CMPSCI 691W Parallel and Concurrent Programming **Spring 2006**

Lecture 2: February 6

Lecturer: Emery Berger *Scribe: Richard Chang*

2.1 Overview

This lecture gives an introduction to processes and threads. Specifically, it covers how both can be used for parallel programming and concurrency. Both can be used to hide latency, maximize CPU utilization and handle mutiple, asynchronous events. Issues such as programming style, communication, and synchronization are discussed. The trade-offs of using either are discussed, and a bake-off of using processes vs. threads is presented.

2.2 Processes

A process can be thought of as a program in execution running on an operating sytem. It consists of an execution context (program counter, registers), an address space, an open files list, process id, group id, etc. Processes can be used for parallel programming by spawning processes to execute concurrently and using some form of inter-process communication to allow them to share data. We will examine the APIs for creating processes, methods of inter-process communication (IPC), and some example programs that illustrate these concepts.

2.2.1 Process API

On UNIX operating systems new processes are created using the `fork()` [6] system call. `Fork()` creates a new copy of the current process. The original process (parent) and the newly created process (child) execute the same program that the parent process was prior to `fork()` being called, but the return value of the `fork()` call will differ in the parent and child processes. The return value of the `fork()` in the parent process will be the process id (pid) of the child process. The return value of the `fork()` in the child process will be 0. This allows the programmer to specify different behavior for the parent process and the child process.

Because processes created using `fork()` continue to execute the same program that their parent was, the `exec()` [6] system call is often used to replace the current process with a executable program after `fork()` has been called. Figure 2.1 shows the source code for a C++ program that uses `fork()` and `exec()` to create a new process and execute a new program. On line 8, the name of the program to be executed by the child process is read from `stdin` using a call to `gets()`. Then a call to `fork()` is made which creates a new process which is a copy of the current process. Both the child and the parent execute concurrently. The if statement on line 10, checks the return value of the `fork()` call. If the return value is 0, then the process currently executing is the child, and a call to `execlp()` is made to run the program whose name was read from `stdin`. In the case when the return value of the `fork()` call was non-zero, then the currently executing process is the parent and the value returned by `fork()` is the pid of the child process. The parent process will sleep for 1 minute, and then wait until the child process terminates.

```c
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>

main() {
  int parentID = getpid(); /* ID of this process */
  char prgname[1024];
  gets(prgname); /* Read the name of the program we want to start */
  int cid = fork();
  if (cid == 0) { /* I'm the child process */
    execlp( prgname, prgname, 0); /* Load the program */
    /* If the program named prgname can be started, we never get
       to this line, because the child program is replaced by prgname */
    printf("I didn't find program %s\n", prgname);
  } else { /* I'm the parent process */
    sleep(1); /* Give my child time to start. */
    waitpid(cid, 0, 0); /* Wait for my child to terminate. */
    printf("Program %s finished\n", prgname);
  }
}
```

Figure 2.1: An example program using `fork()` and `exec()`

On Windows operating systems, processes are not created by forking the current process. Instead, new processes are created using the function `CreateProcess()` [2] which takes 10 arguments that specify parameters such at the name of the application to execute and process attributes.

2.2.2 Translation Lookaside Buffer

Translation lookaside buffer (TLB) is a fast, fully associative memory that is used as a buffer for virtual address to physical address translation [7]. Entries in a process's page table are buffered in TLB in order to speed up memory address translation. If a page number is found in TLB, then the frame number can quickly be determined, but if a page number is not found in a TLB and the page table in memory has to be queried, and a TLB miss has occured which can lead to a huge performance loss.

Because the entries in a TLB are process specific, whenever a context switch occurs all of the TLB entries become invalid. This leads to what is a called a *TLB shootdown*. Initially after a context switch, the entries in the TLB are incorrect because processes have distinct address spaces. This means that memory accesses after a process context switch will be very expensive because of all of the TLB misses that occur.

2.2.3 Copy-on-write

When `fork()` is called, conceptually all resources of the parent process are copied for the child process to have its own address space, execution context, etc. This would potentially mean that all of the page frames used by the parent would have to be copied, and then if `exec()` was called to run a new program, all those copied pages for the child would be invalidated. In order to avoid this, `fork()` is usually implemented using *copy-on-write* [1]. Instead of copying the page frames of the parent as new copies for the child, the parent

```
#include <unistd.h>

int main() {
  int pfds[2];
 pipe(pfds);
 if (!fork()) {
  close(1);     /* close normal stdout*/
 dup(pfds[1]); /* make stdoutsame as pfds[1] */
 close(pfds[0]); /* we don't need this */
 execlp("ls", "ls", NULL);
 } else {
  close(0);     /* close normal stdin*/
 dup(pfds[0]); /* make stdinsame as pfds[0] */
 close(pfds[1]); /* we don't need this */
 execlp("wc", "wc", "-l", NULL);
 }
}
```

Figure 2.2: An example program using pipes for IPC

and child processes initially share page frames. While these page frames are shared, their data cannot be modified, so after a call to `fork()` all shared page frames are marked as read-only. When either process attempts to write to a read-only shared page a new copy is made for that process. The original page frame is then writable to the other process.

2.2.4 Inter-process Communication

Processes can communicate using the system calls we have already seen. One can think of the input to a process as its state before calling `fork()`. Processes can provide simple output by passing an integer argument to the `exit()` function. This value can be read by another process using the `wait()` system call.

There are also methods of communication that allow running processes to communicate during execution. These methods of inter-process communication (IPC) include signals, pipes, sockets, and mmap.

Signals allow processes to send and receive integer values. While there are user-defined signals in UNIX, signals are not terrible useful for parallel or concurrent programming. They are mainly intended for interupts and not general communication.

Using pipes is a method of IPC that is easy and fast. They can be used much like using pipes on the UNIX commandline to communicate between processes. Figure 2.2 shows an example program that uses pipes to redirect the output of a call to `ls` to be the input to a call to `wc -l`. This has the same effect as running `ls | wc -l` on the commandline in UNIX.

Sockets can be used for IPC, but they do require explicit message passing. One benefit of sockets is that processes that are communicating over sockets can be distributed over networks. Running a program written using sockets on a local machine is usually quite efficient while giving the programmer the option to distribute that program over a network.

`Mmap()` is a UNIX system call that allows files to be mapped to memory. This system call can be used for IPC by having multiple processes map the same file to their own distinct address spaces [6]. This way

```
#include <pthread.h>

void * run (void * d) {
  intq = ((int) d);
  intv = 0;
 for (inti = 0; i < q; i++) {
   v = v + expensiveComputation(i);
 }
 return (void *) v;
}

main() {
  pthread_tt1, t2;
  intr1, r2;
  pthread_create(&t1, run, 100);
  pthread_create(&t2, run, 100);
  pthread_join(&t1, (void *) &r1);
  pthread_join(&t2, (void *) &r2);
  printf(r1 = %d, r2 = %d\n, r1, r2);
}
```

Figure 2.3: An example program using pipes for IPC

changes made in the address space of one process that map to the shared file would be reflected in the others that are mapping the same file. Because the mapped file is in memory, disk I/O is avoid and this is relatively efficient. Synchronization can be handled by the `flock()` system call. When a process wants to ensure that no other processes are writing to the shared file, it can obtain a file lock on that file using `flock()`. It should be noted that calls to `flock()` are very expensive.

2.3 Threads

A thread consists of a thread ID, program counter, register values and a stack[7]. Unlike processes which each have a distinct address space, threads share the same address space, files, sockets, etc. Similarly to processes, threads can be used for parallel programming. We will now discuss how threads can be created, how they communicate, and will see an example program using threads on a UNIX operating system.

2.3.1 Threads API

On UNIX operating systems, the threads API used is called `pthreads`, which stands for POSIX threads. Threads are created using the function `pthread_create()` which takes as an argument the name of the function that should be excuted as a separate thread. The function `pthread_join()` is used to wait for a thread to complete. All threads created using `pthread_create()` in a given process execute within that process.

Figure 2.3 shows the an example program that creates two new threads to execute an expensive computation. Because the pthreads API[5] specifies that arguments passed to a function to be started as a new thread

must be a single void * pointer, the function `run()` is defined to take a void * pointer. The `main()` function creates two threads that will run for 100 iterations each. Then, the main thread waits for both threads to finish executing and prints their results.

In Windows there is a function that is used to create threads called `CreateThread()`[3] which takes 6 parameters that specify attributes like function to be executed and stack size.

2.3.2 Thread Communication

In threads, everything is shared except stacks, registers, and thread-specific data. The old way of accessing this thread-specific data was to use the `pthread_setspecific` and `pthread_getspecific` to access and modify data that a programmer wanted to be specific to each thread, and thus not shared. A newer way to achieve the same result is to declare a variable using the `static __thread` modifier. This type of declartion means that variables defined in this way will be thread-specific.

Because data is shared among threads by default, updates to this data must be sychronized. Mutual exclusion to allow only one thread in a critical section are a time can be enforced using calls to `pthread_mutex_lock(&l)` and `pthread_mutex_unlock(&l)`. Critical sections of code that contain updates to shared data can be wrapped in a pair of those calls to obtains a lock and then release it.

2.4 Bake-off

There are trade-offs in almost all design descisions, and the same can be said of using either process or threads for parallel programming. There is not one correct answer for all situations. Whether it is best threads or processes for parallel programming is situation dependent.

2.4.1 Performance

Much of the performance of threads and processes is determined by the time and work done when a context switch occurs to execute a new thread or process.

Context switches for threads a much cheaper because the only data that needs to be stashed and loaded is the registers, the program counter, stack pointer. All other data is shared amongst threads.

For processes all of that data must be stashed and loaded, plus the process context must be stashed and loaded. The TLB shootdown mentioned earlier occurs as well, which causes performance hits every time a page in memory is accessed until the TLB is repopulated with entries for the new process. Because context switches for processes are so expensive, longer time quanta are required to overcome the cost of the context switch. There is a trade-off between time quanta and system responsiveness. Longer time quanta usually means the system is less repsonsive.

2.4.2 Flexibility

Processes are much more flexible than using threads. It is very easy to spawn processes remotely. Parallel programming using sockets and processes can very easily be distributed across a local network or the internet. One downside to using processes is that communication must be done explicitly (sockets) or through some kind of hack (mmap).

Threads, on the other hand, communicate through memory and must be on the same machine which reduces their flexibility. Also, many programmers find it difficult to ensure that their code is *thread-safe*, which means that their code maintains data consistency even when being executed concurrently by mutiple threads[7].

2.4.3 Robustness

Processes in general are much more robust because they are isolated from other processes. In principle if one process dies then the other processes which are executing can just continue to execute with no effect. Apache[4], an open source web server, comes in two versions. The Apache 1.x branch is implemented using multiple processes to handle requests from users. This allows for more robustness. If a user's request causes a process to die, then the server can continue to execute and serve other users' requests.

Conversely, threads are not as robust as processes. If a thread crashes because of a deference of NULL for example, then the entire process terminates. Apache version 2.x is implemented using multiple threads to handle requests from users. The motivation behind this design decision is to increase performance because as we have already seen context switches from threads are much cheaper than for processes. The downside of this descision is the loss in robustness. If a thread crashes, the whole server (process) might terminate.

References

[1] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2002.

[2] Microsoft Corporation.
process and thread functions: Createprocess, 2005.
http://msdn.microsoft.com/library/default.asp?
url=/library/en-us/dllproc/base/createprocess.asp.

[3] Microsoft Corporation.
process and thread functions: Createthread, 2005.
http://msdn.microsoft.com/library/default.asp?
url=/library/en-us/dllproc/base/createthread.asp.

[4] The Apache Software Foundation. Welcome! the apache http server project, 2005. http://httpd.apache.org/.

[5] Lawerence Livermore National Laboratory. Posix threads programming, 2006. http://www.llnl.gov/computing/tutorials/pthreads/.

[6] M. McKusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

[7] A. Silberschatz, P. Galvin, and G. Gagne. *Applied Operating System Concepts*. John Wiley and Sons, 2000.