UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

# CPU VIRTUALIZATION: SCHEDULING

**Questions answered in this lecture:**

What are different scheduling policies, such as:
FCFS, SJF, STCF, RR and MLFQ?

What type of workload performs well with each scheduler?

# ANNOUNCEMENTS

- Reading:
  - Today cover Chapters 7-9
- Project 1: Sorting and System Calls
  - Sorting : Warm-up with using C
    - Finish Part A this week
    - Competition:
      - Free text book or t-shirt to fastest (average) sort in each discussion section
  - Handin directories available
  - Goal is for everyone to learn material – get help if needed!
  - Lots of office hours
- Discussion section switches made tomorrow
- WACM explains Wed and Thu evening

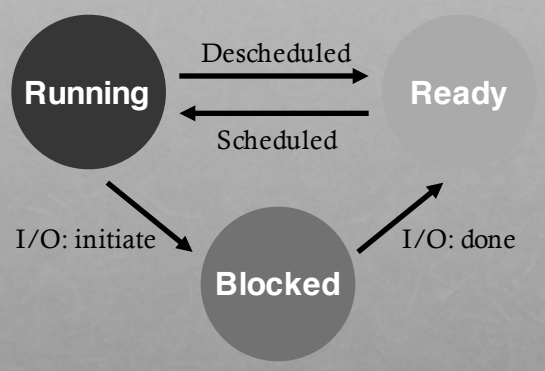
# CPU Virtualization: Two Components

Dispatcher (Previous lecture)

- 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 (Today)
  - Policy to determine which process gets CPU when

# Review: State Transitions




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

# Scheduling Terminology

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

- Job: Assume is 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
- Can't control how long job needs to run; minimize time before scheduled
- Initial_schedule_time – arrival_time

Maximize throughput (jobs completed / second)
- Want many jobs to complete per unit of time

Maximize resource utilization (% time CPU busy)
- Keep expensive devices busy

Minimize overhead (# of context switches and resulting cache misses)
- Reduce number of context switches

Maximize fairness (variation of CPU time across jobs)
- All jobs get same amount of CPU over some time interval

# Workload Assumptions

Start with easiest workload possible, slowly make more realistic and more challenging

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. Run-time of each job is known

# Scheduling Basics

**Workloads**:
- arrival_time
- run_time

**Schedulers**:
- FIFO
- SJF
- STCF
- RR

**Metrics**:
- turnaround_time
- response_time

# EASIEST EXAMPLE

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

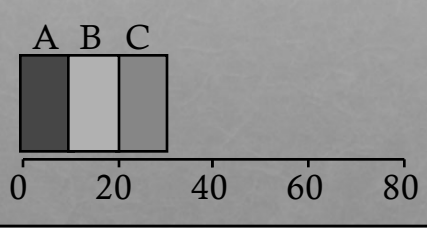
**FIFO**: First In, First Out
- also called FCFS (first come first served)
- run jobs in *arrival_time* order

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

# FIFO FOR IDENTICAL JOBS

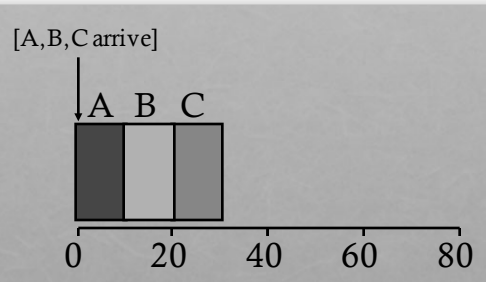
| JOB | arrival_time (s) | run_time (s) |
|---|---|---|
| A | ~0 | 10 |
| B | ~0 | 10 |
| C | ~0 | 10 |

Gantt chart:
Illustrate how jobs are scheduled over time




| 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)




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

# WORKLOAD ASSUMPTIONS

~~1. Each job runs for the same amount of time~~

2. All jobs arrive at the same time

3. The run-time of each job is known

# ANY PROBLEMATIC WORKLOADS FOR FIFO?

**Workload**: ?

**Scheduler**: FIFO

**Metric**: turnaround is high

What problematic workload can you devise?

# EXAMPLE: BIG FIRST JOB

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

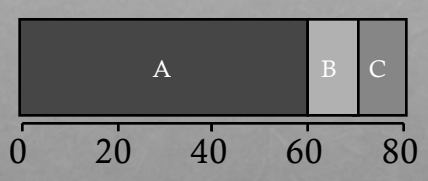
Draw Gantt chart for this workload and policy…
What is the average turnaround time?

# EXAMPLE: BIG FIRST JOB

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

Turnaround time of each job?

A: 60s
B: 70s
C: 80s




Average turnaround time: **70s**

# Convoy Effect

# Passing the Tractor

**Problem with FIFO Scheduler:**

Turnaround time suffers when short jobs must wait for long jobs

**New scheduler**:

SJF (Shortest Job First)

Choose job with smallest *run_time*

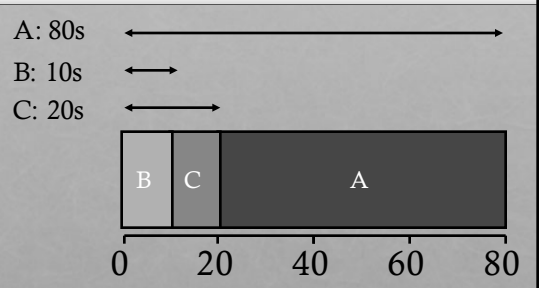
# SHORTEST JOB FIRST

| 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

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




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

# Workload Assumptions

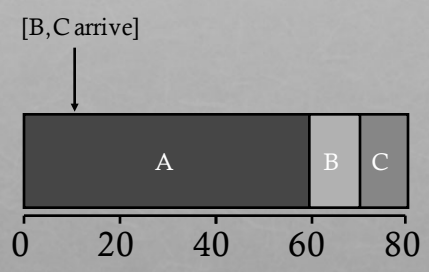
1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. The run-time of each job is known

# 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 a 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**:

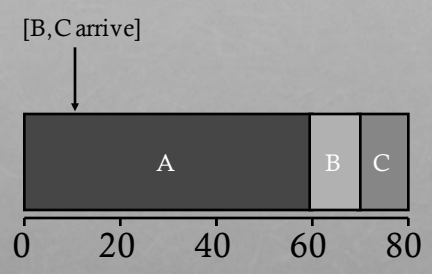
- FIFO and SJF are non-preemptive
- Only schedule new job when previous job voluntarily relinquishes CPU (e.g., performs I/O or exits)

**New scheduler**:

- Preemptive: Potentially schedule different job at any point by taking CPU away from running job
- STCF (Shortest Time-to-Completion First) or SCTF
- Always run 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 |



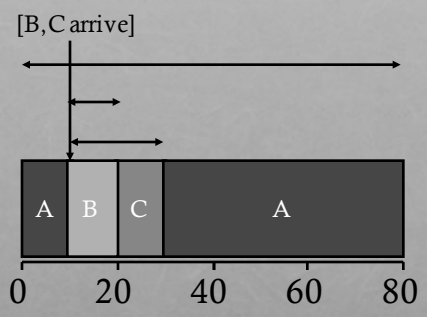

Average turnaround time:

(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 |

A: 80s
B: 10s
C: 20s

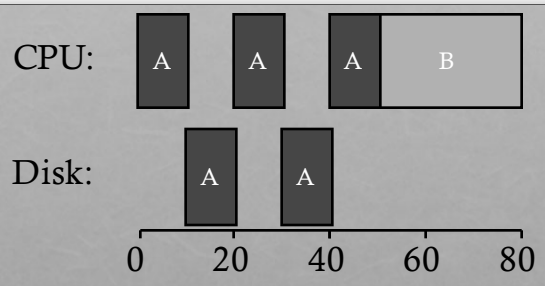



Average turnaround time with SJF: **63.3s**

Average turnaround time with STCF?

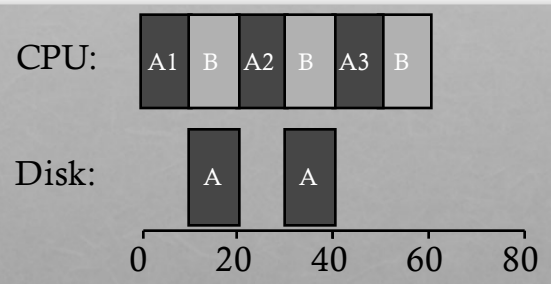
**36.6**

# SCHEDULER ONLY RUNS READY JOBS




Don't let A hold onto CPU while blocked waiting for disk

# I/O AWARE (OVERLAP)




Treat A as 3 separate CPU bursts
When Job Ai completes I/O, Job A(i+1) is ready

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

# BREAK!

- What other courses are your neighbors taking?
- What has been their favorite course in CS? Outside of CS?

# Scheduling Basics

**Workloads**:
- arrival_time
- run_time

**Schedulers**:
- FIFO
- SJF
- STCF
- RR

**Metrics**:
- turnaround_time

# Workload Assumptions

~~1. Each job runs for the same amount of time~~

~~2. All jobs arrive at the same time~~

~~3. The run-time of each job is known~~

# What if do not know job runtime?

- If jobs have same length
  - FIFO is fine
- If jobs have different lengths
  - SJF is better
- How can OS get short jobs to complete first if OS doesn't know which are short?

# Round-Robin Scheduler

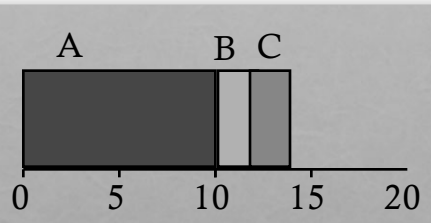
**New scheduler**: RR (Round Robin)

Alternate ready processes every fixed-length time-slice

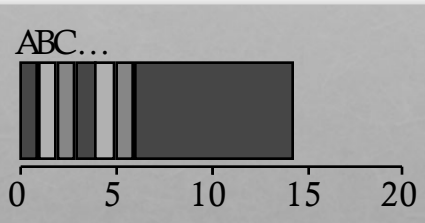
Short jobs will finish after fewer time-slices

Will finish sooner than long jobs

# FIFO VS RR: JOBS DIFFERENT LENGTHS



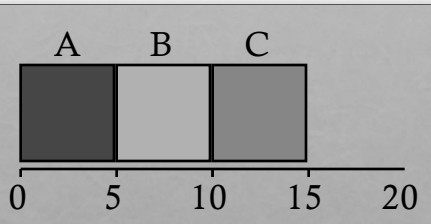

Avg Turnaround Time?
(10+12+14)/3 = **12**



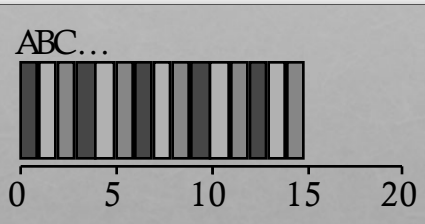

Avg Turnaround Time?
(5+6+14+)/3 = **8.3**

If don't know run-time of each job, gives short jobs a chance to run and finish fast

# FIFO VS RR: JOBS SAME LENGTHS




Avg Turnaround Time?
(5+10+15)/3 = **10**




Avg Turanround Time?
(13+14+15)/3 = **14**

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

# WORKLOAD ASSUMPTIONS

~~1. Each job runs for the same amount of time~~

~~2. All jobs arrive at the same time~~

~~3. The 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" jobs want fast turnaround (or response time)
- "batch" programs want high throughput

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

Q3 → A

Q2 → B

Q1

Q0 → C → D

"Multi-level"

How to know how to set priority?

Approach 1: nice (static!)
Approach 2: history "feedback"

# HISTORY

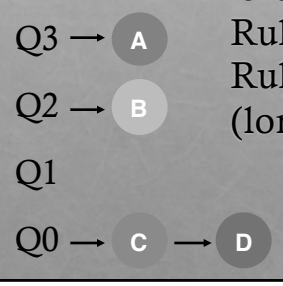
- Use past behavior of process to predict future behavior
  - Common technique in systems
- Processes alternate between I/O and CPU work
- Guess how CPU burst (job) will behave based on past CPU bursts (jobs) of this process

# MORE MLFQ RULES

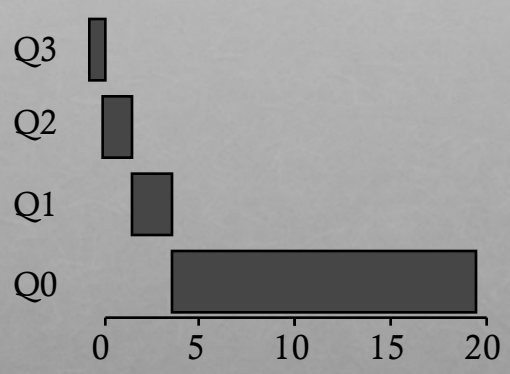
Rule 1: If priority(A) > Priority(B),
A runs

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

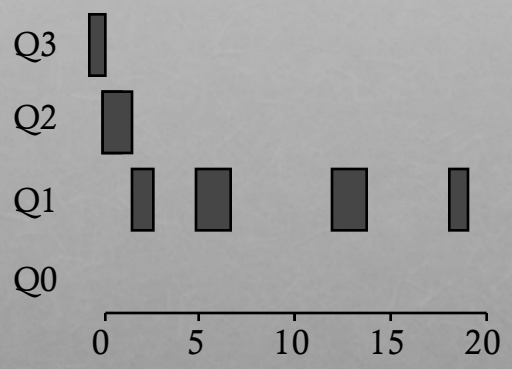
More rules:
Rule 3: Processes start at top priority
Rule 4: If job uses whole slice, demote process
(longer time slices at lower priorities)




# ONE LONG JOB (EXAMPLE)

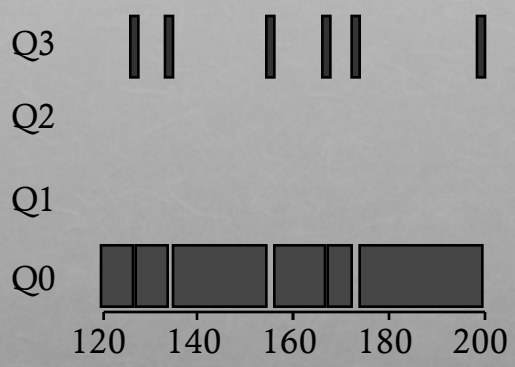


# Job that performs I/O periodically



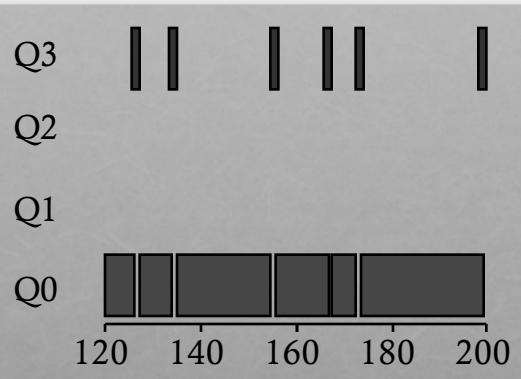

Stays in Q1 queue as long as doesn't use entire Q1 timeslice

# An interactive process joins



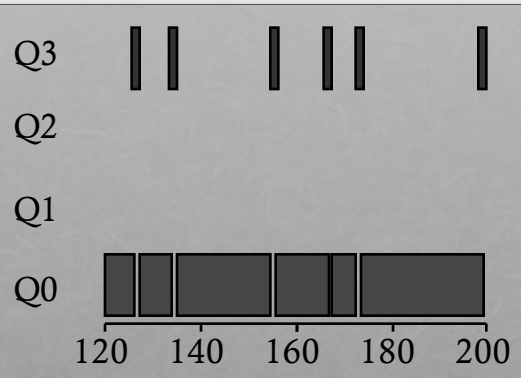

Interactive process never uses entire time slice, so never demoted

# Problems with MLFQ?




Problems
- unforgiving + starvation
- gaming the system

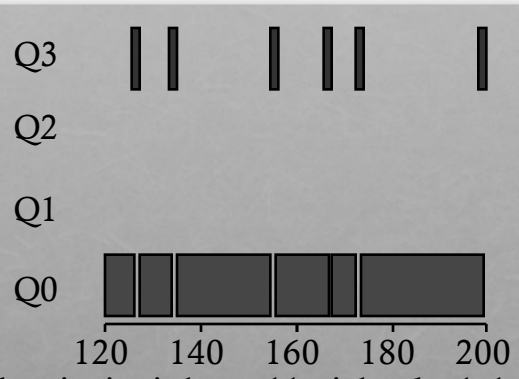
# Prevent Starvation




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 mor
CPU by performing I/O right before time-slice ends

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

# Lottery Scheduling

Goal: proportional (fair) share

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

Amazingly simple to implement

# Lottery Code

```
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t *current = head;
while (current) {
        counter += current->tickets;
        if (counter > winner) break;
        current = current->next;
}
// current is the winner
```

# Lottery Example

```
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t *current = head;
while(current) {
    counter += current->tickets;
    if (counter > winner) break;
    current = current->next;
}
// current gets to run
```

Who runs if winner is:
50
350
0

head → Job A (1) → Job B (1) → Job C (100) → Job D (200) → Job E (100) → null

# OTHER LOTTERY IDEAS

Ticket Transfers

Ticket Currencies

Ticket Inflation

(read more in OSTEP)

# 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.