COL331: Operating Systems
Assignment 2 Report

Sanyam Garg — 2022CS11078
Aneeket Yadav — 2022CS11116

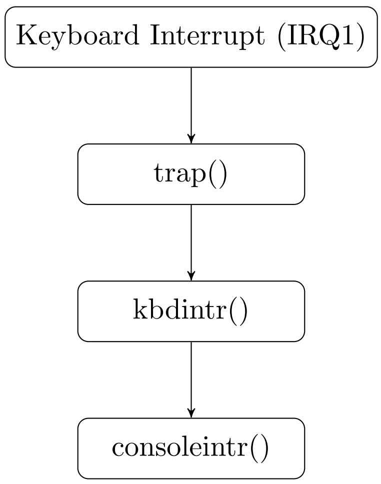
April 10, 2025

1 Introduction

This document details the control flow for custom signal handling in XV6, covering four key signals triggered by Ctrl+C (terminate), Ctrl+B (suspend), Ctrl+F (resume), and Ctrl+G (custom handler).

2 Common Initial Path

All signals follow the same initial path from hardware interrupt to signal dispatch:




```c
// trap.c
void trap(struct trapframe *tf) {
    case T_IRQ0 + IRQ_KBD:
        kbdintr();  // Calls consoleintr()
        lapiceoi();
        break;
}

// console.c
void consoleintr(int (*getc)(void)) {
    switch(c) {
        case C('C'): sigkill();
```

```
        case C('B'): sigstop();
        case C('F'): sigcont();
        case C('G'): sigcustom();
    }
}
```

Listing 1: Common Initial Code Path

3 Signal-Specific Control Flows

3.1 Ctrl+C (Process Termination)

1. `consoleintr()` detects Ctrl+C and calls `sigkill()`
2. `sigkill()` system call:
    (a) Acquires process table lock
    (b) Iterates through all processes (pid > 2)
    (c) Sets `killed=1` and `state=RUNNABLE`
    (d) Releases process table lock
3. On next trap check:
    (a) `trap()` sees `myproc()->killed`
    (b) Calls `exit()` terminating the process

```
// console.c
case C('C'):
    release(&cons.lock);
    sigkill();  // System call entry
    break;

// proc.c
void sigkill(void) {
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid > 2) {
            p->killed = 1;
            p->state = RUNNABLE;
        }
    }
    release(&ptable.lock);
}
```

Listing 2: Termination Sequence

3.2 Ctrl+B (Process Suspension)

1. `consoleintr()` detects Ctrl+B and calls `sigstop()`
2. `sigstop()` system call:
    (a) Acquires process table lock

(b) Sets `suspended=1` for all user processes

(c) Locates shell (pid 2) and wakes it

(d) Releases process table lock

3. Modified `scheduler()`:

    (a) Skips processes with `suspended==1`

    (b) Only schedules `RUNNABLE` processes

```
void sigstop(void) {
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid > 2) p->suspended = 1;
    }
    wakeup(sh->chan);  // Wake shell
    release(&ptable.lock);
}
```

Listing 3: Suspension Sequence

3.3 Ctrl+F (Process Resume)

1. `consoleintr()` detects Ctrl+F and:

    (a) Calls `sigcont()` system call

    (b) Checks `shell_status()` to determine shell state

2. `sigcont()` system call:

    (a) Acquires process table lock

    (b) Sets `suspended=0` for all processes

    (c) Changes state to `RUNNABLE`

    (d) Releases process table lock

3. Back in `consoleintr()`:

    (a) If shell was in `gets()` (sleeping):

        i. Acquires console lock
        ii. Injects 0x11 into input buffer
        iii. Wakes up shell using `wakeup()`
        iv. Releases console lock

    (b) If shell was already in `wait()`:

        i. No action needed (shell will handle resumed processes)

```c
// console.c
case C('F'):
    release(&cons.lock);
    sigcont();  // System call
    cprintf("Ctrl-F detected\n");
    if (!shell_status()) {  // Check after sigcont() returns
        acquire(&cons.lock);
        if (input.e - input.r < INPUT_BUF) {
            input.buf[input.e++ % INPUT_BUF] = 0x11;
            input.w = input.e;
            wakeup(&input.r);
        }
        release(&cons.lock);
    }
    break;

// proc.c
void sigcont(void) {
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->pid > 2 && p->suspended == 1 && p->state != ZOMBIE) {
            p->state = RUNNABLE;
            p->suspended = 0;
        }
    }
    release(&ptable.lock);
}

int shell_status(void) {
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->pid == 2) {
            int state = (p->state == SLEEPING) ? 0 : 1;
            release(&ptable.lock);
            return state;  // 0=in gets(), 1=in wait()
        }
    }
    release(&ptable.lock);
    return 1;
}
```

Listing 4: Resume Sequence

3.4 Ctrl+G (Custom Handler)

1. `consoleintr()` detects Ctrl+G and calls `sigcustom()`
2. `sigcustom()` system call:
   (a) Sets `call_handler=1` for current process if the signal handler has been registered.
3. On next trap:
   (a) `trap()` checks `call_handler`
   (b) Allocates and backs up trapframe

(c) Pushes FAKE_RETURN_ADDR to user stack

(d) Sets eip to handler address

4. Handler execution completes:

(a) Returns to FAKE_RETURN_ADDR

(b) trap() restores original state

```
void sigcustom(void) {
    acquire(&ptable.lock);
    myproc()->call_handler = 1;
    release(&ptable.lock);
}

// In trap():
if(myproc()->call_handler) {
    // Backup state
    myproc()->tf_backup = kalloc();
    memmove(myproc()->tf_backup, tf, sizeof(*tf));

    // Prepare stack
    tf->esp -= 4;
    *(uint*)tf->esp = FAKE_RETURN_ADDR;
    tf->eip = (uint)myproc()->signal_handler;
}
```

Listing 5: Custom Handler Sequence

4 Process State Management

Key modifications to core subsystems:

4.1 Scheduler Modifications

```
void scheduler(void) {
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE || p->suspended)
            continue;  // Skip suspended processes
        // ... normal scheduling ...
    }
}
```

4.2 Wait() System Call

```
int wait(void) {
    // Check for suspended children
    if(all_children_suspended()) return 0;
    // ... normal wait logic ...
}
```

5 Custom Fork

No significant design decisions.

6 Timing Profiler

For each process, time spent in a state is incremented when it transitions to another state. There is one exception to this rule. When the scheduler is invoked, it may be that some process was already in RUNNABLE state, and therefore its waiting time was not incremented. This may lead to incorrect scheduling. Therefore, waiting times for processes in RUNNABLE states are computed at this stage. Only those RUNNABLE processes which have not been suspended are allowed to participate in the scheduling process.

```
void setprocstate(struct proc *p, int new_state)
{
  uint current_time = ticks;
  uint duration = current_time - p->last_state_enter_time;

  switch (p->state)
  {
  case RUNNABLE:
    p->wt += duration; // Add to WT for RUNNABLE state
    break;
  case RUNNING:
    p->execution_time += duration; // Track execution time
    break;
  case SLEEPING:
    p->sleeping_time += duration; // Track execution time
    break;
  case HOLDING:
    p->holding_time += duration; // Track execution time
    break;
  }

  // Set response time on first transition to RUNNING
  if (new_state == RUNNING && p->state == RUNNABLE && !p->rt_recorded)
  {
    p->rt = current_time - p->ctime;
    p->rt_recorded = 1;
  }

  // Update dynamic priority when leaving RUNNING state or entering RUNNABLE state
  if (p->state == RUNNING || new_state == RUNNABLE)
  {
    p->dynamic_priority = calculate_priority(p);
  }

  // Update state and time
  p->state = new_state;
  p->last_state_enter_time = current_time;
}
```

7 Parameter Variation in Scheduling

No statistically significant change in the parameters was observed when ratio of α/β was varied from 1e-6 to 1e6.

8 Conclusion

This implementation provides robust signal handling while maintaining XV6's simplicity. Each signal type follows a clear path from interrupt to final action, with proper state preservation for custom handlers.