Stuff to memorize which Mark said is **definitely** or **probably** on midterm.

Definitely

Context switch diagram: (process and thread notes, page 6)
Test question is context switch diagram:

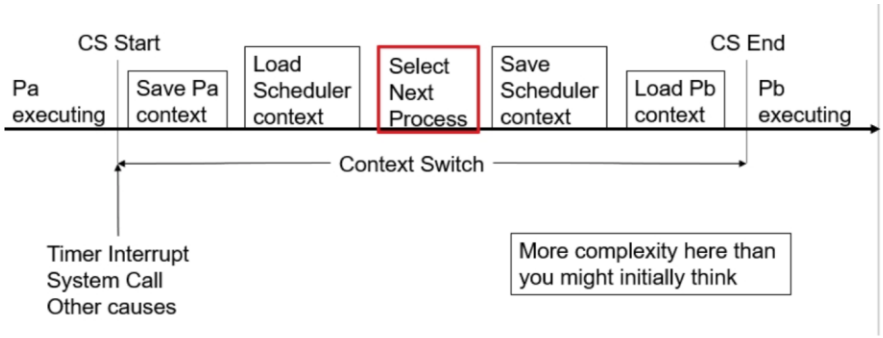

Figure 4: Context Switch

“Question: A context switch is comprised of two context switches. Explain what this means:

Answer: There's a context switch in saving a context and loading a new one!”

Process State Transition Model (process and thread notes, page 11)
“All midterms have asked to reproduce this diagram”

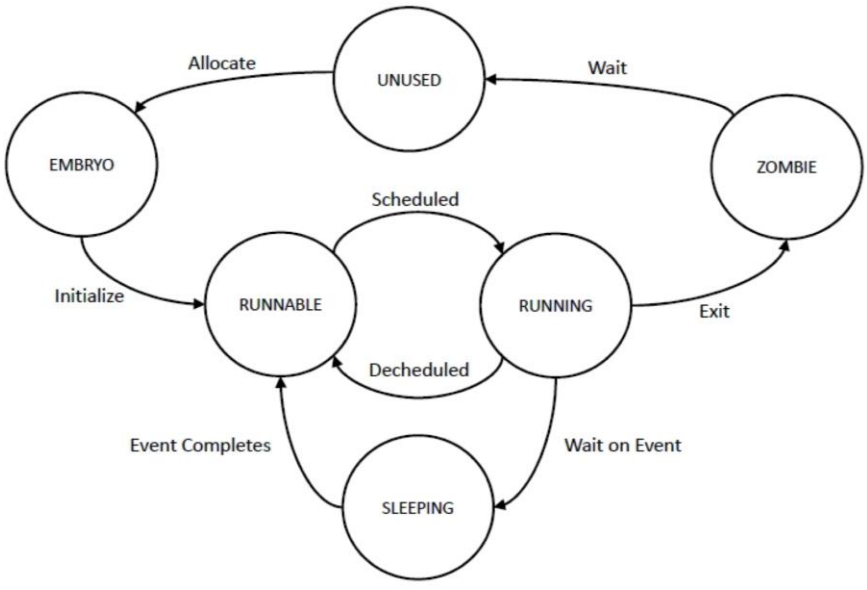

Figure 7: xv6 Process State Transition Model

Stuff to memorize which Mark said is **definitely** or **probably** on midterm.

Concurrency (concurrently notes, page 1)
"Will need to define concurrency on exam"

concurrency the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.

Deadlock (concurrently notes, page 7)
Midterm (and maybe final) test question is listing 4 necessary conditions for it below, some supplemental info:

- If a process is indefinitely denied access to a resource it is called **starvation**, on result of deadlock
- Can only happen if (and only) if all 4 conditions are met
- If they are met, which is often the case for reasonable sized multithreaded code, one of the conditions need to be broken.

The 4 Conditions Necessary and Sufficient for Deadlock

1. Mutual Exclusion. Only one process can use the resource at any one time.
2. Hold and Wait. A process currently holding one exclusive resource is allowed to request another exclusive resource.
3. No preemption. Once a process holds a resource, it cannot be taken away. The held resource must be voluntarily given up by the process.
4. Circular wait. A cycle in the resource allocation graph must exist. Each process must be waiting for a resource which is held by another process, which in turn is waiting for the first process to release the resource. See Fig. 2.

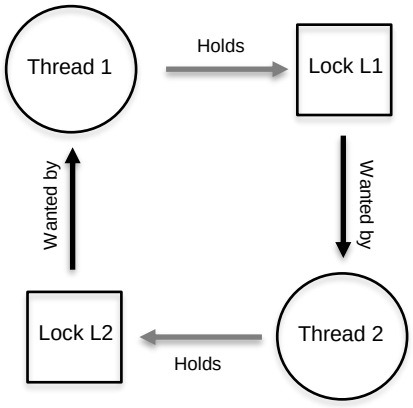

Figure 2: Resource Allocation Graph Cycle

Lock Evaluation (Concurrency Control, page 6)
How do we evaluate locks, what is order of importance and why is that the order of importance? 10% of points allegedly. Screenshot below is for what we evaluate, the why is the following pages.
My take on importance is: if it's not correct it doesn't matter how fair or performant it is, as it will possibly corrupt data. If it's not fair a process could starve and never complete, which isn't acceptable either. Finally, only if everything else is working can speed be conditionally considered.

Stuff to memorize which Mark said is **definitely** or **probably** on midterm.

2.3 Lock Evaluation

1. Correctness. Basically, does the lock work, preventing multiple threads from entering a critical section?

2. Fairness. Does each thread contending for the lock get a fair shot at acquiring it once it is free? Another way to look at this is by examining the more extreme case: does any thread contending for the lock starve while doing so, thus never obtaining it?

3. Performance. What are the costs of using a spin lock? To analyze this more carefully, we suggest thinking about a few different cases. In the first, imagine threads competing for the lock on a single processor; in the second, consider threads spread out across many CPUs.

   For spin locks, in the single CPU case, performance overheads can be quite painful; imagine the case where the thread holding the lock is preempted within a critical section. The scheduler might then run every other thread (imagine there are $N-1$ others), each of which tries to acquire the lock. In this case, each of those threads will spin for the duration of a time slice before giving up the CPU, a waste of CPU cycles.

   However, on multiple CPUs, spin locks work reasonably well (if the number of threads roughly equals the number of CPUs). The thinking goes as follows: imagine Thread A on CPU 1 and Thread B on CPU 2, both contending for a lock. If Thread A (CPU 1) grabs the lock, and then Thread B tries to, B will spin (on CPU 2). However, presumably the critical section is short, and thus soon the lock becomes available, and is acquired by Thread B. Spinning to wait for a lock held on another processor doesn't waste many cycles in this case, and thus can be effective.

5 rules of Multilevel Feedback Queue -

MLFQ Final Rule Set

- 1: If Priority(A) ¿ Priority(B), A runs (B doesn't).
- 2: If Priority(A) = Priority(B), A & B run in RR.
- 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
- 4: Once a job uses up its time allotment (budget) at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- 5: After some time period *S*, promote all jobs
  - In this class, we use one level of promotion

How does MLFQ avoid starvation? -

See 5 above

From midterm review questions, sounded like they'd be on test

Two threads incrementing i, starts at 3 is it possible:

- …to end up at 3?
  - **No,** we're going to experience at least one write (in one round)
- …to end up at 4?
  - **Yes,** could skip one write
- …to end up at 6?
  - **No,** we have two writes, both increments, so max we can put out there is +2

Stuff to memorize which Mark said is **definitely** or **probably** on midterm.

Why do we need hardware designers to make lock? (on exam)

- Cause lock acquisition in hardware looks exactly like i++, 3 steps, so wouldn't be done atomically and could be interrupted, more than 1 thread got a lock
- Hardware designers give us a call that acts atomically in one step

Probably

- Assembly code questions aren't usually considered, if so, it would be **xchng** used in spinlock
- 5 starving philosophers, one solution is one picks up with left hand

Principle advantage of CPU virtualization

- It's giving every process its own CPU, that can only run its process (can't do context switches), they get passed off to CPU
- So, our process doesn't have to worry about context switches or managing hardware, so much that processes doesn't need to worry about
- (programs don't need to worry about writing print driver)