COL331 A2: Signal Handling and Scheduling in xv6

Yash Bansal (2022CS51133)
Nikhil Zawar (2022CS11106)

Contents

1 Introduction

In this assignment, we have implemented keyboard-interrupt-based signal handling and a custom scheduling mechanism in the xv6 operating system. The signal handling feature enables the kernel to respond to specific key combinations such as CTRL+C, CTRL+B, CTRL+F, and CTRL+G, allowing the user to send various signals to processes. These signals correspond to interrupting, backgrounding, foregrounding, and invoking a user-defined handler in a process, respectively.

In addition to signal handling, we have implemented a priority-based scheduler, which alters the way xv6 schedules processes, enabling more control over process execution based on assigned priorities.

2 Signal Handling

Signal handling in xv6 is extended by adding support for:

- **CTRL+C** (Interrupt Signal)
- **CTRL+B** (Background Signal)
- **CTRL+F** (Foreground Signal)
- **CTRL+G** (User-defined Signal Handler)

xv6 already has a built-in signal (CTRL+P) for dumping the process table. We use similar mechanisms to implement our additional signals.

2.1 Detecting Keyboard Interrupts

To detect key combinations, we modify the `consoleintr()` function in `console.c`. Boolean flags are introduced to detect specific key sequences (e.g., CTRL+B, CTRL+F, etc.). When a key combination is detected, the corresponding signal-handling function (defined in `proc.c`) is invoked. These flags are reset after each loop iteration to ensure they don't persist unintentionally.

2.2 CTRL+C — Interrupt Signal

CTRL+C sends an interrupt signal to all processes except the `init` process (PID 1) and the shell process (PID 2), terminating them.

- We define a function `sigint()` in `proc.c`, called from `console.c` upon detecting CTRL+C.
- `sigint()` loops through the process table (`ptable`). For each valid (used) process with PID > 2, it invokes `kill(pid)` to terminate the process.
- The processes are terminated the next time they are scheduled.

2.3 CTRL+B — Background Signal

CTRL+B suspends all user processes (PID > 2), effectively placing them in the background. These processes will not be scheduled again until a foreground signal is issued.

- We define a function `sigbg()` in `proc.c`, called when CTRL+B is detected.
- A new flag `in_bgd` is added to the `proc` structure to indicate if a process is in the background.

- `sigbg()` loops through all processes. For each valid process with PID > 2, it sets `in_bgd = 1`.
- In the `scheduler()`, we skip processes whose `in_bgd` flag is set.
- The shell process (PID 2) is explicitly woken up using `wakeup()` so it remains runnable.
- In `wait()`, background child processes are ignored to ensure the shell can proceed. Zombie processes that couldn't be cleaned in `wait()` are cleared in the scheduler.
- Before killing any background process, we clear its `in_bgd` flag so it becomes schedulable, allowing `kill()` to function correctly.

2.4 CTRL+F — Foreground Signal

CTRL+F brings all background processes back to the foreground, allowing them to resume execution.

- We define `sigfg()`, which resets the `in_bgd` flag for all valid processes.
- These processes can now be scheduled normally.

2.5 CTRL+G — User-defined Signal Handler

CTRL+G allows a process to register and invoke a user-defined signal handler from user space.

- The `proc` structure is extended with the following fields:
  - `handler_exists` — whether a handler is registered.
  - `handler` — function pointer to the user-defined signal handler.
  - `signal_pending` — indicates a pending signal.
  - `within_handler` — whether the process is currently executing the handler.
- A new system call `signal(handler_function)` is added for registering the handler. In the kernel, `sys_signal()` sets the handler and marks `handler_exists = 1`.
- On CTRL+G detection, `syscustom()` is invoked. If a handler is registered, it sets `signal_pending = 1`.
- In `trap.c`, we check if:
  - The signal is pending.
  - The handler is not already running.
  - The process is currently in user mode.
- If all conditions hold:
  - Save trap frame and essential registers in the process struct.
  - Push the return address to the user stack.
  - Redirect the instruction pointer to the registered handler function.
  - Set `within_handler = 1`.
- When the handler finishes (identified via a page fault), we:
  - Set `within_handler = 0`.
  - Restore the saved CPU registers and trap frame.
  - Resume the original execution flow of the process.

3 Scheduling

The implementation of a modified scheduler, first requires the implementation of two system calls. After that we implement the priority-boosted scheduling mechanism. To help in debugging and bookkeeping, we implement the scheduler profiler, which displays the various attributes of any given process.

3.1 Custom Fork Implementation

The custom fork implementation extends the traditional fork system call with two additional parameters:

- `start_later`: Boolean flag to delay process scheduling
- `exec_time`: Maximum execution time in ticks (-1 for unlimited)

3.1.1 Key Data Structures

Added to `struct proc` in `proc.h`:

```
int start_later;        // Flag to delay scheduling
int exec_time;          // Maximum execution time
int ticks_used;         // Ticks consumed so far
```

3.1.2 Implementation Details

The `custom_fork` function in `proc.c`:

```
int custom_fork(int start_later, int exec_time) {
    // ... allocation and setup ...
    np->start_later = start_later;
    np->exec_time = exec_time;
    np->ticks_used = 0;
    // ... rest of fork implementation ...
}
```

3.2 Scheduler Start Implementation

The `scheduler_start()` system call provides explicit control over process activation, enabling:

- Batch process initialization
- Synchronized process starts
- Resource pre-allocation before execution

3.2.1 Algorithm

```
int scheduler_start(void) {
    struct proc *p;
    int found = 0;

    acquire(&ptable.lock);
```

```
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->state == UNUSED && p->start_later) {
            p->state = RUNNABLE;
            p->start_later = 0;
            found = 1;
        }
    }
    release(&ptable.lock);

    return found ? 0 : -1;
}
```

All existing user programs continue functioning unchanged.

3.2.2 Scheduler Start System Call

The companion `scheduler_start` system call activates delayed processes:

```
int scheduler_start(void) {
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->state == UNUSED && p->start_later) {
            p->state = RUNNABLE;
            found = 1;
        }
    }
    return found ? 0 : -1;
}
```

3.3 Scheduler Profiler

3.3.1 Metrics Collected

- Turnaround Time (TAT): Creation to termination
- Waiting Time (WT): Time in ready queue
- Response Time (RT): Creation to first run
- Context Switches (#CS)

3.3.2 Implementation

Added to `struct proc`:

```
int creation_time;
int start_time;
int termination_time;
int context_switches;
int has_started;
```

Automatic metric collection is implemented in scheduler.

4 Priority Boosting Scheduler

The priority scheduler implements:

- Dynamic priority calculation based on CPU usage and wait time
- Priority boosting for starved processes
- Configurable parameters via Makefile

4.1 Priority Calculation

The priority formula implemented:

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

With threshold-based boosting:

```
if(p->waiting_ticks > WAIT_THRESHOLD) {
    p->dynamic_priority = MAX_PRIORITY;
}
```

4.2 Scheduler Modifications

Key changes to the scheduler loop:

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->state != RUNNABLE) continue;

    // Update waiting time
    p->waiting_ticks += ticks - p->last_run_time;
    update_priority(p);

    // Select highest priority process
    if(p->dynamic_priority > highest->dynamic_priority) {
        highest = p;
    }
}
```

5 Scheduler Function Implementation

The modified scheduler implements several critical features:

- **Priority-based selection**: Chooses processes based on dynamic priority calculations
- **Process state management**: Handles RUNNABLE, ZOMBIE, and background processes
- **Time tracking**: Maintains accurate CPU and waiting time measurements
- **Context switch accounting**: Tracks each context switch occurrence

5.1 Core Algorithm

The scheduler follows this workflow:

1. Enable interrupts on the current processor
2. Acquire process table lock
3. Scan process table for:
    - Zombie process cleanup
    - Background process skipping
    - Start-later process exclusion
4. Update process metrics:
    - First-run time tracking
    - Waiting time accumulation
    - Priority recalculation
5. Select highest priority process (with PID tiebreaker)
6. Execute selected process
7. Update runtime statistics
8. Release process table lock

5.2 Key Implementation Details

5.2.1 Priority Selection Logic

```
if(!highest_pri_proc ||
   (p->dynamic_priority > highest_pri_proc->dynamic_priority) ||
   (p->dynamic_priority == highest_pri_proc->dynamic_priority &&
    p->pid < highest_pri_proc->pid)) {
    highest_pri_proc = p;
}
```

5.2.2 Process Execution Block

```
if(highest_pri_proc) {
    highest_pri_proc->context_switches++;
    c->proc = highest_pri_proc;
    switchuvm(highest_pri_proc);
    highest_pri_proc->state = RUNNING;
    // ... context switching ...
    highest_pri_proc->context_switches++;
}
```

The scheduler maintains several performance metrics:

Metric	Implementation
Response Time	Tracks `start_time` on first run
Waiting Time	Accumulates `waiting_ticks`
CPU Time	Updates `cpu_ticks` after execution
Context Switches	Increments `context_switches` per switch

5.3 Performance Considerations

The implementation addresses several performance aspects:

- **Locking**: Uses `ptable.lock` to protect process table access
- **Efficiency**: Single pass through process table per scheduling decision
- **Fairness**: Priority boosting prevents starvation
- **Overhead**: Minimal additional calculations during scheduling

6 Effects of α and β on CPU-bound and I/O-bound Jobs

The parameters α and β govern how CPU usage and wait time influence process priorities in the scheduling system:

- α: This parameter represents the weight given to CPU usage. A higher value of α penalizes processes that consume more CPU time, thereby reducing their priority.
- β: This parameter represents the weight given to wait time. A higher value of β rewards processes that have been waiting longer, increasing their priority.

For testing, we created six custom forked processes: three CPU-bound processes (performing a simple addition loop) and three I/O-bound processes (printing to the console in a loop). All processes were created with a delayed start flag, and were initiated simultaneously. We measured the total time for completion and the waiting time for each process to analyze the effects of varying α and β values. The following observations were made.

6.1 Observations

- As α increases, the time taken by CPU-bound jobs increases, while the time taken by I/O-bound jobs decreases. However, the overall time taken by all jobs increases as α grows.
- As β increases, the time taken by I/O-bound jobs decreases, while the time taken by CPU-bound jobs continues to increase. However, increasing β beyond a certain point does not significantly affect the jobs, as the priority is capped at a maximum value.

6.2 CPU-bound Jobs

CPU-bound jobs are characterized by heavy CPU usage and minimal wait time for I/O operations. The effects of α and β on CPU-bound jobs are as follows:

- A large α significantly reduces the priority of CPU-bound processes, as they consume more CPU time.
- The effect of β is less pronounced for CPU-bound jobs, since these processes spend little time waiting for I/O and are less impacted by wait time.

6.3 I/O-bound Jobs

I/O-bound jobs are characterized by significant wait times for I/O operations. The effects of α and β on I/O-bound jobs are as follows:

- A small α has a limited impact on I/O-bound jobs, as these processes do not consume much CPU time. However, a large α can still penalize them when they use the CPU.
- A large β increases the priority of I/O-bound jobs, especially those that have been waiting for CPU time. This ensures these jobs are scheduled promptly once they are ready after I/O operations.

7 Testing Methodology

Test Cases

- Mixed workload of CPU-bound and I/O-bound processes
- Processes with varying execution time limits
- Priority inversion scenarios
- Stress tests with many concurrent processes

Verification

- Confirmed priority boosting prevents starvation
- Verified execution time limits are enforced
- Validated profiler metrics against manual calculations

8 Conclusion

This implementation enhances xv6 by integrating a signal-handling mechanism for keyboard interrupts and a foreground/background process management system. It also enables user-defined signal handlers and ensures the preservation and restoration of process state during handler invocations, resulting in a more responsive and flexible process management system. Additionally, the scheduler modifications achieve fine-grained process execution control, fair CPU allocation through dynamic priorities, and comprehensive performance monitoring.

Future work could focus on introducing additional scheduling policies, more sophisticated priority calculations, and enhanced visualization of profiler data, further improving process management and system performance. These improvements would provide a more robust and efficient framework for managing processes and resources within xv6.