CS202 Lab 2 Report

Md Abdullah Al Mamun *mmamu003@ucr.edu*
Yicheng Zhang *yzhan846@ucr.edu*

1 THE LIST OF ALL FILES MODIFIED

In this project, the files that we modified are listed below.

- kernel/syscall.h: Define new system call number.
- kernel/syscall.c: Update system call table.
- kernel/sysproc.c: Define system call function.
- kernel/proc.h: Declare a variable named Tickets, Stride, Pass within Proc structure to keep track for each process's tickets, stride size and pass counts.
- kernel/proc.c: Create a new kernel function for allocate tickets within **givetickets** function and for showing the ticks statistics for three programs in **showStatistics** kernel function. Also, we add lottery and stride scheduling algorithms into the scheduler() function.
- kernel/defs.h: Create a new system call.
- user/usys.pl and user/user.h: Update user-space syscall interface.
- user/test.c: Create " test.c" file in the user directory to test functionality.
- Makefile: Append UPROGS in Makefile.

2 EXPLANATION ON WHAT CHANGES WE HAVE MADE IN FILES AND THE SCREENSHOTS

In the first step, we define define new system call number 24 and system call number 25 to our system call — SYS_sched_statistics and SYS_allocateTickets in "`kernel/syscall.h`" and update system call table in "`kernel/syscall.c`", which is shown in figure 1 and 2.

```
18  #define SYS_mknod  17
19  #define SYS_unlink 18
20  #define SYS_link   19
21  #define SYS_mkdir  20
22  #define SYS_close  21
23  #define SYS_hello  22 // hello
24  #define SYS_lab1   23 // First lab
25  #define SYS_sched_statistics   24 // second Lab
26  #define SYS_allocateTickets   25 // second lab
```

Figure 1: Syscall.h

```
134  [SYS_mkdir]   sys_mkdir,
135  [SYS_close]   sys_close,
136  [SYS_hello]   sys_hello,   // hello: system call entry
137  [SYS_lab1]    sys_lab1, // lab1: system call entry
138  [SYS_sched_statistics]   sys_sched_statistics, // lab2: System call entry
139  [SYS_allocateTickets]   sys_allocateTickets,  // lab2 : syscall
140  };
141
```

Figure 2: Syscall.c

In the next step, we firstly define the lab 2 system call function — sys_sched_statistics(void) and sys_allocateTickets(void) in the "`kernel/sysproc.c`", which is shown in figure 3. In "`kernel/proc.c`", we program the int giveTickets(int n) and int showStatistics(int n,int programNum), shown in figure 4.The function named giveTickets just allocate tickets to the prog1.c, prog2.c and prog3.c in the 3:2:1 ratio correspondingly. Also, we update "`kernel/defs.h`" with int giveTickets(int n) and int showStatistics(int n,int programNum), which is shown in figure 5.

```
uint64 sys_sched_statistics(void) // lab 2 syscall definition
{
  int n;
  int programNum;
  argint(0, &n);
  argint(1, &programNum);
  showStatistics(n,programNum);
  return 0;
}


uint64 sys_allocateTickets(void)
{
  int n;
  argint(0, &n);
  giveTickets(n);
  return 0;
}
```

Figure 3: sysproc.c

```
54  int showStatistics(int n,int programNum)
55  {
56
57      if(progFlag==1)
58      {
59      int temp=0;
60      temp = ticksCount[prog1ID]+ticksCount[prog2ID]+ticksCount[prog3ID];
61      printf("\ntotal number of ticks by all 3 programs: %d\nTicks value of each program\n",temp);
62      printf("Prog1 : %d \n",ticksCount[prog1ID]);
63      printf("Prog2 : %d \n",ticksCount[prog2ID]);
64      printf("Prog3 : %d \n",ticksCount[prog3ID]);
65      progFlag = 0;
66      }
67    return 1;
68  }
69
70  int giveTickets(int n)
71  {
72      struct proc *p;
73      p = myproc();
74      p->tickets = n;
75      ticksCount[p->pid] = 0;
76      if(n==30)prog1ID = p->pid,progFlag=1;
77      else if(n==20)prog2ID = p->pid,progFlag = 1;
78      else if(n==10)prog3ID = p->pid,progFlag=1;
79    return 1;
80  }
```

Figure 4: proc.c

```
106  void            procdump(void);
107  void            print_hello(int); //hello
108  int             info(int); // lab1_function
109  int             showStatistics(int, int); //lab2_function
110  int             giveTickets(int); //lab2 function
111
```

Figure 5: defs.h

In the third step, we update user-space syscall interface with new lab2 system call, which is shown in figure 6 and 7.

```
38  entry("uptime");
39  entry("hello"); # hello syscall for user
40  entry("lab1"); # lab 1 syscall for user
41  entry("sched_statistics"); #lab 2 system call
42  entry("allocateTickets"); #lab 2 system call
```

Figure 6: usys.pl

```
int hello(int); // hello
int lab1(int); // lab1
int sched_statistics(int,int); // lab2
int allocateTickets(int); //lab 2
```

Figure 7: user.h

The last step is to create a "`user/prog1.c`","`user/prog2.c`" and "`user/prog3.c`" file in the user directory to show functionality of lab2 system call. Also, Makefile is needed to be modified to gurantee it can be compiled. The modification is shown in figure 8 and 9.

```
int FUNCTION_SETS_NUMBER_OF_TICKETS(int a)
{
  return a/2;
}

int main(int argc,  char *argv[])
{
    int n = FUNCTION_SETS_NUMBER_OF_TICKETS(60); // write your own function here
    allocateTickets(n);
    printf("Prog1.c has been allocated %d Tickets\n",n);
    int i,k;
    const int loop=100000; // 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");
    }
    }
    //printf("%s is performing a system call\n",argv[0]);
    sched_statistics(n,1); // your syscall
    exit(0);
}
```

Figure 8: prog1.c

```
    $U/_sh\
    $U/_stressfs\
    $U/_usertests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_test\
    $U/_prog1\
    $U/_prog2\
    $U/_prog3\
```

Figure 9: Makefile

3 EXPLANATION ON WHAT CHANGES WE HAVE MADE IN CODE IMPLEMENTATION

We had to define two new system call function sys_allocateTickets to allocate the tickets to prog1.c, prog2.c and prog3.c into 3:2:1 ratio tickets and the function named sys_sched_statistics was used to show the statistics of the ticks used by those three user programs (i.e., prog1.c, prog2.c and prog3.c). sys_allocateTickets calls another kernel function named **giveTickets in Kernel/proc.c** to allocate the tickets to those user programs. Then sys_sched_statistics calls another kernel function named **showStatistics in Kernel/proc.c** to show the number of ticks used by per process with showing total ticks used by those three processes. To calculate the allocated ratio per process: (number of ticks per program) / (total number of ticks by all 3 programs) Approximately prog1 ratio was like 1/2, prog2 ratio was like 1/3 and prog3 was like 1/6.

3.1 Process of implementing the lottery Scheduling

To implement this functionalities, we have used the function named " `int giveTickets(int n)`" in " `Kernel/proc.c`". The function receives n (number of ticket for that user process) as its parameter then it allocate those tickets to that corresponding process. We have declared an extra variable named tickets in Process structure to keep track the number of each process's tickets in Xv6 [1] code. Then we allocate 10 tickets for all other programs except the processes prog1.c, prog2.c and prog3.c. We set other process tickets count to 10 inside the allocproc function in " `Kernel/proc.c`". We also declared an array for keeping the track of ticks for each process named int ticksCount[NPROC] which we use to provide the statistics of used ticks for those 3 processes at the end. We were instructed to allocate the tickets to the prog1.c, prog2.c and prog3.c into 3:2:1 ratio. So, based on the criteria we track each process using their ID later on. That is why we saved those ID into a variable to track later on using their ticket allocation pattern, which is shown in figure 10. Finally, we use the function named showStatistics. As we Run 3 test programs (prog1.c - 30 tickets, prog2.c - 20 tickets and prog3.c - 10 tickets), each with a different number of tickets. Whenever one of these processes finishes execution, the system call prints the number of times these processes have been scheduled to run. Our output also shows total ticks used by those processor, shown in figure 14

```
int showStatistics(int n,int programNum)
{

    if(progFlag==1)
    {
    int temp=0;
    temp = ticksCount[prog1ID]+ticksCount[prog2ID]+ticksCount[prog3ID];
    printf("\ntotal number of ticks by all 3 programs: %d\nTicks value of each program\n",temp);
    printf("Prog1 : %d \n",ticksCount[prog1ID]);
    printf("Prog2 : %d \n",ticksCount[prog2ID]);
    printf("Prog3 : %d \n",ticksCount[prog3ID]);
    progFlag = 0;
    }
  return 1;
}

int giveTickets(int n)
{
    struct proc *p;
    p = myproc();
    p->tickets = n;
    ticksCount[p->pid] = 0;
    if(n==30)prog1ID = p->pid,progFlag=1;
    else if(n==20)prog2ID = p->pid,progFlag = 1;
    else if(n==10)prog3ID = p->pid,progFlag=1;
  return 1;
}
```

Figure 10: implementation of lottery scheduling

To implement the lottery scheduling algorithm within scheduler function of Xv6 [1] codebase by replacing the round robin algorithm for process scheduling, we at first commented out the existing ROUND Robin algorithm. Two for loops are contained within the scheduler function. The outer for loop never stops. The inside for loop iterates through all processes and selects the first one that is in the RUNNABLE state. It then continues running this process until the process's time quanta expire or the process yields voluntarily. It then selects the next RUNNABLE process and continues in this manner. Round Robin is implemented in this manner. We require the total number of tickets issued by all processes in the RUNNABLE state for lottery scheduling purposes. To calculate this, we insert a for loop just before the inner for loop executes. This function determines the total number of tickets available. Now, between 0 and the total number of

tickets, we generate a random number. We used the random generator given in the link: https://github.com/avaiyang/xv6-lottery-scheduling/blob/master/rand. We were careful that the random number generator is not biased and provides the number in equal distribution. After obtaining the random number, the for loop is used to execute the processes. While this for loop is iterating over processes, we keep track of the total number of tickets passed or processed that are in an RUNNABLE state. We repeat that process as soon as the total number of tickets passed exceeds the random number we receive. Now that the process has completed, we need to add a break to the for loop that runs all processes. Due to the following. If we do not break, the total number of tickets sold will continue to grow and will always exceed the random number. Additionally, after running a successful process, we must recalculate the total number of tickets issued by all RUNNABLE processes, as this value may have changed.

3.2 Process of implementing the Stride Scheduling

To implement stride scheduling, we modify the original round-robin scheduling algorithm declared within the scheduler function in the file " `Kernel/proc.c`". Firstly, we use one for loop to select the process with the lowest pass counts, which is shown in line 737 to line 742 in figure 11. After that, we keep the pass count of selected process into variable " `minPass`" and struct " `current`". Secondly, another for loop is utilized to switch the selected process into the CPU to execute. This step compares the pass counts of all available processes with the selected pass count recorded in variable " `minPass`". If the process with the lowest pass count is switched into CPU to execute, we add stride size to the pass count of this process. We implement a spinlock inside the second loop function to avoid race conditions between different concurrent processes. The spinlock is implemented using acquire() and release() functions. When context switching is finished, the release function will release the lock, and the critical section will end. Finally, when the scheduling is done, the break function will jump out the for loop and do another round of stride scheduling.

```
733   #ifdef STRIDE
734   struct proc *p = ptable.proc, *current = ptable.proc;
735   int minPass = -1;
736   // acquire(&p->lock);
737   for(p = proc; p < &proc[NPROC]; p++){
738     if(p->state == RUNNABLE && (p->pass <= minPass || minPass < 0)){
739         minPass = p->pass;
740         current = p;
741     }
742   }    // loop all processes to find process with lowest pass.
743
744   for(p = proc; p < &proc[NPROC]; p++){
745     if(p->state != RUNNABLE){
746       continue;
747     }
748     if(p->pass == minPass){
749       acquire(&p->lock);
750
751       current = p;
752       c->proc = current;
753       current->pass += current->stride;  // pass += its stride size
754       current->state = RUNNING;      //switch to this process.
755       ticksCount[current->pid]+=1;   //record one-time scheduing for this process.
756       swtch(&c->context, &current->context);
757       c->proc = 0;
758
759       release(&p->lock);
760       break;
761     }
762   }
763   #endif
```

Figure 11: Implementation of stride scheduling

4 RESULTS OF LOTTERY SCHEDULING

After implementing all of the functionalities that we were asked to implemented, we simulate the OS by using the lottery scheduling as a scheduler. The output is given below in figure 12, figure 13 and figure 14.

```
mamun@LAPTOP-9O07R6MT:/mnt/c/Users/Md Abdullah Al Mamun/Desktop/xv6-riscv$ make qemu
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -MD -mcmodel=medany -ffreestanding -fno-common -nos
tdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie   -c -o kernel/proc.o kernel/proc.c
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -MD -mcmodel=medany -ffreestanding -fno-common -nos
tdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie   -c -o kernel/syscall.o kernel/syscall.c
riscv64-linux-gnu-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel kernel/entry.o kernel/start.o kernel/con
sole.o kernel/printf.o kernel/uart.o kernel/kalloc.o kernel/spinlock.o kernel/string.o kernel/main.o kernel/vm.o kernel/
proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o kernel/fs.o kerne
l/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o k
ernel/virtio_disk.o
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1 -nographic -drive file=fs.img,if=none,
format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting
```

Figure 12: After starting the qemu simulator

```
xv6 kernel is booting

init: starting sh
$ prog1&;prog2&;prog3
Prog1.c has been allocated 30 Tickets
Prog3.c has been allocated 10 Tickets
Prog2.c has been allocated 20 Tickets
```

Figure 13: Ticket allocation in lottery Scheduling

```
total number of ticks by all 3 programs: 127
Ticks value of each program
Prog1 : 68
Prog2 : 34
Prog3 : 25
$
```

Figure 14: Number of ticks used per process

5 RESULTS OF STRIDE SCHEDULING

We simulate the stride scheduling function in the qemu provided by xv6. The result is shown in Figure 15. As the tickets values shown for each program, the program 1, 2 and 3 got 266, 177 and 88 ticket values of total 531. Their values follows the number of allocated tickets, which is 3:2:1.

```
$ prog1&;prog2&;prog3
Prog1.c has been allocated 30 Tickets
Prog2.c has been allocated 20 Tickets
Prog3.c has been allocated 10 Tickets

total number of ticks by all 3 programs: 531
Ticks value of each program
Prog1 : 266
Prog2 : 177
Prog3 : 88
```

Figure 15: Result of Stride Scheduling

6 COMPARISON FIGURES BETWEEN LOTTERY SCHEDULING AND STRIDE SCHEDULING.

In this section, we represent the comparison between Lottery scheduling and Stride Scheduling. The figure 16 and figure 17 illustrate the simulation results for three clients, Prog1, Prog2 and Prog3, with a 3:2:1 allocation. It is clear from the figure 16 that lottery scheduling exists inherent variability at this time scale, due to the algorithm's inherent use of randomization. However, the figure 17 indicates that stride scheduling algorithm is deterministic and produces precise periodic behavior.

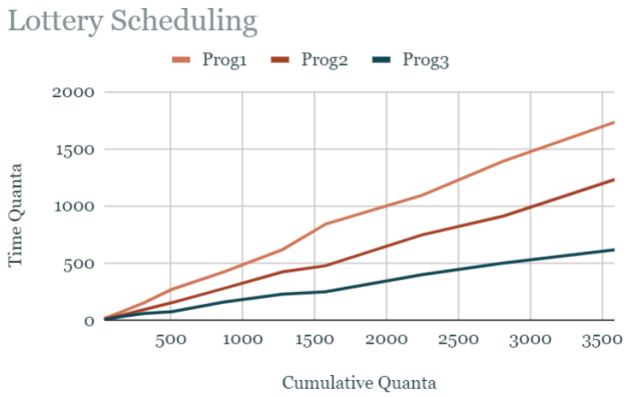

Figure 16: Simulation results for Lottery Scheduling involving three clients

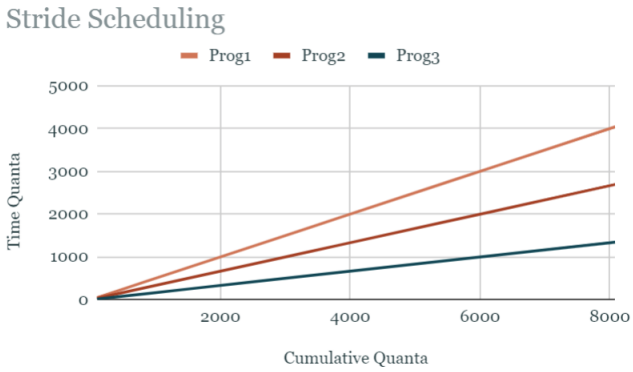

Figure 17: Simulation results for Stride Scheduling involving three clients

The figure 18 and figure 19 plots the absolute error that simulates lottery and stride scheduling algorithms for two clients, prog1 and prog3, with a 3:1 allocation. As figure 18 represented, the absolute error increase linearly without limitation. In contrast, the figure 19 indicates that error for stride scheduling never exceeds a single quantum, which follows a deterministic pattern.

7 HOW WE PRODUCE THOSE FIGURES

In this section, we illustrate how to generate figure 16, figure 17, figure 18 and figure 19. In order to figure 16 and figure 17, we allocate tickets to three program with the ratio of 30:20:10. Then, we change the number of loops ranging from 50000 to 500000, as

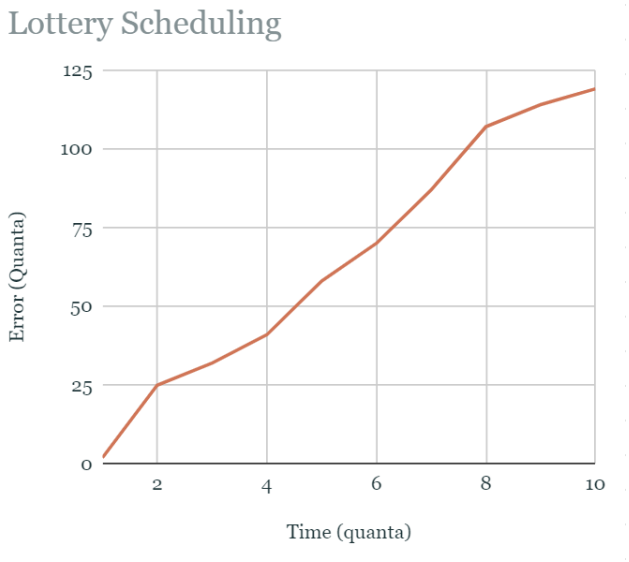

Figure 18: Simulation results for Lottery Scheduling involving two clients

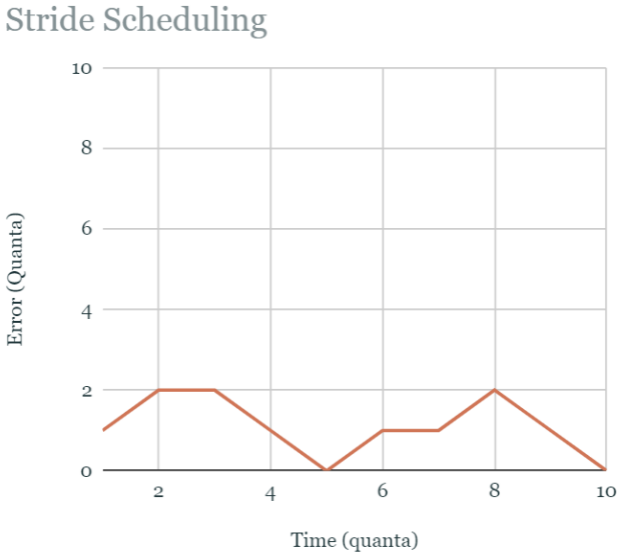

Figure 19: Simulation results for Stride Scheduling involving two clients

the line 22 in figure 20 shows. Next step is to record the ticket values for each program. For figure 18 and figure 19, two programs are scheduled into the qemu simulator. However, their ticket allocation are set to be 3:1. After the ticket value collection, we calculate the mean error for lottery scheduling and error for stride error.

```
14  int main(int argc,  char *argv[])
15  {
16
17
18      int n = FUNCTION_SETS_NUMBER_OF_TICKETS(60); // write your own function here
19      allocateTickets(n);
20      printf("Prog1.c has been allocated %d Tickets\n",n);
21      int i,k;
22      const int loop=200000; // adjust this parameter depending on your system speed
23      for(i=0;i<loop;i++)
24      {
```

Figure 20: Setting different time for quantitative experiments.

8 CONTRIBUTIONS OF EACH MEMBER.

The contributions of Md Abdullah Al Mamun are listed below.

- Implementation of Lottery scheduling algorithm code base.
- Report writing, including section 1.2 and section 3.1 and part of section 6.
- Collecting the data for lottery scheduling for producing the figures for comparison
- Drawing the Figures 8 of the paper for both Lottery Scheduling and Stride Scheduling anf Figure 9 for Lottery Scheduling

The contributions of Yicheng Zhang are listed below.

- Implementation of stride scheduling algorithm code base.
- Report writing, including section 3.2 and section 5, 6 and 7.
- Collecting data for stride algorithm and generating figures in section 6.
- Demo video recording.

REFERENCES

[1] R. Cox, M. F. Kaashoek, and R. Morris. Xv6, a simple unix-like teaching operating system, 2011.