CS1217 - Spring 2023 - Lab 2

Bhumika Mittal, Saptarishi Dhanuka

Contributions of individual team members:
Bhumika: Question 1, MLFQ, Test cases
Saptarishi: Lottery, Test cases (+helped in debugging MLFQ)

Question 1

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
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)
};
```

We can see from proc.h, an xv6 process can be in the following **6 states**: UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE

- ***Zombie*** is a state that is used when a process is done running but is still in the process table and is waiting to be cleaned up by its parent. An exited process remains in the zombie state until its parent calls wait() to find out it exited.
- ***Embryo*** is a state that is used when a process is being created and is not yet runnable.
- ***Unused*** is a state that is used when a process is not being used.
- ***Sleeping*** is a state that is used when a process is sleeping and is not running. It is waiting for some other function to complete such as disk read and is equivalent to the blocked state discussed in class.
- ***Runnable*** is a state that is used when a process is ready to run. It is equivalent to READY state as discussed in the class.
- ***Running*** is a state that is used when a process is running.

These many states are needed because this helps the CPU to identify the processes current state and help in making efficient decisions about the CPU scheduling and allocation time to each

process. For example - scheduling the sleeping process on the CPU while it is still waiting for some is a waste of CPU time.

```
enum procstate state
```

state is the variable which keeps track of the process state. It takes any one of the enum procstate value – UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE, to keep the track of the current state of the process

Question 2

The scheduler() is responsible for the context switching between the processes and selecting the next process that needs to run.
It loops, doing the following:

- choose a process to run
- swtch to start running that process
- eventually that process transfers control via swtch back to the scheduler

The for(; ;) loop is an infinite loop and is required for running the scheduling code continuously, to look for the next process to run. The scheduler selects the process to run (it could be the same process as well), switches the context if needed, and executes it. Once the process is done executing, it changes its state. The infinite loop ensures that the scheduler is always running and can always select the next process to execute.

The default scheduling algorithm that is being implemented in xv6 is the **Round Robin** scheduling algorithm. The loop iterates over the process table, looks for the next process to run that is in the RUNNABLE state. It selects the first process that is RUNNABLE, and switches to its context to execute it.
When the timer interrupt fires, the process is preempted and is sent to the end of the process table, and the next RUNNABLE process is scheduled on the CPU.

Line 329 has an infinite loop. Line 335-337 contains a for loop which looks for the next RUNNABLE process from the process table. Line 342-344 switches the context to the first available runnable process. Line 351 (c->proc = 0;) sets the process to 0.
Line 351 (c->proc = 0;) resets mycpu()'s to be null so that when the loop runs again there is no current process selected and the algorithm can select the next runnable process to run from the table. When the Line 335 for loop finishes, the Line 329 infinite loop just runs it again so it goes through the process table again with the same method.

Question 3

The state of the newly created child process (as a result of the fork() system call) set to **RUNNABLE.**

The initial state for all newly created processes in xv6 is UNUSED. Line 196 sets the state of the newly created process to UNUSED.

The fork creates the new process and copies over the address space, kernel stack, etc, as well as the state of the parent process to the child process. But the child process might not necessarily be in the same state as the parent process (which is usually RUNNING because when the child is forked the parent is running). Therefore, all the child processes are initialized to UNUSED until the child process is fully ready so that they don't run prematurely. Once the process is ready, its state is changed to RUNNABLE (line 217).
It is set to RUNNABLE since the new child process is a completely new process and can be scheduled to run on the CPU. If it was not set to RUNNABLE then it would never be able to be chosen by the scheduling algorithm to actually execute its instructions.

Question 4

The process to which the abandoned child processes (i.e., the processes whose parents have exited) are assigned is "init" as per the following screenshot.

```
255  // Pass abandoned children to init.
256  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
257    if(p->parent == curproc){
258      p->parent = initproc;
259      if(p->state == ZOMBIE)
260        wakeup1(initproc);
261    }
262  }
```

PART 1: MLFQ Scheduler for xv6

To run MLFQ scheduler, go to main.c, line 15:

```
enum which_sched curr_sched = MLFQ;
```

and set curr_sched = MLFQ, as shown above.

Then, just run *make qemu-nox,* the scheduling algorithm is set to MLFQ and it will be executed. This could be tested by writing testpr in the CLI of xv6.

PART 2: LOTTERY Scheduler for xv6

To run MLFQ scheduler, go to main.c, line 15:

```
enum which_sched curr_sched = LOTTERY;
```

and set curr_sched = LOTTERY, as shown above.

Then, just run *make qemu-nox,* the scheduling algorithm is set to MLFQ and it will be executed. This could be tested by writing tickettest in the CLI of xv6.

APPENDIX:

Step by step documentation for MLFQ:

1. Main.c: In mpmain function:

```
// Common CPU setup code.
static void
mpmain(void)
{
 cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
 idtinit();       // load idt register
 xchg(&(mycpu()->started), 1); // tell startothers() we're up

 switch(curr_sched)
 {
   case DEFAULT:
     scheduler();     // start running processes (Default scheduling round robin)
     break;
   case MLFQ:
     mlfqsched();     // MLFQ scheduler
     break;
   case LOTTERY:
     lotterysched();  // Lottery scheduler
     break;
   default:
     scheduler();
 }

}
```

2. Main.c: On top

```
enum which_sched{DEFAULT, MLFQ, LOTTERY};

enum which_sched curr_sched = DEFAULT;
```

3. In defs.h: Below line 115 which is for default scheduler

```
void            mlfqsched(void) __attribute__((noreturn));
void            lotterysched(void) __attribute__((noreturn));
```

==========

4. In syscall.c

```
extern int sys_setpriority(void);
extern int sys_getpriority(void);
```

```
[SYS_setpriority] sys_setpriority,
[SYS_getpriority] sys_getpriority,
```

5. In syscall.h

```
#define SYS_setpriority 22
#define SYS_getpriority 23
```

6. In defs.h

```
int setpriority (int pid , int priority);
int getpriority (int pid);
```

7. In proc.c

```
// set priority of a process
int setpriority(int pid, int priority)
{
 struct proc *p;
 // bound check
 acquire(&ptable.lock);
 if (priority < 0 || priority > MAXPRIORITY)
 {
   return -1; // return error
 }
 // lock the table - no change can be done to the table
 // iterate over the table and find the process with the given pid
 for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
 {
   // if found, set the priority and return
   if (p->pid == pid)
   {
     p->priority = priority;
     p->budget = DEFAULT_BUDGET;
     release(&ptable.lock);
     return 0;
   }
 }
```

```
 release(&ptable.lock); // unlock
 return -1;
}

int getpriority(int pid)
{
 struct proc *p;

 acquire(&ptable.lock); // lock the table
 // iterate over the table and find the process with the given pid
 for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
 {
   // if found, return the priority
   if (p->pid == pid && p->state != UNUSED)
   {
     release(&ptable.lock);
     return p->priority;
   }
 }
 release(&ptable.lock); // unlock
 return -1;
}
```

8. In sysproc.c

```
int sys_getpriority(void)
{
 int pid;
 if(argint(0, &pid) < 0)
   return -1;
 return getpriority(pid);
}

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

```
 return setpriority(pid, priority);
}
```

9. In proc.h: struct proc

```
#define DEFAULT_BUDGET 5
#define MAXPRIORITY 2
```

```
 int priority;                //Process priority
 int budget;                  //budget for each process
```

10. In defs.h

```
#define TICKS_TO_PROMOTE 50
```

11. In proc.c

```
struct
{
 struct spinlock lock;
 struct proc proc[NPROC];
 unsigned int PromoteAtTime;
} ptable;
```

- In allocproc()

```
 p->budget = DEFAULT_BUDGET;
 p->priority = MAXPRIORITY;
```

- In userinit()

```
 ptable.PromoteAtTime = ticks + TICKS_TO_PROMOTE;
```

```
//adjust priority function
void adjustPriority(struct proc *p)
{
 for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
 {
   if (p->state != ZOMBIE)
   {
     if (p->priority < MAXPRIORITY)
     {
       p->priority = p->priority + 1;
       p->budget = DEFAULT_BUDGET;
     }
```

```
    }
  }
}
```

```
// MLFQ scheduler
void mlfqsched(void)
{
 struct proc *p;          // pointer to the process
 struct cpu *c = mycpu(); // pointer to the cpu - mycpu() returns the
address of the current cpu
 c->proc = 0;             // set the current process to 0

 for (;;)
 {
   // Enable interrupts on this processor.
   sti();
   acquire(&ptable.lock);

   int current_priority; // set the current priority to the highest
priority
   for(current_priority = MAXPRIORITY; current_priority >= 0; ){
     int count = 0; // count the number of processes in the priority queue
which are runnable

     for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)     // loop
through the process table
     {
       if (p->state != RUNNABLE)
       {
         continue;
       }
       if (p->priority == current_priority)              // if the process
is runnable and has the current priority (this divides the processes into
different priority queues)
       {
         count++;            // increment the count
         c->proc = p;        // set the current process to p
         switchuvm(p);       // switch to the process's address space
         p->state = RUNNING; // set the state of the process to running
```

```
        int time_in = ticks;                // get the time when the
process starts running
        swtch(&(c->scheduler), p->context); // runs the process p
        int time_out = ticks;               // get the time when the
process stops running
        switchkvm();                        // switch to the kernel's
address space

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0; // set the current process to 0

        // update the budget of the process
        p->budget = p->budget - (time_out - time_in);
        if (p->budget <= 0)
        {
          if (p->priority > 0)
          {
            // if the budget is less than or equal to 0, set the priority
of the process to the next priority and set the budget to the default
budget
            p->priority = p->priority - 1;
            p->budget = DEFAULT_BUDGET;
          }
          else
          {
            // if the budget is less than or equal to 0 and the priority
is 0, set the budget to DEFAULT_BUDGET [policy decision because this isn't
mentioned in the assignment]
            p->budget = DEFAULT_BUDGET;
          }
        }

        //promotion of the process if it yeilds the cpu or if it finishes
its execution
        if (ptable.PromoteAtTime <= ticks)
        {
          adjustPriority(p);
          ptable.PromoteAtTime = ticks + TICKS_TO_PROMOTE;
```

```
          }
        }
      }
      if(count == 0)
      {
        current_priority--; // if there are no processes in the priority
queue, decrement the priority
      }
    }
    release(&ptable.lock);
  }
}
```

LOTTERY SCHEDULER

getpinfo can be used by typing ps in the xv6 shell which shows information about the current processes whose at least one of the pstat fields is nonzero.