Threads

Paul Krzyzanowski
Rutgers University

January 30, 2012 [original September 21, 2010]

1 Introduction

When we looked at the concept of the *process*, we considered the distinction between a *program* and *process*. A process was a program in memory along with dynamically-allocated storage (the heap), the stack, and the execution context, which comprises the state of the processor's registers and instruction pointer (program counter).

If we take a closer look at the process, we can break it into two components:

1. The program and dynamically allocated memory.
2. The stack, instruction pointer, and registers.

The second item is crucial for the execution flow of the program. The instruction pointer keeps track of which instructions to execute next, and those instructions affect the registers. Subroutine call/return instructions as well instructions that push or pop registers on the stack on entry to or exit from a function call adjust the contents of the stack and the stack pointer. This stream of instructions is the process' **thread of execution**.

A traditional process has one thread of execution. The operating system keeps track of the memory map, saved registers, and stack pointer in the process control block and the operating system's scheduler is responsible for making sure that the process gets to run every once in a while.

2 Multithreading

A process may be **multithreaded**, where the same program contains multiple concurrent threads of execution. An operating system that supports multithreading has a scheduler that is responsible for preempting and scheduling all threads of all processes.

In a multi-threaded process, all of the process' threads share the same memory and open files. Within the shared memory, each thread gets its own stack. Each thread has its own instruction pointer and registers. Since the memory is shared, it is important to note that there is no memory protection among the threads in a process.

An operating system had to keep track of processes, and stored its per-process information in a data structure called a process control block (PCB). A multithread-aware operating system also needs to keep track of threads. The items that the operating system must store that are unique to each thread are:

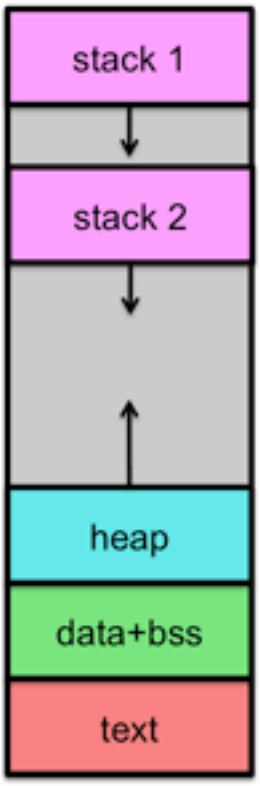

Figure 1: Memory map with two threads

- Thread ID
- Saved registers, stack pointer, instruction pointer
- Stack (local variables, temporary variables, return addresses)
- Signal mask
- Priority (scheduling information)

The items that are shared among threads within a process are:

- Text segment (instructions)
- Data segment (static and global data)
- BSS segment (uninitialized data)
- Open file descriptors
- Signals
- Current working directory

- User and group IDs

3 Advantages of threads

There are several benefits in using threads. Threads are more efficient. The operating system does not need to create a new memory map for a new thread (as it does for a process). It also does not need to allocate new structures to keep track of the state of open files and increment reference counts on open file descriptors.

Threading also makes certain types of programming easy. While it's true that there's a potential for bugs because memory is shared among threads, shared memory makes it trivial to share data among threads. The same global and static variables can be read and written among all threads in a process.

A multithreaded application can scale in performance as the number of processors or cores increases in a system. With a single-threaded process, the operating system can do nothing to make let the process take advantage of multiple processors. With a multithreaded application, the scheduler can schedule different threads to run in parallel on different cores or processors.

4 Thread programming patterns

There are several common ways that threads are used in software:

Single task thread This use of threading creates a thread for a specific task that needs to be performed, usually asynchronously from the main flow of the program. When the function is complete, the thread exits.

Worker threads In this model, a process may have a number of distinct tasks that could be performed concurrently with each other. A thread is created for each one of these work items. Each of these threads then picks of tasks from a queue for that specific work item. For example, in a word processing program, you may have a separate thread that is responsible for processing the user's input and other commands while another thread is responsible for generating the on-screen layout of the formatted page.

Thread pools Here, the process creates a number of threads upon start-up. All of these threads then grab work items off the same work queue. Of course, protections need to be put in place that two threads don't grab the same item for processing. This pattern is commonly found in multithreaded network services, where each incoming network request (say, for a web page on a web server) will be processed by a separate thread.

5 How the operating system manages threads

The operating system saved information about each process in a process control block (PCB). These are organized in a process table or list. Thread-specific information is stored in a data structure called a **thread control block** (**TCB**). Since a process can have one

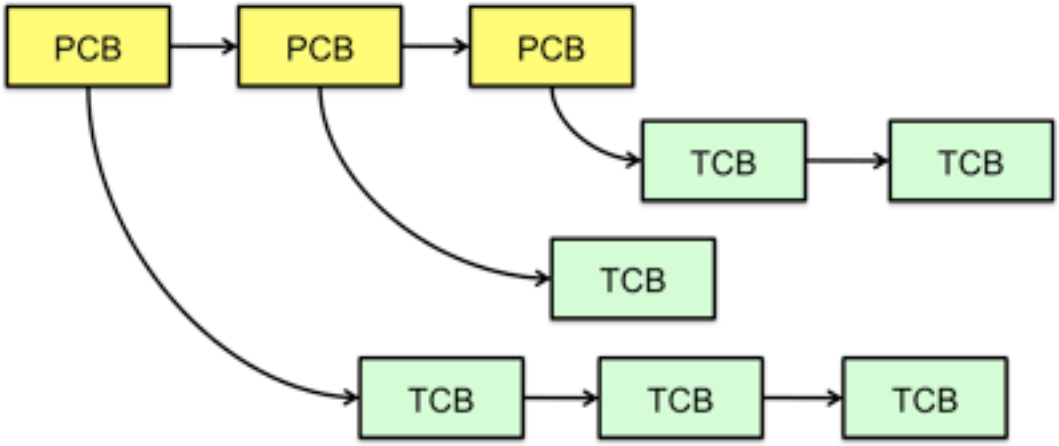

Figure 2: Thread control blocks

or more threads (it has to have at least one; otherwise there's nothing to run!), each PCB will point to a list of TCBs.

6 Scheduling

A traditional, non-multithreaded operating system scheduled processes. A thread-aware operating system schedules threads, not processes. In the case where a process has just one thread, there is no difference between the two. A scheduler should be aware of whether threads belong to the same process or not. Switching between threads of different processes entails a full context switch. Because threads that belong to different processes access different memory address spaces, the operating system has to flush cache memory (or ensure that the hardware supports process tags) and flush the virtual memory TLB (the translation lookaside buffer, which is a cache of frequently-used memory translations), unless the TLB also supports process tags. It also has to replace the page table pointer in the memory management unit to switch address spaces. The distinction between scheduling threads from the same or a different process is also important for hyperthreaded processors, which support running multiple threads at the same time but require that those threads share the same address space.

7 Kernel-level versus user-level threads

What we discussed thus far assumed that the operating system is aware of the concept of threads and offers users system calls to create and manage threads. This form of thread support is known as **kernel-level threads**. The operating system has the ability to create multiple threads per process and the scheduler can coordinate when and how they run. System calls are provided to control the creation, deletion, and synchronization of threads.

Threads can also be implemented strictly within a process, with the kernel treading the process as having a single execution context (the classic process: a single instruction pointer, saved registers, and stack). These threads are known as **user-level threads**. Users

typically link their program with a threading library that offers functions to create, schedule, synchronize, and destroy threads.

To implement user-level threads, a threading library is responsible for handling the saving and switching of the execution context from one thread to another. This means that it has to allocate a region of memory within the process that will serve as a stack for each thread. It also has to save and swap registers and the instruction pointer as the library switches execution from one thread to another. The most primitive implementation of this is to have each thread periodically call the threading library to yield its use of the processor to another thread — the analogy to a program getting context switched only when it requests to do so. A better approach is to have the threading library ask the operating system for a timer-based interrupt (for example, see the *setitimer* system call). When the process gets the interrupt (via the *signal* mechanism), the function in the threading library that registered for the signal is called and handles the saving of the current registers, stack pointer, and stack and restoring those items from the saved context of another thread.

One thing to watch out for with user-level threads is the use of system calls. If any thread makes a system call that causes the process to block (remember, the operating system is unaware of the multiple threads), then every thread in the process is effectively blocked. We can avoid this if the operating system offers us non-blocking versions of system calls that tend to block for data. The threading library can simulate blocking system calls by using non-blocking versions and put the thread in a waiting queue until the system call's data is ready. For example, most POSIX (Linux, Unix, OS X, *BSD) systems have a *O_NONBLOCK* option for the *open* system call that causes a *open* and *read* to return immediately with an *EAGAIN* error code if no data is ready. Also, the *fcntl* system call can set the *O_ASYNC* option on a file that will cause the process to receive a *SIGIO* signal when data is ready for a file. The threading library can catch this signal and "wake up" the thread that was waiting for that specific data. Note that with user-level threads, the threading library will have to implement its own thread scheduler since the non-thread-aware operating system scheduler only schedules at the process granularity.

8 Why bother with user-level threads?

There are several obstacles with user-level threads. One big one is that if one thread executes a system call that causes the operating system to block then the entire process (all threads) is blocked. As we saw above, this could be overcome if the operating system gives us options to have non-blocking versions of system calls. A more significant obstacle is that the operating system schedules the process as a single-threaded entity and therefore cannot take advantage of multiple processors or hyperthreaded architectures.

There are several reasons, however, why user-level threads can be preferable to kernel-level threads. All thread manipulation and thread switching is done within the process so there is no need to switch to the operating system. That makes user-level threading lighter weight than kernel-level threads. Because the threading library must have its own thread scheduler, this can be optimized to the specific scheduling needs of the application. Threads don't have to rely on a general-purpose scheduler of an operating system. Moreover, each multithreaded process may use its own scheduler that is optimized for its own needs. Finally, threading libraries can be ported to multiple operating systems, allowing programmers to write more portable code since there will be less dependence on the system calls of a particular operating

system.

9 Combining user and kernel-level threads

If an operating system offers kernel-level thread support, that does not mean that you cannot use a user-level thread library. In fact, it's even possible to have a program use both user-level and kernel-level threads. An example of why this might be desirable is to have the thread library create several kernel threads to ensure that the operating system can take advantage of hyperthreading or multiprocessing while using more efficient user-level threads when a very large number of threads is needed. Several user level threads can be run over a single kernel-level thread. In general, the following threading options exist on most systems:

1:1 purely kernel threads, where one user thread always corresponds to a kernel thread.

N:1 only kernel threads, where N user-level threads are created on top of a single kernel thread. This is done in cases where the operating system does not support multithreading or where you absolutely do not want to use the kernel's multithreading capabilities.

N:M This is known as **hybrid threading** and maps N user-level threads are mapped onto M kernel-level threads.

10 Example: POSIX threads

One popular threads programming package is **POSIX Threads**, defined as *POSIX.1c, Threads extensions* (also IEEE Std 1003.1c–1995). POSIX is a family of IEEE standards that defines programming interfaces, commands, and related components for UNIX-derived operating systems. Systems such as Apple's Mac OS X, Sun's (Oracle's) Solaris, and a dozen or so other systems are fully POSIX compliant and system such as most Linux distributions, OpenBSD, FreeBSD, and NetBSD are mostly compliant.

POSIX Threads defines an API (application programming interface) for managing threads. This interface is implemented as a native kernel threads interface on Solaris, Mac OS X, NetBSD, FreeBSD, and many other POSIX-compliant systems. Linux also supports a native POSIX thread library as of the 2.6 kernel (as of December 2003). On Microsoft Windows systems, is available as an API library on top of Win32 threads.

We will not dive into a description of the POSIX threads API. There are many good references for that. Instead, we will just cover a few of the very basic interfaces.

11 Create a thread

A new thread is created via:

```
pthread_t t;
pthread_create(&t, NULL, func, arg)
```

This call creates a new thread, *t*, and starts that thread executing function *func(arg)*.

12 Exit a thread

A thread can exit by calling *pthread_ exit* or just by returning from the first function that was invoked when it was created via *pthread_ create*.

13 Join two threads

Joining threads is analogous to the *wait* system call that was used to allow the parent to detect the death of a child process.

```
void *ret_val;
pthread_join(t, &ret_val);
```

The thread that calls this function will wait (block) for thread *t* to terminate. An important differentiator from the *wait* system call that was used for processes is that with threads there is *no parent-child relationship*. Any one thread may join (wait on) any other thread.

14 Stepping on each other

Because threads within a process share the same memory map and hence share all global data (static variables, global variables, and memory that is dynamically-allocated via *malloc* or *new*), **mutual exclusion** is a critical part of application design. Mutual exclusion gives us the assurance that we can create regions in which only one thread may execute at a time. These regions are called **critical sections**. Mutual exclusion controls allow a thread to grab a lock for a specific critical section (region of code) and be sure that no other thread will be able to grab that lock. Any other thread that tries to do so will go to sleep until the lock is released.

The pthread interface provides a simple locking and unlocking mechanism to allow programs to handle mutual exclusion. An example of grabbing and then releasing a critical section is:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
    ...
pthread_mutex_lock(&m);
/* modify shared data */
pthread_mutex_unlock(&m);
```

15 References

- Posix Threads Programming[1], *Blaise Barney*, Lawrence Livermore National Laboratory.

[1] https://computing.llnl.gov/tutorials/pthreads/

- POSIX threads explained, part 1[2], *Gentoo Linux Documentation*, (follow links for Part 2[3] and Part 3[4])
- POSIX[5], Wikipedia article
- Processes and Threads[6], *Microsoft MSDN*, September 23, 2010, 2010 Microsoft Corporation
- Apple Grand Central Dispatch (GCD) Reference[7], *Mac OS X Reference Library*, 2010 Apple Inc.
- Concurrency Programming Guide[8], *Mac OS X Reference Library*, 2010 Apple Inc.
- Intel Hyper-Threading Technology[9]: Your Questions Answered, Intel, May 2009.

[2] http://www.gentoo.org/doc/en/articles/l-posix1.xml
[3] http://www.gentoo.org/doc/en/articles/l-posix2.xml
[4] http://www.gentoo.org/doc/en/articles/l-posix3.xml
[5] http://en.wikipedia.org/wiki/POSIX
[6] http://msdn.microsoft.com/en-us/library/ms684841(v=VS.85).aspx
[7] http://developer.apple.com/library/mac/#documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html
[8] http://tinyurl.com/2brq43o
[9] http://software.intel.com/en-us/articles/intel-hyper-threading-technology-your-questions-answered/