Homework 3: Lottery Scheduler

Task 1: Implementing nice system call

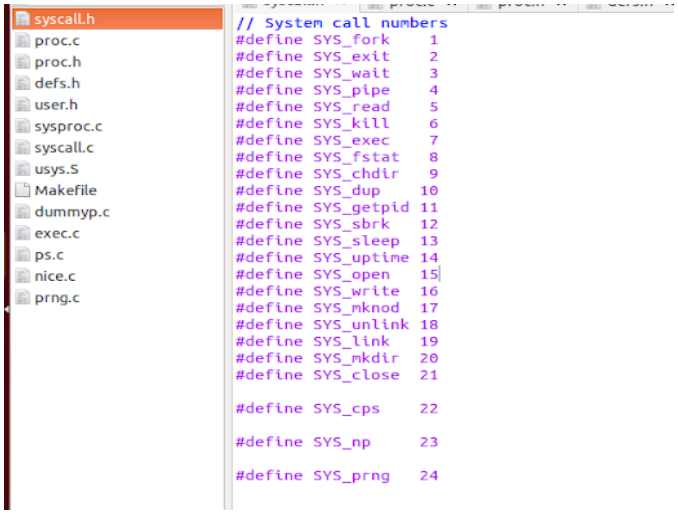
The goal of implementing the nice system call is to enable assigning and changing priorities dynamically to the processes. The way we have implemented it, we take in two arguments to the nice system call - the process id and the priority that we wish to assign to that particular process. The value that we are assigning to priorities i.e the “nice” value, ranges from 0-20, 0 being the highest priority and 20 being the lowest priority. Additionally, we will also create another system call ‘ps’ to get and print the list, state and priorities of the processes in the process table.

To implement this functionality as a system call, we need to make changes to a bunch of kernel files:

- Syscall.h
- Proc.h
- Defs.h
- Users.h
- Exec.h
- Proc.c
- Sysproc.c
- Usys.s
- Syscall.c

Syscall.h:

This is the system call interface that maintains the list of system calls and maps a number to it. This is a crucial step because inside the operating system, the system calls are invoked with a number, not a name. Here we have added two system calls - SYS_cps(printing process info) and SYS_np(nice system call) and have assigned them numbers 22 and 23.



Proc.h:

Next we declare the variable for priority in proc.h file.

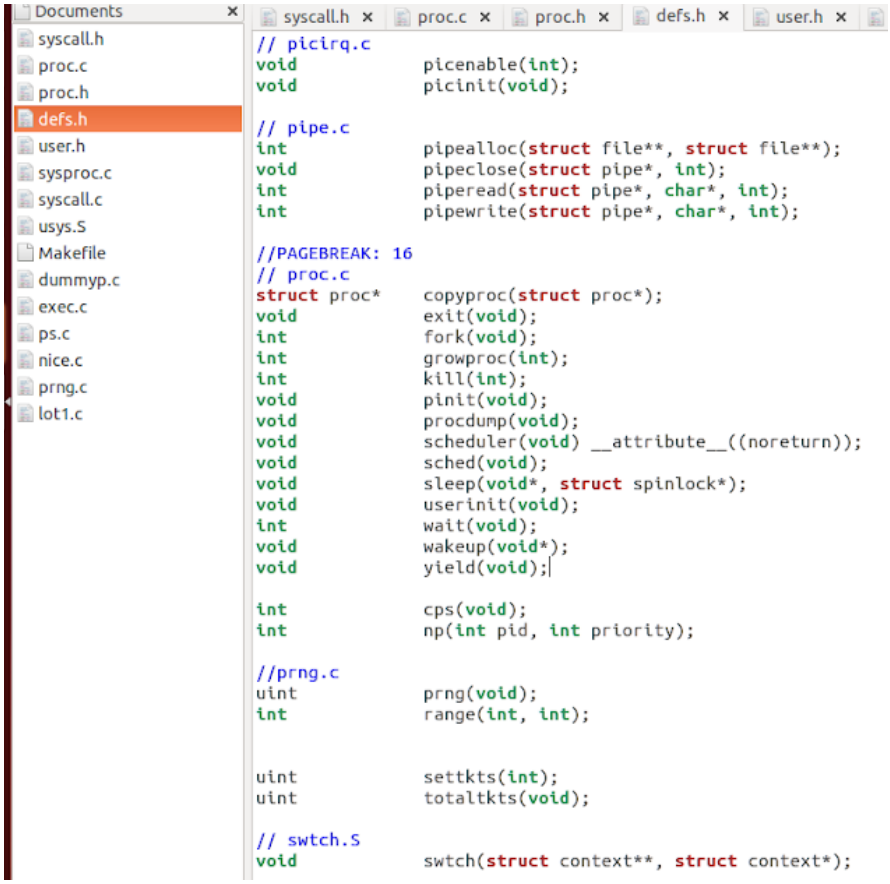
The function cps, does not do anything complicated. It acquires the lock for the table. It loops through the process table, sorts them according to the process states, and prints their process name, process id, priority and state.Finally, it releases the lock.

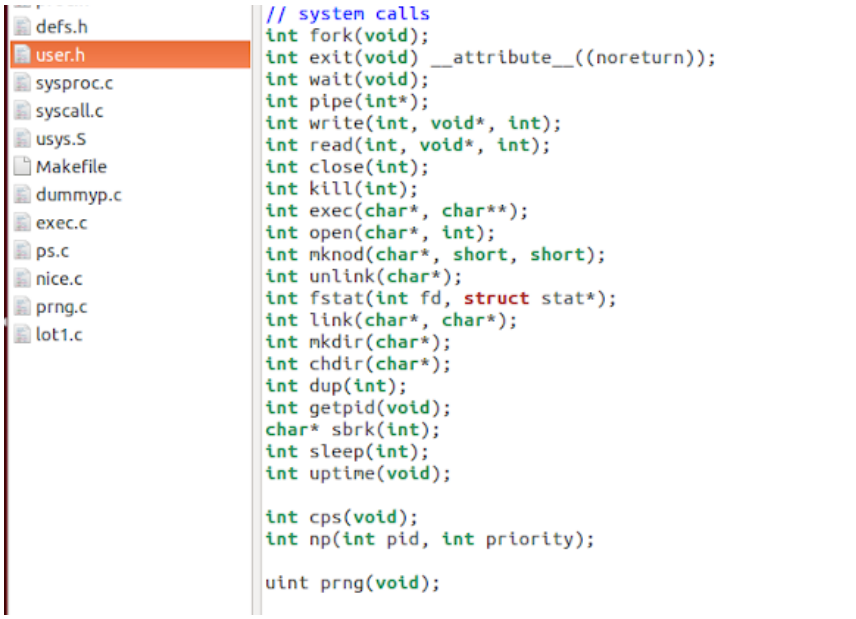
The function np is where the priority assignment to a process happens. The function acquires the lock. It gets two arguments: process id and the priority that is to be assigned. The function then loops through the process table to find the process with the specified process id. Then it assigns the passed value to that particular process's priority attribute. Finally, it releases the lock and returns the process id.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this proc
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

  int priority;

  int tickets;
};

// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap
```

Defs.h and Users.h:

User.h contains the function prototypes for the xv6 system calls and library functions. Defs.h is where function prototypes for kernel-wide function calls, that are not in sysproc.c or sysfile.c, are defined. In both of these places, we need to add our system call declarations for cps and np.





Exec.c

In this file, we simply assign the current process's value as 2. This is the default assignment for any time a current process is running, its default value is 2.

Proc.c:

In this file, we can find a structure called ‘proc’. This is the structure illustrating the process structure in XV6. If we need to assign every process a new attribute, we should create a new entry in this structure to be able to store the priority in here. Here, we added a line ‘int priority’ and this would apply for every process in the system.

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  //Modified
  p->priority = 10;
  //

  release(&ptable.lock);
```

Also we create 2 functions: “cps” and “np” in the proc.c file. Here the function “cps” is called from the kernel and it provides specific information about the processes. The function mainly allows us to know the process details like name of the process, process id (pid), state of the process and its priority. Firstly, the function acquires the lock for reading the process table. Then it displays the content of the table like name of processes, process ids, state and its priority. After that it releases the lock for process table (ptable) and returns all the details back to the kernel. Here the “return 22” represents returning the function to its callee which is ‘sys_cps’ system call and it has the system call number 22 which we assigned earlier in the sys_call.h file.

```
int
cps()
{
        struct proc *p;
        sti();
        acquire(&ptable.lock);
        cprintf("Name \t PID \t State \t Priority \n");
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
          if(p->state == SLEEPING)
                cprintf("%s \t %d \t SLEEPING \t %d\n ", p->name, p->pid, p->priority);
          else if(p->state == RUNNING)
                cprintf("%s \t %d \t RUNNING \t %d\n ", p->name, p->pid, p->priority);
          else if(p->state == RUNNABLE)
                cprintf("%s \t %d \t RUNNABLE \t %d\n ", p->name, p->pid, p->priority);

        }
        release(&ptable.lock);
        return 22;

}
```

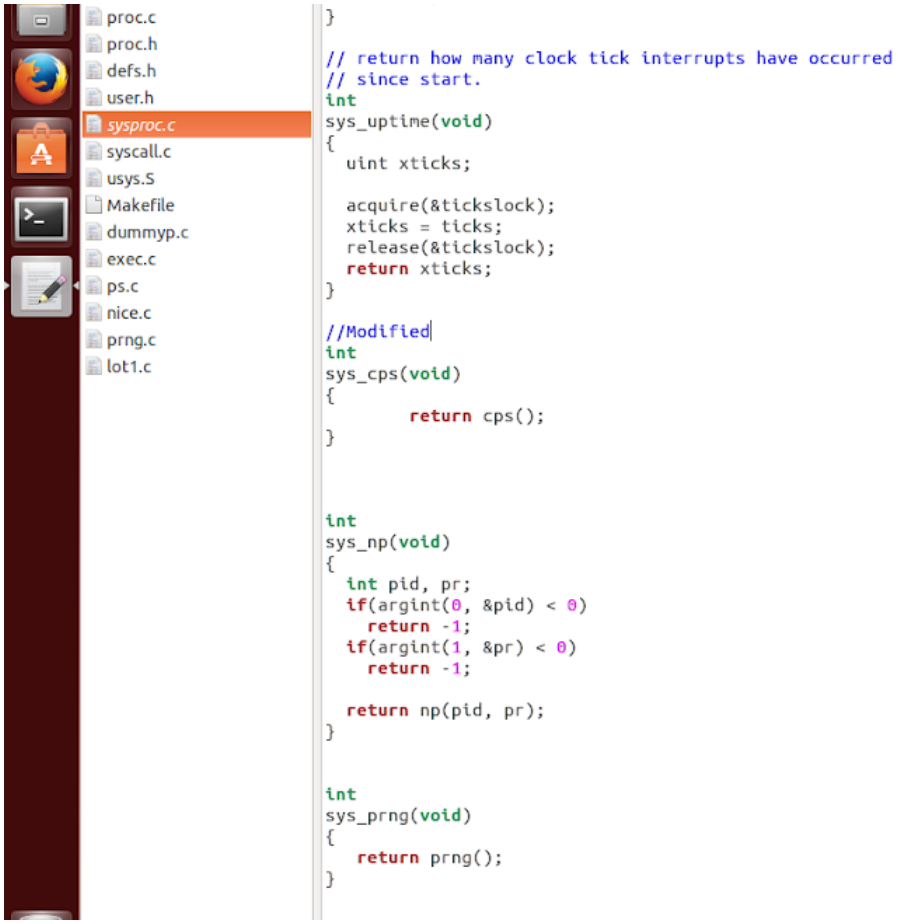
Next we create another function “np” where the priorities of the processes could be modified depending on their nice values. This is most essential as the kernel calls this function to modify

the priorities of the processes. The “np” function takes 2 arguments which is “pid” that is already defined and “priority” which we had defined in the proc.h file. In this function, we modify the priority of the process whose pid we have passed. The main objective of np function is to change the nice value and we achieve doing so by finding that particular process (pid) in the process table and modifying its priority to a given value. Next we return the modified process back.

```
int
np(int pid, int priority)
{
        struct proc *p;
        if(priority>20 || priority < 0){
                return -1;
        }

        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
          if(p->pid == pid){
                        p->priority = priority;
                        break;
                }
        }
        release(&ptable.lock);
        return pid;
}
```

Sysproc.c:

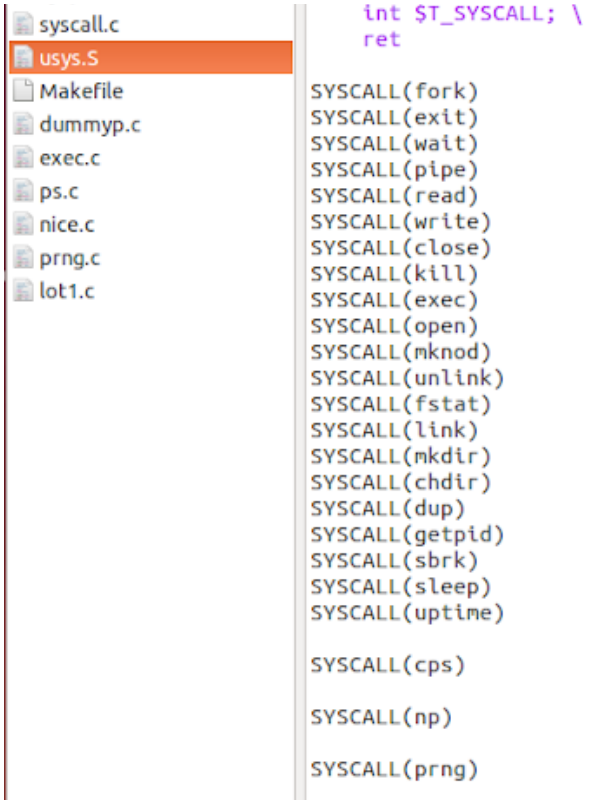
Next, in sysproc.c, we need to define our sys_cps and sys_np, from where our cps and np functions defined in proc.c will be called. In function sys_np, we just do a b=couple of sanity checks, like checking if the passed arguments are valid, and finally issuing a call to np().




```
}

// return how many clock tick interrupts have occurred
// since start.
int
sys_uptime(void)
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}

//Modified
int
sys_cps(void)
{
        return cps();
}



int
sys_np(void)
{
  int pid, pr;
  if(argint(0, &pid) < 0)
    return -1;
  if(argint(1, &pr) < 0)
    return -1;

  return np(pid, pr);
}


int
sys_prng(void)
{
   return prng();
}
```

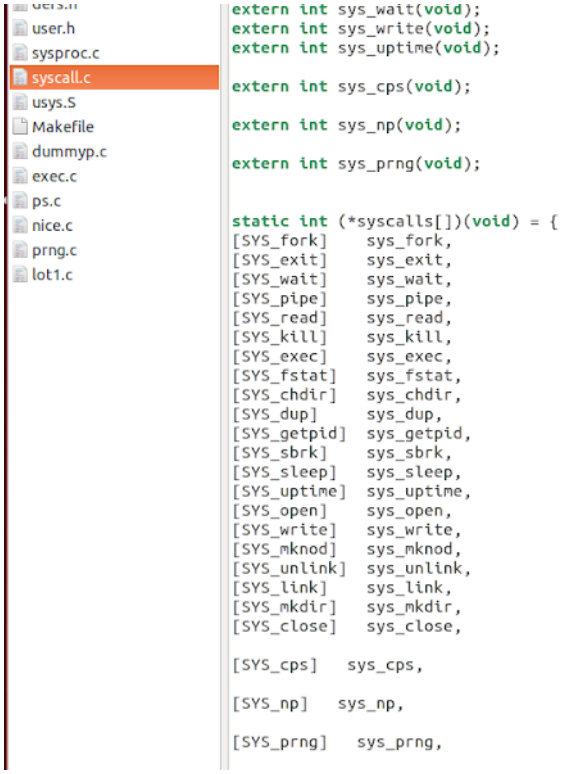
Usys.S:

This file contains some assembly level code(.S extension) to talk with the hardware. Here along with the list of other system calls, we should add np and cps system calls.



Syscall.c:

Here the entry point to the system call is defined. We add the two system calls along with the rest of the 'extern int' definitions. Then in the static int (*syscalls[])(void), we add the two system calls with a similar formatting.



Now we have added all that is necessary to define the system calls in kernel files, we need to create two C files, one for cps and one for nice, from where we would issue the system calls.

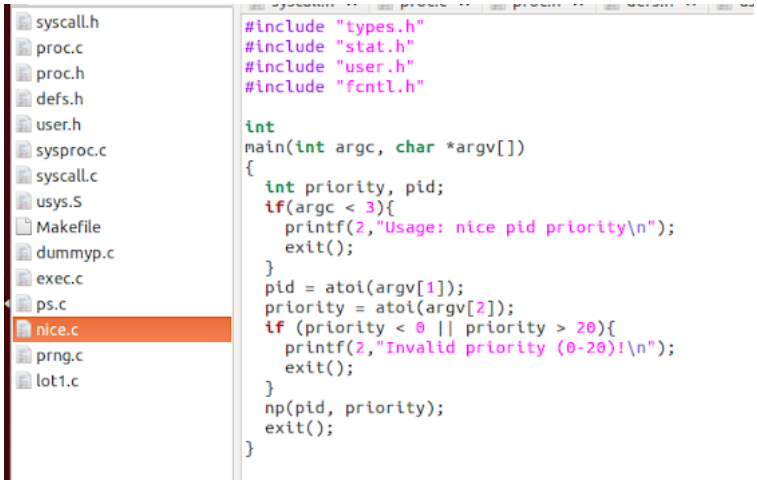
Ps.c:

This is a simple ps.c file, where we issue a function call to the cps() function.

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

int main(void){
  cps();
  exit();
}
```

Nice.c:

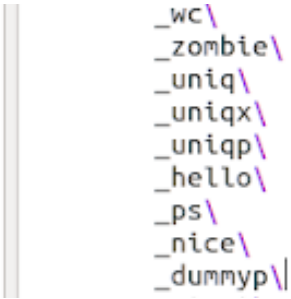
In this .c file, we read the command line arguments that are passed to the nice system call, i.e the process id and priority. We do a sanity check to ensure that the argument priority lies inside the range of 0-20. If yes, we call the np function with the two passed arguments. If the priority is invalid, we issue an error message.



Now, we have done everything to implement the nice system call. We have defined a testcase to test the working of nice in dummyp.c. In this testcase, we are forking the parent to create children. In the child process, we are doing some computations to use up the CPU time. In the parent, we are simply issuing a print command and making the parent wait. We are allowed to pass a value when calling this file. When no value is specified, the loop is executed once. With this we will be able to test the nice system call.

Makefile:

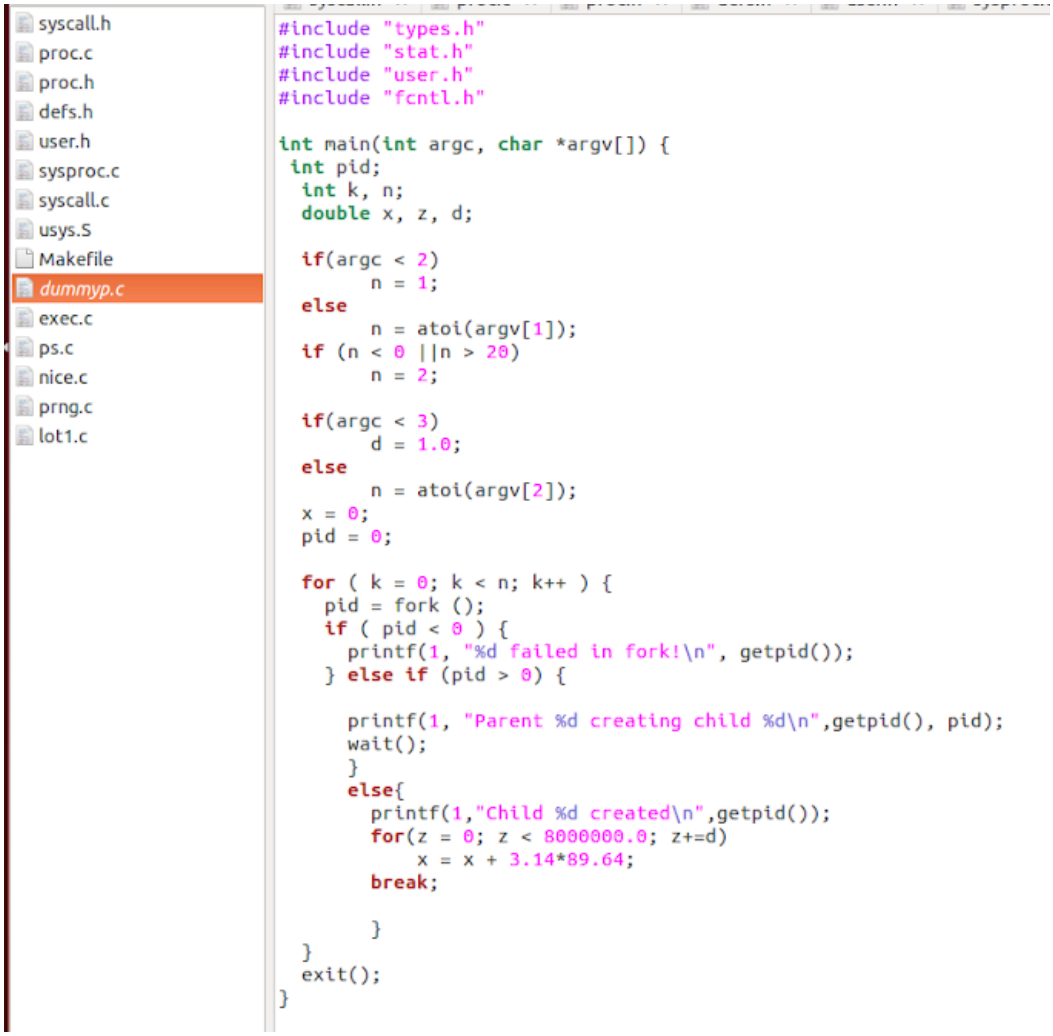
After this, we will have to add the all .c files to the makefile list before running it.



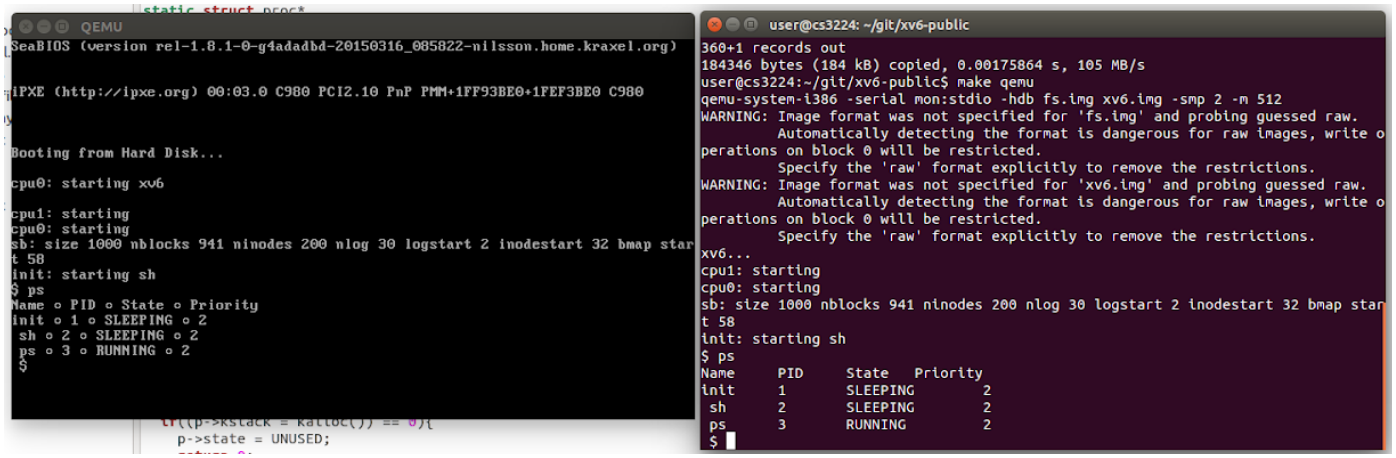
Testing the system call:

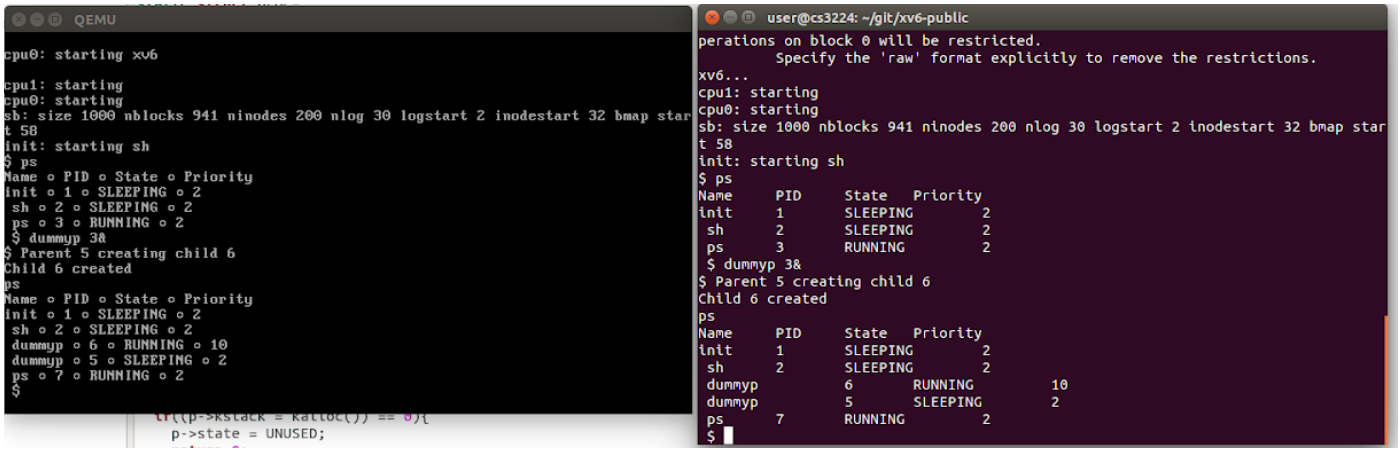
Once we are in QEMU, we can issue a call to the dummy program **(dummyp.c)** that we created. This will essentially create a process, cause it to fork and will create a child as well. We can create two instances of this program if we choose to. Now if we call ps.c file, it will print the list of processes, with their names, id and priorities.Usually, it will contain the init,sh and a few other processes that are running. We can see two entries for the dummy process, if we instantiated it once.One for the parent and one for the child. We can keep calling this function to

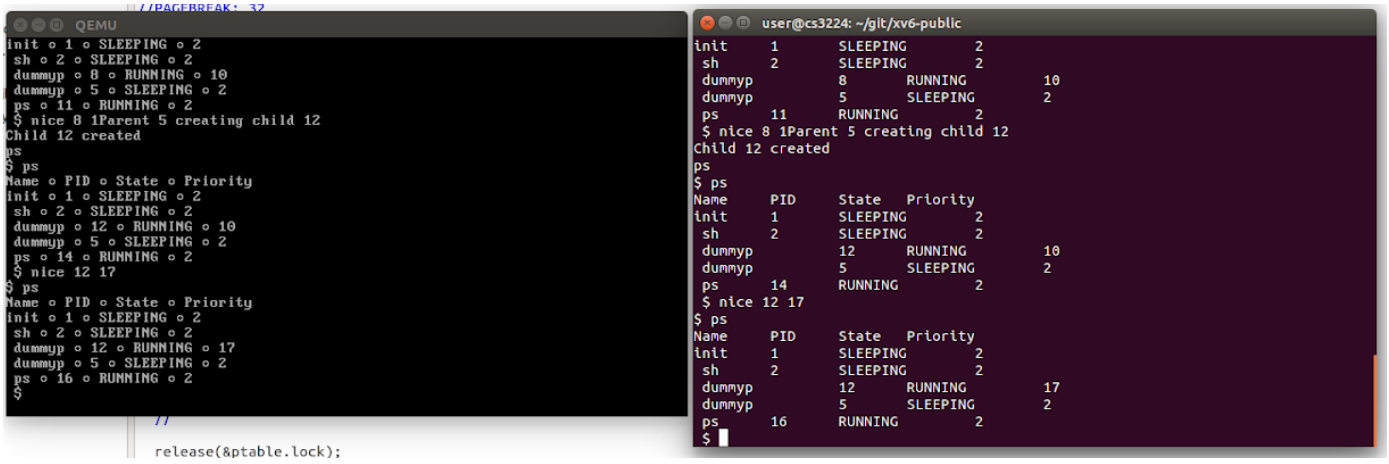
check and monitor the process states. Also, we can note that all the processes have a priority of 2. That is the default priority that we had assigned to all the processes. Now we can try to change the priority of a particular process by issuing the nice system call. By looking at the pid from the list printed, we can issue a nice system call with the pid and the priority that we have assigned. If we give an invalid value, an error message will be printed. Otherwise the priority of that particular process will be updated. We can notice this by calling the ps function again.



OUTPUT :


QEMU
SeaBIOS (version rel-1.8.1-0-g4adadbd-20150316_085822-nilsson.home.kraxel.org)
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF93BE0+1FEF3BE0 C980
Booting from Hard Disk...
cpu0: starting xv6
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ps
Name o PID o State o Priority
init o 1 o SLEEPING o 2
sh o 2 o SLEEPING o 2
ps o 3 o RUNNING o 2
$
user@cs3224: ~/git/xv6-public
360+1 records out
184346 bytes (184 kB) copied, 0.00175864 s, 105 MB/s
user@cs3224:~/git/xv6-public$ make qemu
qemu-system-i386 -serial mon:stdio -hdb fs.img xv6.img -smp 2 -m 512
WARNING: Image format was not specified for 'fs.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
WARNING: Image format was not specified for 'xv6.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
xv6...
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ps
Name PID State Priority
init 1 SLEEPING 2
sh 2 SLEEPING 2
ps 3 RUNNING 2
$


QEMU
cpu0: starting xv6
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ps
Name o PID o State o Priority
init o 1 o SLEEPING o 2
sh o 2 o SLEEPING o 2
ps o 3 o RUNNING o 2
$ dummyp 3&
$ Parent 5 creating child 6
Child 6 created
ps
Name o PID o State o Priority
init o 1 o SLEEPING o 2
sh o 2 o SLEEPING o 2
dummyp o 6 o RUNNING o 10
dummyp o 5 o SLEEPING o 2
ps o 7 o RUNNING o 2
$
user@cs3224: ~/git/xv6-public
operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
xv6...
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ps
Name PID State Priority
init 1 SLEEPING 2
sh 2 SLEEPING 2
ps 3 RUNNING 2
$ dummyp 3&
$ Parent 5 creating child 6
Child 6 created
ps
Name PID State Priority
init 1 SLEEPING 2
sh 2 SLEEPING 2
dummyp 6 RUNNING 10
dummyp 5 SLEEPING 2
ps 7 RUNNING 2
$


QEMU
init o 1 o SLEEPING o 2
sh o 2 o SLEEPING o 2
dummyp o 8 o RUNNING o 10
dummyp o 5 o SLEEPING o 2
ps o 11 o RUNNING o 2
$ nice 8 1Parent 5 creating child 12
Child 12 created
ps
$ ps
Name o PID o State o Priority
init o 1 o SLEEPING o 2
sh o 2 o SLEEPING o 2
dummyp o 12 o RUNNING o 10
dummyp o 5 o SLEEPING o 2
ps o 14 o RUNNING o 2
$ nice 12 17
$ ps
Name o PID o State o Priority
init o 1 o SLEEPING o 2
sh o 2 o SLEEPING o 2
dummyp o 12 o RUNNING o 17
dummyp o 5 o SLEEPING o 2
ps o 16 o RUNNING o 2
$
user@cs3224: ~/git/xv6-public
init 1 SLEEPING 2
sh 2 SLEEPING 2
dummyp 8 RUNNING 10
dummyp 5 SLEEPING 2
ps 11 RUNNING 2
$ nice 8 1Parent 5 creating child 12
Child 12 created
ps
$ ps
Name PID State Priority
init 1 SLEEPING 2
sh 2 SLEEPING 2
dummyp 12 RUNNING 10
dummyp 5 SLEEPING 2
ps 14 RUNNING 2
$ nice 12 17
$ ps
Name PID State Priority
init 1 SLEEPING 2
sh 2 SLEEPING 2
dummyp 12 RUNNING 17
dummyp 5 SLEEPING 2
ps 16 RUNNING 2
$

Task 2: Random Number Generator:

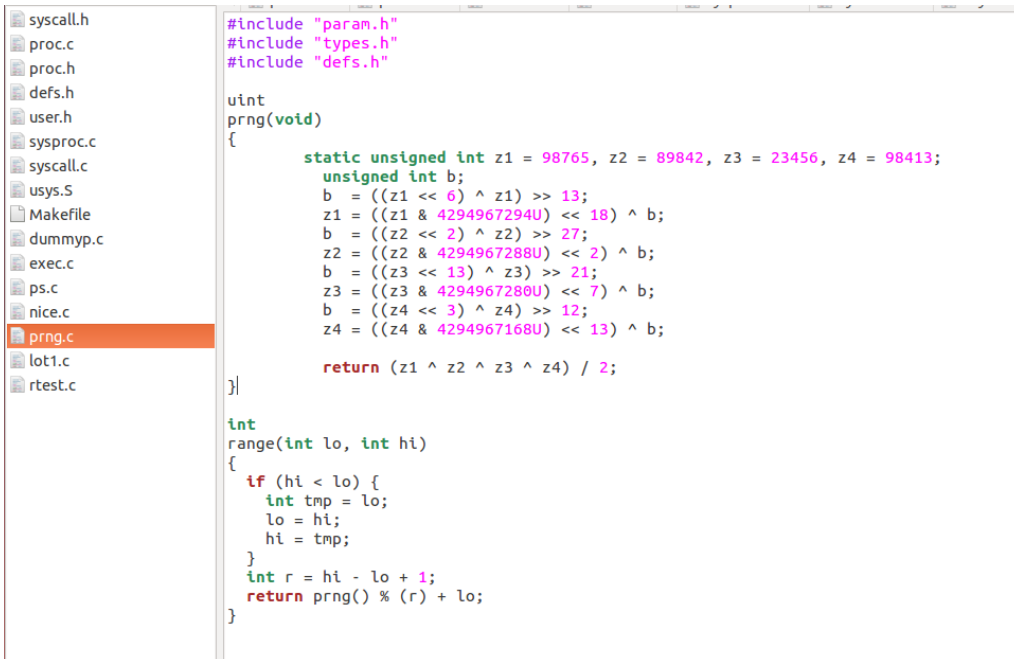
As suggested, we have used a XORshift random number generator. The reason we are implementing this pseudo random number generator is to generate a random ticket number in order to determine which process won and gets to run in the processor.

We are creating a random.c file that has two functions: random() and range(). The range function simply collects the range as arguments, checks it for valid values and adjusts the random number generated by the random() function according to the range.

The random() function takes, when called, generates a random number between range 0 - 2^32. We have given four non zero seed values for the XORshift random number generator to work. This is an already implemented algorithm by George Marsaglia and we have just adapted it to suit XV6 and C language.

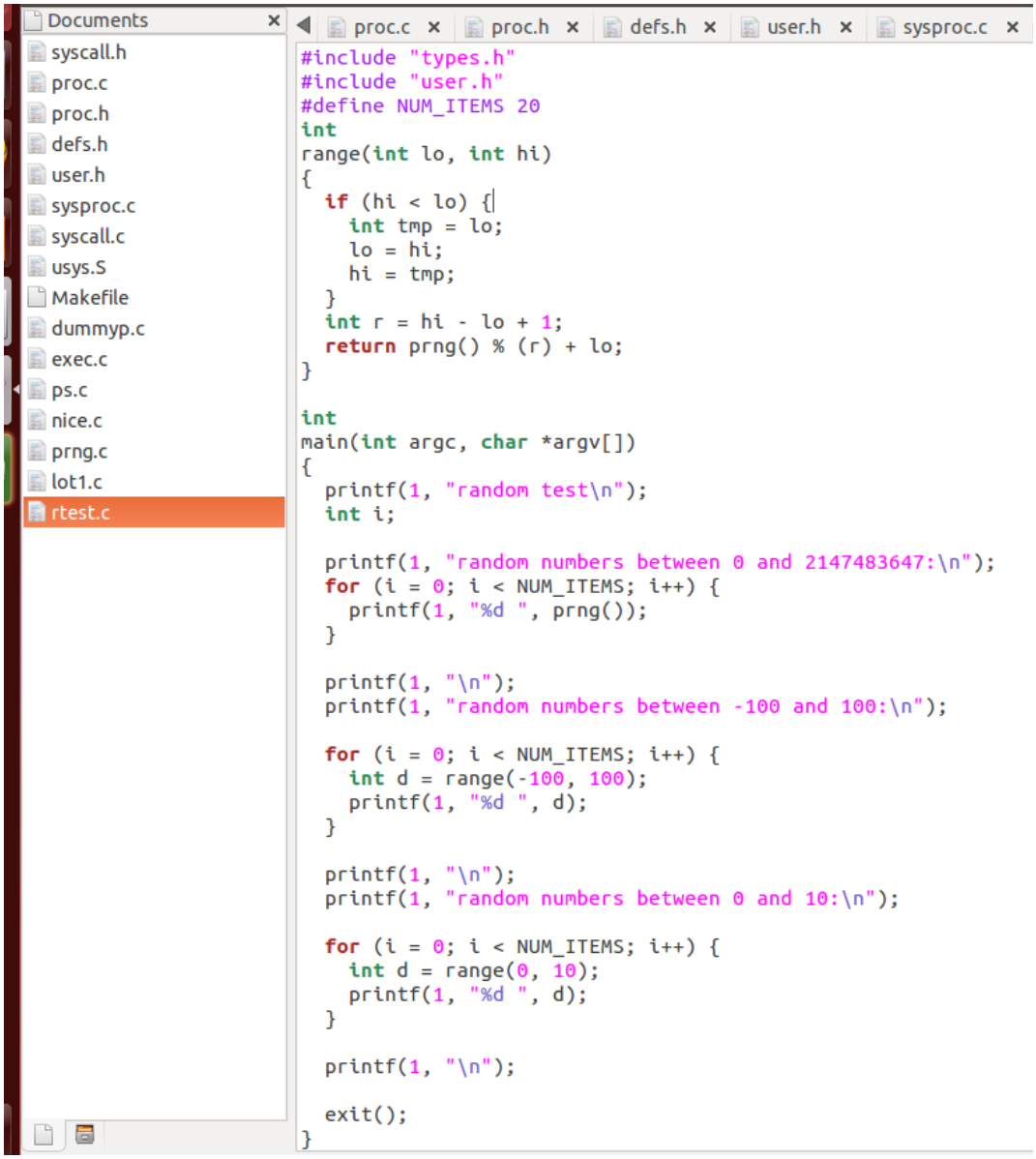
Random() as a system call:

We are adding random() as a system call, the same way we implemented the nice system call. We have added the respective declarations in all the kernel files.

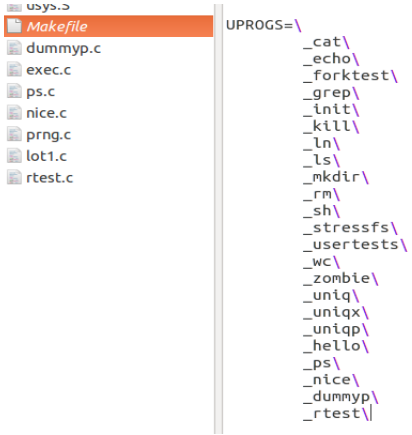


Testcase for PRNG:

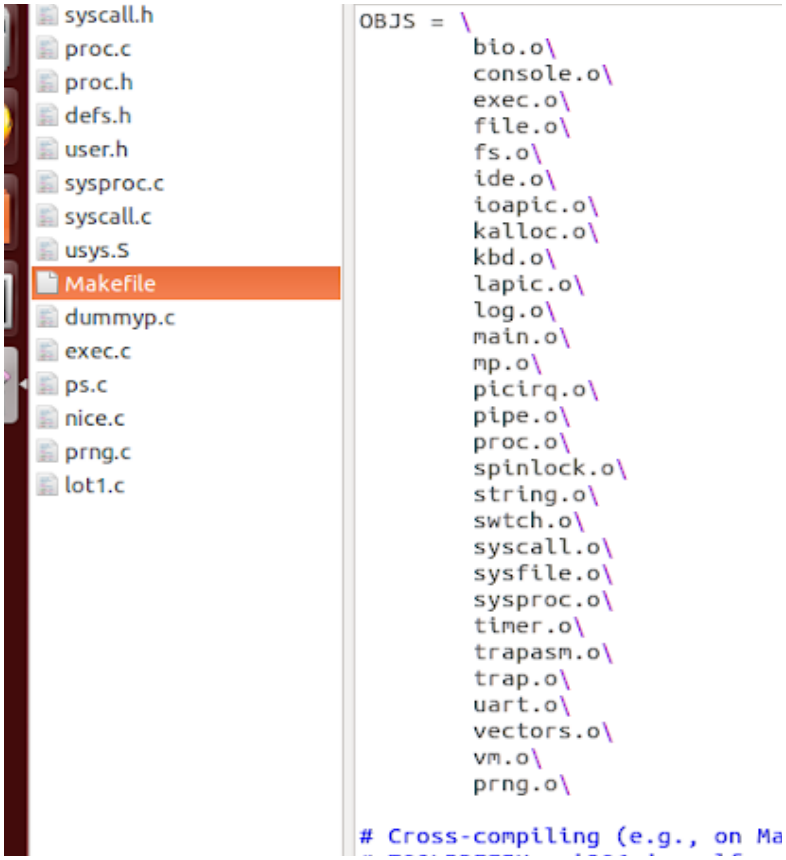
Our test case for testing PRNG is simple. We are just printing random numbers by passing different ranges like 0 to 10, (-100) to 100 and even 0 to 2^32.



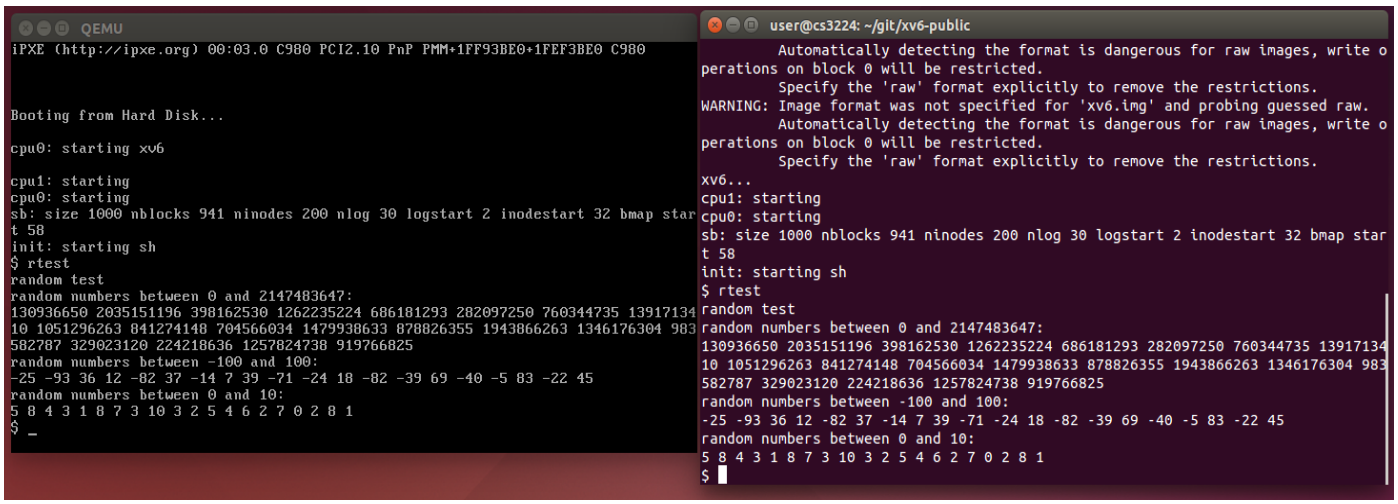
We add this “rtest.c” file in the Makefile to make it executable



Converting prng as an Object in Makefile



OUTPUT:



Task 3: Scheduler

Lottery scheduler is a probabilistic scheduling algorithm that allocates a certain number of tickets to every process in the system. Then, the scheduler uses a random number generator to generate a number based on which a process would be selected as the winner and would be allowed to run until its quantum is over. Lottery scheduling solves the problem of starvation. Giving each process at least one lottery ticket guarantees that it has non-zero probability of being selected at each scheduling operation.

We have implemented the lottery scheduler in proc.c file where the scheduler function of XV6 resides. XV6 employs round robin scheduler and we have replaced the round robin scheduler function with the lottery scheduler. We have added a few functions which perform various steps of the lottery scheduling algorithm.

uint settkts(int nice):

Every process has a priority value associated with it now. This was done in part 1 of the assignment. Every process gets a default priority value of 10. We can also choose to change the priority value assigned to a process using the nice system call.

We have implemented a function, which would allocate a certain number of tickets based on the nice value of each process. Specifically, for every nice value we multiply the tickets by 2. This way, every process gets a certain number of tickets.

```
uint
settkts(int nice)
{
  int tickets = 1;
  int i;
  for (i = 0; i < nice; i++) {
    tickets = tickets * 2;
  }
  return tickets;

}
```

uint totaltkts():

This function has been implemented to calculate the total number of tickets possessed by each process. We loop through the process table, and for every process based on the priority of the process we calculate the number of tickets it has. This can simply be done by passing the priority to numtickets function, since we have a modular function for that.

In this way, the total number of tickets associated with all the processes are calculated. We are calculating this value because we need to know the range in which we should be generating the

random number. Once we know the total tickets, we can use the pseudo random number generator that we implemented in part 2 of this assignment and get it to generate a random number.

```
uint
totaltkts(void)
{
  struct proc *p;
  uint t = 0;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
      continue;
      int n = p->priority;
      t += settkts(n);
  }
  return t;
}
```

Void scheduler():

Here we initially do the necessary declarations, like instantiating the process structure, enabling interrupts on the processor and acquiring the lock for the process table.

Now we need to know the total number of tickets possessed by the process. To get that value, we make a function call to the totaltickets function. Note that based on the priority of the process, the tickets are allocated for that particular process. We add a sanity check condition at this point. If the total number of tickets is zero, it means the processes are dormant, so we release the table lock.

Now that we know the total number of tickets, we can use the PRNG to generate a random number in that range 0 - total number of tickets. We make a system call to the random number generator to get that number. Now to determine the process that won, we should loop through the process table and find all the processes that are runnable. Among the runnable processes, we check the number of tickets that every process has and keep adding that to the counter as long as the counter value is lesser than the random number generated. When the counter value exceeds the winner number that was generated, we stop the loop there and announce that the process whose tickets were added to the counter in the last iteration as the winner.

Then we allow that process to run and use the processor until it's quantum is over. Since the lottery scheduler is a probabilistic algorithm, the processes with lower priority will get tickets as well, but the number of tickets assigned to it will be lesser than the tickets assigned to a higher priority process. Once a process is done running, we decrease its priority so that the next time tickets are assigned to it, it will be a bit lesser than whatever it had. This ensures a fair chance for all the processes in the system. If we don’t do this, there is a chance that the same number of tickets get assigned to the process that actually won. This will create a situation where the same process that ran recently has a higher probability of winning again.


Documents
syscall.h
proc.c
proc.h
defs.h
user.h
sysproc.c
syscall.c
usys.S
Makefile
dummyp.c
exec.c
ps.c
nice.c
prng.c
lot1.c
rtest.c
proc.c
proc.h
defs.h
user.h
sysproc
void
scheduler(void)
{
struct proc *p;
for(;;){
sti();
acquire(&ptable.lock);
uint total = totaltkts();
if (total == 0) {
release(&ptable.lock);
continue;
}
uint counter = 0;
uint winner = range(1, (int) total);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
if(p->state != RUNNABLE)
continue;
int nice = p->priority;
counter += settkts(nice);
if (counter < winner)
continue;
proc = p;
switchuvm(p);
p->state = RUNNING;
swtch(&cpu->scheduler, proc->context);
switchkvm();
//proc->priority = proc->priority - 1;
proc = 0;
break;
}
release(&ptable.lock);
}
}