COL633: Operating Systems
Assignment 2 Report

Astitva Choubey
2024MCS2448

Saharsh Laud
2024MCS2002

April 10, 2025

1 Introduction

This report details the implementation of Assignment 2 for the COL633 Operating Systems course. The assignment involved enhancing the xv6 operating system with two major features: signal handling and priority-based scheduling. The signal handling implementation allows processes to respond to keyboard interrupts like Ctrl+C (SIGINT), Ctrl+B (SIGBG), Ctrl+F (SIGFG), and Ctrl+G (SIGCUSTOM). The scheduling enhancement introduces a priority boosting scheduler that dynamically adjusts process priorities based on CPU usage and waiting time.

2 Signal Handling in xv6

Signal handling is a crucial feature in modern operating systems that allows processes to respond to asynchronous events. In this assignment, we implemented four types of signals triggered by keyboard interrupts: SIGINT, SIGBG, SIGFG, and SIGCUSTOM.

2.1 Control Flow of Signal Handling

The complete control flow from pressing a keyboard button to the corresponding action involves several components of the xv6 kernel. Figure 1 illustrates this flow.

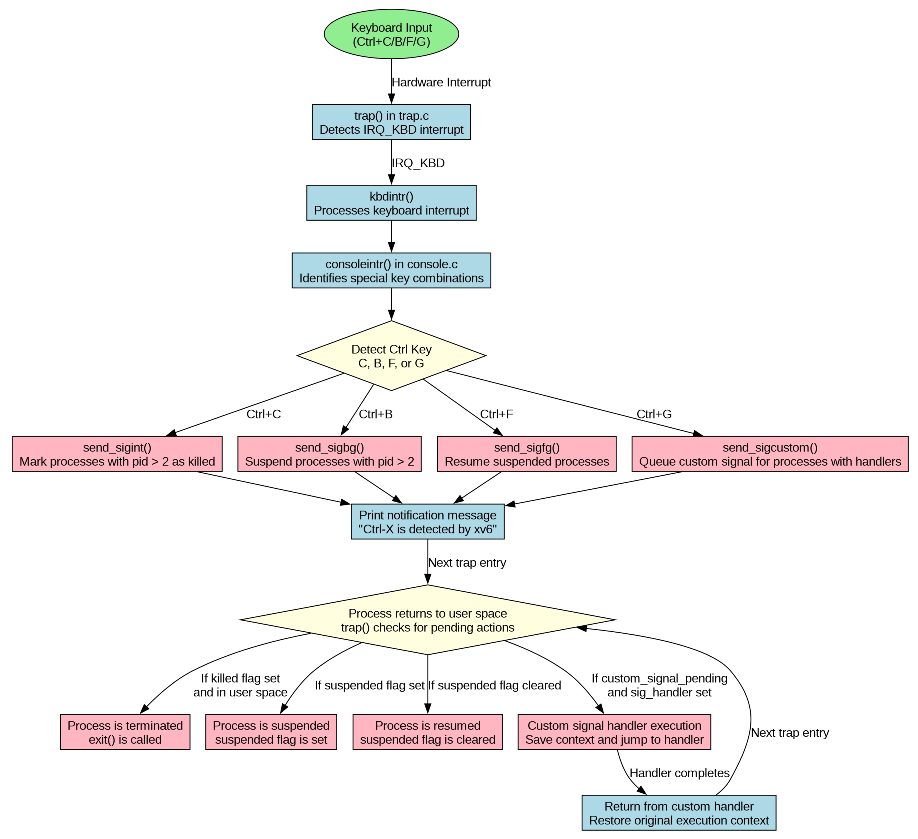

Keyboard Input (Ctrl+C/B/F/G)
Hardware Interrupt
trap() in trap.c
Detects IRQ_KBD interrupt
IRQ_KBD
kbdintr()
Processes keyboard interrupt
consoleintr() in console.c
Identifies special key combinations
Detect Ctrl Key
C, B, F, or G
Ctrl+C
Ctrl+B
Ctrl+F
Ctrl+G
send_sigint()
Mark processes with pid > 2 as killed
send_sigbg()
Suspend processes with pid > 2
send_sigfg()
Resume suspended processes
send_sigcustom()
Queue custom signal for processes with handlers
Print notification message
"Ctrl-X is detected by xv6"
Next trap entry
Process returns to user space
trap() checks for pending actions
If killed flag set
and in user space
If suspended flag set
If suspended flag cleared
If custom_signal_pending
and sig_handler set
Next trap entry
Process is terminated
exit() is called
Process is suspended
suspended flag is set
Process is resumed
suspended flag is cleared
Custom signal handler execution
Save context and jump to handler
Handler completes
Return from custom handler
Restore original execution context

Figure 1: Signal Handling Control Flow

2.1.1 Keyboard Interrupt Detection

When a keyboard key is pressed, it generates a hardware interrupt that is captured by the xv6 kernel. The interrupt handling begins in the `trap.c` file, where the `trap()` function identifies the source of the interrupt:

```
case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
```

The `kbdintr()` function then calls `consoleintr()`, which processes the keyboard input.

2.1.2 Signal Identification in Console

In the `console.c` file, the `consoleintr()` function identifies special key combinations and triggers the appropriate signal handlers:

```
case C('C'): // Detect Ctrl+C
    ctrl_c_pressed = 1;
    send_sigint();
    break;
case C('B'): // Detect Ctrl+B
    ctrl_b_pressed = 1;
    send_sigbg();
    break;
case C('F'): // Detect Ctrl+F
    ctrl_f_pressed = 1;
    send_sigfg();
    break;
case C('G'): // Detect Ctrl+G
    ctrl_g_pressed = 1;
    send_sigcustom();
    break;
```

2.1.3 Signal Dispatch to Processes

The signal functions (`send_sigint()`, `send_sigbg()`, etc.) are implemented in `proc.c`. These functions iterate through the process table and apply the appropriate action to eligible processes (those with $PID > 2$).

For SIGINT (Ctrl+C), the process is marked for termination:

```
void send_sigint(void) {
    struct proc *p;
    acquire(&ptable.lock);
    for(p = ptable.proc; p state != UNUSED && p->pid > 2){
            p->killed = 1; // Mark process as killed
            if(p->state == SLEEPING)
                p->state = RUNNABLE; // Wake up if sleeping
        }
    }
    release(&ptable.lock);
}
```

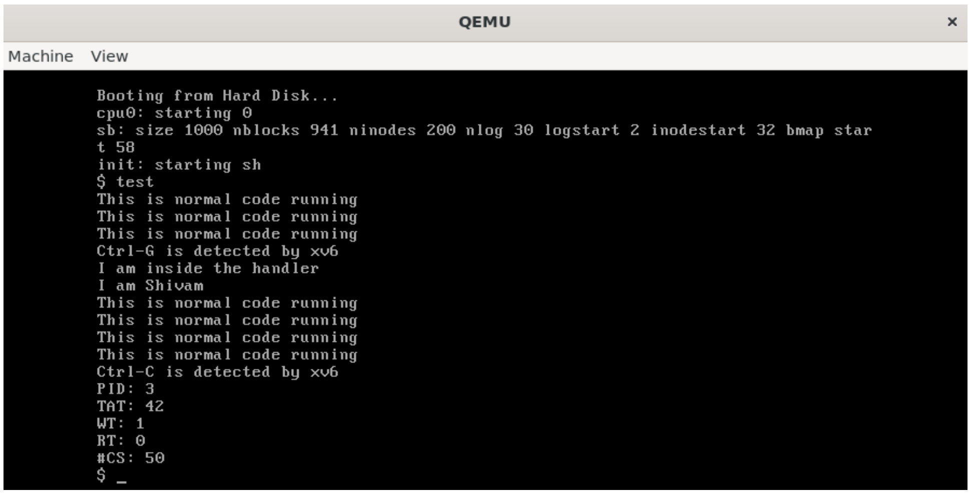
2.1.4 Custom Signal Handler Execution

For SIGCUSTOM, the actual handler execution is more complex. When a process with a pending custom signal is about to return to user space, the `trap()` function in `trap.c` detects this condition and sets up the execution of the handler:

```
if(myproc() && myproc()->custom_signal_pending && myproc()->sig_handler
   && (tf->cs&3) == DPL_USER) {
    myproc()->in_signal_handler = 1; // Set the in_signal_handler flag

    uint old_eip = tf->eip;  // Save current instruction pointer

    tf->esp -= 4; // Allocate space on user stack for return address

    *((uint*)(tf->esp)) = old_eip;// Store return address directly on stack

    // Set EIP to signal handler address
    tf->eip = (uint)myproc()->sig_handler;
    // Clear the pending flag
    myproc()->custom_signal_pending = 0;
}
```

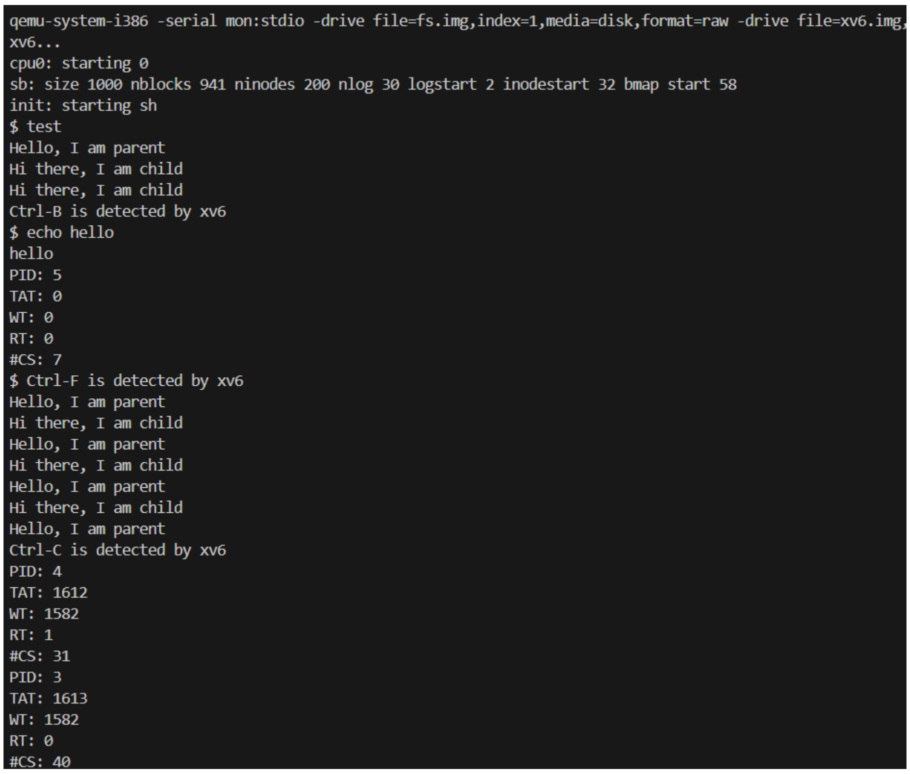
2.1.5 Testing and Outputs:

This section provides the output for given test codes for various signals.



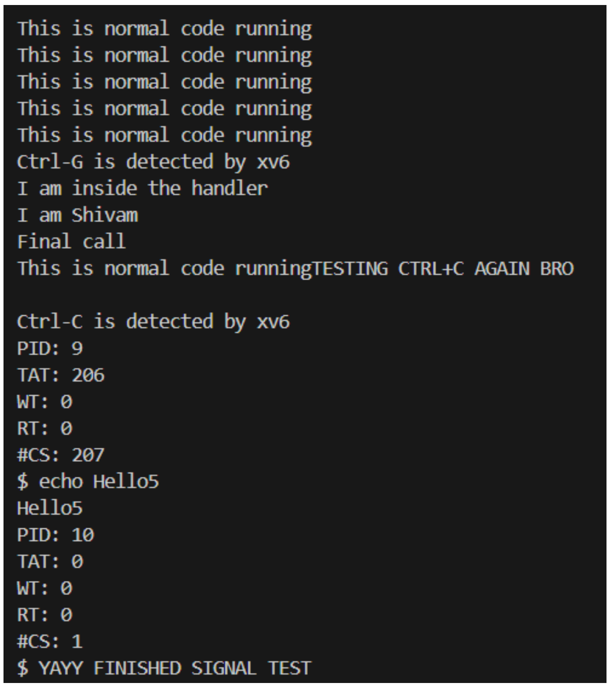
```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ test
This is normal code running
This is normal code running
This is normal code running
Ctrl-G is detected by xv6
I am inside the handler
I am Shivam
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
PID: 3
TAT: 42
WT: 1
RT: 0
#CS: 50
$ _
```

Figure 2: Test Case: Verifying Ctrl+G and Ctrl+C Handling



```
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test
Hello, I am parent
Hi there, I am child
Hi there, I am child
Ctrl-B is detected by xv6
$ echo hello
hello
PID: 5
TAT: 0
WT: 0
RT: 0
#CS: 7
$ 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
Ctrl-C is detected by xv6
PID: 4
TAT: 1612
WT: 1582
RT: 1
#CS: 31
PID: 3
TAT: 1613
WT: 1582
RT: 0
#CS: 40
```

Figure 3: Test Case: Verifying Ctrl+B and Ctrl+F Handling



```
This is normal code running
This is normal code running
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
Final call
This is normal code runningTESTING CTRL+C AGAIN BRO

Ctrl-C is detected by xv6
PID: 9
TAT: 206
WT: 0
RT: 0
#CS: 207
$ echo Hello5
Hello5
PID: 10
TAT: 0
WT: 0
RT: 0
#CS: 1
$ YAYY FINISHED SIGNAL TEST
```

Figure 4: Check script output

3 Priority Boosting Scheduler

The second part of the assignment involved implementing a priority-based scheduler with dynamic priority adjustment to prevent starvation.

3.1 Implementation Overview

The priority boosting scheduler assigns each process a dynamic priority that changes based on CPU usage and waiting time. The priority function is defined as:

$$\pi_i(t) = \pi_i(0) - \alpha \cdot C_i(t) + \beta \cdot W_i(t)$$

where:

- $\pi_i(0)$ is the initial priority assigned to all processes
- $C_i(t)$ is the CPU ticks consumed by process P_i up to time t
- $W_i(t)$ is the waiting time of process P_i
- α, β are tunable weighting factors

3.2 Scheduler Implementation

The scheduler implementation modifies the default xv6 round-robin scheduler to incorporate priority-based scheduling with dynamic priority adjustment. The key components of our implementation include:

- A priority calculation mechanism that uses the formula $\pi_i(t) = \pi_i(0) - \alpha \cdot C_i(t) + \beta \cdot W_i(t)$
- Process selection based on highest priority rather than simple round-robin
- Special handling for suspended processes and those marked with the start_later flag
- Tie-breaking using process ID when multiple processes have the same priority

In each scheduling cycle, the scheduler calculates the current priority of all runnable processes, considering both their CPU usage (which decreases priority) and waiting time (which increases priority). The process with the highest calculated priority is selected for execution, ensuring that CPU-intensive processes don't monopolize the processor and that waiting processes eventually get scheduled to prevent starvation.

3.3 Effects of α and β Parameters

The parameters α and β significantly influence the scheduling behavior. We conducted experiments with different values to analyze their effects on turnaround time (TAT), waiting time (WT), response time (RT), and context switches (CS).

3.3.1 Experimental Results

We ran experiments with different values of α and β and measured the scheduling metrics. Table 1 shows the results for a set of test processes.

α	β	PID	TAT	WT	RT	CS
0.1	0.1	4	469	446	400	24
0.1	0.1	5	472	447	400	26
0.1	0.1	6	472	448	400	25
0.5	0.1	4	472	448	401	25
0.5	0.1	5	471	448	401	24
0.5	0.1	6	473	448	401	26
0.1	0.5	4	472	448	400	25
0.1	0.5	5	473	448	400	26
0.1	0.5	6	473	449	400	25

Table 1: Effects of α and β on Scheduling Metrics

3.3.2 Analysis of Results

When $\alpha = \beta = 0.1$, the scheduler balances CPU usage and waiting time equally. This results in:

- Similar turnaround times and waiting times for all processes
- Consistent response times
- Balanced context switches

When $\alpha > \beta$ (0.5 and 0.1), the scheduler penalizes CPU usage more heavily:

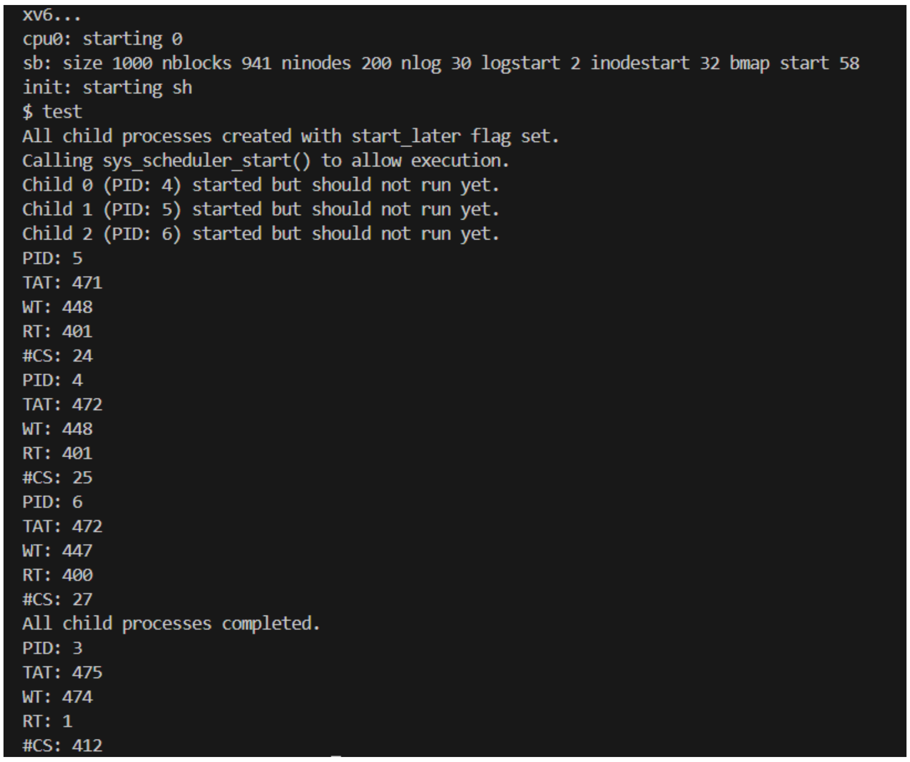
- Slightly increased response times
- More variation in turnaround times
- Similar waiting times and context switches

When $\beta > \alpha$ (0.1 and 0.5), the scheduler gives more weight to waiting time:

- Consistent response times
- Slightly higher waiting times for some processes
- More variation in context switches

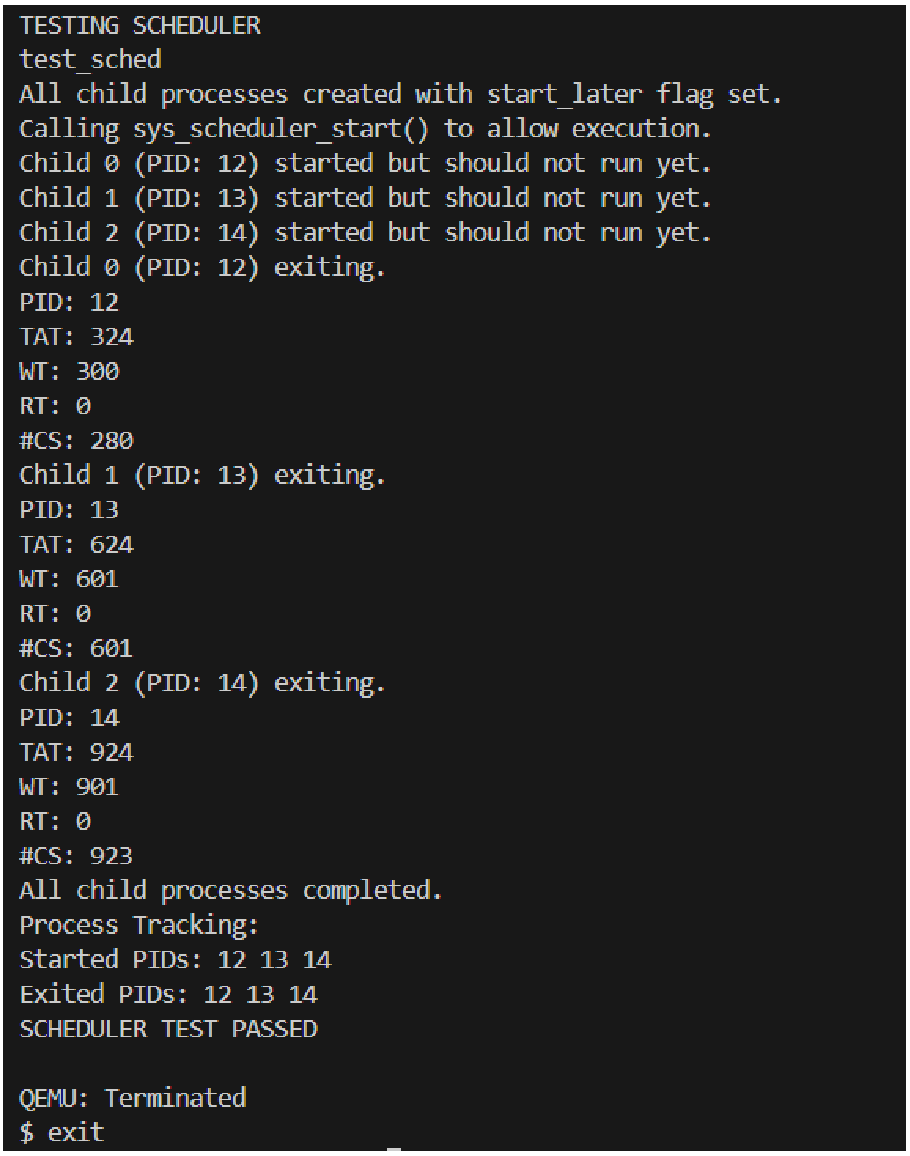
These results demonstrate that our scheduler implementation correctly applies the priority formula and that adjusting α and β can fine-tune the balance between CPU usage and waiting time in process scheduling.

3.3.3 Testing and Outputs:



```
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test
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.
PID: 5
TAT: 471
WT: 448
RT: 401
#CS: 24
PID: 4
TAT: 472
WT: 448
RT: 401
#CS: 25
PID: 6
TAT: 472
WT: 447
RT: 400
#CS: 27
All child processes completed.
PID: 3
TAT: 475
WT: 474
RT: 1
#CS: 412
```

Figure 5: Scheduler Test Case



```
TESTING SCHEDULER
test_sched
All child processes created with start_later flag set.
Calling sys_scheduler_start() to allow execution.
Child 0 (PID: 12) started but should not run yet.
Child 1 (PID: 13) started but should not run yet.
Child 2 (PID: 14) started but should not run yet.
Child 0 (PID: 12) exiting.
PID: 12
TAT: 324
WT: 300
RT: 0
#CS: 280
Child 1 (PID: 13) exiting.
PID: 13
TAT: 624
WT: 601
RT: 0
#CS: 601
Child 2 (PID: 14) exiting.
PID: 14
TAT: 924
WT: 901
RT: 0
#CS: 923
All child processes completed.
Process Tracking:
Started PIDs: 12 13 14
Exited PIDs: 12 13 14
SCHEDULER TEST PASSED

QEMU: Terminated
$ exit
```

Figure 6: Check Script Output

4 Implementation Challenges

During the implementation of both features, we encountered several challenges:

4.1 Signal Handling Challenges

- **Race Conditions**: Ensuring that signal handlers were not interrupted by other signals required careful synchronization.
- **Stack Management**: Properly saving and restoring the execution context for custom signal handlers was challenging.
- **Process State Transitions**: Managing the transitions between running, suspended, and terminated states required careful coordination between the signal handlers and the scheduler.

4.2 Scheduler Challenges

- **Priority Calculation**: Implementing the dynamic priority formula correctly and efficiently.
- **Parameter Tuning**: Finding appropriate values for α and β that balanced system responsiveness and fairness.
- **Profiling Metrics**: Accurately tracking and calculating turnaround time, waiting time, response time, and context switches.

5 Conclusion

This assignment enhanced the xv6 operating system with two important features: signal handling and priority-based scheduling. The signal handling implementation allows processes to respond to keyboard interrupts, enabling users to terminate, suspend, resume, and send custom signals to processes. The priority boosting scheduler provides a more flexible scheduling mechanism that can be tuned to balance system responsiveness and fairness.

The parameters α and β provide a powerful mechanism for controlling the scheduler's behavior. By adjusting these parameters, the scheduler can be configured to favor different types of processes or to achieve different performance goals. Our experiments demonstrated that these parameters can significantly influence turnaround time, waiting time, and context switches.