Project 1C
COM S 352
Spring 2024

(Due: Monday April 1, 2024)

1 Introduction

For this project iteration, you will implement an MLFQ scheduler, which is a scheduler based on multi-level feedback queues. The tasks are summarized in the following table. **You can work invididually or in a pair (i.e., a group of two students).**

Task		**Points (Individual Effort)**	**Points (Working in Pair)**
MLFQ scheduler	**Basic requirements (Implement Rules 1-4 in Section 3)**	**25**	**20**
	Advanced requirement (Implement Rule 5 in Section 3)	**8 (bonus)**	**5**
Integrate MLFQ scheduler with RR scheduler		**3**	**3**
System calls	**startMLFQ**	**4**	**4**
	stopMLFQ	**4**	**4**
	getMLFQInfo	**6**	**6**
Set up system program testsyscall.c for testing		**2**	**2**
Documentation		**6**	**6**

2 Background

2.1 Review of scheduler()

Xv6 implements a round-robin (RR) scheduler. Starting from `main()`, here is how it works. First, `main()` in program main.c performs initialization, which includes creating the first user process, `user/init.c`, to act as the console (shell). As shown below, the last thing `main()` does is call `scheduler()`, a function that never returns.

```
void
main()
{
  // ...
  userinit();      // first user process, runs init.c
  // ...
  scheduler();
}
```

As shown below, the function `scheduler()` in `kernel/proc.c` contains an infinite for-loop `for(;;)`. Another loop inside the infinite loop iterates through the `proc[]` array looking for processes that are in the `RUNNABLE` state. When a `RUNNABLE` process is found, `swtch()` is called to perform a context switch from the scheduler to the user process. The function `swtch()` returns only after context is switched back to the scheduler. This may happen for a couple of reason: the user process blocks for I/O or a timer interrupt forces the user process to yield.

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run.
//  - swtch to start running that process.
//  - eventually that process transfers control
//    via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);

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

2.2 Ticks

Xv6 measures time in ticks, which is a common approach for Operating Systems. A hardware timer is configured to trigger an interrupt every 100 ms, which represents 1 tick of the OS. Every tick, context is switched to the scheduler, meaning the scheduler must decide on each tick: continue running the current user process or switch to a different one.

To follow the full line of calls from the timer interrupt, assuming the CPU is in user mode, the interrupt vector points to assembly code in `kernel/trampoline.S` which calls `usertrap()` which calls `yield()` which calls `sched()` which calls `swtch()`. It is the call to `swtch()` that switches context back to the scheduler. That is when the call to `swtch()` that `scheduler()` made previously returns and the scheduler must make a decision about the next process to run.

```
//
// handle an interrupt, exception, or system call from user space.
// called from trampoline.S
//
void
usertrap(void)
{
  // ...
  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();
// ...
```

The **only reason** yield is called is when there is a timer interrupt; its purpose is to cause a preemption of the current user process. It preempts the current process (kicks it off the CPU) by changing the state from `RUNNING` to `RUNNABLE` (also known as the Ready state). Below is the code for yield from `kernel/proc.c`.

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
  sched();
  release(&p->lock);
}
```

3 Project Requirements

Note: The default `Makefile` runs qemu emulating 3 CPU cores. Concurrency introduces additional concerns that we will **not** deal with in this project. Search for where `CPUS` is set to the default of 3 in `Makefile` and change it to **1**.

You are required to implement an MLFQ scheduler that runs as an alternative to the xv6's default Round-Robin scheduler. Specifically, the MLFQ scheduler should run according to the following rules (**adapted** from the textbook):

- **Rule 1:** Totally **m** levels of priority, denoted as 0, 1, ..., and m-1, are defined in the system. Each process has a priority. The lower is the priority number, the higher is the priority. Accordingly, **m** Ready Queues, denoted as Queue 0, Queue 1, ..., and Queue m-1, are maintained, where Queue x contains the processes whose priority is x.
- **Rule 2:** When a process enters the system, its starting priority is 0.
- **Rule 3:** When the scheduler needs to pick a process to run, it picks the process of higer priority than others. If there are multiple processes of the same highest priority, they should be scheduled

using the Round-Robin policy. Once a process with priority x is scheduled to run, it is assigned to run for a time quantum of 2(x+1) ticks. That is, it can run for the quantum unless it is blocked.

- **Rule 4:** Once a process uses up its time quantum at a its current priority level (regardless of how many times it has given up the CPU), its priority is degraded (i.e., its priority number is incremented by one and it moves down one queue) if its priority number is not m-1 yet.
- **Rule 5**: After a process has been of priority number m-1 (i.e., has stayed at Queue m-1) for **n** ticks, its priority number is boosted to 0.

For individual efforts, implementing rules 1-4 is the basic requirement; but implementing Rule 5 will earn you a bonus. Implementing all the rules is required for pairs.

In addition, you should implement the following system calls to allow a user program to start/stop the MLFQ scheduler and to retrieve the information of the callinger process' running history with the scheduler.

- System call startMLFQ:

  int startMLFQ(int **m**, int **n**)

  This system call is made to start the MLFQ scheduler with parameters m and n, where m is the number of priority levels and n is the maximum ticks that a process can stay at priority number m-1 before it is boosted to 0.

- System call stopMLFQ:

  int stopMLFQ()

  The system call is made to stop MLFQ scheduler and continue to run the default RR scheduler.

- System call getMFLQInfo:

  int getMLFQInfo(struct MLFQInfoReport *report)

  The system call is made by a process to retrieve the process' running history with the MLFQ scheduler. Specifically, struct MLFQInfoReport is defined as

```
#define MLFQ_MAX_LEVEL 10
struct MLFQInfoReport{
        int tickCounts[MLFQ_MAX_LEVEL];
};
```

  Here, for i=0, ..., m-1, tickCounts[i] is the count of ticks that the calling process has run on CPU when it was of priority i.

4 Design and Implementation Guides

You are free to design and implement your system, as long as the above requirements are met. But you can also follow the guides below.

4.1 Data structures in the kernel

Some extra data structures should be introduced to the kernel space. They may include the following.

4.1.1 The global data structure

The MLFQ schedule may maintain the following global data structures for itself.

(1) A **flag** to indicate if the MLFQ scheduler should be running: This flag indicates if MLFQ scheduler has been truned on to run. If it is turned on, the MLFQ scheduler is run; otherwise, the default RR scheduler is run instead. Note that, system call startMLFQ can trun on this flag and system call stopMLFQ can turn off the flag.

(2) The MLFQ **scheduler's parameters**: Two major parameters are **m** – the number of priority levels, and **n** – the max number of ticks that a process can stay at priority level m-1 before it is boosted to 0. These parameters should be set up by system call startMLFQ and then maintained in the kernel space.

(3) The MLFQ **queues**: For the MLFQ scheduler to run, m Ready Queues should be maintained in the kernel space. Each of the queues can be implemented as a (doubly) linked list, where each element contains:
   a. information of a process on the queue, i.e., a pointer to a struct proc or the index of the array proc[NPROC];
   b. pointer to the next element on the queue;
   c. a pointer to the previous element on the queue if the queue is implemented as a doublely-linked list.

   Also, the head elements of the queues should be maintained for the queues.

(4) The **struct MLFQInfoReport**: The structure is defined for system call getMLFQInfo to return the calling process' running history.

You may define the above data structure at file proc.h and/or proc.c.

4.1.2 The additional data structure for each process

You may add the following fields to the PCB of each process (i.e., struct proc) to keep track of the process' priority and running history:

- A flag indicating if the process is already added to a queue of the MLFQ scheduler: During the course of running the MLFQ scheduler, new processes may be created and should be added to a queue of the MLFQ scheduler. This integer can be used to identify such processes.
- The current priority of the process
- The count of ticks that the process has run on the current queue if it is running
- The counts of ticks that the process has run when it is with each of the priority levels: These counts are used to keep track of the process' running history.
- The count of ticks that the process has stayed at priority m-1: This information will be used to determine when the process should be boosted to priority 0.

Note that, you need to initialize these additional fields in function freeproc() in file proc.c, as we should have already learned from Project 1.B.

4.2 Helper functions for managing the MLFQ queues

The basic operations for the MLFQ queues, which may be (doublely) linked lists, should be implemented. The operations may be implemented by the following functions:

- mlfq_enque(int **level**, struct proc ***proc**) or mlfq_enque(int **level**, int **procIndex**): insert a process pointed to by pointer **proc** or with index **procIndex** in array proc[NPROC] to the queue with priority level **level**.
- mlfq_deque(int **level**): delete the head process from the queue with priority level **level**.

Optionally, you may implement the following more general operation:

- mlfq_delete(int **level**, struct proc ***proc**) or mlfq_delete(int **level**, int **procIndex**): delete process pointed to by pointer **proc** or with index **procIndex** in array proc[NPROC] from the queue with priority level **level**.

Note that, implementing a queue data structure and associated operations is a common gramming practice. You can refer to or reuse existing implementations as long as you provide proper acknowledgement in your comments.

4.3 Restructure the kernel code for scheduler

The existing xv6 kernel only implements the RR scheduler in function scheduler(void) in proc.c. In order to support both the new MLFQ scheduler and the default RR scheduler, you may restructure the scheduler code as follows.

(1) Construct the default RR_scheduler(struct cpu *c) from the existing function scheduler(void) function:

```
void
RR_scheduler(struct cpu *c)
{
  struct proc *p;
  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == RUNNABLE) {
      // Switch to chosen process.  It is the process's job
      // to release its lock and then reacquire it
      // before jumping back to us.
      p->state = RUNNING;
      c->proc = p;
      swtch(&c->context, &p->context);
      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
      release(&p->lock);
  }
}
```

(2) Define the new scheduler as MLFQ_scheduler:

Similar to RR_scheduler, the new scheduler can have the following signature:

MLFQ_scheduler(struct cpu *c)

The implementation of this function is to be discussed later.

(3) Restructure the exsting function scheduler(void):

```
void
scheduler(void)
{
        struct cpu *c = mycpu();
        c->proc=0;

        for(;;){
                // Avoid deadlock by ensuring that devices can interrupt.
                intr_on();
                if(mlfqFlag){//the flag indicates if MLFQ scheduler is running
                MLFQ_scheduler(c);
                }else{
                RR_scheduler(c);
                }
        }
}
```

4.4 Implement the MLFQ scheduler: function MLFQ_scheduler(struct cpu *c)

The function may be structured as follows.

```
MLFQ_scheduler(struct cpu *c)
{
  struct proc *p=0; //p is the process scheduled to run; initially it is none

    while(mlfqFlag){//each iteration is run everytime when the scheduler gains the control

       if(p>0 && (p->state==RUNNABLE || p->state==RUNNING)){
         //if the current process is still runnable

         //to add: increment the tick count of p on the current queue

         //to add: 1. check if p's time quantum for the current queue expires or not.
         //2. If expires: (1) updating p's running history accordingly,
         //(2) move p to the queue below it (if p is not at the bottom queue yet),
         //(3) p is set to 0 (to indicate another process should be find to run next).
       }

       //to add for implementing Rule 5:
       //increment the tick counts for the processes at the bottom queue and
       //move each process who has stay there for n ticks to the top queue
```

```
      //to add:
      //add new processes (which haven't been added to any queue yet) to queue 0

      if(p=0){//need to find another process to run

        //to add:
        //find the RUNNABLE process that has the highest priority to run
        //and let p point to this process

      }

      //run the selected process
      if(p>0){
        acquire(&p->lock);
        p->state=RUNNING;
        c->proc=p;
        swtch(&c->context,&p->context);
        c->proc=0;
         release(&p->lock);
      }

    }
}
```

4.5 Implement the system calls startMLFQ, stopMLFQ and getMLFQInfo

Recall that the system calls are implemented in the following steps:

- Declare the unique system call numbers for SYS_startMLFQ, SYS_stopMLFQ and SYS_getMLFQInfo
- Declare the prototypes for functions sys_startMLFQ, sys_stopMLFQ and sys_getMLFQInfo
- Add the implementation functions sys_startMLFQ, sys_stopMLFQ and sys_getMLFQInfo for the new system calls to array syscalls
- In proc.c or sysproc.c, implement functions sys_startMLFQ, sys_stopMLFQ and sys_getMLFQInfo
- In usys.pl in the user directory, add the entries for startMLFQ, stopMLFQ and getMLFQInfo
- In user.h, declare the signatures for startMLFQ, stopMLFQ and getMLFQInfo as well as struct MLFQInfoReport

5 Testing

A sample test program, name testsyscall.c, is attached. You are required to integrate this file into your xv6 package. The test program testsyscall works as follows:

testsyscall <number-of-child-processes> <workload> <max-ticks-at-bottom>

For example:

```
testsyscall 10 10 100
```

Here, the workload is a positive integer; the larger is it, the higher is the workload of each process.

6 Documentation

Documentation is required for the project. Every location that you add/modify code in the kernel must have a comment explaining the purpose of the change. The user programs you write must also have brief comments on their purpose and usage.

Include a `README` file with your names, a brief description, and a list of all files added to the project. If you work in pair, list your group members in this file.

7 Submission

Submit a zip file of the xv6-riscv directory. On pyrite the zip file can be created using:

```
$zip -r project-1c-xv6-riscv.zip xv6-riscv
```

Submit `project-1c-xv6-riscv.zip`. If you work in pair, each pair only submit one copy.