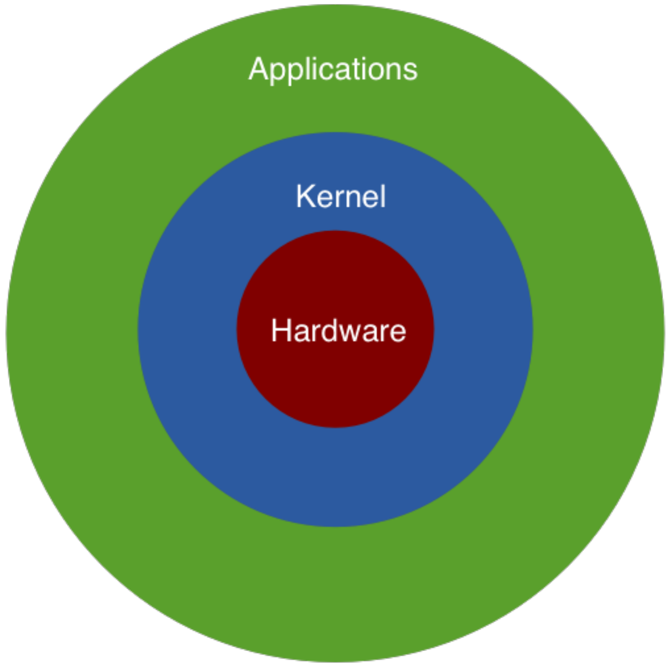
Exam 1 Review

Kernel



Kernel - The part of the OS that implements basic functionality and is always resident in memory.

© 2015 Trevor Bakker and The University of Texas at Arlington

The Operating System as an Extended Machine

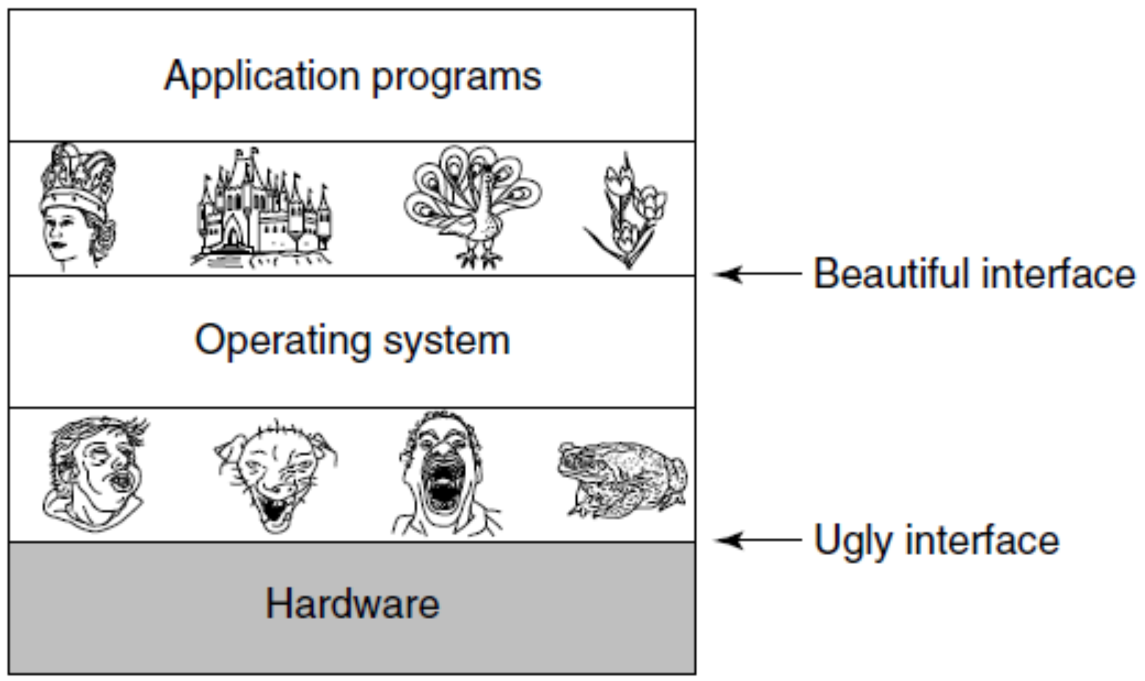


Figure 1-2. Operating systems turn ugly hardware into beautiful abstractions.

The Operating System's Role

- Provide the user with a cleaner model of the computer
  - An abstracted interface to the resources

» Manage the resources

Interrupts

- Mechanism used by the OS to signal the system that a high-priority event has occurred that requires immediate attention.
  - I/O drives a lot of interrupts. Mouse movements, disk reads, etc
- The controller causing the interrupt places the interrupt number in an interrupt register. The OS must then take action

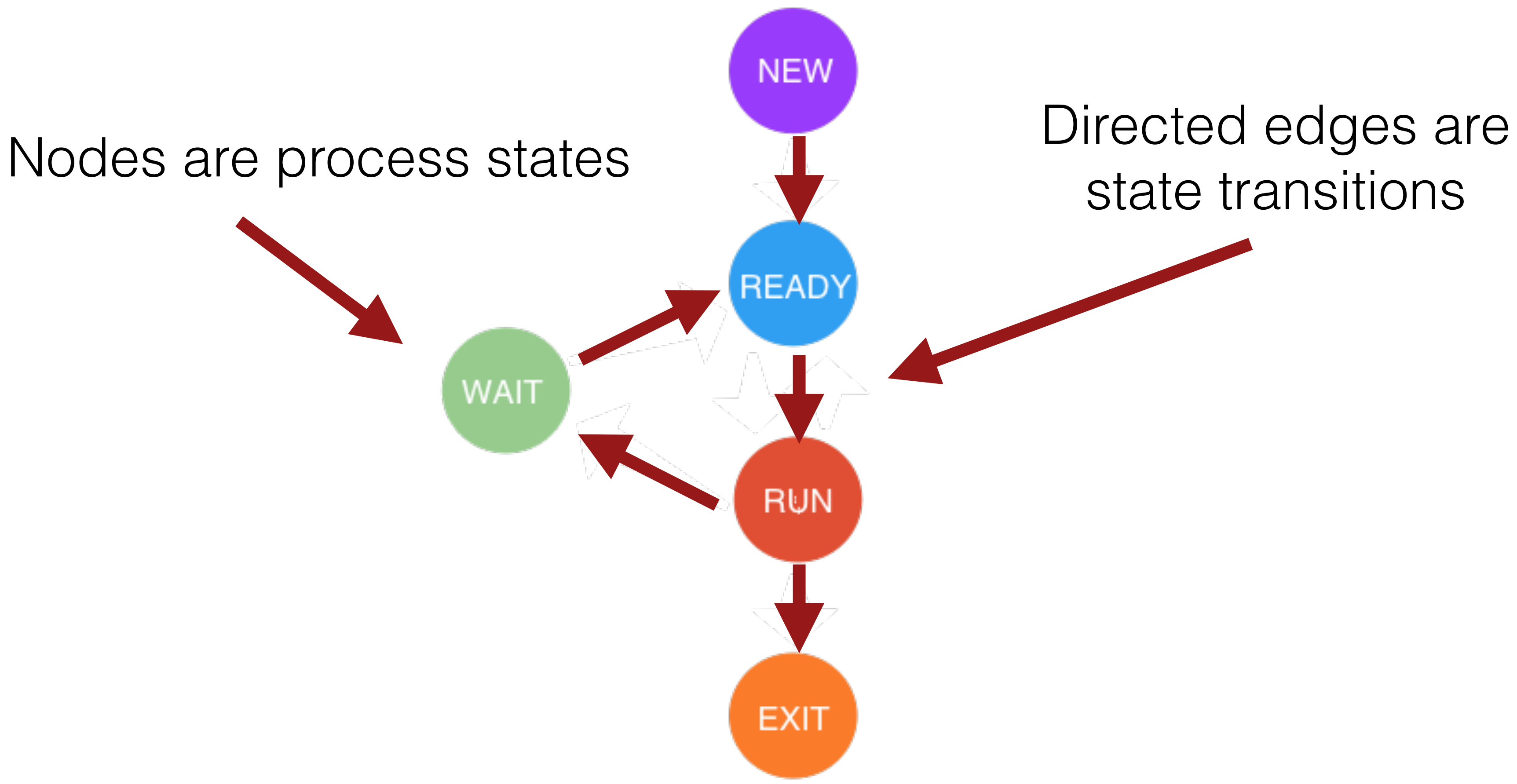
Interrupt Vector

- Normal technique for handling interrupts is a data structure called the interrupt vector.
- One entry for each interrupt.
- Each entry contains the address for the interrupt service routine.
- Some small hardware devices don’t provide an interrupt system and instead uses an event loop. This is known as a status-driven system.

Programs v. Processes

- Program - a sequence of instructions written to perform a specified task.
- Process - an instance of a program in execution.
  - Process is associated with an address space
- A computer program is a passive collection of instructions; a process is the actual execution of those instructions.

Process State Diagram



Parent and children PIDs

- Why does fork return the child's PID to parent processes but it returns 0 to the children?
  - Provides a way to determine which process you are in.
  - Processes can only have one parent. To determine the parent PID you can call getppid().
  - Processes can have multiple children so there is no way for a function to give a parent its child PID

Process Creation

- After fork() both the parent and child continue executing with the instruction that follows the call to fork().
- The child process is a copy of the parent process. The child gets:
    - copy of the parent's data space
    - copy of the parent's heap
    - copy of the parent's stack
    - text segment if it's read-only

Copy-On-Write

- On Linux the parent process's pages are not copied for the child process.
- The pages are shared between the child and the parent process.
    - Pages are marked read-only
- When either process modifies a page, a page fault occurs and a separate copy of that particular page is made for that process which performed the modification.
- This process will then use the newly copied page rather than the shared one in all future references. The other process continues to use the original copy of the page

Inherited Properties

- user ID, group ID
- process group ID
- controlling terminal
- current working directory
- root directory
- file mode creation mask
- signal mask
- environment
- attached shared memory segments

Unique Properties

- the return value from fork()
- the process IDs
- the parent process IDs
- file locks
- pending alarms are cleared for the child
- the set of pending signals for the child is set to zero

Process Execution Modes

- Privileged - OS kernel processes which can execute all types of hardware operations and access all memory
- User Mode - Can not execute low-level I/O. Memory protection keeps these processes from trashing memory owned by the OS or other processes.

How does the OS track a process?

- Each process has a unique process identifier, or PID
  - The POSIX standard guarantees a PID as a signed integral datatype.
  - The datatype is an opaque type called pid_t

Process Control Block

- The kernel maintains a data structure to keep track of all the process information called the process control block or (PCB).
- The PCB also includes pointers to other data structures describing resources used by the process such as files (open files table) and memory (page tables).
- Maintains the state of the process
    - Over 170+ fields

Process Control Block

- Every task also needs its own stack
- So every task, in addition to having its own code and data, will also have a stack-area that is located in user-space, plus another stack-area that is located in kernel-space
- Each task also has a process-descriptor which is accessible only in kernel-space

Process Tables

- The OS holds the process control blocks in the process table
- Usually implemented as an array of pointers to process control block structures
  - Linux calls the PCB task_struct

Process Creation

- Processes are created when an existing process calls the fork() function
- New process created by fork is called the *child process*
- fork() is a function that is called once but returns twice
    - Only difference in the return value. Returns 0 in the child process and the child's PID in the parent process

fork() code example

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
  pid_t pid = fork();

  if (pid == -1) {
    // When fork() returns -1, an error happened.
    perror("fork failed");
    exit(EXIT_FAILURE);
  }
  else if (pid == 0) {
    // When fork() returns 0, we are in the child process.
    printf("Hello from the child process!\n");
    fflush(NULL);
    exit(EXIT_SUCCESS);
  }
  else {
    // When fork() returns a positive number, we are in the parent process
    // and the return value is the PID of the newly created child process.
    int status;
    (void)waitpid(pid, &status, 0);
    printf("Hello form the parent process!");
    fflush(NULL);
  }
  return EXIT_SUCCESS;
}
```

I/O is buffered. Make sure to use `flush()`

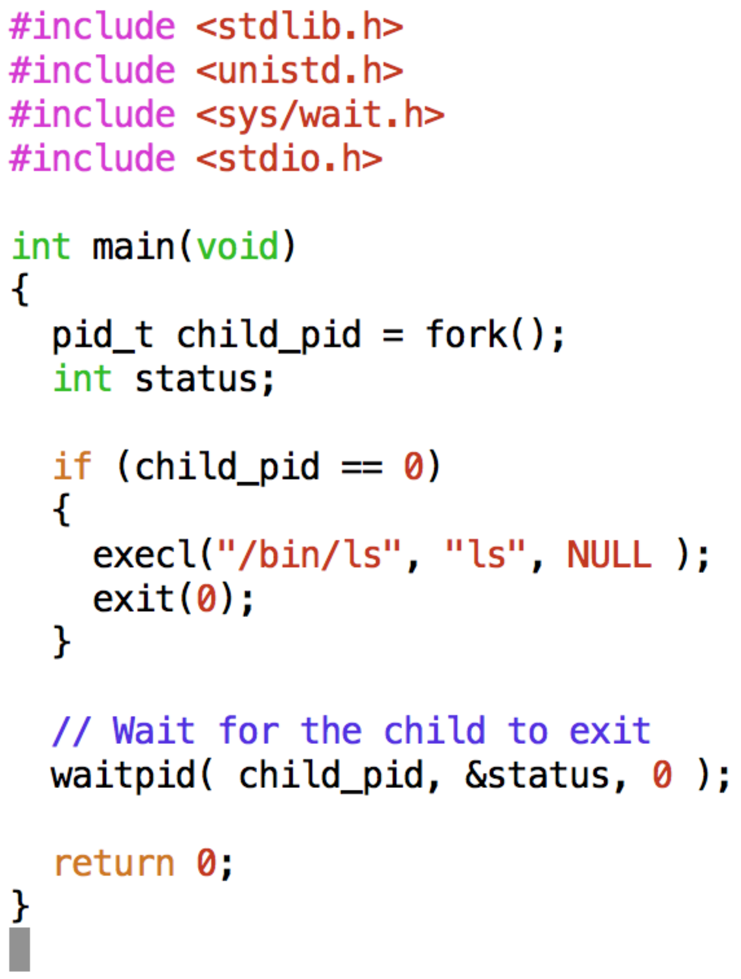
waitpid() code example

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
  pid_t child_pid = fork();
  int status;

  if (child_pid == 0)
  {
    // Sleep for a second
    sleep(1);

    // Intentionally SEGFAULT the child process
    int *p = NULL;
    *p = 1;

    exit(0);
  }

  // Wait for the child to exit
  waitpid( child_pid, &status, 0 );

  // See if the child was terminated by a signal
  if( WIFSIGNALED( status ) )
  {
    // Print the signal that the child terminated with
    printf("Child returned with status %d\n", WTERMSIG( status ) );
  }

  return 0;
}
```

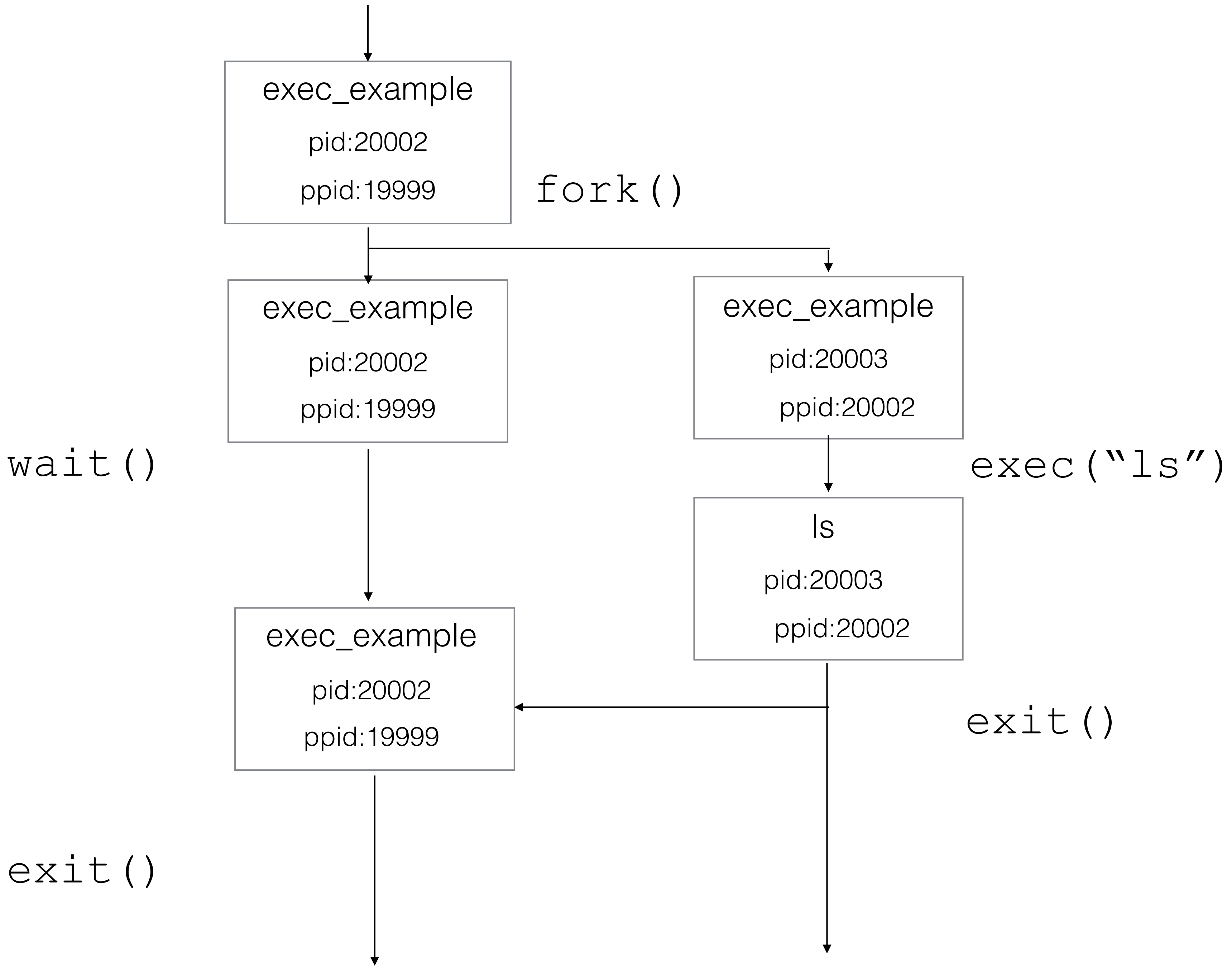
`exec` functions

- When exec is called the new program, specified by exec, completely replaces the running process.
  - text, data, heap and stack are all replaced
  - PID stays the same since it's not a new process

exec code example



```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>

int main(void)
{
  pid_t child_pid = fork();
  int status;

  if (child_pid == 0)
  {
    execl("/bin/ls", "ls", NULL );
    exit(0);
  }

  // Wait for the child to exit
  waitpid( child_pid, &status, 0 );

  return 0;
}
```


exec_example
pid:20002
ppid:19999
fork()
exec_example
pid:20002
ppid:19999
exec_example
pid:20003
ppid:20002
wait()
exec("ls")
ls
pid:20003
ppid:20002
exec_example
pid:20002
ppid:19999
exit()
exit()

Kernel Mode v. User Mode

- A process is executing either in user mode, or in kernel mode. Depending on which privileges, address space a process is executing in, we say that it is either in user space, or kernel space.

- When executing in user mode, a process has normal privileges and can and can't do certain things. When executing in kernel mode, a process has every privilege, and can do anything.

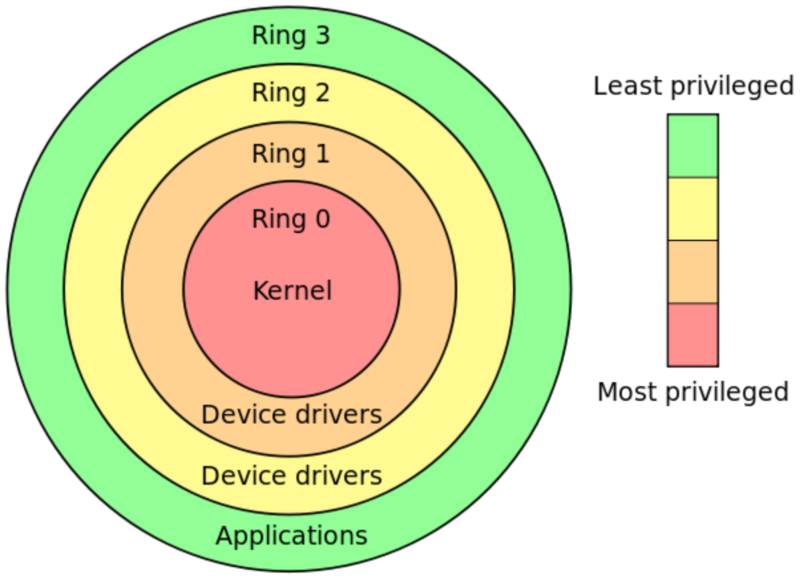
- Processes switch between user space and kernel space using system calls.

Kernel Mode v. User Mode

- These two modes aren't just labels; they're enforced by the CPU hardware.
- If code executing in User mode attempts to do something outside its purview such as accessing a privileged CPU instruction or modifying memory that it has no access to:
    - Trappable exception is thrown. Instead of your entire system crashing, only that particular application crashes.

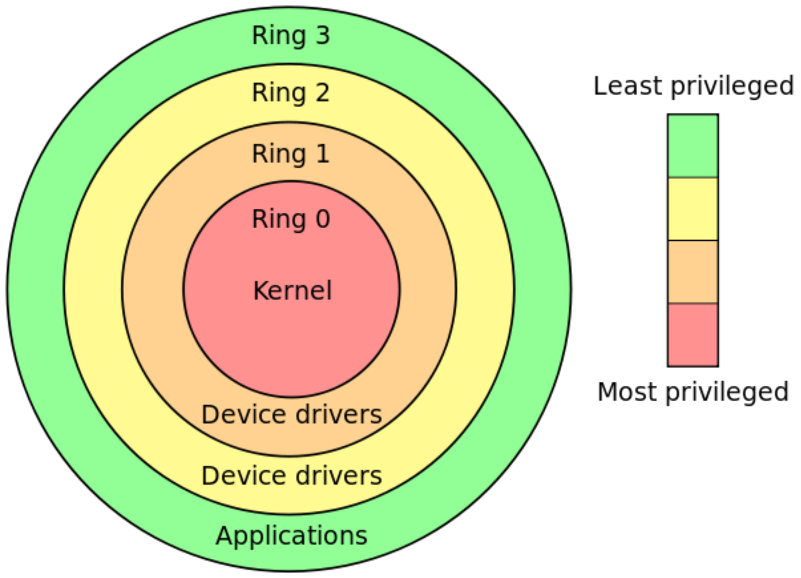
x86 Protection Rings

- Four privilege levels or rings, numbered from 0 to 3, with ring 0 being the most privileged and 3 being the least.
- Rings 1 and 2 weren't used in practice.
  - » VM's changed that



x86 Protection Rings

- Programs that run in Ring 0 can do anything with the system.
- Code that runs in Ring 3 should be able to fail at any time without impact to the rest of the computer system.



CPU Rings and Privilege

- CPU privilege level has nothing to do with operating system users.
- Whether you're root, Administrator, guest, or a regular user, it does not matter.
- All user code runs in ring 3 and all kernel code runs in ring 0, regardless of the OS user on whose behalf the code operates.

CPU Rings and Privilege

- Due to restricted access to memory and I/O ports, user mode can do almost nothing to the outside world without calling on the kernel.
  - It can't open files, send network packets, print to the screen, or allocate memory.
  - User processes run in a severely limited sandbox set up by ring zero.

System Calls

- How do our programs utilize the resources controlled by the OS or communicate with other process?
- Because user mode software can not access hardware devices directly, they must notify the operating system in order to complete system tasks. This includes displaying text, obtaining input from user, printing a document, etc.

System Calls

- Instead of directly calling a section of code the system call instruction issues an interrupt.

- By not allowing the application to execute code freely the operating system can verify that the application has appropriate privileges to call the function.

  - Only system calls enter the kernel. Procedure calls do not.

Monolithic Systems (1)

Basic structure of OS

1. A main program that invokes the requested service procedure.
2. A set of service procedures that carry out the system calls.
3. A set of utility procedures that help the service procedures.

Microkernel

- Only basic functionality is included in the kernel
  - What is basic? Only code that must run in supervisor mode because it must use privileged resources such as protected instructions
- Everything else runs in user space.

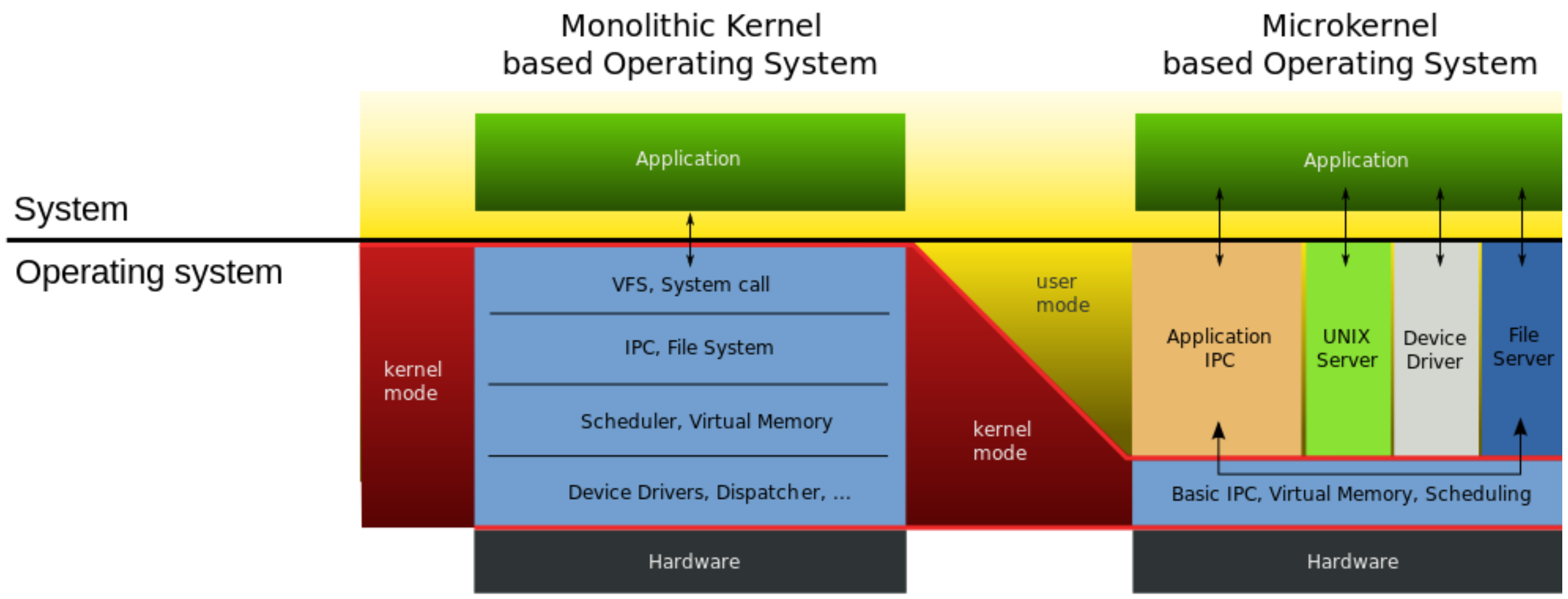
Microkernel

- Put the mechanism in the kernel and the policy in user-mode
  - For example: job scheduling
    - Kernel runs the highest priority process
    - User-mode job scheduler decides priorities

Microkernel

- Theoretically more robust since limiting the amount of code that runs in protected mode limits the number of catastrophic crashes
- Easier to inspect for flaws since a smaller portion of code exists
- May run slower since there are more interrupts from user space to kernel
- Example: Minix

Micro v. Monolithic



"OS-structure2" by Golftheman - http://en.wikipedia.org/wiki/Image:OS-structure.svg. Licensed under Public domain via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:OS-structure2.svg#mediaviewer/File:OS-structure2.svg

Scheduling

- I/O interrupt
  - Hardware clock provides 50 or 60 hz interrupts
- Scheduling algorithms can be divided into 2 categories based on how they deal with clock interrupts
  - Non-preemptive (cooperative)
  - Preemptive

Scheduling Algorithm Goals

- Batch Systems
  - Throughput - maximize jobs per hour
  - Turnaround time - minimize time between submission and termination
  - CPU utilization - you paid for that screaming CPU, use it

Scheduling Algorithm Goals

- Interactive systems
    - Response time
    - Proportionality - meets users expected performance

Scheduling Algorithm Goals

- Real-time systems
  - Meet deadlines
  - Predictability - avoid quality degradation in multimedia systems

Scheduling Metrics

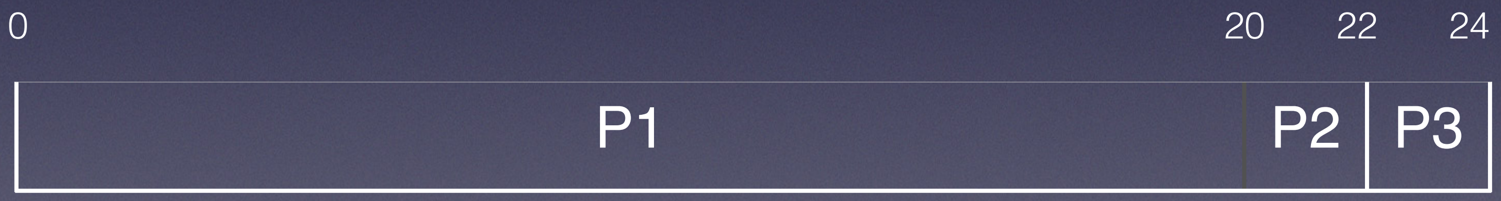
- Throughput - number jobs per unit of time
- Turnaround time - average of time when jobs are submitted to when they complete
- Response time - average of time when jobs are submitted to jobs start running
- Wait time - time jobs spend in the wait queue

FCFS Scheduling

- First come, first served algorithm (FCFS).
- Easy to implement
- Well understood by anyone
- The fairest algorithm. No process is favored over another.

FCFS

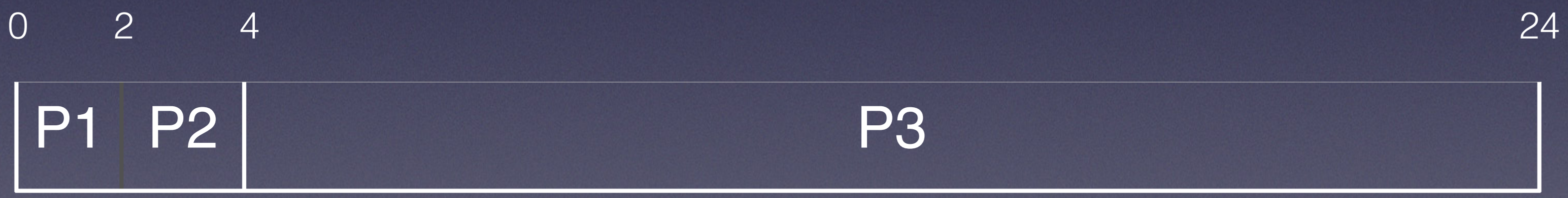
Process ID	Arrival Time	Runtime
1	0	20
2	2	2
3	2	2



Average Waiting Time: (0 + 18 + 20) / 3 = 12.67

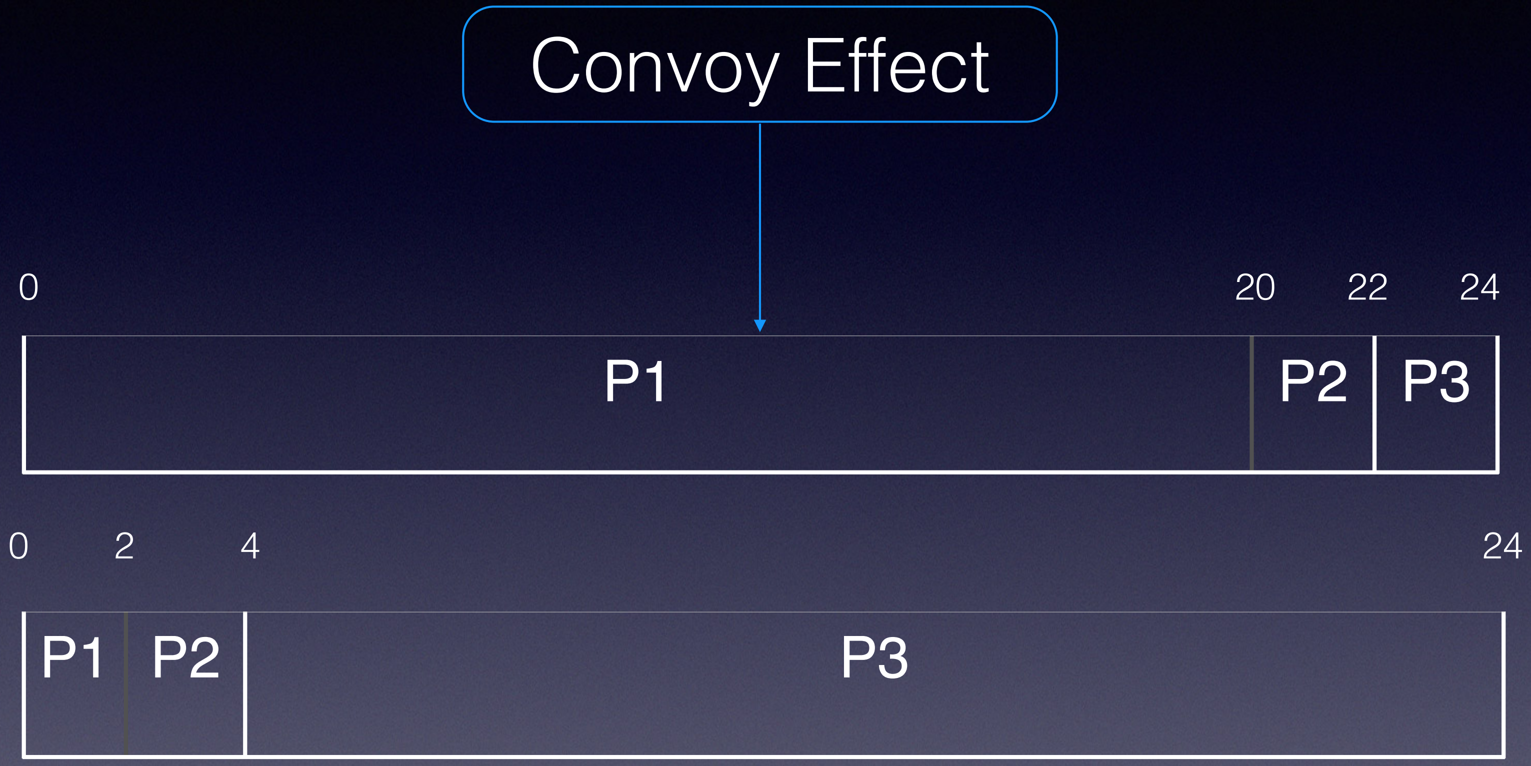
FCFS

Process ID	Arrival Time	Runtime
1	0	2
2	2	2
3	2	20



Average Waiting Time: (0 + 0 + 2) / 3 = 0.67

FCFS



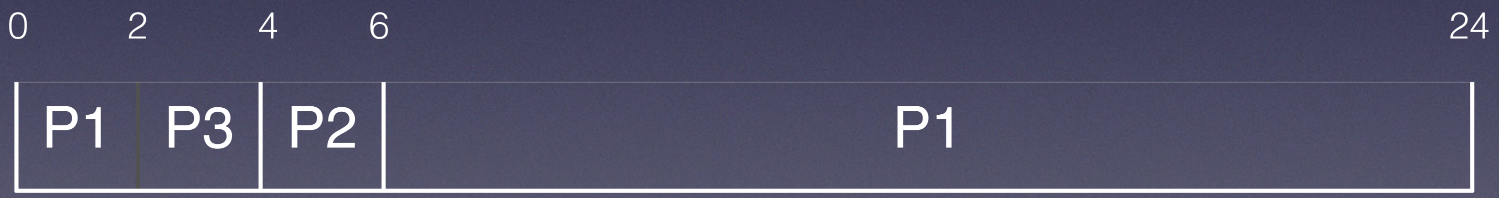
Very Dependent On Job Arrival Time

Preemption

Without Preemption	With Preemption
FCFS	Round Robin
SRTF	Shortest Remaining Time Next
Priority	Priority With Preemption

FCFS w/ Priority (Round Robin)

Process ID	Arrival Time	Runtime	Priority
1	0	20	4
2	2	2	2
3	2	2	1



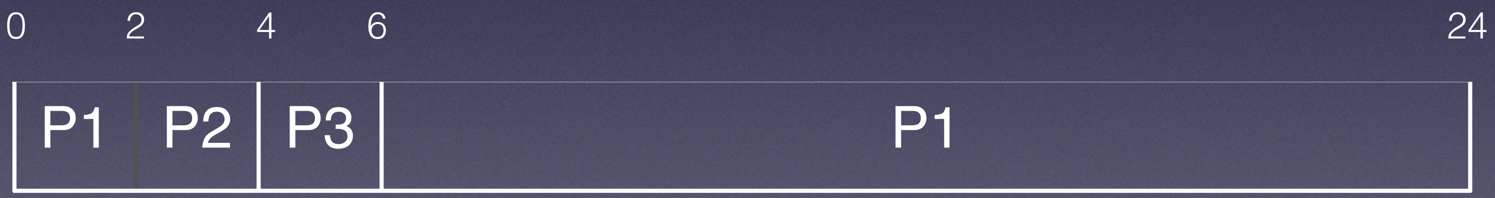
Average Waiting Time: (4 + 2 + 0) / 3 = 2

Time Quantum

- Choosing length is a problem
  - Context switching is expensive
  - Still need responsiveness

SJN with Preemption

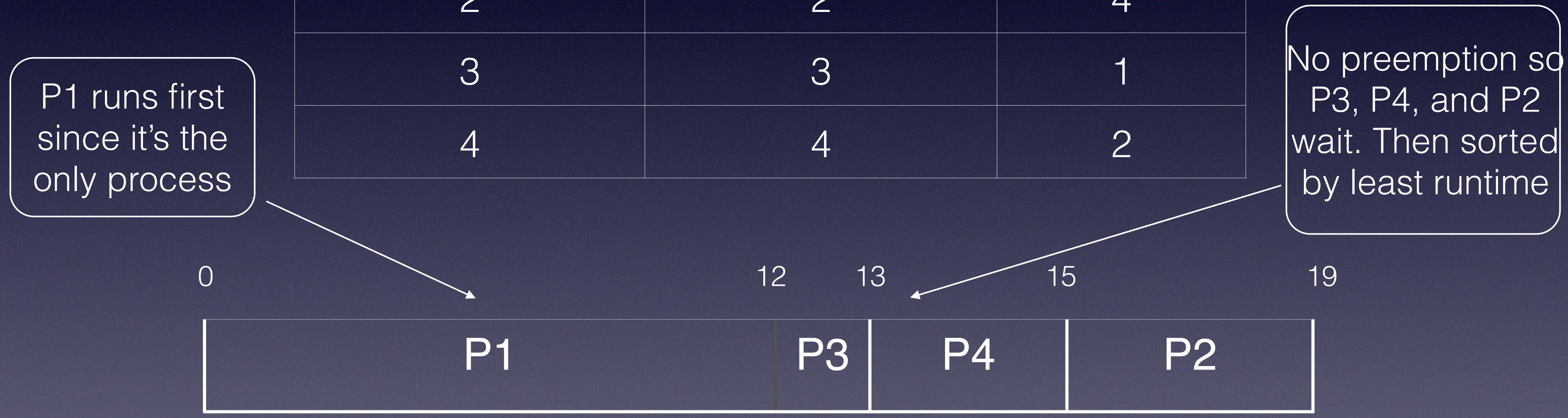
Process ID	Arrival Time	Runtime
1	0	20
2	2	2
3	2	2



Average Waiting Time: (4 + 0 + 2) / 3 = 2

SJN

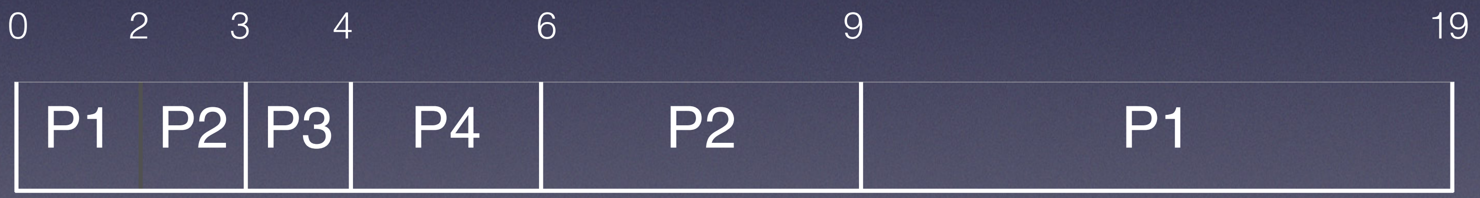
Process ID	Arrival Time	Runtime
1	0	12
2	2	4
3	3	1
4	4	2


P1 runs first since it's the only process
No preemption so P3, P4, and P2 wait. Then sorted by least runtime
0
12
13
15
19
P1
P3
P4
P2

Average Waiting Time: (0+13+9+9)/4 = 7.75

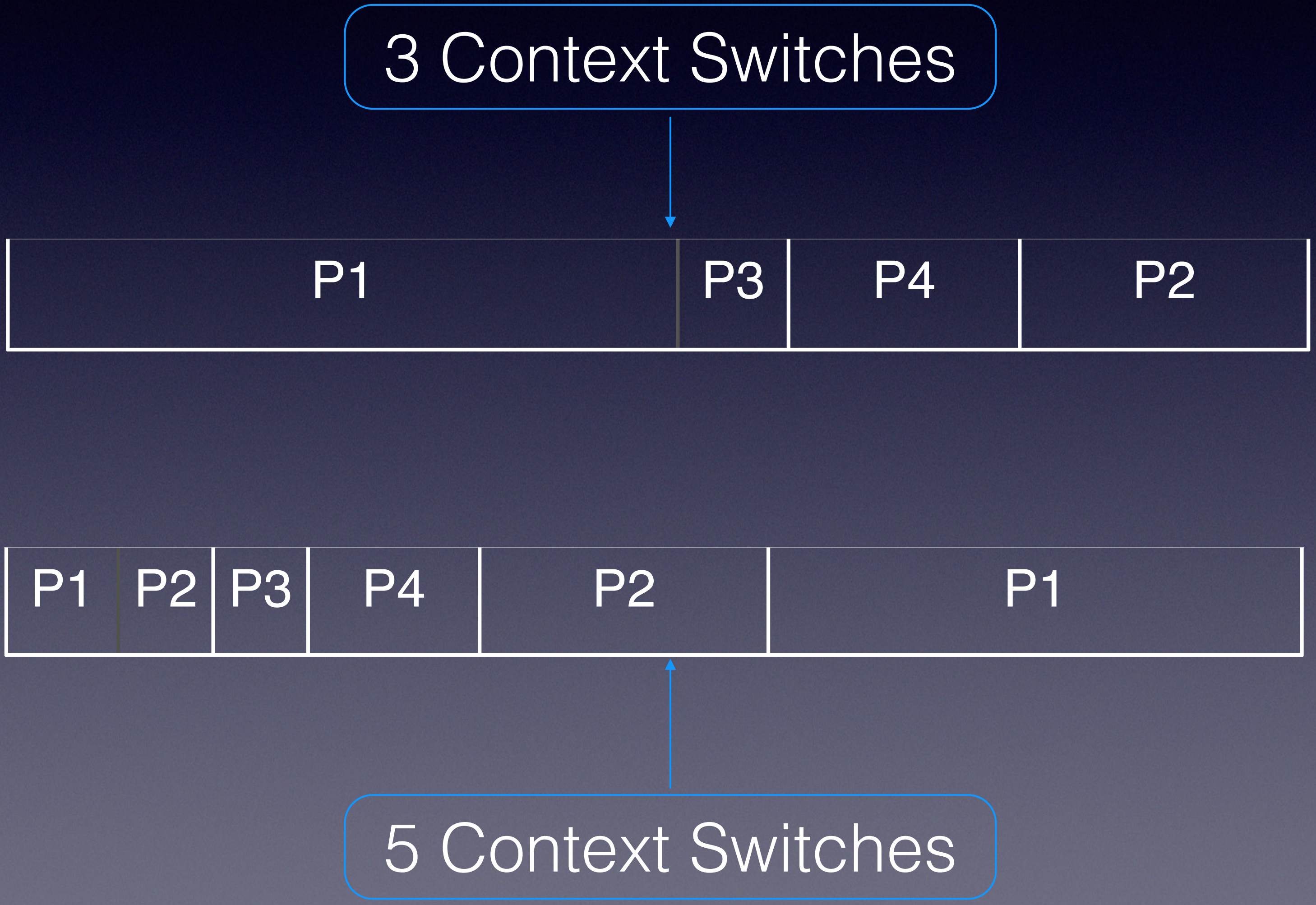
SJN With Preemption

Process ID	Arrival Time	Runtime
1	0	12
2	2	4
3	3	1
4	4	2



Average Waiting Time: (7+3+0+0)/4 = 2.5

SRTF Without and With Preemption


3 Context Switches
P1
P3
P4
P2
P1
P2
P3
P4
P2
P1
5 Context Switches