CS-345 Operating Systems
Fall Semester 2026

Exercise 3

Tutorial: 10/11/2026
Deadline: 20/11/2026

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 its deadline and how much CPU time it needs to execute (estimated runtime). For this exercise, the deadline is measured in seconds from the declaration, while the estimated runtime is measured in milliseconds. For example, a deadline of 5 means "within 5 seconds."

For the scheduling policy, let D be the absolute deadline, E the estimated execution time and C the CPU time already consumed. The remaining execution time is E − C, the slack is (D − now) − (E − C), and the priority is (E − C) / (D − now), using the same time unit throughout. The scheduler runs the process with the highest priority. **The scheduler will be implemented in Exercise 4. In Exercise 3, you store and retrieve the two declared values.**

In this exercise, you will implement two new system calls: **set_proc_info** and **get_proc_info**. The first stores the process parameters that you will also use in the next exercise; the second returns the stored values. 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 uses the new system calls. This program will run on the Linux operating system that you 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;     /* deadline in seconds from now */
int est_runtime; /* estimated CPU time in milliseconds */
```

  The values of these fields are changed by `set_proc_info` and returned to a user program by `get_proc_info`. They are initially zero. A successful setter replaces both values; an unsuccessful setter must leave both unchanged.

```
set_proc_info(int deadline, int est_runtime)
```

This system call changes the corresponding fields in the `task_struct` of the process that made the call. Both arguments must be positive, and the estimated runtime must fit within the deadline: `est_runtime <= 1000LL * deadline`. Equality is valid. Check all arguments before changing either field. Return **0** on success or **−EINVAL** from the kernel on error. The user program sees **−1** with **errno = EINVAL** on error.

```
get_proc_info(struct d_params *params)
```

This system call copies the stored values for the calling process into the structure pointed to by `params`. The user-level program allocates this structure, not the kernel. The call must not change the process fields. Return **0** on success, **−EINVAL** if `params` is NULL, or **−EFAULT** if the values cannot be copied to the supplied user address. On error, the user program receives **−1**, with **errno** set to EINVAL or EFAULT respectively.

Notes:

- For this exercise, you will use the course-patched **Linux kernel 2.6.38.1**.
- `EINVAL` and other error values are located in:
  linux-2.6.38.1/include/asm-generic/errno-base.h
- The `task_struct` structure is defined in:
  linux-2.6.38.1/include/linux/sched.h
- To get the `task_struct` of the process that made the call, use `current`. See arch/x86/include/asm/current.h.
- On every call to **set_proc_info** and **get_proc_info**, use **printk** to print your name, Student ID and the system-call name. You may print additional messages to help debug your implementation. View the messages with **dmesg** inside the guest.
- Define **struct d_params** in a header that you create in **linux-2.6.38.1/include/linux/**. Use the same definition in the local header included by your user-level test programs:

```
struct d_params {
    int deadline;    /* seconds */
    int est_runtime; /* milliseconds */
};
```

Demo programs:

You must create and submit at least one test program that uses both system calls and demonstrates their correct implementation. For example, call `set_proc_info`, then `get_proc_info`, and print the returned fields. Cover successful calls, zero or negative setter arguments, an estimated runtime that exceeds the deadline, repeated calls, and a NULL getter argument. Check that a rejected setter preserves both values by reading them with the getter. Also test a non-NULL address that is not writable and check for EFAULT.

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. The QEMU emulator can create and read a virtual disk image, and on it we can install an operating system. For the purposes of the exercise, we have installed a simple Linux OS (ttylinux distribution) for 32-bit x86 (i386) architecture on a virtual disk image that you should use for your exercise.

First copy the disk image from **~hy345/qemu-linux/hy345-linux.img** to your own directory in **/spare** on the machine you are using. Replace `[username]` with your department account. Check that /spare has free space and make the directory accessible only to you:

```
$ df -h /spare
$ mkdir -p /spare/[username]
$ chmod 700 /spare/[username]
$ cd /spare/[username]
$ cp ~hy345/qemu-linux/hy345-linux.img .
```

The disk image contains the root filesystem (/) and the system's basic programs and tools. From this directory, start the original Linux OS with:

```
$ qemu-system-i386 -hda hy345-linux.img \
    -net nic -net user,hostfwd=tcp:127.0.0.1:2222-:22 \
    -display curses
```

The parameter `-hda hy345-linux.img` makes QEMU use this file as a virtual disk, which appears as /dev/hda in the emulated OS (guest). The network options will allow the file transfers described below. Log in with username **user** and password **csd-hy345**, or with username **root** and password **hy345**. To shut down the guest and exit QEMU, run **/sbin/poweroff** as root. Run only one QEMU instance with your disk image at a time.

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, copy the source code of Linux kernel 2.6.38.1 from the course area to your directory in /spare:

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

After you copy and decompress the source code, you can make the changes required to implement the two new system calls. Then there are two steps 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). Ultimately, the configuration is written to the **.config** file inside the linux-2.6.38.1 directory. We provide a ready-made configuration compatible with the Linux OS installed on the disk image. Copy it from the course area:

```
$ cd linux-2.6.38.1
$ cp ~hy345/qemu-linux/.config .
```

The only configuration parameter you need to change is **CONFIG_LOCALVERSION**. Add your username and, optionally, a revision number to the kernel version, for example `-csdXXXX-a3-r1`. This lets you distinguish your kernel from the original and from earlier builds. After you boot it, check the version with:

```
$ uname -a
```

The default compiler on the department machines is newer than the one needed for this kernel. Use the course's **GCC 4.9.2** compiler. Before compiling, execute the following commands in your host terminal:

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

This makes the system find the course compiler first. Check the selected compiler, then build the kernel:

```
$ which gcc
$ gcc --version
$ make ARCH=i386 bzImage
```

The new kernel image is **linux-2.6.38.1/arch/x86/boot/bzImage**. Since you will not use this kernel on the host OS, you do not need to run **make install**.

In summary, the steps you need to follow are:

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

# Edit the kernel source to implement the two new system calls.
$ cp ~hy345/qemu-linux/.config .
# Edit CONFIG_LOCALVERSION in .config.

$ TOOLCHAIN=/home/misc/courses/hy345/gcc-4.9.2-standalone
$ export PATH="$TOOLCHAIN/bin:$PATH"
$ export PATH="$TOOLCHAIN/libexec/gcc/\
x86_64-unknown-linux-gnu/4.9.2:$PATH"
$ gcc --version
$ make ARCH=i386 bzImage
```

QEMU Parameter Setup

The next step is to use the new kernel image to start Linux with QEMU. You will again use the virtual disk image we gave you, but you will also give QEMU the kernel image to run. First shut down any guest that is already using this disk image. Run the following command from /spare/[username]:

```
$ qemu-system-i386 -hda hy345-linux.img \
    -kernel linux-2.6.38.1/arch/x86/boot/bzImage \
    -append "root=/dev/hda" \
    -net nic -net user,hostfwd=tcp:127.0.0.1:2222-:22 \
    -display curses
```

The **-kernel** option starts QEMU with the new kernel image. With **-append "root=/dev/hda"**, QEMU mounts the root filesystem from /dev/hda, the disk image you are loading as before. After you log in, use **uname -a** to check that the running kernel has your version suffix.

With a custom version suffix, the supplied image may show a module-dependency startup message marked FAILED because it has no module directory for that version. The supplied .config builds the required disk, network and filesystem drivers into the kernel. If the guest boots and these work, check your version and continue with the tests.

Remote working

If you are working remotely on a department machine, connect from your computer with SSH. The QEMU commands above use **-display curses**, which works inside the terminal without a graphical environment:

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

If you prefer a graphical QEMU window, connect with **ssh -Y** to enable X11 forwarding and omit -display curses from the QEMU command. Try running **xterm** first to check that X11 works. In PuTTY, enable “Enable X11 forwarding” under Connection > SSH > X11. Alternatively, you can copy the kernel and disk images to your own computer and run QEMU locally after installing it.

Transferring files between Guest OS and Host OS

To transfer files between the guest and host, keep QEMU running and open **a second SSH terminal on the same department machine**. The **host** is that department machine (for example, Damaskino); the **guest** is Linux inside QEMU. Run both scp commands below **on the host**. The hostfwd option in the QEMU commands above forwards host port 2222 to the guest’s SSH service. If that port is occupied, choose another free port and use it in both QEMU and scp.

The supplied guest has an older SSH server, so use the compatibility options below. The **-O** option selects the SCP protocol supported by the guest. To send a file, run the command from the host directory containing **test1.c**:

```
# Host to guest. Run on the department machine.
$ scp -O -P 2222 \
    -o HostKeyAlgorithms=+ssh-rsa \
    -o KexAlgorithms=+diffie-hellman-group14-sha1 \
    test1.c user@127.0.0.1:/home/user/

# Guest to host. Run on the department machine.
$ scp -O -P 2222 \
    -o HostKeyAlgorithms=+ssh-rsa \
    -o KexAlgorithms=+diffie-hellman-group14-sha1 \
    user@127.0.0.1:/home/user/test1.c .
```

Use the guest password **csd-hy345**. Copy your local headers and Makefile in the same way. **/home/user belongs to the guest**; you do not create it on the host. The final dot means the current host directory. The address **127.0.0.1** refers to the host where QEMU is running, so these commands must run there.

Adding a new system call

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:

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. This entry determines which kernel function is executed when the system call is called from user level and control transfers to the kernel.
3. Declare and implement the kernel function, and add its source file to the kernel build.

For example:

Let's say we want to add a system call named **dummy_sys** which takes one integer argument from the user-level program. It prints the argument and returns its double. This is a worked example; the calls you must implement are **set_proc_info** and **get_proc_info**. Follow the corresponding steps for each call, assigning a distinct unused number to each.

1. Open **linux-2.6.38.1/arch/x86/include/asm/unistd_32.h**, find the existing system call numbers, and add the next unused number after the last one (340 in the supplied kernel):

```
#define __NR_dummy_sys 341
```

   For this one-call example, increment **NR_syscalls** from 341 to 342 in the same file. For your two calls, set it to one more than the highest assigned number. The syscall number selects the corresponding function in the table. Use matching numbers in the kernel and user program; do not register two different calls with the same number.

2. Open **linux-2.6.38.1/arch/x86/kernel/syscall_table_32.S** and add the name of the function that implements the new system call at the corresponding table entry:

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

3. In **linux-2.6.38.1/include/linux/syscalls.h**, add the function prototype before the closing include guard:

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

4. If you have type definitions to add, create a header in **linux-2.6.38.1/include/linux/** and include it where needed. For this exercise, define **struct d_params** there. Make the type visible to the getter prototype and implementation; mark its kernel pointer parameter **__user**.

5. Create the implementation file in **linux-2.6.38.1/kernel/**. For the example, create **dummy_sys.c** with the following code:

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

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

6. For **get_proc_info**, prepare the result in a kernel-local structure and use **copy_to_user()**, declared through **<linux/uaccess.h>**, to copy it to `params`. Check its return value: **0** means the whole structure was copied; a nonzero value means bytes were not copied and the call must return **−EFAULT**. Check NULL separately as specified above. Do not directly dereference a user pointer in kernel code.

7. Finally, change **linux-2.6.38.1/kernel/Makefile** to compile the new source file by adding:

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

Observations:

- It is important to see how 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 helps you navigate the Linux source. Its Search Identifier function can locate functions and structures. Select version **2.6.38.1**:
  https://elixir.bootlin.com/
- To modify files inside the guest, you can use the Vi editor. More information is available at:
  http://linuxfocus.org/~guido/vi/viref.html

Testing system calls

In the last step of the exercise, you will test the new system calls. After you have successfully compiled the kernel and started QEMU with it, write and run your test programs using **set_proc_info** and **get_proc_info** in the guest Linux OS.

Do not forget that the declarations you add are in the kernel and are not visible to user-level programs. Usually, a system call is called through a user-level function implemented in a library such as libc. If this function has not been implemented in a user-level library, as in your case, a program can call the kernel using the **syscall() function** and the system call number. For example, the following code calls the example system call with number 341 and gives it 42 as an argument:

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

#define __NR_dummy_sys 341

int main(void)
{
    long result;
    printf("Trap to kernel level\n");
    result = syscall(__NR_dummy_sys, 42);
    if (result == -1) {
        perror("dummy_sys");
        return 1;
    }
    printf("Back to user level: %ld\n", result);
    return 0;
}
```

Compile and run the program **inside the guest**. For the dummy example, the expected successful return is 84. Adapt your own tests to the arguments and return values of both required calls. Use the getter to check the stored fields and dmesg to inspect kernel messages.

```
$ gcc -Wall -Wextra -o test1 test1.c
$ ./test1
$ dmesg | tail -n 30
```

For this exercise, you must make each system call look like a function call. You can achieve this either by declaring a macro or by creating a wrapper function which accepts the same arguments as the system call. 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 argument. For your assignment, define wrappers for **set_proc_info(int deadline, int est_runtime)** and **get_proc_info(struct d_params *params)**.

Note:

Place both syscall numbers, the wrappers and the definition of **struct d_params** in a local header, for example **proc_info.h**. Include it in your test programs with `#include "proc_info.h"`. The structure must have the same field types and order as its kernel definition. Copy the header to the guest together with your tests. There is no need to modify **/usr/include/unistd.h**.

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 two system calls. This means you will submit all the .c, .h, Makefile, .config, 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 both 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 two 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 (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. The image changes as you use the guest OS, but you do not need to submit it.
2. You **DO NOT** need to submit the entire Linux kernel source code. Submit only the files you modified or created. The kernel image, source and header files, and test programs you submit must be sufficient to run your exercise with the original disk image and QEMU and demonstrate the correct implementation.

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 AI tools, including code produced by AI, is explicitly prohibited.**
3. Write a **README of at most 30 lines**, explaining how the two system calls were implemented.
4. Create a **Makefile** so that **make all** compiles the test programs and produces the executables. **make clean** must remove unnecessary generated files, leaving the files needed for compilation.
5. Place all files to be submitted for Exercise 3 in a **ZIP** archive. Submit the ZIP through **E-Learn**.
6. In many cases, file names are indicative. You can use names that are convenient for you.