



University of California, Riverside
Department of Computer Science and Engineering

Title: Lab 1 - OS structure and Scheduler

Course: Advanced Operation Systems

Mahbod Afarin

Spring 2021

1. Part A: System call and xv6 basic structure

In this part of the project, we have to do some changes in the xv6 in order to count the number of the process in the system, count the total number of the system calls that the current process has made so far, as well as the number of memory pages the current process is using. In the first subsection, I will discuss the details of changing in code, and in the second subsection. I am going to show the results of the sample test file.

1-1 Changes in the code

For counting the number of process in the system, count the total number of the system calls that the current process has made so far, as well as the number of memory pages the current process is using, we have to do some changes in the following files to add `info` system call.

- `proc.c`
- `sysproc.c`
- `syscall.h`
- `syscall.c`
- `user.h`
- `defs.h`
- `usys.S`

In the future sections, first, I will provide the details about what I added to the program.

1-1-1 Changes in *proc.h*

In `proc.h`, I added `NuSysCall` to the struct `proc` to count the number of the system calls per process. Here we have `sz` in `struct proc` which give us the size of the process memory in bytes, and we can simply calculate the number of the memory pages uses by each process with dividing this number by page size (`sz/PGSIZE`).

1-1-2 Changes in *sysproc.c*

In `sysproc.c`, I have defined the system call and named it `sys_info()`. For getting argument from input, I used `argint` and it will pass the input argument to variable `i`.

Listing 1: changes in sysproc.c

```
1  int
2  sys_info(void)
3  {
4      int i;
5      if (argint(0, &i) < 0)
```

```
6         return -1;
7     return info(i);
8 }
```

1-1-3 Changes in *syscall.c*

In *syscall.c*, I added `extern int sys_info(void)` and `[SYS_info] sys_info` in *syscall.c*. I have defined `sys_count(void)` as extern integer, because I already declare it in another file named *sysproc.c*. Also, I have added `++curproc->NuSysCall;` to the system call function to count the number of the system calls for each process.

1-1-4 Changes in *syscall.h*

In *syscall.h*, I have defined the system call and added this line `#define SYS_info 22` at the end of the definitions.

1-1-5 Changes in *user.h*

In *user.h*, I have added `int info(void)`.

1-1-6 Changes in *usys.S*

In *usys.S*, I have added `SYSCALL(info)`.

1-1-7 Changes in *proc.c*

In *proc.c*, I have added the following part. It will get the value of `I` and based on that value, it will return the number of process, number of systems calls for the current process, as well as the number of pages for that process. In the other part of the *proc.c*, I have initialized the `NuSysCall` also, I have defined a variable for counting the number of processes and updated it for each process. Here we have `proc->sz` gives us the size of the process memory in bytes, and we can simply calculate the number of the memory pages uses by each process with dividing this number by page size (`sz/PGSIZE`).

Listing 2: The changes in proc.c

```
1  int
2  info(int i)
3  {
4      struct proc *proc = myproc();
5      if (i == 1)
6          return num_of_Processes;
7      else if (i == 2)
8          return proc->NuSysCall;
9      else if (i == 3)
10         return proc->sz/PGSIZE;
11     else {
```

```
12         exit();
13     }
14     return i;
15 }
```

1-2 Testing the program

For testing the program, I have created a test file called `prog0.c`. This test file is already available in the program directory. As it is clear in the listing 1, it will set three different numbers (1, 2, and 3) for info for two nested loops, and the results should be the number of the process in the system, the number of the system calls for current process so far, and the number of the memory pages used by that specific process.

Listing 3: The test code for testing the first part of the project

```
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int main(int argc, char* argv[])
6  {
7      int i, k;
8      const int loop = 100;
9      for(i = 0; i < loop; i++)
10     {
11         asm("nop");
12         for(k=0; k < loop; k++) {
13             asm("nop");
14         }
15     }
16     printf(1, "The number of Processes: %d \n", info(1));
17     printf(1, "The number of sysCalls for process: %d \n", info(2));
18     printf(1, "The number of the memory pages used by process: %d \n", info(3));
19     exit();
20 }
```

We have to add the `prog0.c` to the make file, so I did the following changes in the make file to do that:

Listing 4: Changes in the make file to add prog0.c

```
1  \_prog0\ // in UPROGS
2  \prog0.c\ // in EXTRA=\
```

Figure 1, show the output screenshot of the `prog0`. As it is clear from the figure, the number of the process in system is 3, the number of the system calls for `prog0` is 31, and the number of the memory pages used by `prog0` is 3.

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ prog0
The number of Processes: 3
The number of sysCalls for process: 31
The number of the memory pages used by process: 3
```

Figure 1: the screenshot of the output results of prog0.c

2 Part B: Scheduling

For this part of the project, I have implemented the lottery and stride scheduling algorithm. In the first part of this section, I will talk about the changes in the code for supporting lottery and stride scheduling. In the second section, I will talk about the results of testing the lottery and stride scheduling.

2-1 Changes in code to support lottery and stride scheduling

For supporting lottery and stride scheduling, I have made some changes in the following files.

- proc.c
- sysproc.c
- syscall.h
- syscall.c
- user.h
- defs.h
- usys.S

1-1-1 Changes in *proc.h*

In [proc.h](#), I did the following changes in the `proc` struct to implement lottery and stride scheduling. I also added `int totalNumTickets;` to store the total number of tickets for each process.

Listing 5: Changes in proc.h

```
1  int myTicket;
2  int numOfTicks;
3  int myStride;
4  int myCount;
5  int stride;
```

1-1-2 Changes in *sysproc.c*

In [sysproc.c](#), I have defined the following system calls. The `sys_SetTicket` system call will get the number of the tickets from the user.

Listing 6: Changes in sysproc.c

```
1  int
2  sys_setTicket(void)
3  {
4      int i;
5      argint(0, &i);
6      return setTicket(i);
7  }
```

1-1-3 Changes in *syscall.c*

In *syscall.c*, I added the following changes to support `sys_setTicket` system call.

Listing 7: Changes in syscall.c

```
1  extern int sys_setTicket(void);
2
3  [SYS_setTicket] sys_setTicket,
```

1-1-4 Changes in *syscall.h*

In *syscall.h*, I have defined the system call and added this line `define SYS_setTicket 23` at the end of the definitions.

1-1-5 Changes in *user.h*

In *user.h*, I have added `int setTicket(int);`

1-1-6 Changes in *usys.S*

In *usys.S*, I have added `SYSCALL(setTicket)`.

1-1-7 Changes in *proc.c*

In *proc.c*, first of all I defined `setTicket` in order to set tickets for each process.

Listing 8: The changes in proc.c

```
1  int setTicket(int i)
2  {
3      totalNumTickets = totalNumTickets - myproc()->myTicket;
4      if(totalNumTickets < 0) totalNumTickets = 0;
5      myproc()->myTicket = i;
6      myproc()->numOfTicks = 0;
7      totalNumTickets = totalNumTickets + i;
8      return 0;
9  }
```

Here is my lottery scheduler implementation:

```

1  void
2  scheduler(void)
3  {
4      struct proc *p;
5      struct cpu *c = mycpu();
6      c->proc = 0;
7
8      //Lab0 second part
9      int lottery;
10     int lotteryVal;
11
12     for(;;){
13         // Enable interrupts on this processor.
14         sti();
15
16         //Lab0 second part
17         lotteryVal = 0;
18         if(totalNumTickets > 0)
19         {
20             lottery = randomGen(totalNumTickets);
21         } else {
22             lottery=0;
23         }
24
25         // Loop over process table looking for process to run.
26         acquire(&ptable.lock);
27         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
28             if(p->state != RUNNABLE)
29                 continue;
30
31             //Lab0 second part
32             lotteryVal += p->myTicket;
33             if (lotteryVal >= lottery)
34             {
35                 p->numOfTicks++;
36
37                 // Switch to chosen process. It is the process's job
38                 // to release ptable.lock and then reacquire it
39                 // before jumping back to us.
40                 c->proc = p;
41                 switchvm(p);
42                 p->state = RUNNING;
43
44                 switch(&(c->scheduler), p->context);
45                 switchkvm();
46
47                 // Process is done running for now.
48                 // It should have changed its p->state before coming back.
49                 c->proc = 0;
50                 break;
51             }
52         }
53         release(&ptable.lock);
54     }
55 }

```

Here is my stride scheduler implementation:

Listing 10: Stride scheduler

```
1  void
2  scheduler(void)
3  {
4      struct proc *p1 = myproc();
5      float StrideMin;
6      struct proc *p;
7      struct cpu *c = mycpu();
8      c->proc = 0;
9
10     for(;;){
11         // Enable interrupts on this processor.
12         sti();
13         StrideMin = 1000000;
14
15         // Loop over process table looking for process to run.
16         acquire(&ptable.lock);
17         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
18             if(p->state != RUNNABLE)
19                 continue;
20
21             if (p->stride < StrideMin) {
22                 //cprintf("%s \n", p->name);
23                 p->numOfTicks++;
24                 StrideMin = p->stride;
25                 p1 = p;
26             }
27         }
28         p1->myCount++;
29         p1->stride += p1->myStride;
30         //cprintf("Process %s stride: %d\n", p1->name, p1->stride);
31         c->proc = p1;
32         switchvm(p1);
33         p1->state = RUNNING;
34
35         swtch(&(c->scheduler), p1->context);
36         switchkvm();
37
38         // Process is done running for now.
39         // It should have changed its p->state before coming back.
40         c->proc = 0;
41         release(&ptable.lock);
42     }
55 }
```

2-2 Testing the lottery and stride scheduling

In order to test the lottery and stride scheduling, I have created three test file named prog1.c, prog2.c, and prog3.c. In these test files we have two nested loops with different number of tickets. We can set the number of tickets using the setTicket().

In the following listing, I showed prog1.c with 30 ticket. The prog2.c and prog3.c are the same as prog1, but with 20 and 10 tickets respectively.

Listing 11: The prog1.c for testing the second phase of the lab

```
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int main(int argc, char* argv[])
6  {
7      setTicket(30);
8      int i, k;
9      const int loop = 43000;
10     for(i = 0; i < loop; i++)
11     {
12         asm("nop");
13         for(k=0; k < loop; k++) {
14             asm("nop");
15         }
16     }
17     exit();
18 }
```

We have to do the following changes in the make file to run these test files. All the three test files are in the program's directory. Also, I change the number of the CPUs from 2 to 1 in the make file.

Listing 12: Changing in the make file to add the three test programs

```
1  \_prog1\ // in UPROGS
2  \_prog2\ // in UPROGS
3  \_prog3\ // in UPROGS
4  \prog1.c\ // in EXTRA=\
5  \prog2.c\ // in EXTRA=\
6  \prog3.c\ // in EXTRA=\
```

The below figure shows the result of running prog1, prog2, and prog3 together. I run all the programs together using the `prog1&prog2&prog3` command. As it is clear from the figure, prog1 with 30 tickets has 23002 ticks, prog2 with 20 tickets has 11936 ticks, and prog3 with 10 tickets has 4476 ticks. The ratio for prog1, prog2, and prog3 is 0.58, 0.30, and 0.11. As we expected and it is clear from the figure, since prog1 has the biggest number of the ticket, it will finish faster, and its ratio is bigger. The prog3 has the lowest number of tickets and it will finish later, and its ratio is lowest.


```

$ prog1&prog2&prog3
Number of ticks for sh is: 5
Number of ticks for sh is: 35
Number of ticks for prog2 is: 23002
zombie!
Number of ticks for prog1 is: 15792
zombie!
Number of ticks for prog3 is: 13404

```

Figure 4: The results of running stride scheduling

Below figure shows the results of the lottery scheduling involving three clients prog1, prog2, and prog3 with 3:2:1 allocation ratio. As it is clear from the figure, the ideal allocation for lottery scheduling is a straight line with a constant slop, but we can see in the figure 5 that the chart is not straight with constant slop and the reason for that is algorithm inherent use of randomization.

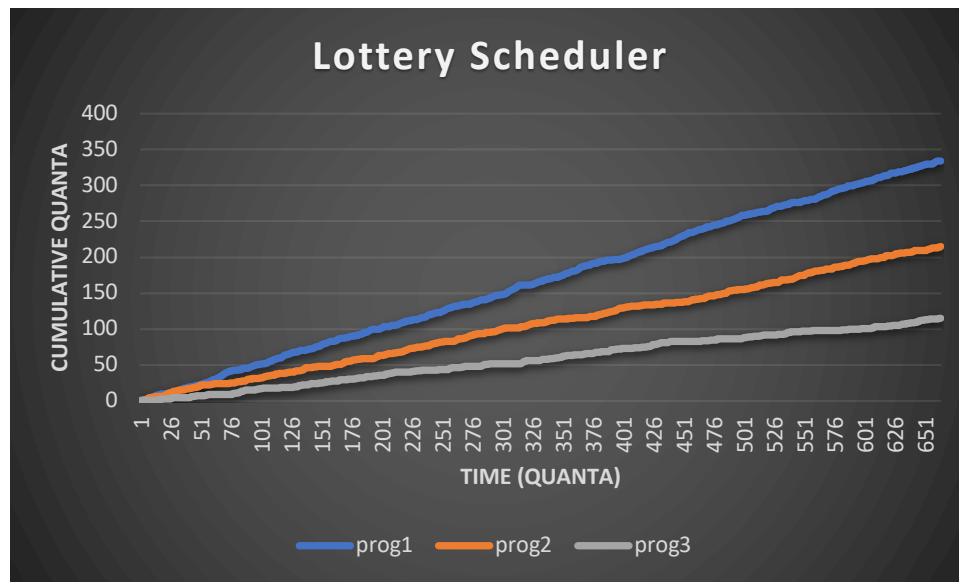


Figure 5: Results of the lottery scheduling involving three clients

Below figure shows the results of the stride scheduling involving three clients prog1, prog2, and prog3 with 3:2:1 allocation ratio. As it is clear from the figure, using stride scheduler we have precise periodic behavior. So, the stride scheduler solve the randomization problem of the lottery scheduler.

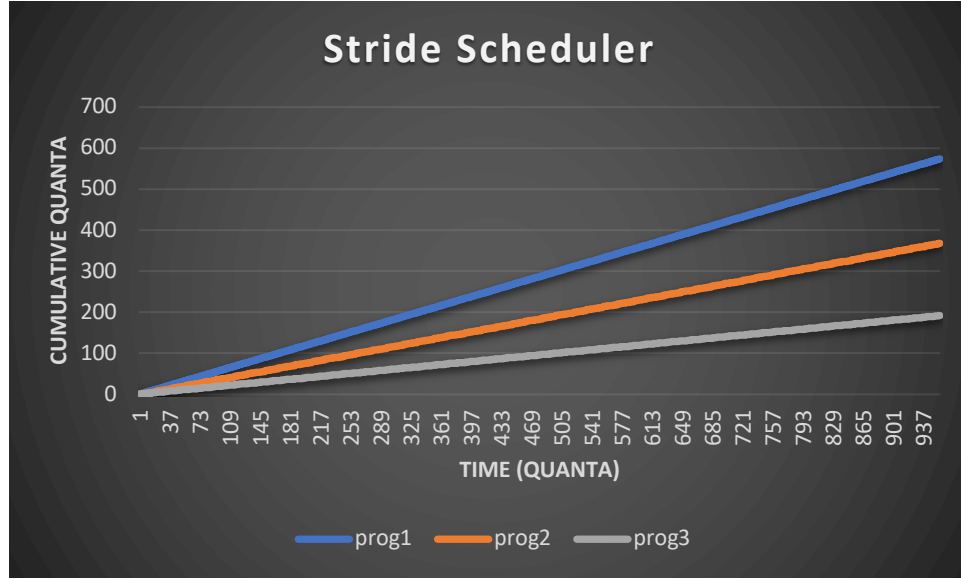


Figure 6: Results of the stride scheduler for prog1, prog2, and prog3

In figure 7 and 8, I showed the results for lottery and stride scheduler error for client with 3:2 ratio. The error will increase slowly in lottery scheduler when we increase the number of allocations. For stride scheduling, the error never exceed a single quantum and it will follow a deterministic pattern. The error in stride scheduling will drop to zero at the end of each period.

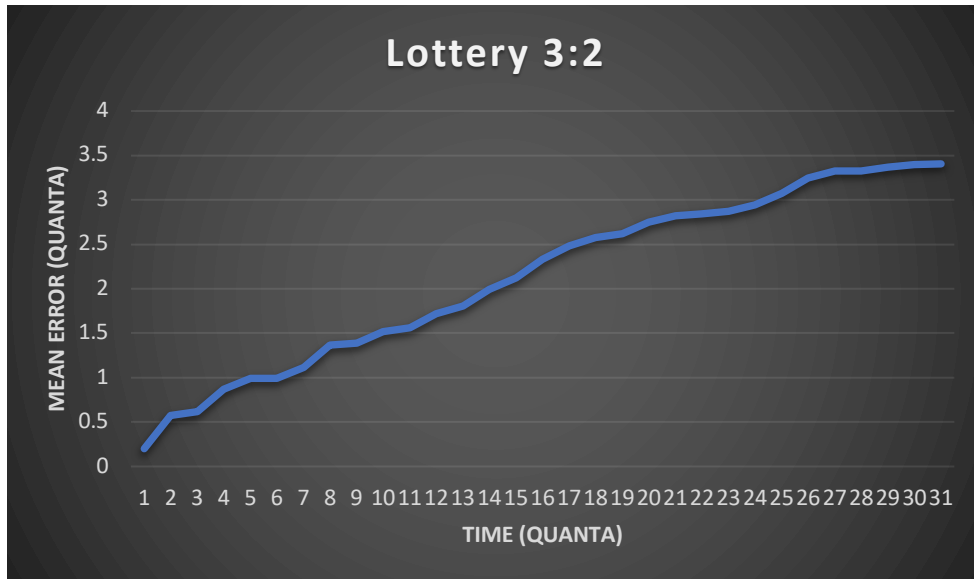


Figure 7: Throughput accuracy for lottery scheduling for running two clients with 3:2 ratio

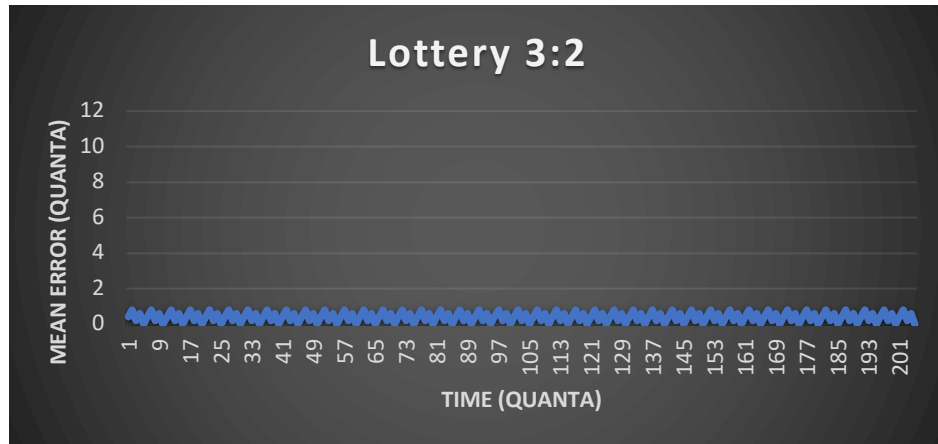


Figure 8: Throughput accuracy for stride scheduling for running two clients with 3:2 ratio