Mark Gameng

CS 450 – Duan Yue

Lab 3 – Process Management

Change Priority

Proc.h

```
int prio;                       // add priority value to a proc, [0, 31], lab 3
int runs;                       // lab 3, aging? how man runs it has done. so when prio has ties, it goes by runs
int timestart;                 // lab 3, keep track of start time
int burst;                    // lab 3 keep track of burst, time its runnning
```

Proc.c

```
void setprio(int priority){
  struct proc *p = myproc();
  // prio has range of 0 and 31
  if(priority >= 0 && priority <= 31){
    acquire(&ptable.lock);
    p -> prio = priority;
    p -> state = RUNNABLE;
    cprintf("PID: %d, Prio: %d\n", p -> pid, priority);
    sched();
    release(&ptable.lock);
  }
}
```

Since were updating the priority of a process, we gotta lock. Also make sure that once it changes the priority value, it then transfers the control to the scheduler, sched(),. Immediately because the priority list has been updated. I also added a print statement, so its easier to see the process id and the priority value when I do the testing afterwards.

With this added system call, have to update other files

User.h

```
void setprio(int);
```

Defs.h

```
void            setprio(int);
```

Usys.s

```
SYSCALL(setprio)
```

Syscall.c

```
extern int sys_setprio(void);
```

```
[SYS_setprio]     sys_setprio,
```

Syscall.h

```
#define SYS_setprio 25
```

Sysproc.c

```
int sys_setprio(void){
  int priority;
  if(argint(0, &priority) < 0){
    return -1;
  }

  setprio(priority);

  return 0;
}
```

Initialization of priority and runs (aging) value:

Proc.c in allocproc()

```
p -> prio = 0; // lab 3
p -> runs = 0; // lab 3
p -> timestart = ticks; // lab 3
p -> burst = 0; // lab 3
```

In fork()

```
np -> prio = curproc -> prio; // lab 3
np -> runs = 0; // lab 3
np -> burst = 0; // lab 3
np -> timestart = ticks; // lab 3
```

These extra variables, runs, burst, timestart allow me to get the stats for each process so I can compare the priority and what these stats are at the end. Tracks the scheduling performance of each process as well as if my priority scheduling works or not.

Update scheduler()

```
struct proc *ptorun = 0; // to store the process to run // highest prio and lowest run time to avoid starvation
int highestprio = 32; // start off as 32 since highest is 31
int lowestruns = 100000; // part of aging

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

  if(p -> prio < highestprio && p -> runs < lowestruns){
    highestprio = p -> prio; // keep track of highest prio
    lowestruns = p -> runs; // keep track of lowest runs
    ptorun = p; // store process to run later
  }
}
// now got the highest prio and lowest run, so know which process to run first
// increase prio of other proc, and add a run to the proc that will run
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p -> state != RUNNABLE)
    continue;

  if(p != ptorun){ // other processes not the proccess that we will run
    if(p -> prio > 0){
      p -> prio --; // increase prio, but not go to negative
    }
  }
}
```

```
  if(ptorun){
    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    p = ptorun; // run the proccess that we found to have highest prio and lowest run.
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    p -> runs++; // add 1 to run lab 3
    if(p -> prio < 31){
      p -> prio++; // decrease the prio by adding 1, making sure not over 31
    }
    uint tmpstart = ticks; // time start

    swtch(&(c->scheduler), p->context);
    switchkvm();

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    uint tmpend = ticks; // time end
    p -> burst = p -> burst + (tmpend - tmpstart); // calc burst time and increment each time
    c->proc = 0;
  }
  release(&ptable.lock);
```

Basically, the scheduler is, it goes through the list of processes and finds and stores the one with the highest priority and least amount of runs. The least amount of runs is for when there are two processes with the same priority. It will then take the one that has ran less. Once it finds the process to run, it then decreases its own priority and increments the runs var while increasing the priority of the other processes by 1. This is the aging of priority. I also calculated the total burst time, by having the ticks before starting and ticks afterwards. This is for calculating the waiting time.

In exit()

```
int timeend = ticks;
int turnaroundtime = timeend - curproc -> timestart;
int waitingtime = turnaroundtime - curproc -> burst;
cprintf("\nPID: %d, Turnaround time: %d | Waiting time: %d\n", curproc -> pid, turnaroundtime, waitingtime);
```

In exit, I then calculate the turnaround times and waiting time as well as printing those with the PID. The PID is so I can cross reference which process had the original priority values and see if my priority scheduling works.

Testing

For testing, I just used the code the TA showed in one of his slides.

```
int main(int argc, char *argv[]){
    if(argc >= 1){
        int prio = atoi(argv[1]);
        setprio(prio);
        int limit = 14300;
        int i, j;
        for(i = 0; i < limit; i++){
            asm("nop");
            for(j = 0; j < limit; j++){
                asm("nop");
            }
        }
    }
    exit();
}
```

Though, I added a way for me to change the priority via the parameters when running in cmd. Also increased the limit to have a longer run time.

These were the results.

```
$ test 1 &;test 10 &;test 20

PID: 17, Turnaround time: 1 | Waiting time: 0

PID: 19, Turnaround time: 0 | Waiting time: 0
PID: 18, Prio: 1
PID: 20, Prio: 10
PID: 16, Prio: 20

PID: 18, Turnaround time: 108 | Waiting time: 71
zombie!

PID: 20, Turnaround time: 109 | Waiting time: 73
zombie!

PID: 16, Turnaround time: 115 | Waiting time: 76
```

```
$ test 10 &;test 20 &;test 1

PID: 40, Turnaround time: 1 | Waiting time: 1
PID: 41, Prio: 10

PID: 42, Turnaround time: 1 | Waiting time: 0
PID: 39, Prio: 1
PID: 43, Prio: 20

PID: 39, Turnaround time: 94 | Waiting time: 50

PID: 41, Turnaround time: 92 | Waiting time: 51
zombie!
$
PID: 43, Turnaround time: 125 | Waiting time: 86
zombie!
```

You can basically ignore the first two prints of stats, for example, PIDs 17, 19, 40, and 42. The pids with the priority is the one that matters. In both images, those with higher priority finished earlier, which is to be expected because all these processes have the same code/ same time to finish. For example, PID 39, with original priority of 1, had a turnaround time of 94 with waiting time 50, while PID 41 with originally priority of 10, had a turnaround time of 92 with a waiting time of 51. Even though, PID 39 started later than PID 41, since its priority is higher, then it ran first and ended first. The turnaround and waiting times tend to increase the lower priority it is, which is to be expected. Thus, my priority scheduling works.