Computer Science 330
Operating Systems
Siena University
Fall 2025

Topic Notes: Processes

In your OS zyBook Chapter 4 readings and activities, you learned about processes, process control blocks, and the basics of how processes are created, how they move throughout the system, and how they are destroyed.

These notes expand on some of those ideas, using FreeBSD 14.3 on noreaster as a case study.

What is a *process*?

Definitions from various textbooks:

- an instance of a program being executed by an OS
- a program in execution
- an abstraction of a running program
- an asynchronous activity
- the "locus of control" of a program in execution
- that which is manifest by the existence of a process control block in the OS
- that entity which is assigned to processors
- the "dispatchable" unit
- the "animated spirit" of a procedure

A process is sequential.

Parts of a process:

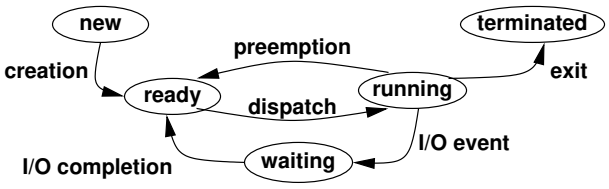
- program code (text section)
- program counter and other registers
- stack (local variables and function call information)
- data (global variables)
- heap (dynamically allocated variables)

A typical multiprogrammed system has many processes at any time. Try `ps -aux` or `ps -ef` to see the processes on your favorite Unix system. Only one of these processes can be on a CPU at a time.

If we look at the `ps` output on a Unix system, we will see a lot of processes owned by `root`, the "super-user". Many of these are essential parts of the system and are intended to continue running as long as the system is up. These processes are called *daemons* and are the motivation for the BSD logo.

So if a process is "in the system" but not executing on the CPU, where is it?

States of a process:



What information do we need to let a process transition among these states?

Think about it in terms of what a person needs to do to get back to what he or she was doing before being interrupted.

- you're sitting in the lab working hard on your OS project and someone interrupts you
- your attention shifts and you go off and do something else for a while
- then when you need to come back to the work, you need to remember *what* you were doing, and *where you were* in the process of doing it

We do this all the time, and many of us are really pretty good at it. A processor can't just pick up where it left off, unless we carefully remember everything it was doing when it was so rudely interrupted.

A *process control block* (PCB) is used to store the information needed to save and restore a process. This information includes:

- Process state (running, waiting, ready)
- Process identifier (PID)
- Program counter
- Other CPU registers
- CPU scheduling information

- Memory-management information
- Accounting information
- I/O status information

In many Unix systems, the PCB is divided between two structures:

- The *proc* structure, which holds the process information needed at all times, even when the process is not active (swapped out).
- The *user* structure, which holds other information about the process.

Historically, it was important to keep as much in the user structure and as little in the proc structure as possible, because of memory constraints. As the size of memory has grown, the amount of memory used by PCBs has become less significant, so this division has become less important.

In FreeBSD, for example, most of the information is now in the proc structure. We can see it on your favorite FreeBSD 14.3 system (noreaster) in `/usr/src/sys/sys/proc.h`. See `struct proc` defined starting at line 653.

The user structure is in `/usr/src/sys/sys/user.h` (line 243), and it is somewhat smaller. It contains per-thread information. Part of this is a `struct kinfo_proc`, which in turn includes a `struct pcb`, which is an architecture-dependent structure defined in `/usr/src/sys/amd64/include/pcb.h` (line 52). Here you find the actual registers that need to be saved when a process is removed from the CPU. We can also look at the other architectures in some other directories such as `arm` and `powerpc` (MIPS is no longer in the FreeBSD kernel, unfortunately).

The transition from one process running on the CPU to another is called a *context switch*. This is pure overhead, so it needs to be fast. Some systems do it faster than others. Hardware support helps - more on that later. Of course, a context switch can only take place when the CPU is running in kernel model.

Process Creation/Deletion

In Unix, every process (except for the first) is created from an existing process. See `ps` again for examples. The processes form a tree, with the root being the `init` process (PID=1).

When a new process is created, what information does it inherit from or share with its parent?

- Does it get any resources that were allocated to the parent?
- Does the parent wait for the child to complete, or do they execute concurrently?
- Is the child a duplicate of the parent, or is it something completely different?
- If it's a duplicate, how much context do they share?

- Can the parent terminate before the child?

Creation of a new process is a highly privileged operation. It requires the allocation of a new PCB, insertion of this PCB into system data structures, among other operations that we would not trust to regular "user" programs.

As we have discussed, such operations are provided through *system calls*. These system calls are functions that are part of the operating system and are permitted to perform some tasks that a normal process is not able to do on its own. If successful, the program temporarily gains a higher privilege level while executing the system call. If the kernel does not grant permission for the process to execute the call, an error return or even program termination could result.

In Unix, the `fork()` system call duplicates a process. The child is a copy of the parent - in execution at the same point, the statement after the return from `fork()`.

The return value indicates if you are the new process (the child) or the original process (the parent).

0 is child, > 0 means parent, -1 means failure (e.g., a process limit has been reached, permission is denied)

A C program that wishes to create a new process will include code similar to this pattern:

```
pid=fork();
if (pid) {
  parent stuff;
}
else {
  child stuff;
}
```

A more complete program that uses `fork()` along with three other system calls (`wait()`, `getpid()`, and `getppid()`) is in `forking.c` in the `forking` subdirectory of `https://github.com/SienaCSISOperatingSystems/procsthreads-examples`.

Some comments about this program:

First, run it to observe what happens. Note that there is only one copy of the printout before the `fork()`, two of the one after. `fork()` is a very unusual function - you call it once, but it returns twice!

How do we know how these work? See the man pages! The `fork` page tells us what we need to include, and how to use it.

How about `wait`? If we issue the command `man wait`, we get the man page for `builtin`, as there is a built-in `wait` command in the shell. To get the page we want, we need to specify a manual "section." Section 2 is the system calls section. The command `man 2 wait` will get us the page we want.

Are there really two processes? Let's look at the output of `ps` as the program runs.

Again, these system calls let you, as a normal user, do things that only the system should be able to do. Your "user mode" process can get access to "kernel mode" functionality through these calls. But since you are accessing this functionality through the system call interface, there is control over your ability to do so.

How many processes can we create on various systems? Where does this limit come from? Can we create enough processes to take down a system?

We can see this in the `forkbomb` program in the same repository. **Note: you are welcome to try this one out on your VM, but please do not run it on a production system like noreaster. It should not be able to crash the system, but let's not take any chances.**

The FreeBSD implementation of `fork()` is in `/usr/src/sys/kern/kern_fork.c`.

Things to note here (for an example – don't worry about the details):

- the action is happening in `fork1()`, which starts on line 849.
- Line 1012: actual allocation of the new `proc` structure with `uma_zalloc`, a special `malloc` basically (see `zalloc(9)`).
- Line 1074: final checks to make sure a new process is allowed based on global system limits, if so, call `do_fork` to create it at line 1080.

  Note that `nprocs` is the kernel variable that stores the number of active processes in the system, `maxproc` is the upper limit on the number of processes allowed.

  Note also how a regular user is restricted from creating a process if the system is within 10 of the overall limit. (line 934)

We will see that in many cases, you will want to use the `vfork(2)` system call instead of `fork()`. This one doesn't copy the address space of the process – it assumes you are going to replace the newly created process' program with a new one. More on this later.

In the Windows world, there is a `CreateProcess()` Windows API call (the Windows API is like a POSIX for Windows) that creates a new process and loads the correct program into that new process.

Soon we will consider more system calls, including ones that let you do things more interesting than making a copy of yourself.

There is also the issue of how all this gets started – how does that first process get started that `fork`s everything else?

Processes may need to communicate with each other in some more interesting way than making copies of themselves. We will see a number of ways this can be done, including the use of a small chunk of POSIX shared memory.

We will see examples of how to create a shared memory segment that can be accessed by your parent and child process.

Another possibility is to have processes pass messages to each other over the network or through the file system.

We will spend a good chunk of time later this semester on cooperating processes.