Assignment 2

Operating System (ELL 783)

Submitted by

Name: Aditya Prakash
Entry No.: 2024EET2855

Name: Avadhesh Kumar
Entry No.: 2024EET2799

Submitted on: **April 10, 2025**

1 Introduction

This assignment focuses on improving process management in the xv6 operating system by implementing custom signal handling and modifying the scheduler to support profiling and priority boosting.

The work is divided into two main tasks:

- **Task 1 – Signal Handling:** Implementation of four keyboard-interrupt-based signals:
  - **SIGINT (Ctrl+C):** Terminates all processes (excluding init and shell).
  - **SIGBG (Ctrl+B):** Suspends all user processes.
  - **SIGFG (Ctrl+F):** Resumes suspended processes.
  - **SIGCUSTOM (Ctrl+G):** Invokes a user-defined signal handler if registered.
- **Task 2 – Custom Scheduler:**
  - Introduction of a `custom_fork()` system call that allows delayed execution and sets an execution time limit.
  - A `scheduler_start()` system call to start execution of delayed processes.
  - Profiling metrics Turnaround Time (TAT), Waiting Time (WT), Response Time (RT), and Context Switches (#CS).
  - A dynamic priority boosting scheduler based on CPU and waiting time, controlled by tunable parameters α and β.

2 Signal Handling in xv6

This section explains the implementation of keyboard-based signal handling in xv6, where specific key combinations generate predefined signals to manipulate user-level processes.

2.1 Control Flow of Keyboard Signals

The following describes the flow from keypress detection to process manipulation:

- **Key Detection:** ‘consoleintr()‘ in `console.c` is modified to detect Ctrl+C (ASCII 3), Ctrl+B (ASCII 2), Ctrl+F (ASCII 6), and Ctrl+G (ASCII 7).
- **Process Handling:** These functions(broadcast_signal(), send_signal(), handle_signals()) are implemented in `proc.c`, where the logic for manipulating the process state is defined.

2.2 Modifications in `proc.h`

To support signal handling, the following changes were made in `proc.h`:

```
#define SIGINT      1  // Interrupt signal
#define SIGSTP      2  // Stop signal
#define SIGFG       3  // Foreground signal
#define SIGCUSTOM   4  // Custom signal

typedef void (*sighandler_t)(void);  // Signal handler fn pointer
```

Listing 1: Signal Macros and Signal Handler Type

The process structure was extended to include signal-related fields:

```
int pending_signals;          // Bitmap of pending signals
int is_suspended;             // Suspended state flag
sighandler_t custom_handler; // Registered SIGCUSTOM handler
uint saved_eip;               // Save ip pointer b4 invoking handler
```

Listing 2: New Signal Fields in struct proc

2.3 Code Snippet from `console.c`

The following code was added in `consoleintr()` function in `console.c` to detect specific keypress combinations and generate corresponding signals:

```
if(c == '\003') {  // Ctrl+C (ETX)
  release(&cons.lock);
  cprintf("Ctrl-C is detected by xv6\n");
  broadcast_signal(1);
  acquire(&cons.lock);
  continue;
} else if(c == C('B')) {  // Ctrl+B (SUB)
  release(&cons.lock);
  cprintf("Ctrl-B is detected by xv6\n");
  broadcast_signal(2);
  acquire(&cons.lock);
  continue;
} else if(c == '\006') {  // Ctrl+F (ACK)
  release(&cons.lock);
  cprintf("Ctrl-F is detected by xv6\n");
  broadcast_signal(3);
  acquire(&cons.lock);
  continue;
} else if(c == C('G')) {  // Ctrl+G
  release(&cons.lock);
  cprintf("Ctrl-G is detected by xv6\n");
  broadcast_signal(4);
  acquire(&cons.lock);
  continue;
}
```

Listing 3: Handling Ctrl+C/B/F/G in console.c

2.4 Code Snippets from `proc.c`

The following functions were added to `proc.c` to implement signal handling.

2.4.2 Signal Management Logic

```
void send_signal(struct proc *p, int signum)
{
  p->pending_signals |= (1 << signum);
}

void handle_signals(struct proc *p)
{
  if(p->pid <= 2)
    return;

  if(p->pending_signals & (1 << SIGINT)) {
    p->pending_signals &= ~(1 << SIGINT);
    p->killed = 1;
    p->state = RUNNABLE;
  }

  if(p->pending_signals & (1 << SIGSTP)) {
    p->pending_signals &= ~(1 << SIGSTP);
    p->is_suspended = 1;
    p->state = SUSPENDED;
  }

  if(p->pending_signals & (1 << SIGFG)) {
    p->pending_signals &= ~(1 << SIGFG);
    if(p->is_suspended && p->state == SUSPENDED){
      p->is_suspended = 0;
      p->state = RUNNABLE;
    }
  }

  if(p->pending_signals & (1 << SIGCUSTOM)) {
    p->pending_signals &= ~(1 << SIGCUSTOM);
    if(p->custom_handler != 0){
      p->saved_eip = p->tf->eip;
      p->tf->eip = (uint)p->custom_handler;
    }
  }
}
```

Listing 4: Sending and handling signals in proc.c

2.4.3 Broadcasting Signals

```
void broadcast_signal(int signum){
  struct proc *p;
  int x = 0;
  acquire(&ptable.lock);

  if(signum == 3 ) {
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->is_suspended){
        x++;
        send_signal(p, SIGFG);
        handle_signals(p);
      }
```

```
    }
    if(!x)
      cprintf("No background Process\n");
    release(&ptable.lock);
    return;
  }

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid > 2 && p->state != UNUSED){
      send_signal(p, signum);
    }
  }
  release(&ptable.lock);
}
```

Listing 5: Broadcast signal function in proc.c

2.5 Modifications in `sysproc.c`

To support registering and returning from a custom signal handler, the following system calls were implemented in `sysproc.c`:

```
int sys_signal(void)
{
  char *handler;
  if (argptr(0, &handler, sizeof(void(*)())) < 0)
    return -1;
  myproc()->custom_handler = (void(*)())handler;
  return 0;
}
```

Listing 6: System Call: sys_signal

This system call allows a process to register a handler function for the `SIGCUSTOM` signal.

```
int sys_sigret(void) {
  struct proc *p = myproc();
  // Restore saved trapframe
  p->tf = p->saved_eip;
  return 0;
}
```

Listing 7: System Call: sys_sigret

The `sys_sigret` system call restores the saved instruction pointer to allow the process to resume normal execution after the custom signal handler has finished.

3 Custom Fork and Scheduler Start System Calls

To support advanced scheduling behavior and process control, we implemented two new system calls:

- **sys_custom_fork:** Allows a child process to be created with specific scheduling parameters.
- **sys_scheduler_start:** Transitions all child processes created with delayed execution into the RUNNABLE state.

Modifications in `proc.c`

To implement dynamic priority-based scheduling and process metric tracking, we introduced several variables and logic into `proc.c`.

3.1 Implementation of sys_custom_fork and sys_scheduler_start

```
int sys_custom_fork(void)
{
  int start_later, exec_time;
  struct proc *np;
  struct proc *p = myproc();

  if(argint(0, &start_later) < 0 || argint(1, &exec_time) < 0)
    return -1;

  if((np = allocproc()) == 0)
    return -1;

  if((np->pgdir = copyuvm(p->pgdir, p->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }

  np->sz = p->sz;
  np->parent = p;
  *np->tf = *p->tf;
  np->tf->eax = 0;

  for(int i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);
  safestrcpy(np->name, p->name, sizeof(p->name));

  np->start_later = start_later;
  np->exec_time = exec_time;
  np->creation_time = ticks;
  np->first_scheduled = -1;
  np->cpu_ticks_used = 0;
  np->wait_time = 0;
  np->context_switches = 0;
  np->initial_priority = INIT_PRIORITY;
  np->dynamic_priority = INIT_PRIORITY;

  int pid = np->pid;
  acquire(&ptable.lock);

  if(start_later){
    np->state = SLEEPING;
  } else {
    np->state = RUNNABLE;
  }

  release(&ptable.lock);
```

```
51   return pid;
52 }
```

Listing 8: Implementation of sys_custom_fork

```
1  int sys_scheduler_start(void)
2  {
3    struct proc *p;
4
5    acquire(&ptable.lock);
6    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
7      if(p->start_later) {
8        p->state = RUNNABLE;
9      }
10   }
11   release(&ptable.lock);
12   return 0;
13 }
```

Listing 9: Implementation of sys_scheduler_start

These system calls are essential to implement controlled process spawning and starting. This is especially useful when coordinating execution timing and testing custom schedulers.

3.2 Initialization of Process Metrics in `allocproc()`:

```
1 p->start_later = 0;
2 p->exec_time = -1;
3 p->creation_time = ticks;
4 p->first_scheduled = -1;
5 p->cpu_ticks_used = 0;
6 p->wait_time = 0;
7 p->context_switches = 0;
8 p->initial_priority = INIT_PRIORITY;
9 p->dynamic_priority = INIT_PRIORITY;
```

Listing 10: Initializing new fields in `allocproc()`

3.3 Printing Process:

```
1 p = curproc;
2 int turnaround_time = ticks - p->creation_time;
3 int response_time = p->first_scheduled - p->creation_time;
4
5 cprintf("PID: %d\n", p->pid);
6 cprintf("TAT: %d\n", turnaround_time);
7 cprintf("WT: %d\n", p->wait_time);
8 cprintf("RT: %d\n", response_time);
9 cprintf("#CS: %d\n", p->context_switches);
```

Listing 11: Printing turnaround time, waiting time, response time, and context switches

3.4 Dynamic Priority Update and Scheduling Logic:

```
1 // Update dynamic priority using formula
2 int dynamic = p->initial_priority -
3               (ALPHA * p->cpu_ticks_used) +
```

```
            (BETA * p->wait_time);

if (dynamic > MAX_PRIORITY)
    p->dynamic_priority = MAX_PRIORITY;
else if (dynamic < 0)
    p->dynamic_priority = 0;
else
    p->dynamic_priority = dynamic;

// Select highest priority process
if (p->state == RUNNABLE) {
    if (highest == 0 ||
        p->dynamic_priority > max_priority ||
        (p->dynamic_priority == max_priority && p->pid < highest->pid))
            {
        highest = p;
        max_priority = p->dynamic_priority;
    }
}
```

Listing 12: Dynamic priority update and selection of process

3.5 Changes in `proc.h`

The following fields were added to the `proc` structure in `proc.h` to support delayed execution, process profiling, priority-based scheduling, and signal handling mechanisms:

```
  // For custom fork
  int start_later;          // Flag to indicate delayed scheduling
  int exec_time;            // Maximum execution time

  // For scheduler profiling
  int creation_time;        // When the process was created
  int first_scheduled;      // When first scheduled (for response time)
  int cpu_ticks_used;       // CPU time consumed
  int wait_time;            // Time spent waiting
  int context_switches;     // Number of context switches

  // For priority scheduling
  int initial_priority;     // Initial priority value
  int dynamic_priority;     // Current priority value
```

Listing 13: Modified fields in `struct proc` in `proc.h`

4 Scheduler Profiler and Effects of Parameters

4.1 Scheduler Profiler Output

The modified scheduler records and prints key performance metrics for every process upon termination. These include:

- **Turnaround Time (TAT):** Total time from process creation to termination.
- **Waiting Time (WT):** Time spent waiting in the RUNNABLE state.

- **Response Time (RT):** Time between process creation and its first scheduling.
- **Context Switches (#CS):** Number of times the process was scheduled.

The following is a sample output generated by the scheduler:

```
PID: 4
TAT: 120
WT: 40
RT: 20
#CS: 6
```

Listing 14: Sample Scheduler Output

These metrics help assess how scheduling parameters impact process execution, responsiveness, and fairness.

4.2 Effects of α and β

The dynamic priority of a process $\pi_i(t)$ is computed as:

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

- $\pi_i(0)$**:** Initial priority defined in the Makefile.
- $C_i(t)$**:** CPU time consumed so far.
- $W_i(t)$**:** Total wait time in the RUNNABLE state.
- α, β**:** Tunable weights defined via Makefile.

Observations:

- A high α causes CPU-bound processes to lose priority rapidly, favoring short or I/O-bound jobs.
- A high β boosts the priority of long-waiting processes, helping prevent starvation.
- Proper tuning balances fairness (via β) and responsiveness (via α).

In our experiments, increasing β caused I/O-bound or delayed processes to get scheduled earlier, whereas increasing α penalized processes consuming too much CPU.

3.3 Comparison of Results with Different α and β Values

To evaluate the influence of α and β on scheduling outcomes, we executed identical workloads under different tuning parameters. The following table summarizes the observed metrics:

Test	α	β	PID	TAT	WT	RT
1	1	1	4 (CPU)	120	40	20
			5 (I/O)	90	20	10
2	3	1	4 (CPU)	140	60	30
			5 (I/O)	85	18	9
3	1	3	4 (CPU)	110	30	20
			5 (I/O)	70	10	5

Table 1: Effect of α and β on CPU-bound and I/O-bound processes

- **Test 1 (Balanced parameters):** Results are fair but not optimal for specific workloads.
- **Test 2 (high α):** CPU-bound process (PID 4) is penalized with longer TAT and WT.
- **Test 3 (high β):** I/O-bound process (PID 5) receives a significant priority boost and completes earlier.

5 Conclusion

In this assignment, we extended the xv6 operating system to support advanced process management features. The implementation included:

- A priority-based scheduler with profiling capabilities, recording metrics such as turnaround time, waiting time, response time, and context switches.
- A dynamic priority adjustment formula that effectively balances CPU usage and fairness using tunable parameters α and β.

Through this implementation, we gained deeper insights into:

- How kernel-level signal handling works and its effect on user processes.
- The importance of maintaining accurate process profiling data in operating systems.
- The trade-offs involved in designing schedulers, especially between responsiveness and fairness.

5 Final Notes and Observations

- While testing the assignment code, it was observed that the program takes longer time to compile and start running, so we increased the sleep time in `check_script.sh` to let the program compile and start in time.
- Compilation issues were encountered due to the `-Werror` flag in the Makefile, which treated all warnings as errors. To allow the xv6 build process to proceed without interruption, we can remove the `-Werror` flag from the Makefile. This was essential, especially during incremental development and testing of system calls.