Section 3
October 12th

Concurrency within a Pipe:
Currently, we assume there is only one process who can access the buffer in pipe. In lab2, we need to change the implementation of our pipe so it can safely be accessed by multiple processes (i.e., multiple readers and multiple writers).

We will use spinlock to ensure consistency and integrity of the `pipe structure`, and use conditional variables to coordinate the multiple readers and writers.

The expected behavior is that read on the pipe should block while there is nothing to read. Meaning read should wait until the pipe is written to (using `sleep)`. However, if the write side of the pipe is closed, the reader should be notified (using `wakeup)`.

**Remember*:* We won't write more than 512 bytes, so don't worry about handling arbitrary amounts of writing (...unless you want to handle that case!)

Question: What condition should the reader wait on? What condition should the writer wait on?

When there are multiple writers to the pipe, a single write call should be atomic. For example, one process write "aaa", another process write "bbb", the content should be either "aaabbb" or "bbbaaa" there should be no interleaving such that the pipe contains "ababba".

Fork:
Fork creates another copy of the current process. This means all the kernel data structure of a process has to be duplicated including: `struct proc, user memory, memory region, trapframe, open files.`

Question: Where will the new process start executing in user mode?

To differentiate the new processes from the old process, we call the new process a child of the parent old process. The return value of `fork` is different between the child and the parent. The parent will return the current process id and the child will return 0.

Question: How can we alter the return value of the fork function to simulate the situation above?

Exit, Wait:
`exit` and `wait` are two important synchronization methods in xk. `wait` waits until a child has exited and returns the process id of the child. `exit` exits the current process.

xk cannot immediately free all the kernel data structures associated with the currently running process because the physical machine has page table pointer (register `%cr3`) pointing to a field inside `struct proc` of the current process.

Question: How can the kernel clean up the data structures associated with the currently running process when the process calls `exit?`