Project 1C
COM S 352
Fall 2023

CONTENTS

1. INTRODUCTION **1**

2. BACKGROUND **1**

2.1 The PCB (Process Control Block) and Process Table 1
2.2 Getting to Know scheduler( ) 2
2.3. How does preemption work? 3

3. PROJECT REQUIREMENTS **4**

3.1 Accounting for Process Runtime (15 points) 4
3.2 Add system call getruntime (15 points) 4
3.3 A queue data structure for the new schedulers (20 points) 4
3.4 Preparing for multiple schedulers (10 points) 6
3.5 A queue-based round-robin scheduler (20 points) 7
3.6 Stride scheduler (20 points) 7
3.6.1 Computing stride *7*
3.6.2 Using pass *8*
3.6.3 Initializing pass *8*
3.7 Testing (15 points) 8
3.8 Documentation (5 points) 9

4. SUBMISSION **9**

1. Introduction

For the final Project 1 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 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 per-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. How does preemption work?

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 same 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()` previously made returns and the scheduler makes a decision about the next process to run.

From `kernel/usertrap.c`:
```
//
// 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 purpose of `yield()` 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).

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 if you are not using the VirtualBox image: The default `Makefile` runs qemu emulating 3 CPU cores. Concurrency introduces additional concerns that we will not deal with in this project iteration. If you are not using the VirtualBox image, search for `CPUS` in `Makefile` and set it to 1.

3.1 Accounting for Process Runtime (15 points)

To complete this section, the things to do are:

- Add a runtime field to `struct proc`.
- Make sure runtime is initialized to 0 for a new process.
- Increment the runtime by 1 every time a process is preempted.

A process runtime is the time (number of ticks) spent running on the CPU. It must be tracked per process. As with all per-process information, it would be a good idea to add this 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 at that time gets its runtime incremented by 1 tick. Note that there may have been no processes running at the time of the interrupt; this can happen on a system with no CPU-bound (only I/O-bound) processes. This method of account for time is only an approximation, but it works well enough for most schedulers.

3.2 Add system call getruntime (15 points)

For testing purposes, add a new system call that can be used as follows.

```
int pid = 5; // the process id of interest
int runtime = getruntime(pid); // returns the runtime of the process
printf("runtime of process %s is %d\n", pid, runtime);
```

Note that this system call is almost identical to `nice()` in project 1B, review those instructions if you are not sure how to implement the system call.

3.3 A queue data structure for the new schedulers (20 points)

You will need a queue data structure to implement the two new schedulers for this project. There are multiple ways to implement queues in C. You are free to experiment with your own approach, however, the following is recommended.

A word of caution, the xv6 kernel does not provide 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, therefore, will not work.

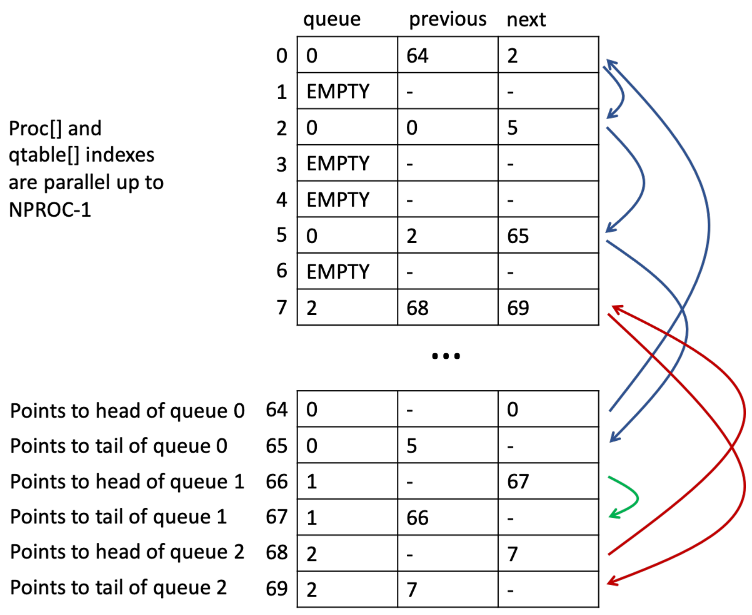
In xv6, the maximum number of processes is fixed at `NPROC` (default 64) as defined in `kernel/param.h`. Because the maximum number of processes is small and fixed, it suggests using static memory for the data structures in this project. It is a common theme in system programming to prefer static memory 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];
```



Proc[] and qtable[] indexes are parallel up to NPROC-1

		queue	previous	next
	0	0	64	2
	1	EMPTY	-	-
	2	0	0	5
	3	EMPTY	-	-
	4	EMPTY	-	-
	5	0	2	65
	6	EMPTY	-	-
	7	2	68	69
		•••		
Points to head of queue 0	64	0	-	0
Points to tail of queue 0	65	0	5	-
Points to head of queue 1	66	1	-	67
Points to tail of queue 1	67	1	66	-
Points to head of queue 2	68	2	-	7
Points to tail of queue 2	69	2	7	-

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.

3.4 Preparing for multiple schedulers (10 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 versions of `scheduler()` in `kernal/proc.c`, name them `scheduler_rr()` and `scheduler_stride()`.
- Use `SCHEDULER` in `main()` in `kernel/main.c` to select which scheduler to use.

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
```

In `kernal/proc.c`, make a two copies of `scheduler()` called `scheduler_rr()` and `scheduler_stride()`. The next sections describe how to modify these schedulers.

Section 2.2 describes how the scheduler originally gets called when the system boots. Use `SCHEDULER` to call the correct scheduler.

3.5 A queue-based round-robin scheduler (20 points)

Rules of the round-robin scheduler:

- Use FIFO ordering. The next process to run should be the process added to the queue longest ago.
- A process is allowed to run for a time quanta of 2 ticks before it is replaced by the next process. You will need to consider how you will handle the case when there is only one RUNNABLE process. Does the process get enqueued and immediately dequeued or does it simply go back to running without the queue?

Modify `scheduler_rr()` to use the queue to select from processes in the `RUNNABLE` state. There are a some 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 any statements that do something like the following:

```
p->state = RUNNABLE
```

The next question that needs to be answered is how to use the queue to pick the next process to run. The `scheduler_rr()` 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.

Finally, when should a process be preempted? Note that the timer gives an interrupt every tick. When that happens the scheduler must have some way of knowing that a timer interrupt has been called and it must then decide if the current process (if there is one) should be replaced with a new process. See how `yield()` is used in Section 2.3.

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;
```

3.6 Stride scheduler (20 points)

Now you are ready to implement a stride scheduler in `scheduler_stride()`. In a stride scheduler every process needs 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.

3.6.1 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,
};
```

3.6.2 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.

Modify `scheduler_stride()` to dequeue the process with the smallest pass value to run next. It may be very similar to the round-robin version.

3.6.3 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*.

3.7 Testing (15 points)

Create a utility program in a new file `user/schedtestc.c` (and add `$U/_shedtestc` 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 and set their nice values to 14 and 19. Both processes then enter long CPU-bursts that lasts several ticks. See the project 1B tests for how to simulate a long CPU-burst.

Based on the `nice_to_tickets` table, we would expect the child with nice 14 to get about 3 times as much runtime. Call the `getruntime()` system call for each process and print the results to verify this.

Test 2:

Use fork to create two new child processes and set their nice values to 14 and 19. One process then enters a long CPU-burst like in the previous test while the other process simulates being I/O bound. An I/O-bound process can be simulated with the following:

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

Collect the results from the test using `getruntime()`. You may also want to use `getlog()` implemented in Project 1B. Add a report of the test results in the README file. Are the results what you expected?

3.8 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 a brief description and a list of all files added to the project. The `README` must also contain the test results from Section 3.7.

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`.