

# Part1

Christina Pavlopoulou  
cpavl001@ucr.edu

Andres Calderon  
acald013@ucr.edu

Niloufar Hosseini Pour  
nhoss003@ucr.edu

January 30, 2016

## 1 What files we changed to solve each part?

we changed the following files:

- 1- proc.h
- 2- proc.c
- 3- syscall.h
- 4- syscall.c
- 5- sysproc.c 6- user.h
- 7- defs.h
- 8- usys.S
- 9- Makefile

- 1- In proc.h file we added an integer variable inside the proc structure.

```
struct proc {  
  
    int counter;  
    .  
    .  
    .  
}
```

- 2- In proc.c file, inside the fork() function, after allocating process, we initialized counter variable of proc structure to zero.

```
int fork(void){  
  
    struct proc *np;  
    \ \ Allocate process  
    if((np = allocproc()) == 0)  
        return -1;
```

```

        np -> counter=0;
    }

```

3- In syscall.h file, we define the position of the system call vector that connect to our implementation.

```

#define SYS_ counter 22

```

4- In syscall.c file, we define externally the function that connect the shell and the kernel, use the position defined in syscall.h to add the function to the system call vector.

```

extern int sys_ counter(void);

static int (*syscalls[])(void) = {
    .
    .
    .
    [SYS_ counter] sys_ counter, };

void
syscall(void)
{
    sys_ counter();
}

```

5- In sysproc.c, we added the real implementation of our system call method.

```

int
sys_ counter(void){
    proc-> counter++;
    return proc->counter;
}

```

6- In user.h file, we defined the function that can be called through the shell. Our system call function prototype.

```

int counter(void);

```

7- In defs.h file, we added a forward declaration for our new system call under proc.c section.

\\ proc.c

```

int counter(void);

```

8- In USYS.S, we used the macro to connect the call of user to the system call function.

```

SYSCALL(counter)

```

9- In Makefile file, we told make how to compile and link the program. Under UPROGS=\ section we added our program.

```
UPROGS=\
    - count\
```

## 2 What files we added?

We added count.c file as follows:

```
#include "types.h"
#include "user.h"
#include "syscall.h"

int main()
{
    printf(1, "my system call %d \n", counter());
    return 0;
}
```

## 3 Output

For compiling our program and seeing the output we opened two terminals, in the first terminal we wrote: qemu-nox-gdb

In the second terminal we wrote gdb -q -iex "set auto-load safe-path /home/csgrads/nhoss003/xv6/" then we entered continue

then we came back to the first terminal and wrote the name of our program without .c extension.

Our output is:

```
my system call 4
```

Here is a screen shot of our output:

```

nhoss003@sledge:~/xv6
[nhoss003@sledge xv6]$ make qemu-nox-gdb
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
512000 bytes (5.1 MB) copied, 0.031415 s, 163 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000179552 s, 2.9 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
273+1 records in
273+1 records out
139804 bytes (140 kB) copied, 0.000810136 s, 173 MB/s
*** Now run 'gdb'.
qemu -nographic -hdb fs.img xv6.img -smp 2 -m 512 -S -gdb tcp::25052
Could not open option rom 'sgabios.bin': No such file or directory
xv6...
cpu1: starting
cpu0: starting
init: starting sh
$ coun
my system call 4
pid 3 coun: trap 14 err 5 on cpu 1 eip 0xffffffff addr 0xffffffff--kill proc
$ 

```

Figure 1: First Terminal Results

```

nhoss003@sledge:~/xv6
echo.d      ioapic.o      mp.d        string.d    wc.asm
echo.o      kalloc.c      mp.h        string.o    wc.c
echo.sym    kalloc.d      mp.o        swtch.o     wc.d
elf.h       kalloc.o      Notes       swtch.S     wc.o
entry.o     kbd.c         param.h     symlink.patch wc.sym
entryother kbd.d         picirq.c    syscall.c   x86.h
entryother.asm kbd.h        picirq.d    syscall.d   xv6.img
entryother.d kbd.o         picirq.o    syscall.h   _zombie
entryother.o kernel        pipe.c      syscall.o   zombie.asm
entryother.S kernel.asm    pipe.d      sysfile.c   zombie.c
entry.S      kernel.ld     pipe.o      sysfile.d   zombie.d
exec.c       kernel.sym    printf.c    sysfile.o   zombie.o
exec.d       _kill        printf.d    sysproc.c   zombie.sym
exec.o       kill.asm     printf.o    sysproc.d
fcntl.h      kill.c       printpcs    sysproc.o
file.c       kill.d       proc.c      timer.c
[nhoss003@sledge xv6]$ gdb -q -iex "set auto-load safe-path /home/csgnads/nhoss003/xv6/"
+ target remote localhost:25052
The target architecture is assumed to be i8086
[f000:fff0] 0xfffff0: ljmp $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb) c
Continuing.
Remote connection closed
(gdb) 

```

Figure 2: Second Terminal

## 4 Part 2

### 4.1 Lottery Scheduler

In the second part of the lab we had to implement the lottery scheduler. In the lottery scheduler, the user assigns to each process a number of tickets. We decided to assign one ticket by default to each process. For a process to win the lottery, it should have the ticket that the lottery generates. We generate the winning ticket using a random number generator function. To find the winner, we iterate through the tickets of all the processes until we find the winning ticket.

Now that we described the procedure of the lottery scheduler, we need to explain which parts of the code we changed to implement it. Firstly, we assigned the default numbers of tickets (to one) in the function `allocproc` of the `proc.c` file. All the other changes were implemented in the function `scheduler` of the `proc.c` file. At first, we iterated through all the processes, to find the total of all the tickets (`getTotalTickets` function in the `proc.c`). Then, we implemented a second iteration in which we look for the winner process. We find the number of tickets accumulated till the process that we currently are and we check if this number is greater or equal to the random number that the random number generator has produced (`randomGen` function in the `proc.c`). If this is true, then we found the winner process, otherwise we continue our search.

### 4.2 Ticket Inflation

In this part of the assignment we implemented the ticket inflation. In the ticket inflation, if the user notices that a process that she needs does not win as much as she wants, she could increase the number of its tickets. We do that by adding a system call which increases the number of tickets of the process that invokes it.

Our system call is called `sys_numtickets` and it is implemented in the `sysproc.c` file. We increase by one the number of tickets that the process already has and we return its current ticket numbers. We also had to make this system call visible to the `syscall.c` file. Then, we added it in the `syscall.h`, `user.h`, `defs.h` and `usys.S` files.

## 5 Tests

We design two tests to evaluate the validity of our implementation. The main idea is to run processes which perform long loops, allocate different amount of ticket to each of them and output their performance in fixed intervals of time. We implement a number of system calls in the `sysproc.c` file to achieve these goals.

### 5.1 System calls

A first system call (`top`) was implemented to run the `procdump` function every 5 seconds. We modify the default `procdump` function (at the end of `proc.c`) to output information about the running and runnable processes. Specifically, it shows the pid, name, tickets and number of lottery wins of each of the process. This system call receives as parameters the number of requested outputs and its number of tickets. We use the first parameter to set how many samples we want to collect during the test.

A second system call (`secs`) will return the instant in which this system call was invoked. It uses the `cmostime` function and structs provide by the `date.h` and `lapic.c` files. For convenience, it just returns the number of seconds during the current hour.

A third system call (`work`) will receive two parameters: a number of iterations for the loop and a number of tickets to be assigned for the current process. It just will perform a simple assignment during the number of specified iterations.

### 5.2 Test 1. Ticket allocation and fairness share

The first test evaluates the ratio of the ticket allocation. We implement a command line function (file `test.c`) that runs the system call `work`. Three processes were launched with different number of tickets running at the same time (using the `&` wildcard). The selected ratio was 8:3:1. Figure 3 shows the details of the setting for this test. Figure 4 shows the number of lotteries each process wins during a 200 seconds test.

In order to test the fairness share of the allocation, we plot the proportion of lottery wins for each process. For a ratio of 8:3:1 the expected proportions are 0.666, 0.250 and 0.083. Figure 5 illustrates the results. We can see that at the beginning of the test the proportions suffer some fluctuations but, as the number of lotteries increase, the proportions remains constant around the expected values.

```

and@and-Aspire-4520: ~/Documents/PhD/AOS/v2/xv6-public
and@and-Aspire-4520: ~/Documents/PhD/AOS/v2/xv6-public 80x24
qemu-system-i386 -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512
xv6...
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
init: starting sh
$ test 1000000000 1 &
$ test 1000000000 3 &
$ top 40 8
7-top_8,8,4-test_1,825,6-test_3,427,2923
7-top_8,321,4-test_1,853,6-test_3,534,2928
7-top_8,646,4-test_1,909,6-test_3,648,2933
7-top_8,975,4-test_1,936,6-test_3,785,2938
7-top_8,1300,4-test_1,981,6-test_3,909,2943
7-top_8,1603,4-test_1,1029,6-test_3,1042,2948
7-top_8,1916,4-test_1,1067,6-test_3,1174,2953
7-top_8,2223,4-test_1,1099,6-test_3,1310,2958
7-top_8,2544,4-test_1,1142,6-test_3,1437,2963
7-top_8,2867,4-test_1,1187,6-test_3,1545,2968
7-top_8,3205,4-test_1,1228,6-test_3,1657,2973
7-top_8,3537,4-test_1,1263,6-test_3,1782,2978
7-top_8,3877,4-test_1,1301,6-test_3,1894,2983
7-top_8,4212,4-test_1,1338,6-test_3,2006,2988
7-top_8,4535,4-test_1,1373,6-test_3,2127,2993

```

Figure 3: Details of the settings for test 1.

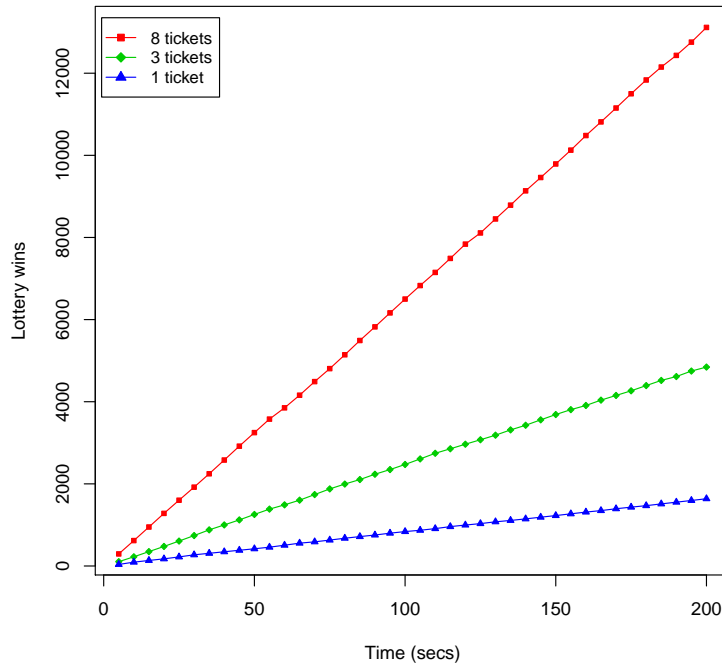


Figure 4: Lottery wins for three different processes. The proportional ration was 8:3:1.

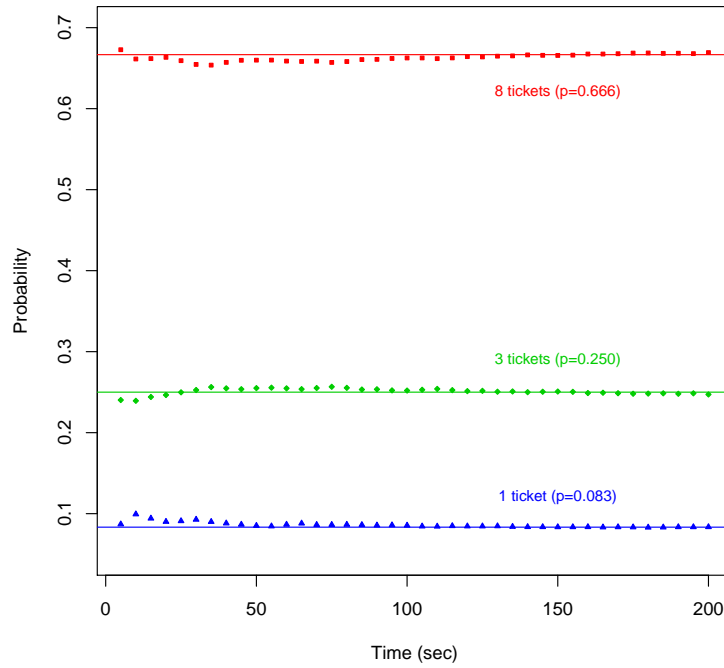


Figure 5: Lottery wins for three different processes. The proportional ratio was 8:3:1.

### 5.3 Test 2. Ticket inflation

To test ticket inflation we implement a command line function (file test2.c) which runs a long loop. We track the number of iterations inside the loop and call the numtickets system call (section XX) every 1000000 iterations to increase its number of tickets. Then, we launch two process: test2 (starting with 2 tickets) and top, which remains with a fixed number of tickets. Figure 6 shows the settings for this test.

Figure 7 illustrate the number of lottery wins for the two processes. We can see that the process with ticket inflation increases faster its number of wins according to the increase in its number of tickets. The probability of the fixed-ticket process of winning lotteries start to decrease compared with the increase number of tickets of the inflation process.

```
and@and-Aspire-4520: ~/Documents/PhD/AOS/v2/xv6-public
and@and-Aspire-4520: ~/Documents/PhD/AOS/v2/xv6-public 80x24
qemu-system-i386 -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512
xv6...
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
init: starting sh
$ test2 2000000000 2 &
$ top 41 5
5-top_5,8,4-test2_2,348,2885
5-top_5,345,4-test2_2,507,2890
5-top_5,676,4-test2_3,676,2895
5-top_5,994,4-test2_3,856,2900
5-top_5,1310,4-test2_3,1040,2905
5-top_5,1586,4-test2_4,1262,2910
5-top_5,1857,4-test2_4,1492,2915
5-top_5,2138,4-test2_4,1710,2920
5-top_5,2415,4-test2_5,1933,2925
5-top_5,2640,4-test2_5,2205,2930
5-top_5,2893,4-test2_6,2451,2935
5-top_5,3132,4-test2_6,2713,2940
5-top_5,3361,4-test2_6,2980,2945
5-top_5,3555,4-test2_7,3286,2950
5-top_5,3766,4-test2_7,3571,2955
5-top_5,3978,4-test2_8,3846,2960
```

Figure 6: Details of the settings for test 2.

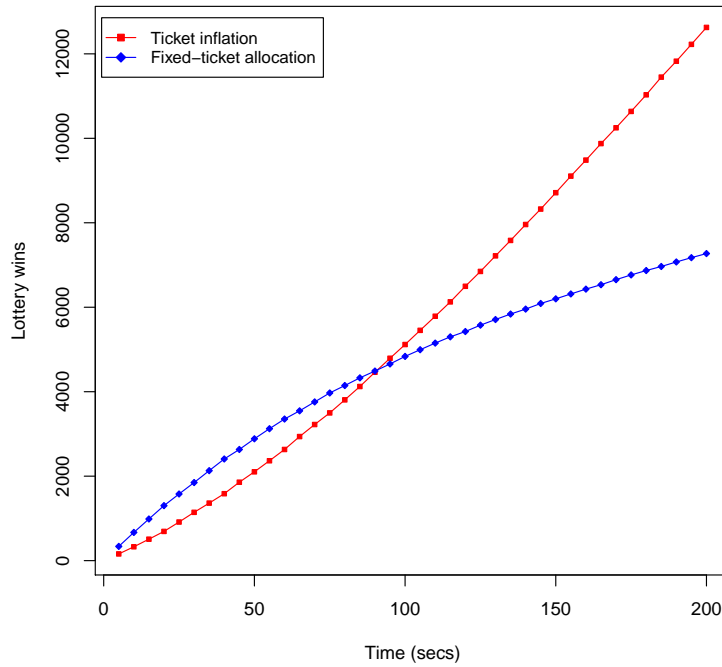


Figure 7: Response from two processes with different ticket allocation (Fixed vs Inflation).