

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

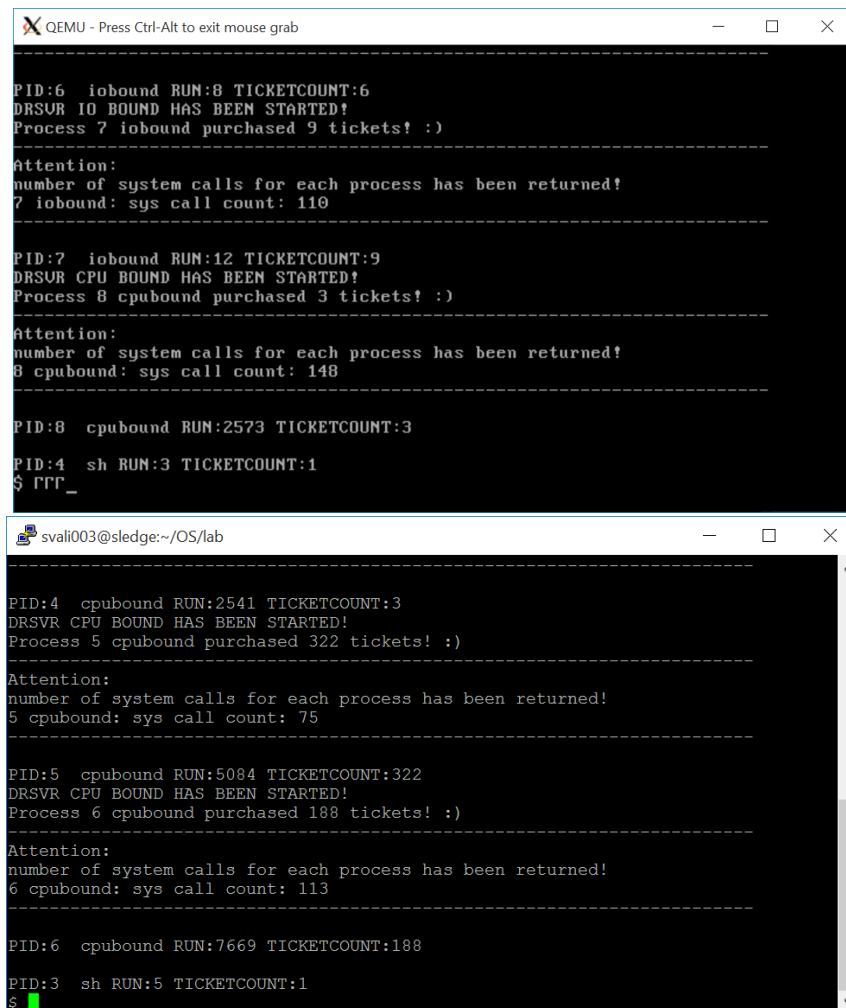
You can see the code above. You can also take a look at the generate random function that I write.

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
$ !TT_

-----
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.