Project 1C
COM S 352
Spring 2022

CONTENTS

1. Introduction

For this project iteration, you will implement two schedulers: a queue-based round-robin scheduler and a stride scheduler that achieves fair-share (aka proportional-share) scheduling. Both schedulers will use a queue data structure that you implement. Implementing and testing the schedulers requires understanding how preemption and time keeping works in xv6. Section 2 provides this background.

2. Background

2.1 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);
    }
  }
}
```

2.2. 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. 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.

3.1 Process runtime tracking (15 points)

For this section, the things to do are:

- Add a runtime field to `struct proc`.
- Initialize the runtime to 0 for a new process.
- Increment the runtime by 1 every time a process is interrupted by the timer.

To implement any kind of fair scheduler, there needs to be some way to keep track of how much runtime each process has had on the CPU. Because xv6 counts time in ticks (see Section 2.2) that is also how we will account for runtime. To keep it simple, use the rule that whichever process is interrupted by a timer interrupt gets one tick added to its runtime. There is not sub-tick accounting of time; runtime does not increase when a process runs for less than a single tick.

Add a field to the `struct proc` in `kernel/proc.h` to store runtime. Find the location in `kernel/proc.c` where the fields of a process are initialized and set the runtime to 0. Also add code to increment the runtime of the process that is interrupted on each tick. Where would be a good place to add this code? See Section 2.2 for a function that only gets called from the timer interrupt and knows about the process that was running at the time of the interrupt.

3.2 An improved round-robin scheduler (15 points)

For this section, the things to do are:

- Add `#define SCHEDULER` to `kernel/param.h` to select among different schedulers when testing.
- Create an alternative version of `scheduler()` in `kernal/proc.c`, name it `scheduler_rr()`.
- Use `SCHEDULER` in `main()` in `kernel/main.c` to select the scheduler to use.

Before implementing the stride scheduler, first implement a queue-based round-robin scheduler. The only difference between the new round-robin scheduler and the default scheduler in xv6 is that the new one will use a queue. The same queue data structure (as a priority queue sorted by `pass` value) will also be used to implement the stride scheduler, allowing multiple schedulers to share a single data structure. With this in mind, we will implement a simple but reusable queue as a doubly linked list with the ability to store and insert by priority (`pass` value).

We will create two new schedulers, to choose which one to use when testing add the following line to `kernel/param.h`.

```
#define SCHEDULER    2   // 1 - original, 2 - round-robin with queue, and 3 - stride
```

3.2.1 Implementing a Queue

There are multiple ways to implement a queue in C. You are free to experiment with your own approach. The following describes one simple approach.

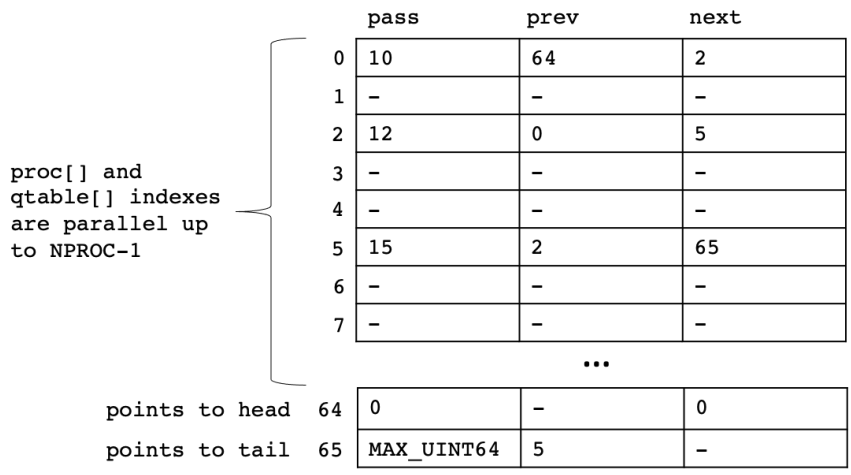
In xv6 the maximum number of processes is fixed at `NPROC` (default 64) as defined in `kernel/param.h`. Because the maximum number of processes is small, it suggests using static memory for the data structures in this project. It is a common theme in system programming to prefer static memory and in-place 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. Furthermore, xv6 does not provide easy dynamic memory management in kernel code, so if you do what to use dynamic memory in the kernel you are on your own.

The following approach for building a queue for a fixed number of processes is adapted from Chapter 4 of Comer, Douglas. Operating System Design: The Xinu Approach, Linksys Version. 2011 (available digitally 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. Although you may be better off learning the general approach and then implementing it yourself rather than trying to adapt the Xinu code into xv6. 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

// a node of the linked list
struct qentry {
    uint64 pass; // used by the stride scheduler to keep the list sorted
    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[]
qentry qtable[NPROC+2];
```



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. Maintaining the queue this way simplifies inserting a process into an already sorted list.

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

3.2.2 Using the Queue in a Round-Robin Scheduler

Now modify the existing round-robin scheduler to use the queue for storing processes in the `RUNNABLE` state (i.e., we are creating a *ready 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. In `kernel/proc.c`, do a search for the code:
`p->state = RUNNABLE`

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 `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. To make your modifications, make a copy of `scheduler()` called `scheduler_rr()`. For testing, use the constant `SCHEDULER` (previously defined in `kernel/param.h`) to call the chosen scheduler at the end of `main()` in `kernel/main.c`.

3.3 Stride scheduler (10 points)

For this section, the thigs to do are:

- Create an alternative version of `scheduler()` in `proc.c`, name it `scheduler_stride()`.
- Use `SCHEDULER` in `kernel/main.c` to select the correct scheduler to use.

Now you are ready to implement a stride scheduler. In a stride scheduler every process has a counter called pass. Every time the scheduler needs to select the next process to run it chooses the process with the lowest pass value. On each run, pass is incremented by the stride of the process. Stride is set small for high priority processes (so they will run more often) and high for low priority processes.

3.3.1 Computing stride

First, give each process a stride by adding a field to `struct proc` to store it. Update the stride whenever nice is set. The stride is calculated by as:

```
stride = 1000000 / tickets; // stride is 1 million / tickets
```

Tickets are assigned based on the nice value from the table (taken from the CFS scheduler) below. Note that nice values start at -20, so the table lookup requires first adding 20 to the nice value.

```
static const int nice_to_tickets[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

3.3.3 Using pass

Each process has a pass value that is incremented by its stride every time the process runs. Start by adding a pass field to `struct proc`. For now, initialize the pass to 0.

When creating the round-robin scheduler, we enqueued a process every time its state was set to `RUNNABLE` in `kernel/proc.c`. The locations to modify were identified by the code: `p->state = RUNNABLE`

For the stride scheduler, the queue should be kept in sorted order so that it is easy to get the process with the lowest stride from the head of the queue. To do this created an enqueue function that traverses the queue starting at the head until it finds where to put the process and then inserts it at that location.

Create a modified version of `scheduler()` called `scheduler_stride()` that dequeues the process with the smallest pass value to run next. It may be very similar to the round-robin version.

3.3.2 Initializing pass

In the previous section the pass for a new process was initialized to 0. Imaging a system where all of the processes have been running for a while and have high pass values. A new process with a pass of zero will be able to take over the CPU for a long time. Instead, it is a better strategy to initialize (e.g., in `allocproc()`) a new process' pass value to *the current lowest pass* + *the stride*.

3.4 Testing (5 points)

For this section, the things to do are:

- Update `pstat` to hold the new information added to `struct proc`.
- Run some tests and report the results in a `README` file.

From project iteration 1B, update `pstat` to reflect the new information stored in `struct proc`.

```
/**
 * 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 nice[NPROC]; // the nice value of the process

  int runtime[NPROC]; // number of ticks process has been on CPU
  int stride[NPROC]; // the stride calculated from nice
  int pass[NPROC]; // the pass value used by the stride scheduler
};

#endif // _PSTAT_H_
```

Test your solution. For example, create a user program that forks two children and sets their nice values to 14 and 19. Use the parent to check on the `pstat` after a while (`sleep()` can be used to wait an amount of time). For the experiment both children should be CPU bound, this can be done with an infinite loop, for example, `while(1)`.

Based on the `nice_to_tickets` table, we would expect the child with nice 14 to get about 3 times as much runtime.

Report your results in a `README` file.

3.5 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 3.4.

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 pyrite the zip file can be created using:

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

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