Reading Reference:

OSPP Book: Chapter 7 (here is the link of this chapter)

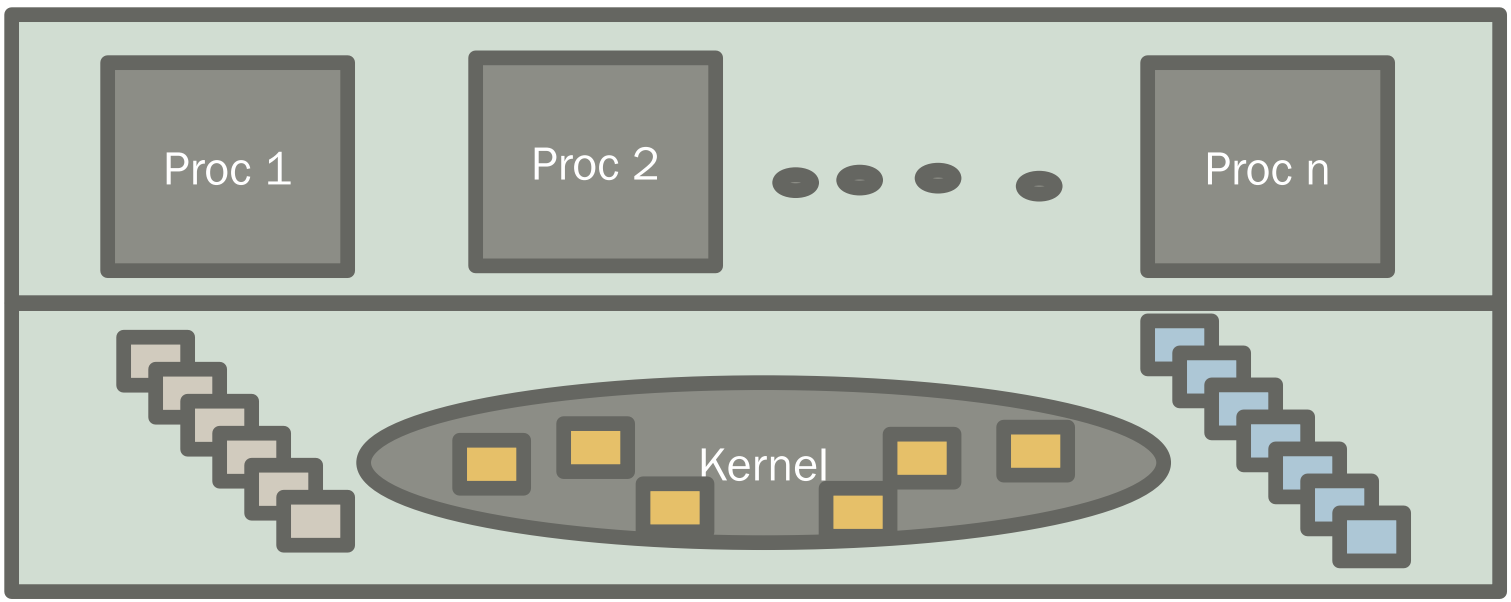
UNIX PROCESS SCHEDULING

Tanzir Ahmed

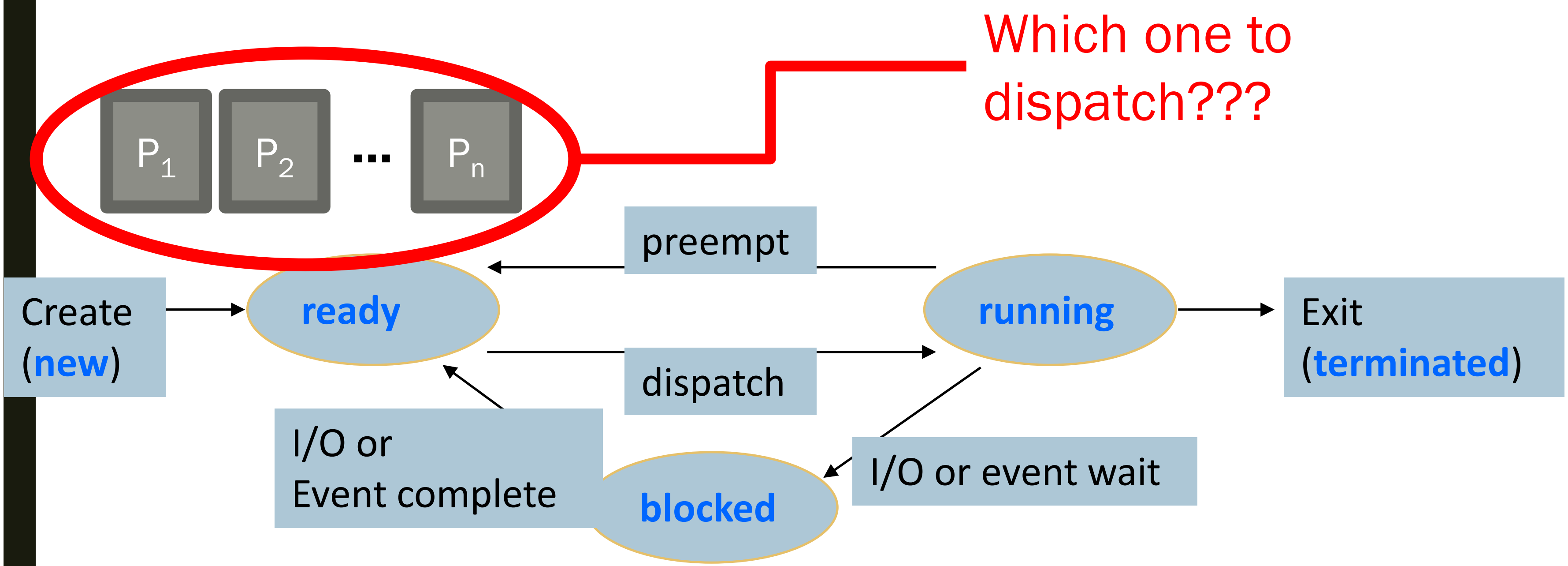
CSCE 313 Spring 2021

Process Scheduling

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

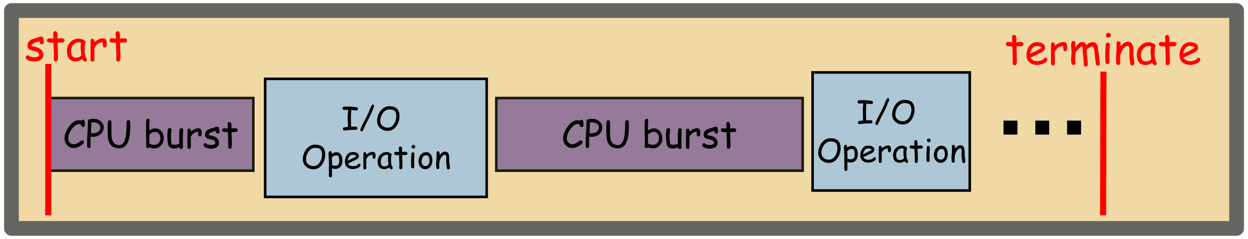


Process Scheduling – More Specifically



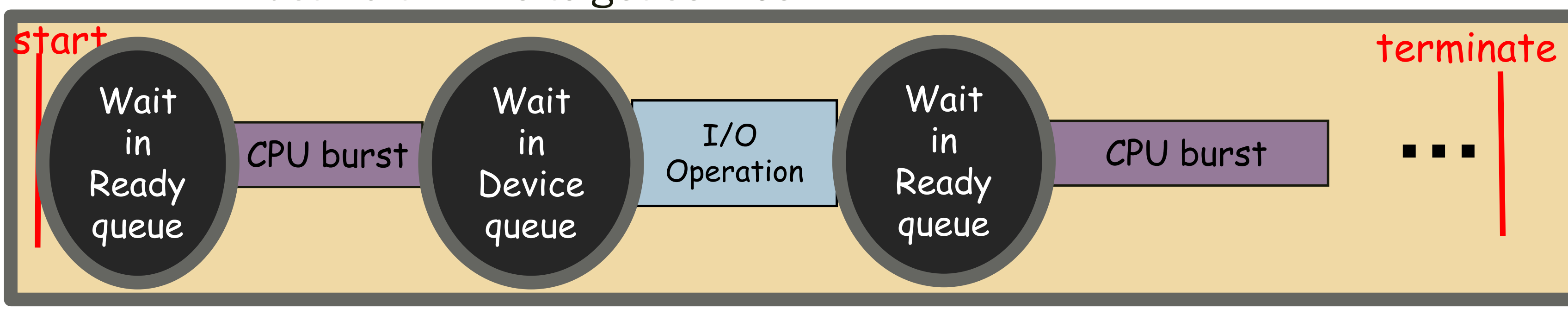
Overall Idea

If there were no other processes, a process's life would look like the following:



However, since there are other processes, each process is slowed down by waits in Ready and Blocked queues:

- *Think of CPU and I/O devices as servers in the bank, where people must wait in line to get service*



Now, the scheduler's goal is to minimize these Wait times in some aggregate sense, especially in the Ready queue

- *Our focus is CPU scheduler*
- *There are other schedulers (e.g., I/O schedulers) are not our focus*

Some Thoughts – Why the Reality has Hope

- Based on which queue a process spends most of its time, we can classify processes to 2 types:
- I/O-bound Processes
  - *Spend most time in blocked state as the I/O device performs I/O*
  - *Example: A web-server reading requests from network (I/O), reading files from disk (I/O), writing response back to network (I/O), and some request parsing (CPU)*
- CPU-bound Processes
  - *Spend most time in ready queue or as running*
  - *Example: A program computing large prime numbers – a lot of computations involving CPU, but no/less I/O*

Scheduling Metrics

- 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?*

How to Measure?

- **Waiting Time:** Time spent in Ready queue
  - *In other words, time between job's arrival in the Ready (or Blocked) queue and start of service*
- **Service (Execution) Time:** Time spent in CPU (or I/O device)
- **Response (Completion) Time:**

$$Response\ Time\ =\ Finish\ Time\ -\ Arrival\ Time$$

  - *Response time is what the user experiences:*
    - Time to echo a keystroke in editor
    - Time to compile a program
  - *Another view of Resp Time is the time it waits in line plus the time it is processed:*

$$Response\ Time\ =\ Waiting\ Time\ +\ Service\ Time$$

- Because of timer, a process can be kicked out of CPU many times and each time it wait in the Ready queue behind other processes

$$Response\ Time\ = \sum Waiting\ Time\ + \sum Service\ Time$$

- **Throughput:** number of jobs completed per unit of time
  - *The more the device utilization, the higher the throughput*
  - *Throughput related to response time, but not same thing:*
    - Minimizing response time and maximizing throughput can be contradictory (examples coming up)

Scheduling Metrics and Goals

- Scheduling metrics are important for both individual processes and the system as a whole
  - ***Customer-Centric***: *Response Time*
  - ***System-Centric***: *Throughput , Average Response Time, Fairness*
- Scheduling Goals

1. Minimize **Average Response Time (ART)**
   - *Measure from the response time of several jobs*
   - *Also applies to 1 job as – i.e., we want to minimize the response time for processing a mouse click, a file copy, to compile a program etc.*
2. **Maximize Throughput**
   - *By keeping CPU and I/O devices as utilized as possible*
3. Ensure **Fairness**
   - *Share CPU in some equitable way*
   - *This policy can contradict with others (we will see examples)*

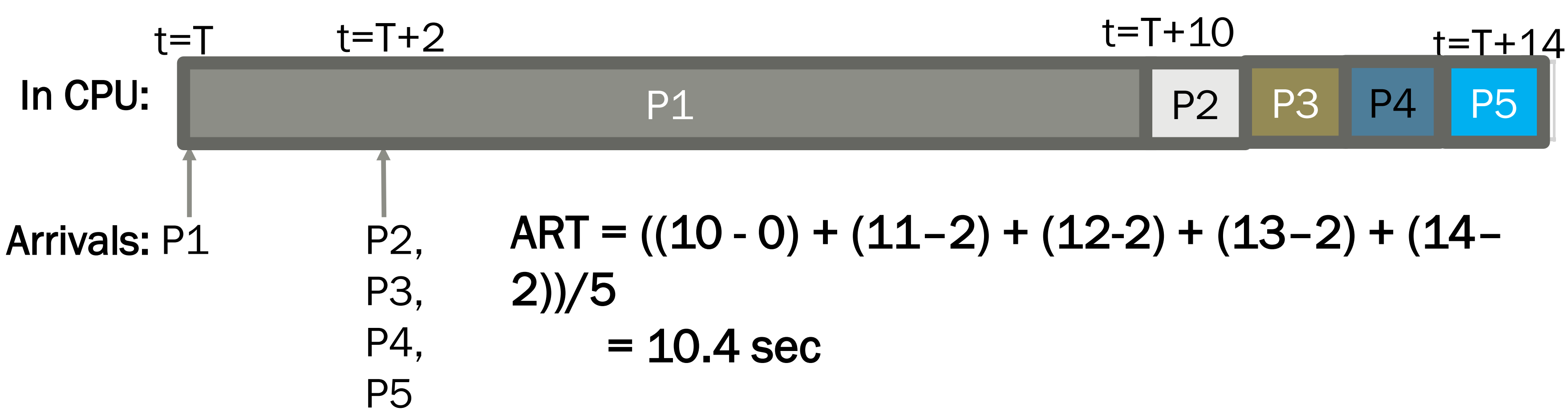
Scheduling: Decision Points

- There can be 3 points in time when the scheduler runs:
  - *The current process voluntarily gives up (by requesting I/o) CPU*
  - *Timer expired - the current process is kicked out*
  - ***Arrival of a new process (new!!)***
- A scheduler can be preemptive or not (related to the above):
  - ***Preemptive:*** *scheduler runs as soon as new arrival*
  - ***Non-preemptive:*** *scheduler waits until timer expires, or the current process yields CPU*
- Some scheduling algorithms can run in both preemptive and non-preemptive mode
  - *SRTF can be run in both preemptive or non-preemptive manner. FIFO would be the same with or w/o preemption*
  - *Round Robin is non-preemptive by definition*

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*
- Uses:
  - *Read/write requests to a disk file*
  - *Network packets in a NIC*
- An Example:

Job	Arrival Time (sec)	Service Time (sec)
P1	T	10
P2	T+2	1
P3	T+2	1
P4	T+2	1
P5	T+2	1



ART = ((10 - 0) + (11–2) + (12-2) + (13–2) + (14–2))/5

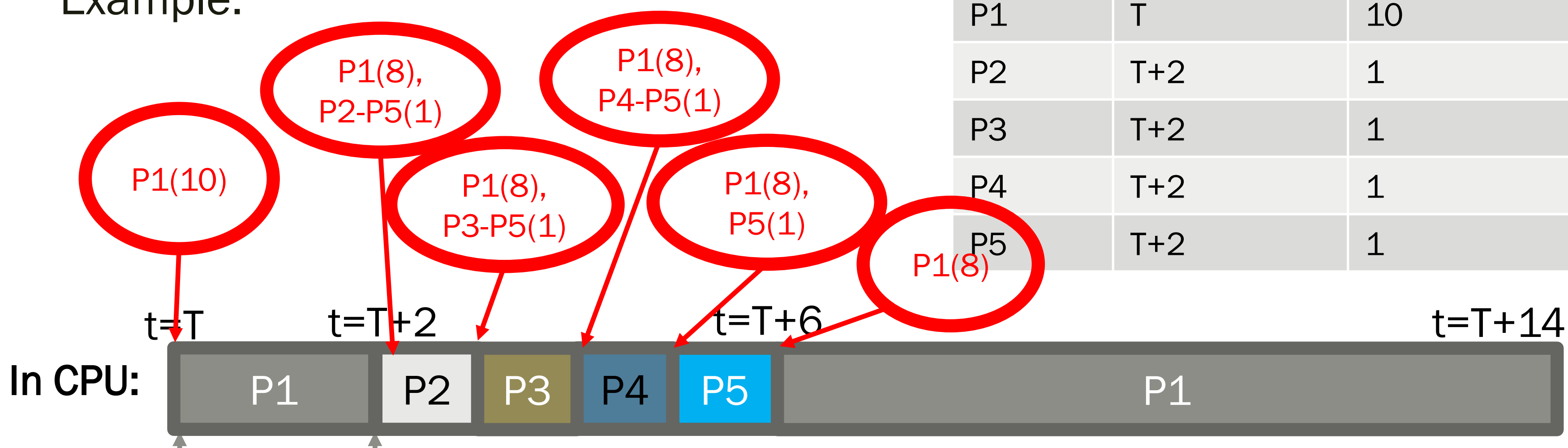
= 10.4 sec

P2: **S**hortest **R**emaining **T**ime **F**irst (SRTF)

- Always do the task that has the shortest remaining amount of work to do
  - *Aka. **S**hortest **J**ob **F**irst (SJF)*
  - *Note: Remaining time of a job changes every time it gets service in CPU*

Job	Arrival Time (sec)	Service Time (sec)
P1	T	10
P2	T+2	1
P3	T+2	1
P4	T+2	1
P5	T+2	1

Example:



Arrivals: P1

P2,
P3,
P4,

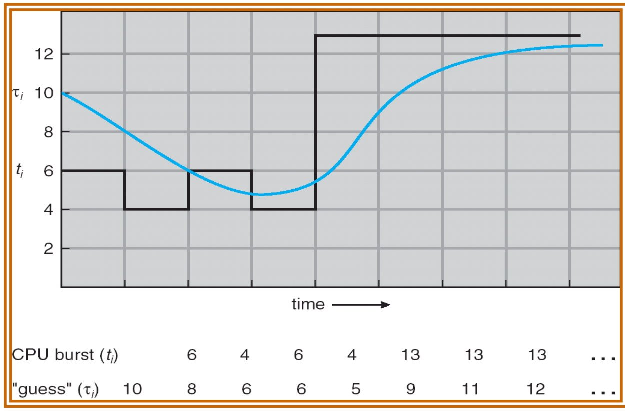
ART = ((14 - 0) + (3 – 2) + (4 - 2) + (5 – 2) + (6 – 2))/5 = 4.8 sec

Some Thoughts

- Claim: SRTF is optimal for ART
  - *SRTF always picks the shortest job; if it did not, then, by definition, it would result in higher average response time. <<see notes for details>>*
- When is FIFO optimal?
  - *When all jobs are equal in length*
  - *SRTF gives the same schedule, but incurs many context switches*
- Does SRTF have any downsides?
  - ***Starvation:** longer jobs would suffer. Imagine a supermarket that implements SRTF in checkout lines! Longer lines will keep waiting*
  - ***Implementation:*** No exact algorithm to implement SRTF *(how would you know how much is remaining???)*

Practical Implementation of SRTF

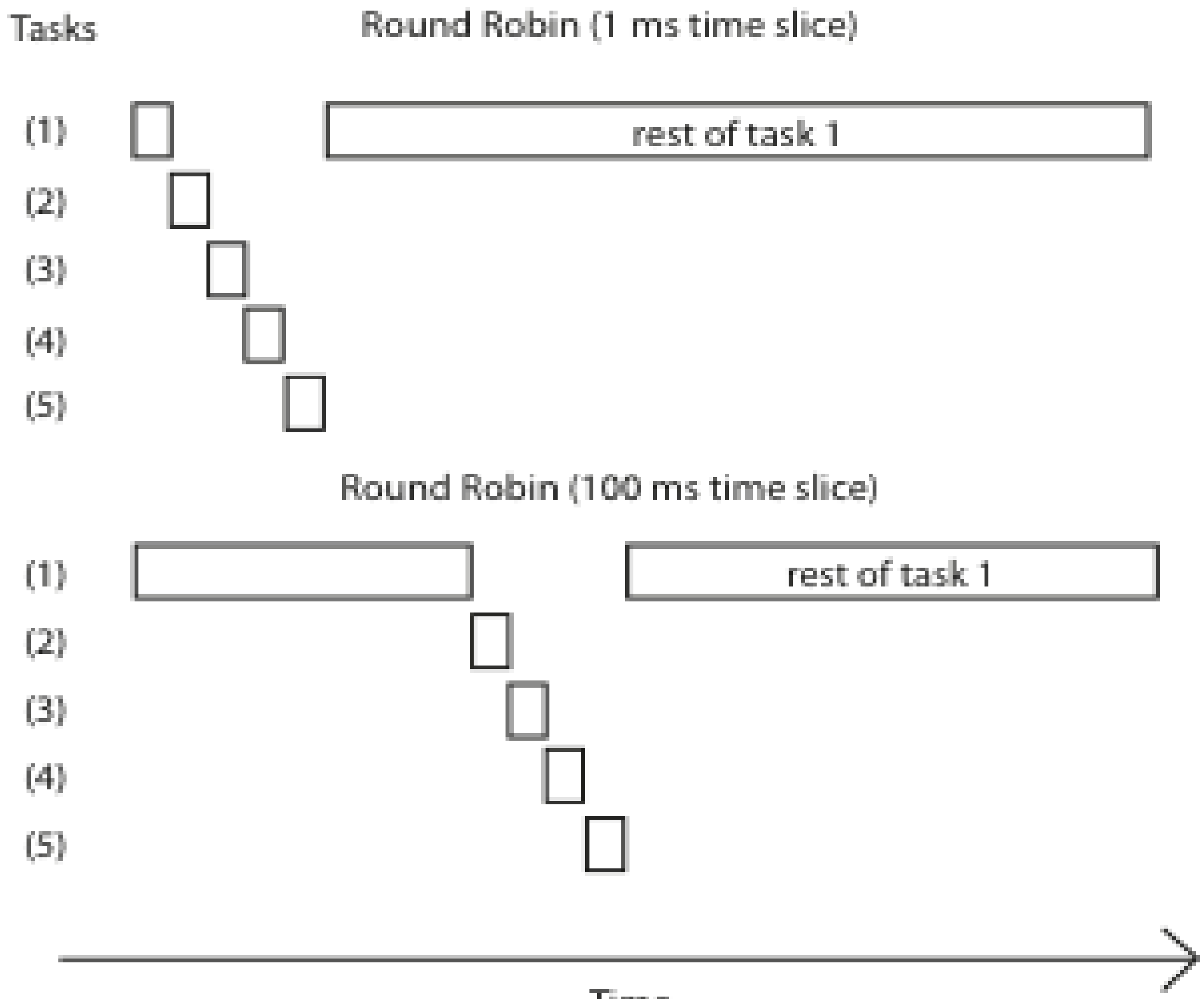
- Issue: How do we know the remaining time?
  - *User provides job runtime*
    - System kills job if takes too long (i.e., to stop cheating)
  - *But even for non-malicious users, it is hard to predict runtime accurately*
- Adaptive Algorithm without user input: Predict the next **CPU burst** (not the whole task length) based on the recent past
  - *Works because programs have predictable behavior*
    - If program was I/O bound in past, probably it will be in future
- 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 $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots)$*
  - *Function f could be one of many different time series estimation schemes (Kalman filters, etc.)*



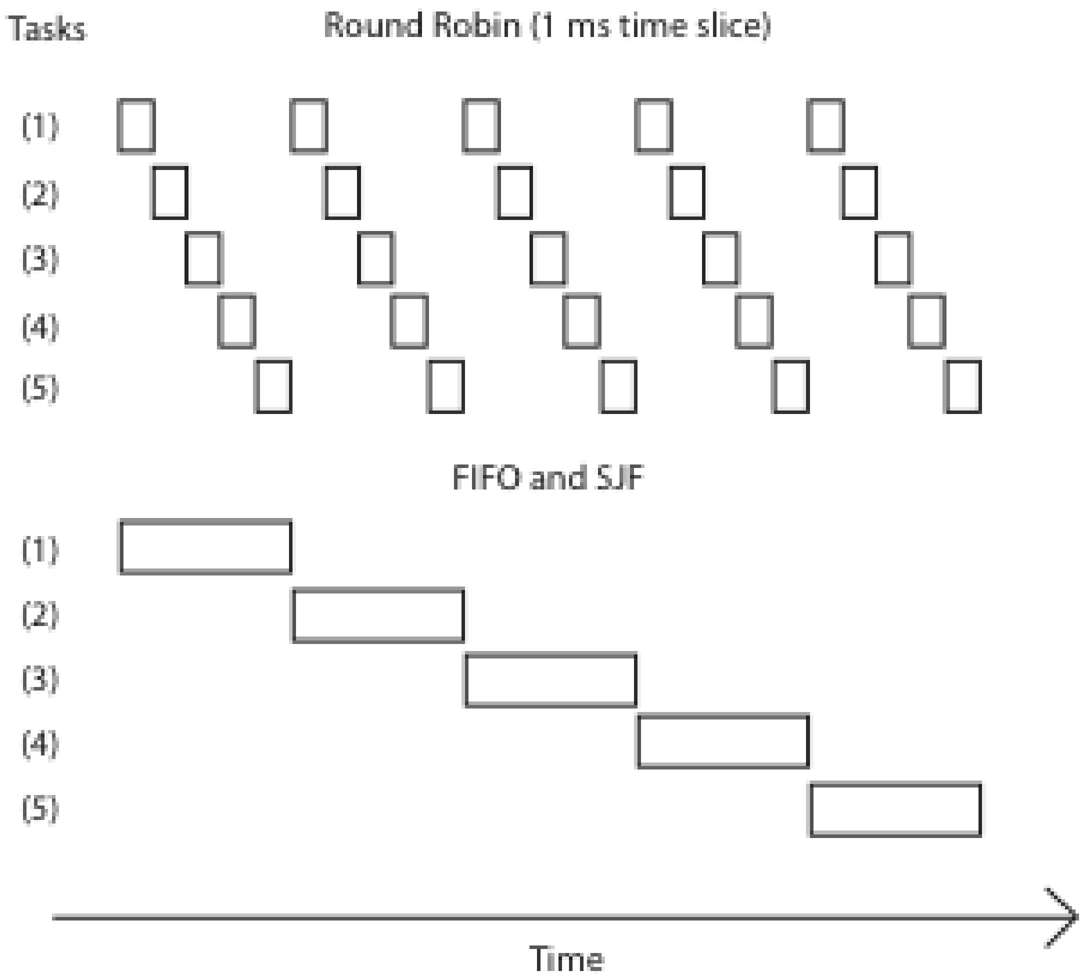
P3: Round Robin

- Each task gets resource for a fixed **time quantum**
  - *If task doesn't complete, it goes back in line*
  - *If it finishes the CPU burst because of I/O, it gets out before the quantum expires*
- Now, we need a timer, right???
  - *So far, we have been operating w/o a timer*
- But, **how to we choose a good time quantum???**
  - *<u>Too long (i.e., Infinite)</u>?*
    - *Then it will be equivalent to FIFO (as if there is no timer and no preemption)*
  - *<u>Too short (i.e. One instruction)</u>?*
    - *Too much overhead of swapping processes*
    - *But tasks finish approximately in the order of their length (approximating SRTF)*
- Thus, RR is sort of a compromise between FIFO and SRTF

Round Robin – Varying Time Slice



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. FIFO

- Even after ignoring context-switch overhead, Round Robin can be worse than FIFO in cases like the above (i.e., equal jobs):
  - *Round robin is better when there is a mix of short and long jobs. However, it is poor for jobs that are the same length*
  - *Of course, context switches are not zero-cost, adding more overhead to RR*
- However, Round Robin is **fair**
  - *Everyone gets equal share of CPU per unit of time*
  - *Of course, shorter tasks finish sooner because they need fewer turns*

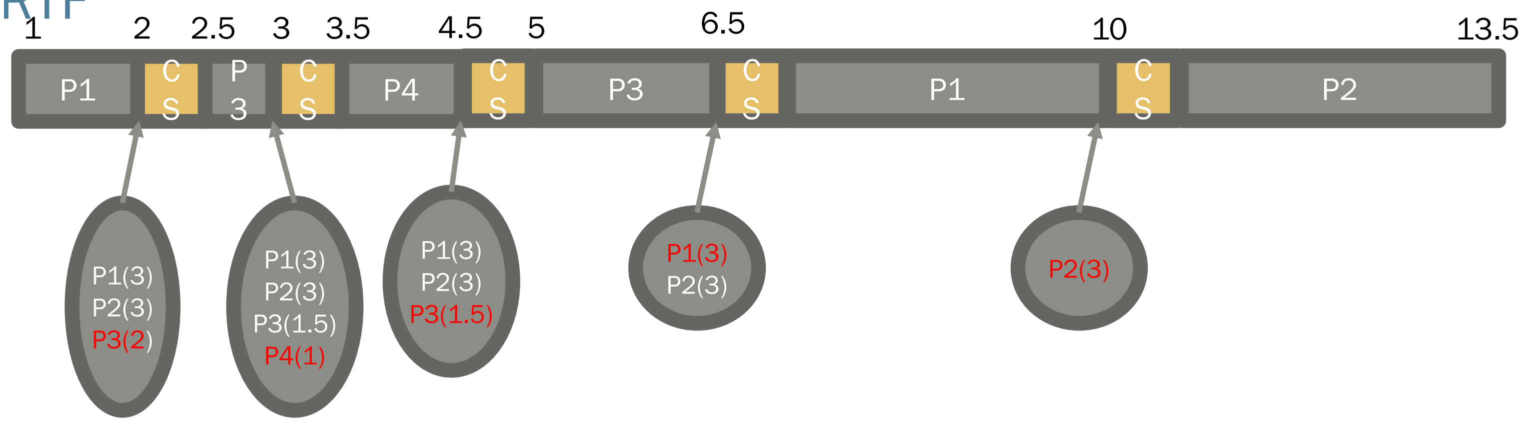
Assuming 0.5 sec overhead per context switch (i.e., time between 2 user processes) in a 1-CPU-1-core system, schedule the following workload and compute Average Response Time (ART) under:

a. Shortest Remaining Time First (SRTF) (preemptive)

b. Round-Robin (RR) with time quantum=2sec

Process	Arrival Time	Service Time
P1	1	4
P2	2	3
P3	2	2
P4	3	1

- SRTF



- ART = ((10-1) + (13.5 – 2) + (6.5 – 2) + (4.5-3))/4 = **26.5/4**

Assuming 0.5 sec overhead per context switch (i.e., time between 2 user processes) in a 1-CPU-1-core system, schedule the following workload and compute Average Response Time (ART) under:

a. Shortest Remaining Time First (SRTF) (preemptive)

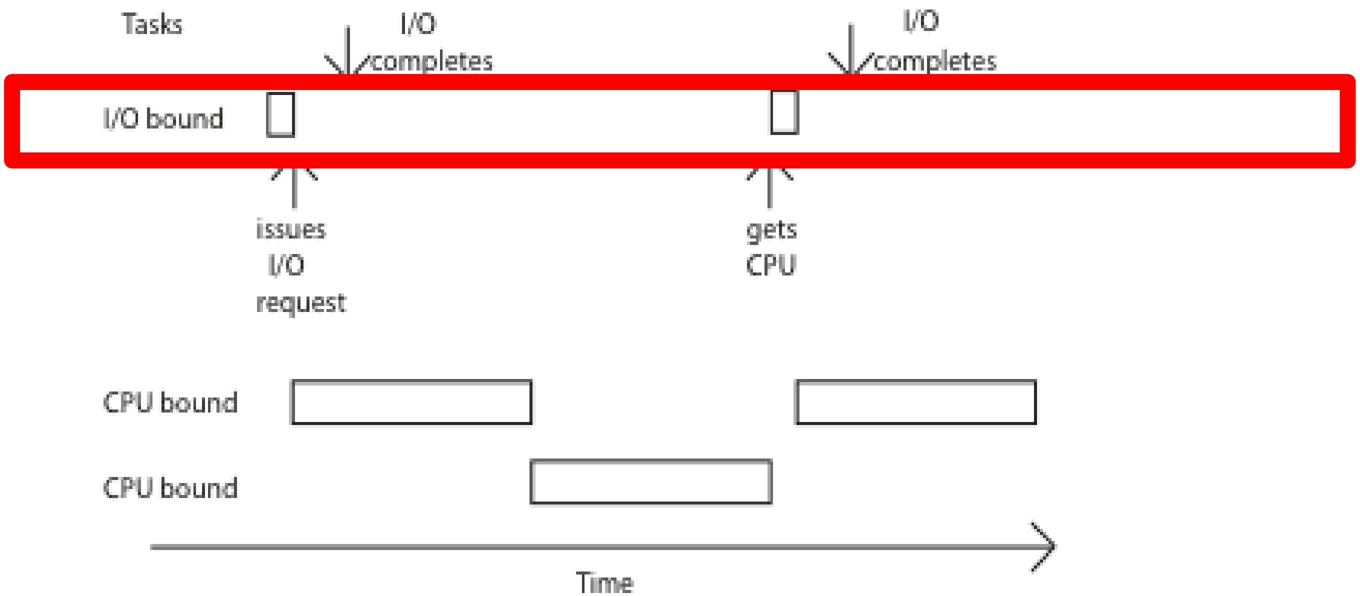
b. Round-Robin (RR) with time quantum=2sec

Process	Arrival Time	Service Time
P1	1	4
P2	2	3
P3	2	2
P4	3	1

- RR

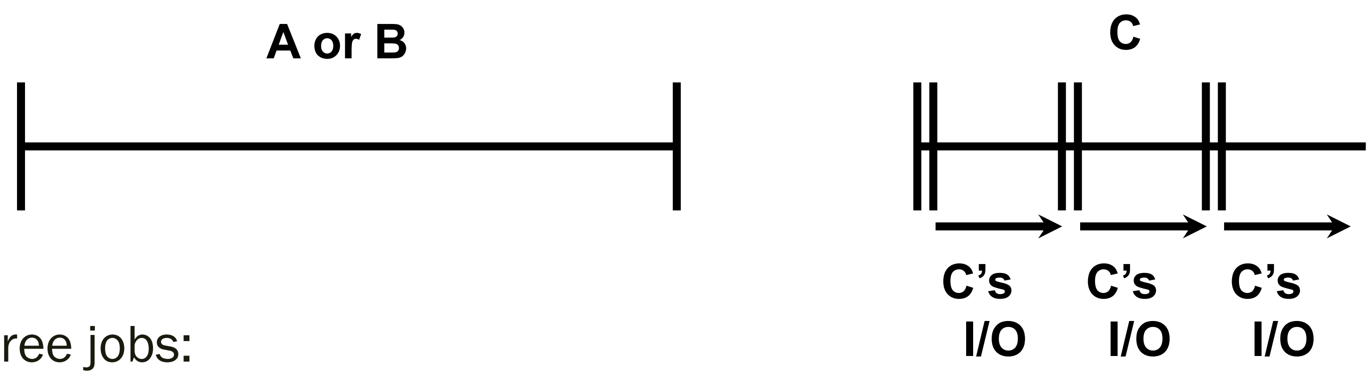
- ART = ((12-1) + (13.5 – 2) + (8 – 2) + (9.5-3))/4 = **35/4**

Another Downside of RR – Mixed Workload



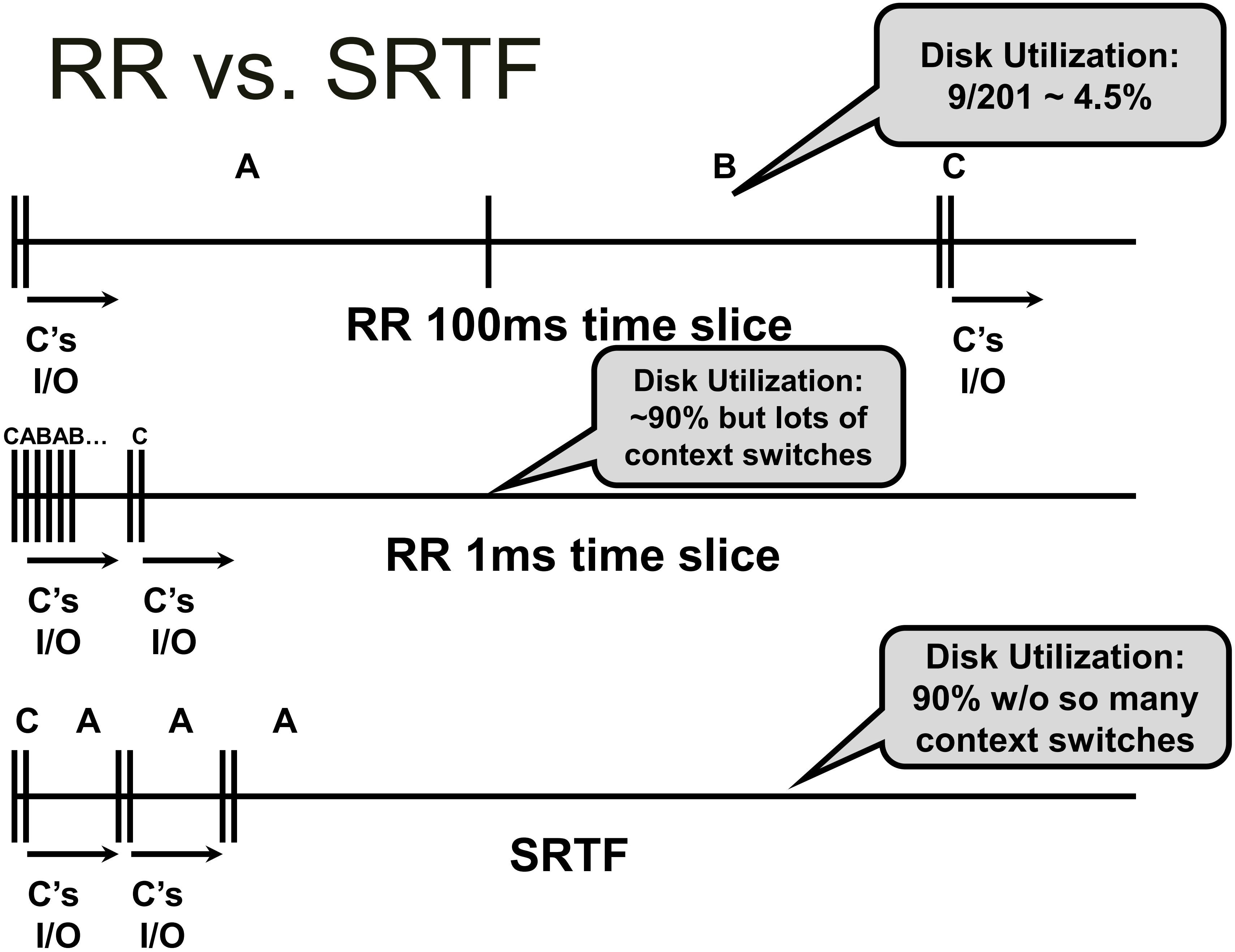
- I/O task (e.g., keystroke) must wait for its turn for the CPU
  - Gets a tiny fraction of the performance it could get
- We could shorten the RR quantum - would help, but it would increase overhead
- What would this do under **SRTF**?
  - Every time the task is ready, it is scheduled immediately!

Example - Benefits of SRTF

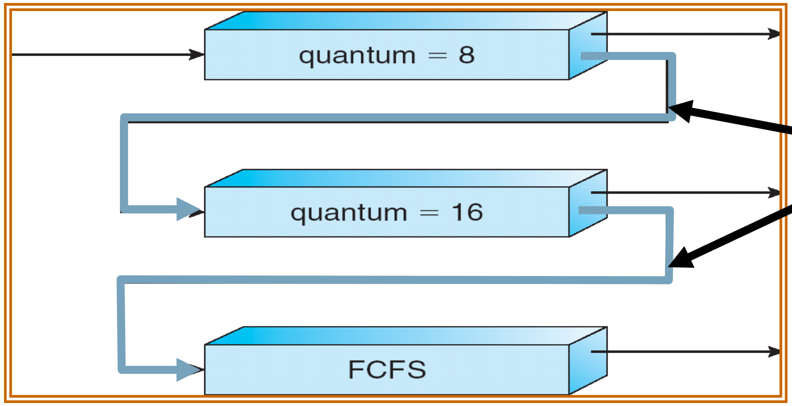


- Three jobs:
  - *A,B: CPU bound, each run for **100ms***
    *C: I/O bound, loop **1ms** CPU, 9ms disk I/O*
  - *If only one at a time, C uses 90% of the disk, A or B use 100% of the CPU*
- With FIFO:
  - *Once A or B get in, keep CPU for 100ms*
- What about RR or SRTF?
  - *Easier to see with a timeline*

RR vs. SRTF


Disk Utilization:
9/201 ~ 4.5%
A
B
C
RR 100ms time slice
C's
I/O
C's
I/O
Disk Utilization:
~90% but lots of
context switches
CABAB…
C
RR 1ms time slice
C's
I/O
C's
I/O
Disk Utilization:
90% w/o so many
context switches
C
A
A
A
SRTF
C's
I/O
C's
I/O

Multi-Level Feedback Scheduling



Long-Running Compute tasks demoted to low priority

- 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*

Countermeasure

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

- Example: MIT Othello game project (simpler version of Go game)
  - *Computer playing against competitor's computer, so key was to do computing at higher priority than the competitors.*
    - Cheater (or Winner!!!) put in carefully crafted cout statements, stayed high up in priority

MLFQ 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.
    - But this tends to be unfair
    - Example: In Multics, people found a 10-year old job while shutting down
  - *Time slice:*
    - Each queue gets a certain amount of CPU time
    - e.g., 70% to highest, 20% next, 10% lowest
    - More fair compared to fixed priority, so this one used in practice

Summary

- FCFS is simple and minimizes overhead.
- If tasks are variable in size, then FCFS can have very poor ART
- If tasks are equal in size, FCFS is optimal in terms of ART
  - *SRTF becomes equivalent to FCFS*
  - *RR would do increase ART significantly*
- SRTF is optimal in terms of ART
  - *The only way to implement is for Kalman filter-like approximations for the individual CPU bursts*
  - *But NOT fair*
  - *We also do not have preemption – longer jobs can have very long waiting times, making them unresponsive*

Summary (contd.)

- If tasks are equal in size, RR will have poor ART
- RR works poorly on a mix of CPU and I/O bound tasks
  - *SJF is hugely beneficial in this case*
- RR avoids starvation and is fair
- To avoid the downsides of RR and SRTF, we want something in the middle
  - *That would combine the good sides of both*
- MFQ scheduler is the most practical – answer to our prayer
  - *Approximates SJF while running just RR for each queue, no need to run any adaptive algorithm got the CPU bursts*
  - *Achieves a balance between responsiveness, low overhead, and fairness*
  - *Good for mixed workloads (user typing and long CPU computations running simultaneously)*