Lab #2: System Call Implementation Report

Group Members:

- Bhagyesh Ravindra Gaikwad (bgaik001@ucr.edu, NetID: bgaik001)
- Dipro Chakraborty (dchak006@ucr.edu, NetID: dchak006)

Objective:

1. Implement Basic Lottery and stride scheduler.
2. To print out the number of times each process in the system has been scheduled to run, using a new system call *sched_statistics()*
3. Run the user-level program with 3 different ticket values, and display the number of ticks for each program.

Steps of Execution

Setting up the System Call

- We create two new system calls, *settickets()* and *schedstatistics()* with numbers 22 and 23 in our syscall.h file, as an indication of our system calls.

syscall.h

```
// 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_settickets 22
#define SYS_schedstatistics 23
```

- This is followed by updating the system call functions in the kernel/syscall.c file, providing the function declarations.

```
extern uint64 sys_uptime(void);
extern uint64 sys_settickets(void); //settickets syscall declaration
extern uint64 sys_schedstatistics(void); //schedstatistics syscall declaration

static uint64 (*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_settickets] sys_settickets, //syscall entry
[SYS_schedstatistics] sys_schedstatistics, //schedstatistics entry
};

void
```

- Now, we define these syscall functions in the kernel/sysproc.c file. The explanation of the code is in the next section.

```
{
  uint xticks;

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

//set tickets syscall definition
uint64
sys_settickets(void)
{
    int n;
    argint(0,&n);
    allocateTickets(n);
    return 0;
}

//sched statistics syscall definition
uint64
sys_schedstatistics(void)
{
    int n;
    int prog_num;
    argint(0,&n);
    argint(1,&prog_num);
    displayStats(n,prog_num);
    return 0;
}
```

- This is followed by defining the allocateTickets() function and the displayStats() function in the kernel/proc.c file. We also add new parameters int tickets (to store the number of tickets), int pass (stores the intermediate pass for stride scheduling) and int stride (a long integer used). They are defined in the proc.c files in the two methods (explanation later).

```c
  uint64 kstack;                //
  uint64 sz;                    //
  pagetable_t pagetable;        //
  struct trapframe *trapframe;  //
  struct context context;       //
  struct file *ofile[NOFILE];   //
  struct inode *cwd;            //
  char name[16];                //

  int tickets; // Stores number o

  int pass; //pass count
  int stride; //stride size
};
```

```c

int ticksArray[NPROC]; // Store count of the ticks
int prog1id,prog2id,prog3id,pflag=0;


// Set tickets function
int allocateTickets(int n)
{
    struct proc *p = myproc();
    p->tickets=n;
    p->stride = 30000/n;
    p->pass = p->stride;
    ticksArray[p->pid]=0;
    if(n==30){
        prog1id=p->pid;
        pflag=1;
        #ifdef STRIDE
        printf("Pass for prog 1 : %d\n",p->pass);
        #endif
    }
    else if(n==20){
        prog2id=p->pid;
        pflag=1;
        #ifdef STRIDE
        printf("Pass for prog 2 : %d\n",p->pass);
        #endif
    }
    else if(n==10){
        prog3id=p->pid;
        pflag=1;
        #ifdef STRIDE
        printf("Pass for prog 3 : %d\n",p->pass);
        #endif
    }
    return 1;
}

// Scheduler statistics function
int displayStats(int n, int prog_num)
{
    if(pflag==1)
    {
       printf("Ticks in prog 1 : %d\n",ticksArray[prog1id]);
       printf("Ticks in prog 2 : %d\n",ticksArray[prog2id]);
       printf("Ticks in prog 3 : %d\n",ticksArray[prog3id]);
       printf("Total ticks : %d\n",(ticksArray[prog1id]+ticksArray[prog2id]+ticksArray[prog3id]));
       pflag=0;

    }
    return 1;
}

```

- The final step in the kernel-space syscall interface is to add the allocateTickets(int) and displayStats(int,int) function definitions in the kernel/defs.h file.

```
int             piperead(struct pipe*, uint64, int);
int             pipewrite(struct pipe*, uint64, int);

// printf.c
void            printf(char*, ...);
void            panic(char*) __attribute__((noreturn));
void            printfinit(void);

// proc.c
int         allocateTickets(int);   //settickets
int         displayStats(int,int);  //schedstatistics
int             cpuid(void);
void            exit(int);
int             fork(void);
int             growproc(int);
void            proc_mapstacks(pagetable_t);
```

- After the kernel-space, we update the user-space syscall interface. This is done by adding the settickets() and schedstatistics() entries in user/usys.pl files and also defining settickets(int) and schedstatistics(int,int) functions in user/user.h

```
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int settickets(int);
int schedstatistics(int,int);
```

```
entry("link");
entry("mkdir");
entry("chdir");
entry("dup");
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");
entry("settickets"); #settickets entry
entry("schedstatistics"); #schedstatistics entry
```

- The last additions to make in the system call files is to add the prog1.c, prog2.c and prog3.c files in user/ and also make sure to add them into the Makefile in order to execute them for our scheduling algorithm.

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int FUNCTION_SETS_NUMBER_OF_TICKETS(int a)
{
    return a;
}
int main(int argc, char *argv[])
{
    int n = FUNCTION_SETS_NUMBER_OF_TICKETS(30);    // write your own function here
    settickets(n);
    int i,k;
    const int loop=50000; // adjust this parameter depending on your system speed
    for(i=0;i<loop;i++)
    {
        asm("nop");  // to prevent the compiler from optimizing the for-loop
        for(k=0;k<loop;k++)
        {
           asm("nop");
        }
    }
    //sched_statistics(); // your syscall
    schedstatistics(n,1);
    exit(0);
}
```

```makefile
# Prevent deletion of intermediate files,
# that disk image changes after first buil
# details:
# http://www.gnu.org/software/make/manual/
.PRECIOUS: %.o

UPROGS=\
	$U/_cat\
	$U/_echo\
	$U/_forktest\
	$U/_grep\
	$U/_init\
	$U/_kill\
	$U/_ln\
	$U/_ls\
	$U/_mkdir\
	$U/_rm\
	$U/_sh\
	$U/_stressfs\
	$U/_usertests\
	$U/_grind\
	$U/_wc\
	$U/_zombie\
	$U/_prog1\
	$U/_prog2\
	$U/_prog3\
```

- Also, we are implementing the random number generator which is courtesy (https://github.com/siddharthsingh/OS/tree/master/XV6/lottery%20scheduling). We are also adding the rand.o file in the kernel space of Makefile.

syscall.h | Makefile | rand.c | usys.S | prog1.c | proc.c | usys.pl

```c
/* Copyright (C) 1997 Makoto Matsumoto and Takuji Nishimura.      */
/* Any feedback is very welcome. For any question, comments,      */
/* see http://www.math.keio.ac.jp/matumoto/emt.html or email      */
/* matumoto@math.keio.ac.jp                                       */

/* 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

static unsigned long mt[N]; /* the array for the state vector  */
static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */

/* initializing the array with a NONZERO seed */
void
sgenrand(unsigned long seed)
{
    /* setting initial seeds to mt[N] using         */
    /* the generator Line 25 of Table 1 in          */
    /* [KNUTH 1981, The Art of Computer Programming */
    /*    Vol. 2 (2nd Ed.), pp102]                  */
    mt[0]= seed & 0xffffffff;
    for (mti=1; mti<N; mti++)
        mt[mti] = (69069 * mt[mti-1]) & 0xffffffff;
}

long /* for integer generation */
genrand()
{
    unsigned long y;
    static unsigned long mag01[2]={0x0, MATRIX_A};
    /* mag01[x] = x * MATRIX_A  for x=0,1 */

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1)   /* if sgenrand() has not been called, */
            sgenrand(4357); /* a default initial seed is used   */
```

Explanation of the Code

We created the function allocateTickets(int n), which receives an integer n and it assigns 30 tickets to program prog1, 20 tickets to program prog2, and 10 tickets to program prog3. As far as the other processes are concerned, they are given a default value of 10 in the allocproc() function of kernel/proc.c file. We also declared an array of tickArray[] which stores the tick values in the different processes, and will be used to display the scheduler statistics. Also we store the processes ids of prog1,2, and 3 and three id variables to use them in the scheduler statistics function. We are also calculating the pass value, by dividing the stride (30000) with the number of tickets. So the pass is inversely proportional to the number of tickets.

In the displayStats(int n,int prog_num) function, we are calculating the ticks used in each program and calculating its sum, to display to the user as output. The pflag variable is a status variable indicating the program is in execution. Initially we are giving the ticket ratio as 3:2:1 which eventually turns out as a ratio (½):(⅓):(⅙) with respect to the total number of ticks.

syscall.h | proc.c | proc.h | sysproc.c | syscall.c

```c
int ticksArray[NPROC]; // Store count of the ticks
int prog1id,prog2id,prog3id,pflag=0;


// Set tickets function
int allocateTickets(int n)
{
    struct proc *p = myproc();
    p->tickets=n;
    p->stride = 30000/n;
    p->pass = p->stride;
    ticksArray[p->pid]=0;
    if(n==30){
        prog1id=p->pid;
        pflag=1;
        #ifdef STRIDE
        printf("Pass for prog 1 : %d\n",p->pass);
        #endif
    }
    else if(n==20){
        prog2id=p->pid;
        pflag=1;
        #ifdef STRIDE
        printf("Pass for prog 2 : %d\n",p->pass);
        #endif
    }
    else if(n==10){
        prog3id=p->pid;
        pflag=1;
        #ifdef STRIDE
        printf("Pass for prog 3 : %d\n",p->pass);
        #endif
    }
    return 1;
}

// Scheduler statistics function
int displayStats(int n, int prog_num)
{
    if(pflag==1)
    {
       printf("Ticks in prog 1 : %d\n",ticksArray[prog1id]);
       printf("Ticks in prog 2 : %d\n",ticksArray[prog2id]);
       printf("Ticks in prog 3 : %d\n",ticksArray[prog3id]);
       printf("Total ticks : %d\n",(ticksArray[prog1id]+ticksArray[prog2id]+ticksArray[prog3id]));
       pflag=0;

    }
    return 1;
}
```

Lottery Scheduling

The scheduler function has two for loops, one inside the other. The outer loop runs forever. In the Round Robin algorithm the inside for loop iterates over all the processes and chooses the first process in a RUNNABLE state. It then runs this process till its time quanta expire or the process yields voluntarily. It then selects the next process in RUNNABLE state and so on.

For lottery scheduling, we need the total number of tickets of the processes that are in the RUNNABLE state. So, to calculate this we use a for loop inside the outer for loop, before the inside for loop executes, which calculates the total number of tickets. This is followed by generating a random number between 0 and the total number of tickets. On getting the random number we execute the for loop that runs processes. When this for loop loops over processes, we keep counting the total number of tickets passed or processed that are in a RUNNABLE state. As soon as the total tickets passed get higher than the random number, we run that process. Lastly, after the process runs, the break at the end of the for loop which executes all processes, makes sure the total tickets passed does not exceed the random number. Also, after we run a winning process we need to recompute the total number of tickets of all RUNNABLE processes as that value might have changed. We also add locks to avoid situations like race conditions.

```c
c->proc = 0;

for(;;)
{
    intr_on();
    #ifdef LOTTERY
    struct proc *p;
    int totalTickets = 0 ;
    for(p=proc;p<&proc[NPROC];p++)
    {
      if(p->state==RUNNABLE){
        totalTickets+= p->tickets;
      }
    }
    int winner = random_at_most(totalTickets);

    int temp=0;
    for(p=proc;p<&proc[NPROC];p++)
    {
      if(p->state==RUNNABLE)
        temp+=p->tickets;
      if(temp>winner)
      {
        acquire(&p->lock);
        p->state = RUNNING;
        c->proc = p;
        ticksArray[p->pid]+=1;
        swtch(&c->context,&p->context);

        c->proc=0;
        release(&p->lock);

        break;
      }
    }
    #endif
```

```
bhagy_linux@ITSloan-J09VNTK: ~/lottery_stride/xv6-riscv
qemu-system-riscv64 -machine virt -bios none -kernel ker
format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=

xv6 kernel is booting

init: starting sh
$ prog1&;prog2&;prog3
Ticks in prog 1 : 30
Ticks in prog 2 : 16
Ticks in prog 3 : 9
Total ticks : 55
$
```

Stride Scheduling

For stride scheduling, again we need to modify the void scheduler() function in the proc.c file. We first select the process, which has the lowest pass value. In the following for loop, once we reach the minimum pass process, we add the stride size to the pass counts and add spinlocks to avoid race conditions between concurrent processes. After context switching, the lock is released, thus ending the critical section. Lastly, the break jumps out of the loop to proceed to another round of stride scheduling.

proc.c | Makefile | rand.c | usys.S | prog1.c | usys.pl

```c
    #endif
    #ifdef STRIDE
    struct proc *p,*current_proc;
    int minPass = -1;

    for(p=proc;p<&proc[NPROC];p++){
        if(p->state == RUNNABLE &&(p->pass <= minPass || minPass<0))
        {
            minPass = p->pass;
            current_proc = p;
        }
    }

    for(p=proc; p<&proc[NPROC];p++){
        if(p->state!=RUNNABLE){
            continue;
        }
        if(p->pass == minPass){
            acquire(&p->lock);
            current_proc=p;
            c->proc=current_proc;
            current_proc->pass+=current_proc->stride;
            current_proc->state=RUNNING;
            ticksArray[current_proc->pid]+=1;
            swtch(&c->context,&current_proc->context);
            c->proc=0;

            release(&p->lock);
            break;
        }
    }
    #endif
}
```

bhagy_linux@ITSloan-J09VNTK: ~/lottery_stride/xv6-ri

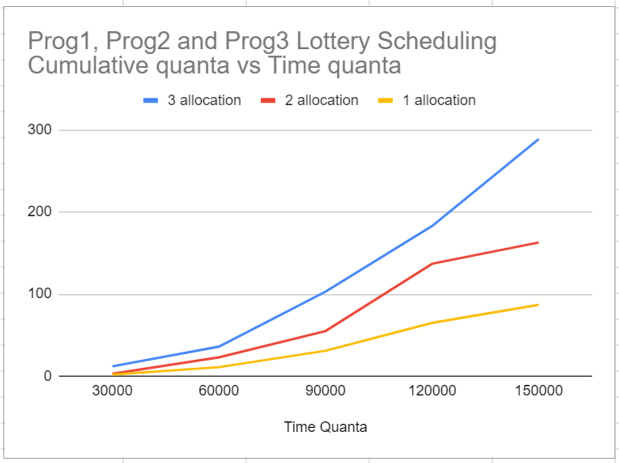
```
format=raw,id=x0 -device virtio-blk-device

xv6 kernel is booting

init: starting sh
$ prog1&;prog2&;prog3
Pass for prog 3 : 3000
Pass for prog 1 : 1000
Pass for prog 2 : 1500
Ticks in prog 1 : 32
Ticks in prog 2 : 21
Ticks in prog 3 : 10
Total ticks : 63
$
```

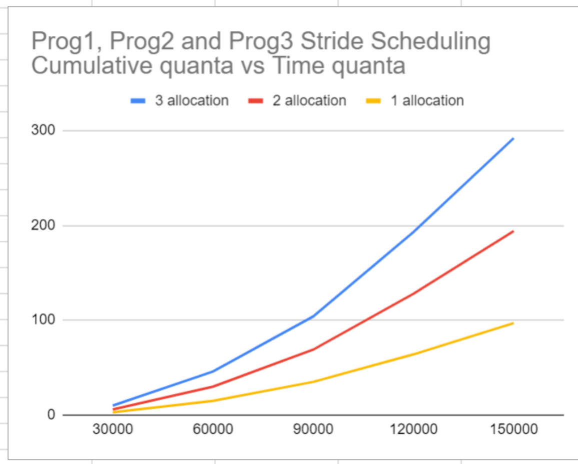
Graphs similar to Figure 8 in paper :

Below are the values used that we obtained after running the lottery and stride scheduling :

		LOTTERY				STRIDE	
Time Quanta	Prog1	Prog2	Prog3	Time Quanta	Prog1	Prog2	Prog3
30000	12	3	2	30000	10	6	3
60000	36	23	11	60000	46	30	15
90000	103	55	31	90000	104	69	35
120000	183	137	65	120000	193	128	64
150000	289	163	87	150000	292	194	97

Their corresponding graphs :



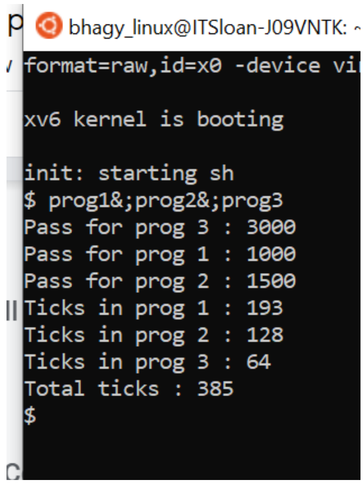


We change the loops in time quanta from 30000 to 150000 in intervals of 30000, and plot the corresponding tick values to get the same result. The stride scheduling graph has a regular nature because of the deterministic nature, while the lottery graph can turn a little skewed due to the probabilistic nature.

Eg of input in prog files :

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int FUNCTION_SETS_NUMBER_OF_TICKETS(int a)
{
    return a;
}
int main(int argc, char *argv[])
{
    int n = FUNCTION_SETS_NUMBER_OF_TICKETS(10);    // write your own functic
    settickets(n);
    int i,k;
    const int loop=120000; // adjust this parameter depending on your system
    for(i=0;i<loop;i++)
    {
        asm("nop");  // to prevent the compiler from optimizing the for-loop
        for(k=0;k<loop;k++)
        {
           asm("nop");
        }
    }
    //sched_statistics(); // your syscall
    schedstatistics(n,3);
    exit(0);
}
```

Eg of o/p after changing const int loop in prog files :



Contributions of Group Members

Bhagyesh Ravindra Gaikwad

- Implementation of lottery scheduling.
- Screenshots of program files and outputs in the project report.
- Report writing- Objectives and Lottery Scheduling

Dipro Chakraborty

- Implementing the stride scheduling algorithm.
- Recording the youtube video (https://www.youtube.com/watch?v=in1sBJYs7FM&t=20s&ab_channel=DiproChakraborty).
- Report writing- Setting up the system call and Stride scheduling

Figure 8 implementation was done by both.

****