Operating Systems (CS3013) Notes

Christopher Myers

June 13, 2020

Contents

1 Virtualization

Virtualization is a technique by which the operating system takes existing hardware resources (e.g. the processor, the disk, etc.) and *virtualizes* it, i.e. makes multiple virtual copies of it or otherwise highly abstracts access to it. For instance, the OS can virtualize access to the CPU by creating processes for each running program and allotting them processing time (timesharing).

1.1 Processes

Processes are tasks that the computer is running, with various pieces of data tied to them such as state or PID. Note that processes as given here are processes on unix-like systems.

1.1.1 Creating processes

Processes are often launched from a shell, a text-based interactive program that the user can use to interact with the computer. Command facilities include the ability to navigate through the filesystem, launch programs, view files, and a variety of helpful features to make interaction easier (e.g. autocomplete, autojump, syntax highlighting, etc.)

Shells are able to start new processes using fork() and exec() calls. The actual process of executing a new process involves transforming the current process into the process whose execution is desired, done by reinitializing the stack, the heap, and by replacing the text section of the code in memory. This can actually be inefficient in many cases, so sometimes alternative functions like clone() are used instead.

Since exec() will transform the process that it runs on, it is usually desirable to call fork() to create a duplicate process *before* calling exec(). The fork() call will return differently depending on whether it's in the child process or the parent process - 0 is the parent, anything positive is the child.

After a child has been spawned off to do its exec() call, the parent must wait for the child to die before continuing execution or else the input/output of the two can overrun and resources that don't need to be occupied will remain occupied.

1.1.2 State

Processes have state, meaning what their status is or what they're doing. These states can happen over different timespans - for example, a very small timespan might be for context switching, while a larger timespan might be a block due to an I/O request. Some of these states are listed below:

- Running - one of the process threads is currently executing on the CPU.
- Ready - the process is ready to run, but has not yet been scheduled to receive time on the CPU.
- Blocked - the process is waiting for something (like a disk read) before it continues.

Moving between running and ready is done by the OS, specifically by the process scheduler in a process predictably called *scheduling*. The process of determining what processes should run next is actually very complex and depends on various factors.

1.2 Limited Direct Execution

Limited direct execution is the means by which the OS is able to implement preemptive timesharing. It does this with hardware support in the form of two special instructions.

The first kind is a trap, which raises the privilege level and jumps into the OS. The second is a return-from-trap, which drops privilege level and returns back to the process. This return jump is accomplished in part using the program counter (sometimes called the instruction pointer), a register found in modern processors.

In order to execute system calls, the OS uses a trap handler that allows programs to interrupt and call system calls, supplying the given call with all the arguments it needs to complete and return (including the system call number, which tells the OS which system call to actually execute).

2 Threads & Multithreading

Threads can be viewed as separate processes that share address space with their parent process. In that sense creating a new thread is roughly analogous to fork()ing, except there is no duplication of the process that calls fork(). Threads share heap space and the text section, but they do *not* share stacks, so care must be taken not to use stack variables for communication between threads.

2.1 Atomicity

Atomicity is the illusion that a set of separate actions occur all at once as a solid unit, i.e. atomically, and is the solution to the problem of race conditions in threads. It requires allowing certain threads to continue at certain times or preventing some from running until a set time, in effect giving the programmer a very small amount of control over process scheduling. By using various tools such as semaphores and mutexes, programmers can better protect critical sections (areas of code where shared resources are modified).

2.2 Race conditions

Race conditions occur when two or more threads try to access and modify the same code at the same time. This causes issues because most operations, even simple adding to something in memory, are not atomic. Consequently the results of such non-atomic operations can be non-deterministic when two threads are in critical sections at the same time. Obviously, this is undesirable and race conditions are usually things to be eliminated.

Critical sections, then, are areas where two threads can't be in the same area at the same time or a race condition will occur – this property is called *mutual exclusion*. No matter how this is controlled for, all threads should be able to eventually pass through this critical section.

2.3 Locks & other synchronization methods

Locks are ways of keeping competing threads from operating within the same critical section.

2.3.1 Spinlocks

Spinlocks work by setting a lock when one thread enters and releasing it when it exits. Other threads that try to enter will get caught in a loop while they wait to enter - hence the "spin" part of spinlock. This is very inefficient and not optimal as the CPU could be using that time spinning to do something more useful.

3 Memory Management

Memory is managed in a per-process virtual address space that starts at zero for all processes. The goal of the system is to allow for easy memory management within processes, at cost of a bit of overhead for the OS and the hardware. This presents the challenge, then, of mapping a process's virtual address to the actual physical address in RAM, alongside the challenge of dividing up memory among processes and handling fragmentation.