CS 537: Intro to Operating Systems

Analyzing xv6 Locks and Conditions

Aalyaan Feroz[1] and Salman Munaf[1]

[1]University of Wisconsin - Madison

November 23, 2021

Introduction

In this report, we will talk about how locks are implemented in xv6. Two types of locks are present in xv6. These are either spinlocks or sleep locks. For part 1 of this project, we will be analyzing two instances of spin locks. For part 2, we will be analyzing how two instances of the sleep() and wakeup() routines work in the kernel.

1 Part 1: Spinlocks

Spinlock is a locking system mechanism. It allows a thread to acquire it to simply wait in loop until the lock is available i.e. a thread waits in a loop or spin until the lock is available. These locks are initialized using the initlock() function:

```
void
initlock(struct spinlock *lk, char *name)
{
  lk->name = name; // passed for debugging
  lk->locked = 0;
  lk->cpu = 0;
}
```

Spinlock is usually held for a short period of time so that not a lot of CPU cycles are wasted spinning. To implement spinlocks, xv6 relies on a special 386 hardware instruction, xchg. This instruction returns the old value present in the locked variable, and simultaneously updates said value to new value. The point to note here is that this instruction is done **atomically**. So, in a while loop a thread will check if the lock is held. If not, it will keep spinning and keep testing until it is released by another thread. Once the lock is released (xchg will return 0), this new thread will set the locked variable again to 1 atomically through the xchg instruction making sure no other thread can enter the critical section.

Now lets look at the **acquire** function below. The acquire function disables interrupts before trying to acquire a lock. This is done by the routine pushcli() to avoid deadlocks. Once it does that, it checks to see if the lock is already held. It raises panic if this is true. After that, the acquire function uses the xchg() function to check if lock is held (lk->locked will be 1). If this is the case, the xchg function returns 1 and keeps looping. If xchg returns 0, this means that the lock has been successfully acquired (lk->locked **was** 0, now is 1) and the while loop terminates. In the end, the acquire function records some details such as the cpu holding the lock and the call stack. It uses the getcallerpcs() function for this.

```
void
acquire(struct spinlock *lk)
{
  pushcli();
  if(holding(lk))
    panic("acquire");

  while(xchg(&lk->locked, 1) != 0)
    ;

  __sync_synchronize();

  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

The release function does essentially the opposite of the acquire function. It checks if this cpu is holding the lock. If that is not the case, it throws a panic. Then it resets cpu and call stack for the lock. It also sets the locked variable to 0 atomically. Lastly, it enables interrupts using the popcli() function.

```
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  __sync_synchronize();

  asm volatile("movl $0, %0" : "+m" (lk->locked) : );

  popcli();
}
```

Performance

We will consider the costs of using spinlocks. To analyze this more carefully, imagine threads competing for the lock on a single processor. 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. This results in a waste of CPU cycles, causing inefficient performance. This is the reason behind using spin locks for small critical sections.

File table: ftable.lock

The global open file table ftable is an array of file structures, protected by a global ftable.lock. The per-process file table in struct proc stores pointers to the struct file in the global file table. All the open files in the system are kept in the file table. The file table has a function to allocate a file (filealloc), create a duplicate reference (filedup), release a reference (fileclose), and read and write data (fileread and filewrite). Here, we will evaluate the critical sections and locks of these functions, and also why locks are needed in these functions.

```
25 //code in file.c
26 struct file*
27 filealloc(void)
28 {
29   struct file *f;
30
31   acquire(&ftable.lock);
32   for(f = ftable.file; f < ftable.file + NFILE; f++){
33     if(f->ref == 0){
34       f->ref = 1;
35       release(&ftable.lock);
36       return f;
37     }
38   }
39   release(&ftable.lock);
40   return 0;
41 }
```

The filealloc() function iterates over the file table to find an unreferenced file. If it is found, it returns a new reference. Before it does that, it acquires the ftable lock. The critical section starts when it acquires the lock (line 31) and has two cases for it to end: One where this function updates the reference for the unreferenced and releases the lock (line 35), and the other case where no unreferenced file is found and the function releases the lock (line 39). Now we will consider why it is very important for the function to have a lock. It is possible for this function to be called by two threads. In figure 1, we find that this function can be called by sys_open() and sys_pipe() -> pipealloc(). If two threads call filealloc() and enter the critical section, and if there is no lock present, both threads can end up pointing to the same reference. For example, both threads can read the value of f->ref as 0 for the same file and then set it to 1 returning the same file reference. If a lock is present, then a thread will first find an unreferenced file and set its reference count to 1 and then release the lock. Then the second thread can enter the critical section and find a different unreferenced file. The lock here is to prevent from multiple threads pointing to the same reference and guaranteeing mutual exclusion.

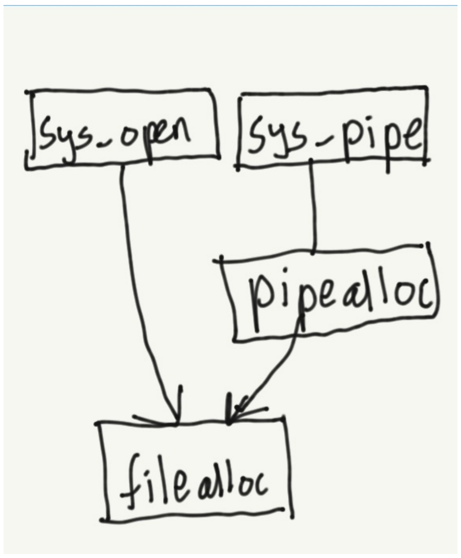

Figure 1: Trace of filealloc()

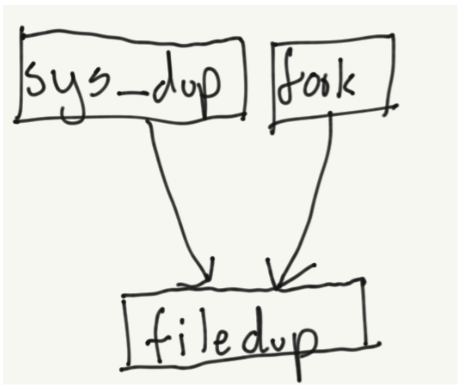

Figure 2: Trace of filedup()

```
43 // code in file.c
44 struct file*
45 filedup(struct file *f)
46 {
47   acquire(&ftable.lock);
48   if(f->ref < 1)
49     panic("filedup");
50   f->ref++;
51   release(&ftable.lock);
52   return f;
53 }
```

The filedup() function creates a duplicate reference to a file. Firstly, it checks to see whether the file is unreferenced. If it is, the function raises panic. If not, it increments the reference count for that file and returns the reference. The critical section for this function starts when the function acquires the lock (line 47). Checking if the file is unreferenced and incrementing the reference count, both these lines are part of the critical section. The critical sections ends when the lock is released (line 51) right after incrementing f->ref. It is also important for this function to have a lock otherwise the file reference count will be inconsistent. In figure 2 (below), we find that this function can be called by sys_dup() and fork(). If two threads enter the function, and if there is no lock present, they can read the same value from memory, increment it

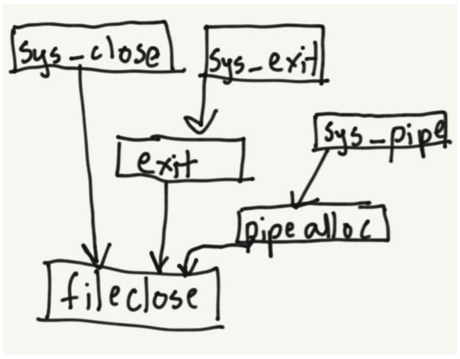

Figure 3: Trace of fileclose()

and then store it back to the memory location. This will result in an increment by only 1 even though two threads called the filedup() function. Lock ensures that the count remains consistent and only one thread can read and write the value at a time.

```
55 // code in file.c
56 void
57 fileclose(struct file *f)
58 {
59   struct file ff;
60
61   acquire(&ftable.lock);
62   if(f->ref < 1)
63     panic("fileclose");
64   if(--f->ref > 0){
65     release(&ftable.lock);
66     return;
67   }
68   ff = *f;
69   f->ref = 0;
70   f->type = FD_NONE;
71   release(&ftable.lock);
72
73   if(ff.type == FD_PIPE)
74     pipeclose(ff.pipe, ff.writable);
75   else if(ff.type == FD_INODE){
76     begin_op();
77     iput(ff.ip);
78     end_op();
79   }
80 }
```

The fileclose() function closes the file. It does so by decrementing the reference count, and closes the file completely when the reference count reaches zero. If the reference count is not zero, it does not completely close the file. The critical section for this function starts when the ftable lock is acquired (line 61) and ends in two different cases: one where the reference count is not zero after decrementing. In this case the function decrements the ref count, and checks whether it is greater than zero. If so, it releases the lock (line 65). The other case is when the reference count is 0 after decrementing it. In this case, the ref count is decremented and the reference is set to 0 (file is unreferenced), type of file is set to FD_NONE and the lock is released (line 71). We will see again why the lock is important for this function. In figure 3, we find that this function can be called by sys_close(), sys_exit(), and sys_pip(). If two threads call the fileclose() function, and a lock is not present, they can read the same value from memory, decrement it and then store it back to the memory location. This will result in a decrement by only 1 even though two threads called the function. This leads to inconsistent count and it can lead to a file being left open even though it has no reference. A lock ensures that only one thread can manipulate the file table at a time.

All the threads are sharing the same file table so it is necessary to use locks whenever entries are manipulated. In xv6, the above critical sections are run whenever open(), close() or dup() system calls are invoked. In order to investigate the behavior of locks further, we created a program in which two processes open different files. We added some print statements and extended the length of critical section as well in filealloc() by adding a while loop which incremented a count variable till 100000. We noticed that only one of the thread could enter the critical section and the other thread was waiting for the lock to be released by spinning and using the CPU cycles.

Buffer cache: bcache.lock

The buffer cache is simply an array of buffers and is used to cache disk contents. The buffer cache is a linked list of buf structures holding copies of disk block contents. Caching disk blocks in memory reduces the number of disk reads and also provides a synchronization point for disk blocks used by multiple processes. The buffer cache has a function to look for a particular sector in the buffer cache (bget) and a function to releases the lock (brelse). We will assess the critical section of these functions, how locks work in these functions, and why the locks are needed.

```
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;

  acquire(&bcache.lock);

  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }

  for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
    if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
      b->dev = dev;
      b->blockno = blockno;
      b->flags = 0;
      b->refcnt = 1;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }
  panic("bget: no buffers");
```

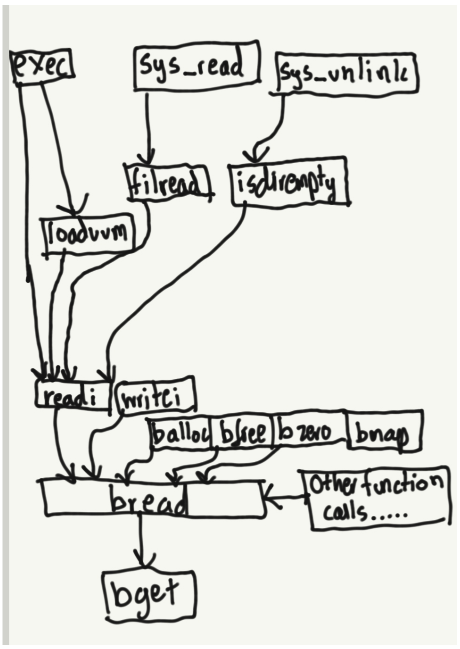

Figure 4: Trace of bget()

To read a disk block, the function bread() is called, which calls the bget() function (See figure 4). There are three possibilities in the bget() function. In the first possibility, bget() checks if the block is already cached. If that is true, it increments the reference count for the block and returns it. The second is if the block is cached but it is busy; bget() then sleeps on it, and after waking up it locks bcache. It does so by checking if the buffer corresponding to the sector number exists in the cache. It has to complete this check again because it possible that the respective sector could have been reused for another block while bget() slept on it. The third possibility is the block is not cached. In this case, the sector number must be fetched from the disk. In order to do this, the buffer cache needs to recycle an unused buffer. It looks for an unused buffer from the end of the linked list (due to LRU policy), and looks for the first clean, non busy buffer. When it is found, it allocates this buffer for the current block and marks it as invalid. The critical section for this function is from line 66-72 if the block is cached, and 66-87 if the block is not cached, where it recycles an unused buffer.

We stated before that the bget() function is called by bread(). Now, we will analyze the role of locks in the bget() function Let's consider when the two threads each call bread(). If the buffer cache is not locked, a race condition could occur where the two threads could point to the same block b or refcnt cant be inconsistent in case the block is cached. If a lock is present, let's say that a thread uses the bread() function to enter the bget() function. If another thread enters the bget() function and tries to acquire the lock, it cannot do so and the thread that requests to acquire the lock, will keep spinning. This thread will only be able to acquire the lock when the initial thread releases the lock.

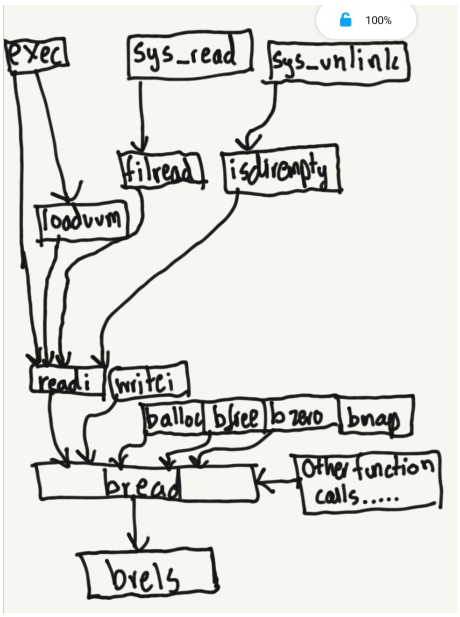

Figure 5: Trace of brelse()

This way, mutual exclusion is guaranteed and multiple threads are restricted from modifying the buffer cache at a single point in time.

```
119 //code in bio.c
120 void
121 brelse(struct buf *b)
122 {
123   if(!holdingsleep(&b->lock))
124     panic("brelse");
125
126   releasesleep(&b->lock);
127
128   acquire(&bcache.lock);
129   b->refcnt--;
130   if (b->refcnt == 0) {
131     // no one is waiting for it.
132     b->next->prev = b->prev;
133     b->prev->next = b->next;
134     b->next = bcache.head.next;
135     b->prev = &bcache.head;
136     bcache.head.next->prev = b;
137     bcache.head.next = b;
138   }
139
140   release(&bcache.lock);
141 }
```

Another function for the buffer cache is brelse(). When a process is done using a disk block, it calls brelse(). This function releases the lock by clearing the busy flag, so other processes sleeping on this block can start using this disk block contents again (whether clean or dirty). This function also moves this block to the head of the buffer cache, so that the LRU replacement algorithm will not replace it immediately. The critical section for this function starts at line 128 and ends at line 140. There are 18 function calls to brelse() (Figure 5) but we will only consider a few important ones. The first is balloc(). The function balloc() reads the free block bit map, finds a free block, fetches it into the buffer cache using bread(), zeroes it, and returns it. Another function call to brelse() is bfree(). This frees the block, which mainly involves updating the bitmap on disk. Note that all of these functions rely on the buffer cache and log layers, and do not access the device driver or disk directly. Two other functions are readi() and writei(). These used to read and write data at a certain offset in a file. Given the inode, the data block of the file corresponding to the offset is located, after which the data is read/written. The function bmap() returns the block number of the n-th block of an inode, by traversing the direct and indirect blocks. If the n-th block doesn't exist, it allocates one new block, stores its address in the inode, and returns it. This function is used by the readi() and writei() functions. Considering the readi() and writei() function call, we will consider why locks are needed for the brelse() function. Consider a thread that calls brelse() from the readi() function call. When it does this, it acquires the lock. Now, if another thread enters the brelse() function from the writei() function, it tries to acquire the lock, but cannot do so because the lock is already held, and the thread keeps spinning. The thread calling the writei() function will only be able to acquire the lock when the thread calling the readi() function will be released. This makes sense, as we do not want the writei() and readi() functions to be simultaneously called and affect the same buffer cache block. This also affirms mutual exclusion, which we have seen in locks.

These critical sections are invoked whenever a file is read or write from/to disk. As the call graph above shows that read() system call eventually invokes the bget() function. We created a user program where one process was reading data from a file and the other process was writing the data to the same file. We added some print statements and extended the length of critical section as well in bget() by adding a while loop which incremented a count variable till 100000. We noticed that only one of the thread could enter the critical section and the other thread was waiting for the lock to be released by spinning and using the CPU cycles.

For all the critical sections above, we should note that the length of these sections may be longer when compiled into assembly language. Each line of C code can translate into multiple instructions in x86 assembly.

2 Part 2: sleep() and wakeup() routines

xv6 provides sleep() and wakeup() functions, that are equivalent to the wait and signal functions of a conditional variable. The sleep and wake up functions must be invoked with a lock that ensures that the sleep and wake up procedures are completed atomically. A process that wishes to block on a condition calls sleep(chan, lock), where chan is any opaque handle, and lock is any lock being used by the code that calls sleep/wakeup. The sleep function releases the lock the process came with, and acquires a specific lock that protects the scheduler's process list (ptable.lock), so that it can make changes to the state of the process and invoke the scheduler. Once the scheduler is called, there is a context switch, and control goes back to where the scheduler was called once the process wakes up. This is done while the ptable.lock is held. Simultaneously, the sleep function release this lock not passed to it, reacquires the original lock passed to it, and returns to the process that has just woken up. A process that makes the condition true will invoke wakeup(chan) while it holds the lock. The wakeup() call makes all waiting processes runnable. Once wakeup makes the processes runnable, these processes can actually run when the scheduler is invoked at a later time.

wait() in proc.c

```
// code in proc.c
int
wait(void)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

The first instance where we see the use of the sleep() function is in the wait() function in proc.c. When a parent calls wait, the wait function acquires ptable.lock, and looks over all processes to find any of its zombie children. If none is found, it calls sleep(). When a child calls exit, it acquires ptable.lock, and wakes up its parent. When the parent wakes up in wait, it does the job of cleaning up its zombie child, and frees up the memory of the process. The wait() and exit() system calls provide a good use case of the sleep() and wakeup() functions.

```
void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

This sleep() function is called whenever a process calls wait() on a running children. On the other hand, the wakeup() function is called when a process calls the exit() function. It wakes up the parent waiting for it.

iderw() and ideintr() in ide.c

```
void
iderw(struct buf *b)
{
  struct buf **pp;

  if(!holdingsleep(&b->lock))
    panic("iderw: buf not locked");
  if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
    panic("iderw: nothing to do");
  if(b->dev != 0 && !havedisk1)
    panic("iderw: ide disk 1 not present");

  acquire(&idelock);  //DOC:acquire-lock

  // Append b to idequeue.
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
    ;
  *pp = b;

  // Start disk if necessary.
  if(idequeue == b)
    idestart(b);

  // Wait for request to finish.
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
    sleep(b, &idelock);
  }


  release(&idelock);
}
```

The iderw() function in ide.c keeps the list of pending disk requests in a queue. It uses interrupts to find out when each request has finished. Although iderw() maintains a queue of requests, the simple IDE disk controller can only handle one operation at a time. The disk driver maintains the invariant that it has sent the buffer at the front of the queue to the disk hardware; the others are simply waiting their turn. iderw() adds the buffer b to the end of the queue. If the buffer is at the front of the queue, iderw() must send it to the disk hardware by calling idestart; otherwise the buffer will be started once the buffers ahead of it are taken care of. We can see that polling is not an efficient use of the CPU. Instead, iderw sleeps, waiting for the interrupt handler to record in the buffer's flags that the operation is done. While this process is sleeping, xv6 will schedule other processes to keep the CPU busy.

```
void
ideintr(void)
{
  struct buf *b;

  // First queued buffer is the active request.
  acquire(&idelock);

  if((b = idequeue) == 0){
    release(&idelock);
    return;
  }
  idequeue = b->qnext;

  // Read data if needed.
  if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
    insl(0x1f0, b->data, BSIZE/4);

  // Wake process waiting for this buf.
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);

  // Start disk on next buf in queue.
  if(idequeue != 0)
    idestart(idequeue);

  release(&idelock);
}
```

At some point in time, the disk will finish its operation. When this happens, an interrupt will be triggered. To handle this, the ideintr() function is called. Ideintr consults the first buffer in the queue to find out the operation that was occurring. If the buffer was being read and the disk controller has data waiting, ideintr reads the data into the buffer with insl. Now the buffer is ready: ideintr sets B_VALID, clears B_DIRTY, and wakes up any process sleeping on the buffer. Finally, ideintr must pass the next waiting buffer to the disk.

Appendix

Some programs were written for debugging purposes. These are test_open.c and test_read.c files in the xv6-public directory.