CS-345 Operating Systems

Fall Semester 2025

Exercise 3

Tutorial: 13/11/2025
Deadline: 27/11/2025

Introduction

The Linux operating system kernel contains a Scheduler which decides which will be the next process to be executed on the computer’s processor (CPU). The scheduler makes decisions according to the Scheduling Policy, which helps in the efficient use of the processor.

One process scheduling policy is the following: each process declares to the scheduler what its deadline is (by what time it must have been executed) and also how much time it needs to execute (estimated runtime). For this exercise, assume that the deadline is measured in seconds, while the estimated runtime is measured in milliseconds. The scheduler runs the process that has the smallest (Current Time - D) / (Remaining execution time). If a process has (Current Time - D) <1, then it is killed.

At any given time, multiple processes may need to be executed. In such cases, the scheduler prioritizes based on these two fields of each process. More information on how the priority of each process will be calculated will be given to you in the next exercise (4th).

n this exercise, you will have to implement two new system calls, which you will also use in the next exercise of the course. More specifically, you will need to add two new system calls to the Linux operating system kernel: “**set_proc_info**” και “**get_proc_info**”. Then you will test them, using the QEMU emulator, by compiling the Linux kernel with your changes and running Linux with the new kernel. Finally, you are asked to write and run a user-level demo program that will use the new system calls. This program will run on the Linux operating system that you will load with QEMU. The goal of the exercise is to familiarize you with the Linux kernel source code, the way system calls are defined and implemented in Linux, kernel compilation, and the use of an emulator.

Implementation

You are asked to add the following functionality to the Linux kernel:

- Inside the Linux kernel, each process is represented as a task_struct. This struct contains all the necessary information for a process (e.g., process id, parent process, etc.). You must add two new fields to this struct:

  ```
  int deadline      /* the process's deadline, in seconds from now */.
  int est_runtime   /* the expected execution time, in milliseconds */
  ```

  The values of these fields will be changed by the system call `set_proc_info` system call, while they can be returned to a program via the `get_proc_info` `system call`.

- `set_proc_info(int deadline, int est_runtime)`
  This system call changes the values of the corresponding fields in the task_struct of the specific process that made the system call. You must check that the values of the given arguments are valid and also that the est_runtime can be met within the deadline. The system call must return 0 if the process information has been stored successfully, while it will return `EINVAL` in case of an error.

- `get_proc_info(struct d_params * params)`
  This system call returns, via the params pointer, the values that have been set for the process that made the system call. The memory for this struct will be allocated by the user-level program that called the system call and not by the kernel. The kernel will simply change the struct's values according to the corresponding values it finds in the process's task_struct and will return the information (by reference) to the user-level program. If the execution of the system call is successful, it returns the value 0. If the params argument is NULL, the system call must return the value `EINVAL`.

Notes:

- For this exercise, you will use the Linux kernel 2.6.38.1

- `EINVAL` as well as other error values are located in the file:
  linux-2.6.38.1/include/asm-generic/errno-base.h

- The `task_struct` structure is defined in the
  file: linux-2.6.38.1/include/linux/sched.h

- To get the task_struct of the current process that made the system call, you can look in the file:
  arch/x86/include/asm/current.h

- The Linux kernel stores detailed information for all current processes in a list of task_struct objects. To access all the system processes in this list, you can use the for_each_process macro.

- In each call of `set_proc_info, get_proc_info` your name, your Student ID and the name of the system call must be printed at the kernel level. To achieve this, use the printk function. In the same way, you can print any other messages you want from the kernel (e.g., a simple way to debug the system call you are creating). You can see these messages when you have loaded Linux with the specific kernel, by running dmesg or executing the command "cat /var/log/messages".

- You will need to define the `struct d_params` in a file that you will create in the linux-2.6.38.1/include/linux folder. This structure will contain 2 fields that will store the parameters of a process.

Demo programs:

You must create and submit at least one test program that will make use of the system calls and demonstrate their correct implementation. For example, your program can first call set_proc_info, then get_proc_info, to print all the parameters set by the previous system call. This program should cover various possible calls of the system calls and present all possible execution cases (e.g., correct and incorrect argument declaration).

Running Linux in QEMU

Emulators are widespread for many reasons. They allow us to install and run an operating system as simple users on a computer that has another operating system, without needing to change anything on it. This computer can be used by several users, and each one can run a different operating system with an emulator without affecting the other users. Especially when we want to test some changes in the kernel of an operating system, as we will do in this exercise, the emulator is quite useful for another reason. If, due to a programming error in the kernel, the system crashes (e.g., kernel panic), we can easily and quickly restart the operating system with some change (debugging) without affecting the computer's basic operating system (host operating system).

QEMU is already installed on the department's machines (man qemu for more information). The QEMU emulator can create and read a virtual disk image, and on it, we can install any operating system (e.g., from a virtual cd rom). For the purposes of the exercise, we have installed for you a simple Linux OS (ttylinux distribution) for 32-bit x86 (i386) architecture on a virtual disk image that you should use for your exercise.

You should first copy this disk image (63 MB) from the course's area (~hy345/qemu-linux/hy345-linux.img) o a directory in /spare on the machine you are using (e.g., /spare/[username]), so you don't have space problems (e.g., quota exceeded) with your area.

```
$ cp ~hy345/qemu-linux/hy345-linux.img /spare/[username]/
```

Make sure you have the appropriate permissions on this directory so that only you have access:

```
$ chmod 700 /spare/[username]
```

The above disk image has the Linux OS you will use installed. It contains the root filesystem (/) in which the system's basic programs and tools exist. So, using this image, you can try to start this Linux OS with the QEMU emulator, simply with the following command:

```
$ qemu-system-i386 -hda hy345-linux.img
```

The parameter -hda hy345-linux.img makes QEMU use the file hy345-linux.img as a virtual disk image, which will appear as the /dev/hda device in the emulated OS (guest operating system). Running the above command, you will see the Linux OS we are emulating start-up. When it asks you to log in, use the account with username "user" and password "csd-hy345". You can also log in with the "root" account with the password "hy345". To exit QEMU and shut down the Guest OS, use the poweroff command.

Compiling the Linux kernel

The next step is to change the Linux kernel, compile it, and create a new kernel image with which you can start the guest Linux OS with QEMU, instead of the original kernel image that exists in the disk image we provide. You could change and compile the kernel from within the guest OS. For convenience, however, you will work on the host OS, i.e., directly on the department machine you are using for the exercise. First, you will need the source code of the Linux kernel 2.6.38.1 to make the necessary changes and compile it. So, you should copy the source code from the course's area to the directory used in /spare/[username] on the machine.

```
$ cp ~hy345/qemu-linux/linux-2.6.38.1-patched.tar.bz2 /spare/
[username]/
```

After you copy and decompress the Linux kernel 2.6.38.1 source code, you can make whatever changes are required to implement the new system calls. Then there are two simple steps you must follow to compile the changed kernel and create a new kernel image: configure and make. There are various ways to configure the Linux kernel (e.g., make menuconfig, make config, etc.). Ultimately, the kernel's configuration is written to the .config file inside the linux-2.6.38.1 directory. Because there are many options for the Linux kernel configuration, we are giving you a ready-made configuration that is compatible with the Linux OS we have installed on the disk image. You will copy the configuration file from the course's area to use it for the kernel you will build.

The only thing you need to change in the .config file is the CONFIG_LOCALVERSION parameter. There you must add a suffix to the name (version) of the new kernel you will build, so that you are sure you are using your own kernel when you load it with QEMU (and not the original kernel). Also, if you repeat the process more times, you will be able to distinguish the different kernel revisions you have tested, by changing the kernel version before each compilation. So, in CONFIG_LOCALVERSION you should put your username, and if you want, a revision number. You will be able to see the kernel version when the OS starts, or after you have logged in with the command:

```
$ uname -a
```

The gcc available on the school's machines is version 10.2.1. However, with this version, we will not be able to successfully compile the Linux kernel. For this reason, we have added an older version gcc compiler to the course's area, which you must use in this exercise. So, before you compile the kernel, you will execute the commands:

```
$ export PATH="/home/misc/courses/hy345/gcc-4.9.2-standalone/bin/:
$PATH"
$ export
PATH="/home/misc/courses/hy345/gcc-4.9.2-standalone/libexec/gcc/
x86_64-unk nown-linux-gnu/4.9.2/:$PATH"
```

This way, you tell the system to look in the specific path to find the compiler, so it will first find the version installed in the course's area. To make sure you are using the correct gcc (4.9.2), just run:
`$ which gcc` ή **`$ gcc --version`**

Finally, to compile the kernel with your own changes, you will run:

```
$ make ARCH=i386 bzImage
```

The new kernel image (bzImage) that will be created from the compilation will be the file **linux2.6.38.1/arch/x86/boot/bzImage**. Since you will not be using the new kernel on the host OS, you do not need to do `make install`.

In summary, the steps you need to follow are:

```
$ cp ~hy345/qemu-linux/linux-2.6.38.1-patched.tar.bz2 /spare/
[username]/
$ cd /spare/[username]
$ tar -jxvf linux-2.6.38.1-patched.tar.bz2
$ cd linux-2.6.38.1

# Edit kernel source code to implement the new system calls

$ cp ~hy345/qemu-linux/.config . # Mind the dot at the end!

# Edit .config, find CONFIG_LOCALVERSION="-hy345", and append to the kernel's
version name your username and a revision number

$ export PATH="/home/misc/courses/hy345/gcc-4.9.2-standalone/bin/:
$PATH"
$ export
PATH="/home/misc/courses/hy345/gcc-4.9.2-standalone/libexec/gcc/
```

QEMU Parameter Setup

The next step is to use the new kernel image, which is located in the file linux2.6.38.1/arch/x86/boot/bzImage to start Linux using QEMU. You will again use the virtual disk image we gave you, but you will also give QEMU the kernel image to run.

```
$ qemu-system-i386 -hda hy345-linux.img -append "root=/dev/hda"
-kernel linux-2.6.38.1/arch/x86/boot/bzImage
```

With the `-kernel linux-2.6.38.1/arch/x86/boot/bzImage` QEMU will start with the new kernel image. With `-append "root=/dev/hda"` QEMU will mount the root filesystem from /dev/hda, which is the disk image you are loading as before. After you login, with the command `uname -a` ,you see the kernel version the system is running.

Remote working

If you are working remotely on a department machine, to start QEMU on the remote machine you will need to connect with X11 forwarding from your own computer. X11 forwarding can be enabled when you connect to the machine:

```
$ ssh [username]@[host].csd.uoc.gr -Y
```

Try running the xterm command and an xterm window should open.As long as xterm works, you can use QEMU. Alternatively, you can simply run QEMU without a graphical environment (much less lag) with the ncurses library:

```
$ qemu-system-i386 -hda hy345-linux.img —display curses
```

Alternatively, you can copy the kernel image (after you have changed and compiled the Linux kernel on a department machine) and the disk image, and then run QEMU (after installing it) locally on your computer. If you are using putty, you must enable the "Enable X11 forwarding" option in the Connection tab and then SSH.

Transferring files from Guest OS to Host OS

To transfer files from the guest OS (which you run with QEMU) to the host OS (where you do your main implementation) and vice versa, you can use the scp program. From within the guest OS, you can access the host OS with the (virtual) IP address 10.0.2.2. For example, to transfer the file test1.c from the guest OS to the host OS in your area in a hy345 directory, you can execute from within QEMU (i.e., from the Guest OS) the command:

```
$ scp test1.c [username]@10.0.2.2:~/hy345
```

The [username] is the username you have on the department's machines. You will need to provide the password you have on the department's machines to complete the copy with scp. Correspondingly, to copy from the host OS (e.g., a department machine) the file test1.c from the hy345 directory in your area to the Linux OS running in QEMU, you will run from within QEMU the command:

```
$ scp [username]@10.0.2.2:~/hy345/test1.c .
```

Pay attention to the period after test1.c. It is not a typo.

Adding a new system call

General information on the steps you need to follow and the files you need to change or create to add a system call to the Linux kernel 2.6 can be found here. The guide below briefly describes how a system call is structured in the Linux kernel and how you can add a new one. There are three basic steps to implement a new system call in the Linux kernel:

1. Add a new system call number to the kernel that corresponds to your system call.

2. Add an entry to the kernel's system call table with your system call's system call number. This entry will determine which kernel function will be executed when a trap occurs with your system call number (i.e., when the system call is called from

user level and control transfers to the kernel for the execution of the specific system call).

3. You must add code to the kernel that implements the functionality the system call will offer. You must also add the appropriate header files to define new types and structs that the system call uses to transfer information between kernel and user space. Furthermore, you will need to copy arguments and results between kernel space and user space using the corresponding functions available in the kernel.

For example:

Lets say we want to add a system call named **dummy_sys** which takes one argument from the user-level program that called it, specifically an integer number. The dummy_sys system call will simply print this number given as an argument and will return its double to the user-level program. We will follow the steps below:

1. We open the file **linux-2.6.38.1/arch/x86/include/asm/unistd_32.h** with an editor, find the system call numbers for the system calls that already exist in the kernel, and after the last system call number (e.g., 340 in our kernel) we add a line with the next number.

```
#define _NR_dummy_sys 341
```

   Also, we increment NR_syscalls by one (e.g. from 341 to 342 in our kernel). Thus we defined the number 341 for the dummy_sys system call. This number will be used after a trap so that the kernel can find the appropriate function (system call function pointer) that implements the system call in the system call table.

2. We open the file **linux-2.6.38.1/arch/x86/kernel/syscall_table_32.S** and there we add, on the last line, the name of the function that implements the new system call.

```
.long sys_dummy_sys /* 341 */
```

3. In the third step, we will define the dummy_sys system call. At the end of the file **linux-2.6.38.1/include/asm-generic/syscalls.h** we add the function prototype of the system call.

```
asmlinkage long sys_dummy_sys(int arg0);
```

4. If you have type definitions to add, you must also create a header file in the **linux-2.6.38.1/include/linux/** folder, which you will include where needed. For this example, this is not necessary.

5. Next, we create a new file in the **linux-2.6.38.1/kernel/** folder which will contain the implementation of the system call. In this example, the file will bel inux-2.6.38.1/ kernel/dummy_sys.c and will contain the code:

```
#include <linux/kernel.h>

asmlinkage long sys_dummy_sys(int arg0) {
    printk("Called dummy_sys with argument: %d\n", arg0);
    return ((long)arg0 * 2);
}
```

6. If you have arguments that are passed by reference from user space to kernel space, you will need to copy them after calling `access_ok()`. The copy can be done by calling the `copy_from_user()`function. You will need to do a corresponding procedure to copy the data back to user space using the `access_ok()` and `copy_to_user()` functions.

7. Finally, you will need to change the **linux2.6.38.1/kernel/Makefile** file to include and compile the new source code file by adding a line in the appropriate place:

```
obj-y += dummy_sys.o
```

Observations:

- It is important to see how some similar system calls that already exist in the Linux kernel have been implemented (e.g., gettimeofday, times, getpid)
- You can learn and follow the Linux Kernel coding style when implementing system calls: https://www.kernel.org/doc/Documentation/process/coding-style.rst
- The Elixir Cross Reference will help you navigate the Linux kernel source code. Its search (Search Identifier) will likely be useful for you to locate functions and structures in the various source code files. Be sure to select the correct kernel version we are using in the exercise. https://elixir.bootlin.com/
- To modify files from within the Guest OS, you can use the Vi editor, which works similarly to the Vim editor. More information on using Vi can be found at: http://linuxfocus.org/~guido/vi/viref.html

Testing system calls

In the last step of the exercise, you will need to test the new system calls. After you have successfully compiled the kernel with the system calls you created, and have started QEMU with the new Linux kernel, you will need to write some test programs that use `set_proc_info, get_proc_info` in the guest Linux OS.

Do not forget that the declarations of the system calls you will make are in the kernel and are not visible to user-level programs. Usually, a system call is called via a function that runs at user level and is implemented in some library (e.g., libc). Then, this user-level function calls the `syscall()` macro with its correct system call number, to transfer control to the kernel (trap). There, the system call's code will run. If this function has not been implemented in a user-level library (as in your case), a program can call the desired system call using the `syscall()` macro and the number of the system call it wants to call. For example, the following piece of code calls the system call with number 341 and gives it the value 42 as an argument.

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

#define _NR_dummy_sys 341
```

```
int main(void) {
    printf("Trap to kernel level\n");
    syscall( NR_dummy_sys, 42);
    printf("Back to user level\n");
    return 0;
}
```

For this exercise, you must make the system calls look like function calls. You can achieve this either by declaring macros or by creating the appropriate wrapper functions which will accept the same arguments as the system calls. For example:

```
#define _NR_dummy_sys 341

/* Use either a macro */
#define dummy_sys(arg1) syscall(_NR_dummy_sys, arg1)

/* Or a wrapper function */
long dummy_sys(int arg1) {
    return syscall(_NR_dummy_sys, arg1);
}
```

This way, in your test program, you can simply write `dummy_sys(42)`and the correct system call will be called with the correct arguments.

Note:

If you have header files with definitions for new types and structs, these must also be included in the demo programs you write. For this to happen, you will need to transfer them to the Guest OS and you may need to provide the path to these header files during compilation. Specifically, you will need to define the `d_params` struct so that it is visible at the user level. For this reason, you can add the definition for this struct along with the macros or wrapper functions of the system calls. All these definitions can be placed in the /usr/include/unistd.h file that exists in the Guest OS and which you will include in all your test programs.

Submission

After you do the exercise, you must submit the following:

1. The new kernel image that resulted from the compilation, i.e., the file linux-2.6.38.1/arch/x86/boot/bzImage.

2. All the files you had to modify or create in the Linux kernel source code to implement the system calls. This means you will submit all the .c, .h, Makefile, etc. files in which you made any change or created yourself. Caution, do not submit files that you did not need to modify for your implementation (e.g., the entire rest of the kernel source tree).

3. The code from all the test programs you wrote and ran inside the guest Linux OS to test the system calls you implemented. Also, any header files you used for type and function definitions and any Makefiles needed for the compilation of these programs. You do not need to submit the executable files.

4. A README file in which you briefly (but comprehensively and clearly) describe all the steps you followed for the addition and implementation of the new system calls. Also, you must comment on what you observed from the test programs you ran. If you have done something different or more than what we mention in the exercise description at any step, you can also mention it in the README.

5. You can create a folder with the modified kernel source code files (if you want, it would be good to keep the directory structure that exists in the linux kernel), a folder with the test programs and header files from the guest OS, and finally transfer them to a folder together with the bzImage and the README file to submit the exercise in the known way.

Caution:

1. You **DO NOT** need to submit the disk image (hy345-linux.img) even if it has been modified. Indeed, the disk image may change as you use the guest OS, but you do not need to submit it.

2. You **DO NOT** need to submit any file with the entire Linux kernel source code. You must note and submit only the files you modified or created. The kernel image (bzImage), the source and header files, as well as the test programs you submit, must be sufficient so that your exercise can run with the original disk image and QEMU so that the correct implementation of the exercise is visible.

Observations

1. The exercise is individual. Any copying can be easily detected by appropriate software and will be given a grade of zero. Include your name and your account in all files.

2. The use of code produced by AI is explicitly prohibited.

3. Write a README file, at most 30 lines, with explanations on how the system calls were implemented.

4. Construct a Makefile file, so that by typing `make  all`, the compilation of the programs that use the system calls is done and the executable files are produced. Also, by typing `make clean`, all unnecessary files should be cleaned up, leaving only the files needed for compilation.

5. Place all the files to be submitted for exercise 3 in a zip folder. Submit the zip file via E-Learn.

6. In many cases, the file names are indicative. You can use whatever names are convenient for you.