

## Homework 3 – Lottery Scheduler

### Task 1: Implement Nice System Call

Updates made in the following system files

A `foo.c` file is created to create child process and test the nice code. The `foo.c` program takes the number of child processes to be created from the user through the command line. A simple check is done to see if the user has correctly given the input, if the number given is a negative number, then terminate the process and such.

Here, we use `fork` system call to create the required number of processes. To make the child process a CPU bound process, a calculation part is included in the code so that CPU utilization is forced to occur.

Now, we have to include the priority of the process in the process control block which is stored in **proc.h** under struct `proc`

`Int priority;`

In **proc.c** we have to make appropriate changes under `allocproc` function. CPU is allocated the process using this function. Here we set the default priority of any process that comes into memory as 20.

Next, we change the priority of the child process when it is created such that the priority of the child process is greater than the parent process. Thus we make these changes in **exec.c** where we change

`proc->priority = 15`

Now, to implement nice system call, we first need to update the table that is stored in system call interface in **syscall.h**. Here I have defined

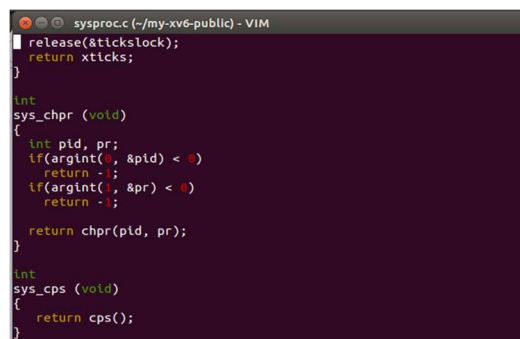
`SYS_chpr = 22`

Now we need to include the declaration of `chpr` in **defs.h** and **user.h** as

`Int chpr(int pid, int priority)`

Include the definition of the system call in **proc.c**. We write function definition of `chpr` in this file.

After this, we need to include the definition of `syschpr` in **sysproc.c** which will then call the `chpr` function.



```
sysproc.c (/my-xv6-public) - VIM
release(&tickslock);
return xticks;
}

Int
sys_chpr (void)
{
    Int pid, pr;
    if(argint(0, &pid) < 0)
        return -1;
    if(argint(1, &pr) < 0)
        return -1;
    return chpr(pid, pr);
}

Int
sys_cps (void)
{
    return cps();
}
```

Then we need to call the system call in **usys.S**, I added

```
SYSCALL (chpr)
```

Now, we open **syscall.c** and declare the function we defined in sysproc.c i.e

```
extern int sys_chpr(void)
```

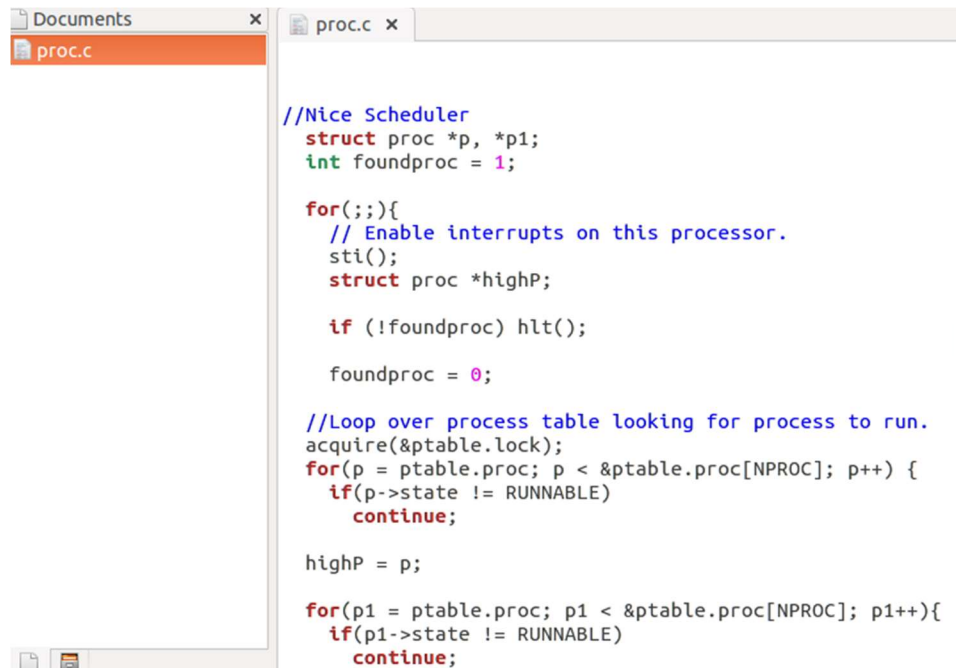
```
[SYS_chpr] sys_chpr,
```

Now, we create a new file nice.c which calls the chpr function. Here we take pid and priority from the command line and check if the user has given correct values for the same.

The scheduler function has been included in **proc.c**

Thus, this changes the scheduling algorithm of xv6 from round robin to priority scheduling.

Scheduler function for Nice:



```
//Nice Scheduler
struct proc *p, *p1;
int foundproc = 1;

for(;;){
    // Enable interrupts on this processor.
    sti();
    struct proc *highP;

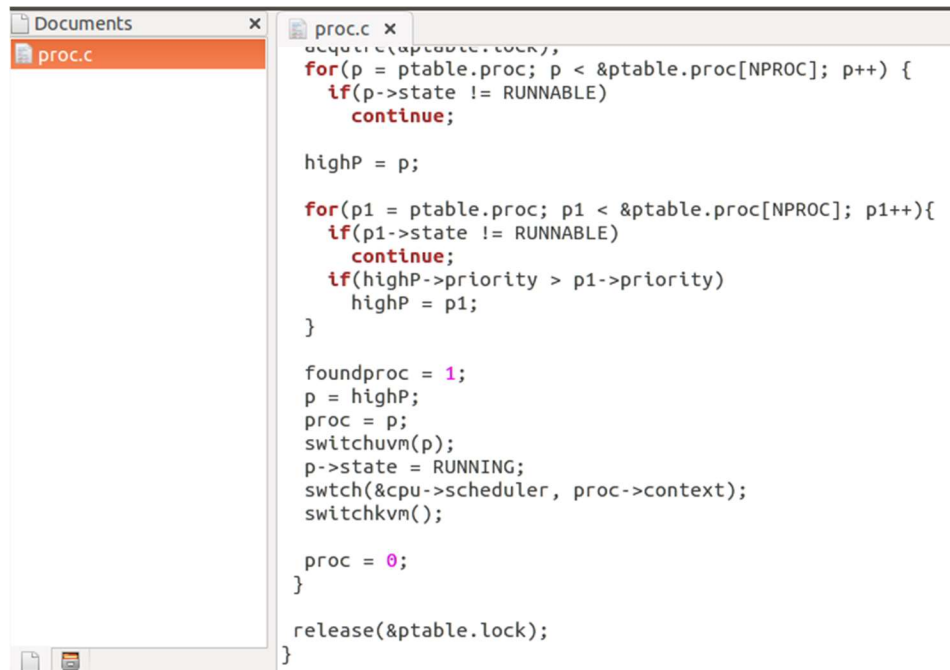
    if (!foundproc) hlt();

    foundproc = 0;

    //Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->state != RUNNABLE)
            continue;

        highP = p;

        for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
            if(p1->state != RUNNABLE)
                continue;
```



```
Documents x proc.c x
proc.c
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->state != RUNNABLE)
        continue;

    highP = p;

    for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
        if(p1->state != RUNNABLE)
            continue;
        if(highP->priority > p1->priority)
            highP = p1;
    }

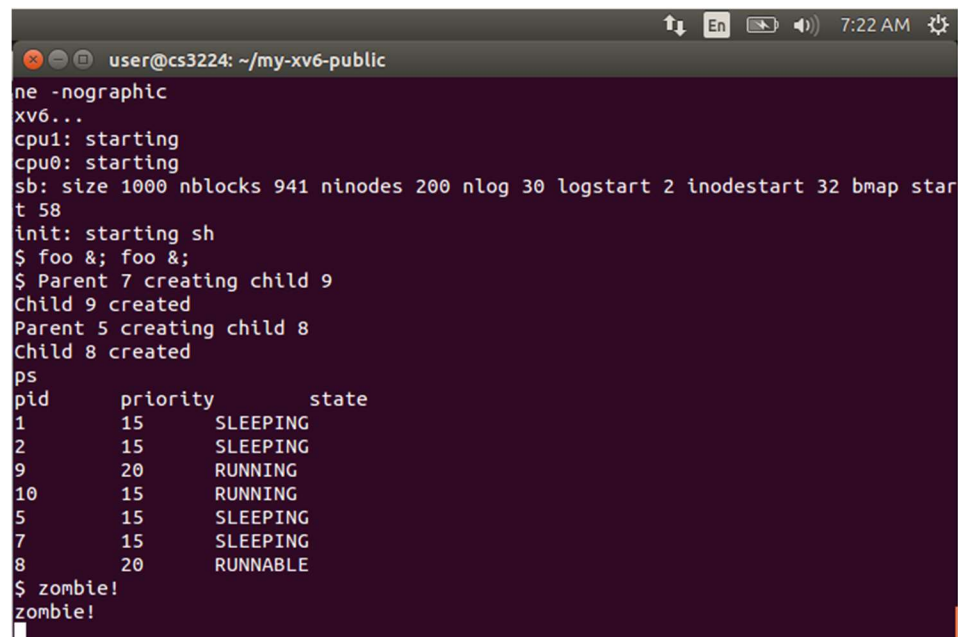
    foundproc = 1;
    p = highP;
    proc = p;
    switchvm(p);
    p->state = RUNNING;
    switch(&cpu->scheduler, proc->context);
    switchkvm();

    proc = 0;
}

release(&ptable.lock);
}
```

Testing Nice:

We run foo a couple of times where we fork child processes and we simultaneously run ps function to see the priority values and the states of the different processes.



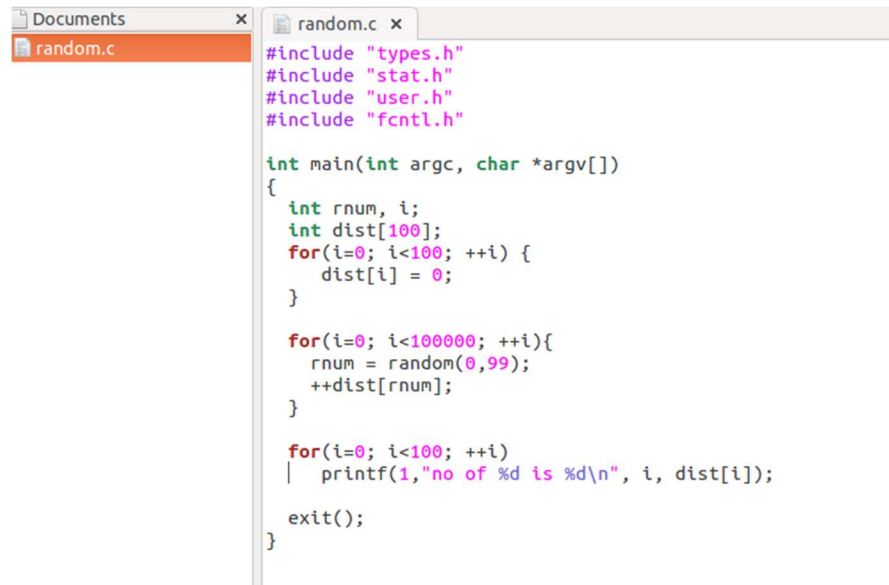
```
user@cs3224: ~/my-xv6-public
ne -nographic
xv6...
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ foo & foo &
$ Parent 7 creating child 9
Child 9 created
Parent 5 creating child 8
Child 8 created
ps
pid      priority  state
1         15      SLEEPING
2         15      SLEEPING
9         20      RUNNING
10        15      RUNNING
5         15      SLEEPING
7         15      SLEEPING
8         20      RUNNABLE
$ zombie!
zombie!
```

## Task 2: Random Number Generator

A 'random' system call is written in proc.c

A 32 bit LFSR generator is used to randomly generate numbers. A range is defined and then using this range a seed is generated.

A simple test case is written to randomly generate the numbers between 0-99 and observe the distribution.



```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

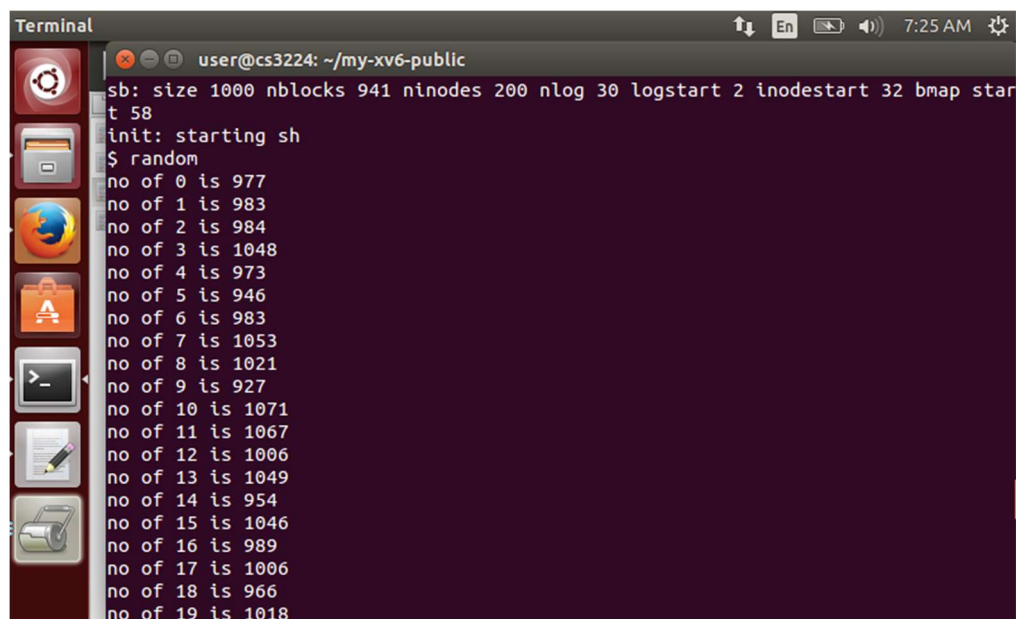
int main(int argc, char *argv[])
{
    int rnum, i;
    int dist[100];
    for(i=0; i<100; ++i) {
        dist[i] = 0;
    }

    for(i=0; i<100000; ++i){
        rnum = random(0,99);
        ++dist[rnum];
    }

    for(i=0; i<100; ++i)
        printf(1, "no of %d is %d\n", i, dist[i]);

    exit();
}
```

Output:



```
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ random
no of 0 is 977
no of 1 is 983
no of 2 is 984
no of 3 is 1048
no of 4 is 973
no of 5 is 946
no of 6 is 983
no of 7 is 1053
no of 8 is 1021
no of 9 is 927
no of 10 is 1071
no of 11 is 1067
no of 12 is 1006
no of 13 is 1049
no of 14 is 954
no of 15 is 1046
no of 16 is 989
no of 17 is 1006
no of 18 is 966
no of 19 is 1018
```

This can be called by 'random' in command prompt.

### **Task 3: Implementing lottery scheduler**

The following scheduler function has been added to implement the lottery scheduler in proc.c

```

random.c x  proc.c x
/Lottery Scheduler
struct proc *p;
int foundproc = 1;

long total_tickets = 0;
long counter = 0;
long winner = 0;

for(;;){
    // Enable interrupts on this processor.
    sti();

    if (!foundproc) hlt();

    foundproc = 0;

    acquire(&ptable.lock); // acquire the lock

    counter = 0;
    total_tickets = lottery();

    if(total_tickets <=0)
    {
        release(&ptable.lock);
        continue;
    }

    winner = random(1, total_tickets);

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

        if((counter + p->tickets) < winner)
        {
            counter = counter + p->tickets;
            continue;
        }
        foundproc = 1;
        proc = p;
        switchvm(p);
        p->state = RUNNING;
        swtch(&cpu->scheduler, proc->context);
        switchvm();

        proc = 0;
        break;
    }
    release(&ptable.lock);
}

```

- A function is defined to run through the process table to check the runnable processes and thus add the number of tickets accordingly

- A list is maintained to map the priority values to the number of tickets. More tickets meaning higher the priority of the process
- After counting the total number of tickets, if it is 0, lock is released, and loop runs again to find a runnable process
- Post this step, a random winner is generated, and corresponding process is identified
- The process table is the data structure used to iterate through runnable process. The leaving and entering of the processes is also maintained by the process table along with the state information.
- A sum of all the runnable process is maintained to match the winning number back to the runnable process

Testing the scheduler:

A simple test case is written to check the functioning. Even though lottery scheduler is random, it only logically makes sense that process with higher priority will get more tickets and thus in turn more CPU time to complete processing. Lotterytest1 is run for testing.

```

user@cs3224: ~/my-xv6-public
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.00017243 s, 3.0 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
372+1 records in
372+1 records out
190509 bytes (191 kB) copied, 0.000701728 s, 271 MB/s
qemu-system-i386 -serial mon:stdio -drive file=xv6.img,media=disk,index=0,format=
=raw -drive file=fs.img,index=1,media=disk,format=raw -smp 2 -m 512 -display no
ne -nographic
xv6...
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ lotterytest1
Start: Priority 40
parent waiting
Start: Priority 4
End: Priority 40
End: Priority 4
$

```

Another way to test would be to use foo test file to fork child processes and then try to change the priority of the processes using the nice function

```
user@cs3224: ~/my-xv6-public
init: starting sh
$ foo &; foo &;
Parent 5 creating child 7
Child 7 created
$ Parent 8 creating child 9
Child 9 created
ps
pid      priority    state
1         15          SLEEPING
2         15          SLEEPING
10        15          RUNNING
5         15          SLEEPING
7         20          RUNNING
8         15          SLEEPING
9         20          RUNNABLE
$ zombie!
zombie!
nice 2 1
$ ps
pid      priority    state
1         15          SLEEPING
2         1          SLEEPING
12        15          RUNNING
```

Changes made for system calls in the following files:

**syscall.h**

```
syscall.h (~my-xv6-public) - VIM
#define SYS_pipe      4
#define SYS_read      5
#define SYS_kill      6
#define SYS_exec      7
#define SYS_fstat     8
#define SYS_chdir     9
#define SYS_dup      10
#define SYS_getpid    11
#define SYS_sbrk     12
#define SYS_sleep    13
#define SYS_uptime   14
#define SYS_open     15
#define SYS_write    16
#define SYS_mknod    17
#define SYS_unlink   18
#define SYS_link     19
#define SYS_mkdir    20
#define SYS_close    21
#define SYS_chpr     22
#define SYS_gettime  23
#define SYS_settickets 24
#define SYS_cps      25
#define SYS_random   26
```

**defs.h**

```
//PAGEBREAK: 16
// proc.c
struct proc* copyproc(struct proc*);
void exit(void);
int fork(void);
int growproc(int);
int kill(int);
void pinit(void);
void procdump(void);
void scheduler(void) __attribute__((noreturn));
void sched(void);
void sleep(void*, struct spinlock*);
void userinit(void);
int wait(void);
void wakeup(void*);
void yield(void);
int chpr(int pid, int priority);
int cps(void);
int random(int l, int h);
```

user.h

```
user.h (~my-xv6-public) - VIM
int mknod(char*, short, short);
int unlink(char*);
int fstat(int fd, struct stat*);
int link(char*, char*);
int mkdir(char*);
int chdir(char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int chpr(int pid, int priority);
int cps(void);
int random(int l, int h);
int gettimeofday(struct rtcdate *);
int settickets(int);

// ulib.c
int stat(char*, struct stat*);
char* strcpy(char*, char*);
void *memmove(void*, void*, int);
char* strchr(const char*, char c);
int strcmp(const char*, const char*);
```

37,1 66%

usys.S

```
usys.S (~my-xv6-public) - VIM
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(chpr)
SYSCALL(cps)
SYSCALL(random)
SYSCALL(gettime)
SYSCALL(settickets)
```

36,1 Bot