CS 5600 Computer Systems

Spring 2026

Virtualization: The CPU and Scheduling
Lecture 2
January 20, 2026

Prof. Scott Valcourt

s.valcourt@northeastern.edu

603-380-2860 (cell)

Portland, ME

The Roux Institute
Northeastern University

Lecture Agenda

- Abstraction – Chapter 4 – The Process
- Interlude – Chapter 5 – Process API
- Mechanism – Chapter 6 – Limited Direct Execution
- Scheduling – Chapter 7 – Introduction
- Scheduling – Chapter 8 – Multi-Level Feedback Queue
- Scheduling – Chapter 9 – Proportional Share

- Last Week: Chapter 1 - Introduction

What is a Process?

Process: An execution stream in the context of a process state

What is an execution stream?

- Stream of executing instructions
- Running piece of code
- “thread of control”

What is process state?

- Everything that the running code can affect or be affected by
- Registers
  - General purpose, floating point, status, program counter, stack pointer
- Address space
  - Heap, stack, and code
- Open files

Processes vs. Programs

A process is different than a program

- Program: Static code and static data
- Process: Dynamic instance of code and data

Can have multiple process instances of the same program

- Can have multiple processes of the same program
  Example: many users can run “ls” at the same time

Process API

- These APIs are available on any modern OS.
  - **Create**
    - Create a new process to run a program
  - **Destroy**
    - Halt a runaway process
  - **Wait**
    - Wait for a process to stop running
  - **Miscellaneous Control**
    - Some kind of method to suspend a process and then resume it
  - **Status**
    - Get some status info about a process

Process Creation

1. **Load** a program code into memory, into the address space of the process.
   - Programs initially reside on disk in *executable format*.
   - OS performs the loading process lazily.
     - Loading pieces of code or data only as they are needed during program execution.

2. The program's run-time **stack** is allocated.
   - Use the stack for *local variables*, *function parameters*, and *return address*.
   - Initialize the stack with arguments → `argc` and the `argv` array of `main()` function

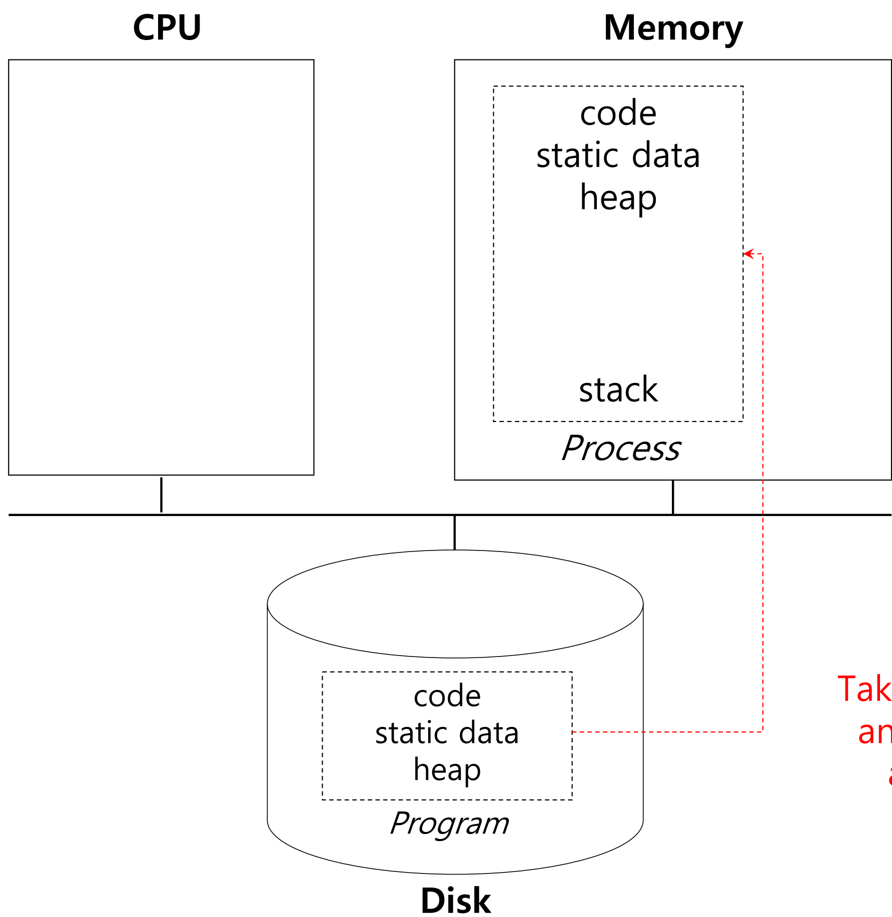
Process Creation (Cont.)

3. The program's **heap** is created.
   - Used for explicitly requested dynamically allocated data.
   - Program request such space by calling `malloc()` and free it by calling `free()`.

4. The OS does some other initialization tasks.
   - input/output (I/O) setup
     - Each process by default has three open file descriptors.
     - Standard input, output, and error

5. **Start the program** running at the entry point, namely `main()`.
   - The OS *transfers control* of the CPU to the newly-created process.

Loading: From Program To Process

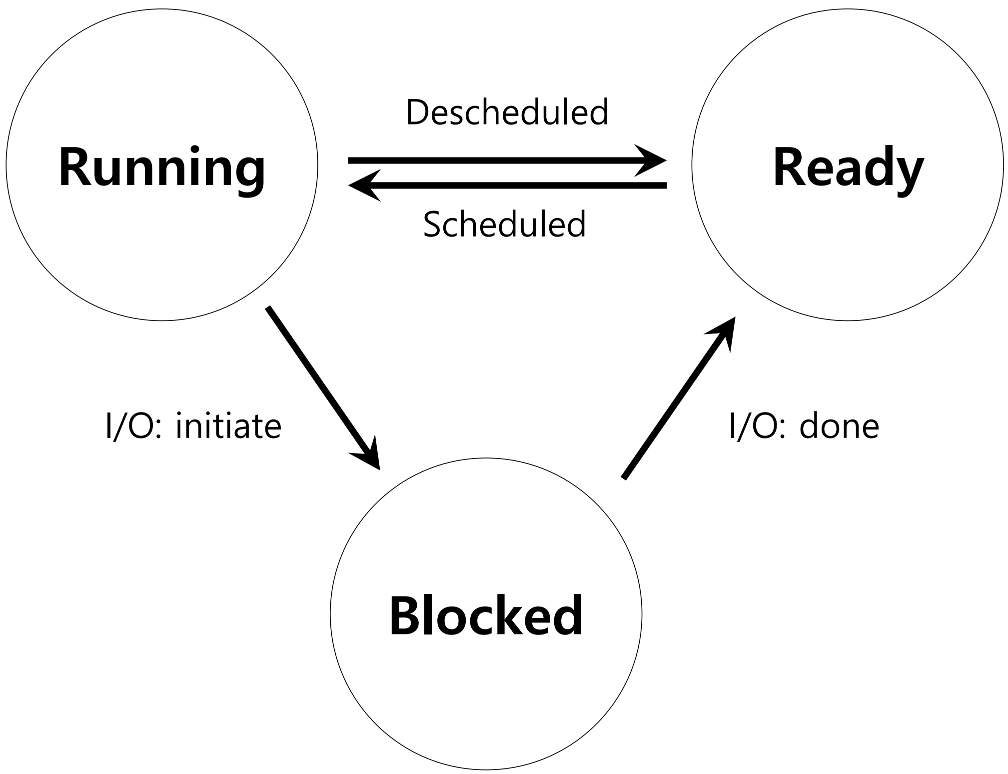


Loading:
Takes on-disk program and reads it into the address space of process

Process States

- A process can be one of three states.
  - **Running**
    - A process is running on a processor.
  - **Ready**
    - A process is ready to run but for some reason the OS has chosen not to run it at this given moment.
  - **Blocked**
    - A process has performed some kind of operation.
    - When a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

Process State Transition



Data structures

- The OS has some key data structures that track various relevant pieces of information.
  - **Process list**
    - Ready processes
    - Blocked processes
    - Current running process
  - **Register context**

- PCB (Process Control Block)
  - A C-structure that contains information about each process.

Example) The xv6 kernel Proc Structure

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;    // Index pointer register
    int esp;    // Stack pointer register
    int ebx;    // Called the base register
    int ecx;    // Called the counter register
    int edx;    // Called the data register
    int esi;    // Source index register
    int edi;    // Destination index register
    int ebp;    // Stack base pointer register
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```

Example) The xv6 kernel Proc Structure (Cont.)

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;          // Start of process memory
    uint sz;            // Size of process memory
    char *kstack;       // Bottom of kernel stack
                // for this process
    enum proc_state state;    // Process state
    int pid;            // Process ID
    struct proc *parent;  // Parent process
    void *chan;         // If non-zero, sleeping on chan
    int killed;         // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;    // Current directory
    struct context context;   // Switch here to run process
    struct trapframe *tf; // Trap frame for the
                    // current interrupt
};
```

How to provide the illusion of many CPUs?

- CPU virtualizing
  - The OS can promote the illusion that many virtual CPUs exist.
  - **Time sharing**: Running one process, then stopping it and running another
    - The potential cost is performance.

Processes vs. Threads

- A process is different than a thread
- Thread: "Lightweight process" (LWP)
  - An execution stream that shares an address space
  - Multiple threads within a single process
- Example:
  - Two **processes** examining same memory address 0xffe84264 see **different** values (i.e., different contents)
  - Two **threads** examining memory address 0xffe84264 see **same** value (i.e., same contents)

Virtualizing the CPU

Goal:
- Give each process the impression it alone is actively using CPU
- Resources can be shared in **time** and **space**
- Assume single uniprocessor
  - Time-sharing (multi-processors: advanced issue)
- Memory?
  - Space-sharing (later)
- Disk?
  - Space-sharing (later)

How to Provide Good CPU Performance?

Direct execution

- Allow user process to run directly on hardware
- OS creates process and transfers control to starting 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 and hardware maintain some control

Problem 1: Restricted OPS

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

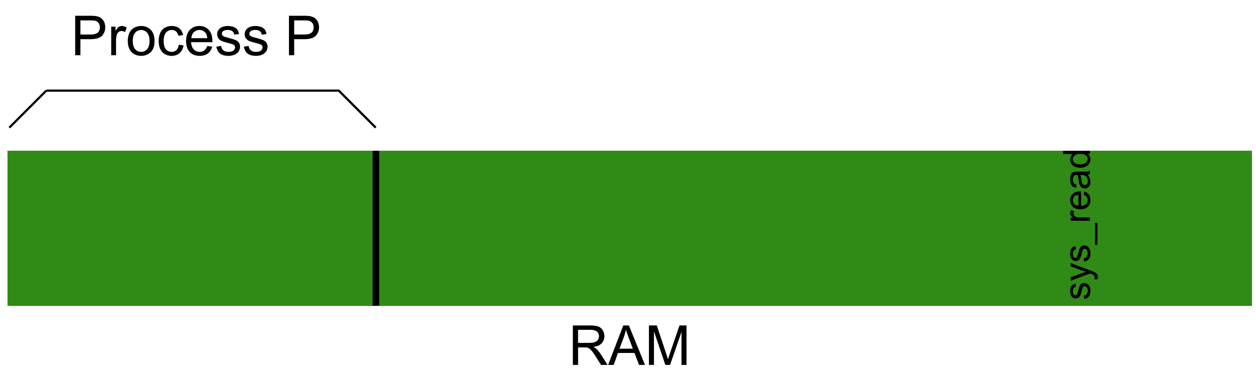
Solution: privilege levels supported by hardware (bit of status)

- User processes run in user mode (restricted mode)
- OS runs in kernel mode (not restricted)
  - Instructions for interacting with devices
  - Could have many privilege levels (advanced topic)

How can process access device?

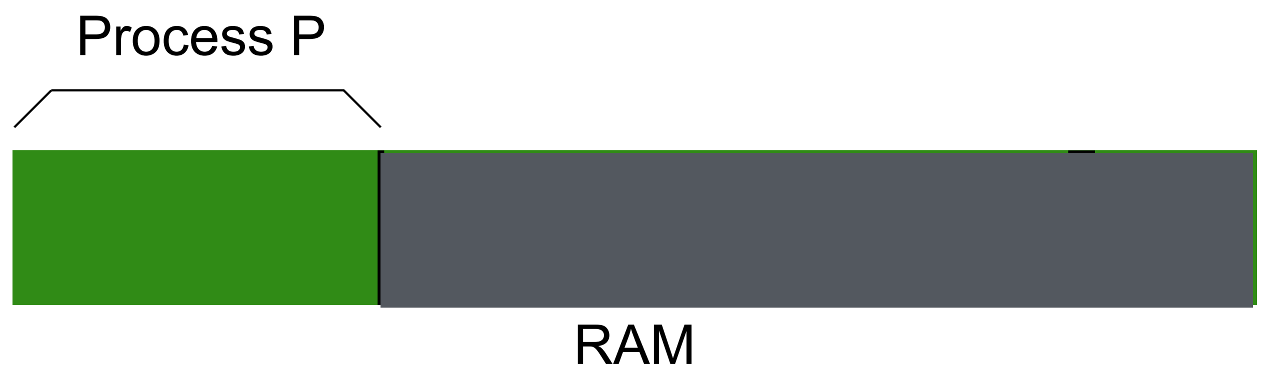
- System calls (function call implemented by OS)
- Change privilege level through system call (trap)

System Call



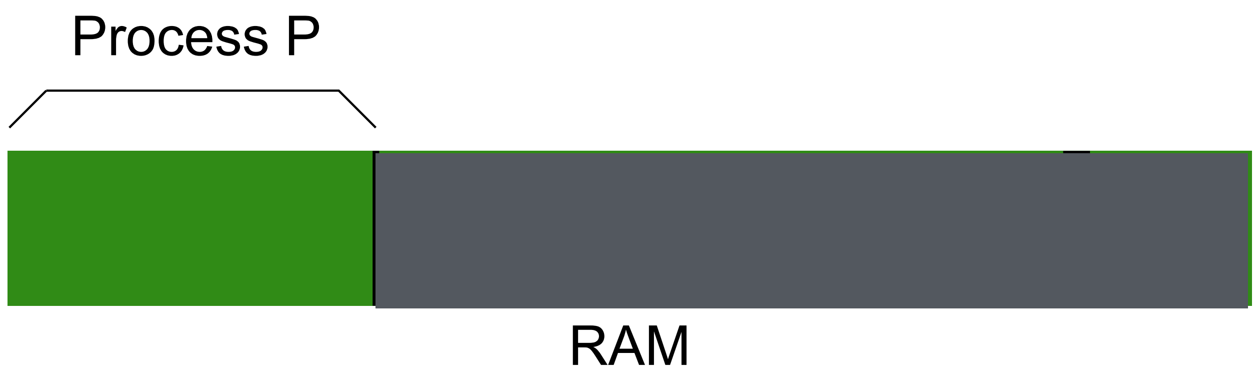
P wants to call read()

System Call



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

System Call



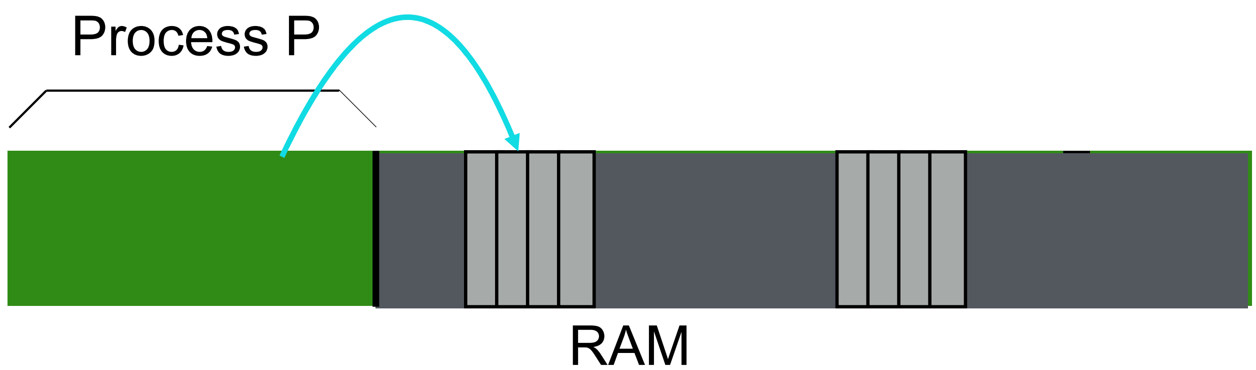
P wants to call read() but no way to call it directly

System Call

read():

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

System Call



read():

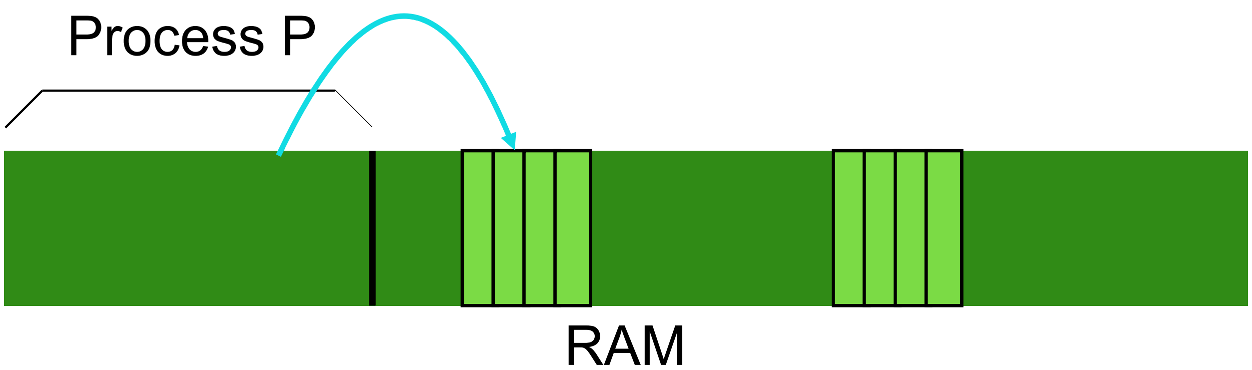
```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

System Call

Kernel mode: we can do anything!



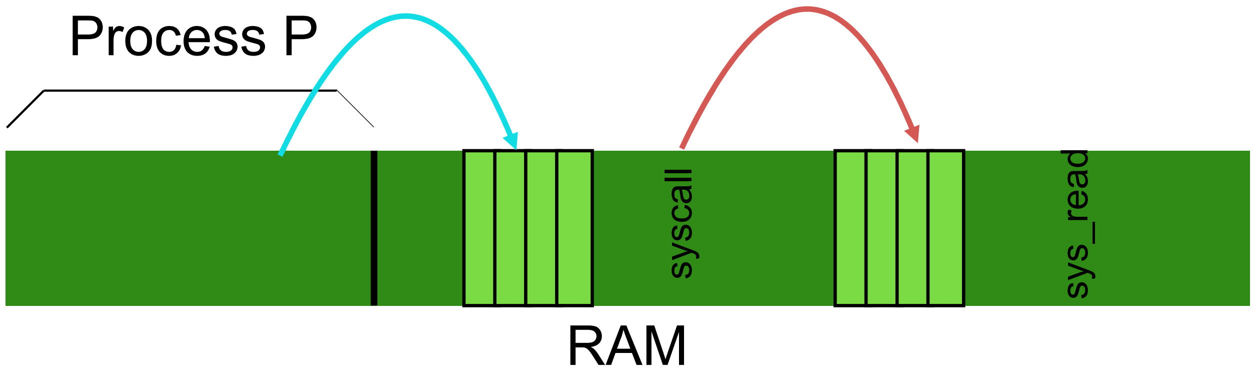
read():

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

syscall-table index   trap-table index

System Call

Follow entries to correct system call code



read():

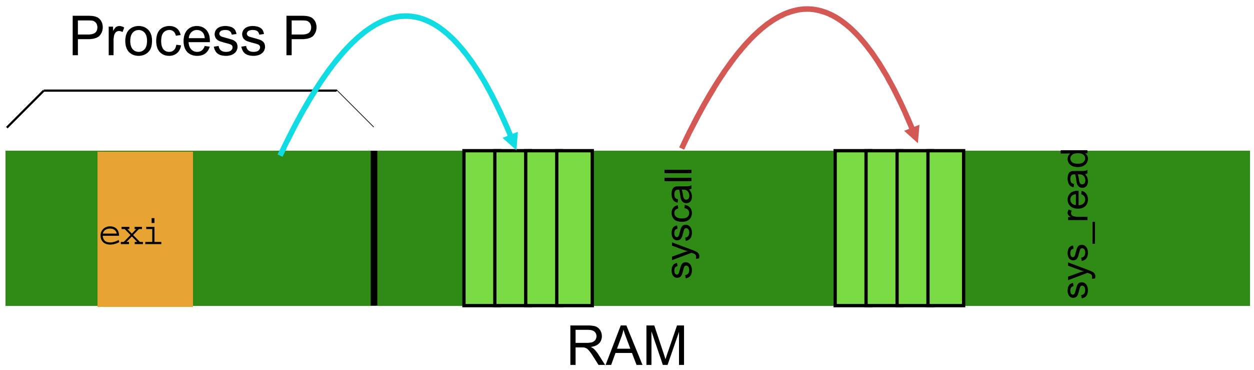
```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

System Call

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



read():

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

syscall-table index   trap-table index

What to limit?

User processes are not allowed to perform:

- General memory access
- Disk I/O
- Special x86 instructions like `lidt` (load interrupt descriptor table)

What if process tries to do something restricted?

Problem 2: How to take CPU AWAY?

OS requirements for **multiprogramming** (or multitasking)

- Mechanism
  - To switch between processes
- Policy
  - To decide which process to schedule when

Separation of policy and mechanism

- Reoccurring theme in OS
- Policy: Decision-maker to optimize some workload performance metric
  - Which process when?
  - Process **Scheduler**: Future lecture
- Mechanism: Low-level code that implements the decision
  - How?
  - Process **Dispatcher**: Today's lecture

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 dispatcher gain control?

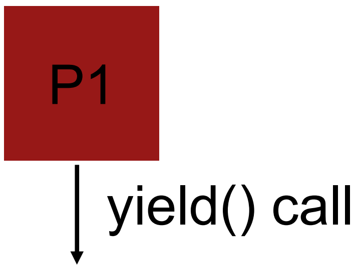
Question 2: What execution context must be saved and restored?

Q1: How does Dispatcher get CONTROL?

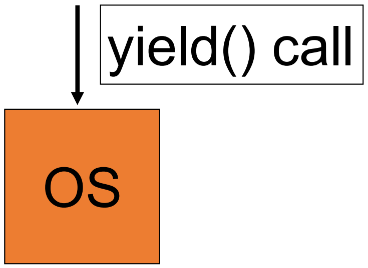
Option 1: Cooperative Multi-tasking

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

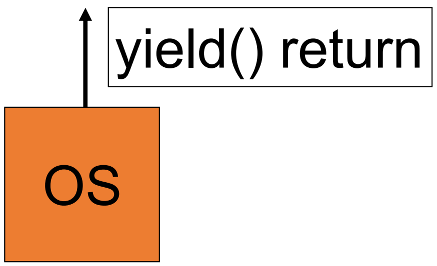
Cooperative Approach



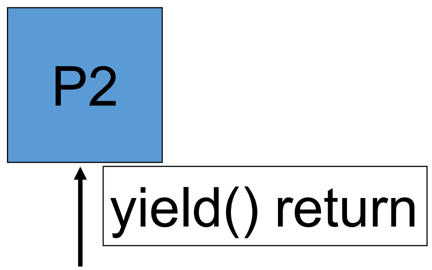
Cooperative Approach



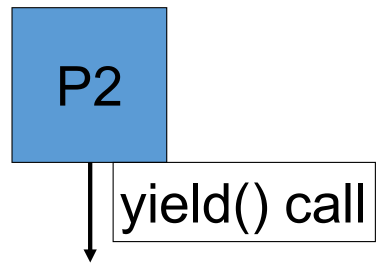
Cooperative Approach



Cooperative Approach



Cooperative Approach



Q1: How Does the Dispatcher RUN?

- 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 the Dispatcher RUN?

Option 2: True Multi-tasking

- 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 10 ms to 200 ms

Q2: What Context must be Saved?

Dispatcher must track context of process when not running

- Save context in process control block (PCB) (or, process descriptor)

What information is stored in PCB?

- PID
- Process state (I.e., running, ready, or blocked)
- Execution state (all registers, PC, 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)

Requires special hardware support

- Hardware saves process PC and PSR on interrupts

Context Switching

Operating System	Hardware	Program
		Process A ...

Context Switching

Operating System	Hardware	Program
		Process A …
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

Context Switching

Operating System	Hardware	Program
		Process A …
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call **switch()** routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) return-from-trap (into B)		

Context Switching

Operating System	Hardware	Program
		Process A …
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call **switch()** routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) return-from-trap (into B)		
	restore regs(B) from k-stack(B) move to user mode jump to B's IP	

Context Switching

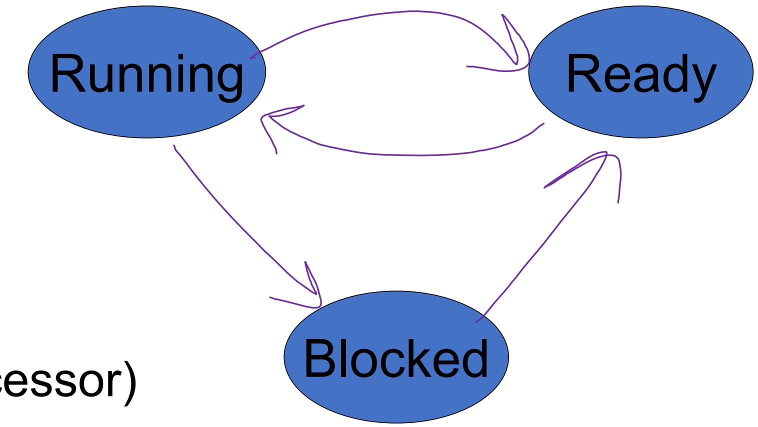
Operating System	Hardware	Program
		Process A …
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call **switch()** routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) return-from-trap (into B)		
	restore regs(B) from k-stack(B) move to user mode jump to B's IP	
		Process B …

The xv6 Context Switch Code

```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
   # Save old registers
   movl 4(%esp), %eax       # put old ptr into eax
   popl 0(%eax)        # save the old IP
   movl %esp, 4(%eax)       # and stack
   movl %ebx, 8(%eax)       # and other registers
   movl %ecx, 12(%eax)
   movl %edx, 16(%eax)
   movl %esi, 20(%eax)
   movl %edi, 24(%eax)
   movl %ebp, 28(%eax)

   # Load new registers
   movl 4(%esp), %eax       # put new ptr into eax
   movl 28(%eax), %ebp # restore other registers
   movl 24(%eax), %edi
   movl 20(%eax), %esi
   movl 16(%eax), %edx
   movl 12(%eax), %ecx
   movl 8(%eax), %ebx
   movl 4(%eax), %esp       # stack is switched here
   pushl 0(%eax)       # return addr put in place
   ret            # finally return into new ctxt
```

Problem 3: Slow Operations such as I/O?

When running process performs op that does not use CPU, OS switches to process that needs CPU (policy issues)



OS must track mode of each process:

- Running:
  - On the CPU (only one on a uniprocessor)
- Ready:
  - Waiting for the CPU
- Blocked
  - Asleep: Waiting for I/O or synchronization to complete

Transitions?

Problem 3: Slow Operations such as I/O?

OS must track every process in system
- Each process identified by unique Process ID (PID)

OS maintains queues of all processes
- 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

Next Topic: Policy for determining which **ready** process to run

Worried About Concurrency?

- What happens if, during interrupt or trap handling, another interrupt occurs?
- OS handles these situations:
  - **Disable interrupts** during interrupt processing
  - Use several sophisticated **locking** schemes to protect concurrent access to internal data structures.

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

Break

Process Creation

Two ways to create a process

- Build a new empty process from scratch
- Copy an existing process and change it appropriately

Option 1: New process from scratch

- Steps
  - Load specified code and data into memory;
    Create empty call stack
  - Create and initialize PCB (make look like context-switch)
  - Put process on ready list
- Advantages: No wasted work
- Disadvantages: Difficult to setup process correctly and to express all possible options
  - Process permissions, where to write I/O, environment variables
  - Example: WindowsNT has call with 10 arguments

Process Creation

Option 2: Clone existing process and change

- Example: Unix fork() and exec()
  - Fork(): Clones calling process
  - Exec(char *file): Overlays file image on calling process
- Fork()
  - Stop current process and save its state
  - Make copy of code, data, stack, and PCB
  - Add new PCB to ready list
  - Any changes needed to child process?
- Exec(char *file)
  - Replace current data and code segments with those in specified file
- Advantages: Flexible, clean, simple
- Disadvantages: Wasteful to perform copy and then overwrite of memory

Unix Process Creation

How are Unix shells implemented?

```
While (1) {
 Char *cmd = getcmd();
 Int retval = fork();
 If (retval == 0) {
   // This is the child process
   // Setup the child’s process environment here
   // E.g., where is standard I/O, how to handle signals?
   exec(cmd);
   // exec does not return if it succeeds
   printf(“ERROR: Could not execute %s\n”, cmd);
   exit(1);
} else {
   // This is the parent process; Wait for child to finish
   int pid = retval;
   wait(pid);
}
}
```

The fork() System Call

- Create a new process
  - The newly-created process has its own copy of the **address space**, **registers**, and **PC**.

p1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {      // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {           // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n",
        rc, (int) getpid());
    }
    return 0;
}
```

Calling fork() example (Cont.)

Result (Not deterministic)

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

or

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

The wait() System Call

- This system call won't return until the child has run and exited.

p2.c

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

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {      // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {           // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
        rc, wc, (int) getpid());
    }
    return 0;
}
```

The wait() System Call (Cont.)

Result (Deterministic)

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

The exec() System Call

- Run a program that is different from the calling program

p3.c

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

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {      // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");      // program: "wc" (word count)
        myargs[1] = strdup("p3.c");    // argument: file to count
        myargs[2] = NULL;          // marks end of array
        …
```

The exec() System Call (Cont.)

p3.c (Cont.)

```c
…
        execvp(myargs[0], myargs); // runs word count
        printf("this shouldn't print out");
    } else {                 // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
            rc, wc, (int) getpid());
    }
    return 0;
}
```

Result

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
29 107 1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

All of the above with redirection

p4.c

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

int
main(int argc, char *argv[]){
    int rc = fork();
    if (rc < 0) {     // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child: redirect standard output to a file
        close(STDOUT_FILENO);
        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
        …
```

All of the above with redirection (Cont.)

p4.c

```
        …
        // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");      // program: "wc" (word count)
        myargs[1] = strdup("p4.c");    // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);     // runs word count
    } else {                    // parent goes down this path (main)
        int wc = wait(NULL);
    }
    return 0;
}
```

Result

```
prompt> ./p4
prompt> cat p4.output
32 109 846 p4.c
prompt>
```

Break

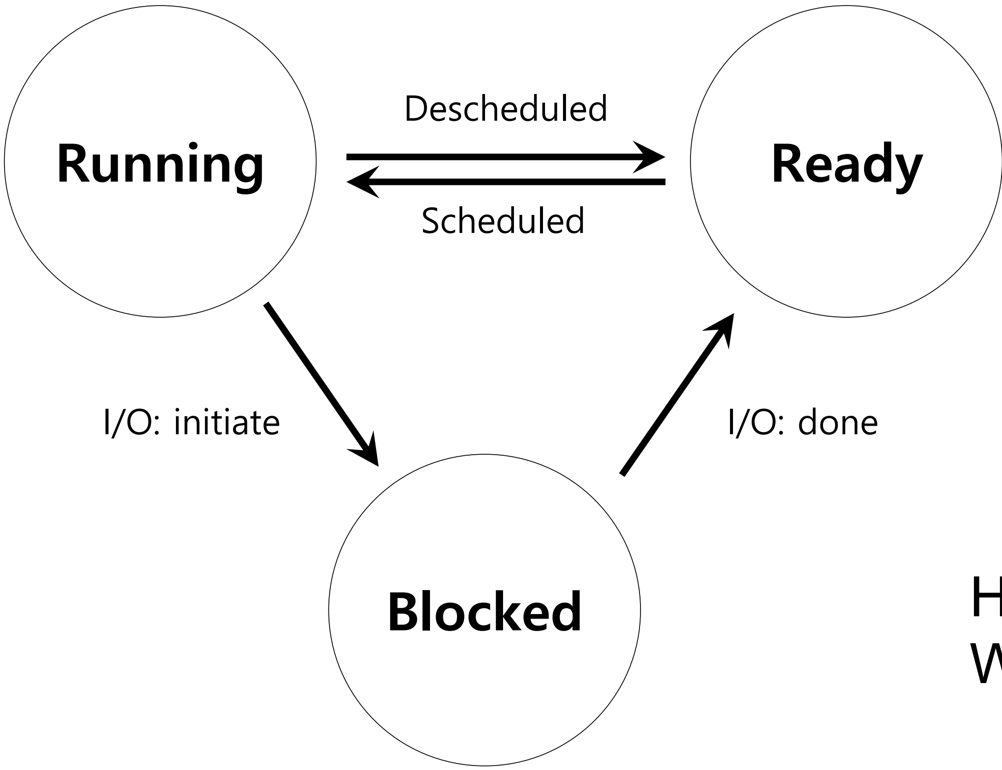
CPU Virtualization: Two Components

Dispatcher

- Low-level mechanism
- Performs context-switch
  - Switch from user mode to kernel mode
  - Save execution state (registers) of old process in PCB
  - Insert PCB in ready queue
  - Load state of next process from PCB to registers
  - Switch from kernel to user mode
  - Jump to instruction in new user process

- Scheduler
  - Policy to determine which process gets CPU when

Review: State Transitions



How to transition? ("mechanism")
When to transition? ("policy")

Vocabulary

Workload: set of **job** descriptions (arrival time, run_time)

- Job: Viewed as the current CPU burst of a process
- Process alternates between CPU and I/O
  process moves between ready and blocked queues

Scheduler: logic that decides which ready job to run

Metric: measurement of scheduling quality

Scheduling Performance Metrics

Minimize turnaround time
- Do not want to wait long for job to complete
- Completion_time – arrival_time

Minimize response time
- Schedule interactive jobs promptly so users see output quickly
- Initial_schedule_time – arrival_time

Minimize waiting time
- Do not want to spend much time in Ready queue

Maximize throughput
- Want many jobs to complete per unit of time

Maximize resource utilization
- Keep expensive devices busy

Minimize overhead
- Reduce number of context switches

Maximize fairness
- All jobs get same amount of CPU over some time interval

Workload Assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Example: workload, scheduler, metrics

JOB	arrival_time (s)	run_time (s)
A	~0	10
B	~0	10
C	~0	10

FIFO: First In, First Out
- also called FCFS (first come first served)
- run jobs in *arrival_time* order

What is our turnaround?: *completion_time* - *arrival_time*

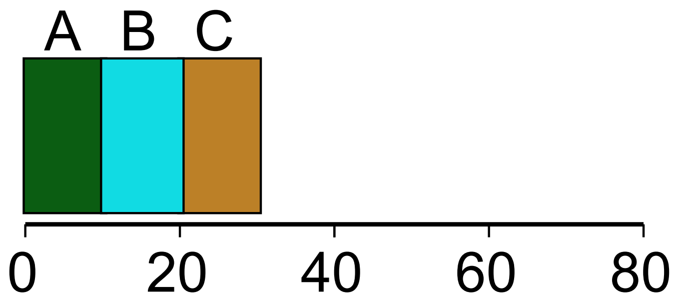
FIFO Event Trace

JOB	arrival_time (s)	run_time (s)
A	~0	10
B	~0	10
C	~0	10

Time	Event
0	A arrives
0	B arrives
0	C arrives
0	run A
10	complete A
10	run B
20	complete B
20	run C
30	complete C

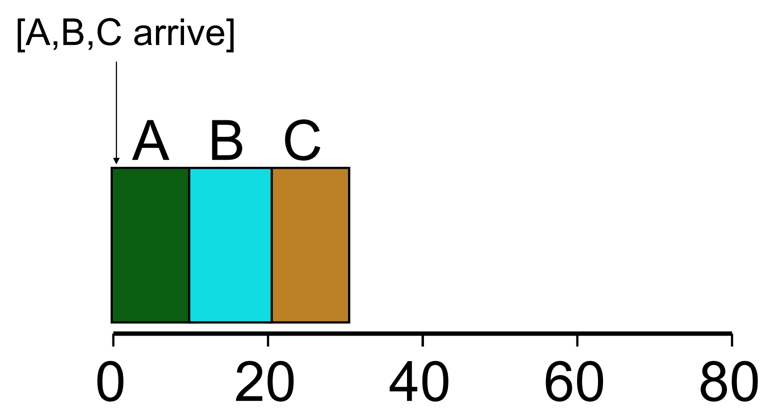
FIFO (Identical Jobs)

JOB	arrival_time (s)	run_time (s)
A	~0	10
B	~0	10
C	~0	10



Gantt chart:
Illustrates how jobs are scheduled over time on a CPU

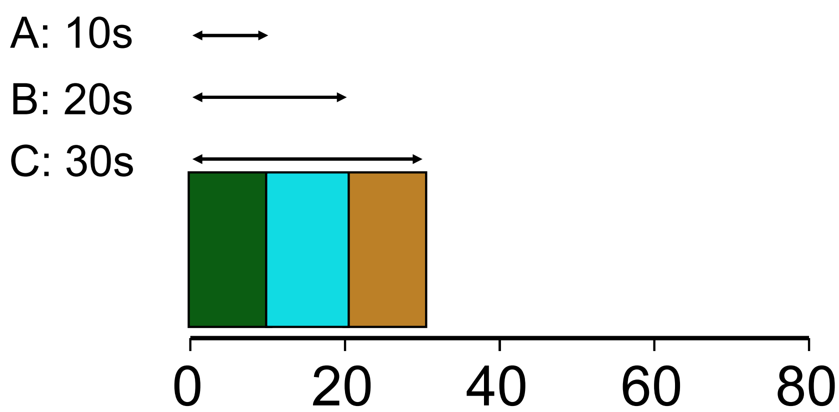
FIFO (Identical Jobs)



What is the average turnaround time?

Def: *turnaround_time = completion_time - arrival_time*

FIFO (Identical Jobs)



What is the average turnaround time?

Def: *turnaround_time* = *completion_time* - *arrival_time*

(10 + 20 + 30) / 3 = **20s**

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Workload Assumptions

~~1. Each job runs for the same amount of time~~
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

Any Problematic Workloads for FIFO?

- **Workload**: ?
- **Scheduler**: FIFO
- **Metric**: turnaround is high

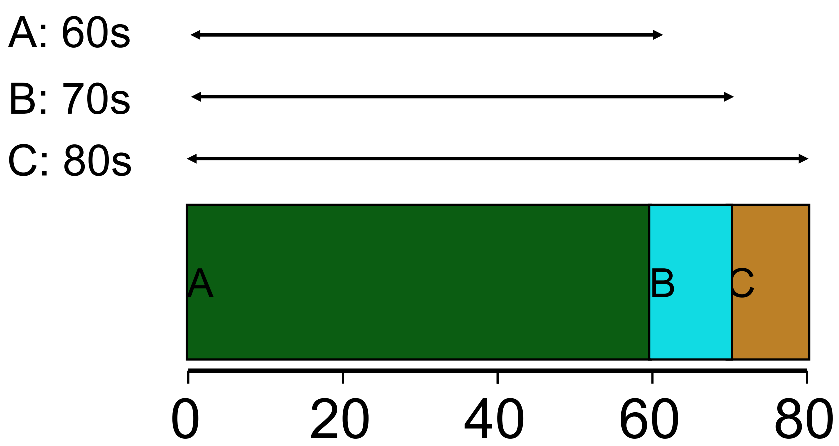
Example: Big First Job

JOB	arrival_time (s)	run_time (s)
A	~0	**60**
B	~0	10
C	~0	10

Draw Gantt chart for this workload and policy…

What is the average turnaround time?

Example: Big First Job



Average turnaround time: **70s**

Convoy Effect – Passing the Tractor

Problem with Previous Scheduler:

FIFO: Turnaround time can suffer when short jobs must wait for long jobs

New scheduler:

SJF (Shortest Job First)

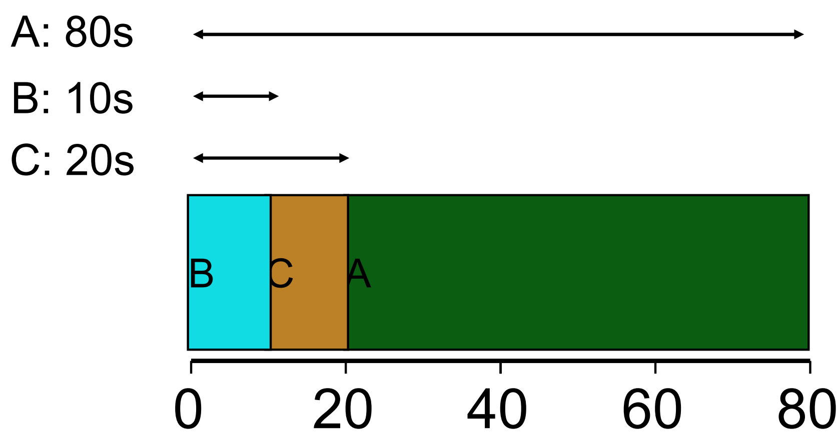
Choose job with smallest *run_time*

Example: Shortest Job First

JOB	arrival_time (s)	run_time (s)
A	~0	**60**
B	~0	10
C	~0	10

What is the average turnaround time with SJF?

SJF Turnaround Time



What is the average turnaround time with SJF?

(80 + 10 + 20) / 3 = **~36.7s**

Average turnaround with FIFO: 70s

For minimizing average turnaround time (with no preemption): SJF is provably optimal

Moving shorter job before longer job improves turnaround time of short job more than it harms turnaround time of long job

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Workload Assumptions

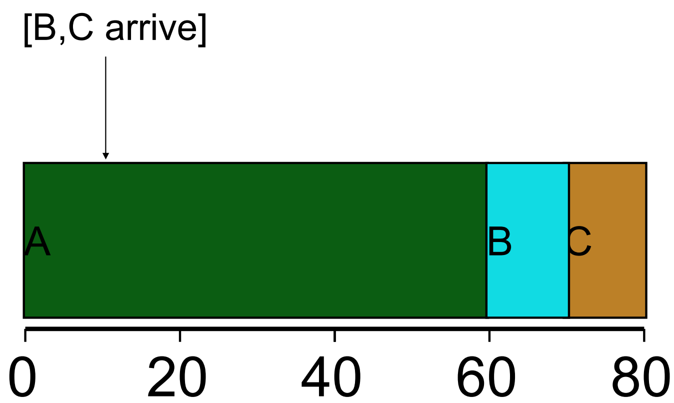
1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

Example: Shortest Job First (Arrival Time)

JOB	arrival_time (s)	run_time (s)
A	~0	60
B	**~10**	10
C	**~10**	10

What is the average turnaround time with SJF?

Stuck Behind that Tractor Again



JOB	arrival_time (s)	run_time (s)
A	~0	60
B	**~10**	10
C	**~10**	10

What is the average turnaround time?

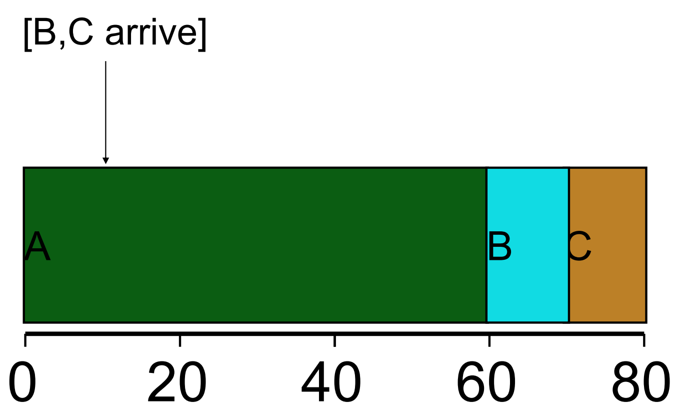
(60 + (70 – 10) + (80 – 10)) / 3 = **63.3s**

Preemptive Scheduling

Prev schedulers:
- FIFO and SJF are non-preemptive
- Only schedule new job when previous job voluntarily relinquishes CPU (performs I/O or exits)

New scheduler:
- Preemptive: Potentially schedule different job at any point by taking CPU away from running job
- STCF (Shortest Time-to-Completion First)
- Always run a job that will complete the quickest

Non-Preemptive: SJF

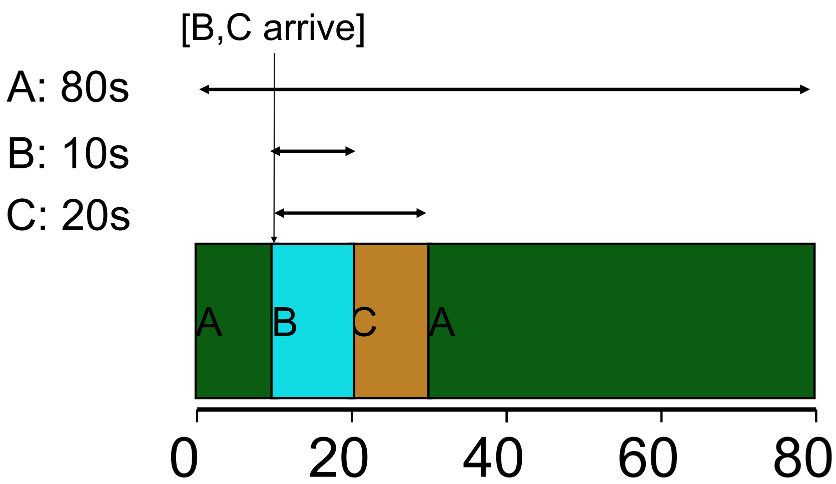


JOB	arrival_time (s)	run_time (s)
A	~0	60
B	**~10**	10
C	**~10**	10

What is the average turnaround time?

(60 + (70 – 10) + (80 – 10)) / 3 = **63.3s**

Preemptive: STCF



JOB	arrival_time (s)	run_time (s)
A	~0	60
B	**~10**	10
C	**~10**	10

Average turnaround time with STCF?

36.6s

Average turnaround time with SJF: **63.3s**

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

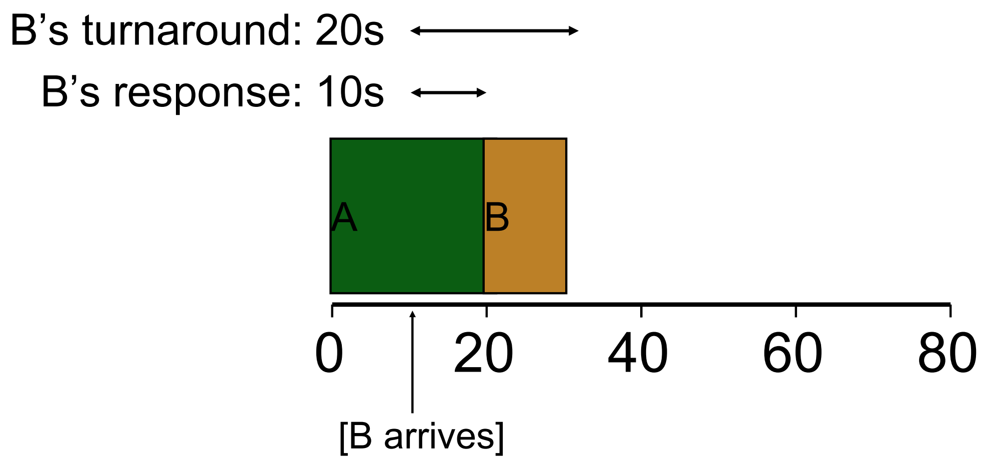
Response Times

Sometimes care about when job starts instead of when it finishes

New metric:

response_time = first_run_time - arrival_time

Response versus Turnaround



Round Robin Scheduler

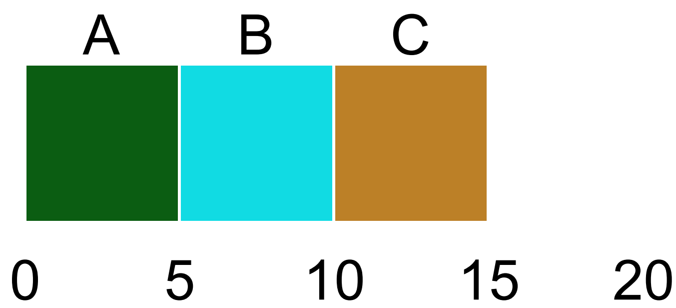
Prev schedulers:

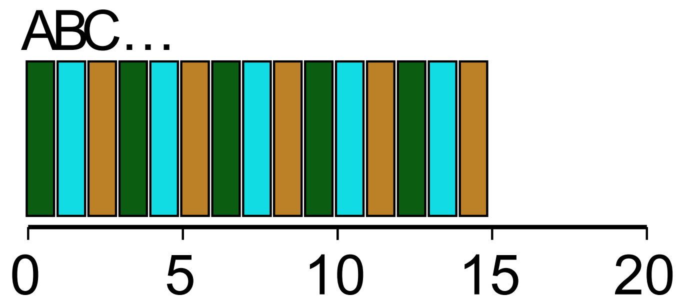
FIFO, SJF, and STCF can have poor response time

New scheduler: RR (Round Robin)

Alternates ready processes every fixed-length time-slice

FIFO versus RR





Avg Response Time?
(0+5+10)/3 = **5**

Avg Response Time?
(0+1+2)/3 = **1**

In what way is RR worse?
Average turn-around time with equal job lengths is horrible

Other reasons why RR could be better?
If don’t know run-time of each job, gives short jobs a chance to run and finish fast

Scheduling Basics

Workloads:
- arrival_time
- run_time

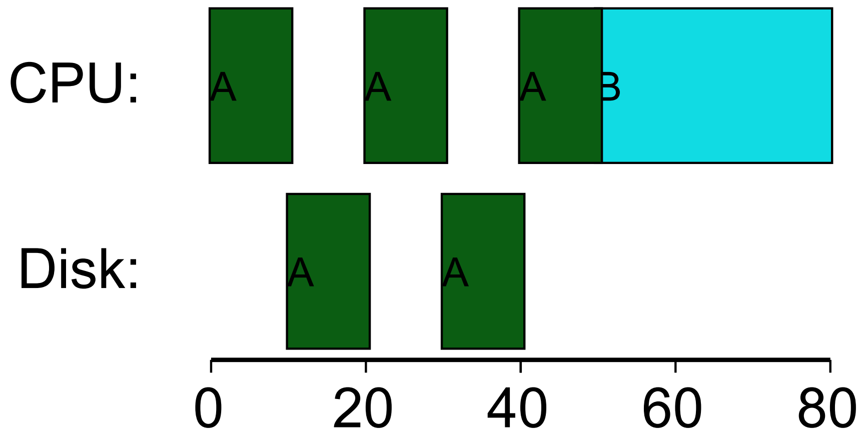
Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Workload Assumptions

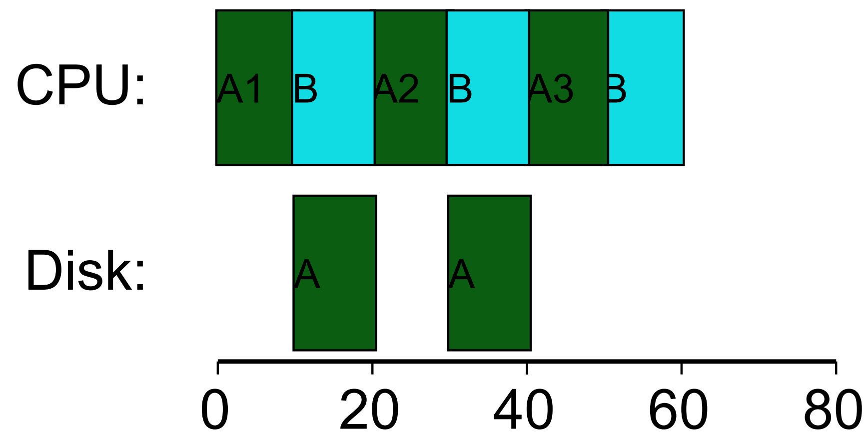
1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~All jobs only use the CPU (no I/O)~~
4. Run-time of each job is known

Not I/O Aware



Don’t let Job A hold on to CPU while blocked waiting for disk

I/O Aware (Overlap)



Treat Job A as 3 separate CPU bursts
When Job A completes I/O, another Job A is ready

Each CPU burst is shorter than Job B, so with SCTF, Job A preempts Job B

Workload Assumptions

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~All jobs only use the CPU (no I/O)~~
4. ~~Run-time of each job is known~~

    (need smarter, fancier scheduler)

MLFQ (Multi-Level Feedback Queue)

Goal: general-purpose scheduling

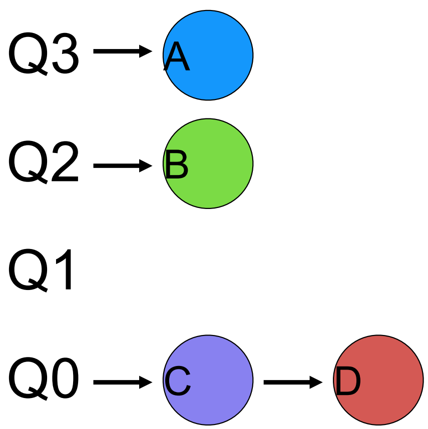
Must support two job types with distinct goals

- "interactive" programs care about response time
- "batch" programs care about turnaround time

Approach: multiple levels of round-robin; each level has higher priority than lower levels and preempts them

Priorities

Rule 1: If priority(A) > Priority(B), A runs
Rule 2: If priority(A) == Priority(B), A & B run in RR



“Multi-level”

How to know how to set priority?

Approach 1: nice
Approach 2: history “feedback”

History

- Use past behavior of process to predict future behavior
  - Common technique in systems
- Processes alternate between I/O and CPU work
- Guess how CPU burst (job) will behave based on past CPU bursts (jobs) of this process

More MLFQ Rules

Rule 1: If priority(A) > Priority(B), A runs

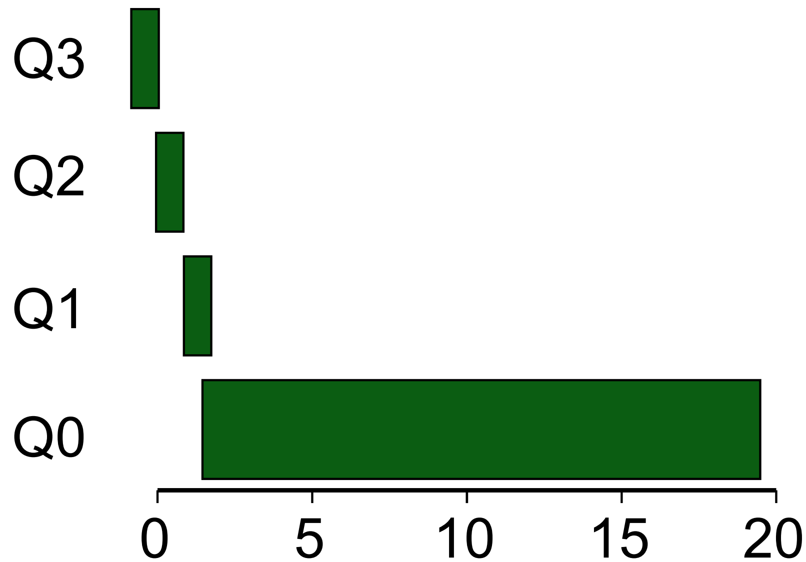
Rule 2: If priority(A) == Priority(B), A & B run in RR

More rules:

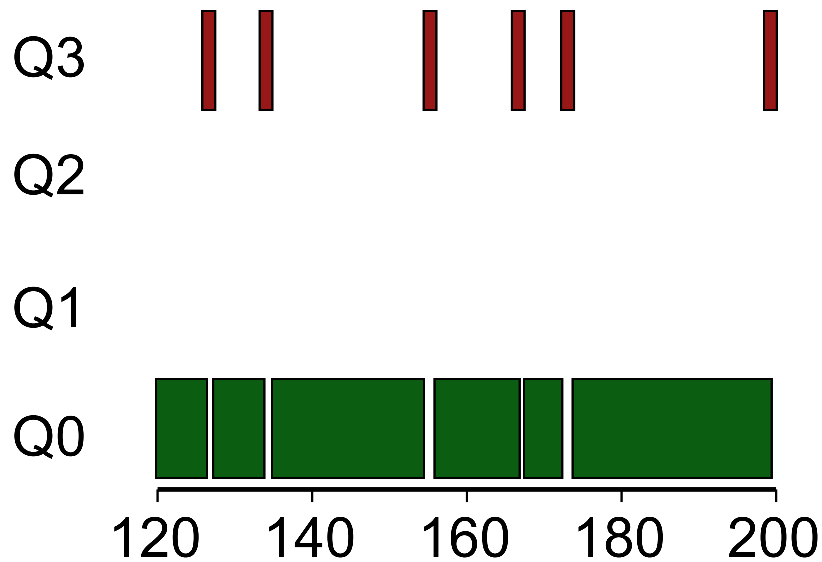
Rule 3: Processes start at top priority

Rule 4: If job uses whole slice, demote process
(longer time slices at lower priorities)

One Long Job Example

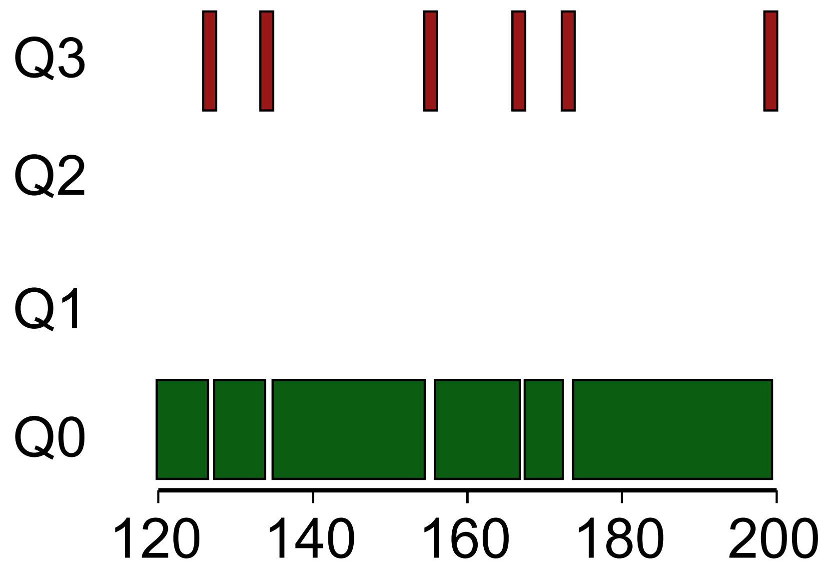


An Interactive Process Joins



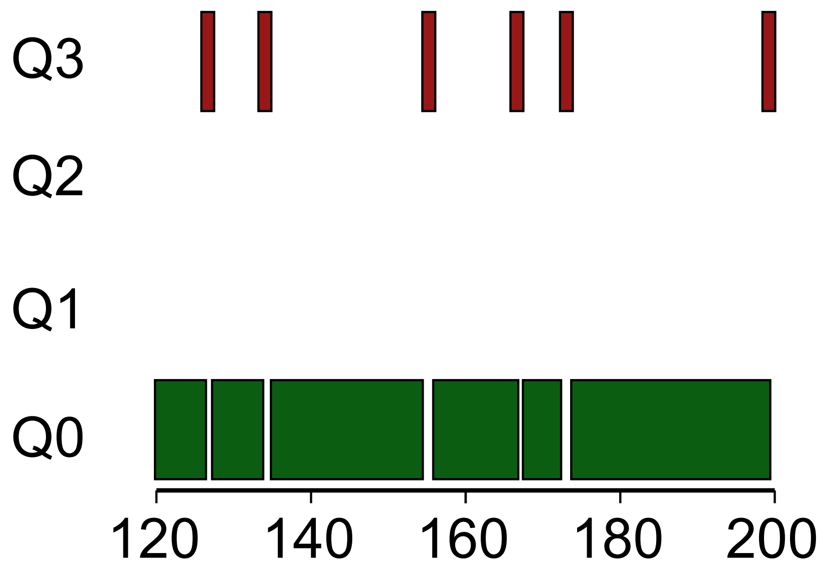
Interactive process never uses entire time slice, so never demoted

Problems with MLFQ?



Problems
- unforgiving + starvation
- gaming the system

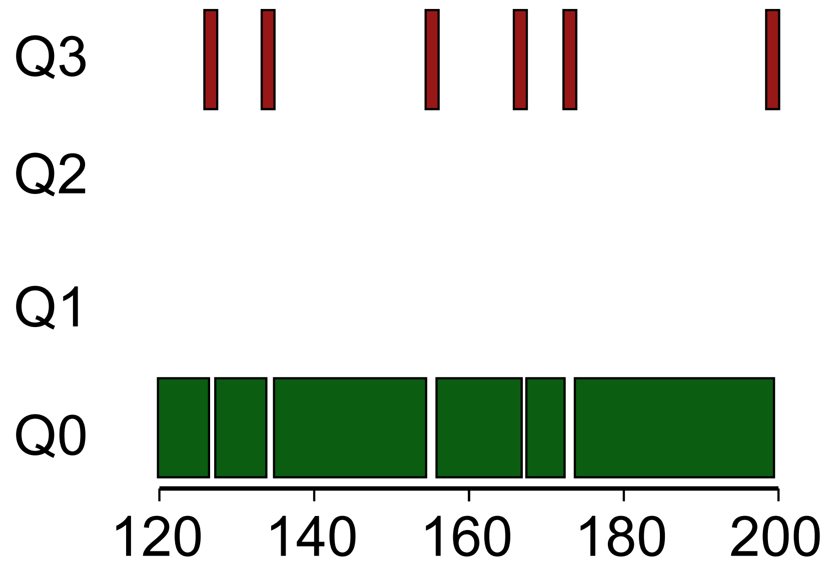
Prevent Starvation



Problem: Low priority job may never get scheduled

Periodically boost priority of all jobs (or all jobs that haven't been scheduled)

Prevent Gaming



Problem: High priority job could trick scheduler and get more CPU by performing I/O right before time-slice ends

Fix: Account for job's total run time at priority level (instead of just this time slice); downgrade when exceeding threshold

Proportional Share Scheduler – Lottery

- Fair-share scheduler
  - Guarantee that each job obtain *a certain percentage* of CPU time.
  - Not optimized for turnaround or response time

Approach:
- give processes lottery tickets
- whoever wins runs
- higher priority => more tickets

Amazingly simple to implement

Basic Concept

- Tickets
  - Represent the share of a resource that a process should receive
  - The percent of tickets represents its share of the system resource in question.

- Example
  - There are two processes, A and B.
    - Process A has 75 tickets → receive 75% of the CPU
    - Process B has 25 tickets → receive 25% of the CPU

Lottery scheduling

- The scheduler picks <u>a winning ticket</u>.
  - Load the state of that *winning process* and runs it.
- Example
  - There are 100 tickets
    - Process A has 75 tickets: 0 ~ 74
    - Process B has 25 tickets: 75 ~ 99

Scheduler's winning tickets:	63	85	70	39	76	17	29	41	36	39	10	99	68	83	63
Resulting scheduler:	A	B	A	A	B	A	A	A	A	A	A	B	A	B	A

The longer these two jobs compete,
The more likely they are to achieve the desired percentages.

Ticket Mechanisms

- Ticket currency
  - A user allocates tickets among their own jobs in whatever currency they would like.
  - The system converts the currency into the correct global value.
  - Example
    - There are 200 tickets (Global currency)
    - Process A has 100 tickets
    - Process B has 100 tickets

User A → *500* (A's currency) to A1 → *50* (global currency)
→ *500* (A's currency) to A2 → *50* (global currency)

User B → *10* (B's currency) to B1 → *100* (global currency)

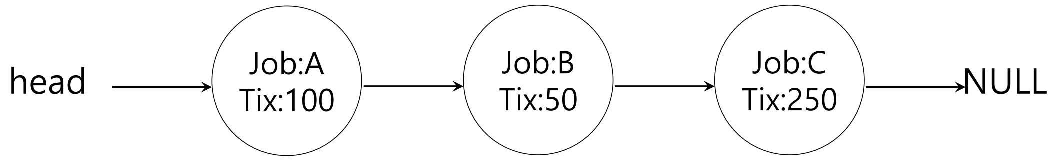
Ticket Mechanisms (Cont.)

- Ticket transfer
  - A process can temporarily hand off *its tickets* to another process.

- Ticket inflation
  - A process can temporarily raise or lower the number of tickets is owns.
  - If any one process needs *more CPU time*, it can boost its tickets.

Implementation

- Example: There are there processes, A, B, and C.
  - Keep the processes in a list:



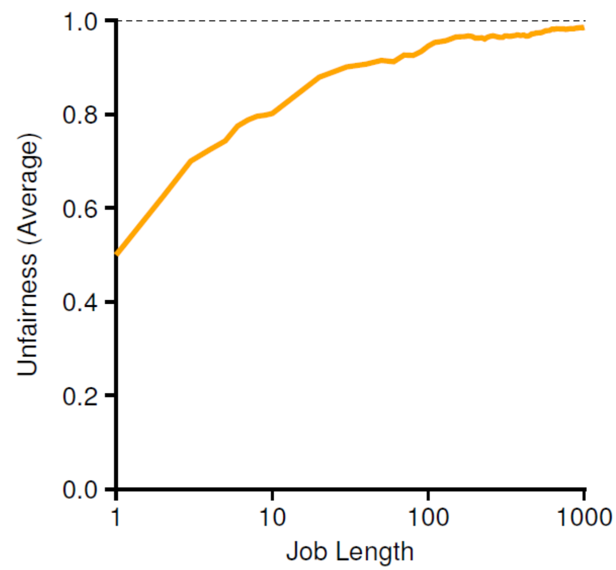

```c
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
// get a value, between 0 and the total # of tickets
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
     counter = counter + current->tickets;
     if (counter > winner)
          break; // found the winner
     current = current->next;
}
// 'current' is the winner: schedule it...
```

Implementation (Cont.)

- `U`: unfairness metric
  - The time the first job completes divided by the time that the second job completes.
- Example:
  - There are two jobs, each jobs has runtime 10.
    - First job finishes at time 10
    - Second job finishes at time 20
  - `U`$= \frac{10}{20} = 0.5$
  - `U` will be close to 1 when both jobs finish at nearly the same time.

Lottery Fairness Study

- There are two jobs.
  - Each jobs has the same number of tickets (100).



When the job length is not very long,
average unfairness can be quite severe.

Stride Scheduling

- Stride of each process
  - (A large number) / (the number of tickets of the process)
  - Example: A large number = 10,000
    - Process A has 100 tickets → stride of A is 100
    - Process B has 50 tickets → stride of B is 200

- A process runs, increment a counter(=pass value) for it by its stride.
  - Pick the process to run that has the lowest pass value

```
current = remove_min(queue);        // pick client with minimum pass
schedule(current);                  // use resource for quantum
current->pass += current->stride;   // compute next pass using stride
insert(queue, current);             // put back into the queue
```

A pseudo code implementation

Summary

- Understand goals (metrics) and workload, then design scheduler around that
- General purpose schedulers need to support processes with different goals
- Past behavior is good predictor of future behavior
- Random algorithms (lottery scheduling) can be simple to implement and avoid corner cases.

What's Next

- Module Software Preparation (HW0)
  - Due by Sunday, January 25 by 11:59pm ET
  - Install Virtual Machine (VirtualBox or UTM or something else)
  - Install GitHub repository
  - Verify C Compiler works
  - Push file into GitHub private repo

- Programming Assignment 1
  - Due by Sunday, February 1 by 11:59pm ET
  - Write a UNIX Shell

- Homework Assignment 1
  - Due by Sunday, February 1 by 11:59pm ET

- Quiz 1
  - Due by Sunday, February 1 by 11:59pm ET