CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Fall 2021 Lecture 9
CPU Scheduling

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

Questions from last time

- Scheduling time unit: often millisec (1/1000 of a sec)
- Estimation & probabilistic approaches in computing
  optimal algorithms, cache, virtual memory, data centers etc. Based on field/recent data.
- Prediction of next burst
  - Based on actual recent duration and predicted value (which is based on past actual values)
  - More recent data points get more weight (based on alpha).
  - Initial prediction? Prior field data
- Shortest Job First (SJF) vs Preemptive SJF
  - SJF is not preemptive
  - Preemptive SJF (also termed **Shortest remaining time first**)
  - Priority scheduling can also be preemptive or non-preemptive

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible: Maximize
- **Throughput** – # of processes that complete their entire execution per time unit: Maximize
- **Turnaround time** –time to execute a process from submission to completion: Minimize
- **Waiting time** – total amount of time a process has been waiting in the ready queue: Minimize
- **Response time** –time it takes from when a request was submitted until the first response is produced (assumption: beginning of execution), not final output (for time-sharing environment): Minimize

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3 *but almost the same time 0*.
  The Gantt Chart for the schedule is:

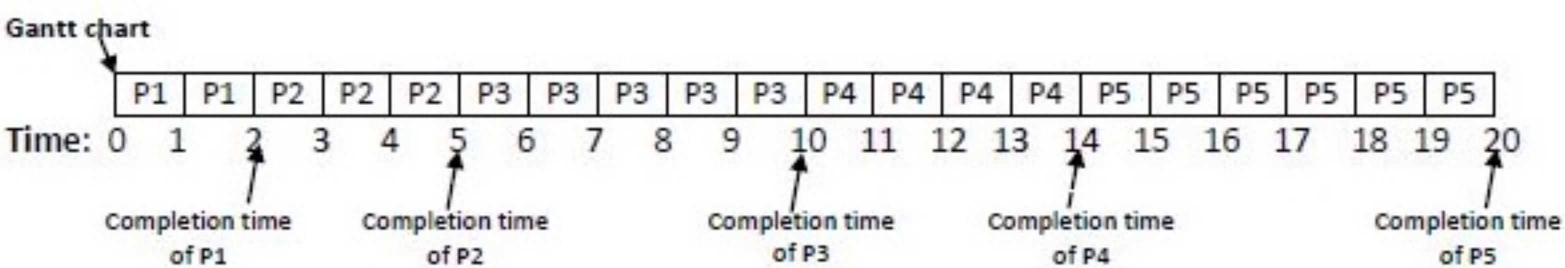
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
  - Average waiting time: (0 + 24 + 27)/3 = 17
- Throughput: processes finished per unit time 3/30 = 0.1 per unit
- Turnaround time for P1, P2, P3 = 24, 27, 30 thus average = 8.2
- *Response* time for P1, P2, P3 = 0, 24, 27 assuming .. Thus the average is ..

Turnaround time –time to execute a process from submission to completion.
Response time –time it takes from when a request was submitted until the first response is produced (assumption: beginning of execution), not final output.

Colorado State University

Example: FCFS (from IC Q)

Given			From Gantt chart		Calculation	
Process ID	**Arrival Time**	**Burst time**	Begins	Completion time	Turnaround time	Waiting time
P1	**0**	**2**	0	2	2-0=2	0
P2	**1**	**3**	2	5	5-1=4	2-1=1
P3	**2**	**5**	5	10	10-2=8	3
P4	**3**	**4**	10	14	14-3=11	7
P5	**4**	**6**	14	20	20-4=16	10
Av					41/5=8.2	21/5=4.2



Note: Processes arrive when they want to. They have to wait when CPU is busy.

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- Reduction in waiting time for short process *GREATER THAN* Increase in waiting time for long process
- SJF is optimal – gives **minimum average waiting time** for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Estimate or could ask the user

Colorado State University

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- All arrive at time 0.
- SJF scheduling chart

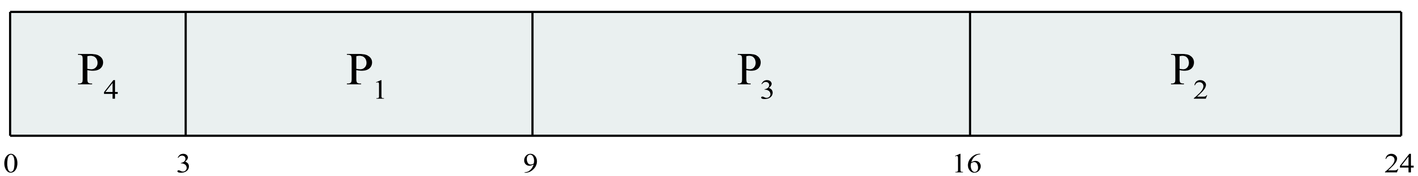
- Average waiting time for P_1,P_2,P_3,P_4 = (+   +   +   ) /   =

Pause for students to do the computation

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- All arrive at time 0.
- SJF scheduling chart



- Average waiting time for P_1,P_2,P_3,P_4 = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

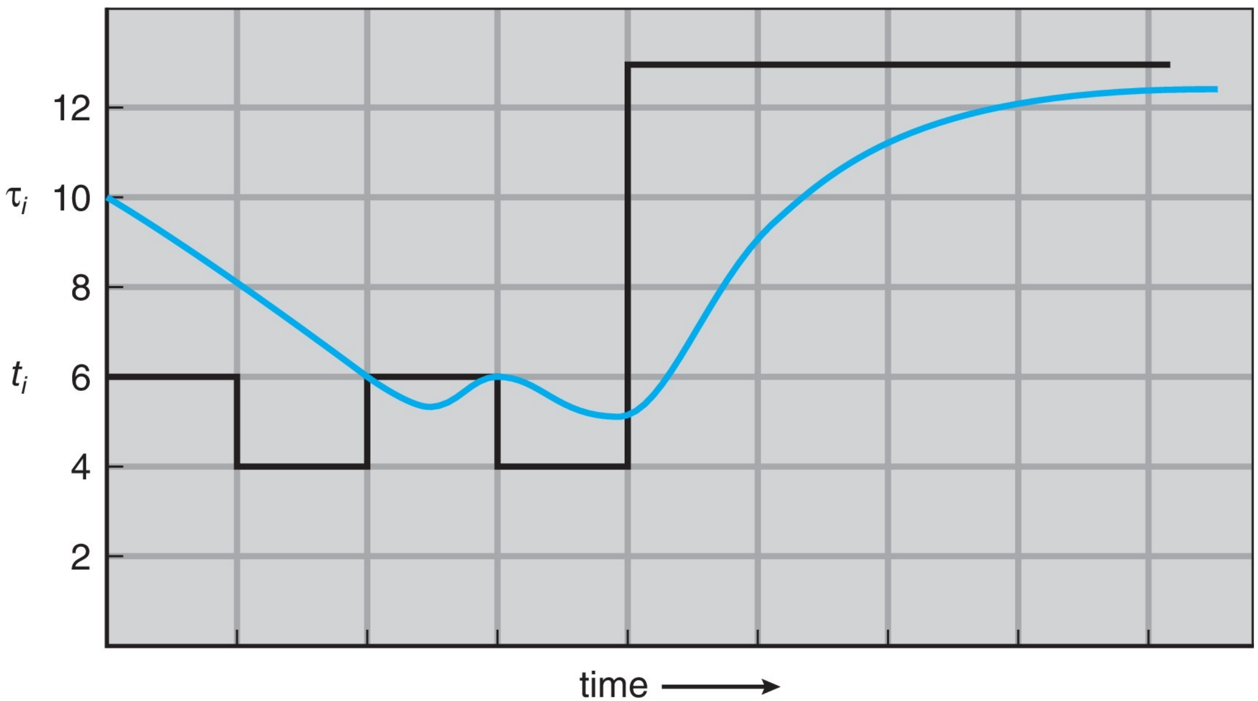
- Can only estimate the length – should be similar to the recent bursts
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using *exponential averaging*

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define: $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n .$

- Commonly, α set to ½

Colorado State University

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Blue points: guess
Black points: actual
$\alpha = 0.5$

Ex:
0.5x6 +0.5x10 = 8

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$
- If we expand the formula, substituting for τ_n, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Widely used for predicting stock-market etc

Shortest-remaining-time-first (**preemptive** SJF)

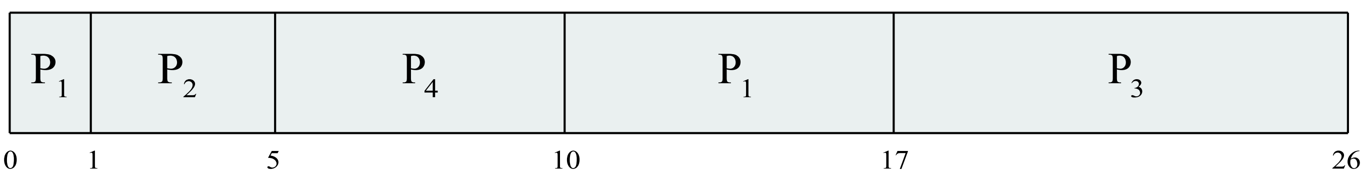
- Preemptive version called **shortest-remaining-time-first**

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4 (will preempt because 4<7)
P_3	2	9 (will not preempt)
P_4	3	5

0	P1
1	P2 preempts P1
2	P3 doesn't P2
3	..
4	..
5	RT: P1=7, P3:9, P4:5. Thus ..

- *Preemptive* SJF Gantt Chart



- Average waiting time for P1,P2,P3,P4

= [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 msec

Preempted process gets into Ready Queue (not FCFS here)

Colorado State University

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive

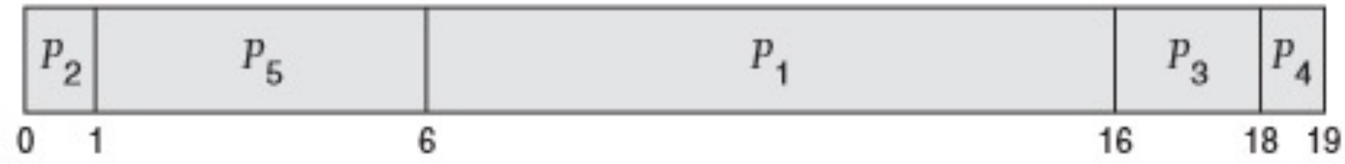
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
  - Solution ≡ **Aging** – as time progresses increase the priority of the process

MIT had a low priority job waiting from 1967 to 1973 on IBM 7094! ☺

Ex Priority Scheduling non-preemptive

Process	Burst Time	Priority
P_1	10	3
P_2	1	1 (highest)
P_3	2	4
P_4	1	5
P_5	5	2

- P1,P2, P3, P4,P5 all arrive at time 0.
- Priority scheduling Gantt Chart



- Average waiting time for P1, .. P5: (6+0+16+18+1)/5 = 8.2 msec

Colorado State University

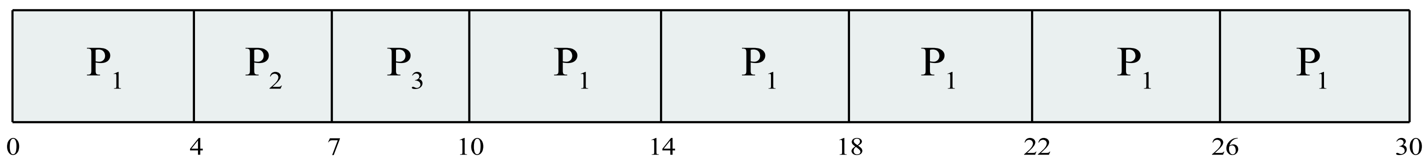
Round Robin (RR) with time quantum

- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds. After this, the process is preempted, added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high (overhead *typically in 0.5% range*)

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Arrive a time 0 in order P1, P2, P3: The Gantt chart is:

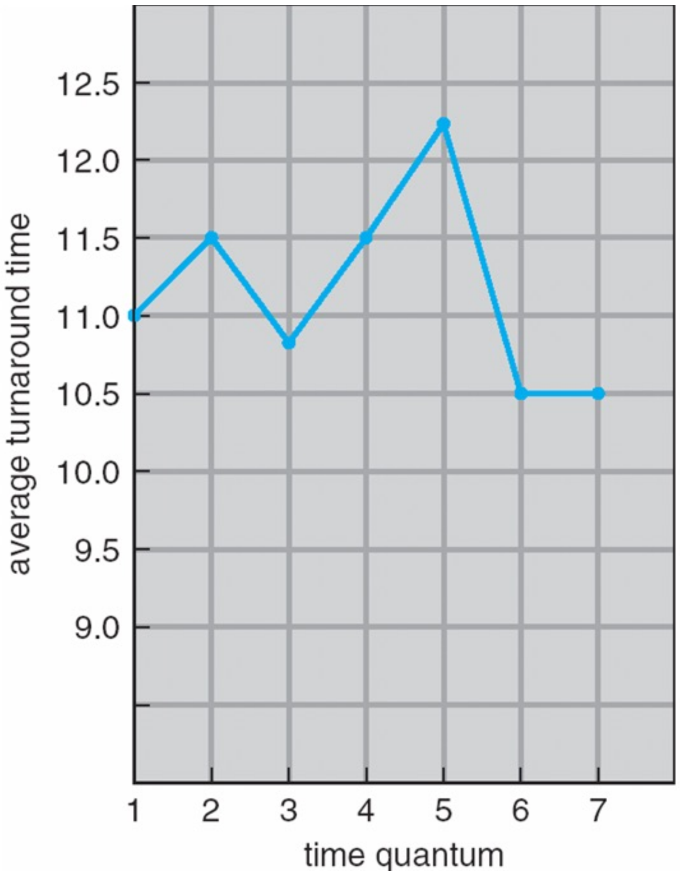


- Waiting times: P1:10-4 =6, P2:4, P3:7, average 17/3 = 5.66 units
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 µsec

Response time: Arrival to beginning of execution
Turnaround time: Arrival to finish of execution

Colorado State University

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Rule of thumb: 80% of CPU bursts should be shorter than q

Ex: Round robin with quant q=7.
All processes arrive at about the same time.
Turnaround time for P1,P2,P3,P4:
6,9,10,17 av = 10.5
Similarly for q =1, ..6 (try at home)

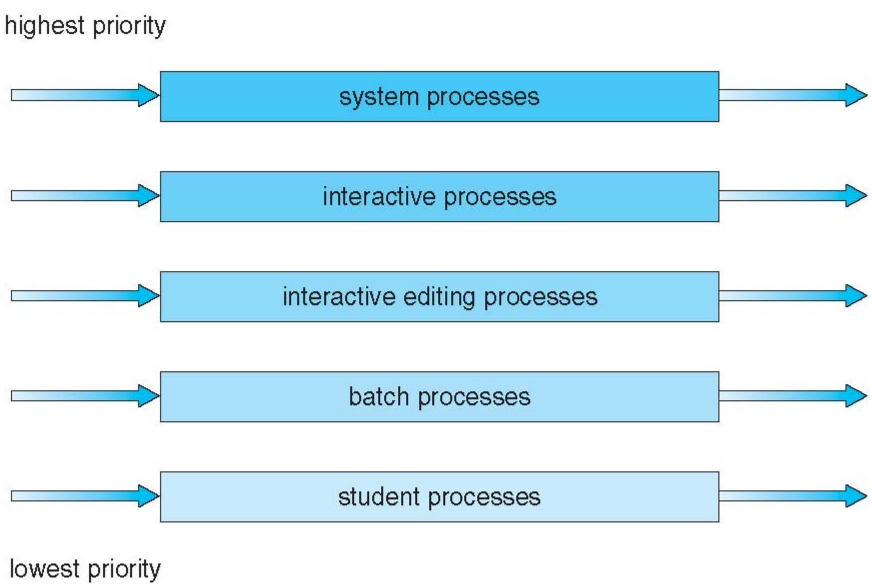
Response time: Arrival to *beginning* of execution
Turnaround time: Arrival to finish of execution

Colorado State University

Multilevel Queue

- Ready queue is partitioned into separate queues, e.g.:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm, e.g.:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation. Or
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

Multilevel Queue Scheduling



Real-time processes may have the highest priority.

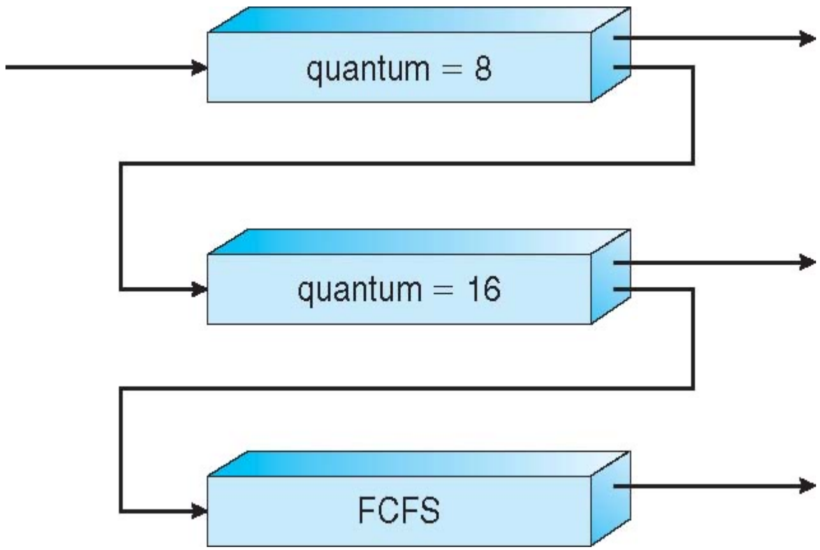
Multilevel *Feedback* Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service
  - Details at ARPACI-DUSSEAU

Inventor FJ Corbató won the Touring award!

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS (no time quantum limit)



- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

Upgrading may be based on aging. Periodically processes may be moved to the top level.

Variations of the scheme were used in earlier versions of Linux.

Completely fair scheduler Linux 2.6.23

Goal: fairness in dividing processor time to tasks (Con Kolivas, Anaesthetist)

- Variable time-slice based on number and priority of the tasks in the queue.
  - Maximum execution time based on waiting processes (Q/n).
  - Fewer processes waiting, they get more time each
- Queue ordered in terms of “virtual run time”
    - execution time on CPU added to value
  - smallest value picked for using CPU
  - small values: tasks have received less time on CPU
  - I/O bound tasks (shorter CPU bursts) will have smaller values
- *Balanced (red-black) tree* to implement a ready queue;
  - Efficient. O(log n) insert or delete time
- Priorities (*niceness*) cause different decays of values: higher priority processes get to run for longer time
  - virtual run time is the weighted run-time

Scheduling schemes have continued to evolve with continuing research. A comparison.

Thread Scheduling

- Thread scheduling is similar
- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes

Scheduling competition

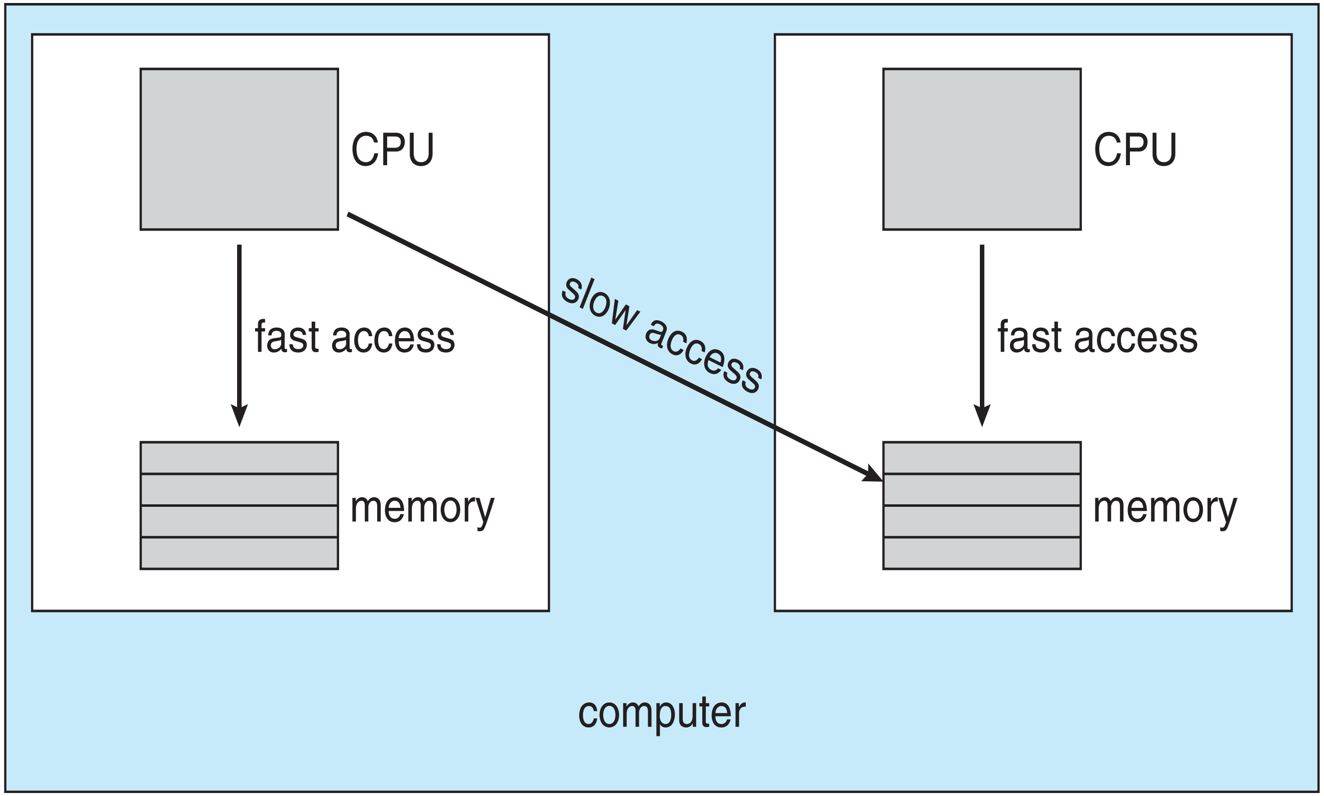
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system
- Pthread API allows both, but Linux and Mac OSX allows only SCS.

LWP layer between kernel threads and user threads in some older OSs

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available.
- **Assume Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – individual processors can be dedicated to specific tasks at design time
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling,
  - all processes in common ready queue, **or**
  - each has its own private queue of ready processes
    - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running because of info in cache
  - **soft affinity: try but no guarantee**
  - **hard affinity can specify processor sets**

NUMA and CPU Scheduling



Note that memory-placement algorithms can also consider affinity
Non-uniform memory access (NUMA), in which a CPU has faster access to some parts of main memory.

Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
  - **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
  - **Pull migration** – idle processors pulls waiting task from busy processor
  - Combination of push/pull may be used.

Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core
  - Concurrent
  - Parallel: with hyper-threading hardware

Real-Time CPU Scheduling

- Can present obvious challenges
  - **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
  - **Hard real-time systems** – task must be serviced by its deadline
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
  - **periodic** ones require CPU at constant intervals

RTOS: real-time OS. QNX in automotive, FreeRTOS etc.

Virtualization and Scheduling

- Virtualization software schedules multiple guests OSs onto CPU(s)
- Each guest doing its own scheduling
  - Not knowing it doesn't own the CPUs
  - Can affect time-of-day clocks in guests
- Virtual Machine Monitor has its own scheduler
- Various approaches have been used
  - Workload aware, Guest OS cooperation, etc.

Operating System Examples

- Solaris scheduling: 6 classes, Inverse relationship between priorities and time quantum
- Windows XP scheduling: 32 priority levels (real-time, non-real-time levels)
- Linux scheduling schemes have continued to evolve.
  - Linux Version 2.5: Two multilevel priority (“nice values”) queue sets
  - Linux Completely fair scheduler (CFS, 2007):

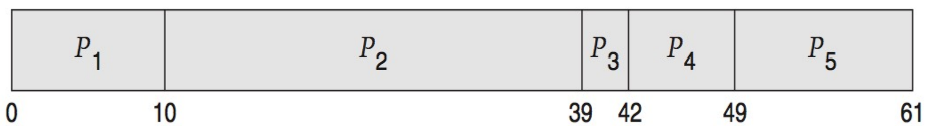
Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of analytic evaluation
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

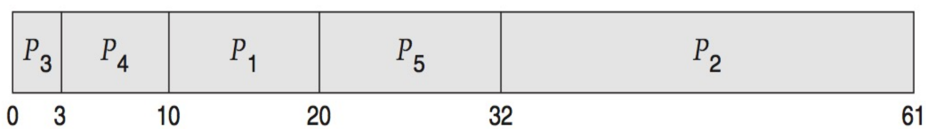
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

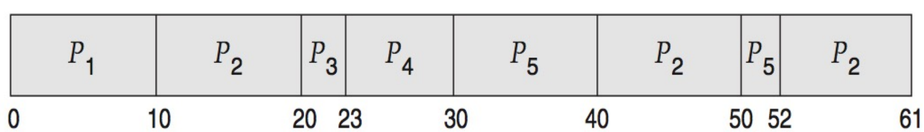
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:



  - Non-preemptive SFJ is 13ms:



  - RR is 23ms:



Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Probabilitistic Models

- Assume that the arrival of processes, and CPU and I/O bursts are random
  - Repeat deterministic evaluation for many random cases and then average
- Approaches:
  - Analytical: Queuing models
  - Simulation: simulate using realistic assumptions

Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

Little's Formula for av Queue Length

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

  $$n = \lambda \times W$$

  - Valid for any scheduling algorithm and arrival distribution
- Example: average 7 processes arrive per sec, and 14 processes in queue, then average wait time per process $W = n/\lambda = 14/7 = 2$ sec

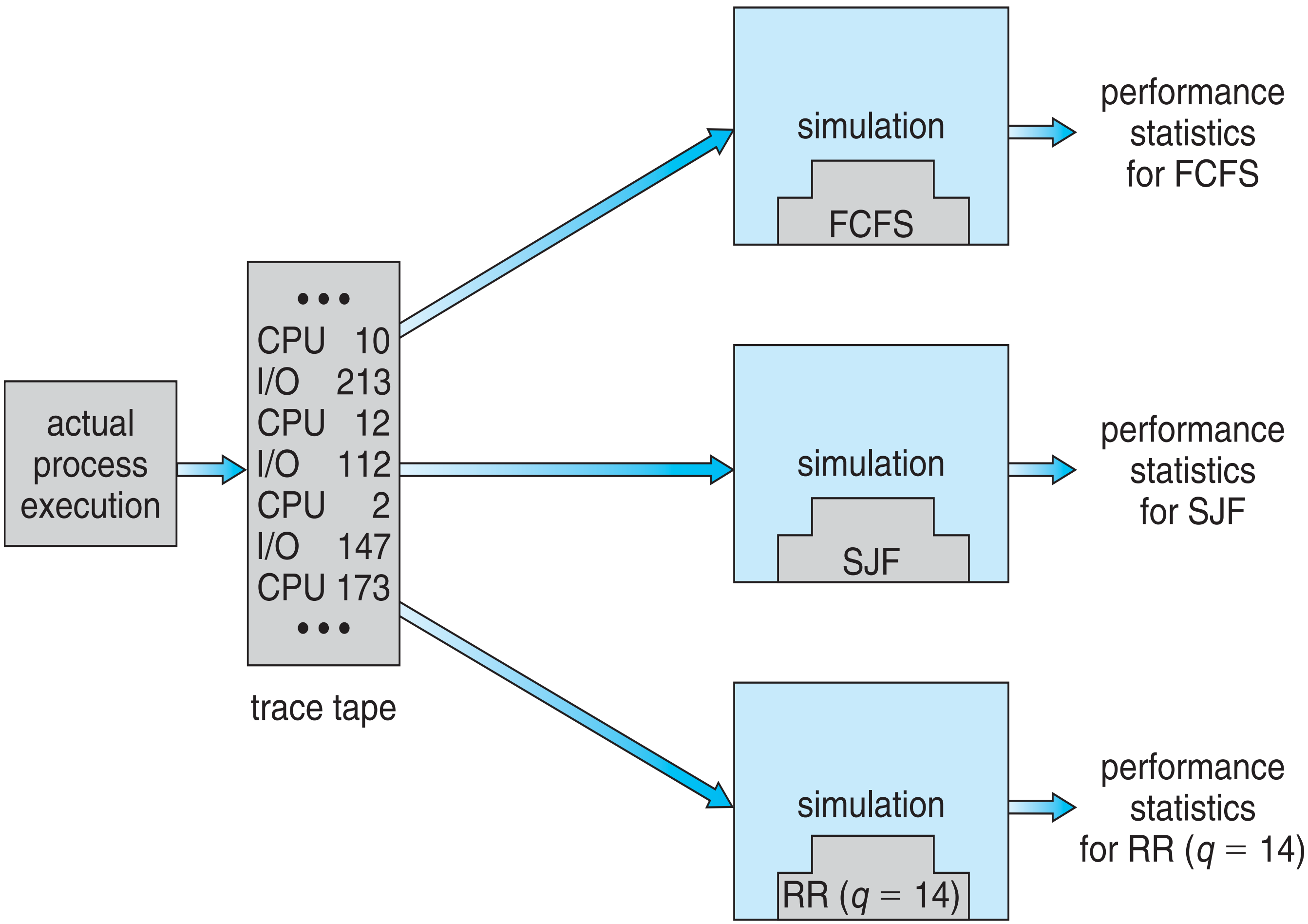
Each process takes $1/\lambda$ time to move one position.
Beginning to end delay $W = n \times (1/\lambda)$

Colorado State University

Simulations

- Queueing models limited
- **Simulations** more versatile
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - *Trace tapes* record sequences of real events in real systems

Evaluation of CPU Schedulers by Simulation



Actual Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary

ICQ Thurs

Q1

i. Pthreads are a POSIX standard API for thread creation and synchronization. True

ii. A Pthread library is always implemented in the user space. False

Q2.

In a thread with deferred cancellation, cancellation only occurs when

A: The thread reaches the Cancellation point

CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Synchronization

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

Process Synchronization: Objectives

- Concept of process synchronization.
- The critical-section problem, whose solutions can be used to ensure the consistency of shared data
- Software and hardware solutions of the critical-section problem
- Classical process-synchronization problems
- Tools that are used to solve process synchronization problems

Process Synchronization

EW Dijkstra *Go To Statement Considered Harmful*

Colorado State University

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	Look in fridge. Out of milk.
12:40	Arrive at store.	Leave for store
12:45	Buy milk.	Arrive at store.
12:50	Arrive home, put milk away.	Buy milk
12:55		Arrive home, put milk away. Oh no!

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- **Illustration**: we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers.
  - have an integer **`counter`** that keeps track of the number of full buffers.
  - Initially, **`counter`** is set to 0.
  - It is incremented by the producer after it produces a new buffer
  - decremented by the consumer after it consumes a buffer.

  Will it work without any problems?