# Project 1C
# COM S 352
# Fall 2022

**CONTENTS**



## 1. Introduction

For the final Project 1 iteration, you will implement a Multi-Level Feedback Queue (MLFQ) scheduler. The project may be completed in pairs (2 maximum). There are no extra tasks for working in pairs this time.

The scheduler must follow the rules listed below, which are adapted from the book.

For processes A and B:
**Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
**Rule 2:** If Priority(A) = Priority(B), A & B run in round-robin fashion using the quantum length of the given queue.
**Rule 3:** When a process enters the system, its starting queue is determined by its nice value (see starting queue nice range in the table below).
**Rule 4:** Once a process uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
**Rule 5:** After every 60 ticks, all processes in the system are moved back to their starting queues (aka, priority boost).
**Rule 6:** Whenever the nice value of a process is updated (e.g., `nice()` system call), its queue level is also updated according to the table below. Note that this may result in the process being

moved out of one queue and into another. In that case it should be enqueued onto the back of the new queue.

**Queue Levels**

| | starting queue nice range | quanta |
|---|---|---|
| Queue 2 | nice ≤ -10 | 1 |
| Queue 1 | -10 < nice ≤ 10 | 10 |
| Queue 0 | nice > 10 | 15 |

# 2. Background

## 2.1 The PCB (Process Control Block) and Process Table

The `struct proc` defined in `proc.h` is xv6's implementation of a PCB (Process Control Block). It holds all information for an individual process. In `proc.c` an array of these structures is declared called `proc` to serve as the process table. The process table holds information about all processes and its size is statically set at `NPROC` (default 64). In other words, xv6 can have a maximum of `NPROC` processes. Initially, the state of all elements in the process table array are set to `UNUSED`. This simply means that a slot in the array is empty (it has no process). When a new process is created, the first empty slot in the `proc` array is found and used for the process.

How is it useful for the project? Whenever you need to store information for each process, for example, the process runtime, it would be a good idea to add this information to `struct proc`.

## 2.2 Getting to Know scheduler( )

Xv6 currently implements a round-robin (RR) scheduler. Starting from `main()`, here is how it works. First, `main()` performs initialization, which includes creating the first user process, `user/init.c`, to act as the console. 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 of 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);
    }
  }
}
```

## 2.3. What are Ticks?

Xv6 measures time in ticks, which a common approach for Operating Systems. A hardware timer is configured to trigger an interrupt every 100ms, 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, which means changing the process 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

## 3.1 Accounting for Process Runtime in Queue (5 points)

A process runtime is the time (number of ticks) spent running on the CPU while at a specific queue level. It must be tracked for each process. As with all per-process information, it would be a good idea to add this information to the PCB (see Section 2.1).

In xv6 we can only account for time at the granularity of ticks (100ms by default). So, we will adopt the following rule. On every timer-based interrupt (see Section 2.3 for how you would know timer-based interrupt has occurred) whatever process was running on the CPU gets its runtime incremented by 1 tick. Note that there may have been no process running at the time of the interrupt; this can happen on a system with no CPU-bound (only I/O-bound) processes.

Whenever a process changes queue level (either up or down) for any reason, its runtime should be set back to 0.

## 3.2 A Data Structure for the Multi-level Feedback Queue (10 points)

You will need some data structure to implement multi-level feedback queues. There are multiple ways to implement queues in C. You are free to experiment with your own approach, however, the following is highly recommended. A word of caution, the xv6 kernel provide not easy

dynamic memory management in kernel code – there is no malloc() system call available in the kernel. Most examples of queues in C you will find on the Internet assume dynamic memory and will not work.

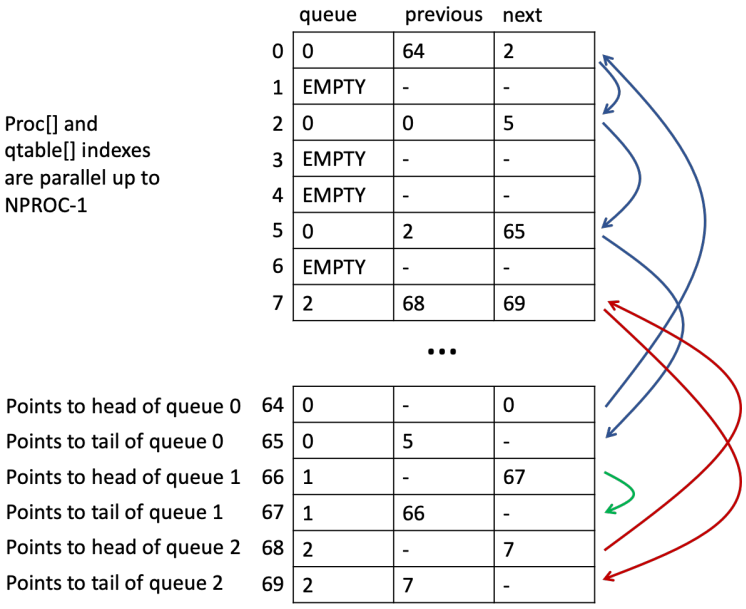
In xv6, the maximum number of processes is fixed at `NPROC` (default 64) as defined in `kernel/param.h`. Because the maximum number of processes is small and fixed, it suggests using static memory for the data structures in this project. It is a common theme in system programming to prefer static memory and in-place algorithms over dynamic memory when appropriate. Static memory can be far more efficient and less error prone than dynamic memory and its performance is more predictable, particularly when compared to the use of automatic garbage collection found in some languages.

The following approach for building multiple queues for a fixed number of processes is adapted from Chapter 4 of *Comer, Douglas. Operating System Design: The Xinu Approach, Linksys Version. 2011.* It is available digitally from the library: https://quicksearch.lib.iastate.edu/permalink/01IASU_INST/174tg9m/alma990018965010102756 Xinu is an embedded operating system designed for teaching at Purdue University. You may use any code from this book in your project with proper citation. However, it is difficult to extract and adapt just the parts of the code you need, so you may be better off learning the general approach described in the book (and below) and then implementing your own version. The following are some definitions you may use to get started; they can be placed near the top of `kernel/proc.c`.

```
#define MAX_UINT64 (-1)
#define EMPTY MAX_UINT64
#define NUM_QUEUES 3

// a node of the linked list
struct qentry {
    uint64 queue; // used to store the queue level
    uint64 prev; // index of previous qentry in list
    uint64 next; // index of next qentry in list
};

// a fixed size table where the index of a process in proc[] is the same in qtable[]
struct qentry qtable[NPROC + 2*NUM_QUEUES];
```




The first `NPROC` elements of `qtable[]` correspond directly with the elements in `proc[]` (i.e., they are parallel arrays). The values in `prev` and `next` represent indexes of `qtable[]` that point to previous and next elements in a doubly linked list. The end of `qtable[]` is set aside to store the indexes that point to the head and tail of the queue.

A good way to get started is to define some functions to manage the queue. For example, implement `enqueue` and `dequeue` functions.

### 3.3 Using a Queue in a Round-Robin Scheduler (10 points)

Before jumping into the full multi-level feedback queue implementation, it would be a good idea to modify the existing round-robin scheduler (see Section 2.2) to use just a single queue.

There are a couple of questions to address. First, when to enqueue a process? The answer is any time a process is set to `RUNNABLE` it needs to be added to the queue. To identify all locations where a process is set to `RUNNABLE`, do a search in `kernel/proc.c` for the code:
`p->state = RUNNABLE`
At all these locations enqueue the process into the correct queue (it would be a good idea to make a helper function that does this).

**A helpful hint:** the xv6 code in `kernel/proc.c` uses pointers rather than array indexes to reference the elements of `proc[]`. Because `proc[]` and `qtable[]` are being used as parallel arrays it is useful to be able to find the array index for a given pointer. It is easy to convert from a pointer to an array index with the following pointer arithmetic.

```
// assume p is a pointer to a process in proc[]
uint64 pindex = p - proc;
```

The other question that needs to be answered is how to use the queue to pick the next process to run. The existing `scheduler()` method should be modified so that rather than searching for a runnable process in the `proc[]` array it now picks the next runnable process by dequeuing it from the queue.

### 3.4 Implement All MLFQ Rules (15 points)

Now you are ready to implement the full MLFQ. Refer to the rules and queue level table given in the introduction.

**How to know what queue level to start a new process at?** See the queue level table to test which range the process nice value falls.

**How to know when one quanta of time has expired?** Sometimes context is switched back to `scheduler()` due to a voluntary action (e.g., the process is blocked or finished) and sometimes it is because the timer interrupt has preempted the process (see Section 2.3). It is up to you to devise a strategy and add code where needed such that `scheduler()` can tell the difference. The context switches due to a timer interrupt check if the current process runtime has exceeded one quanta and act according to the rules.

**How to know when to perform a priority boost?** In addition to keeping track of per-process runtimes you will also want some way to keep track of the time since the last priority boost. When the scheduler sees sufficient time has passed, it performs the priority boost before choosing the next process to run.

### 3.5 Testing (5 points)

Create a utility program in a new file `user/schedtest.c` (and add `$U/_shedtest` to `UPROGS` in `Makefile`). You may want to implement some of your own tests. At a minimum test the following two scenarios.

**Test 1:**

Use fork to create two new child processes. Both child processes then set their nice values to -19 and enter long CPU-bursts that lasts at least 30 seconds.

A simple way of simulating a process doing useful work on the CPU (i.e., a CPU burst) is a long counting loop. For example:

```
uint64 acc = 0;
for (uint64 i=0; i<count; i++) {
  acc += i;
}
```

**Test 2:**

Use fork to create two new child processes. Both child processes then set their nice values to -19. One process the enters a log CPU-burst like in the previous test while the other process simulates being I/O bound. For example:

```
for (uint64 i=0; i<count; i++) {
  sleep(1);
}
```

Collect the results from the test using the `startlog` and `getlog` systems calls implemented in Project 1B. Add a report of the test results in the README file. Are the results what you expected?

### 3.6 Documentation (5 points)

Documentation is required for the project. Every location that you add/modify code in the kernel must have a comment explain 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.

The `README` must also contain the test results from Section 3.5.

## 4. Submission

Make sure the code compiles, focus on working code over having every feature complete. We will look at the code for partial credit. Document anything that is incomplete in the `README` file.

Submit a zip file of the xv6-riscv directory. On the Linux command line, the zip file can be created using:

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

Submit `project-1c-xv6-riscv.zip`.