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.

The Data Structures of Processes

When we develop a user-space software and we don't know the size of the data that this software is going to store while it's running, we usually use dynamic memory allocation, that is, regions of memory are allocated at run-time in case we need to store more data that we didn't know that it will be needed to be stored. We have encountered the run-time stack previously, and you may recall that this region of memory is dedicated for local variables, parameters and some information that make function invocation possible. The other region of a process is known as run-time heap, which is dedicated for the data that we decided to store in memory while the software is running. In C, for instance, the function `malloc` is used to allocate bytes from the run-time heap and maintains information about free and used space of the heap so in the next use of this function the allocation algorithm can decide which region should be allocated based on the required bytes to allocate. This part that allocates memory dynamically and manages the related stuff is known as *memory allocator* and one of well-known allocators is Doug Lea's memory allocator. For programming languages that run the program by using a virtual machine, like Java and C#, or by using interpreters like PHP and Python, they usually provides its users an automatic dynamic memory allocation instead of the manual memory allocation which is used by languages such as C. However, the virtual machine or the interpreter needs to allocate dynamic memory by itself and frees the region of the heap that are not used any more through a mechanism known as *garbage collection*. For those who don't know, in static memory allocation, the size of data and where will it be stored in the memory are known in compiling time, global variables and local variables are examples of objects that we use static memory allocation for them. In dynamic memory allocation, we cannot decide in compiling time the size of the data or whether it will be stored in the first place, these important information will only known while the software is running, that is, in run-time. Due to that, we need to use dynamic memory allocation for them since this type of allocation doesn't require these information in the compiling time.

Processes table is an example of data structures (objects) that we can't know its size in compile time and this information can be only decided while the kernel is running. Take your current operating system as an example, you can run any number of processes [2], your system may run just two processes for example, and you can run more and more without the need of recompiling the kernel that you use. When a new process is created at run-time, an entry for this process in the processes tables is needed, a number of bytes are allocated by the memory allocator to be used to store the information of this process. When we are done with this process, the memory region that is used to stored its information is marked as free space so it can be used to store something else in the future, for example, the entry of another process.

In our current situation, we don't have any means of dynamic memory allocation in 539kernel, this is a topic to come when we start discussing memory management. Due to that, our current implementations of processes table and process control block are going to use static memory allocation through global variables. That of course, restrict us from creating a new process on-the-fly, that is, at run-time. But our current goal is to implement a basic multitasking that will be extended later to be similar to the ones that available in modern operating systems. To start our implementation, we need to create new two files, `process.c` and its header file `process.h`. Any function or data structure that is related to processes belong to these file.

Process Control Block

A process control block (PCB) is an entry in the processes table, it stores that information that is related to a specific process. The context and the state of the process are stored in this entry, we already have discussed the concepts of process' context and state. In 539kernel, currently, there are two possible states of a process, either a process is *running* or *ready*. When a context switch is needed to be performed, the context of the current process, that it will be suspended, should be stored on its PCB. Currently, the context of the process in 539kernel is represented by the values which were stored in the processor's register before interrupting the process. Each process in 539kernel, as in most modern kernels, has a unique identifier known as *process id* or PID for short, this identifier is also stored in the PCB of the process. Now, let's define the general structure of PCB and its components in 539kernel. These definitions should reside in `process.h`.

```
typedef enum process_state { READY, RUNNING } process_state_t;

typedef struct process_context
{
    int eax, ecx, edx, ebx, esp, ebp, esi, edi, eip;
```

[2] To some limit of course.

```
} process_context_t;

typedef struct process
{
    int pid;
    process_context_t context;
    process_state_t state;
    int *base_address;
} process_t;
```

As you can see, we start by a type known as `process_state_t`, any variable that has this type may have two possible values, `READY` or `RUNNING`, they are the two possible states of a process and this type will be used for the state field in PCB definition.

Next the type `process_context_t` is defined. It represents the context of a process in 539kernel and you can see it is a C structure that intended to store a snapshot of x86 registers that can be used by a process.

Finally, the type `process_t` is defined which represents a process control block, that is, an entry in the processes table. A variable of type `process_t` represents one process in 539kernel environment. Each process has a `pid` which is its unique identifier. A `context` which is the snapshot of the environment before suspending the process. A `state` which indicates whether a process is `READY` to run or currently `RUNNING`. Any finally, a `base_address` which is the memory address of the process' code starting point [3], that is, when the kernel intend to run a process for the first time, it should jump to the `base_address`, in other words, set `EIP` to `base_address`.

Processes Table

In the current case, as we mentioned earlier, we are going to depend on static memory allocation since we don't have any way to employ dynamic memory allocation. Due to that, our processes table will be too simple, it is an array of type `process_t`. Usually, more advanced data structure is used for the processes list based on the requirements which are decided by the kernelist, *linked list data structure* is a well-known choice, but we can't implement that now due to the lack of dynamic memory allocation in 539kernel. The following definition should be reside in `process.h`. Currently, the maximum size of 539kernel processes table is `15` processes, feel free to increase it but don't forget, it will, still, be a static size.

```
process_t *processes[ 15 ];
```

[3] Think of `main()` in C.