Locks

We searched inside the xv6 project using the phrase "acquire(" and after carefully looking at the results, we realized that xv6 uses "spinlock" primarily. Although, there is also a "sleeplock" in xv6, it is just a wrapper around the actual spinlock. Hence, for this project, we are focusing only on spinlock, because we believe that it is easy to understand sleeplock once we have understood the underlying spinlock (in life, we should focus on understanding simple thing first, right?)

There are many instances of spinlock initialization throughout xv6 code, but for this project we are limiting our analysis to 2 instances.

Analysis #1: Locks usage in console.c API

There is total 4 instances of lock usages in console.c

1. cprintf
2. consoleintr
3. consoleread
4. consolewrite

We are aware about the existence of cprintf() from the earlier xv6 projects, and we are naturally curious about how multiple writes are synchronized across all processes.[Snippet – 1]

Routine: cprintf
Task: Print to the console. only understands %d, %x, %p, %s.

Step#1: Boundary Condition Check
console module has a variable "locking", which gets initialized to 1 during consoleinit() via main(). This variable determines if lock/unlock attempt is even allowed, essentially it shows if the system is in panic mode or normal working mode.

Step#2: Lock Acquiring
If locking=1 then there is an attempt of lock acquiring.

Step#3: Critical Section
Here cprintf() does the business logic of printing character to screen and serial port(if available, but we shall not go into details for the brevity).
It iterates through each character that was passed to it and determines how to interpret it.

Step#4: Lock Release
If locking=1 then lock is released.

```
locking = cons.locking;
if(locking)
  acquire(&cons.lock);

if (fmt == 0)
  panic("null fmt");

argp = (uint*)(void*)(&fmt + 1);
for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
  if(c != '%'){
    consputc(c);
    continue;
  }
  c = fmt[++i] & 0xff;
  if(c == 0)
    break;
  switch(c){
  case 'd':
    printint(*argp++, 10, 1);
    break;
  case 'x':
  case 'p':
    printint(*argp++, 16, 0);
    break;
  case 's':
    if((s = (char*)*argp++) == 0)
      s = "(null)";
    for(; *s; s++)
      consputc(*s);
    break;
  case '%':
    consputc('%');
    break;
  default:
    // Print unknown % sequence to draw attention.
    consputc('%');
    consputc(c);
    break;
  }
}

if(locking)
  release(&cons.lock);
```

Snippet 1 console.c - cprintf() routine

The routine consoleintr() follows the cprintf() style, minus additional check. There is a spinlock acquire, followed by critical section, followed by spinlock release.

The routines consoleread and consolewrite works on inode, and hence, have few extra steps.

1. release spinlock on inode,
2. acquire spinlock for business logic
3. business logic critical section
4. release spinlock acquired in step#2
5. acquire spinlock released in step#1

Analysis #2: Locks usage in kalloc.c API

Another module that we were interested was "kalloc", an xv6 variant of linux's calloc() API. There were a lot of instances of discussion in class saying "malloc is a thread-safe" so this was a perfect opportunity for us to analyze its working xv6.

It has 2 routines where locks are used.

1. kfree (remember free in c?)
2. kalloc (remember calloc in c?)

kalloc has a spinlock and a variable "use_lock" as an optimization switch. We will talk about it once we look at the phases of kalloc initialization.

Unlike most modules in xv6, kalloc initialization happens in 2 phases.

1. main() calls kinit1() while still using entrypgdir to place just the pages mapped by entrypgdir on free list.
2. main() calls kinit2() with the rest of the physical pages after installing a full page table that maps them on all cores.

Phase#1and #2 are initialized just once (why? main.c is called just once), and there is no need for synchronization at this point. Hence, routines kfree and kalloc checks if use_lock variable is set or not. If they are not set, then it just short-circuits the lock acquire/release and proceed as usual. However, once the system is completely started, we shall use locks always, and hence use_lock is set in kinit2().

API-1 kalloc()

```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

Snippet 2 kalloc

It takes a node from the linkedlist freelist, and gives back to caller. Hence, lock is used for taking node out of freelist. It is a relatively small critical section. (What is kmem.use_lock doing here? Remember we talked about the optimization? Yes, we cannot afford that optimization anymore)

Table 1 kalloc usages

No	File	Routine	Race cond	Description
1	Main.c	Startothers	No	Runs just once. Allocate stack.
2	Proc.c	Allocproc	Yes	Allocate kernel stack for a new process.
3	Vm.c	walkpgdir	Yes	Get the page, if not found, then create one.
4		Setupkvm	Yes	Setup kernel part of the process
5		Inituvm	Yes	Load the initcode into address 0 of pgdir
6		Allocuvm	Yes	Allocate page tables and physical memory to grow process
7		Copyuvm	Yes	Given a parent process's page table, create a copy. Used in fork() from proc.c

API-2 kfree()

```
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

Snippet 3 kfree

It takes a paseed node, reset the memory, and inserts back to the linkedlist freelist. Hence, lock is used for inserting node into the freelist. It is a relatively small critical section. (What is kmem.use_lock doing here? Remember we talked about the optimization? Yes, we cannot afford that optimization anymore)

Table 2 kfree usage

No	File	Routine	Race cond	Description
1	Proc.c	fork	Yes	Copied the parent process state to child process state. If there is a failure, then kfree() is used to free the memory.
2		wait	Yes	If there is any child in zombie state, it's memory is freed using kfree().
6	vm.c	Allocuvm	Yes	Grows process from old size to new size. If mapping after growing size fails, then calls kfree() to memory.
7		deallocuvm	Yes	Reverses the growing phase of the process size. Frees newly allocated memory using kfree()
		freevm	Yes	First frees every entry within page table, then frees entire pagetable for a process.
		copyuvm	Yes	If copying from parent to child process fails, uses kfree() to free child pages.

Sleep and Wakeup

```
426  void
427  sleep(void *chan, struct spinlock *lk)
428  {
429    struct proc *p = myproc();
430
431    if(p == 0)
432      panic("sleep");
433
434    if(lk == 0)
435      panic("sleep without lk");
436
437  >   /* Must acquire ptable.lock in order to-
443    if(lk != &ptable.lock){  //DOC: sleeplock0
444      // cprintf("proc.c::sleep:acquire lock: [proc:%s]\n", p->name);
445      acquire(&ptable.lock);  //DOC: sleeplock1
446      release(lk);
447    }
448
449    // Go to sleep.
450    p->chan = chan;
451    p->state = SLEEPING;
452
453    sched();
454
455    // Tidy up.
456    p->chan = 0;
457
458    // Reacquire original lock.
459    if(lk != &ptable.lock){  //DOC: sleeplock2
460      // cprintf("proc.c::sleep:release lock: [proc:%s]\n", p->name);
461      release(&ptable.lock);
462      acquire(lk);
463    }
464  }
```

Snippet 4 sleep

```
469  static void
470  wakeup1(void *chan)
471  {
472    struct proc *p;
473
474    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
475      if(p->state == SLEEPING && p->chan == chan)
476        p->state = RUNNABLE;
477  }
478
479  // Wake up all processes sleeping on chan.
480  void
481  wakeup(void *chan)
482  {
483    acquire(&ptable.lock);
484    wakeup1(chan);
485    release(&ptable.lock);
486  }
487
```

Snippet 5 wakeup and wakeup1

Sleep routine is called with “chan” and “lock”. “chan” is a variable through which the process which called sleep will wake up. It can be used for typical problems like producer/consumer, reader/writer etc. With “lock” different processes working on the same “chan” can work mutually exclusively. We can compare this with standard conditional_variable pattern.

In sleep()[Snippet 4] Line 443, it first verifies if the passed lock is not ptable.lock to avoid deadlock. It releases the lock passed to sleep(). Then it sets the channel for this process and make it SLEEPING so that it can be woken up by some other process [Snippet 4: – 4: Line 450-451]. At this point, it calls sched() which releases the control of the CPU to scheduler thread. Once, the process is run by the scheduler again, it comes back to the sleep(), and then it releases ptable.lock and acquires the lock.

[Snippet 5] wakeup and wakeup1 are just variant which wakes up all thread waiting on the channel. It is similar to pthread_cond_broadcast. If the lock is already is acquired in ptable.lock then we can directly call wakeup1. If not, it is first acquired within wakeup, then it calls wakeup1 and then release ptable.lock. In, wakeup1 lock acquire and release needs to be handled explicitly by caller.

Analysis #1: sleep/wakeup usage in pipe.c API

Xv6 supports pipe api, in which bytes are written at one end, and read at the other end (hence the name pipe). It is a typical producer/consumer problem where bytes are written(produce) by writer(producer) and are read(consume) by reader(consumer).

Each pipe is represented by the struct with some fields, but we are interested in the following fields.

1. Spinlock lock – for exclusive lock in data field, because it is shared b/w writer and reader
2. Data[PIPESIZE] – holds the bytes written or bytes to be read
3. Nread – number of bytes read
4. Nwrite – number of bytes written

Pipealloc() and pipeclose() is pretty straightforward, so we won’t discuss them for the brevity of our producer/consumer analysis.

```
100  int
101  piperead(struct pipe *p, char *addr, int n)
102  {
103    int i;
104
105    acquire(&p->lock);
106    while(p->nread == p->nwrite && p->writeopen){  //DOC: pipe-empty
107      if(myproc()->killed){
108        release(&p->lock);
109        return -1;
110      }
111      sleep(&p->nread, &p->lock); //DOC: piperead-sleep
112    }
113    for(i = 0; i < n; i++){  //DOC: piperead-copy
114      if(p->nread == p->nwrite)
115        break;
116      addr[i] = p->data[p->nread++ % PIPESIZE];
117    }
118    wakeup(&p->nwrite);  //DOC: piperead-wakeup
119    release(&p->lock);
120    return i;
121  }
122
```

Snippet 6 pipe.c – piperead()

```
79  pipewrite(struct pipe *p, char *addr, int n)
80  {
81    int i;
82
83    acquire(&p->lock);
84    for(i = 0; i < n; i++){
85      while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
86        if(p->readopen == 0 || myproc()->killed){
87          release(&p->lock);
88          return -1;
89        }
90        wakeup(&p->nread);
91        sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
92      }
93      p->data[p->nwrite++ % PIPESIZE] = addr[i];
94    }
95    wakeup(&p->nread);  //DOC: pipewrite-wakeup1
96    release(&p->lock);
97    return n;
98  }
```

Snippet 7 pipe.c – pipewrite()

1. There is an uber level pipe lock which is used for mutexing piperead and pipewrite [piperead: line 105, 119, pipewrite: line 83, 96]
2. Piperead:line106 checks if the pipe is empty, if yes, then piperead caller process has to sleep() with waiting on n->read channel and p->lock. Within sleep() it is going to acquire ptable.lock, release the mutex p->lock, set itself on channel p->nread, and then switch back to scheduler. Scheduler then may attempt to schedule writer process in round-robin scheduling.
3. Writer checks if pipe is full or not (p->nwrite == p->nread + PIPESIZE), if it is not full, then it starts filling pipe with items. p->data[p->nwrite++ % PIPESIZE] = addr[i];
4. Let's say if producer find that now the pipe is full, it wakes the consumer up, using the same channel the consumer starts waiting. This happens in wakeup() routine, where it just finds all the processes that are in SLEEPING stage and have the same channel. It sets their state to RUNNABLE. However, the real magic happens in the next step.
5. To relinquish the CPU, it needs to sleep with the same lock but different channel, p->nwrite. Because it needs to wait for consumer to make pipe not-full. So it calls sleep() with same lock so that p->lock can be released, and scheduler can schedule the consumer again.
6. Now, the consumer resumes just after it called sched() during sleep(). There it re-aquires the p->lock, set channel to 0, and releases ptable.lock, and resumes the business as usual.
7. Once, it has consumed all the possible items, it wakes the producer up and releases the p->lock. It simply exits then.

One critical observation here is that pipe code works on 2 different channels, one for read and one for write.

Analysis #2: wait/exit usage in proc.c API

Xv6 provides an api for parent-child problem, where parent needs to wait for child to end, before proceeding further, and child needs to exit gracefully.

```
232  exit(void)
233  {
234    struct proc *curproc = myproc();
235    struct proc *p;
236    int fd;
237
238    if(curproc == initproc)
239      panic("init exiting");
240
241    // Close all open files.
242  > for(fd = 0; fd < NOFILE; fd++){…
248
249    begin_op();
250    iput(curproc->cwd);
251    end_op();
252    curproc->cwd = 0;
253
254    acquire(&ptable.lock);
255
256    // Parent might be sleeping in wait().
257    wakeup1(curproc->parent);
258
259    // Pass abandoned children to init.
260    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
261      if(p->parent == curproc){
262        p->parent = initproc;
263        if(p->state == ZOMBIE)
264          wakeup1(initproc);
265      }
266    }
```

Snippet 8 proc.c - exit API

```
276  int
277  wait(void)
278  {
279    struct proc *p;
280    int havekids, pid;
281    struct proc *curproc = myproc();
282
283    acquire(&ptable.lock);
284    for(;;){
285      // Scan through table looking for exited children.
286      havekids = 0;
287      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
288        if(p->parent != curproc)
289          continue;
290        havekids = 1;
291      > if(p->state == ZOMBIE){…
306      }
307
308      // No point waiting if we don't have any children.
309      if(!havekids || curproc->killed){
310        // cprintf("proc.c::wait::release lock: [proc:%s]\n", p->name);
311        release(&ptable.lock);
312        return -1;
313      }
314
315      // Wait for children to exit.  (See wakeup1 call in proc_exit.)
316      sleep(curproc, &ptable.lock);  //DOC: wait-sleep
317    }
318  }
```

Snippet 9 proc.c – wait API – some code blocks are folded

This is how wait() works.

1. First, it acquires ptable.lock so that if there is any child process is in ZOMBIE state, it can cleanup it. (Cleaning is important, and preventing ZOMBIE apocalypse is even more important.)

2. It iterates through all process and if there is any child who is in ZOMBIE state, it cleans up that child process in ptable and mark it as UNUSED, so that it can be used to allocate a new process. Later, it releases the lock, and simply returns.
3. If there is no children, or the parent process itself is killed, then it simply releases the lock returns.
4. It is possible that child is taking some to complete its execution, in that case, there is no point consuming CPU cycles, and it just simply sleeps with parent (current process) as a channel and ptable.lock as a common lock.

This is how exit() works.

1. It acquires ptable.lock as it needs to mark all the child processes ZOMBIE in case the current process is a parent one. (Ideally parent should wait for all of its children to exit gracefully before exiting, but we have to deal with not-so-ideal situations as well, that is just life, no?)
2. It wakes up parent using wakeup1, (why wakeup1 instead of wakeup, well, we already have the ptable.lock at this point, so if we call wakeup, then it will be a deadlock, and deadlocks are deadly, so we need to take some smart action here).
3. Now, one weird case here is, what if the exiting process is a parent process? Who will take care of cleaning up its children? Well, there is an ever-running process, that was set up initially, “initproc”. We just ask initproc to take care of them. How nice of “initproc”.
4. Finally it marks itself ZOMBIE, and then calls sched(), only to never return back.
5. Why is the parent process the one to cleanup, and not the child process? Well, a process never clean itself while it is running, because of kstack and pgdir are in use while it is running. That is why parent has to do cleanup. Hence, the parent can only ever cleanup once the child process is done running for the last time.