COL633 - Operating Systems Assignment 2 – Easy

Enhancing Process Management in xv6

Group Members:

2024MCS2459-Vaibhav Katendra

2024MCS2445-Adithya R. Narayan

Course: Operating Systems
April 10, 2025

Chapter 1

Introduction

This report provides a comprehensive explanation of the modifications and enhancements introduced into the xv6 operating system as part of Assignment 2 – Easy. The assignment consists of two main tasks:

1. **Signal Handling:** Implement custom signal handling that maps specific keyboard inputs (Ctrl+C, Ctrl+B, Ctrl+F, Ctrl+G) to corresponding actions—process termination, suspension, resumption, or invoking a user-defined signal handler.

2. **xv6 Scheduler Enhancements:** Introduce a custom fork implementation with the ability to defer process scheduling and limit execution time. This is paired with an enhanced scheduler that provides performance metrics (turnaround time, waiting time, response time, and number of context switches) and a dynamic priority boosting mechanism governed by parameters α and β.

The report focuses on two crucial aspects:

- A detailed, step-by-step control flow analysis for each signal handling event.
- An in-depth discussion on how the tuning parameters α and β affect the scheduler's profiled metrics.

Chapter 2

Detailed Control Flow in Signal Handling

This section explains the complete journey from when the user presses one of the designated keyboard keys until the corresponding process action is completed. Each signal is handled with its own specific logic in the xv6 kernel.

2.1 Ctrl+C (SIGINT) – Process Termination

2.1.1 Key Detection

- The keyboard driver in xv6 is modified to detect the key combination `Ctrl+C`.
- When the keyboard interrupt occurs, the driver identifies the ASCII value 3 (ETX) as Ctrl+C.
- Immediately after detection, the system prints:

```
Ctrl-C is detected by xv6
```

2.1.2 Kernel Level Handling

- The interrupt service routine (ISR) receives the Ctrl+C event.
- The handler determines that the event corresponds to SIGINT.
- The kernel traverses the process table to select processes with `pid` $>$ `2` (excluding the `init` and `shell` processes).
- For each eligible process, the kernel sets the `killed` flag to 1.

2.1.3 Process Action

- When the killed process next enters the scheduler, it checks the killed flag.
- The process then enters the exit routine, where resources are released.
- Before termination, the process metrics are calculated and displayed:
  - PID: Process ID
  - TAT: Turnaround Time (end_time - creation_time)
  - WT: Waiting Time (time spent in RUNNABLE state)
  - RT: Response Time (first_scheduled_time - creation_time)
  - #CS: Number of Context Switches
- Control returns to the shell, and the termination is reflected on the console.

2.2 Ctrl+B (SIGBG) – Process Suspension

2.2.1 Key Detection

- The driver recognizes the `Ctrl+B` keystroke (ASCII value 2).
- An immediate console message is printed:

```
Ctrl-B is detected by xv6
```

2.2.2 Kernel Level Handling

- The ISR processes the event and categorizes it as SIGBG.
- The kernel identifies all processes (with `pid` $>$ `2`) and flags them for suspension.
- For each eligible process, the `suspended` flag is set to 1.

2.2.3 Process Action

- Processes with the suspended flag set continue to exist in the process table.
- However, the scheduler skips these processes during scheduling decisions.
- The suspended processes do not accumulate waiting time or total ticks while suspended.
- Control is explicitly returned to the shell, allowing the user to enter new commands.

2.3 Ctrl+F (SIGFG) – Process Resumption (Foregrounding)

2.3.1 Key Detection

- When the user presses `Ctrl+F`, the keyboard driver detects ASCII value 6.
- A message is printed:

```
Ctrl-F is detected by xv6
```

2.3.2 Kernel Level Handling

- The ISR discerns the signal as SIGFG.
- The kernel locates all processes that have the `suspended` flag set to 1.
- For each suspended process, the kernel resets the `suspended` flag to 0.

2.3.3 Process Action

- With the suspended flag cleared, processes become eligible for scheduling again.
- The scheduler resumes considering these processes during its scheduling decisions.
- Resumed processes continue execution from the point they were suspended.
- Metrics tracking (waiting time, total ticks) resumes for these processes.

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

2.4.1 Key Detection

- The detection routine intercepts the `Ctrl+G` key combination (ASCII value 7).
- The console immediately displays:

```
Ctrl-G is detected by xv6
```

2.4.2 Kernel Level Handling

- The ISR interprets this event as SIGCUSTOM.
- The kernel checks if the foreground process has registered a signal handler via the `signal()` system call.
- If a handler is registered, the kernel prepares to transfer control to this handler.

2.4.3 Process Action

- If a custom handler is present, the kernel saves the current execution context.
- The process's user stack is modified to include a return address that will restore normal execution after the handler completes.
- Control is transferred to the user-defined handler function.
- The handler executes in user space, potentially calling other functions (as seen in the test case where it calls `sv()` which then calls `more()`).
- Once the handler chain completes, control returns to the normal flow of the process.
- If no custom handler is registered, the process continues normal execution.

2.5 Implementation Details

2.5.1 Key Data Structures

- The `proc` structure was extended to include:
  - `suspended`: Flag indicating if the process is suspended
  - `sighandler`: Pointer to the custom signal handler function
  - `waiting_time`: Accumulated time in RUNNABLE state
  - `total_ticks`: Total CPU time consumed
  - `creation_time`: Time when the process was created
  - `end_time`: Time when the process exited
  - `first_scheduled`: Flag indicating if the process has been scheduled
  - `response_time`: Time from creation to first scheduling
  - `context_switches`: Number of context switches

2.5.2 Key Functions

- `consoleintr()`: Modified to detect special key combinations
- `signal()`: System call to register a custom signal handler
- `custom_fork()`: Enhanced fork with start_later flag and exec_time
- `scheduler_start()`: System call to activate deferred processes
- `scheduler()`: Modified to track metrics and implement priority boosting

2.6 Summary of Signal Handling Flow

The complete signal handling mechanism involves:

1. Immediate detection and console feedback on the key press.
2. Kernel-level signal interpretation and process table scanning.
3. Contextual actions: termination (SIGINT), suspension (SIGBG), resumption (SIGFG), or custom handling (SIGCUSTOM).

4. Metrics collection and reporting for process performance analysis.

This design guarantees a responsive and manageable process control system that seamlessly connects low-level hardware events to high-level process management.

Chapter 3

xv6 Scheduler Enhancements and Parameter Effects

3.1 Overview

The second major task involves enhancing the scheduler by introducing:

- A custom fork system call that supports deferred scheduling (via a `start_later_flag`) and enforces execution time limits using an `exec_time` parameter.
- A `scheduler_start` system call that activates all processes created with the deferred scheduling flag.
- An integrated scheduler profiler that outputs important metrics for each process after termination:
  - Turnaround Time (TAT): end_time - creation_time
  - Waiting Time (WT): Time spent in RUNNABLE state
  - Response Time (RT): first_scheduled_time - creation_time
  - Number of Context Switches (#CS): Count of preemptions
- A dynamic priority boosting mechanism to ensure fairness.

3.2 Implementation of Custom Fork and Scheduler Start

3.2.1 Custom Fork

The `custom_fork(int start_later, int exec_time)` system call extends the standard fork functionality:

- It creates a new process just like the regular fork.
- If `start_later` is set to 1, the process is created but not made RUNNABLE until `scheduler_start()` is called.
- The `exec_time` parameter sets a limit on how long the process can execute before being terminated.
- The new process inherits all properties from its parent, including file descriptors and memory layout.

3.2.2 Scheduler Start

The `scheduler_start()` system call activates all processes created with the `start_later` flag:

- It traverses the process table to find all processes with `start_later` set to 1.

- For each such process, it changes the state to RUNNABLE, making them eligible for scheduling.
- This allows for synchronized starting of multiple processes, which is useful for benchmarking and testing.

3.3 Metrics Tracking

3.3.1 Turnaround Time (TAT)

- Definition: The total time from process creation to completion.
- Implementation: `TAT = end_time - creation_time`
- When a process exits, its end_time is recorded, and TAT is calculated and displayed.

3.3.2 Waiting Time (WT)

- Definition: The time a process spends in the RUNNABLE state but not actually running.
- Implementation: The waiting_time counter is incremented in the scheduler for each process that is RUNNABLE and not suspended.
- This metric is crucial for evaluating scheduler fairness.

3.3.3 Response Time (RT)

- Definition: The time from process creation until it is first scheduled.
- Implementation: When a process is scheduled for the first time, its response_time is calculated as `RT = current_time - creation_time`.
- This metric is important for interactive processes where quick initial response is desirable.

3.3.4 Context Switches (#CS)

- Definition: The number of times a process is preempted (switched out) by the scheduler.
- Implementation: When the scheduler switches from one process to another, the context_switches counter of the previous process is incremented.
- This metric helps evaluate scheduler overhead and efficiency.

3.4 Dynamic Priority Boosting Model

In the enhanced scheduler, each process P_i has a dynamic priority computed as:

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

where:

- $\pi_i(0)$: The initial priority value assigned to the process.
- $C_i(t)$: Cumulative CPU ticks consumed by the process until time t.
- $W_i(t)$: Total waiting time—the time the process spends in the ready queue.
- α: Weight factor that penalizes CPU usage.
- β: Weight factor that rewards waiting time.

3.5 Effects of α and β on Scheduler Metrics

The tuning of α and β is critical to achieving a balance between resource utilization and process fairness.

3.5.1 Turnaround Time (TAT) and Waiting Time (WT)

- A high α value increases the penalty on CPU-bound processes:
  - **Effect:** As CPU-bound processes quickly accumulate CPU ticks, their priority drops significantly. This may lead to increased waiting time as they are frequently preempted in favor of other processes.
  - **Resulting TAT/WT:** Elevated waiting times can contribute to a longer turnaround time for CPU-bound tasks.
- A high β value rewards processes that accumulate waiting time:
  - **Effect:** Processes that remain in the ready queue are boosted, reducing their effective waiting time.
  - **Resulting TAT/WT:** This mechanism lowers waiting time, thereby potentially decreasing turnaround time.

3.5.2 Response Time (RT)

- An optimal α ensures that a process does not lose too much priority immediately after arrival:
  - **Effect:** If α is set too high, interactive or I/O-bound processes might be delayed even if they have just arrived. A moderate α allows new processes to get scheduled swiftly.
  - **Resulting RT:** Lower response times when new processes are not heavily penalized.
- A correctly tuned β further helps by boosting waiting processes:
  - **Effect:** Processes waiting in the run queue receive a higher chance of being scheduled, improving responsiveness.
  - **Resulting RT:** Reduced response times for processes that might otherwise suffer from prolonged scheduling delays.

3.5.3 Number of Context Switches (#CS)

- A high α may lead to aggressive preemption:
  - **Effect:** CPU-bound processes are preempted more frequently, increasing the total number of context switches.
  - **Resulting #CS:** Excessive context switching may introduce overhead and can degrade overall system performance.
- A high β provides a counterbalance:
  - **Effect:** By boosting waiting processes, β helps reduce rapid preemptions, leading to a more stable scheduling environment.
  - **Resulting #CS:** A balanced β contributes to maintaining a moderate level of context switches.

3.5.4 Experimental Results

Our test results from the scheduler test demonstrate the effects of our chosen α and β values:

- For Child 0 (PID 12): TAT = 329, WT = 301, RT = 1, #CS = 285
- For Child 1 (PID 13): TAT = 630, WT = 601, RT = 1, #CS = 604
- For Child 2 (PID 14): TAT = 931, WT = 902, RT =
- For Child 2 (PID 14): TAT = 931, WT = 902, RT = 1, #CS = 605

These results demonstrate the effects of our chosen α and β values:

- All processes have a very low Response Time (RT = 1), indicating that our scheduler quickly attends to new processes.
- The Waiting Time (WT) increases significantly for each subsequent child process, showing how CPU-bound processes accumulate waiting time under our priority scheme.
- The number of Context Switches (#CS) is relatively high, especially for the later processes, indicating frequent preemption.
- Turnaround Time (TAT) increases for each child, reflecting both the increased waiting time and the cumulative effect of the workload.

3.6 Balancing α and β

Finding the right balance between α and β is crucial for optimal scheduler performance:

- If α is too high relative to β:
  - CPU-bound processes may starve, leading to increased waiting times and turnaround times.
  - The system may become overly responsive to short-burst, I/O-bound processes at the expense of throughput for CPU-intensive tasks.
- If β is too high relative to α:
  - Long-waiting processes might monopolize the CPU once scheduled, potentially delaying newly arrived, critical processes.
  - The system may not adequately penalize CPU-hogging processes, leading to unfair resource distribution.

3.7 Recommendations for Tuning

Based on our observations and experimental results, we recommend:

- Start with moderate values for both α and β, such as $\alpha = 0.75$ and $\beta = 0.25$.
- Monitor the scheduler metrics (TAT, WT, RT, #CS) under various workloads.
- If long-running, CPU-bound processes are showing excessive waiting times, consider decreasing α slightly.
- If interactive or I/O-bound processes are experiencing high response times, consider increasing β.
- Aim for a balance where no single metric (TAT, WT, RT, or #CS) shows extreme values across different process types.