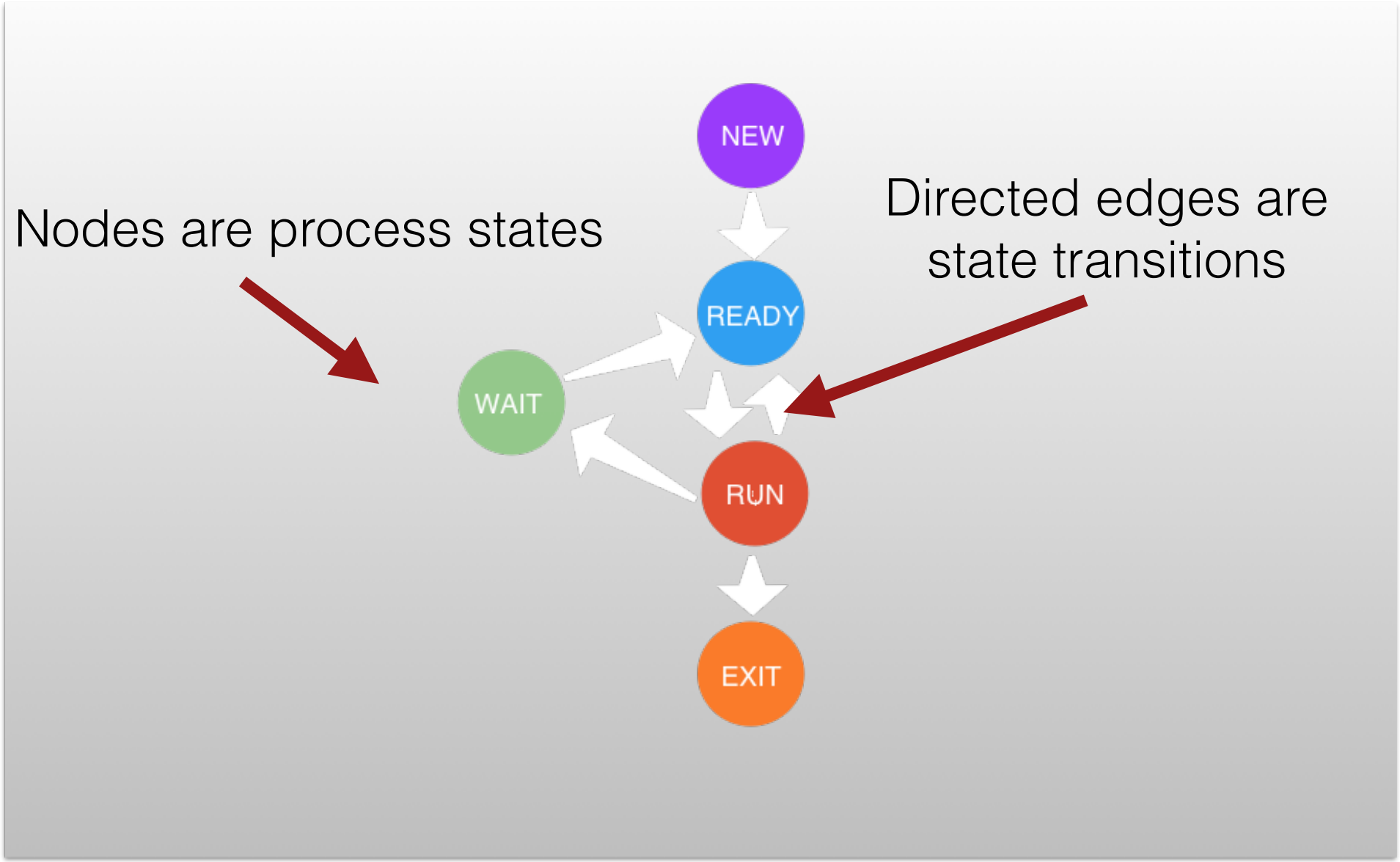
CSE 3320

Chapter 8: Process Management, Part 1

Trevor Bakker

The University of Texas at Arlington

Process State Diagram



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() man page

FORK(2) Linux Programmer's Manual FORK(2)

NAME

fork - create a child process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION

fork() creates a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited.

fork() return values

RETURN VALUE

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.

ERRORS

EAGAIN **fork**() cannot allocate sufficient memory to copy the parent's page tables and allocate a task structure for the child.

EAGAIN It was not possible to create a new process because the caller's **RLIMIT_NPROC** resource limit was encountered. To exceed this limit, the process must have either the **CAP_SYS_ADMIN** or the **CAP_SYS_RESOURCE** capability.

ENOMEM **fork**() failed to allocate the necessary kernel structures because memory is tight.

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

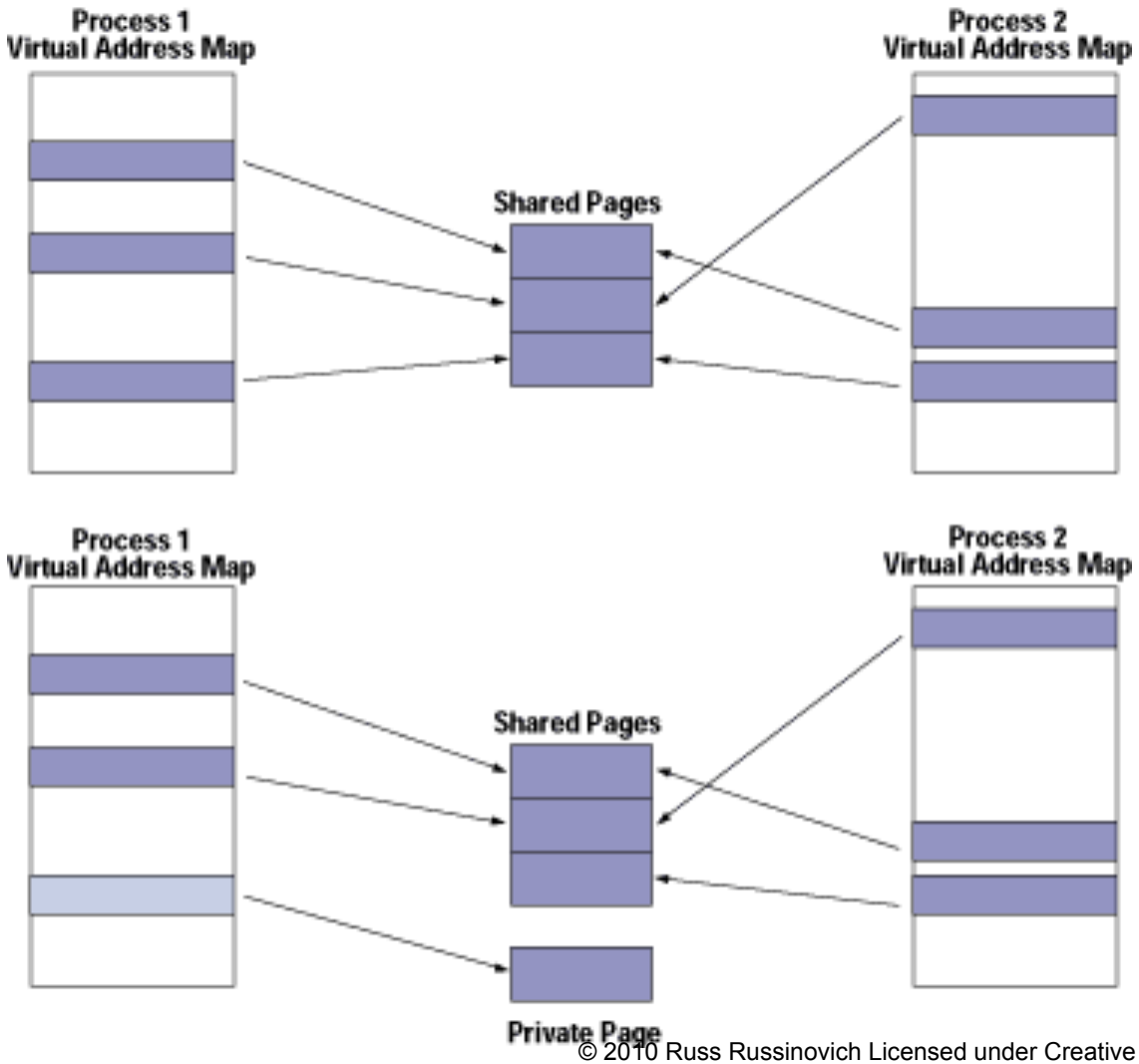
Process Creation

- 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

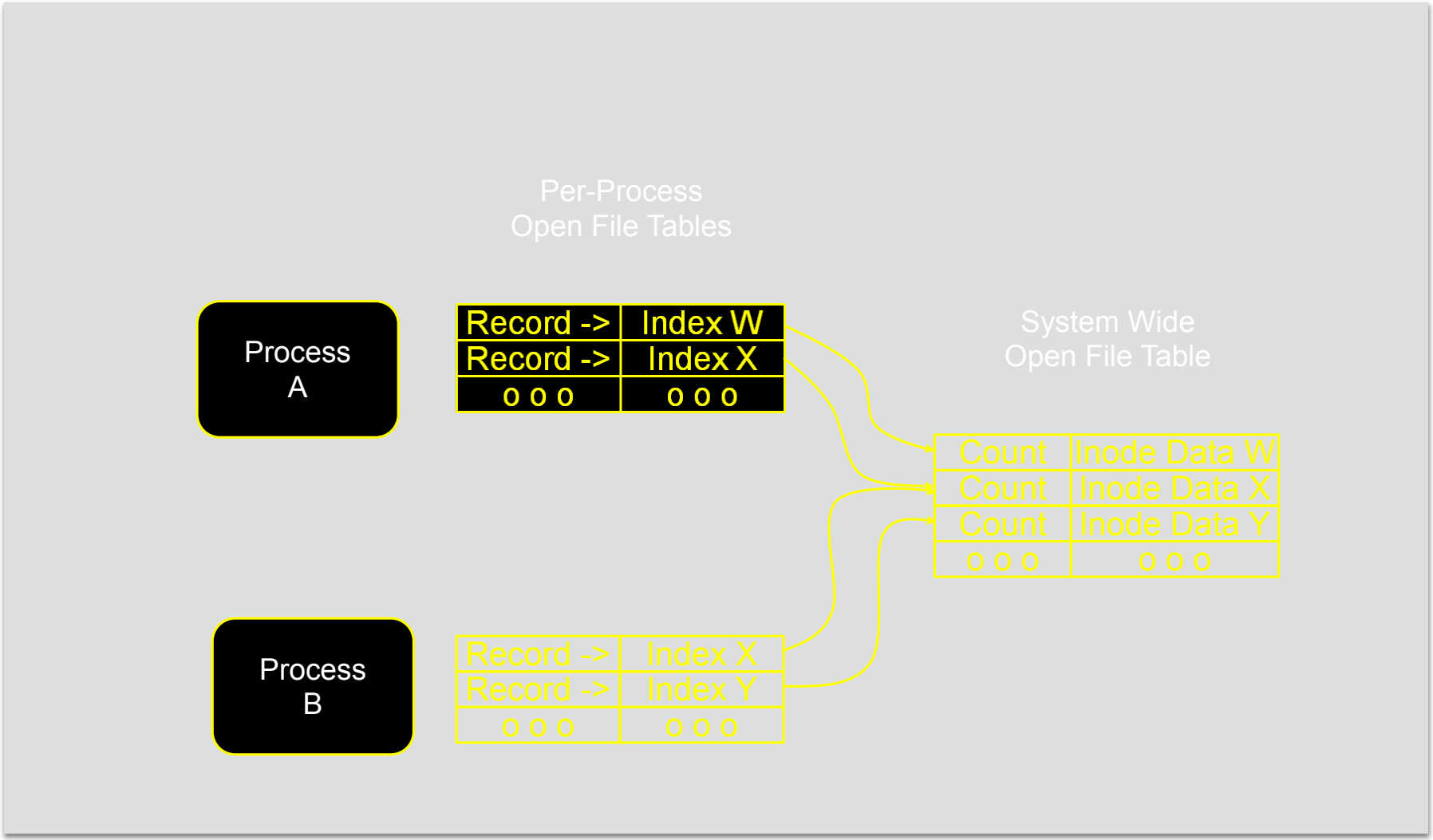
Copy-On-Write

- 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

Copy-On-Write



File Control Blocks



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
- resource limits

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

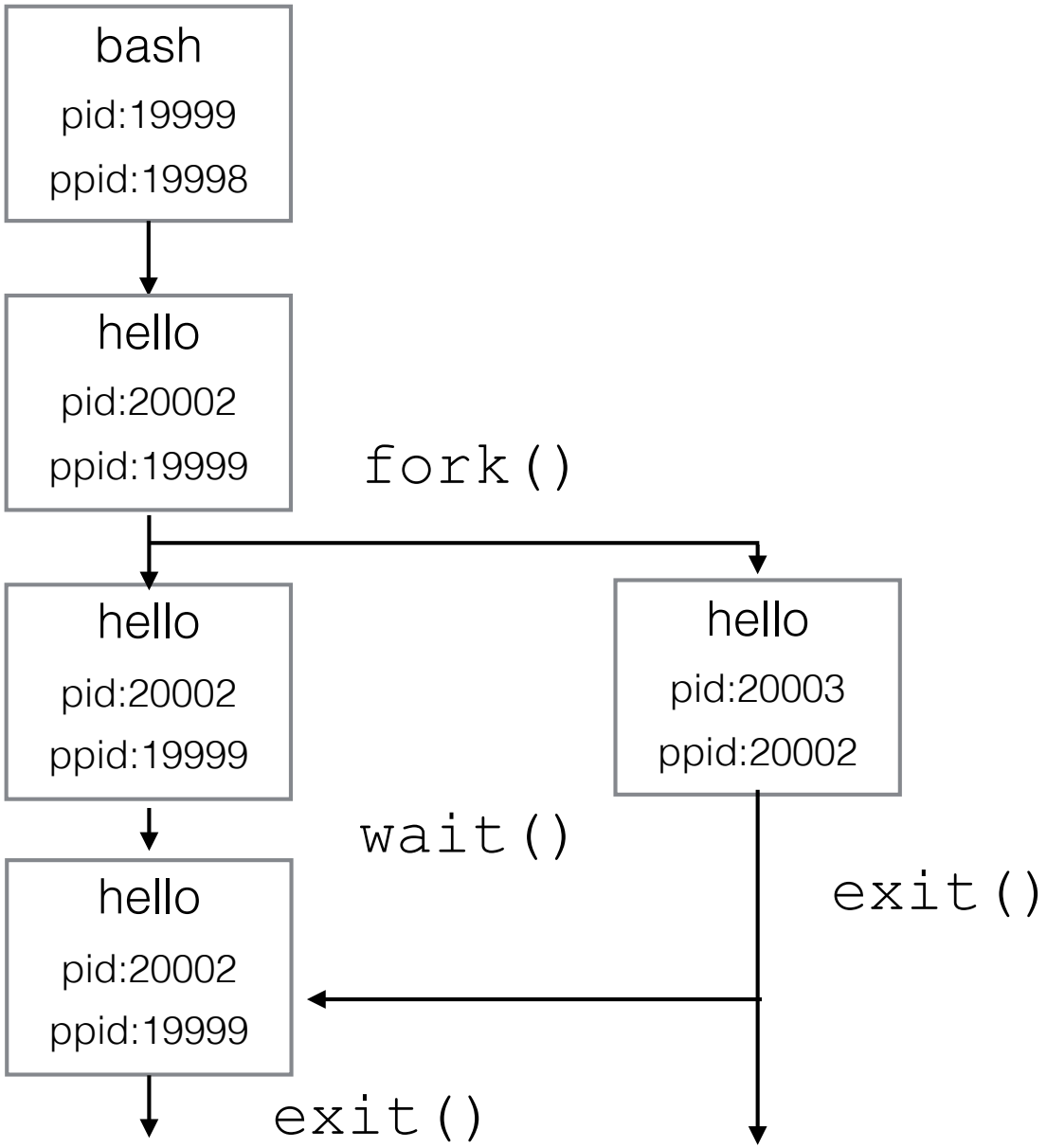
fork() code example

```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;
}
```

Hello World


bash
pid:19999
ppid:19998
hello
pid:20002
ppid:19999
fork()
hello
pid:20002
ppid:19999
hello
pid:20003
ppid:20002
wait()
exit()
hello
pid:20002
ppid:19999
exit()

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()`

`fork()` failures

- Too many processes already.
  - Usually a sign something else has gone wrong
  - `cat /proc/sys/kernel/pid_max`
- Too many processes for the current user
  - `ulimit -a`

Terminating a Process

- Three normal ways

1. Executing a return from the `main` function
2. Calling the `exit` function. C library function.
3. Calling the `_exit` function. System call.
    1. You should use _exit to kill the child program when the exec fails. Otherwise the child process may interfere with the parent process' external data by calling its signal handlers, or flushing buffers.
    2. You should also use _exit in any child process that does not do an exec, but those are rare.

Terminating a Process

- Two abnormal ways

1. Calling `abort`. Generates a SIGABORT signal.

2. The process received a signal

What if the parent terminates first?

- `exit` and `_exit` return the termination status of child processes to the parent. Who gets the status if the parent has already terminated?

- The `init` process becomes the parent of any process whose parent terminates.

  - The orphaned process is inherited by the `init` process

  - When a process is terminated the kernel iterates through all the active processes to see if the terminating process is the parent of any remaining processes. If so, it inherits that process

Zombie Process

- A process that has completed execution but still has an entry in the process table.
  - The entry is still needed to allow the parent process to read its child's exit status
- A child process always becomes a zombie before being removed from the resource table.
  - Normally, zombies are immediately waited on by their parent and then reaped by the system

`wait()` and `waitpid()`

- When a process terminates the parent is notified by the operating system sending a SIGCHLD signal.
  - Default handling is to ignore the signal
- Child termination is asynchronous
- POSIX provides two system calls `wait` and `waitpid`

wait()

```
WAIT(2)                    Linux Programmer's Manual                    WAIT(2)

NAME
       wait, waitpid - wait for process to change state

SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *status);
       pid_t waitpid(pid_t pid, int *status, int options);
       int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- `wait` suspends the parents operation until the next child process terminates.
- If there is a child process already terminated and waiting for reaping `wait` will return immediately
- PID of the terminated child process is returned on success. -1 if failure.

`wait()` status parameter

WIFEXITED(status**)**
returns true if the child terminated normally, that is, by calling **exit**(3) or **_exit**(2), or by returning from main().

WEXITSTATUS(status**)**
returns the exit status of the child. This consists of the least significant 16–8 bits of the status argument that the child specified in a call to **exit**() or **_exit**() or as the argument for a return statement in main(). This macro should only be employed if **WIFEXITED** returned true.

WIFSIGNALED(status**)**
returns true if the child process was terminated by a signal.

WTERMSIG(status**)**
returns the number of the signal that caused the child process to terminate. This macro should only be employed if **WIFSIGNALED** returned true.

`wait()` status parameter

WCOREDUMP(status**)**
returns true if the child produced a core dump. This macro should only be employed if **WIFSIGNALED** returned true. This macro is not specified in POSIX.1-2001 and is not available on some Unix implementations (e.g., AIX, SunOS). Only use this enclosed in #ifdef WCOREDUMP ... #endif.

WIFSTOPPED(status**)**
returns true if the child process was stopped by delivery of a signal; this is only possible if the call was done using **WUNTRACED** or when the child is being traced (see **ptrace**(2)).

WSTOPSIG(status**)**
returns the number of the signal which caused the child to stop. This macro should only be employed if **WIFSTOPPED** returned true.

WIFCONTINUED(status**)**
(Since Linux 2.6.10) returns true if the child process was resumed by delivery of **SIGCONT**.

waitpid()

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

- Allows the parent process to wait on a specific child process.

`wait()` 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;
}
```

Process Creation

`fork()` creates an exact copy of a process. How do we create a unique process?

By combining `fork` with an `exec` function

`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 family

```
int execl(char const *path, char const *arg0, ...);
int execle(char const *path, char const *arg0, ..., char const *envp[]);
int execlp(char const *file, char const *arg0, ...);
int execv(char const *path, char const *argv[]);
int execve(char const *path, char const *argv[], char const *envp[]);
int execvp(char const *file, char const *argv[]);
```

Meaning of the letters after exec:
l – A list of command-line arguments are passed to the function.
e – An array of pointers to environment variables is passed to the new process image.
p – The PATH environment variable is used to find the file named in the path argument.
v – Command-line arguments are passed to the function as an array of pointers.

exec code example

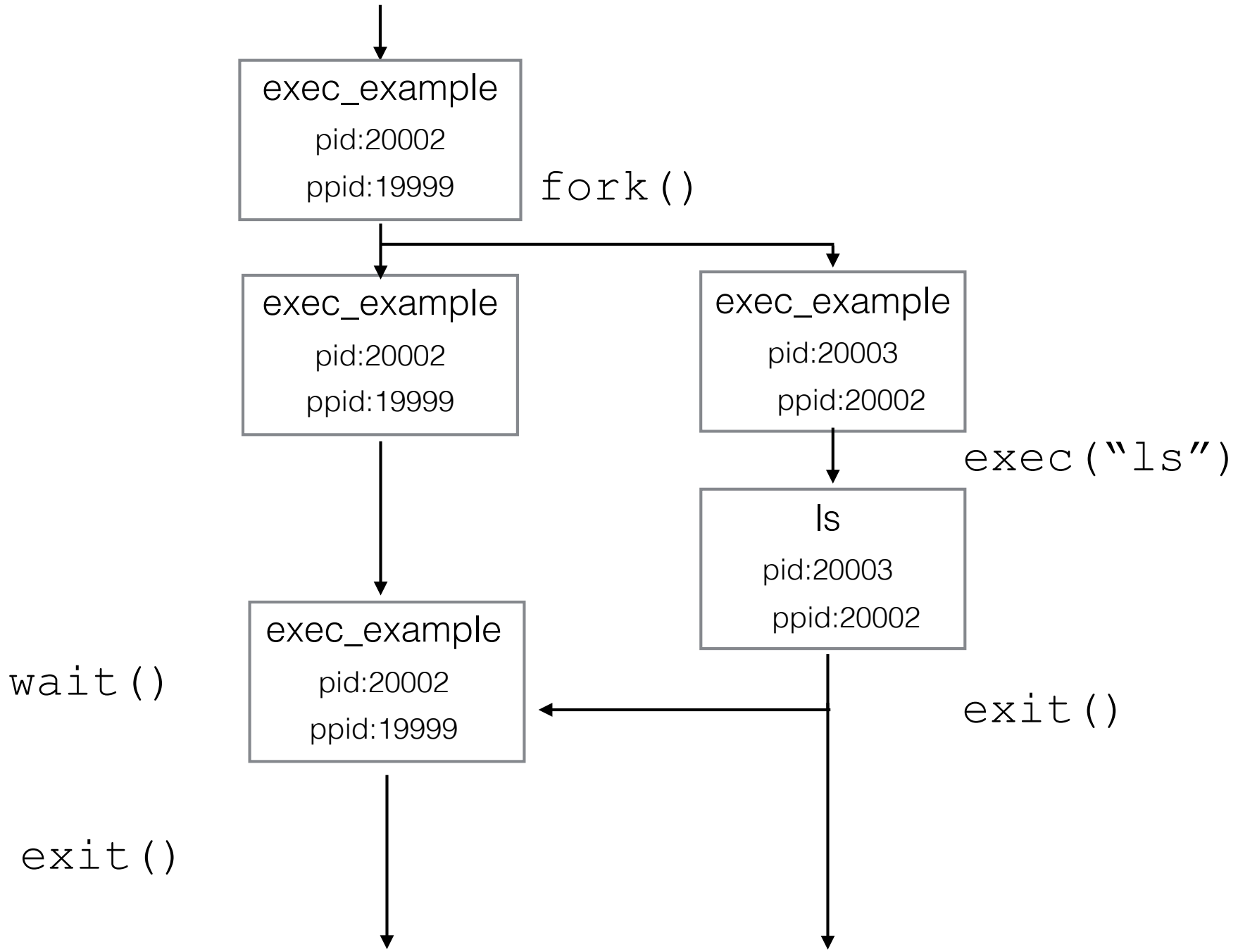
```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>

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

Pseudo parallelism

- In a single CPU system the CPU is only running one process at a time
  - Process are switched over the course of time giving the illusion of of parallelism.
- Contrast with a multiprocessor system in which multiple CPU share the same memory and execute processes in parallelism.

Process Model

- All running software on the computer is organized into a number of sequential processes.
- Conceptually each process has its own virtual CPU.
- In reality, the CPU is shared by the processes.
  - Multiprogramming: The rapid switching back and forth

The Process Model (2)

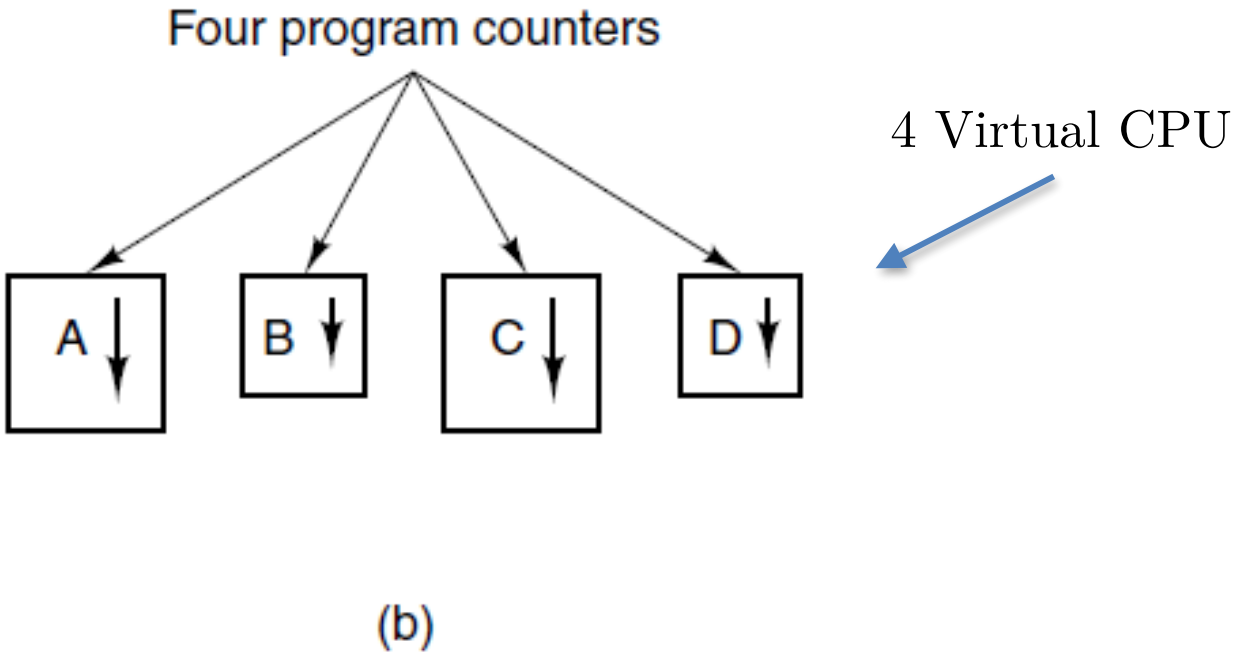


Figure 2-1. (b) Conceptual model of four independent, sequential processes.

The Process Model (1)

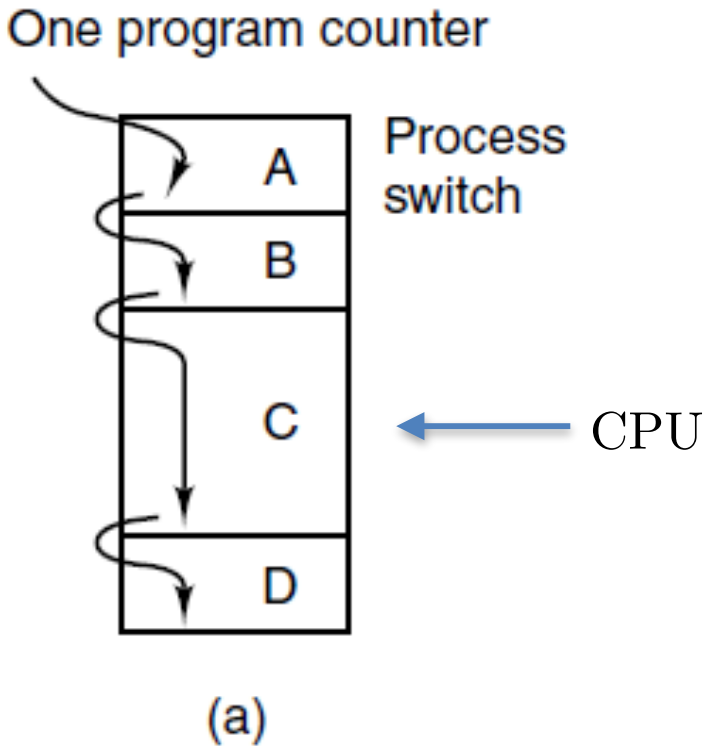


Figure 2-1. (a) Multiprogramming of four programs.

The Process Model (3)

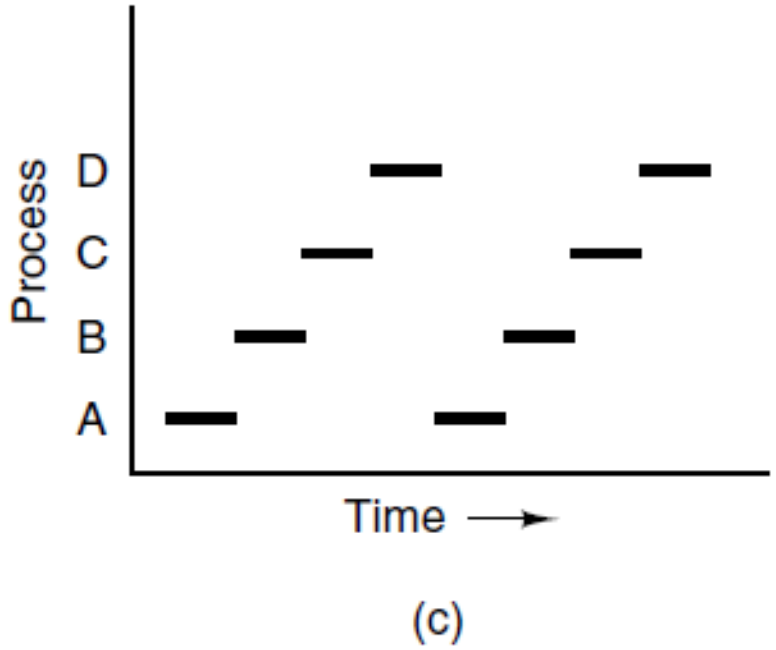
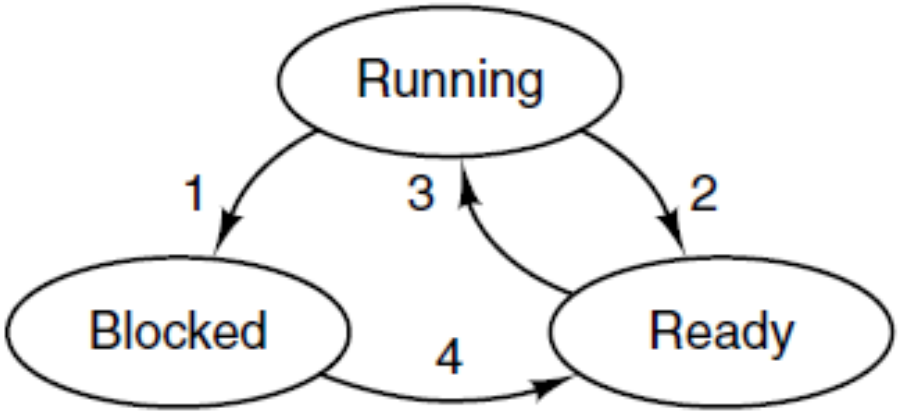


Figure 2-1. (c) Only one program is active at once.

Process States (2)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

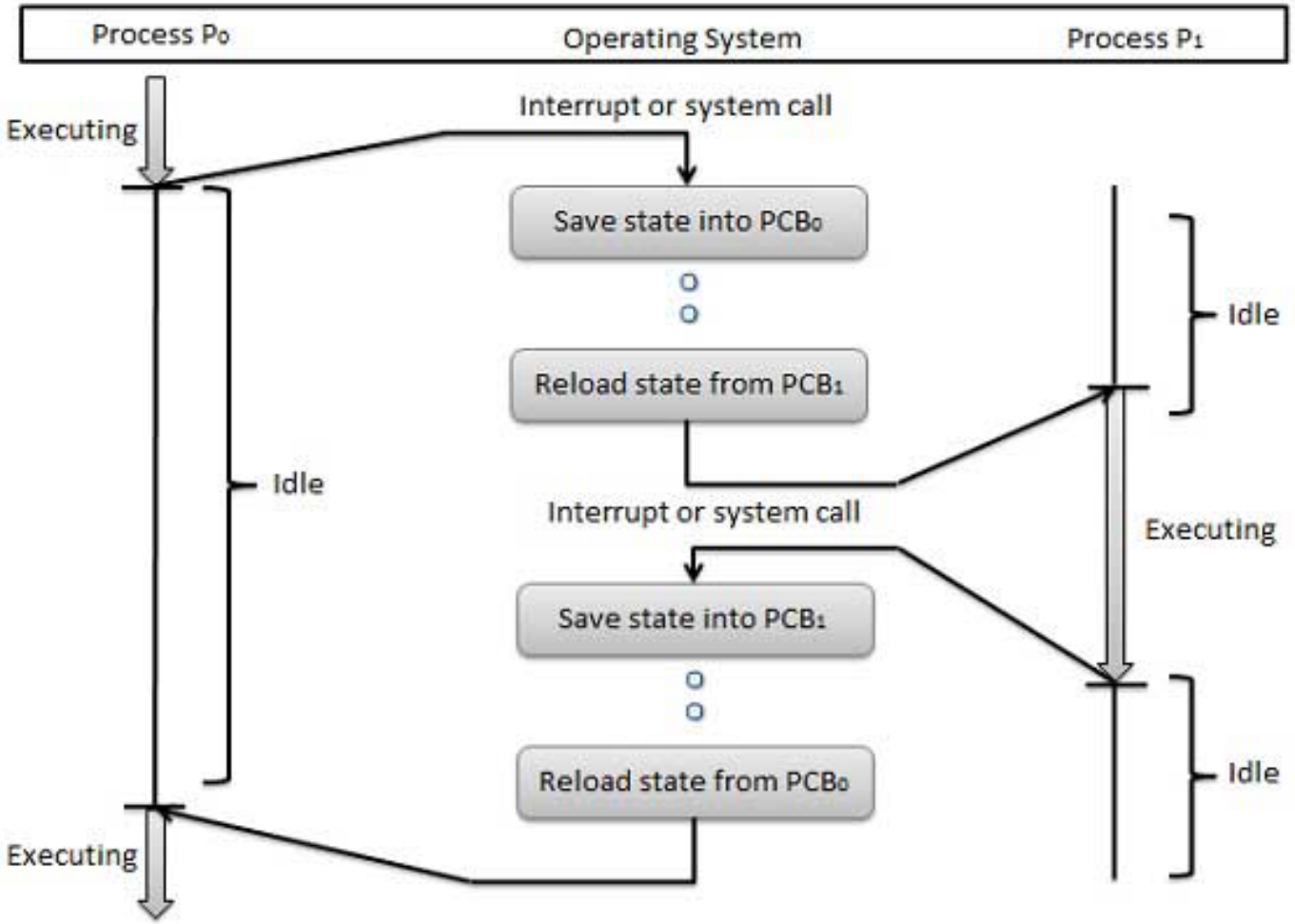
Context Switch

- A context switch (also sometimes referred to as a process switch or a task switch) is the switching of the CPU from one process or thread to another.
- A context is the contents of a CPU's registers and program counter at any point in time.

Context Switch in more detail

1. Suspend the progression of one process and storing the CPU's state (i.e., the context) for that process somewhere in memory.
2. Retrieve the context of the next process from memory and restoring it in the CPU's registers
3. Return to the location indicated by the program counter (i.e., returning to the line of code at which the process was interrupted) in order to resume the process.

Context Switch



Process States (3)

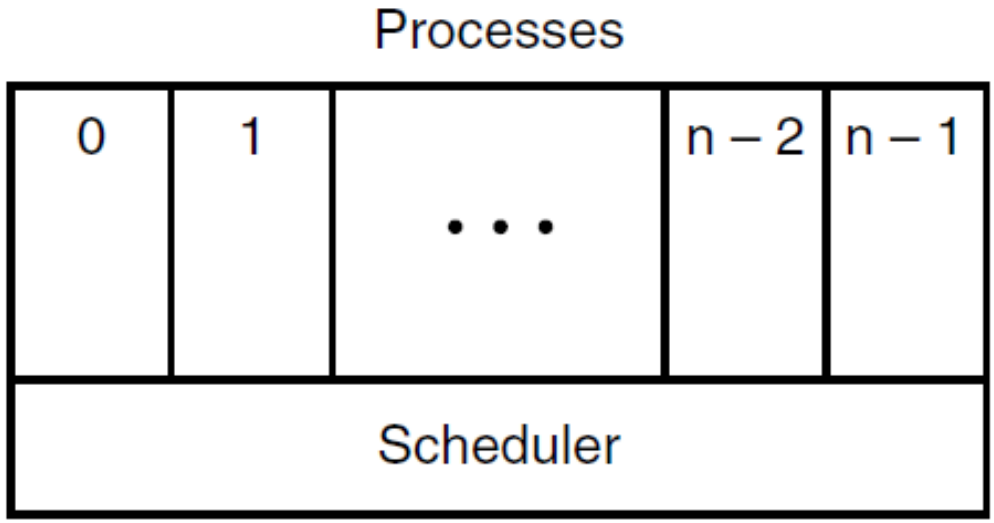
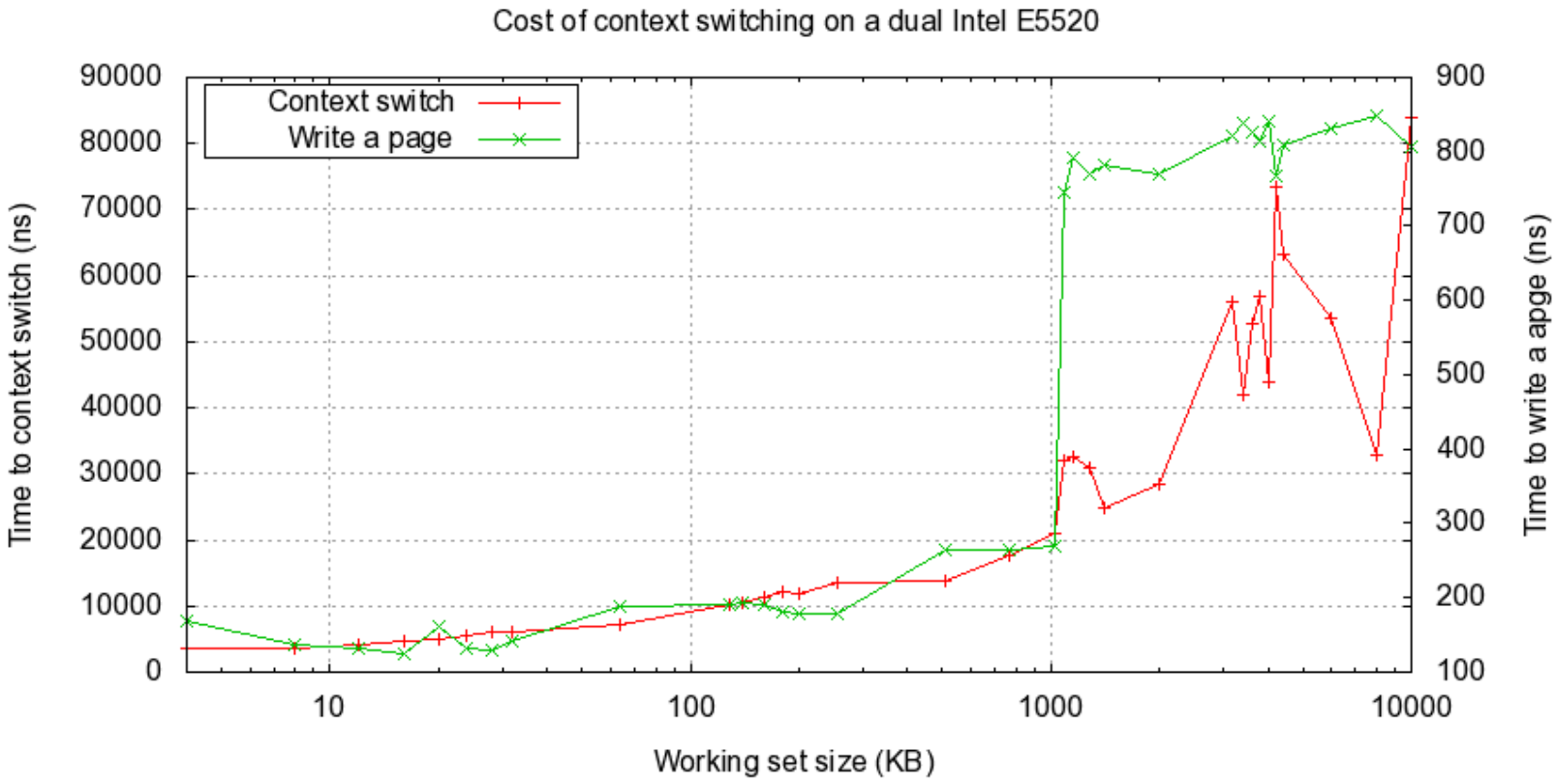


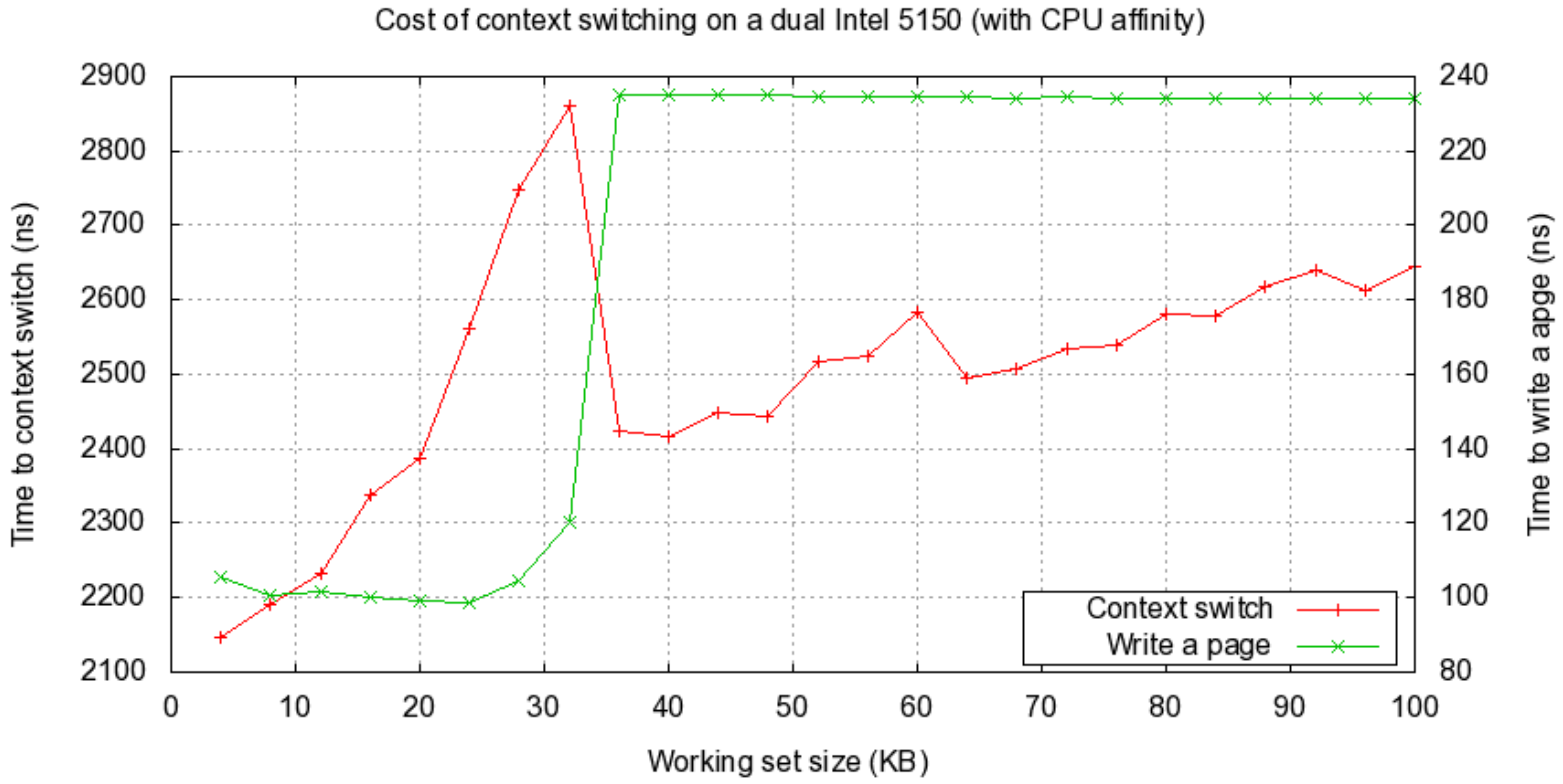
Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Cost of Context Switch



30µs of CPU overhead is a good rule of thumb for worst case context switch

Cost of Context Switch



CPU affinity can help context switching cost, but linux is pretty poor at scheduling CPU affinity.

Process States (1)

Three states a process may be in:

1.Running (actually using the CPU at that instant).

2.Ready (runnable; temporarily stopped to let another process run).

3.Blocked (unable to run until some external event happens).

CPU Utilization

- Multiprogramming improves CPU utilization
  - If the average process computes 20% of the time, then 5 processes are needed to fully utilize the CPU.
    - Unrealistic as it assumed that all 5 process won't be waiting for I/O at the same time.

CPU Utilization

» Better model is a probabilistic view.

» Suppose a process ends a fraction p of its time waiting for I/O

» With n processes in memory, the probability that all n processes are waiting for I/O is p^n

$$\text{CPU utilization} = 1 - p^n$$

Modeling Multiprogramming

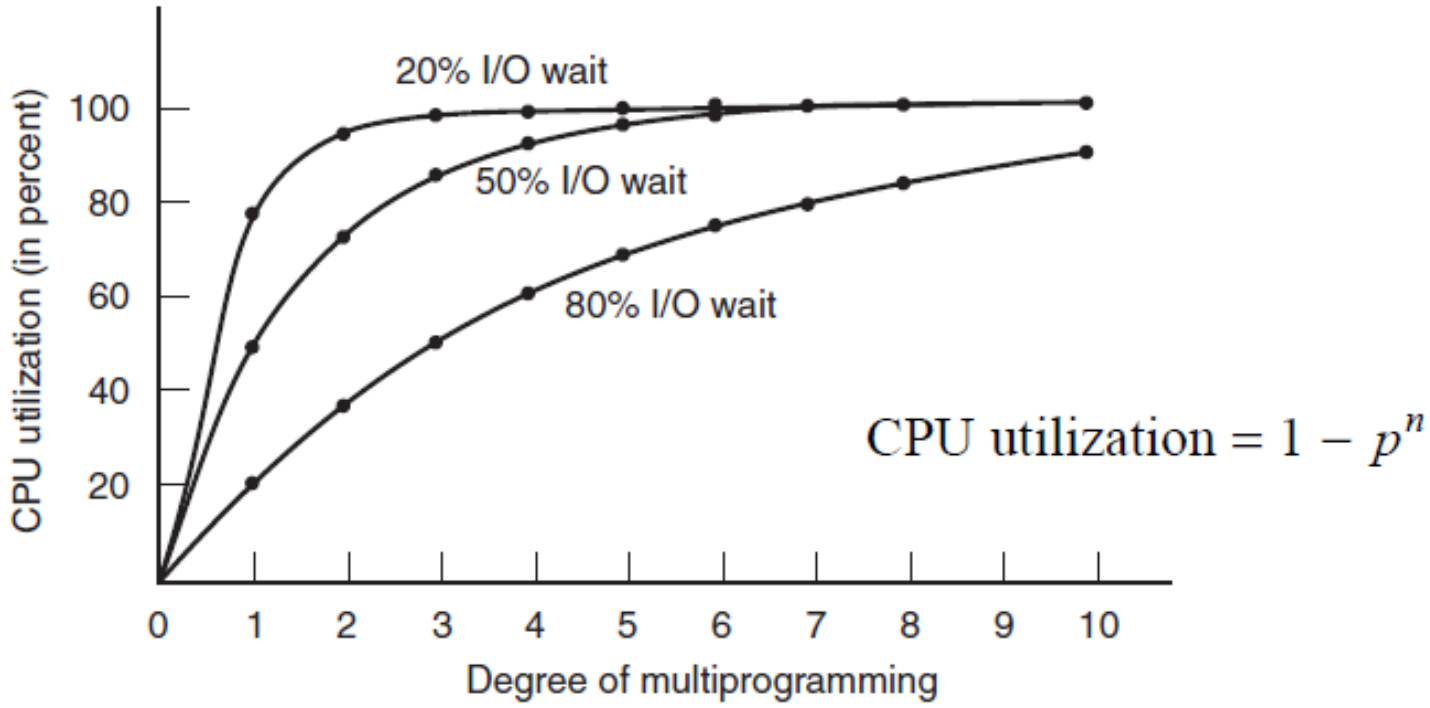


Figure 2-6. CPU utilization as a function of the number of processes in memory.

CPU Utilization

Example: Given 8GB of RAM, the OS taking 2GB of RAM and each user program taking 2GB we can run 3 user programs at once. With an 80% average I/O wait the CPU utilization is:

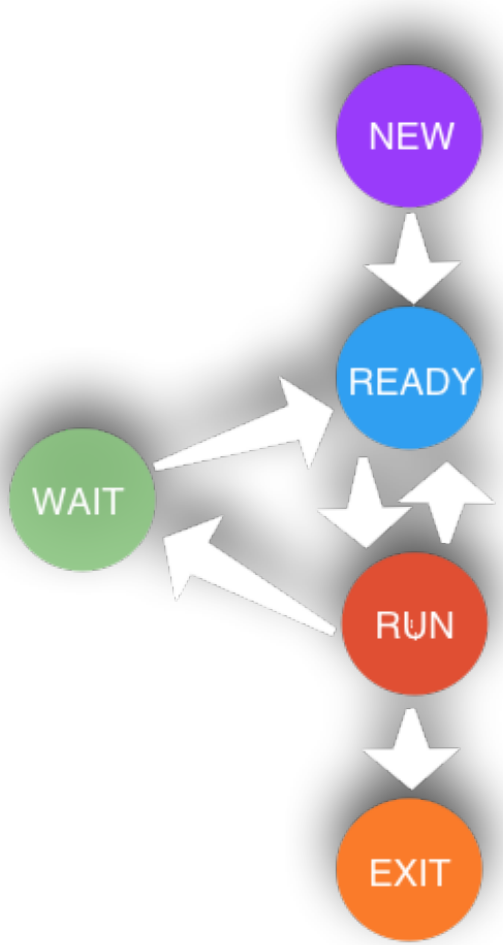
$$1 - 0.8^3 = 48.8\%$$

CPU Utilization

Example continued: Adding an additional 8 GB of memory would allow 7 processes at once and raise the CPU utilization:

$$1 - 0.8^7 = 79\%$$

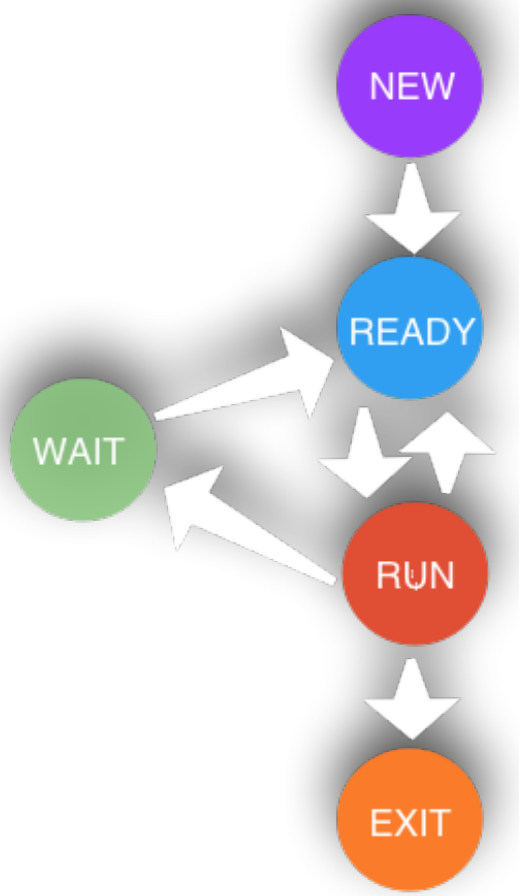
Process State Diagram



Transitioning to Run

How do processes transition from Ready to Run?

What decides when it should run in relation to other processes that are ready?



Scheduling

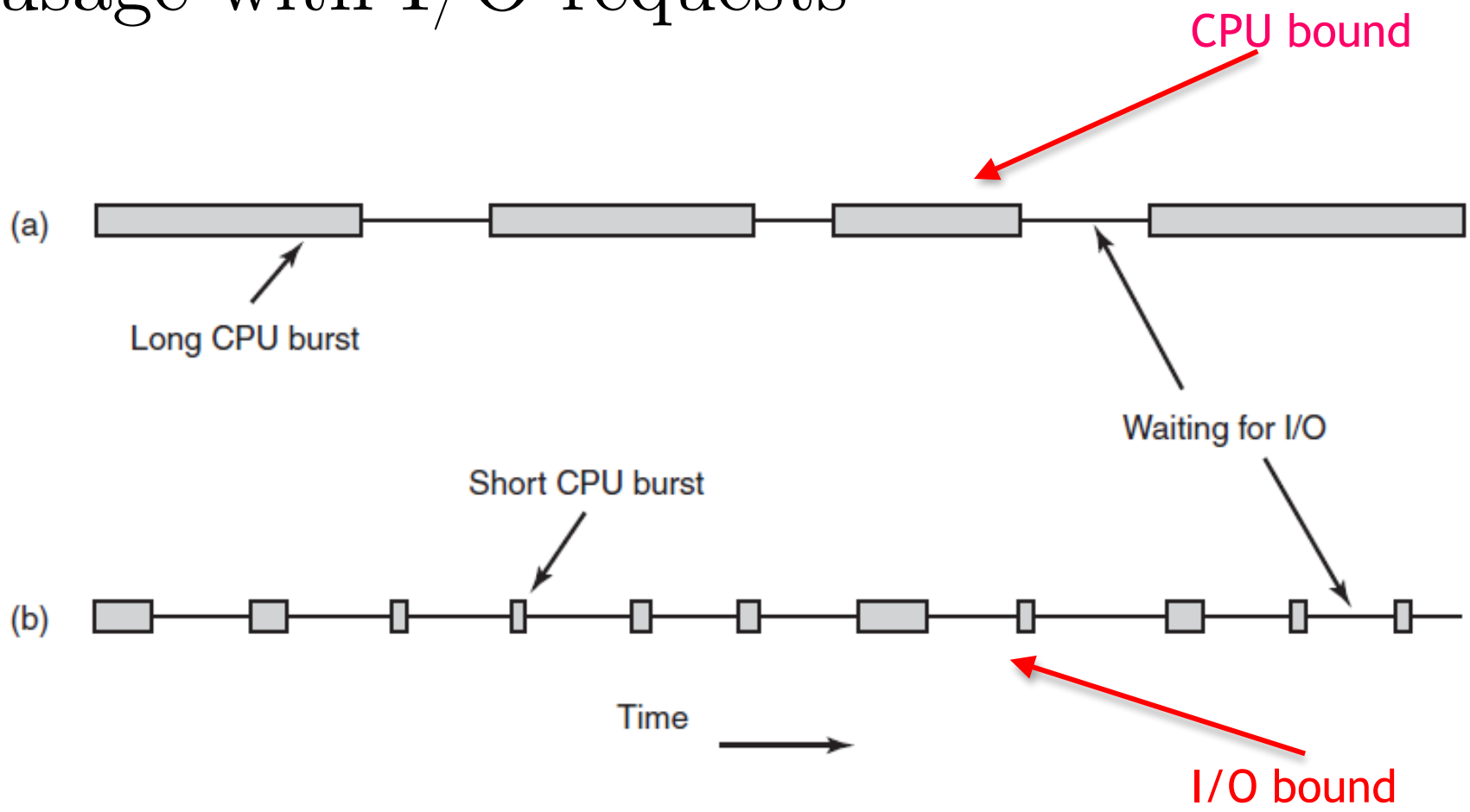
- Cooperative Multitasking
- Preemptive Multitasking

Scheduling

- Multiple processes or threads competing for the CPU
  - Choice needs to be made of which to run next
    - Scheduler - part of the OS that makes that decision
    - Scheduling algorithm

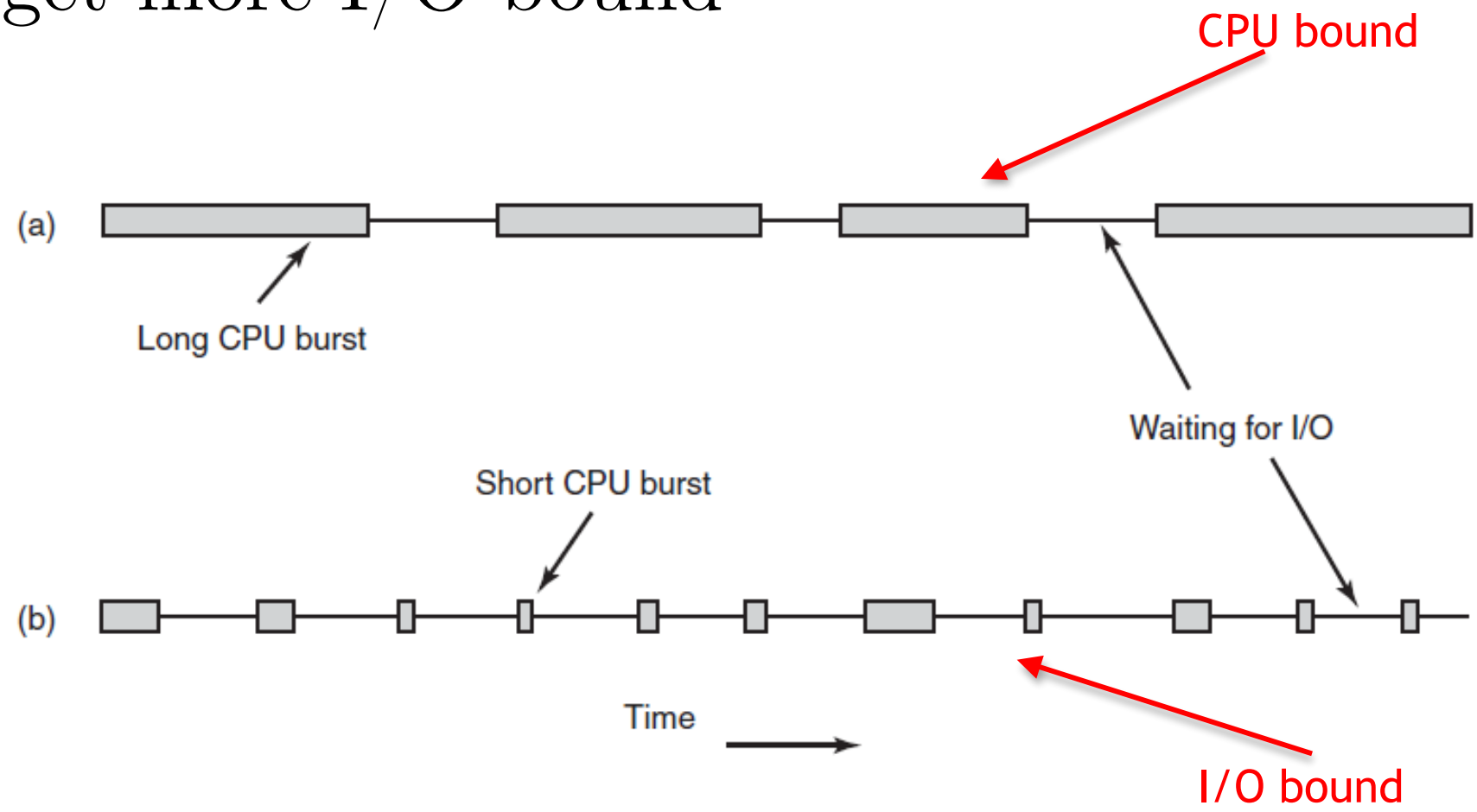
Scheduling

- Nearly all processes alternate burst of CPU usage with I/O requests



Scheduling

- As processors get faster, processes tend to get more I/O bound



Scheduling

- Key issue: when to make the scheduling decision

  3. When a process blocks on I/O, semaphore, mutex, etc.
     - Scheduler doesn't know process dependencies
  4. When an I/O interrupt occurs

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

Categories of Scheduling

1. Batch
   - Still in widespread use
   - Nonpreemptive or preemptive with long time quantum ok
2. Interactive
   - Preemptive to provide responsiveness
   - General purpose
3. Real Time
   - Preemptive not always needed

Scheduling Algorithm Goals

- All Systems
  - Fairness - give each process a fair share
  - Policy enforcement - see the stated policy is carried out
  - Balance - keep all parts of the system busy

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

Selecting the best algorithm

- Criteria:
  1. throughput - jobs run per hour or per minute
  2. average turnaround time - time from start to end of job
  3. average response time - time from submission to start of output
  4. CPU utilization - percent of time the CPU is running real jobs (not switching between processes or doing other overhead; of more interest in large systems)
  5. average wait time - the time that processes spend in the ready queue

Selecting the best algorithm

- Criteria:
  1. throughput - jobs run per hour or per minute
  2. average turnaround time - time from start to end of job
  3. average response time - time from submission to start of output

Depends on job mix, so difficult to compare fairly and accurately

Selecting the best algorithm

- Criteria:
  4. CPU utilization - percent of time the CPU is running real jobs (not switching between processes or doing other overhead; of more interest in large systems)

Interesting and easy to measure, but in personal computers we really don't care about it.

Selecting the best algorithm

- Criteria:
  5. average wait time - the time that processes spend in the ready queue

Makes the most sense. We want to make sure the that most computing is getting done in the least amount of wasted time

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

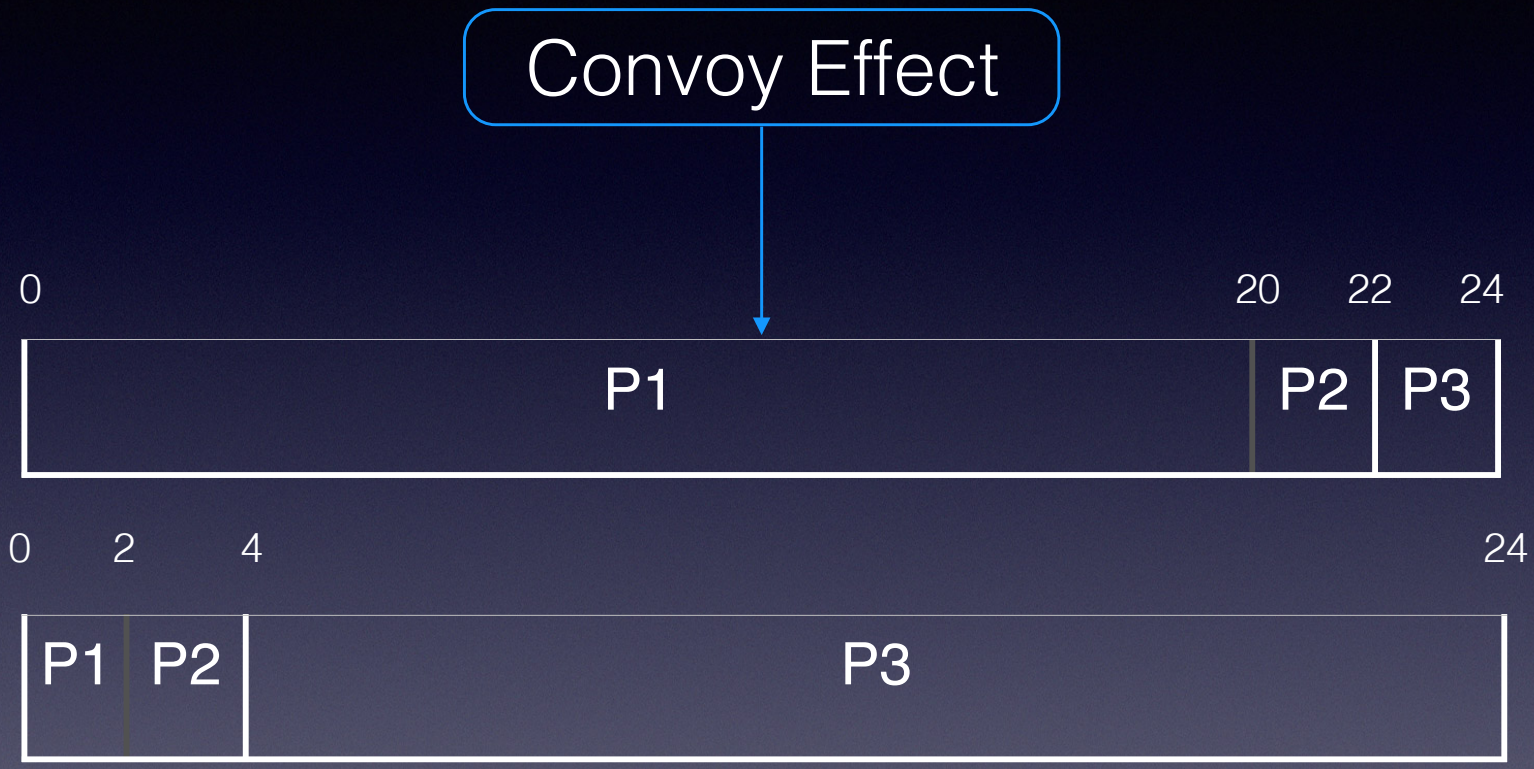
Average Waiting Time: (0 + 18 + 20) / 3 = 12.67

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



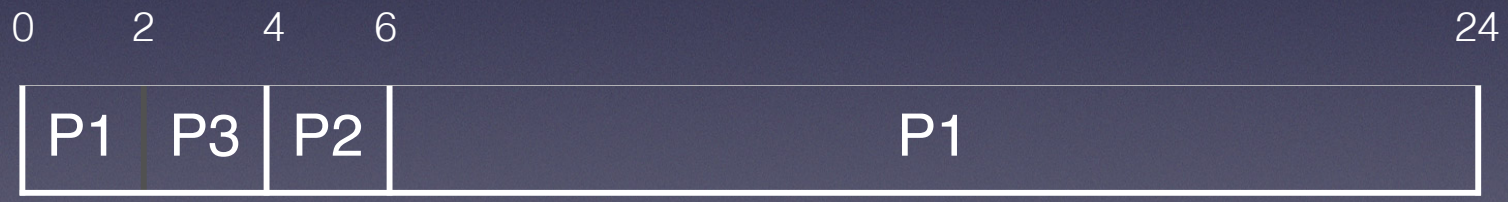
Very Dependent On Job Arrival Time

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



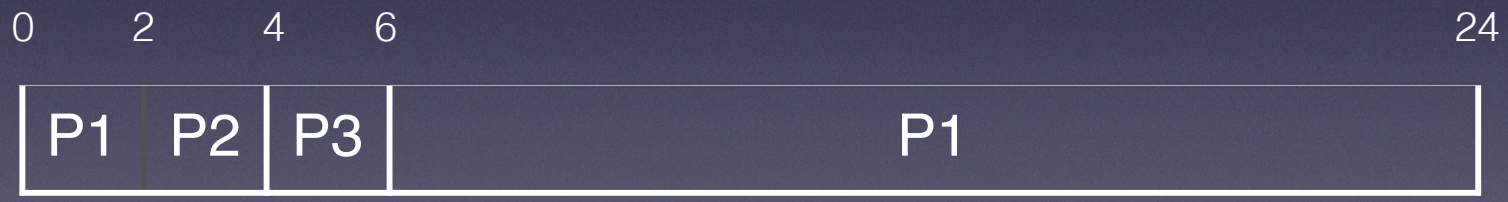
Average Waiting Time: (4 + 2 + 0) / 3 = 2

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



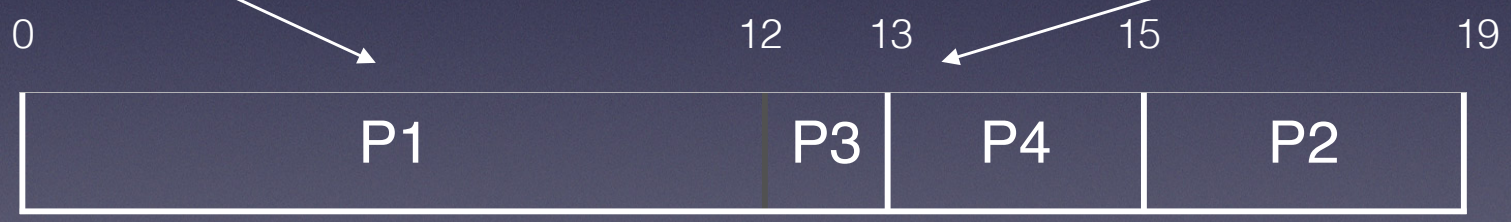
Average Waiting Time: (4 + 0 + 2) / 3 = 2

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

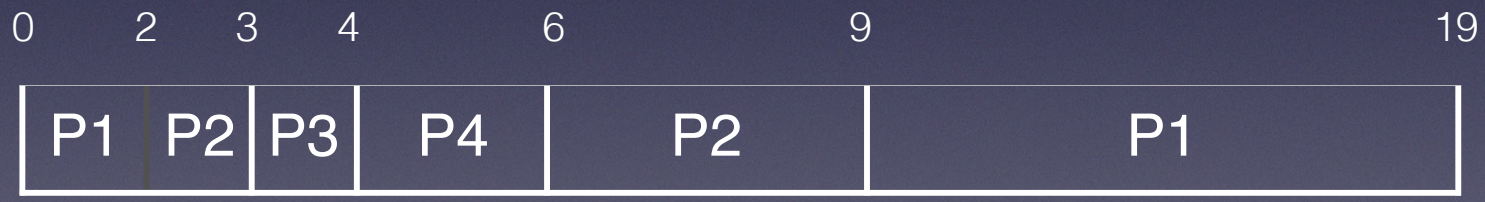
No preemption so P3, P4, and P2 wait. Then sorted by least runtime



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

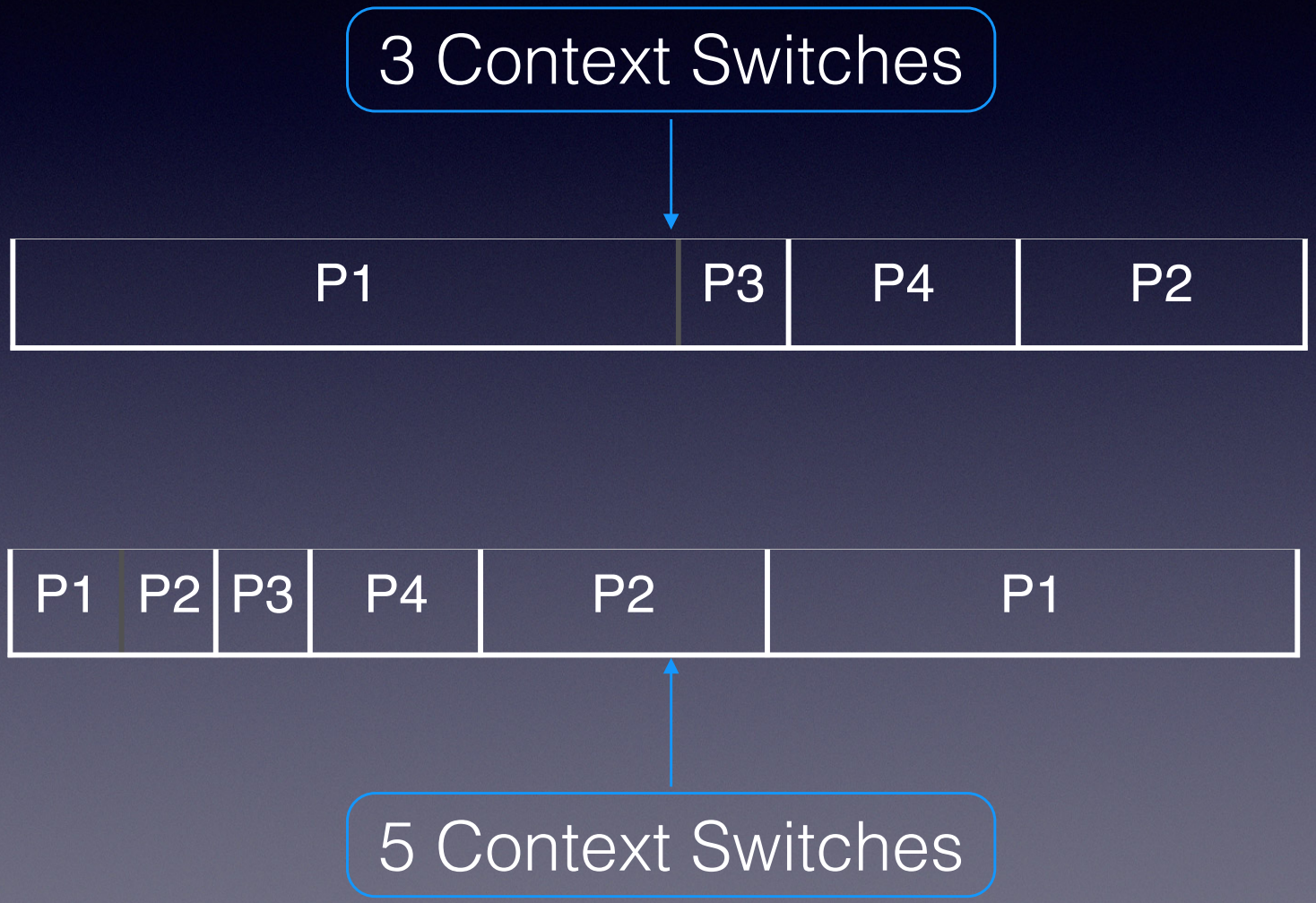
SJN With Preemption

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



Priority Scheduling

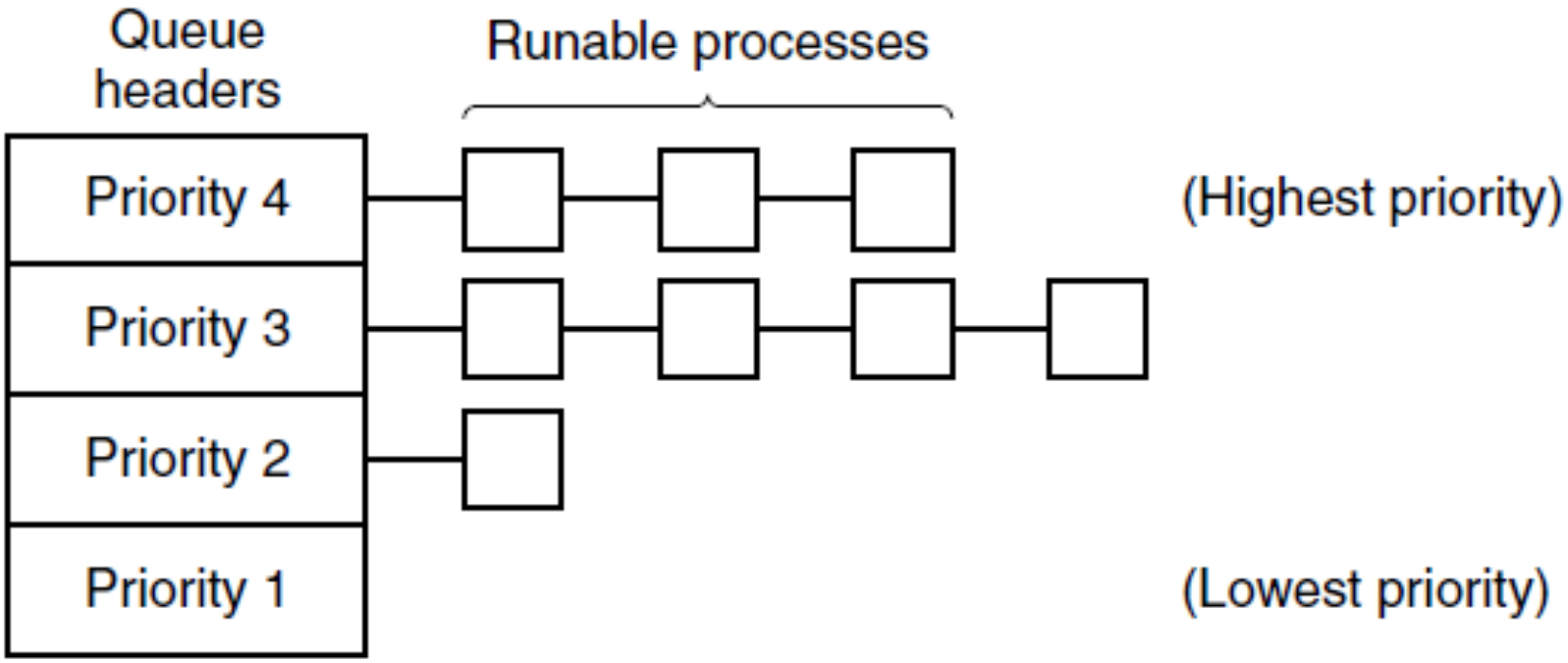


Figure 2-43. A scheduling algorithm with four priority classes.

Guaranteed Scheduling

If n users are logged in, you will receive 1/n of the CPU power.

Lottery Scheduling

- Process receives a lottery ticket
- Lottery ticket chosen at random and the winning process is allowed to run.
- More important processes can be given more lottery tickets.
- Highly responsive.
- Tickets can be exchanged by cooperating processes

Lottery Scheduling

- Lottery scheduling can solve problems difficult to handle by other schedulers.
- Video server with video streams of different frame rates. (10, 20, 25 frames per second)
  - Processes get 10, 20, 25 tickets