Mehrnaz Ayazi
Prof. Nael Abughazaleh
CS202
5/2/2021

Lab 1 report: OS Structure and scheduler

This project consists of two parts. The first part is to add a system call named info to the XV6 operating system. The second part is adding lottery and stride scheduler to XV6 instead of its default Round Robin scheduler.

INFO SYSTEM CALL

Info system call gets an integer as an input and does one of the following:

1. If the input is equal to 1 the output would show **count of the processes in the system**
2. If the input is equal to 2 the output would show **count of the total number of system calls that the current process has made so far**
3. If the input is equal to 3 the output would be **the number of memory pages the current process is using**

   To add this system call we need to modify the following files.

   1. syscall.h
   2. syscall.c
   3. sysproc.c
   4. usys.S
   5. User.h

Syscall.h

Adding the following line to syscall.h file will define a new system call to the operating system with number 22 assigned to it.

```
#define SYS_info 22
```

Syscall.c

We need to add the following lines to this file

```
106 extern int sys_info(void);


133 [SYS_info] sys_info,
```

Sysproc.c

The function linked to this system call is written in the sysproc.c file.

But before writing the system call function we need to define a few new variables and functions in the system.

We add a new variable to proc struct in the **`proc.h`** named **`sysnum`**

```
int sysnum;                  //number of syscalls the process made
```

This variable keeps the count of system call each process has called and is incremented in the beginning of all system calls. To do so, we have to add the following line to these system call functions:

- sys_fork
- sys_exit
- Sys_wait
- sys_kill
- Sys_getpid
- sys_sbrk
- sys_sleep
- sys_uptime
- sys_info
- sys_settickets(Added Later)
- sys_setstrides(Added Later)

```
myproc()->sysnum +=1;
```

Next, we add a new function to our system that counts the processes when called. We define the process in the **defs.h** file.

```
int             getNumProc(void);
```

Now that we have defined the function we can write the function in the **proc.c** file and call in **sysproc.c** where we need it. The reason to do so and not embedding the function in the **sys_info()** function is because we cannot access **ptable** in **sysproc.c.**

```
int getNumProc(void)
{
   struct proc *p;
   int count = 0;

   acquire(&ptable.lock);

   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
   {
       if(p->state != UNUSED)
           count++;
   }

   release(&ptable.lock);

   return count;
}
```

By calling **getNumProc()** in **sys_info()** function we can get the number of processes in the system.

Following is the **sys_info()** function in the **sysproc.c** file.

The function checkers the input in **argint(0)** and in case of no error proceeds and checks the input.

If the input is equal to 1 returns the count of processes in the system by calling **getNumProc()**.

If the input is equal to 2, return **`myproc()->sysnum`** which is the count of the system calls current process made so far.

If the input is equal to 3, returns the number of page tables the current process is using by returning the value of **`myproc()->sz/PGSIZE.`**

And finally, if the input is not equal to 1,2 or 3 the function returns -1.

```
int
sys_info(void)
{
   myproc()->sysnum +=1;
   int func;
   if(argint(0, &func) < 0)
       return -1;
   if(func == 1){
       return getNumProc();
   }
   else if (func == 2){
       return myproc()->sysnum;
   }
   else if(func == 3){
       return myproc()->sz/PGSIZE;
   } else{
       return -1;
   }

}
```

To connect this system call to user applications we need to add it to the **`Usys.S`** file,

```
SYSCALL(setstrides)
```

And **user.h** file.

```
int info(int func);
```

Now by calling **`info(int func)`** in a user program **`sys_info`** is called.

To test this system call a new user program was added to the files.

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

int
main(void)
{
   printf(1, "num of processes %d\n", info(1));
   printf(1, "count of syscalls %d\n", info(2));
   printf(1, "page table  %d\n", info(3));
   exit();
}
```

SCHEDULER

Lottery Scheduler

To add the lottery scheduler to our system we first need to define a few variables in the **`proc.h`** file for process struct.

```c
int tickets;
```

Tickets shows the number of tickets each process has. To set the tickets we should first define a default ticket value for all processes and also add a new system call to change the number of tickets of a process.

To add a default ticket value when a process is created, we add the following line to **`allocproc()`** function in **`proc.c`**

```c
92 p->tickets = 3;
```

Then we add a new system call that allows processes to set a new ticket value.

```
int sys_settickets(void)
{
   myproc()->sysnum +=1;
   int n;
   if(argint(0, &n) < 0)
       return -1;
   myproc()->tickets = n;
   return n;
}
```

Before we can add the scheduler function we need to define it in **def.h** file.

```
void        lottery_scheduler(void) __attribute__((noreturn));
```

We can now add a new function to **proc.c** that can be used as lottery scheduler of the system.

```
void
lottery_scheduler(void)
{

   struct proc *p;
   int foundproc = 1;
   struct cpu *c = mycpu();
   c->proc = 0;
   int chosenTicket;

  cprintf("Calling proc::lottery_scheduler\n");

   int total_tickets = 0;
   int counter = 0;

   for(;;) {
       if (!foundproc) hlt();
       foundproc = 0;
       // Enables interrupts on this processor
       sti();

       // Get the lock
       acquire(&ptable.lock);

       // Find the total number of tickets in the system
       total_tickets = 0;
       for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
```

```
        if (p->state != RUNNABLE) {
            continue;
        }
        total_tickets += p->tickets;
    }
    if(total_tickets <=1){

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

            // Switch to chosen process.  It is the process's job
            // to release ptable.lock and then reacquire it
            // 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;
            break;
        }
        release(&ptable.lock);
        continue;

    }

    // Grab a random ticket from the ticket list
    chosenTicket = rand() % total_tickets;
    cprintf("chosen ticket = %d\n",chosenTicket);

    counter = 0;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->state != RUNNABLE)
            continue;

        //find the process which holds the lottery winning ticket
        if ((counter + p->tickets) < chosenTicket){
            counter += p->tickets;
            continue;
        }

        // Schedule this process
```

```
            foundproc = 1;
            c->proc = p;
            switchuvm(p);
            p->state = RUNNING;
            swtch(&(c->lottery_scheduler),p->context);

            switchkvm();
            c->proc = 0;
            break;
        }
        release(&ptable.lock);
    }
}
```

After initialization and acquiring the lock we find the total number of tickets using this for loop.

```
total_tickets = 0;
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->state != RUNNABLE) {
        continue;
    }
    total_tickets += p->tickets;
}
```

Next to make the scheduler robust to unpredicted situations and to make sure that the system always runs a process we embed the Round Robin scheduler in the lottery scheduler in case the total number of tickets is less than 2.

```
if(total_tickets <=1){

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

        // Switch to chosen process.  It is the process's job
        // to release ptable.lock and then reacquire it
        // 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;
        break;
    }
    release(&ptable.lock);
    continue;

}
```

If the total number of tickets is greater than 2 then we proceed with the lottery scheduler and search the processes for the winner.

```
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->state != RUNNABLE)
        continue;

    //find the process which holds the lottery winning ticket
    if ((counter + p->tickets) < chosenTicket){
        counter += p->tickets;
        continue;
    }
...
```

When the process in possession of the ticket is found the cpu switches to the winner process and the state of the process is changed to RUNNING.

```
foundproc = 1;
c->proc = p;
switchuvm(p);
p->state = RUNNING;
swtch(&(c->lottery_scheduler),p->context);
```

Changing the Scheduler

To change the scheduler the following changes need to take place.

First, the following line needs to be added to cpu struct in proc.h

```
struct context *lottery_scheduler;
```

Second, the following line in **`sched()`** in **`proc.c`** needs to be replaced:

```
swtch(&p->context, mycpu()->scheduler);
```

With this line

```
swtch(&p->context, mycpu()->lottery_scheduler);
```

Finally we change **`main.c`** file so that the system use **`lottery_scheduler()`** as the default scheduler.

We replace the following line in **`mpmain()`** function,

```
scheduler();
```

With this line

```
lottery_scheduler();
```

Testing the Scheduler

To test the scheduler three almost identical programs with three different allocated tickets were written.

```
#include "types.h"
#include "stat.h"
#include "user.h"
int main(int argc, char *argv[]) {
   settickets(30);
   int i,k;
   const int loop=43000;
   for(i=0;i<loop;i++) {
       printf(1,"prog1 %d\n", i);
       asm("nop"); //in order to prevent the compiler from optimizing the for loop
       for(k=0;k<loop;k++) {
           asm("nop");
       }
   }
   exit();
}
```

Printing **`i`** gives an estimation of the allocated CPU to that particular program.

This is the output of running three programs simultaneously(the program was run for 10 seconds).

1. Prog1 : 30 tickets
2. Prog2: 20 tickets
3. Prog3 : 5 tickets

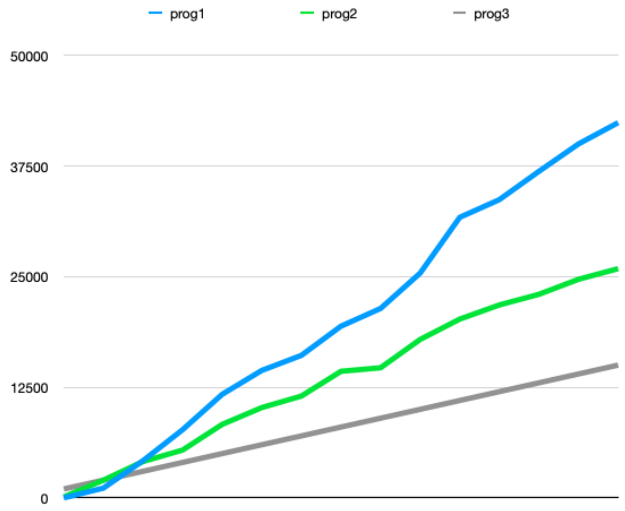
```
prog1 8276
prog2 5036
prog3 1558
```

The ratios to the actual ratios are as follows:

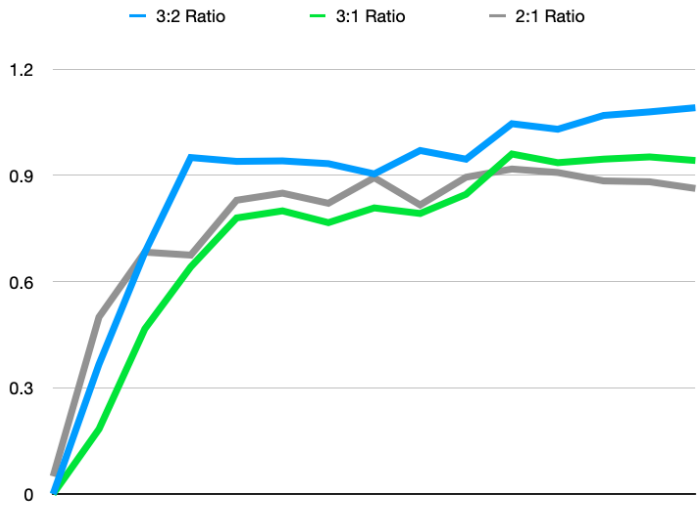
1. prog1/prog2 = 1.09
2. prog1/prog3 = 0.88
3. prog2/prog3 = 0.80

lottery scheduler CPU allocation ratios

	prog1	prog2	prog3
	0	100	1000
	1100	2000	2000
	4200	4100	3000
	7700	5400	4000
	11700	8300	5000
	14400	10200	6000
	16100	11500	7000
	19400	14300	8000
	21400	14700	9000
	25400	17900	10000
	31700	20200	11000
	33700	21800	12000
	36900	23000	13000
	40000	24700	14000
	42400	25900	15000



In the next figure errors can be seen for lottery scheduler for three ratio of 3:1, 3:2 and 2:1.



Stride Scheduling

Just like the lottery scheduler we need a few new variables for stride scheduling.

The following variables are added to **`proc.h`** process struct.

```
int default_stride;
int stride;
```

`Default_stride` is the value that has the initial value of the strides assigned to the process which is added to the process strides each time the process gets the CPU.

`Stride` is the value of current strides the process has.

Next, a system call is added so the processes can change their **`default_stride`** value.

```
int sys_setstrides(void)
{
   myproc()->sysnum +=1;
   int n;
   if(argint(0, &n) < 0)
       return -1;
   myproc()->default_stride = n;
   myproc()->stride = n;
   return n;
}
```

This system call gets an integer as input and sets the **`default_value()`** to input.

Next we need to assign a default stride value to each process when it is created.

To do this, we add the following lines to **`allocproc()`** function in **`proc.c`**.

```
 p->default_stride = 100/(p->tickets);
```

```
p->stride = p->default_stride;
```

Before we can write the stride scheduler we need to add the following variable to cpu struct in **`proc.h`**

```
struct context *stride_scheduler;
```

We also need to define the scheduler function in **`defs.h`**

```
void        stride_scheduler(void) __attribute__((noreturn));
```

Now that we defined everything we need we can write the stride scheduler

```
void
stride_scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;
    float min_stride;
    struct proc *chosen_proc = myproc();

    cprintf("Calling proc::stride_scheduler\n");

    for (;;) {

        // Enables interrupts on this processor
        sti();

        acquire(&ptable.lock);

        min_stride = INT_MAX;
```

```
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if (p->state != RUNNABLE) continue;

            if (p->stride < min_stride) {
                min_stride = p->stride;
                chosen_proc = p;
            }
        }
        chosen_proc->stride += chosen_proc->default_stride;
        c->proc = chosen_proc;
        switchuvm(chosen_proc);
        chosen_proc->state = RUNNING;

        swtch(&(c->stride_scheduler), chosen_proc->context);
        switchkvm();

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

After initialization and acquiring the lock the scheduler looks for the process with minimum strides.

```
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
   if (p->state != RUNNABLE) continue;

   if (p->stride < min_stride) {
       min_stride = p->stride;
       chosen_proc = p;
   }
}
```

When the minimum stride process is found, the CPU switches to that process and increments the stride by default_stride value.

```
chosen_proc->stride += chosen_proc->default_stride;
```

```
c->proc = chosen_proc;
switchuvm(chosen_proc);
chosen_proc->state = RUNNING;
```

To switch to stride scheduler as the default scheduler, we need to change the following line in **`sched()`** in **`proc.c`**

```
swtch(&p->context, mycpu()->scheduler);
```

To

```
swtch(&p->context, mycpu()->stride_scheduler);
```

And change the following line in **`mpmain()`** in **`main.c`**

```
scheduler();
```

To

```
stride_scheduler();
```

Testing the scheduler

To test the stride scheduler the same program for testing lottery scheduler was used. The strides were set to 10,20 and 30 for prog1, prog2 and prog3 respectively.

The output is as follows:

```
prog1 6094
prog2 2709
prog3 1823
```

The ratio of CPU ratios to expected ratios are as follows:

1. prog1/prog2 = 1.12
2. prog1/prog3 = 1.11
3. prog2/prog = 0.99

The programs were run for 10 seconds.

Stride Scheduler time slots

prog1	prog2	prog3
3200	1700	1000
5800	2900	2000
9000	4500	3000
11900	6100	4000
14800	7500	5000
17600	8900	6000
20800	10600	7000
23800	12000	8000
26700	13500	9000
29500	14800	10000
32600	16400	11000
35500	17900	12000
38600	19600	13000
41700	21100	14000