CS202 Lab 2 Report

Yingfan Gao
ygao195@ucr.edu
UC Riverside

Yufei Li
yli927@ucr.edu
UC Riverside

Shahab Nikkhoo
snikk002@ucr.edu
UC Riverside

1 THE LIST OF ALL FILES MODIFIED

In this assignment, we have modified the following files:

- kernel/syscall.h: Define new system call numbers.
- kernel/syscall.c: Update system call table.
- kernel/sysproc.c: Define system call function.
- kernel/proc.c: Create new kernel function.
- kernel/proc.h: Create new process states such as stride, tickets for scheduling.
- kernel/defs.h: Create new system call.
- user/usys.pl and user/user.h: Update userspace system call interface.
- user/lab2.c: Write a test program.
- Makefile: Lab2 operating with different scheduling algorithms.

2 EXPLANATION ON WHAT CHANGES WE MADE

Similar to the previous assignment, we first define new system call #25 [SYS_sched_statistics] and #26 [SYS_sched_tickets] (see Figure 1), and then update the corresponding system call table (see Figure 2).

```
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_hello  22
#define SYS_sysinfo 23 //lab1
#define SYS_procinfo 24 //lab1
#define SYS_sched_statistics 25 //lab2
#define SYS_sched_tickets 26 //lab2
```

Figure 1: Syscall.h

```
[SYS_close]   sys_close,
[SYS_hello]   sys_hello,
[SYS_sysinfo] sys_info, // lab1
[SYS_procinfo] sys_procinfo, // lab1
[SYS_sched_statistics] sys_sched_statistics, // lab2
[SYS_sched_tickets] sys_sched_tickets, // lab2
```

Figure 2: Syscall.c

Next, we defined system call function [sys_sched_statistics(void)] as well as [sys_sched_tickets(void)] and program it in "kernel/proc.c" (as shown in Figure 3, Figure 4 and Figure 5).

```
uint64
sys_sched_statistics(void) //lab2
{
  sched_statistics();
  return 0;
}


uint64
sys_sched_tickets(void) //lab2
{
  int tickets;
  // int ticks;
  argint(0, &tickets);
  // argint(0, &ticks);
  sched_tickets(tickets);
  return 0;
}
```

Figure 3: Sysproc.c

```
int sched_statistics(void)
{
  struct proc * p = myproc();
  // acquire(&p->lock);
  for(p = proc; p < &proc[NPROC]; p++) {
    if(p->state != UNUSED)
    {
      if(p->tickets != 60) // child process
      {
        printf("%d(%s): tickets: %d, ticks: %d\n",
        p->pid, p->name, p->tickets, p->ticks);
      }
      if (p->tickets == 60) // parent process
      {
        printf("%d(%s): tickets: xxx, ticks: %d\n", p->pid, p->name, p->ticks);
      }
    }
  }
  // release(&p->lock);
  return 0;
}

// This system call sets the caller process's ticket value to the given parameter.
int sched_tickets(int tickets)
{
  struct proc * p = myproc();
  // acquire(&p->lock);
  p->tickets = tickets;
  #ifdef STRIDE
  p->stride = 10000/tickets;
  #endif
  // release(&p->lock);
  return 0;
}
```

Figure 4: Proc.c (system call)

```
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID
  int tickets;                 // Number of tickets for lottery scheduler
  int ticks;                   // Number of time slices for lottery scheduler
  #ifdef STRIDE
  int stride;                  // Stride for stride scheduler
  int pass;                    // Pass for stride scheduler
  #endif
```

Figure 5: Proc.h

Also, we add these functions into "kernel/defs.h" that enable the new created system call available. Then, we update userspace system call interface with the new system calls (as shown in Figure 6, Figure 7, and Figure 8).

```
int            sysinfo(int); //lab1
extern uint    systemcallcount;
int            procinfo(struct pinfo*); //lab1
int            sched_statistics(void); //lab2
int            sched_tickets(int); //lab2
unsigned short rand(void); //lab2
```

Figure 6: Defs.h

```
entry("sysinfo"); # lab1
entry("procinfo"); # lab1
entry("sched_statistics"); # lab2
entry("sched_tickets"); # lab2
```

Figure 7: Usys.pl

```
int hello(void);
int sysinfo(int); //lab1
int procinfo(struct pinfo*); //lab1
int sched_statistics(void); //lab2
int sched_tickets(int); //lab2
```

Figure 8: User.h

The two scheduling algorithms, i.e., lottery scheduling and stride scheduling are shown in Figure 9 and Figure 10, respectively.

```
// Do lottery scheduling
#ifdef LOTTERY
// Get total number of tickets
int total_tickets = 0;
for(p = proc; p < &proc[NPROC]; p++){
  acquire(&p->lock);
  if(p->state == RUNNABLE){
    total_tickets += p->tickets;
  }
  release(&p->lock);
}
int winner = 1 + ((int)rand() % total_tickets);
int current = 0;
// Find the scheduled process
for(p = proc; p < &proc[NPROC]; p++){
  acquire(&p->lock);
  if (p->state != RUNNABLE) {
    release(&p->lock);
    continue;
  }
  current += p->tickets;
  if(current < winner){
    release(&p->lock);
    continue;
  }
  p->state = RUNNING;
  c->proc = p;
  p->ticks += 1; // Increment ticks
  swtch(&c->context, &p->context);
  c->proc = 0;
  release(&p->lock);
  break;
}
#endif
```

Figure 9: Proc.c (Lottery scheduling)

For lottery scheduling, we first statistic the total number of tickets by iterating all runnable processes. Then we generate a winning score using the rand() function (provided in Part2 pseudo code). The accumulative tickets for a process, if larger than the winning score, would indicate that process is scheduled to be executed. We then add one more time slice to its ticks and implement context switch, with the lock then being released.

```
#ifdef STRIDE
// Do stride scheduling
int current = 100000;
struct proc *chosen = proc; // Pointer to the chosen process
// Find the scheduled process
for(p = proc; p < &proc[NPROC]; p++){
  acquire(&p->lock);
  if (p->state != RUNNABLE) {
    release(&p->lock);
    continue;
  }
  if (p->pass < current) {
    current = p->pass;
    chosen = p;
  }
  release(&p->lock);
}
acquire(&chosen->lock);
chosen->state = RUNNING;
c->proc = chosen;
chosen->ticks += 1; // Increment ticks
chosen->pass += chosen->stride; // Increment pass
swtch(&c->context, &chosen->context);
c->proc = 0;
release(&chosen->lock);
#endif
```

Figure 10: Proc.c (Stride scheduling)

For stride scheduling, we use a process pointer "chosen"" to keep track of all runnable processes and point at the one with minimum pass (cummulative stride). Then, we add one more time slice to that pointer and implement context switch, with the lock then being released.

3 EXPERIMENT FIGURES AND DISCUSSION

In this section, we represent the comparison between Lottery scheduling and Stride Scheduling. We plot lottery scheduling (Figure 11 and Figure 12), and stride scheduling (Figure 13 and Figure 14), both under two time allocations (8:4:2:1 and 1:1:1:1).

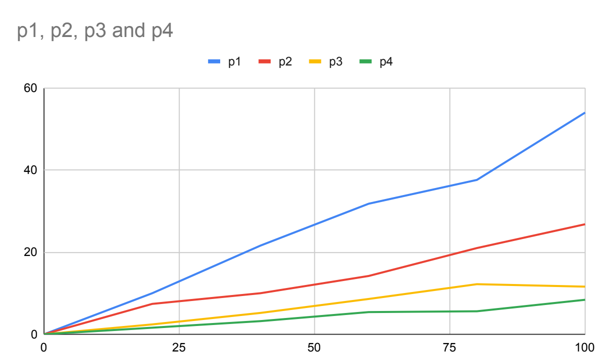

Figure 11: Lottery scheduling (8-4-2-1)

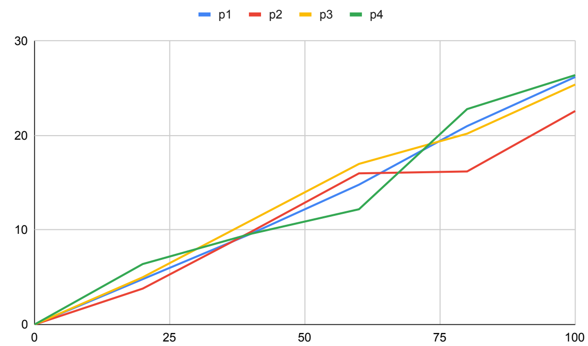

Figure 12: Lottery scheduling (1-1-1-1)

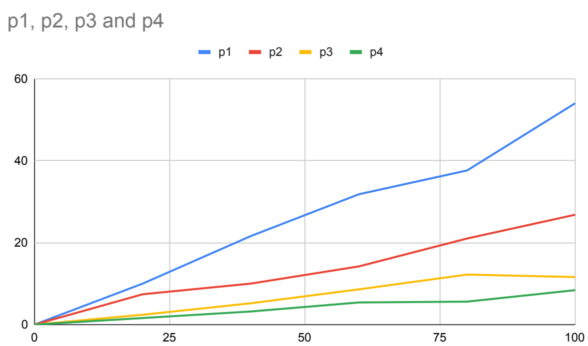

Figure 13: Stride scheduling (8-4-2-1)

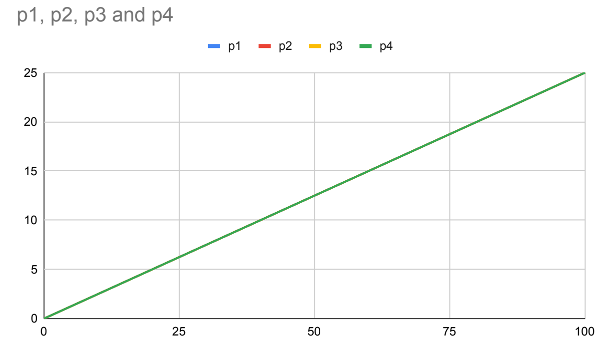

Figure 14: Stride scheduling (1-1-1-1)

From the above figures, we can see that stride scheduling is much more deterministic than lottery scheduling. For each scheduling algorithm, we run the two experiments for 5 times to get the average.

4 SUMMARY OF CONTRIBUTIONS

In this lab assignment, Yufei took charge of the first part of the lab, while Shahab completed the second part. And Yingfan was responsible for the writting of lab report.