Lab 06 - Windows Processes and Threads

Objectives

1. Learn how to create and work with processes in Windows
2. Learn how to create and work with threads in Windows

Pre-reading

The following material contains commands and/or other information that could be essential to completing this and future labs. Students are strongly encouraged to take a look at the material below *before* coming to the lab, since knowing this material before you start will likely make task of completing the lab much faster and easier.

Creating a Process

One process can create another process by calling the Win32 API **`CreateProcess()`** function (which, in turn, calls the Native API functions **`NtCreateProcess()`** and **`NtCreateThread()`**). Whenever a process is created, the calling process is directing the Executive to perform a large amount of work: to set up a new address space and allocate resources to the process and to create a new base thread. Once the new process has been created, the old process will continue using its old address space while the new one operates in a new address space with a new base thread. This means that there can be many different options for creating the process, so the **`CreateProcess()`** function has many parameters, some of which can be quite complex. After the Executive has created the new process, it will return a handle for the child process and a handle for the base thread in the process.

Following is a copy of the function prototype for **`CreateProcess`** taken verbatim form the Win32 API reference manual – found in this link below. (You will find the Microsoft Developers Network -- MSDN – a very useful website for on line documentation)

http://msdn2.microsoft.com/en-us/library/ms682425.aspx

```
BOOL WINAPI CreateProcess(
  __in          LPCTSTR lpApplicationName,
  __in_out      LPTSTR lpCommandLine,
  __in          LPSECURITY_ATTRIBUTES lpProcessAttributes,
  __in          LPSECURITY_ATTRIBUTES lpThreadAttributes,
  __in          BOOL bInheritHandles,
  __in          DWORD dwCreationFlags,
  __in          LPVOID lpEnvironment,
  __in          LPCTSTR lpCurrentDirectory,
  __in          LPSTARTUPINFO lpStartupInfo,
  __out         LPPROCESS_INFORMATION lpProcessInformation
);
```

First you should notice that these are not standard C types, they are defined in the **windows.h** file (see the "Simple Types" entry in the on line Visual C++ documentation), many of which are implemented as standard C types. This level of indirection in name types creates an abstract interface that Windows NT implementers can use as they wish.

The ten parameters in **`CreateProcess()`** provide great flexibility to the programmer, but for the simple case, most of these can be **`NULL`**. In this lab, you will use a relatively simple set of parameters because we are only dealing with a console application (as in a program run from the command prompt, **`cmd.exe`**).

The first two parameters, **`IpApplicationName`** and **`IpCommandLine`**, provide two different ways to define the name of the file that is to be executed by the process's base thread. The reasons for providing both ways are related to compatibility with multiple subsystems. This is an example of how the pressures of commercial products cause software to have compromises in its design. **`IpApplicationName`** is a string representation of the name of the file to be executed, and **`IpCommandLine`** is a string representation of the C-style command line to run the process from **`cmd.exe`**. A set of rules explains the conditions under which each name should be used. That is, if you pass a **`NULL`** for **`IpApplicationName`** and a command line string for **`IpCommandLine`**, your code will be consistent with most C environments (including UNIX environments).

For example, suppose you want to create a process and have it run the notepad program on a file name "temp.txt". You set **`IpCommandLine`** to point at the string with the filename for notepad as: (note that your PC may have a different path)

```
include string.h
...
strcpy( IpCommandLine, "C:\\WINNT\\SYSTEM32\\NOTEPAD.EXE temp.txt" );
CreateProcess(NULL, IpCommandLine, .. );
```

The two security attribute parameters, **`IpProcessAttributes`** and **`IpThreadAttributes`** default to **`NULL`** (we won't be using this feature in this lab). For the inheritance attribute "**`bInheritHandles`**", we set to **`FALSE`**. The previous **`CreateProcess()`** call can now be refined to:

```
CreateProcess(NULL, IpCommandLine, NULL, NULL, FALSE, …);
```

Next, the **`dwCreationFlags`** parameter is used to control the priority and other aspects of the new process. The child process can be created with any of the four priority classes by passing one of the following:

- **`NORMAL_PRIORITY_CLASS`**: default priority
- **`IDLE_PRIORITY_CLASS`**: run if there are no other threads to run
- **`HIGH_PRIORITY_CLASS`**: usually time critical, pre-empt idle and normal threads
- **`REALTIME_PRIORITY_CLASS`**: these threads pre-empt all other classes of threads

Other flags can be passed using the **`dwCreationFlags`** parameter (by logically OR'ing their values together). See the MSDN documentation for all of the possible values.

One flag that you will find useful for this exercise is the **CREATE_NEW_CONSOLE** flag, which causes the new process to be created with its own console (cmd.exe) window. If the goal was to create a new child process with a high-priority class and its own console window, the **CreateProcess()** call would be refined as follows.

```
CreateProcess(NULL, IpCommandLine, NULL, NULL, FALSE,
              HIGH_PRIORITY_CLASS | CREATE_NEW_CONSOLE, …);
```

The **IpEnvironment** parameter is used to pass a new block of environment variable to the child. If the parameter is **NULL**, the child uses the same environment variable as the parent. If it is not **NULL**, it must point to a **NULL**-terminated block of **NULL**-terminated strings, each of the form **Name=value**. If you use a non-**NULL** value for this parameter, be careful to distinguish between ordinary 8-bit ASCII and 16 bit Unicode characters.

The **IpCurrentDirectory** string specifies the current drive and full pathname of the current directory in which the child should execute. The drive is prefixed to the full pathname using conventional MS-DOS notation. A **NULL** value causes the child to execute using the same current drive and directory as the parent. Using a **NULL** for the current drive and directory refines the sample **CreateProcess()** call to the following.

```
CreateProcess(NULL, IpCommandLine, NULL, NULL, FALSE,
              HIGH_PRIORITY_CLASS | CREATE_NEW_CONSOLE, NULL, NULL, …);
```

Ok, now a parameter we need to initialize, **IpStartupInfo**. The startup information is a C struct used to pass a set of miscellaneous parameters regarding window characteristics and redirection information for the console and keyboard. The information is stored in a structure of the following type:

```
typedef struct _STARTUPINFO {
      DWORD cb;
      LPTSTR lpReserved;
      LPTSTR lpDesktop;
      LPTSTR lpTitle;
      DWORD dwX;
      DWORD dwY;
      DWORD dwXSize;
      DWORD dwYSize;
      DWORD dwXCountChars;
      DWORD dwYCountChars;
      DWORD dwFillAttribute;
      DWORD dwFlags;
      WORD wShowWindow;
      WORD cbReserved2;
      LPBYTE lpReserved2;
      HANDLE hStdInput;
      HANDLE hStdOutput;
      HANDLE hStdError;
} STARTUPINFO,
 *LPSTARTUPINFO;
```

An instance of this data structure must be created in the calling program. Then its address must be passed as a parameter in **`CreateProcess()`**. Again, we will not need to fill in this structure if we are not creating a window. You can find the details on line here:

http://msdn2.microsoft.com/en-us/library/ms686331.aspx

Warning: **`CreateProcess()`** does not presume to know the length of the structure whose address is passed as the **`IpStartupInfo`** parameter. Be sure to initialize the size field in the **`STARUPINFO`** structure before passing it to **`CreateProcess()`**.

The last parameter is the **`IpProcessInformation`** parameter, which points to another Win32 structure that must be supplied by the calling program, as follows.

```
typedef struct _PROCESS_INFORMATION {
      HANDLE hProcess;
      HANDLE hThread;
      DWORD dwProcessId;
      DWORD dwThreadId;
} PROCESS_INFORMATION,
 *LPPROCESS_INFORMATION;
```

There is no length field in this structure, so the calling process simply needs to allocate an instance of **`PROCESS_INFORMATION`** and pass a pointer to the instance into **`CreateProcess()`**. When the function returns, the four parameters will have been filled in to reflect.

- the handle of the newly created process (the **`hProcess`** field)
- the handle of the base thread (**`hThread`**)
- a global process identifier (**`dwPrcessId`**) that can be used by another process to locate the newly created process
- a global thread identifier (**`dwThreadId`**)

`CreateProcess()` allocates a handle to the process and the thread when it creates them; they are returned in the **`PROCESS_INFORMATION`** data structure. To avoid the possibility of creating a “handle leak” (whereby the handle is implicitly allocated to the parent process), you must explicitly deallocate each handle, for example by calling **`CloseHandle()`**. Of course, when a process terminates, all of its handles (kept in a list in the process’s descriptor) will be automatically released. Even so, since a handle is a system resource, you should make it a practice to always explicitly close handles as soon as you are through using them.

Notice that **`CreateProcess()`** returns zero if it fails and nonzero otherwise Win32 provides an integer error number for the last error that occurred. It is good practice to check for an error on a Win32 API call and then to check the last error if the call failed.

Here is the code for a full call to create a process.

```
char* path = "C:\\WINNT\\SYSTEM32\\NOTEPAD.EXE temp.txt";

STARTUPINFO si;
ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);

PROCESS_INFORMATION pi;
ZeroMemory( &pi, sizeof(pi) );

if( CreateProcess(

      NULL,    // Application name
      path,    // Application arguments
      NULL,    // Process handle not inheritable
      NULL,    // Thread handle not inheritable
      FALSE,   // Set handle inheritance
      CREATE_NEW_CONSOLE ,  // Creation flags
      NULL,    // Use parent's environment block
      NULL,    // Use parent's starting directory
      &si,     // Pointer to STARTUPINFO structure
      &pi )    // Pointer to PROCESS_INFORMATION structure
      == FALSE ){

            // most common problem is that the program was not found.
            printf( "could not create process with path = (%s)\n", path );

            // send error information to stderr stream
            fprintf(stderr, “CreateProcess() failed on error %d\n”,
GetLastError());

 }

// Wait until child process exits. (look this up on MSDN)
WaitForSingleObject( pi.hProcess, INFINITE );

// Close process and thread handles.
CloseHandle( pi.hProcess );
CloseHandle( pi.hThread );
```

Creating a Thread

You can also create additional threads in the current process using the **`CreateThread()`** Win32 API function (which uses the **`NtCreateThread`** Native API call). Since each thread represents an independent computation of a shared program on shared data, all within the context of an existing process address space, creating a thread requires that the programmer supply information relating to the execution environment for this thread while presuming all of the process-specific information.

The best way to discuss the procedure for creating a thread is to first look at the function prototype – found here:

http://msdn2.microsoft.com/en-us/library/ms682453.aspx

```
HANDLE WINAPI CreateThread(
  __in   LPSECURITY_ATTRIBUTES lpThreadAttributes,
  __in   SIZE_T dwStackSize,
  __in   LPTHREAD_START_ROUTINE lpStartAddress,
  __in   LPVOID lpParameter,
  __in   DWORD dwCreationFlags,
  __out  LPDWORD lpThreadId
);
```

This prototype uses six parameters to describe the characteristics of the new thread. In this case, the function will create only one handle (the one for the thread) and return it as the result of the call. Again, this means that a system resource is implicitly allocated on a successful call to **`CreateThread()`**, so the programmer is obliged to explicitly close the handle when it is no longer being used. To see an example of how **`CreateThread()`** is called, let's look at these parameters in turn.

`lpThreadAttributes` is the same as the security attribute parameter used in **`CreateProcess`**, except of course there is a value only for the thread. For the simple case, it should be set to **`NULL`**. As mentioned earlier, the use of the security attributes is beyond the scope of this lab.

`dwStackSize` parameter gives the programmer a chance to set the size of the stack, though usually one would just use the default, signified by setting the value of the parameter to zero. Since each thread executes independently of the other threads in the process, each has its own stack. Again, this is an optional parameter.

`lpStartAddress` is the address of an entry point for a function that has a prototype of the form:

```
DWORD WINAPI threadWork(LPVOID);
```

With a process, it was necessary to provide the name of an executable file. However, for a thread, it is only necessary to provide the OS with an address in the current address space where the new thread should begin to execute. In conventional programming languages (such as C), it is generally not possible for a computation to simply start off in the middle of some procedure. A branch to a new logical context is handled by bundling the new code in a procedure and then calling the procedure at it entry point.

That is, in a language that checks the type of a called entry point compared to the function call (as is done in C++ and ANSI C), there must be a function prototype before an entry point address can be used as a parameter. Of course, this means that there must also be a function to implement the prototype, the function that the new thread will begin executing after it is created. So, in the above prototype, our thread will start running the procedure called, **`threadWork()`**.

`IpParameter` is how we pass arguments to the procedure called **`threadWork()`**. In order to send any data type we wish, the prototype is expecting a void pointer. The type **`LPVOID`** is defined as **`void*`**, so the compiler is happy, but the programmer must take care to cast it carefully when inside the **`threadWork()`** procedure's body. Pay attention, this parameter is an address, not a value. Now the example call will take the following form:

```
int theArg = 99;
CreateThread(NULL, 0, threadWork, &theArg, …);
```

dwCreationFlags is the parameter used to control the way the new thread is created. Currently, there is only one possible flag value that can be passed for this parameter: **CREATE_SUSPENDED**. This value will cause the new thread to be created but to be suspended until another thread executes on the new thread. So, calling:

```
ResumeThread(targetThreadHandle);
```

Where **targetThreadHandle** is the handle of the new thread being created. Or, we can just use the default value of zero. This will cause the thread to be active when it is created, and saves the need to start each thread. Adding this parameter, the example call becomes:

```
CreateThread(NULL, o, threadWork, &theArg, 0, …);
```

LpThreadId is the last parameter. It is a pointer to a system-wide thread identification of type **DWORD**. This is analogous to the **dwThreadID** field in the **PROCESS_INFORMATION** record returned by **CreateProcess()**. So, looking back at the **CreateThread()** prototype above, we see that **LpThreadId** is an output – it will be filled in for us. Now, we see a full example on how to start a thread:

```
DWORD threadId;
HANDLE handle = CreateThread(NULL,0,threadWork,&theArg,0,&threadId);
if ( handle == NULL ){
      printf("Failed to create thread");
      ExitProcess(0);
} else {
      printf("handle   = %i \n", handle);
      printf("threadId = %i \n", threadId);
}
```

Note that the **CreateThread()** can fail, and you should test if the returned **HANDLE** is valid. Also, you can see that **threadId** parameter has been returned with a system wide ID value.

The **threadWork()** function, the entry point for the thread, looks like this:

```
DWORD WINAPI threadWork( LPVOID param ) {
      // cast param to the correct type, in this case an integer, but
      // depends on the fourth parameter given to the CreateThread function
      int arg = *((int*)param);
      ...          // do the work
      return 0;    // the thread finishes
}
```

Note: Before we get into programming something with threads, let's look at some complications in windows programming. A full discussion is available here:

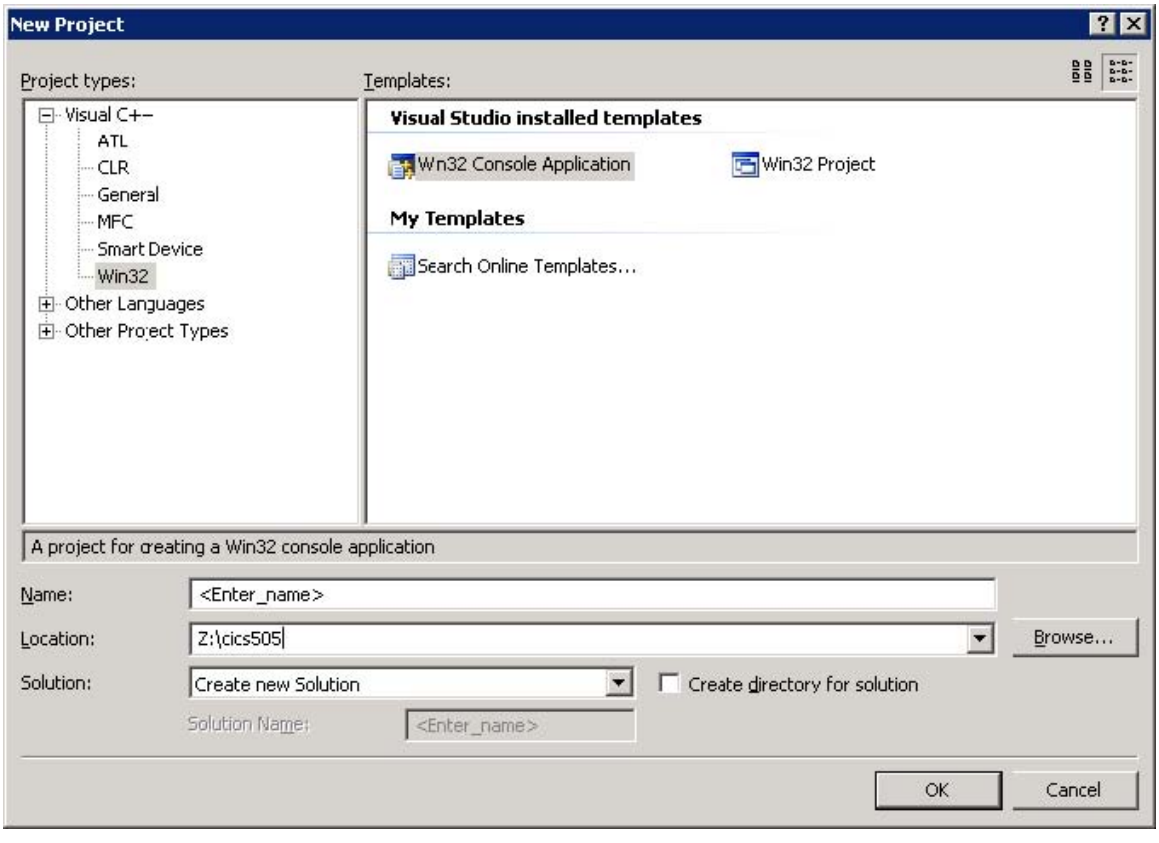
http://support.microsoft.com/kb/104641

Basically put, the C runtime library was derived in a UNIX context in which there is no distinction between processes and threads. In the Windows NT context, many threads can be executing in a single address space. All threads have access to all the variables not in the thread's stack. This can be useful, and is a simple way for threads to communicate, but creates possible race conditions. Such as a use of the global variable called **`errno`**, which is set by the runtime functions if there was an error in the last call made. As we saw in full code example above (page 5), we can retrieve this error number and print it. This is fine if there is a single thread running, but what if there are many threads calling these C runtime functions? Worse, there are memory leakage issues using **`CreateThread()`** and C runtime functions (see link above), so in Visual Studio C++ you can use **`_beginthreadex()`** and **`_endthreadex()`** if needed. To avoid these conditions, **`_beginthreadex()`** creates copies of the C standard library global variables for each thread before starting the new thread. You can complete this lab without these functions if you don't make use of the C runtime functions. Here is an example of how to use **`_beginthreadex()`**:

```
//
// notice the return type is not DWORD WINAPI
// when using _beginthreadex()
//
unsigned __stdcall threadWork( LPVOID param ){

      // cast this pointer with care!
      int data = *((int*)param);
      ...
      // exit the thread, not needed when using CreateThread();
      _endthreadex(0);
      return 0;
}

//
// start the thread...
//
HANDLE handle = (HANDLE)_beginthreadex(NULL, 0, threadWork,&theArg,0,&threadId);

// test if thread handle is valid
if( handles == NULL ){
      printf("Could not CreateThread\n");
      ExitThread(ERROR);
}
else {
      printf("handle[%i] threadId [%i]\n\n", handle, threadId);
}
```

Short Introduction to Visual Studio

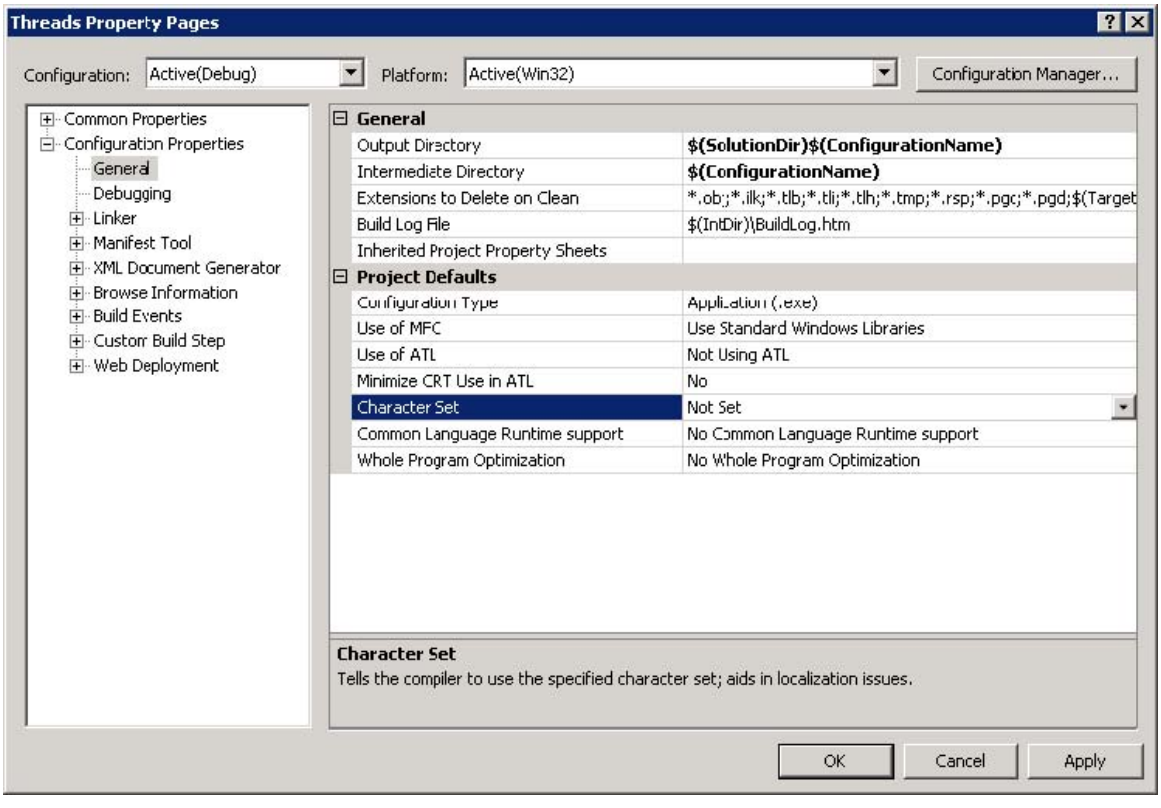
In this section you are going to see how to create a new project in **Visual Studio**. To start a new project you choose **File** → **New** → **Project**. The following dialog box will open:



You must select **Win32** project type and the **Win32 Console Application** template. Enter a project name and choose the location of the project. Make sure the **Create directory for solution** checkbox is unchecked.

On the window that appears next click on **Application Settings** and then uncheck **Precompiled header** and check **Empty project** options. Then click on the **Finish** button and an empty project will be created for you.

For some of the examples in this lab to work as they are presented you must change an additional project setting: you must configure the project so that it is not using the Unicode character set. For that you must click on the project name in the **Solution Explorer** and select **Properties** from the popup-menu. A new window will open and here you must first select the **Configuration Properties/General** option in the tree on the left and then change the **Character Set** property to have the value **Not Set**. After you do this the window should look similar to the following image:



Now we can add a new source file to the project. To do that go to the **Project → Add New** Item menu and add a new C++ file to the project. A new empty file is created and added to the project. This is the place where you can start typing your code.

Procedure

Part A

For Part A you must write a very simplified mini-shell for windows. In one of the previous labs you dealt with process creation under Linux. Now we'll do a simplified version of that to demonstrate how to launch an application from our code in windows. Instead of reading the commands from the console, as you would do in case of a normal shell, you must read them from a file and execute each one in turn. You don't have to worry about searching the PATH for the files to execute, you may use the full path in the text file.

You application should work as follows:

```
> launch.exe list.txt
```

where *list.txt* is a text file with applications and arguments to launch (one per line):

```
C:\WINNT\SYSTEM32\NOTEPAD.EXE test.txt
C:\Program Files\Mozilla\firefox.exe http://www.google.com
...
```

Your program (launch.exe) will launch in turn each application specified in the list.txt file and wait for it to finish. After it finished it launches the next one, and so on, until the end of the file.

The marker will use a file with applications that are valid for that machine. Even though you must submit your test file, don't worry about the paths existing on a given machine.

Part B

In this part we will write a program (separate from launch.exe) that creates N threads that race to get to a give target value T. Each thread is given two random integer values: an increment and a sleep time. Each thread has its own counter that it will increment with the given increment value after which will sleep for the given sleep time. The first thread that reaches the target number T will be the winner and will have to print a message notifying that it has won after which it must terminate. All the other threads must print a message notifying that they have lost and terminate.

Example:

```
> threads.exe 5 100
```

This will create 5 threads that are racing from 0 to 100.

There should be only one thread that reaches the target value first, so only one line with the winning message should be seen in the output. (Note that two threads could run at the same time because we will not worry about synchronization between threads here. We consider that unlikely for now… we will learn about synchronization primitives in a later lab).

Deliverables

Part A and Part B should be done as separate projects in Visual Studio.

Submit just the source and executable files for Part A and Part B.