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 switching is performed after that. Finally, we are going to create a number of processes to make sure that everything works fine.

Before getting started in the plan that has been just described, 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`.

```c
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` will be called by the kernel once it starts running, the function 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

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 defining this entry, the TSS's 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 get 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 `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
```

[1] Since it is a segment.

```
    ret
```

As you can see, it's 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 (inside run-time heap) 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 provide their users an automatic dynamic memory allocation instead of the manual memory allocation which is used by languages such as C, that is, the programmer of these languages don't need to explicitly call a function (such as `malloc`) to allocate memory in the heap at run-time instead the virtual machine or the interpreter allocates dynamic memory by itself and frees the region of the heap that are not used anymore 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 be 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], maybe your system is running just two processes now but 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 store 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 topic will be covered 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, restricts 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 state and context of the process are examples of these information. In 539kernel, there are two possible states for a process, either a process is *running* or *ready*. When a context switch is needed to be performed, the context of the currently running process, which will be suspended, should be stored on its PCB. Based on our previous discussions, the context of the process in 539kernel consists the values which were stored in the processor's registers before suspending 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`.

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

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

[2] To some limit of course.

```
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 it right now now due to the lack of dynamic memory allocation in 539kernel. The following definition should 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 the current number of processes in the environment, this value will become handy when we write the code of the scheduler which uses round-robin algorithm, simply, whenever a process is created in 539kernel, the value of this variable is increased and since deleting a process will not be implemented, for the sake of simplicity, the value of this variable will not be decreased anywhere in the current code of 539kernel.

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 when creating a new process and its value is increased by one completing the creation.

The function `process_init` is called when the kernel starts, and it initializes the process management subsystem by just initializing the two global variables that we mentioned.

The function `process_create` is the one that creates 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 memory 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++;
}
```

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 or 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`.

After initializing the context, the state of the process is set as `READY` to run and the base address of the process is stored in a separate field. Then, the freshly-created PCB is added to the processes list 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 the 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 term for the process which was running right before calling the kernel to handle the interrupt, we 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 choosing the next process, 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.

```c
#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 three 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, same as `process_init`, it will be called when the kernel starts.

The core function is `scheduler` which represents 539kernel's scheduler, this

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

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 and `run_next_process` is called in order to 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()
{
    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 returns the PCB of the process that should run right now and prepare the value of `next_sch_pid` for the next context switching by using round-robin policy.

```
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, `5 = ( 2 * 2 ) + 1 (the remainder)`.

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 be `2`, and we perform the following modulo operation `x % 2` where `x` can be

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

any number, the possible result values of this operation are only `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 so on to infinity!

As you can see, modulo gives us 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 get 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 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 the current 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`.

```c
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 divide it 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[6]. After that, the function `get_next_process` is called to obtain the PCB of the process that will run this time, that is, the next process.

[6] And that's why the global variables are considered evil.

As you can see, scheduler receives nine parameters, each one of them has a name same as one of the processor's 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 have 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. How did we get these values and passed them as parameters to scheduler? This will be discussed later.

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 by using the passed parameter. 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` [7] after `extern interrupt_handler`.

```
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`. Now, we can start to implement the code of interrupt 32's handler which calls the scheduler with the needed parameters. In the file `idt.asm` the old code of the routine `isr_32` should be changed to the following.

```
isr_32:
    ; Part 1

    cli ; Step 1

    pusha ; Step 2

    ; Step 3
    mov eax, [esp + 32]
    push eax

    call scheduler ; Step 4

    ; ... ;

    ; Part 2

    ; Step 5
    mov al, 0x20
    out 0x20, al

    ; Step 6
    add esp, 40d
    push run_next_process
```

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

```
    iret ; Step 7
```

There are two major parts in this code, the first one is the code which is executed before calling the scheduler, that is, the one before the line `call scheduler`. The second one is the code which is executed after the scheduler returns. The first step of part one is to disable the interrupts via the instruction `cli`, when we are handling an interrupt, it is better to not receive any other interrupt. If we don't disable interrupts here, while handling a system timer interrupt, another system timer interrupt can occur even before calling the scheduler in the first time, you may imagine the mess that can be as a result of that. Before explaining steps two and three of this routine, we need to answer a vital question: When this interrupt handler is called, what is the context of the processor? In other words, the context of which code of the system is loaded into memory? The answer is, the suspended process, that is, the process that was running before the system timer emitted the interrupt. That means all values that were stored by the suspended process on the general purpose registers will be there when `isr_32` starts executing and we can be sure that the processor will not change any of these values during suspending the process and calling the handler of the interrupt. What assures us that the context of suspended process will be there on the processor when `isr_32` starts is the fact that we have defined all ISRs gate descriptors as interrupt gates in the IDT table, if we have defined them as task gates, the context of the suspended process will not be available directly on processor's registers. Defining an ISR descriptor as an interrupt gate makes the processor to call this ISR as a normal routine by following the calling convention. It's important to remember that when we discuss obtaining the value of `eip` of the suspended process later on in this section.

By knowing that the context of suspended process is reachable via the registers (e.g `eax`) we can store a copy of them in the stack in order to be used later by the scheduler to copy the context of the suspended process to the memory as we have seen, also, pushing them into stack gives as two more benefits. First we can start to use the registers as we like without the fear of losing the suspended process context, it is stored in the stack and we can refer to it anytime we need it. Second, according to C calling convention that we have discussed in chapter these pushed values can be considered as parameters for a function that will be called and that's exactly how we pass the context of suspended process to the function `scheduler` as parameters, simply by pushing the values of general purpose registers into the stack. Now, instead of writing `8` push instructions to push these values into the stack, for example `push eax`, there is an x86 instruction named `pusha` which pushes the current values of all general purpose registers into the stack, that exactly what happens in the second step of `isr_32` in order to send them as parameters to the function `scheduler`. The reverse operation of `pusha` can be performed by the instruction `popa`, that is, the values on the stack will be loaded into the registers. The instruction pushes the values of the registers in the following order: `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, and `edi`. Based on C calling convention they will be received as parameters in the reversed order [8], so, the parameter that contains the value of edi will be before esi and so on, you can see that in an obvious way in the parameters of the function scheduler. The only missing piece now is the value of the instruction pointer eip, which is handled by the third step of isr_32, as you know, it is too important to store the last eip of the suspended process, we need to know here did the execution of the suspended process code stop so we can resume its work later from the same point, and this information can be known through eip. Not like the general purpose registers, the value of eip will not be pushed into the stack by the instruction pusha, furthermore, the current eip is by no means a pointer to where the suspended process stopped, as you know, the current value of eip is a pointer to the current instruction which is being executed, that is, one of isr_32 instructions. So, the question is, where can we find the value of eip which was there just before the suspended process has been suspended? The answer again can be found in the calling convention. First, we need to mention

Let's assume that a process named A was running and a system timer interrupt occurred which caused process A to be suspended and isr_32 to start, as we have mentioned earlier, isr_32 will be called as a normal routine and the calling convention will be followed by the processor. Figure shows the stack at the time after executing pusha in isr_32. As you can see, the context of process A is on the stack, for example, to reach the value of esi which was stored right before the process A has been suspended, we can do that by refer to the memory address esp + 4, since the current esp stores the memory address of the top of the stack, the size of the value of edi and all other registers is 4 bytes and that the value of esi next to the top of the stack. The same technique can be used with any value in stack. As you may have noticed in the figure, that the return address to where process A suspended is stored in the stack, and that's due to the calling convention which requires the return address of the caller to be stored in the stack so we can return to it, as you can see, here, the process A was considered as the caller and isr_32 as the callee. So, to obtain the value of process A's return address, we simply can do that by reading the value in esp + 32, and that exactly what we have done in the third step of isr_32 code, we first read this value and then we push it into the stack so the function scheduler can receive it as the first parameter.

The fourth and fifth steps are simple, in the fourth step we call the function scheduler which we have already discussed, after the function scheduler returns, we need to tell PIC that we finished the handling of and IRQ by sending end of interrupt command to the PIC and that's we is performed in the fifth step, we have already discussed the of send end of interrupt command to PIC in chapter .

The final thing to do after choosing the next process and performing the context switch is to gives a CPU time for the code of the next process. This is usually performed by jumping to the memory address in which the selected process where suspended. There are multiple ways to do that, the way which we have used

[8] That is, the first pushed values will be the last to receive.

in 539kernel is to exploit the calling convention, again. As we have mentioned before, the return address to the caller is stored in the stack, in our previous example, the return address to process A was stored in the stack right before the values of process A context which have been pushed by the instruction pusha. When a routine returns by using the instruction ret or iret, this address will be jumped to by the processor, we exploit this fact to make the next process runs after isr_32 finishes instead of process A, this is too simple to be done, the return address of process A should be removed from the stack and the resume point of the next process is pushed in its position, and that's what we do in the sixth step of isr_32, we remove all values that we have pushed on the stack while running isr_32, this is performed by just adding 40 to the current value of esp, we have already discussed this method of removing values from the stack , why adding 40? You may ask. The number of values that have been pushed by the instruction pusha is 8 values, each one of them of size 4 bytes (32-bit), that means the total size of them is 4 * 8 = 32. Also, we have pushed the value of eip which also has the size of 4 bytes, so, until now the total size of pushed items in isr_32 is 32 + 4 = 36 and these are all what we have pushed in purpose, we also need to remove the return address which has been pushed into the stack before calling isr_32, the size of memory addresses in 32-bit architecture has the size 4 bytes (32-bit), that means 36 + 4 = 40 bytes should be removed from the stack to ensure that we remove all pushed values and the return address. After that, we simply push the memory address of the function run_next_process. In the seventh step, the routine isr_32 returns indicating that handling an interrupt has been completed, but instead of returning to the suspended code before calling the interrupt handler, the code of the function run_next_process will be called, which is, as we have seen, enables the interrupts again and jumps to the resume point of the next process. In this way, we have got a basic multitasking!

Running Processes

In our current environment, we will not be able to test our process management by using the normal ways, I mean, we can't run a user-space software to check if its process has been created and being scheduled or not. Instead, we are going to create a number of processes by creating their PCBs via process_create function, and their code will be defined as functions in our kernel, the memory address of these functions will be considered as the starting point of the process. Our goal of doing that is just to test that our code of process management is running well. All code of this section will be in main.c unless otherwise is mentioned. First, we define prototypes for four functions, each one of them represents a separate process, imaging them as a normal use-space software. These prototypes should be defined before kernel_main.

```
void processA();
void processB();
void processC();
void processD();
```

Inside `kernel_main`, we define four local variables. Each of them represents the PCB of one process.

```
    process_t p1, p2, p3, p4;
```

Before the infinite loop of `kernel_main` we create the four processes in the system by using the function `process_create` as the following.

```
    process_create( &processA, &p1 );
    process_create( &processB, &p2 );
    process_create( &processC, &p3 );
    process_create( &processD, &p4 );
```

The code of the processes is the following.

```
void processA()
{
    print( "Process A," );

    while ( 1 )
        asm( "mov $5390, %eax" );
}

void processB()
{
    print( "Process B," );

    while ( 1 )
        asm( "mov $5391, %eax" );
}

void processC()
{
    print( "Process C," );

    while ( 1 )
        asm( "mov $5392, %eax" );
}

void processD()
{
    print( "Process D," );

    while ( 1 )
        asm( "mov $5393, %eax" );
}
```

Each process starts by printing its name, then, an infinite loop starts which keeps setting a specific value in the register `eax`. To check whether multitasking is working fine, we can add the following lines the beginning of the function `scheduler` in `scheduler.c`.

```
    print( " EAX = " );
    printi( eax );
```

Each time the scheduler starts, it prints the value of `eax` of the suspended process. When we run the kernel, each process is going to start by printing its name, and before a process starts executing the value of `eax` of the previous process will be shown. Therefore, you will see a bunch of `EAX = 5390`, `EAX = 5391`, `EAX = 5392` and `EAX = 5393` keep showing on the screen which indicates that the process, `A` for example in case `EAX = 5390` is printed, was running and it has been suspended now to run the next one and so on.

Finishing up Version `T`

And we have got version `T` of 539kernel which provides us a basic process management subsystem. The last piece to be presented is the makefile to compile the whole code.

```
ASM = nasm
CC = gcc
BOOTSTRAP_FILE = bootstrap.asm
INIT_KERNEL_FILES = starter.asm
KERNEL_FILES = main.c
KERNEL_FLAGS = -Wall -m32 -c -ffreestanding -fno-asynchronous-unwind-tables -fno-pie
KERNEL_OBJECT = -o kernel.elf

build: $(BOOTSTRAP_FILE) $(KERNEL_FILE)
    $(ASM) -f bin $(BOOTSTRAP_FILE) -o bootstrap.o
    $(ASM) -f elf32 $(INIT_KERNEL_FILES) -o starter.o
    $(CC) $(KERNEL_FLAGS) $(KERNEL_FILES) $(KERNEL_OBJECT)
    $(CC) $(KERNEL_FLAGS) screen.c -o screen.elf
    $(CC) $(KERNEL_FLAGS) process.c -o process.elf
    $(CC) $(KERNEL_FLAGS) scheduler.c -o scheduler.elf
    ld -melf_i386 -Tlinker.ld starter.o kernel.elf screen.elf process.elf scheduler.elf -o 5
    objcopy -O binary 539kernel.elf 539kernel.bin
    dd if=bootstrap.o of=kernel.img
    dd seek=1 conv=sync if=539kernel.bin of=kernel.img bs=512 count=8
    dd seek=9 conv=sync if=/dev/zero of=kernel.img bs=512 count=2046
    qemu-system-x86_64 -s kernel.img
```

Nothing new in here but compiling the new C files that we have added to 539kernel.