Assignment 2 Report: Enhancing Signal Handling and Custom Scheduling in xv6

Group Members
Anand Sharma 2024JCS2049
Surajprakash Narwariya 2024JCS2044

Course: Operating Systems

April 10, 2025

Contents

1 Introduction

This report describes our approach to enhancing the xv6 operating system by implementing robust signal handling and custom scheduling mechanisms. The modifications allow the system to detect keyboard interrupts corresponding to four control commands:

- **Ctrl+C (SIGINT)** – terminates processes.
- **Ctrl+B (SIGBG)** – suspends processes.
- **Ctrl+F (SIGFG)** – resumes suspended processes.
- **Ctrl+G (SIGCUSTOM)** – invokes a custom user-defined signal handler.

Additionally, our scheduler is extended with a custom fork mechanism that supports delayed process start and a dynamic priority scheduler. The dynamic priority is computed as:

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

where $C_i(t)$ is the CPU time consumed and $W_i(t)$ is the waiting time. The tunable parameters α and β balance CPU usage against waiting time, ensuring fairness and responsiveness.

The report is organized as follows:

1. Introduction and Problem Statement.
2. Design and Implementation Overview.
3. Detailed Control Flow for Each Command.
4. Dynamic Priority Scheduling and Effects of α and β.
5. Experimental Observations and Limitations.
6. Conclusion and Future Work.

2 Design and Implementation Overview

To address the assignment requirements, we modified several xv6 source files:

- **proc.h:** Extended the process structure to include new fields for signal handling (e.g., `sigint_killed`, `sigsuspended`), delayed start, execution time, profiling metrics, and dynamic priority.
- **proc.c:** Modified process creation functions (`fork` and `custom_fork`), integrated signal dispatch via the `sendsig()` function, and updated the scheduler loop to compute dynamic priorities based on CPU usage and waiting time.
- **sysproc.c:** Added system call wrappers for `custom_fork`, `scheduler_start`, and `signal` (for user registration of custom signal handlers).
- **trap.c:** Updated the trap handler to check for signals and redirect execution to a custom handler when required.
- **console.c:** Enhanced the keyboard interrupt handler (`consoleintr()`) to detect Ctrl+C, Ctrl+B, Ctrl+F, and Ctrl+G events, print appropriate messages, and invoke `sendsig()` with the corresponding signal.

3 Detailed Control Flow for Keyboard Commands

This section describes the control flow for each of the four keyboard commands.

3.1 Ctrl+C (SIGINT) – Process Termination

1. **Keyboard Event Detection:**
   - In `console.c`, the function `consoleintr()` reads characters from the input device.
   - When the user presses Ctrl+C, the macro `C('C')` is detected.
   - The variable `ctrlc_requested` is set to SIGINT and `print_ctrlc_flag` is raised; a message such as "Ctrl-C is detected by xv6" is printed.
2. **Signal Dispatch:**
   - In `trap.c`, during trap handling (e.g., on a timer interrupt), the flag `ctrlc_requested` is checked.
   - If set, `sendsig(SIGINT)` is invoked (implemented in `proc.c`), which iterates over the process table and sets the `sigint_killed` flag for eligible processes (ignoring processes with pid 1 and 2).
3. **Process Termination:**
   - In the scheduler loop (in `proc.c`), a process with the `sigint_killed` flag is marked for termination.
   - When scheduled, the process transitions to the exit routine, where profiling metrics (turnaround time, waiting time, response time, and context switches) are computed before cleanup.
4. **Outcome:** Ctrl+C triggers SIGINT, leading to the termination of processes.

3.2 Ctrl+B (SIGBG) – Process Suspension

1. **Keyboard Event Detection:**
   - In `console.c`, `consoleintr()` detects the Ctrl+B combination via `C('B')`.
   - A message such as "Ctrl-B detected" is printed.
2. **Signal Dispatch:**
   - The function `sendsig(3)` is invoked to dispatch the SIGBG signal.
   - Within `sendsig()`, the kernel iterates over the process table and sets the `sigsuspended` flag for each eligible process (pid > 2).
3. **Process Suspension:**
   - In the scheduler loop, processes marked as suspended (i.e., with `sigsuspended` set) are skipped and not scheduled for execution.
   - This effectively suspends the processes until a resume signal is received.
4. **Outcome:** Ctrl+B results in SIGBG being sent, and the affected processes are suspended.

3.3 Ctrl+F (SIGFG) – Process Resumption

1. **Keyboard Event Detection:**
   - In `console.c`, `consoleintr()` detects Ctrl+F (via `C('F')`).
   - A message like "Ctrl-F detected" is printed.
2. **Signal Dispatch:**
   - The function `sendsig(4)` is invoked, dispatching SIGFG.
   - Within `sendsig()`, the kernel iterates through the process table and clears the `sigsuspended` flag for any suspended process.
3. **Process Resumption:**
   - Once the `sigsuspended` flag is cleared, the processes become eligible for scheduling again.
   - The scheduler then selects these processes in subsequent cycles, and they resume normal execution.
4. **Outcome:** Ctrl+F effectively resumes the suspended processes by clearing their suspension flags.

3.4 Ctrl+G (SIGCUSTOM) – Custom Signal Handler Invocation

1. **Keyboard Event Detection:**
   - In `console.c`, when Ctrl+G (via `C('G')`) is pressed, a message such as "CTRL + G pressed (SIGCUSTOM)" is printed.
2. **Marking for Custom Handling:**
   - The kernel iterates over the process table (acquiring `ptable.lock`) and, for each process with a registered custom handler (i.e., where `custom_handler_function` is not NULL), the flag `pending_custom_handler` is set.
3. **Trap Handling and Custom Handler Invocation:**
   - In `trap.c`, during trap handling, if the current process has its `pending_custom_handler` flag set, the trap frame's instruction pointer (`eip`) is modified to point to the address stored in `custom_handler_function`.
   - The custom signal handler is then executed in user space.
4. **Outcome:** Ctrl+G delivers SIGCUSTOM to processes with registered handlers, invoking the user-defined function.

4 Dynamic Priority Scheduling and Effects of α and β

Our custom scheduler computes the dynamic priority of each process using:

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

where:

- $\pi_i(0)$ is the initial priority.
- $C_i(t)$ is the CPU ticks consumed (tracked by `ticks_run`).
- $W_i(t)$ is the waiting time, computed as the difference between the current tick count and the sum of the process's creation and run times.

4.1 Role of α (CPU Usage Penalty)

- **Concept:** α penalizes processes based on CPU consumption. A higher α value reduces the dynamic priority more steeply as a process accumulates CPU ticks.
- **Effects on CPU-bound Processes:**
  - CPU-bound processes that consume many ticks see their dynamic priority drop significantly.
  - This reduction prevents such processes from monopolizing the CPU, forcing them to yield more often.
- **Profiling Metrics:**
  - **Turnaround Time (TAT):** May be reduced if processes are preempted early.
  - **Waiting Time (WT):** May increase due to frequent preemptions.
  - **Context Switches (#CS):** A high α can lead to increased context switches.

4.2 Role of β (Waiting Time Boost)

- **Concept:** β boosts the dynamic priority of processes based on their waiting time. A higher β value increases the dynamic priority for processes that have waited longer in the ready queue.
- **Effects on I/O-bound and Interactive Processes:**
  - These processes typically experience longer wait times.
  - A higher β ensures they are scheduled more promptly, reducing response time.
- **Profiling Metrics:**
  - **Response Time (RT):** Improved, as waiting processes are given CPU time faster.
  - **Waiting Time (WT):** Generally reduced for processes that are boosted by waiting.

4.3 Tuning Trade-offs and Profiling Implications

Balancing α and β is critical:

- A high α relative to β may excessively penalize CPU-bound processes, leading to frequent context switches and increased waiting times.
- Conversely, a high β may boost waiting processes too much, potentially starving CPU-bound processes.
- Our profiling metrics (TAT, WT, RT, #CS) provide feedback for fine-tuning these parameters, allowing us to optimize the balance between throughput and responsiveness.

5 Experimental Observations and Limitations

5.1 Experimental Observations

During our testing, we observed the following:

- **Interactive Responsiveness:** Processes that were I/O-bound or interactive benefited from a higher β, as they received prompt CPU allocation and showed improved response times.
- **CPU-bound Behavior:** When α was set higher, CPU-bound processes experienced more frequent preemptions, which improved overall system fairness but sometimes resulted in increased context switching overhead.
- **Profiling Feedback:** The profiling outputs provided clear metrics (turnaround time, waiting time, response time, and context switches) that guided our tuning of α and β. This allowed us to strike a balance where neither interactive nor CPU-bound tasks were starved.

5.2 Limitations

Despite our improvements, several limitations were identified:

- **Static Parameter Tuning:** The values for α and β are set at compile time, which may not adapt well to rapidly changing workloads.
- **Overhead of Context Switching:** High preemption rates, while improving fairness, can incur significant context switching overhead, which may degrade performance in some scenarios.
- **Signal Handling Robustness:** Our current implementation focuses on specific signals; additional signal types and error handling may be required for a more robust system.

6 Conclusion

In this report, we presented our enhancements to the xv6 operating system by integrating robust signal handling and custom scheduling. We detailed the control flows for the following keyboard commands:

- **Ctrl+C (SIGINT):** Triggers process termination and cleanup, with profiling metrics reported upon exit.
- **Ctrl+B (SIGBG):** Suspends processes by setting a suspension flag.
- **Ctrl+F (SIGFG):** Resumes suspended processes by clearing the suspension flag.
- **Ctrl+G (SIGCUSTOM):** Invokes a custom user-defined signal handler for processes with registered handlers.

Furthermore, our custom scheduler employs a dynamic priority function that uses the tunable parameters α and β to balance CPU consumption and waiting time. Profiling metrics such as turnaround time, waiting time, response time, and context switches provide essential feedback for tuning these parameters.