CSE 312 Operating Systems Spring 2020
Homework 3 Report

Ahmed Semih Özmekik
171044039

June 18, 2020

Abstract

Demonstration of small units and system calls and then the whole system will be shown. A better and extensive tests has been applied on the demo video. This is a quick demonstration report.

Demonstration Link: https://youtu.be/vdarl43GURU

1 Introduction

We will show that the requests stated in the homework are provided one by one. A demo video containing the unit test examples described here has been prepared. If you wish, you can watch the video from the given link above. It is almost impossible to demo micro kernels only with screenshots, so the demonstration link for 3 micro kernels that make up the main part of the assignment is included in the shared video.

2 Assembly Design

Some parts of the design will look like task 2. In this report, I will talk about the design choices I have changed especially in this new assignment by transferring most of the workload to the assembly. There were two main things expected from us: The first is to store the Process Table in a data structure in the assembly, and the second is to write the interrupt handler to the assembly routine.

First, let's talk about the data structures used in the assembly. We need data structures where we can both store the context of the processes and find out which process can be executed in order, that is, we can achieve robin round scheduling. We will use a simple array for the process table, and a queue for the interrupt handling, as I did in the previous assignment. They were defined as follows, in each kernel flavor:

```
## declarations
.data
process_limit: .word  1024
process_table: .space 4096  # contains reference to processes resources.
process_queue: .space 4096  # queue for robin round scheduling.

...
```

```
## mutators and accessors
enqueue:
addi    $t0, $s4, 1
div     $t0, $s1
mfhi    $s4             # back = (back + 1) % limit.
la      $t1, process_queue
sll     $t2, $s4, 2
add     $t1, $t1, $t2
sw      $a0, ($t1)        # process_table[back] = $a0.
addi    $s2, $s2, 1
jr     $ra

dequeue:
la      $t0, process_queue
sll     $t2, $s3, 2
add     $t0, $t2, $t0
lw      $v0, ($t0)        # $v0 = process_table[front].
li      $t3, 0xffff
sw      $t3, ($t0)
addi    $t1, $s3, 1
div     $t1, $s1
mfhi    $s3             # front = (front + 1) % limit.
addi    $s2, $s2, -1
jr     $ra

front:
la      $t0, process_table
sll     $t2, $s3, 2
add     $t0, $t2, $t0
lw      $v0, ($t0)        # $v0 = process_table[front].
jr $ra

is_empty:
addi    $v0, $s2, 0
jr              $ra

init_process_table:
la      $t0, process_table
```

```
la      $t6, process_queue
li      $t1, 0xffff
li      $t2, 0
loop_init_process_table:
sub     $t5, $t2, 4096
beqz    $t5, exit_init_process_table
add     $t3, $t0, $t2
add     $t7, $t6, $t2
sw      $t1, ($t3)      # process_table[i] = 0xffff
sw      $t1, ($t7)
addi    $t2, $t2, 4
j               loop_init_process_table
exit_init_process_table:
jr $ra
```

In this code snippet, we see both defined data structures. I haven't shared accessors and mutators for the process table yet. We will discuss them later. But here is an initialization procedure that initializes with a predetermined value for both process table and queue. And some mutation routines for queue.

Now, let's examine the interrupt handler and fork routines.

```
.globl fork
fork:
addi    $a0, $s2, 1       # $a0 = process index.
jal     enqueue
j       kernel_loop

.globl interrupt_handler   # $a1 = cont flag, $a0 = current process id.
interrupt_handler:
jal     is_empty
beqz    $v0, no_interrupt
beqz    $a1, not_cont
jal enqueue
not_cont:
jal     dequeue            # next process to execute from queue.
li      $a2, 1             # print.
move    $a1, $v0           # $a1 = next process id.
li      $v0, 22            # next process from queue to cpu.
syscall
no_interrupt:
move    $a1, $a0
li      $a2, 0             # no print.
li      $v0, 22
```

```
syscall
j kernel_loop
```

Both are routines that kernel will frequently call, when a system call comes in from another process.

Some part of the fork operation takes place in .cpp and some part in the assembly. Although the details of the processes are stored in the assembly in the process table, when the fork operation is performed, since the access to the context is only possible on the .cpp side, the copying of the context of the current process takes place on the .cpp side. But no matter what, data structures are stored in the assembly. Hence, the fork is bounced to the kernel process after the process calling the system call, and from there the process table in the assembly is accessed.

```
void readTable()
  {
    process_table = (mem_word *)mem_reference(table_addr);
    process_queue = (mem_word *)mem_reference(queue_addr);
  }

  Process *get(int idx) const
  {
    assert(idx <= index);
    return (Process *)mem_reference(process_table[idx]);
  }

  void set(Process *process) const
  {
    assert(process->getID() <= index);
    char *data = (char *)process;

    mem_addr addr = process_table[process->getID()];
    for (size_t i = 0; i < sizeof(Process); i++)
      set_mem_byte(addr + i, *(data + i));
  }
```

As seen in the code snippet above, it is necessary to move the kernel to the CPU and access its data segments in the assembly in order to access the information of the processes. With this constructor, and two accessors and mutators, set and get, we can access processes by such, form the .cpp side.

On the interrupt handler side, necessary actions are taken to achieve robin round scheduling (such as taking the next process from queue and putting the blocked process in queue). But since some of the context switch must still be on the .cpp side, a system call was designed for it.

3 Results

As the output of all kernels is too long, their descriptions and demonstrations are left to the demo. All demonstrations were made in detail in the demo video.