CS 326: Operating Systems

Processes and Daemons

Lecture 6

Today's Schedule

- Processes
- Daemons

Today's Schedule

- **Processes**
- Daemons

Processes

- We briefly touched on **processes** when we studied booting up
- Processes are created with the **fork** system call
- This creates a **clone** of an existing process
- After creating the clone, we know two things:
  - Which process is the parent
  - Which process is the child
- Logic branches from here, allowing the two processes to do different work

Dealing with Clones

- The cloning approach is particularly nice if you want to make your application work on multiple CPUs
- It doesn't quite help us if we want to launch a completely different program, though
- For instance, our program wants to start the `top` command
  - There is another function to accomplish this: `exec`

exec

- The `exec` family of functions allows us to launch other applications
- `exec` replaces the memory space of a clone with a new program and begins its execution
- After `fork()`: copy of my_program
- After `exec()`: separate process running `top` … or whatever you wanted to run!

Demo: fork + exec

Why Split fork + exec?

- Why not just have a nice C function called `launch_program` (or something like that) instead?
  - Or in other words: why does this need to be broken into two steps?
- Advantages of operating this way:
  - While the new process is still a clone, it can set up the target *environment* for the new application
    - Unix pipe mechanism is based on this
  - No restriction on which process will be replaced (could be the parent or child… usually child)

Setting up the Environment

- The new process can inherit several aspects of its predecessor
- Environment variables: the system path, current working directory, global program options
- Redirection: the new process may be set up to receive input on its `stdin` stream from the parent process

Demo: env

Running Processes

- The OS **scheduler** is responsible for ensuring each process gets a share of the CPU
- Rather than letting processes coordinate this, the scheduler **preempts** them
  - **Context switch**: swapping **process control blocks**
  - Each process can pretend it owns the CPU
- When a process is waiting for an I/O device, it gets switched out until the operation completes
  - Interleaves I/O and CPU usage
- Different **scheduling algorithms** have different performance properties

Scheduler Implementation

- scheduler() in proc.c
- Process control block in proc.h
- swtch.S

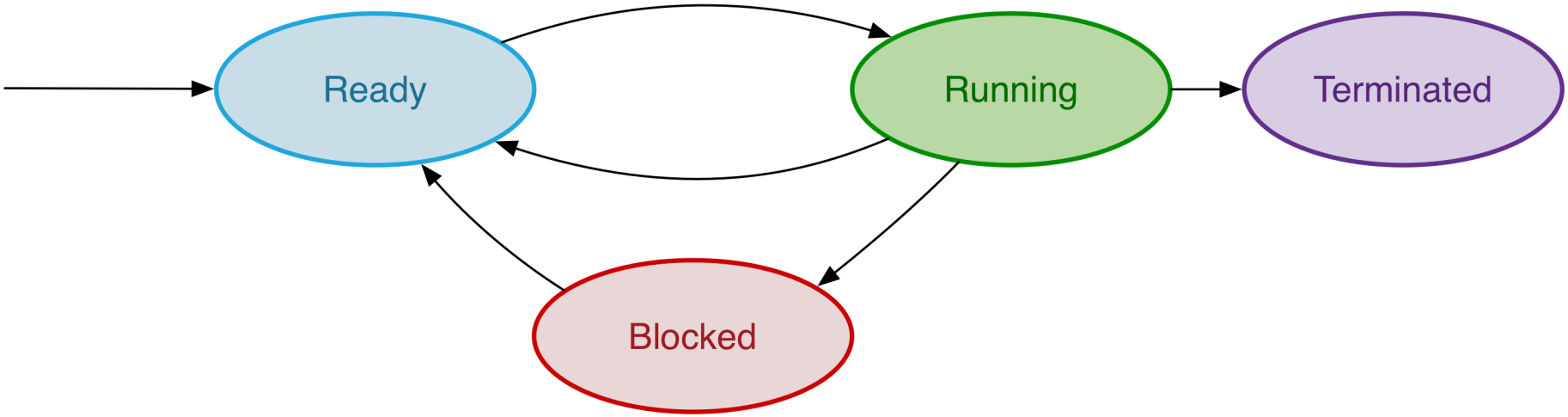
Process Control Block

- Executable machine code
- Memory (isolated from other processes)
  - Call stack
  - Heap
- Metadata
  - Permissions, ownership
  - Open files and access isolation
  - Environment variables
- Context
  - Registers
  - Stack pointer, program counter
  - Memory addressing

State Information

- Information about running processes is stored in **Process Control Blocks (PCBs)**
  - This includes, variables, call stack, heap, etc.
  - Only **one** active PCB per CPU
- As a program runs, the OS may interrupt it to **pause** its execution
  - To make this mechanism work, we need help from the hardware

Basic State Transitions



Process States

On Linux, we have the following possibilities:

- R (running)
- S (sleeping)
- D (disk sleep)
- T (stopped)
- T (tracing stop)
- Z (zombie)
- X (dead)

xv6 States

- Unused
- Used
- Sleeping
- Runnable
- Running
- Zombie

Braaaiiiiiiins

- Zombie processes?! What are those?
- A zombie is a child process that has terminated but still has an entry in the OS process table
- Generally happens when the parent doesn't call `wait()`
  - The process table entry lingers until the exit status is read by the parent
- What's an exit status? Well, that 'return 0' at the end of your C program ain't just there for show!
  - (it returns the exit status)

Termination

- There are three ways a process can terminate:
  1. Clean exit
  2. Error exit
  3. Involuntary exit
- Regardless of the exit type, the OS:
  - Deallocates the memory assigned to the process
  - Closes any open files
  - Releases locks
  - Cancels any callbacks/timers

Today's Schedule

- Processes
- **Daemons**

What's next (after init)?

- Ok, so we have booted our computer and we have processes running!
- Once we're actually running our OS, the **init** system starts several **daemons**
  - Background processes
  - Generally have a 'd' suffix:
    - system**d**
    - syslog**d**
    - dhcpc**d**

Daemons

- Daemons (pronounced demons) are responsible for all kinds of **non-interactive** tasks
  - Logging error messages
  - Configuring the network
  - Accepting ssh sessions
- Similar: Windows **services**, look for `svchost.exe` in task manager



Why “Daemons”?

- If you look up the origin of daemon, you might find “**D**isk **A**nd **E**xecution **MON**itor”
  - Backronym
- They were originally named after Maxwell’s Demon from a thought experiment on the 2nd law of thermodynamics
- Two bodies of different temperatures will eventually reach equilibrium when brought together
- Maxwell’s demon arranges fast- and slow- moving molecules into separate containers to violate this rule

Confronting our Daemons

- Let’s say we want to know what daemons are running on a machine
  - We can use the ps command to list all processes, and we can assume daemons will *usually* end in a ‘d’
- Useful commands:
  - `ps -e --no-header`
  - `awk to print columns`
  - `sed` to remove extra information:
    - `sed 's:\(.*\)/.*:\1:g’`
- Maybe we can turn this into a script?

Are Daemons Processes?

- Daemons do special things, so they're probably what makes up the rest of the OS, right?
- Not really… Most daemons are regular old "user space" programs / processes.
- To perform privileged operations, we need **system calls**
  - Basically the API of your OS
  - Special function calls that traverse the bounds of space and time (or user space, at least…)
    - This is what we'll talk about next