Project 1C
COM S 352
Spring 2023

(Due: Friday March 31)

1. Introduction

For this project iteration, you will implement a fair scheduler following the Linux's complete fair scheduler (CFS) design principles. The tasks are summarized in the following table.

Task		**Points**
3.1 share scheduler	**3.1.5 Help Function: weight_sum function**	**3**
	3.1.5 Help Function: shortest_runtime_proc	**6**
	3.1.6 Function: cfs_sheduler	**10**
	Others	**6**
3.2 Integrate share scheduler with RR scheduler		**2**
3.3 System calls	**nice**	**6**
	startcfs	**4**
	stopcfs	**4**
3.4 Set up test		**3**
3.5 Documentation		**6**

Implementing and testing the schedulers requires understanding how preemption and time keeping works in xv6. Section 2 provides this background. Second 3 provides detailed guidance on how to implement our fair scheduler.

2. Background

2.1 Review of scheduler( )

Xv6 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. 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 and Guides

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 Implement a fair scheduler

The core of this project is to implement a fair scheduler, according to the Linux CFS design principles. We will implement the scheduler in kernel/proc.c, somewhere before the function of scheduler(void). The implementation includes the steps as follows.

3.1.1 Add nice and vruntime to struct proc

For this step, the things to do are:

- Add a **nice** field to `struct proc` of kernel/proc.h to indicate the priority of the process.
- Add a **vruntime** field to `struct proc` to keep track the virtual runtime of the process.
- Initialize both the **nice** and the **vruntime** to 0 for a new process in function freeproc in kernel/proc.c.

Note that, in later steps we will introduce a system call to set the **nice** value for a process and will update the **vruntime** in the implementation of a fair scheduler.

3.1.2 Declare the following system parameters for the fair scheduler to kernel/proc.c

int cfs_sched_latency = 100; //default length of scheduling latency
int cfs_max_timeslice = 10; //max number of timeslices for a process per scheduling latency
int cfs_min_timeslice = 1; //min number of timeslices for a process per scheduling latency

3.1.3 Define the nice-to-weight conversion table in kernel/proc.c

Recall that the CFS algorithm derives the weight from the nice of a process, and uses weight to determine the priority. Following is the array for the conversion and it should be added to kernel/proc.c before the array is referenced. Note that, the array entries are in the range of 0~39, but nice values are in the range of –20~19; hence, nice x should be converted to nice_to_weight[x+20].

```
int nice_to_weight[40] = {
        88761, 71755, 56483, 46273, 36291,  /*for nice = -20, …, -16*/
        29154, 23254, 18705, 14949, 11916, /*for nice = -15, …, -11*/
        9548, 7620, 6100, 4904, 3906, /*for nice = -10, …, -6*/
        3121, 2501, 1991, 1586, 1277, /*for nice = -5, …, -1*/
        1024, 820, 655, 526, 423, /*for nice = 0, …, 4*/
        335, 272, 215, 172, 137, /*for nice = 5, …, 9*/
        110, 87, 70, 56, 45, /*for nice = 10, …, 14*/
        36, 29, 23, 18, 15, /*for nice = 15, …, 19*/
};
```

3.1.4 Declare variables for fair scheduler in kernel/proc.c

To facilitate our implementation of fair scheduler, the following variables should be declared and initialized.

int cfs = 0; //indicate if the fair scheduler is the current scheduler, 0 by default
struct proc *cfs_current_proc=0; //the process currently scheduled to run by the fair scheduler and is initialized to 0
int cfs_proc_timeslice_len=0; //number of timeslices assigned to the above process

int cfs_proc_timeslice_left=0; //number of timeslices that the above process can still run

3.1.5 Develop help functions in kernel/proc.c

```
int weight_sum(){
        //to add: compute the sum of the weights of all the RUNNABLE processes,
        //and return the sum
}

struct proc* shortest_runtime_proc(){
        //to add: find the RUNNABLE process that has the shortest vruntime and return it.
        //If no process is RUNNABLE, return 0.
}
```

3.1.6 Implement scheduler function cfs_scheduler in kernel/proc.c

At the core of the fair scheduler implementation is the cfs_scheduler function. The framework and some important parts of the functions have been provided. Please fill the missing parts following the "**to add:**" signs.

```
void cfs_scheduler(struct cpu *c)
{
   //initialize c->proc, which is the process to be run in the next timeslice
   c->proc = 0;

   //decrement the current process' left timeslice
   cfs_proc_timeslice_left -=1;

   if(cfs_proc_timeslice_left > 0 && cfs_current_proc->state == RUNNABLE){

        //when the current process hasn't used up its assigned timeslices and is runnable
        //it should continue to run the next timeslce
        c->proc = cfs_current_proc;

   }else if(cfs_proc_timeslice_left == 0 ||
        (cfs_current_proc != 0 && cfs_current_proc->state != RUNNABLE)){

        //when the current process uses up its timeslices or becomes not runnable
        //it should not be picked to run next and its vruntime should be updated
        int weight = nice_to_weight[cfs_current_proc->nice+20]; //convert nice to weight
        int inc = (cfs_proc_timeslice_len - cfs_proc_timeslice_left) * 1024 / weight;
                        //compute the increment of its vruntime according to CFS design
        if(inc<1) inc=1; //increment should be at least 1
        cfs_current_proc->vruntime += inc; //add the increment to vruntime
        //prints for testing and debugging purposes
        printf("[DEBUG CFS] Process %d used up %d of its assigned %d timeslices and is
        swapped out!\n",
                cfs_current_proc->pid,
                cfs_proc_timeslice_len - cfs_proc_timeslice_left,
                cfs_proc_timeslice_len
        );
    }

    if(c->proc==0){

        //to add:
        //(1) Call shortest_runtime_proc() to get the proc with the shorestest vruntime
        //(2) If (1) returns a valid process, set up cfs_current_proc, cfs_proc_timeslice_len,
        //    cfs_proc_timeslice_left and c->proc accordingly.
        //    Notes: according to CFS, a process is assigned with time slice of
        //    ceil(cfs_sched_latency * weight_of_this_process / weights_of_all_runnable_process)
        //    and the timeslice length should be in [cfs_min_timeslice, cfs_max_timeslice]
        //(3) If (1) returns 0, do nothing.


        if(c->proc > 0){
          //prints for testing and debugging purposes
          printf("[DEBUG CFS] Process %d will run for %d timeslices next!\n",
                        c->proc->pid,
                        cfs_proc_timeslice_len);

          //schedule c->process to run
          acquire(&c->proc->lock);
          c->proc->state=RUNNING;
          swtch(&c->context, &c->proc->context);
          release(&c->proc->lock);
        }

    }
}
```

3.2 Integrate the fair scheduler with the current scheduler of xv6

Replace the current scheduler(void) function with the following two functions:

```
//The original RR scheduler is moved to old_scheduler
void
```

```
old_scheduler(struct cpu *c)
{
  struct proc *p;
  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);
  }
}


//The scheduler runs the original RR scheduler (if cfs==0) or our new fair scheduler (if cfs==1)
void
scheduler(void)
{
        struct cpu *c = mycpu();
        c->proc=0;

        for(;;){
            // Avoid deadlock by ensuring that devices can interrupt.
            intr_on();
            if(cfs){
                        cfs_scheduler(c);
            }else{
                        old_scheduler(c);
            }
        }
}
```

3.3 Add system calls to set nice value for process and to start/stop the fair scheduler

Add the following system calls to allow user programs to provide inputs to the scheduler. Please follow the procedures introduced in Project 1.B to add them to the relevant files in both kernel and user folders.

3.3.1 System call nice

```
int nice(int new_nice);
```

If new_nice is an integer between –20 and 19, the caller's nice is set to new_nice. Otherwise, the nice value is unchanged. The system returns the nice value (after update) of the caller.

Hint: in your kernel-side implementation function sys_nice(void), you may use the following snippet of code for fetching the argument (i.e., int new_nice) from user space:

```
int new_nice;
argint(0, &new_nice);
```

3.3.2 System calls startcfs and stopcfs

```
int startcfs(void);
```

This system call starts the share scheduler by setting the variable cfs to 1 in proc.c. It returns 1.

```
int stopcfs(void);
```

This system call stops the share scheduler by setting the variable cfs to 0 in proc.c. It returns 1.

3.4 Test the system

Incorporate the following snippet of test code to a test user program, named **testsyscall.c**, and make necessary changes to make testsyscall runnable in xv6 with make qemu.

```
int main()
{
    //start the share scheduler
    startcfs();

    //create 10 child processes of nice=10 (note: the parent process is of nice=0 by default)
    int ret=0;
    for(int i=0; i<10; i++){
        ret=fork1();
        if(ret==0){
            nice(10);
            break;
        }
    }
    //all processes run the same code as follows
    printf("process (pid=%d) has nice = %d\n", getpid(),nice(-30));
    int t=0;
    while(t++<2){
        double x=987654321.9;
        for(int i=0; i<100000000; i++){
            x /= 12345.6789;
        }
    }

    //parent process waits for child processes to terminate and then stop share scheduler
    if(ret>0){
        wait(0);
        stopcfs();
    }

    return 0;
}
```

3.5 Documentation

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`.