

HOMEWORK 3 : LOTTERY SCHEDULER

Part 1 : Implementing Nice System Call

'Nice' system call is used to dynamically assign and modify the 'Nice' value or the 'Priority' of the processes. In this task, we have to assign/modify the priority of a given process by using the following rule – Higher the Nice value is, lower is the actual priority of the process.

In this task, I am assuming that Nice value is equal to the Priority value of the process. But functionally, what this means is that a process with Nice/Priority value of 5 is actually more important than a process with a Nice/Priority value of 20.

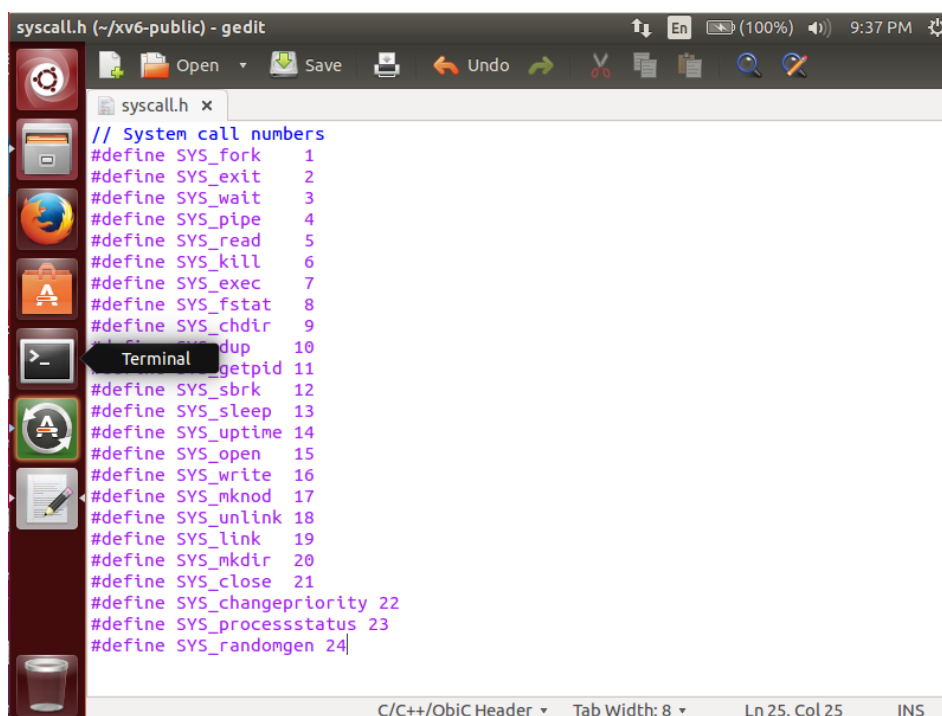
The range of these values is from 0 to 20. Initially, all the processes are assigned a value of 10 and the child processes are assigned a value of 4 (since the child processes have more priority over the parent process).

Implementation :

To complete this task, I had to implement two system calls – 'nice' and 'ps'. PS call is to retrieve and display information (PID, State and Priority value) of all the processes running in the system. To do this, I modified the following files :

1. SYSCALL.H

To start off, System call interface maintains the table of all the system call and associates each with a number. So first, I updated this table. In syscall.h, I added the following two system calls. Here, I have given the numbers 22 and 23 to the system calls. The 'changepriority' is for the nice system call and 'processtatus' (change priority) is for the ps system call.



```
syscall.h (~/.xv6-public) - gedit
// System call numbers
#define SYS_fork 1
#define SYS_exit 2
#define SYS_wait 3
#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_changepriority 22
#define SYS_processtatus 23
#define SYS_randomgen 24
```

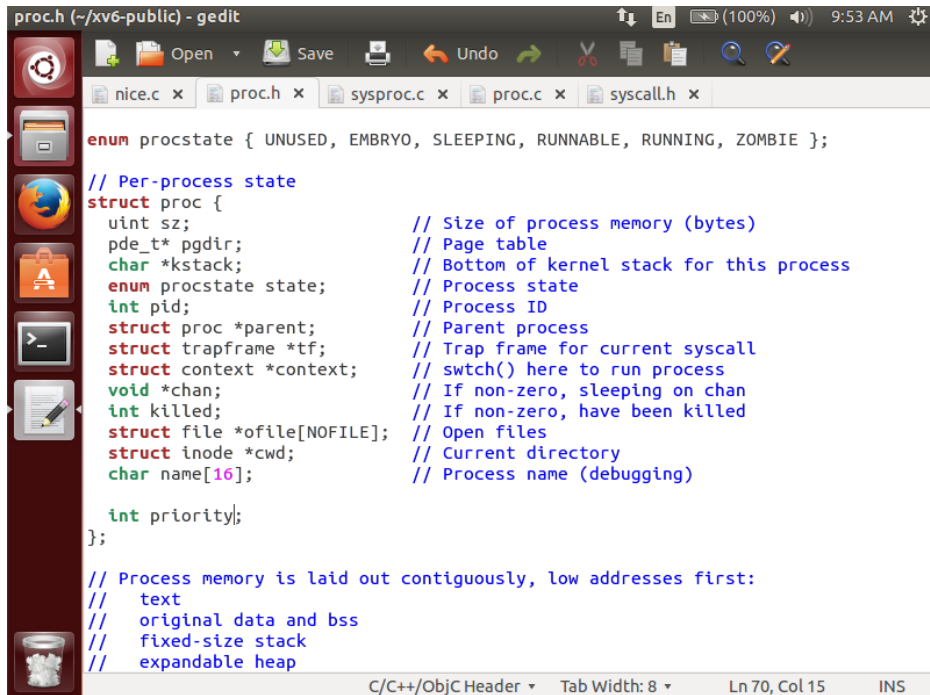
The screenshot shows a gedit window titled 'syscall.h (~/.xv6-public) - gedit'. The editor contains a list of system call numbers defined as macros. The list includes SYS_fork through SYS_randomgen. Two new entries have been added: SYS_changepriority with value 22 and SYS_processtatus with value 23. The status bar at the bottom indicates 'C/C++/ObjC Header', 'Tab Width: 8', 'Ln 25, Col 25', and 'INS'.

2. PROC.H

This file contains all the headers required to execute the system calls. These include all the required declarations, etc. Since we are implementing the nice and ps calls, we have to add the new attribute 'Priority' to the process structure present in this file.

This attribute will store the priority or the nice value of that particular process which will be used by the nice and ps calls.

On line 70, I have added the int variable, priority.



```
proc.h (~/.xv6-public) - gedit
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// 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 process
    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)

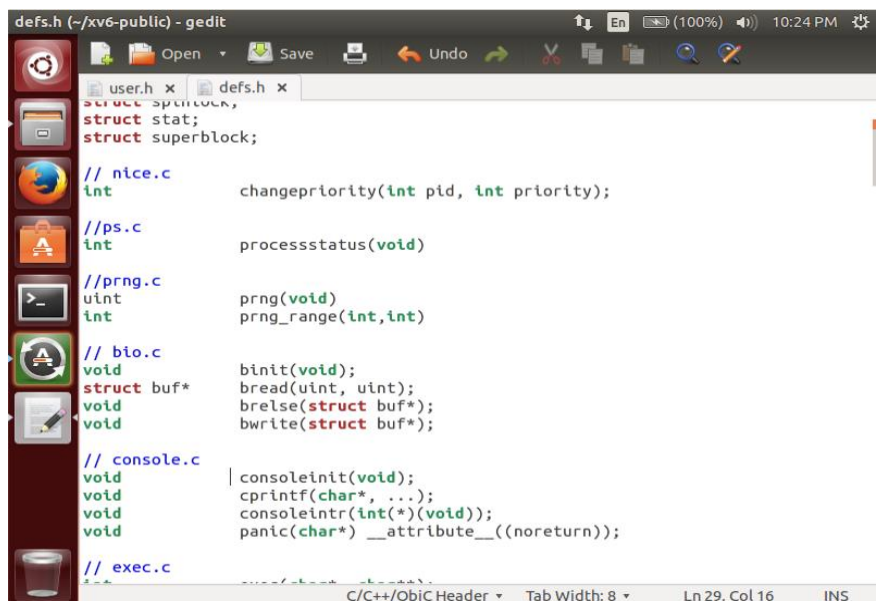
    int priority;

};

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

3. DEFS.H and USER.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 changepriority and processstatus.



```
defs.h (~/.xv6-public) - gedit
struct spinlock;
struct stat;
struct superblock;

// nice.c
int changepriority(int pid, int priority);

// ps.c
int processstatus(void);

// prng.c
uint prng(void);
int prng_range(int, int);

// bio.c
void binit(void);
struct buf* bread(uint, uint);
void brelse(struct buf*);
void bwrite(struct buf*);

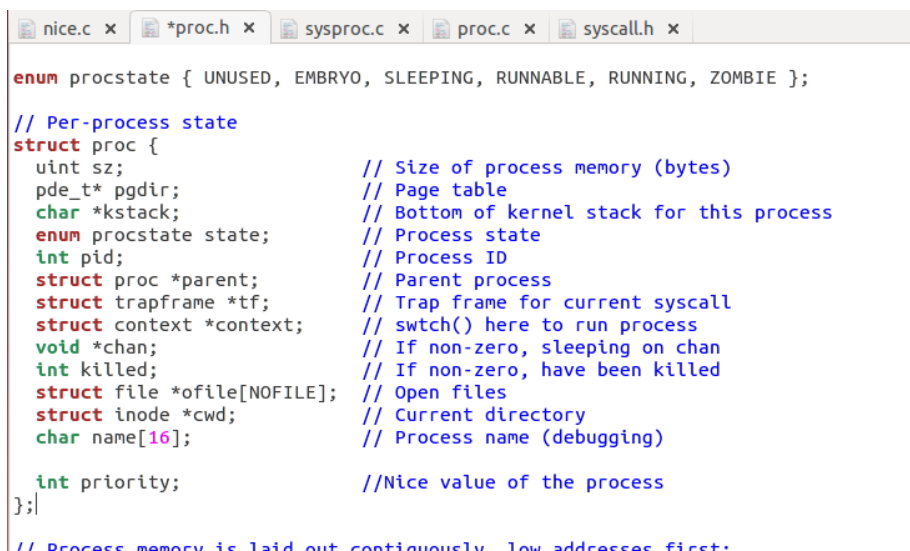
// console.c
void consoleinit(void);
void cprintf(char*, ...);
void consoleintr(int (*)(void));
void panic(char*) __attribute__((noreturn));

// exec.c
```

4. PROC.C and PROC.H

PROC.C file contains the definitions for the system calls that we declared and that were present in the previous files. In this file, we will define the functions – `changepriority` and `processstatus`, along with updating the other attributes.

In the PROC.H file, we see a structure called `proc`. This structure contains the attributes of each process initialized in the system. Since we are talking about a new property ‘priority’, we have to define it inside the `proc` structure.



```
nice.c x *proc.h x sysproc.c x proc.c x syscall.h x
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// 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 process
    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; // switch() 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; // Nice value of the process
};

// Process memory is laid out contiguously, low addresses first
```

Under the “found:” section of the PROC.C file, we assign a default value to this new attribute by using the command `p->priority = 10;`

I am assigning 10 as the default priority for all newly created processes.

It is also important to reflect this attribute when we create child processes. To do this, we head over to the EXEC.C file. Under this file, there is a portion of the code that sets up the parameters of the child process. Here we add the line `p->priority = 4;`

I am assigning 4 as the default priority for the child process because functionally, a child process should have more priority than the parent process. Hence the Process/Nice attribute of this process is lower than the parent process.

Coming back to the PROC.C, we have to add the definition for the two system calls we are implementing.

Here the function “`processstatus`” 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.* At first, 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 23” represents returning the function to its callee which is ‘`sys_cps`’ system call and it has the system call number 23 which we assigned earlier in the `sys_call.h` file.

Process Status :

```
nice.c x proc.h x sysproc.c x proc.c x syscall.h x
processstatus(void)
{
    struct proc *p;

    //Enable interrupt on the processor
    sti();

    //Looping over processes to match with the pid
    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 23;
}
```

Change Priority :

```
nice.c x proc.h x sysproc.c x proc.c x syscall.h x
    }
    cprintf(" %p", pc[i]);
}
cprintf("\n");
}

//Adding the definition for priority change
int
changepriority(int pid, int priority)
{
    struct proc *process;
    acquire(&ptable.lock);
    for(process = ptable.proc; process < &ptable.proc[NPROC]; process++){
        if(process->pid == pid){
            process->priority = priority;
            break;
        }
    }
    release(&ptable.lock);
    return pid;
}

//Adding the definition for process status
int
processstatus(void)
{
    struct proc *p;
```

In the changepriority, we have two arguments to the function – PID and the Nice/Priority value. This is also a simple code that acquires lock on the process table, loops through it to find the process whose PID matches with the one provided to the function, and then updates the priority of that particular process. We return the PID of the modified process.

5. SYSPROC.C

In this file, we define the “sys_<>” functions that will be calling the system calls we created. I created two functions – sys_changepriority and sys_processtatus. These functions in turn call the changepriority() and the processtatus() that we defined in the PROC.C file.

In the sys_processtatus(void), we are directly calling the processtatus system call we defined earlier.

Things are a tad different in the sys_changepriority function. Rather than directly calling the function, we are initializing the two variables by passing some integers in it using the argint() functions since we cannot pass uninitialized variables in the function definition.

sys_processtatus :

```
int
sys_processtatus(void)
{
    return processtatus();
}
```

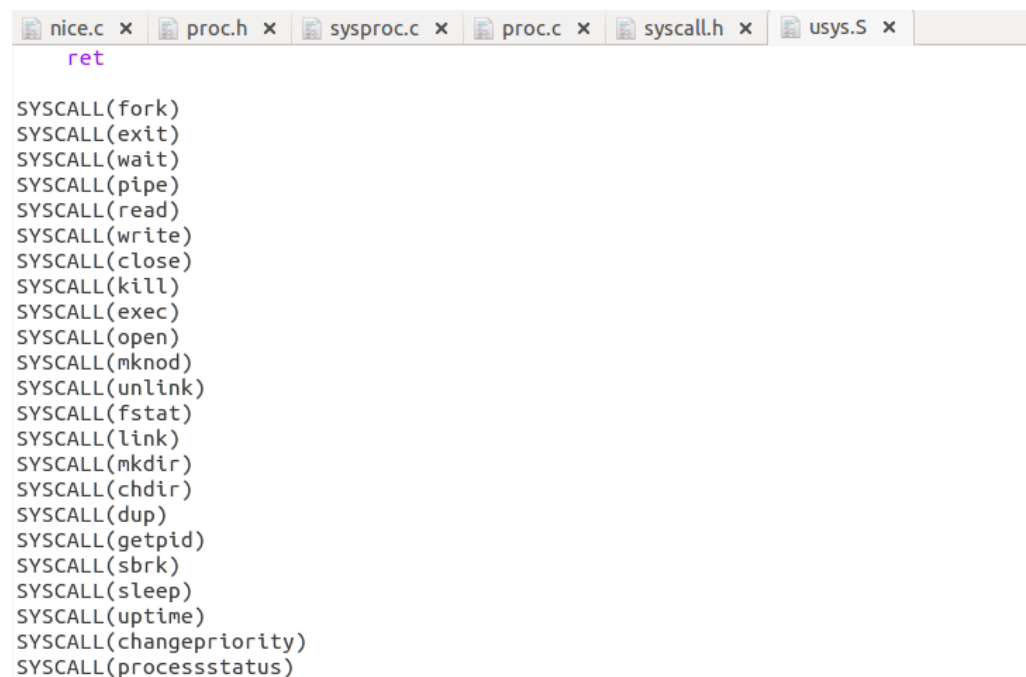
sys_changepriority :

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

    return changepriority(pid, pr);
}
```

6. USYS.S

Next, we have to make some minor changes in the usys.S file. The '.S' extension indicates that this file has assembly level code and this file interacts with the hardware of the system.



```
nice.c x  proc.h x  sysproc.c x  proc.c x  syscall.h x  usys.S x
ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
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(changepriority)
SYSCALL(processstatus)
```

7. SYSCALL.C

This file is the entry point for the definition of the system call. 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.

```
extern int sys_changepriority(void);
extern int sys_processstatus(void);
```

```
static int (*syscalls[])(void) = {
[SYS_fork]      sys_fork,
[SYS_exit]      sys_exit,
[SYS_wait]      sys_wait,
[SYS_pipe]      sys_pipe,
[SYS_read]      sys_read,
[SYS_kill]      sys_kill,
[SYS_exec]      sys_exec,
[SYS_fstat]     sys_fstat,
[SYS_chdir]     sys_chdir,
[SYS_dup]       sys_dup,
[SYS_getpid]    sys_getpid,
[SYS_sbrk]      sys_sbrk,
[SYS_sleep]     sys_sleep,
[SYS_uptime]    sys_uptime,
[SYS_open]      sys_open,
[SYS_write]     sys_write,
[SYS_mknod]     sys_mknod,
[SYS_unlink]    sys_unlink,
[SYS_link]      sys_link,
[SYS_mkdir]     sys_mkdir,
[SYS_close]     sys_close,
[SYS_changepriority] sys_changepriority,
[SYS_processstatus] sys_processstatus,
```

8. PS.C and NICE.C

This is the last step in the implementation of the system calls. We have modified all the required kernel files, but now we create the program files that will be used to trigger the system call. I am calling these 'ps' and 'nice', keeping true to the linux versions.

PS.C →

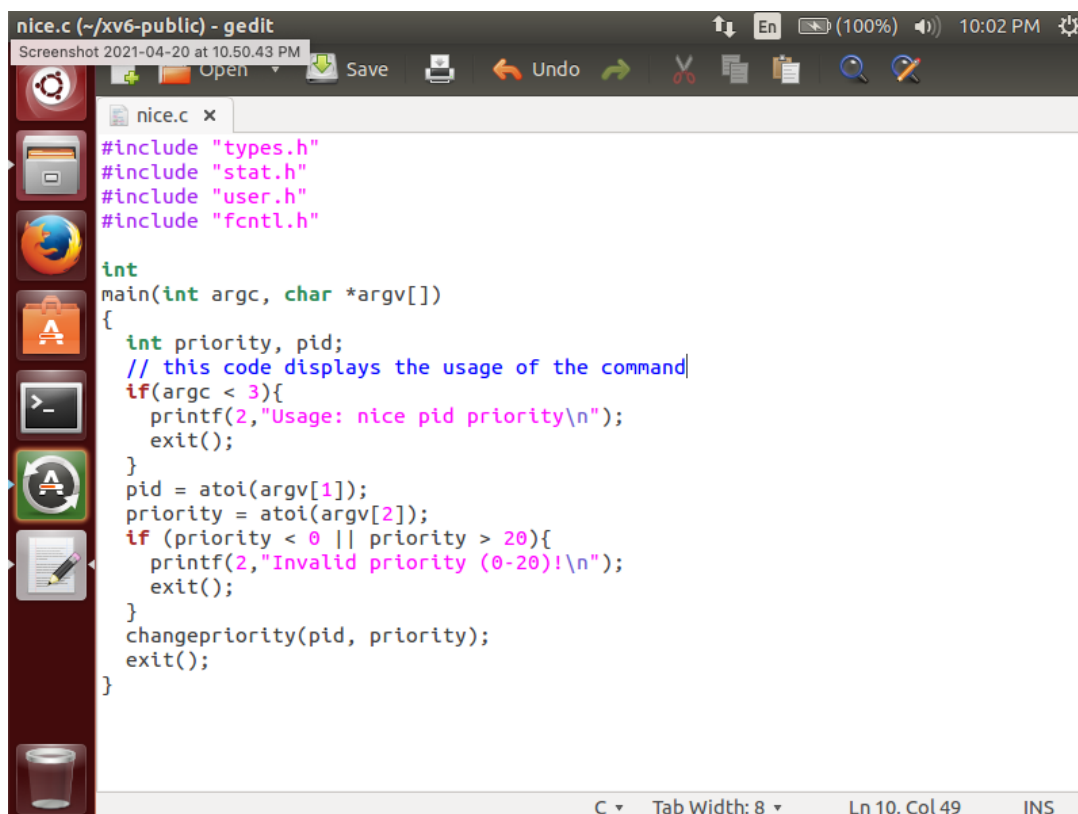
This is a simple file that directly calls the 'processstatus' command that we defined previously.

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

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

NICE.C →

This file, we read the command line arguments that are passed to the nice system call, i.e the PID and priority. We do a check to ensure that the argument priority lies inside the range of 0-20. If yes, we call the changepriority function with the arguments provided. If the priority is invalid, we print an error message.



```
nice.c (~/.xv6-public) - gedit
Screenshot 2021-04-20 at 10:50:43 PM

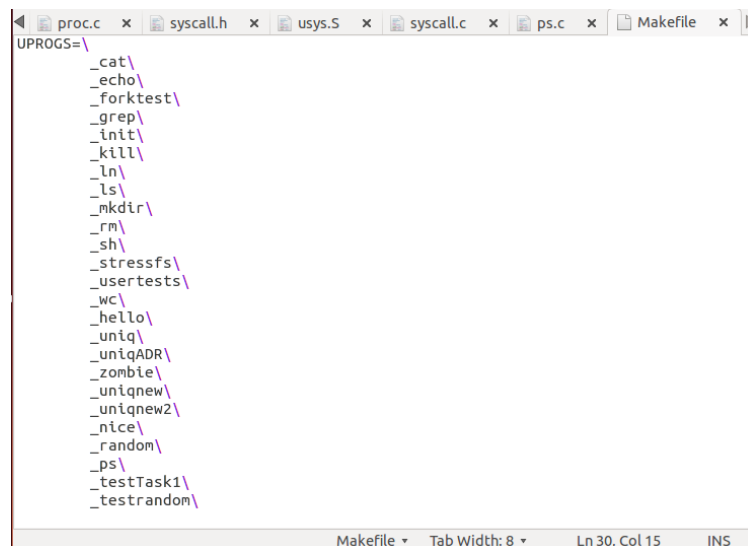
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int
main(int argc, char *argv[])
{
    int priority, pid;
    // this code displays the usage of the command
    if(argc < 3){
        printf(2, "Usage: nice pid priority\n");
        exit();
    }
    pid = atoi(argv[1]);
    priority = atoi(argv[2]);
    if (priority < 0 || priority > 20){
        printf(2, "Invalid priority (0-20)!\n");
        exit();
    }
    changepriority(pid, priority);
    exit();
}
```

FINAL Steps and TEST CASE :

Once we have created/modified all these files, it is time to update the MAKEFILE of xv-6 to include the new files we created, namely the ps.c and nice.c and the testcase.

Add the names in the makefile list in the proper syntax as follows :

A screenshot of a text editor window with multiple tabs. The active tab is 'Makefile'. The content shows a list of programs under the variable 'UPROGS=':

```
UPROGS=\n_cat\n_echo\n_forktest\n_grep\n_init\n_kill\n_ln\n_ls\n_mkdir\n_rm\n_sh\n_stressfs\n_usertests\n_wc\n_hello\n_uniq\n_uniqADR\n_zombie\n_uniqnew\n_uniqnew2\n_nice\n_random\n_ps\n_testTask1\n_testrandom
```

 The status bar at the bottom indicates 'Makefile', 'Tab Width: 8', 'Ln 30, Col 15', and 'INS'.

The Test Case :

I have created a file “testTask1.c” that contains a test case. In this case, we are creating a parent process and we are forking it to create child processes, the number of these child processes is determined by the argument we provide to the test case.

In this case, after the child process is created, we are running some CPU- wasting computations. This is because we want to check the status of the processes while the case is running, and therefore we need some time delay where the processes are running for us to issue the commands to test their status. This will give us time to modify the priority of the tasks dynamically, thus testing the function calls.

To run the test case, issue the following command :

```
$testTask1 3*
```

This will create a parent task that will spawn three child processes, each with a delay of almost 10 seconds.

While the child process executes, we can issue the commands

\$ps : This will print the current status of the processes active in the system. This will print a list of processes along with their PID, their STATE and their PRIORITY(which is the NICE value)

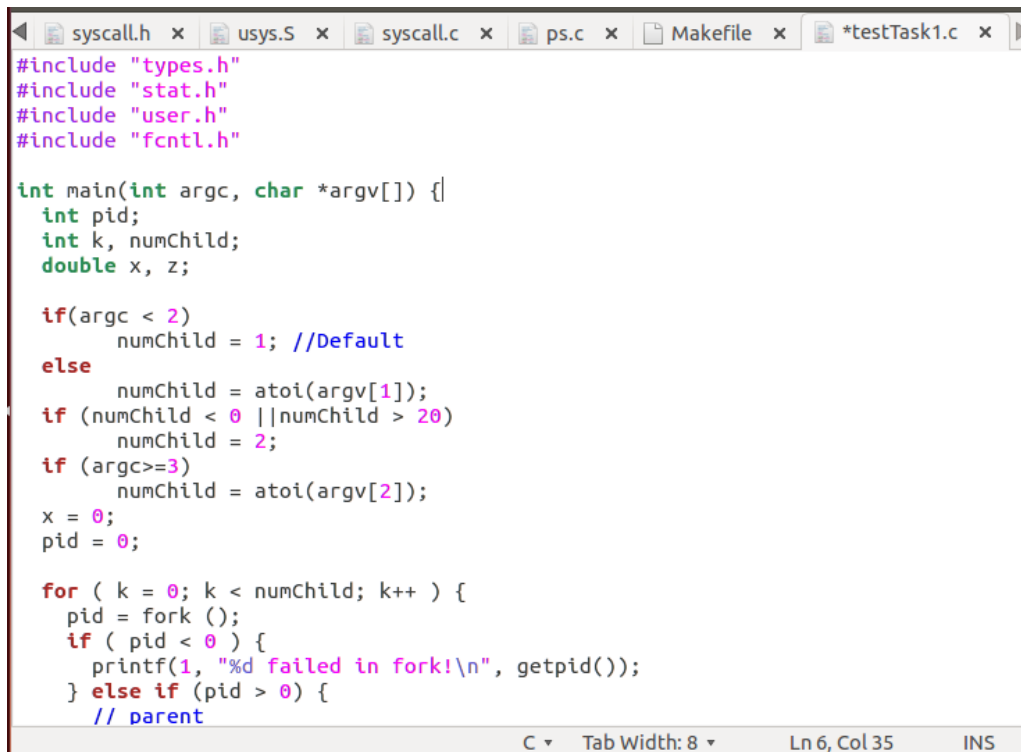
\$nice <pid> <new_nice_value> : This command will modify the priority or the nice value of the process mentioned in the PID.

While the child processes are running, issue a ‘ps’ command to list the active processes.

Note down the PID of one of the processes, and issue a ‘nice’ command to modify the

priority of that particular process. Issue another 'ps' command. This should list the process with its new Nice/priority value.

TEST CASE CODE :

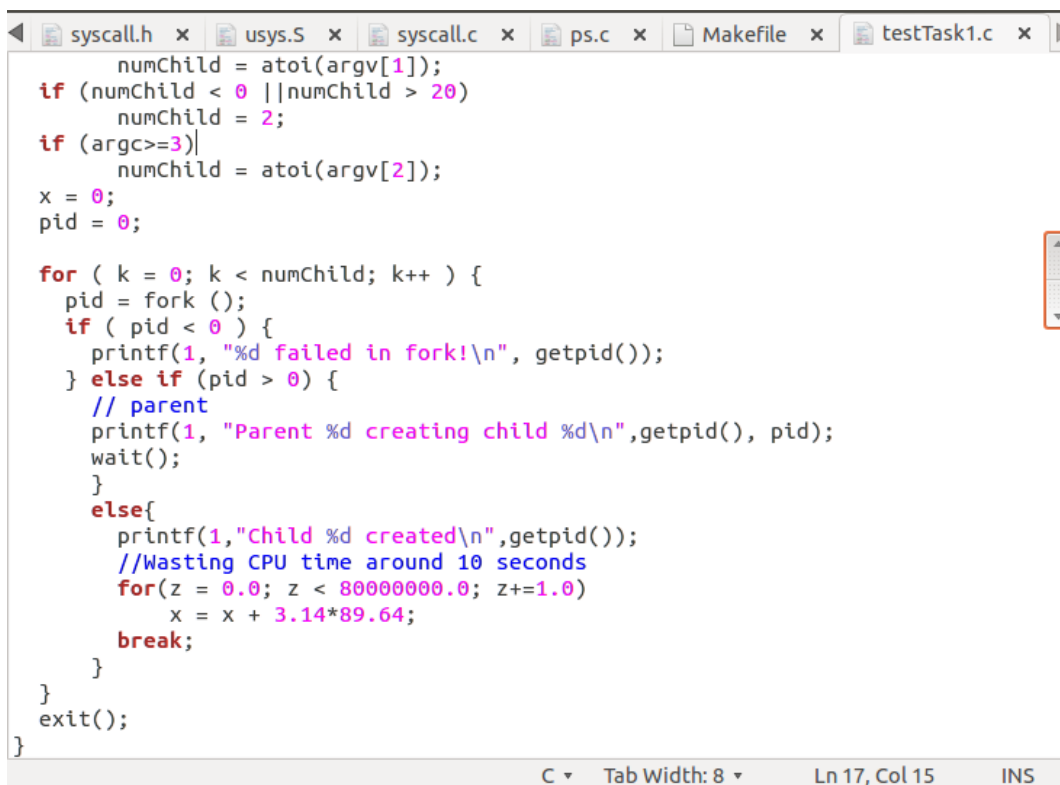


```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int main(int argc, char *argv[]) {
    int pid;
    int k, numChild;
    double x, z;

    if(argc < 2)
        numChild = 1; //Default
    else
        numChild = atoi(argv[1]);
    if (numChild < 0 || numChild > 20)
        numChild = 2;
    if (argc>=3)
        numChild = atoi(argv[2]);
    x = 0;
    pid = 0;

    for ( k = 0; k < numChild; k++ ) {
        pid = fork ();
        if ( pid < 0 ) {
            printf(1, "%d failed in fork!\n", getpid());
        } else if (pid > 0) {
            // parent
```



```
        numChild = atoi(argv[1]);
    if (numChild < 0 || numChild > 20)
        numChild = 2;
    if (argc>=3)
        numChild = atoi(argv[2]);
    x = 0;
    pid = 0;

    for ( k = 0; k < numChild; k++ ) {
        pid = fork ();
        if ( pid < 0 ) {
            printf(1, "%d failed in fork!\n", getpid());
        } else if (pid > 0) {
            // parent
            printf(1, "Parent %d creating child %d\n",getpid(), pid);
            wait();
        }
        else{
            printf(1,"Child %d created\n",getpid());
            //Wasting CPU time around 10 seconds
            for(z = 0.0; z < 80000000.0; z+=1.0)
                x = x + 3.14*89.64;
            break;
        }
    }
    exit();
}
```

OUTPUTS :

```
user@cs3224: ~/xv6-public
init: starting sh
$ ps
Name    PID    State  Priority
init    1      SLEEPING  4
sh      2      SLEEPING  4
ps      3      RUNNING   4
$ testTask1 4&
$ Parent 5 creating child 6
Child 6 created
$ ps
Name    PID    State  Priority
init    1      SLEEPING  4
sh      2      SLEEPING  4
testTask1 6      RUNNING   10
testTask1 5      SLEEPING   4
ps      7      RUNNING   4
$ nice 5 2
$ ps
Name    PID    State  Priority
init    1      SLEEPING  4
sh      2      SLEEPING  4
testTask1 6      RUNNING   10
testTask1 5      SLEEPING   2
ps      9      RUNNING   4

user@cs3224: ~/xv6-public
$ ps
Name    PID    State  Priority
init    1      SLEEPING  4
sh      2      SLEEPING  4
testTask1 6      RUNNING   10
testTask1 5      SLEEPING   2
ps     10      RUNNING   4
$ Parent 5 creating child 11
Child 11 created
$ ps
Name    PID    State  Priority
init    1      SLEEPING  4
sh      2      SLEEPING  4
testTask1 11     RUNNING   10
testTask1 5      SLEEPING   2
ps     12      RUNNING   4
$ nice 11 1
$ ps
Name    PID    State  Priority
init    1      SLEEPING  4
sh      2      SLEEPING  4
testTask1 11     RUNNING    1
testTask1 5      SLEEPING   2
ps     14      RUNNING   4
$ Parent 5 creating child 15
Child 15 created
$ ps
Name    PID    State  Priority
init    1      SLEEPING  4
sh      2      SLEEPING  4
testTask1 15     RUNNING   10
testTask1 5      SLEEPING   10
ps     17      RUNNING   4
$ Parent 5 creating child 18
Child 18 created
```

As we can see, we are able to dynamically modify the priority of the running tasks using the NICE command.

END OF TASK 1

Part 2 : Implementing a Pseudo Random Number Generator

As mentioned in the task, I researched on how XORshift Random Numbers are generated. There is a research paper which is published that shows an easy way to implement this using XOR shift. This method makes sure that we generate a random number in the range of 0 to $2^{32} - 1$. This is in-kernel implementation, which means that the random numbers being generated are slight faster in terms of execution.

The reason we need this PRNG is because we are implementing the Lottery based Scheduler technique. In this, we assign something known as 'Tickets' to each running process in the process table, which is nothing but an integer that tells us how much tickets each process has. There are several ways to assign tickets to these processes, one of which I have mentioned in the next task few pages ahead. Once tickets have been assigned, we pick a random ticket/number in the range of the total tickets, and the process which has the Golden ticket, gets the CPU scheduling. This is similar to a real life Lottery system, hence the name Lottery based Scheduling.

In this task, I have created two calls, one is the 'randomgen()' which returns a random number between 0 and $2^{32} - 1$. The other is the 'randomgenrange(int low, int high)' which returns a random number in the range provided by the integers low and high.

IMPLEMENTATION :

This is very similar to the previous task, we create these calls and define them in the kernel files as mentioned above.

The code for randomgen() is :

```
uint
randomgen(void)
{
    static unsigned int z1 = 12345, z2 = 12345, z3 = 12345, z4 = 12345;
    unsigned int b;
    b = ((z1 << 6) ^ z1) >> 13;
    z1 = ((z1 & 4294967294U) << 18) ^ b;
    b = ((z2 << 2) ^ z2) >> 27;
    z2 = ((z2 & 4294967288U) << 2) ^ b;
    b = ((z3 << 13) ^ z3) >> 21;
    z3 = ((z3 & 4294967280U) << 7) ^ b;
    b = ((z4 << 3) ^ z4) >> 12;
    z4 = ((z4 & 4294967168U) << 13) ^ b;
    return (z1 ^ z2 ^ z3 ^ z4) / 2;
}
```

The code for randomgenrange(int,int) is :

```
int
randomgenrange(int low, int high)
{
    static unsigned int z1 = 12345, z2 = 12345, z3 = 12345, z4 = 12345;
    unsigned int b;
    b = ((z1 << 6) ^ z1) >> 13;
    z1 = ((z1 & 4294967294U) << 18) ^ b;
    b = ((z2 << 2) ^ z2) >> 27;
    z2 = ((z2 & 4294967288U) << 2) ^ b;
    b = ((z3 << 13) ^ z3) >> 21;
    z3 = ((z3 & 4294967280U) << 7) ^ b;
    b = ((z4 << 3) ^ z4) >> 12;
    z4 = ((z4 & 4294967168U) << 13) ^ b;

    uint randomnumber = (z1 ^ z2 ^ z3 ^ z4) / 2;
    if (high < low) {
        int temp = low;
        low = high;
        high = temp;
    }
    int range = high - low + 1;
    return randomnumber % (range) + low;
}
```

TESTCASE :

I have created a test case called 'testrandom.c'.

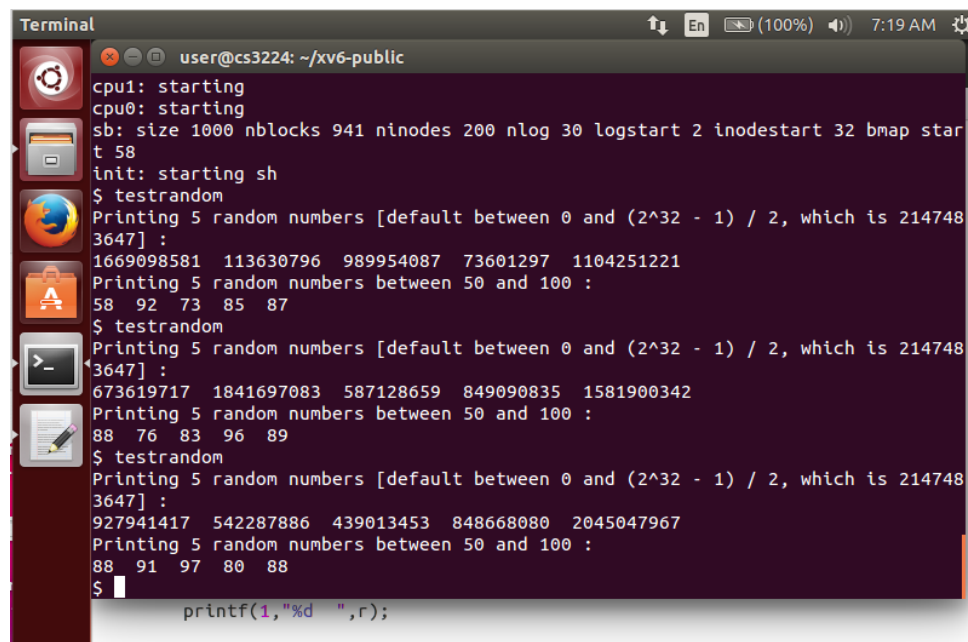
```
int main(int argc, char *argv[]) {  
  
    int r,i;  
  
    printf(1,"Printing 5 random numbers [default between 0 and (2^32 - 1) / 2,  
which is 2147483647] : \n");  
    for(i=0;i<5;i++)  
    {  
        r = randomgen();  
        printf(1,"%d ",r);  
    }  
  
    printf(1,"\nPrinting 5 random numbers between 50 and 100 : \n");  
    for(i=0;i<5;i++)  
    {  
        r = randomgenrange(50,100);  
        printf(1,"%d ",r);  
    }  
    printf(1,"\n");  
    exit();  
}
```

This is a simple test case that prints 5 random numbers using the randomgen() function and then prints 5 random numbers between the range of 50 and 100.

EXECUTING THE TEST CASE :

Run the command in QEMU: \$testrandom

OUTPUT :



```
Terminal
user@cs3224: ~/xv6-public
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ testrandom
Printing 5 random numbers [default between 0 and (2^32 - 1) / 2, which is 214748
3647] :
1669098581 113630796 989954087 73601297 1104251221
Printing 5 random numbers between 50 and 100 :
58 92 73 85 87
$ testrandom
Printing 5 random numbers [default between 0 and (2^32 - 1) / 2, which is 214748
3647] :
673619717 1841697083 587128659 849090835 1581900342
Printing 5 random numbers between 50 and 100 :
88 76 83 96 89
$ testrandom
Printing 5 random numbers [default between 0 and (2^32 - 1) / 2, which is 214748
3647] :
927941417 542287886 439013453 848668080 2045047967
Printing 5 random numbers between 50 and 100 :
88 91 97 80 88
$
printf(1,"%d ",r);
```

As we can see, it is printing random numbers each time we call the program.

END OF TASK 2

TASK 3 : Implementing a Lottery Based Scheduler

Lottery Based Scheduler is a Probabilistic Scheduling algorithm, as it randomly picks the process for execution based on the value of the "Golden Ticket". In this algorithm, we provide each process with a certain range of 'tickets'. Then, we pick the 'Golden ticket', which is nothing but a random ticket number in the range of the total tickets available. Whichever process holds this particular ticket, it gets selected for execution till its quantum is over.

The implementation of my algorithm is an update over the previous round robin algorithm, which means that the data structure used for the processes is similar. The way they get picked for execution is probabilistic.

Some of the quirks of this algorithm :

- The way of mapping the tickets for each process depends on the Nice value or the Priority value, and it is recorded not as an attribute, but it is calculated each time the scheduler runs to allow for dynamic allocation of tickets depending on the current priority of the process.
- The number of tickets held by a process is calculated by $2^{(20 - \text{nice}/\text{priority})}$. Hence, higher the nice/priority value, lesser the amount of tickets held by that process. If a process has priority 20, then the tickets held by the process are $2^{(20-20)} = 1$. And if the process has priority 0, then the tickets held by the process are 2^{20} .
- The order of processes in the process queue is similar to the round robin algorithm.
- Total number of tickets are calculated by iterating through all the processes in the process list and tickets held (by the above mentioned calculation using the priority) are added to the total sum. Then a Golden Ticket is picked randomly in the range of 1 and totalTickets.
- When the scheduler iterates through the list of processes, it checks if the golden ticket lies in the count of each process's tickets. If yes, then that particular process is selected for execution. If not, then the tickets held by that task is deducted from the golden ticket count and the iteration continues with the next process.
- For example, if there are 3 processes with 10 tickets each, then the total tickets are 30. A golden ticket is selected randomly, let's say it is 25. When the first process is read, the tickets it has (10) is lesser than the Golden ticket (25). Hence, 10 is deducted from the Golden Ticket count and the first process is discarded. Second process also has 10 tickets, which is still less than the golden count (15 now). Hence 10 is deducted and the third process is checked. NOW, the golden count is at 5, and the ticket count of the third process is 10. This means the process holds the golden ticket. Hence it gets selected for execution.

Then we allow that process to run and use the processor until its 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. This approach leads to a situation where the processes are less likely to starve, which handles the edge cases too.

IMPLEMENTATION :

Number of Tickets calculation :

```
//Given the priority/nice value, calculating the number of tickets.
//Since lower the value, higher the actual priority,
//I'm calculating num of ticks as 2^(20-nice/priority)
//Hence lower the nice value, higher the number of tickets.
uint
numtickets(int priority)
{
    int n = 20-priority;
    int tickets = 1;
    int i;
    for (i = 0; i < n; i++) {
        tickets = tickets * 2;
    }
    return tickets;
}
```

Total Tickets calculation :

```
//Calculating total available tickets
uint
totaltickets(void)
{
    struct proc *p;
    uint total = 0;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
            continue;
        int nice = p->priority;
        total += numtickets(nice);
        //printf("%d numtickets: %d\n", p->pid, numtickets(nice));
    }
    return total;
}
```

Scheduler :

```
void
scheduler(void)
{
    struct proc *p;
    for(;;){
        // Enable interrupts on this processor.
        sti();

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);

        // get total number of tickets by iterating through every process
        uint total = totaltickets();
        if (total == 0) {
            release(&ptable.lock);
            continue;
        }
        //cprintf("total tickets: %d\n", total);

        // hold lottery
        uint counter = 0; // used to track if we've found the winner yet
        uint winner = randomgenrange(1, (int) total);
        //cprintf("winner ticket: %d\n", winner);

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

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

    int priority = p->priority;
    counter += numtickets(priority);
    if (counter < winner)
        continue;
    // Switch to chosen process. It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    proc = p;

    switchvm(p);
    p->state = RUNNING;

    swtch(&cpu->scheduler, proc->context);
    switchkvm();
    p->priority = p->priority - 1;
    // Process is done running for now.
    // It should have changed its p->state before coming back.
    proc = 0;
    break;
}
release(&ptable.lock);
}
```