Operating Systems
Assignment 2 – *Easy*

Instructions:

1. The assignment has to be done in a group of 2 members.
2. This assignment has 2 tasks, and both tasks have internal subtasks.
3. You can use Piazza for any queries related to the assignment; avoid asking queries on the last day.

1 Introduction

This assignment focuses on enhancing process management in xv6 by implementing signal handling and a custom scheduling mechanism. In the first part, we introduce signal handling for keyboard interrupts ('SIGINT,' 'SIGBG,' 'SIGFG' and 'SIGCUSTOM') to allow process termination, suspension, resuming and custom handling. In the second part, we modify the xv6 scheduler to incorporate priority-based scheduling and profiling.

2 Signal Handling in xv6

By default, xv6 does not handle signals like `Ctrl+C` (SIGINT) and `Ctrl+Z` (SIGSTP). In this task, you must implement keyboard interrupt-based signal handling to allow process termination and suspension.

SIGINT (Signal Interrupt): Sent when the user presses Ctrl+C. This signal should be sent to all the processes in the system with $pid > 2$. The init process and the initial shell have pid as 1 and 2, respectively, so the signal should not be sent to those processes. When this signal is handled, the corresponding process should be killed.

SIGBG (Signal Background): Sent when the user presses Ctrl+B. This signal should be sent to all processes in the system with $pid > 2$. The init process and the initial shell have pid as 1 and 2, respectively, so the signal should not be sent to those processes. When this signal is handled, the corresponding process should be suspended instead of being killed. The execution of this process should be paused and it should not be scheduled. After the processes are suspended, the shell (sh, pid = 2) **should become active again** and be ready to execute commands.

SIGFG (Signal Foreground): Sent when the user presses Ctrl+F. It should be sent to all the processes that were earlier suspended using the SIGBG signal. These processes should be made runnable again.

SIGCUSTOM (Signal Custom): Sent when the user presses Ctrl+G. If the user has registered a signal handler, then that handler should be called whenever this signal is handled; otherwise, this signal should be ignored.
For registering a signal handler, you need to implement the **void signal(sighandler_t handler)** system call where:

1. **sighandler_t** is a typedef defined the following way: typedef void (*sighandler_t)(void)
2. **handler**: A pointer to the function that will handle the signal. The handler function takes no parameter and returns void.

Refer to this link for more information. Note that, we will test this signal with only one user invoked process, which means there will be 3 processes in the system (init, sh, user-process) and hence the signal should be sent to the user-process. The registered signal handler should be invoked in user space.

2.1 Implementation Overview

Whenever a keyboard event is detected, it should be printed immediately in the following format:

```
Ctrl-<key> is detected by xv6
```

where $< key >$ could be one of the following: 'C', 'B','F' or 'G' (in capitals).

Upon handling the **SIGINT** signal, the process should be terminated.

Upon handling the **SIGBG** signal, the process should be sent to the background and should no longer be scheduled.

Upon receiving **SIGFG**, the background process should be made runnable again.

Upon handling the **SIGCUSTOM** signal, the handler should be invoked if registered otherwise take no action.

2.2 Testing

We will test the signal handling for CPU intensive processes. The process is busy doing some work (calculating fibonacci series) instead of sleeping.

2.2.1 Test Case: Verifying `Ctrl+G` and `Ctrl+C` Handling

```c
#include "types.h"
#include "stat.h"
#include "user.h"


int fib(int n) {
    if (n <= 0) return 0;
    if (n == 1) return 1;
    if (n == 2) return 1;
    return fib(n - 1) + fib(n - 2);
}

void sv() {
    printf(1, "I am Shivam\n");
}

void myHandler(){
    printf(1, "I am inside the handler\n");
    sv();
}

int main() {
    signal(myHandler); // you need to implement this syscall for registering
        signal handlers
    while (1) {
        printf(1, "This is normal code running\n");
        fib(35); // doing CPU intensive work
    }
}
```

We press Ctrl+G while the process is running and since we have registered a signal handler for this particular signal, the custom handler should be invoked and after that the normal execution of the program should be continued. Finally when we press Ctrl+C the process should be suspended and the shell should become active again. We present the expected output ahead.

Expected Output

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap
    start 58
init: starting sh
$ test1
This is normal code running
This is normal code running
This is normal code running
Ctrl-G is detected by xv6
I am inside the handler
I am Shivam
This is normal code running
This is normal code running
This is normal code running
This is normal code running
Ctrl-C is detected by xv6
$
```

2.2.2 Test Case: Verifying `Ctrl+B` and `Ctrl+F` Handling

```c
#include "types.h"
#include "stat.h"
#include "user.h"

int fib(int n) {
    if (n <= 0) return 0;
    if (n == 1) return 1;
    if (n == 2) return 1;
    return fib(n - 1) + fib(n - 2);
}

int main() {
    int pid = fork();
    if (pid != 0) {
        while (1) {
            printf(1, "Hello, I am parent\n");
            fib(35);
        }
    }
    else {
        while (1) {
            printf(1, "Hi there, I am child\n");
            fib(35);
        }
    }
}
```

So here both the child and parent processes are running and printing to the shell (there might be some overlaps due to race condition, which is not an issue). When we press Ctrl+B, both these processes are suspended and the shell becomes active again. This can be verified as 'echo' and 'ls' commands are executed. The '$' may be misplaced due to buffer flushing. Then, when we press Ctrl+F, the suspended processes are made runnable again, and they start executing. Finally Ctrl+C kills them.

Expected Output

```
init: starting sh
$ test2
Hi there, I am child
Hello, I am parent
```

```
Hi there, I am child
Hello, I am parent
Hi there, I am child
Hello, I am parent
Hi there, I am child
Ctrl-B is detected by xv6
$ echo shivam
$ shivam
ls
$ .              1 1 512
..             1 1 512
README         2 2 2875
cat            2 3 15616
echo           2 4 14496
forktest       2 5 8936
grep           2 6 18460
init           2 7 15116
kill           2 8 14580
ln             2 9 14476
.
.
.
Ctrl-F is detected by xv6
Hello, I am parent
Hi there, I am child
Hello, I am parent
Hi there, I am child
Hello, I am parent
Hi there, I am child
Hello, I am parent
Hi there, I am child
Ctrl-C is detected by xv6
$
```

3 xv6 Scheduler

The current scheduler in xv6 is a round robin scheduler. The scheduler code for xv6 may be found in **proc.c** and its accompanying header file, **proc.h**.

In this assignment, you need to use xv6 with the following specifications:

1. The system shall use a single core. To enable xv6 to run on a single core, you need to **set** the **CPUS** variable to **1** in the Makefile.
2. Assume a set of n pre-emptable tasks $\mathcal{T}$, such that the tasks do not share resources, and no precedence ordering exists among the tasks.
3. By default, all processes are initialized with the default xv6 scheduling policy. But you will implement some custom scheduling algorithms in this assignment.
4. The xv6 kernel may safely terminate a process when it completely executes.

Before implementing any of the further tasks you should implement these 2 things:

- `custom_fork(start_later_flag, exec_time)` : Create a new implementation of fork to take two arguments. The first argument is a boolean flag which when set to true, should tell the scheduler to **not** immediately start scheduling the forked process. The process should only be scheduled after explicitly invoked using the `sys_scheduler_start` system call (refer to the next point). The second argument will be used to specify the execution time (number of ticks) of the forked process. Once the exec_time of the process is over, the kernel should terminate the process. If exec_time is set to -1, the process should execute indefinitely without being killed by the kernel. Note: when the

start_later flag is set to 0, and exec_time is set to -1, custom_fork should behave like the normal fork call.

- `sys_scheduler_start()` : Implement this system call, to indicate that the processes created using the custom_fork system call with start_later_flag set to True will start executing from now on (after the execution of the system call completes). This system call should do nothing if there are no custom_forked processes.

A sample file `test_sched.c` that makes use of both the calls is given as an example below :

```
#include "types.h"
#include "stat.h"
#include "user.h"

#define NUM_PROCS 3  // Number of processes to create

int main() {
    for (int i = 0; i < NUM_PROCS; i++) {
    int pid = custom_fork(1, 50); // Start later, execution time 50
    if (pid < 0) {
    printf(1, "Failed to fork process %d\n", i);
    exit();
    } else if (pid == 0) {
    // Child process
    printf(1, "Child %d (PID: %d) started but should not run yet.\n", i, getpid());
    for (volatile int j = 0; j < 100000000; j++); // Simulated work
    exit();
    }
    }

    printf(1, "All child processes created with start_later flag set.\n");
    sleep(400);

    printf(1, "Calling sys_scheduler_start() to allow execution.\n");
    scheduler_start();

    for (int i = 0; i < NUM_PROCS; i++) {
        wait();
    }

    printf(1, "All child processes completed.\n");
    exit();
}
```

Expected output for the code above should be:

```
$ test_sched
All child processes created with start_later flag set.
Calling sys_scheduler_start() to allow execution.
Child 0 (PID: 4) started but should not run yet.
Child 1 (PID: 5) started but should not run yet.
Child 2 (PID: 6) started but should not run yet.
All child processes completed.
```

3.1 Scheduler Profiler

Create a scheduler profiler that prints out all the metrics –turnaround time(TAT), waiting time(WT), response time(RT) and the number of context switches(#CS) – for each PID, after the process has finished

execution.
The format of the output should be like this:

```
$echo
PID: 3
TAT: 11
WT: 21
RT: 0
#CS: 21
```

3.2 Priority Boosting Scheduler

Modify the `xv6` scheduler to implement priority boosting:

- Every process has a **dynamic priority** that decreases as it consumes CPU time.
- If a process waits for too long, its priority is **boosted** to avoid starvation.

3.2.1 Priority Model

Each process P_i is assigned a **dynamic priority** $\pi_i(t)$ at time t, which changes based on the CPU usage and the wait time. The priority function is defined as:

$$\pi_i(t) = \pi_i(0) - \alpha \cdot C_i(t) + \beta \cdot W_i(t), \tag{1}$$

where:

- $\pi_i(0)$ is the **initial priority** assigned to all processes. This parameter should be specified in the Makefile.
- $C_i(t)$ is the **CPU ticks consumed** by process P_i up to time t ticks.
- $W_i(t)$ is the **waiting time** (the duration for which a process is waiting for the CPU core; starting from the time of its creation).
- α, β are **tunable weighting factors** that balance CPU consumption and waiting time. These must be specified using the **Makefile**. Discuss the effects of parameters on CPU-bound jobs and I/O-bound jobs.

Note: The process with the **highest priority** is selected: If multiple processes have the same priority, the one with the lowest process ID is chosen.

4 Report: 10 Marks

Page limit: 10

- For first task, explain the complete control flow from pressing the keyboard buttons (Ctrl + C/B/F/G) to the process termination, suspension, foreground or invoking custom handler.
- Discuss the effects of α and β parameters with respect to the parameters profiled in the first sub-part of the second task.

Submit a PDF file with the name *A2_report.pdf* that contains all relevant details. Also, you must **ensure** that the group members' **entry numbers** are listed on the **cover page**.

5 Submission Instructions

- We will run MOSS on the submissions. Any cheating will result in a zero in the assignment, a penalty as per the course policy and possibly much stricter penalties (including a fail grade).
- There will be NO demo for assignment 2. Your code will be evaluated using a check script (check.sh) on hidden test cases and marks will be awarded based on that. You can find the test scripts here.

How to submit:

1. Copy your report to the xv6 root directory.
2. Then, in the root directory run the following commands:

```
make clean
tar czvf \
  assignment2_easy_<entryNumber1>_<entryNumber2>.tar.gz *
```

   This will create a tarball with the name, $assignment2\_easy\_ < entryNumber1 > \_ < entryNumber2 > .tar.gz$ in the same directory that contains all the xv6 files and the PDF document. Entry number format: 2020CSZ2445 (*All English letters will be in capitals in the entry number.*). **Only one** member of the group is required to **submit** this tarball on Moodle.

3. Please note that if the report is missing in the root directory, then no marks will be awarded for the report.
4. If you are attempting the assignment as an individual, you do not need to mention the entryNumber_2 field.

Run the following commands to validate your submission:

```
sudo apt install expect
mkdir check_scripts
tar xzvf check_scripts.tar.gz -C check_scripts
cp assignment2_easy_<entryNumber1>_<entryNumber2>.tar.gz \
check_scripts
cd check_scripts
bash check.sh \
assignment2_easy_<entryNumber1>_<entryNumber2>.tar.gz
```