Multithreading — An Overview

In most modern operating systems it is possible for an application to split into many "threads" that all execute concurrently. It might not be immediately obvious why this is useful, but there are numerous reasons why this is beneficial.

When a program is split into many threads, each thread acts like its own individual program, except that all the threads work in the same memory space, so all their memory is shared. This makes communication between threads fairly simple, but there are a few caveats that will be noted later.

So, what does multithreading do for us?

Well, for starters, multiple threads can run on multiple CPUs, providing a performance improvement. A multithreaded application works just as well on a single-CPU system, but without the added speed. As multi-core processors become commonplace, such as Dual-Core processors and Intel Pentium 4's with HyperThreading, multithreading will be one of the simplest ways to boost performance.

Secondly, and often more importantly, it allows the programmer to divide each particular job of a program up into its own piece that operates independently of all the others. This becomes particularly important when many threads are doing blocking I/O operations.

A media player, for example, can have a thread for pre-buffering the incoming media, possibly from a harddrive, CD, DVD, or network socket, a thread to process user input, and a thread to play the actual media. A stall in any single thread won't keep the others from doing their jobs.

For the operating system, switching between threads is normally cheaper than switching between processes. This is because the memory management information doesn't change between threads, only the stack and register set do, which means less data to copy on context switches.

Multithreading — Basic Concepts

Multithreaded applications often require synchronization objects. These objects are used to protect memory from being modified by multiple threads at the same time, which might make the data incorrect.

The first, and simplest, is an object called a mutex. A mutex is like a lock. A thread can lock it, and then any subsequent attempt to lock it, by the same thread or any other, will cause the attempting thread to block until the mutex is unlocked. These are very handy for keeping data structures correct from all the threads' points of view. For example, imagine a very large linked list. If one thread deletes a node at the same time that another thread is trying to walk the list, it is possible for the walking thread to fall off the list, so to speak, if the node is deleted or changed. Using a mutex to "lock" the list keeps this from happening.

Computer Scientist people will tell you that Mutex stands for Mutual Exclusion.

In Java, Mutex-like behaviour is accomplished using the synchronized keyword.

Technically speaking, only the thread that locks a mutex can unlock it, but sometimes operating systems will allow any thread to unlock it. Doing this is, of course, a Bad Idea. If you need this kind of functionality, read on about the semaphore in the next paragraph.

Similar to the mutex is the semaphore. A semaphore is like a mutex that counts instead of locks. If it reaches zero, the next attempt to access the semaphore will block until someone else increases it. This is useful for resource management when there is more than one resource, or if two separate threads are using the same resource in coordination. Common terminology for using semaphores is "uping" and "downing", where upping increases the count and downing decreases and blocks on zero.

Java provides a Class called Semaphore which does the same thing, but uses acquire() and release() methods instead of uping and downing.

With a name as cool-sounding as semaphore, even Computer Scientists couldn't think up what this is short for. (Yes, I know that a semaphore is a signal or flag ;)

Unlike mutexes, semaphores are designed to allow multiple threads to up and down them all at once. If you create a semaphore with a count of 1, it will act just like a mutex, with the ability to allow other threads to unlock it.

The third and final structure is the thread itself. More specifically, thread identifiers. These are useful for getting certain threads to wait for other threads, or for getting threads to tell other threads interesting things.

Computer Scientists like to refer to the pieces of code protected by mutexes and semaphores as Critical Sections. In general, it's a good idea to keep Critical Sections as short as possible to allow the application to be as parallel as possible. The larger the critical section, the more likely it is that multiple threads will hit it at the same time, causing stalls.

In POSIX, the types we'll be dealing with are pthread\_t for thread identifiers, pthread\_mutex\_t for mutexes, and sem\_t for semaphores. We use the word "pthread" a lot because it stands for POSIX Threads.

Compiling Multithreaded Programs

Compiling multithreaded applications will require a few minor tweaks to our build setup. First, we'll need to include the appropriate header file. For POSIX systems, this header is called pthread.h. This header defines all the functions we'll be using to make threads. If we're using semaphores we'll also need to include semaphore.h.

#include <pthread.h>

#include <semaphore.h>

The next change is that we'll need to link our program with the pthread library to use its functions. For a compiler like gcc we simply use the -l option, like this:

gcc myProgram.o -o myProgram -lpthread

Now that we've got the header in place, and we know how to link our program, let's get started.

Creating a thread

Creating a pthread is fairly easy. The function pthread\_create is used, and it takes 4 arguments.

int pthread\_create(pthread\_t \* pth, pthread\_attr\_t \*att, void \* (\*function), void \* arg);

The first argument is a pointer to a pthread\_t, where the function stores the identifier of the newly-created thread. The next argument is the attribute argument. This is typically NULL, but can also point to a structure that changes the thread's attributes. the third argument is the function the new thread will start at. If the thread returns from the function, the thread is terminated as well. You can think of the function as main, since it behaves similarly. The final argument is passed to the function when the thread is started. this is similar to the argc/argv command line arguments to main, but it can be any data type. Zero is returned on success, otherwise a failure of some variety happened.

Inside the thread function, a thread can terminate itself by returning from the thread function or by calling pthread\_exit. They behave identically.

A thread can also be "detached", which frees all the resources the thread acquired while it was running as soon as it terminates. This is accomplished with pthread\_detach. A detached thread can't be waited on.

Stopping a thread

Sometimes an application may wish to stop a thread that is currently executing. The function pthread\_cancel can help us accomplish this.

int pthread\_cancel(pthread\_t thread);

The only argument to pthread\_cancel is the thread identifier for the thread to be cancelled. It returns zero if successful, or an error code otherwise.

A thread can set whether or not it can be cancelled by using int pthread\_setcancelstate.

Mutexes and Semaphores

Mutexes are fairly easy to create. The function we use is pthread\_mutex\_init, which takes 2 parameters. The first is a pointer to a mutex\_t that we're creating. The second parameter is usually NULL, but can also be a pthread\_mutexattr\_t structure that specifies different attributes for it.

To lock and unlock a mutex, use pthread\_mutex\_lock and pthread\_mutex\_unlock. These both take 1 parameter: a pointer to the mutex being operated on. pthread\_mutex\_trylock is similar to pthread\_mutex\_lock, except that if it can't lock the mutex, it returns a error instead of blocking.

When the mutex is no longer needed, it can be freed with pthread\_mutex\_destroy.

Semaphores follow a similar paradigm. They are initialized with sem\_init, which takes 3 parameters. The first is a pointer to the semaphore being initialized. The second is always zero. This argument is used to denote semaphores shared between processes, but it isn't always supported. The third argument specifies the initial value of the newly created semaphore.

To "Up" a semaphore, use sem\_post. To "Down" a semaphore, use sem\_wait. These kind of parallel pthread\_mutex\_lock and pthread\_mutex\_unlock.

sem\_destroy is used to destroy a semaphore once it is no longer needed.

Multithreading — Waiting for other threads

It is also possible to make one thread stop and wait for another thread to finish. This is accomplished with pthread\_join. This function takes a pthread\_t identifier to pick which thread to wait for, and takes a void \*\* parameter to capture the return value. Joining a thread that has already exited is possible, and performing this will free any resources the thread had not already deallocated. In GNU/Linux, as well as other UNIX-like operating systems, these unjoined threads are called zombies.

Note that only 1 thread can wait for any other thread. A detached thread (with pthread\_detach) can't be waited on either.

Here's some example code to illustrate pthread\_join:

#include <stdio.h>

#include <pthread.h>

/\* This is our thread function. It is like main(), but for a thread \*/

void \*threadFunc(void \*arg)

{

char \*str;

int i = 0;

str=(char\*)arg;

while(i < 10 )

{

usleep(1);

printf("threadFunc says: %s\n",str);

++i;

}

return NULL;

}

int main(void)

{

pthread\_t pth; // this is our thread identifier

int i = 0;

/\* Create worker thread \*/

pthread\_create(&pth,NULL,threadFunc,"processing...");

/\* wait for our thread to finish before continuing \*/

pthread\_join(pth, NULL /\* void \*\* return value could go here \*/);

while(i < 10 )

{

usleep(1);

printf("main() is running...\n");

++i;

}

return 0;

}

Running this code will produce a bunch of text from threadFunc(), and then a bunch from main().

Multithreading — Example Source

Here's some example code to illustrate thread creation:

#include <pthread.h>

#include <stdio.h>

/\* This is our thread function. It is like main(), but for a thread\*/

void \*threadFunc(void \*arg)

{

char \*str;

int i = 0;

str=(char\*)arg;

while(i < 110 )

{

usleep(1);

printf("threadFunc says: %s\n",str);

++i;

}

return NULL;

}

int main(void)

{

pthread\_t pth; // this is our thread identifier

int i = 0;

pthread\_create(&pth,NULL,threadFunc,"foo");

while(i < 100)

{

usleep(1);

printf("main is running...\n");

++i;

}

printf("main waiting for thread to terminate...\n");

pthread\_join(pth,NULL);

return 0;

}

The output will be (mostly) alternating lines as the main() and threadFunc() threads execute and pause. Without the usleep()'s they'll not switch because we aren't doing anything that takes long enough to consume our whole time slice.

We could capture the return value in the pthread\_join() call if we used a variable instead of NULL for the second argument.

Performance Considerations

When designing an application for threads, or converting an existing program, there are some considerations to keep in mind when it comes to threads.

First, thread creation tends to be expensive -- spawning thousands of threads with short lifetimes usually isn't time-effective. If you need to create threads frequently, a common pattern used to reduce this cost is a "Thread Pool". At startup, the application will spawn a number of threads and supply them on demand. When the thread task completes, the thread returns to the pool for reuse later. Fancier implementations will dynamically close threads when there's too much of a surplus, or spawn additional threads when there's a shortage.

Each additional thread also gets its own stack. This stack space can be large, which can consume a lot of memory space (especially in 32bit applications). There are methods to reduce a thread's stack size using the pthreads API. For small numbers of threads this usually isn't a concern, but it's something to keep in mind.

Lock contention (when two or more threads are trying to acquire the same lock) requires skillful design to keep as many threads operating in parallel as possible. There are several volumes of literature on ways to design locks, lock heirarchies, and other variations to mitigate this cost.

Multithreading Terms

There are many terms used when writing multithreaded applications. I'll try to describe a few of there here.

Deadlock — A state where two or more threads each hold a lock that the others need to finish. For example, if one thread has locked mutex A and needs to lock mutex B to finish, while another thread is holding mutex B and is waiting for mutex A to be released, they are in a state of deadlock. The threads are stuck, and cannot finish. One way to avoid deadlock is to acquire necessary mutexes in the same order (always get mutex A then B). Another is to see if a mutex is available via pthread\_mutex\_trylock, and release any held locks if one isn't available.

Race Condition — A program that depends on threads working in a certain sequence to complete normally. Race Conditions happen when mutexes are used improperly, or not at all.

Thread-Safe — A library that is designed to be used in multithreaded applications is said to be thread-safe. If a library is not thread-safe, then one and only one thread should make calls to that library's functions.