In discussing processes and threads, we talked about context switching—
an interrupt occurs (device completion, timer interrupt)—a thread causes a trap or exception—
may need to choose a different thread/process to run.

We glossed over the choice of which process or thread is chosen to be run next—
"some thread from the ready queue".

This decision is called scheduling—
- scheduling is a policy—
- context switching is a mechanism

**Classes of Schedulers**

- **Batch**
  - Throughput / utilization oriented
  - Example: audit inter-bank funds transfers each night, Pixar rendering, Hadoop/MapReduce jobs
- **Interactive**
  - Response time oriented
  - Example: attu.cs
- **Real-time**
  - Deadline driven
  - Example: embedded systems (cars, airplanes, etc.)
- **Parallel**
  - Speedup-driven
  - Example: "space-shared" use of a 1000-processor machine for large simulations

We'll be talking primarily about interactive schedulers.

**Multiple levels of scheduling decisions**

- **Long term**
  - Should a new "job" be "initiated," or should it be held?
    - typical of batch systems
  - what might cause you to make a "hold" decision?
- **Medium term**
  - Should a running program be temporarily marked as non-runnable (e.g., swapped out)?
- **Short term**
  - Which thread should be given the CPU next? For how long?
  - Which I/O operation should be sent to the disk next?
  - On a multiprocessor:
    - should we attempt to coordinate the running of threads from the same address space in some way?
    - should we worry about cache state (processor affinity)?

**Scheduling Goals I: Performance**

- Many possible metrics / performance goals (which sometimes conflict)
  - maximize CPU utilization
  - maximize throughput (requests completed / s)
  - minimize average response time (average time from submission of request to completion of response)
  - minimize average waiting time (average time from submission of request to start of execution)
  - minimize energy (joules per instruction) subject to some constraint (e.g., frames/second)

**Scheduling Goals II: Fairness**

- No single, compelling definition of "fair"
  - How to measure fairness?
    - Equal CPU consumption? (over what time scale?)
    - Fair per-user? per-process? per-thread?
    - What if one process is CPU bound and one is I/O bound?
- Sometimes the goal is to be unfair:
  - Explicitly favor some particular class of requests (priority system), but...
  - avoid starvation (be sure everyone gets at least some service)
The basic situation

- Schedulable units
- Resources

Scheduling:
- Who to assign each resource to
- When to re-evaluate your decisions

When to assign?

- Pre-emptive vs. non-pre-emptive schedulers
  - Non- preemptive
    - Once you give somebody the green light, they’ve got it until they relinquish it
    - An I/O operation
    - Allocation of memory in a system without swapping
  - Pre-emptive
    - You can re-visit a decision
    - Setting the timer allows you to preempt the CPU from a thread even if it doesn’t relinquish it voluntarily
    - In any modern system, if you mark a program as non-runnable, its memory resources will eventually be re-allocated to others
    - Re-assignment always involves some overhead
    - Overhead doesn’t contribute to the goal of any scheduler

- We’ll assume “work conserving” policies
  - Never leave a resource idle when someone wants it

Before we look at specific policies

- There are some simple but useful “laws” to know about …

  - The Utilization Law: \( U = X \times S \)
    - Where \( U \) is utilization, \( X \) is throughput (requests per second), and \( S \) is average service requirement
    - Obviously true
    - This means that utilization is constant, independent of the schedule, so long as the workload can be processed

  - Little’s Law: \( N = X \times R \)
    - Where \( N \) is average number in system, \( X \) is throughput, and \( R \) is average response time (average time in system)
    - Better average response time implies fewer in system, and vice versa

- Response Time at a single server under FCFS scheduling: \( R = S / (1-U) \)
  - Clearly, when a customer arrives, her response time will be the service time of everyone ahead of her in line, plus her own service time: \( R = S \times (1+A) \)
  - Assumes everyone has the same average service time
  - Assume that the number you see ahead of you at your instant of arrival is the long-term average number in line; so \( R = S \times (1+N) \)
  - By Little’s Law, \( N = X \times R \)
  - So \( R = S \times (1 + X \times R) = S + S \times X \times R = S / (1 - X \times S) \)
  - By the Utilization Law, \( U = X \times S \)
  - So \( R = S / (1-U) \)
  - And since \( N = X \times R \), \( N = U / (1-U) \)

(Not quite a law – requires some assumptions)

- Response Time at a single server under FCFS scheduling: \( R = S / (1-U) \)
  - Clearly, when a customer arrives, her response time will be the service time of everyone ahead of her in line, plus her own service time: \( R = S \times (1+A) \)
  - Assumes everyone has the same average service time
  - Assume that the number you see ahead of you at your instant of arrival is the long-term average number in line; so \( R = S \times (1+N) \)
  - By Little’s Law, \( N = X \times R \)
  - So \( R = S \times (1 + X \times R) = S + S \times X \times R = S / (1 - X \times S) \)
  - By the Utilization Law, \( U = X \times S \)
  - So \( R = S / (1-U) \)
  - And since \( N = X \times R \), \( N = U / (1-U) \)
Kleinrock’s Conservation Law for priority scheduling:
\[ \sum_{p} U_p \cdot R_p = \text{constant} \]
- Where \( U_p \) is the utilization by priority level \( p \) and \( R_p \) is the time in system of priority level \( p \)
- This means you can’t improve the response time of one class of task by increasing its priority, without hurting the response time of at least one other class.

**Algorithm #1: FCFS/FIFO**

- First-come first-served / First-in first-out (FCFS/FIFO)
  - schedule in the order that they arrive
  - "real-world" scheduling of people in (single) lines
    - supermarkets, McD’s, Starbucks ...
  - jobs treated equally, no starvation
    - In what sense is this "fair"?
- Sounds perfect!
  - in the real world, when does FCFS/FIFO work well?
    - even then, what’s it’s imitation?
  - and when does it work badly?

**FCFS/FIFO example**

- Suppose the duration of A is 5, and the durations of B and C are each 1
- average response time for schedule 1 (assuming A, B, and C all arrive at about time 0) is \((5+6+7)/3 = 18/3 = 6\)
- average response time for schedule 2 is \((1+2+7)/3 = 10/3 = 3.3\)
- consider also "elongation factor" – a "perceptual" measure:
  - Schedule 1: A is 5/5, B is 6/1, C is 7/1 (worst is 7, ave is 4.7)
  - Schedule 2: A is 7/5, B is 1/1, C is 2/1 (worst is 2, ave is 1.5)

**FCFS/FIFO drawbacks**

- Average response time can be lousy
  - small requests wait behind big ones
- May lead to poor utilization of other resources
  - if you send me on my way, I can go keep another resource busy
  - FCFS may result in poor overlap of CPU and I/O activity
    - E.g., a CPU-intensive job prevents an I/O-intensive job from doing a small bit of computation, thus preventing it from going back and keeping the I/O subsystem busy
- Note: The more copies of the resource there are to be scheduled, the less dramatic the impact of occasional very large jobs (so long as there is a single waiting line)
  - E.g., many cores vs. one core

**Algorithm #2: SPT/SJF**

- Shortest processing time first / Shortest job first (SPT/SJF)
  - choose the request with the smallest service requirement
- Provably optimal with respect to average response time
  - Why do we care about "provably optimal"?
SPT/SJF optimality – The interchange argument

- In any schedule that is not SPT/SJF, there is some adjacent pair of requests f and g where the service time (duration) of f, \( s_f \), exceeds that of \( g, s_g \).
- The total contribution to average response time of f and g is \( 2t_k + 2s_f + s_g \), which is smaller because \( s_g < s_f \).
- If you interchange f and g, their total contribution will be \( 2t_k + 2s_g + s_f \), which is smaller because \( s_g < s_f \).
- If the variability among request durations is zero, how does FCFS compare to SPT for average response time?

Algorithm #3: RR

- Round Robin scheduling (RR)
  - Use preemption to offset lack of information about execution times
    - I don’t know which one should run first, so let’s run them all!
  - Ready queue is treated as a circular FIFO queue
    - What signifies the end of a quantum?
    - Time-division multiplexing (time-slicing)
    - Great for timesharing
    - No starvation
  - Sounds perfect!
    - How is RR an improvement over FCFS?
    - How is RR an improvement over SPT?
    - How is RR an approximation to SPT?

RR drawbacks

- What if all jobs are exactly the same length?
  - What would the pessimal schedule be (with average response time as the measure)?
- What do you set the quantum to be?
  - No value is “correct”
    - If small, then context switch often, incurring high overhead
    - If large, then response time degrades
- Treats all jobs equally
  - If I run 100 copies of SETI@home, it degrades your service
  - How might I fix this?

Algorithm #4: Priority

- Assign priorities to requests
  - Choose request with highest priority to run next
    - If tie, use another scheduling algorithm to break (e.g., RR)
    - Goal: non-fairness (favor one group over another)
- Abstractly modeled (and usually implemented) as multiple “priority queues”
  - Put a ready request on the queue associated with its priority
- Sounds perfect!

Priority drawbacks

- How are you going to assign priorities?
- Starvation
  - If there is an endless supply of high priority jobs, no low-priority job will ever run
- Solution: “age” threads over time
  - Increase priority as a function of accumulated wait time
  - Decrease priority as a function of accumulated processing time
  - Many ugly heuristics have been explored in this space
Program behavior and scheduling

• An analogy:
  – Say you’re at the airport waiting for a flight
  – There are two identical ATMs:
    • ATM 1 has 3 people in line
    • ATM 2 has 6 people in line
  – You get into the line for ATM 1
  – ATM 2’s line shrinks to 4 people
  – Why might you now switch lines, preferring 5th in line for ATM 2 over 4th in line for ATM 1?

Residual Life

• Given that a job has already executed for X seconds, how much longer will it execute, on average, before completing?

Multi-level Feedback Queues (MLFQ)

• It’s been observed that workloads tend to have increasing residual life – “if you don’t finish quickly, you’re probably a lifer”
• This is exploited in practice by using a policy that discriminates against the old (with apologies to the EEOC)
• MLFQ:
  – there is a hierarchy of queues
  – there is a priority ordering among the queues
  – new requests enter the highest priority queue
  – each queue is scheduled RR
  – requests move between queues based on execution history

UNIX scheduling

• Canonical scheduler is pretty much MLFQ
  – 3-4 classes spanning ~170 priority levels
  – timesharing: lowest 60 priorities
  – system: middle 40 priorities
  – real-time: highest 60 priorities
  – priority scheduling across queues, RR within
  – processes with highest priority always run first
  – processes with same priority scheduled RR
  – processes dynamically change priority
  – increases over time if process blocks before end of quantum
  – decreases if process uses entire quantum
• Goals:
  – reward interactive behavior over CPU hogs
  – interactive jobs typically have short bursts of CPU

Scheduling the Apache web server SRPT

• What does a web request consist of? (What’s it trying to get done?)
• How are incoming web requests scheduled, in practice?
• How might you estimate the service time of an incoming request?
• Starvation under SRPT is a problem in theory – is it a problem in practice?
  – “Kleinrock’s conservation law”

(Work by Bianca Schroeder and Mor Harchol-Balter at CMU)
Summary

- Scheduling takes place at many levels
- It can make a huge difference in performance
  - this difference increases with the variability in service requirements
- Multiple goals, sometimes conflicting
- There are many "pure" algorithms, most with some drawbacks in practice – FCFS, SPT, RR, Priority
- Real systems use hybrids that exploit observed program behavior
- Scheduling is still important, and there are still new angles to be explored – particularly in large-scale datacenters for reasons of cost and energy