Lab 2 Overview

Section 3: 01/23/20

Please pick up section handout as you come in :)

Lab 2

- Two parts:
  - Design on fork, wait, exit due 1/24/2020 (Tomorrow!) at 11:59pm
  - Design on pipe due 1/31/2020 (Next Friday) at 11:59pm
- The whole Lab2 Due 2/7/2020 (Next next Friday) at 11:59pm
- Lab2 will be ***time consuming and difficult***

Late Days

- Don't tag submission until you are done (or plan to use late days)
- Email cse451-staff@cs.washington.edu with

  [Late Day + Group Member netid] in subject line after you tagged the late submission (send us an interrupt :p)

Design Document

Due Tomorrow**(1/23)**:

- This is for you, whatever will prepare you for success should be on the document.
- It will be hard the first time knowing what to include, that's ok. You will learn from the earlier labs to (hopefully) become more successful in later labs.
- Office hours are a great time to talk about design. It's easier to see your approach in words instead of spread throughout many files.
- Use labs/designdoc.md as a reference on what to include in your design docs!
- Template is at labs/lab2design.md

OSV synchronization primitives

- include/kernel/synch.h
- kernel/synch.c
- spinlock, sleeplock, condition variable

Spinlocks

- Spin until it acquires the lock
- Interrupt is disabled while holding a spinlock
- don't worry about the intrlock parameter in spinlock_init, not used currently

Sleeplocks(mutex)

- Blocks until it can acquire the lock.
- On `sleeplock_acquire`, if the lock is already acquired by another thread, the current thread adds itself to the lock's waiter list, sets its state to `SLEEPING` and blocks. It won't get scheduled until there is an opportunity to grab the lock.
- On `sleeplock_release`, the thread holding the sleeplock wakes up a thread from the lock's waiter list. This will set the sleeping thread's state to `READY`, so the sleeping thread can wake up, check the lock condition, grab the lock if it's still free, block again otherwise.

Condition Variables

- Always paired with a spinlock in osv, CV itself is essentially a list of waiters
- `condvar_wait`, blocks until a condition is no longer true, release lock before blocking and grabs lock before it returns
- `condvar_signal`, wake up a thread from CV list, because condition have changed
- `condvar_broadcast`, same as above but wake up all threads on the CV list

Condition Variable Example

```
int ice_cream = 0;

void get_ice_cream() {
   spinlock_acquire(&fridge_lock);
   while(ice_cream <= 0) {
     condvar_wait(&icecream_condvar, &fridge_lock);
   }
   // CONSUME ICE CREAM
   ice_cream--;
   spinlock_release(&fridge_lock);
}
```

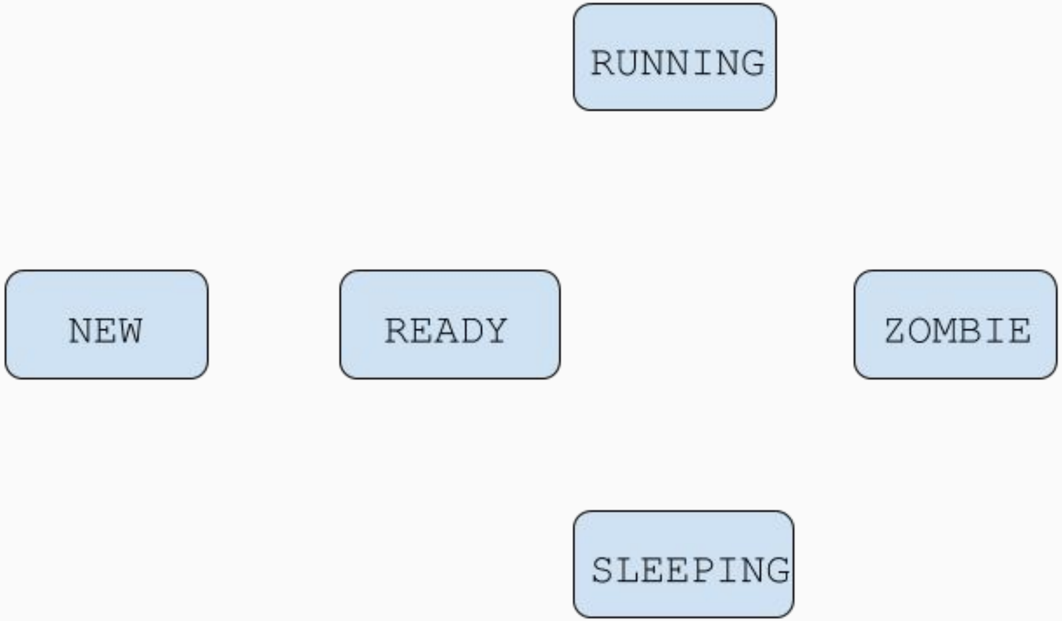
```
void stock_ice_cream() {
   spinlock_acquire(&fridge_lock);
   // put ice cream in fridge
   ice_cream++;
   condvar_signal(&icecream_condvar);
   spinlock_release(&fridge_lock);
}
```

- condvar_wait in a while loop because it is possible that there is no more ice cream (condition is no longer true) after the thread wakes up.
- once the while loop is exited, the thread has guaranteed mutual exclusion and you KNOW there is ice cream (condition is held).

More on Locks and Synchronization

See Chapter 5 in Operating Systems: Principles and Practice

Process Life Cycle (exercise)



RUNNING

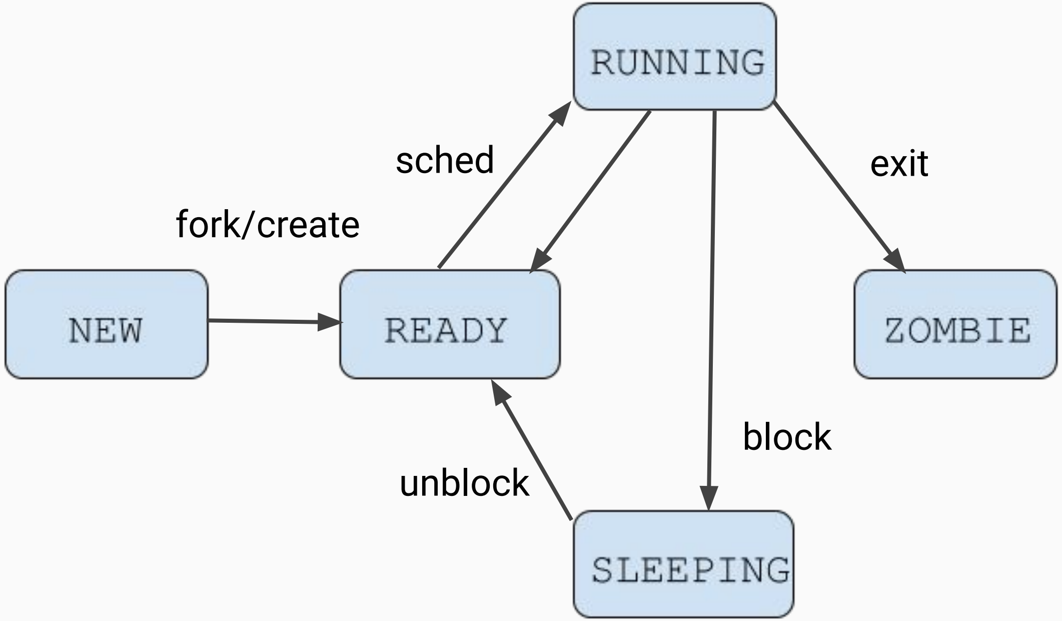
NEW

READY

ZOMBIE

SLEEPING

Process Life Cycle



fork()

Creates a new process by duplicating the calling process. Returns 0 in child, and child PID in parent.

You need a way to track parent child relationship.

What does this entail? What needs to be created and what/how do we copy parent process state?

wait()/exit()

wait() - Waits until a child process terminates and returns that child’s PID and exit status
exit() - Halts program and reclaims resources consumed by the program.

Cases to consider:

- parent waits before child exits
- parent waits after child exits
- parent exits without waiting for child

wait

- Parent can only wait for its children
- Parent cannot wait for the same child twice
- Parent needs to block until the child exits (a condition)

exit (exercise)

- What resources need to be cleaned up when a process exits?
- When can a process's resources be safely cleaned up? Who will free these resources?

Keep in mind that a process can be both a parent to other processes and a child of another process. Think through both roles the process needs to play when it exits.

exit (exercise)

- If a parent process calls exit before it's child finishes executing, how does the child process need to be modified to guarantee that someone will wait for the child?

pipe(fds)

- Creates a pipe (internal buffer) for reading to/writing from.
- Similar to bounded buffer
- From user's perspective: Two new files will be allocated, one will be the "read end" (not writable), and one will be the "write end" (not readable).
- Will need to create a set of pipe file operations so fs_read_file on pipe file can be directed to the right function.

```
static struct file_operations pipe_file_operations = {
    .read = pipe_read,
    .write = pipe_write,
    .close = pipe_close
};
```