The Processes Abstraction

Andrew Quinn

Learning Objectives:

1. What is a process?
2. How does an operating system use limited direct execution so that processes are safe and efficient?

Announcements:

1. HW2 due tonight!

Many of you will have heard of the idea of a process before—it's a word that I've certainly used frequently during lectures in this class. You're probably used to thinking of a process informally as a *running program*—an instantiation of a program that actually performs some computation. This intuition is spot on.

However, things get a bit more complex from an operating system perspective. Each process has the illusion that it has access to its own machine. Among other things, this includes:

1. Each process thinks that it has its own CPU, with its own registers and CPU context. That is, on an x86 machine, each process that runs will have its own unique value for `eax`.
2. Each process thinks that it has its own memory. That is, each process that runs can have unique values assigned to the same memory address (e.g., `*(0x1000)` can be `10` in process A but `12` in process B).
3. Each process thinks that it has its own open files. That is, even if two processes open the same file, they each have their own *file cursors* indicating their current file offset.

An operating system makes this possible through *virtualization*. It has to keep track of all of the state of all of the processes to make the system run smoothly and provide this illusion.

THE OS PERSPECTIVE

Let's get into the operating system's perspective of processes in a system, starting by describing the *states* that a process can be in:

1. **Running**: The process is currently running on a CPU.
2. **Ready**: The process is not currently running, but it is *schedulable* by the OS (it would have things to do).
3. **Blocked**: The process is waiting on some other event, such as I/O or a synchronization primitive.
4. **Finished**: Called a *zombie process*, the process has finished executing, but no other process has observed its exit code (the term used in the literature is that no other process has *reaped* it).

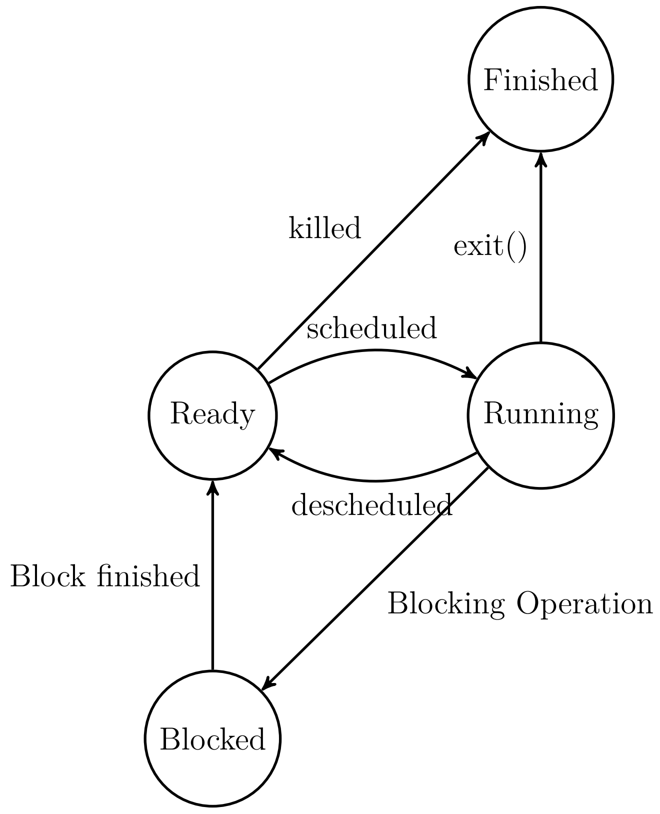

Figure 1: The possible process states.

Figure 1 describes these states pictorially. We see that the operating system can move a process between running and ready, whereas a process moves itself to blocked by executing a *blocking* operation. Your book has an example of how processes move through these states, which we will discuss (Figure 4.4 in OSTEP).

LIMITED DIRECT EXECUTION

We saw that the operating system provides the illusion of multiple hardware resources for each of the processes in the system. To do this, the operating system needs to be a resource manager of the system. One idea that has been proposed would be for the operating system to *emulate* the actions of processes, where every instruction that a process executes would pass through the operating system.

This would work, but emulation is very slow! So, there seems to be a tradeoff: how can operating systems manage resources while also retaining efficiency?

The answer is *limited direct execution.* With limited direct execution, the operating system allows a process to execute directly on the hardware, without *any* intervention by the system. There are two problems: (1) How can the OS ensure that the process does access resources in an unsafe manner? and (2) How can the OS gain control of the hardware to perform its actions? We'll walk through both of these ideas in more detail:

SAFETY THROUGH RESTRICTED OPERATIONS

Operating systems ensure safety by restricting the operations that a process can perform on its own. Modern hardware introduces modes: user mode and kernel model. The hardware only allows certain operations to be executed while in kernel mode. Our systems then execute processes in user mode, preventing the processes from doing *bad stuff*.

But, what if a process has a valid reason for using a privileged instruction? For example, what if a process wants to read a file! To make this possible, operating systems provide *system calls*, usually a few hundred of them. You may think that system calls look like normal procedure calls, but that is just because you have probably always used the wrappers for system calls provided by libc.

Instead, system calls are implemented using special *trap* instructions. These instructions simultaneously jump into a specific function in the kernel and change the mode of the hardware into kernel mode. When finished with the system call, the operating system can execute a *return-from-trap* instruction to reverse the process. The exact semantics of these instructions (e.g., where the arguments to the systemcall go, where to place the context of the process when the trap occurs, etc.) vary across architectures, but the high-level behavior is the same.

One key question remains, though: how does the system know where in the kernel to jump? Surly the user process should not be able to directly specify an address. Instead, on boot, the operating system populates an *interrupt table*, which specifies which functions to execute on each of the various interrupts that can happen. These functions are called interrupt handlers.

One note: your book uses the word “trap” to refer to interrupts triggered by a program. Other sources call these “software interrupts” or “internal interrupts”. The book uses the word “interrupt” to refer to interrupts triggered by hardware, some sources call these “hardware interrupts” or “external interrupts”. I will try to be consistent with your book.

There is not a separate interrupt handler for every possible systemcall, as there are not enough entries in the interrupt table for such a design. So, processes instead pass a systemcall number when executing a trap to specify which system call. We’ll walk through Figure 6.2 in class as an example of executing such a systemcall; you will be asked to do this in your second project/assignment.

CONTROL THROUGH CONTEXT SWITCHES

Let’s say that the operating system wants to switch from one process to another; how? Limited direct execution makes this difficult—the operating system is, by definition, *not* executing. We could try assume that all processes will eventually execute system calls. This is called cooperative scheduling; it has been used in many systems, but can easily be exploited by bad actors.

So, instead an operating system would like to have *preemption*—the ability to force a process off of the CPU. This is also called non-cooperative scheduling. The approach taken by most operating systems (including PintOS) is to have an external interrupt called a timer interrupt. The process of using the timer interrupt is almost identical to the process of executing a trap instruction. The key difference is that the operating system may switch to a different process, called a *context switch*, while processing the timer interrupt.

This approach allows the system to perform *time multiplexing*, in which it schedules different processes on its hardware resources. We'll talk more about context switching and scheduling (basically this topic) in the next class.