Project 1B
COM S 352
Fall 2024

CONTENTS

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. How System Calls Work

An application cannot do everything itself; it needs some way to request action that only the OS is allowed to perform. A simple function call to an OS library API would be the most efficient solution, however, allowing arbitrary function calls to OS code (which must run with kernel mode level privileges) would compromise the OSes ability to provide security and reliability to

the system. Instead, system call are used, which in the end acts as a function call, but provides a secure mechanism to switch between user and kernel mode.
Here is an example from xv6, to demonstrate the point. Consider the user application `user/zombie.c`, which on line 12 calls `sleep(5)`. The end goal is to call the function `sys_sleep(void)` in `kernel/sysproc.c` with the CPU running in kernel mode. We will now explore how the system call is performed from start to end.

The implementation of `sleep(int time)` is written in assembly code in `user/usys.S`.

```
sleep:
   li a7, SYS_sleep
   ecall
   ret
```

There is a good reason for it to be written in assembly code. It must perform two important machine instructions. First, it must load a constant `SYS_sleep` (defined as 13 in `kernel/syscall.h`) into the register `a7`. We will see that the system call trap handler of the kernel specifically looks for the value saved in the processes `a7` register to know which system call the user wants to perform. Finally, the `ecall` instruction is called. This is an instruction specifically designed to cause a trap which will result in the CPU changing from user mode to kernel mode and executing the trap handler code for system calls.

The trap handler is located at `uservec` in `trampoline.S`. This code is again in assembly because we need to save the current CPU registers to the "trapframe" so that they can be restored once we return form the trap. Importantly, `uservec` ends with a call to `usertrap()`, which is implemented in `trap.c`. When the purpose of the trap is a system call, we find that `usertrap()` calls `syscall()` which is located in `syscall.c`. Here we find an interesting line of code:

```
num = p->trapframe->a7;
```

Do you recall what register `a7` of the calling process contains and that all of the registers of the calling process were saved to the trapframe? That's right, by investigating the saved `a7` register we find out what system call the user code wanted to be called. We then look into the array of functions pointers to system call functions and call the corresponding function (at index `SYS_sleep` or 13). The code for that function happens to be located in `sysproc.c` and is called `sys_sleep`.

There is still one more important detail. Because `sys_sleep()` is called by the interrupt handler (which doesn't know what arguments the user indented to call the system call with), it is called with no (or `void`) arguments). Recall that the user application called the function `sleep(5)`. What happened to that 5? When a function is called its arguments are pushed onto the call stack. But every process has its own stack, we need to investigate the user processes stack to find what argument was pushed right before the system call. A helper function is called to get the first (index 0) argument and put it into the memory location pointed to by the address of `n`:

```
argint(0, &n);
```

2.2 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`.

2.3 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);
    }
  }
}
```

2.4 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

This section is a detailed guide to implement the required changes. It is recommended to complete the steps in the order they are presented.

IMPORTANT: This project is much easier if you only run the emulator in single CPU mode. Multiple CPUs add complications (e.g., race condition bugs) you don't want to deal with.

☐ Go to `xv6-project/ Makefile` and search for `CPUS`. Change the default from 3 to 1.

```
ifndef CPUS
CPUS := 1
endif
```

3.1 Accounting for Process Runtime

☐ To complete this section, the things to do are:

- Add a field to store runtime to each processes PCB (see section 2.2)
- 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 2.2).

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 2.4 for a location in the code that gets called on every timer-based interrupt), add code at that location to increment the runtime stored in the PCB of the process that was running on the CPU 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.

The next section will create a system call `getruntime` to test what you have done.

3.2 Add system calls getruntime and stride

Now add two system calls `stride` and `getruntime`. Here is an example of how the system calls can be used in a user level application.

```
int pid = 5; // the process id of interest
stride(pid, 100); // sets the stride of process 5 to 100
int runtime = getruntime(pid); // returns the runtime of the process
printf("runtime of process %s is %d\n", pid, runtime);
```

The following steps explain how to add system calls.

First, we will add the kernel side code of the system calls.

kernel/syscall.h

☐ Each system call has a unique number that identifies it. This is used by the user side of the system call to indicate to the kernel code which system call to execute. Add the following system call ids to `kernel/syscall.h`.

```
#define SYS_stride 22
#define SYS_getruntime 23
```

The system call numbers are used by `kernel/syscall.c` as an index to the `syscalls[]` array of function pointers, which we will modify next.

kernel/syscall.c

☐ Look at lines 89 to 129 in `kernel/syscall.c`. They declare the functions of the system calls that will be called on the kernel side and add them to the `syscalls[]` array. Follow the pattern in the code to add the new system call functions.

```
extern uint64 sys_stride(void);
extern uint64 sys_getruntime(void);
...
[SYS_stride]  sys_stride,
[SYS_getruntime]  sys_getruntime,
```

kernel/proc.h

☐ The Process Control Blocks (PCBs) in xv6 are stored in an array called `proc` (we will refer to this as the process table) which is declared in `kernel/proc.c`. The `struct proc` is defined in `kernel/proc.h`. We want each process to have a stride value, so add an `int stride` to the `struct proc`.

kernel/proc.c

For simplicity, most of the code will go into `kernel/proc.c`. There are a few updates that need to be made in this file.

1. ☐ The default stride value of a process should be 0. Find the function `freeproc()`. Notice that this function zeros out many fields of proc structure so it can be reused. Follow the example and set the stride value of the process to 0.

2. Now create the full definitions for the two new system calls. This code should also go near the top of `kernel/proc.c` (or `kernel/sysproc.c`). Keep in mind that C is sensitive to the order in which variables and functions are declared (something can't be used before it is declared).

a. ☐ The first system call has the following function signature on the kernel side. Note that the function takes no parameters, that is because the user can't call this function directly, it gets executed by an interrupt handler that doesn't know what parameters to call it with!

```
uint64
sys_stride(void) {
```

The goal of the system call `sys_stride` is to set the stride value. See section 2.1 for the reason the parameter cannot be obtained directly. Instead `argint` should be used. Learn from `sys_sleep` in `sysproc.c` how this function is used.

You can also learn from `sys_sleep` how to obtain the current user process. When you are obtained it, set the stride to the value passed by the system call.

b. ☐ The second system call has the following function signature on the kernel side.

```
uint64
sys_getruntime(void) {
```

This system call has no parameters, but it does return a value. Simply return from `sys_getruntime` with the value obtained from the PCB.

For a user application to call a system call the call must be declared as a function. A Perl script takes care of generating the assembly code. Update the following two user side files.

`user/usys.pl`

☐ Add the new system call to the Perl script that automatically generates the required assembly code. Follow the example of `uptime` to add entries for `getruntime` and `stride`.

user/user.h

☐ Add the following system call declarations.

```
int getruntime();
int stride(int pid, int stride);
```

3.3 A queue data structure for the new schedulers

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.

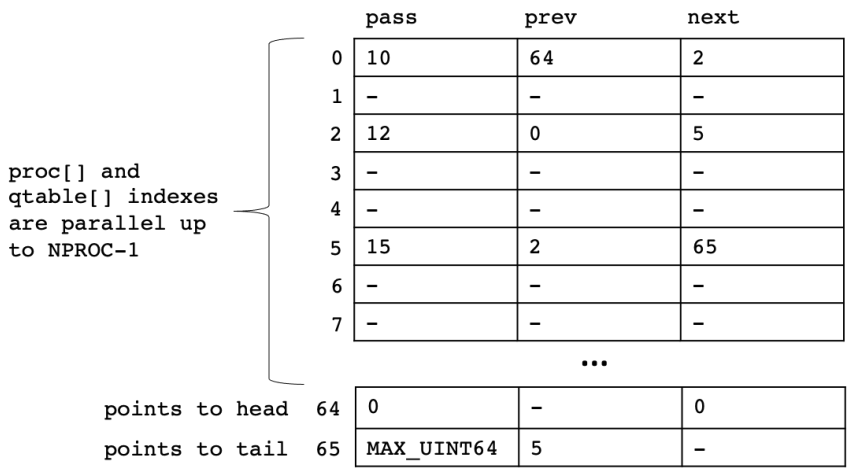
In xv6 the maximum number of processes is fixed at `NPROC` (default 64) as defined in `kernel/param.h`. Because the maximum number of processes is small, it suggests using static memory for the data structures in this project. It is a common theme in system programming to prefer static memory and in-place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

☐ Add 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

For this section, the things to do are:

- Add `#define SCHEDULER` to `kernel/param.h` to select among different schedulers when testing.
- Create two 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 2.3 describes how the scheduler originally gets called when the system boots. Use `SCHEDULER` to call the correct scheduler.

3.5 Testing

☐ Before implementing the new schedulers create some tests that you will use to evaluate them. 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 stride values to 4 and 12. Both processes then enter long CPU-bursts that lasts several ticks, for example, spinning in a long loop.

We would expect the child with stride of 4 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 stride values to 4 and 12. 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); // sleep puts process into blocked state
}
```

Collect the results from the test using `getruntime()`.

3.6 Create a new queue-based round-robin scheduler

Before starting this make sure you know how the scheduler works, see Section 2.3.

The rules for the new round-robin scheduler are:

- 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;
```

3.7 Stride scheduler

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.

3.7.1 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.

3.7.2 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*.

3.8 Documentation

☐ Include a `README` file with a brief description and a list of all files added to the project. Add a report of the test results from section 2.5 in the README file. Are the results what you expected?

☐ 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.

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-1b-xv6-riscv.zip xv6-riscv
```

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