Project 1
Wiki Documentation

This project focuses on the implementation of a system scheduler in a RISC-V xv6 architecture. Given that a Round-Robin (RR) scheduler is used by default, the objective of this project is to implement a First-Come, First-Served (FCFS) and Multi-Level Feedback Queue (MLFQ) scheduler. In this case, there are five system calls to be implemented, with each serving a purpose in implementing the two schedulers.

Given that the built-in scheduler function in the proc.c file uses a RR scheduler, I decided to reference its implementation merely as a guide in defining and implementing the FCFS and MLFQ schedulers. I approached this assignment by implementing the following system calls: yield(), getlev(), setpriority(), fcfsmode(), and mlfqmode() by modifying the following files: sysproc.c, syscall.c, syscall.h, defs.h, user.h, user.pl, proc.c, and proc.h. Below are the following steps I took to implement these.

Step 1: Define the system calls in the sysproc.c file

In order to create a new system call, it should be defined in the sysproc.c file as the file specifies the implementation of these system calls and overall contains the system calls that handle processes and its management.

I implemented five system calls: sys_yield(void), sys_getlev(void), sys_setpriority(void), sys_fcfsmode(void), and sys_mlfqmode(void). Each serves a different role in implementing system schedulers. Since the yield() function exists as a built-in function, I called the function in the syscall. For getlev and set priority, I made it so that getlev returns the queue level of the process and setpriority sets the priority of the process with the pid. Additionally, the argint() call in sys_setpriority() extracts integer arguments passed from a user program to the kernel syscall, thus it needed to be changed to return a value (void → int). More specific implementations of the functions were made in the proc.c file.

```c
extern int current_mode;
uint64
sys_yield(void)
{
  yield();
  return 0;
}

uint64
sys_getlev(void)
{
  struct proc *p = myproc();
  if (current_mode == FCFS)
      return 99;
  return p->queue_level;
}

uint64
sys_setpriority(void)
{
  int pid, priority;
  if(argint(0, &pid) < 0 || argint(1, &priority) < 0)
    return -1;

  return setpriority(pid, priority);
}

uint64
sys_mlfqmode(void)
{
  return mlfqmode();
}
uint64
sys_fcfsmode(void)
{
  return fcfsmode();
}
```

Figure 1. sysproc.c file

Step 2: Define the functions in the defs.h file

I defined the functions in the defs.h file so that I could access these files when initializing in the syscall.c file. Since ‘yield’ is an in-built function, I just created four definitions for the remaining four functions. Also, I changed the argint(int, int*) from void to an int function so that it returned a value when used in the setpriority() system call.

```
void            yield(void);
int             either_copyout(int user_dst, uint64 dst, void *src, uint64 len);
int             either_copyin(void *dst, int user_src, uint64 src, uint64 len);
void            procdump(void);
int             getlev(void);
int             setpriority(int pid, int priority);
int             mlfqmode(void);
int             fcfsmode(void);
```

```
// syscall.c
int             argint(int, int*);
```

Figure 2. defs.h file

Step 3: Register the system calls in the syscall.c file

I made a few changes to the syscall.c file, which added/registered the new functions as new system calls. Moreover, I changed the argint() function to an int function and included a “return 0” at the end so that it can be used when it is called in the setpriority function.

```
extern uint64 sys_yield(void);
extern uint64 sys_getlev(void);
extern uint64 sys_setpriority(void);
extern uint64 sys_mlfqmode(void);
extern uint64 sys_fcfsmode(void);
```

```
[SYS_yield]   sys_yield,
[SYS_getlev]  sys_getlev,
[SYS_setpriority] sys_setpriority,
[SYS_mlfqmode] sys_mlfqmode,
[SYS_fcfsmode] sys_fcfsmode,
```

```
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
  *ip = argraw(n);
  return 0;
}
```

Figure 3. syscall.c file

Step 4: Register the functions in the system call numbers

In addition to registering the functions in the syscall.c file, I also registered it in the syscall.h file by defining its system call number to the latest available number, which in this case is 22-26. This is done so that a number corresponds to each specific system call.

```
#define SYS_yield  22
#define SYS_getlev  23
#define SYS_setpriority 24
#define SYS_mlfqmode    25
#define SYS_fcfsmode    26
```

Figure 4. syscall.h file

Step 5: Initializing variables in the proc.h file

In order to use certain variables in the proc.c file, I initialized the following variables in the proc.h file. I defined RR as 0, FCFS as 1, and MLFQ as 2, and I defined an extern variable current_mode that indicates which scheduler mode is being used.

```c
#define RR 0
#define FCFS 1
#define MLFQ 2
extern struct cpu cpus[NCPU];
extern int current_mode;
```

```c
  int priority;
  int queue_level;
  int ctime;
  int time_quantum;
  int ticks;
```

Figure 5. proc.h file

Step 6: Adding required specifications to the functions in the proc.c file

Several changes were made to the proc.c file. First of all, I initialized the current_mode as FCFS in order to schedule processes within FCFS after the first xv6 boot. I also created a boost_ticks variable that will be used in the MLFQ scheduler.

```c
int current_mode = FCFS;
struct spinlock pid_lock;
int boost_ticks = 0;
```

Figure 6. proc.c file (initializing variables)

Next, I added additional initializations to the allocproc function that will be used in the scheduler. Here, ctime helps manage process scheduling based on their creation time, queue_level and priority allow for adjustments based on the process's behavior, and time_quantum is used to manage how long processes run before being preempted.

```c
  // Set up new context to start executing at forkret,
  // which returns to user space.
  memset(&p->context, 0, sizeof(p->context));
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;
  p->ctime = ticks;
  p->queue_level = 0;
  p->priority = 3;
  p->time_quantum = 0;
  return p;
}
```

Figure 7. proc.c file (allocproc() function)

Afterwards, I implemented the following functions: setpriority, mlfqmode, and fcfsmode. setpriority() returns 0 when priority of the process was set with the given pid properly, -1 if theres no process with the given pid, and -2 if the priority value is not between 0 and 3. mlfqmode() and fcfsmode() returns 0 when it successfully transitions modes, and -1 if the system is already in MLFQ/FCFS mode.

```c
int
setpriority(int pid, int priority)
{
  struct proc *p;
    if (priority <= 0 || priority >= 3) {
        return -2;
    }
    for (p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if (p->pid == pid) {
            p->priority = priority;
            release(&p->lock);
            return 0;
        }
        release(&p->lock);
    }
    return -1;
}
```

```c
int mlfqmode(void) {
  if (current_mode == MLFQ) {
    return -1;
  }
  current_mode = MLFQ; //switch
  ticks = 0; //reset ticks
  for (struct proc *p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if (p->state != UNUSED) {
      p->queue_level = 0; //Lo queue
      p->priority = 3; //reset
      p->time_quantum = 0; //reset
    }
    release(&p->lock);
  }
  return 0;
}
```

```c
int fcfsmode(void) {
  if (current_mode == FCFS) {
    return -1;
  }
  current_mode = FCFS;
  ticks = 0;
  for (struct proc *p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if (p->state != UNUSED) {
      p->queue_level = 0;
      p->priority = 3;
      p->time_quantum = 0;
    }
    release(&p->lock);
  }
  return 0;
}
```

Figure 8. proc.c file (setpriority(), fcfsmode(), mlfqmode() function)

The scheduler() function runs in an infinite loop (for(;;)) to continuously check and schedule processes. The intr_on() enables interrupts to prevent deadlocks. It switches between three different scheduler modes: RR, FCFS, and MLFQ. It first checks if the current_mode is the RR mode, which is the default scheduler of xv6. It then checks each process to see if it is runnable and if a process is found, it is switched to the RUNNING state and the context is switched to that process. If no runnable process is found, the CPU waits for an interrupt using asm volatile("wfi").

```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    intr_on();
    if (current_mode == RR){
      // The most recent process to run may have had interrupts
      // turned off; enable them to avoid a deadlock if all
      // processes are waiting.
      int found = 0;
      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;
          found = 1;
        }
        release(&p->lock);
      }
      if(found == 0) {
        // nothing to run; stop running on this core until an interrupt.
        intr_on();
        asm volatile("wfi");
      }
    }
```

Figure 9. proc.c file (scheduler() RR)

Next, it checks if the current_mode is the FCFS mode. Processes are executed in the order they are created based on their creation time (ctime). The loop iterates over the processes and selects the one with the minimum creation time, which is the process created the earliest. If no process is runnable, the CPU waits for an interrupt.

```
else if (current_mode == FCFS) {
  struct proc *minp = 0; //minimum process
  for (p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if (p->state == RUNNABLE) {
      if (minp == 0 || p->ctime < minp->ctime) {
        if (minp) {
          release(&minp->lock);
        }
        minp = p;
      }
      else {
        release(&p->lock);
      }
    }
    else {
      release(&p->lock);
    }
  }
  if (minp) {
    minp->state = RUNNING;
    c->proc = minp;
    swtch(&c->context, &minp->context);
    c->proc = 0;
    release(&minp->lock);
  }
  else {
    intr_on();
    asm volatile("wfi");
  }
}
```

Figure 10. proc.c file (scheduler() FCFS)

Lastly, it checks if the current_mode is the MLFQ mode. It uses multiple queues, each representing a different priority level. Processes are initially placed in higher-priority queues and are demoted to lower-priority queues if they exhaust their time slices. In order to prevernt starvation, all processes are boosted by resetting their queue level to 0 (highest priority) every 50 ticks (priority boosting). The scheduler looks for runnable processes in the highest priority queue (level 0) and if no processes are found in level 0, it checks level 1 then level 2. If a process uses up its time slice, it is demoted to a lower priority level and if no processes are found in any of the levels, the system defaults to FCFS by calling fcfsmode().

```c
      else if (current_mode == MLFQ) {
        struct proc *p;
        struct proc *chosen = 0;
        int level;

        boost_ticks++;
        if (boost_ticks >= 50) { //priority boost every 50 ticks
          for(p = proc; p < &proc[NPROC]; p++) { //reset after 50 ticks
            acquire(&p->lock);
            if (p->state == RUNNABLE || p->state == RUNNING) {
              p->queue_level = 0;
              p->ticks = 0;
            }
            release(&p->lock);
          }
          boost_ticks = 0;
        }
        for (level = 0; level <= 2; level++) { //checking eqch queue level
          for(p = proc; p < &proc[NPROC]; p++) {
            acquire(&p->lock);
            if (p->state == RUNNABLE && p->queue_level == level) {
              chosen = p;
              goto found_mlfq; //if process found -> exit loop
            }
            release(&p->lock);
          }
        }
        fcfsmode();

        found_mlfq:
          if (chosen) {
            chosen->state = RUNNING;
            c->proc = chosen;
            chosen->ticks++; //increment ticks

            swtch(&c->context, &chosen->context); //context swtich
            c->proc = 0;

            if (chosen->state == RUNNABLE) { //demote to lower level if exhausted time slice
              if ((chosen->queue_level == 0 && chosen->ticks >= 1) ||
                  (chosen->queue_level == 1 && chosen->ticks >= 2) ||
                  (chosen->queue_level == 2 && chosen->ticks >= 4)) {
                if (chosen->queue_level < 2) {
                  chosen->queue_level++;
                }
                chosen->ticks = 0; //reset
              }
            }
          release(&chosen->lock);
        }
        else {
          intr_on();
          asm volatile("wfi");
        }
      }
    }
  }
```

Figure 11. proc.c file (scheduler() MLFQ)

Step 7: Declare the function in the user.h file

In order to make the system call available at the user program, I declared the function in the user/user.h file.

```
void yield(void);
int getlev(void);
int setpriority(int, int);
int mlfqmode(void);
int fcfsmode(void);
```

Figure 12. user.h file

Step 8: Add a new macro in the usys.pl file

Afterward, I added new macros in the usys.pl file so that it generates a system call stub that allows user programs to invoke these system calls. The usys.pl file overall defines how to preprocess when using the system call in a user program.

```
entry("yield");
entry("getlev");
entry("setpriority");
entry("mlfqmode");
entry("fcfsmode");
```

Figure 13. usys.pl file

Step 9: Add the code file to the makefile

Adding the code $U/_test\ into the makefile ensures that the user program test.c is properly compiled and linked once I run the makefile.

```
UPROGS=\
	$U/_cat\
	$U/_echo\
	$U/_forktest\
	$U/_grep\
	$U/_init\
	$U/_kill\
	$U/_ln\
	$U/_ls\
	$U/_mkdir\
	$U/_rm\
	$U/_sh\
	$U/_stressfs\
	$U/_usertests\
	$U/_grind\
	$U/_wc\
	$U/_zombie\
	$U/_test\
```

Figure 14. Implementing the function

RESULTS

After implementing the getppid() function as a system call and ppid.c user program, I booted the xv6 kernel to execute the ppid user program. The results are as follows:

```
project01-2023049998 — qemu-system-riscv64 ◂ make qemu — 101×41
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1  -nographic -global
 virtio-mmio.force-legacy=false -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device
,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

init: starting sh
$ test
FCFS & MLFQ test start

[Test 1] FCFS Queue Execution Order
Process 4 executed 100000 times
Process 5 executed 100000 times
Process 6 executed 100000 times
Process 7 executed 100000 times
[Test 1] FCFS Test Finished

nothing has been changed
successfully changed to MLFQ mode!

[Test 2] MLFQ Scheduling
Process 8 (MLFQ L0-L2 hit count):
L0: 10201
L1: 10788
L2: 79011
Process 9 (MLFQ L0-L2 hit count):
L0: 10869
L1: 11388
L2: 77743
Process 10 (MLFQ L0-L2 hit count):
L0: 16497
L1: 22616
L2: 60887
Process 11 (MLFQ L0-L2 hit count):
L0: 16567
L1: 22485
L2: 60948
[Test 2] MLFQ Test Finished

FCFS & MLFQ test completed!
$
```

Figure 15. Execution results

As seen in the photo, the program tests the FCFS scheduling, switches modes to MLFQ, then testing the MLFQ scheduling.

As for the flow of operation, given the test.c file, it starts with the fork() process being called to create four child processes that involves incrementing its own counter until it reaches NUM_LOOP (= 100,000). Here, the FCFS makes it so that each proces is run until its complete before starting the next one, and it also means that processes are run in their PID/creation order.

Afterwards, the mode checks to see if it is in FCFS mode then if it is in MLFQ mode. Since it was in FCFS mode to begin with (because current_mode is assigned to be FCFS when the xv6 is booted), it prints "nothing has been changed". Then, it switches to MLFQ mode and prints out the confirmation.

In the MLFQ scheduler, it creates four child process then it calls getlev() to check its current level. It increments count[level] based on the result then it prints how many times it was scheduled at each level.

TROUBLESHOOTING

When trying to run the test.c file twice in a row, I was getting a message that despite starting and testing the FCFS mode, it says “successfully changed to FCFS mode!” before switching to MLFQ mode, which shouldn’t be the case considering that after MLFQ is tested and a new test is called, it should switch back to FCFS mode.

```
project01-2023049998 — qemu-system-riscv64 • make qemu — 101×34
$ test
FCFS & MLFQ test start

[Test 1] FCFS Queue Execution Order
Process 13 executed 100000 times
Process 14 executed 100000 times
Process 15 executed 100000 times
Process 16 executed 100000 times
[Test 1] FCFS Test Finished

successfully changed to FCFS mode!
successfully changed to MLFQ mode!

[Test 2] MLFQ Scheduling
Process 17 (MLFQ L0-L2 hit count):
L0: 1056
L1: 11401
L2: 87543
Process 18 (MLFQ L0-L2 hit count):
L0: 5635
L1: 11368
L2: 82997
Process 19 (MLFQ L0-L2 hit count):
L0: 11408
L1: 22483
L2: 66109
Process 20 (MLFQ L0-L2 hit count):
L0: 11415
L1: 22536
L2: 66049
[Test 2] MLFQ Test Finished

FCFS & MLFQ test completed!
$
```

I tried to fix this issue by calling fcfsmode() after mlfq scheduler is run, given the specification that “the fcfsmode system call is invoked in MLFQ mode, processes will be scheduled within the FCFS again.”

```
    for (level = 0; level <= 2; level++) { //checking eqch queue level
      for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if (p->state == RUNNABLE && p->queue_level == level) {
          chosen = p;
          goto found_mlfq; //if process found -> exit loop
        }
        release(&p->lock);
      }
    }
    fcfsmode();

```

By doing so, when running the test consecutively, it prints out “nothing has been changed” before switching to MLFQ mode.

```
xv6 kernel is booting

init: starting sh
$ test
FCFS & MLFQ test start

[Test 1] FCFS Queue Execution Order
Process 4 executed 100000 times
Process 5 executed 100000 times
Process 6 executed 100000 times
Process 7 executed 100000 times
[Test 1] FCFS Test Finished

nothing has been changed
successfully changed to MLFQ mode!

[Test 2] MLFQ Scheduling
Process 8 (MLFQ L0-L2 hit count):
L0: 1054
L1: 10546
L2: 88400
Process 9 (MLFQ L0-L2 hit count):
L0: 5086
L1: 10617
L2: 84297
Process 10 (MLFQ L0-L2 hit count):
L0: 7988
L1: 20562
L2: 71450
Process 11 (MLFQ L0-L2 hit count):
L0: 10578
L1: 20682
L2: 68740
[Test 2] MLFQ Test Finished

FCFS & MLFQ test completed!
```

```
$ test
FCFS & MLFQ test start

[Test 1] FCFS Queue Execution Order
Process 13 executed 100000 times
Process 14 executed 100000 times
Process 15 executed 100000 times
Process 16 executed 100000 times
[Test 1] FCFS Test Finished

nothing has been changed
successfully changed to MLFQ mode!

[Test 2] MLFQ Scheduling
Process 17 (MLFQ L0-L2 hit count):
L0: 9405
L1: 21295
L2: 69300
Process 18 (MLFQ L0-L2 hit count):
L0: 10505
L1: 21222
L2: 68273
Process 19 (MLFQ L0-L2 hit count):
L0: 15927
L1: 32015
L2: 52058
Process 20 (MLFQ L0-L2 hit count):
L0: 15751
L1: 26301
L2: 57948
[Test 2] MLFQ Test Finished

FCFS & MLFQ test completed!
```

Figure 16. test 1 (left) and test 2 (right)