CS537 P3b Report

Linxiu Zeng and Hairong Yin

November 24, 2021

1 Lock Analysis

1.1 ftable.lock

The *ftable.lock* is a spinlock defined in *struct ftable* within *file.c*, and it can be initialized when the function *fileinit* is called.

```
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
```

Code 1 (from file.c)

```
#define NFILE       100  // open files per system
```

Code 2 (from param.h)

The *struct ftable* includes one spinlock named lock and an array of *struct file* defined in *file.h*. Namely, it is a table that is able to contain *NFILE*(a number defined in *param.h*) opening files per system.

Within the file, there are in total 3 functions that contains all 3 critical sections which apply this spin lock. Those functions are *filealloc()*, *filedup()*, and *fileclose()*. More detail explanations of how this lock is implemented will be provided in the following subsections.

1.1.1 filealloc() – First critical section

```
1 // Allocate a file structure.
2 struct file*
3 filealloc(void)
4 {
5   struct file *f;
6 
7   acquire(&ftable.lock);
8   for(f = ftable.file; f < ftable.file + NFILE; f++){
9     if(f->ref == 0){
10      f->ref = 1;
11      release(&ftable.lock);
12      return f;
13    }
14  }
15  release(&ftable.lock);
16  return 0;
17 }
```

Code 3 (from file.c)

(a)Critical section length analysis:

This critical section goes from line 7 to line 15 in Code 2 with length of 3 instructions, while that those instructions are encompassed by a for loop. The for loop may be executed many times with lock held before hitting the release instruction.

(b)Critical section behavior analysis:

The *filealloc()* function itself creates new files and also allocate space for this file in the file table. The for loop traverses through the entire ftable and check whether the current file has already been referenced. If ref is 0, meaning that the current file is not in-use, grab this file, mark its status to 1 (in-use), and return the renewed file table. Or, if the entire file table has not empty space, return 0.

The lock is acquired before the for loop and has two release conditions. The first condition is that if there exists at least 1 available file, release the lock before the return statement. The other is that if the entire file table is busy, then release the lock before returning 0. Therefore, the lock here ensures that only one thread can loop through the current file table, hence there is only one thread can make change of the file table in one system at once.

If multiple threads are running this section at the same time, what each of them can do is to create 1 pointer pointing to the current file table. So multiple pointers will exist in one specific time, but the ftable lock limits the number of threads accessing and updating the file table to 1 to protect the possible race condition occurred in the file table. Assume this section is no surrounded by the ftable.lock and two threads run this section concurrently. Then, for example, when the first thread finish executing the if statement on line 9, meaning that the for loop does find an available file at location f=x. At this time, time interrupt occurs, the second thread therefore finds the same f=x location since the file table has not been updated by thread 1. So thread 2 takes location x and exits the function. But then when thread 1 is released, it overwrites the info and exits the function. Now one information leak occurs! The change done by thread 2 is overwritten by thread 1, which creates a race condition.

Oppositely, if the lock is correctly placed and is held by thread 1, then when thread 2 wants to access the critical section, it will spins (stuck in the while loop) and wait for thread 1 to finish execution.

(c)Frequency analysis:
This function is called by the *sys_open()* function in *sysfile.c* and by the *pipealloc()* function in *pipe.c*. Basically, how often the lock is acquired and released is the same as how often the function that contains this critical section is called by corresponding higher level functions. If the number of higher level functions that calls this function is large, then the frequency of the lock is relatively high. Since this function is only involved in two higher level function calls, its frequency is relatively low.

1.1.2 filedup() – Second critical section

```
1 // Increment ref count for file f.
2 struct file*
3 filedup(struct file *f)
4     {
5         acquire(&ftable.lock);
6         if(f->ref < 1)
7             panic("filedup");
8         f->ref++;
9         release(&ftable.lock);
10        return f;
11    }
```

Code 4 (from file.c)

(a)Critical section length analysis:
The critical section goes from line 5 to line 9 in Code 4, so the instruction length is 3.

(b)Critical section behavior analysis:
What the critical section is trying to do is to duplicate the file. In order to do that, it needs to increase the ref count of the file by 1. However, before increasing, we need to check whether the ref count of the file is at least 1, else that it means this file is not accessible, then we simply throw error (line 7 in Code 4).
Now we analyze the potential contention for the *ftable.lock* by case if the lock is not added.

<u>Case 1:Threads contention for line 6 in Code 4</u>
Assume that there are two threads running at the same time, called thread 1 and thread 2. Thread 1 is trying to call *filedup()* function on an open file, denoted as file x, while that thread 2 is trying to close file x. Let thread 1 runs first, and it then hits line 6 in Code 4. Now the ref count of input file x is 1, so thread 1 jumps to line 8, but haven't execute line 8. Now scheduler interrupt thread 1 and let thread 2 start to run to finish. After thread 2 finished running, file x has been closed. Then the scheduler let thread 1 continue running line 8 in Code 4. However, since now the file x has been closed, then it is not accessible. Thus it will lead to error when thread 1 is trying to increase ref count of file x.

<u>Case 2:Threads contention for line 8 in Code 4</u>
Assume that there are two threads running at the same time, called thread 1 and thread 2. Both of them are trying to call *filedup* on open file x at the same time. Let thread 1 run first, but is interrupted by scheduler it when thread 1 hasn't yet updated the increased number to the register that stores the value of *f->ref*. Denote this number as m. Now suppose the scheduler let thread 2 to run to finish, then the register that stores the value of *f->ref* has been increased by 1. Then thread 1 runs again. But it would not recheck the current number of the register, but simply put m into the register. Therefore, even though line 9 in Code 4 has been executed twice by both of the threads, the number stored in the register is still *f->ref+1*, which is incorrect.

Oppositely, when the lock is placed correctly, if the lock is already held by thread 1, then when thread 2 wants to access the critical section, it will spins (stuck in the while loop) and wait for thread 1 to finish execution. In this case, both of the cases work correctly.

(c) Frequency analysis:
The function *filedup()* is called by the *sys_dup()* function in *sysfile.c* and by the *fork()* function in *proc.c*. Basically, how often the lock is acquired and released is the same as how often the function that contains this critical section is called by corresponding higher level functions. If the number of higher level functions that calls this function is large, then the frequency of the lock is relatively high. Since this function is only involved in two higher level function calls, its frequency is relatively low.

1.1.3 fileclose() – Third critical section

```
// Close file f.  (Decrement ref count,
    close when reaches 0.)
void
fileclose(struct file *f)
{
  struct file ff;

```

```
7   acquire(&ftable.lock);
8   if(f->ref < 1)
9     panic("fileclose");
10  if(--f->ref > 0){
11    release(&ftable.lock);
12    return;
13  }
14  ff = *f;
15  f->ref = 0;
16  f->type = FD_NONE;
17  release(&ftable.lock);
18
19  if(ff.type == FD_PIPE)
20    pipeclose(ff.pipe, ff.writable);
21  else if(ff.type == FD_INODE){
22    begin_op();
23    iput(ff.ip);
24    end_op();
25  }
26 }
```

Code 6 (from file.c)

(a)Critical section length analysis:

The first part of its critical section goes from line 7 to line 10 before the first potential release instruction; whereas the second part goes from line 14 to 16 in Code 6 before the second release instruction, and the total length of the instructions is 6.

(b)Critical section behavior analysis:

The *fileclose()* function is called while closing up a file, given the location of that file in the current file table *f* as parameter. It first creates a file *ff* and put the critical section before the first two condition checks and ends after the values of *ff*, *f -> ref*, and *f -> type* are changed. So, the logic of this critical section is as following: the function first checks the status of the current file table, if it is available, then it decrements the ref value and compares it with 0, otherwise it throws error message and continues. If the *p->ref* value is larger than 0, meaning that it is still in use, the function releases the lock and exits this function. If not, then this file is not in use (value 0 or lower after decrementing *f->ref*), so the function refers the pointer to current file table to the new-created file, assigns 0 to *f->ref*, and changes the file type to *FD_NONE*.

Therefore, the lock ensures that race conditions will not occur while doing all previous steps. To be more clear, let's assume there are two threads executing this critical section when no locks are provided. When thread 1 runs to line 10 in Code 6, the value of *f->ref* decreases from 1 to 0, and is about to assign the content of *f* to *ff*. At this time, context switch occurs, and thread 2 runs into the critical section and is caught by the first if condition. So thread 2 panics. But at this time, the file type whose value should be *FD_NONE* is not yet changed, which is incorrect. Therefore, it is clear that the lock is necessary.

Oppositely, if the lock is correctly placed and is held by thread 1, then when thread 2 wants to access the critical section, it will spins (stuck in the while loop) and wait for thread 1 to finish execution.

(c)Frequency analysis:

This function is called by *pipealloc()* in *pipe.c*; *exit()* in *proc.c*; *sys_close()*, *sys_open()*, and *sys_pipe()* in *sysfile.c*. Basically, how often the lock is acquired and released is the same as how often the function that contains this critical section is called by corresponding higher level functions. If the number of higher level functions that calls this function is large, then the frequency of the lock is relatively high. In total the function *fileclose()* has 5 upper functions, so its frequency of being called is relatively low.

1.1.4 Overall Frequency Analysis

The file table uses function *filealloc()* to allocate a file, function *filedup()* to create a duplicate reference, and function *fileclose()* to release a reference. The frequency of acquire and release of a lock depends on how often the function they lied in is called by higher-level function, which we have analyzed in the previous subsection. Therefore, the way we analyze the frequency of a single lock is to combine the frequency of every single critical section which we have already analyzed. In this case, the overall frequency of *ftable.lock* is low.

1.2 bcache.lock

The *bcache.lock* is a spinlock, and it is defined in *struct bcache* within the file *bio.c*, in which the lock can be initialized when the function *binit()* is called. There are two functions inside *bio.c* that acquired the bcache.lock, i.e. there are two critical sections which has used the bcache.lock. The two functions are *bget()* and *brelse()*.More detail explanations of how this lock is implemented will be provided in the following subsections.

1.2.1 bget() – First critical section

```
// Look through buffer cache for block
    on device dev. Return locked buffer.
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;
  acquire(&bcache.lock);
```

```
7   for(b = bcache.head.next; b != &bcache
      .head; b = b->next){
8     if(b->dev == dev && b->blockno ==
      blockno){
9       b->refcnt++;
10      release(&bcache.lock);
11      acquiresleep(&b->lock);
12      return b;
13    }
14  }
15  // Not cached; recycle an unused
      buffer.
16  for(b = bcache.head.prev; b != &bcache
      .head; b = b->prev){
17    if(b->refcnt == 0 && (b->flags &
      B_DIRTY) == 0) {
18      b->dev = dev;
19      b->blockno = blockno;
20      b->flags = 0;
21      b->refcnt = 1;
22      release(&bcache.lock);
23      acquiresleep(&b->lock);
24      return b;
25    }
26  }
27  panic("bget: no buffers");
28 }
```

Code 7 (from bio.c)

(a)Critical section length analysis:

The first part of critical section goes from line 7 to line 9 before the first potential release instruction, and the second part of the critical section goes from line 16 to line 21 in Code 7 before the second release instruction. The total length of instructions of the critical section is 9, while that those instructions are encompassed by a for loop. The for loop may be executed many times with lock held before hitting the release instruction.

(b)Critical section behavior analysis:

What the critical section is trying to do is to look through buffer cache for block on device dev. If a cached block is found, increase its ref count and then release the spinning lock. Else it means that no cached buffer for the indicated block, then we allocate an unused buffer and release the spinning lock.

Now we analyze the potential contention for the *bcache.lock* by case if the lock is not added.

<u>Case 1:Threads contention for line 9 in Code 7</u>

Assume that there are two threads running at the same time, called thread 1 and thread 2. Both of them are trying to call *bget()* at the same time and they are trying to get the same block that is cached already. Then if line 9 in Code 7 is not locked by *bcache.lock*, it means that thread 1 may be interrupted by the scheduler when it is executing line 9 but haven't yet been able to store the updated value back into the register that stores the value of *b->refcnt*. Now the scheduler let thread 2 run to finish, and then let thread 1 run again. But thread 1 would not recheck the current number of the register, but simply put the previous value into the register, in which it still equals to *b->refcnt+1*. Therefore, even though line 9 in Code 7 has been executed twice by both of the threads, the number stored in the register is still *b->refcnt+1*, which is incorrect.

<u>Case 2:Threads contention for line 17 in Code 7</u>

Assume that there are two threads running at the same time, called thread 1 and thread 2. Thread 1 is trying to recycle an unused buffer, denoted as buffer x, while that thread 2 is trying to cache buffer x. Then if line 17 and the following lines are not locked, when thread 1 just finished executing line 17 and get TRUE for the if statement, but not yet go into the following lines to cache the block, scheduler may interrupt it and let thread 2 run to finish. Now the buffer x is cached, when thread 1 continues to run, it would falsely recycle buffer x in which that it has now been cached by thread 2.

<u>Case 3:Threads contention for loops in Code 7</u>

Assume that there are two threads running at the same time, called thread 1 and thread 2. Then it may happen that when thread 1 just updated *b* to be *b->next*, it is interrupted by the scheduler and thread 2 runs, changing *b->next* to be another block. Now when thread 1 runs again, it may not be able to correctly loop through all the blocks in the buffer. W.L.O.G. for the other loop that check by updating *b* to be *b->prev*.

Oppositely, if the lock is correctly placed and is held by thread 1, then when thread 2 wants to access the critical section, it will spins (stuck in the while loop) and wait for thread 1 to finish execution, so all of the case will work correctly.

(c)Frequency analysis:

This function is onlyc called by *bread()* in *bio.c*. Basically, how often the lock is acquired and released is the same as how often the function that contains this critical section is called by corresponding higher level functions. If the number of higher level functions that calls this function is large, then the frequency of the lock is relatively high.

1.2.2 brelse() – Second critical section

```
// Release a locked buffer.
// Move to the head of the MRU list.
void
brelse(struct buf *b)
{
```

```
6    if(!holdingsleep(&b->lock))
7      panic("brelse");
8
9    releasesleep(&b->lock);
10
11   acquire(&bcache.lock);
12   b->refcnt--;
13   if (b->refcnt == 0) {
14     // no one is waiting for it.
15     b->next->prev = b->prev;
16     b->prev->next = b->next;
17     b->next = bcache.head.next;
18     b->prev = &bcache.head;
19     bcache.head.next->prev = b;
20     bcache.head.next = b;
21   }
22
23   release(&bcache.lock);
24 }
```

Code 8 (from bio.c)

(a)Critical section length analysis:

This critical section goes from line 12 to line 21 in Code 8, so the instruction length is 8.

(b)Critical section behavior analysis:

This critical section mainly does two things for which the atomicity should be ensured. The first one is to decrease the value of *b->refcnt* by 1, the other is to assign the position of node d from its current position to the second node of the entire linked list if the value of *b->refcnt* is 0, that is, if there is no one waiting for buffer b.

According to the previous discussion, there are two possible race conditions. And this can be seen by cases with the prerequisites of removing the lock from each of them and assign two threads to run this code block concurrently.

<u>Case 1:Threads contention for line 12 in Code 8</u>

If moving the critical section after the minus instruction, then the following situation may occur. That is, the timer interrupt may occurs right after thread 1 just finish loading and incrementing the number but not yet storing the number back. At this time, thread 2 enters the critical section and runs till complete the entire section. Then, the same location will be updated twice when thread 1 rewrites the same value of *b->refcnt*, which results in a different answer as expected.

<u>Case 2:Threads contention occurs in any line between line 15 and line 20 in Code 8</u>

If removing the lock around this chunk of code (line 15 to line 20), then the following situation may occur. Thread 1 and thread 2 may execute any line concurrently in this chunk, which ends up with a random and incorrect answer due to the race condition of switching the assigned value of nodes within the linked list, similarly as explained in section 1.2.1 Case 3.

Oppositely, if the lock is correctly placed and is held by thread 1, then when thread 2 wants to access the critical section, it will spins (stuck in the while loop) and wait for thread 1 to finish execution, so all cases will work correctly.

(c)Frequency analysis:

This function is called by the following functions: *readsb()*, *bzero()*, *balloc()*, *bfree()*, *ialloc()*, *iupdate()*, *ilock()*,*bmap()*, *itrunc()*, *readi*, and *writei* in file *fs.c*; *install_trans*, *read_head*, and *write_log* in *log.c*. Basically, how often the lock is acquired and released is the same as how often the function that contains this critical section is called by corresponding higher level functions. If the number of higher level functions that calls this function is large, then the frequency of the lock is relatively high.

1.2.3 Overall Frequency Analysis

The frequency of acquire and release of a lock depends on how often the function they lied in is called by higher-level function, which we have analyzed in the previous subsection. If the frequency of higher level functions calls of this function is high, then the frequency of the lock is relatively high, or vice versa. In our case, the *bget()* function is only called by *bread()* while checking if a block is cached, and the *bread()* is called if and only if some indicated block is required.

2 *sleep* and *wakeup* Analysis

2.1 Function Analysis

(a)Function *sleep()*:

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();
  if(p == 0)
    panic("sleep");
  if(lk == 0)
    panic("sleep without lk");
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
  //Go to sleep
  p->chan = chan;
  p->state = SLEEPING;
  sched();
  //Tidy up
  p->chan = 0;
  // Reacquire original lock.
  if(lk != &ptable.lock){
    release(&ptable.lock);
    acquire(lk);
  }
```

```
24 }
```

Code 9 (from proc.c)

This function mainly is trying to atomically release lock and sleep on the input channel. Since the process of going to sleep needs to change the current process's state into SLEEPING and update its channel to the input channel (line 14-15 in Code 9), we need to make sure it is done atomically. Thus, before releasing the input spinlock, we first need to check whether the *ptable.lock* has been required. If the input spinlock is not *ptable.lock*, then we need to acquire *ptable.lock* first, and then release the original lock in the next step (line 9-12 in Code 9). Then we call the function *sched()* to enter the scheduler. When we are returned from the scheduler, it means that this process has been waked up, so we change its channel back to 0 and release the *ptable.lock* if it is not the original lock, while also reacquire the original lock.

(a)Function *wakeup()*:

```
// Wake up all processes sleeping on chan.
static void
wakeup1(void *chan)
{
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

Code 10 (from proc.c)

The function *wakeup1()* mainly is trying to wake up all the processes that are sleeping on the input channel by iterating each process on the ptable and change its state to be RUNNABLE if it is sleeping. Since this needs to be done atomically, we also use another function *wakeup()* which simply acquire the *ptable.lock* before we call function *wakeup1()*, and release *ptable.lock* after *wakeup1()* has done.

2.2 Example 1: pipe.c

```
1 int
2 pipewrite(struct pipe *p, char *addr, int n)
3 {
4   int i;
5   acquire(&p->lock);
6   for(i = 0; i < n; i++){
7     while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
8       if(p->readopen == 0 || myproc()->killed){
9         release(&p->lock);
10        return -1;
11      }
12      wakeup(&p->nread);
13      sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
14    }
15    p->data[p->nwrite++ % PIPESIZE] = addr[i];
16  }
17  wakeup(&p->nread);  //DOC: pipewrite-wakeup1
18  release(&p->lock);
19  return n;
20 }
21
22 int
23 piperead(struct pipe *p, char *addr, int n)
24 {
25  int i;
26  acquire(&p->lock);
27  while(p->nread == p->nwrite && p->writeopen){  //DOC: pipe-empty
28    if(myproc()->killed){
29      release(&p->lock);
30      return -1;
31    }
32    sleep(&p->nread, &p->lock); //DOC: piperead-sleep
33  }
34  for(i = 0; i < n; i++){  //DOC: piperead-copy
35    if(p->nread == p->nwrite)
36      break;
37    addr[i] = p->data[p->nread++ % PIPESIZE];
38  }
39  wakeup(&p->nwrite);  //DOC: piperead-wakeup
40  release(&p->lock);
41  return i;
42 }
```

Code 11:(from pipe.c)

Function Analysis:

The write function begins by acquiring the spinlock to avoid any possible race conditions. Then, it checks whether the current buffer is full on line 7 of Code 11 — yes means that there are no space available to write any extra contents in, and no means there are space available to write more things down.

Therefore, the write must be put to sleep and wait until the read function to clean up the buffer if the buffer is full if the buffer is filled up to its most extent. This process has been done in line 12 and 13 of Code 11, where the *pipewrite()* function calls *wakeup()* to alert any sleeping reader and calls *sleep()* to put the current writer to sleep while releasing the spinlock. Then, after executing the entire for loop, one reader needs to be waken up to consume the content inside the current buffer, which is shown by the *wakeup()* function in line 17. Now, the write function is done with its job, so it release the lock and shifts the role to the *piperead()* function.

W.L.O.G. the read function acts like the write function because the overall structure and arrangement of the *sleep()* and *wakeup()* in *piperead()* is symmetrical to what is demonstrated in *pipewrite()*.

The *sleep()* and *wakeup()* functions in the *pipe.c* use the producer and consumer structure, whereas the write plays the role of producer and the read plays the role of consumer.

Interaction Analysis:

Let's suppose there are two threads (i.e. thread 1 and thread 2) calls *pipewrite()* and *piperead()* simultaneously on two different CPUs.

Suppose thread 1 calls the function *piperead()* acquires the *p->lock* successfully, then thread 2 that calls function *pipewrite()* cannot acquire it, so thread 2 just wait and spins. However, now the pipe is empty, if the process is not killed, thread 1 will go to line 32 in Code 11 and goes to sleep. Now the function *sleep* will release *p->lock*, and now thread 2 is able to acquire *p->lock*. Since the pipe is currently empty, thread 2 will loop over the bytes being written (line 15 in Code 11), adding each to the pipe. During this loop, it could happen the the buffer is full. In this case, thread 2 calls *wakeup()* in line 12 of Code 11 to alert any sleeping readers. Then thread 2 call *sleep()* to wait for a reader to take some bytes out of the buffer. Same, the *sleep()* function releases *p->lock*.

Now the *sleep()* function called by thread 1 keeps running and acquires the *p->lock* when returning. Then thread 1 keeps running in *piperead()* and executes line 34-38 in Code 11 to copy data out of the pipe, in which it also increments *nread* by the number of bytes copied. Now we have those bytes that are ready for writing, so thread 1 now calls *wakeup()* to wake any sleeping writers before it returns. According to the implementation of *wakeup()*, it finds a thread sleeping on *pi->nwrite*, which is thread 2. It marks thread 2 as RUNNABLE.

When there are multiple readers or writers, each call of *wakeup()* only wakes up the first thread and all the other threads will find the condition is still false and sleep again. This is because the pipe code sleeps inside a loop checking the sleep condition.

2.3 Example 2: pipe.c

```
1 // called at the start of each FS system call.
2 void
3 begin_op(void)
4 {
5   acquire(&log.lock);
6   while(1){
7     if(log.committing){
8       sleep(&log, &log.lock);
9     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
10      // this op might exhaust log space; wait for commit.
11      sleep(&log, &log.lock);
12    } else {
13      log.outstanding += 1;
14      release(&log.lock);
15      break;
16    }
17  }
18 }
19
20 // called at the end of each FS system call.
21 // commits if this was the last outstanding operation.
22 void
23 end_op(void)
24 {
25   int do_commit = 0;
26   acquire(&log.lock);
27   log.outstanding -= 1;
28   if(log.committing)
29     panic("log.committing");
30   if(log.outstanding == 0){
31     do_commit = 1;
32     log.committing = 1;
33   } else {
34     // begin_op() may be waiting for log space,
35     // and decrementing log.outstanding has decreased
36     // the amount of reserved space.
37     wakeup(&log);
38   }
39   release(&log.lock);
40   if(do_commit){
41     // call commit w/o holding locks, since not allowed
42     // to sleep with locks.
43     commit();
44     acquire(&log.lock);
45     log.committing = 0;
46     wakeup(&log);
47     release(&log.lock);
48   }
49 }
```

Code 12:(from log.c)

Function Analysis:

The *begin_op()* begins by acquiring the spinlock to avoid any possible race conditions. And then the function will increment the number of *log.outstanding*(number of FS system calls that is currently executing) showed by line 13 in Code 12 and also releases the acquired lock in line 14 in Code 12 unless the logging system is currently committing, or there is no enough unreserved log space to hold the writes from this call. The log sleeps in both of the failing conditions as described above.

The *end_ op()* should be called at the end of each FS system call. It also acquires the lock when function starts to ensure the safety of data within the global struct log. Then the function decrements the *log.outstanding* (number of FS system calls that is currently executing). If the count is now zero, it commits the current transaction by changing the value of *do_ commit* and *log.committing* to 1 (Line 31 and 32) and calling *commit()* (Line 43) in Code 12. Returned from *commit()*, the value of *log.committing* is changed back to 0 to mark the completion of committing the current process. Then, the *end_ op()* wakes up the log process to start the next round of transaction (Line 46 in Code 12).

Else, if the count is not zero, then it means that the current one is not the last outstanding operation, therefore, this function only needs to wake up the log system call and releases the lock (Line 37 in Code 12). Since in this condition, the value of *do_ commit* has not been set to 1, the function will not enter the if condition.

The *sleep()* and *wakeup()* functions in these two functions use the producer and consumer structure.

Interaction Analysis:

Let's suppose there are two threads (i.e. thread 1 and thread 2) calls *begin_ op* and *end_ op()* simultaneously on two different CPUs, in which that thread 1 just create an FS system call, and thread 2 has already *begin_ op()* once and now comes to the end of its FS system call.

Suppose at some time thread 1 calls the function *begin_ op()* acquires the *p->lock* successfully, then thread 2 that calls function *end_ op()* cannot acquire it, so thread 2 just wait and spins. It could happen that thread 1 find the op exhaust log space (line 9 of Code 12), then it also call *sleep()*. In both cases, *sleep()* will release *p->lock*, and now thread 2 could acquire the lock and keeps running.

After decreasing *log.outstanding* by one, it equals to 0 now. So thread 2 has set *do_ commit* to be 1 and *log_ committing* to be. Then thread 2 will go into the if statement from line 40 to line 48 in Code 12. Among them, when thread 2 executed *wakeup()* (line 46) thread 1 has been waken up. However, thread 1 can only return from the *sleep()* when thread 2 has finished running and release the *log.lock*, in which that now thread 1 get the lock again. By running through the while loop again, now thread 1 is able to add one to *log.outstanding* successfully.

Line 7 in code 12 that checks *log.committing* guarantees that all other threads sleeps until the committing thread finished. Also, in the case that there are multiple threads that are sleeping when running *begin_ op*, there is still only one thread could be waken up by the thread that just finished calling *end_ op* because of the while loop that wraps all the *sleep()* function.