CMPSCI 377 Operating Systems **Fall 2008**

Lecture 6: September 18

Lecturer: Prashant Shenoy *Scribe: Shashi Singh*

6.1 Interprocess Communication

Remember that by using fork() we can split one process into two, and that the input of each process will be basically whatever is the state of the program right before the fork(). Also, notice that after the fork() each process is free to follow its own path of execution, and that its output will be given by whatever value is return via the exit() call. Now, considering that a machine can have several processes running in parallel, we might want to consider make them communicate with each other. On last class we discussed some of the possibilities for doing that: signals (sending/receiving simple integer numbers), mmap (implicit communication by memory sharing), pipes (unidirectional communication channels) and sockets (explicit message passing).

6.2 Threads

First, remember that different processes keep their own data on a distinct address spaces. Threads, on the other hand, explicitly share their entire address space with one another. Although this can make things a lot faster, it comes with the cost of making programming **a lot** more complicated.

In Unix/POSIX, the threads API is composed by two main calls:

- pthread_create(), which starts a separate thread;
- pthread_join(), which waits for a thread to complete.

Notice that the general syntax for using these is:

```
pid = pthread_create(&tid, NULL, function_ptr, argument);
pthread_join(tid, &result);
```

Example:

```
void *run(void *d)
{
     int q = ((int) d);
     int v = 0;
     for (int i=0; i<q; i++)
         v = v + some_function_call();
     return (void *) v;
}
```

```
int main()
{
   pthread_t t1, t2;
   int *r1, *r2;
   pthread_create(&t1, NULL, run, (int *)100);  // the last parameter is a hack;
        // it *should* be a pointer, but we can pass the desired data (say, an int)
        // as if it were a pointer

   pthread_create(&t2, NULL, run, (int *)666);

   pthread_join(t1, (void **) &r1);
   pthread_join(t2, (void **) &r2);

   cout << r1= << *r1 << r2= << *r2 << endl;
}
```

Notice that the above threads maintain *different* stacks and different sets of registers; except for those, however, they share *all* their address spaces. Also notice that if you were to run this code in a 2 core machine, it would be expected that it ran sort of twice as fast as it would in a single core machine. If you ran it in a 4 core machine, however, it would run as fast as in the 2 core machine, since there would be no sufficient threads to exploit the available parallelism.

6.2.1 Processes vs threads

One might argue that in general processes are more flexible than threads. For one thing, they can live in two different machines, and communicate via sockets; they are easy to spawn remotely (eg: ssh foo.cs.umass.edu "ls -l"); etc. However, processes requires explicit communication and risky hackery. Threads also have their own problems: because they communicate through shared memory, they must run on same machine and require thread-safe code. So even though threads they are faster, they are much harder to program. In a sense, we can say that processes are far more robuts than threads, since they are completely isolated from other another. Threads, on the other hand, are not that safe, since whenever one thread crashes the whole process terminates.

When comparing processes and threads, we can also analyse the context switch cost. Whenever it is needed to switch between two processes, we must invalidate the TLB cache (the so called *TLB shootdown*). This of course makes everything slower. When we switch between two threads, on the other hand, it is not needed to invalidate the TLB, because all threads share the same address space, and thus have the same contents in the cache. In other words, the cost of switching between threads is **much** smaller than the cost of switching between proceses.

6.2.2 Kernel Threads and User-Level Threads

OS-managed threads are called kernel-level threads or lightweight processes. All thread operations are implemented in the kernel. The OS schedules all of the threads in the system. Example - Solaris: lightweight processes (LWP). Kernel-level threads make concurrency much cheaper than processes. This is because, as compared to processes, there is much less state to allocate and initialize. However, for fine grained concurrency, kernel-level threads still suffer from too much overhead. Thread operations still require system calls (ideally we want thread operations to be as fast as a function call). Kernel-level threads have to be

general to support the needs of all programmers, languages, runtimes, etc. For such fine-grained concurrency, we need even *cheaper* threads. To make threads cheap and fast, they need to be implemented at user level. User-level threads are managed entirely by the run-time system (user-level-library). A thread is simply represented by a program counter, registers, stack, and small thread control block (TCB). Creating a new thread, switching between threads, and synchronizing threads are done via function calls (without any kernel involvement). User-level threads are about 100 times faster than kernel threads. But, user-level threads are not a perfect solution. They are *invisble* to the OS. As a result, the OS cana make poor decisions like - scheduling a process with idle threads; blocking a process whose thread initiated an I/O, even though the process has other threads that can execute; unscheduling a process with a thread holding a lock, even when other threads do not hold any locks. Solving this problem requires communication between the kernel and the user-level thread manager.