Project 1B
COM S 352
Fall 2023

1. Introduction

For this project iteration, you will prepare system calls and utilities that will be used to implement and evaluate new schedulers for the xv6 operating system. Below is an overview of the tasks to complete with the section numbers of where they are described in more detail.

Task	Document Section(s)

1. Add a new system call `getlog` with the following specification.
 	
`int getlog(struct logentry*);`
 	
The `logentry` is a **pointer to an array**. The array is updated by the kernel to contain the log before return.
 	
`logentry` – points to the first entry of an **array** of log entries
Returns: the number of valid log entries that have been added to the log
 	
Notes:
`struct logentry {`
`  int pid; // process id`
`  enum procstate fromstate;`
`  enum procstate tostate;`
`  int time; // time measured in ticks`
`};`
	2.1 – 2.3

2. Add a new system call `nice` with the following specification. `int nice(int inc);` Adds `inc` to the current nice value. The range of the nice value is +19 (low priority) to -20 (high priority). Attempts to set a nice value outside the range are clamped to the range. `inc` – the amount to add to the current nice value Returns: the updated nice value	2.1 – 2.3
3. Instrument the scheduler to record all process state transitions to the log. For example, the log will contain entries like entries like `{pid=5, fromstate=RUNNABLE, tostate=RUNNING, time=100}`.	2.4

4. Modify the current round-robin scheduler to also consider priority.	2.5
5. Create a command line utility `schedtest` for testing the system calls. The test will use `fork` to create multiple processes.	2.6
6. Documentation (see submission instructions below)	3

2. Implementation Guide

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

2.1. Setup

`kernel/schedlog.h and kernel/schedlog.c`

☐ Download the provided code and add `schedlog.h` and `schedlog.c` to the kernel.

`Makefile`

☐ Search for `OBJS` and add `$K/schedlog.o` to the list of kernel files. Follow the pattern of the list.

2.2. Adding System Calls on the Kernel Side

Systems calls allow user code to access functions of the kernel. A direct function call would not be secure. Instead, the system call provides a mechanism to transition CPU execution from user mode to kernel mode. To implement a system call we need to add code on both the user and kernel side. First, add the kernel side code as described here.

`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_getlog 22
#define SYS_nice   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_getlog(void);
extern uint64 sys_nice(void);
...
[SYS_getlog]  sys_getlog,
[SYS_nice]    sys_nice,
```

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 nice value, so add an `int nice` 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. ☐ Add the following two lines to the end of the includes.

   ```
   #include "schedlog.h"
   ```

2. ☐ The default nice 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 nice value of the process to 0.

3. Now create the full definitions for the two new system calls. This code should also go near the top of `kernel/proc.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_getlog(void) {
      ```

      The goal of the system call `sys_getlog` is to provided the user with a copy of the log. The log (defined in `kernel/schedlog.c`) is contained in kernel memory, it is not possible for an application to access it directly because it is

outside of the applications virtual address space. To provided data to an application we must copy the data into its virtual address space. The application must facilitate this by allocating the data structure and passing the virtual address with the system call. A helper function in xv6 knows how to extract the system call argument from the stack.

```
uint64 user_log_virt_addr;

argadd(0, &user_log_virt_addr);
```

Copying the data from kernel memory to application memory must be done with care, because the applications memory is broken into pages which may be located anywhere in physical memory. What happens if the data is spread across multiple pages? In other words, a simple call to a C library function like `memcpy()` does not work. Another helper function in xv6 copies a given number of bytes from a specified location in physical memory to a given process' virtual memory.

```
struct proc *p = myproc();
int log_entry_size = sizeof(struct logentry);
int log_size = schedlognext;

int copy_result = copyout(p->pagetable,
    user_log_virt_addr,
    (char *)schedlog,
    log_entry_size*log_size) < 0);
```

The function `copyout()` returns 0 on success or -1 on error. An example of an error would be if the copy went past the allocated memory for the process. `sys_getlog` should return the length of the log (in number of log entries) or -1 on error. Add that now to finish up the definition of `sys_getlog()`.

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

```
uint64
sys_nice(void) {
```

As before, the parameters must be manually extracted from the applications call stack. In this case, the parameter to `sys_nice` is a single int that represents how much to add to the current value of nice. An int can be extracted from the call stack with the xv6 helper function `argint`.

```
int inc; // the increment
// set inc to the argument passed by the user
argint(0, &inc);
```

Now update the nice value of the process. Recall that you added nice to the struct proc in

a previous step. You can get the current (interrupted) user process with the following.

```
struct proc *p = myproc();
```

The math is *NewNiceValue* = *OldNiceValue* + *Increment*, where the *Increment* may be positive or negative. However, the nice value is clamped to the range -20 to 19. In other words, set the value to -20 if it is below that value or 19 if it is above that value. Update the nice value in the struct pointed to by `p`. Recall that you added nice to the struct proc in a previous step.

Finally, this system call must return the new (updated) nice value. We have not implemented the nice system call to behave in the same way as it does in Linux https://man7.org/linux/man-pages/man2/nice.2.html.

2.3. Adding System Calls on the User Side

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 `getlog` and `nice`.

`user/user.h`

☐ At the top of the file add the following.

```
struct logentry;
```

☐ Add the following system call declarations.

```
int getlog(struct logentry*);
int nice(int inc);
```

2.4. Instrumenting the Scheduler

☐ Process state transitions are controlled in `kernel/proc.c`. For example, the line 128 transitions the state from `UNUSED` to `USED`.

```
p->state = USED;
```

The USED state indicates a new process has been created and is not occupying the slot in the process table array. Line 254 transitions the new process from `USED` to `RUNNABLE`.

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

Search for "state" to identify every state transition in the file and log each one. The initialization of the process table, specifically on line 58 in the `procinit()` function should not be logged.

Call `logproc()` which is defined in `/kernel/schedlog.c`. It has the signature below. The returned error code can be ignored. The state is transition from `fromstate` to `tostate`.

```
int logproc(struct proc* p, enum procstate fromstate,
                            enum procstate tostate);
```

2.5. Modify the Scheduler to be Priority Based

☐ Before making any modifications, get familiar with the xv6 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);
    }
  }
}
```

☐ Modify the `scheduler()` function so that any runnable process with a negative nice value always executes before any process with a nice value of 0 or greater. If there are multiple processes with negative nice values, they should execute in round-robin fashion with each other. Likewise, if there are multiple processes with non-negative nice values, they should also execute in round-robin fashion with each other.

2.6. Implement the schedtest Command

Implement a utility program in a file named `user/schedtest.c`.

From the command line, a user should be able to start a program and set its nice value like this:
`$ schedtest`

☐ In `Makefile`, search for `UPROGS` and add `$U/_schedtest` to the list of utilities. Follow the pattern of the list.

☐ Add the required includes.

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "kernel/user.h"
#define USER_LOG
#include "kernel/schedlog.h"
```

☐ Setup main.

```
int
main(int argc, char *argv[])
{
        exit(0);
}
```

Special notes about xv6 applications: xv6 is a real operating system, but it is very simple compared to Linux. You will not be able to include many of the libraries you are used to, for example there is no `stdio.h`. Instead, you must make do with what is provided in the xv6 libraries or make your own. Look at the other utility user programs for examples. One additional thing to note is that the system call `exit()` must be called to end an application in xv6. Executing a `return` from `main` will not correctly terminate the application.

☐ Now create a test. Use `fork()` to create 6 child processes. On 3 of the children use the new `nice()` system call to set their nice value to a negative number; on the other 3 set the nice value to a positive number. Have the 6 processes use up some CPU time. One way to do that is with a loop like this.

```
volatile int calc=0;
for (int i=0; i<100000; i++) {
     for (int j=0; j<10000; j++) {
          calc += i;
     }
}
```

☐ The parent process should wait for all 6 child processes to finish. Then it should call the new `getlog()` system call and print the log in a readable format. For example:

```
pid=1, fromstate=RUNNABLE, tostate=RUNNING, time=100
pid=1, fromstate=RUNNING, tostate=RUNNABLE, time=101
```

The getlog system call requires an array on the user side to store the logs, create one.

```
struct logentry schedlog[SCHED_LOG_SIZE];
```

The `wait()` system call in xv6 waits for any child to terminate. It takes one parameter, a pointer to status information, which is typically set to 0. The function can be called multiple times in the same application. Review Lecture 2 material for more information about the system calls or study the xv6 code for how they are implemented.

3. 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 program you write must also have brief comments on its purpose and usage.

Include a `README` file with your name, a brief description, and the results from running `schedtest`. Did you get the result you expected?

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`.