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,' 'SIGSTP,' 'SIGFG') to allow process termination, suspension, and resuming. In the second part, we modify the xv6 scheduler to incorporate priority-based scheduling and profiling along with the use of signals for terminating processes that have overrun their predefined execution time.

2 Signal Handling in xv6

By default, xv6 does not handle `Ctrl+C`, (SIGINT) `Ctrl+Z` (SIGSTP) and `Ctrl+F`. 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, terminating the foreground process by default.

SIGSTP (Signal Stop): Sent when the user presses Ctrl+Z, suspending the foreground process instead of terminating it. The process can be resumed with **SIGFG**.

SIGFG (Signal Foreground): Sent when the user presses Ctrl+F, it will resume the last process that was suspended with the **SIGSTP** signal.

2.1 Implementation Overview

On the high level, an OS's signal handling happens in the following way. An event of interest occurs (for example, pressing Ctrl+C, Ctrl+Z, Ctrl+F in our case) and is captured. The kernel detects this event and sends a signal to the foreground process. When the user process is next scheduled, the pending signals are checked, and the respective action is taken. Whenever a keyboard event is detected, it should be printed immediately in the following format:

```
Ctrl+<key> detected
```

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

Upon handling the **SIGSTP** 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".

2.2 Testing

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

```
#include "types.h"
#include "stat.h"
#include "user.h"
int main(void)
{
    while (1)
    {
        printf(1, "Hello");
        sleep(100);
    }
}
```

2.2.2 Expected Output

```
$ test
HelloHelloHelloHelloHello
Ctrl + Z detected

Ctrl + F detected
HelloHelloHelloHelloHello
Ctrl + C detected
$
```

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 the 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 of the forked process. Once the exec_time of the process is over, the kernel sends the SIGINT signal to terminate it.
- `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.

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, waiting time, response time and the number of context switches – for each user 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
TRT: 12
```

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 the first task, the report should clearly mention the design decisions taken while implementing the signal handler and the code snippets of essential changes made in the xv6 code. In your report, you should explain the complete control flow from pressing the keyboard buttons (Ctrl + C/Z/F) to the process termination/suspension/foreground.
- Discuss the effects of α and β parameters with respect to the parameters profiled in the first subpart.

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
```