Announcements

- I've finally been assigned a permanent office: 3050 Kemper. Office hours remain the same (T/Th from 3:15-4:30 PM and W from 9:30-10:30 AM)
- The first programming project will be handed out on Monday at discussion section (and posted on the web); Bhume will be giving an introduction to FreeBSD kernel hacking in discussion.

Key Points From Last Lecture...

- **kernel** — the small nucleus of system software that runs in privileged mode and serves as an interface between user programs and physical hardware
- two system states — kernel mode and user mode; when a program is running in "user mode," there are hardware-enforced restrictions on what memory can be accessed and which machine instructions can be executed
- the OS kernel is further divided into a "top half" and a "bottom half"
  - "top half" runs synchronously for a process; system calls are part of the kernel "top half"
  - "bottom half" runs asynchronously for a process; interrupt handlers for hardware events are part of the "bottom half"

What is a process?

process — a program in execution

- program's executable code ("text segment")
- program counter
- program's static and global variables ("data segment")
- stack — local variables and function parameters
- OS bookkeeping information; in FreeBSD, this data is divided into two structures:
  - **process structure** — information that must always remain resident in memory at all times (process credentials, open files, memory allocation assignments, ...)
  - **user structure** — information that is only required while the process is running (signal handlers, debugging information)

How are processes created?

New processes are created by calling the `fork()` system call:

```
int fork(void);
```

`fork()`'s purpose is simple — it makes an identical copy of the currently running process, including all memory in the process' address space, the current program counter, etc.

Execution of both processes (parent and child) continues from the point of the `fork` call.

fork() example

```
int main(void) {
    int x = 0;

    printf("x = %d\n", x);
    fork();
    x++;
    printf("x+1 = %d\n", x);
}
```

fork() example

```
int main(void) {
    int x = 0;

    printf("x = %d\n", x);
    fork();
    x++;
    printf("x+1 = %d\n", x);
}
```

Output:

```
x = 0
x+1 = 1
x+1 = 1
```

`fork()`, continued

But what's the point of cloning a process if they do the exact same thing? Can we distinguish between the parent and the child process?

Yes! From the `fork()` manpage:

> *On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created, and errno will be set appropriately.*

So we can test the return value to figure out whether the currently executing process is the parent or child.

fork() example

```
int main(void) {
    int pid;

    printf("x = %d\n", x);
    pid = fork();
    if (pid == 0) {
        printf("I am the child process!\n");
    } else {
        printf("I am the parent process; my child
                is process #%d\n", pid);
    }
}
```

fork() example

```
int main(void) {
    int pid;

    printf("x = %d\n", x);
    pid = fork();
    if (pid == 0) {
        printf("I am the child process!\n");
    } else {
        printf("I am the parent process; my child
                is process #%d\n", pid);
    }
}
```

Output:

```
I am the child process!
I am the parent process; my child is process 31100
```

execve()

Spawning identical children would obviously be useful in some situations (e.g., a web server that has to handle multiple simultaneous requests), but what if we want to execute a *different* program?

Another system call:

```
int execve(char* filename,
           char* argv,
           char* envp);
```

`execve()` wipes the current address space clean, then starts execution of the specified program.

(so if you don't want the current program destroyed, you need to fork a second copy before calling execve)

waitpid()

`fork()` and `execve()` can be used together to spawn a new program. But what if we want our original program to stop and wait for the second program to finish before continuing execution?

One more system call:

```
pid_t waitpid(pid_t pid, int* status,
              int options);
```

- `pid` is the Process ID of the process to wait for; use -1 to wait for any child process
- `status` is a pointer to an integer where the exit status of the process will be stored (i.e., successful completion, killed by a signal, etc)
- `options` specifies how this system call should behave; see the manpage (`man 2 waitpid`) for details

A simple Unix shell

You should now have all the tools you need to write a (very simple) Unix shell. Try to fill in the following missing parts of this simple shell program:

```
int main(void) {
    char command[1024];
    char parameters[1024];
    // define any extra vars you need

    while(1) {
        printPrompt();
        getCommand(&command, &parameters);

        // your code goes here
    }
}
```

System call prototypes

```
int fork(void);
int execve(char* filename,
           char* argv,
           char* envp);
pid_t waitpid(pid_t pid, int* status,
              int options);
```

Solution:

```
int main(void) {
    char command[1024];
    char parameters[1024];
    int status;

    while(1) {
        printPrompt();
        getCommand(&command, &parameters);

        if (fork() != 0)    // parent process
            waitpid(-1, &status, 0);
        else                // child process
            execve(command, parameters, NULL);

    }
}
```

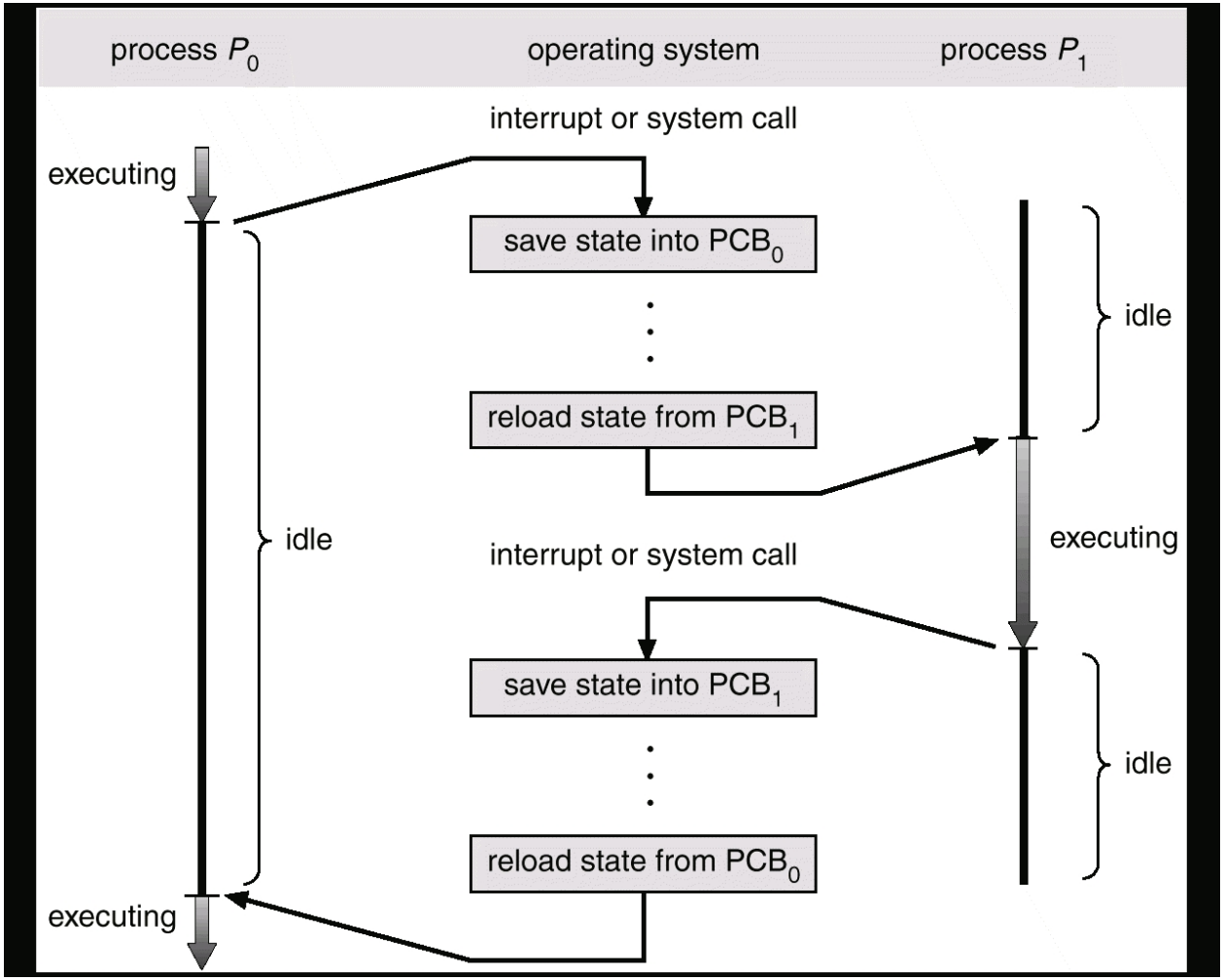
Context Switching

FreeBSD (and all modern operating systems) support transparent multiprogramming — the *illusion* of concurrent process execution.

> **context switch** — *the act of storing the state (context) of the CPU for the currently running process and loading the saved state of another process.*

Context switching is a relatively expensive operation due to the amount of copying that needs to be performed.

Context Switching



Scheduling

Clearly minimizing the number of unnecessary context switches is a important to system performance.

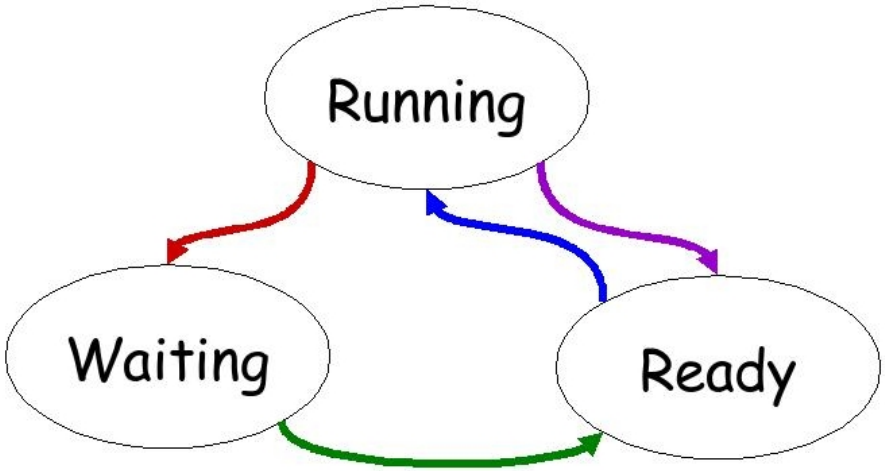
The basic problem:

- at any given time, several processes are ready to run
- only N processes can run at a time (where N=number of processors)

Selection of which process to run at any given time is handled by the operating system's scheduler; many different scheduling algorithms exist.

Process States

- running — process is currently executing on the processor
- ready — process is not currently executing, but is ready to be scheduled
- waiting — (or "blocked") process is waiting for an event before it can continue running (I/O, child process completion, etc.)



Process Scheduling

Process scheduling can be implemented in one of two ways:

- **preemptive scheduling** — the running process can be forced to temporarily stop executing and let another process run
- **non-preemptive scheduling** — a running process continues to run until it blocks on an event, voluntarily yields control of the processor, or finishes execution

We'll focus on preemptive scheduling.

Preemptive Scheduling

The system clock is the key to preemptive scheduling.

- a chip on the motherboard of the computer
- sends a periodic signal to the CPU; frequency (HZ) can be set by the OS; Unix-based operating systems usually use values of 100, 1000, or something in between.
- signal triggers a timer interrupt, which causes execution to jump into the bottom half of the kernel and make scheduling decisions

The amount of time that a process is allowed to run before being interrupted is called the **quantum**, or **time slice**; usually measured in number of clock ticks.

How do we decide which process to schedule?

One option: simple first come, first serve scheme

- place all runnable processes on a single queue called the "ready" list
- when a scheduling decision needs to be made, simply take the process on the head of the queue and execute it
- whenever a running process uses up its quantum or a waiting process is unblocked, it will be placed on the tail of the queue

Pros: very simple scheduling algorithm; easy to implement and quick to execute (remember that process scheduling decisions may be made thousands of times per second)

Cons:

How do we decide which process to schedule?

One option: simple first come, first serve scheme

- place all runnable processes on a single queue called the "ready" list
- when a scheduling decision needs to be made, simply take the process on the head of the queue and execute it
- whenever a running process uses up its quantum or a waiting process is unblocked, it will be placed on the tail of the queue

Pros: very simple scheduling algorithm; easy to implement and quick to execute (remember that process scheduling decisions may be made thousands of times per second)

Cons: Not fair for programs that are heavily I/O based. . . can result in high latency and "lag"

Process Classifications

We need some way to distinguish between different types of processes so that we can prioritize them. Here's one possible categorization:

- **I/O bound** — programs that perform lots of I/O operations with relatively little processing inbetween (text editors, web browsers, etc.)
- **CPU bound** — programs with little or no I/O (e.g., scientific software and simulations

Process Classifications

We need some way to distinguish between different types of processes so that we can prioritize them. Here's one possible categorization:

- **I/O bound** — programs that perform lots of I/O operations with relatively little processing inbetween (text editors, web browsers, etc.)
- **CPU bound** — programs with little or no I/O (e.g., scientific software and simulations
- **hybrid** — multimedia programs (video players, music players, etc.)

The operating system can make guesses at which type of application a process is based on how often it uses up its entire quantum and is preempted vs how often it blocks on I/O.

Process Classifications

Other classifications are also possible:

- **interactive** — a process where responsiveness should be the priority and actual throughput is secondary
- **batch** — processes for which throughput and overall execution time is most important (may or may not do a lot of I/O)
- **realtime** — processes that **need** to complete by a specific deadline; most often encountered in embedded systems

Performance Measurement Metrics

Important metrics for performance evaluation:

- **throughput** — # of processes completed per unit time
- **turnaround time (T)** — real world time to complete a process
- **service time (S)** — total amount of time a process needs to run on the CPU to finish
- **response ratio** — "normalized turnaround time" $= \frac{T}{S}$
- **response time** — time taken to respond to user input (keystrokes, command entered into shell, etc.)