SIGNAL HANDLING AND SCHEDULER IN xv6

OPERATING SYSTEM: ASSIGNMENT 2-EASY

2022CS11599 – Aniket Chattopadhyay ||||| 2022CS11605 – Avni Aggarwal

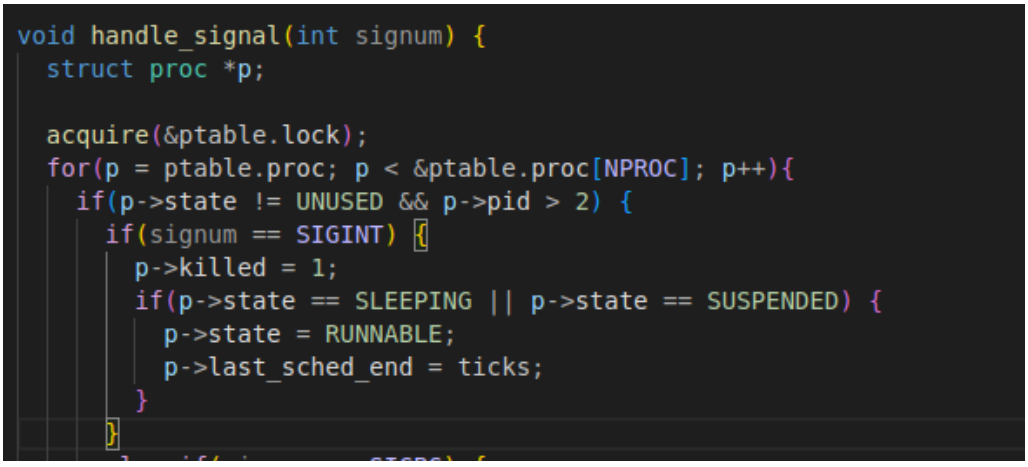
1 Introduction

2 Signal Handling in xv6

2.1 Implementation of SIGINT (CTRL+C)

The role of SIGINT signal is to kill all the process apart from initproc() and shell() (i.e. pid = 1 and pid = 2)

When **CTRL+C** is pressed, the terminal driver recognizes it as the interrupt (INTR) character and sends a SIGINT signal to the processes with pid > 2 (signal is sent to this process group). In the kernel, handle_signal(int signum) in proc.c sets the killed flag of the process. If some process to be killed is sleeping, it is woken up so the signal can be handled. The process will then exit safely—either immediately if in user space, or after returning from kernel mode and releasing any held locks.



```
void handle_signal(int signum) {
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != UNUSED && p->pid > 2) {
      if(signum == SIGINT) {
        p->killed = 1;
        if(p->state == SLEEPING || p->state == SUSPENDED) {
          p->state = RUNNABLE;
          p->last_sched_end = ticks;
        }
      }
```

Fig 1: handle_signal() to kill process implemented in proc.c

2.2 Implementation of SIGBG (CTRL+B) and SIGFG(CTRL_F)

The role of SIGBG signal is to pause the execution of all the process apart from initproc() and shell() (i.e. pid = 1 and pid = 2) and get the shell to start running. Note that even if a process is in SLEEPING state, it is getting paused. In the kernel, handle_signal(int signum) in proc.c sets the suspended flag of the process.

When **CTRL+B** is pressed, the terminal driver recognizes it as the interrupt (INTR) character and sends a SIGBG signal to the processes with pid > 2 (signal is sent to this process group).

The suspended flag is handled when it is trapped via scheduling or any other means. So, whenever the process is comfortable to handle the suspended flag, it would handle it (i.e. when it is not doing any critical operations such as holding locks, handling interrupts etc.)

To ensure that the shell (i.e. pid=2) wakes up after all the processes are suspended, we modify the wait() for the shell. To do this, if all children are suspended, we return with a –2 value to allow the shell to take more input. Also, we explicitly wake up the shell once the required processes are suspended.

To resume execution, simply **CTRL+F** would restore the state of ALL the paused processes to runnable.

```
else if(signum == SIGBG) {
  p->suspended = 1;
}
else if(signum == SIGFG) {
  if(p->state == SUSPENDED && p->killed == 0) {
    p->state = RUNNABLE;
    p->suspended = 0;
    p->last_sched_end = ticks;
  }
}
```

Fig 1: handle_signal() to pause/resume execution of process implemented in proc.c

```
// Trap code for handling SIGBG (CTRL+B)
if(myproc() && myproc()->suspended) {
  myproc()->state = SUSPENDED;
  wakeup(myproc()->parent);
  yield_nR();
}
```

Fig 2: trap code for handling SIGBG

```
int
wait(void)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;
      }
      if(p->state == SUSPENDED) {
        release(&ptable.lock);
        return -2;
      }
    }
```

Fig 3: modified wait() in proc.c to handle suspended children

```
// Read and run input commands.
while(getcmd(buf, sizeof(buf)) >= 0){
  if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
    // Chdir must be called by the parent, not the child.
    buf[strlen(buf)-1] = 0;  // chop \n
    if(chdir(buf+3) < 0)
      printf(2, "cannot cd %s\n", buf+3);
    continue;
  }
  if(fork1() == 0)
    runcmd(parsecmd(buf));
  if(wait()==-2)
    continue;
}
```

Fig 4: modified sh.c to go along

2.3 Implementation of SIGCUSTOM (CTRL+G)

The role of signal(sighandler_t handler) is to register a handler function with a user process. The user registers this handler function via the signal system call, which calls the sys_signal function in the kernel, which stores the handler function's address in the process struct (in the kernel space, so that it can be used later)

To call this custom handler, we simply press **CTRL+G** to link to the handle_signal() function via keyboard interrupt. Here, a simple flag is set active for similar reasons as in above cases. Now, we handle the remaining code in trap.c file. Firstly, we need to save the entire context and trap frame of the process so that we can resume execution normally after executing the code of the handler. However, due to limited stack space, this may sometimes lead to a page fault. In that case, we simply reuse the stored trap frame so that code execution resumes normally.

```
int
sys_signal(void) {
  sighandler_t handler;
  // Fetch the handler address from user space
  if (argptr(0, (void*)&handler, sizeof(sighandler_t)) < 0) {
    return -1;
  }
  // Validate the handler is within user space
  if ((uint)handler >= KERNBASE || handler == 0) {
    return -1;
  }
  // Assign the handler to the current process
  struct proc *curproc = myproc();
  curproc->custom_handler = handler;
  return 0;
}
```

Fig 1: handle the signal system call in the kernel space

```
if (myproc() && (tf->cs & 3) == DPL_USER) {
  struct proc *p = myproc();

  // Handle SIGCUSTOM if pending and handler is registered
  if (p->sigcustom_pending && p->custom_handler) {
      if(p->fb_tf == 0) {
          p->fb_tf = (struct trapframe*) kalloc();
          memmove(p->fb_tf, tf, sizeof(struct trapframe));
      }
      else {
          // Cast to char* to match kfree's expected parameter type
          kfree((char*)p->fb_tf);
          p->fb_tf = (struct trapframe*) kalloc();
          memmove(p->fb_tf, tf, sizeof(struct trapframe));
      }

      uint* esp = (uint*)(p->tf->esp);
      *--esp = p->tf->esp;
      *--esp = p->tf->eflags;
      *--esp = p->tf->eip; // Push the return address onto the stack
      *--esp = 0;

      tf->esp = (uint)esp; // Update the stack pointer
      tf->eip = (uint)p->custom_handler; // Set instruction pointer to handler

      p->sigcustom_pending = 0; // Clear pending signal

  }
}
```

Fig 2: trap handling for SIGCUSTOM – load the context parameters (in user stack) and save the existing trapframe for use in page faults

3 xv6 Scheduler

3.1 Custom Fork and Scheduler Start

We implemented a custom fork system call that takes two arguments:

Start Later Flag: If set to true, the forked process is not scheduled immediately. It becomes runnable only after calling the scheduler_start() system call.

Execution Time: Specifies how many ticks the process should run before being terminated by the kernel. If set to -1, the process runs indefinitely.

Additionally, we implemented the scheduler_start() system call to activate all deferred processes created with the custom fork. It has no effect if no such processes exist.

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

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == EMBRYO && p->start_later == 1){
      p->state = RUNNABLE;
      p->last_sched_end = ticks;
    }
  }
  release(&ptable.lock);

  return 0;
}
```

Fig 1. Scheduler_start function implementation

3.2 Scheduler Profile:

TAT (Turnaround Time) : The time process execution finishes – creation time

WT (Waiting time) : Time the process spent in RUNNABLE state

RT (Response time) : Time the process was first scheduled – creation time

#CS (No. of context switches) : Whenever an executing user process (P1) gets switched to another user process (P2) by the scheduler, the count of #CS in P1 increases.

Creation time is initialized in allocproc().

```
// For custom fork
int start_later;          // Flag to delay scheduling
int exec_time;            // Execution time in ticks
int cpu_ticks_used;       // CPU time used so far

// For profiling
int creation_time;        // Time when process was created
int first_scheduled;      // Time when process is first scheduled
int exit_time;            // in exit()
int total_waiting_time;   // WT
int context_switches;     // #CS
int last_sched_end;       // when process last stopped running
int last_run_start;       // when process started running

// For priority scheduling
int initial_priority;     // π(0)
int dynamic_priority;     // Current priority
```

Fig 2. proc.h flags

3.3 Priority Boosting Scheduler:

Dynamic priority is calculated like this:

$$\pi i(t) = \pi i(0) - \alpha \cdot Ci(t) + \beta \cdot Wi(t)$$

• $\pi i(0)$ is the initial priority assigned to all processes. This parameter should be specified in the Makefile.

• $Ci(t)$ is the CPU ticks consumed by process Pi up to time t ticks.

• $Wi(t)$ is the waiting time (the duration for which a process is waiting for the CPU core; starting from the time of its creation).

• α, β are tunable weighting factors that balance CPU consumption and waiting time.

```
// Priority scheduler logic
struct proc *highest_priority = 0;
int highest_value = -1000000; // Very low initial value

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

  // Calculate waiting time since last scheduled
  if (p->last_sched_end < ticks) {
    int waited = ticks - p->last_sched_end;
    p->total_waiting_time += waited;
    p->last_sched_end = ticks; // Reset to avoid double-counting
  }

  // Calculate dynamic priority
  p->dynamic_priority = p->initial_priority - (ALPHA * p->cpu_ticks_used) + (BETA * p->total_waiting_time);

  if (p->dynamic_priority < 0) {
    p->dynamic_priority = 0;
  }

  if (p->dynamic_priority > highest_value) {
    highest_value = p->dynamic_priority;
    highest_priority = p;
  }
  else if (p->dynamic_priority == highest_value) {
    // Break ties with lowest PID
    if (highest_priority == 0 || p->pid < highest_priority->pid) {
      highest_priority = p;
    }
  }
}

// If we found a process to run
if(highest_priority != 0){
  p = highest_priority;

  if (c->last_proc != 0 && c->last_proc != p) {
    c->last_proc->context_switches++;
  }
  c->last_proc = p;

  // Switch to chosen process.
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;

  // Record first execution time for response time calculation
  if(p->first_scheduled == -1){
    p->first_scheduled = ticks;
  }


  swtch(&(c->scheduler), p->context);
  switchkvm();
```

Fig 3. Priority handling in scheduler

The effect of alpha and beta on the profiling outputs: (initial priority is set to be 100)

Alpha	Beta	Avg TAT	Avg WT	Avg RT	Avg #CS
0	0	628	11	2	502
0	1	634	10	3	507
1	0	630	12	2	510
1	1	632	9	2	505
2	1	635	11	2	509
1	2	629	9	3	511
1	5	627	10	2	506
5	1	636	11	3	514

Conclusion from the results:

Since the values are quite close as there are not many processes and heavy load, we are not able to distinguish as much. But theoretically, these are the results.

Scenario (α, β)	Effect on Metrics	Ideal For
High α, High β	- Frequent context switches (↑ **#CS**) - Balanced turnaround time (**TAT**) for both CPU- and I/O-bound processes	Mixed workloads
High α, Low β	- CPU-bound processes may **starve** - Low waiting time (**WT**) for I/O-bound tasks	I/O-heavy workloads
Low α, High β	- CPU-bound processes **dominate** - High waiting time (**WT**) for I/O-bound tasks	CPU-heavy workloads
Low α, Low β	- Potential for **unfair scheduling** and **starvation** - High turnaround time (**TAT**) for waiting processes	**Not recommended**