Chapter 3: Processes
Wrapping up chapter 2

- We wrapped up all of the critical information in chapter 2 last time.
  - Virtual machines should be read about in the chapter, but we’re not going to talk about them in class.

- One final topic: OS-assisted debugging.
  - Let’s look at dtrace and strace on a real program, and see our program interact with the underlying OS via system calls.
Here forward, we focus on specific topics in detail.

Today, we start with processes.

Processes are the fundamental unit of computing that we as end-users see.
Processes

- What is a process?

- A process is an active instance of a program.

- A program on its own just sits on disk doing nothing.
  - To do something useful, we execute the program.

- Executing a program means:
  - Associating information with the sequence of instructions that give the program context.
  - Allowing the program to modify this context.
Parts of a process

- What goes into a process?
  - Or, to put it another way, what goes into making a program run?

- Program code itself. This is often called the text segment of a process.
- A program counter and set of registers that represent the CPU state.
- A stack used for function invocations by the program.
- A heap for dynamically allocated memory.
- A data segment for global variables.
Process in Memory

max

stack

heap

data

text
Process State

As a process executes, it changes *state*

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution
Diagram of Process State

new → admitted → ready → running → terminated

waiting → I/O or event completion

ready → interrupt → running

running → scheduler dispatch → ready

ready → I/O or event wait → waiting

terminated → exit
Context

- Earlier we said that a process is a program that is executing.

- The execution is represented by some context.

- The OS defines a data structure for representing this context.

- This context is often called the **Process Control Block**.
Process Control Block (PCB)

Information associated with each process

- Process state
  - Ready, waiting, running, etc…
- Program counter
- CPU registers
- CPU scheduling information
  - Example: priorities
- Memory-management information
- Accounting information
- I/O status information
  - Open file handles, I/O devices used by process
Process Control Block (PCB)

- process state
- process number
- program counter

registers

memory limits

list of open files

...
Let’s look at a real one!

- Processes inside OSX.
  - The kernel (XNU) for OSX is open source.

- We might as well check out the system call header so you can see a real implementation of what we talked about last time.

  - Off to the editor…
  - /Users/matt/Desktop/OS415/xnu-1228.9.59/bsd/sys/proc_internal
Why all of this state?

- One of the critical uses of the PCB is to allow processes to be swapped in and out of the CPU.

- You can think of the PCB as everything necessary to reconstitute a running program from being frozen.

- Let’s look at what goes into swapping processes.
CPU Switch From Process to Process

- **process** $P_0$
  - executing
  - interrupt or system call
    - save state into PCB$_0$
    - reload state from PCB$_0$
  - idle

- **operating system**
  - interrupt or system call
    - save state into PCB$_1$
    - reload state from PCB$_1$
  - idle

- **process** $P_1$
  - executing
So, given a process, we want to manage them as they go about their lifetimes.

Recall the state transition table:

Different states represent different groups of processes.
- Those that are running
- Those that are ready to run
- Those that are waiting for something and not ready to run.
Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues
Ready Queue And Various I/O Device Queues

![Diagram showing different queues and PCBs](image)
Representation of Process Scheduling
Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
  - Most often seen in batch environments, like a cluster or server.

- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
Addition of Medium Term Scheduling

This scheduler is responsible for *swapping*. 
Schedulers (Cont)

- Short-term scheduler is invoked very frequently (milliseconds) \(\Rightarrow\) (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) \(\Rightarrow\) (may be slow)
- The long-term scheduler controls the degree of multiprogramming
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
Process mix

- In a multiprogramming environment, we can achieve good utilization if we have a nice mix of processes.
  - IO-bound and CPU-bound

- When IO-bound processes are forced to wait, CPU bound ones can use the time in between.

- Running only IO bound processes will lead to lots of idle time and many in the waiting state.

- Running only CPU bound processes will lead to slow execution of each due to contention for processors.
Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**.
- **Context** of a process represented in the PCB.
- Context-switch time is overhead; the system does no useful work while switching.
- Time dependent on hardware support.
  - Some CPUs provide multiple register sets, which makes swapping faster if processes already loaded into them.
Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Let’s look at the family of processes on a Linux machine to see processes that spawned other processes.
Process Creation (Cont)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it

- UNIX examples
  - `fork` system call creates new process
  - `exec` system call used after a `fork` to replace the process’ memory space with a new program
Process Creation

- fork() initiates a new process (child).
- The child process calls exec().
- The parent process calls wait() to wait for the child process to complete.
- The child process calls exit().