ECE 350: Real-Time Operating Systems Spring 2023

Lecture 2 — Review of Processes and Threads

Jeff Zarnett *2023-02-27*

Past is Prologue

In prerequisite courses (should you have taken them as per the standard program!), we covered three important things that are relevant to the operating system: processes, threads, and hardware. Here, we're going to take some time to review these things and make sure we're on solid ground before moving on.

Remember, It's a Trap

Operating systems run, as previously discussed, on interrupts. In addition to the interrupts that will be generated by hardware and devices (e.g., a keyboard signalling that the F1 key has been pressed), there are also interrupts generated in software. These are often referred to as a *trap* (or, sometimes, an exception). The trap is usually generated either by an error like an invalid instruction or from a user program request.

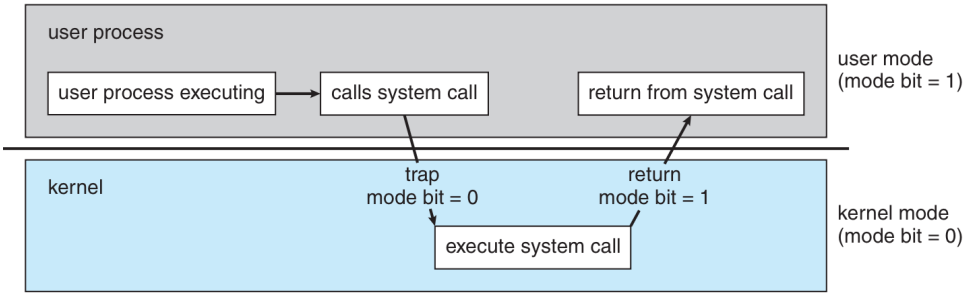
If it is simply an error the operating system will decide how to deal with it, and in desktop/laptop OSes, the usual strategy is sending the exception to the program that caused it, and this is usually fatal to the offending program. Your programming experience will tell you that you can sometimes deal with an exception (perhaps through the language equivalent of the Java/C# `try-catch-finally` syntax), but often an exception is unhandled and terminates the program.

The more interesting case is the intentional use of the trap: this is how a user program gets the operating system's attention. When a user program is running, the operating system is not; we might even say it is "sleeping". If the program running needs the operating system to do something, it needs to wake up the OS: interrupt its sleep. When the trap occurs, the interrupt handler (part of the OS) is going to run to deal with the request.

Already we saw the concept of user mode vs. supervisor mode instructions: some instructions are not available in user mode. Supervisor mode, also called kernel mode, allows all instructions and operations. Even something seemingly simple like reading from disk or writing to console output requires privileged instructions. These are common operations, but they involve the operating system every time.

Modern processors keep track of what mode they are in with the mode bit. This was not the case for some older processors and some current processors have more than two modes, but we will restrict ourselves to dual-mode operation with a mode bit. Thus we can see at a glance which mode the system is in. At boot up, the computer starts up in kernel mode as the operating system is started and loaded. User programs are always started in user mode. When a trap or interrupt occurs, and the operating system takes over, the mode bit is set to kernel mode; when it is finished the system goes back to user mode before the user program resumes [SGG13].

Suppose a text editor wants to output data to a printer. Management of I/O devices like printers is the job of the OS, so to send the data, the text editor must ask the OS to step in, as in the diagram below:



Transition from user to supervisor (kernel) mode [SGG13].

So to print out the data, the program will prepare the data for printing. Then it calls the system call. You may think of this as being just like a normal function call, except it involves the operating system. This triggers the operating system (with a trap). The operating system responds and executes the system call and dispatches that data to the printer. When this job is done, operation goes back to user mode and the program returns from the system call.

Motivation for Dual Mode Operation. Why do we have user and supervisor modes, anyway? As Uncle Ben told Spiderman, "with great power comes great responsibility". Many of the reasons are the same as why we have user accounts and administrator accounts: we want to protect the system and its integrity against errant and malicious users.

An example: multiple programs might be trying to use the same I/O device at once. If Program 1 tries to read from disk, it will take time for that request to be serviced. During that time, if Program 2 wants to read from the same disk, the operating system will force Program 2 to wait its turn. Without the OS to enforce this, it would be up to the author(s) of Program 2 to check if the disk is currently in use and to wait patiently for it to become available. That may work if everybody plays nicely, but without someone to enforce the rules, sooner or later there will be a program that does something nasty, like cancel another program's read request and perform its read first.

This doesn't come for free, of course: there is a definite performance trade-off. Switching from user mode to kernel mode requires some instructions and some time. It would be faster if everything ran in kernel mode because we would spend no time switching. Despite this, the performance hit for the mode switch is judged worthwhile for the security and integrity benefits it provides.

The Process and the Thread

A process is a program in execution. It is composed of three things:

1. The instructions and data of the program (the compiled executable).
2. The current state of the program.
3. Any resources that are needed to execute the program.

Having two instances of the same program running counts as two separate processes. Thus, you may have two windows open for Microsoft Word, and even though they are the same program, they are separate processes. Similarly, two users who both use Firefox at the same time on a terminal server are interacting with two different processes.

The Process Control Block

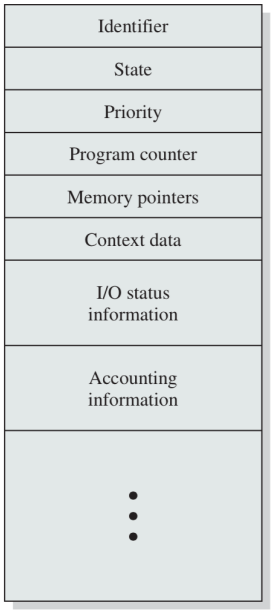
We will take a slight detour to the behind-the-scenes view of how the operating system manages a process, so that we can have a mental model of what may happen when a program is executing. The operating system's data

structure for managing processes is the *Process Control Block* (PCB). This is a data structure containing what the OS needs to know about the program. It is created and updated by the OS for each running process and can be thrown away when the program has finished executing and cleaned everything up. The blocks are held in memory and maintained in some container (e.g., a list) by the kernel.

The process control block will (usually) have [Sta18]:

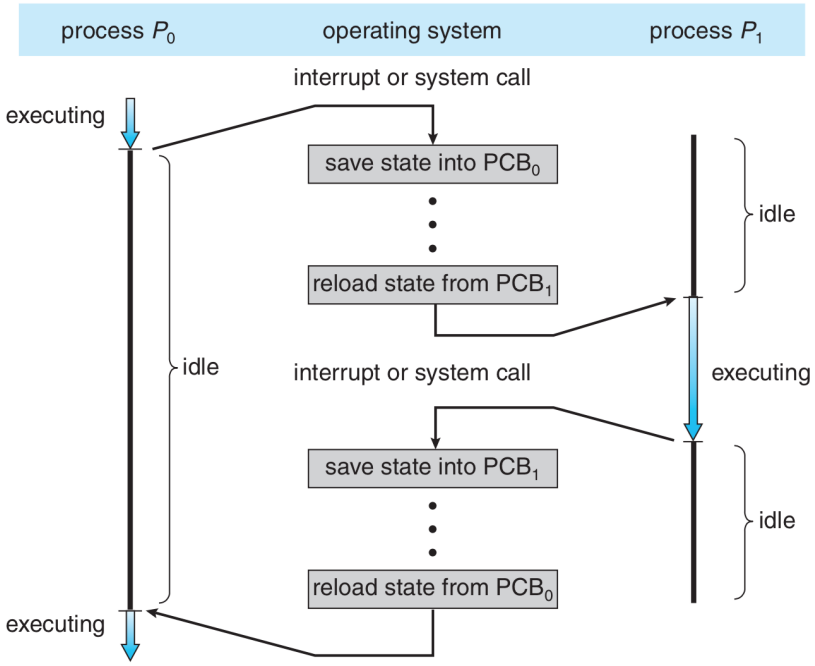
- **Identifier.** A unique ID associated with the process; usually a simple integer that increments when a new process is created and reset when the system is rebooted.
- **State.** The current state of the process.
- **Priority.** How important this process is (compared to the others).
- **Program Counter.** A place to store the address of the next instruction to be executed (*when needed).
- **Register Data.** A place to store the current values of the registers (*when needed); also called context data.
- **Memory Pointers.** Pointers to the code as well as data associated with this process, and any memory that the OS has allocated by request.
- **I/O Status Information.** Any outstanding requests, files, or I/O devices currently assigned to this process.
- **Accounting Information.** Some data about this process's use of resources. This is optional (but common).

To represent this visually:



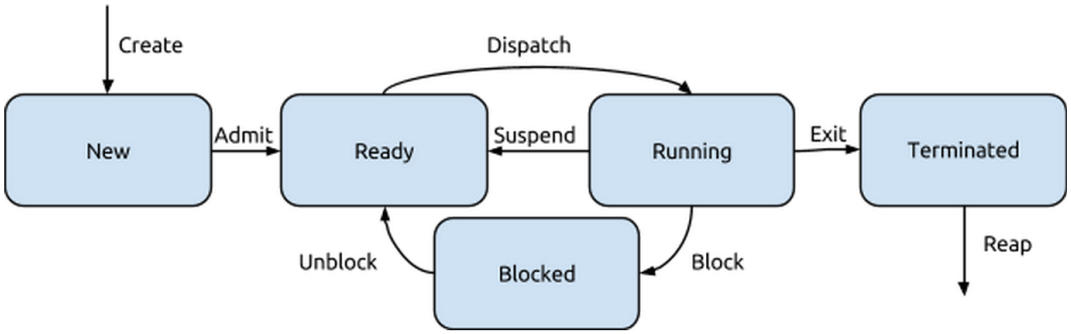
A simplified Process Control Block [Sta18].

Almost all of the above will be kept up to date constantly as the process executes. Two of the items, notably the program counter and the register data are asterisked with the words "when needed". When the program is running, these values do not need to be updated. However, when a system call (trap) or process switch occurs, and the execution of that process is suspended, the OS will save the state of the process into the PCB. This includes the Program Counter variable (so the program can resume from exactly where it left off) and the Register variables (so the state of the CPU goes back to how it was). The diagram below shows the sequence as the OS switches between the execution of process P_0 and process P_1.



A process switch from P_0 to P_1 and back again [SGG13].

Process States. The diagram below shows the five-state model:



State diagram for the five-state model.

There are now eight transitions, most of which are similar to what we have seen before:

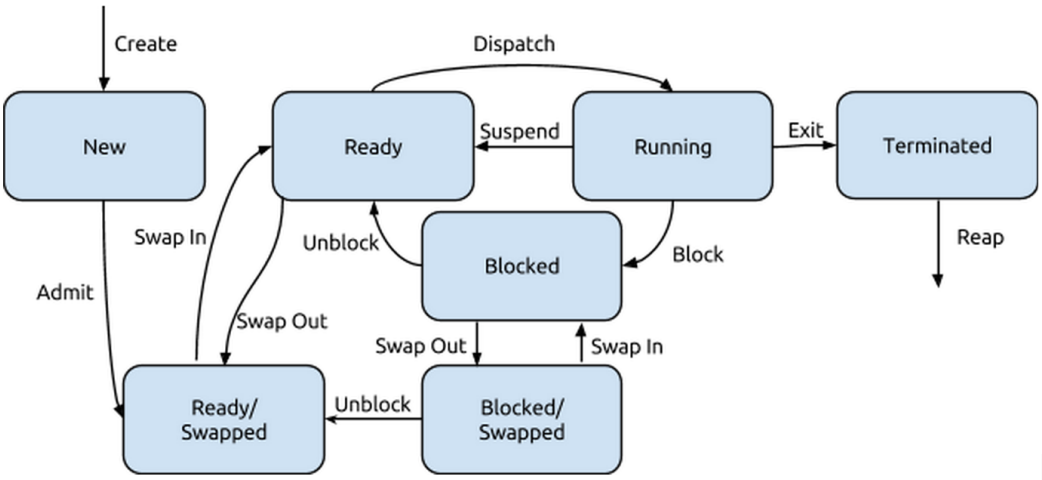
- **Create:** The process is created and enters the New state.
- **Admit:** A process in the New state is added to the list of processes ready to start, in the Ready state.
- **Dispatch:** A process that is not currently running begins executing and moves to the Running state.
- **Suspend:** A running program pauses execution, but can still run if allowed, and moves to the Ready state.
- **Exit:** A running program finishes and moves to the Terminated state; its return value is available.
- **Block:** A running program requests a resource, does not get it right away, and cannot proceed.
- **Unblock:** A program, currently blocked, receives the resource it was waiting for; it moves to the Ready state.

- **Reap:** A terminated program's return value is collected by a `wait` and its resources can be released.

There are two additional "Exit" transitions that may happen but are not shown. In theory, a process that is in the Ready or Blocked state might transition directly to the Terminated state. This can happen if a process is killed, by the user or by its parent (recall that parent processes can generally kill their children at any time, something the law thankfully does not permit). It may also happen that the system has a policy of killing all the children of a parent process when the parent process dies.

Remember that this model works for a thread, but the process has two additional ones:

Ready/Swapped (ready to run, and currently not in memory) and Blocked/Swapped (not ready to run, and currently not in memory). That gives us, finally, the seven-state model, a minor variation of the five-state model:



State diagram for the seven-state model.

As in the five-state model, there are additional "Exit" transitions that may happen but are not shown. If a process is killed, for example, regardless of whether it is in memory or on disk, it will move to the Terminated state.

At this point I assume you remember how to use `fork()` and related functions like `wait()` and there's no need to recap it here. If you are uncertain about it, please check the ECE 252 notes!

And the Thread

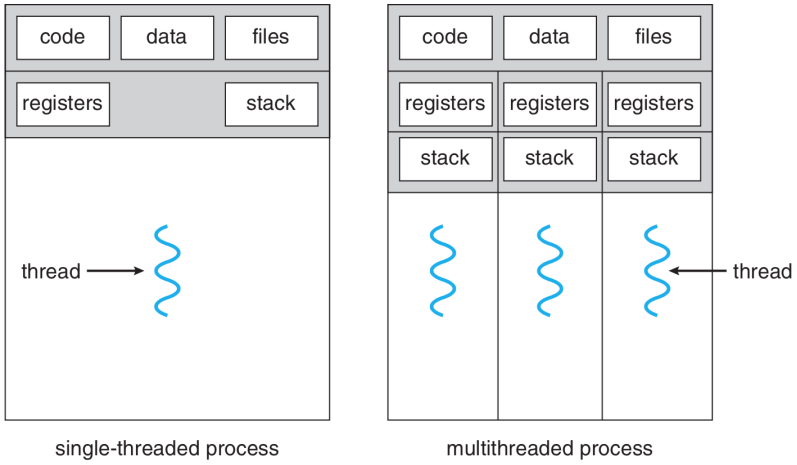
The term "thread" is a short form of *Thread of Execution*. A thread of execution is a sequence of executable commands that can be scheduled to run on the CPU. Threads also have some state (where in the sequence of executable commands the program is) and some local variables. Most programs you have written until now probably had only one thread; that is, your program's code is executed one statement at a time, sequentially in some order.

A multithreaded program is one that uses more than one thread, at least some of the time. When a program is started, it begins with an initial thread (where the `main` function is) and that main thread can create some additional threads if needed. Note that threads can be created and destroyed within a program dynamically: a thread can be created to handle a specific background task, like writing changes to the database, and will terminate when it is done. Or a created thread might be persistent.

In a process that has multiple threads, each thread has its own [Sta18]:

1. Thread execution state (like process state: running, ready, blocked...).
2. Saved thread context when not running.
3. Execution stack.
4. Local variables.
5. Access to the memory and resources of the process (shared with all threads in that process).

Or, to represent this visually:



A single threaded and a multithreaded process compared side-by-side [SGG13].

As you know, the primary motivation for threads over processes is performance. They are much faster to create and clean up than processes, and there's no overhead of establishing shared-memory communication. But of course, there are risks, like any one thread crashing the whole program.

Like with processes, I'll assume you remember how the various pthread functions work from ECE 252 – if not, please go back and look at that – it will save you a lot of headache...

References

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.

[Sta18] William Stallings. *Operating Systems Internals and Design Principles (9th Edition)*. Pearson, 2018.