CS5460: Operating Systems

Lecture 3: Context Switch

(Chapters 4, 5, 6)

Slide Credit: Andrea Arpaci-Dusseau

Assignment 1

- Due Tue Feb 2
  - 1 week left!

Providing Good CPU Performance?

Direct execution

- Allow user process to run directly on hardware
- OS creates process, transfers control to start point (i.e., main())

Problems with direct execution?

1. Process could do something restricted

   Could read/write other process data (disk or memory)

2. Process could run forever (slow, buggy, or malicious)

   OS needs to be able to switch between processes

3. Process could do something slow (like I/O)

   OS wants to use resources efficiently and switch CPU to other process

Solution:

Limited direct execution – OS & hw maintain some control

Problem #1: Restricted Ops

How can we ensure user process can't harm others?

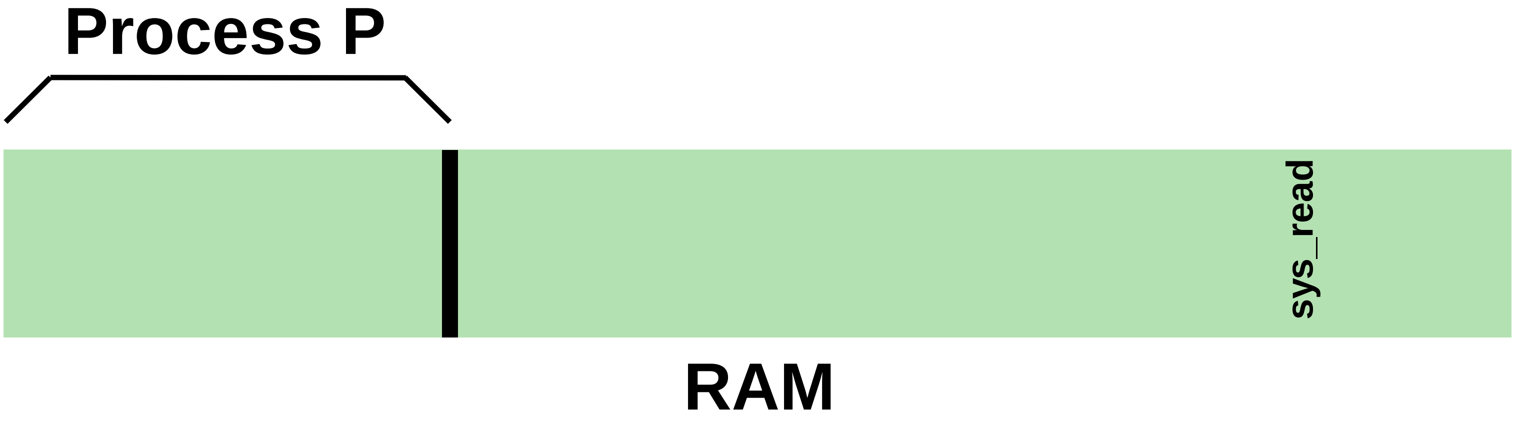
Solution: privilege levels supported by hw (status bit)

- User processes run in user mode (restricted mode) (Ring 3)
- OS runs in kernel mode (not restricted) (Ring 0)
  - Instructions for interacting with devices
  - Access to all memory
  - Ability to reconfigure CPU control registers (IDT, PTBR/CR3)

How can processes access devices?

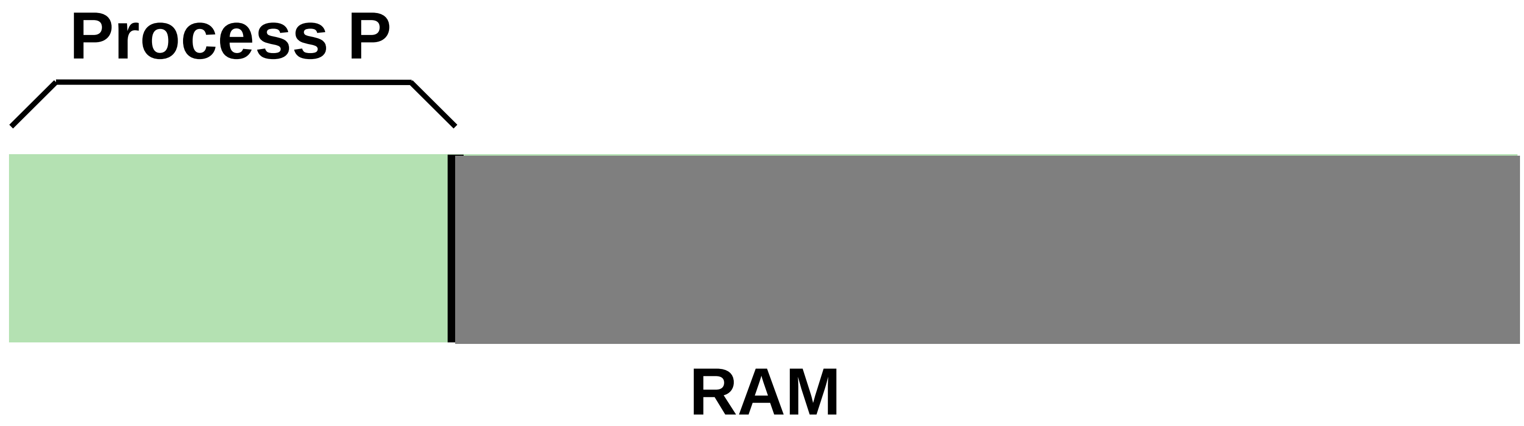
- System calls (function call implemented by OS)
- Change privilege level through system call (trap)

System Call



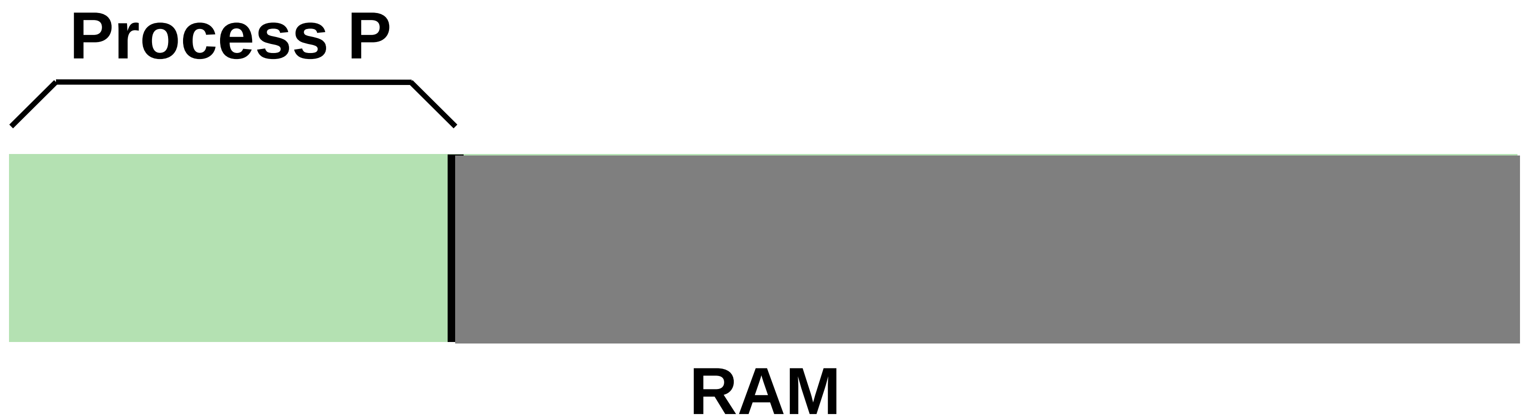
P wants to call read()

System Call



P can only see its own memory because of user mode (other areas, including kernel, are hidden)

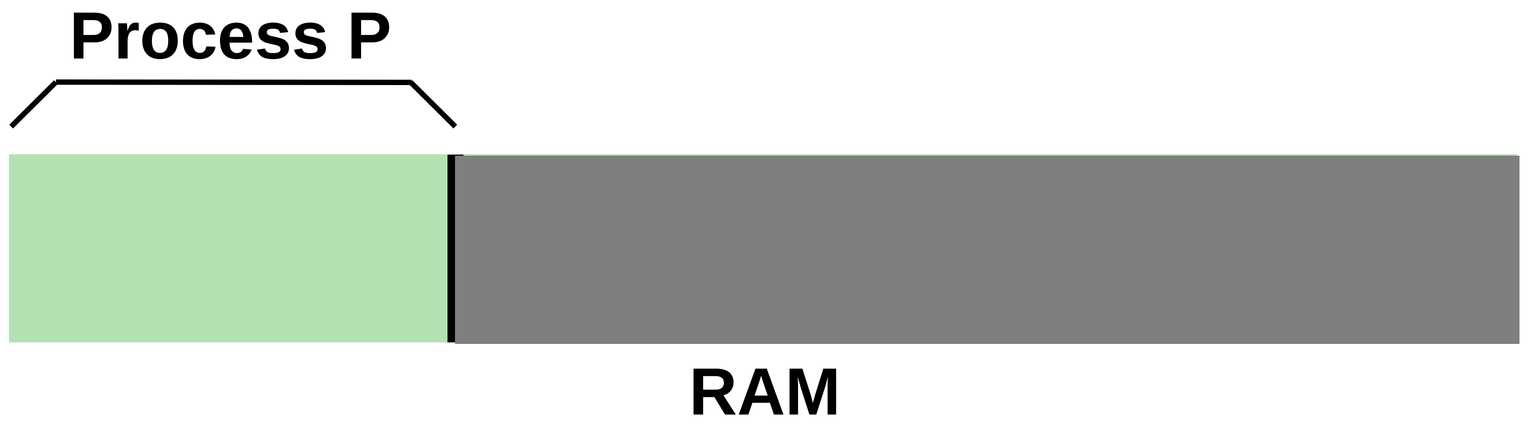
System Call



P wants to call read() but no way to call it directly

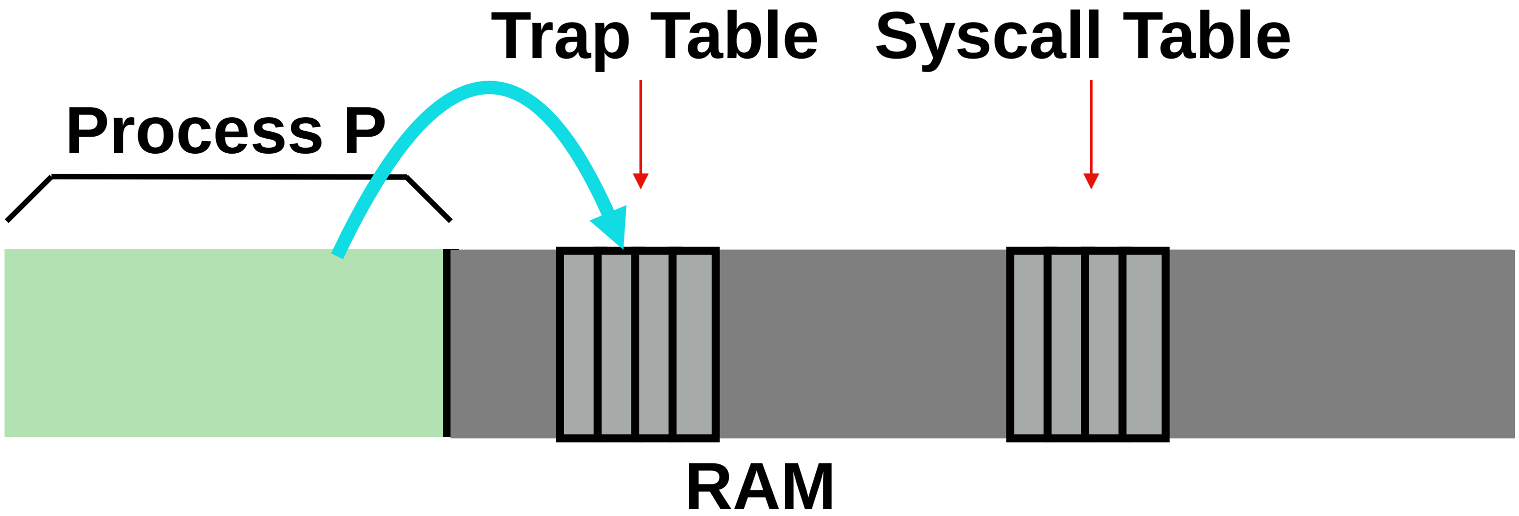
~~`callq sys_read`~~

System Call



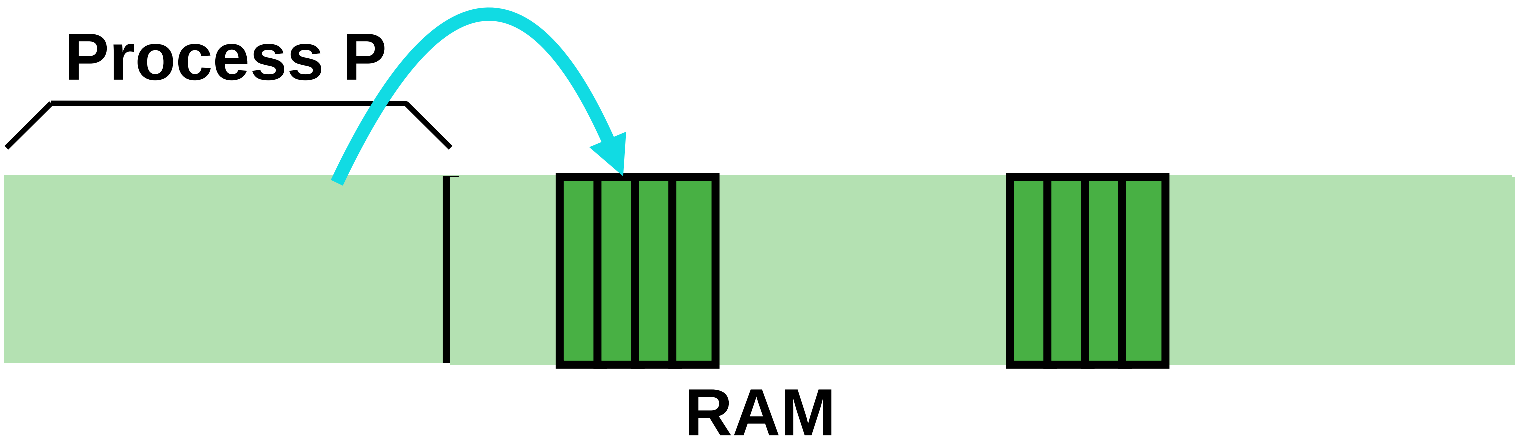

```
movl $6, %eax;   int $64
```

System Call




```
movl $6, %eax;    int $64
```

System Call



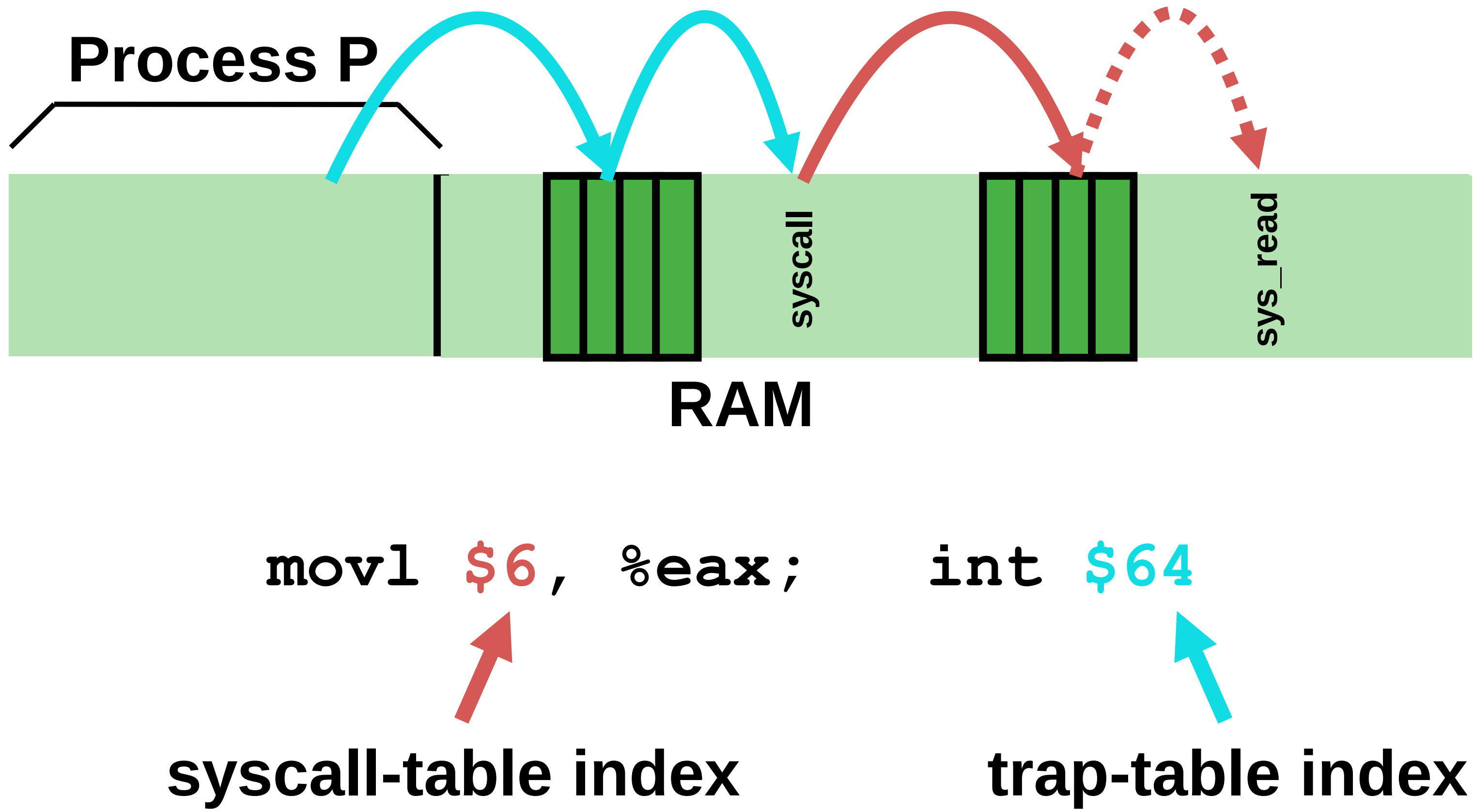

```
movl $6, %eax;    int $64
```

syscall-table index　　　　trap-table index

Trap instruction → kernel mode, vectors to trap handler

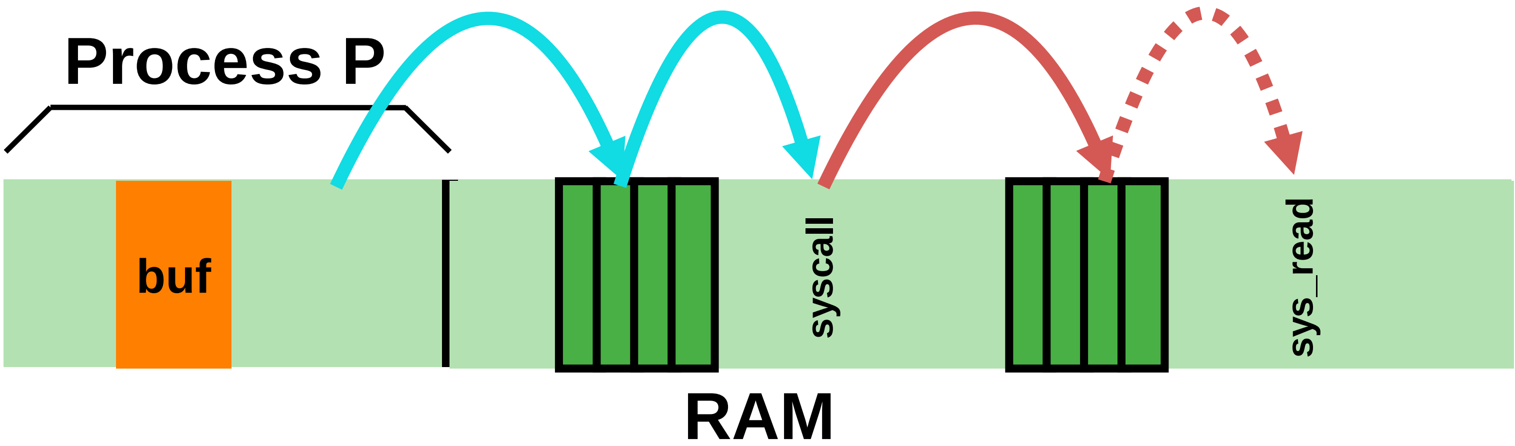
Kernel mode: we can do anything!

System Call



Follow entries to correct system call code

System Call




```
movl $6, %eax;   int $64
```

syscall-table index   trap-table index

Kernel can access user memory to fill in user buffer
return-from-trap at end to return to Process P

What do we need to limit?

User processes are not allowed to perform:

- General memory access
- Disk I/O
- Special x86 instructions like `lidt`

What if process tries to do something restricted?

Problem #2: Take CPU Away?

OS requirements for multiprogramming (multitasking):

- Policy: Decision-maker optimizing a performance metric
  - Process Scheduler: Which process when?
- Mechanism: Low-level code that implements the decision
  - Dispatcher and Context Switch: How?

Example of separation of policy and mechanism

Dispatch Mechanism

OS runs dispatch loop

```
while (1) {
      run process A for some time-slice
      stop process A and save its context
      load context of another process B
}
```

Context-switch

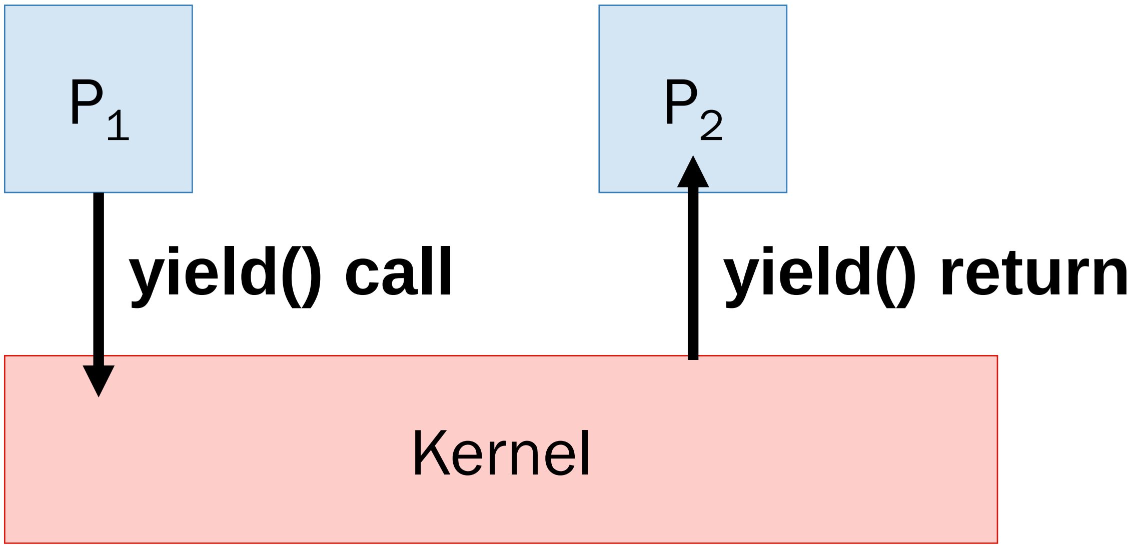
Question 1: How does OS/dispatcher gain control?

Question 2: What execution context must be saved /restored?

Q1: How does OS get control?

Option 1: Cooperative Multitasking

- Trust process to relinquish CPU to OS through traps
  - Examples: System call, page fault (access page not in main memory), or error (illegal instruction or divide by zero)
  - Provide special `yield()` system call



Q1: How does OS get control?

- Problem with cooperative approach?
- Disadvantages: Processes can misbehave
  - By avoiding all traps and performing no I/O, can take over entire machine
  - Only solution: Reboot!
- Not performed in modern operating systems

Q1: How does OS get control?

Option 2: Preemptive Multitasking

- Guarantee OS can obtain control periodically
- Enter OS by enabling periodic alarm clock
  - Hardware generates timer interrupt (CPU or separate chip)
  - Example: Every 10ms
- User must not be able to mask timer interrupt
- Dispatcher counts interrupts between context switches
  - Example: Waiting 20 timer ticks gives 200 ms time slice
  - Common time slices range from 4 ms to a few hundred ms

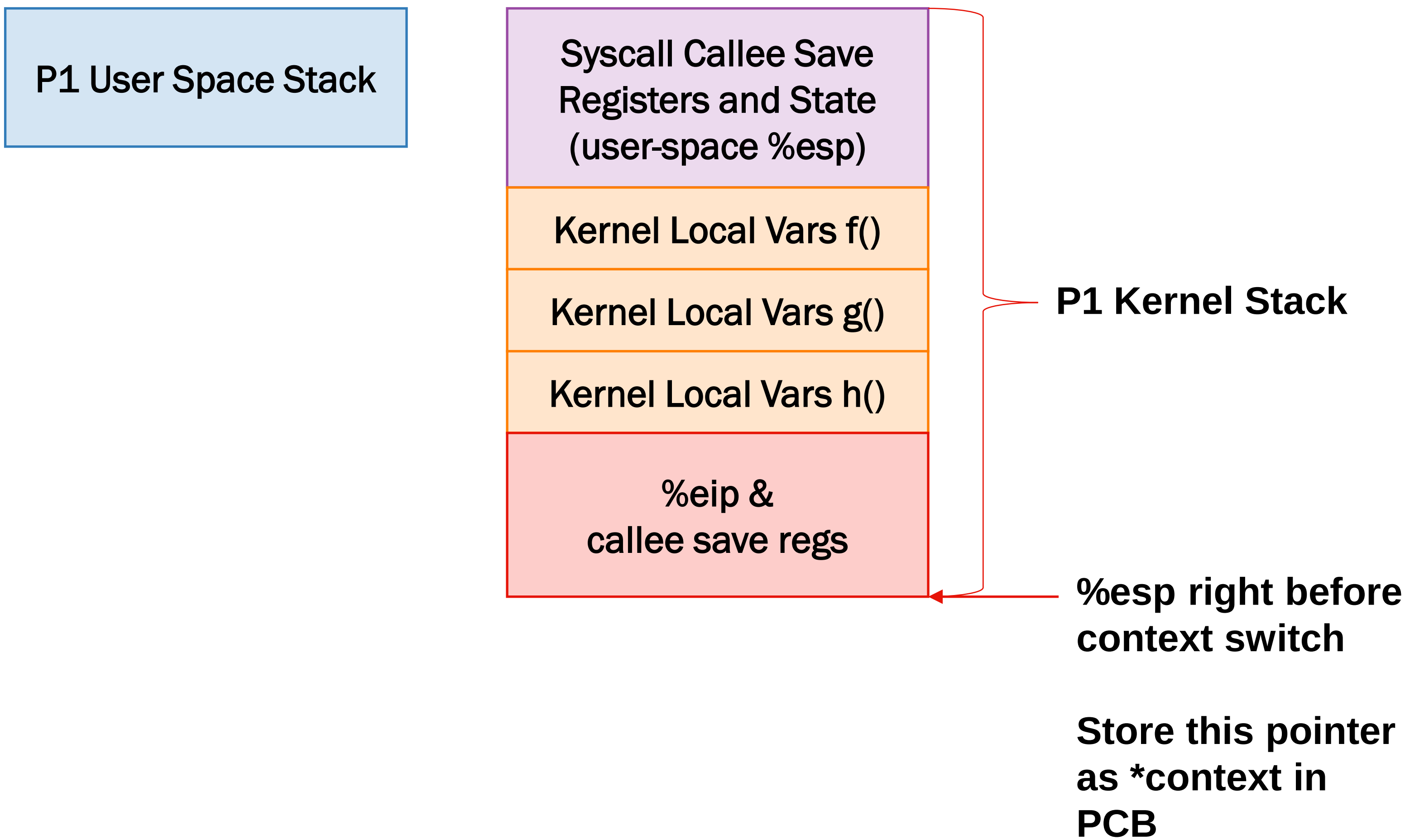
Q2: What context to save?

- Process Control Block: where dispatcher stores context of process when not running; contains
  - PID
  - Process state (i.e., running, ready, or blocked)
  - **Execution state (all registers, instruction ptr, stack ptr)**
  - Scheduling priority
  - Accounting information (parent and child processes)
  - Credentials (which resources can be accessed, owner)
  - Pointers to other allocated resources (e.g., open files)

How this stuff is handled is a bit tricky

- On fork: allocate PCB, initialize, put on ready queue (queue of runnable processes)
- On exit: clean up all process state (close files, release memory, page tables, etc)

Zooming In: Registers & KStack



Operating System	Hardware	Program
		Process A **…**
	Syscall or timer interrupt Hw switches to kstack Raises to kernel mode Save regs(A) to kstack(A) Jump to trap handler	
Handle the trap Call **swtch()** routine Save regs(A) to PCB(A) Restore regs(B) from PCB(B) Switch to kstack(B) Return-from-trap (into B)		
	Restore regs(B) from kstack(B) Move to user mode Jump to B's IP	
		Process B **…**

xv6 PCB

```
struct context
{
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

```
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
struct proc {
  uint sz;                     // Proc mem size (bytes)
  pde_t* pgdir;                // Page table
  char* kstack;                // Bottom of kstack
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc* parent;         // Parent process
  struct trapframe* tf;        // Trap frm for syscall
  struct context* context;     // swtch() here to run
  void* chan;                  // If !0, sleep on chan
  int killed;                  // If !0, been killed
  struct file* ofile[NOFILE];  // Open files
  struct inode* cwd;           // Current directory
  char name[16];               // Process name
};
```

Context Switch

- Context switches are fairly expensive
  - Time sharing systems do 100-1000 context switches per second
  - When? Timer interrupt, packet arrives on network, disk I/O completes, user moves mouse, …

- lab2-15 3.8 μs
- gamow 1.6 μs
- home 1.0 μs

- How might one go about measuring this?

Problem #3: Slow Ops (I/O)?

On op that does not use CPU,
OS switches to other processes

OS must track process states:

Running:

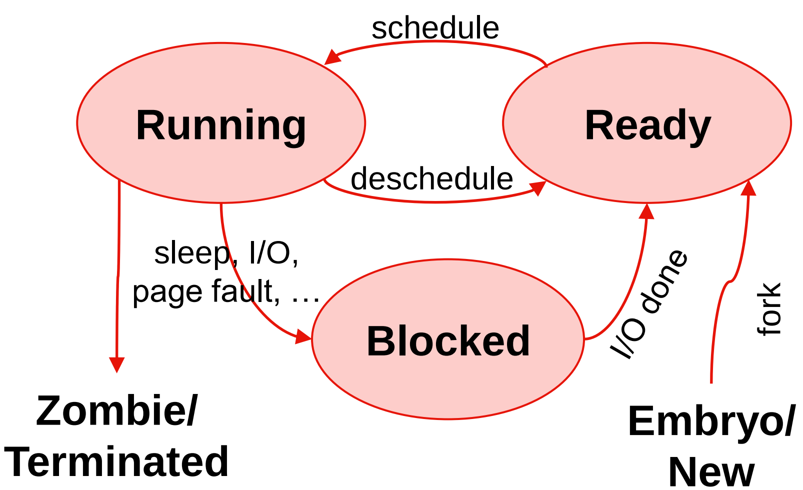
On the CPU (1 on a uniprocessor)

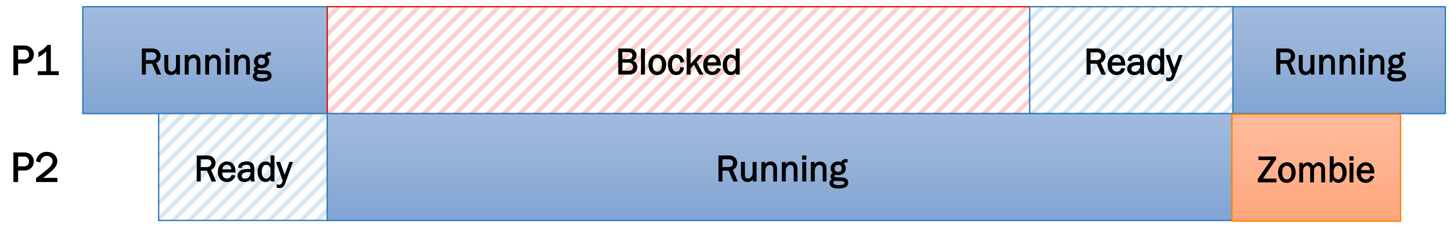
Ready:

Waiting for the CPU

Blocked:

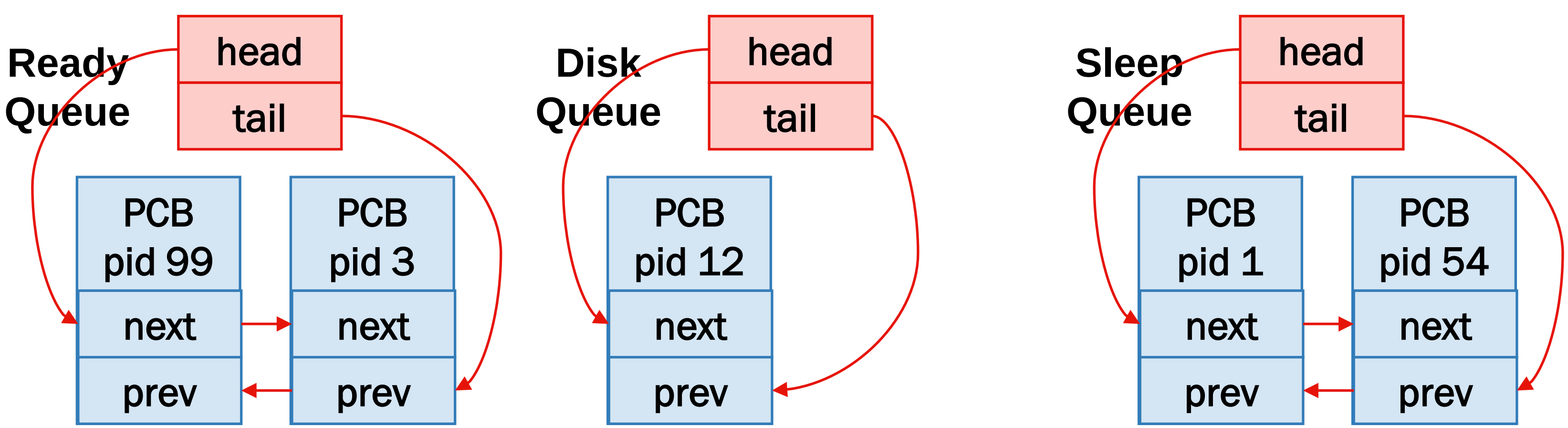
Asleep: Waiting for I/O or synchronization to complete





Problem #3: Slow Ops (I/O)?

- OS maintains queues of all PCBs
  - Ready queue: Contains all ready processes
  - Event queue: One logical queue per event
    - e.g., disk I/O and locks
    - Contains all processes waiting for that event to complete
- Invariant: each process in 1 state and on 1 queue



CPU Virtualization Summary

- Virtualization:
  Context switching gives each process impression it has its own CPU
- Direct execution makes processes fast
- Limited execution at key points to ensure OS retains control
- Hardware provides a lot of OS support
  - user vs kernel mode
  - timer interrupts
  - automatic register saving on syscall

Process Management

- OS manages processes:
  - Creates, deletes, suspends, and resumes processes
  - Schedules processes to manage CPU allocation
  - Manages inter-process communication and synchronization
  - Allocates resources to processes (and takes them away)

- Processes use OS functionality to cooperate
  - Signals, sockets, pipes, files to communicate

Practical Process Management

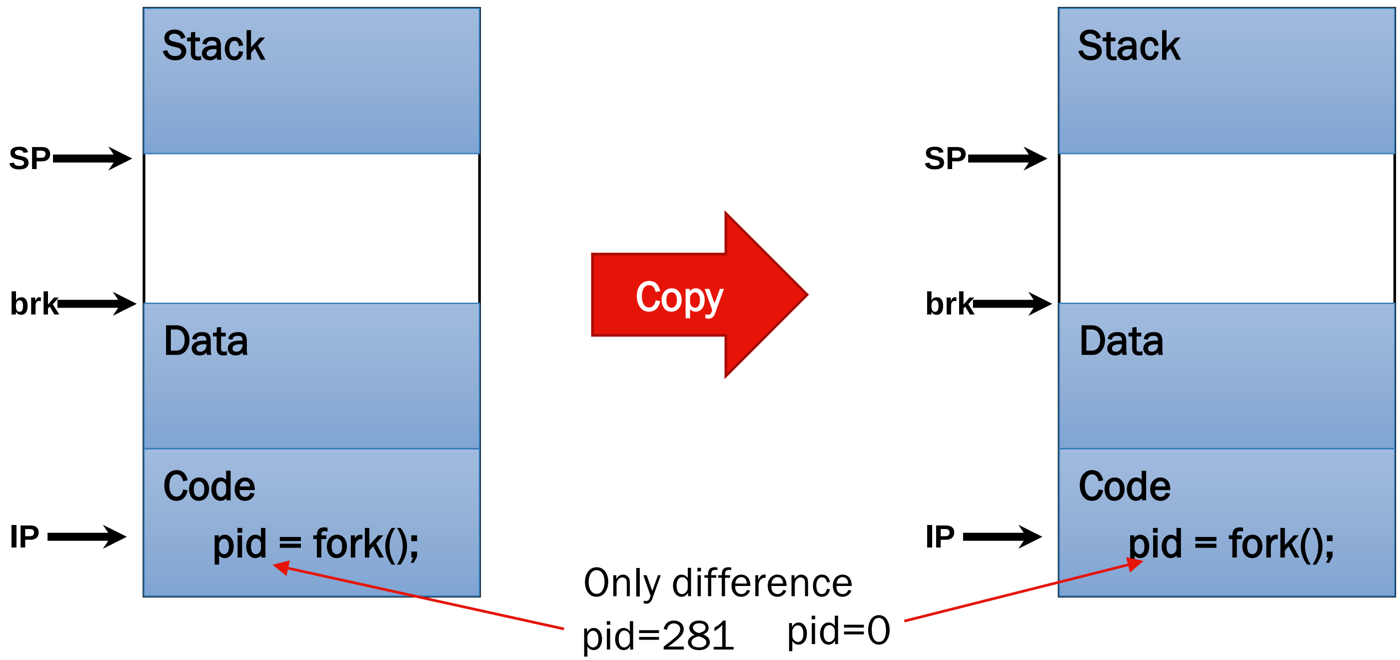
- On a Unix machine, try:
  - ps –Af: lots of info on all running processes
  - kill –9 547: terminates process with PID 547
  - top : displays dynamic info on top running jobs
  - Write a program that calls:
    - getpid(): returns current process's PID (process id)
    - fork(): create a new process
    - wait(): wait for exit of a child process
    - exec(): load a new program into the current process
    - sleep(): puts current process to sleep for specified time
- Commands work on macOS
- On Windows → Task manager (CTL-ALT-DEL)

Creating Processes: fork()

- Creates process that is near-clone of forking parent
  - Address space and running state is cloned
- Return of fork() differs:
  - 0 in child
  - child PID in parent
- Many kernel resources are shared
  - open files and sockets

- wait() lets a process wait for the exit of a child
- To spawn new program, use some form of exec()

Semantics of fork()



- fork(), exit(), and exec() are weird!
  - fork() returns twice – once in each process
  - exit() does not return at all
  - exec() usually “does not return”: replaces process’ program

fork() and wait()

```
int main(int argc, char *argv) {
  printf("parent %d\n", (int)getpid());
  pid_t rc = fork();
  assert(rc >= 0)

  if (rc == 0) {                   // child
    printf("child %d\n", (int)getpid());
  } else {                         // parent
    wait(NULL);
    printf("%d is parent of %d\n", (int)getpid(), rc);
  }
  return 0;
}
```

```
$ ./fork-wait
parent 13037
child 13039
13037 is parent of 13039
```

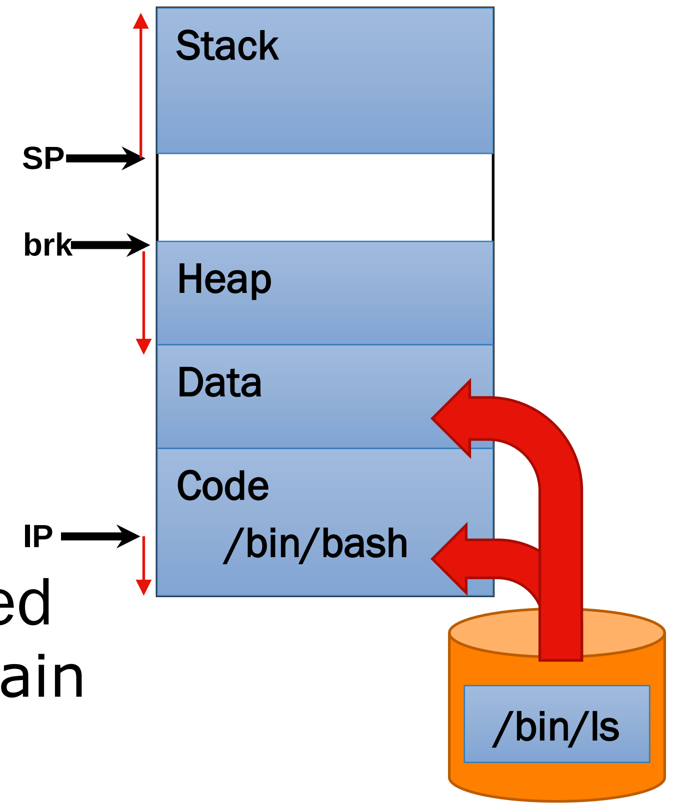
What is the output without wait()?

exec(): Run Another Program

- Can't write entire system in one program!
  - Need to replace process with another program

- Loads program from filesystem
  - Replaces code, data, bss
  - Put argv on stack; reset %esp
  - Release heap memory
  - Reset %eip to main
    (really _start)

- exec() only returns to caller if failed
  - Otherwise process is now in a new main



Why Separate fork and exec?

- Lots of parameters on creating a process
  - Shell may want to
  - redirect output of children
  - change child environment
  - change child working directory
  - run child as a different user
- Hard to create simple, expressive-enough API
- Separation allows policy to be expressed in parent's program but in child's process
- Tradeoff: child may inherit things it doesn't need
  (or shouldn't have)

Example: Output Redirection

```
pid_t rc = fork();

if (rc == 0) {                  // child
  close(STDOUT_FILENO);         // close fd 1
  open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU); // new fd 1

  const char *myargs[3];
  myargs[0] = "wc";
  myargs[1] = "p4.c";
  myargs[2] = NULL;
  execvp(myargs[0], myargs);   // runs “wc p4.c > p4.output”
} else {                        // parent
  wait(NULL);
}
```

Termination: exit(), kill()

- When process dies, OS reclaims resources
  - Record exit status in PCB
  - Close files, sockets
  - Free memory
  - Free (nearly) all kernel structures

- Process terminates with exit()
- Process terminates another with kill()

```
int main(int argc, char* argv[]) {
  pid_t pid = fork();
  if (pid == 0) {
    sleep(10);
    printf("Child exiting!\n");
    exit(0);
  } else {
    sleep(5);
    if (kill(pid, SIGKILL) != -1)
      printf("Sent kill!\n");
  }
}
```

Orphans and Zombies

```
systemd─┬─acpid
        ├─5*[agetty]
        ├─cron
        ├─login───bash───compute
        ├─rsyslogd
        ├─sshd─┬─sshd───sshd───rsync───rsync
        │      └─sshd───sshd───bash───pstree
        ├─systemd-journal
        └─systemd-udevd
```

- Parent `wait()` on child returns status
- Must keep around PCB with status after child exit
- **Zombie**: exited process with uncollected status
- Parent exits before child? **Orphaned**
  - `init` adopts orphans
  - Collects and discards status of reparented children after exit
  - Useful for “daemons” (nohup)

Important Terms and Ideas

- Process, programs
- Process Control Blocks
- syscall, user/kernel mode
- Dispatcher, context switch
- Process State Machine
- New, Ready, Running, Blocked, Terminated
- fork(), wait(), exec(), exit(), kill()
- Orphans, zombies, init