Project 1
COM S 362
Fall 2021

1. Introduction

For this project, you will modify the xv6 kernel to implement a Multi-Level Feedback Queue (MLFQ). The MLFQ will consist of 3 queues having time quanta of 1 tick (100ms), 2 ticks (200ms) and 4 ticks (400ms). It will also perform a priority boost every 32 ticks (3,200ms). You are also required to create a system call that provides per process scheduler statistics. The system call will be used by a utility program to collect experimental results.

2. Getting to Know scheduler( )

Xv6 implements a round robin (RR) scheduler. Starting at `main()`, here is how it works. First `main()` performs initialization, which includes creating the first user process, `init.c`, to act as the console. The last thing `main()` does is call `scheduler()`, a function that never returns.

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

The function `scheduler()` in `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);
    }
  }
}
```

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 `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();
// ...
```

Yield is only called when there is a timer interrupt, its purpose is to cause a preemption of the current user process. It gives up the CPU (preempts) for the current process by changing the state from RUNNING to RUNNABLE (also known as the Ready state).

```
// 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);
}
```

4. Detailed Requirements

Important: Your solution only needs to work on single CPU machines. If you are working from the command line, use the following command to compile and start xv6.

```
$ export CPUS=1; make qemu
```

For VS Code users, update `tasks.json` with the following:

```
"command": "export CPUS=1; make qemu-gdb",
```

4.1. Documentation (20 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 utility programs you write must also have brief comments on their purpose and usage.

A README file must be submitted, see Section 5 for the details.

4.2. Create the System Call `getpstat()` (20 points)

Add the following system call that will get statistics about processes on the system. For this step to be complete, actual values do not need to be put in the `pstat`, simply return from the system call.

```
// populates pstat with information about all processes
// returns 0 on success, -1 on error
int getpstat(struct pstat*);
```

System calls are defined in two places in the code, on the kernel side and user side. On the kernel side, system calls start with `sys_`. Do a search in all files for `sys_uptime`; use it as an example for where `sys_getpstat()` will need to be added.

On the user side, add `getpstat()` to `user.h` and `usys.pl`, again use other system calls as examples.

The `pstat` struct is defined as follows (put it in a file `kernel/pstat.h`).

```
/**
 * Adapted from https://github.com/remzi-arpacidusseau/ostep-
projects/tree/master/scheduling-xv6-lottery
 **/
#ifndef _PSTAT_H_
#define _PSTAT_H_

#include "param.h"

struct pstat {
  int inuse[NPROC]; // whether this slot of the process table is in use (1 or 0)
  int pid[NPROC];   // the PID of each process
  int ticks[NPROC]; // the number of ticks each process has accumulated
  int queue[NPROC]; // the current queue of each process or -1 if not RUNNABLE
};

#endif // _PSTAT_H_
```

Finally, test the new system call by creating a simple user program called `pstat.c`. Calling it on the command line should result in output that looks like the following, except the values may be all 0 at this point because the `pstat` struct has not been populated (also don't worry about exact formatting).

```
$ pstat
pid ticks queue
1   10    2
…
```

4.3. Return Useful Information from `getpstat()` (10 points)

Now populate `pstat` with useful information. The values of `inuse` and `pid` can be obtained directly from the process table `proc[]`. Tracking the number of process ticks is something you will need to add to `scheduler()`.

Tick accounting rule: when a process is preempted by the timer interrupt to run the scheduler, its tick count is increased by 1.

How do we know when a process is being preempted by the timer interrupt? See Section 3 which discusses how the timer interrupt causes a preemption of a running user process. Identify a good place in the chain of function calls to add a tick counter.

At the end of this step the `pstat` command should display correct values for everything except for queue. Modify `pstat` so that it only displays process information when `inuse` is 1.

4.4. Adding a First Queue (20 points)

For this step try to get a RR scheduler working with at least one queue. Recall that xv6 already uses RR scheduling, so if this step works there should be no significant change in the scheduling outcome. The only difference is that the scheduler will be using a queue instead of an array. The obvious data structure for the queue is a linked list. It enables the enqueue and deque operations required to add a process and get the next process to run. There are a few questions to be answered.

When to add a process to the queue? Any time a process is set to the RUNNABLE state it needs to be added to the queue. In `proc.c`, do a search for `p->state = RUNNABLE;`.

How to use the queue? The scheduler itself must be modified to use the queue. See Section 2 for details on how `scheduler()` works.

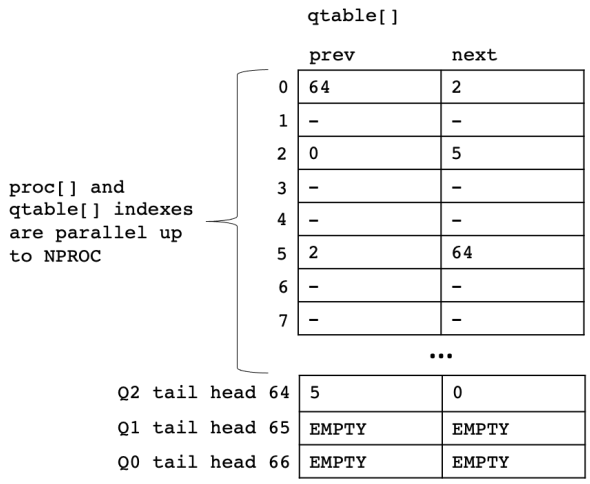
How to implement the queue data structure? It is up to you how to implement the queues. One approach would be to add them directly to the `proc[]` table. Another approach is as follows.

There are a maximum `NPROC=64` processes as defined in `param.h`. Because the maximum number of processes is small, it suggests using static memory for the data structures in this project. Using static memory is common in systems programming, particularly for smaller embedded systems. 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 process queues is adapted from Chapter 4 of Comer, Douglas. Operating System Design: The Xinu Approach, Linksys Version. 2011 (available from the library). 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. The approach shown here is simplified because we do not need to implement priority queues. It could be simplified even further, because we do not need doubly linked lists for this project. (We would if killed processes needed to be immediately removed from the queue, good for you if you can figure out why this is not required).

```
#define EMPTY (-1)
struct qentry {
    uint64 prev;
    uint64 next;
}

qentry qtable[NPROC + NQUEUES];
```



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 head and tail entries for `NQUEUES` number of queues. For example, the table in the figure above shows Q2 is a list of 3 elements, Q1 is empty and Q0 is empty.

A helpful hint, the xv6 code uses pointers rather than array indexes to reference elements of `proc[]`. It is easy to convert from a pointer to an array index with the following simple pointer arithmetic.

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

4.5. Implement MLFQ (15 points)

Implement the MLFQ with 3 three priority levels, the quanta for highest to lowest priority is 1 tick, 2 ticks and 4 ticks. There are a few things to keep in mind. When a process is moved from `RUNNABLE` to `RUNNING` it must maintain a record of what queue it was on, that way it can be placed back onto the same queue when it returns to the `RUNNABLE` state. A process drops to a lower queue only when its tick count increases by one quantum. For this step, it is not possible for a process to return to a higher priority queue.

4.6. Add Priority Boost (5 points)

Now add priority boost. Every 32 ticks, all processes return to the highest priority queue.

4.7. Test the Scheduler (10 points)

Create a utility program called `schedtest.c`. It takes two arguments: the number of ticks to sleep and a number to count.

```
$ schedtest 10 1000000000
```

The program should have loop that alternates between sleep and work. For sleep, if the passed argument is greater than zero, call the `sleep()` system call. For work, the count number indicates how much work to do, for example:

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

For the experiment, start three `schedtest` processes in the background (that is done by ending the command with the symbol `&`). The arguments for each process should be chosen such that it targets a different queue. QEMU is not a cycle accurate emulator, the speed of the host influences the simulate time, so you will need to experiment with how long different counts take to complete. For example:

```
$ schedtest 4 10000000 &
$ schedtest 2 20000000 &
$ schedtest 0 40000000 &
```

Now run `pstat` to verify that each process is placed in a different queue. The process with 0 sleep should be continusly running, except when it is preempted by the shorter runtime processes.

```
$ pstat
```

Put the results of this step in the README file.

5. Submission

Create a **README** file that states the progress on the project and the results of your experiments and/or testing. **Don't forget the documentation** described in Section 4.1.

Make sure the code compiles, focus on working code over having every feature complete.

Copy only the `kernel` and `user` folders into another folder and zip the contents for submission. This will reduce the large binary files uploaded and downloaded in the submissions.