Project 1B
COM S 352
Fall 2022

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.

The project may be completed in pairs (2 maximum). The tasks for this project iteration are listed below. Tasks 1-3 are required when working as an individual and in pairs. All pairs must also complete task 4.

Task	Individual	Pair
1. Add the system call `startlog`. `int startlog(void);` Starts logging every context switch between user processes by the scheduler. The log consists of a fixed size array of `struct logentry`. Logging continues until the log buffer is full (e.g., `LOG_SIZE` entries). Returns: 0 if successful, -1 on error (e.g., the log has already been started)	15	11

2. Add the system call `getlog`.
 	
`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`
`  int time; // time measured in ticks`
`};`
	15	11

3. Add the system call nice. `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	15	11
4. Add the command line utility `nice`. **nice** *N PROG [ARG]...* `N` – the nice value (between -20 and 19) `PROG` – the program to execute `[ARG]…` – zero or more arguments that are passed to the program to execute Example usage: `$ nice 19 wc README` The example command results in `wc README` being executed at the nice value 19. Attempts to set the nice value outside the range of -20 to 19 should result in an error message and the command not being executed.	0	11
5. Documentation (see submission instructions below)	5	6

2. Implementation Guide

2.1. Adding System Calls on the Kernel Side

To add new systems calls on the kernel side, modify the following files.

kernel/syscall.h

Each system call has a unique number that identifies it. This is used by the user side application to indicate the desired system call. Add the following system call numbers.

```
#define SYS_startlog 22
#define SYS_getlog   23
#define SYS_nice     24
```

The system call numbers are used as an index into the `syscalls[]` array of function pointers, which we will modify next.

kernel/syscall.c

Follow the example of the other systems calls to add the system call declarations and function pointers in `syscalls[]`.

```
...
extern uint64 sys_startlog(void);
extern uint64 sys_getlog(void);
extern uint64 sys_nice(void);
...
[SYS_startlog] sys_startlog,
[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/log.h

Create a new header file to declare the log structure.

```
#define LOG_SIZE 100
struct logentry {
        int pid; // process id
        int time; // number of ticks
};
```

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. Include the new header file `log.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. We need some way to tell time so timestamps can be recorded in the log. Xv6 marks time in ticks, which a common approach for operating systems. Specifically, a hardware timer is configured to trigger an interrupt at some periodic interval (by default every 100ms),

which represents 1 tick. On each tick, context is switched to the scheduler, so the scheduler can decide: continue running the current user process or switch to a different one?

The function that is called by the timer interrupt is `yield()`. Find this function in `kernel/proc.c` and add to it to increment a timer variable. You need to create the timer variable; it can be declared near the top of `kernel/proc.c`. Initialize time to 0.

Now create the full definitions for the three 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). Fill in the TODOs in the example code below with appropriate code according to your approach.

```
struct logentry schedlog[LOG_SIZE];
// TODO: add additional variables as needed here

uint64
sys_startlog(void)
{

  // TODO: initiate logging
  // See Section 1 for the specifications of this function.

}

uint64
sys_getlog(void) {
    uint64 userlog; // hold the virtual (user) address of
                    // user's copy of the log
    // set userlog to the argument passed by the user
    argaddr(0, &userlog);


    // copy the log from kernel memory to user memory
    struct proc *p = myproc();
    if (copyout(p->pagetable, userlog, (char *)schedlog,
            sizeof(sturct logentry)*LOG_SIZE) < 0)
        return -1;

    // TODO: most of this function is done for you, just
    //       return the number of valid entries in the log
    //       (i.e., how much has been recorded since
    //       startlog was called)
}
```

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

  // get the current user process
  struct proc *p = myproc();

  // TODO: update and return the nice value of the current
  //       user process (p)
  // See Section 1 for the specifications of this function.

}
```

4. The final step is to record context switches to the log. The main function of the scheduler is appropriately called `scheduler()`. Study the code. When does a context switch happen? Add a log entry for the process that is being switched into.

2.2. 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 an entries for `startlog`, `getlog` and `nice`.

`user/user.h`

At the top of the file add the following.

```
struct logentry;
```

Add the following system call declarations.

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

2.3. Testing the System Calls (Not required, but recommended)

Modify an existing utility or create your own (see the nice command bellow for how to do that). An example of typical usage of the system calls is the following (be sure to include `kernel/log.h`).

```
struct logentry schedlog[LOG_SIZE];
...
nice(10);
startlog();
if (fork() > 0)
     sleep(5);
     getlog(schedlog);
}
```

2.4. Implement the Nice Command

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

From the command line, a user should be able to start a program and set its nice value like this:
`$ nice 19 wc README`

In the example, the command "`nice 19 wc README`" executes the `nice` utility program (the one you will implement), which in turn sets the nice value to 19 and then executes the command "`wc README`". The command "`wc README`" executes the `wc` (word count) utility program passing it the argument `README`.

The specification for the `nice` command is the following.

`nice` *`N PROG [ARG]...`*

`N` – the nice value
`PROG` – the program to execute
`[ARG]…` – zero to many arguments that are passed to the program to execute

How to start the code?
Create a new user file `user/nice.c`. To get the header files correct, a good starting point is to copy the code from a simple existing utility such as `user/zombie.c`. In the file `Makefile` search for `zombie` and add a similar line for `nice`.

How to get and parse the parts of the nice command?
Command line arguments are available in the char array `argv[]` passed into `main(int argc, char *argv[])`. For example, the command:

```
$ nice 19 wc README
```

results in main's `argv[]` looking like this:

```
argv[0] = "nice"
argv[1] = "19"
argv[2] = "wc"
```

```
argv[3] = "README"
argv[4] = 0 // the array ends with a null string
```

To convert a string into an integer use the function `atoi()`. Unfortunately, the xv6 version of atoi doesn't understand the negative sign, so you will need to check for that yourself and adjust accordingly.

The best way to learn how to write systems code is by looking at examples. In this case, two good examples are `user/wc.c` and `user/kill.c`.

How can `nice` start another program?
A process can change the program it is executing by calling `exec()`. The xv6 implement of `exec()` has the following definition.

```
int exec(char* pathname, char* argv[]);
```

The first parameter is the name of the command to execute, for example "`wc`". The second parameter takes an array of strings that must end with the last string being only a null character. Notice the similarity between the parameters of `exec()` and the arguments passed to `main(char* pathname, char* argv[])`. Remember that you can get the address of something with the symbol `&` and pointers can be treated as arrays. Using the example from the previous question `char **arr = (char **)&argv[2]` produces:

```
arr[0] = "wc" // the first element should be the program name
arr[1] = "README"
arr[2] = 0 // the array ends with a null
```

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 programs you write (if any) must also have brief comments on their purpose and usage.

Include a `README` file with your name(s), a brief description, and a list of all files added or modified in the project.

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`.