Reading Reference:
Textbook: Chapter 7

W6 – UNIX PROCESS SCHEDULING

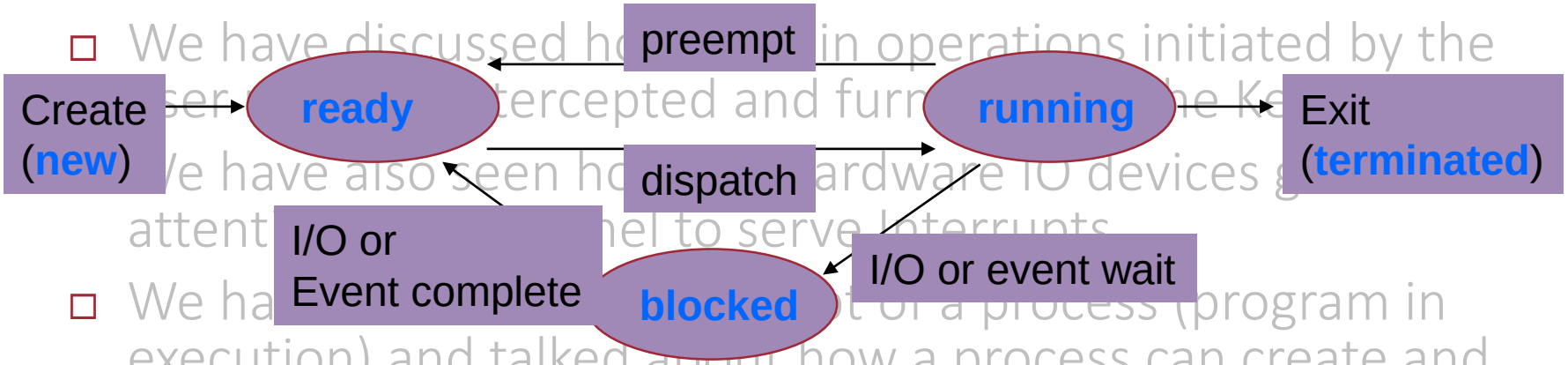
Aakash Tyagi
CSCE 313 Spring 2017

Theme – What we have explored so far

- We have seen how computer and operating systems have evolved, especially in doing multiple things at once (concurrency and parallelism)
- We have talked about the separation of user application space, hardware, and the middle protective layer called Kernel
- We have discussed how certain operations initiated by the user must be intercepted and furnished by the Kernel
- We have also seen how the hardware IO devices get the attention of the Kernel to serve Interrupts
- We have discussed the concept of a process (program in execution) and talked about how a process can create and execute a child process
- Finally, we have seen what a process lifecycle looks like with the different stages a process can be in and its various transition modes

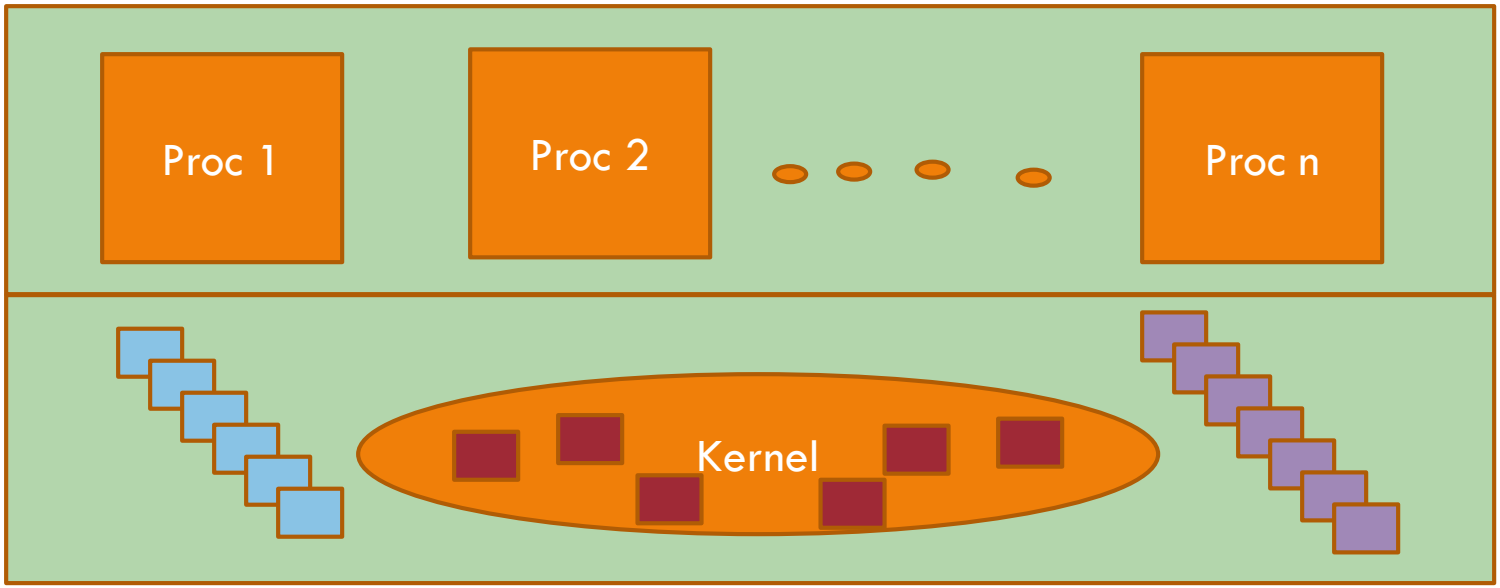
Theme – What we have explored so far

- We have seen how computer and operating systems have evolved, especially in doing multiple things at once (concurrency and parallelism)
- We have talked about the separation of user application space, hardware, and the middle protective layer called Kernel
- We have discussed how [illegible] in operations initiated by the [illegible] intercepted and furn[illegible]he Ke[illegible]
- We have also seen ho[illegible]ardware IO devices g[illegible] attent[illegible]nel to serve interrupts
- We ha[illegible]t of a process (program in execution) and talked about how a process can create and execute a child process
- Finally, we have seen what a process lifecycle looks like with the different stages a process can be in and its various transition modes



Theme – What are we moving on to next?

- Today we will ask how does a Kernel juggle the (often) competing requirements of Performance, Fairness, Utilization, etc. in dealing with concurrency



Outline of the Lecture

- Scheduling policy: what to do next, when there are multiple processes ready to run
  - Or multiple packets to send, or web requests to serve, or...
- Definitions
  - response time, throughput, predictability
- Scheduling policies
  - FIFO, round robin, optimal
  - multilevel feedback queues as approximation of optimal

Adapted from contemporary courses in OS/Systems taught at Berkeley, UW, TAMU, UIUC, and Rice. Special acknowledgment to Profs. Guo/Bettati at TAMU, Joseph at Berkeley, Anderson & Dahlin (Chapter 7)

A Conversation about Scheduling

- You manage a web site, that suddenly becomes wildly popular. Do you?
  - Buy more hardware?
  - Implement a different scheduling policy?
  - Turn away some users? Which ones?
- How much worse will performance get if the web site becomes even more popular?
- When does scheduling become important?
  - Multiple consumers
  - Diverse needs for shared resources
- Consideration must be paid for a number of performance measures
  - **Customer-Centric**: Response Time (wait time + service time)
  - **System-Centric**: Response Time, Fairness, Throughput
- Overarching Goal: Minimize Response time while maximizing throughput

A Conversation about Scheduling

- How do these systems operate in real life –
  - Order counter at a fast food restaurant
  - Water server at a sit-down restaurant
  - Checkout counter at a supermarket

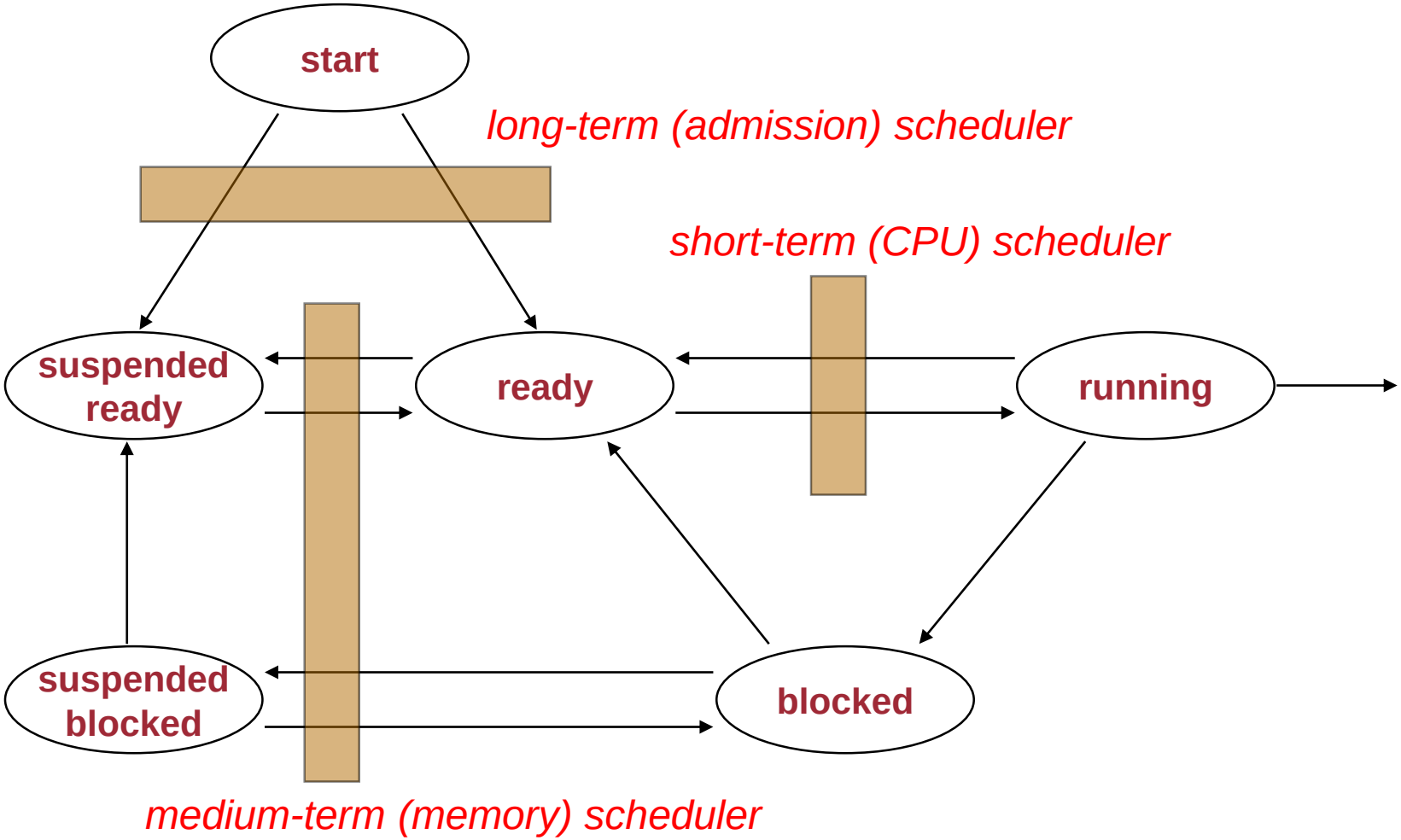
Terms and Definitions

- Task/Job
  - User request: e.g., mouse click, web request, shell command, …
- Latency/response time
  - How long does a task take to complete?
- Throughput
  - How many tasks can be done per unit of time?
- Overhead
  - How much extra work is done by the scheduler?
- Fairness
  - How equal is the performance received by different users?
- Predictability
  - How consistent is the performance over time?

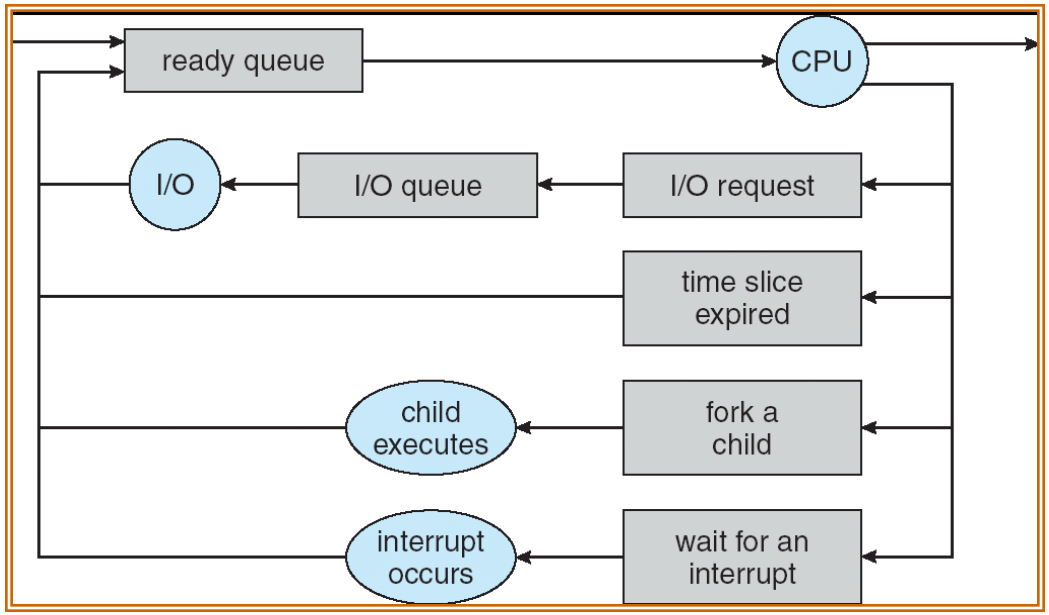
More Terms and Definitions

- Workload
  - Set of tasks for system to perform
- Preemptive scheduler
  - If we can take resources away from a running task
- Work-conserving
  - Resource is used whenever there is a task to run
- Scheduling algorithm
  - takes a workload as input
  - decides which tasks to do first
  - Performance metric (throughput, latency) as output
  - Only preemptive, work-conserving schedulers to be considered

Schedulers


start
long-term (admission) scheduler
short-term (CPU) scheduler
suspended ready
ready
running
suspended blocked
blocked
medium-term (memory) scheduler

CPU Scheduling



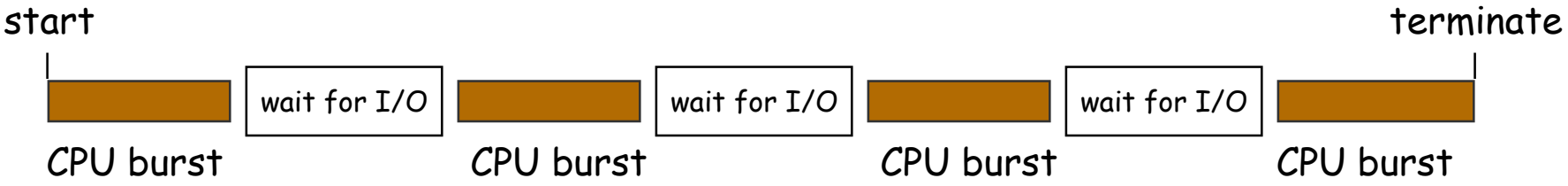
- Question: How is the OS to decide which of several processes to take off a queue?
  - Obvious queue to worry about is ready queue

Scheduling Assumptions

- The high-level goal: Dole out CPU time to optimize some desired parameters of system

Focus: Short-Term Scheduling

- Recall: Motivation for multiprogramming -- have multiple processes in memory to keep CPU busy.
- Typical execution profile of a process:



- CPU scheduler is managing the execution of CPU bursts, represented by processes in ready or running state.

Scheduling Metrics

- Waiting Time: time the job is waiting in the ready queue
  - Time between job's arrival in the ready queue and launching the job
- Service (Execution) Time: time the job is running
- Response (Completion) Time:
  - Time between job's arrival in the ready queue and job's completion
  - Response time is what the user sees:
    - Time to echo a keystroke in editor
    - Time to compile a program

Response Time = Waiting Time + Service Time

- Throughput: number of jobs completed per unit of time
  - Throughput related to response time, but not same thing:
    - Minimizing response time will lead to more context switching than if you only maximized throughput

Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
- Maximize Throughput
  - Two parts to maximizing throughput
    - Minimize overhead (for example, context-switching)
    - Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among processes in some equitable way
  - Fairness is not minimizing average response time

P1: First In First Out (FIFO) or FCFS (First Come First Served)

- Schedule tasks in the order they arrive
  - Continue running them until they complete or give up the processor
- Example: memcached
  - Facebook cache of friend lists, …

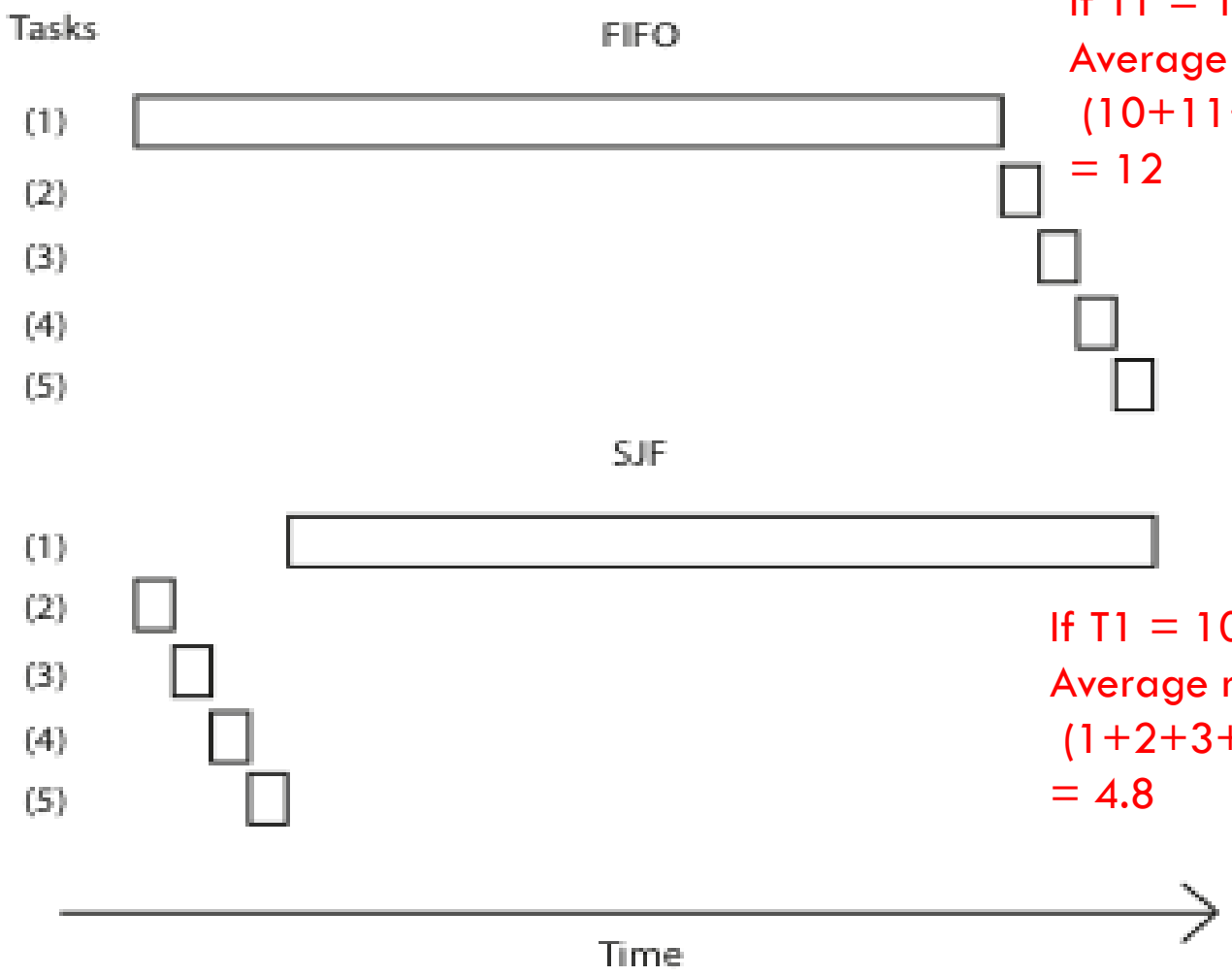
- On what workloads is FIFO particularly bad?
  - *One really long task, remaining tiny tasks. If the long task comes first, the rest would wait.*

P2: Shortest Job First (SJF)

- Always do the task that has the shortest remaining amount of work to do
  - Also called **Shortest Remaining Time First (SRTF)**

- Suppose we have five tasks arrive one right after each other, but the first one is much longer than the others
  - Which completes first in FIFO? Next?
    - *As name implies, first task in will finish first without pre-emption. Next will be the one that came right after, and so on*
  - Which completes first in SJF? Next?
    - *The shortest task always finishes first. Next shortest task will finish second, and so on*

FIFO vs. SJF – Example showing extremes



If T1 = 10, T2..T5 = 1
Average response time = (10+11+12+13+14)/5 = 12

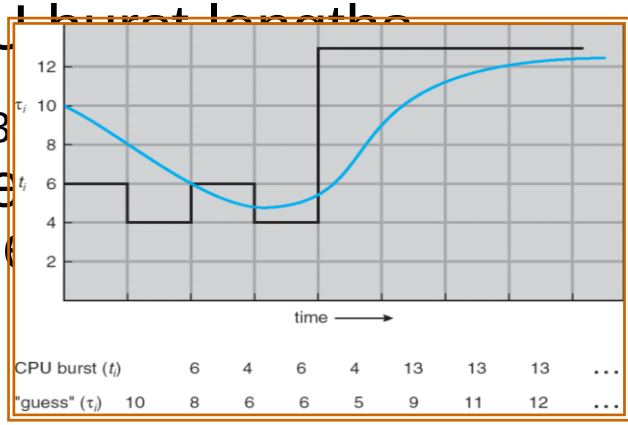
If T1 = 10, T2..T5 = 1
Average response time = (1+2+3+4+14)/5 = 4.8

Shortest Job First

- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    - When you submit a job, have to say how long it will take
    - To stop cheating, system kills job if takes too long
  - But: even non-malicious users have trouble predicting runtime of their jobs
- Claim: SJF is optimal for average response time
  - *Why? SJF always picks the shortest job; if it did not, then by definition it would result in higher average response time. <<see notes for details>>*
- For what workloads is FIFO optimal?
  - *Why? FIFO is optimal for jobs that have identical characteristics in which case it does not matter who goes first.*
- Does SJF have any downsides?
  - *Yes, SJF can lead to starvation because longer jobs would never get any allocated resources. Imagine a supermarket that implemented SJF!*

Predicting the Length of the Next CPU Burst

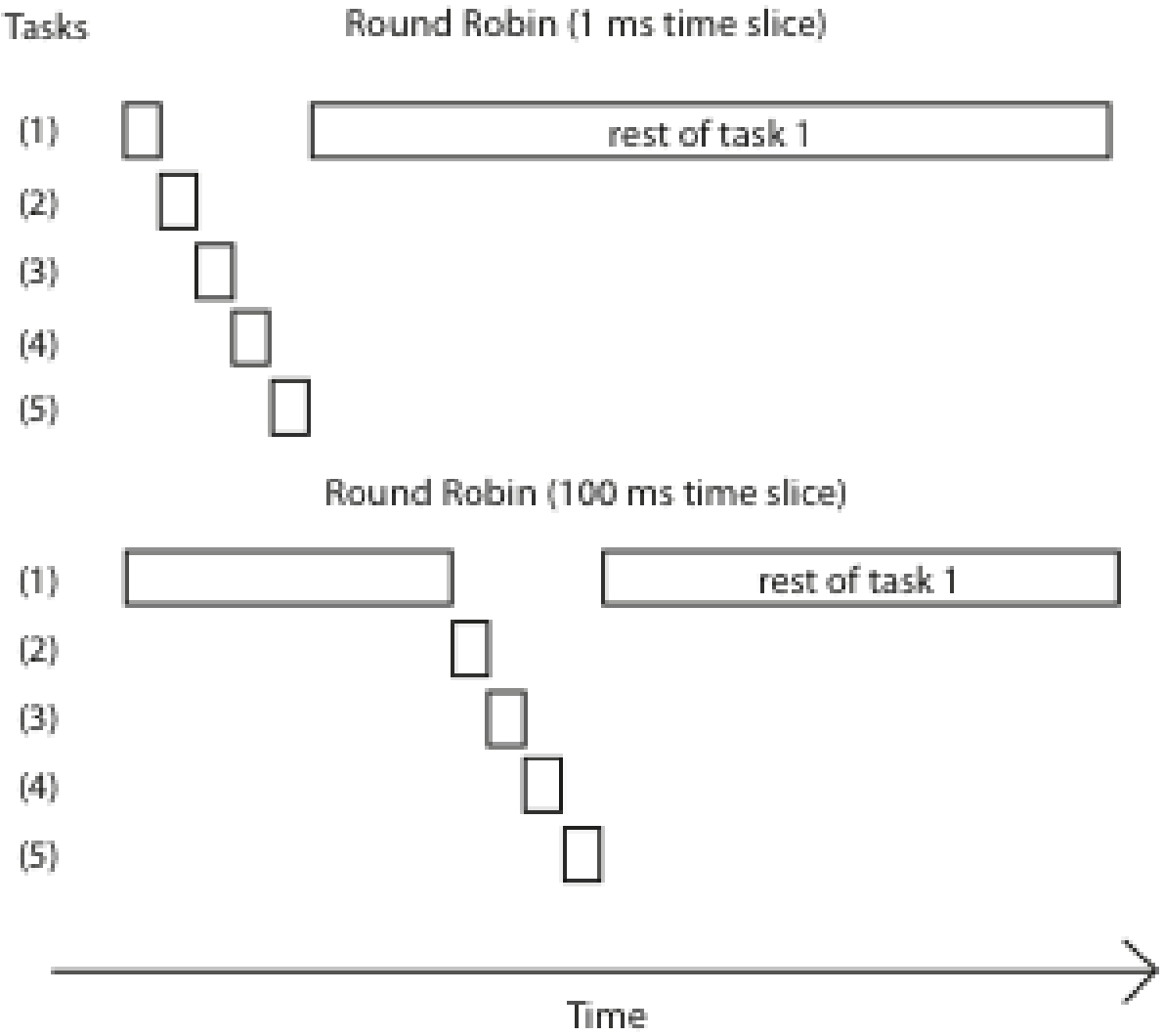
- Adaptive: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc.
  - Works because programs have predictable behavior
    - If program was I/O bound in past, likely in future
    - If computer behavior were random, wouldn't help
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts:
    Let t_{n-1}, t_{n-2}, t_{n-3}, etc. be previous CPU burst lengths.
    Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}$
  - Function f could be one of many diffe
    estimation schemes (Kalman filters,



P3: Round Robin

- Each task gets resource for a fixed period of time (time quantum)
  - If task doesn't complete, it goes back in line
- Need to pick a time quantum
  - What if time quantum is too long?
    - Infinite?
      - *Then it will be equivalent to FCFS or FIFO*
  - What if time quantum is too short?
    - One instruction?
      - Too much overhead of swapping processes

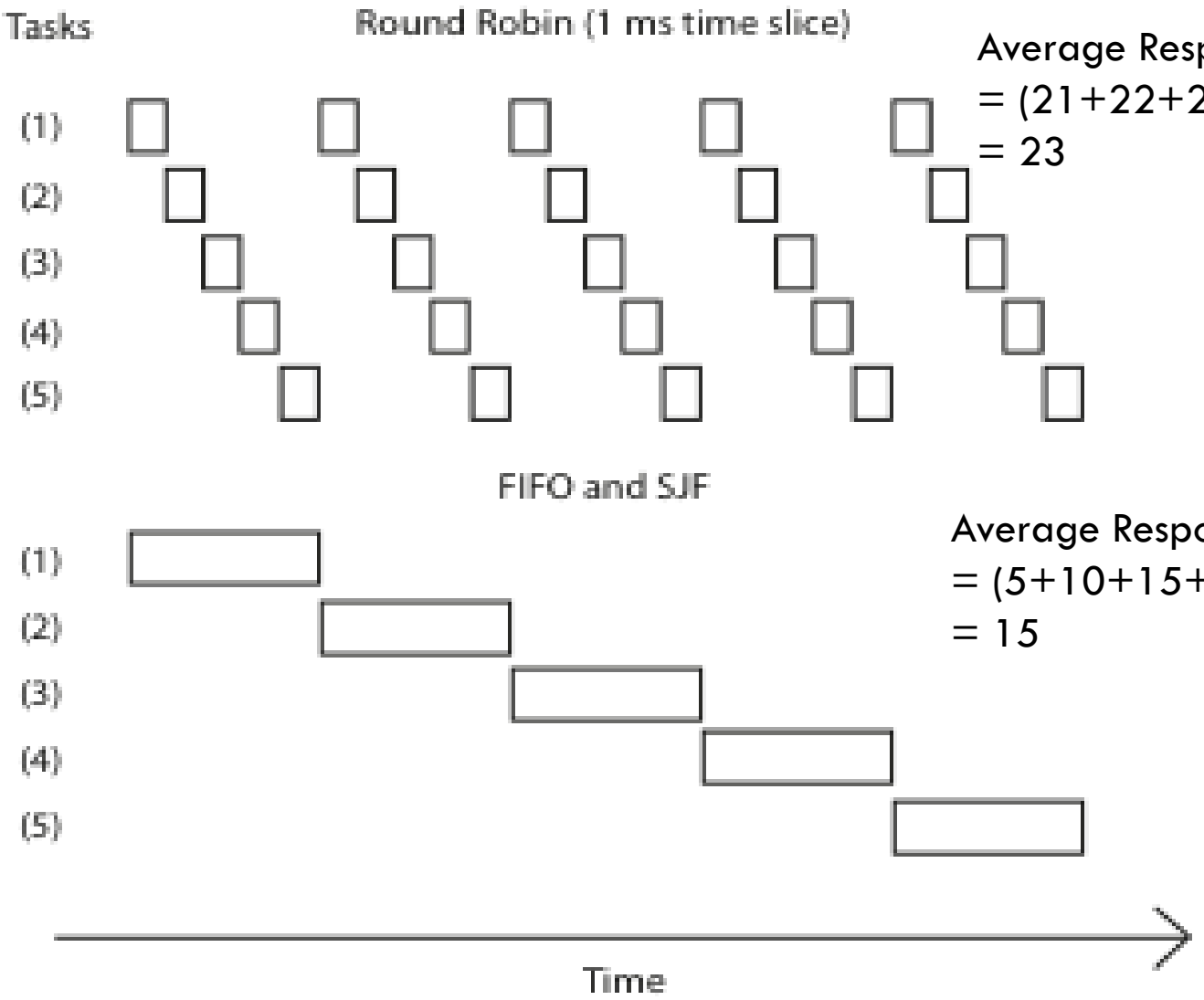
Round Robin


Tasks
Round Robin (1 ms time slice)
(1)
rest of task 1
(2)
(3)
(4)
(5)
Round Robin (100 ms time slice)
(1)
rest of task 1
(2)
(3)
(4)
(5)
Time

Round Robin vs. FIFO

- Assuming zero-cost time slice, is Round Robin always better than FIFO?
  - *No. Round robin is better for short jobs, but it is poor for jobs that are the same length.*
- What's the worst case for Round Robin?
  - CPU devoted to Overhead

Round Robin vs. FIFO



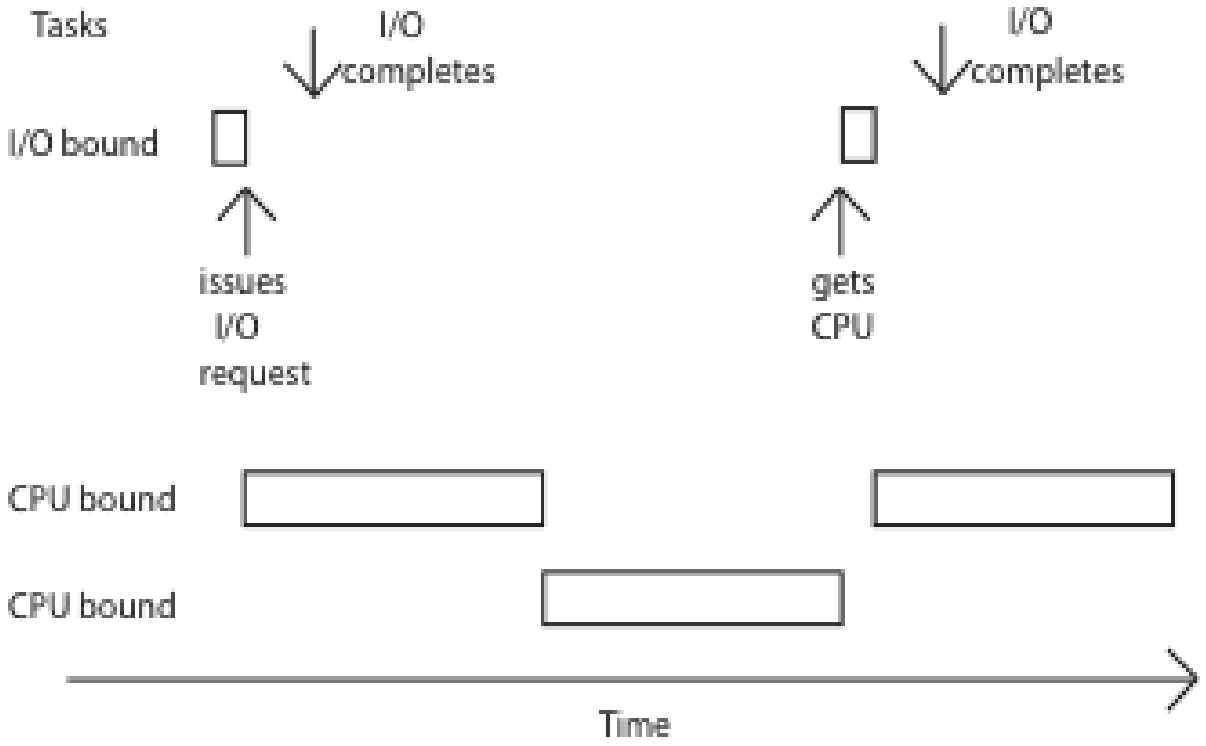
Average Response Time
= (21+22+23+24+25)/5
= 23

Average Response Time
= (5+10+15+20+25)/5
= 15

Round Robin vs. Fairness

- Is Round Robin always fair?
  - *round robin ensures we don't starve, and gives everyone a turn, but lets short tasks complete before long tasks*

Mixed Workload



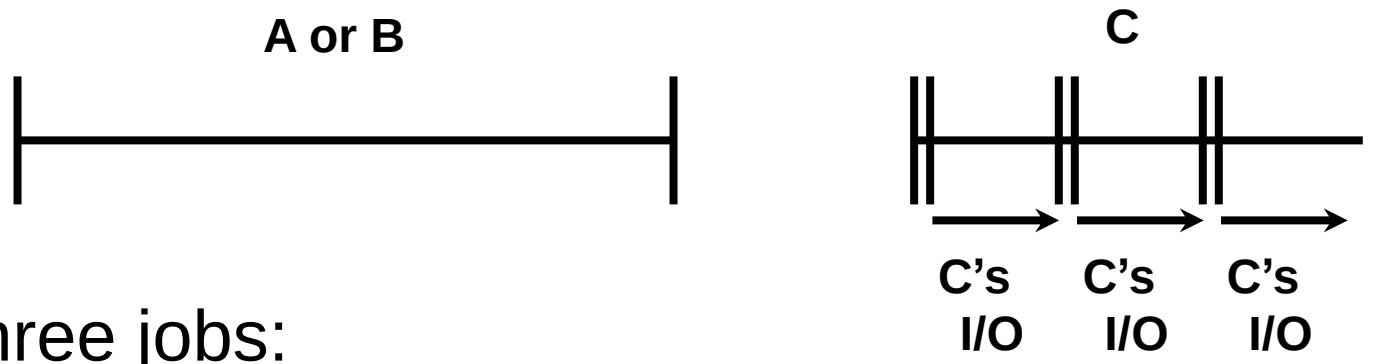
- I/O task has to wait its turn for the CPU, and the result is that it gets a tiny fraction of the performance it could get.
- We could shorten the RR quantum, and that would help, but it would increase overhead.
- What would this do under SJF
  - Every time the task returns to the CPU, it would get scheduled immediately!

Discussion

- SJF is the best you can do at minimizing average response time
  - Provably optimal

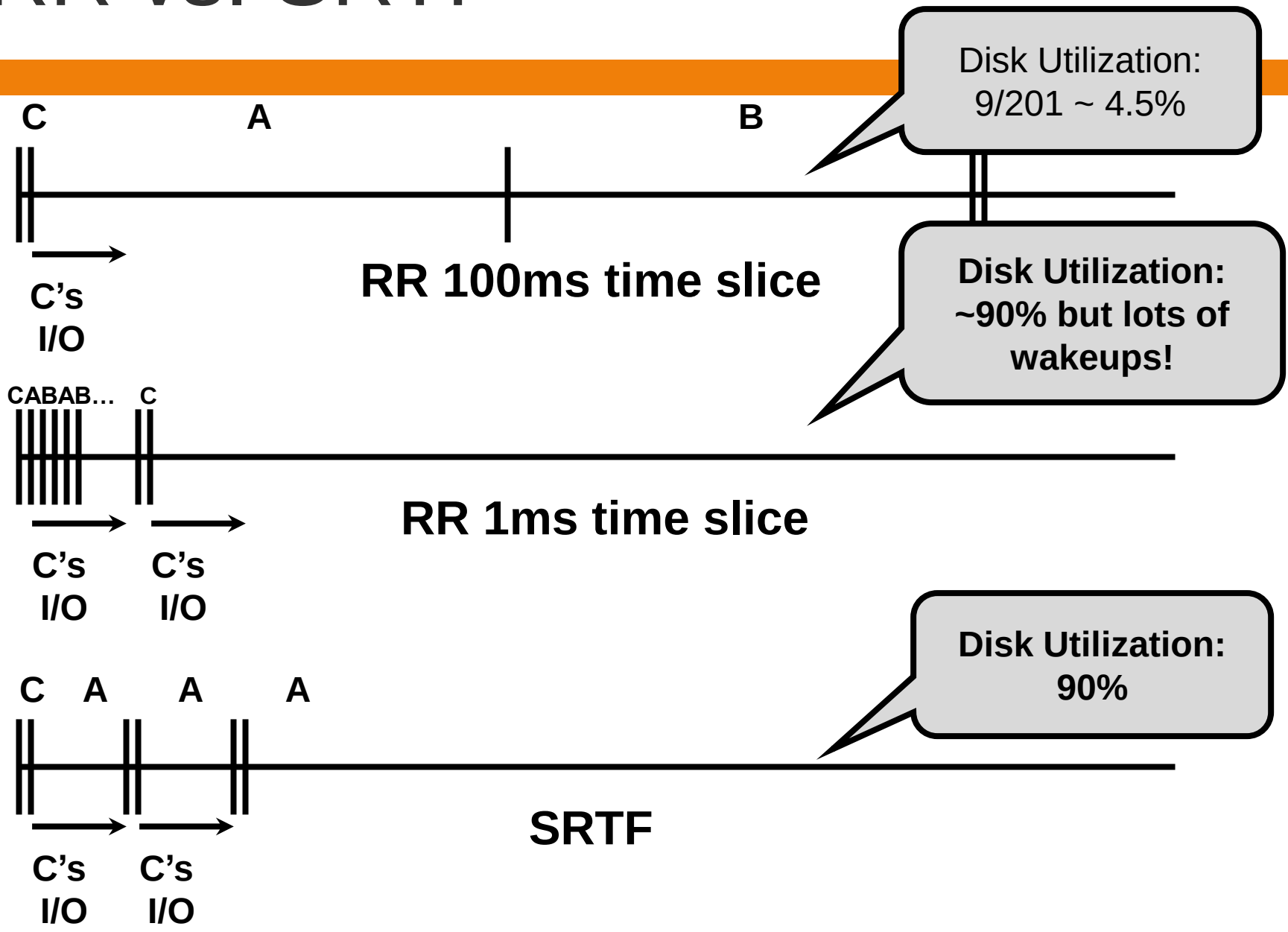
- Comparison of SJF with FCFS and RR
  - What if all jobs the same length?
    - SJF becomes the same as FCFS (i.e., FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    - SJF (and RR): short jobs not stuck behind long ones

Example to illustrate benefits of SRTF

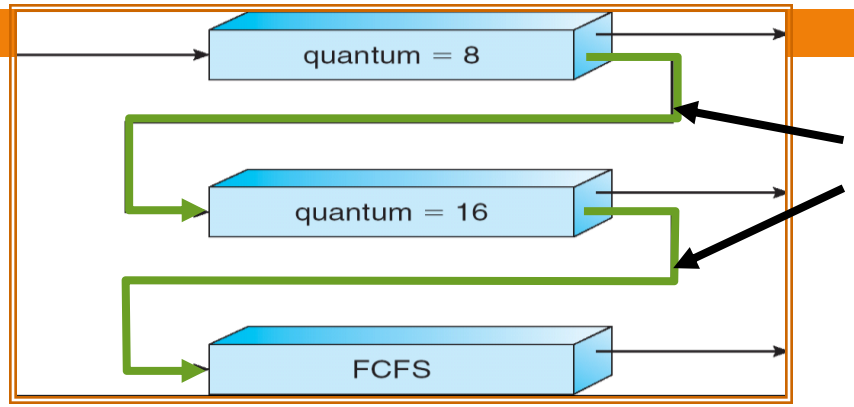


- Three jobs:
  - A,B: CPU bound, each run for a week
    C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B use 100% of the CPU
- With FIFO:
  - Once A or B get in, keep CPU for one week each
- What about RR or SRTF?
  - Easier to see with a timeline

RR vs. SRTF


C
A
B
Disk Utilization:
9/201 ~ 4.5%
C's
I/O
RR 100ms time slice
Disk Utilization:
~90% but lots of
wakeups!
CABAB…
C
C's
I/O
C's
I/O
RR 1ms time slice
C
A
A
A
Disk Utilization:
90%
C's
I/O
C's
I/O
SRTF

Multi-Level Feedback Scheduling



- Another method for exploiting past behavior
  - First used in Cambridge Time Sharing System (CTSS)
  - Multiple queues, each with different priority
    - Higher priority queues often considered “foreground” tasks
  - Each queue has its own scheduling algorithm
    - e.g., foreground – RR, background – FCFS
    - Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc.)
- Adjust each job’s priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level

Scheduling Details

- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top

- Scheduling must be done between the queues
  - Fixed priority scheduling:
    - Serve all from highest priority, then next priority, etc.
  - Time slice:
    - Each queue gets a certain amount of CPU time
    - e.g., 70% to highest, 20% next, 10% lowest

Countermeasure

- Countermeasure: user action that can foil intent of the OS designer
  - For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
  - Of course, if everyone did this, wouldn't work!

- Ex: MIT Othello game project (simpler version of Go game)
  - Computer playing against competitor's computer, so key was to do computing at higher priority the competitors.
    - Cheater put in printf's, ran much faster!

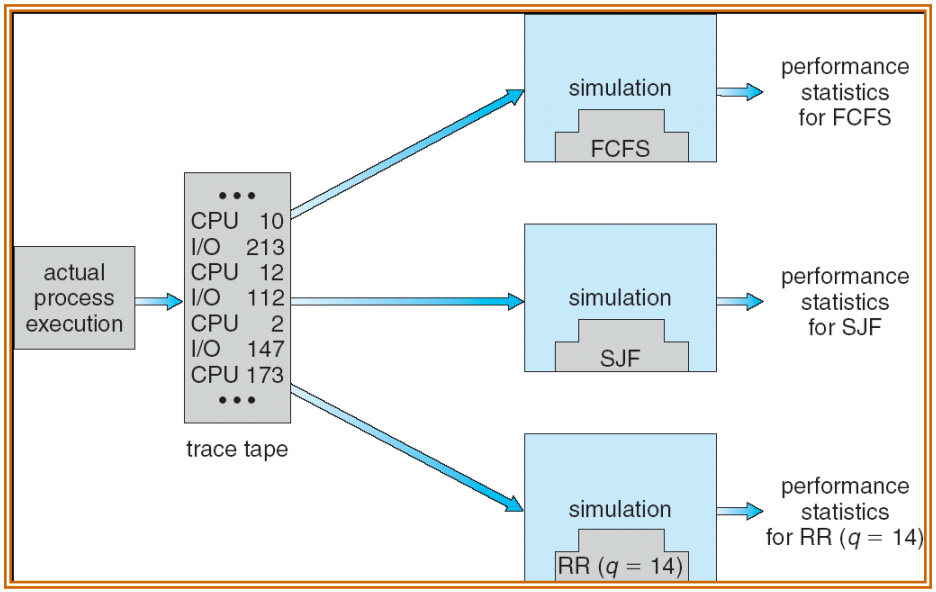
Scheduling Fairness

- What about fairness?
  - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    - Long running jobs may never get CPU
    - In Multics, shut down machine, found 10-year-old job
  - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
  - Tradeoff: fairness gained by hurting average response time!

- How to implement fairness?
  - Could give each queue some fraction of the CPU
    - What if one long-running job and 100 short-running ones?
    - Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
  - Could increase priority of jobs that don't get service
    - What is done in UNIX
    - This is ad hoc—what rate should you increase priorities?

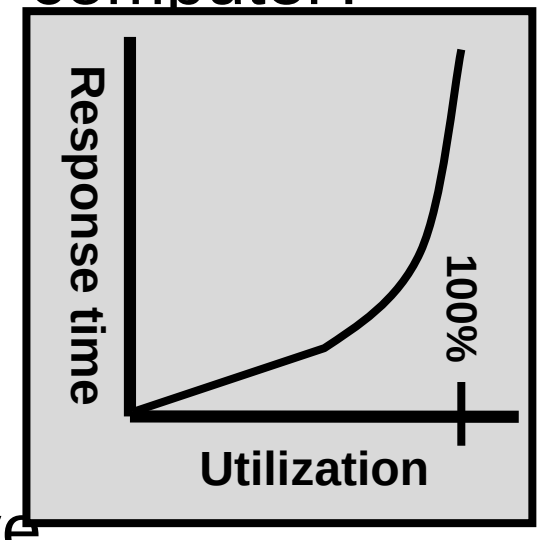
How to Evaluate a Scheduling algorithm?

- Deterministic modeling
  - Takes a predetermined workload and compute the performance of each algorithm for that workload
- Queuing models
  - Mathematical approach for handling stochastic workloads
- Implementation/Simulation:
  - Build system which allows actual algorithms to be run against actual data. Most flexible/general.



A Final Word On Scheduling

- When do the details of the scheduling policy and fairness really matter?
  - When there aren't enough resources to go around
- When should you simply buy a faster computer?
  - (Or network link, or expanded highway, or …)
  - One approach: Buy it when it will pay for itself in improved response time
    - Assuming you're paying for worse response time in reduced productivity, customer angst, etc…
    - Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization⇒100%



- An interesting implication of this curve:
  - Most scheduling algorithms work fine in the "linear" portion of the load curve, fail otherwise
  - Argues for buying a faster X when hit "knee" of curve

Summary

- FCFS is simple and minimizes overhead.
- If tasks are variable in size, then FCFS can have very poor average response time.
- If tasks are equal in size, FCFS is optimal in terms of average response time.
- Considering only the processor, SJF is optimal in terms of average response time.
- SJF is pessimal in terms of variance in response time.

Summary (contd.)

- If tasks are variable in size, Round Robin approximates SJF.
- If tasks are equal in size, Round Robin will have very poor average response time with short time slices.
- Tasks that intermix processor and I/O benefit from SJF and can do poorly under Round Robin.
- Round Robin avoids starvation.
- By manipulating the assignment of tasks to priority queues, an MFQ scheduler can achieve a balance between responsiveness, low overhead, and fairness.