DWORD  waiter;
```

Example code (Continued)

```
// Here are the three functions that are defined.
// They do trivial things and should be mostly self explanatory.

DWORD WINAPI genericThreadFunc1(LPVOID n)
{
      cout << "Thread started (genericThreadFunc1)..." << endl;
      for(int i = 0; i < 100; i++) {
            cout << "threadFunc1 says: " << i << endl;
      }
      cout << "...(genericThreadFunc1) Thread terminating." << endl;
      return (DWORD)n;
}
```

Example code (Continued)

```
DWORD WINAPI printString(LPVOID n)
 {
      cout << "Thread started (printString)..." << endl;

 // NOTE: In the next line, we make a pointer and cast what was passed in.
 // This is how you use the LPVOID parameters passed into the
 // CreateThread call (below).
      char* str = (char*)n;
      for(int i = 0; i < 50; i++) {
           cout << "printString says: " << str << endl;
      }
      cout << "...(printString) Thread terminating." << endl;
      return (DWORD)n;
 }
```

Example code (Continued)

```
DWORD WINAPI printNumber(LPVOID n)
{
        cout << "Thread started (printNumber)..." << endl;
        int num = (int)n;
        for (int i = num; i < (num + 100); i++) {
                cout << "printNumber says: " << i << endl;
        }
        cout << "...(printHello) Thread terminating." << endl;
        return (DWORD)n;
}
```

Example code (Continued)

```
// Get ready, because here's where all the *REAL* magic happens
 int main(int argc, char* argv[ ])
 {
       int CONSTANT = 2000;
       char myString[20];
       strcpy(myString,"Threads are Easy!");

// Here is where we call the CreateThread Windows API Function that actually
// creates and begins execution of a thread.
//  Please read your help files for what each parameter does on
//  your Operating system.
```

Example code (Continued)

```
// Here's some basics:
 // Parameter 0: Lookup
 // Parameter 1: Stack size (0 is default which means 1MB)
 // Parameter 2: The function to run with this thread
 // Parameter 3: Any parameter that you want to pass to the thread function
 // Parameter 4: Lookup
 // Parameter 5: Once thread is created, an id is put in this variable passed in

hThreads[0] = CreateThread(NULL,0,genericThreadFunc1,(LPVOID)0,NULL,&id[0]);
hThreads[1] = CreateThread(NULL,0,printString,(LPVOID)myString,NULL,&id[1]);
hThreads[2] = CreateThread(NULL,0,printNumber,(LPVOID)CONSTANT,NULL,&id[2]);
```

Example code (Continued)

```
// Now that all three threads are created and running, we need to stop
// the primary thread (which is this program itself - Remember that once
// "main" returns, our program exits)
// so that our threads have time to finish.  To do this, we do what is
// called "Blocking".
// We're going to make main just stop and wait until all three threads
// are done.
// This is done easily with the next line of code.  Please read the help
// file about the specific API call "WaitForMultipleObjects".


waiter = WaitForMultipleObjects(MAX_THREADS, hThreads, TRUE, INFINITE);
```

Example code (Continued)

```
// After all three threads have finished their task, "main" resumes and
// we're now ready to close the handles of the threads.  This is just a
// bit of clean up work.
// Use the CloseHandle (API) function to do this.  (Look it up in the
// help files as well)

      for(int i = 0; i < MAX_THREADS; i++) {
            CloseHandle(hThreads[i]);
      }

      return 0;
}
```

Example code using Windows API in the Textbook

```
// First, always include <windows.h> for all the Windows specific thread
// information
 #include <windows.h>
 #include <stdio.h>

 DWORD Sum; /* data is shared by the thread(s) */
 /* the thread runs in this separate function */

 DWORD WINAPI Summation(LPVOID Param)
 {
      DWORD Upper =  *(DWORD*)Param;
      for (DWORD i=0; i <= Upper; i++)
            Sum += i;
      return 0;
 }
```

Example code using Windows API in the Textbook (Continued)

```
int main(int argc, char* argv[ ])
{
      DWORD ThreadId;
      HANDLE ThreadHandle;
      int Param;
       /* perform some basic error checking */
      if (argc != 2) {
           fprintf(stderr, “An integer parameter is required\n”);
           return -1;
      }

      Param = atoi(argv[1]);
      if (Param < 0) {
           fprintf(stderr, “An integer >= 0 is required\n”);
           return -1;
      }
```

Example code using Windows API in the Textbook (Continued)

```
    /* create the thread */
    ThreadHandle = CreateThread(
          NULL, // default security attributes
          0, // default stack size
          Summation, // thread function
          &Param, // parameter to thread function
          0, // default creation flags
          &ThreadId);  // returns the thread identifier

     if (ThreadHandle != NULL) {
          // now wait for the thread to finish
          WaitForSingleObject(ThreadHandle, INFINITE);
          // close the thread handle
          CloseHandle(ThreadHandle);
          printf("sum = %d\n", Sum);
       }
}
```