Operating Systems and Networks

Assignment 5
Enhancing xv6 OS

Name: Samruddhi Shastri
Roll no: 2019111039

Run the operating system

```
$ make clean
$ make qemu SCHEDULER=<scheduler>
```

OR

```
$ make clean && make qemu SCHEDULER=<scheduler>
```

<scheduler> = DEFAULT (For Round Robin scheduling)
<scheduler> = FCFS (For First-come first-serve scheduling)
<scheduler> = PBS (For Priority Based Scheduling)
<scheduler> = MLFQ (For Multi-level Feedback Queue Scheduling)

Files Modified / Added

i) benchmark.c - Create process to test scheduling
ii) defs.h - add syscalls
iii) Makefile - Macro, compilation of newly added files
iv) proc.c - modified scheduling algorithms and added syscalls
v) ps.c - user program for printing information about processes
vi) setPriority.c - user program for changing priority of the process
vii) Syscall.c - add syscalls
viii) syscall.h - add syscalls
ix) sysproc.c - add syscalls
x) time.c - user function to test waitx syscall
xi) trap.c - increment rtime, modification related to preemption
xii) user.h - add syscalls
xiii) usys.S - add syscalls

Explanation of code

waitx (system call)

i) Add ctime, rtime, etime to the proc struct (File: proc.h)

```
 int ctime;                  // changed
 int etime;                  // changed
 int rtime;                  // changed
```

ii) Initialize ctime, rtime, etime in allocproc (File: proc.c)

```
 p->ctime = ticks; // changed
 p->rtime = 0;     // changed
 p->etime = 0;     // changed
```

iii) Increment the runtime after each clock cycle if the state is RUNNING (File: trap.c) (File: trap.c)

```
if(myproc() != 0 && myproc()->state == RUNNING){ // changed
  myproc()->rtime = myproc()->rtime + 1; // changed
}
```

iv) Set value of the etime when the process exits (File: proc.c)

```
curproc->etime = ticks; // changed
```

v) Iterate through all the processes in the ptable, if any child of the current process is found, whose state is ZOMBIE, reset all the values and compute wtime and rtime for the process.

`wtime = etime - (rtime + ctime)`
`rtime = rtime`
(File: proc.c)

```
int
waitx(int* wtime, int* rtime){
 struct proc *p;
 int havekids, pid;
 struct proc *curproc = myproc();
 acquire(&ptable.lock);
 for(;;){
   havekids = 0;
   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
     if(p->parent != curproc)
```

```
      continue;
    havekids = 1;
    if(p->state == ZOMBIE){
      pid = p->pid;
      kfree(p->kstack);
      p->kstack = 0;
      freevm(p->pgdir);
      #ifdef MLFQ
      for(int j=0; j<num_proc_in_q[p->queue]; j++){
        if(p == queue[p->queue][j]){
          for(int k=j; k<num_proc_in_q[p->queue]-1; k++){
            queue[p->queue][k] = queue[p->queue][k+1];
          }
          queue[p->queue][num_proc_in_q[p->queue]-1]->pid = 0;
          num_proc_in_q[p->queue]--;
          break;
        }
      }
      #endif
      p->pid = 0;
      p->parent = 0;
      p->name[0] = 0;
      p->killed = 0;
      p->state = UNUSED;
      *wtime = p->etime - (p->ctime + p->rtime);
      *rtime = p->rtime;
      release(&ptable.lock);
      return pid;
    }
  }
  if(!havekids || curproc->killed){
    release(&ptable.lock);
    return -1;
  }
  sleep(curproc, &ptable.lock);  //DOC: wait-sleep
 }
}
```

time (system call)

```
time <command>
```

It creates a child process and executes the command entered with the ‘time’. The parent process waits until the child process exits. It calls waitx to return the rtime and wtime. (File: time.c)

```
int
main(int argc, char *argv[])
{
 int a, b;
 if(argc <= 1){
   printf(1,"time: invalid arguments\n");
   exit();
 }
  int pid = fork();
 if(pid == 0){
   exec(argv[1], argv + 1);
   exit();
 }
 waitx(&a, &b);

 printf(1, "\nwtime:       %d\n", a);
 printf(1, "rtime:       %d\n", b);

 exit();
}
```

Scheduling Algorithms

i) Round Robin (default)

Iterate through all the processes and find the process that is RUNNABLE. If found, allocate CPU to it. (File: proc.c)

```
#ifdef DEFAULT
     acquire(&ptable.lock);
     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
       if(p->state != RUNNABLE)
         continue;
       p->n_run++;
       c->proc = p;
       switchuvm(p);
       p->state = RUNNING;
       swtch(&(c->scheduler), p->context);
       switchkvm();
```

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

ii) ***First-Come First-Serve scheduling***

Iterate through all the processes to find the one with minimum ctime. Then, allocate the CPU to it. (File: proc.c)

```
#ifdef FCFS
    acquire(&ptable.lock);
    struct proc *pmin_ctime = 0;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if (p->state != RUNNABLE)
        continue;
      if(p->pid != 0){
        if (pmin_ctime==0){
          pmin_ctime = p;
        }
        else{
          if (p->ctime < pmin_ctime->ctime)
            pmin_ctime = p;
        }
      }
    }
    p = pmin_ctime;
    if (p != 0){
      p->n_run++;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(c->scheduler), p->context);
      switchkvm();
      c->proc = 0;
    }
    release(&ptable.lock);
  #endif
```

iii) ***Priority Based scheduling***

a) Declare priority for each process in proc structure. (File: proc.h)

```
int priority;                // changed
```

b) Initialise the priority to 60 in allocproc (File: proc.c)

```
p->priority = 60; // changed
```

c) Iterate through all the processes to check for the one with least priority (value). If found, allocate CPU to it. (File: proc.c)

```
#ifdef PBS
  acquire(&ptable.lock);
  struct proc *pmax_priority = 0;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->state != RUNNABLE)
      continue;
    if(p->pid != 0){
      if (pmax_priority==0){
        pmax_priority = p;
      }
      else{
        if (p->priority < pmax_priority->priority)
          pmax_priority = p;
      }
    }
  }
  p = pmax_priority;
  if (p != 0){
    p->n_run++;
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    switchkvm();
    c->proc = 0;
  }
  release(&ptable.lock);
#endif
```

d) setPriority

Take the inputs from the command line and call set_priority system call with the arguments. (File: setPriority.c)

```
int
main(int argc, char *argv[])
{
 int priority, pid;
```

```
 if(argc <= 2){
   printf(1,"setPriority: invalid arguments\n");
   exit();
 }
 priority = atoi(argv[1]), pid = atoi(argv[2]);
 if(priority < 0 || priority > 100){
   printf(1, "setPriority: Invalid priority\n");
   exit();
 }
 int ret = set_priority(priority, pid);
   if(ret >= 0 && ret <=100 )
   printf(1,"Successfully set the priority of process with pid %d from
%d to %d\n", pid, ret, priority);
 exit();
}
```

e) set_priority

Iterate through the ptable to check for the process with the given pid. If found, update the priority of the process. If the new priority of the process is less than the old one, yield the process running in the CPU and reschedule. (File: proc.c)

```
int
set_priority(int new_priority, int pid){
 struct proc *p;
 acquire(&ptable.lock);
 int a = -1;
 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
   if(p->pid == pid){
     a = p->priority;
     p->priority = new_priority;
     break;
   }
 }
 release(&ptable.lock);
 if(a > new_priority){
   yield();
 }
 return a;
}
```

iv) *Multi-level Feedback queue scheduling*

i) Add ticks[5], queue, cpu_ticks, q_push_time to the proc struct (File: proc.h)

- ticks[i] represent the number of ticks the process ran in queue i.
- queue is the current queue in which the process is present
- cpu_ticks is the number of ticks spent by the process in a CPU before it relinquishes.
- q_push_time is the time (in ticks) when the process enters the queue in which it is currently present.

```
int ticks[5];
int queue;
int cpu_ticks;
int q_push_time;
```

ii) Create queues. (File: proc.c)

- q_num_proc stores the number of processes in the queue.
- max_ticks stores the maximum number of ticks that are allowed for the queue.

```
struct proc *queue[5][NPROC];
int q_num_proc[5] = {0}, max_ticks[5] = {1, 2, 4, 8, 16};
```

iii) Initialise the arrays and variables. (File: proc.c)

```
#ifdef MLFQ
   p->cpu_ticks = 0;
   p->queue = 0;
   p->q_push_time = 0;
   for(int i=0; i<5; i++)
     p->ticks[i] = 0;
 #endif
```

iv) Add the process in the queue 0 when they are created. (File: proc.c)

Function: userinit

```
#ifdef MLFQ
   p->q_push_time = ticks;
   p -> queue = 0;
   queue[0][q_num_proc[0]] = p;
   q_num_proc[0]++;
 #endif
```

Function: fork

```
#ifdef MLFQ
   np->q_push_time = ticks;
   np->queue = 0;
```

```
  queue[0][q_num_proc[0]] = np;
  q_num_proc[0]++;
#endif
```

v) Schedule the processes (File: proc.c)

- Check for aging. If a process has not been allocated CPU for 100 ticks since it entered the queue, promote it to the upper queue (incase the current queue is non zero)
- Next, iterate over all the queues. If the length of queue is greater than zero, schedule the process which is at the head of queue.
- After the process has exited or finished its time slice, add it to the queue.
- The change of queue takes place in the trap function.

```
  acquire(&ptable.lock);

  for(int i=1; i <= 4; i++){
    for(int j=0; j < q_num_proc[i]; j++){
      struct proc *p = queue[i][j];
      if(ticks - p->q_push_time > 100){
        for(int k = j; k < q_num_proc[i]-1; k++)
          queue[i][k] = queue[i][k+1];
        q_num_proc[i]--;
        //}
        p->q_push_time = ticks;
        p->queue = i-1;
        queue[i-1][q_num_proc[i-1]] = p;
        q_num_proc[i-1]++;
      }
    }
  }

  struct proc *p =0;
  for(int i=0; i <= 4; i++){
    if(q_num_proc[i] > 0 && queue[i][0]!=0){
      p = queue[i][0];
      for(int j = 0; j < q_num_proc[i]-1; j++)
        queue[i][j] = queue[i][j+1];
      q_num_proc[i]--;
      //}
      break;
    }
  }
```

```
  if(p!=0 && p->state==RUNNABLE){
    p->n_run++;
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    swtch(&c->scheduler, p->context);
    switchkvm();
    c->proc = 0;
    if(p->state == RUNNABLE){
      p->q_push_time = ticks;
      queue[p->queue][q_num_proc[p->queue]] = p;
      q_num_proc[p->queue]++;
    }
  }
  release(&ptable.lock);
  #endif
```

vi) Once the process is scheduled, check if it has exhausted its time slice. If yes, demote its queue and yield the process. Else, increment the cpu_ticks and continue its operation. (File: trap.c)

```
if (myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0 +
IRQ_TIMER)
 {
   #ifdef MLFQ
     if(myproc()->cpu_ticks >= max_ticks[myproc()->queue]){
       myproc()->cpu_ticks = 0;
       if(myproc()->queue != 4)
         myproc()->queue++;
       yield();
     }
     else{
       myproc()->cpu_ticks++;
       myproc()->ticks[myproc()->queue]++;
     }
   #else
     yield();
   #endif
 }
```

iv) If the process has slept during its execution and the time slice has not been expired, reschedule the process to the same queue once its state changes to RUNNABLE.

```
static void
wakeup1(void *chan){
 struct proc *p;
 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
   if(p->state == SLEEPING && p->chan == chan){
     p->state = RUNNABLE;
     #ifdef MLFQ
       p->cpu_ticks = 0;
       p->q_push_time = ticks;
       queue[p->queue][q_num_proc[p->queue]] = p;
       q_num_proc[p->queue]++;
     #endif
   }
 }
}
```

```
int
kill(int pid)
{
 struct proc *p;
 acquire(&ptable.lock);
 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
   if(p->pid == pid){
     p->killed = 1;
     // Wake process from sleep if necessary.
     if(p->state == SLEEPING){
       p->state = RUNNABLE;
       #ifdef MLFQ
         p->q_push_time = ticks;
         queue[p->queue][q_num_proc[p->queue]] = p;
         q_num_proc[p->queue]++;
       #endif
     }
     release(&ptable.lock);
     return 0;
   }
 }
 release(&ptable.lock);
 return -1;
}
```

benchmark (user program)

```
$ benchmark
```

It creates programs to check the scheduling of programs.

```
int number_of_processes = 10;

int main(int argc, char *argv[]){
 int j;
 for (j = 0; j < number_of_processes; j++){
   int pid = fork();
   if (pid < 0){
     printf(1, "Fork failed\n");
     continue;
   }
   if (pid == 0){
     volatile int i;
     for (volatile int k = 0; k < number_of_processes; k++){
       if (k <= j){
         sleep(2000); //io time
       }
       else{
         for (i = 0; i < 100000000; i++){
           ; //cpu time
         }
       }
     }
     printf(1, "\nProcess: %d Finished\n", getpid());
     exit();
   }
   else{
     set_priority(100-(20+j),pid);
   }
 }
 for (j = 0; j < number_of_processes+5; j++){
   wait();
 }
 exit();
}
```

ps (syscall)

Iterate through all the processes and print the details related to it. (File: proc.c)

```
void
ps(void)
{
 #ifdef PBS
   static char *states[] = {
     [UNUSED]    "UNUSED",
     [EMBRYO]    "EMBRYO",
     [SLEEPING]  "SLEEPING",
     [RUNNABLE]  "RUNNABLE",
     [RUNNING]   "RUNNING",
     [ZOMBIE]    "ZOMBIE"
   };
   struct proc *p;
   cprintf("\n pid \t\t priority \t state \t\t\t rtime \t\t
n_run\n\n");
   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
     if(p->pid!=0){
       cprintf(" %d \t\t %d \t\t %s \t\t %d \t\t %d\n", p->pid,
p->priority, states[p->state], p->rtime, p->n_run);
     }
   }
 #endif
 #ifdef FCFS
   static char *states[] = {
     [UNUSED]    "UNUSED",
     [EMBRYO]    "EMBRYO",
     [SLEEPING]  "SLEEPING",
     [RUNNABLE]  "RUNNABLE",
     [RUNNING]   "RUNNING",
     [ZOMBIE]    "ZOMBIE"
   };
   struct proc *p;
   cprintf("\n pid \t\t state \t\t\t rtime \t\t n_run\n\n");
   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
     if(p->pid!=0){
       cprintf(" %d \t\t %s \t\t %d \t\t %d\n", p->pid,
states[p->state], p->rtime, p->n_run);
     }
   }
 #endif
 #ifdef DEFAULT
   static char *states[] = {
     [UNUSED]    "UNUSED",
```

```
    [EMBRYO]    "EMBRYO",
    [SLEEPING]  "SLEEPING",
    [RUNNABLE]  "RUNNABLE",
    [RUNNING]   "RUNNING",
    [ZOMBIE]    "ZOMBIE"
  };
  struct proc *p;
  cprintf("\n pid \t\t state \t\t\t rtime \t\t n_run\n\n");
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid!=0){
      cprintf(" %d \t\t %s \t\t %d \t\t %d\n", p->pid,
states[p->state], p->rtime, p->n_run);
    }
  }
 #endif
 #ifdef MLFQ
  static char *states[] = {
    [UNUSED]    "UNUSED",
    [EMBRYO]    "EMBRYO",
    [SLEEPING]  "SLEEPING",
    [RUNNABLE]  "RUNNABLE",
    [RUNNING]   "RUNNING",
    [ZOMBIE]    "ZOMBIE"
  };
  struct proc *p;
  cprintf("\n pid \t\t state \t\t\t rtime \t\t wtime \t\t n_run \t\t
cur_q \t\t q0 \t\t q1 \t\t q2 \t\t q3 \t\t q4\n\n");
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid!=0){
      cprintf(" %d \t\t %s \t\t %d \t\t %d \t\t %d \t\t %d \t\t %d
\t\t %d \t\t %d \t\t %d \t\t %d\n", p->pid, states[p->state], p->rtime,
ticks - p->enter, p->n_run, p->queue, p->ticks[0], p->ticks[1],
p->ticks[2], p->ticks[3], p->ticks[4]);
    }
  }
 #endif
}
```

ps (user program)

Call the ps system call. (File: ps.c)

```
$ps
```

```
int
```

```
main(int argc, char *argv[]){
 if(argc != 1){
   printf(1, "ps: invalid arguments\n");
   exit();
 }
 ps();
 exit();
}
```