CS 630 - 002
Operating Systems Design

Lecture 3
CPU Scheduling

Jing Li

jingli@njit.edu

GITC 4106

New Jersey Institute of Technology

In This Lecture

- CPU Scheduling
  - Scheduling policy goals
  - Scheduling policy options
  - Practical considerations for implementations

Slides courtesy of Hung Daochuan, Chris Gill, David Ferry, Tarek Abdelzaher, Ion Stoica, John Kubiatowicz, Peter Dennings, Anthony Joseph, Jonathan Ragan-Kelley, Peter Troger, Insup Lee.

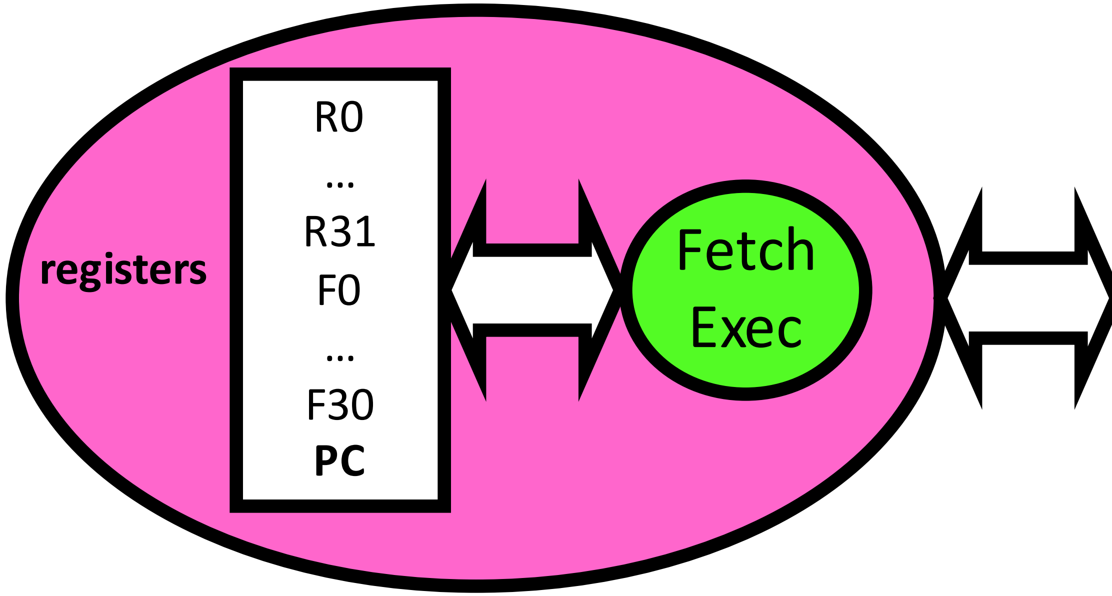
Recap

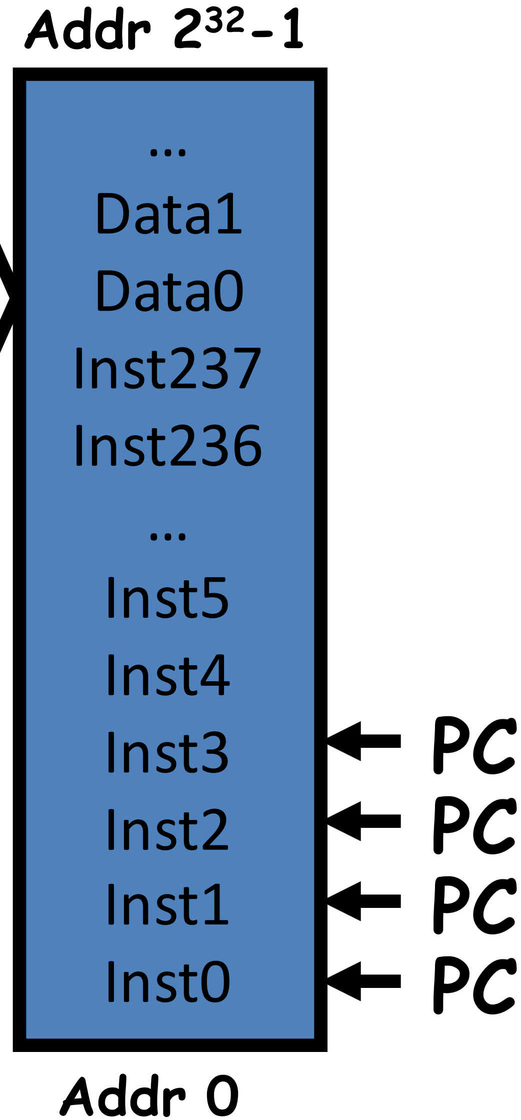
Different Operating Systems Designs

- Simple:
  - Only one or two levels of code
- Layered:
  - Lower levels independent of upper levels
- Microkernel:
  - OS built from many user-level processes
- Modular:
  - Core kernel with Dynamically loadable modules
- ExoKernel:
  - Separate protection from management of resources
- Cell-based OS:
  - Two-level scheduling of resources

NJIT

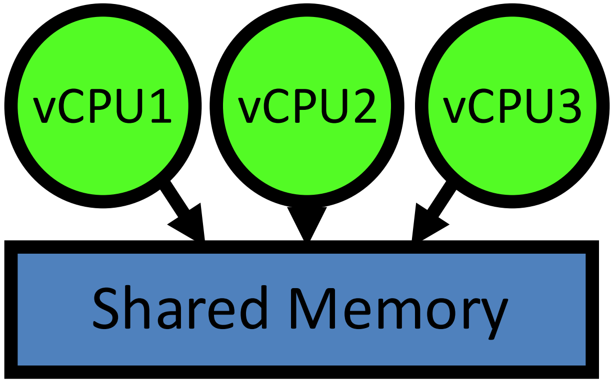
Recall: What happens during program execution?

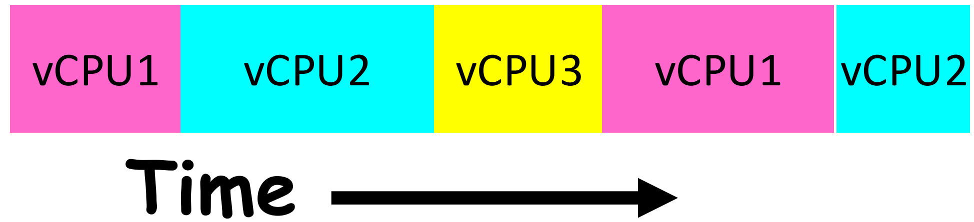




- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

How to give the illusion of multiple processors?





- Assume a single processor. How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual "CPU" needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
  - Call result a "Thread" for now…
- How to switch from one CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, other things

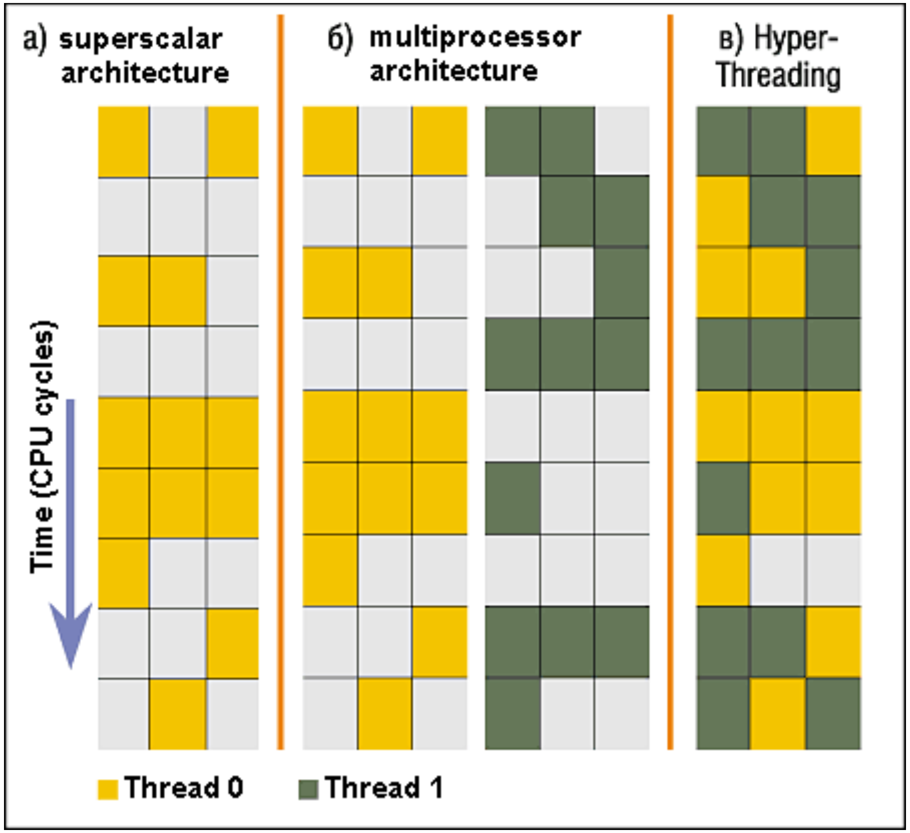
NJIT

Protecting Threads from Each Other

- Problem: Run multiple applications in such a way that they are protected from one another
- Goal:
  - Keep User Programs from Crashing OS
  - Keep User Programs from Crashing each other
  - [Keep Parts of OS from crashing other parts?]
- (Some of the required) Mechanisms:
  - Address Translation
  - Dual Mode Operation
- Simple Policy:
  - Programs are not allowed to read/write memory of other Programs or of Operating System

Modern Technique: SMT/Hyperthreading

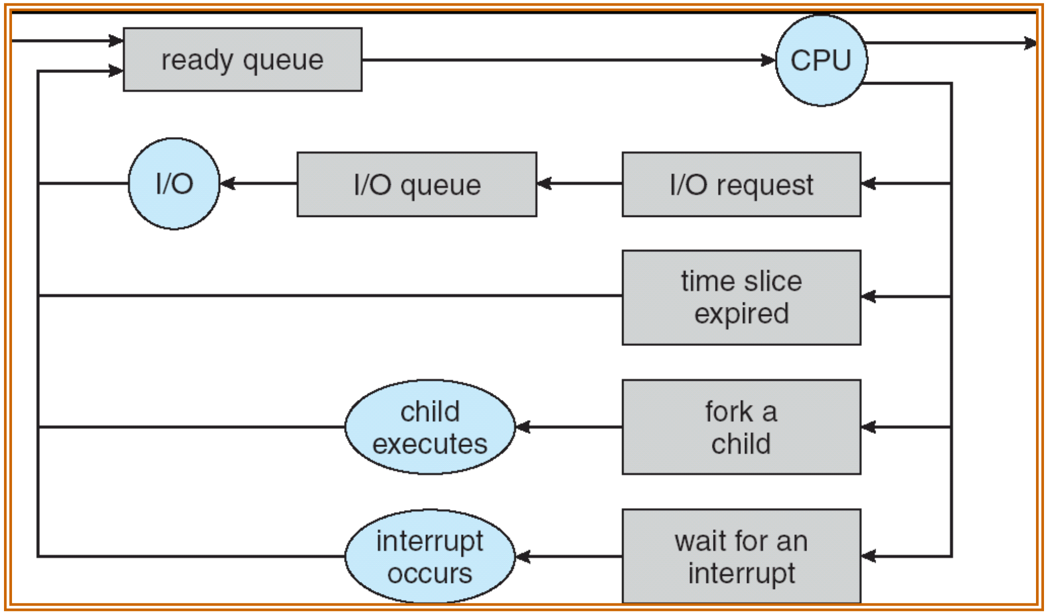
- Hardware technique
  - Exploit natural properties of superscalar processors to provide illusion of multiple processors
  - Higher utilization of processor resources
- Can schedule each thread as if were separate CPU
  - However, not linear speedup!
  - If have multiprocessor, should schedule each processor first



- Original technique called "Simultaneous Multithreading"
  - See http://www.cs.washington.edu/research/smt/
  - Alpha, SPARC, Pentium 4 ("Hyperthreading"), Power 5

Scheduling

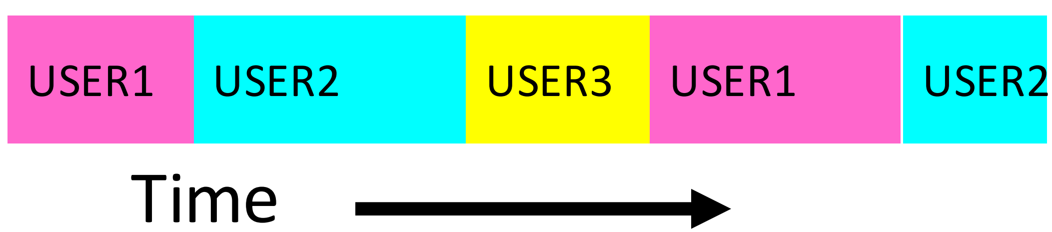
CPU Scheduling



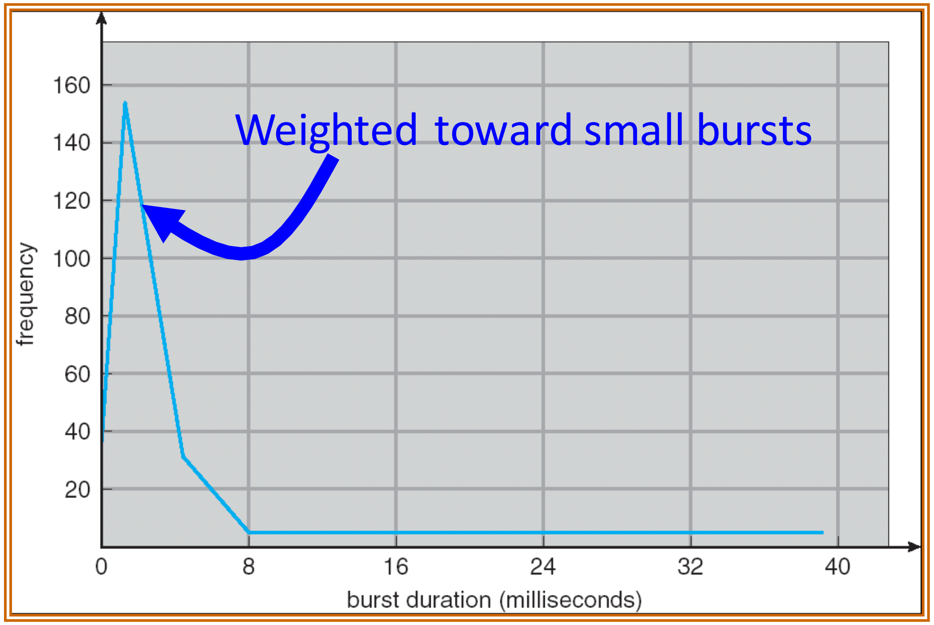
- Question: How is the OS to decide which of several tasks to take off a queue?
  - Obvious queue to worry about is ready queue
  - Others can be scheduled as well
- Scheduling: deciding which threads are given access to resources from moment to moment

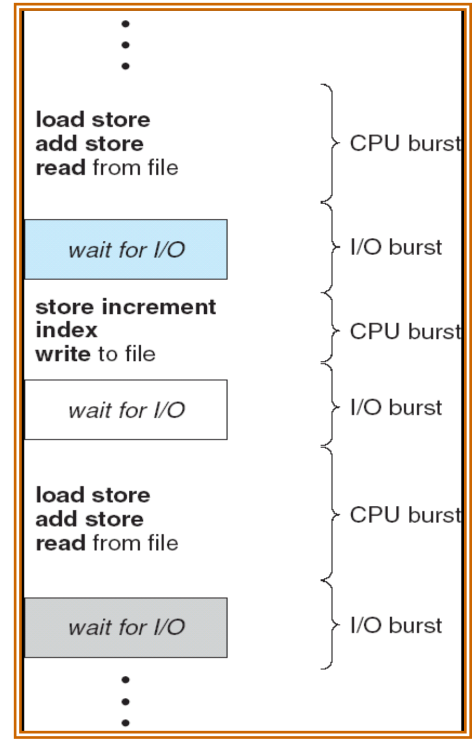
(Simplified) Scheduling Assumptions

- CPU scheduling big area of research in early 70's
- Many implicit assumptions for CPU scheduling:
  - One program per user
  - One thread per program
  - Programs are independent
- Clearly, these are unrealistic but they simplify the problem so it can be solved
  - For instance: is "fair" about fairness among users or programs?
    - If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system



Assumption: CPU Bursts





- Execution model: programs alternate between CPU & I/O bursts
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With time slicing, thread may be forced to give up CPU before finishing current CPU burst
- CPU burst distribution is of main concern

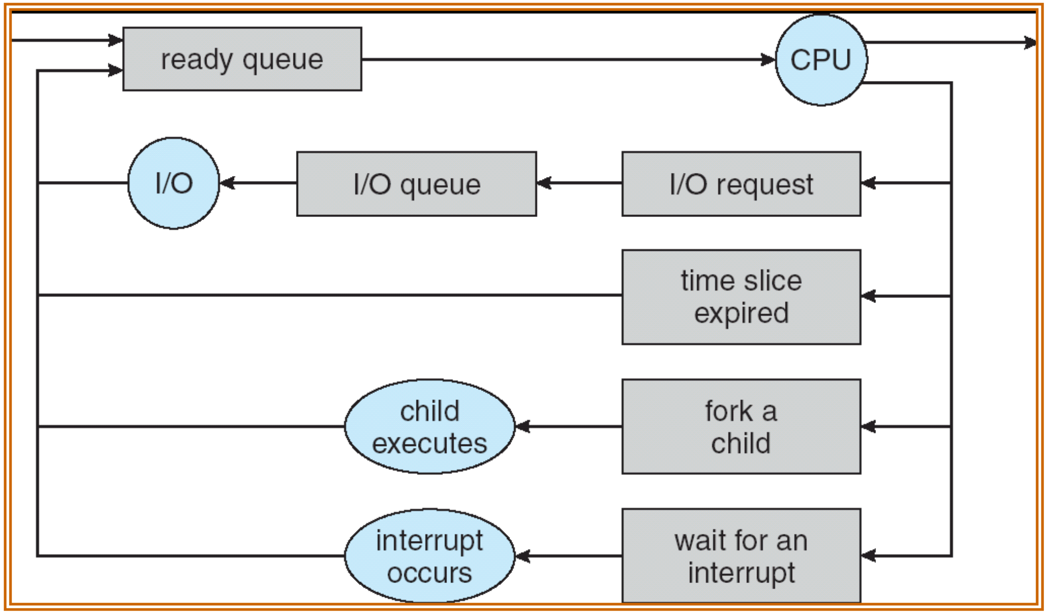
NJIT

Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    - Time to echo a keystroke in editor
    - Time to compile a program
    - Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    - Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - Minimize overhead (for example, context-switching)
    - Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - Better *average* response time by making system *less* fair

NJIT

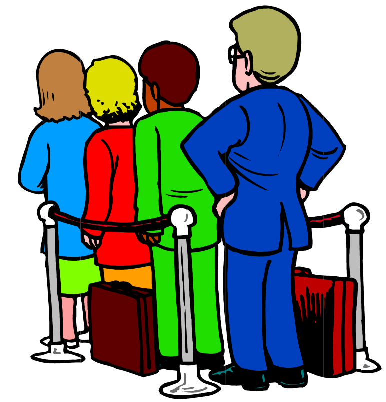
CPU Scheduler



- CPU scheduling decisions may take place when a process:
  - 1. Switches from running to waiting state
  - 2. Switches from running to ready state
  - 3. Switches from waiting to ready (i.e. by an interrupt)
  - 4. Terminates
- Scheduling under 1 and 4 is nonpreemptive
- All other scheduling is preemptive

First-Come First-Served (FCFS) Scheduling

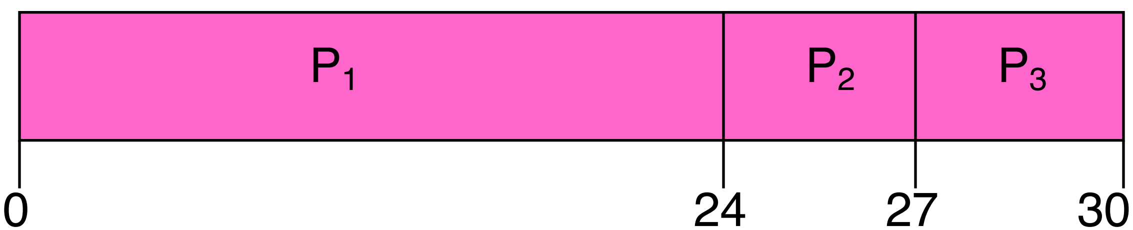
- First-Come First-Served (FCFS)
  - Also "First In First Out" (FIFO) or "Run until done"
    - In early systems, FCFS meant one program scheduled until done (including I/O)
    - Now, means keep CPU until thread blocks



- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

  - Suppose processes arrive in the order: P_1 , P_2 , P_3
    The schedule is:

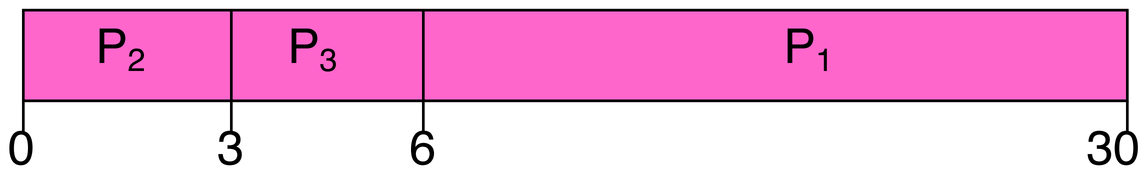


  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17
  - Average response time: (24 + 27 + 30)/3 = 27
- *Convoy effect:* short process behind long process

NJIT

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P_2 , P_3 , P_1



  - Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
  - Average waiting time: $(6 + 0 + 3)/3 = 3$
  - Average response time: $(3 + 6 + 30)/3 = 13$
- In second case:
  - Average waiting time is much better (before it was 17)
  - Average response time is better (before it was 27)
- FIFO Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)

Round Robin (RR)

- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submission order
  - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand…

- Round Robin (RR) Scheme
  - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
  - After quantum expires, the process is *preempted* and added to the end of the ready queue.
  - n processes in ready queue and time quantum is $q \Rightarrow$
    - Each process gets $1/n$ of the CPU time
    - In chunks of at most q time units
    - No process waits more than $(n-1)q$ time units

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

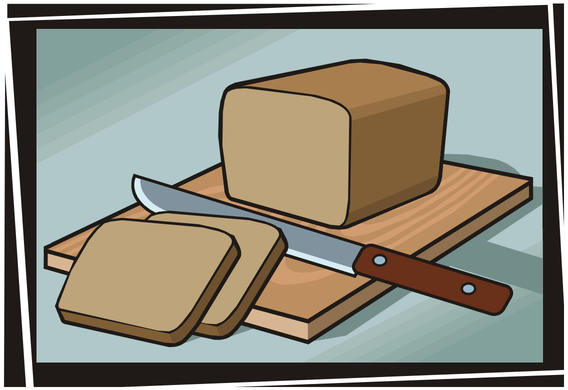
- The schedule is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 28 48 68 88 108 112 125 145 153

- Waiting time for P_1=(68-20)+(112-88)=72 P_2=(20-0)=20
  P_3=(28-0)+(88-48)+(125-108)=85
  P_4=(48-0)+(108-68)=88
- Average waiting time = (72+20+85+88)/4=66¼
- Average response time = (125+28+153+112)/4 = 104½

Round-Robin Discussion



- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (∞)?
    - Get back to FCFS
  - What if time slice too small?
    - Throughput suffers!
- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - Worked ok when UNIX was used by one or two people.
    - What if three compilations going on? 3 seconds to echo each keystroke!
  - In practice, need to balance short-job performance and long-job throughput:
    - Typical time slice today is between 10ms – 100ms
    - Typical context-switching overhead is 0.1ms – 1ms
    - Roughly 1% overhead due to context-switching

NJIT

Comparisons between FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example:
- Response Times:

10 jobs, each take 100s of CPU time
RR scheduler quantum of 1s
All jobs start at the same time

Job #	FCFS	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

  - Both RR and FCFS finish at the same time
  - Average response time is much worse under RR!
    - RR is bad when all jobs have the same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FCFS
  - Total time for RR longer even for zero-cost switch!

NJIT

Break

45 min.	Break	45 min.	Break	50 min.	End

Time

New Jersey Institute of Technology

Administrivia

Lab 1: Hello, Linux!

New Jersey Institute of Technology

Purpose of Lab 1

- Will be released tonight and will be due on Feb 19 23:55

- Your first trial of forming your group

- Ease you into programming in Linux

- Bases for a later Lab – writing a shell program

In this lab assignment, you will

- 1. Learn some basic UNIX/Linux commands
- 2. Use the built-in Linux manual (man pages) to look up certain functions and commands
- 3. Try to learn and use an editor to create and write files
- 4. Compile a "Hello, Linux!" program using the C standard library

Requirements

- 1. Linux machine (or a MacOS, but not Windows)

- 2. following the exact order

- 3. record your answers by taking a screenshot

```
yuchen@yuchen-VirtualBox:/bin$ ls |head -1
bash
yuchen@yuchen-VirtualBox:/bin$
```

- 4. submit your report in the PDF form to Canvas
  - Late penalty: see first lecture slides or syllabus
  - Only person in the team need to submit

Earlier Example with Different Time Quantum

Best FCFS:

P_2 [8]	P_4 [24]	P_1 [53]	P_3 [68]

0 8 32 85 153

	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Response Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

What if we Knew the Future?

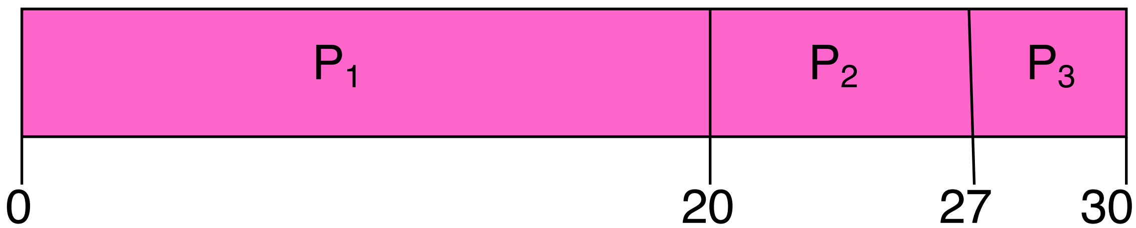
- Could we always mirror best FCFS?
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average response time
- (*Non-preemptive*) Shortest Job First (SJF, Shortest Process Next):
  - Run whatever job has the least amount of computation to do
- Shortest Remaining Time First (SRTF, SRPT):
  - If job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
- What is the difference between (non-preempt) SJF and SRTF?
  - What if when jobs arrive at the same time?
  - What if when a short job arrives before a long job's completion?

Difference between SJF and SRTF

- Example:

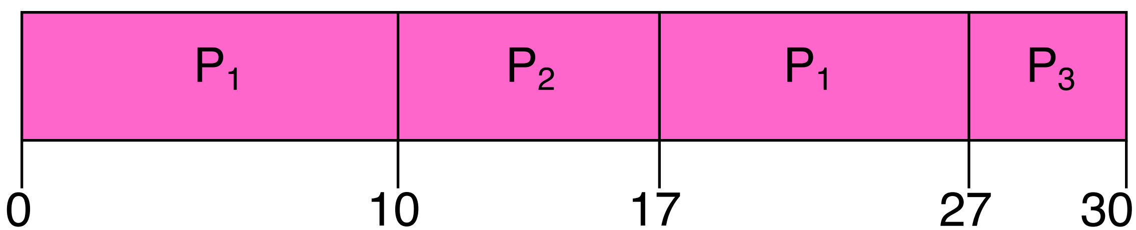
Process	Arrival Time	Burst Time
P_1	0	20
P_2	10	7
P_3	26	3

- (Non-Preemptive) SJF:



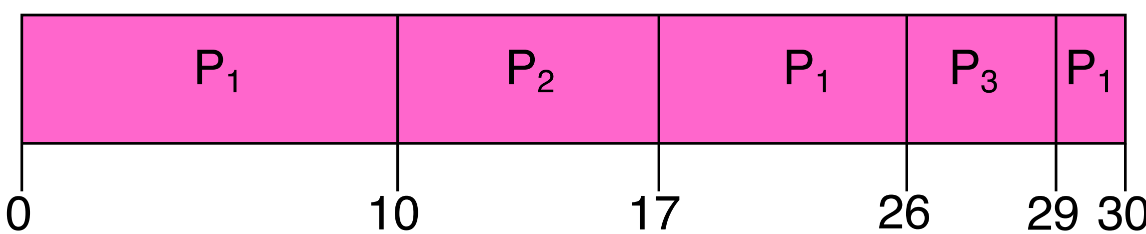
Average response time: (20 + 17 + 4)/3 = 13.67

- SRTF



Average response time: (27 + 7 + 4)/3 = 12.67

- (Preemptive SJF: strictly speaking, different from SRTF)

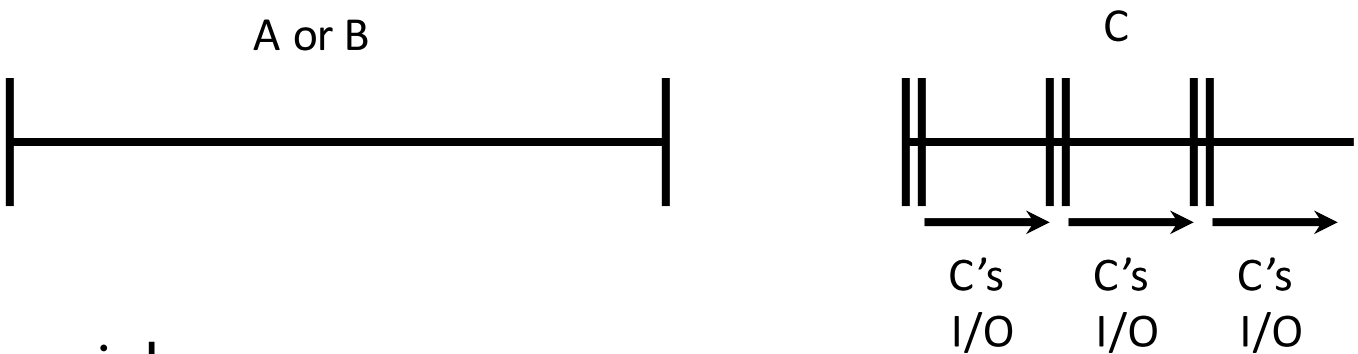


Average response time: (30 + 7 + 3)/3 = 13.3

Discussion

- SJF/SRTF are the best for minimizing average response time
  - Provably optimal
    (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - SRTF becomes the same as FCFS
      (i.e. FCFS is the best one can do if all jobs have the same length)
  - What if jobs have varying length?
    - SRTF (and RR): short jobs not stuck behind long ones

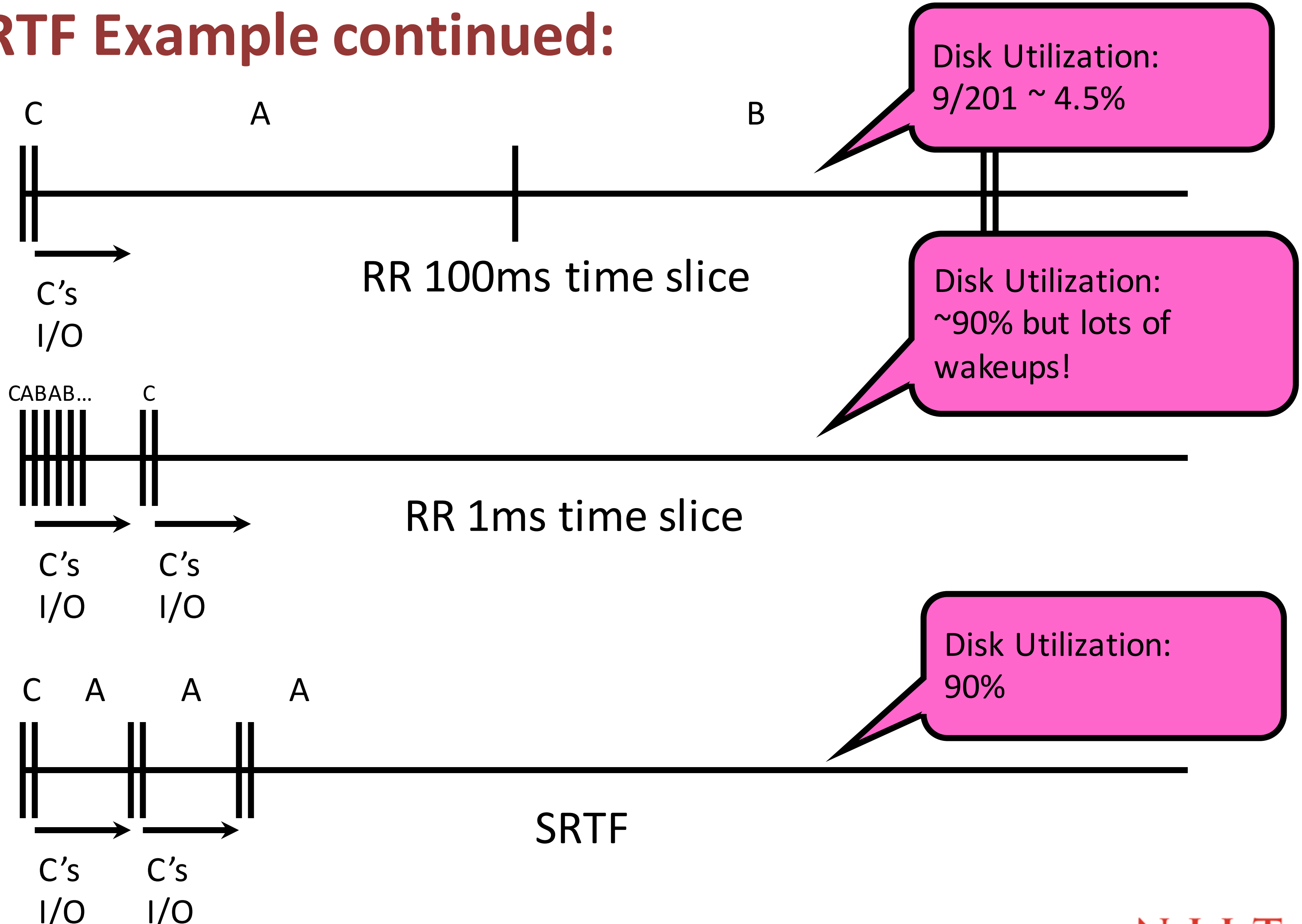
Example to illustrate benefits of SRTF



- Three jobs:
  - A,B: both CPU bound, 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 could use 100% of the CPU
- With FCFS:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline

NJIT

SRTF Example continued:



SRTF Further discussion

- Starvation
  - Large jobs never get to run (starvation), if many small jobs
- 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
- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal, so can't do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

Summary

- Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it
- FCFS Scheduling:
  - Run jobs to completion in order of submission
  - Pros: Simple (+)
  - Cons: Short jobs get stuck behind long ones (-)
- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs (+)
  - Cons: Poor when jobs are same length (-)
- Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time) (+)
  - Cons: Hard to predict future, Unfair (-)

NJIT