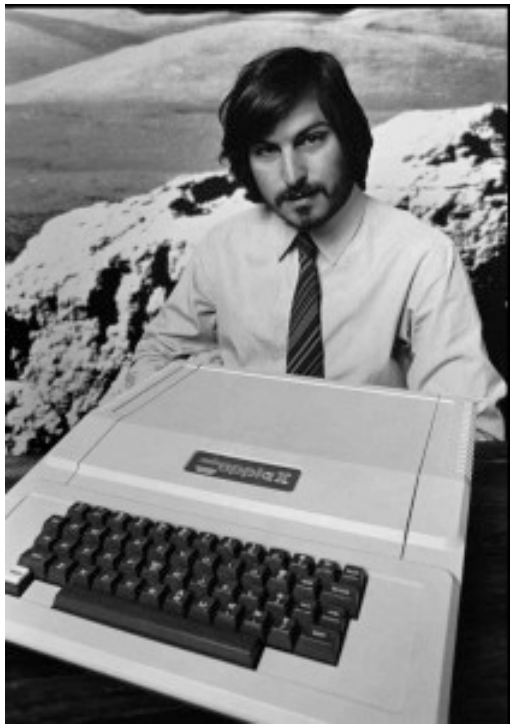
COS 318: Operating Systems

CPU Scheduling

Prof. Margaret Martonosi
Computer Science Department
Princeton University

http://www.cs.princeton.edu/courses/archive/fall11/cos318/

Steve Jobs 1955-2011



“Your time is limited, so don't waste it living someone else's life. Don't be trapped by dogma — which is living with the results of other people's thinking. Don't let the noise of others' opinions drown out your own inner voice. And most important, have the courage to follow your heart and intuition. They somehow already know what you truly want to become. Everything else is secondary…”

Stanford University Commencement Address 2005
http://news.stanford.edu/news/2005/june15/jobs-061505.html

Announcements

- No rest for the weary… ☺
  - Project 2 now posted. Due Oct 19.

- Upcoming Seminar:
- Prof. Tom Wenisch, University of Michigan
- "Making Enterprise Computing Green: Efficiency Challenges in Warehouse-Scale Computers"
  - Range of novel techniques for managing data centers in order to reduce energy consumption or peak power dissipation.
  - CS Small Auditorium, 4:30pm, Oct 18.

- Also, he'll be doing a session at 1pm that day (before this class) on grad school opportunities at UMichigan.

Today's Topics

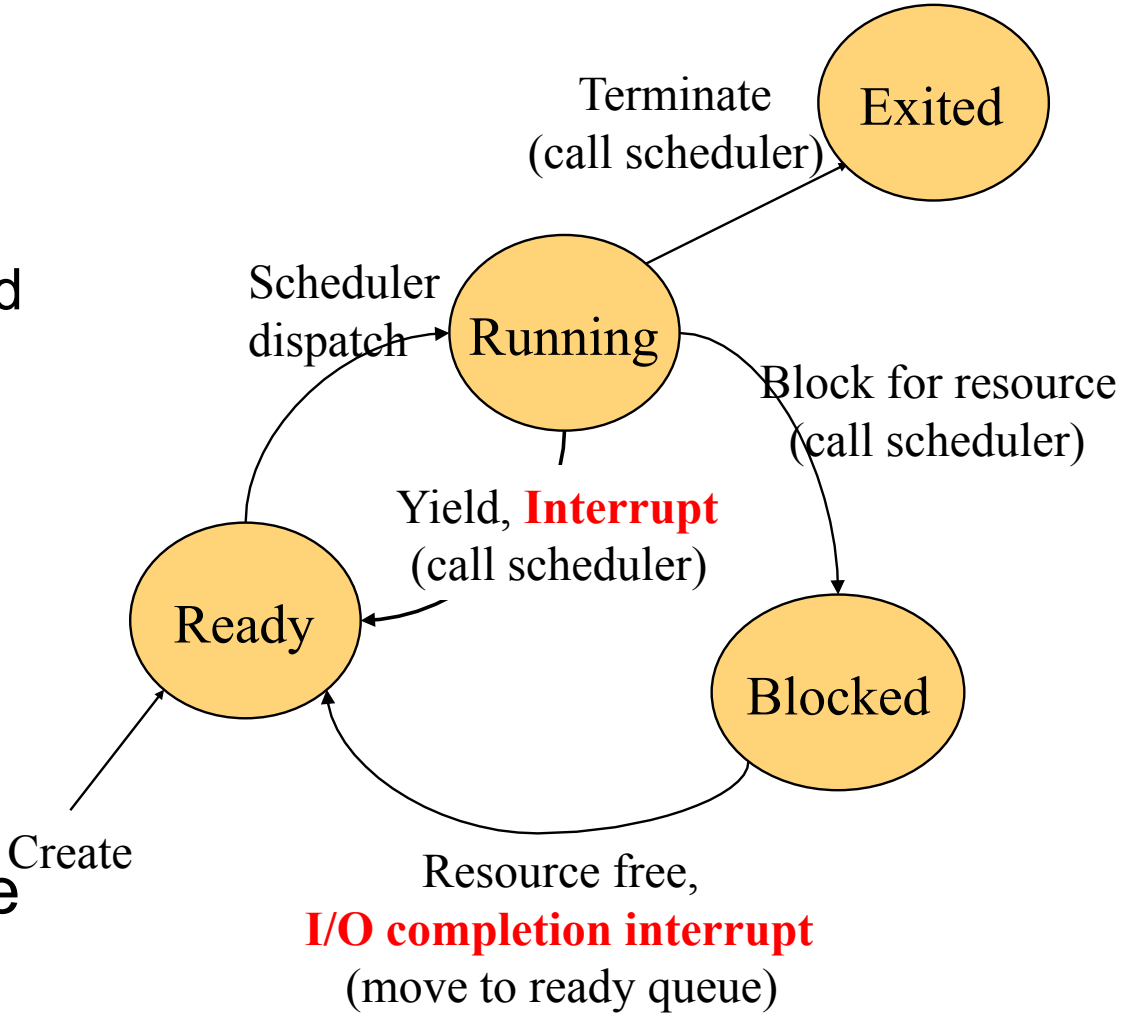
- CPU scheduling basics
- CPU Scheduling algorithms

When to Schedule?

- Process/thread creation
- Process/thread exit
- Blocking on I/O or synchronization
- I/O interrupt
- Clock interrupt (pre-emptive scheduling)

Preemptive vs. Non-Preemptive Scheduling

- Preemptive scheduling
  - Running ⇒ ready
  - Blocked ⇒ ready
  - Running ⇒ blocked
  - Terminate
- Non-preemptive scheduling
  - Running ⇒ ready
  - Blocked ⇒ ready
- Batch vs interactive vs real-time



Separation of Policy and Mechanism

- “Why and What” vs. “How”
- Objectives and strategies vs. data structures, hardware and software implementation issues.
- Process abstraction vs. Process machinery

Policy and Mechanism

- Scheduling policy answers the question:
  Which process/thread, among all those ready to run, should be given the chance to run next?
- Mechanisms are the tools for supporting the process/ thread abstractions and affect *how* the scheduling policy can be implemented. (this is review)
  - How the process or thread is represented to the system - process or thread control blocks.
  - What happens on a context switch.
  - When do we get the chance to make these scheduling decisions (timer interrupts, thread operations that yield or block, user program system calls)

CPU Scheduling Policy

- The CPU scheduler makes a sequence of "moves" that determines the interleaving of threads.
  - Programs use synchronization to prevent "bad moves".
  - …but otherwise scheduling choices appear (to the program) to be *nondeterministic*.
- The scheduler's moves are dictated by a *scheduling policy*.

Scheduler Policy: Goals & Metrics of Success

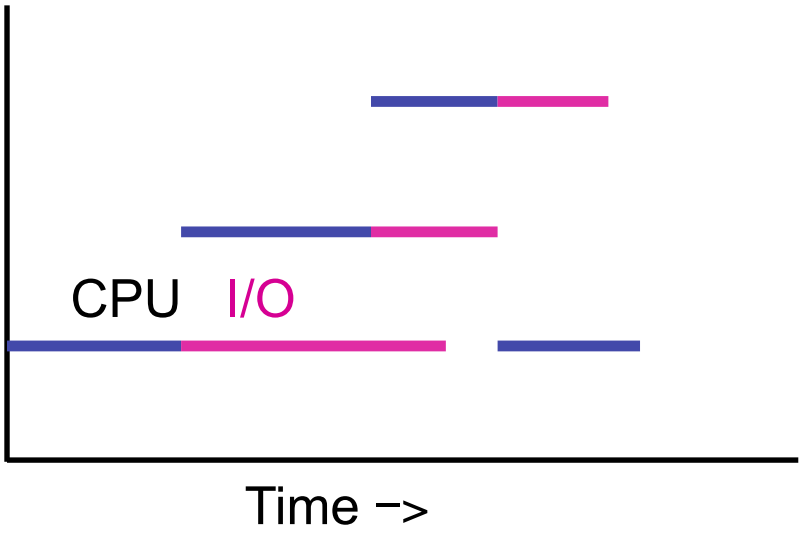
- Response time or latency (to minimize the average time between arrival to completion of requests)
  - How long does it take to do what I asked? (R) Arrival -> done.
- Throughput (to maximize productivity)
  - How many operations complete per unit of time? (X)
- Utilization (to maximize use of some device)
  - What percentage of time does the CPU (and each device) spend doing useful work? (U)
    time-in-use / elapsed time
- Fairness
  - What does this mean? Divide the pie evenly? Guarantee low variance in response times? Freedom from starvation?
- Meet deadlines and guarantee jitter-free periodic tasks
  - real time systems (e.g. process control, continuous media)

Articulating Policies

- Given some of the goals just mentioned, what kind of policies can you imagine?
- What information would you need to know in order to implement such a policy?
- How would you get the information?

Multiprogramming and Utilization

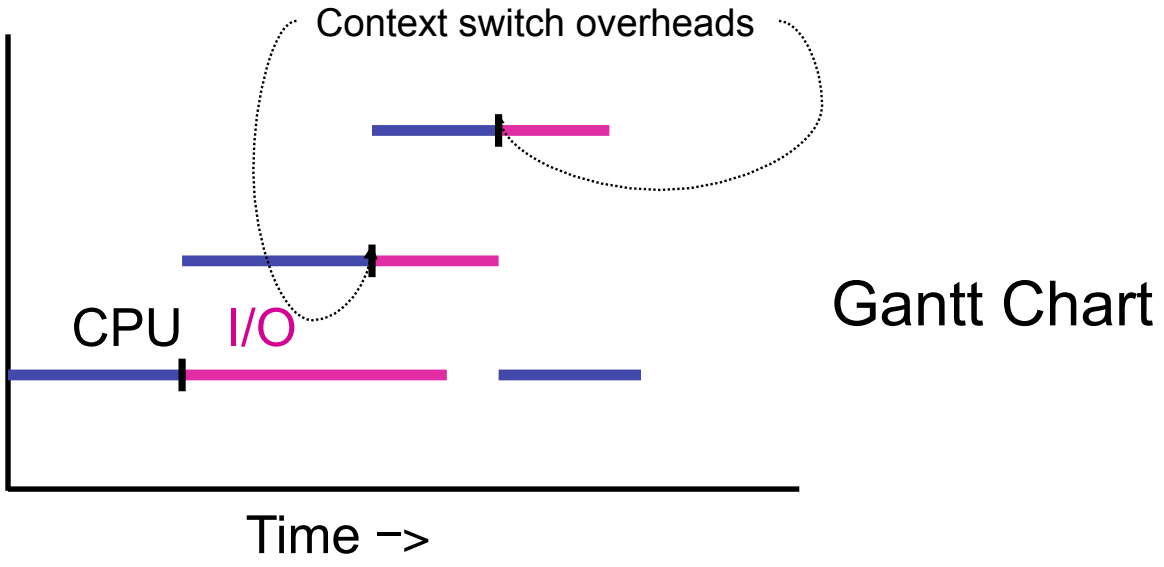
- Early motivation: *Overlap* of computation and I/O
- Determine *mix* and *multiprogramming level* with the goal of "covering" the idle times caused by waiting on I/O.



Gantt Chart

Multiprogramming and Utilization

- Early motivation: *Overlap* of computation and I/O
- Determine *mix* and *multiprogramming level* with the goal of "covering" the idle times caused by waiting on I/O.



Flavors

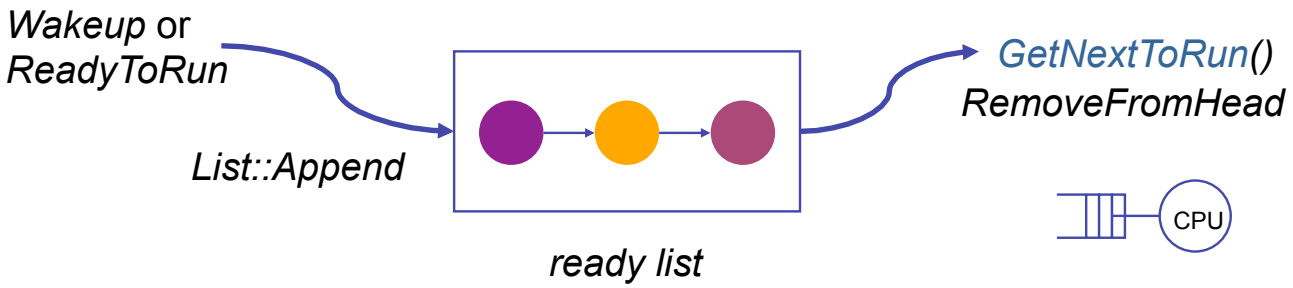
- ***Long-term scheduling*** - which jobs get resources (e.g. get allocated memory) and the chance to compete for cycles (be on the ready queue).
- ***Short-term scheduling or process scheduling*** - which of those gets the next slice of CPU time
- ***Non-preemptive*** - the running process/thread has to explicitly give up control
- ***Preemptive*** - interrupts cause scheduling opportunities to reevaluate who should be running now (is there a more “valuable” ready task?)

Scheduling Algorithms

- FIFO, FCFS
- SJF - Shortest Job First (provably optimal in minimizing average response time, assuming we know service times in advance)
- Round Robin
- Multilevel Feedback Queuing
- Priority Scheduling

A Simple Policy: FCFS

- The most basic scheduling policy is *first-come-first-served,* also called *first-in-first-out* (FIFO).
  - FCFS is just like the checkout line at the QuickiMart.
    - Maintain a queue ordered by time of arrival.
    - *GetNextToRun* selects from the front of the queue.
  - FCFS with preemptive timeslicing is called *round robin*.



First-Come-First-Serve (FCFS) Policy

- What does it mean?
  - Run to completion (old days)
  - Run until blocked or yields
- Example 1
  - P1 = 24sec, P2 = 3sec, and P3 = 3sec, submitted together
  - Average response time = (24 + 27 + 30) / 3 = 27

- Example 2
  - Same jobs but come in different order: P2, P3 and P1
  - Average response time = (3 + 6 + 30) / 3 = 13

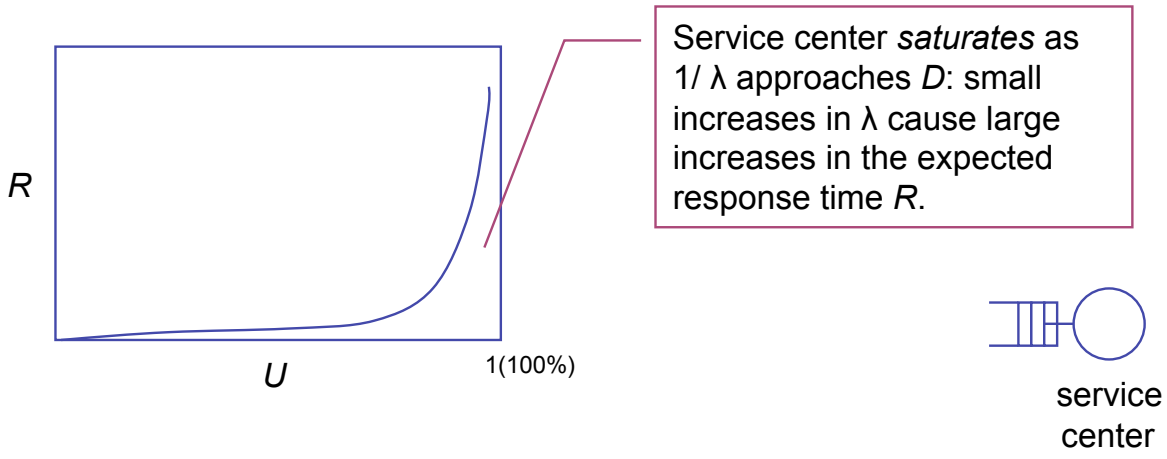
(Gantt Graph)

Behavior of FCFS Queues

Assume: stream of normal task arrivals with mean *arrival rate* λ.
Tasks have normally distributed *service demands* with mean *D*.

Then: *Utilization U = λD* (**Note**: 0 <= *U* <= 1)
Probability that service center is idle is 1-*U.*
"Intuitively", *R = D/(1-U)*

λ=1/60 (i.e. 1 task every 60s)
D = 30 (service time of 30s)
U=50%



Little's Law

For an unsaturated queue in steady state, queue length *N* and response time *R* are governed by:

Little's Law: N = λR.

While task ***T*** is in the system for *R* time units, λ*R* new tasks arrive.
During that time, *N* tasks depart (all tasks ahead of ***T***).
But in steady state, the flow in must balance the flow out.
(**Note**: this means that throughput *X* = λ).

Little's Law gives response time R = D/(1 - U).

Intuitively, each task ***T***'s response time $R = D + DN$.
Substituting λ*R* for *N*: $R = D + D\lambda R$
Substituting *U* for λD: $R = D + UR$
$R - UR = D \dashrightarrow R(1 - U) = D \dashrightarrow R = D/(1 - U)$

Why Little's Law Is Important

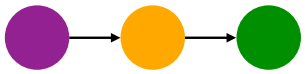
- 1. Intuitive understanding of FCFS queue behavior.
  - Compute response time from demand parameters (**λ**, ***D***).
  - Compute *N*: tells you how much storage is needed for the queue.
- 2. Notion of a *saturated* service center. If D=1: $R = 1/(1-\lambda)$
  - Response times rise rapidly with load and are unbounded.
  - At 50% utilization, a 10% increase in load increases ***R*** by 10%.
  - At 90% utilization, a 10% increase in load increases ***R*** by 10x.
- 3. Basis for predicting performance of *queuing networks.*
  - Cheap and easy "back of napkin" estimates of system performance based on observed behavior and proposed changes, e.g., *capacity planning*, "what if" questions.

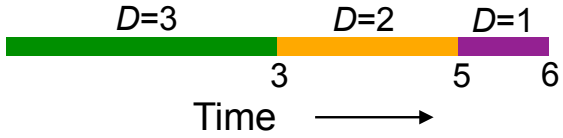
Scheduler Policy: Goals & Metrics of Success

- Response time or latency (to minimize the average time between arrival to completion of requests)
  - How long does it take to do what I asked? (R) Arrival -> done.
- Throughput (to maximize productivity)
  - How many operations complete per unit of time? (X)
- Utilization (to maximize use of some device)
  - What percentage of time does the CPU (and each device) spend doing useful work? (U)
    time-in-use / elapsed time
- Fairness
  - What does this mean? Divide the pie evenly? Guarantee low variance in response times? Freedom from starvation?
- Meet deadlines and guarantee jitter-free periodic tasks
  - real time systems (e.g. process control, continuous media)

Evaluating FCFS

- How well does FCFS achieve the goals of a scheduler?
  - throughput. FCFS is as good as any non-preemptive policy.
    - ….if the CPU is the only schedulable resource in the system.
  - fairness. FCFS is intuitively fair…sort of.
    - "The early bird gets the worm"…and everyone else is fed eventually.
  - response time. Long jobs keep everyone else waiting.

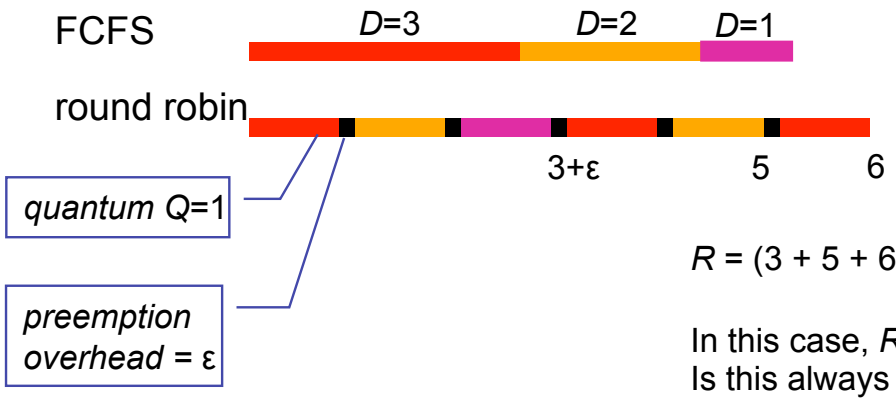




$R = (3 + 5 + 6)/3 = 4.67$

Preemptive FCFS: Round Robin

- Preemptive timeslicing is one way to improve fairness of FCFS.
  - If job does not block or exit, force an involuntary context switch after each quantum *Q* of CPU time.
  - Preempted job goes back to the tail of the ready list.
  - With infinitesimal *Q* round robin is called *processor sharing*.

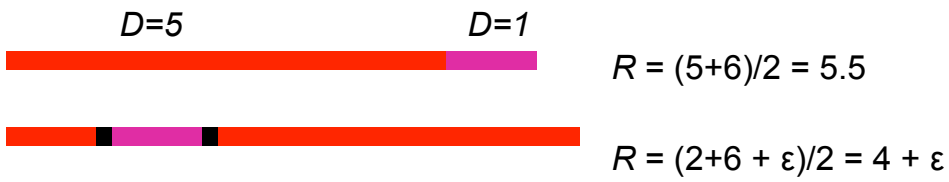


$R = (3 + 5 + 6 + \varepsilon)/3 = 4.67 + \varepsilon$

In this case, *R* is unchanged by timeslicing.
Is this always true?

Evaluating Round Robin

- Response time. RR reduces response time for short jobs.
  - For a given load, a job's wait time is proportional to its D.
- Fairness. RR reduces variance in wait times.
  - But: RR forces jobs to wait for other jobs that arrived later.
- Throughput. RR imposes extra context switch overhead.
  - CPU is only Q/(Q+ε) as fast as it was before.
  - Degrades to FCFS with large Q.



FCFS vs. Round Robin

- Example
  - 10 jobs and each takes 100 seconds
- What is the average response time for FCFS and RR?
  - FCFS: non-preemptive
  - RR: time slice 1sec and no overhead
- FCFS (non-preemptive scheduling)
  - job 1: 100s, job2: 200s, ... , job10: 1000s
- Round Robin (preemptive scheduling)
  - time slice 1sec and no overhead
  - job1: 991s, job2: 992s, ... , job10: 1000s
- Comparisons
  - Round robin is much worse (turnaround time) for jobs about the same length
  - Round robin is better for short jobs (relative to slice)
  - What is *good* about round robin?

CPU + I/O: Resource Utilization Example

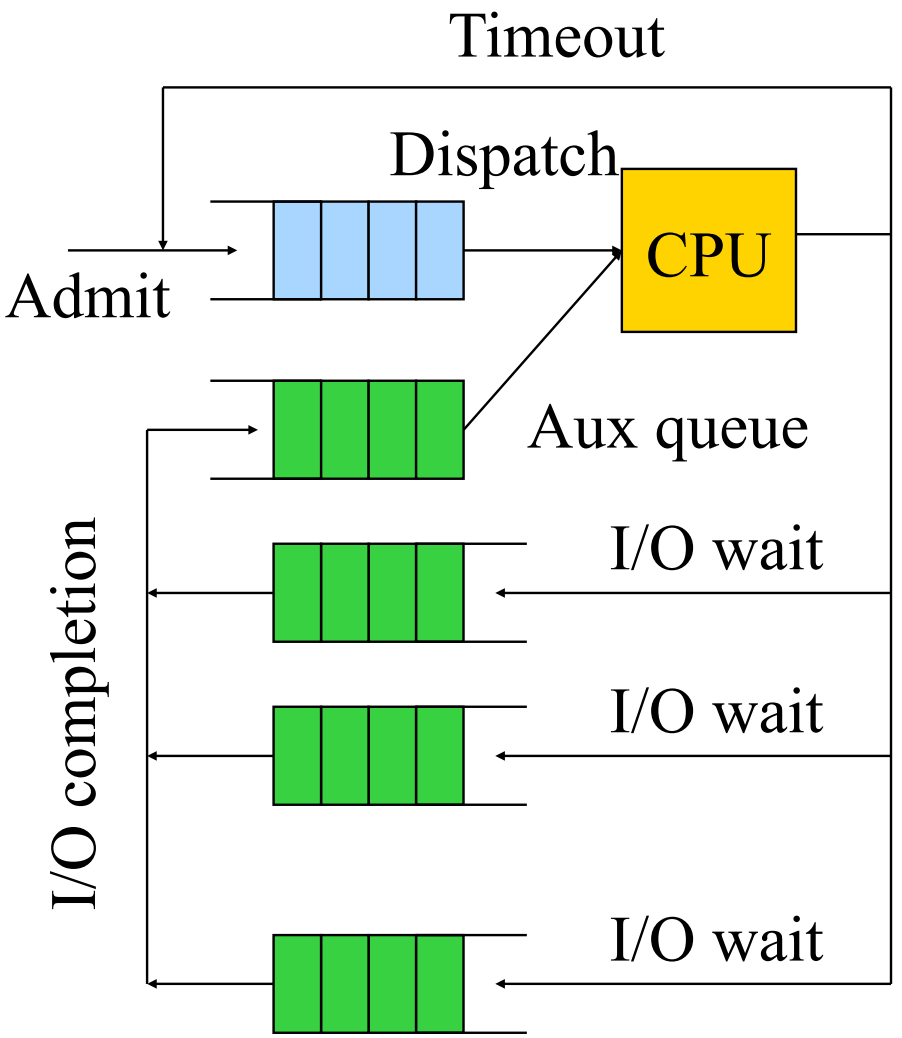
- A, B, and C run forever (in this order)
  - A and B each uses 100% CPU forever
  - C is a CPU plus I/O job (1ms CPU + 10ms disk I/O)
- Time slice 100ms
  - A (100ms CPU), B (100ms CPU), C (1ms CPU + 10ms I/O),
- Time slice 1ms
  - A (1ms CPU), B (1ms CPU), C (1ms CPU),
    A (1ms CPU), B (1ms CPU), C(10ms I/O) || A, B, …, A, B

- 100ms time slice:
  - CPU Util=201ms of CPU / (201 + I/O syscall) = ~99.9%
  - Disk Util: 10ms of disk usage over ~201 ms = ~5%
- 1ms time slice:
  - CPU Util = ~99.9%
  - Disk Util: 10ms disk usage over 15ms

What do we learn from this example?

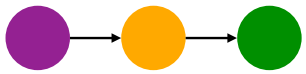
Virtual Round Robin

- Aux queue is FIFO
- I/O bound processes go to aux queue (instead of ready queue) to get scheduled
- Aux queue has preference over ready queue



Shortest Job First/STCF

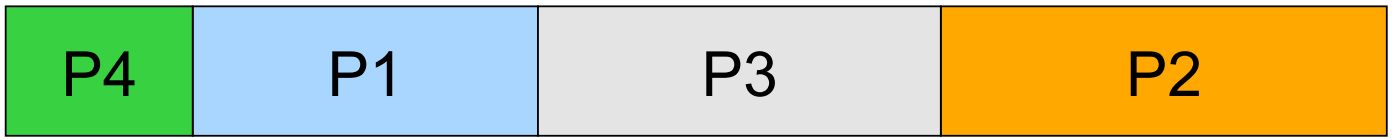
- Shortest Job First (SJF) aka Shortest Time to Completion First (Shortest Job First)
  - Non-preemptive
- is provably optimal if the goal is to minimize R.
  - Example: express lanes at the MegaMart
- Idea: get short jobs out of the way quickly to minimize the number of jobs waiting while a long job runs.
  - Intuition: longest jobs do the least possible damage to the wait times of their competitors.



$R = (1 + 3 + 6)/3 = 3.33$

SRTCF/SRPT

- Shortest Remaining Time to Completion First (Shortest Remaining Processing Time)
  - Preemptive version
- Example
  - P1 = 6sec, P2 = 8sec, P3 = 7sec, P4 = 3sec
  - All arrive at the same time

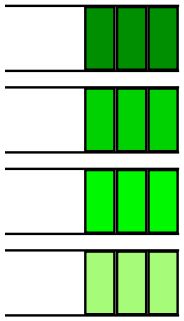


- Can you do better than SRTCF in terms of average response time?
- Issues with this approach?

Priority Scheduling

- Obvious
  - Not all processes are equal, so rank them
- The method
  - Assign each process a priority
  - Run the process with highest priority in the ready queue first
  - Adjust priority dynamically (I/O wait raises the priority, reduce priority as process runs)
- Why adjusting priorities dynamically
  - T1 at priority 4, T2 at priority 1 and T2 holds lock L
  - Scenario
    - T1 tries to acquire L, fails, blocks.
    - T3 enters system at priority 3.
    - T2 never gets to run!

Multiple Queues: One Approach



Priority	Time slices
4	1
3	2
2	4
1	8

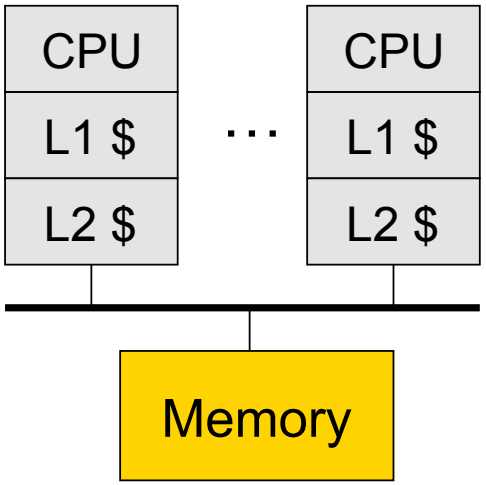
- Jobs start at priority=4 queue (high priority, but short time slice)
- If timeout expires, decrement to priority 3, and double time slice
- If timeout doesn't expires, stay or pushup one level

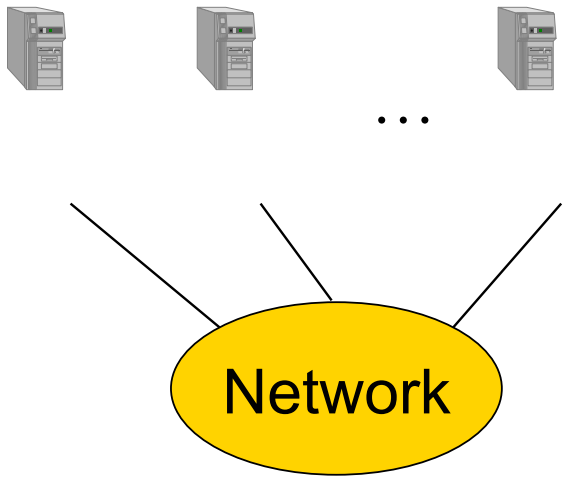
- What does this method do?

Lottery Scheduling

- Motivations
  - SRTCF does well with average response time, but unfair
- Lottery method
  - Give each job a number of tickets
  - Randomly pick a winning tickets
  - To approximate SRTCF, give short jobs more tickets
  - To avoid starvation, give each job at least one ticket
  - Cooperative processes can exchange tickets
- Question
  - How do you compare this method with priority scheduling?

Multiprocessor and Cluster





Multiprocessor architecture

- Cache coherence
- Single OS

Cluster or multicomputer

- Distributed memory
- An OS in each box

Multiprocessor/Cluster Scheduling

- Design issue
  - Process/thread to processor assignment
- Gang scheduling (co-scheduling)
  - Threads of the same process will run together
  - Processes of the same application run together
- Dedicated processor assignment
  - Threads will be running on specific processors to completion
  - Is this a good idea?

Real-Time Scheduling

- Two types of real-time
  - Hard deadline
    - Must meet, otherwise can cause fatal error
  - Soft Deadline
    - Meet most of the time, but not mandatory
- Admission control
  - Take a real-time process only if the system can guarantee the "real-time" behavior of all processes
  - The jobs are schedulable, if the following holds:

$$\sum \frac{C_i}{T_i} \leq 1$$

where C_i = computation time, and T_i = period

Rate Monotonic Scheduling (Liu & Layland 73)

- Assumptions
  - Each periodic process must complete within its period
  - No process is dependent on any other process
  - Each process needs the same amount of CPU time on each burst
  - Non-periodic processes have no deadlines
  - Process preemption occurs instantaneously (no overhead)
- Main ideas of RMS
  - Assign each process a fixed priority = frequency of occurrence
  - Run the process with highest priority
  - Only works if CPU utilization is not too high
- Example
  - P1 runs every 30ms gets priority 33 (33 times/sec)
  - P2 runs every 50ms gets priority 20 (20 times/sec)

Earliest Deadline First Scheduling

- Assumptions
  - When a process needs CPU time, it announces its deadline
  - No need to be periodic process
  - CPU time needed may vary
- Main idea of EDF
  - Sort ready processes by their deadlines
  - Run the first process on the list (earliest deadline first)
  - When a new process is ready, it preempts the current one if its deadline is closer
  - Provably optimal
- Example
  - P1 needs to finish by 30sec, P2 by 40sec and P3 by 50sec
  - P1 goes first
  - More in MOS 7.5.3-7.5.4

4.3 BSD Scheduling with Multi-Queue

- "1 sec" preemption
  - Preempt if a process doesn't block or complete within 1 second
- Priority is recomputed every second
  - P_i = base + (CPU_i-1) / 2 + nice, where CPU_i = (U_i + CPU_i-1) / 2
  - Base is the base priority of the process
  - U_i is process utilization in interval i
- Priorities
  - Swapper
  - Block I/O device control
  - File operations
  - Character I/O device control
  - User processes

Linux Scheduling

- Time-sharing scheduling
  - Two priority arrays: active and expired
  - 40 priority levels, lower number = higher priority
  - Priority = base (user-set) priority + “bonus”
    - Bonus between -5 and +5, derived from *sleep_avg*
    - Bonus decremented when task sleeps, incremented when it runs
    - Higher priority gets longer timeslice
  - Move process with expired quantum from active to expired
  - When active array empty, swap active and expired arrays

- Real-time scheduling
  - 100 static priorities, higher than time sharing priorities
  - Soft real-time

Windows Scheduling

- Classes and priorities
  - Real time: 16 static priorities
  - User: 16 variable priorities, start at a base priority
    - If a process has used up its quantum, lower its priority
    - If a process waits for an I/O event, raise its priority
- Priority-driven scheduler
  - For real-time class, do round robin within each priority
  - For user class, do multiple queue
- Multiprocessor scheduling
  - For N processors, normally run N highest priority threads
  - Threads have hard or soft affinity for specific processors
  - A thread will wait for processors in its affinity set, if there are other threads available (for variable priorities)

Summary

- Different scheduling goals
  - Depend on what systems you build
- Scheduling algorithms
  - Small time slice is important for improving I/O utilization
  - STCF and SRTCF give the minimal average response time
  - Priority and its variations are in most systems
  - Lottery is flexible
  - Real-time depends on admission control