CS537 Analyzing xv6 Locks and Conditions

Zhuocheng Sun & Gefei Shen

Part 1

Xv6 implements a spinlock for synchronization and mutual exclusion of access for critical sections. In the spinlock struct, when the variable locked is 1, it means that the lock has been occupied; when the variable locked is 0, it means that the lock is not occupied.

Before using the lock, the lock has to be initialized by initlock(). When a program wants to acquire a lock, it will call acquire() function, which is basically implemented by an atomic exchange statement, to occupy a lock. When a program wants to release a lock, it will call release() function to release the lock. All three functions initlock(), acquire(), release() are implemented in spinlock.c in xv6.

Spinlock in Pipe.c and critical section analysis:

The first lock I want to analyse is the spinlock used in pipe.c. A pipe is a small kernel buffer exposed to process a pair of file descriptors, one for reading and one for writing. Writing data to one end of the pipe can make data readable from the other end of the pipe[1]. There are three functions in pipe.c that use spinlock to guard their critical sections.

```
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);
  }
}
```

As we can see in the above function, it acquires the spinlock at the very beginning. When both file descriptors of read end and write end are not open anymore, it will release the lock and free the memory. When either file descriptors of read end or write end are still open, it will release the lock but not free the memory. Therefore, in this function, the critical section is line 5-line 11(consider void as line 1). In this critical section, the pipe struct will check if it is still writable, if it is still writable, it will set writeopen, the file descriptor of the write end, to 0, and wake up corresponding process (details will be discussed later in sleeplock section); if it is not, it will set readopen, the file descriptor of read end, to 0 and wake up corresponding process (details will be discussed later in sleeplock section).

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

In this function, after the initialization for integer i, it acquires the spinlock. When the pipe is full of written data, which means no more space to write, it will check if the file descriptor of the read end is still open and the current process still exists, if not, it will release the lock and return -1. Another situation is that it will write all n bytes into the address and release the lock and return n. Therefore, in this function, the critical section is the for loop and the second wakeup function call. In the critical section, it will iteratively write data to the pipe buffer from the write end until the for loop ends or the pipe is full (sleep and wakeup details will be discussed later).

[1] p.10. Cox, Russ, Frans Kaashoek, and Robert Morris. 2012. “xv6 Book.” xv6 Book.

```
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 piperead function also uses spinlock. After the initialization of integer i, it acquires the lock. When there is no data to read (p->nread == p->nwrite) and the file descriptor of the write end has closed, it will enter the while loop. If the current process has been killed, it will release the lock and return -1. Another scenario to release the lock is that the process reads all the data available in the pipe buffer and releases the lock. Therefore, in this function, the critical section includes the while loop, for loop and the wakeup call. In this critical section, it will iteratively read data from the pipe buffer from the read end until there is no data to read (sleep and wakeup details will be discussed later).

Understand the lock behavior in pipe.c dynamically

To understand how often the lock is held and released when a process is running, I decided to add print statements after acquiring a lock and before releasing a lock. Additionally, I create a global integer variable called lock_num (initialize to 1 for each process) to record how many acquisitions have happened in the running process. Take the modified pipeclose function as an example:

```
void
pipeclose(struct pipe *p, int writable)
{
  acquire(&p->lock);
  cprintf("lock %d acquired in pipeclose. ", lock_num);
  if(writable){
    p->writeopen = 0;
    wakeup(&p->nread);
  } else {
    p->readopen = 0;
    wakeup(&p->nwrite);
  }
  if(p->readopen == 0 && p->writeopen == 0){
    cprintf("lock %d released in pipeclose\n", lock_num)
;
    lock_num++;
    release(&p->lock);
    kfree((char*)p);
  } else {
    cprintf("lock %d released in pipeclose\n", lock_num)
;
    lock_num++;
    release(&p->lock);
  }
}
```

Then, I ran some commands/test cases that use pipe to analyse the lock behavior.

The first one is : *echo loveCS537 | wc*

The second one is: *echo abcde | grep c*

```
$ echo loveCS537 | wc
spinlock initialized
lock 1 acquired in pipewrite. lock 1 released in pipewrite
lock 2 acquired in pipewrite. lock 2 released in pipewrite
lock 3 acquired in pipewrite. lock 3 released in pipewrite
lock 4 acquired in pipewrite. lock 4 released in pipewrite
lock 5 acquired in pipewrite. lock 5 released in pipewrite
lock 6 acquired in pipewrite. lock 6 released in pipewrite
lock 7 acquired in pipewrite. lock 7 released in pipewrite
lock 8 acquired in pipewrite. lock 8 released in pipewrite
lock 9 acquired in pipewrite. lock 9 released in pipewrite
lock 10 acquired in pipewrite. lock 10 released in pipewrite
lock 11 acquired in pipeclose. lock 11 released in pipeclose
lock 12 acquired in piperead. lock 12 released in piperead
lock 13 acquired in piperead. lock 13 released in piperead
1 1 10
lock 14 acquired in pipeclose. lock 14 released in pipeclose
```

In this example, we can see it first acquires the lock and releases the lock in pipewrite 10 times. So during the echo process, for every single pipewrite function call, it writes one character to the pipe buffer each time. After writing all characters in “loveCS537” and the “\n” added by the echo program, this process called pipeclose function. It acquire the lock in pipeclose and close the file descriptor of the write end, and then release the lock.

For the wc process, it needs to read the data in, so it is on the read end of the pipe buffer. As we can see in the above graph, it acquires the lock in piperead and reads the data in and releases the lock. After the data is available for the program wc, it counts how many lines, how many words and how many characters in the data and prints the output. Then finally, closing the file will call pipeclose again. It requires the lock, close the file descriptor of the read end and release the lock.

```
$ echo abcde | grep c
spinlock initialized
lock 1 acquired in pipewrite. lock 1 released in pipewrite
lock 2 acquired in pipewrite. lock 2 released in pipewrite
lock 3 acquired in pipewrite. lock 3 released in pipewrite
lock 4 acquired in pipewrite. lock 4 released in pipewrite
lock 5 acquired in pipewrite. lock 5 released in pipewrite
lock 6 acquired in pipewrite. lock 6 released in pipewrite
lock 7 acquired in pipeclose. lock 7 released in pipeclose
lock 8 acquired in piperead. lock 8 released in piperead
abcde
lock 9 acquired in piperead. lock 9 released in piperead
lock 10 acquired in pipeclose. lock 10 released in pipeclose
```

In this example, we can see it first acquires the lock and releases the lock in pipewrite 10 times. So during the echo process, for every single pipewrite function call, it writes one character to the pipe buffer each time. After writing all characters in “abcde” and the “\n” added by the echo program, this process called pipeclose function. It acquire the lock in pipeclose and close the file descriptor of the write end, and then release the lock.

For the grep process, it needs to read the data in, so it is on the read end of the pipe buffer. As we can see in the above graph, it acquires the lock in piperead and reads the data in and releases the lock. After the data is available for the program grep, it finds where ‘c’ is in the data and prints the output. Then finally, closing the file will call pipeclose again. It requires the lock, close the file descriptor of the read end and release the lock.

Why do we need the spinlock in pipewrite() and piperead()?

pipewrite() function in pipe.c is always called by filewrite() function in file.c, and filewrite() is always called by the system call sys_write() in sysfile.c. So the call graph for pipewrite is:

pipewrite() <- filewrite() <- sys_write().

When a pipe write operation task is assigned to multiple threads, there could be multiple calls for sys_write in different threads, and these sys_write() calls would lead to multiple filewrite() calls, which eventually, would lead to multiple calls of pipewrite() and a contention for it. To ensure the data is written to the pipebuffe correctly, we need the spinlock before writing the data in pipewrite() and release the lock after writing the data.

piperead() function in pipe.c is always called by fileread() function in file.c, and fileread() is always called by the system call sys_read() in sysfile.c. So the call graph for piperead is:

piperead() <- fileread() <- sys_read().

As we know, piperead() operation is not a simple read operation that does not change anything. piperead() will consume the data on the read end of the pipe buffer, so it is also important to mutually exclude multiple piperead() calls at the same time.

Therefore, when a pipe read operation task is assigned to multiple threads, there could be multiple calls for sys_read in different threads, and these sys_read() calls would lead to multiple fileread() calls, which eventually, would lead to multiple calls of piperead() and a contention for it. To ensure the data is conusmed from the read end of the pipebuffe correctly, we need the spinlock before reading the data in piperead() and release the lock after reading the data.

How do Spinlocks in pipe.c affect the overall performance?

In some scenarios, all the data in the pipe buffer has been consumed by the reader from the read end, so at this time, pipe’s reader will wait for new data to be written in to consume. Since both piperead() and pipewrite() acquire the same spinlock, while the reader is waiting for new data to be written, it must release the lock such that pipewrite() can acquire the lock and write data in. If we just put the reader to spin while waiting for the event that new data has been written, it will waste many CPU cycles.

Rather than make the waiting process waste CPU by repeatedly checking whether the desired event has happened, xv6 allows a process to give up the CPU and sleep waiting for an event, and allows another process to wake the first process up[2]. This is also why we need wakeup and sleep calls.

Part 2 Wakeup & Sleep in details

How wakeup(), wakeup1(), and sleep() are implemented in xv6?

The implementation of wakeup, wakeup1 and sleep can be found in the proc.c. Screenshots of these code are shown below:

[2] P.50, Cox, Russ, Frans Kaashoek, and Robert Morris. 2012. “xv6 Book.” xv6 Book.

```
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

  // Tidy up.
  p->chan = 0;

  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```

The sleep() function does its sanity check for no process and no lock conditions. Then it acquires the ptable lock to ensure no other thread can modify the ptable. It assigns chan to the process as instructed by the parameter, then it sets the state to sleep.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

The chan is defined in a struct in proc.h, which is shown above. The pointer chan refers to the channel, programs that are put into sleep can be put into the same channel together for batch coordination on wakeup and wakeup1 calls[3]. For example, thread T1 and T2 are assigned with channel 12, and thread T3 is assigned with channel 10. When calling wakeup(12), both T1 and T2 will get woken up, while T3 remains sleeping.

After changing the ptable, it calls sched(), which calls context switch to save the current state and restore the previous state. Then the process enters the sleep stage. When the contact switch is switched again to this process, this process's chan is set to 0, which indicates that it's not sleeping anymore. Thus, the sleep() reacquires the original lock and continues its execution[4].

```
//PAGEBREAK!
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
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;
}

// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

The wakeup call is divided into two parts, wakeup() that acquires the ptable lock and the wakeup1() that actually wakes the process. The implementation of wakeup is relatively easy, it acquires the ptable lock, and calls wakeup1().

Wakeup1() loop through all the processes in the ptable list to find a matching process. If a process is sleeping and its chan is equal to the parameter, its state will be set to RUNNABLE. Then the wakeup1() returns and wakeup() releases ptable lock.

Sleep and wake in pipe.c

As introduced above, pipe is an OS object that handles processing of file descriptors, it can be opened in multiple processes at a time. The pipe not only needs locks to handle multiple requests at the same time, but also needs to make sure it does not run out of buffer space. There are three functions in pipe.c that use sleep and wake to dynamically handle read and write requests.

[3] P.55, Cox, Russ, Frans Kaashoek, and Robert Morris. 2012. "xv6 Book." xv6 Book.

[4] P.57-58, Cox, Russ, Frans Kaashoek, and Robert Morris. 2012. "xv6 Book." xv6 Book.

```
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: pipewrite-full
      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;
}
```

As we can see in the above function, after entering the for loop, it checks if the pipe is full. When the pipe is full, it means the program does not have sufficient space to handle the write request. Thus, it wakes the piperead() process (if it has) to free up the space and put itself to sleep else it would just sleep and wait for another method to wake it up. If the pipe is not full, it would write into the pipe. In addition to the sleep and wake call in the whole loop, there is a wake at the end of the function to wake up any sleeping piperead() function, which could possibly happen in the following function.

```
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;
}
```

As we can see above, the piperead() function will check if the pipe is empty with the while loop. When the pipe is empty, it puts itself to sleep and waits for someone to wake it up. If the pipe is not empty, then it processes data and empties the pipe. The wakeup() call at the end of this function wakes any potentially sleeping pipewrite().

```
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);
  }
}
```

In this function, after acquiring the spinlock, it checks whether it is writable. If it is writable, it sets writeopen to 0, then wakes up sleeping piperead() thread. Else, it sets readopen to 0 and wakes up any sleeping pipewrite() thread it has.

Understand Wakeup and Sleep Behavior in pipe.c Dynamically

To better understand the mechanism of sleep and wake in the file pipe.c, System output with a detailed log of sleep and wake activities is a clear way to illustrate activities happening within the CPU. Console output with dedicated description at each code section is added. In order to increase the occurrence of sleep(), buffer size is also reduced to a much smaller size. In addition to that, to show all the external calls of pipe.c functions, I've added console outputs whenever any function of pipe.c is called. By entering command

*grep pipeclose *.c*

I found file.c is the only one calling the pipeclose() function.

```
[[gefei@rockhopper-05] (15)$ grep pipeclose *.c
file.c:    pipeclose(ff.pipe, ff.writable);
pipe.c:pipeclose(struct pipe *p, int writable)
```

Similar grep commands were also conducted on piperead() and pipewrite(); they've also indicated that file.c is the only file that is directly calling the pipe functions. Hence, some console output was added to file.c.

All the modifications are shown below.

In file.c

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

  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE) {
    cprintf("Calling READ\n");
    return piperead(f->pipe, addr, n);
  }
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("fileread");
}

//PAGEBREAK!
// Write to file f.
int
filewrite(struct file *f, char *addr, int n)
{
  int r;

  if(f->writable == 0)
    return -1;
  if(f->type == FD_PIPE) {
    cprintf("Calling WRITE\n");
    return pipewrite(f->pipe, addr, n);
  }
  if(f->type == FD_INODE){
    // write a few blocks at a time to avoid exceeding
    // the maximum log transaction size, including
    // i-node, indirect block, allocation blocks,
    // and 2 blocks of slop for non-aligned writes.
    // this really belongs lower down, since writei()
    // might be writing a device like the console.
    int max = ((MAXOPBLOCKS-1-1-2) / 2) * 512;
    int i = 0;
    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;
    }
    return i == n ? n : -1;
  }
  panic("filewrite");
}
```

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

  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);

  if(ff.type == FD_PIPE) {
    cprintf("Calling CLOSE\n");
    pipeclose(ff.pipe, ff.writable);
  }
  else if(ff.type == FD_INODE){
    begin_op();
    iput(ff.ip);
    end_op();
  }
}
```

In pipe.c

```
#define PIPESIZE 3
```

```
void
pipeclose(struct pipe *p, int writable)
{
  acquire(&p->lock);
  if(writable){
    p->writeopen = 0;
    cprintf("Wake up READ thread\n");
    wakeup(&p->nread);
  } else {
    p->readopen = 0;
    cprintf("Wake up WRITE thread\n");
    wakeup(&p->nwrite);
  }
  if(p->readopen == 0 && p->writeopen == 0){
    release(&p->lock);
    kfree((char*)p);
  } else
    release(&p->lock);
}

//PAGEBREAK: 40
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: pipewrite-full
      if(p->readopen == 0 || myproc()->killed){
        release(&p->lock);
        return -1;
      }
      cprintf("Pipe full, wake up READ thread\n");
      wakeup(&p->nread);
      cprintf("Pipe full, put WRITE to sleep\n");
      sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  cprintf("Finish writing, wake up READ thread\n");
  wakeup(&p->nread);  //DOC: pipewrite-wakeup1
  release(&p->lock);
  return n;
}
```

```
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;
    }
    cprintf("Pipe empty, put READ to sleep\n");
    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];
  }
  cprintf("Finish reading, wake up WRITE thread\n");
  wakeup(&p->nwrite);  //DOC: piperead-wakeup
  release(&p->lock);
  return i;
}
```

Same prompt from Section 1 is used to analyze the wake and sleep behavior. The output of

echo CS537 | wc

is attached below:

```
$ echo CS537 | wc
Calling WRITE
Finish writing, wake up READ thread
Calling WRITE
Finish writing, wake up READ thread
Calling WRITE
Finish writing, wake up READ thread
Calling WRITE
Pipe full, wake up READ thread
Pipe full, put WRITE to sleep
Calling READ
Finish reading, wake up WRITE thread
Finish writing, wake up READ thread
Calling READ
Calling WRITE
Finish reading, wake up WRITE thread
Finish writing, wake up READ thread
Calling READ
Calling WRITE
Finish reading, wake up WRITE thread
Finish writing, wake up READ thread
Calling READ
Calling CLOSE
Finish reading, wake up WRITE thread
Wake up READ thread
Calling READ
Finish reading, wake up WRITE thread
1 1 6
Calling CLOSE
Wake up WRITE thread
```

The wc function counts the total character count and the word count and line count. As the above output indicates, pipe writes 6 characters (including the '\n') into the buffer one character each time by pipewrite(). When the fourth write was executed, the pipe was already full since I'd manually set the pipe buffer to 3. Hence, the pipewrite() is put to sleep and it attempts to wake up any sleeping piperead() thread. However, since there is no sleeping read(), it does not do anything. Then a call of piperead() reads the buffer, and attempts to wake up the sleeping pipewrite() thread. This time, since there was a sleeping pipewrite() thread, it was successfully woken up.

Then multiple pipewrite() and piperead() was called, followed by a pipeclose(). Since the last piperead() might get executed before the last pipewrite() finished its execution, it put itself to sleep. Hence, the pipeclose() function wakes the sleeping piperead() to finish reading all of the pipe buffer. Then the pipeclose() was called to close the reading by the system, attempting to wake up sleeping pipewrite(). Since no pipewrite was sleeping, hence, it does nothing and the program terminates.

Why do we need wakeup() and sleep() in pipe.c?

The call graphs of piperead() and pipewrite() are described in the above section. The pipeclose() follows a similar format, pipeclose() is called by the fileclose() from file.c, and fileclose() is always called by sysfile.c via system functions sys_close(), sys_open() and sys_pipe(). The call graph is shown below:

pipeclose() <- fileclose() <- sys_functions().

When multiple calls of piperead(), pipewrite() or pipeclose() occurs, each of them will do their own condition check. If the preconditions are not met, they will put themselves to sleep and wait for another process to wake it up when preconditions are met. Thus, sleep() and wakeup() will ensure that each thread will be executed whenever appropriate, without wasting excessive CPU power.

How do wakeup() and sleep() in pipe.c affect the overall performance?

The major advantages of wakeup() and sleep() compared to lock are its flexibility and relatively less consumption of CPU power while waiting for execution. It does not require a specific function to wake up a certain process. For example, both pipeclose() and piperead() can wake up a sleeping pipewrite() function. As long as the channel is assigned and managed properly, it is much more efficient and straightforward for coding and maintenance.

Bibliography

Cox, Russ, Frans Kaashoek, and Robert Morris. 2012. "xv6 Book." xv6 Book. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-828-operating-system-engineering-fall-2012/lecture-notes-and-readings/MIT6_828F12_xv6-book-rev7.pdf