Operating Systems Assignment 2 - Easy

Aditya Jha 2022CS11102
Rishit Jakharia 2022CS11621

1 Signal Handling in xv6

1.1 Console input

Updated the *console.c* file to accept inputs;
Using definitions like `#define SIGINT 1`
Code being as follows;

```
// console.c
void consoleintr(int (*getc)(void)){
...
    switch(c){
    ...
    case C('C'):
      dosignal = SIGINT;
      break;
    case C('B'):
      dosignal = SIGBG;
      break;
      ... // Similarly for Ctrl+F and Ctrl+G
    }
...
  switch(dosignal){
  case SIGCUSTOM:
    cprintf("Ctrl-G is detected by xv6\n");
    customsig();
    break;
  case SIGINT:
    cprintf("Ctrl-C is detected by xv6\n");
    procsignal(SIGINT);
    break;
  ... // Similarly for SIGBG and SIGFG
  default:
    break;
  }
}
```

1.2 SIGINT, SIGBG, and SIGFG

The process structure (`struct proc`) was extended with:
`suspended` and `ever_suspended` flags

Console Handling (*console.c*)

```
case C('C'): dosignal = SIGINT;
case C('B'): dosignal = SIGBG;
case C('F'): dosignal = SIGFG;
```

Later processed through:

```
procsignal(SIGINT/SIGBG/SIGFG/...);
```

Signal Specific details

1) <u>SIGINT</u>
   Key Effects:

   - Terminates all non-system processes
   - Cleans up suspended processes via `cleanup_suspended()`
   - Immediate process table sweep with priority given to active processes

```
// proc.c void procsignal(int)
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state == UNUSED || p->pid <= 2) continue;
  p->killed = 1;
  if(p->state == SLEEPING) p->state = RUNNABLE;
}
cleanup_suspended();
```

2) <u>SIGBG</u>
   Behavior:

   - Suspends non-critical processes
   - Maintains process state for later resumption
   - Maintains another process state for eventually reaping
   - Suspended processes return a special code `'-2'` in the `wait()` for the shell to run.

   Below is the code for setting the process as suspended.

```c
// proc.c void procsignal(int)
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  // Skip unused processes and kernel/system processes (pid
     <= 2).
  if(p->state == UNUSED || p->pid <= 2)
    continue;

  // Skip processes with reserved names.
  if(strcmp(p->name, "sh") == 0 ||
     strcmp(p->name, "init") == 0 ||
     strcmp(p->name, "console") == 0)
    continue;

  // Suspend the process.
  p->suspended = 1;
  p->ever_suspended = 1;

  // Wakeup its parent process, if necessary.
  wakeup1(p->parent);
}
```

The following is how this process is handled in the wait function.

```c
// proc.c int wait(void)
...
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc || p->suspended)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        ...
        if(p->ever_suspended)
          return -2;

        return pid;
      }
    }
...
    if(all_done && havekids){
      // Return special code to indicate suspended children
         exist
      release(&ptable.lock);
      return -2;
    }
...
  }
}
```

3) <u>SIGFG</u>
Foreground restoration

```
// proc.c void procsignal(int)
  case SIGFG:
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != UNUSED && p->suspended){
        p->suspended = 0;
        if(p->state == SLEEPING && !p->killed){
          p->state = RUNNABLE;
        }
      }
    }
```

1.3 SIGCUSTOM

Registering the signal handler
For registering a signal handler, a system call is added as follows:

```
// syscall.h
#define SYS_signal 22

// syscall.c
extern int sys_signal(void);

... // (*syscalls)
[SYS_signal] sys_signal,
...

// usys.S
SYSCALL(signal)
```

To keep track of the signal handler for each process, added the handler to the process structure. Then defined a user-space wrapper function.

```
// proc.h
typedef void signal(sighandler_t handler);

struct proc{ ...
  sighandler_t handler;
  ... // some other useful fields
}

// user.h
void signal(sighandler_t handler);
```

When the syscall is made, the address pf the handler is saved to the handler attribute of the process.

```
// sysproc.c
int sys_signal(void){
  int addr;
  if(argint(0, &addr) < 0)
    return -1;

  myproc()->handler = (sighandler_t)addr;
  return 0;
}
```

When the console registers a `Ctrl+G` interrupt, it runs the function `void customsig(void)` in *proc.c*.

```
// proc.c void customsig(void)
if(p->handler){
// Save current user context so that sigret can restore it.
    p->saved_eip = p->tf->eip;
    p->saved_esp = p->tf->esp;
    p->in_handler = 1;

// Adjust the user stack: make room for one return address.
    uint *ustack = (uint*)p->tf->esp;
    ustack -= 1; // Reserve space for return address.
    ustack[0] = 0;

    p->tf->esp = (uint)ustack;
    p->tf->eip = (uint)p->handler;
}
```

The `p->saved_eip` stores the return address and similarly for the stack pointer(`p->saved_esp`). The `p->in_handler` flag is used for the return from the handler.

```
// trap.c void trap(struct trapframe *tf)

default:
// When the custom handler completes execution,
// the poiters are restored for the trapframe
  tf->eip = p->saved_eip;
  tf->esp = p->saved_esp;
  p->in_handler = 0;
  break;
```

2 xv6 Scheduler

2.1 Implementation for the `syscalls`

The `custom_fork` system call extends the standard fork operation with two parameters:

- `start_later_flag`: Controls whether the process starts immediately (`0`) or is created but not yet runnable (`1`)
- `exec_time`: Limits the maximum execution time of the process (`-1` for unlimited execution)

Key implementation details include:

- Adding a new process state `CREATED` for processes that are fully initialized but not yet runnable
- Implementing a `check_exec_time` function called from the timer interrupt handler to enforce execution time limits
- Handling the special case where `start_later_flag = 0` and `exec_time = -1` by directly calling the standard fork

The `scheduler_start` system call allows processes in the `CREATED` state to become `RUNNABLE`.

```
int sys_scheduler_start(void){
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == CREATED)
      p->state = RUNNABLE;
  }
  release(&ptable.lock);
  return 0;
}
```

2.2 Priority Boosting Scheduler

The priority boosting scheduler implements a dynamic priority system, calculating process priorities using:

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

Key implementation details include:

- Adding priority-related fields to the process control block
- Updating process priorities in both the scheduler and the timer interrupt handler
- Implementing highest-priority-first selection with PID as a tiebreaker

To update a process `p`'s priority, the following is done:

```
p->priority = p->init_priority - (ALPHA * p->cpu_ticks) + (BETA *
    p->wait_time);
```

2.3 Effects of α and β Parameters

1) Effects on Profiled Parameters

- Turnaround Time (TAT): Higher α/β ratio generally reduces average `TAT` by preventing excessive waiting.
- Waiting Time (WT): Higher β values directly reduce average `WT`.
- Response Time (RT): Higher β values improve `RT` by giving waiting processes a priority boost.
- Context Switches (#CS): Higher α and β values generally increase `#CS`.

2) Effects on CPU-bound Jobs

- Higher α values significantly reduce the priority of CPU-bound processes as they accumulate CPU time. This prevents CPU-bound jobs from monopolizing the processor.
- As α increases, CPU-bound jobs experience increased turnaround times, more frequent preemption, longer overall completion times and higher waiting times.
- CPU-bound jobs benefit less from β since they rarely wait voluntarily. This prevents complete starvation of CPU-intensive tasks.

3) Effects on I/O-bound Jobs

- I/O-bound processes accumulate less CPU time, so they're less affected by α.
- These processes benefit significantly from β as they frequently wait.
- As β increases, these jobs experience lower response times and smaller waiting-to-execution time ratios.

The current parameters (α = 1, β = 2) create a scheduler that provides good interactive performance while still allowing background CPU-intensive tasks to make reasonable progress.

Note: *Code snippets in the report might differ from what is actually implemented in the xv6 kernel. The snippets here are kept simpler for easier understanding.*