The scheduling problem (Chapter 9)

○ Decide which processes are allowed to run when.

○ Optimize throughput, response time, etc.

○ Subject to constraints including context switch time, swap time, etc.
What's a schedule?
Input: some processes contending for resources.
Output: a "schedule" of which process does what when.
A "scheduler" is anything that produces a schedule.
Some kinds of scheduling:

- **Short term:** which process, already in memory, gets to execute next?
- **Long term:** which commands or requests are "admitted" for later processing? A.K.A. "admission control"
- **Medium term:** which processes are asked to wait for others to finish.
- **I/O:** which pending I/O request should be handled first? (when writing disk blocks, which block should go first?)
Run-queue: a set of processes that are "ready to run". The scheduling problem: give each process in the run queue a time to execute that is
• Prioritized according to the importance of the process.
• Fair to other processes.
A model of queueing

λ: arrival rate: arrivals per second.
μ: service rate: jobs processed per second

1/λ: seconds between arrivals
1/μ: seconds to complete a job

If arrival rate = departure rate, we say that the system is in **steady state**.
A queueing model of scheduling:

Processes that aren't done get queued again. Processes that exit aren't queued again.
The most fundamental and useful result: Little's law

- Assume we have a **steady-state** system.
- Let $L$ be the **average number of requests pending** inside the system. $L = "load"
- Let $\lambda$ represent **average arrival (and exit) rate** for requests.
  Let $W$ be the **average time-in-system** for a request (i.e., entrance to exit). $W = "wait"
- Then $L = \lambda W$.

No distribution assumptions on arrival or service.
- Really easy to understand, but exceedingly hard to prove!
Using Little's law

If you know any two of average time-in-system, average requests-in-system, or average arrival rate, you can get the other one.

Example:

A web service receives about 100 requests per minute and you observe that it takes 1/2 second to service a request. How many requests are pending at a time?

\[ \lambda = \frac{100}{60} \text{ requests/second} \]

\[ W = \frac{1}{2} \text{ second} \]

\[ L = \lambda W = \frac{5}{6} \text{ requests in system, on average.} \]
Another example of Little's law
Round-robin scheduling takes as input a sequence of runnable processes, processes them, and puts them back into the queue (closed loop)

If there are 6 runnable processes and a time-slice of 1/100 second, what is the time in system for a process to get one slice?

λ = 100 arrivals/second
L = 6 slice requests in system
L = λ W, so W = L/λ = 6/100 of a second between slices.

Trivial result, but in general, Little's laws are powerful.
Suppose there are three processes in the run queue with priorities 1, 2, and 3, and their priorities are the same as the number of slices they get in each sequence of 6 slices. Each slice takes 100 ms. What is the average rate at which processes are processed?

$L=3$ things in system at a time
$W=time$ for total of 6 slices $= 600ms$
$\lambda=L/W = 3/(600/1000) = 3000/600 = 5$ per second.
There are a number of related laws to the main Little's Law: [http://en.wikipedia.org/wiki/Little's_law](http://en.wikipedia.org/wiki/Little's_law)

Assume that the system is in **steady state**

Let $\lambda =$ average arrival rate

= average departure rate

Let $W =$ average time in system ("wait")

Let $L =$ average jobs ("load")

Then $L=\lambda W$

But the system has two parts:

A processing unit.

A queue.

But looking only at the queue:

$L-1 =$ average number of jobs waiting for processing.

$(L-1)/\lambda =$ average time in queue
Let $\mu = \text{average service rate (requests per second)}$

Then $1/\mu = \text{average service time (seconds per request)}$

Then, assuming serial processing:

$W-(1/\mu) = \text{average time spent in queue before processing}$

$W-(1/\mu) = (L-1)/\lambda$

In general, any time you can put a box around a subsystem and the box is in balance, Little's law applies.

Mnemonics:

$W=\text{wait}$

$L=\text{load}$

$\mu = \text{service rate}$

$\lambda = \text{arrival rate}$
To understand scheduling, it's helpful to start with a physical metaphor.

**work** = cpu cycles expended on a goal. Analogous to "distance traveled".

**throughput** = \( \text{work} / \text{wallclock time} \) = work per unit time. (= work cycles / second)

**efficiency** = \( \text{work} / \text{total cycles} \) = % of time used wisely.

**processor speed** = \( \text{total cycles} / \text{wallclock time} \)

**overhead** = \( 1 - \text{work} / \text{total cycles} \) = % of time "wasted": used for purposes other than accomplishing computational goals.

**efficiency + overhead = 1**

**throughput = efficiency \times processor speed**

**throughput and response time are independent variables.**
The run queue is not a COMP15 queue

In scheduling theory, a queue is any mechanism that takes in jobs and releases one at a time, according to some "queueing discipline".

Some queueing disciplines:
- First-in, First-out (FIFO)
- Prioritized - chosen according to univariate priority.
- Strategized - chosen according to multivariate rubrics.
- Randomized - chosen at random.
Inside the google search engine
Thursday, November 03, 2011  4:39 PM

Scheduling Page 16
Fairness

○ For N processes, how equally is work distributed?
○ Only meaningful for processes at the same priority.
○ Some authors define unfairness as the difference between maximum and minimum "work" runtimes allocated at the same priority level.
  ○ 0 unfairness: everything is equal
  ○ high unfairness: things are unequal

Opposite of fairness is **starvation**: one or more processes don't get to run at all.
Goals of scheduling

Minimize unfairness and latency.
Maximize throughput and efficiency.

Except that these are **conflicting** goals!
Common to all short-term strategies

A **system clock** that interrupts process execution at predetermined intervals and invokes the scheduler.

Whenever a process **blocks**, control is returned to the scheduler even if its scheduling interval is not done.

The scheduler only acts on processes in the **run queue**, i.e., processes ready to execute. When a process becomes runnable, **it is not executed immediately**; it is returned to the run queue for eventual execution.
The basic conflict in scheduling
maximize efficiency => minimize scheduler overhead
and context switches.
reduce unfairness and latency => make context
switches more frequent!

The fairest scheduler in the world has large overhead,
while the most efficient one is very unfair, i.e., one
process runs until it must sleep.
1. Batch scheduling is not fair. Waiting until the end of each process before scheduling the next makes user response times really long.

2. Round-robin is not fair. When a process blocks due to I/O, it loses the rest of its scheduling cycle. There is never any point at which it catches up.
Scheduling is an active research topic
Satisfy particular goals....
Better, stronger, faster....
Two Linux schedulers of note:
"The O(1) scheduler" (<2009)
"The Completely Fair Scheduler (>2009)
The O(1) linux scheduler (pre-2009)
http://joshaas.net/linux/linux_cpu_scheduler.pdf

**Epoch:** a scheduling event in which every process in the run queue gets a timeslice. **Priority array:** an array of linked lists of tasks, where each linked list corresponds to a priority.

**Preconditions:**
Two priority arrays:
- `waiting[]`: processes to be scheduled in this epoch.
- `done[]`: processes that have been scheduled in this epoch.

**The algorithm:**
Repeat forever
While `waiting[]` is non-empty,
   choose a process from the highest priority list in `waiting[]`.
   schedule it for a time slice with a length determined by its priority.
   move it to the `done[]` array.
end while
Swap `waiting[]` and `done[]` roles and start over.
End repeat

Why is this $O(1)$?
- $O(1)$ to choose a non-empty list.
- $O(1)$ to unlink an element from the head.
- $O(1)$ to relink it elsewhere.
Scheduling priority

In all Linux scheduling, each Linux process has a static priority between -19 (highest priority) and +20 (lowest priority).

In O(1) scheduling, there is also a **dynamic priority** that is +/- 5 of a process's static priority.

In O(1) scheduling, a process's priority determines the length of its time slice:

```
#define BASE_TIMESLICE(priority)
(MIN_TIMESLICE + \n ((MAX_TIMESLICE - MIN_TIMESLICE) * \n (MAX_PRIO-1 - priority) / (MAX_USER_PRIO-1)))
```

In other words, linearly proportional to priority.

Static priority is set via the nice(1,2) command and system call.

dynamic priority is increased for I/O bound processes that are blocked waiting for I/O
decreased for CPU-bound processes that compute without blocking.

This is a dirty hack to address the intrinsic unfairness of O(1) scheduling.
From whence comes unfairness?

Monday, November 6, 2017  5:18 PM

[Hand-drawn diagram with annotations]

Heats of $1/0 \rightarrow$ smaller skew
Why O(1) scheduling is unfair

Processes that block a lot get much less CPU time.
When they unblock, they're given only what they deserve if they didn't block at all.
There is no "catch up" for CPU time when a process unblocks.

So, an I/O intensive process gets -- on average -- much less CPU time than a compute-bound process.
... because it's my ball...

O(1) scheduling
  minimizes overhead
  lacks fairness

How could we
  compensate for processes that wait a lot.
  give them higher priority
  according to a process lifecycle
definition of fairness?
The Completely Fair Scheduler (Linux > 2009)

https://en.wikipedia.org/wiki/Completely_Fair_Scheduler

Track **total runtime** allotted to a process. Store each priority of run queue as a priority queue where the priority is **time spent so far**. At each point, the process that has had the least runtime so far is scheduled. A process that hasn't run lately can "catch up" by consuming several contiguous slices.

Implementation:
A runqueue for each process priority.

**Each runqueue is a red-black tree whose key is time spent running so far.**

For each epoch:
For each runqueue, highest to lowest priority:
Repeat until runqueue's time allotment ends:
Schedule the leftmost process (least key) for a slice.
Remove it from the tree.
Update its time spent computing.
Reinsert it into the tree.
End repeat
End for

Basic idea can be applied to
Fairness of processes
Fairness of process groups.
Fairness among users
End of lecture on 10/31/2018

Wednesday, October 31, 2018    5:18 PM
O(1): simple, but not fair.
CFS: fair, but not simple: If \( n \) is the number of processes in all run queues,
    fairness is bounded by \( O(n) \)
    runtime of scheduler is bounded by \( O(\log n) \) per time slice, \( O(n \log n) \) per epoch.

Few other details
    Fairness is assured over a small window of epochs.
    That window rolls over as time advances.
Other short-term scheduling strategies

**Round-robin**: give a slice of time to each runnable process.

**Prioritized round-robin**: give longer or more frequent slices to higher-priority processes.

**Run-until-blocked**: process in memory gets priority until it must sleep.

**Pre-emptive**: higher-priority processes exclude lower-priority ones until finished, then lower-priority processes can continue.

**Real-time**: schedule according to behavior of external devices rather than an internal clock.
Long-term scheduling
  ○ Also known as "admission control".
  ○ A big research topic in grid/cloud/autonomic computing.

While a short-term scheduler decides what to do with active processes, a long-term scheduler decides when a job will become a process.

Questions asked by a long-term scheduler:
  ○ How many concurrent processes can the system handle?
  ○ How much memory is needed for each?
  ○ How important is response time versus throughput versus efficiency?
  ○ How important is it to eventually admit a job?
An apparent paradox:
Marketing data suggests that if you make a user wait more than 2-5 seconds for a travelocity search, the sale is lost.

For optimal business, it is better to drop the late request and process the not-yet-late ones.

This is an admission-control policy!
Long-term scheduling factors

○ **Deadlines**: how long of a wait time is acceptable?
  - Example: weather forecasting must be finished before the 7pm news.
  - Example: rendering of today's animation must be complete by tomorrow at 8 am.

○ **Throughput**: how much of the cpu can one use via latency hiding?
  - Example: How fast can one finish n predictions of geological substructure, where n is large?
  - Example: how fast can we render a scene in a Pixar movie?
  - Example: How fast can we complete both accounts payable and accounts receivable for the night?
Typical admission control algorithm
  - Predict resource utilization.
  - Predict overlap and latency hiding.
  - Admit process only if within bounds.
  - Otherwise, defer process until there are enough resources to assure completion, or delete if deadlines are past.

Application: grid computing
  - **Process**: a parallel program that runs on a grid.
  - **Resource utilization**: # of nodes required, amount of memory on each node (I/O bandwidth to each node)
  - Resources must be known in advance

Application: cloud computing
  - **Process**: services one request
  - Resource utilization: sum of current active processes + new one.
  - Resource needs known in advance.
  - **Capacity**: some prediction of how many processes one can handle.
  - **Admission**: only admit jobs you think you can handle.
  - **Elasticity**: add processing power as number of concurrent requests increases.
Tension between "theory" and "practice" 
"theory": things we can understand and analyze.
  simple scheduling algorithms
  models of task arrival, completion, and resource utilization 
"practice": what works.
  much more complex mechanisms than can be analyzed.
  much more complex inputs than can be modeled. 
"dirty hacks that work"

Example: admission control algorithms
theory: predicts what happens when all instances of a problem are autonomous.
practice: **instances aren't autonomous!**