Implementing a Thread Library on Linux

[Path: > Evan Jones' Scratch Pad > Software | Written by Evan Jones]

[2003-December-10 11:12]

When a program is executed on a computer, it is executed a line at a time, which each line containing a command to execute, or directions about which piece of code to execute next. Sometimes, this is not sufficient. There are many programs which must be able to do many things at once. For example, a GUI application must repaint the display and respond to user input while doing any processing, and web servers must be able to send information to hundreds of clients simultaneously. This is the problem that threads try to solve. Threads allow multiple parts of a program to run at the same time. All modern operating systems supply some sort of thread library, such as standard Posix threads. However, I was curious about how threads actually work. So armed with a little bit of knowledge about C and the Linux kernel, I went out to build my own thread library.

Multitasking

In traditional operating systems, there are two tools for multitasking provided by the operating system. The first is a process, and the second is a thread.

Processes

Each program is a seperate process. The operating system shares computing resources between all the processes that are running. It also keeps each process seperate, preventing one process from modifing another process's resources. However, it is quite common that many processes can belong to a single program. For example, the **Figure 1** shows some of the processes currently running on my Linux system, and there are four processes executing `mozilla-bin` . In this case, one process will need to communicate with another. This is facilitated through operating system routines for inter-process communication (IPC), such as signals (**`signal`**), shared memory (**`shmop`**), and pipes (**`pipe`**).

Process ID	Command
1	init
142	/sbin/syslogd
183	/usr/bin/X11/X
266	esd
270	sawfish
275	panel
277	gmc
279	gnome-terminal
291	bash
319	/usr/bin/mozilla-bin
341	/usr/bin/scite
344	/usr/bin/mozilla-bin
345	/usr/bin/mozilla-bin
346	/usr/bin/mozilla-bin
2568	ps

Figure 1: Linux System Process List

Threads

Inter-process communication is simple and easy to use when it is used occasionally. However, if there are many processes and many resources to be shared between them, the model quickly becomes cumbersome. Threads were created to make this sort of resource sharing simple and efficient. The concept is that a single process can have a number of threads, and everything is shared between them except the execution context (the stack and CPU registers). This way, if a shared resource is modified in one thread, the change is visible in all other threads. The disadvantage is that care must be taken to avoid problems where two threads try to access a shared resource at the same time.

Kernel Threads

Modern operating systems directly support threads in the kernel, which means that the scheduling of which process is supposed to run is done by the operating system. Kernel threads allow the operating system to schedule different threads to run on different processors in multiprocessor computers, which can be an enormous performance increase. The disadvantage of kernel threads is that each time the operating system switches between threads, a context switch, there is a penalty due to the overhead of saving and restoring the state of the threads. Therefore, many simultaneous threads will be slower than a single thread which distributes its time between different tasks. This is the reason that single threaded web servers, like **thttpd**, perform better than multi-threaded web servers, like **Apache**. The **JAWS research project** has an excellent analysis of the performance of various multitasking approaches with respect to web servers.

User Space Threads

Threads can also be built in user space. This means that a library or program is responsable for scheduling and executing threads. When this is done in user space, there is still a penalty for a context switch. However, the cost is less than an operating system context switch. Sometimes, user space threads are called fibers, to suggest that they are "lighter" than kernel threads. For the remainder of this article, I will refer to user space threads as fibers, and kernel threads as threads. Fibers have an additional advantage over kernel threads: Only one thread can modify a shared resource at a time, since only one fiber can be executing at a time. This means that some of the locks required for threads may not be needed. However, care is still required. As mentioned in "**Cooperative Task Management**", a 2002 USENIX paper, *"Cooperative task management [user space threads] avoids the concurrency problems of locks only if tasks can complete without having to yield control to any other task."* Fibers also cannot take advantage of multiprocessor systems. The author of GNU Pth, Ralf Engelschall, has an excellent paper about his work **implementing user space threads**, which covers the differences between different threading models in detail.

A Basic Thread Library

To play around with these concepts, a basic C thread library, **libfiber**, was written. It is implemented using two techniques for fibers and Linux kernel threads. This library provides an extremely simple implementation for creating, destroying and scheduling fibers or threads. It should only be used as an example for learning about how threads are implemented, since there are many issues with signals and synchronization. For real applications there are more polished libraries such as **pthreads** for kernel threads or **GNU Portable Threads (Pth)** for user space threads.

libfiber interface

void initFibers();

Initializes the internal structures inside the library. Should be called before using any library routines.

int spawnFiber(void (*func)(void));

Creates a new thread which will execute the given function. Returns zero on success, or nonzero on error.

void fiberYield();

Yield execution control to another fiber. This simply calls `sched_yield` in the kernel thread implementation.

void waitForAllFibers();

Waits for all the fibers to exit, and frees resources associated with each one. If called from a child fiber in the user space implementations, it will return once all other fibers have exited. If called from a child fiber in the kernel implementation, it will return immediately. Returns zero on success, or nonzero on error.

Figure 2 shows the main function from the sample application included with the library. It creates three fibers executing different functions, waits until they have all completed, then returns.

```
int main()
{
        // Initialize the fiber library
        initFibers();

        // Go fibers!
        spawnFiber( &fiber1 );
        spawnFiber( &fibonacchi );
        spawnFiber( &squares );

        // Since these are not preemptive, we must allow
them to run
        waitForAllFibers();

        // The program quits
        return 0;
}
```

Figure 2: Example program using the fiber library

Implementing Kernel Threads on Linux

Linux takes a unique approach to threads. On Linux, threads and processes are treated the same. They are both considered to be tasks. The difference is that threads share the same memory space, file mappings and (in general) signal handlers. There is an excellent post from Linus Torvalds to the **Linux Kernel Mailing List** that explains the **advantages of this implementation**. This means that in the process list in **Figure 1**, the processes may actually be threads. For example, the four mozilla-bin processes are actually threads inside a single process.

On Linux, kernel threads are created with the `clone` system call. This system call is similar to `fork` in that it creates a task which is executing the current program. However it differs in that `clone` specifies which resources should be shared. To create a thread, we call `clone` to create a task which shares as much as possible: The memory space, file descriptors and signal handlers. The signal to be sent when the thread exists is `SIGCHLD` so `wait` will return when a thread exits.

The first challenge is allocating a stack for the thread. The simplest solution, used in libfiber, is allocating the stack on the heap using `malloc`. This means that an estimate of the maximum stack size must be made. If the stack grows larger, memory corruption will occur. A solution used by **bb_threads**, is to create a barrier at the bottom of the stack with `mprotect`. This way, the thread will cause a segmentation fault when it overflows the stack. The best solution, used by the Linux pthreads implementation, is to use `mmap` to allocate memory, with flags specifying a region of memory which is allocated as it is used. This way, memory is allocated for the stack as it is needed, and a segmentation violation will occur if the system is unable to allocate additional memory.

The stack pointer passed to `clone` must reference the top of the chunk of memory, since on most processors the stack grows down. This is done by adding the size of the region to the pointer returned by `malloc`. To avoid a memory leak, the stack must be freed once the thread has exited. The libfiber library waits for threads to exit using **`wait`**, then frees the stack using **`free`**. **Figure 3** shows an example of creating a thread. See `libfiber-clone.c` in the **libfiber package** for the full implementation. For details about how Linux's pthread implementation creates threads, see "**The Fibers of Threads**" from Linux Magazine.

```
#include <malloc.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <sched.h>
#include <stdio.h>

// 64kB stack
#define FIBER_STACK 1024*64

// The child thread will execute this function
int threadFunction( void* argument )
{
        printf( "child thread exiting\n" );
        return 0;
}

int main()
{
        void* stack;
        pid_t pid;

        // Allocate the stack
        stack = malloc( FIBER_STACK );
        if ( stack == 0 )
        {
                perror( "malloc: could not
allocate stack" );
                exit( 1 );
        }

        printf( "Creating child thread\n" );

        // Call the clone system call to create the child
thread
	pid = clone( &threadFunction,
(char*) stack + FIBER_STACK,
		SIGCHLD | CLONE_FS |
CLONE_FILES | CLONE_SIGHAND |
CLONE_VM, 0 );
	if ( pid == -1 )
	{
		perror( "clone" );
		exit( 2 );
	}

	// Wait for the child thread to exit
	pid = waitpid( pid, 0, 0 );
	if ( pid == -1 )
	{
		perror( "waitpid" );
		exit( 3 );
	}

	// Free the stack
	free( stack );

	printf( "Child thread returned and
stack freed.\n" );

	return 0;
}
```

Figure 3: Simple clone Thread Example

Implementing Fibers Using makecontext

Modern Unix systems include library functions for manipulating the execution context, contained in `ucontext.h`. These functions are **`getcontext`**, **`setcontext`**, **`swapcontext`** and **`makecontext`**. Essentially, **`getcontext`** saves the current execution context, **`setcontext`** switches to the specified execution context, **`swapcontext`** will save the current context and switch to another and **`makecontext`** creates a new execution context. To create a new fiber, we call `getcontext` to retrieve the current context then modify the members of the `ucontext_t` struct to specify the new context. Again, we must allocate a new stack, but this time the library takes care of the direction of the stack growth. Then we call `makecontext` and specify the function and arguments to execute in the fiber.

The remaining challenge is that in this implementation, the fibers must explicitly yield control to allow other fibers to run. The `swapcontext` function can be used to stop executing one fiber and continue executing another. **Figure 4** shows a simple program which creates a child fiber and switches between the child and the parent. See `libfiber-uc.c` in the **libfiber package** for the full implementation.

```
#include <malloc.h>
#include <ucontext.h>
#include <stdio.h>

// 64kB stack
#define FIBER_STACK 1024*64

ucontext_t child, parent;

// The child thread will execute this function
void threadFunction()
{
        printf( "Child fiber yielding to parent"
);
        swapcontext( &child, &parent );
        printf( "Child thread exiting\n" );
        swapcontext( &child, &parent );
}

int main()
{
        // Get the current execution context
        getcontext( &child );
        // Modify the context to a new stack
        child.uc_link = 0;
        child.uc_stack.ss_sp = malloc(
FIBER_STACK );
        child.uc_stack.ss_size =
FIBER_STACK;
        child.uc_stack.ss_flags = 0;
        if ( child.uc_stack.ss_sp == 0 )
        {
                perror( "malloc: Could not
allocate stack" );
                exit( 1 );
        }

        // Create the new context
        printf( "Creating child fiber\n" );
        makecontext( &child,
&threadFunction, 0 );

        // Execute the child context
        printf( "Switching to child fiber\n" );
        swapcontext( &parent, &child );
        printf( "Switching to child fiber
again\n" );
        swapcontext( &parent, &child );

        // Free the stack
        free( child.uc_stack.ss_sp );

        printf( "Child fiber returned and stack
freed\n" );

        return 0;
}
```

Figure 4: Simple makecontext Fiber Example

Implementing Fibers Using longjmp

ANSI C provides functions for saving and restoring the execution context for graceful error handling. `setjmp` saves the current execution context, and `longjmp` returns to a previously saved context. The one new challenge to implementing fibers with these functions is that they do not provide a way to create a new stack. They simply restore a previous stack frame. To create the stack, we use sigaltstack to allocate an alternate stack for the signal handler, set up a signal handler, then call it on the new stack using `raise`. The signal handler saves its current state hen returns to get rid of the "signal handler context". Finally, the fiber is ready to run. **Figure 5** shows a minimal program that uses this technique to set up an alternate stack. See `libfiber-clone.c` in the **libfiber package** for the full implementation. This technique is used by **GNU Pth** to implement fibers in a very portable fashion.

```
#include <malloc.h>
#include <setjmp.h>
#include <signal.h>
#include <stdio.h>

// 64kB stack
#define FIBER_STACK 1024*64

jmp_buf child, parent;

// The child thread will execute this function
void threadFunction()
{
	printf( "Child fiber yielding to
parent\n" );
	if ( setjmp( child ) )
	{
		printf( "Child thread exiting\n"
);
		longjmp( parent, 1 );
	}
	longjmp( parent, 1 );
}

void signalHandler( int arg )
{
	if ( setjmp( child ) )
	{
		threadFunction();
	}

	return;
}

int main()
{
	stack_t stack;
	struct sigaction sa;

	// Create the new stack
	stack.ss_flags = 0;
	stack.ss_size = FIBER_STACK;
	stack.ss_sp = malloc( FIBER_STACK
);
	if ( stack.ss_sp == 0 )
	{
		perror( "malloc: Could not
allocate stack." );
		exit( 1 );
	}
	sigaltstack( &stack, 0 );

	// Set up the custom signal handler
	sa.sa_handler = &signalHandler;
	sa.sa_flags = SA_ONSTACK;
	sigemptyset( &sa.sa_mask );
	sigaction( SIGUSR1, &sa, 0 );

	// Send the signal to call the function on the new
stack
	printf( "Creating child fiber\n" );
	raise( SIGUSR1 );
```

```
        // Execute the child context
        printf( "Switching to child fiber\n" );
        if ( setjmp( parent ) )
        {
                printf( "Switching to child fiber
again\n" );
                if ( setjmp( parent ) == 0 )
longjmp( child, 1 );
        }
        else longjmp( child, 1 );

        // Free the stack
        free( stack.ss_sp );
        printf( "Child fiber returned and stack
freed\n" );
        return 0;
}
```

Figure 5: Simple longjmp Fiber Example

Conclusion

The implementation of libfiber can be examined to understand how threads and fibers are created by libraries which provide them. This can be useful when analyzing the performance impact of various multiprocessing techniques. The library was developed and tested on Linux, however, the fiber implementations should be portable across most Unix platforms. The thread implementation, on the other hand, is Linux specific, due to the `clone` system call.

Resources

- **Pthreads Tutorial Part 1**, **Part 2**, **Part 3** - How to use the pthreads library
- **LinuxThreads FAQ** - Frequently asked questions about Linux's pthread implementation

- **The Fibers of Threads** - An article explaining how Linux's pthread library is implemented
- **Bare-bones threads** [**local mirror**] - A simple Linux thread library that uses clone and provides mutexes
- **GNU Portable Threads (Pth)** - User space thread library for nearly any platform imaginable
- **Portable Multithreading** - A excellent paper describing how to create user space threads
- **Linux Kernel Mailing List** - The definitive resource for Linux operating system discussions, and development.
- **Kernel Traffic** - A summary of the activity on the Linux Kernel Mailing List
- **JAWS: Understanding High Performance Web Systems** - The performance implications of processes, threads and fibers
- **Cooperative Task Management** - A 2002 USENIX paper from Microsoft that describes the software engineering advantages and disadvantages of different concurrency models in great detail.

Comments

The following are responses to questions about this document, some of which resulted in some changes.

- 2002-10-21: **Re: About locking in multi-threaded environments**
- 2002-10-21: **Re: thread library [passing multiple parameters to new threads]**

Revision History

- **2003-12-10**: Whoops. Fixed another IBM thread link.
- **2003-02-20**: IBM's Posix Threads tutorial has moved. I've also linked to all three parts.
- **2002-10-24**: Clarified the language in the section discussing the need for locks with user space threads. Concurrency errors can occur with user space threads. However, code between two calls to yield() (or functions that in turn call yield) will always execute atomically from the perspective of the program. One sentence was modified to use softer language as follows: This means that ~~many~~ some of the locks required for threads ~~are not~~ may not be needed. A reference was added to the Microsoft Research paper on **Cooperative Task Management**, which has a good quote on this subject and is a good paper on this subject in general.
- **2002-10-24**: Added a comments section with responses to questions about this document.

[Previous: Google Contest SiteMapper | Up to Software | Next: thttpd Web Server Patches]