POSIX

Implementation

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;

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;

}

Mutual exclusion:

Unix POSIX threads rely on mutex for mutual exclusion. A mutex variable is decalred globally using a call to pthread\_mutex\_t myMutex. To guarentee 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 calles 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.

When mutex is called, the following happens: (come back soon)

*Cover page: including title, authors, student numbers, and who wrote what"*

*Table of contents*

Introduction/Context/BackGround

- What is a Process.

- What is a Thread.

- Issues with Threads

- X

- Y

- Z

Win32 Threads

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

- Key features of Win32 Threads

- A

- B

- C

- Issue Handling

- X

- Y

- Z

- Example Implementation (C programming language)

- int main();

POSIX Threads (pthreads)

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

- Key features of POSIX Threads

- A

- B

- C

- Issue Handling

- X

- Y

- Z

- Example Implementation (C programming language)

- int main();

The Battle: PThreads vs. WIN32 Threads

Who wins and why at:

- Performance

- Complexity

- Implementation

- Major Bonuses of one over the other

- Major Letdowns of one over the other

Conclusion

- Short and sweet really, as the battle will conclude pretty well,

"We feel X because A, B, C".

Introduction

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 PCD 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 above code, while writing multi-threaded programs one must consider and handle:

* 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

My References:

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

Modern Multi-threading - Carver, Tai (2006)

Modern Operating Systems - Tanenbaum (2009)

Good Web Pages:

ARTICLE:

Why windows threads are better than pthreads:

https://software.intel.com/en-us/blogs/2006/10/19/why-windows-threads-are-better-than-posix-threads

OPINIONS:

Stack overflow rant about the two:

http://stackoverflow.com/questions/5644912/posix-threads-vs-win32-threads

CODE:

Sample Win32 thread implementation:

http://www.cs.rpi.edu/courses/netprog/WindowsThreads.html

CODE:

Win32 API tutorial

https://www.relisoft.com/win32/active.html

ARTICLE:

The microsoft docs on Multithreading with windows:

https://msdn.microsoft.com/en-us/library/y6h8hye8.aspx

CODE:

LOADS of info on pthread implementation:

https://computing.llnl.gov/tutorials/pthreads/#PthreadsAPI

CODE:

Similar to above, in a tutorial format:

http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

*Introduction*

What is a thread, what is a process? what issues arise that need handling?

*Systems overview;*

Win32 threads:

*features*

*structure*

*hardware/software requirements*

*evolution*

Pthreads:

*features*

*structure*

*hardware/software requirements*

*evolution*

*Comparisons;*

Need to talk about the pro's and cons of the two different inmplementations,

dont think we'll be able to write this until the above 3 sections are written.

*suggested subsections:*

*performance*

*complexity*

*implementation*

*advantages and disadvantages*

*. . .*

*Conclusions/Discussion*

*. . .*

*References (books, papers, technical reports, websites); references should be cited*

*within the text where needed*

*. . .*