CS 202 Lab 1

Nafis Mustakin - nmust004@ucr.edu

May 2021

Task 1

The system call info(int) was implemented as bellow -

```
//Get system info
//param = 1: return count of processes in the system
//param = 2: return count of total system calls made so far
//param = 3: return number of memory pages being used by the current process

int
info(int param)
{
  int count = 0;
  struct proc *p;
  switch(param)
  {
    case 1:
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
          if(p->state != UNUSED){
            count++;
          }
        }
        release(&ptable.lock);
        break;
    case 2:
        p = myproc();
        count = p->nosyscall;
        break;
    case 3:
        p = myproc();
        count = (int)(p->sz/PGSIZE);
        if(p->sz % PGSIZE) count++;
        break;
    default:
        cprintf("Unrecognised parameter\n");
        break;
  }
  return count;
}
```

Here we introduce a new variable *nosyscall* in *structproc* which keeps track of the number of system calls made by the process. This variable is updated whenever a system call is made from the *syscall* method in the *syscall.c* file. The update is a simple increment as below -

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();
```

```
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
    curproc->nosyscall++; //update number of system calls made
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

I implemented a *demo.c* program which calls the info() function a number of times with different parameters. The program can be found in Appendix A. The output is shown here-

```
checkpoint1
checkpoint2
number of processes:3

checkpoint3
page tables of current process:50

checkpoint3a
page tables of current process:98

checkpoint3b
page tables of current process:147

checkpoint3c
page tables of current process:197

checkpoint4
system calls made by current process:4

checkpoint4a
system calls made by current process:4

checkpoint4b
system calls made by current process:4

checkpoint4c
system calls made by current process:4


checkpoint5
page tables of current process:521, pid:3

checkpoint6
system calls made by current process:4

Child checkpoint7
page tables of child process:0, pid:0

Child checkpoint7a
page tables of child process:58, pid:0

Child checkpoint7b
page tables of child process:118, pid:0

Child checkpoint7c
page tables of child process:179, pid:0

Child checkpoint8
```

```
system calls made by child process:4

Child checkpoint 9
Number of processes: 4

Parent checkpoint 7
page tables: 632
Parent checkpoint8
system calls made by parent process:4
```

Observations

- The initial number of processes is 3. This includes the *init* and *sh* system programs along with the current program.
- The number of processes increase as we call $fork()$
- The number of pagetables per process increase as we call $printf()$ each time, this is due to the fact that $printf()$ actually outputs to a file held by the process instead of directly to the console (which can be done using $cprintf()$). More print statements increases the size of the file therefore the number of pagetables held by the process.
- The number of system calls actually does not increase as we make system calls in the form of *settickets*() or $fork()$. I have been unable to debug what is leading to this behaviour since I cannot find any obvious theoretical flaw in my program.

Task 2

The lottery scheduling is implemented as below-

Lottery Scheduling

```
uint turn = rand(getRunnableTickets());
uint sum = 0;

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;
  if(turn > sum + p->tickets){
    sum += p->tickets;
    continue;
  }
  // Switch to chosen process.  It is the process's job
  // to release ptable.lock and then reacquire it
  // before jumping back to us.
  //noproc = 0;
  c->proc = p;
  p->ticks++;
  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;
  break;
}
```

I generated a random number between 0 and the number of total tickets held by runnable processes (disregard processes that are not runnable since those cannot be scheduled anyway) using the *rand*(*intmax*) and *getRunnableTickets*() function as shown in Appendix A. Then we loop through the process list to find which process holds the winning lottery. Upon finding the process, we schedule it and break the loop.

To test the scheduler I implemented another system call called getticks() which prints the number of ticks each RUNNING or RUNNABLE process has ran for (doing so ignores the *init* and *sh* process). The system call is shown here-

```
//get number of ticks the process has ran for
int
getticks()
{
  struct proc *p;
  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != UNUSED && p->state != SLEEPING){
      cprintf("%s %d, ", p->name, p->ticks);
    }
  }
  cprintf("\n");
  release(&ptable.lock);
  return 1;
}
```

I called this system call from the first test program which is named *ticktest*1.*c*. The *getticks*() function is called once every 1000 iteration of the outer loop. When we run multiple processes simultaneously it gives us the ticks each process has ran for at the time that line gets executed in *ticktest*1.*c* which means it is not have an equal time interval, but it serves our purpose well as we do not care about the time interval as long as we monitor all 3 processes at that time.

The output of the program is a bit messy, so I processed it using some search and replace and saved it in a spreadsheet. And using simple charting functions of the spreadsheet we generate Figure 1.

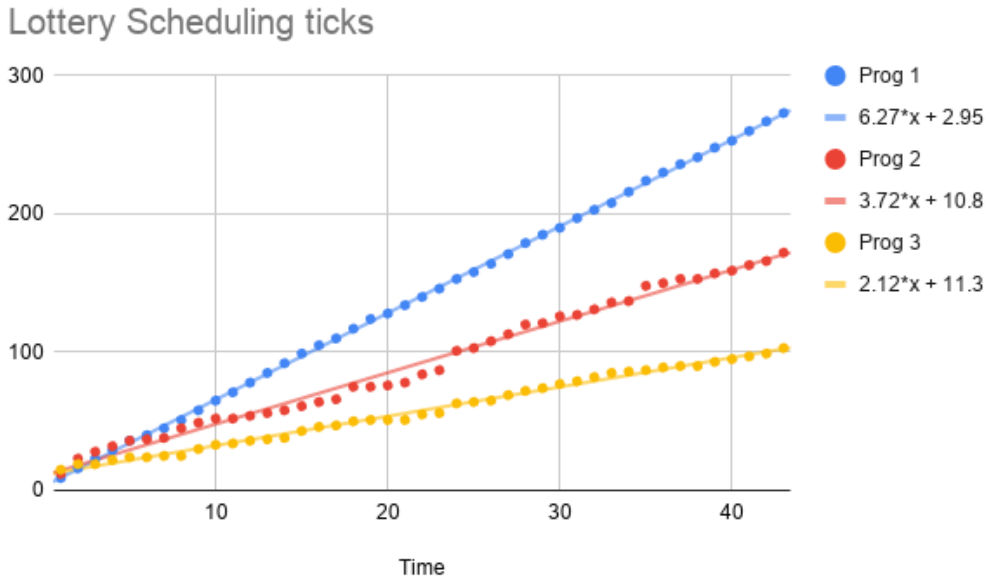

Figure 1: Ticks of 3:2:1 Ticket ratio with Lottery Scheduling

From the trendlines of the figure, which is the linear regression of the data, we can find the slope of each program. From there we can find the ratios -

$Prog1 : Prog2 = 6.27 : 3.72 = 1.685 : 1$ which is close to $3 : 2$
$Prog1 : Prog3 = 6.27 : 2.12 = 2.957 : 1$ which is close $3 : 1$

Stride Scheduling

I implemented stride scheduling as below-

```
struct * min;
double minpass;

for(p = ptable.proc, min = ptable.proc, minpass = 10000.0;
p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;
  if(p->pass <= minpass){
    minpass = p->pass;
    min = p;
  }
}
double stride = (double) (getRunnableTickets()/(min->tickets));
min->pass += stride;
```

The stride for each process is dynamically calculated from the number of tickets held by runnable process at that moment, instead of keeping it static as a variable within the process. Using the same testing method as lottery scheduling, we generate Figure 2.

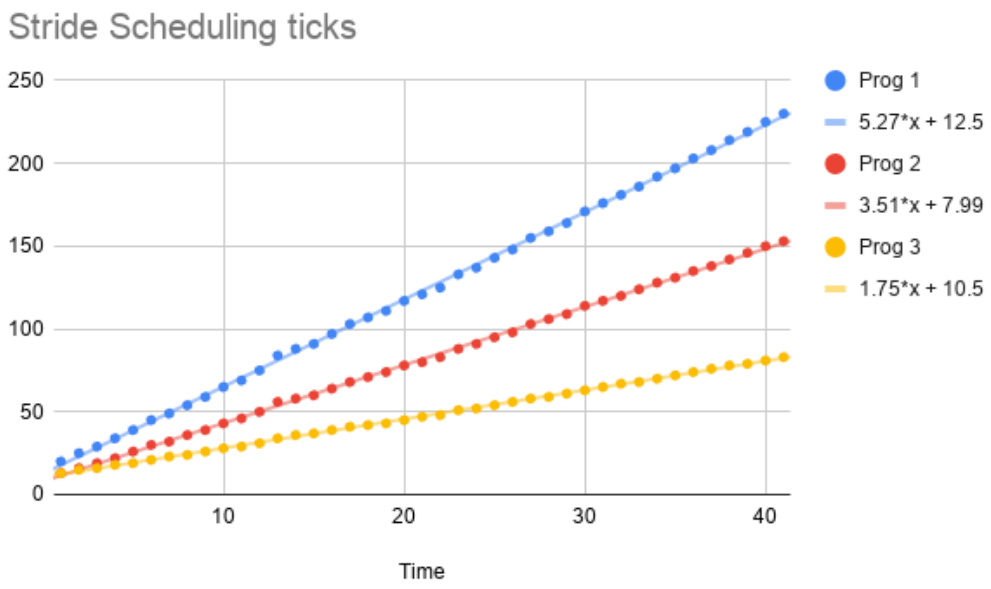

Figure 2: Ticks of 3:2:1 Ticket ratio with Stride Scheduling

As we can see from the plot, the ratio are as follows -

$prog1 : prog2 = 5.27 : 3.51 = 1.501 : 1$
$prog1 : prog3 = 5.27 : 1.75 = 3.011 : 1$

which is more accuare than what we got from lottery scheduling and as we can see from the plot, it is also more fair as it does not rely on running for longer period of time to ensure fairness.

Appendix A

demo.c

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

int main(int argc, char* argv[]){

    printf(1, "checkpoint1\n");

    printf(1, "checkpoint2\nnumber of processes:%d\n\n", info(1));

    printf(1, "checkpoint3\npage tables of current process:%d\n\n", info(2));

    printf(1, "checkpoint3a\npage tables of current process:%d\n\n", info(2));
    printf(1, "checkpoint3b\npage tables of current process:%d\n\n", info(2));
    printf(1, "checkpoint3c\npage tables of current process:%d\n\n", info(2));

    printf(1, "checkpoint4\nsystem calls made by
        current process:%d\n\n", info(3));
    settickets(30);
    printf(1, "checkpoint4a\nsystem calls made by current process:%d\n\n", info(3));
    settickets(40);
    printf(1, "checkpoint4b\nsystem calls made by current process:%d\n\n", info(3));
    settickets(60);
    printf(1, "checkpoint4c\nsystem calls made by current process:%d\n\n", info(3));

    int pid = getpid();
    printf(1, "checkpoint5\npage tables of current process:
    %d, pid:%d\n\n", info(2), pid);

    printf(1, "checkpoint6\nsystem calls made by current process:%d\n\n", info(3));
    int pid2= fork();
    if(pid2 == 0){
        printf(1, "Child checkpoint7\npage tables of child process:
            %d, pid:%d\n\n", info(2), pid2);

        printf(1, "Child checkpoint7a\npage tables of child process:
            %d, pid:%d\n\n", info(2), pid2);
        printf(1, "Child checkpoint7b\npage tables of child process:
            %d, pid:%d\n\n", info(2), pid2);
        printf(1, "Child checkpoint7c\npage tables of child process:
            %d, pid:%d\n\n", info(2), pid2);
        printf(1, "Child checkpoint8\nsystem calls made by child process:
            %d\n\n", info(3));
        printf(1, "Child checkpoint 9\nNumber of processes: %d\n\n", info(1));
        exit();
    }
    else{
        sleep(10);
        printf(1, "Parent checkpoint 7\npage tables: %d\n", info(2));

        printf(1, "Parent checkpoint8\n
            system calls made by parent process:%d\n\n", info(3));
        exit();
    }

  return 0;
}
```

Updates to struct proc

```
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)
  uint nosyscall;              // Number of system calls made by this process
  uint tickets;                // Number of tickets the process holds
  uint ticks;                  // Number of ticks the process has run for
  double pass;                 // Total pass of this process
};
```

Total tickets of runnable processes

```
//get the number of total tickets held by
//runnable processes
uint
getRunnableTickets()
{
  struct proc *p;
  uint total = 0;

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

Random Number Generator

```
//Random number generator using BlumBlumShub
uint
rand(int max){

  static uint seed = 65521;
  uint m = 2903*2879;

  seed = (seed*seed)% m;
  if(max <= 0) return 1;
  return seed%max;
}
```