Operating Systems
Assignment 2 – Easy

Nikhil Gupta (2024JCS2611)
Abhishek Gupta (2024JCS2047)

April 10, 2025

1 Signal Handling in xv6

1.1 Overview

We extended `xv6` to support signal handling for certain keyboard-triggered inputs. These signals mimic standard UNIX-style behavior for process control and signal handling. We implemented the following signals:

- `Ctrl+C` → `SIGINT` — terminate user processes
- `Ctrl+B` → `SIGBG` — suspend user processes
- `Ctrl+F` → `SIGFG` — resume suspended processes
- `Ctrl+G` → `SIGCUSTOM` — invoke a user-defined handler

Kernel Extensions for Signal Handling

To support the signal mechanism, we made the following changes to the kernel:

- **Global Signal State** (`proc.h`):

```
// Signal Identifiers
enum kibs { NOSIG, SIGINT, SIGBG, SIGFG, SIGCUSTOM };
extern enum kibs curkibs;
```

`curkibs` is a global variable that tracks the currently pending signal. It is updated in the console interrupt handler and consumed in trap handling.

- **New Process State**:

```
// procstate enum
UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE, SUSPENDED
```

We introduced a new state `SUSPENDED` to pause processes without terminating them.

- **Per-Process Signal Handler**:

```
// struct proc
void (*signal_handler)();
```

A new function pointer was added to each process to store the address of a user-registered custom signal handler for `SIGCUSTOM`.

1.2 Signal Detection (`console.c`)

The console input handler `consoleintr()` is invoked on every keyboard interrupt. It reads the incoming character, checks if it's a control signal, and maps it to an appropriate signal enum using `dispatchsig()`.

`dispatchsig()` updates the global kernel signal tracker `curkibs`, which is later processed in the trap handler to perform signal-specific behavior such as termination, suspension, or user-defined handler invocation.

```
case C('C'):
  printkey('C');
  dispatchsig(SIGINT);
  break;

case C('G'):
  printkey('G');
  dispatchsig(SIGCUSTOM);
  break;
```

The `printkey()` function formats the message "Ctrl -X is detected by xv6" for debug visibility.

1.3 SIGINT (Ctrl+C)

Purpose

Terminate all user processes (PID `>` 2).

Additionally, `Ctrl+C` injects the string `sw\n` into the shell's input buffer. This causes the shell to invoke `wait()`, helping it collect zombie processes without needing user input.

Control Flow

1. Ctrl+C triggers `SIGINT` via `dispatchsig()`.
2. All user processes are marked `killed = 1` and set to `RUNNABLE`.
3. The shell receives `sw` in its input buffer, parses it, and calls `wait()`.
4. On the next syscall or trap, killed processes terminate.

```
// Signal dispatch for SIGINT
if (signal == SIGINT) {
  for (...) {
    if (p->pid > 2 && valid_state(p->state)) {
      p->killed = 1;
      p->state = RUNNABLE;
    }
  }
}

// Inject 'sw' into input buffer (console.c)
input.buf[input.e++ % INPUT_BUF] = 's';
input.buf[input.e++ % INPUT_BUF] = 'w';
input.buf[input.e++ % INPUT_BUF] = '\\n';
input.w = input.e;

// Shell handles 'sw' to clean up zombies (sh.c)
if (strcmp(buf, "sw\\n") == 0) {
  wait();
  continue;
}
```

1.4 SIGBG (Ctrl+B)

Purpose

Suspend user processes and wake up the shell.

Control Flow

1. Ctrl+B triggers `SIGBG`.
2. All user processes (PID `> 2`) are set to `SUSPENDED`.
3. The shell process (PID `= 2`), if sleeping on itself, is made `RUNNABLE`.
4. At the end of `trap()`, before returning to user mode, the kernel checks if the current process is suspended. If so, it yields the CPU.

```
if (signal == SIGBG) {
  for (...) {
    if (p->pid > 2)
      p->state = SUSPENDED;
  }
  for (...) {
    if (p->pid == 2 && p->state == SLEEPING && p->chan == p)
      p->state = RUNNABLE;
  }
}

// In trap.c
if(myproc() && myproc()->state == SUSPENDED && (tf->cs & 3) == DPL_USER)
  yield();
```

Why suspended processes are skipped in `wait()`: When a process is suspended using `Ctrl+B`, it is moved to the `SUSPENDED` state but not terminated. To ensure that the shell doesn't block while waiting for such suspended children, the `wait()` system call explicitly skips over all children in the `SUSPENDED` state.

This allows control to return to the shell immediately after suspension, enabling it to accept further input or resume processes later.

```
// In wait() function
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->parent != curproc || p->state == SUSPENDED)
    continue;
  ...
}
```

Yield behavior: A process only becomes RUNNABLE again if it is not in the `SUSPENDED` state.

```
// In yield()
void yield(void) {
  acquire(&ptable.lock);
  if(myproc()->state != SUSPENDED)
    myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

1.5 SIGFG (Ctrl+F)

Purpose

Resume suspended processes by making them `RUNNABLE` again.

Control Flow

1. Ctrl+F triggers `SIGFG`.
2. Every process in `SUSPENDED` state is changed to `RUNNABLE`.

```
if (signal == SIGFG) {
  for (...) {
    if (p->state == SUSPENDED)
      p->state = RUNNABLE;
  }
}
```

1.6 SIGCUSTOM (Ctrl+G)

Purpose

Invoke a user-defined signal handler by modifying the trapframe.

Control Flow

1. Ctrl+G is captured and mapped to `SIGCUSTOM`.
2. `dispatchsig()` sets the global `curkibs = SIGCUSTOM`.
3. During the next trap (e.g., timer interrupt), if `curkibs == SIGCUSTOM` and the process has a registered handler, the trapframe is modified:
   - Push the current `eip` onto the user stack
   - Set `eip` to the handler function address
   - Clear the signal flag
4. The user-defined handler executes in user space.
5. When the handler returns, it pops the old `eip`, returning to the original user instruction.

```
if (myproc()->signal_handler && curkibs == SIGCUSTOM) {
  myproc()->tf->esp -= 4;
  *(uint*)(myproc()->tf->esp) = myproc()->tf->eip;
  myproc()->tf->eip = (uint)myproc()->signal_handler;
  curkibs = NOSIG;
}
```

Stack Layout

```
      Before Trap
     +-------------+
esp  |  user stuff |
     +-------------+
eip  normal execution

      During Trap
     +-------------+
esp-4|  old eip    | manually pushed
     +-------------+
eip  handler address

      In Handler
     +-------------+
     |  old eip    | esp
     +-------------+
eip  handler()

      After return
     +-------------+
esp+4| back to user| execution resumes where it left off
     +-------------+
```

2 xv6 Scheduler

2.1 Overview

In this task, we enhanced the default round-robin scheduler in `xv6` to support:

- Delayed process execution using a new process state (`WAITING_TO_START`).
- A specified execution time for processes (`exec_time`).
- Dynamic priority-based scheduling using parameters α and β.
- Detailed process profiling of TAT, WT, RT, and #CS.

These changes allow more fine-grained control over how processes are scheduled, run, and eventually terminated.

2.2 Kernel Modifications

New Process State: `WAITING_TO_START` We added a new state in `procstate`:

```
// in proc.h
enum procstate {
  UNUSED, EMBRYO, SLEEPING, RUNNABLE,
  RUNNING, ZOMBIE, SUSPENDED,
  WAITING_TO_START
};
```

This defers the process from being scheduled immediately.

Process Profiling Fields In `struct proc`, we track:

```
// struct proc
int arrival_time;      // The creation time of the process
int waiting_time;      // Time spent in RUNNABLE state
int run_time;          // CPU time used so far
int exec_time;         // Max CPU time allowed (-1 if unlimited)
int completion_time;   // Time process finishes
int first_run_time;    // Time process first runs
int cs;                // Context switch count
```

Trap Handling and `updatewaittime()` In `trap.c`, the timer interrupt increments `run_time` for the running process, checks if it's exceeded `exec_time`, and calls `updatewaittime()` for `RUNNABLE` processes:

```
// In trap.c - IRQ_TIMER handler
updatewaittime();  // increment waiting_time for RUNNABLE processes

if(myproc() && myproc()->state == RUNNING) {
  myproc()->run_time++;
  if(myproc()->exec_time > 0 &&
     myproc()->run_time >= myproc()->exec_time)
    myproc()->killed = 1; // Exceeded allotted exec_time
}
```

`updatewaittime()` simply iterates over the process table, incrementing `waiting_time` for all `RUNNABLE` processes:

```
void updatewaittime(void) {
  acquire(&ptable.lock);
  for(struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == RUNNABLE)
      p->waiting_time++;
  }
  release(&ptable.lock);
}
```

2.3 Custom Scheduler and Process Management

2.3.1 Scheduler Modifications

We replaced round-robin logic with a priority-based selection. Each `RUNNABLE` process's priority is:

$$\text{priority} = \text{INIT\_PRIORITY} - \alpha \cdot \text{run\_time} + \beta \cdot \text{waiting\_time}.$$

Highlights:

- **Scanning the table:** We compute each `RUNNABLE` process's priority. The one with the highest priority is chosen:

```
int highest_priority = -1;
struct proc *sched_proc = 0;

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

  p->priority = INIT_PRIORITY
              - ALPHA * p->run_time
              + BETA * p->waiting_time;
  if(p->priority < 0)
    p->priority = 0;

  if(p->priority > highest_priority){
    highest_priority = p->priority;
    sched_proc = p;
  }
}

// If no candidate found, fallback to shell (pid=2) etc.
```

- **Fallback to Shell:** If no user process is found, we pick the shell (PID=2).
- **First Run Time:** If `first_run_time` == -1, we set it to `ticks` the first time we run the process:

```
if(sched_proc->first_run_time == -1) {
  sched_proc->first_run_time = ticks;
}
```

- **Context Switch Counting:** If we schedule a different process than last time, we increment `cs`:

```
if(last_sched_proc != sched_proc) {
  sched_proc->cs++;
}
last_sched_proc = sched_proc;
```

2.3.2 `custom_fork()` System Call

We created `custom_fork(`*`start_later_flag, exec_time`*`)` so new child processes can:

- **Set** `exec_time`: We store `exec_time` in `np->exec_time` before deciding the process state:

```
np->exec_time = exec_time; // e.g., 50 ticks or -1 for unlimited
```

- **Start Later if Requested**: If `start_later_flag` is `true`, the child goes to `WAITING_TO_START`; otherwise it's `RUNNABLE`:

```
if(start_later_flag)
  np->state = WAITING_TO_START;
else
  np->state = RUNNABLE;
```

The `-1` default for `exec_time` means "no limit".

2.3.3 Process Profiling

At `exit()`, we log each process's:

- Turnaround Time (TAT) = `completion_time - arrival_time`
- Waiting Time (WT) = `waiting_time`
- Response Time (RT) = `first_run_time - arrival_time`
- Context Switches (#CS) = `cs`

```
// Inside exit()
cprintf("PID: %d\\n", curproc->pid);
cprintf("TAT: %d\\n", curproc->completion_time - curproc->arrival_time);
cprintf("WT: %d\\n", curproc->waiting_time);
cprintf("RT: %d\\n", curproc->first_run_time - curproc->arrival_time);
cprintf("#CS: %d\\n", curproc->cs);
```

2.4 Discussion of α and β

- **High α, low β:** Strongly penalizes CPU-heavy jobs. I/O-bound or short jobs are favored.
- **Low α, high β:** Boosts priority of processes with long waiting times, preventing starvation.
- **Balanced $\alpha \approx \beta$:** Offers an in-between approach for fairness between CPU-bound and waiting processes.