ECE254: Operating Systems and Systems Programming Spring 2015

Lecture 4 — Processes

Jeff Zarnett

Processes

Early computers, as well as many modern embedded systems, did exactly one thing, or at least, exactly one thing at a time. At that time, the program had access to all the resources available in the system. Now, we expect that the OS supports multiple programs running concurrently. For that to work reliably, we need a way to manage the complexity and this has resulted in the notion of a *process*.

A process is a program in execution. It is more than just the compiled instructions, though that is an important part. A process has 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.

The process is a very important concept in operating systems. Practically everything the OS does involves one or more processes and all the other concepts we will examine in the future will depend on this subject. When we discuss scheduling, for example, the problem is scheduling of processes. How important are they?

> *The concept of process is fundamental to the structure of modern computer operating systems. Its evolution in analyzing problems of synchronization, deadlock, and scheduling in operating systems has been a major intellectual contribution of computer science.*
>
> - What Can Be Automated?: The Computer Science and Engineering Research Study, MIT Press, 1980

Most requirements of an operating system revolve around processes. Some examples from [Sta14]:

- **Scheduling:** The OS must choose which processes run at what time, to maximize processor utilization and/or minimize response time.
- **Resource Allocation:** The OS must allocate resources to processes, keeping track of what resources are owned by which process.
- **Inter-Process Communication:** The OS must support inter-process communication: the transfer of data from one process to another (by whatever means).

Note that 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.

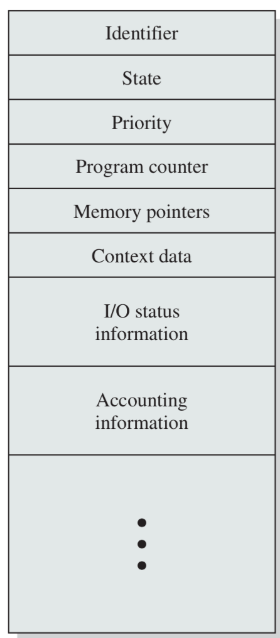
The Process Control Block

The operating system's data structure for managing processes is the *Process Control Block* (PCB). This is a data structure containing everything 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 [Sta14]:

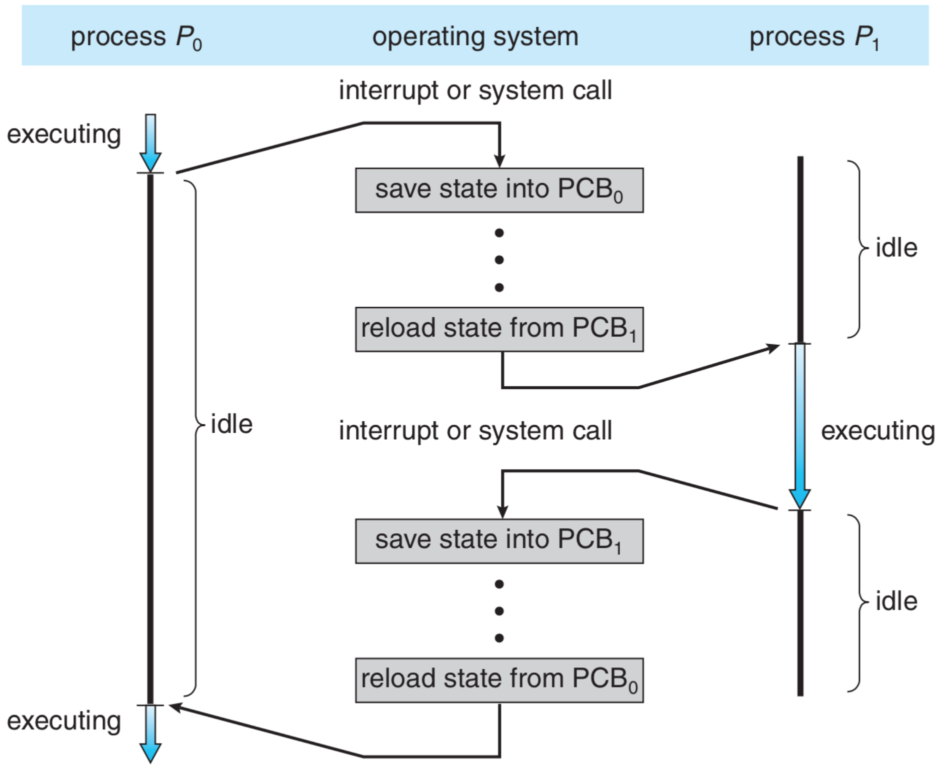
- **Identifier.** A unique ID associated with the process; usually a simple integer that increments when a new process is created and reset when the system is rebooted.
- **State.** The current state of the process.
- **Priority.** How important this process is (compared to the others).
- **Program Counter.** 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 [Sta14].

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 an system call (trap) or process switch occurs, and the execution of that process is suspended, we will need a way to restore the state of the program back to what it was. So what we will do is save the state of the process into the PCB. We save the state into the Program Counter variable (so we can remember what instruction was next) and the Register variables (so everything 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].

The Circle of Life

Unlike energy, processes may be created and destroyed. Upon creation, the OS will create a new PCB for the process and initialize the data in that block. This means setting the variables to their initial values: setting the initial program state, setting the instruction pointer to the first instruction in `Main`, and so on. The PCB will then be added to the set of PCBs the OS maintains. After the program is terminated and cleaned up, the OS may collect some data (like a summary of accounting information) and then it can remove the PCB from its list of active processes and carry on.

Process Creation

There are, generally speaking, three main events that may lead to the creation of a process [Tan08]:

1. System boot up.
2. User request to start a new process.
3. One process spawns another.

When the computer boots up, the OS is started and begins creating processes. An embedded system might have all the processes it will ever run created by this initialization process, but general-purpose operating systems will allow at least one of the other routes (if not all of them).

At boot time the OS starts up various processes, some of which will be in the foreground (visible to the user) and some in the background. A user-visible process might be the log in screen; a background process might be the server that shares media on the local network. The UNIX term for a background process is *Daemon*. You have already worked with one of these if you have ever used the `ssh` (Secure Shell) command to log into a Linux system; when you attempt to connect it is the `sshd` (Secure Shell Daemon) that responds to your connection attempt.

Users are well known for starting up processes whenever they feel like it, much to the chagrin of system designers everywhere. Every time you double-click an icon or enter a command line command (like `ssh` above) that will result in the creation of a process.

An already-executing process may spawn another. If you receive an e-mail with a link in it and click on that link[1], the e-mail program will start up the web browser (another process) to open the web page. Or a program may break its work up into different logical parts to be parcelled out to subprograms that run as their own process (to promote parallelism or fault tolerance). When an already-executing program spawns another process, we say the spawning process is the *parent* and the one spawned is the *child*. Later on, we will return to the subject of relations between processes in UNIX.

Process Destruction

Eventually, most processes die. This is sad, but it can happen in one of four ways [Tan08]:

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal Error (involuntary)
4. Killed by another process (involuntary)

Most of the time, the process finishes because they are finished or the user asks them to. If the command is to compile some piece of code, when the compiler process is finished, it terminates normally. When you are finished writing a document in a text editor, you may click the close button on the window and this will terminate the program normally.

Sometimes there is voluntary exit, but with an error. If the user attempts to run a program that requires write access to the temporary directory, and it checks for the permission on startup and does not find it, it may exit voluntarily with an error code. Similarly, the compiler will exit with an error if you ask it to compile a non-existent file [Tan08]. In either case, the program has chosen to terminate (not continue) because of the error and it is a voluntary termination.

The third reason for termination is a fatal error occurring in the program, like a stack overflow error or division by zero. The OS will detect this error and send it to the program. Very often, this results in the involuntary termination of the offending program. A process may tell the OS it wishes to handle some kinds of errors (like in Java/C# with the `try-catch-finally` syntax) in which case the OS will send the error to the program which can hopefully deal with it. If so, the process may continue, otherwise, the unhandled exception will result in the involuntary termination.

The last reason for termination is that one process might be killed by another (yes, processes can murder one another. Is no-one safe?!). Typically this is a user request: a program is stuck or consuming too much CPU and the user opens task manager in Windows or uses the `ps` command (in UNIX) to find the offender and then terminates it with the “End Process” button (in Windows) or the `kill` command (in UNIX). However, programs can, without user intervention, theoretically kill other processes, such as a parent process killing a child it believes to be stuck (or timed out).

Obviously, there are restrictions on killing process: a user or process must have the rights to execute the victim. Typically a user may only kill a process he or she has created, unless that user is a system administrator. While killing processes may be fun, it is something that should be reserved for when it is needed.

Sometimes when a process is killed, all the process it has spawned are killed as well (this is really cruel). Neither UNIX nor Windows works this way, however: a parent can outlive the death of its child and vice-versa.

[1] Security advice: don’t click on links you receive by e-mail.

Process Family Tree

In UNIX, but not in Windows, the relationship between the parent process and child process(es), if any, is maintained, forming a hierarchy. A process, unlike most plants and animals, reproduces asexually: a process has one parent, but may have zero or more children. A process and all its descendants form a *process group* and certain operations like sending a signal (e.g., the terminate signal `Ctrl-C`) can be sent to the whole group, letting each process decide what to do with it. [Tan08].

In UNIX the first process created is called `init` and it is the parent of all processes (eventually), much like the `Object` class in Java is the superclass of all classes in the system. Thus in UNIX we may represent all processes as a tree structure, where each node is a process, each node may have zero or more children, and moving up the hierarchy will eventually take us to `init`.

In Windows, a process that spawns another process gets a reference to its child, allowing it to exercise some measure of control over the child. However, this reference may be given to another process (so, the concept of adoption exists) meaning there is no real hierarchy. A process in UNIX cannot disinherit a child [Tan08].

When a process terminates, voluntarily or otherwise, it does so with a return code, just as a function often returns a value. If the command is issued on the command line (e.g., `cat /var/log/syslog` or from double clicking an icon, the return value is generally ignored (or at least, not presented to the user). In UNIX, when a parent process spawns a child, it can get the code that process returns. Usually, a return value of zero indicates success and other values indicate an error of some sort. Normally there is some sort of understanding between the parent and child processes about what a particular code means.

The workflow in UNIX is as follows. First, the parent spawns the child process with the `fork` system call. If it is interested in waiting for the child process to finish, it will use the system call `wait`, in which case the parent will be awaiting the completion of the child process. When the child process is finished, it returns a value with the `exit` system call. The parent process will then get this as the return value of the `wait` call and may proceed.

When a child process finishes execution, until such time as the parent comes by to collect the return value, the child continues in a state of "undeath" we call a *zombie*. This does not mean that the process then shuffles around the system attempting to eat the brains of other processes; it just means that the process is dead but not gone. The program has finished executing, there is still an entry, the PCB list and the process holds on to its allocated resources until such time as the return value is collected. Only after the return value is collected can it be cleaned up. Usually, a child process's result is eagerly awaited by its parent and the `wait` call collects the value right away, allowing the child to be cleaned up (or, more grimly, "reaped"). If there is some delay for some reason, the process is considered a zombie until that value is collected.

If a child process's parent should die before the child does, the process is called an *orphan*. In UNIX any orphan process is automatically adopted by the `init` process, making sure all processes have a good home. By default, `init` will just `wait` on all its child processes (and do nothing with the return values), ensuring that when they are finished, they do not become zombies. Sometimes a program is intentionally orphaned: it is spawned to run in the background (e.g., when starting up a service or daemon on the system). This would be cruel, except that processes, as far as anyone knows, do not have feelings.

References

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

[Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.

[Tan08] Andrew S. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Prentice Hall, 2008.