3

The Abstraction: Processes

In this note, we discuss one of the most fundamental abstractions that the OS provides to users: the **process**. The definition of a process, informally, is quite simple: it is a **running program**. The program itself is a lifeless thing: it just sits there on the disk, a bunch of instructions (and maybe some static data), waiting to spring into action. It is the operating system that takes this bunch of bytes and gets it running, thus transforming the program into something useful.

The OS provides this abstraction by **virtualizing** the CPU. By running one process, then stopping it and running another, and so forth, the OS can promote the illusion that many virtual CPUs exist when in fact there is only one physical CPU (or a few).

We'll now discuss some of the important concepts underlying the notion of a process. We'll start with process context, continue by discussing the different states a process can be in, and then discuss key issues in how the OS gains control of the CPU. We'll then briefly discuss the OS API for creating and managing processes.

3.1 Process Context

To understand what constitutes a process, we have to understand its **context** (sometimes called **state**); that is, what parts of

the system does the OS need to save in order to be able to later re-start the process seamlessly? To understand this better, think of what a program can read or update when it is running; at any given time, what in the machine is an important component of the execution of this program?

One obvious component of process context is *memory.* All instructions lie in memory; the data that the running program reads and updates sits in memory as well. Thus memory and any related pieces of information the OS should track about memory are a part of the context of the process.

Also part of the process context are *registers*; many instructions explicitly read or update registers and thus clearly they are important to the execution of the process.

Note that there are some particularly special registers that also comprise context beyond general-purpose registers. For example, the **program counter (PC)** (sometimes called the **instruction pointer**) tells us which instruction of the program is currently being executed; similarly a **stack pointer** and associated **base pointer** are often used to manage the stack for local variables, function parameters, return addresses, and the like.

Finally, programs often access persistent storage devices too. Such *I/O information* might include a list of the files the process currently has open.

Of course, there are many other pieces of information that the OS tracks about a process. For example, scheduling priority information, relationship to other processes (e.g., which process created this one), information about which event a process is blocked upon (if it is blocked), and many other similar things may also comprise process context; the exact details depend on the system in question.

3.2 Process States

Now that we have some idea of what a process is (though we will continue to refine this notion), let us talk about the different **states** a process can be in at a given time.

In a simplified view, a process can be in one of three states:

- **Running:** In the running state, a process is running on a processor. This means it is executing instructions.
- **Ready:** In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.
- **Blocked:** In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place. A common example: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

If we were to map these states to a diagram showing the possible transitions, we would arrive at something like the picture in Figure 3.1.

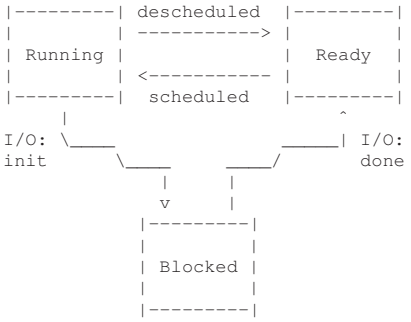

Figure 3.1: Process: State Transitions

As you can see in the diagram, a process can be moved between the ready and running states at the discretion of the OS. Being moved from ready to running means the process has been

DATA STRUCTURE: THE READY QUEUE
Operating systems are replete with various important **data structures** that we will discuss in these notes. The **ready queue** is the first such structure. It is one of the simpler ones, but certainly any OS that has the ability to run multiple programs at once will have something akin to this structure.

scheduled; being moved from running to ready means the process has been **descheduled**. Once a process has become blocked (e.g., by initiating an I/O operation), the OS will keep it as such until some event occurs (e.g., I/O completion); at that point, the process moves to the ready state again (and potentially immediately to running again, if the OS so decides).

Note that the OS must track the states of these processes. To do so, the OS likely will keep some kind of **ready queue** for all processes that are ready, as well as some additional information to track which process is currently running. The OS must also track, in some way, blocked processes, and when an I/O event completes, make sure to wake the correct process and move it on to the ready queue.

We should also note that there are some other states a process can be in. Sometimes a system will have an **initial** state that the process is in when it is being created. Also, a process could be placed in a **final** state where it has exited but has not yet been cleaned up (in UNIX-based systems, this is called the **zombie** state[1]). This final state can be useful as it allows other processes (usually the **parent** that created the process) to examine the return code of the process and see if it executed successfully. When finished, the parent will then make one final call to indicate to the OS that it can completely forget about the now-extinct process (the `wait()` system call in UNIX does this).

[1] Yes, the zombie state. Don't worry – these zombies are not too scary.

3.3 Gaining Control of the CPU

At this point, you should have some understanding of the lifetime of a process. It gets created, starts running on the CPU, perhaps issues some I/O requests (moving to the blocked state and back as I/Os complete), then runs again, and so forth, until it completes. While a process is running, though, sometimes the OS would like to be able to stop it and run some other process (some other deserving process from the ready queue that is).

This sounds simple but it is actually a little bit tricky: specifically, if a process is running on the CPU, this by definition means the OS is *not* running. If the OS is not running, how can it do anything at all? It sounds almost philosophical, but it is a real problem: there is clearly no way for the OS to take an action if it is not running. Thus we arrive at the crux of the problem.

THE CRUX: HOW TO REGAIN CONTROL OF THE CPU
How can the operating system **regain control** of the CPU so that it can switch between processes?

A Cooperative Approach: Wait For System Calls

One approach that some systems have taken in the past (for example, early versions of the Macintosh operating system) is known as the **cooperative** approach. In this style, the OS *trusts* the processes of the system to behave reasonably. Processes that run for too long are thus assumed to periodically give up the CPU so that the OS can thus decide to run some other task.

Thus, you might ask, how does a friendly process give up the CPU in this utopian world? Most processes, as it turns out, transfer control of the CPU to the OS quite frequently by making **system calls**, for example, to open a file and subsequently read it, or to send a message to another machine, or to create a new process. Systems like this often include an explicit **yield** system

DESIGN TIP: DEALING WITH MISBEHAVIOR
Operating systems often have to deal with misbehaving processes, those that either through design (maliciousness) or accident (bugs) attempt to do something that they shouldn't. In modern systems, the way the OS tries to handle such malfeasance is to simply terminate the offender. One strike and you're out! Perhaps a little brutal, but what else should the OS do when you try to access memory illegally or execute an illegal instruction?

call, which does nothing except to transfer control to the OS so it can run other processes.

Applications also transfer control to the OS when they do something illegal. For example, if an application divides by zero, or tries to access memory that it shouldn't be able to access, it will generate a **trap** to the OS. The OS will then have control of the CPU again (and likely terminate the offending process).

Thus, in a cooperative scheduling system, the OS regains control of the OS by waiting for a system call or an illegal operation of some kind to take place. You might also be thinking: isn't this passive approach less than ideal? What happens, for example, if a process (whether malicious, or just full of bugs) ends up in an infinite loop, and never makes a system call? What can the OS do then?

A Non-Cooperative Approach: The OS Takes Control

Without some additional help from the hardware, it turns out the OS can't do much at all when a process refuses to make system calls (or mistakes) and thus return control to the OS. In fact, in the cooperative approach, your only recourse when a process gets stuck in an infinite loop is to resort to the age-old solution to all problems in computer systems: **reboot the machine**. Thus, we again arrive at a subproblem of our general quest to gain control of the CPU.

THE CRUX: GAINING CONTROL WITHOUT COOPERATION
How can the OS gain control of the CPU even if processes are not being cooperative?

The answer turns out to be simple and was discovered by a number of people building computer systems many years ago: a **timer interrupt**. A timer device can be programmed to raise an interrupt every so many milliseconds; when the interrupt is raised, the currently running process is halted, and a preconfigured **interrupt handler** in the OS runs. At this point, the OS has regained control of the CPU, and thus can do what it pleases: stop the current process from running, and start a new one running.

There are three important aspects of managing the timer interrupt that the OS and hardware must perform together. First, the hardware must know which OS code to run when the interrupt takes place. The OS establishes such a **trap table** at boot time, basically giving the hardware a pointer to some memory where the code that should run lives. Of course, this is a privileged operation; only the OS should be able to perform the operations needed in order to set up this table.

Second, the OS must start the timer, also a privileged operation. Once begun, the OS can thus feel safe in that control will eventually be returned to it. Note that the timer can also be turned off, something we will discuss later when we understand concurrency in more detail.

HARDWARE SUPPORT: THE TIMER INTERRUPT
The addition of a **timer interrupt** gives the OS the ability to regain control of the CPU even if processes act in a non-cooperative fashion. Thus, this hardware is key in helping the OS maintain control of the system.

Finally, the OS must be very careful when first running in such an interrupt handler. In particular, the OS must be able to

correctly save the context of the running process, and, if it decides to switch to another process, restore the context of the new process. The saving of the context of one process and restoring of another is called a **context switch**, which we now discuss in more detail.

3.4 The Tricky Part: Saving and Restoring Context

When a timer interrupt goes off, the hardware takes control and does a number of things before transferring control to the OS. In particular, the OS needs to be able to successfully save the context of the currently-running process and restore the context of the about-to-be-run process. This code has to be written carefully in assembly, as it deals with the intricacies of the machine upon which it runs.

One thing the hardware absolutely must do is *save the program counter* for the OS somewhere. Once the hardware jumps to the interrupt handler, the PC (by definition) changes; once changed, the old PC is lost and the OS will not be able to save the context of the current process correctly. Thus, most hardware has some machinery to put the PC at the time of the interrupt someplace the OS can access it (e.g., on the kernel stack, or in a special register).

The code that runs also must be careful to avoid overwriting any registers that the process was using before it saves them. Unfortunately, just to execute instructions, registers must be used, and thus the code to save them must again be quite careful. Some hardware systems reserve a few registers for the OS to use in this piece of code; others have instructions that can directly save registers to well-known memory locations.

Thus, when switching from one process to another, the OS must be careful so as not to overwrite any registers before it saves them. But where should it save the values? The data structure that the OS saves such values to is usually called the **process control block (PCB)**, which contains such information. This data structure thus contains room for all the registers of the

process including special ones like the PC and stack and base pointers. Thus, to switch between processes, the OS must know the locations of the PCB structures for each; once known, the OS can save the state of the current process to its PCB, restore the state of the about-to-be-run process, and then run that process.

To start running a process, the OS must execute one more special instruction that the hardware provides: a **return-from-trap instruction**. This instruction not only returns execution to the proper PC but also reduces the privilege level of the process to user mode as needed.

3.5 The Dispatcher

One final note: all of these low-level actions, like taking switching between processes, and mucking around with the run queue, and so forth, are found in a part of the OS often referred to as the **dispatcher** (or **low-level scheduler**). The dispatcher is thus responsible for managing the state of the processes (as described above) and for performing context switches.

3.6 Summary

We have introduced the most basic abstraction of the OS: the process. We have further discussed what process context is, and how the OS can virtualize the CPU by switching from one process to another via a context switch. We have also discussed a key issue: how to forcefully regain control of the CPU when a process is not cooperating. This key issue allows the OS to control the hardware as desired, instead of trusting processes to work correctly and non-maliciously.