Last time, we covered many of the important facets of an operating system:

An **operating system** mediates between **hardware** and **processes** that use the hardware.

A **process** is a running version of a **program**, and has an **execution context**, including **register values** and **memory map**.

**Context switching** (between processes) accomplishes **latency hiding**, where one process accomplishes work while another is waiting for something.

**System (kernel)** and **user memory** set limits on what processes can and cannot do.

A process communicates with an OS (and devices) via **system calls**.

An OS sends messages to a process via **signals**.

A **device driver** serves as a **single point of contact** for external devices, maintains a definitive concept of **device state**, and avoids **state botches**.
So, writing an operating system is relatively easy!

**Compartmentalize** the concept of **device state** in **drivers**.

Figure out how **processes** call **drivers** via **system calls**.

Figure out how to **load and run processes**.

Figure out how to **send a signal to a process**.

Figure out how **processes share resources** (time, space, bandwidth, cores, ...).

Done! (sort of, but I am lying!).
Except that these basic steps hide some difficult quandaries:

Design of an OS is a **multivariate optimization problem**. Can't maximize both **time efficiency** and **interactivity/sharing**. Can't streamline processing of both **generic** and **special-purpose computation**.

...etc...

So

Some **difficult design decisions** need to be made. There is an intimate relationship between **OS code** and **hardware capabilities**. Everything has to run **really, really fast!**
This lecture, we will discuss:

The notion of **time** in an OS
How the OS manages **multiple processes**
How to measure whether the **system is busy**.
**Scheduling** and **context switching**.
Three notions of time:

**Wallclock time**: the usual notion of time, in elapsed seconds.

**User time**: the time spent in a process.

**System time**: the time spent in system calls, at the request of a process.

(OS time or "overhead": the time spent outside processes)
I will frequently express time in terms of Gantt charts:

- **Time** is on the X axis.
- **Entities** are on the Y axis.
- **Activities** are boxes whose vertical extent indicates the entities involved, and whose horizontal extent indicates the time that passed.
In talking about time, I will often use the word "state" and the word "event". A **state** is a situation that persists over some time period. An **event** signals a change in states and has no duration in time. A **state transition diagram** has states represented as nodes and events as arrows between nodes.
There are two concepts of time in this diagram:
How long one stays in each circle.
Elapsed time between some two chosen arrows occurring.
There are two conceptions of time in an OS

Time for which a state persists.
Time between two events (that might not be otherwise related).
The notions of time are measured differently:

**Wallclock time** is typically measured via a **hardware clock** that resides within the computer. **User Time** and **System Time** are typically measured by checking what the CPU is doing at periodic intervals (typically, every 1/100 or 1/1000 of a second)

If the CPU is **executing user code for a process**, increment that process's **user time** by one tick. If it is **executing a system call for a process** in the OS, increment the process's **system time** by one tick. If no process is running, count that time toward the OS itself ("idle").

Note: in multi-core operating systems, **more than one process can be executing during a tick.**
The calls and algorithms we are going to study today are of two kinds:

- measuring by polling.
- measuring by adding high-resolution times.

The primitive linux that we started with only supported polling.

- there weren't good hardware clocks.
- time for measuring process states precisely was excessive.
- SO, there are system calls that are based upon counting ticks.
- EVEN though the modern operating system uses the hardware clock!

For now, I will describe the jiffie system.
The quantized notion of time

Thursday, September 08, 2011   2:33 PM
Example: measurement versus timing of sleeps

Thursday, September 08, 2011        2:38 PM

[Hand-drawn diagram with time markings and labels]
Measuring time

In the shell:
  times ./a.out

In C:
  http://www.cs.tufts.edu/comp/111/examples/Time/times.c
  http://www.cs.tufts.edu/comp/111/examples/Time/getrusage.c
  http://www.cs.tufts.edu/comp/111/examples/Time/clock_gettime.c
Some time functions use seconds. Others use seconds + **milliseconds**. Still others use seconds + **nanoseconds**. Reason: legacy code and increasing needs for time accuracy.
"times" can't measure anything under a clock tick. Fast operations seem to take zero time. To measure something that happens very quickly, must repeat it many times, then divide:

```c
struct tms start, finish;
times(&start);
for (i=0; i<1000000; i++)
    { something_to_measure(); } 
times(&finish);
clock_t elapsed = (finish.tms_utime - start.tms_utime) + (finish.tms_stime - start.tms_stime);
```
Some time quandaries

Time measurements are probabilistic, not deterministic.
Measurements depend upon exactly when a process is running or waiting for input.
The times system call itself takes time.
The act of repeating an operation takes time.
The concept of process state

Every process has a "state"
User and system time are logged whenever the process is in the state "running".
The simplest concept of process state:
running: accomplishing useful work.
sleeping: waiting for something to happen.

Examples of sleeping:
Waiting for user input.
Waiting a set amount of time (sleep() system call).
Waiting for another process to complete.
"Ready"

In a single core architecture, only one process can run at a time.

Thus there are (at least) three states for a process:
- **Running**: accomplishing useful work
- **Sleeping**: waiting for something to happen
- **Ready**: ready to run, but not running now.
Process state rules

Only one process can run on one core at a time. An arbitrary number of processes can be sleeping and/or ready. The number of processes that are running and/or ready is referred to as the instantaneous load on the machine. The average of instantaneous load over time is called a load average. This is a measure of how busy the OS is.
An example

30 emacs instances
  20 waiting for a keypress
  10 have received a keypress and are waiting to run.
6 cores
  => 6 running ones.
  4 ready ones.
Instantaneous load is 10.
"uptime"

reports the load average on a Linux system
independent of the number of cores(!)
Averages for 1 min, 5 min, 10 min

1 core:
load average=1: pretty good
load average=40: awful.

Watch out:
on an 8-core system, a load average of 40 is 8 times
less awful than on a 1-core system!
Example: load average for a one-core machine

\[
\frac{1 + 1 + 2 + 2 + 2}{2}
\]
The Five-state model

The Five-state model (almost all OS's)
What the states mean
Monday, September 11, 2017 9:22 AM

New: not scheduled yet
Running: on a core and doing something.
Blocked: waiting for something.
Ready: ready to run but not running yet.
Exit: done and unloaded.
Not all state transitions are possible

Can't go from blocked to running (must be ready first).
Can't go from blocked to exit (must run in order to exit)!
Somewhat more refined than the 5-state model.

Four distinct forms of blocked:
- **S**: sleeping: interruptable wait.
- **D**: uninterruptable wait (for disk operations)
- **W**: page wait for memory
- **T**: trace wait: stopped for debugging purposes.

A state before exit:
- **Z**: "zombie": dead but awaiting reporting status to parent.
ps: lists processes in the process table, by number.
Most useful form:
   ps -ef
e: extended listing: all processes
f: full information
The /proc pseudo filesystem
Contains a directory for each process, by process number.
That lists everything the operating system knows about the process.
Including its current state.
   R - in run queue
   S - sleeping
   D - disk wait
   W - paging
   T - paused
   Z - zombie (dead, but waiting for reaping of status)
See /proc/{pid}/status where {pid} is the process identifier (an integer).

ps uses /proc.
Scheduling

Refers to the algorithm by which the next runnable process is chosen to run.

Two really common scheduling algorithms:

**Round-robin:** take turns running parts of the available runnable processes.

**Batch:** run the highest priority process until it completes or waits; then run the next-highest priority process until the highest priority process becomes runnable.
Examples of scheduling

Round-robin

Batch
Time and context switching

Only time spent running counts as user or system time.
Time spent context switching counts as OS time.

Efficiency = time spent doing useful work / wallclock elapsed time (per core).
Metrics of scheduling

**Responsiveness:** the extent to which a system seems to respond to your commands.

- **response time (latency):** the time between a command and its completion.
- **average response time:** measure of responsiveness.

**Throughput:** how much useful work is actually accomplished.

- **efficiency:** the ratio of user+system time / wallclock time, as a measure of how much work is done.
- **overhead:** the ratio of OS time / wallclock time, as a measure of time "wasted" on non-user tasks.

**Fairness:** to what extent is time equally shared between runnable processes?
To understand scheduling, it's helpful to start with a physical metaphor.

work = cpu cycles expended on a goal.

Analogous to "distance traveled".

throughput = work / wallclock time = average work per unit time.

efficiency = work / total cycles = % of time used productively.

processor speed = total cycles / wallclock time

overhead = 1 - work / total cycles = % of time "wasted": used for purposes other than accomplishing computational goals.

efficiency + overhead = 1

throughput = efficiency * processor speed
Latency

- the amount of time between input to a process and output.
- this is the time from becoming unblocked to being blocked again.
- computed as an average.
For N processes, how equally is work distributed?

Perhaps easier to measure **unfairness**: the standard deviation of efficiency over all processes.

\[
\sqrt{\frac{\sum_{p=1}^{N} (e_p - \bar{e})^2}{N-1}}
\]

where \( e_p \) is efficiency of process \( p \)
and \( \bar{e} \) is mean efficiency \( \frac{\sum_{p=1}^{N} e_p}{N} \)

Optimally fair => unfairness is 0

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

Note that in this case, the efficiency of executing a process \( e_p \) is the number of cycles spent on that process alone / total cycles available.
Low latency and high throughput are in conflict.

Low latency means frequent context switching and lower throughput. Higher throughput means less context switching, and thus higher latency.

(And round-robin scheduling isn't fair.)
Rest of course:

○ Better: faster, more reliable, more secure.
○ Some particular confusion as to what "better" means!
○ Subtleties abound; let's try to demystify them as we go....