Operating Systems and Architecture (CO527) Spring 2016

Processes and Threads

Fred Barnes and Radu Grigore

Operating systems ensure (1) that all programs access the processor, and (2) that one crashing program does not bring down others. To achieve these tasks, operating system implement two concepts, threads and processes. Two threads running at the same time should get, roughly, the same processor time. A process is a group of threads which are isolated from threads in other processes.

The informal notion of a program usually corresponds to an operating system process. There are exceptions. For example, Google Chrome uses one process for each tab. This way, it takes advantage of the isolation between processes: if one tab crashes, the other tabs of the browser continue running. Thus, one reason for implementing a program using multiple processes is to increase its reliability. Each process contains at least one thread, so any multi-process program is also a multi-thread program. Apart from reliability, there are two further reasons to implement programs using multiple threads: efficiency and code organisation.

The operating system sees a number of threads trying to run on several processor cores. The operating system does not know what each thread tries to achieve. If there are, say, 8 threads trying to run on 4 cores, then the operating system ensures that each thread gets access to the equivalent of half a core. Thus, if a program consists of more threads, then the program gets access to a bigger share of the processing resources. It should be noted, however, that having a bigger share of the processor does not automatically make programs faster. The reason is that the processor is only one of several computational resources: A program also needs access to memory and, perhaps, the file system and the network. In general, to speed up a program, one needs to look at which resource acts as a bottleneck. When the processor is the bottleneck, which is rarely the case, a program will generally run faster if it has multiple threads. (Although even this comes with the proviso that we don't know how to parallelize certain tasks: We don't know if NC = P.)

The other reason for using multiple threads is code organisation. Here, the standard example is that of a web server. A web server has to handle multiple clients at the same time. Yet, it is convenient to implement the server as if there was a single client. Then, you ask the operating system to run several copies of the same server, in parallel, one for each client. The alternative would be to explicitly handle multiple clients, for example by using the select function, as we saw in the lecture about using files.

Notice that the two reasons for using threads, efficiency and code organisation, are complementary. We can organise the code into threads even if there is only one processing core. But, we can gain efficiency from multiple threads only if there are multiple cores. Some people insist we should be using different words to refer to these different uses of threads: concurrency to refer to the efficiency-guided uses, and parallelism to refer to the code organisation-guided uses. However, although the two goals are clearly different, the terminology concurrency/parallelism is not entirely standard. At least in the context of garbage collection, these words are used in a completely different way. Terminology aside, it is important to realise that threads serve two different purposes.

Let us now look at how to use multiple processes and multiple threads on a POSIX operating system.

Processes

Here is a simple program that starts several worker processes:

```
setpgid(0, 0);
for (int i = 0; i < workers_count; ++i) if (fork() == 0) { do_work(); return; }
```

After a call to fork, the parent process is cloned. The clones differ in one thing: in the parent process, fork returns a positive integer, which is the identifier of the child process; in the child process, fork returns 0. Hence, the function do_work is called only by the newly forked children.

Just like threads are grouped into processes, processes are grouped into process groups. The call setpgid(0,0) creates a new process group, and makes the current process its leader. This comes in handy when you detect some irrecoverable error, and you want to terminate the main process and all worker processes. Because they are in the same process group, you can say

```
killpg(getpgrp(), SIGKILL);
```

The getpgrp gets the identifier of the current process group, and killpg sends the SIGKILL signal to all processes in the group.

Finally, the main process will typically wait for all the worker processes to terminate.

```
while (wait(NULL) != -1);
```

The call to wait returns −1 when there is no (more) child to wait for. It may also return −1 if some other error occurs. Be warned that all the snippets from above don't handle errors, which real code should do.

Occasionally, processes of the same program will need to communicate. There are many communication mechanisms: signals, pipes, named pipes, POSIX message queues, socket pairs; and also regular files, including memory mapped ones. We saw above an example of communication via signals: The default action on receiving the SIGKILL signal is to terminate the process, but a process could also register a handler that does something else. And, of course, there are other signals that could be sent.

Now let us look at another communication mechanism: POSIX message queues. This is an asynchronous communication mechanism: the sender and the receiver do not need to be synchronized. We can send a message as follows:

```
mqd_t q = mq_open("/message-queue-name", O_WRONLY | O_CREAT, S_IRWXU, NULL);
mq_send(q, "hello", 6, 0);
mq_close(q);
```

Message queue names must start with '/'. If the queue does not exist, it will be created (O_CREAT). If it will be created, then the current user will have rwx permissions (S_IRWXU). The last argument of mq_open can be used to specify the maximum number of messages in the queue and the maximum size of a message. Here, we use the defaults (NULL). To send a message, we specify the queue to which to send it (q), what is the message ("hello"), its length (6), and its priority (0).

To receive a message, we do something very similar:

```
#define buffer_size (1<<20)
char buffer[buffer_size];
mqd_t q = mq_open("/message-queue-name", O_RDONLY);
mq_receive(q, buffer, buffer_size, NULL);
mq_close(q);
```

As usual, be warned that real code must also handle exceptional situations, such as the case in which the queue does not exist and such as the case in which the buffer is too small to hold a message.

You can find a more or less complete example in the accompanying pgrep.c.

Threads

Processes are nicely isolated from one another by the operating system. But, on the negative side, forking a process is a fairly expensive operation, because the whole state of the process needs to be cloned. Linux does the cloning in a lazy fashion, so the price is not paid upfront. Threads are not nicely isolated from one another: they share the same state. But, on the positive side, creating a process is a fairly cheap operation. The difference in speed between processes and threads applies not only to creation: communication also tends to be more expensive between processes than it is between threads. So, let's take a look at how to use threads.

Here we create a number of identical threads:

```
pthread_t threads[thread_count];
for (int i = 0; i < thread_count; ++i) {
  pthread_create(&threads[i], NULL, thread_main, NULL);
}
```

The function thread_main is the entry point of the threads, and we need to implement it.

```
void * thread_main(void * unused) { ++counter; return NULL; }
```

Our threads increment a counter and then terminate. Recall that, unlike processes, threads share the same state — they operate in the same virtual memory space. So, if counter is some global variable, then all threads refer to the same place in memory. Now let's wait for all threads to finish:

```
for (int i = 0; i < thread_count; ++i) {
  pthread_join(threads[i], NULL);
}
```

Observe that we have the following correspondence between operations on processes and operations of threads:

Processes	Threads	Effect
fork()	pthread_create(&id, NULL)	spawns a new process/thread
waitpid(id, 0)	pthread_join(id, NULL)	waits for process/thread id to terminate

But recall there are differences. The most important one is that fork clones the virtual memory space, but pthread_create does not. Now, after joining all the threads, let us print the value of the counter.

```
printf("%d", threads_count);
```

If the counter starts at 0, then we expect the printed value to be threads_count, because the each thread increments the counter once. Alas, it turns out that counter is *sometimes* a bit smaller than threads_count. What is going on? The issue is that the increment operation is not atomic. It involves fetching data from memory to a register in the processor, actually incrementing the value, and then moving the data back to memory. It is possible, although not very likely, that a thread is interrupted in the middle of this process. When the interrupted thread continues, it will operate on a stale value of counter. And all modifications done by other threads to counter are lost because the interrupted thread now writes back to memory an old value of counter.

To ensure that counter is incremented properly, we can use a mutex. We bracket the increment operation by locking and unlocking a mutex.

```
pthread_mutex_lock(&counter_mutex);
++counter;
pthread_mutex_unlock(&counter_mutex);
```

The mutex is initialised as follows:

```
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;
```

What does a mutex do? When a thread locks the mutex, we say that it *acquires* the mutex; when a thread unlocks a mutex, we say that it *releases* the mutex. Imagine the mutex as a token that sits on a table, and threads as people walking around the table. Acquiring a mutex means taking the token from the table and holding it; releasing a mutex means putting the token back on the table. The main guarantee offered by a mutex is that only one thread can hold it, at any time. In our case, since all threads that operate on counter do so while holding counter_mutex, it follows that the operations on counter are atomic: they appear as if they are done in one step.

Now observe what is happening here. On the one hand, communication between threads would appear to be simpler: We do not need to instantiate a message queue and call special functions — we simply modify variables which are shared and visible to all threads. On the other hand, if we do not do some extra work to synchronize threads, our modifications to shared variables can have surprising effects.

Mutexes are just one synchronization primitive. Another one are *condition variables*. A call to

```
pthread_cond_wait(&condition_variable, &mutex);
```

will release mutex (which must be held), wait until the condition variable is signaled, and then reaquire mutex and continue. To signal a condition variable, you call

```
pthread_cond_signal(&condition_variable);
```

A combination of a mutex with a condition variable is called a *monitor*.

It's worth observing that Java objects are essentially monitors. Each Java object has associated a mutex and a condition variable, which are not directly accessible. However, adding the synchronized keyword on a method, has the same effect as locking the mutex (associated with object this) at the beginning of the method, and unlocking the mutex at the end of the method. Finally, the wait and notify methods of java.lang.Object have exactly the same effect as pthread_cond_wait and pthread_cond_signal. Of course, the pthreads interface is a lot more flexible (and, hence, potentially confusing). For example, (1) you can lock a mutex in one function and unlock in a different one; and (2) you can associate a condition variable with one mutex, and later in the execution associate it with a different mutex.

Another interesting synchronisation primitive is the readers–writer lock. The relevant functions are rdlock, wrlock, and unlock — all prefixed by pthread_rwlock_. So, a readers–writer lock is much like a mutex, except it has two types of locking. In our analogy with the object on the table, you can imagine that one may 'hold' the token in two ways. One way is to simply touch the token, without moving it from its place on the table. Multiple people can touch the token at the same time, as long as they don't move it. The other way to hold a token is the traditional one: you take it of the table and hold it in your hand. Only one person at a time can do this; moreover, no other person can touch the token while you took it off the table. To just touch the token, you call pthread_rwlock_rdlock. To take it off the table, you call pthread_rwlock_wrlock. If you want to stop touching the token, of if you want to place the token back on the table, you call pthread_rwlock_unlock. But why the operations called rdlock (acquire read lock) and wrlock (acquire write lock). That's because of the typical way a RW-lock is used. It is used to protect some variable or data structure, which we can inspect and modify. If we just inspect it, then we acquire the RW-lock in read mode. If we also modify it, then we acquire the RW-lock in write mode.

Exercises

1. Implement a variant of pgrep that uses threads instead of processes. You will have to implement a concurrent (message) queue, using synchronization primitives such as mutexes.
2. ► What is the main difference between inter-thread communication and inter-process communication?
3. ► What is a readers-writer lock?
4. ► What is a condition variable?
5. Read 'man mq_overview'.
6. Read 'man pthreads'. Which other synchronisation primitives are there available in pthreads?

References

[1] Brouwer, *The Linux Kernel*, *Processes*, 2003