Introduction

This is a project report submission in partial fulfillment of the requirements for the course COMP SCI 537 under Prof. Remzi H. Arpaci-Dusseau, Department of Computer Sciences. It aims at analyzing locks and sleep/wakeup routines in xv6 operating system. The report has two major parts. The first part outlines the analysis on two particular lock instances and the second part focuses on understanding two particular processes that sleep and then wakeup.

[Part 1] Locking in xv6

Xv6 uses locks in many places to avoid race conditions. There are two types of locks in xv6 – Spinlock and Sleeplock. I have picked two instances of spinlock that shall be analyzed in the later sections. As a recap, a lock is used to ensure that only one thread (CPU in this case since xv6 doesn't support threads) is running within the critical section at a time, or when we want longer sequences of instructions to be atomic.

[Part 1] Lock Instances

Picked the following lock instances for analysis.
Instance A: tickslock in *trap.c*
Instance B: ftable.lock in *file.c*
We have the following objectives for each critical section in each lock instance:

1. Find out where it is used.
2. How long the critical sections are.
3. A general understanding of what they are doing and which high-level functions contend for the lock.
4. How often the lock is held and released when a process is running.

Remarks for objective 1:
Lock usage will be represented using a hierarchy structure. <filename> → <method name> → <use case>

Remarks for objective 2:
The length of the critical section will be represented in terms of instructions in a very rudimentary way. Note that only for tickslock, there is extra analysis for duration. For other lock instances/critical sections, it will be done as mentioned above.

Remarks for objective 3:
We will mainly focus into the lock usage, functionalities and higher level callers.

Remarks for objective 4:
This objective is aimed at how multiple threads/CPUs interact with the lock in the critical section. We will use print statements within critical sections to determine how often a lock gets acquired by the particular process for each CPU.

[Part 1] Analysis of Lock Instance A

There is only one critical section for this lock instance.

```
acquire(&tickslock);
ticks++;
wakeup(&ticks);
release(&tickslock);
```

Following sub-sections provide analysis of the above-mentioned lock instance.

Objective 1:

trap.c
→ *trap(struct trapframe *tf)*
→ Interrupt by hardware clock every tick handled by case T_IRQ0 + IRQ_TIMER.

Objective 2:

Clearly, this runs for one tick duration. Or, the minimum time possible between two timer interrupts. Currently, a tick increments every time there is a hardware timer interrupt represented by the case T_IRQ0 + IRQ_TIMER.

If we observe the following lines of code in *lapic.c* file,

```
lapicinit(void)
{
  ...
  // The timer repeatedly counts down at bus frequency
  // from lapic[TICR] and then issues an interrupt.
  // If xv6 cared more about precise timekeeping,
  // TICR would be calibrated using an external time
source.
  lapicw(TDCR, X1);
  lapicw(TIMER, PERIODIC | (T_IRQ0 +
IRQ_TIMER));
  lapicw(TICR, 10000000);
  …
```

it can be seen that lapic's register TICR controls the time slice duration which can be modified to change tick duration or make timer interrupts more spaced. The time will depend on bus frequency. For instance, if the bus is at 1MHz, its period will be at 1us (1micro second), so the time slice will be at
1us * 10 000 000 = 10 000 000 us = 10 seconds.

Objective 3:

trap(), when it's called for a time interrupt, does just two things: increment the ticks variable, and call wakeup. At the end of trap, xv6 calls yield that may cause the interrupt to return in a different process. tickslock serializes operations on the ticks counter.

trap() is called in three situations. Firstly, when there is a system call, and a user program requires the kernel to do something for it. Secondly, an exception i.e. an instruction (user or kernel) does something illegal, such as divide by zero or use an invalid virtual address. Thirdly, a device interrupt which is a hardware event (the one we are interested in).

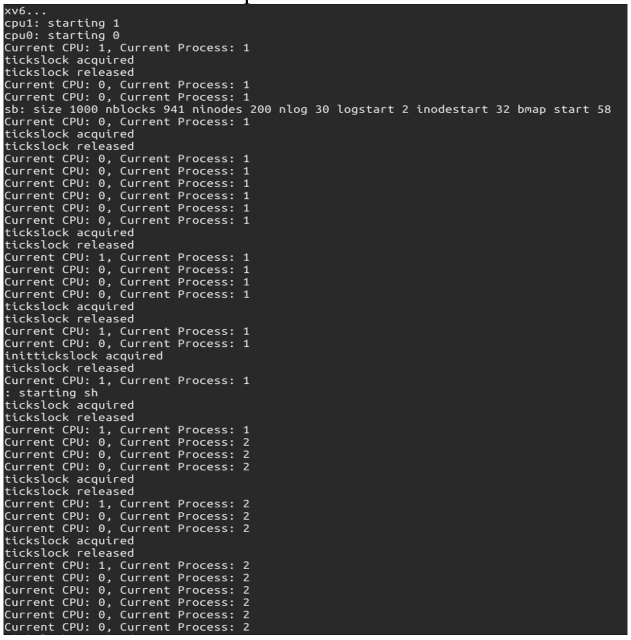
As for the lock usage, I have made the following observations regarding threads and lock accesses.

- The timer interrupt handler might increment ticks at about the same time that a kernel thread reads ticks in *sleep()* or *uptime()* system call. The lock tickslock serializes the two accesses.

- The interaction of spinlocks and interrupts raises a potential danger. Suppose *sleep()* holds tickslock, and its CPU is interrupted by a timer interrupt. Timer interrupt handler would try to acquire tickslock, see it was held, and wait for it to be released. In this situation, tickslock will never be released: only *sleep()* can release it, but sys_sleep will not continue running until timer interrupt handler returns. So the CPU will deadlock, and any code that needs either lock will also freeze.
- To avoid this situation, if a spinlock is used by an interrupt handler, a CPU must never hold that lock with interrupts When a CPU acquires any lock, xv6 always disables interrupts on that CPU. Interrupts may still occur on other CPUs, so an interrupt's acquire can wait for a thread to release a spinlock; just not on the same CPU.

Objective 4:

Following illustration shows how 2 CPUs interact with tickslock at startup. It is acquired every time there is a hardware timer interrupt.



```
xv6...
cpu1: starting 1
cpu0: starting 0
Current CPU: 1, Current Process: 1
tickslock acquired
tickslock released
Current CPU: 0, Current Process: 1
Current CPU: 0, Current Process: 1
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
Current CPU: 0, Current Process: 1
tickslock acquired
tickslock released
Current CPU: 0, Current Process: 1
Current CPU: 0, Current Process: 1
Current CPU: 0, Current Process: 1
Current CPU: 0, Current Process: 1
Current CPU: 0, Current Process: 1
Current CPU: 0, Current Process: 1
tickslock acquired
tickslock released
Current CPU: 1, Current Process: 1
Current CPU: 0, Current Process: 1
Current CPU: 0, Current Process: 1
Current CPU: 0, Current Process: 1
tickslock acquired
tickslock released
Current CPU: 1, Current Process: 1
Current CPU: 0, Current Process: 1
inittickslock acquired
tickslock released
Current CPU: 1, Current Process: 1
: starting sh
tickslock acquired
tickslock released
Current CPU: 1, Current Process: 1
Current CPU: 0, Current Process: 2
Current CPU: 0, Current Process: 2
Current CPU: 0, Current Process: 2
tickslock acquired
tickslock released
Current CPU: 1, Current Process: 2
Current CPU: 0, Current Process: 2
Current CPU: 0, Current Process: 2
tickslock acquired
tickslock released
Current CPU: 1, Current Process: 2
Current CPU: 0, Current Process: 2
Current CPU: 0, Current Process: 2
Current CPU: 0, Current Process: 2
Current CPU: 0, Current Process: 2
Current CPU: 0, Current Process: 2
```

[Part 1] Analysis of Lock Instance B

There are three critical sections for this lock instance. ftable.lock serializes allocation of a struct file in file table. Every open file in xv6 is represented by a struct file which is simply a wrapper around inode/pipe, along with read/write offsets and reference counts. The global open file table 'ftable' is an array of file structures, protected by a global ftable.lock.

Following sub-sections provide analysis of the above-mentioned lock instance for each critical section.

Critical Section 1

```
  acquire(&ftable.lock);
  for(f = ftable.file; f < ftable.file + NFILE; f++){
          if(f->ref == 0) {
              f->ref = 1;
              release(&ftable.lock);
              return f;
          }
     }
  release(&ftable.lock);
```

[Critical Section 1] Objective 1:

file.c
  → *file *filealloc(void)*
      → *Find and return address of first free slot in the global file table.*

[Critical Section 1] Objective 2:

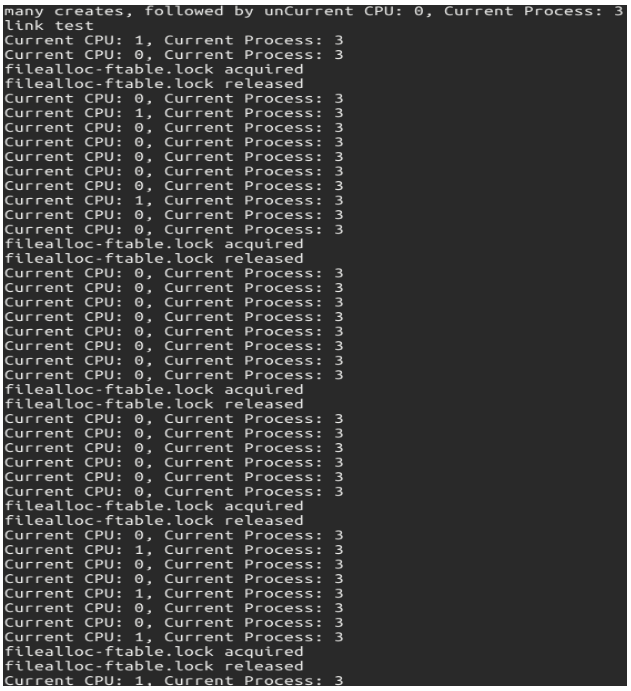
It is a loop over file table and with a single if statement for each iteration. It runs till it finds the first free slot. If free slot doesn't exist, the corresponding function *filealloc* returns 0.

[Critical Section 1] Objective 3:

filealloc gets called by *pipealloc* which gets called by *pipe()* system call. It finds the first two available (using *filealloc*) positions in the process's open file table and allocates them for the read and write ends of the pipe. It also gets called by *open()* system call which creates or opens inode. It creates struct for inode, add it to ftable and set file struct data.

[Critical Section 1] Objective 4:

Refer to *file_test1.c*. Add it to UPROGS in makefile before running xv6. This test calls multiple create system calls which internally use the open system call. Thus we can observe how 2 CPUs perform multiple creates while interacting with ftable.lock.
Note that test process ID is 3.



```
many creates, followed by unCurrent CPU: 0, Current Process: 3
link test
Current CPU: 1, Current Process: 3
Current CPU: 0, Current Process: 3
filealloc-ftable.lock acquired
filealloc-ftable.lock released
Current CPU: 0, Current Process: 3
Current CPU: 1, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 1, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
filealloc-ftable.lock acquired
filealloc-ftable.lock released
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
filealloc-ftable.lock acquired
filealloc-ftable.lock released
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
filealloc-ftable.lock acquired
filealloc-ftable.lock released
Current CPU: 0, Current Process: 3
Current CPU: 1, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 1, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 1, Current Process: 3
filealloc-ftable.lock acquired
filealloc-ftable.lock released
Current CPU: 1, Current Process: 3
```

Critical Section 2

```
acquire(&ftable.lock);
if(f->ref < 1)
   panic("filedup");
f->ref++;
release(&ftable.lock);
```

[Critical Section 2] Objective 1:

file.c
   *→ file *filedup(struct file*)*
         → If passed file reference is valid, gets incremented else panic.

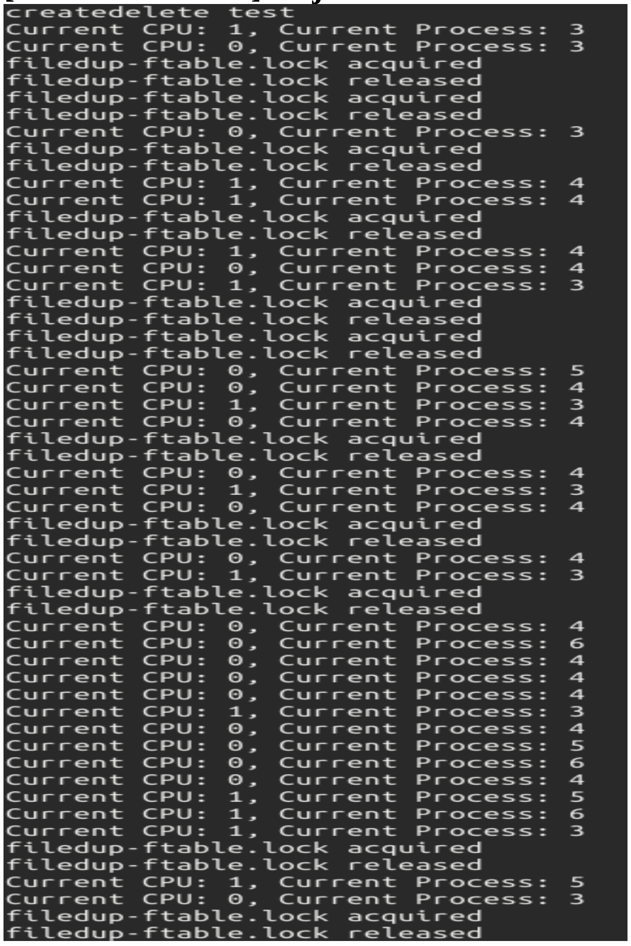
[Critical Section 2] Objective 2:

It is a single if statement and an increment instruction that operates on the reference of the passed file structure pointer.

[Critical Section 2] Objective 3:

The reference count is the number of processes currently accessing the file. It gets called by the *dup()* system call which duplicates proc's reference to file and calls *filedup* to increment the file's reference count. It also gets called by *fork()* system call. Whenever a child process is created, it gets a copy of the file descriptor table from the parent process. So, if a file is open in master process and a child process is created, the reference count increments, as it is now open in child process as well, and when it is closed in any of the processes, it decrements. A file is finally closed when the reference count reaches zero.

[Critical Section 2] Objective 4:



```
createdelete test
Current CPU: 1, Current Process: 3
Current CPU: 0, Current Process: 3
filedup-ftable.lock acquired
filedup-ftable.lock released
filedup-ftable.lock acquired
filedup-ftable.lock released
Current CPU: 0, Current Process: 3
filedup-ftable.lock acquired
filedup-ftable.lock released
Current CPU: 1, Current Process: 4
Current CPU: 1, Current Process: 4
filedup-ftable.lock acquired
filedup-ftable.lock released
Current CPU: 1, Current Process: 4
Current CPU: 0, Current Process: 4
Current CPU: 1, Current Process: 3
filedup-ftable.lock acquired
filedup-ftable.lock released
filedup-ftable.lock acquired
filedup-ftable.lock released
Current CPU: 0, Current Process: 5
Current CPU: 0, Current Process: 4
Current CPU: 1, Current Process: 3
Current CPU: 0, Current Process: 4
filedup-ftable.lock acquired
filedup-ftable.lock released
Current CPU: 0, Current Process: 4
Current CPU: 1, Current Process: 3
Current CPU: 0, Current Process: 4
filedup-ftable.lock acquired
filedup-ftable.lock released
Current CPU: 0, Current Process: 4
Current CPU: 1, Current Process: 3
filedup-ftable.lock acquired
filedup-ftable.lock released
Current CPU: 0, Current Process: 4
Current CPU: 0, Current Process: 6
Current CPU: 0, Current Process: 4
Current CPU: 0, Current Process: 4
Current CPU: 0, Current Process: 4
Current CPU: 1, Current Process: 3
Current CPU: 0, Current Process: 4
Current CPU: 0, Current Process: 5
Current CPU: 0, Current Process: 6
Current CPU: 0, Current Process: 4
Current CPU: 1, Current Process: 5
Current CPU: 1, Current Process: 6
Current CPU: 1, Current Process: 3
filedup-ftable.lock acquired
filedup-ftable.lock released
Current CPU: 1, Current Process: 5
Current CPU: 0, Current Process: 3
filedup-ftable.lock acquired
filedup-ftable.lock released
```

Refer to *file_test2.c*. Add it to UPROGS in makefile before running xv6. This test calls multiple create-delete(unlink) system calls over 4 processes which call *fork()* while file is open thereby internally incrementing reference by calling filedup. Thus we can observe how 2 CPUs perform multiple create-deletes while interacting with ftable.lock.
Note that parent process ID is 3.
Child process IDs are 4, 5 and 6.

Critical Section 3

```
acquire(&ftable.lock);
if(f->ref < 1)
   panic("fileclose");
if(--f->ref > 0){
   release(&ftable.lock);
   return;
}
ff = *f;
f->ref = 0;
f->type = FD_NONE;
release(&ftable.lock);
```

[Critical Section 3] Objective 1:

file.c
   *→ file *fileclose(struct file*)*
         → Decrements passed file reference. When a If reaches zero, underlying pipe or inode is released.

[Critical Section 3] Objective 2:

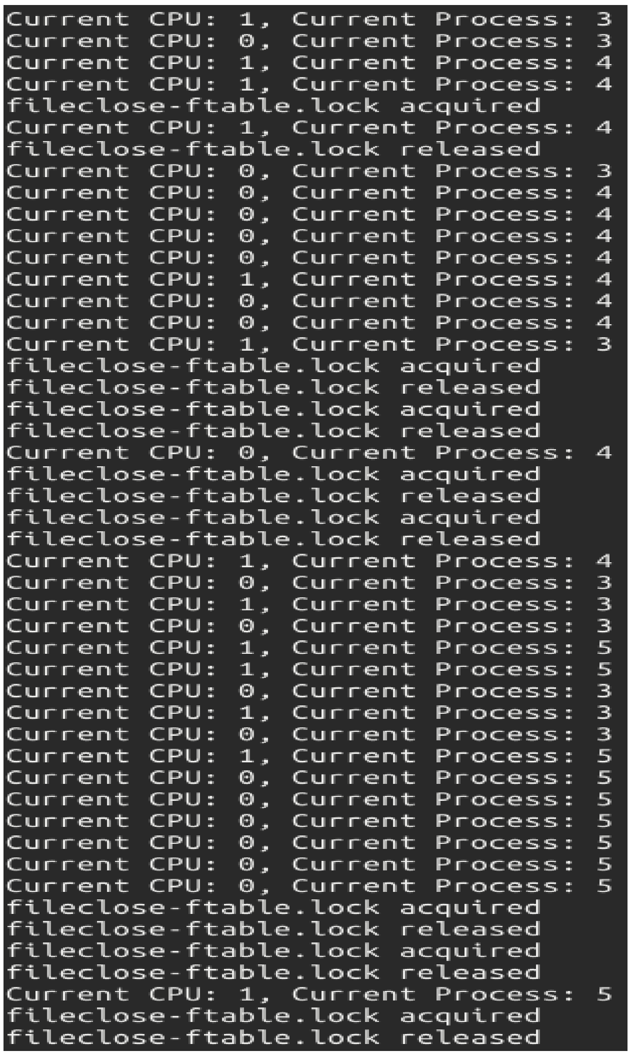
It is two if statements and three assignment instructions that operate on the reference of the passed file structure pointer.

[Critical Section 3] Objective 3:

This area has three callers. Firstly, it gets called by the *close()* system call. It releases the corresponding file descriptor, making it free for reuse by a future *open()*, *pipe()*, or *dup()* system calls. Secondly, it is also called upon *exit()* which terminates the calling process and any open file descriptors belonging to the process are closed thereby internally calling *close()* system call. Lastly, it is also called by *pipealloc()* system call if read/write file descriptors don't get allocated successfully, thereby releasing them by calling *fileclose.*

[Critical Section 3] Objective 4:

Refer to *file_test3.c*. Add it to UPROGS in makefile before running xv6. This test performs concurrent create/link/unlink of the same file resulting in multiple *close()* system calls across all the processes.
Note that parent process ID is 3.
Child process IDs are greater than 3. There are total 80 processes being run by the two CPUs. The diagram below doesn't represent the full thing, it is advised to run the test to get better insight.



```
Current CPU: 1, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 1, Current Process: 4
Current CPU: 1, Current Process: 4
fileclose-ftable.lock acquired
Current CPU: 1, Current Process: 4
fileclose-ftable.lock released
Current CPU: 0, Current Process: 3
Current CPU: 0, Current Process: 4
Current CPU: 0, Current Process: 4
Current CPU: 0, Current Process: 4
Current CPU: 0, Current Process: 4
Current CPU: 1, Current Process: 4
Current CPU: 0, Current Process: 4
Current CPU: 0, Current Process: 4
Current CPU: 1, Current Process: 3
fileclose-ftable.lock acquired
fileclose-ftable.lock released
fileclose-ftable.lock acquired
fileclose-ftable.lock released
Current CPU: 0, Current Process: 4
fileclose-ftable.lock acquired
fileclose-ftable.lock released
fileclose-ftable.lock acquired
fileclose-ftable.lock released
Current CPU: 1, Current Process: 4
Current CPU: 0, Current Process: 3
Current CPU: 1, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 1, Current Process: 5
Current CPU: 1, Current Process: 5
Current CPU: 0, Current Process: 3
Current CPU: 1, Current Process: 3
Current CPU: 0, Current Process: 3
Current CPU: 1, Current Process: 5
Current CPU: 0, Current Process: 5
Current CPU: 0, Current Process: 5
Current CPU: 0, Current Process: 5
Current CPU: 0, Current Process: 5
Current CPU: 0, Current Process: 5
Current CPU: 0, Current Process: 5
fileclose-ftable.lock acquired
fileclose-ftable.lock released
fileclose-ftable.lock acquired
fileclose-ftable.lock released
Current CPU: 1, Current Process: 5
fileclose-ftable.lock acquired
fileclose-ftable.lock released
```

[Part 2] Sleep and Wakeup in xv6

Xv6 provides Sleep and Wakeup functions, that are equivalent to the wait and signal functions of a Condition Variable (CV). Sleep and Wakeup functions must be invoked with a lock that ensures that the sleep and wakeup procedures are completed atomically. A process that wishes to block on a condition calls *sleep(chan, mutex)*, where *chan* is any opaque handle which refers to some event that must happen for the process to continue, and *mutex* is any lock being used by the code that calls sleep/wakeup.

Sleep routine can be summarized as follows:

1. Initial check if current process is valid and the *mutex* is held. Else, panic on sleep.
2. Acquire *ptable.lock* so that it can make changes to the process state (mark it as SLEEP) and invoke the scheduler.
3. Release the *mutex* so that another thread on this lock can run and can wake this thread up once the *chan* event is complete.
4. Context switch out by calling *shed()*
5. Control returns to this line once the process is woken up by some other thread/CPU and context switched in by the scheduler again, with *ptable.lock* held.
6. Release *ptable.lock.*
7. Reacquires the original *mutex* and returns back in the woken up process.

When the *chan* event is complete a process will invoke wakeup while it holds the *mutex* which gets released after wakeup is executed so that the woken up process can reacquire *mutex* as described in step 7 in sleep routine.

Wakeup routine can be summarized as follows:

1. Acquire *ptable.lock.*
2. Mark all sleeping processes waiting on *chan* event as RUNNABLE. (wakeup1 routine)
3. Release *ptable.lock.*

[Part 2] Sleep Instances

I have picked two instances of sleep-wakeup in code that will be analyzed in the following subsections:
Instance A: pipewrite in pipe.c
Instance B: piperead in pipe.c
sleep and wakeup are used here to synchronize producers and consumers in xv6’s implementation of pipes.

Note that unlike lock instance analysis, this analysis will be performed together as it is more intuitive to do so.
We have the following objectives for each section:

1. Find out where it is used.
2. Length of the section.
3. Identify the mutex on which sleep and wakeup are operating.
4. General flow of sleep-wakeup interaction.
5. A simulation of how the CPUs interact with pipes using the lock and sleep-wakeup routines

[Part 2] Analysis of Sleep Instances

Instance A: pipewrite

```
int
pipewrite(struct pipe *p, char *addr, int n)
{
  int i;
  acquire(&p->lock);
  for(i = 0; i < n; i++){
    while(p->nwrite == p->nread + PIPESIZE){
      if(p->readopen == 0 || myproc()->killed){
        release(&p->lock);
        return -1;
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->nread);
  release(&p->lock);
  return n;
}
```

Instance B: piperead

```
int
piperead(struct pipe *p, char *addr, int n)
```

```
{
  int i;

  acquire(&p->lock);
  while(p->nread == p->nwrite && p->writeopen){
    if(myproc()->killed){
      release(&p->lock);
      return -1;
    }
    sleep(&p->nread, &p->lock);
  }
  for(i = 0; i < n; i++){
    if(p->nread == p->nwrite)
      break;
    addr[i] = p->data[p->nread++ % PIPESIZE];
  }
  wakeup(&p->nwrite);
  release(&p->lock);
  return i;
}
```

Objective 1:
Both are in the file *pipe.c*. Code snippets for the two methods are given above.

Objective 2:
piperead is comprised of two loops with conditional statements and CV routines.
pipewrite is comprised of a nested loop with conditional statements and CV routines.

Objective 3:
Lock used: *p→lock*
piperead wakes *p→nwrite*
pipewrite wakes *p→nread*

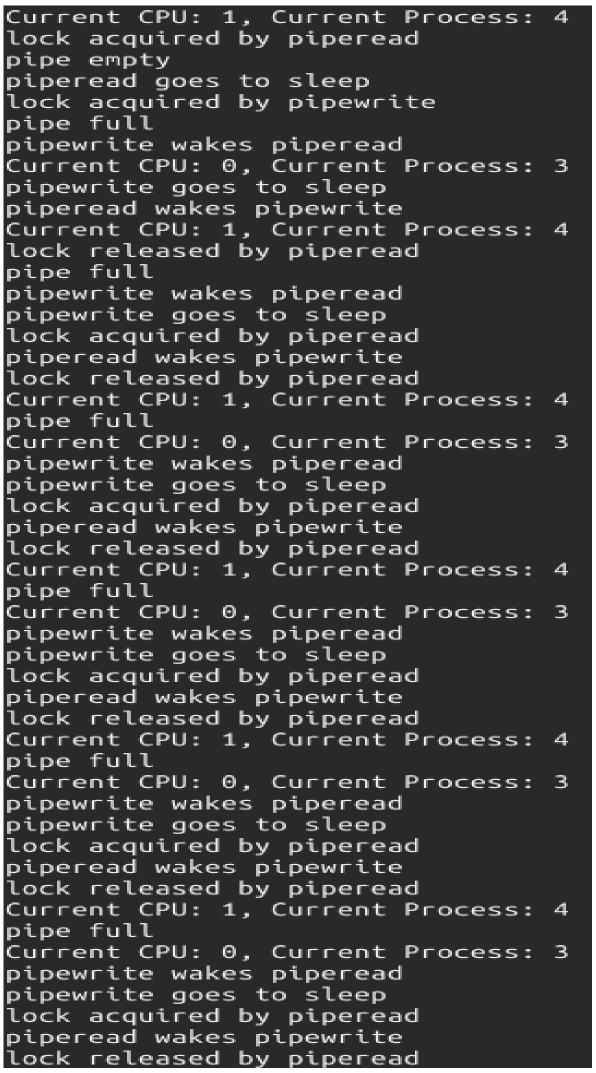
Objective 4:
Each pipe is represented by a struct pipe, which contains a lock and a data buffer. The fields *nread* and *nwrite* count the total number of bytes read from and written to the buffer. The buffer wraps around but the counts do not wrap, which implies we must use circular indexing.
Condition for full buffer: (*nwrite* == *nread* + PIPESIZE)
Condition for empty buffer: (*nwrite* == *nread*)

Assume that calls to *piperead* and *pipewrite* happen simultaneously on two different CPUs. *pipewrite* begins by acquiring the pipe's lock, which protects the counts, the data, and their associated invariants. *piperead* cannot acquire the lock at this moment. It spins in acquire waiting for the lock. While *piperead* waits, *pipewrite* loops over the bytes being written as shown in the code snippet, adding each to the pipe in turn. During this loop, it could happen that the buffer gets filled. In that case, case, *pipewrite* calls wakeup to alert any sleeping readers to the fact that there is data waiting in the buffer and then sleeps on &*p→nwrite* to wait for a reader to take some bytes out of the buffer as described in the sleep routine summary. Sleep releases *p→lock* as part of putting *pipewrite*'s process to sleep.

Now that *p→lock* is available, *piperead* manages to acquire it and enters its critical section: it finds that buffer is not empty it falls through to the for loop, copies data out of the pipe and increments *nread* by the number of bytes copied. That many bytes are now available for writing, so *piperead* calls wakeup to wake any sleeping writers before it returns. Wakeup finds a process sleeping on *&p→nwrite*, the process that was running *pipewrite* but stopped when the buffer filled. It marks that process as RUNNABLE as described in the wakeup routine summary.

The pipe code uses separate sleep channels for reader and writer (*p→nread* and *p→nwrite*). This might make the system more efficient in the unlikely event when there are lots of readers and writers waiting for the same pipe. The pipe code sleeps inside a loop checking the sleep condition. If there are multiple readers or writers, all but the first process to wake up will see the condition is still false and sleep again.

Objective 5:
Refer to *sleep_test1.c*. Add it to UPROGS in makefile before running xv6. This test performs simple fork and pipe read/write.
Note: Test process IDs are 3 and 4.



```
Current CPU: 1, Current Process: 4
lock acquired by piperead
pipe empty
piperead goes to sleep
lock acquired by pipewrite
pipe full
pipewrite wakes piperead
Current CPU: 0, Current Process: 3
pipewrite goes to sleep
piperead wakes pipewrite
Current CPU: 1, Current Process: 4
lock released by piperead
pipe full
pipewrite wakes piperead
pipewrite goes to sleep
lock acquired by piperead
piperead wakes pipewrite
lock released by piperead
Current CPU: 1, Current Process: 4
pipe full
Current CPU: 0, Current Process: 3
pipewrite wakes piperead
pipewrite goes to sleep
lock acquired by piperead
piperead wakes pipewrite
lock released by piperead
Current CPU: 1, Current Process: 4
pipe full
Current CPU: 0, Current Process: 3
pipewrite wakes piperead
pipewrite goes to sleep
lock acquired by piperead
piperead wakes pipewrite
lock released by piperead
Current CPU: 1, Current Process: 4
pipe full
Current CPU: 0, Current Process: 3
pipewrite wakes piperead
pipewrite goes to sleep
lock acquired by piperead
piperead wakes pipewrite
lock released by piperead
Current CPU: 1, Current Process: 4
pipe full
Current CPU: 0, Current Process: 3
pipewrite wakes piperead
pipewrite goes to sleep
lock acquired by piperead
piperead wakes pipewrite
lock released by piperead
```

Running Tests [IMPORTANT]

There are 4 tests attached along with the report.
file_test1.c → Testing Critical Section 1 of ftable.lock
file_test2.c → Testing Critical Section 2 of ftable.lock
file_test3.c → Testing Critical Section 3 of ftable.lock
sleep_test1.c → Simulating piperead and pipewrite.

You need to use the attached xv6 build to run the following tests. Uncomment the print statements at the locations described in objective 1 for each section to get necessary prompts of when a lock is being acquired, released, pipe full/empty etc.

Refer to the screenshot images along with report if the image quality in the report is poor.