

LAB 1 – SYSTEM CALL & LOTTERY

Part1.

For the first part, I edited several files.

1) **Sysproc.c:** The syscall implemented in this file

```
int
sys_returncount(void)
{
    cprintf("-----\n");

    cprintf("Attention:\nnumber of system calls for each process has been
returned!\n");

    cprintf("%d %s: sys call count: %d\n",
proc->pid, proc->name, proc->sysCount);

    cprintf("-----\n");

    return 0;
}
```

I designed returncount(void) for returning the number of system calls for each process. As you can see I defined a sysCount variable in proc struct.

- 2) **Proc.h:** I add sysCount variable in proc.
- 3) **Syscall.h:** I defined the position of system call vector 22
- 4) **Syscall.c:** Each system call will use syscall(void) function, and I will add system call variable here. I also add 2 line to define system call in this file.
- 5) **User.h:** he I will define a function that can be called through the command prompt
- 6) **Usys.s:** define a macro to define connect the call of user to the system call function
- 7) **Defs.h:** add a forward declaration for your new system call [From Stack overflow 😊]

I hate to document code inside a report! You can find each function definition inside each program by searching DRSVR or cs202. 😊

Check hello.c for test case, to return number of system call count, you should run the system call function from the user program. The picture below shows the number of sys calls for

```
QEMU - Press Ctrl-Alt to exit mouse grab
iPXE v1.0.0-591-g7aee315
iPXE (http://ipxe.org) 00:03.0 C900 PCI2.10 PnP PMM+1FFC8D60+1FF88D60 C900

Booting from Hard Disk...

cpu0: starting xv6

ioapicinit: id isn't equal to ioapicid; not a MP
cpu0: starting
init: starting sh
$ hello 13:
DRSVR TEST 0 HAS BEEN STARTED!
Process 4 hello purchased 8 tickets! :)

Attention:
number of system calls for each process has been returned!
4 hello: sys call count: 34

PID:4 hello RUN:10
PID:3 sh RUN:3
$ -
```

LAB 1 – SYSTEM CALL & LOTTERY

Part2.

For the second part, I implement a system call called buy ticket() that sets the number of tickets for each process when the user program calls it.

I add two variable to proc, ticketCount and runCount. TicketCount stores number of dedicated tickets, and runCount will store number of time slots that the scheduler will dedicate to the process.

```
int test = 0;

int totalTickets = 0;

// Calculate total ticket count
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    totalTickets = totalTickets + p->ticketCount;
}

release(&ptable.lock);
acquire(&ptable.lock);

if (LOOPAVOID == 0){
    SUM = 1;
    MAX = 1;
}

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE){
        //test++;
        continue;
    }

    if (totalTickets>0) {
        MAX = totalTickets;
    }

    int JACKPOT = generateRandom(MAX);

    // Switch to chosen process. It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.

    if ( p->ticketCount > 0) {
        //cprintf("\n1SUM:%d JACKPOT:%d PID:%d RUN:%d\n", SUM,JACKPOT,p->pid,p->runCount);
        SUM += p->ticketCount;
        //cprintf("\n2SUM:%d JACKPOT:%d PID:%d RUN:%d\n", SUM,JACKPOT,p->pid,p->runCount);
        test++;
    }
    else {
        //cprintf("\n0SUM:%d JACKPOT:%d\n", SUM,JACKPOT);
        SUM++;
    }

    if (SUM < JACKPOT){
        LOOPAVOID = 1;
        //cprintf("\nContinue!");
        continue;
    }

    LOOPAVOID = 0;
    proc = p;

    p->runCount++;
}
```

LAB 1 – SYSTEM CALL & LOTTERY

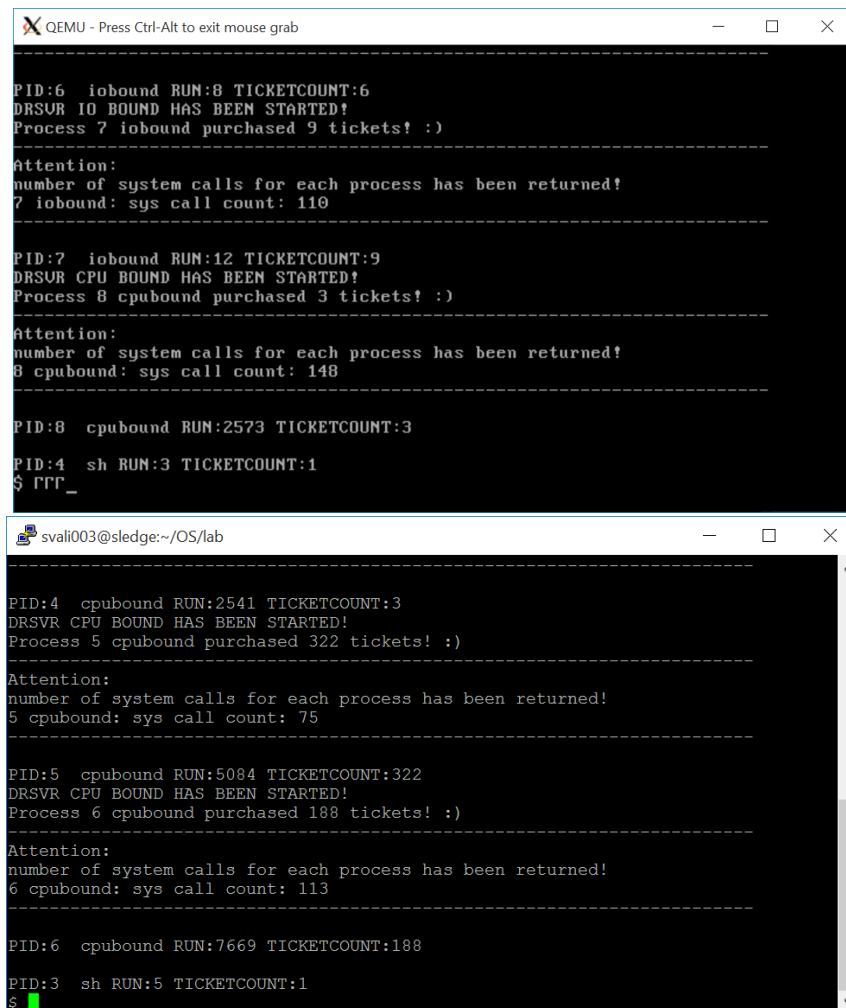
You can see the code above. You can also take a look at the generate random function that I write.

Algorithm:

- 1- Count total number of tickets
- 2- Hold the lottery and set the jackpot ticket
- 3- Reset number of sum
- 4- For each process add ticketcount of each process to the sum
- 5- The winner is the process that has more tickets than the jackpot

Note:

I add LOOPAVOID for situation that the process queue has no winner, so the number of sum will reset to 1, so always we loop and we never have a winner. So I add this value to stop resetting sum when we have unscheduled process.



```
QEMU - Press Ctrl-Alt to exit mouse grab

PID:6 iobound RUN:8 TICKETCOUNT:6
DRSVR IO BOUND HAS BEEN STARTED!
Process 7 iobound purchased 9 tickets! :)

-----
Attention:
number of system calls for each process has been returned!
7 iobound: sys call count: 110
-----

PID:7 iobound RUN:12 TICKETCOUNT:9
DRSVR CPU BOUND HAS BEEN STARTED!
Process 8 cpubound purchased 3 tickets! :)

-----
Attention:
number of system calls for each process has been returned!
8 cpubound: sys call count: 148
-----

PID:8 cpubound RUN:2573 TICKETCOUNT:3
PID:4 sh RUN:3 TICKETCOUNT:1
$ !!!_

svali003@sledge:~/OS/lab

PID:4 cpubound RUN:2541 TICKETCOUNT:3
DRSVR CPU BOUND HAS BEEN STARTED!
Process 5 cpubound purchased 322 tickets! :)

-----
Attention:
number of system calls for each process has been returned!
5 cpubound: sys call count: 75
-----

PID:5 cpubound RUN:5084 TICKETCOUNT:322
DRSVR CPU BOUND HAS BEEN STARTED!
Process 6 cpubound purchased 188 tickets! :)

-----
Attention:
number of system calls for each process has been returned!
6 cpubound: sys call count: 113
-----

PID:6 cpubound RUN:7669 TICKETCOUNT:188
PID:3 sh RUN:5 TICKETCOUNT:1
$
```

LAB 1 – SYSTEM CALL & LOTTERY

Simulation:

I have two type of process. CPU Bound, and IO BOUND. CPU bound will multiply 2, 1000 times. And IO BOUND will allocate 50 array. Base on the resource limitation, I could not use more than 50.

The point is that the number of RUN of time slices are proportional to the number of tickets (1.33 for IOBOUND). For the cpu bound we have more time slices.

Note: I disabled the second CPU.

I wanted to draw the graph, bud it was just time consuming. For the IOBOUND the ratio was constant ☺

TICKET	RUN(CPU)
6 (IO)	8
12 (IO)	9
3 (CPU) concurrent io too	2573
3 (CPU) just cpu bound process	2541
188 (CPU)	7669
322 (CPU)	5084

Conclusion:

- 1- When we have IO BOUND and CPU BOUND processes at the same time, the number of run times is more with the same amount of ticket.
- 2- When we have IO BOUND processes, the ratio is constant (At least in my observation)
- 3- When we have CPU Bound processes, the ration of RUN to TICKET was not linear. As you can see a process with less cpu had lower run time slices. As it depends on the lottery value. But basically when we have higher number of ticket, we have more chance to win (sum>jackpot), but sometimes it depend on the order. (A process with large number of tickets before one make a sum rise near the jackpot, so the next process with lower number of tickets can win the lottery.
- 4- Another point is that the order, As I noticed, the order was the same as the order of calling. Unfortunately I did not have time to design a good testbench, to make sure about the correct order.

LAB 2 – NULL POINTER DEREFERENCE

Changes

Before telling you about which files that I change, let take a look at how things working in the xv6. The first thing is forking a new process, as a memory perspective, the important function is “copyvm” that you can find it in a vm.c. Also we should change the location that program loads in exec() from 0 to PGSIZE.

To stop compiler from load into page 0 we should also change the make file.

- 1) **vm.c:** Instead of staring copying in copyvm from page 0, i.e. ‘i=0’, we should start from i=PGSIZE which is the next page, i.e. page 1;

```
for(i = PGSIZE; i < sz; i += PGSIZE){
```

- 2) **exec.c:** Load program into page 1 of the memory (userspace);

```
sz = PGSIZE;
```

- 3) **MakeFile:** Change start execution address to 0x1000 (4096) [each page is 4k]

```
_%. %.o $(ULIB)
$(LD) $(LDFLAGS) -N -e main -Ttext 0x1000 -o $@ $^
$(OBJDUMP) -S $@ > $.asm
$(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* //; /^$$/d' > $.sym
```

```
_forktest: forktest.o $(ULIB)
# forktest has less library code linked in - needs to be small
# in order to be able to max out the proc table.
$(LD) $(LDFLAGS) -N -e main -Ttext 0x1000 -o _forktest forktest.o ulib.o usys.o
$(OBJDUMP) -S _forktest > forktest.asm
```

These changes are enough for Null pointer dereference without kernel, but when we need have a syscall, we should change the required code for add syscall (like lab 1), plus additional changes in the syscall.c for checking pointers before passing (argptr());

- 4) **Sysproc.c:**

```
int
sys_getdrsvrptr(void)
{
    char *pointer ;

    if(argptr(0, &pointer, 1) < 0) {
        cprintf("DRSVR Found NULL POINTER IN KERNEL! :( \n");
        return -1;
    }
    else {
        cprintf("DRSVR Pointer has been set in kernel! :) \n");
        return 0;
    }
}
```

LAB 2 – NULL POINTER DEREFERENCE

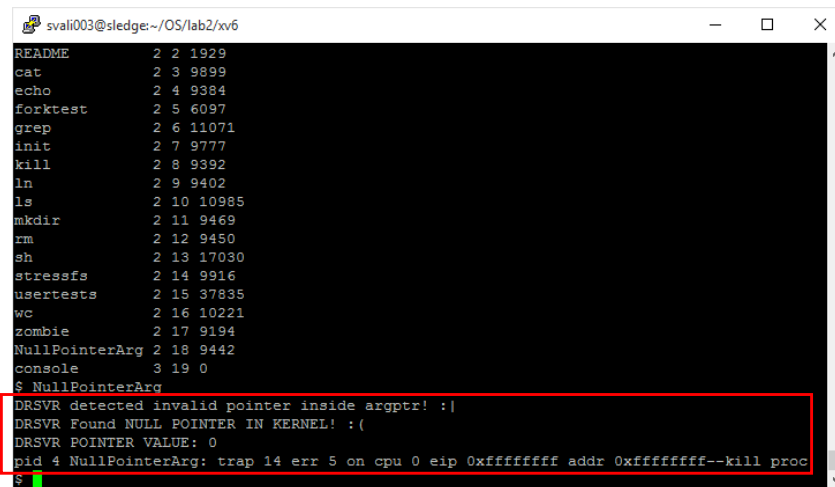
5) **Syscall.c:**

```
if((uint)i >= proc->sz || (uint)i+size > proc->sz || (uint)i == 0 ) {  
    // DRSVR - CS202 - LAB2  
    cprintf("DRSVR detected invalid pointer inside argptr! :| \n");  
    return -1;  
}
```

6) **Proc.h, Syscall.h, User.h, Usys.s, Defs.h; [For adding Syscall]**

7) **NullPointerArg.c:** User code for testing my job ☺

```
#include "types.h"  
#include "user.h"  
  
int main(int argc, char* argv[]){  
  
    int * myPointer = 0; // for null dereference  
    char * myPointer2 = 0; // for kernel test  
  
    getdrsvrptr(myPointer2); // syscall kernel test  
  
    printf(1,"DRSVR POINTER VALUE: %d\n",myPointer); // user test  
  
    return 0;  
}
```



```
svali003@sledge:~/OS/lab2/xv6  
README      2 2 1929  
cat          2 3 9899  
echo        2 4 9384  
forktest    2 5 6097  
grep        2 6 11071  
init        2 7 9777  
kill        2 8 9392  
ln          2 9 9402  
ls          2 10 10985  
mkdir       2 11 9469  
rm          2 12 9450  
sh          2 13 17030  
stressfs    2 14 9916  
usertests   2 15 37835  
wc          2 16 10221  
zombie      2 17 9194  
NullPointerArg 2 18 9442  
console     3 19 0  
$ NullPointerArg  
DRSVR detected invalid pointer inside argptr! :|  
DRSVR Found NULL POINTER IN KERNEL! :(  
DRSVR POINTER VALUE: 0  
pid 4 NullPointerArg: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffffffff--kill proc  
$
```

You can find my changes by searching for DRSVR or CS202 inside the code ☺ As you can see process has been trapped and killed because of Null pointer dereference.

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.",
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     return 0;
45 }
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

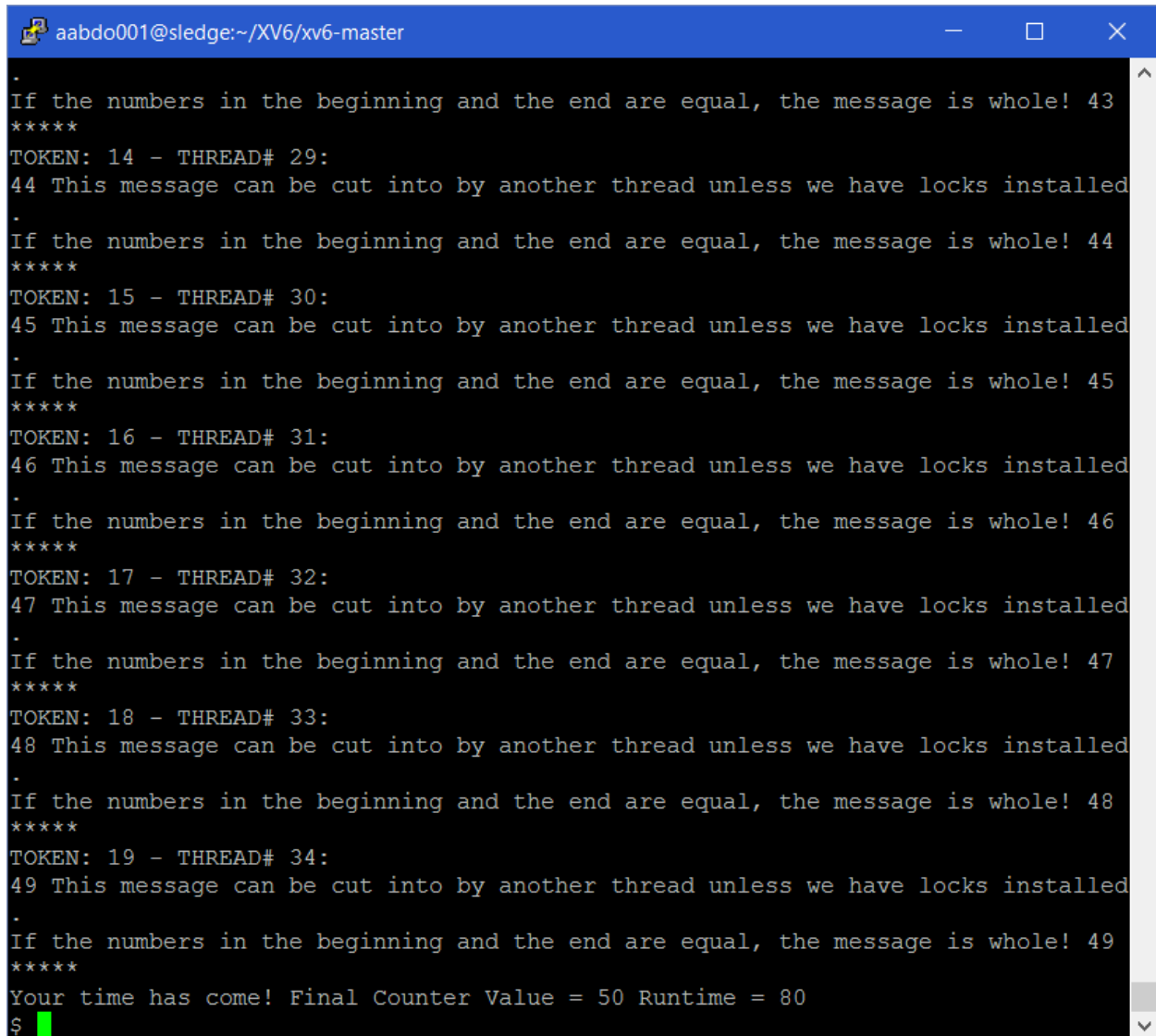
```
aabdo001@sledge:~/XV6/xv6-master
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000159116 s, 3.2 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
274+1 records in
274+1 records out
140562 bytes (141 kB) copied, 0.000727544 s, 193 MB/s
qemu -serial mon:stdio -hdb fs.img xv6.img -smp 2 -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)

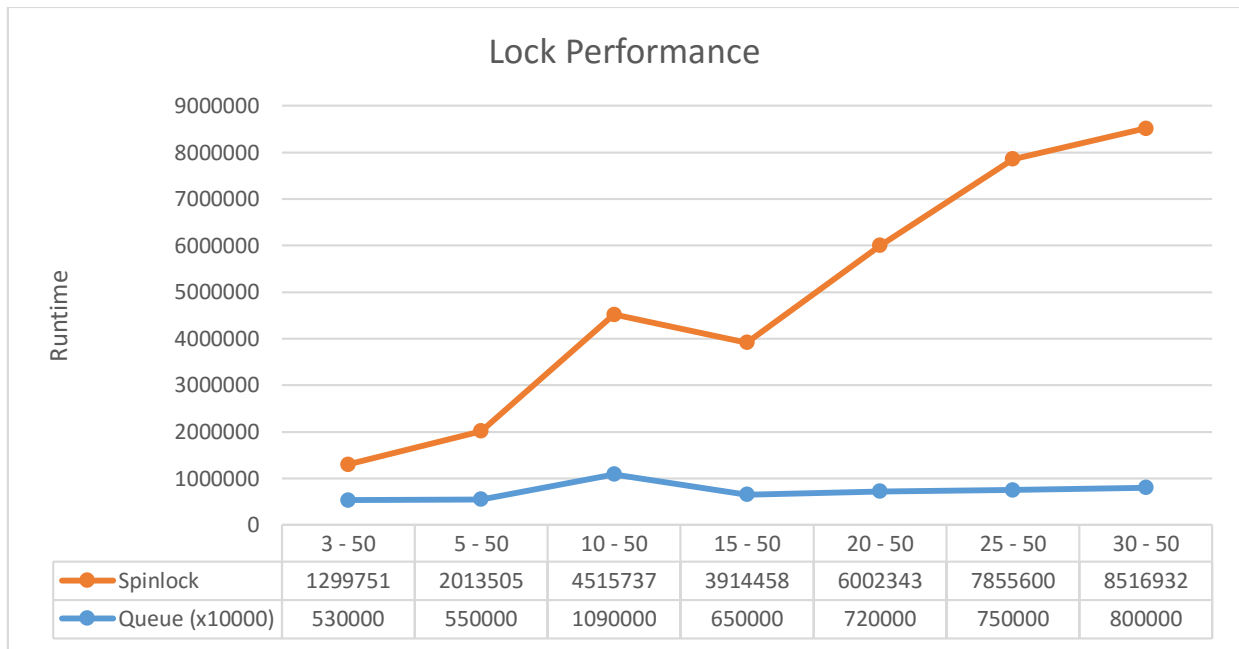
```

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.