developerWorks®

POSIX threads explained

A simple and nimble tool for memory sharing

Daniel Robbins

July 01, 2000

POSIX (Portable Operating System Interface) threads are a great way to increase the responsiveness and performance of your code. In this series, Daniel Robbins shows you exactly how to use threads in your code. A lot of behind-the-scenes details are covered, so by the end of this series you'll really be ready to create your own multithreaded programs.

Threads are fun

Knowing how to properly use threads should be part of every good programmer's repertoire. Threads are similar to processes. Threads, like processes, are time-sliced by the kernel. On uniprocessor systems the kernel uses time slicing to simulate simultaneous execution of threads in much the same way it uses time slicing with processes. And, on multiprocessor systems, threads are actually able to run simultaneously, just like two or more processes can.

So why is multithreading preferable to multiple independent processes for most cooperative tasks? Well, threads share the same memory space. Independent threads can access the same variables in memory. So all of your program's threads can read or write the declared global integers. If you've ever programmed any non-trivial code that uses fork(), you'll recognize the importance of this tool. Why? While fork() allows you to create multiple processes, it also creates the following communication problem: how to get multiple processes, each with their own independent memory space, to communicate. There is no one simple answer to this problem. While there are many different kinds of local IPC (inter-process communication), they all suffer from two important drawbacks:

- They impose some form of additional kernel overhead, lowering performance.
- In almost all situations, IPC is not a "natural" extension of your code. It often dramatically increases the complexity of your program.

Double bummer: overhead and complication aren't good things. If you've ever had to make massive modifications to one of your programs so that it supports IPC, you'll really appreciate the simple memory-sharing approach that threads provide. POSIX threads don't need to make expensive and complicated long-distance calls because all our threads happen to live in the same house. With a little synchronization, all your threads can read and modify your program's existing data structures. You don't have to pump the data through a file descriptor or squeeze it into a tight, shared memory space. For this reason alone you should consider the one process/multithread model rather than the multiprocess/single-thread model.

Threads are nimble

But there's more. Threads also happen to be extremely nimble. Compared to a standard fork(), they carry a lot less overhead. The kernel does not need to make a new independent copy of the process memory space, file descriptors, etc. That saves a lot of CPU time, making thread creation ten to a hundred times faster than new process creation. Because of this, you can use a whole bunch of threads and not worry too much about the CPU and memory overhead incurred. You don't have a big CPU hit the way you do with fork(). This means you can generally create threads whenever it makes sense in your program.

Of course, just like processes, threads will take advantage of multiple CPUs. This is a really great feature if your software is designed to be used on a multiprocessor machine (if the software is open source, it will probably end up running on quite a few of these). The performance of certain kinds of threaded programs (CPU-intensive ones in particular) will scale almost linearly with the number of processors in the system. If you're writing a program that is very CPU-intensive, you'll definitely want to find ways to use multiple threads in your code. Once you're adept at writing threaded code, you'll also be able to approach coding challenges in new and creative ways without a lot of IPC red tape and miscellaneous mumbo-jumbo. All these benefits work synergistically to make multithreaded programming fun, fast, and flexible.

I think I'm a clone now

If you've been in the Linux programming world for a while, you may know about the __clone() system call. __clone() is similar to fork(), but allows you to do lots of things that threads can do. For example, with __clone() you can selectively share parts of your parent's execution context (memory space, file descriptors, etc.) with a new child process. That's a good thing. But there is also a not-so-good thing about __clone(). As the __clone() man page states:

> "The __clone call is Linux-specific and should not be used in programs intended to be portable. For programming threaded applications (multiple threads of control in the same memory space), it is better to use a library implementing the POSIX 1003.1c thread API, such as the Linux-Threads library. See pthread_create(3thr)."

So, while __clone() offers many of the benefits of threads, it is not portable. That doesn't mean you shouldn't use it in your code. But you should weigh this fact when you are considering using __clone() in your software. Fortunately, as the __clone() man page states, there's a better alternative: POSIX threads. When you want to write **portable** multithreaded code, code that works under Solaris, FreeBSD, Linux, and more, POSIX threads are the way to go.

Beginning threads

Here's a simple example program that uses POSIX threads:

thread1.c

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void *thread_function(void *arg) {
  int i;
  for ( i=0; i<20; i++ ) {
    printf("Thread says hi!\n");
    sleep(1);
  }
  return NULL;
}

int main(void) {

  pthread_t mythread;

  if ( pthread_create( &mythread, NULL, thread_function, NULL) ) {
    printf("error creating thread.");
    abort();
  }

  if ( pthread_join ( mythread, NULL ) ) {
    printf("error joining thread.");
    abort();
  }

  exit(0);

}
```

To compile this program, simply save this program as thread1.c, and type:

```
$ gcc thread1.c -o thread1 -lpthread
```

Run it by typing:

```
$ ./thread1
```

Understanding thread1.c

thread1.c is an extremely simple threaded program. While it doesn't do anything useful, it'll help you understand how threads work. Let's take a step-by-step look at what this program does. In main() we first declare a variable called mythread, which has a type of pthread_t. The pthread_t type, defined in pthread.h, is often called a "thread id" (commonly abbreviated as "tid"). Think of it as a kind of thread handle.

After mythread is declared (remember that mythread is just a "tid", or a handle to the thread we are about to create), we call the pthread_create function to create a real, living thread. Don't be confused by the fact that pthread_create() is inside an "if" statement. Since pthread_create() returns zero on success and a non-zero value on failure, placing the function call in an if() is just an elegant way of detecting a failed pthread_create() call. Let's look at the arguments to pthread_create. The first one is a pointer to mythread, &mythread. The second argument, currently set to NULL, can be used to define certain attributes for our thread. Because the default thread attributes work for us, we simply set it to NULL.

Our third argument is the name of a function that the new thread will execute when it starts. In this case the name of the function is thread_function(). When this thread_function() returns, our new thread will terminate. In this example our thread function doesn't do anything remarkable. It just prints out "Thread says hi!" 20 times and then exits. Notice that thread_function() accepts a void * as an argument and also returns a void * as a return value. This shows us that it's possible to use a void * to pass an arbitrary piece of data to our new thread, and that our new thread can return an arbitrary piece of data when it finishes. Now, how do we pass our thread an arbitrary argument? Easy. We use the fourth argument to the pthread_create() call. In this example, we set it to NULL because we don't need to pass any data into our trivial thread_function().

As you may have guessed, the program will consist of two threads after pthread_create() successfully returns. Wait a minute, **two** threads? Didn't we just create a single thread? Yes, we did. But our main program is also considered a thread. Think of it this way: if you write a program and don't use POSIX threads at all, the program will be single-threaded (this single thread is called the "main" thread). By creating a new thread we now have a total of two threads in our program.

I imagine you have at least two important questions at this point. The first is what the main thread does after the new thread is created. It keeps on going and executes the next line of our program in sequence (in this case that line is "if (pthread_join(...))"). The second question you may be wondering about is what happens to our new thread when it exits. It stops and waits to be merged or "joined" with another thread as part of its cleanup process.

OK, now on to pthread_join(). Just as pthread_create() split our single thread into two threads, pthread_join() merges two threads into a single thread. The first argument to pthread_join() is our tid mythread. The second argument is a pointer to a void pointer. If the void pointer is non-NULL, pthread_join will place our thread's void * return value at the location we specify. Since we don't care about thread_function()'s return value, we set it to NULL.

You'll notice that our thread_function() takes 20 seconds to complete. Long before thread_function() completes, our main thread has already called pthread_join(). When this occurs our main thread will block (go to sleep) and wait for thread_function() to complete. When thread_function() completes, pthread_join() will return. Now our program has one main thread again. When our program exits, all new threads have been pthread_join()ed. This is exactly how you should deal with every new thread you create in your programs. If a new thread isn't joined it will still count against your system's maximum thread limit. This means that not doing the proper cleanup will eventually cause new pthread_create() calls to fail.

No parents, no children

If you've used the fork() system call you're probably familiar with the concept of parent and child processes. When a process creates another new process, using fork(), the new process is considered the child and the original process is considered the parent. This creates a hierarchical relationship that can be handy, especially when waiting for child processes to terminate. The waitpid() function, for example, will tell the current process to wait for the termination of any child processes. waitpid() is used to implement a simple cleanup routine in your parent process.

Things are a little more interesting with POSIX threads. You may have noticed that I have intentionally avoided using the terms "parent thread" and "child thread" so far. That's because with POSIX threads this hierarchical relationship doesn't exist. While a main thread may create a new thread, and that new thread may create an additional new thread, the POSIX threads standard views all your threads as a single pool of equals. So the concept of waiting for a child thread to exit doesn't make sense. The POSIX threads standard does not record any "family" information. This lack of genealogy has one major implication: if you want to wait for a thread to terminate, you need to specify which one you are waiting for by passing the proper tid to pthread_join(). The threads library can't figure it out for you.

To many people this isn't very good news because it can complicate programs that consist of more than two threads. Don't let it worry you. The POSIX threads standard provides all the tools you need to manage multiple threads gracefully. Actually, the fact that there is no parent/child relationship opens up some creative ways to use threads in our programs. For example, if we have a thread called thread 1, and thread 1 creates a thread called thread 2, it's not necessary for thread 1 itself to call pthread_join() for thread 2. Any other thread in the program can do that. This allows for interesting possibilities when you write heavily multithreaded code. You can, for example, create a global "dead list" that contains all stopped threads and then have a special cleanup thread that simply waits for an item to be added to the list. The cleanup thread calls pthread_join() to merge it with itself. Now your entire cleanup will be handled neatly and efficiently in a single thread.

Synchronized swimming

Now it's time to take a look at some code that does something a little unexpected. Here's thread2.c:

thread2.c

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int myglobal;

 void *thread_function(void *arg) {
  int i,j;
  for ( i=0; i<20; i++ ) {
    j=myglobal;
    j=j+1;
    printf(".");
    fflush(stdout);
   sleep(1);
    myglobal=j;
  }
  return NULL;
}

int main(void) {

  pthread_t mythread;
  int i;

  if ( pthread_create( &mythread, NULL, thread_function, NULL) ) {
    printf("error creating thread.");
    abort();
  }

  for ( i=0; i<20; i++) {
    myglobal=myglobal+1;
    printf("o");
    fflush(stdout);
    sleep(1);
  }

  if ( pthread_join ( mythread, NULL ) ) {
    printf("error joining thread.");
    abort();
  }

  printf("\nmyglobal equals %d\n",myglobal);

  exit(0);

}
```

Understanding thread2.c

This program, like our first one, creates a new thread. Both the main thread and the new thread then increment a global variable called myglobal 20 times. But the program itself produces some unexpected results. Compile it by typing:

```
$ gcc thread2.c -o thread2 -lpthread
```

And run it:

```
$ ./thread2
```

Here's some output from my system:

```
$ ./thread2
..o.o.o.o.oo.o.o.o.o.o.o.o.o.o..o.o.o.o.o
myglobal equals 21
```

Quite unexpected! Since myglobal starts at zero, and both the main thread and the new thread each increment it by 20, we should see myglobal equaling 40 at the end of the program. Since myglobal equals 21, we know that something fishy is going on here. But what is it?

Give up? OK, I'll show you why it happens. Take a look at thread_function(). Notice how we copy myglobal into a local variable called "j"? And see how we increment j, then sleep for one second, and only then copy our new j value into myglobal? That's the key. Imagine what happens if our main thread increments myglobal just **after** our new thread copies the value of myglobal into j. When thread_function() writes the value of j back to myglobal, it will overwrite the modification that our main thread made.

When writing threaded programs, you want to avoid useless side effects like the one we just looked at because they're a waste of time (except when you are writing articles on POSIX threads, of course :). Now, what can we do to eliminate this hassle?

Since the problem occurs because we copy myglobal to j and hold it there for one second before writing it back, we could try to avoid using a temporary local variable and incrementing myglobal directly. While this solution will probably work in this particular example, it is not correct. And if we were performing a relatively complex mathematical operation on myglobal instead of just incrementing it, it would certainly fail. But why?

To understand the problem, you need to remember that threads run simultaneously. Even on a uniprocessor machine (where the kernel uses time slicing to simulate true multitasking) we can, from a programmer's perspective, imagine both threads as executing at the same time. thread2.c has problems because the code in thread_function() relies on the fact that myglobal will not be modified during the ~1 second before it is incremented. We need some way for one thread to tell the other to "hold off" while it's making changes to myglobal. I'll show you exactly how to do this in my next article. See you then.

Related topics

- See documentation on Linux threads, by Sean Walton, KB7rfa
- In An Introduction to Pthreads-Tcl, see changes to Tcl that enable it to be used with POSIX threads
- Always go to your friendly Linux pthread man pages ("man -k pthread")
- Refer to the home page for POSIX and DCE threads for Linux
- See The LinuxThreads Library
- Take a look at David R. Butenhof's book Programming with POSIX Threads, in which he covers, among other things, the possible permutations of not using mutexes
- Check out W. Richard Stevens' book UNIX Network Programming: Network APIs: Sockets and XTI, Volume 1