Scheduling is a matter of great controversy.
  - Should it be fair? (O(log n))
  - Should it instead be fast? (O(1))
As well,
  - Should it maximize throughput? (batch)
  - Should it minimize response time? (some variant of round-robin)

Today, we develop some tools for understanding scheduling.
  Mathematical estimates.
  Experimental approaches.
Scheduling is viewed as a queueing problem, where the queue represents work to be accomplished and scheduling is the act of choosing some task on the queue to process.

Queues are represented as follows:

The most fundamental theoretical result for a queueing system is **Little's law**: If

- a queueing system is in balance, in the sense that the average entry rate = the average exit rate.
- \( \lambda \) represents the average input and exit rate.
- \( L \) represents the average jobs in system (either in the queue or being processed)
- \( W \) represents the average wait time between entry and exit.
- Then \( L = \lambda W \)
In Little's law \( L = \lambda W \), \( \lambda \), \( W \), and \( L \) are all averages.

The fact that there are \( n \) things in a queue doesn't mean automatically that it will take \( n \) times the amount of time for one thing.
  - The rate could vary.
  - The processing time could be shorter or longer in some special case.
  - Processing times aren't necessarily independent; the fact that a task precedes another can change how long the later task takes.
Assume steady state => system is in balance.

1. Make **cleverly chosen assumptions** on what's happening, statistically.
2. Make **predictions** based upon those assumptions

3. **Critique the accuracy** of the assumptions.
Good news: Little's law is very general and applies to a majority of systems we can think of.

Bad news: going beyond Little's law requires some kind of distribution assumption as to how jobs will arrive and how processing will be done.

Effectiveness of scheduling is very dependent on how and when processes spawn, block, unblock, and die!

spawn, unblock: enqueue in run queue.
die: permanent dequeue.
We can analyze the response of systems with **queueing theory**.

Mathematical model of batch processing.
Overly precise models that almost model reality.
Precise answers to a number of specific questions.
No information in some **realistic cases**.

Caveats

Batch environment.
"Fairness" is irrelevant.
Steady-state behavior.

Queueing theory assumptions:

- **Exponential (or Poisson) inter-arrival time** distribution for jobs.
- **Exponential (or Poisson) service time** distribution for completing jobs
- **subsystems are independent of one another**.
- arrivals are **independent**.
- processing of jobs is **independent**.

**Whoa there! What is going on here?**
An ideal queue

Thursday, November 18, 2010  4:45 PM

Diagram:

- Exponential arrival rate
- FCFS
- Exponential service rate
Real systems aren't like that

Not memoryless
Not Poisson or exponential arrivals.
Arrivals aren't even independent (in worst case, a flash crowd is the opposite of independent arrivals!). Processing is certainly not independent.

=> everything I will talk about today is an approximation.

The reason this particular approximation is important is that it is easy to compute.

It typically underestimates the processing time needed (but there are some cases where it over-estimates).
Why this model is important

It is analogous to big-O notation in algorithms, but for the vastly different field of capacity planning.

• While **Algorithms** is the study of how to make things run faster without buying a bigger, faster computer,
• **Capacity Planning** is the study of how to buy a sufficiently fast computer to run an algorithm acceptably.

Queueing theory allows us to approximate the effects of "buying a faster computer" without actually buying it or trying out the software on it.

Much of this lecture is taken from Menasce et al, **Performance by Design**, [https://www.amazon.com/Performance-Design-Computer-Capacity-Planning/dp/0130906735](https://www.amazon.com/Performance-Design-Computer-Capacity-Planning/dp/0130906735)
Exponential inter-arrival time distribution:
- Models arrival time as a set of completely independent trials.
- Let $t$ be the time of arrival of the next event, as a difference between now and then. Then:
  $$\text{Prob}(\text{arrival time } t \leq q) = 1 - e^{-\lambda q}$$

$\lambda$: the rate of the process; a measure of how fast things happen.
  - Small $\lambda$: events are spaced far apart.
  - Large $\lambda$: events are spaced close together.
Facts about exponential distributions
- Mean = $1/\lambda$
- Standard deviation = $1/\lambda$
- Memoryless: $\text{Prob}(T \leq x+t | T > t) = \text{Prob}(T \leq x)$
  - $T =$ time of arrival
  - $t =$ time you started looking for an arrival
  - In other words, waiting longer ($T > t$) doesn't change things.

Some useful properties (of memorylessness):
- Suppose $X, Y, Z$ are independent exponential inter-arrival time processes with parameters $\lambda_X, \lambda_Y, \lambda_Z$. Then the union of $X, Y, Z$ is exponential with parameter $\lambda_X + \lambda_Y + \lambda_Z$
- In other words, the union of a set of exponential arrivals is exponential, with the obvious parameter.
- This is equivalent mathematically to "memorylessness".

Examples:

Suppose
- I type at an average of 100 chars/min
- My mother types at an average of 200 chars/min
- Our inter-arrival times follow the exponential distribution.

Then
- Both of us together type at an average of 300
chars/min with exponential inter-arrival time.

Translating into OS terminology,
lambda is a rate, like "5 per second". 
1/\lambda is a time, "1 second per 5". 
users' rates add provided that the users are acting independently and without memory.

Actually a very simple idea:
if 5 people type an average of one command/10 seconds, then
the total arrival rate is 5 commands per 10 seconds, and
a command arrives on average every 2 seconds...
Exponential service time distribution:
\[ \text{Prob(Processing time } \leq t) = 1 - e^{-\mu t} \]
- \( \mu = \text{parameter of jobs = service rate = jobs/time} \)
- Mean = \( \frac{1}{\mu} \)
- Standard deviation = \( \frac{1}{\mu} \)

Memoryless: \( \text{Prob}(T \leq x+t|T>t) = \text{Prob}(T \leq x) \)

In OS terms:
- There is a "service", e.g., "writing to disk", that takes a given average time \( \frac{1}{\mu} \)
- Example: page in or page out.
- Can predict, within limits, a reasonable \( \mu \) for swap in.
Same formula, different meanings:

○ A exponential inter-arrival time process is one in which the **inter-event arrival times** are exponential.

○ An exponential service process is one in which the **service time itself** (and not the time between service requests) follows an exponential distribution.

○ Same equation, two meanings.
  - \( \text{Prob(} \text{next arrival before time } t \text{)} = 1 - e^{-\lambda t} \)
  - \( \text{Prob(} \text{processing time less than } t \text{)} = 1 - e^{-\mu t} \)

○ This symmetry makes analysis a lot easier than if these were different!
\( \lambda \) is exponential
- exponentially distributed inter-arrival times
- independence of arrival events: "memoryless" (first M)

\( \mu \) is exponential
- exponentially distributed service times
- independence of service times: "memoryless" (second M)

Watch out!
These are ideal conditions
In reality, arrivals are not necessarily exponentially distributed.
  - There is coupling between arrivals
    - Example: web hits for breaking news
    - Example: flash crowds, e.g., slashdot.
In reality, service times are not necessarily independent
  - Shared resources, e.g., disk and memory.
  - Context-switching overhead.
Combination/multiplexing:

\[ \pi_1 + \pi_2 + \ldots + \pi_n = 1.0 \]
\[ 0 \leq \pi_i \leq 1.0 \]
\[ \pi_i \text{ is a constant probability of taking branch } i. \]

Note that this works because the processes are *independent* and *memoryless*, and does not hold if they are not!

Splitting/demultiplexing:

\[ \sum \pi_i = 1 \]
Kinds of queues
  o Kendall notation: A/B/c/k/m/Z
  o A=arrival process; default is M ("Memoryless", Exponential inter-arrival times)
  o B=service process; default is M ("Memoryless", Exponential service time)
  o c=number of servers; default is 1
  o k=maximum queue size; default is infinite
  o m=customer population; default is infinite
  o Z=name of queueing discipline, default is FCFS

Previous queue is M/M/1
Here's M/M/3:
Some startling mathematical results:

For M/M/1 queue,
- Queue reaches steady state only if \( p = \frac{\lambda}{\mu} < 1 \)
- Prob(queue length is \( k \)) = \( p^k (1-p) \) where \( p = \frac{\lambda}{\mu} \)
- The mean jobs in system are \( n = \frac{p}{1-p} \) (including jobs in queue and jobs being processed).
- The mean waiting-line length (jobs not yet being served) is \( w = \frac{p^2}{1-p} \) (including jobs in process of being queued and dequeued).

Little's laws for M/M/1 queue:
- Mean time in system = \( \frac{n}{\lambda} \)
- Mean wait time before service = \( \frac{w}{\lambda} \)

Key to proofs: principle of balance:
- Input rate = output rate if system is in balance (\( \lambda < \mu \))
- Probability of increasing queue size by 1 = \( \frac{\lambda}{\mu} \)
- Probability of decreasing by 1 = \( 1 - \frac{\lambda}{\mu} \)


Question: if \( p = \frac{\lambda}{\mu} > 0.5 \), won't queue blow up?
Answer: no, because the probability of high queue sizes becomes vanishingly small.
- The events of growing by 1 are independent.
- Growing by 2 has probability \( p^2 < p \)
- Growing by 3 has probability \( p^3 < p^2 \)
The cosmic thing here:

- Given estimates of input rate and service rate, it is possible to derive an estimate of throughput.
- This means that if one knows \( \lambda \), and knows desired time in system, then one can compute \( \mu \)!
- **One can compute how big and fast one's computer should be!**
Generalization: $c$ processors:

Steady state: $\lambda/c\mu < 1$

$S_0 = \text{Prob}(\text{queue empty}) = \frac{1}{\sum_{i=0}^{c-1} \frac{(\lambda/\mu)^i}{i!} + \frac{1}{c!} \frac{(\lambda/\mu)^c}{1 - \lambda/(c\mu)}}$

Mean time prior to service =

$$\frac{(\lambda/\mu)^c S_0}{c! c\mu (1 - \lambda/(c\mu))^2}$$
Limiting case: M/M/∞

Infinite servers, memoryless arrival (monkeys on keyboards)
- Waiting line length = $w = 0$
- Mean time in system = $1/\mu$
- Mean jobs in system = $\lambda/\mu$
- Value of this: how well one can do if one increases processing power without bound!
Networks of queues
  ○ Can connect queueing systems together into networks
  ○ Output of one system is input to another.
  ○ Surprising result: can compute the parameters of the network provided one knows the distribution parameters of inputs and service parameters of the queues.

Key concept: "stability" is simpler to analyze than dynamic state.
  ○ To achieve stability, system must come into "equilibrium" subject to "balance equations".
  ○ Key equilibrium: input=output.
  ○ In absence of this balance, system cannot be analyzed easily.
  ○ When equilibrium is present, analysis is easy!
If we can compute the labels on the edges, then Little's laws and elementary queueing theory tell us the time in system, and serial times sum, while parallel times average.
Create a graph representing processing of jobs.
Label the edges
Compute individual time-in-system
Sum and average to get total time-in-system.

Two basic cases for labeling edges via the principle of balance:
An open network with feedback.
A closed network.
First case: no feedback

\[ \frac{X}{C \mu} < 1.0 \]

Key: \( X \) outputs because output is Exponential when and if system is in steadystate.

\[ \lambda \]

\[ \lambda_1 + \lambda_2 \]

\[ \lambda_2 \]

\[ \lambda_1 \]

\[ \prod \lambda \leq \pi \]

\[ \prod \lambda_2 \leq \pi \]

\[ \prod (\text{choose branch}) = \pi_1 \]

\[ \prod (\text{choose branch}) = \pi_2 \]

\[ (\pi_1 + \pi_2) = 1.0 \]
Second case: simple feedback.

\[ \lambda + (1 - \psi)e = e \]

Equilibrium constraint:
\[ e = \lambda + (1 - \psi)e \Rightarrow e = \frac{\lambda}{\psi} \]

Problem: what is raw output rate \( e \)?

Problem limits:

1. if there are two outcomes with probability \( \psi \) and probability \((1-\psi))\), then output rates are \( e\psi \) and \( e(1-\psi) \) for the two cases.
2. if rates \( \lambda, \delta \) are added, resulting rate is \( \lambda + \delta \)
3. if there is feedback with probability \((1-\psi))\), then input rate = output rate, so \( e = \lambda + (1-\psi)e \)
4. Thus \( e = \lambda/\psi \)
Bad news: no information on "time in system" from balance equations.
Case 2: feedback in open networks (input from outside, output to outside)
Queues 1 to q with server constants $\mu_i, c_i$ servers.
Each queue gets input from all other queues, and input from outside world.
Prob(queue i gets input from queue j) = $\pi_{ji}$
Prob(queue i output exits network) is $\psi_i$

Each queue's view of the world:

Note:
- $\lambda_k, \psi_k, \pi_j, \mu_{jk}, c_k$ constant.
- Variables are $e_1, \ldots, e_q$
- $q$ equations in $q$ unknowns $\rightarrow$ unique solution.
- If this solution exists, it is the equilibrium point! ("Jackson's algorithm")
- Model of multiple-priority processing.
Two parts

The equations for M/M/c determine what happens inside a black box.
The principle of balance determines labels on the edges outside the box.

Thus

We can predict the behavior of a fairly complex network.
A whole course is possible
(formerly on the qualifying examination
for a Ph.D. in computer science)
(fortunately, no longer!)

What we understand
Queueing of load-independent servers
(tasks in queue don't affect response
time)
Queueing of load-dependent servers
(tasks in queue affect response time)
Distribution-independent results for very
simple networks.
Exponential-interarrival (Poisson)
memoryless inter-arrival times

What we don't understand
Models other than FCFS.
Non-exponential response.
Non-equilibrium behavior (at least, from
this perspective).
"Non-product systems" : behavior of the
whole is not a simple addition of behavior
of the parts.
Etc.

Solution: simulation.