P3B: ANALYZING XV6 LOCKS AND CONDITIONS

Nishit Saraf
Ayan Deep Hazra

LOCKS

FS.C (SLEEPLOCK)

INITIALIZATION:iinit()

```
for(i = 0; i < NINODE; i++) {
  initsleeplock(&icache.inode[i].lock, "inode")
}
```

The SleepLock is initialized in the iinit method. It is initialized for every Inode, where NINode is the total number of Inodes. So that all of the unnamed files have an inode therefore have a sleeplock assigned to it. And allows the lock to be acquired by the first inode which tries to enter a critical section.

ACQUIRING THE LOCK

INSTANCE 1: ilock()

```
acquiresleep(&ip->lock);

if(ip->valid == 0){
  bp = bread(ip->dev, IBLOCK(ip->inum, sb));
  dip = (struct dinode*)bp->data + ip->inum%IPB;
  ip->type = dip->type;
  ip->major = dip->major;
  ip->minor = dip->minor;
  ip->nlink = dip->nlink;
  ip->size = dip->size;
  memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
  brelse(bp);
  ip->valid = 1;
  if(ip->type == 0)
    panic("ilock: no type");
}
```

The sleeplock is acquired in the ilock method, this is done so that the only one inode is able to progress through the critical section.

The critical section checks whether the inode is valid or not, and if it is then it initializes the inode with the data from the disk. By doing a memmove it copies the address of the inode from the disk to the actual inode.

This method requires a lock to be acquired, otherwise 2 different nodes might get the same address and data from the disk. As a result this needs to be done atomically.

This is separate from the iget() method because only this is a critical section so by bifurcating this from iget() method we keep the critical sections small and maximize concurrency by allowing multiple threads to run iget() method concurrently. This lock is released in the iunlock method, which is explained below.

```
int consolewrite(struct inode *ip, char *buf, int n)
{
        int i;
        iunlock(ip);
        acquire(&cons.lock);
        for(i = 0; i < n; i++)
                consputc(buf[i] & 0xff);
        release(&cons.lock);
        ilock(ip);
        return n;
}
```

This method ilock is called by consoleread() and consolewrite() in the console.c, exec() in the exec.c, filestat() and filewrite() in the file.c, and sys_link(), sys_unlink(), create(), sys_open() and sys_chdir() in the sysfile.c.

Since, this function is being called by so many other functions in other files, whenever they together call the ilock() method, all of the code would run concurrently. Since ilock() only allows one thread/inode to gain the lock, all of the other threads and processes would be put to sleep, as a result we guarantee that only one inode would be able to gain the lock. This would also work when two or more threads would call the methods which would then call ilock(), for example two different threads calling consolewrite() which would then call ilock(), this would still work since only thread would be able to gain the lock, and all other threads would be put to sleep and be awoken when the iunlock() is called.

The Critical Section is From Line 296-311

INSTANCE 2:iput()

```
void
iput(struct inode *ip)
{
  acquiresleep(&ip->lock);
  if(ip->valid && ip->nlink == 0){
    acquire(&icache.lock);
    int r = ip->ref;
    release(&icache.lock);
    if(r == 1){
      // inode has no links and no other references: truncate and free.
      itrunc(ip);
      ip->type = 0;
      iupdate(ip);
      ip->valid = 0;
    }
  }
  releasesleep(&ip->lock);
```

The sleeplock is acquired in the iput method, this is done so that only one inode is able to progress through the critical section.

The critical section checks whether the inode is valid and if it has no links or not. If it doesn't have any links, it tries to acquire a spinlock of the cache to recycle the cache reference. If it happens to enter the if block, it means that the inode has no links and no other references.
So we can free the inode and its details on the disk as well. Then the sleeplock is released so that it can wake up another thread which is currently asleep and can access these methods atomically.

This method needs to be in a lock, otherwise two different threads (or in this case two different inodes) which are linked to each other enter this section together then we wouldn't be able to clear the contents from the disk and free the inode, since it becomes a race condition.

We would free both of the inodes, however while checking for the number of links both would result in one (if there was a timer interrupt and then a context switch before the just after the if condition), as a result, even though the intention was to clear the inode data from the disk, it wouldn't occur.

As a result, this is a critical section and thus needs a lock around it. By using a sleeplock rather than spinlock, we guarantee less time wasted as precious CPU cycles are not wasted spinning not doing anything but instead puts the inodes to sleep, which are woken up by the release method.

This method iput() is called by unlockput() in fs.c, fileclose() in file.c, sys_link() in sysfile.c. This function is called by unlockput() which is inside the same file as iput(), this denotes that the processes even from the same file would be treated the same as the processes from the other files, and would be put to sleep if they are unable to acquire the sleep lock.

Since in the beginning of the function we acquire the sleeplock just like the ilock() function we guarantee that only one thread in a single file would be able to acquire the lock at any given single point in time, and all other processes would be put to sleep, which would be woken up later in the function where the critical section is over.

```
  if(ff.type == FD_PIPE)
    pipeclose(ff.pipe, ff.writable);
  else if(ff.type == FD_INODE){
    begin_op();
    iput(ff.ip);
    end_op();
  }
}
```

The Critical Section is From Line 334-347

RELEASING THE LOCK:iunlock()

```
void
iunlock(struct inode *ip)
{
  if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)
    panic("iunlock");

  releasesleep(&ip->lock);
}
```

The sleeplock is released in the iunlock method which is called by an inode after accessing the ilock method, in which it acquires the lock.

This results in one of the sleeping threads/inodes to be woken up and releases the lock to be held by one of the sleeping inodes.

If there are no sleeping inodes, then it doesn't wake any inodes, just releases the lock to be held by another inode.

After the ilock method but before the iunlock method an inode might call many methods which need to be done atomically, like adding an inode to the disk or removing an inode from disk.

This method also checks the holdingsleep for the inode, which tells if the inode currently holds the lock or not, if not and it tries to call the iunlock method, something has gone wrong and we raise an exception by calling panic. This is equivalent to an exception handler on intel x-86.

This method is called by unlockput() in the same file(fs.c), consoleread() and consolewrite() in console.c, exec() in exec.c, filestat(), fileread() and filewrite() in file.c, and sys_link() in sysfile.c.

Since this function is called by all these functions, any of them could have concurrently as a result inside the iunlock function, there is a check to see if the ip is valid or not otherwise returns an exception (As described above), and the releasesleep() inside

sleeplock.c has a spinlock so that only one of these threads can run at a singular point.

```
while(i < n){
  int n1 = n - i;
  if(n1 > max)
    n1 = max;

  begin_op();
  ilock(f->ip);
  if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
    f->off += r;
  iunlock(f->ip);
  end_op();

  if(r < 0)
    break;
  if(r != n1)
    panic("short filewrite");
  i += r;
}
```

The Critical Section is on line 321.

PROC.C (SPINLOCK)

INITIALIZATION: pinit()

```
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
}
```

The spinlock is initialized with the initlock function, with and address to the ptable struct's lock element and the name "ptable" passed in as parameter.

INSTANCE 1: allocproc()

In allocproc, the lock is acquired over a very small critical section, one that goes through the ptable structure and looks for an empty spot and assigns the state and pid to the process. If there are no available spots, then it releases the lock without doing anything and returns. This return is required else, you would be releasing a lock twice, and that would lead to a seg fault.

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  release(&ptable.lock);

  // Allocate kernel stack.
```

The Critical Section is between line 79 and 92.

Overall, a lock is required because if multiple threads entered the for loop, they would all concurrently go over the same elements in the loop and therefore cause logical errors. Therefore, when the lock is held, and a different thread tries to enter the critical section, it keeps spinning and waiting for the first thread to complete execution.

INSTANCE 2: userinit()

userinit() indirectly uses a lock by calling allocproc, when initializing the first user process. And in the concluding lines, it calls a lock to change the state of the process in question to RUNNING. This makes intuitive sense as you would not want multiple concurrent assignments to an important parameter like state, this could lead to very bad consequences starting with indeterminism.

INSTANCE 3: fork()

Like userinit(), fork() acquires and releases a lock around setting the state assignment, presumably for the same reason as before. It also calls allocproc() and therefore indirectly acquires and releases the same lock a second time.

```
acquire(&ptable.lock);

np->state = RUNNABLE;

release(&ptable.lock);
```

The Critical Section is between line 215 and 219.

INSTANCE 4: exit()

exit() is interesting as it only acquires a lock and calls the wakeup1() method or sched() function depending on whether the parent of p is the current process versus if none of it is true. If none, then the state is set to zombie and sched() is called. We do not release the lock in this function, but do it in either of the two functions that is called from exit.

INSTANCE 5: wait()

wait() calls acquire before executing the for loop that scans the table for exit-ted children. If the process is in a zombie state then the lock is released after freeing up and defaulting the spot's variables and returning the pid of the now defunct process. After the loop if we determine that a process is childless or killed, we release the lock and return an error code. If none of the above is true then the process is dormant and we make the current process go to sleep and also pass the lock as a parameter. In sleep, the lock is conditionally released.

```
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;
  }
```

The Critical Section is between line 279 and 305.

INSTANCE 6: scheduler()

Outside a main for loop, the critical section is located, which contains a main for loop, one that does the bulk of the scheduler's job.

The lock is acquired and releases a rather large critical section involving a for loop that iterates through the table and examines every process. If a process is not runnable, then we ignore it. If it is, then we choose it to run (schedule it) and wait till a timer interrupt has occurred or the process completes or surrenders. We do this for all processes and exit the loop and release this lock.

```
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
      continue;

    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;

    swtch(&(c->scheduler), p->context);
    switchkvm();

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0;
  }
  release(&ptable.lock);
```

The Critical Section is between line 334 and 352.

As the comments note, handling the lock is entirely up to the process and not the kernel. After having loaded the process switchuvm(p), the process is marked running (p->state = RUNNING;) and the processor switches to execute it (swtch(&(c->scheduler), p->context);). When the process comes back to scheduler the kernel loads its memory by invoking switchkvm();

INSTANCE 7: sched()

This function does not do any locking but checks if the lock is held elsewhere in the code and if not, it invokes the panic() function.

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

Checks holding() function on line 371.

As described in the function definition, this function enters the scheduler function and tries to gauge the status of the scheduler at the instance when it is called. The use of the holding function occurs at the

first of the if else clauses. If the lock is not held then the panic function is called for the “ptable.lock” argument. The remaining functions also call panic() in some capacity depending on the state of the scheduler.

The remaining instances are not as interesting and have only 1 or 2 lines in the Critical Section.

INSTANCE 8: yield()

In this function the lock is around a state assignment and invocation of the sched() function. We can understand this as the sched() function should only be invoked by one thread, and by extension, one process, at a time.

INSTANCE 9: forkret()

This function just releases the ptable lock that must have been held elsewhere.

forkret() is called in the concluding lines of allocproc().

INSTANCE 10: sleep()

There are 2 instances of the function checking if the lock passed in as a param is not the same instance as the current lock, and if it is true, then the current ptable lock is released and the lock parameter is acquired.

INSTANCE 11: wakeup()

This function acquires a lock, calls wakeup1 and releases the lock.

INSTANCE 12: kill()

This is intuitive from the previous instances where a lock was needed to initialize a process; a lock is also needed to kill a process, to prevent multiple threads from trying to kill the same process and putting the computer in an indeterminate state by shooting a segmentation fault.

SLEEP/WAKEUP

ANALYSIS OF SLEEP AND WAKEUP FUNCTIONS

The sleep() function releases the lock that a process is holding up, and puts it in the sleeping state by assigning the state variable to SLEEPING. The sched() function is then called, and this only returns when the process is scheduled to run again, else it sits in an infinite loop within the interior of the sched() function.

On returning, which we know it has to, what alternative does it have, the state variables are tidied up and the lock that was formerly released is held again by using the acquire() function of spinlock.c. It does this by first comparing if the lock being held is by that process or not. If it is held by some other process, then that lock is released and the same lock is acquired by our process in question.

The wakeup() is written in an interesting manner by the original authors of xv6. The only thing it seemingly does is hold the lock and calls the similarly named wakeup1() function.

The wakeup1() function iterates through all the processes in the ptable and compares the chan variable of each and every one to the function’s chan parameter. If it matches, then the process is set to RUNNING, or for our purposes “woken up”. A lock seems like the logical choice to wrap this bit of code around in. One would not want multiple threads changing the state of any process, concurrently.

LOG.C

The sleep() function is called twice in this file and both are in the begin_op() function, albeit in different if case statements.

The begin_op() function starts off with first acquiring a lock and then embedding a while loop within the critical section of that lock. The while loop in question is an infinite while loop that only terminates if the interior of the loop calls a break command.

```
void
begin_op(void)
{
  acquire(&log.lock);
  while(1){
    if(log.committing){
      sleep(&log, &log.lock);
    } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
      // this op might exhaust log space; wait for commit.
      sleep(&log, &log.lock);
    } else {
      log.outstanding += 1;
      release(&log.lock);
      break;
    }
  }
}
```

The Critical Section is between line 334 and 352.

Inside the while, there is a if-else if branch and it first checks if the parameter committing of log is 1 or not. If it is, then the sleep function is called with the address of the log struct and the address of the log struct’s lock element, passed in as parameters.

As we know from our analysis of these two functions, we can understand that the process is first set to SLEEPING if it is executing the commit() function (represented by committing == 1).

Next, the function checks if the spaces in the log space are exhausted with the if condition. If it is exhausted, then it is again made to sleep by calling the sleep() function.

The else code increases the outstanding parameter's value by 1 and releases the lock that was being held at the start of the function, and calls a break in the while loop.

Taking a step back, we see that the interior of the while is set up so that there are 3 alternatives, 2 of which result in a sleep() and a third which releases the lock and breaks from the loop entirely. So, effectively we continue sleeping till neither of the conditions are met, in which case we leave the function after releasing the lock.

In a pleasing symmetry, the two times that wakeup() is called is within the end_op() function.

```
// called at the end of each FS system call.
// commits if this was the last outstanding operation.
void
end_op(void)
{
  int do_commit = 0;

  acquire(&log.lock);
  log.outstanding -= 1;
  if(log.committing)
    panic("log.committing");
  if(log.outstanding == 0){
    do_commit = 1;
    log.committing = 1;
  } else {
    // begin_op() may be waiting for log space,
    // and decrementing log.outstanding has decreased
    // the amount of reserved space.
    wakeup(&log);
  }
  release(&log.lock);

  if(do_commit){
    // call commit w/o holding locks, since not allowed
    // to sleep with locks.
    commit();
    acquire(&log.lock);
    log.committing = 0;
    wakeup(&log);
    release(&log.lock);
  }
}
```

The Critical Section is between line 334 and 352.

This function is called at the end of every File System call and commits if the process was the last outstanding one.

The first time wakeup() is called in the function, it is within the spinlock's critical section. We first check if a process is currently being committed, and if it does, we panic(), as we are within a lock and should be the only ones who have this power. There must be some bug in the code if someone outside the critical section can edit our variables. Next we check if the outstanding variable is 0, if it is, then we change committing to 1 (or basically start the commit process) and do_commit also to 1. If none of the above is true, then we call the wakeup() function and wake this thread up.

This is interesting, as while we are still inside a lock, on calling wakeup, we now hold another lock within that lock from wakeup(). Within our level 2 lock, we call wakeup1() with the parameter chan, which was passed into wakeup(). wakeup1() then iterates through the process table and checks each one to see if the chan parameter matches or not. If it does then that process' state is set to RUNNING. If it never matches, then we finish the iteration and go back to the wakeup() function, where we release the lock (level 2 lock) and return to the calling function (end_op).

After calling wakeup, we exit the if-else if cases, and release the main lock.

Before exiting the end_op function, we check if do_commit was set to 1 from the previous if-else if block. If it was, then we enter a new if block, which first calls the commit() function. It then acquires the log's lock and calls the wakeup() function from within the critical section, with the address of the log struct as the parameter.

Within the wakeup, we do a similar thing as before, we compare the chan parameter and if it matches, we set that process in question to running. Like this, we have now "woken up" a process that was formerly sleeping.

PIPE.C

The sleep() function is called in two instances within pipe.c, and we can bifurcate them into two instances. The wakeup() function is called in three instances within pipe.c, and although two of these instances include the sleep() function, one of these instances only includes the wakeup() function and not the sleep() function, so we would bifurcate this file into a total of three instances.

INSTANCE 1: 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){  //DOC: pipewr
      if(p->readopen == 0 || myproc()->killed){
        release(&p->lock);
        return -1;
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->nread);  //DOC: pipewrite-wakeup1
  release(&p->lock);
  return n;
}
```

The function begins by trying to acquire a spinlock which ensures that everything that follows it is a critical section, therefore needs to be done atomically rather than multi-threaded.

Within the critical section are two loops, a for loop and a while loop. The for loop runs as the outer loop and is the total number of addresses that need to be written in the other file. The while loop checks whether each particular address has enough space to read an address and the PIPESIZE for every written entry.

After that it tries to open the particular pipe file, if it results in an error the function releases the spinlock and exits the function. Otherwise it calls the wakeup method, on the read of the pipe. This results in all the processes on the read to be woken up, and if it's already awake nothing happens. Then it calls the sleep() method on the write of the pipe, thus this results in all of the processes on the write to be asleep, until woken up by the piperead() method. This makes sure that every pipewrite() method wakes up a read process and not a write process. This is similar to the producer, consumer problem we encountered in class where a producer would wake up a consumer and vice-versa.

After the end of the while loop, the data is finally written and the particular address is stored within the pipe data block. However this array index uses a modulus PIPESIZE which makes sure that array indices don't go out of bounds. Since we reset the array address whenever p->write is more than PIPESIZE, the array is never out of bounds.

After this the wakeup method is called once more, which wakes up the nth process since that isn't included in the for loop, and here again the spinlock is released since this is the end of the function and if every file is valid (and can be read from), the lock is finally released properly.

Pipewrite is called by the filewrite() in file.c. As a result whenever multiple threads within filewrite() call pipewrite together they would first need to acquire a lock which would make sure that only one of the threads is able to get the lock at any given point of time. This makes sure that the critical section is done atomically and no two threads enter the section concurrently.

```
int
filewrite(struct file *f, char *addr, int n)
{
  int r;

  if(f->writable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return pipewrite(f->pipe, addr, n);
```

The critical section in total in pipewrite is from line 83 till line 96.

INSTANCE 2: piperead()

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

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

This function is quite similar to pipewrite() but instead of writing the address we rather read into address and still the spinlock is required to do this portion atomically since this section is a critical section.

However the main difference between this method and pipewrite() is that in this method we first do the while loop then the for loop rather than the while loop being inside of the for loop. This helps in making it faster, since we check every process and check if it's KILLED status or not, and if any one process within the pipe is KILLED, then we release the spinlock and return from the function. Otherwise the sleep function is called on all of the reading processes, which is woken in the pipewrite method as seen above.

The for loop has an additional check this time which checks whether the total read blocks are equal to the

total written blocks, and if it is then to exit the for loop. This is done so that we don't accidentally go out of bounds where we read an entry which doesn't exist. Then the address is filled with the correct data entry within the process by using the same looping back mechanism which was used in pipewrite() to not reach out of bounds.

After the for loop the wakeup method is called writing processes which as described above awakens a sleeping writing process and releases the spinlock in case it hasn't been released before and exits the function. As we can see from these two functions pipewrite and piperead the producer and consumer problem can be seen on full display. By using these condition variables such as whether the file is valid or not, we make sure to wake the opposite thread as opposed to the one which was just called.

Piperead is called by fileread() in file.c. As a result, whenever multiple threads call the piperead() call, only one single thread is actually able to attain the lock. Just, as before, helps with running the critical section as atomically as possible since no context switches would affect this portion of the code.

```
int
fileread(struct file *f, char *addr, int n)
{
  int r;

  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
```

The critical section in total in piperead is from Line 105 till Line 119.

INSTANCE 3: pipeclose()

```
void
pipeclose(struct pipe *p, int writable)
{
  acquire(&p->lock);
  if(writable){
    p->writeopen = 0;
    wakeup(&p->nread);
  } else {
    p->readopen = 0;
    wakeup(&p->nwrite);
  }
  if(p->readopen == 0 && p->writeopen == 0){
    release(&p->lock);
    kfree((char*)p);
  } else
    release(&p->lock);
}
```

The function begins again acquiring a lock which again tells us that the code following this is a critical section and needs to be done atomically. However, between these function calls there are more functions within pipe.c which allow for multi-threads and therefore don't require locking and other methods such as CV's and Semaphores.

Then the function checks if writable is valid or not, if it is then it sets the writeopen of the pipe process as 0 and then wakes up a reading process otherwise it sets the readopen of the function as 0 and wakes up a writing process which then again acquires the lock after being woken up. This is similar to the above functions which would wake up the opposite thread, rather the same process which calls the function.

After that the function checks whether the readopen and writeopen for the pipe process are 0 or not, if it is then releases the spinlock and frees the pipe from the heap. This can only happen if the pipeclose method is at least called twice once where the writable is valid and at least one more time where writable is not valid, since this is the only place which sets p->writeopen and p->readopen as 0 in this whole file. This tells us that after reading everything and calling pipeclose, it would set readopen as 0 and the same with writeopen as 0 and after that we would free the pipe struct from the memory.

Pipeclose is called by the fileclose() inside of file.c. As to maintain the atomicity of the critical section, to enter pipeclose, a thread would require to acquire a lock whereas the other threads would have to spin. This ensures that at any given time only thread is running the critical section and no other thread/process can interrupt it.

```
  if(ff.type == FD_PIPE)
    pipeclose(ff.pipe, ff.writable);
  else if(ff.type == FD_INODE){
    begin_op();
    iput(ff.ip);
    end_op();
  }
}
```

The critical section in total in pipeclose is from Line 62 till Line 74.