Recall from last time that Linux process state includes "runnable" ("R") and several kinds of "blocked"

- **S**: sleeping: interruptible wait.
- **D**: uninterruptable wait (for disk operations)
- **W**: page wait for memory
- **T**: trace wait: stopped for debugging purposes.

Note the big change from last time. My diagram from last time turns out to be a common misconception. I repaired the previous notes!

The state "D" occurs in the short time that a disk transfer is running. It is not a scheduler state. More about this later.

How do these states come about? What do they mean? How can one manipulate process state?
Every process uses memory that is organized in a whole number of pages.

If you ask for part of a page from the OS, you get a whole page.

At any one time, a specific page may be in memory or "swapped out" to disk.

For a process to make progress in running, the pages containing the place in the program that is running, and data needed by that program, need to be "resident" in memory and not "swapped out".

Referencing a page that is not resident is called a "page fault".
Prerequisites are met for the process to run:
   Required pages are in memory now.
   Requested I/O (if any) has completed.
   Requested waits (if any) have completed.
   Etc.
Runnable is not the same as running. In particular, since any process monitoring state may have taken the place of another runnable process, it is rather likely that processes other than the one querying state are not running at the same time.
A regular sleep can occur as a result of
Calling the sleep(n) system call to sleep for
n seconds (n is an integer).
Trying to read something that is not ready
to read.

Some very important misconceptions to
correct now:
read doesn't sleep unless there is nothing
to read yet.
write doesn't sleep very often, and only
when it is out of room in a FIFO (first-in,
first-out queue).
A process enters page wait if a required page is not resident.
Upon reference to the page, the process generates a (software) interrupt.
The OS takes over and tries to load the page from disk.
If that fails because the page doesn't exist, we get a "segmentation fault".
If that succeeds, the process becomes runnable again.
The trace state is entered when:
You type control-Z to the shell or send the SIGTSTP signal to the process (which is precisely equivalent). Control-Z does exactly the same thing as

```bash
kill -TSTP {pid}
```

where `{pid}` is the process id to stop.
A debugger stops execution of the process by similar means.

Some notes:
The opposite of control-Z is to either type:
- `fg`: continue the process in the foreground.
- `bg`: continue the process in the background.

Both of these send a SIGCONT (continue).
One can also continue a stopped process (in either foreground or background) via

```bash
kill -CONT {pid}
```
This one is really difficult to see.

Modern disk controllers work via "direct memory access" (DMA): they write data directly from disk to memory. During this (very short) time, the DMA cannot be interrupted.

So this is a very different state than "W" or "S":

"S" and "W" are interruptable waits.
If a process receives a signal during an interruptable wait, it processes the signal.
"D" is an uninterruptable wait. It can occur during "W" and "S" states where the disk is concerned.

So -- and this is different than I said on monday(!) -- one cannot get to the "D" state from the "R" state. One has to be in "W" or "S" first!
A process in state "D" returns to the other state ("W" or "S") when the disk transfer is over.
Signals delivered to the process are deferred until the "D" state ends.

It is very unlikely that your program will
be able to even capture a "D" state. They are too short-lived!
In assignment 1, you are unlikely to see this one. And if you do, it means trouble!

A zombie process is one that has ended, but whose exit status has not been read by its parent. In assignment 1, your program is the parent, so if you allow the monitored process to become a zombie, your program is incorrect!
Scheduling

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

Three common scheduling policies:

**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.

**Real-time:** interrupt whatever you're doing to respond to outside stimuli, including other processes with batch and/or round-robin scheduling policies.
Examples of scheduling

Thursday, September 09, 2010   4:46 PM

Round-robin

Batch

Real-time + batch

real-time event
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).
The scheduling quandary

There is no scheduling policy that is a panacea.

- Round-robin scheduling slows down batch processes (why?)
- Batch scheduling makes interactive processes unusable (why?)
- Real-time scheduling is only applicable to real-time events and not to batch scheduling.
Prioritized scheduling: allow every process a slice of time proportional to its priority. "Fair" scheduling: allow all processes at the same priority a total amount of time that is roughly equal, regardless of blocking due to sleeps, page waits, or I/O.

Both of these are variants of round robin that address weaknesses in same.
In process scheduling, an "epoch" is a length of time in which there is some concept of fairness.

- Round-robin: all processes have gotten a slice.
- Fair scheduling: all processes at the same priority have received the same amount of CPU time.
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.

```
request latency response
```

```
for a web page
```
For N processes, how equally is work distributed?

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

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

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.