Chapter 6: Process Management in 539kernel

Introduction

The final result of this chapter is what I call version `T` of 539kernel which has a basic multitasking capability. The multitasking style that we are going to implement is time-sharing multitasking. Also, instead of depending on x86 features to implement multitasking in 539kernel, a software multitasking will be implemented. Our first step of this implementation is to setup a valid task-state segment, while 539kernel implements a software multitasking, a valid TSS is needed. As we have said earlier, it will not be needed in our current stage, but we will set it up anyway. Its need will show up when the kernel lets user-space software to run. After that, basic data structures for process table and process control block are implemented. These data structures and their usage will be as simple as possible since we don't have any mean for dynamic memory allocation, yet! After that, the scheduler can be implemented and system timer's interrupt can be used to enforce preemptive multitasking by calling the scheduler every period of time. The scheduler uses round-robin algorithm to choose the next process that will use the CPU time, and the context switch is performed after that. Finally, we are going to create a number of processes to make sure that everything works fine. But before that, we need to organize our code a little bit since it's going to be larger starting from this point. New two files should be created, `screen.c` and its header file `screen.h`. We move the printing functions that we have defined in the progenitor and their related global variables to `screen.c` and their prototypes should be in `screen.h`, so, we can `include` the latter in other C files when we need to use the printing functions. The following is the content of `screen.h`.

```
volatile unsigned char *video;

int nextTextPos;
int currLine;

void screen_init();
void print( char * );
void println();
void printi( int );
```

As you can see, a new function `screen_init` has been introduced while the others are same as the ones that we already wrote. The function `screen_init` is called by the kernel once it starts running and it initializes the values of the global variables `video`, `nextTextPos` and `currLine`. Its code is the following and it should be in `screen.c`, of course in the beginning of this file, `screen.h` should be included by using the line `#include "screen.h"`.

```
void screen_init()
```

```
{
    video = 0xB8000;
    nextTextPos = 0;
    currLine = 0;
}
```

Nothing new in here, just some organizing. Now, the prototypes and implementations of the functions `print`, `println` and `printi` should be removed from `main.c`. Furthermore, the global variables `video`, `nextTextPos` and `currLine` should also be removed from `main.c`. Now, the file `screen.h` should be included in `main.c` and in the beginning of the function `kernel_main` the function `screen_init` should be called.

Initializing the Task-State Segment

In our current case this step, as I have mentioned earlier, is optional. The TSS will be handy when a switch is performed between a user-space code which runs in privilege level `3` and the kernel which runs in privilege level `0`. However, since we are on the topic of process management, then the best time to deal with TSS is now.

Setting TSS up is too simple. First we know that the TSS itself is a region in the memory [1]. So, let's allocate this region of memory. The following should be added at end of `starter.asm`. A label named `tss` is defined, and inside this region of memory, which its address is represented by the label `tss`, we put a double-word of `0`, recall that a word is `2` bytes while a double-word is `4` bytes. So, our TSS contains nothing but a bunch of zeros.

```
tss:
    dd 0
```

As you may recall, each TSS needs an entry in the GDT table after that its segment selector can be loaded into the task register. Then the processor is going to think that there is one process (one TSS entry in GDT) in the environment and it is the current process (The segment selector of this TSS is loaded into task register). Now, let's define the TSS entry in our GDT table. In the file `gdt.asm` we add the following entry under the label `gdt`. You should not forget to modify the size of GDT under the label `gdt_size_in_bytes` under `gdtr` since the sixth entry has been added to the table.

```
tss_descriptor: dw tss + 3, tss, 0x8900, 0x0000
```

Now, let's go back to `starter.asm` in order to load TSS' segment selector into the task register. In `start` routine and below the line `call setup_interrupts` we add the line `call load_task_register` which calls a new routine named

[1] Since it is a segment.

`load_task_register` that loads the task register with the proper value. The following is the code of this routine.

```
load_task_register:
    mov ax, 40d
    ltr ax

    ret
```

As you can see, its too simple. The index of TSS descriptor in GDT is `40 = (entry 6 * 8 bytes) - 8 (since indexing starts from 0)`. So, the value `40` is moved to the register `ax` which will be used by the instruction `ltr` to load the value `40` into the task register.