Information Technology University of the Punjab
SE301T Operating Systems – Spring 2025
Assignment 2 [CLO2]

Deadline: **Solution** **Total Marks: 60**

Instructions:

i. This is an individual assignment so every student must submit their own solution.
ii. You will need a Linux-based environment to attempt this assignment (WSL/dual-boot/VMWare etc.).
iii. All questions are based on the xv6 OS, so you would need qemu to run it. Install it using `apt install qemu-system`. Take help from instructor/TA to setup your system to run xv6 if you are unable to do it on your own.
iv. As submission, you will be required to submit a single file in PDF format on Google Classroom which should contain all the code, its description and output screenshots (if any).
v. Clearly label the start of all questions in your report. Also, make sure to highlight the important bits in your solution.
vi. You can take help from the textbook/reference books or the Linux man pages (man fork etc.). Discussion among peers without showing the solution is acceptable, but you must attempt individually. Your submissions will be checked for plagiarism including AI generated content and if found, it will be dealt with according to the anti-plagiarism policy.
vii. Make sure to submit it by the deadline. Late submissions will not be accepted.

---

1. [15] The `kill` system call in xv6 takes the PID of the process to be killed as an argument. The `sys_kill()` function in sysproc.c calls the `kill()` function in `proc.c`. (The `kill` function in `proc.c` is defined inside the kernel code, so it is not accessible to the process directly; the user program can only invoke the `kill` *system call* because the header file `user.h` has its prototype and usys.S has its assembly definition.) This `kill()` function (in `proc.c`) changes the value of the `killed` member of the respective process's `proc struct` to `1`. The process is exited inside the `trap` function in `trap.c`.

   You are required to trace the `kill` system call. You may use GDB to trace it into the kernel and back to the caller or you can simply search through the xv6 source code as shown in class. In your solution, present code at each step and explain what it does.

See video: https://youtu.be/WhUC5dpZeN8

2. [15] Trace the kernel code from the point a process's `sleep` system call is executed inside the kernel by the `sleep` function in `proc.c` to the point when the next RUNNABLE process, which presumably had called `yield` in past, resumes. Note down all the related code in your solution and explain what each part of the code does.

See video: https://youtu.be/kwCOim5heQE

3. [30] The scheduler in xv6 follows the *round-robin scheduling policy*. You are required to change it to *ticket-based lottery scheduling*. You can follow the steps given below to implement this:

   a. Add an integer member named `mytickets` in `struct proc`. This would hold the number of assigned tickets to a process.
   b. Add a system call `settickets` to the xv6 code that allows a process to assign tickets to itself. The `settickets` system call should take an integer argument. You can look at other system calls that take arguments to know how to do this, e.g. the `kill` system call takes `pid` (an integer) as an argument.
   c. In scheduler, traverse the `ptable` to sum the tickets of all RUNNABLE processes in a variable `ticketsum`.
   d. Then, generate a random number between `0` and `ticketsum`. Functions for random number generation are provided with this assignment in a C file and a header file.
   e. Find the process to run as shown in lottery scheduling pseudocode in Figure 9.1 (OSTEP Chapter 9).

   Once you have changed the scheduler, create a test for your implementation. This can be done by creating a user program that uses `fork` to create a child process. You can then assign different tickets to both processes in a proportion such that one process has a much higher chance of running than the other. For example, assign 10 tickets to the child and 100 tickets to the parent. Both processes should also print some message repeatedly in an infinite loop that helps to identify them. Running such a user program should show that the process that has a high number of tickets gets to print its message more often than the process with a low number of tickets.

Add new member “tickets” to struct proc:

D: > ITU > OS_S25 > xv6-public > proc.h

```c
struct proc {
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int tickets;
};
```

Add system call settickets:

D: > ITU > OS_S25 > xv6-public > user.h

```c
int uptime(void);
int settickets(int);
int printtickets(void);
```

D: > ITU > OS_S25 > xv6-public > usys.S

```
SYSCALL(settickets)
SYSCALL(printtickets)
```

D: > ITU > OS_S25 > xv6-public > syscall.h

```c
#define SYS_mkdir   20
#define SYS_close   21
#define SYS_settickets 22
#define SYS_printtickets 23
```

D: > ITU > OS_S25 > xv6-public > sysproc.c

```c

int
sys_settickets(void)
{
  int n;

  if(argint(0, &n) < 0)
    return -1;
  myproc()->tickets = n;
  return 0;
}

int
sys_printtickets(void)
{
  return myproc()->tickets;
}
```

D: > ITU > OS_S25 > xv6-public > syscall.c

```c
extern int sys_settickets(void);
extern int sys_printtickets(void);
```

D: > ITU > OS_S25 > xv6-public > syscall.c

```c
[SYS_close]    sys_close,
[SYS_settickets]   sys_settickets,
[SYS_printtickets]   sys_printtickets,
```

In allocproc (proc.c), assign default tickets:

```c
found:
  p->context->eip = (uint)forkret;
  p->tickets = 10;

  return p;
}
```

Include rand.h and copy code from rand.c to the end of proc.c file:

```c
#include "spinlock.h"
#include "rand.h"
```

```c

/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0df   /* constant vector a */
#define UPPER_MASK 0x80000000 /* most significant w-r bits */
#define LOWER_MASK 0x7fffffff /* least significant r bits */

/* Tempering parameters */
#define TEMPERING_MASK_B 0x9d2c5680
#define TEMPERING_MASK_C 0xefc60000
#define TEMPERING_SHIFT_U(y)  (y >> 11)
#define TEMPERING_SHIFT_S(y)  (y << 7)
#define TEMPERING_SHIFT_T(y)  (y << 15)
#define TEMPERING_SHIFT_L(y)  (y >> 18)

#define RAND_MAX 0x7fffffff
```

proc.c

D: > ITU > OS_S25 > xv6-public > proc.c

```c
static unsigned long mt[N]; /* the array for the sta
static int mti=N+1; /* mti==N+1 means mt[N] is not i

/* initializing the array with a NONZERO seed */
void
sgenrand(unsigned long seed)
{ ···
}

long /* for integer generation */
genrand()
{ ···
}

// Assumes 0 <= max <= RAND_MAX
// Returns in the half-open interval [0, max]
long random_at_most(long max) { ···
```

Include this line so that sys_uptime may be used to generate a seed for the random number generation.

proc.c

D: > ITU > OS_S25 > xv6-public > proc.c

```c
extern void trapret(void);
extern int sys_uptime(void);
```

Modify scheduler according to lottery scheduling algorithm:

proc.c

D: > ITU > OS_S25 > xv6-public > proc.c

```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  int total_tickets;
  int ticketcount;

  for(;;){
    // Enable interrupts on this processor.
    sti();
    total_tickets = 0;
    ticketcount = 0;

    // Loop over process table looking for process to ru
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
```

proc.c

D: > ITU > OS_S25 > xv6-public > proc.c

```c
{
  for(;;){
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if (p->state == RUNNABLE)
        total_tickets += p->tickets;
    }

    unsigned int seed = sys_uptime();
    sgenrand(seed);
    long winner = random_at_most(total_tickets);

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


```

proc.c

D: > ITU > OS_S25 > xv6-public > proc.c

```c
{
  for(;;){
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
        }
      }
    }
    release(&ptable.lock);
  }
```

Add user program to test lottery scheduling:

myprog.c

D: > ITU > OS_S25 > xv6-public > myprog.c

```c
#include "types.h"
#include "user.h"

int main(){
    int rc = fork();
    if (rc == 0){
        settickets(100);
        for (int i = 0; i< 50; i++)
            printf(1, "Child count: %d\n", i);
        exit();
    }
    else{
        settickets(10);
        for (int i = 0; i< 50; i++)
            printf(1, "Parent count: %d\n", i);
        wait();
    }
    exit();
}
```

Run in Xv6:

```
$ myprog
Parent count: 0Child count: 0
Child count: 1
Child count: 2
Child count: 3
Child count: 4
Child count: 5
Child count: 6
Child count: 7
Child count: 8
Child count: 9
Child count: 10
Child count: 11
Child count: 12
Child count: 13
Child count: 14
Chil
Parent count: 1
Parent count: 2
Pad count: 15
Child count: 16
Child crent count: 3
Parent count: 4
Parent counount: 17
Child count: 18
Child count: 19
Child count: 20
```

```
Child count: 38
Child count: 39
Child count: 40
Child count: 41
Child count: 42
Child count: 43
Child count: 44
Child count: 45
Child count: 46
Child count: 47
Child count: 48
Child count: 49
t: 5
Parent count: 6
Parent count: 7
Parent count: 8
Parent count: 9
Parent count: 10
Parent count: 11
Parent count: 12
Parent count: 13
Parent count: 14
Parent count: 15
Parent count: 16
Parent count: 17
Parent count: 18
```

Child completes its count first; parent completes after child finishes.