CMPSCI 377 Operating Systems **Fall 2012**

Lecture 4: September 13

Lecturer: Prashant Shenoy *TA: Sean Barker & Demetre Lavigne*

4.1 Process State

4.1.1 Process

A process is a dynamic 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 (text segment), its static data, its heap and the heap pointer (HP) where dynamic data is kept, program counter (PC), stack and the stack pointer (SP), value of CPU registers, set of OS resources in use (list of open files etc.), and the current process execution state (new, ready, running etc.). Some state may be stored in registers, such as the program counter.

4.1.2 Process Execution States

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 the *ready* state. Once the process has been assigned to a processor by the OS 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, I/O, or waiting for a file to become available), it is moved into the *waiting* state until it no longer needs to wait - then it is moved back into the *ready* state. The ready state means that the process is ready to run, but some other process is already running. A process may also transition from the running state to the ready state due to a context switch (the OS has scheduled another process even though the current process hasn't finished). 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*.

4.1.3 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. A PCB is created in the kernel whenever a new process is started. The OS maintains a queue of PCBs, one for each process running in the 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; scheduling information such as 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. The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

4.1.4 Process State Queues

The OS maintains the PCBs of all processes in *state queues*. The OS maintains a queue for each of the states described in Section 4.1.2. PCBs of all processes in the same *execution state* are 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 queue for each state is unbounded in length except for the run queue. The run queue's length is always less than or equal to the number of processor in the system.

Some queues such as the wait or ready queues may control many PCB entries. The OS can use different policies to manage each queue (FIFO, Round Robin, Priority etc). Each I/O device has its own wait queue. The run queue, however, can only have one entry per processor core on the system (since each core can only run a single process at a time). The OS scheduler determines how to move processes between the ready and run queues.

4.2 Process Management

4.2.1 Context Switch

A context switch is the step required to move a process between the run and ready queues. The context switch is an essential feature of a multitasking operating system so that multiple processes can share a single CPU resource. In a context switch, the state of the first process must be saved 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. The state from the running process is stored into the process control block. After this completes, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can begin executing.

Context switches are computationally intensive since register and memory state must be saved and restored. Much of the design of operating systems is to optimize the use of context switches since they can occur 10 to 1000 times per second in modern operating systems. Context switches occur when processes change to the wait queue for I/O, due to interrupt handling, and for user and kernel mode switching. In some systems a context switch occurs after the current running process has run for some pre-defined amount of time (called a quantum). If the process hasn't finished and hasn't started some I/O then after the quantum is up, the OS does a context switch to another process. A quantum is typically a few milliseconds to a few hundred milliseconds (Linux uses around 200ms). The context switch can take on the order of 100 microseconds

4.2.2 Creating a Process: *fork* System Call

A process can create other processes to do work. All processes are created by some other process; each process is considered to have a single parent, and it may have several children if it creates multiple processes.

The first process on a Unix/Linux system is the Init process. This process starts up during the boot process and initiates various system daemons and eventually the login process. When you subsequently log into a system, you are connected to a shell process which will in turn spawn additional processes for each command you run.

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". The *pstree* command in Linux can be used to see the process hierarchy. 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 calls the *exec* function to allow it to start a new application or function. 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 an argument to the function. As a new process is not created, the process ID (PID) does not change, 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.

4.2.3 Process Termination

On process termination, OS reclaims all resources assigned to the process. In Unix, a process can terminate itself using the *exit* system call. Alternatively, a process can terminate another process (if it has the privilege to do so) using the *kill* system call. Note that if a process is killed, its child processes may or may not be killed. If they are not, then they will be assigned a new parent process, either a "grand parent" or the Init process.

4.2.4 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 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. Modern web browsers such as Google Chrome also use cooperating processes to allow different windows or tabs to be isolated from one another for stability and security reasons. 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. To effectively solve the producer-consumer problem, the processes can use either shared memory or interprocess communication to coordinate.

4.2.5 Process Communication

Message passing is very much like e-mail. One process will use a *send* system call to send a message to another process. The other process uses a *receive* system call to recieve the message. Each process needs to be able to name the other processes that it would like to communicate with. The OS is responsible for keeping track of all of the messages (copying, notifying, etc.). Distributed systems generally will use message passing to communicate.

Shared memory is another process communication method that establishes a mapping between process address space and a named memory object. This named memory object can then be shared by multiple processes. The system call *mmap* is used to create the memory object. A typically example usage would be a process creating a shared memory object with *mmap* then using *fork* to create other processes that will share the data structure. The processes involved still need to do some sort of synchronization to regulate reading and writing to the shared memory.