CMPSCI 377 Operating Systems **Fall 2008**

Lecture 5: September 16

Lecturer: Prashant Shenoy *Scribe: Shashi Singh*

5.1 Process Management

5.1.1 Process

A process is an instance of a computer program that is being sequentially executed by a computer system that has the ability to run several computer programs concurrently. A computer program itself is just a passive collection of instructions, while a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several windows of the same program typically means more than one process is being executed. The state of a process consists of - code for the running program, its static data, heap and the heap pointer (HP), program counter, stack and the stack pointer, value of CPU registers, set of OS resources in use (list of open files etc), process execution state (new, ready, running etc).

5.1.2 Process Execution State

Processes go through various process states which determine how the process is handled by the operating system kernel. The specific implementations of these states vary in different operating systems, and the names of these states are not standardised, but the general high-level functionality is the same. When a process is first started/created, it is in *new* state. It needs to wait for the process scheduler (of the operating system) to set its status to "new" and load it into main memory from secondary storage device (such as a hard disk or a CD-ROM). Once it is loaded into memory it enters *ready* state. Once the process has been assigned to a processor by a short-term scheduler, a context switch is performed (loading the process into the processor) and the process state is set to *running* - where the processor executes its instructions. If a process needs to wait for a resource (such as waiting for user input, or waiting for a file to become available), it is moved into the em wating state until it no longer needs to wait - then it is moved back into the em ready state. Once the process finishes execution, or is terminated by the operating system, it is moved to the *terminated* state where it waits to be removed from main memory. The OS manages multiple active process using *state queues*.

5.1.3 Process Control Block

A Process Control Block is a data structure in the operating system kernel containing the information needed to manage a particular process. The PCB is "the manifestation of a process in an operating system". A PCB will include: the identifier of the process (a process identifier, or PID); register values for the process including the program counter; the address space for the process; priority; process accounting information, such as when the process was last run, how much CPU time it has accumulated, etc; pointer to the next PCB i.e. pointer to the PCB of the next process to run; I/O Information (i.e. I/O devices allocated to this process, list of opened files, etc). Since the PCB contains the critical information for the process, it must be

kept in an area of memory protected from normal user access. In some operating systems the PCB is placed in the beginning of the kernel stack of the process since that is a convenient protected location.

5.1.4 Process State Queues

The OS maintains the PCBs of all processes in *state queues*. PCBs of all processes in the same *execution state* is placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue. The OS can use different policies to manage each queue (FIFO, Round Robin, Priority etc). Each I/O device has its own wait queue.

5.1.5 Context Switch

A context switch is the computing process of storing and restoring the state (context) of a CPU such that multiple processes can share a single CPU resource. The context switch is an essential feature of a multitasking operating system. Context switches are usually computationally intensive and much of the design of operating systems is to optimize the use of context switches. There are three scenarios where a context switch needs to occur: multitasking, interrupt handling, user and kernel mode switching. In a context switch, the state of the first process must be saved somehow, so that, when the scheduler gets back to the execution of the first process, it can restore this state and continue. The state of the process includes all the registers that the process may be using, especially the program counter, plus any other operating system specific data that may be necessary. Often, all the data that is necessary for state is stored in one data structure, called a process control block.

5.1.6 Creating a Process: *fork* System Call

A process can create other processes to do work. In computing, when a process forks, it creates a copy of itself, which is called a "child process". The original process is then called the "parent process". More generally, a fork in a multithreading environment means that a thread of execution is duplicated, creating a child thread from the parent thread. Under Unix and Unix-like operating systems, the parent and the child processes can tell each other apart by examining the return value of the fork() system call. In the child process, the return value of fork() is 0, whereas the return value in the parent process is the PID of the newly-created child process. The fork operation creates a separate address space for the child. The child process has an exact copy of all the memory segments of the parent process, though if copy-on-write semantics are implemented actual physical memory may not be assigned (i.e., both processes may share the same physical memory segments for a while). Both the parent and child processes possess the same code segments, but execute independently of each other. The child process usually executes the *exec* function to do something useful. The exec functions of Unix-like operating systems are a collection of functions that causes the running process to be completely replaced by the program passed as argument to the function. As a new process is not created, the process ID (PID) does not change across and execute, but the data, heap and stack of the calling process are replaced by those of the new process. In the *execl*, *execlp*, *execv*, and *execvp* calls, the child process inherits the parent's environment. The parent process, after creating the child process, may issue a wait system call, which suspends the execution of the parent process while the child executes. When the child process terminates, it returns an exit status to the operating system, which is then returned to the waiting parent process. The parent process then resumes execution.

5.1.7 Process Termination

On process termination, OS reclaims all resources assigned to the process. In Unix, a process cana terminate itself using the *exit* system call. A process can terminate another process (if it has the privilege to do so) using the *kill* system call.

5.1.8 Cooperating Processes

Cooperating processes work with each other to accomplish a single task. This may improve performance by overlapping activities or performing work in parallel. It helps tp easily share information between tasks. It can enable an application to achieve a better program structure as a set of cooperating processes, where each is smaller than a single monolithic program. Distributed systems are examples of cooperating processes in action. In computer science, the producer-consumer problem (also known as the bounded-buffer problem) is a classical example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data (i.e. removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

5.1.9 Interprocess Communication

Inter-Process Communication (IPC) is a set of techniques for the exchange of data among two or more threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for message passing, synchronization, shared memory, and remote procedure calls (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated.

5.1.9.1 Message Passing

message passing is a form of communication used in interprocess communication. Communication is made by the sending of messages to recipients. Each process should be able to name the other processes. The consusmer is assumed to have an infnite buffer size. The sender typically uses *send()* system call to send messages, and the receiver uses *receive()* system call to receive messages. These system calls can be either synchronous or asynchronous.

5.1.9.2 Shared Memory

Shared memory is a memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them. One process will create an area in RAM which other processes can access (this is typically done using system calls *mmap*, *shmget* etc). Since both processes can access the shared memory area like regular working memory, this is a very fast way of communication (as opposed to other mechanisms of IPC). On the other hand, it is less powerful, as for example the communicating processes must be running on the same machine (whereas other IPC methods can use a computer network), and care must be taken to avoid issues if processes sharing memory are running on separate CPUs and the underlying architecture is not cache coherent. Since shared memory is inherently non-blocking, it can't be used to achieve synchronization.