COMP20180 - Intro to Operating Systems

Case Study

POSIX threads (Unix) versus Win32 threads (Windows)

Gerard Fogarty - 13303911

Joe Duffin - 13738019

Edwin Keville - 13718661

Table of Contents

Page 3: Introduction/Context/Background - *Joe Duffin*

- What is a Process.

- What is a Thread.

- Issues with Threads

Page 6: Win32 Threads - *Edwin Keville*

- What uses them and History

- Key features of Win32 Threads

- Issue Handling

- Example Implementation (C programming language)

Page 11: POSIX Threads (pthreads) - *Gerard Fogarty*

- What uses them (kinda obvious but key) and History

- Key features of POSIX Threads

- Issue Handling

- Example Implementation (C programming language)

Page 13: The Battle: Pthreads vs. WIN32 Threads - *Group Effort*

Who wins and why at:

- Performance

- Complexity

- Implementation

- Major Bonuses of one over the other

- Major Letdowns of one over the other

Page 14: Conclusion - *Group Effort*

- Our opinion on the best implementation

Page 14: References

*Introduction/Context/Background*

Multiprogramming is the process in which a system increases CPU utilization by organizing jobs, or code and data, so that the CPU always has one to execute. The key feature of systems with multiprogramming are processes, and, a further extension, threads. Processes and threads allow many programs, most of which the user is unaware of, to be executed with the appearance of parallelism, sometimes referred to as pseudoparallelism. This dramatically increases the speed at which a system executes several jobs. For example, while a given process is waiting on user input, and therefore in a waiting state, another processes' calculations' code can be executed by the CPU.

When a program is executed a process is created by the operating system. A process is just an instance of an executing program. Processes come under two categories:

* User Processes
* System Processes

User processes have a logical address space, different from the kernel space used for system processes. This logical address space is then mapped to an actual physical address space. Where two processes may have an identical logical address space, they always have unique physical addresses. A direct implication of this is that there is no automatically shared memory between processes. At a processes' address space a quantity of information is stored in a dynamic data structure called a process control block (PCB). The PCB must be created and initialized upon creation of a process. This is a very expensive operation.

A PCB would typically contain:

* The process ID (PID)
* The processes state (ready, running, blocked, suspended)
* The program counter (PC)
* CPU register values
* Memory management information
* Os resources allocated
* Accounting/scheduling Information

A process has one thread, or unit, of control. Where multiprogramming allows several processes to run concurrently, multi-threading allows each process to carry out several tasks related to the given process concurrently. In this way, threads are to processes as processes are to systems.

All instances of a thread belonging to a process share almost all of the information in the PCB, most importantly a processes' physical address space is shared across all threads. This sharing causes thread creation to be a very light weight operation, with very few overheads, making multi-threaded programs a very attractive multiprogramming implementation. A threads unique attributes, listed below, allow the thread to run independently within a process; carrying out instructions, calculating and amending variables.

Unique attributes of each thread (belonging to a process).

* The PC
* A stack pointer
* Interrupt Vectors
* Stack
* State
* Child threads

With several threads accessing and amending the same physical address space concurrently the process becomes very vulnerable to error. Race conditions can cause havoc and the non-deterministic behavior of an operating systems scheduler makes it difficult to program multiple threads of control effectively. Some of the potential problems are highlighted in the two code snippets below. It can be clearly seen that due to the non atomic behavior of the while loops in threads A and B, that there is the potential for neither thread to print "BOOM" or even terminate the while loop!

|  |  |
| --- | --- |
| Example Process | |
| **int** i = 0; | |
| Thread A | Thread B |
| **while** (i != 15) {  i++;  **if** (i == 4) *printf*("BOOM");  } | **while** (i != 15) {  i++;  **if** (i == 7) *printf*("BOOM");  } |

In order to avoid disastrous consequences such as the potentials for the given code, while writing multi-threaded programs one must consider and handle several factors.

* Synchronization

- Forcing deterministic behavior so the order of execution of certain threads and sections of code can be controlled.

* Mutual Exclusion

- Defining sections of code as an atomic critical section in order to avoid race conditions such as those highlighted above.

* Resource Sharing

- Seeing as the OS resources are allocated to the process, not the thread, threads can and do have to wait for other threads to relinquish them.

This essay is going to explore two different implementations of threads across two different operating systems and the way in which the above issues are handled. Code, as well as verbose examples, will will be used to demonstrate the main features of:

* POSIX threads (pthreads)

Utilized in Unix systems

* Win32 threads

Utilized in Microsoft Windows 32bit systems

----------------------------------------------

*Win32 Threads*

The Win32 API was first introduced alongside Windows NT 3.1. It succeeded Win16 which accompanied windows 3.1 and older. While it was available earlier, it was mainly the extremely popular Windows 95 operating system that resulted in its widespread use and a great many applications being ported over to it.

Windows, as the name would suggest, introduced a window based graphical environment with various applications running concurrently. So while initially many thought there would not be much need for multiprocessing on personal computers this new way of working showed that it was in fact essential.

Win16 did offer multiprocessing but it was extremely limited. This limited environment however provided a lot of the logic and methods that would later be implemented on the more advanced API’s which followed.

The first attempts from windows at multiprocessing were OS/2 with Presentation Manager. But the serialized inputs from elements such as the keyboard and mouse meant that one application could possibly tie up the entire system. Deserializing the messaging queue required a new fix, and this fix was multithreading.

While every new version of windows makes additions to the API, the naming remains mostly the same. Win16 was followed by Win32, with minor variations(Win32s and Win32c). and these were followed by Win64

For the purpose of our case study we will be looking at threads as defined in the Win32 API.

Windows threads themselves are a basic unit of CPU utilization and comprise of a thread ID, a program counter. A register set and a stack. They are a path of execution and every process requires at least one thread. They exist mainly on two levels, the user level and the Kernel level. The win32 API operates on a kernel level, meaning it is handled directly by the operating system.

There is a close link between user and kernel level threads and Win32 uses a one-to-one model whereby each user thread is linked to a kernel thread. This allows for concurrency as if one thread makes a blocking system call, the other threads may still run. Although this method does mean that for each thread there must be a corresponding kernel thread. This overhead means that Win32 restricts the amount of threads supported.

The Win32 library also exists within the Kernel and contains all the information such as code and data structures that the programmer must use.

Programs are mostly written in the C/C++ language and the threads themselves are representations of functions. A program begins execution with its main thread and new threads can be created from within.

The API function which creates a new thread is called CreateThread.

For example

hThread = CreateThread(&security\_attributes,

dwStackSize, ThreadProc, pParam, dwFlags &idThread);

security\_attributes: these are just that. The security attributes for the thread. This can be set to null in windows NT and is ignored altogether in windows 98. It is a pointer to a structure and setting to null will give it the default security values.

dwStacksize: will declare the initial size of the stack size in bytes although windows can dynamically expand this when it is required.

threadProc: is a pointer to the thread function

pParam: is the main argument for threadProc and is how the threads can share data between main and secondary threads.

dwFlags: Here a flag can be used such as CREATE\_SUSPENDED. If this happens then the thread is created in a suspended state and will remain so until ResumeThread is called.

&idThread: this is simply a pointer to a variable that will hold the ID for the thread.

in the process.H header file there is also \_beginthread. This is much simpler and works perfectly fine for most applications

hThread = \_beginthread (ThreadProc, uiStackSize, pParam)

Threads such as this are then ended using the \_endthread function. When CreateThread is used, it is recommended to close with ExitThread otherwise deadlocks may occur.

Thread priorities are also very important when it comes to handling and they come in 5 variations: Highest, Above Normal, Normal, Below Normal and Idle

(eg: THREAD\_PRIORITY\_HIGHEST, THREAD\_PRIORITY\_ABOVE\_NORMAL…)

HANDLE is specific to win32 and it differs from the thread ID.It is a token which allows you to perform tasks such as waiting for the thread or to kill it.

Looking at a code sample can give us a better idea.

|  |
| --- |
| **#include**<stdio.h>  **#include**<stdlib.h>  **void** **ThreadProc**(**void** \*param);  **int** **main**() {  **int** val = 7;  HANDLE handle;  //here the thread is created  handle = (HANDLE) \_beginthread(ThreadProc, 0, &val);  //main blocked until thread completion  WaitForSingleObject(handle, INFINITE);  **return** 0;  }  **void** **ThreadProc**(**void** \*param) {  **int** h = \*((**int**\*) param);  **printf**("thread is Running, value passed is %d", h);  \_endthread();  } |

The WaitForSingleObject() function will wait for the completion of another thread. As shown here the parameter passed to it is the handle.

Win32 threads handle mutual explusion by implementing Critical Sections. These are blocks of code that are atomic (cannot be interrupted until completion). They are defined as a critical section object and use the global variable CRITICAL\_SECTION.

To get a better idea of how this is implemented in code we can look at this toy c code example with expiation of the functions below.

|  |
| --- |
| CRITICAL\_SECTION gCS; // global variable  **const** **int** gcMaxCount = 10;  **volatile** **int** gCount = 0;  **void** **ThreadMain**(**char** \*name) {  **while** (gCount < gcMaxCount) {  EnterCriticalSection(&gCS); /// Critical Section of Code  **printf**("I, \"%s\", am in critical Section\n", name);  gCount++;  Sleep(200);  LeaveCriticalSection(&gCS);  }  } |

Variables and functions from previous code explained below.

CRITICAL\_SECTION gCS: Here the structure is declared as a global variable. It is shared across all threads.

EnterCriticalSection(&gCS): Then when the thread wants to enter the critical section it is required to call this function. Passing it the structure declared above. The thread which calls this then ‘owns’ the critical section

At this point any other thread that tries to call the same critical section will be suspended. It will remain that way until the critical section owner has left that section. I

LeaveCriticalSection(&gCS): In order for the owner to leave they then call this function. Ownership of the section is then passed to the second thread which is now no longer suspended.

Another important function with regards to critical sections is deleteCriticalSection().This allows for the critical section to be deleted and the system resources used to be freed.

When using this kind of mutual exclusion, care needs to be taken. There is a possibility for starvation to occur where a thread takes up the critical section for an extremely long time leaving the other thread suspended.

There are limitations involved as these critical sections only work for one particular process at a time. If several processes are required to share resources then a mutex object must be used. They are very similar to the critical section in that they are a synchronization object. They have a state that, when it is not owned, is set to signaled. Then when it is owned it is set to non signaled. Each thread needs to wait for ownership. The wait is not necessarily FIFO (First In First Out) but is decided by a number of factors within the system.

----------------------------------------------

*POSIX Threads (pthreads)*

Initially hardware vendors would create their own versions of threads depending on their needs for a specific project. Of course this caused problems for developers as it greatly reduced the portability of applications. The solution to this as to create a standard. This was specified by the IEEE POSIX 1003.1c standard(1995) for UNIX systems and implementations which adhered to the standard were referred to as POSIX thread or Pthreads.

This standard has continued to grow and evolve with todays programming. The API itself is implemented using the pthreads.h header and is a set of C language programming types and procedure calls.

POSIX threads are initialized with the method call:

pthread **create**(thread, attr, start\_routine, arg);

With passed parameters defined as follows:

thread: a memory location or data type pthread\_t where a thread ID will be stored

attr: A pointer to a pthread\_attr\_t which specifies a set of attributes for a thread

start: A pointer to the function that the thread should execute

arg: An argument which can be passed to a the thread.

The code sample below demonstrates thread creation.

|  |
| --- |
| **#include**<pthread.h>  **#include**<stdio.h>  **#include**<stdlib.h>  **#include**<assert.h>  **#define** NUM\_SQUARES 100  **void** \***TaskCode**(**void**\* r) {  **int**\* val = \*((**int**) r);  **printf**("Hello from thread, passed value is %d", val);  **return** NULL;  }  **int** **main** (**void**) {  pthread\_t thread;  **int** val = 7;  **int** rc = pthread\_create(&thread, NULL, TaskCode, &val);  assert(0 == rc);  //the main thread blocks its self while waiting  rc = pthread\_join(thread, NULL);  assert( 0 == rc);  **return** 0;  } |

Unix POSIX threads rely on mutex for mutual exclusion. A mutex variable is declared globally using a call to pthread\_mutex\_t myMutex. To guarantee mutual exclusion a thread must 'own' this variable when accessing a critical section of code. Ownership is granted with a call to pthread\_mutex\_lock(&myMutex). Only one thread is ever granted ownership and other subsequent calls to pthread\_mutex\_lock by other threads results in that thread being blocked until the owner of myMutex calls pthread\_mutex\_unlock(&myMutex). When a mutex is no longer needed a call to pthread\_mutex\_destroy(&myMutex) is used.

The following code snippet demonstrates the pthread\_mutex being used in a function executed by a thread used to carry out a block of code atomically.

|  |
| --- |
| pthread\_mutex\_t count\_mutex; //the mutex is global  **long** **long** count;  **void**  **increment\_count**(){  **pthread\_mutex\_lock**(&count\_mutex); //lock  count = count + 1;  **pthread\_mutex\_unlock**(&count\_mutex); //unlock  } |

----------------------------------------------

*The Battle: PThreads vs. WIN32 Threads*

There is much controversy over which implementation of threads is better. There are arguments using a given feature of either Win32 or Pthreads to advocate both for and against that particular implementation. It just depends what camp you are hearing it from. At face value, both implementations offer a very similar end result. Here some of the differences in four area's are highlighted and examined.

- Persistence of signals

There is a dramatic difference between the two APIs here. Win32 threads signals' persist. Once created they must be handled at some point by the programmer, otherwise the signal will cause havoc and some unexpected behavior in the program. On the contrary, pthreads must signal in a while loop in order for the signal to persist. This allows non looped signals to query, and, if the signal is not met, the signal dies. The two different implementations can be compared to the tree falling in the woods problem. Should a pthread fall in the woods (signal) and no one is around to hear it, the pthread gets forgotten about. However, should a Win32 thread fall in the woods, it will continue to scream in pain until some one finds it.

- Data Types

Win32 threads have a single main data type, the HANDLE. The HANDLE is the data typed used for threads, mutex's, semaphores, etc. When reading c code implementing Win32 threads the Object type that a method call is referring to may not be apparent. It is possible to have an array containing both threads and mutexs.

Pthreads are very explicit in their data types using separate types for each object:

pthread\_t, pthread\_mutex\_t, pthread\_cond\_t, etc.

- Windows ability to Wait for Object(s)

The single data type of windows allows for very simple signal handling. If a Win32 program is waiting for a thread termination or a mutex unlock, or even multiples of either, a call to WaitForSingleObject() or WaitForMultipleObjects() will block the calling thread until signaled. This allows for apparently simple and elegant code but at the huge cost of the difficulty to implement. Very careful and particular logic must be implemented in order to avoid unwanted behavior. In unix, each object, whether it be a mutex or a thread, must be explicitly handled by its corresponding function. If the main thread is to wait for the completion of a set of threads, it must loop through a series of calls to pthread\_join() in order to unblock.

- System calls and Library functions

This is a small discrepancy between the two. The POSIX standard declares all of its thread related functions and variables in the pthreads.h header file. This file contains system call wrappers and must be included in any multi-threaded program. The Win32 implementation of threads relies on direct system calls. These direct system calls limit the portability of of the program written for the Win32 API, where with a program written with pthreads.h will run on any system following the POSIX standards.

----------------------------------------------

*Conclusion*

Its a subjective decision to define one or the other as 'better'. It can be argued that the simple single data type of windows threads makes coding simpler or the lack of explicitness makes it more difficult, depending on what side of the fence you are one. The portability, in our opinion, is a huge issue. However, die hard windows users are adamant that windows is the only operating system ever needed. They've never clearly never seen the light. In respect for our sense of disdain towards windows users, we must vote in favor of the POSIX defined standard for threads. We believe its explicitness, efficient signal handling and portability defines it as the best implementation.

----------------------------------------------

*References:*

Books:

Operating System Concepts - Silberschatz, Gavlin & Gagne (2005)

Modern Multi-threading - Carver, Tai (2006)

Modern Operating Systems - Tanenbaum (2009)

Programming Windows - Petzold (1999)

Websites:

Pthreads vs Win32 Threads - http://www.slideshare.net/abufayez/pthreads-vs-win32-threads

Forums at - http://software.intel.com/forums/

Lecture Slides from the course were also used