OPERATING SYSTEMS PROJECT ASSIGNMENT INSTRUCTIONS

Background: Creating a Process

The creation of a process or thread is an operating specific task. This discussion provides background on how to create them using the Win32 API.

A process can create another process by calling the appropriate functions in the Win32 API, specifically CreateProcess (which calls the native functions NtCreateProcess and NtCreateThread). Creating a process requires the OS to do a number of things, including setting up the address space (allowable memory) for the process, allocating resources, and creating the base thread. After creating a process, the original process continues using its old address space while the new one operates in the new address space allocated to it. Many options are available in the creation process, so CreateProcess has a number of parameters to allow these to be specified. (This is in marked contrast to fork() used in UNIX, which has no parameters because the child is a clone of the parent until it changes its own characteristics.) Regardless of the options specified by CreateProcess, after the OS has started the new process, it returns a handle to the process and a handle to the base thread in the process that can be used to reference the process and thread in future operations.

The following prototype is taken from the Win32 API Reference Manual. This manual is available online in MSDN and contains other information about functions, parameters, and pathnames that may be useful. The set of data types used for the parameters are defined in *windows.h.*

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,      //Ptr to name of exe module
  LPTSTR lpCommandLine,           //Ptr to command line string
  LPSECURITY_ATTRIBUTES lpProcessAttributes, //Ptr to procs security attrib
  LPSECURITY_ATTRIBUTES lpThreadAttributes,  //Ptr to thread security attr
  BOOL bInheritHandles,             // handles  inheritance flag
  DWORD dwCreationFlags,          // creation flags
  LPVOID lpEnvironment,           // Ptr to new environment block
  LPCTSTR lpCurrentDirectory,     // Ptr to current directory name
  LPSTARTUPINFO lpStartupInfo,  // Ptr to startUpInfo
  LPPROCESS_INFORMATION lpProcessInformation //Ptr to proc_info
);
```

While there are many parameters, the good news is that a default value can be used for many of them that will give acceptable results for normal process creations. The parameters that are relevant to us are:

lpApplicationName and *lpCommandLine*
These provide two different ways to define the name of the file that is to be executed by the process's base thread. lpApplicationName is a string representation of the name of

the file to be executed and lpCommandLine is a string representation of the C-style command line that could be used to start the process if it was started from *cmd.exe*. If you pass a NULL for lpApplicationName and a command line string for lpCommandLine, your code will be consistent with typical C environments, including UNIX. This is what we will use. Figure 1 illustrates how to set this parameter up correctly.

lpProcessAttributes, *lpThreadAttributes*, and *bInheritHandles*
lpProcessAttributes and lpThreadAttributes are security attributes and they, along with bInheritHandles, can all be set to their default values: NULL for the security attributes and FALSE for the inheritance. Refer to Figure 1 for an illustration.

dwCreationFlags
This parameter is used to control the priority and other aspects of the new process. Four priority classes are available: HIGH_PRIORITY_CLASS, IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, or REALTIME_PRIORITY_CLASS. The default value is NORMAL.., unless the parent has IDLE.. in which case the child is also IDLE... Details of the flags are available in the API manual. Other flags can be passed in this parameter (by ORing their values together). One useful flag is CREATE_NEW_CONSOLE, which causes the new process to be created with its own console (cmd.exe) window. See Figure 1 for illustrations of these.

lpEnvironment
This parameter is used to pass a block of environment variables to the child process. If the parameter is NULL, the child inherits the parent's environment variables. If it is not NULL, it must point to a NULL-terminated block of NULL-terminated strings, each of the form name=value. Use NULL for this parameter.

lpCurrentDirectory
This string specifies the current drive and full pathname of the current directory in which the child should execute. A NULL value causes the child to execute using the same drive and directory as the parent. See Figure 1 for illustration.

lpStartupInfo
This field is a struct that contains miscellaneous parameters describing window characteristics. An instance of this struct must be created in the calling program, and its address passed as a parameter in lpStartupInfo. While there are many fields in the struct, for our purposes only the first field (cb) that defines the length of the struct must be initialized before it is passed to CreateProcess. See Figure 1 for an illustration of how the instance of the struct is created and initialized.

lpProcessInformation
The last parameter 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;
```

There is no length field in this structure so the calling process needs to allocate an instance of the struct and pass a pointer to the instance in the parameter lpProcessInformation. When the CreateProcess call returns, the four fields in the struct will be initialized with the:

- the handle of the newly created process (the hProcess field) for use within the address space of the process to refer to it in system calls
- the handle of the base thread (hThread)
- a global process identifier (dwProcessId) that can be used by another process to locate the new process, and
- a global thread identifier (dwThreadId).

CreateProcess allocates handles to processes and threads when it creates them and returns them in the PROCESS_INFORMATION structure. To avoid creating a "handle leak" (processes that are no longer needed, but are still taking up system resources), you should explicitly deallocate a handle (call CloseHandle) when the process or thread is no longer needed. When a process terminates, all of its handles (kept in the PCB) are released by the OS.

A CreateProcess request (since it results in a system call) can fail. If it does, CreateProcess returns a nonzero code that indicates the type of failure. For successful requests, the return code is zero.

The shell of the code to create a process is shown below in Figure 1:

```
#include  <windows.h>
#include <stdio.h>
#include <string.h>
.....
STARTUPINFO  startInfo;
PROCESS_INFORMATION processInfo;
..
strcpy(lpCommandLine, "C:\\WINNT\\SYSTEM\\NOTEPAD.EXE temp.txt");
ZeroMemory(&startInfo, sizeof(startInfo) );   // zero bytes starting at
                                              // startInfo. Number of bytes
                                              // to zero is second parameter
startInfo.cb = sizeof(startInfo);
// Start Process. If it fails, log message to display
if (!CreateProcess(NULL, lpCommandLine, NULL, NULL, FALSE,
     HIGH_PRIORITY_CLASS | CREATE_NEW_CONSOLE, NULL, NULL,
     &startInfo, &processInfo) ) {
   fprintf(stderr, "CreateProcess failed on error %d \n", GetLastError() );
   ExitProcess(1);    //Exit with error flag
};
...
CloseHandle(&processInfo.hThread);
CloseHandle(&processInfo.hProcess);
....
```

Figure 1. Shell of code used to create a process.

Creating a Thread

A process can create additional threads in itself by calling the appropriate function in the Win32 API, specifically CreateThread (which calls the native function NtCreateThread). When a thread is created, the execution environment for the thread is assumed by the OS to be the same as the process's within which it exists. There is, however, additional information that must be supplied. The following prototype is for CreateThread.

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,          //Ptr to security attrib
  DWORD dwStackSize,                 // initial thread stack size, in bytes
  LPTHREAD_START_ROUTINE lpStartAddress,          // Ptr to thread functn
  LPVOID lpParameter,                // argument for new thread
  DWORD dwCreationFlags,             // creation flags
  LPDWORD lpThreadId                 //  Ptr to returned thread identifier
);
```

The concepts of a handle being returned, a return code to indicate success or failure, and needing to close the thread when it is no longer needed apply to threads just as they do to processes. Specifics of the function parameters are:

lpThreadAttributes
This is the same as for CreateProcess. The NULL value is acceptable for us.

dwStackSize
Since each thread operates independently of other threads each has its own stack. This parameter allows the size of the stack to be set, although the default is usually acceptable. The default can be used by setting the value to 0.

lpStartAddress and *lpParameter*
For a process, it was necessary to provide the name of an executable file from which to load the process. Since a thread is created in an already existing process, the OS only needs the address in the current address space where the thread is to begin execution (essentially the info needed to initialize the program counter for the thread). lpStartAddress allows the thread to begin at any function within the current process. lpStartAddress is the address of an entry point for a function that has a prototype of the form:

```
DWORD WINAPI ThreadFunc(LPVOID);
```

It is possible to pass information to this function that will be executed by the new thread. Since there is a function call and a prototype, the type of the parameter passed must be known and declared or a *void* * type must be used to tell the compiler that the type of the parameter to the function is unknown at compile time. CreateThread uses the void * approach which is why the parameter type is LPVOID which is an alias for void *. The lpParameter value is passed to the function when the new thread begins to execute it.

dwCreationFlags
This parameter is used to control the way the new thread is created. There is only one possibility currently, namely CREATE_SUSPENDED. This causes the new thread to be suspended after creation until another thread executes the function

```
ResumeThread(targetThreadHandle);
```

on the new thread where targetThreadHandle is the handle of the new thread. The default value of dwCreationFlags is zero, which causes the thread to be active (not suspended) when it is created.

lpThreadId
This parameter is analogous to the dwThreadId field returned by CreateProcess.

Putting this all together, a call to create a new thread might look like:

```
DWORD WINAPI myFunc(LPVOID);   //Prototype for entry level thread
                               // function
...
int theArg;                    //argument being passed to the function
DWORD targetThreadID;
...
CreateThread(NULL, 0, myFunc, &theArg, 0, &targetThreadID);
```

The good news is that the preceding explains how to create threads. The bad news is that if you're using the C runtime library (which we will), this doesn't quite work. The reason is that the C library was developed in a UNIX environment which does not distinguish between processes and threads. Since in the thread world all threads in a process have access to any variable that is not in a thread stack, any thread can access information available to another. Richter (1997) has an example of such a variable, *errno*. errno is a variable set by runtime functions when they complete that shows the status of the completion. In the UNIX world, you do not need to access a function to check errno. You simply access it as any other variable. In the Win32 API, the function GetLastError() is used to access this variable. Suppose that two threads are executing in a process and both call C runtime functions. Both of these functions will set errno on their return, but only the last return value will remain in errno. This could result in one thread thinking the status of its system call was something that it was not (e.g., if one thread's call succeeded, resulting in an errno = 0, and the other's failed, resulting in an errno = -1). This phenomenon is known as a *race condition* and is difficult to reproduce or debug.

Microsoft has provided an alternative function to CreateThread called *_beginthreadex* to be used by programs that use multiple threads at the same time they use the C runtime library. The previously discussed race condition problem only occurs with any globally accessible variable used by this library, but there are several of these including errno. Microsoft's solution is to provide a copy of each of these variables for each thread. Then, when the thread interacts with the runtime library, the variables are shared only between the runtime code and the thread, but not among all the threads. The prototype for _beginthreadex is:

```
unsigned long _beginthreadex(
  void *security,
  unsigned stack_size,
  unsigned ( _stdcall *start_address) (void *),
  void *arglist,
  unsigned initflag,
  unsigned *thrdaddr
);
```

Although the parameter lists for CreateThread and _beginthreadex are different, it is relatively straightforward to transfer the parameter list from one to the other. Figure 2 illustrates this.

```
DWORD WINAPI myFunc(LPVOID);
...
LPSECURITY_ATTRIBUTES lpThreadAttributes = NULL;
DWORD stackSize = 0;
int theArg;
DWORD dwCreationFlags = 0;
DWORD targetThreadID;
...
        unsigned long threadHandle =_beginthreadex(
          (void *) lpThreadAttributes,
          (unsigned) stackSize,
          (unsigned (_stdcall *) (void *))  myFunc,
          (void *) &theArg,
          (unsigned) dwCreationFlags,
          (unsigned *) &targetThreadID
);
```

Figure 2. Shell code for use of _beginthreadex in C

Project

Your will need to build your program so that it accepts information passed to *main* via a command line like a traditional C program. Do **not** use cin or standard input to input information. To do this under Visual Studio, you must create your project as a console application. You must also tell Visual Studio that you are building a multithreaded program using _beginthreadex so it includes the appropriate libraries. How you do this depends on the IDE version you are using to build the program. The following are suggestions of things that you **may or may not** need to do:

1. Signal the compiler that this is a threaded program by specifying the multithread option (e.g., /MDd under Code Generation in the C/C++ folder/Runtime Library).
2. Notify the linker to include the library. (e.g, in the "Link" tab the list of Command Line modules must include "/libcmt.lib" or "/libcmtd.lib". The second lib is for use if you are building the debug version rather than the release version of the executable.

In addition, the prototype for *main* must have the form

```
int main (int argc, char * argv[]);
```

where *argc* is a count of the number of strings on the command line (including the executable name). For example, the command line entry of

```
hello 2
```

where hello is the executable name would result in a value for *argc* of two. *argv*[] is an array of null-terminated (traditional C) strings, with *argv*[0] holding the command name, *argv*[1] holding the first parameter, and so on. In our example, *argv*[0] would contain the null-terminated string *hello* and *argv*[1] would contain the string *2*. Note that while the

following examples are in C, the project can in fact be implemented in C++. Be careful to pay attention to the difference in how C and C++ handle passing parameters by reference.

Write a C/C++ program that takes three DOS command line parameters: the first is an integer specifying the number of threads to be initiated up to a maximum of 100; the second is the number of iterations that each thread should execute before terminating up to a maximum of 100000000. The third is an integer which if 1 means enable mutex checking before entering the critical section, and if 0 means disable mutex checking. A sample command line execution would be RunProgram 98 1000000 1.
Completing the project requires that you write a program that does the following:

- Read the command line parameters NumberOfThreads, NumberOfIterations, and EnableMutex
- For 1 to NumberOfThreads:
    - Create a new thread to execute simulated work. Each thread initiated should perform the activity described in the function *threadWork* in this handout to simulate CPU activity and generate possible race events.
    - Log its creation with an informative message that contains a numeric identifier for the thread.
    - Log a message with the thread's numeric identifier as each terminates
- The main thread should wait until all threads it initiated have terminated before outputting a message that displays the current value of the global variable count, and also the theoretical value that count should be if no race events had occurred. I will let you determine how to calculate the theoretical value.
- Terminate.

- Each thread should perform the action in the code shell given below (basically it's incrementing a count in a way that makes race condition collisions more likely) for the duration specified by NumberOfIterations

Here is the shell:

```
#include <process.h>            // for _beginthreadex
#include <windows.h>
#include <cstdio>
#include <cmath>
#include <cstdlib>
using namespace std;

bool enableMutex = false;      //Command line parameter
count = 0;       //Actual count after update by threads
unsigned long numberIter = 0;          //Command line parameter

int main( int argc, char * argv[]) {
        // Local variables
```

```
        // Get the command line parameters
        // If correct, argc should be 4 and argv[1] should be numberOfThreads,
                // argv[2] is number of Iterations, and argv[3] is enableMutex
        ....

        Create a Mutex  //See your text for a description of how to do this with
        Win32 API

        // For each thread
        for (i = 0; i < numberOfThreads; i++) {
                Create a new thread to execute simulated work
                Sleep(10);       // Let the new thread run
        }

        // Loop while the threads run to completion
        while (any thread is running) {
            Sleep(1000);
        }
        Output the actual value of count and the theoretical value
}
```

The above code uses a Win32 API function called Sleep(X). This function causes the current thread to yield its timeslice and block until X ms have elapsed. The thread will then be placed back into the appropriate scheduling queue. The placement of this call in the code above allows the new thread to run, by causing the calling thread to block for 100 ms. This technique is common in multithreaded applications.

The shell of the code for the simulation threads is in Figure 3.

```
DWORD WINAPI threadWork(LPVOID threadNo)
{
   // Create a CPU load by burning some cycles. Hope to cause contention
   for (unsigned int i = 0; i < numberIter; i++) {
        <Prepare to enter critical section. Check mutex if enableMutex is true.
        You provide this code.>

        // Critical section. The following loop is the CS. Do not change its' code.
        for (int j=0; j<10; j++)
                count = count + abs ( GetCurrentThreadId());  //Update the count

        <Exit critical section. You provide this code.>
    }
    ..
    // notify main this thread is done
    return;
}
```

Figure 3. Simulation thread shell

References:

Dr. Terry Metzgar

Part of the idea and discussion for this lab was taken from Gary Nutt, Operating Systems Projects using Windows NT, Addison Wesley, 1999.

Richter, Jeffrey, Advanced Windows, Third Edition, Microsoft Press, Redmond, WA, 1997.

Deliverables:

In addition to your program, create an aareadme.txt file that describes how to run your program and also the number of threads and iterations it takes on your computer to generate a race condition.

Submit all files necessary to run your program, **including the aareadme.txt,** your source code, and your program's executable. Put your submission in a zip folder named yourNameCSIS443.zip and upload your zipped folder to the appropriate assignment link in Canvas. All source code must comply with good standard coding practices.