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.

Process Creation

Now, we are ready to write the function that creates a new process in 539kernel. Before getting started in implementing the required functions, we need to define their prototypes and some auxiliary global variables in `process.h`.

```
int processes_count, curr_pid;

void process_init();
void process_create( int *, process_t * );
```

The first global variable `processes_count` represents that current number of processes in the environment, this value will become handy when we write that code of the scheduler which uses round-robin algorithm, simply, whenever a process is created in 539kernel, the value of this variable is increased. The global variable `curr_pid`, contains the next available process identifier that can be used for the next process that will be created. The current value of this variable is used for when creating a new process and its value is increased by one after that.

The function `process_init` is called when the kernel starts, and it initializes the process management subsystem, currently, by just initializing the two global variables that we mentioned. The function `process_create` is the one that create a new process in 539kernel, that is, it is equivalent to `fork` in Unix systems. As you can see, it takes two parameters, the first one is a pointer to the base address of the process, that is, the starting point of the process' code. The second parameter is a pointer to the process control block, as we have said, currently, we use static memory allocation, therefore, each new PCB will be either stored in the as a local or global variables, so, for now, the caller is responsible for allocating a static memory for the PCB and passing its memory address in the second parameter. In the normal situation, the memory of a PCB is allocated dynamically by the creation function itself, but that's a story for another chapter. The following is the content of `process.c` as we have described.

```
#include "process.h"

void process_init()
{
    processes_count = 0;
    curr_pid = 0;
}

void process_create( int *base_address, process_t *process )
{
    process->pid = curr_pid++;

    process->context.eax = 0;
```

```
    process->context.ecx = 0;
    process->context.edx = 0;
    process->context.ebx = 0;
    process->context.esp = 0;
    process->context.ebp = 0;
    process->context.esi = 0;
    process->context.edi = 0;
    process->context.eip = base_address;

    process->state = READY;
    process->base_address = base_address;

    processes[ process->pid ] = process;

    processes_count++;
}
```

As you can see, `process_init` just set the initial values to the global variables. In `process_create`, a new process identifier is assigned to the new process. Then the context is initialized, this structure will be used later in context switching, either by copying the values from the processor to the structure of vice versa. Since the new process has not been run yet, hence, it didn't set any value to the registers, then we initialize all general purpose registers with `0`, later on, when this process runs and the scheduler decides to suspend it, the values that this process wrote on the real registers will be copied in here. The structure field of program counter `EIP` is initialized with the starting point of the process' code, in this way we can make sure that when the scheduler decides to run this process, it loads the correct value to the register `EIP` via context switching process. After that, the state of process is set as `READY` to run, the base address is stored, to PCB is added to the processes list, which is a simple array and finally the number of processes in the system is increased by one. That' all we need for now to implement multitasking, in real cases, there will be usually more process states such as *waiting*, the data structures are allocated dynamically to make it possible to create virtually any number of processes, the PCB may contains more fields and more functions to manipulate processes table (e.g. delete process) are implemented. However, our current implementation, tough too simple, is enough as a working foundation. Now, in `main.c`, the header file `process.h` is needed to be included, and the function `process_init` should be called in the beginning of the kernel, after the line `screen_init();`.

The Scheduler

Right now, we have all needed components to implement the core of multitasking, that is, the scheduler. As mentioned multiple times before, round-robin algorithm is used for 539kernel's scheduler. Let's present two definitions to make our next

discussion more clear. The term *current process* means that process that is using the processor now, at some point of time, the system timer emits an interrupt which suspend the current process and calls the kernel to handle the interrupt [4], at this point of time, we keep the same name for the suspended process, we sill call it the current process. By using some algorithm, the scheduler chooses the *next process*, that is, the process that will run after the scheduler finishes its work and the kernel returns the processor to the processes. After making this choices of the next process by the scheduler, performing the context switching and jumping to the process code, this chosen process will be the current process instead of the suspended one, and it will be the current process until the next run of the scheduler and so on. Now, we are ready to implement the scheduler, let's create a new file `scheduler.c` and its header file `scheduler.h` for the new code. The following is the content of the header file.

```
#include "process.h"

int next_sch_pid, curr_sch_pid;

process_t *next_process;

void scheduler_init();
process_t *get_next_process();
void scheduler( int, int, int, int, int, int, int, int, int );
void run_next_process();
```

First, `process.h` is included since we need to use the structure `process_t` in the code of the scheduler. Then two global variables are defined, the global variable `next_sch_pid` stores the PID of the next process that will run after next system timer interrupt, while `curr_sch_pid` stores the PID of the current process. The global variable `next_process` stores a reference to the PCB of the next process, this variable will be useful when we want to move the control of the processor from the kernel to the next process which is the job of the function `run_next_process`. The function `scheduler_init` sets the initial values of the global variables, and similar to `process_init`, it will be called when the kernel starts. The core function is `scheduler` which represents 539kernel's scheduler, this function will be called when the system timer emits its interrupt. It chooses the next process to run with the help of the function `get_next_process`, performs context switching by copying the context of the current process from the registers to the memory and copying the context of the next process from the memory to the registers. Finally, it returns to give `run_next_process` to be called and jump the the next process' code. In `scheduler.c`, the file `scheduler.h` should be included to make sure that everything works fine. The following is the implementation of `scheduler_init`.

```
void scheduler_init()
{
```

[4] In this case the kernel is going to call the scheduler.

```
    next_sch_pid = 0;
    curr_sch_pid = 0;
}
```

It's too simple function that initializes the values of the global variables by setting the PID `0` to both of them, so the first process that will be scheduled by 539kernel is the process with PID `0`. Next, is the definition of `get_next_process` which implements round robin algorithm, it selects which process to run next and returns a pointer to the PCB of this process.

```
process_t *get_next_process()
{
    process_t *next_process = processes[ next_sch_pid ];

    curr_sch_pid = next_sch_pid;
    next_sch_pid++;
    next_sch_pid = next_sch_pid % processes_count;

    return next_process;
}
```

Too simple, right! [5] If you haven't encountered the symbol `%` previously, it represents an operation called *modulo* which gives the remainder of division operation, for example, `4 % 2 = 0` because the reminder of dividing `4` on `2` is `0`, but `5 % 2 = 1` because `5 / 2 = 2` and remainder is `1`, so, `2 * 2 = 4 + 1 (the remainder) = 5`, in modulo operation, any value `n` that has the same position of `2` in the previous two examples is known as *modulus*. For instance, the modulus in `5 % 3` is `3` and the modulus in `9 % 10` is `10` and so on. In some other places, the symbol `mod` is used to represent modulo operation instead of `%`. The interesting thing about modulo that its result value is always between the range `0` and `n - 1` given that `n` is the modulus. For example, let the modulus is `2`, and we perform the following modulo operation `x % 2` where `x` can be any number, the possible result values of this operation are `0` or `1`. Using this example with different values of `x` gives us the following results, `0 % 2 = 0`, `1 % 2 = 1`, `2 % 2 = 0`, `3 % 2 = 1`, `4 % 2 = 0`, `5 % 2 = 1`, `6 % 2 = 0` and do on to infinity! As you can see, this operation gives as a cycle that starts from `0` and ends at some value that is related to the modulus and starts all over again with the same cycle given an ordered sequence of values for `x`, sometimes a clock is used as metaphor to describe the modulo operation. However, in mathematics a topic known as *modular arithmetic* is dedicated to the modulo operation. You may noticed that modulo operation can be handy to implement round-robin algorithm.

Let's go back to the function `get_next_process` which chooses the next process to run in a round-robin fashion. As you can see, it assumes that the PID of the next process can be found directly in `next_sch_pid`. By using this assumption

[5] Could be simpler, but the readability is more important here.

it fetches the PCB of this process to return it later to the caller. After that, the value of `curr_sch_pid` is updated to indicate that, right now, the current process is the one that we just selected to run next. The next two lines are the core of the operation of choosing the next process to run, it prepares which process will run when next system timer interrupt occurs, assume that the total number of processes in the system is `4`, that is, the value of `processes_count` is `4`, and assume that the process that will run in this system timer interrupt has the PID `3`, that is `next_sch_pid = 3`, PIDs in 539kernel start from `0`, that means there is no process with PID `4` and process `3` is the last one. In line `next_sch_pid++` the value of the variable will be `4`, and as we mentioned, the last process is `3` and there is no such process `4`, that means we should start over the list of processes and runs process `0` in the next cycle, we can do that simply by using modulo on the new value of `next_sch_pid` with the modulus `4` which is the number of processes in the system `process_count`, so, `next_sch_pid = 4 % 4 = 0`. In the next cycle, process `0` will be chosen to run, the value of `next_sch_pid` will be updated to `1` and since it is lesser than `process_count` it will be kept for the next cycle. After that, process `1` will run and the next to run will be `2`. Then process `2` will run and next to run is `3`. Finally, the same situation that we started our explanation with occurs again and process `0` is chosen to run next. The following is the code of the function `scheduler`.

```
void scheduler( int eip, int edi, int esi, int ebp, int esp, int ebx, int edx, int ecx, int
{
    process_t *curr_process;

    // ... //

    // PART 1

    curr_process = processes[ curr_sch_pid ];
    next_process = get_next_process();

    // ... //

    // PART 2

    if ( curr_process->state == RUNNING )
    {
        curr_process->context.eax = eax;
        curr_process->context.ecx = ecx;
        curr_process->context.edx = edx;
        curr_process->context.ebx = ebx;
        curr_process->context.esp = esp;
        curr_process->context.ebp = ebp;
        curr_process->context.esi = esi;
        curr_process->context.edi = edi;
```

```
        curr_process->context.eip = eip;
    }

    curr_process->state = READY;

    // ... //

    // PART 3

    asm( "  mov %0, %%eax;  \
            mov %0, %%ecx;  \
            mov %0, %%edx;  \
            mov %0, %%ebx;  \
            mov %0, %%esi;  \
            mov %0, %%edi;"
            : : "r" ( next_process->context.eax ), "r" ( next_process->context.ecx ), "r" (
                "r" ( next_process->context.esi ), "r" ( next_process->context.edi ) );

    next_process->state = RUNNING;
}
```

I've commented the code to divided into three parts for the sake of simplicity in our discussion. The first part is too simple, the variable `curr_process` is assigned to a reference to the current process which has been suspended due to the system timer interrupt, this will become handy in part 2 of scheduler's code, we get the reference to the current process before calling the function `get_next_process` because, as you know, this function changes the variable of current process' PID (`curr_sch_pid`) from the suspended one to the next one. After that, the function `get_next_process` is called to the PCB of the process that will be run this time, that is, the next process.

As you can see, `scheduler` receives nine parameters, each one of them has a name same as one of the processors registers. We can tell from these parameters that the function `scheduler` receives the context of the current process before being suspended due to system timer's interrupt. For example, assume that process `0` was running, after the quantum finished the scheduler has been called, which decides that process `1` should run next. In this case, the parameters that has been passed to the scheduler represent the context of process `0`, that is, the value of the parameter `eax` will be same as the value of the register `eax` that process `0` set at some point of time before being suspended.

In part `2` of scheduler's code, the context of the suspended process, which `curr_process` represents it right now, is copied from the processor into its own PCB, as we mentioned, the values of the parameters represent the context of the suspended process as it was in the processor before suspending the process. How did we get these values and passed them as parameters to `scheduler`? This will be discussed later. Storing current process' context into its PCB is

simple as you can see, we just store the passed values in the fields of the current process structure. These values will be used later when we decide to run the same process. Also, we need to make sure that the current process is really running by checking its `state` before copying the context from the processor to the PCB. At the end, the `state` of the current process is switched from `RUNNING` to `READY`.

Part `3` performs the opposite of part `2`, it uses the PCB of the next process to retrieve its context before the last suspension if this process, then this context will be copied to the registers of the processor. Of course, not all of them are being copied to the processor, for example, the program counter `EIP` cannot be written to directly, so we will see later how to deal with it. Also, the registers that are related to the stack, `ESP` and `EBP` were skipped in purpose. They will be handled later on we we start discussing memory management. As a last step, the `state` of the next process is changed from `READY` to `RUNNING`. The following is the code of `run_next_process` which is last function remain in `scheduler.c`.

```
void run_next_process()
{
    asm( "  sti;             \
            jmp *%0" : : "r" ( next_process->context.eip ) );
}
```

It is a simple function that executes two assembly instructions. First it enables the interrupts via the instruction `sti`, then it jumps to the memory address which is stored in the `EIP` of next process' PCB. The purpose of this function will be discussed after a short time.

To make everything runs properly, `scheduler.h` need to be included in `main.c`, note that, when we include `scheduler.h`, the line which includes `process.h` should be remove since `scheduler.h` includes its by itself. After that, the function `scheduler_init` should be called when initializing the kernel, say after the line which calls `process_init`.

Calling the Scheduler

So, "how the scheduler is being called" you may ask. The answer to this question has been mentioned multiple times before. In 539kernel, when the system timer decides that it is the time to interrupt the processor, the interrupt `32` is being fired. This is where the scheduler being called, in each period of time it will be called to schedule another process and gives it CPU time. In this part, we are going to write a special interrupt handler for interrupt `32` that calls 539kernel's scheduler. First we need to add the following lines in the beginning of `starter.asm` [6] after `extern interrupt_handler`.

[6] I'm about to regret that I called this part of the kernel the starter! obviously it's more than that!

```
extern scheduler
extern run_next_process
```

As you may guessed, the purpose of these two lines is to make the functions `scheduler` and `run_next_process` usable by the assembly code of `starter.asm`.