

Assignment 3: Multithreading

By:

AmirAli Abdolrashidi

Shahriyar Valielahi Roshan

Introduction:

To implement this project, we have mainly configured proc.c. Makefile is also changed slightly. Three new files have been created (thread.c, thread.h, hat.c). Finally, to implement new system calls, we have modified the usual files (sysproc.c, syscall.c, syscall.h, user.h, defs.h, and usys.S).

[We did a lot of work to implement this project. It took at least 30 hours 😊]

Let's dive right into details starting with the most configured file.

Proc.c:

➤ clone()

In this file, we created the clone() function that creates child threads. It is almost identical to fork() with two major differences:

1. Sharing the process memory instead of copying it

```
473 // Allocate process.
474 if((np = allocproc()) == 0)
475 return -1;
476
477
478 //CS202 TODO: Share page table from parent with child.
479 np->pgdir=proc->pgdir;
480 //CS202 - The following lines are the same as fork()
481 np->sz = proc->sz;
482 np->parent = proc;
483 *np->tf = *proc->tf;
```

2. Cloning the stack with parent's stack on top

```
487
488 //CS202 - TODO: Set up stack here
489 //Parent's stack boundary
490 void *startParentStack=(void*) proc->tf->ebp+16; //((Base Pointer, argc, *argv, Program Counter)*4
491 void *endParentStack=(void*) proc->tf->esp;
492 unsigned int parentStackSize= (unsigned int)(startParentStack-endParentStack);
493 //Initialize thread's stack
494 np->tf->esp=(unsigned int)(stack+size-parentStackSize);
495 np->tf->ebp=(unsigned int)(stack+size-16);
496 memmove(stack+size-parentStackSize,endParentStack,parentStackSize);
497
498 // Clear %eax so that fork returns 0 in the child.
499 np->tf->eax = 0;
500
```

To briefly describe what our clone() function does, it first allocates a process with the same address space as the parent's by sharing the parent's page table. Then it creates a stack in the child's address space which has been linked to the end of the parent's stack. To do so, we first calculate the boundaries of the parent's stack and, having allocated the stack memory using malloc() beforehand, we copy the parent stack's content to the top of the new stack. Please note that the first four blocks of the parent stack are reserved for parent arguments and special variables (presumably argc, *argv, parent base pointer and the return address, hence the offset of 16 in the code), marked by the base pointer where the stack pointer starts from.

Next it defines the file descriptors and sets several variables related to the thread like name, and status and finally it returns the PID of the new created thread. At the same time, it also returns 0 to the child so that it knows it is a clone and not the actual parent.

➤ wait()

Sharing the parent page table with the threads implies a trickier deallocation since freeing the memory while others are still using it could even result in a crash, restarting the system! To prevent this, we make sure that only the parent can deallocate its own page table after all threads have exited. Any process whose page table is not the same as its parent can be considered as a parent itself.

```
227     p->kstack = 0;
228
229     if(p->pgdir != p->parent->pgdir)    //CS202 - TODO: If it is not a thread
230         freevm(p->pgdir);
231
232
233     p->state = UNUSED;
```

➤ thsleep()

Based on sleep(), this system call is used to put a thread to sleep in a queue-based lock only. It then returns the control to the scheduler.

```
512 int thsleep()    //CS202 - TODO
513 { //Based upon sleep()
514     if(proc == 0) panic("sleep");
515     acquire(&ptable.lock);
516     // Go to sleep.
517     proc->state = SLEEPING;
518     sched();
519     release(&ptable.lock);
520     return 0;
521 }
```

➤ thwake()

This system call awakens a thread with a known PID when it is time to assign it the queue-based lock. It is based on wakeup().

```

523 int thwake(unsigned int pid)    //CS202 - TODO
524 { //Based upon wakeup()
525     struct proc *p;
526     acquire(&ptable.lock);
527     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
528         if(p->state == SLEEPING && p->pid == pid)
529             p->state = RUNNABLE;
530     release(&ptable.lock);
531     return 0;
532 }

```

Thread.h

This is our main library function declarations. Here, two important structs are defined; “queue” which is basically a linked list for the queue-based lock, and “lock_t” which can be used for creating critical sections. “lock_t” can be used for both spinlock and queue lock.

```

1 //CS202 - TODO
2 typedef struct lock_t lock_t;
3 typedef struct queue queue;
4 struct queue
5 {
6     //int isEmpty;
7     unsigned int pid;
8     queue* next;
9 };
10 struct lock_t
11 {
12     unsigned int flag;
13     unsigned int guard;
14     queue* q;
15 };
16

```

Thread.c

➤ thread_create()

In the thread_create() function, we get a callback function and its argument struct variable. Here we create a stack with a predetermined value (We used 4kB), and clone the process using the clone() system call. The parent then returns, but the child, receiving 0 from clone() will be directed to start the callback routine with the given argument. After it returns, it exits and prepares for termination.

```

1 //CS202 - TODO
2 #include "types.h"
3 #include "user.h"
4 #include "thread.h"
5 #include "x86.h" //For xchg(), the atomic set function
6 #include <stddef.h> //For NULL
7
8 void thread_create(void *(*start_routine)(void*), void* arg)
9 {
10     int size=4096*sizeof(void);
11     void* stack= (void*)malloc(size);
12
13     int childID;
14     childID=clone(stack,size);
15
16     if(childID==0)
17     {
18         //printf(1, "Thread %d Created!!\n",getpid());
19         (*start_routine)(arg);
20         exit();
21     }
22 }
23

```

➤ Spinlock:

In the following figure, you can see the definition of three functions related to the spinlock. lock_init() initializes the spin lock by setting flag to zero (guard and q are used for queue-based locks only); threads use lock_acquire() to claim the lock by using atomic test- and-set (called xchg()) on the flag; and lock_release() can be used to release the lock by setting the flag to zero atomically.

```

24 void lock_init(lock_t *lock)
25 {
26     lock->flag=0;
27     lock->guard=0;
28     lock->q=NULL;
29 }
30
31
32 void lock_acquire(lock_t *lock) //Spinlock
33 {
34     while(xchg(&lock->flag,1)==1);
35 }
36
37 void lock_release(lock_t *lock)
38 {
39     xchg(&lock->flag,0);
40 }

```

➤ Queue-based lock

Similar to spinlock, this lock is also initialized using `lock_init()`. The functions `lock_qacquire()` and `lock_qrelease()` are derived from the TEP book. Every thread in `lock_qacquire()` checks to see if the lock is available and if it is unable to get it, the thread's PID is added to the end of the queue of the lock and put to sleep using `thsleep()`.

```
44 void lock_qacquire(lock_t *lock) //Mr. Anderson's lock!
45 {
46     while(xchg(&lock->guard, 1) == 1);
47     if(lock->flag == 0)
48     {
49         lock->flag = 1;
50         lock->guard = 0;
51     }
52     else
53     {
54         qput(&lock->q, getpid());
55         lock->guard = 0;
56         thsleep();
57     }
58 }
```

`lock_qrelease()` checks if the queue is empty using the `qlen()` function (which returns the length of the queue; 0 is empty). If the queue is empty, it releases the lock. Otherwise, it pops the PID in the front of the queue out and wakes the corresponding thread to capture the lock.

```
59 // Releasing the queue lock
60 void lock_qrelease(lock_t *lock)
61 {
62     while(xchg(&lock->guard, 1) == 1);
63     if(qlen(lock->q) == 0)
64     {
65         lock->flag = 0;
66     }
67     else
68     {
69         unsigned PID = qget(&lock->q);
70         if(PID != 0) thwake(PID);
71     }
72     lock->guard = 0;
73 }
74 }
```

The key functions to implement and utilize the queue are `qput()` and `qget()`, which add and remove elements in the linked list respectively.

```

77 void qput(queue** Q,unsigned int PID)
78 {
79     queue* qtemp;
80     int c=0;
81
82     qtemp=(queue*)malloc(sizeof(queue));
83     qtemp->pid=PID;
84     qtemp->next=NULL;
85
86     if(*Q==NULL)
87     {
88         *Q=qtemp;
89         // printf(1,"QUEUE ANNOUNCEMENT: QUEUE CREATED!\n");
90     }
91     else
92     {
93         queue* qtemp2=*Q;
94         while(qtemp2->next!=NULL) {qtemp2=qtemp2->next;c++;}
95         qtemp2->next=qtemp;c++;
96     }
97
98     // printf(1,"QUEUE ANNOUNCEMENT: %d has been put in location %d.\n",qtemp->pid,c);
99 }

101 // Get the element at the front of the queue
102 unsigned int qget(queue** Q)
103 {
104     if(*Q!=NULL)
105     {
106         unsigned int targetPID=(*Q)->pid;
107         if((*Q)->next!=NULL)
108         {
109             queue* qtemp=*Q;
110             *Q=(*Q)->next;
111             free(qtemp);
112         }
113         else
114         {
115             free(*Q);
116             *Q=NULL;
117             // printf(1,"QUEUE ANNOUNCEMENT: QUEUE EMPTY!\n");
118         }
119
120         // printf(1,"QUEUE ANNOUNCEMENT: %d has been removed.\n",targetPID);
121         return targetPID;
122     }
123     return 0;
124 }

```

Hat.c

This is our user program which simulates “a game of Frisbee” between the created threads.

➤ Argument bundle struct

Since we have only one slot to pass the arguments to the callback function, we define a struct “argThread” to send all them at once. It consists of a local variable, the thread ID (id) and shared variables such as

number of threads (totalT), intended number of passes (iterNum), current number of passes (counter), lock (lock), runtime counter (runtime) and of course the “Frisbee” itself (token).

```
1 //TODO
2 #include "types.h"
3 #include "stat.h"
4 #include "user.h"
5 #include "thread.h"
6
7 //int counter=0;
8
9 struct argThread
10 {
11     int* iterNum; // Shared
12     int* counter; // Shared
13     int* token; // Shared
14     int* totalT; // Shared
15     int* runtime; // Shared
16     lock_t *lock; // Shared
17     int id; // Local
18 }
19 };
20 typedef struct argThread argThread;
```

➤ Callback function for threads (*ball)

In this function, threads iteratively acquire a lock to get access to the token. If the value of the token is the same as a thread’s id, it can claim it and then increment it so the next thread can get access to it (the last thread sets it to zero). Subsequently the thread releases the lock. This iteration continues and the token is circulated between our created threads and “counter” is incremented every time until it reaches “iterNum”.

```
22 void* ball(void* arg)
23 {
24     argThread* n=(argThread*) arg;
25     int pid=getpid();
26     while(1)
27     {
28         lock_acquire(n->lock);
29         (*n->runtime)++;
30         if(*n->counter>=*n->iterNum)
31         {
32             lock_release(n->lock);
33             break;
34         }
35
36         if (*n->token == n->id){
37             printf(1, "TOKEN: %d - THREAD# %d:\n%d This message can be cut into by another thread unless we have locks installed.\n",
38                 *n->token, n->id, *n->runtime);
39             *n->token = (*n->token+1)%(*n->totalT);
40             (*n->counter)++;
41         }
42         lock_release(n->lock);
43     }
44
45     return 0;
46 }
```


➤ Main function

The main function creates the argument bundles for each thread's callback function, uses them to create the threads and waits for each thread to exit.

```
48 int
49 main(int argc, char *argv[])
50 {
51     //Argument initialization
52     int maxLimit=10,numThread=2,counter=0,frisbee=0,runtime=0,i;
53     lock_t lock;
54     if(argc>1) numThread=atoi(argv[1]);
55     if(argc>2) maxLimit=atoi(argv[2]);
56     lock_init(&lock);
57
58     //Argument bundle for threads
59     argThread *arg_t = (argThread*)malloc(numThread*sizeof(argThread));
60     for (i=0; i<numThread; i++)
61     {
62         arg_t[i].counter=&counter;
63         arg_t[i].iterNum=&maxLimit;
64         arg_t[i].lock=&lock;
65         arg_t[i].runtime=&runtime;
66         arg_t[i].totalT=&numThread;
67         arg_t[i].token = &frisbee;
68         arg_t[i].id=i;
69     }
70
71     //Begin weaving the threads!
72     //void* t=(void*)&arg_t;
73     for(i=0;i<numThread;i++)
74     {
75         thread_create(ball,(void*)&(arg_t[i]));
76     }
77     for(i=0;i<numThread;i++) wait(); //I wait for every thread
78
79     //End
80     printf(1, "Your time has come! Final Counter Value = %d Runtime = %d\n",counter,runtime);
81     free(arg_t);
82     exit();
83 }
```

➤ To run hat on xv6, you may pass two arguments: number of threads (2 by default) and number of passes (10 by default):

- o hat <#Threads=2> <#Passes=10>

Makefile

We added the thread.o to the user libraries and also added "hat" as a user program.

```
135 ULIB = ulib.o usys.o printf.o umalloc.o thread.o
136 #TODO: thread.o added to the user libraries
137
138 UPROGS=\
139     _cat\
140     _echo\
141     _forktest\
142     _grep\
143     hat\
144     _init\
145     _kill\
```



```
241 EXTRA=\
242 mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c hat.c\
243 ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
244 printf.c umalloc.c\
245 README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
246 .gdbinit.tmpl gdbutil\
247
```

Sample Run

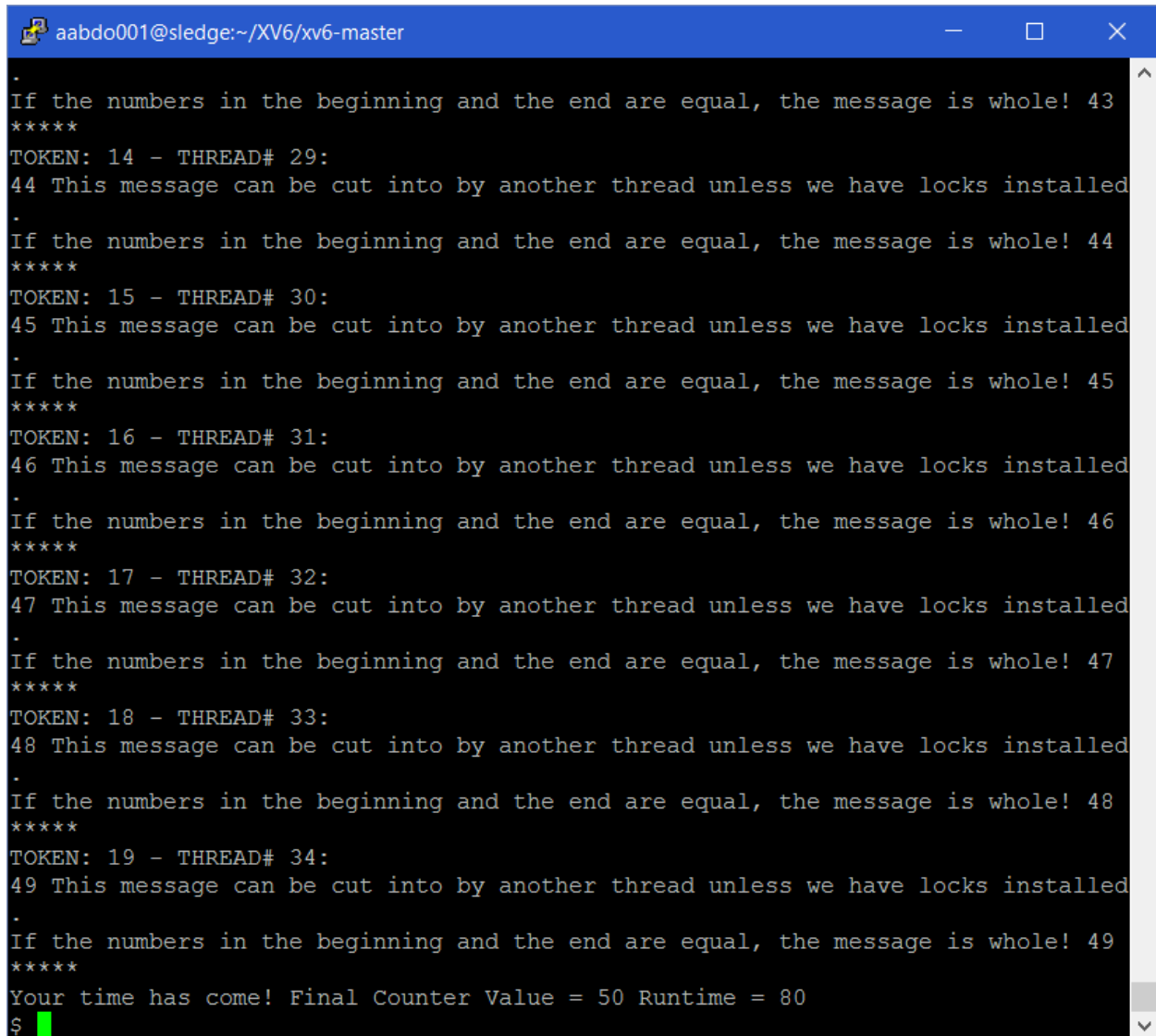
```
aabdo001@sledge:~/XV6/xv6-master
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000159116 s, 3.2 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
274+1 records in
274+1 records out
140562 bytes (141 kB) copied, 0.000727544 s, 193 MB/s
qemu -serial mon:stdio -hdb fs.img xv6.img -smp 2 -m 512
xv6...
cpu1: starting
cpu0: starting
init: starting sh
$ hat 2 5
TOKEN: 0 - THREAD# 4:
0 This message can be cut into by another thread unless we have locks installed.
If the numbers in the beginning and the end are equal, the message is whole! 0
*****
TOKEN: 1 - THREAD# 5:
1 This message can be cut into by another thread unless we have locks installed.
If the numbers in the beginning and the end are equal, the message is whole! 1
*****
TOKEN: 0 - THREAD# 4:
2 This message can be cut into by another thread unless we have locks installed.
If the numbers in the beginning and the end are equal, the message is whole! 2
*****
TOKEN: 1 - THREAD# 5:
3 This message can be cut into by another thread unless we have locks installed.
If the numbers in the beginning and the end are equal, the message is whole! 3
*****
TOKEN: 0 - THREAD# 4:
4 This message can be cut into by another thread unless we have locks installed.
If the numbers in the beginning and the end are equal, the message is whole! 4
*****
Your time has come! Final Counter Value = 5
$
```

Performance Analysis

We have tested the performance of the two implemented locks using the “runtime” variable. It does not always give us the same result, but it remains in the same order.

Below you can see sample screenshots for the two methods.

- Queue: (hat 30 50)



```
aabdo001@sledge:~/XV6/xv6-master
.
If the numbers in the beginning and the end are equal, the message is whole! 43
*****
TOKEN: 14 - THREAD# 29:
44 This message can be cut into by another thread unless we have locks installed
.
If the numbers in the beginning and the end are equal, the message is whole! 44
*****
TOKEN: 15 - THREAD# 30:
45 This message can be cut into by another thread unless we have locks installed
.
If the numbers in the beginning and the end are equal, the message is whole! 45
*****
TOKEN: 16 - THREAD# 31:
46 This message can be cut into by another thread unless we have locks installed
.
If the numbers in the beginning and the end are equal, the message is whole! 46
*****
TOKEN: 17 - THREAD# 32:
47 This message can be cut into by another thread unless we have locks installed
.
If the numbers in the beginning and the end are equal, the message is whole! 47
*****
TOKEN: 18 - THREAD# 33:
48 This message can be cut into by another thread unless we have locks installed
.
If the numbers in the beginning and the end are equal, the message is whole! 48
*****
TOKEN: 19 - THREAD# 34:
49 This message can be cut into by another thread unless we have locks installed
.
If the numbers in the beginning and the end are equal, the message is whole! 49
*****
Your time has come! Final Counter Value = 50 Runtime = 80
$
```

- Spinlock: (hat 30 50)

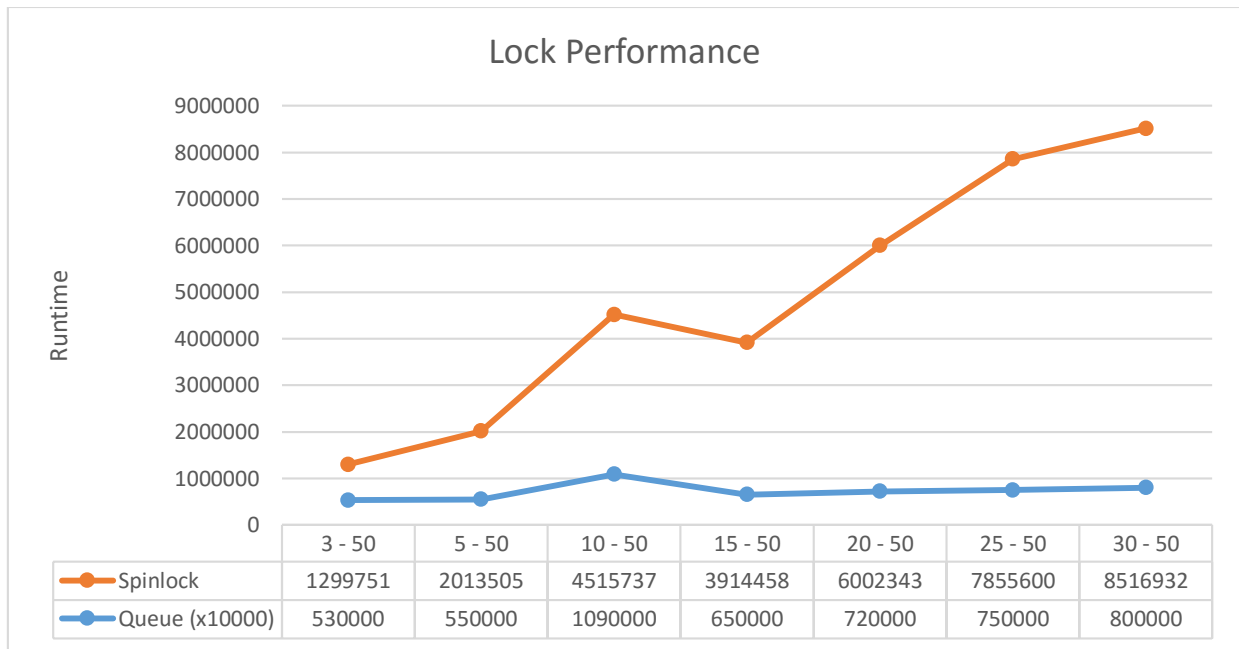
```

aabdo001@sledge:~/XV6/xv6-master
.
If the numbers in the beginning and the end are equal, the message is whole! 43
*****
TOKEN: 14 - THREAD# 18:
44 This message can be cut into by another thread unless we have locks installed
.
If the numbers in the beginning and the end are equal, the message is whole! 44
*****
TOKEN: 15 - THREAD# 19:
45 This message can be cut into by another thread unless we have locks installed
.
If the numbers in the beginning and the end are equal, the message is whole! 45
*****
TOKEN: 16 - THREAD# 20:
46 This message can be cut into by another thread unless we have locks installed
.
If the numbers in the beginning and the end are equal, the message is whole! 46
*****
TOKEN: 17 - THREAD# 21:
47 This message can be cut into by another thread unless we have locks installed
.
If the numbers in the beginning and the end are equal, the message is whole! 47
*****
TOKEN: 18 - THREAD# 22:
48 This message can be cut into by another thread unless we have locks installed
.
If the numbers in the beginning and the end are equal, the message is whole! 48
*****
TOKEN: 19 - THREAD# 23:
49 This message can be cut into by another thread unless we have locks installed
.
If the numbers in the beginning and the end are equal, the message is whole! 49
*****
Your time has come! Final Counter Value = 50 Runtime = 8516932
$

```

Number of threads – Number of passes

Method	3 - 50	5 - 50	10 - 50	15 - 50	20 - 50	25 - 50	30 - 50
Spinlock	1299751	2013505	4515737	3914458	6002343	7855600	8516932
Queue	53	55	109	65	72	75	80



Note: Search for “CS202” or “TODO” to find our changes in xv6.