Lab 1 Report

Christina Pavlopoulou   Niloufar Hosseini Pour   Andres Calderon

cpavl001@ucr.edu   nhoss003@ucr.edu   acald013@ucr.edu

February 1, 2016

1 Adding a system call

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`

In `proc.h` file we added an integer variable inside the *proc* structure.

```
1    struct proc {
2      int counter;
3      .
4      .
5      .
6    }
```

In `proc.c` file, inside the *fork* function, after allocating process, we initialized counter variable of *proc* structure to zero.

```
1    int fork(void){
2      struct proc *np;
3      if((np = allocproc()) == 0)
4        return -1;
5      np -> counter=0;
6    }
```

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

```
1    # define SYS_counter 22
```

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.

```
1    extern int sys_counter(void);
2    static int (*syscalls[])(void) = {
3       .
4       .
5       .
6    [SYS_counter] sys_counter, };
7
8    void syscall(void)
9    {
10     sys_counter();
11   }
```

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

```
1    int sys_counter(void){
2      proc-> counter++;
3      return proc->counter;
4    }
```

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

```
1        int counter(void);
```

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

```
1        int counter(void);
```

In `usys.S`, we used the macro to connect the call of user to the system call function.

```
1      SYSCALL(counter)
```

In `Makefile` file, we link the program. Under *UPROGS=* section we added our program.

```
1    UPROGS=\
2     _count\
```

Finally, we added `count.c` file as follows:

```
1    #include "types.h"
2    #include "user.h"
3    #include "syscall.h"
4
5    int main()
6      printf(1,"my system call %d \n", counter());
7      return 0;
8    }
```

Our output is:

```
1    my system call 4
```

Figure 1 shows our output. We can see that our process outputs 4 as the number of its system calls.

2 Lottery Scheduler and ticket inflation

2.1 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.

Figure 1: First Terminal Results.

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 modify the *proc* structure in the `proc.h` file to add two new variables: *tickets* (the number of tickets) and *wins* (the number of lottery wins). Then, 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. Every time a process wins a lottery, we increment its value of *wins* by 1.

2.2 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 as it is described in section 1.

3 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. In addition, we modify the number of CPUs in the `Makefile` to work with just one CPU.

3.1 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 *rtcdate* structure provide by the `date.h` and `lapic.c` files. For convenience, it just returns the number of seconds during the current hour.

3

Figure 2: Details of the settings for test 1.

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 will just perform a simple assignation during the number of specified iterations.

3.2 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 2 shows the details of the setting for this test. Figure 3 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 4 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.

3.3 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 2.2) 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 5 shows the settings for this test.

Figure 6 illustrates the number of lottery wins for the two processes. We can see that the process with ticket inflation increases faster its number of wins proportionally to the increase of 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.
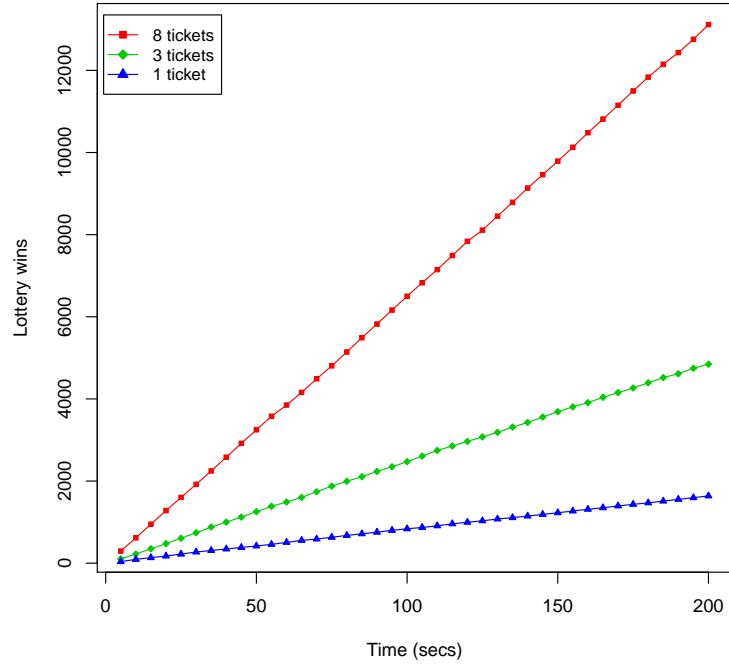
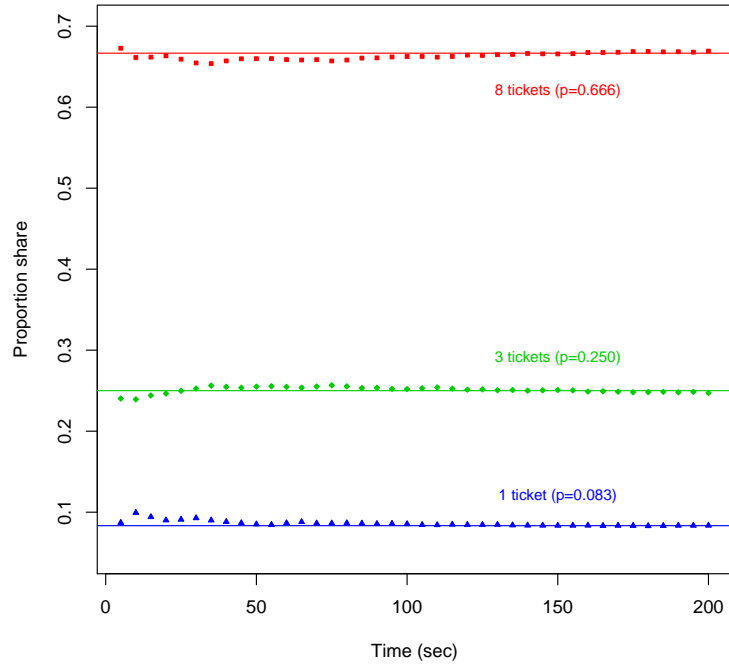Figure 3: Lottery wins for three different processes. The proportional ration was 8:3:1.

Figure 4: Proportion share for three different processes. The horizontal lines show the expected values.

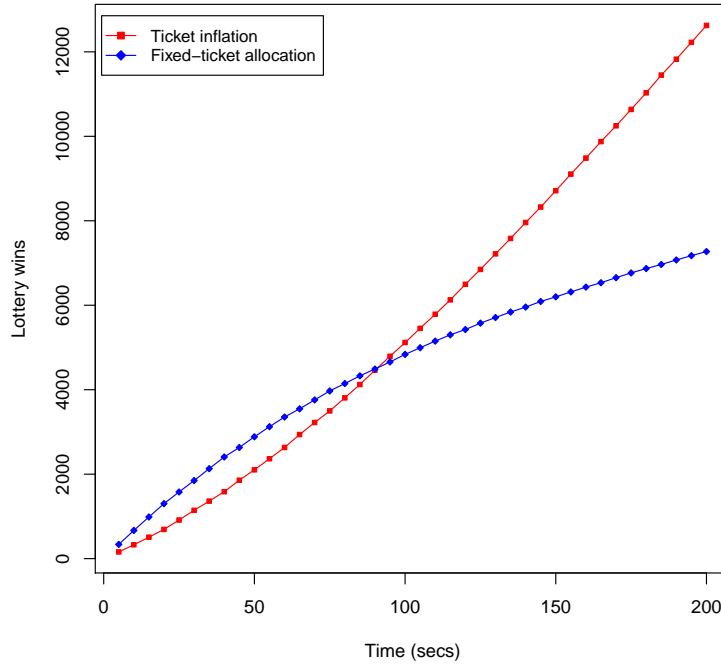Figure 5: Details of the settings for test 2.

Figure 6: Response from two processes with different ticket allocation (Fixed vs Inflation).