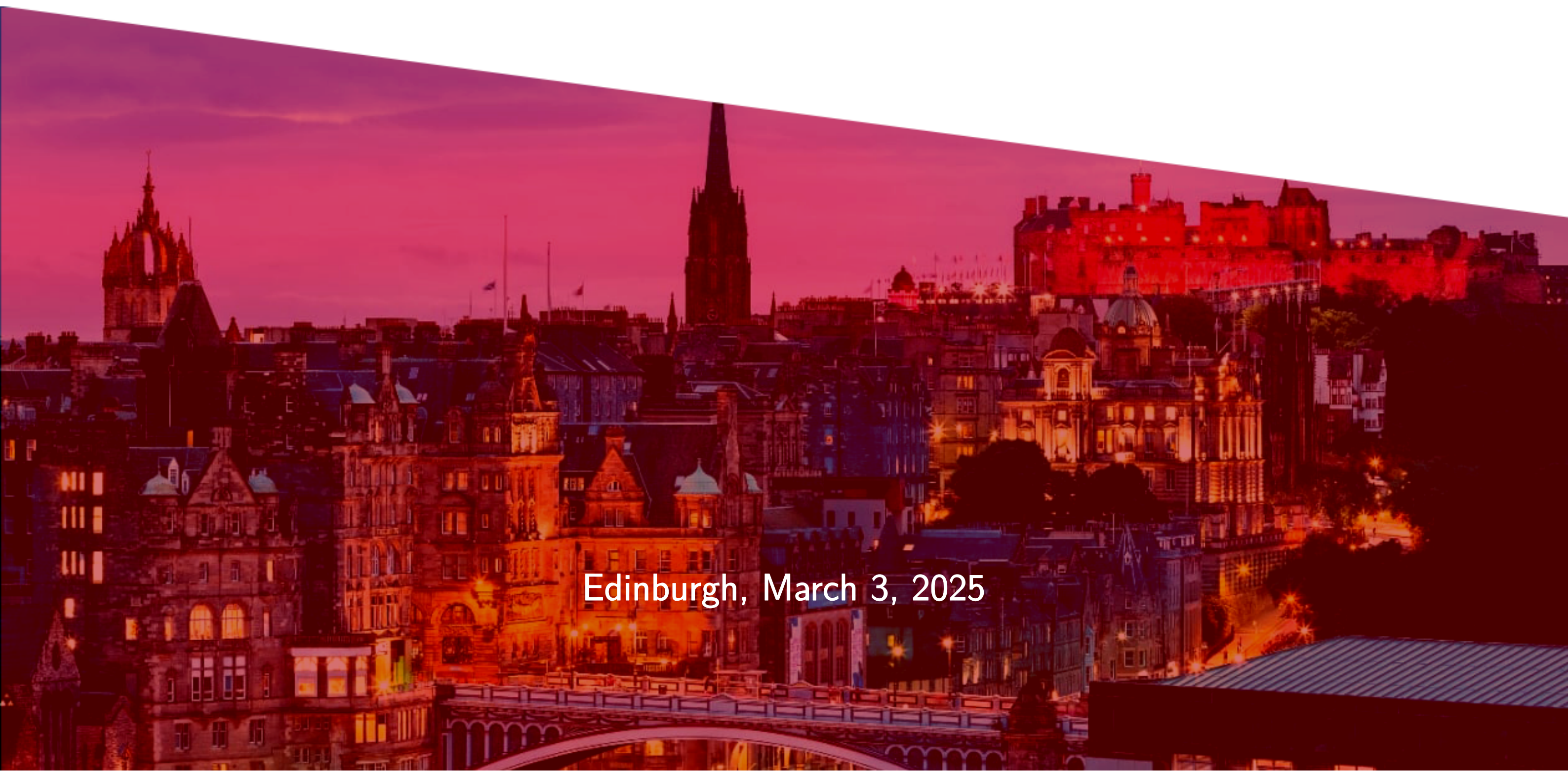
Operating Systems Courseworks 2024-2025

Course Number: INFR100792023

Semester Number: 2

Score Out of 100: 50%

Authors: Amir Noohi



Contents

1 | Introduction

Welcome to `INFR100792023` - Operating Systems in the Spring semester of the academic year 2023-24. In this course, you shall dive head-first into the exciting and complex world of operating systems, gaining both a strong theoretical foundation and hands-on expertise. This document contains information about the coursework assignments which account for half of your total grade in this course. There are four assignments - Coursework 0, 1, 2, and 3.

This Chapter shall outline the aims, expectations, guidelines, and rules we (the course faculty) expect you (the students taking this course) to abide by, along with the necessary background and skills we expect you to possess beforehand or develop as the course progresses. Chapter 2 will describe the environment setup on which you shall do the coursework. Chapters 3, 4, 5, and 6 pertain to Coursework 0, 1, 2, and 3 respectively.

1.1 | Aims of the Coursework

The goal of the coursework is to complement the theoretical side of the course (which will be covered in lectures and evaluated in the final exam) with a practical side. The lectures will imbibe you with an abstracted-out view of how an operating system (OS) is structured, and how its various parts interplay with each other. While doing the coursework, however, you shall perceive how this conceptual design is fleshed out into a real implementation. Your understanding of an Operating System will be complete as you will be able to map the implementation to the concepts, and vice-versa.

As an Operating Systems developer, you must be able to navigate through large code bases with many connected and interacting components. The OS is the most privileged software on any computer system, and it directly manages and controls access to the physical resources on a system, such as memory, processor cores, and storage. When you try to implement a seemingly small modification to an existing OS, you must first understand how the relevant parts interact, because a misunderstanding may result in a kernel that does not boot or crashes unexpectedly. A buggy OS can be catastrophic if deployed in a real-world use-case.

On the other hand, the OS is a very extensive and complex piece of software. The Linux kernel, which you shall use for your coursework, has over 28 million lines of code spread out over 169 thousand files. Comprehending the purpose of every line in the entire source code is a futile effort, and ultimately unnecessary if you just want to modify some part of this kernel. Thus, you must be able to judge what to explore and what to ignore, what is relevant to your task and what is secondary, what must be perceived comprehensively and what can be safely set aside as an abstraction. This is a very subtle yet critical skill that differentiates an amateur programmer from a seasoned veteran of systems programming. Doing the coursework with due diligence shall nudge you to the latter class.

1.2 | Timeline

The below table outlines the schedule for each coursework. Coursework 0, focusing on preparation, is not marked, but its completion is strongly recommended. Please note that regulations for late submissions can be found in Section 1.4.1, and all assignments have a deadline of 12 PM.

Table 1.1: Coursework Schedule

Assignment	Score	Effort	Release Date	Deadline	Feedback By
CW0	0%	N/A	28/01/2024	11/02/2024	N/A
CW1	17%	16 Hours	11/02/2024	04/03/2024	25/03/2024
CW2	17%	16 Hours	04/03/2024	18/03/2024	08/04/2024
CW3	16%	16 Hours	18/03/2024	01/04/2024	22/04/2024

1.3 | Required Background

The coursework for this Operating Systems class demands a strong foundation in the C programming language. As we delve into the intricacies of the Linux kernel, which is predominantly written in C, a high level of fluency in this language is essential. While it is not a prerequisite to have prior experience with the Linux Operating System itself, even as a user, the key requirement is a robust capability in expressing complex programming concepts in C.

To support your learning curve, we will provide some basic tutoring in C programming. However, it's important to recognize that this assistance will have its boundaries in terms of how comprehensively a language can be taught within the constraints of the course. The nature of the coursework is such that it will inherently enhance your proficiency in C. This practical approach to learning through direct engagement with the Linux source code is designed to solidify your understanding and skills.

However, if you are at the very beginning of your journey with the C language, particularly if concepts such as pointers are challenging for you, and if you doubt your ability to rapidly acquire a deep understanding of these concepts, it is crucial to carefully consider your participation in this course. This course is intensive and assumes a certain level of pre-existing knowledge and comfort with C. For those who are not confident in their current level of proficiency in C, especially in handling advanced programming constructs, it might be advisable to seek additional preparation before embarking on this course.

This course is not just about learning how to use the Linux kernel; it is about understanding and modifying its very foundation. This requires not only a theoretical understanding of programming concepts but also the practical ability to apply these concepts in a complex, real-world environment. If you are ready for this challenge and confident in your C programming skills, we welcome you to a journey of deep learning and practical application in the world of operating systems.

1.4 | Guidelines and Rules

This course operates under three key guidelines and rules: "Late coursework & extension requests," "Declaration of Own Work," and "Guide to the Principled Code." Adherence to these guidelines is critical for the successful completion of the course. Late submissions and extension requests are governed by specific rules, academic integrity is paramount in all submissions, and the ability to produce compilable and runnable code is essential. Each of these aspects will be detailed in their respective subsections, outlining the expectations and standards required for your coursework.

1.4.1 | Late Coursework & Extension Requests

This course enforces specific policies for late submissions and extension requests. Assignments should be submitted by the set deadline. However, understanding that unforeseen circumstances can arise, each coursework has defined late submission rules. These rules, outlining the possibilities for late submissions, incurred penalties, maximum extension lengths, and other relevant information, are available on the Course LEARN Page.

In instances impacting your ability to complete coursework on time where late submissions are permitted, you may request an extension through the online Assessment Support Tool. These requests are reviewed by the University Extensions and Special Circumstances Service (ESC) team. Due to the time taken for decision-making, it's advisable to request extensions as early as possible.

Students registered with the Disability & Learning Support Service with entitlements for extra time on coursework should apply these adjustments for late submissions. For circumstances preventing timely submission, you may need to apply for Special Circumstances.

The School of Informatics follows a Late Submission Rules and Penalties Policy in line with the University Taught Assessment Regulations. For example, Rule 1 allows for a 3-day extension and a 7-day Extra Time Adjustment (ETA), both combinable. Late submissions without an approved extension incur a 5% penalty per calendar day for up to 7 days post-deadline, after which a zero score is assigned.

Marks are typically returned within 28 calendar days of the submission deadline. For detailed information on late submission rules, penalties, and scenarios involving extensions and ETAs, visit https://web.inf.ed.ac.uk/node/4533.

1.4.2 | Declaration of Own Work

By submitting assignments in this course, you affirm that your work complies with the University's code of conduct and the Student Conduct and Assessment regulations. This declaration confirms that, except where indicated, all submitted work is your own and that you have:

- Read and understood the University's regulations concerning academic misconduct.
- Where relevant to the assessment style:
  - Clearly referenced/listed all sources.
  - Correctly referenced and marked all quoted text (from books, web, etc.) with quotation marks.
  - Cited the sources of all images, data, etc., not created by you.
- Not communicated about the work with other students, either electronically, verbally, or through any other means, nor allowed your work to be seen by other students.
- Not used any assessment material from other students, past or present.
- Not submitted work previously presented for this or any other course, degree, or qualification.
- Not incorporated work from or sought assistance from external professional agencies, except for attributed source extracts where appropriate.
- Complied with all examination requirements as outlined in the examination paper and course/programme handbooks.
- Understood that the University of Edinburgh and TurnitinUK may electronically copy your submitted work for assessment, similarity reporting, and archival purposes.
- Understood that it is a violation of the Code of Student Conduct to:
  - Use or assist in using unfair means in any University examination.
  - Engage in any action that disrupts the integrity of the examination process.
  - Impersonate another student or allow another student to impersonate you.
- Understood that cheating is a grave offence. Any student found to have cheated or attempted to cheat in an examination may face severe penalties, including failing the examination or the entire examination diet, or any other penalty deemed appropriate by the University.

For further guidance on plagiarism, you are encouraged to:

- Consult your course organiser or supervisor for personalized advice and clarification.
- Visit the University of Edinburgh's Institute for Academic Development website for resources on referencing and citations: Referencing and Citations Resources.
- Refer to the University's guidelines on academic misconduct to understand what constitutes plagiarism and how to avoid it: Academic Misconduct Guidelines.

1.4.3 | Guide to the Principled Code

In this course, the emphasis on producing principled and functional code is critical. Adherence to the following guidelines is essential for a successful evaluation of your coursework:

- **Compilability:** The primary requirement is that your code must compile without errors. Submissions that fail to compile will automatically receive a zero score. It is your responsibility to ensure that your code is free of compilation errors before submission.
- **Adherence to Skeleton:** Each assignment comes with a provided code skeleton which you are required to follow. Non-compliance with the provided skeleton will lead to an automatic score of zero. This policy is strict, and no objections to the score will be entertained on grounds of non-adherence.
- **Code Readability:** Your code must be neat, clean, and well-organized. Proper formatting, consistent naming conventions, and logical structuring are expected. Readability is key to both effective coding practices and ease of assessment.
- **Commenting:** Ambiguous or complex sections of code should be accompanied by clear comments. These comments should explain the rationale behind the chosen approach or clarify complex parts of the code. The goal is to make your code as understandable as possible.

- **Functional Completeness:** Beyond just compiling, your code should correctly implement the required functionalities as outlined in the assignment brief. Functional completeness will be a significant factor in the assessment.

Adhering to these principles is not only crucial for your success in this course but also forms the foundation of good software development practices.

1.5 | Technology Stack

The diagram presented in Figure 1.1 illustrates a high-level design of a computer system. It encapsulates the multi-layered architecture that defines the interaction between user applications and the physical hardware of a computer. At the foundation, there is the hardware layer, composed of the essential physical components such as the CPU, memory, and storage devices. Building upon this, the operating system layer takes precedence, providing essential services such as process management, memory allocation, and I/O operations. Further abstracted, the standard library offers a set of predefined functions for performing fundamental tasks, like file handling and process control, which are utilized by the utilities layer to furnish tools and applications for end-user interaction.

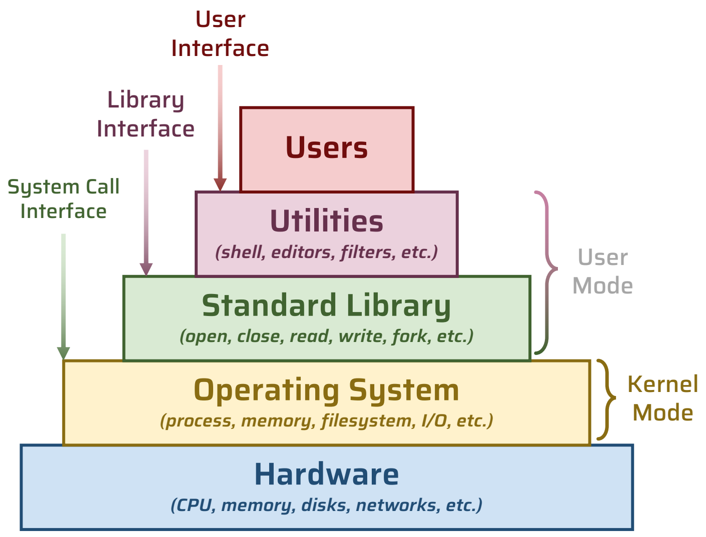

User Interface
Library Interface
System Call Interface
Users
Utilities
(shell, editors, filters, etc.)
Standard Library
(open, close, read, write, fork, etc.)
Operating System
(process, memory, filesystem, I/O, etc.)
Hardware
(CPU, memory, disks, networks, etc.)
User Mode
Kernel Mode

Figure 1.1: Computer System Diagram

In the context of a virtualized system, the depicted model extends by incorporating additional layers to represent the virtualization aspect (see Figure 1.2). Atop the hardware layer resides the host operating system, which is responsible for the direct management of the hardware resources. The virtualization layer, typically provided by software like QEMU, allows for the creation and management of virtual machines (VMs). Each VM encapsulates a complete set of the layers described above, including its own guest operating system that operates oblivious to the virtualized nature of the underlying hardware. This encapsulation allows multiple isolated operating system instances to coexist on a single physical machine, optimizing resource utilization and providing robust environment isolation.

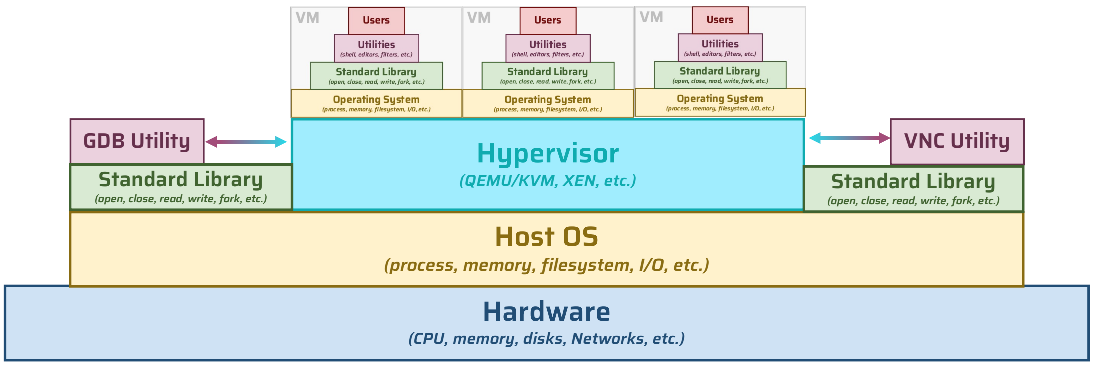

VM
Users
Utilities
(shell, editors, filters, etc.)
Standard Library
(open, close, read, write, fork, etc.)
Operating System
(process, memory, filesystem, I/O, etc.)
VM
Users
Utilities
(shell, editors, filters, etc.)
Standard Library
(open, close, read, write, fork, etc.)
Operating System
(process, memory, filesystem, I/O, etc.)
VM
Users
Utilities
(shell, editors, filters, etc.)
Standard Library
(open, close, read, write, fork, etc.)
Operating System
(process, memory, filesystem, I/O, etc.)
GDB Utility
Standard Library
(open, close, read, write, fork, etc.)
Hypervisor
(QEMU/KVM, XEN, etc.)
VNC Utility
Standard Library
(open, close, read, write, fork, etc.)
Host OS
(process, memory, filesystem, I/O, etc.)
Hardware
(CPU, memory, disks, Networks, etc.)

Figure 1.2: Virtualized System Diagram

The coursework in this course involves the use of a technology stack consisting of three essential components: Linux, QEMU, and GDB. Detailed familiarity with these tools is crucial for developing and testing your code effectively.

1.5.1 | Linux

For your coursework, you will be working with the Linux kernel version 6.1, which is a long-term support (LTS) release. This version provides stability and a comprehensive feature set, making it ideal for educational purposes. It's important to note that throughout the coursework, we will only support the x86_64 architecture. This focus allows for a standardized development and testing environment that is widely applicable and relevant in the field.

Remember, all course-related development and testing will be expected to be compatible with the Linux kernel 6.1 on the x86_64 architecture. This requirement ensures consistency in the coursework and aligns your learning experience with current industry standards.

1.5.2 | QEMU

Working with an operating system, especially during development and testing phases, inevitably involves encountering crashes and bugs. Running a buggy OS on actual hardware can be risky. Hence, you will use *virtual machines* (VMs) for your coursework. QEMU is a *hypervisor* that allows the creation and management of VMs, providing a safe environment for testing. For detailed instructions on using QEMU, refer to Chapter **??**.

1.5.3 | GDB

The GNU Debugger (GDB) is a powerful tool for debugging C programs. It enables you to trace the execution of a program step-by-step, pause it, and inspect various symbols at runtime. This capability is invaluable for identifying and fixing errors in your code. You will become familiar with GDB during Coursework 0, as detailed in Chapter 2.

Each of these tools is integral to the development process in this course. Mastery of Linux, QEMU, and GDB will not only aid in your coursework but also equip you with valuable skills applicable to a wide range of software development and systems engineering tasks.

2 | Coursework 0

2.1 | Goal of the Coursework

The primary goal of this coursework is to deepen students' understanding of the Linux kernel's structure, focusing on how to navigate, modify, and extend its functionalities. This objective is divided into two key areas:

1. **Understanding and Modifying the Linux Kernel:** Students will learn how the Linux kernel is organized, identifying its critical components and their roles. The coursework will guide them through the process of downloading and compiling the kernel, providing insights into the kernel architecture. A significant emphasis will be placed on how to modify the kernel for specific purposes. This includes adding new modules, integrating them into the kernel source, and understanding the interaction between different kernel parts. Students will also learn how to create and add menu entries in the kernel configuration, which is essential for managing new modules.
2. **Kernel Module Development and Function Hierarchy:** An essential part of this coursework is the development of new kernel modules. Students will be taught how to write, compile, and insert modules into the Linux kernel. The course will also delve into the order of functions within the kernel, explaining how different components interact at the code level. This will provide students with a comprehensive view of the kernel's functional flow and how their custom modules can seamlessly integrate into it.

Through a combination of theoretical knowledge and hands-on practice, the coursework is structured to progressively enhance the students' understanding and skills in Linux kernel development. This approach aims to prepare students for advanced tasks in systems programming and operating system development, with a strong foundation in kernel customization and debugging techniques.

2.2 | Background

For this coursework, you will work with the Linux kernel version 6.1..

Important aspects to consider:

- **Linux Kernel Version:** You should use Linux kernel version 6.1.74 for this coursework.
- **Use of DICE Server:** The compilation of the Linux kernel should be done on a DICE server. This is a more efficient approach compared to compiling within a VM. The DICE server provides a faster and more suitable environment for such intensive tasks.
- **Role of the VM:** The VM, which will be running on the same DICE server, is solely for testing your compiled kernel and for running user-space applications. It should not be used for compiling or downloading the kernel.
- **Pre-installed Dependencies:** All required dependencies for compiling the Linux kernel are already installed on the DICE servers. You do not need to install any additional software or libraries.
- **Kernel Images:** Post-compilation, you will primarily work with two types of kernel images:
  1. **bzImage:** The compressed kernel image, which you will use with QEMU for testing the compiled kernel.
  2. **vmlinux:** The uncompressed kernel image, essential for kernel debugging with GDB.

2.3 | Tasks

This section outlines two practical tasks to enhance your understanding of Linux kernel debugging. The first task is simpler, involving a modification to the kernel's boot process. The second task is more advanced, requiring the creation of a kernel module.

2.3.1 | Task 1: Showing Boot Message

The goal of this task is to modify the Linux kernel's 'start_kernel' function to print a custom message during the boot process.

1. **Locate the Function:** Find the 'start_kernel' function in the kernel source code. It's typically located in the 'init/main.c' file.
2. **Modify the Function:** Add a line of code to print a message. For example:

   ```
   printk(KERN_INFO "Hello, Kernel World!\n");
   ```

   This line uses the 'printk' function, which is the kernel's equivalent of 'printf'.
3. **Recompile the Kernel:** After making your changes, recompile the kernel as described in earlier sections.
4. **Test Your Changes:** Boot your VM with the newly compiled kernel and observe the message during the boot process.

You can easily confirm this by checking your VM logs using *dmesg* command and then you can find your text using *grep* like below

```
dmesg | grep "YOUR MESSAGE"
```

2.3.2 | Task 2: Writing and Loading a Kernel Module

This advanced task involves writing a simple kernel module that prints a message and loading it during the boot process.

1. **Create the Module:** Write a kernel module. Here's a simple example:

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

   static int __init my_module_init(void) {
       printk(KERN_INFO "Load Message from YOUR_STUDENT_NUMBER.\n");
       return 0;
   }

   static void __exit my_module_exit(void) {
       printk(KERN_INFO "Unload Message from YOUR_STUDENT_NUMBER.\n");
   }

   module_init(my_module_init);
   module_exit(my_module_exit);
   ```

2. **Compile the Module:** Add the module to the kernel's build system. You can do this by placing your module source in an appropriate directory(drivers/misc) within the kernel source tree, and updating the 'Makefile' to include your module for compilation and Kconfig to have an option in menuconfig.
3. **Configure the Kernel:** When configuring your kernel (using 'make menuconfig'), ensure that the option to include your module is star(*), not *m*, since we want that the module be part of the linux kernel image.
4. **Recompile and Boot:** Recompile your kernel, boot your VM with this kernel, and verify that your module is loaded and the message is printed during boot.

These tasks provide hands-on experience with kernel modification and module development, enhancing your understanding of the Linux kernel's architecture and operation.

2.4 | Skeleton

The submission for this coursework should be organized in a specific structure to ensure correct compilation and ease of evaluation. The submission must be a compressed zip file named ‘CW0_STUDENTNUMBER.zip‘ (for example, ‘CW0_S1234567.zip‘). The ‘S‘ in the student number should be capitalized. The zip file should contain two folders, each corresponding to one of the tasks. The folder names are case-sensitive.

- **Task 1 Folder:** In the folder named ‘Task1‘, you should include the following files:
  1. A ‘.config‘ file which is your kernel configuration file.
  2. The modified ‘main.c‘ file. This file should be placed within a subfolder named ‘init‘ following the Linux kernel’s directory structure. For example, the path to ‘main.c‘ should be ‘Task1/init/main.c‘. Placing ‘main.c‘ directly in the ‘Task1‘ folder, or any other structure, will result in a failure to compile, and you will automatically receive a score of 0 for this task.
- **Task 2 Folder:** In the folder named ‘Task2‘, the following should be included:
  1. Your custom kernel module source code. Place this inside a folder named ‘drivers/misc‘. It is crucial that your module is correctly located in this folder for it to be compiled.
  2. A ‘.config‘ file, which is the configuration file for your kernel including your module.
  3. The ‘Makefile‘ used for compiling your module. This should be in the same directory as your module source code.
  4. The ‘Kconfig‘ used for adding new entry. This should be in the same directory as your module source code.

Please ensure that the directory structure and file names are exactly as specified. Failure to adhere to this structure will result in an automatic score of 0 for the respective task. It is crucial to follow these guidelines for your code to be compiled and assessed correctly.

3 | Coursework 1: Processes and Scheduling

3.1 | Introduction

In this coursework, you will explore and manipulate the various data structures that Linux uses to control processes, threads and make scheduling decisions. The goal of the coursework is to give you hands-on experience in development inside the Linux kernel.

This coursework contributes 17% to your total grade in this course.

A detailed breakdown of the score, along with the key dates associated with this coursework is listed in Chapter 3.2. Chapter 3.3 contains some instructions you must abide by for this coursework. The specifications for this coursework are listed in Chapter 3.4. Your implementation must conform to this list of specifications. Deviations from this list will be penalized. Chapter 3.5 lists out some essential reading material that constitutes essential background for this coursework.

3.2 | Overview

3.2.1 | Scoring

This Coursework has 3 tasks. They are modular, i.e. they will be scored independently of each other. The full specification for both parts is detailed in Chapter 3.4.

Table 3.1: Score Division

Component	**Weight**
Task 1	20
Task 2	30
Task 3	50
Maximum Score	100

3.3 | Instructions

3.3.1 | Base Kernel

We provide you a base kernel, upon which you shall make your modifications for the coursework. This kernel is Linux *v*6.10.0 with some changes. These changes are discussed in detail in Chapter 3.4. You can download this kernel in either of two ways:

- If you prefer to maintain your code as a `git` repository, you may download our patch file `cw1-base.patch` from https://os.systems-nuts.com/cw1-base.patch. You may apply the patch as a new commit to your Linux source tree repository by running the following command within the source folder.

```
USER@HOSTNAME:~/$ git clone --branch v6.10 https://github.com/torvalds/linux.git
USER@HOSTNAME:~/$ cd linux
USER@HOSTNAME:~/linux$ git am --keep-cr < /path/to/cw1-base.patch
```

- You can also download the entire base Linux kernel for this coursework as a `cw1-linux.zip` file from https://os.systems-nuts.com/cw1-linux.zip to the computer on which you will do the coursework, and unzip it there.

NOTE: This is not the same kernel that you used for Coursework 0. Please make sure you are working on the correct kernel source tree.

3.3.2 | Config Options

You must set all the config options that you will need for debugging and testing. Additionally, for this coursework, you must also set `CONFIG_SCHED_STATS`, `CONFIG_SCHED_DEBUG`, and `CONFIG_SMP`.

3.4 | Tasks

3.4.1 | Task 1: Whats My Ancestry?

System calls are function calls that allow user-space processes to communicate with the kernel. A system call is implemented inside the kernel, but it may be invoked from user-space. Each syscall is identified by a unique number.

In Task 1, you are expected to implement a new system call.
In the base kernel provided, in `arch/x86/entry/syscalls/syscall_64.tbl`, we have declared a new syscall with ID 463. This syscall is named `ancestor_pid` and is implemented by a function named `sys_ancestor_pid`. `sys_ancestor_pid` has been declared in `include/linux/syscalls.h`. It takes a *pid_t* variable and an *unsigned int* as argument, and returns a *long*. The definition of this syscall is incomplete - you can find it in `kernel/sched/core.c`. Your task is to complete this definition.

In Linux, a process can only spawn from other existing processes (via system calls such as *fork*, *execve*, and so on). When a process makes a syscall to spawn a new process, the new process is called the *child*, while the caller is the *parent* process. As a corollary, every process has a parent (though not necessarily a *live* one). An *ancestor* of a process is either that process itself (0^{th} order ancestor), or its parent (1^{st} order ancestor), or its parent's parent (2^{nd} order ancestor), and so on.

`sys_ancestor_pid` accepts a PID and an unsigned int n as its arguments. Let this PID belong to a process p. `sys_ancestor_pid` should return the PID of the n^{th}-order ancestor of p.

To ensure that you receive full credit for this Task, please abide by the following specifications.

1. If the input PID is zero, then the PID of the calling process is used.
2. If the input PID is negative, then return `-EINVAL` (Invalid Argument).
3. If any of the ancestors with order less than or equal to n do not exist, then return `-ESRCH` (No Such Process).
4. If n exceeds the order of the oldest ancestor, return `-ESRCH`.

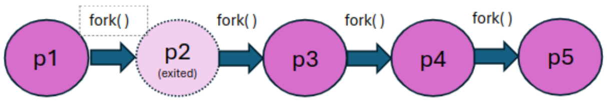


Figure 3.1: P1 forks P2, P2 forks P3, P3 forks P4, P4 forks P5. P2 has exited, while rest are still alive.

As an illustration, consider Figure 3.1. The following should hold.
`sys_ancestor_pid`($p5$, 0) = $p5$.
`sys_ancestor_pid`($p5$, 1) = $p4$.
`sys_ancestor_pid`($p5$, 2) = $p3$.
`sys_ancestor_pid`($p4$, 2) = `-ESRCH`.
`sys_ancestor_pid`($p4$, 3) = `-ESRCH`.

3.4.2 | Task 2: Trickle-Down Niceness

The Linux Scheduler has the important task of scheduling jobs (threads) on CPU cores. To determine which jobs must be prioritized, a *priority* is assigned to each. *Normal* jobs have a priority value between 100 and 139 (both inclusive). This corresponds to a *niceness* value between -20 and 19. A higher *niceness* means the job is of less importance, and vice versa. You can find the niceness of any process using the `getpriority` system call. You can also modify increment the niceness of the calling process with the `nice` system call.

In Task 2, you must implement yet another system call. This system call is named `propagate_nice`. Its ID is 464 (see `arch/x86/entry/syscalls/syscall_64.tbl`), and has been declared as `sys_propagate_nice` in `include/linux/syscalls.h`. It takes a single *int* argument n, and returns a *long*. Its implementation in `kernel/sched/core.c` is incomplete. You must complete this.

The system call's functionality is as follows: When a process makes this system call, its niceness is increased by the given value (exactly like the `nice` syscall). Additionally, this increment in niceness must trickle down to all its live descendants, halving across each generation. However, if any of the caller's descendants do not exist anymore, then *its* descendants will not be affected.

The syscall can be considered successful if it manages to change the niceness of at least one process, in which case, it must return 0. In all other cases, it is considered a failure, in which case it must return a negative integer.

If the increment is negative, the syscall must fail.

As an example, consider Figure 3.1.
If `sys_propagate_nice`(3) is called from $p3$, then the nice value of $p3$ increases by 3, that of $p4$ increases by 1, while $p5$ is unaffected. If `sys_propagate_nice`(2) is called from $p1$, then the nice value of $p1$ increases by 2, while no other process is affected (since $p2$ has exited, there is no route to its further descendants).

Sample output showing the effect of `sys_propagate_nice`:

Before syscall	**After syscall**
`pid1:1273, pid2:-1, ni:0`	`pid1:1273, pid2:-1, ni:5`
`pid1:0, pid2:1274, ni:0`	`pid1:0, pid2:1274, ni:2`
`pid1:0, pid2:0, ni:0`	`pid1:0, pid2:0, ni:1`

3.4.3 | Task 3: Scheduling History

The *proc* file system acts as an interface to internal data structures in the kernel. It can be used to obtain information about the system and to change certain kernel parameters at runtime. On a live Linux system, go to `/proc/` to checkout the various entries. You will find files which contain information, such as `/proc/stat`, `/proc/vmstat`, and so on. In addition, you can find process-specific information inside `/proc/<PID>/`.

In particular, take a look at the contents of `/proc/<PID>/schedstat`. It shows 4 pieces of process-specific information. The first is the total time (in nanoseconds) that this process spent executing on a CPU. The second is the total time (in nanoseconds) that this process spent waiting in a runqueue. The third is the number of timeslices run on this current CPU. The fourth piece of information (enclosed by square brackets) should be the list of CPUs that this process was scheduled on in this *epoch* (an *epoch* is defined as a period of 10 seconds of runtime). However, the functionality pertaining to this piece of information is yet unimplemented. Your task is to implement this.

We have modified `struct task_struct` inside `/include/linux/sched.h` in the base kernel, to add two new members:

1. `cpumask_t used_cpus` is a bit-mask that should store the list of all CPUs on which this process got scheduled. To manipulate `cpumask_t`, you should make use of the functions declared in `include/linux/cpumask.h`.
2. `unsigned int epoch_ticks` is a cyclic counter that should increment at every scheduler tick, and resets to zero at the start of a new epoch (10 seconds of runtime, counted from the start of the program). Time spent waiting on the runqueue is not included in the epoch. When a new epoch starts, `used_cpus` must be updated by removing all the CPUs that the task previously ran on in the last epoch.

You are expected to use the aforementioned variables to capture the required information. The contents of `used_cpus` will automatically be fetched into `/proc/<PID>/schedstat`.

3.5 | Helpful Reading

The following are some resources that you can use to improve your practical understanding of scheduling, required for this coursework.

1. Sched-Stats `Documentation/scheduler/sched-stats.rst`

3.6 | Submission

First you must create a folder named `CW1_UUN` where `UUN` is your university ID (eg. $S1234567$). This folder must contain one file as shown below.

```
USER@HOSTNAME:~$ ls CW1_S1234567/
core.c
```

The file `core.c` is the modified file `kernel/sched/core.c` from the Linux source code pertaining to your submission. You should restrict all your modifications to this singular file.

You must then create a `.zip` archive of this folder as shown below.

```
USER@HOSTNAME:~/CW1_S1234567$ cd ..
USER@HOSTNAME:~$ zip -r CW1_S1234567.zip CW1_S1234567
```

This `.zip` file should be submitted on the Learn portal before the deadline.

4 | Coursework 2: Memory Subsystem

4.1 | Introduction

In this coursework, you will be tasked with two main objectives designed to enhance your understanding and practical skills in managing and manipulating the memory subsystem of the Linux kernel. The first task involves collecting statistics about page table operations across the page table hierarchy, while the second task focuses on tracking and reporting different types of page faults on a per-process basis. Through these tasks, you will gain hands-on experience with critical components of the Linux kernel's memory management, including but not limited to *virtual memory*, *physical memory*, *page tables*, *memory mapping*, *memory allocation*, *page walking*, and *fault handling mechanisms.*

The tasks will require you to modify specific kernel files to implement counters, track events, and expose the collected statistics through the `/proc` filesystem. This practical experience will deepen your understanding of how modern operating systems manage memory resources and handle memory-related exceptions, providing valuable insights into the complex interactions between user processes and the kernel's memory subsystem.

This coursework contributes 17% **to your total grade in this course.**

4.2 | Scoring

This Coursework has 2 tasks - Task 1 and Task 2. The 2 tasks are modular, i.e. they will be scored independently of each other. The implementation of Task 1 does not depend on that of Task 2 in any manner. Even within each task, there exist modular components that are graded independently. This structure allows you to secure partial marks by completing part of the coursework in case you are unable to complete the full coursework. The full specification for both tasks is detailed in Section 4.3.

Table 4.1: Score Division

Component	**Weight**
Task 1: Memory Management Statistics	7
Task 2: Fault Statistics	10
Maximum Score	17

4.3 | Specification

4.3.1 | Base Kernel

You shall implement this coursework upon the original **Linux v6.13** kernel. We will not provide you a modified base kernel for this coursework. You will have to make all changes required to implement the specification by yourself on this kernel. Please ensure that you are working on the correct kernel source.

4.3.2 | Task 1: Memory Management Statistics

In Task 1, you are required to collect statistics on the number of actions performed on a page table for each process. To translate a virtual address (VA) to a physical address (PA), the operating system must execute a "page table walk" — a series of in-memory translation look-ups. Linux implements a 4-level page table hierarchy for x86-64 architectures. During virtual address translation, the virtual address is partitioned into multiple fields, each corresponding to a specific level in the page table hierarchy:

1. PGD (Page Global Directory) Offset: Indexes into the PGD.
2. PUD (Page Upper Directory) Offset: Indexes into the PUD.
3. PMD (Page Middle Directory) Offset: Indexes into the PMD.
4. PTE (Page Table Entry) Offset: Indexes into the PTE.
5. Offset within Page: Determines the exact byte within the physical page.

Figure 4.1 illustrates the page table walk process. The first VA offset indexes into the PGD, which is unique for each process. The value retrieved from this lookup points to the corresponding PUD. The PUD offset derived from the virtual address is then used to index an entry from this PUD, which points to the corresponding PMD. This pattern continues through the hierarchy until the complete translation from virtual address to physical address is obtained.

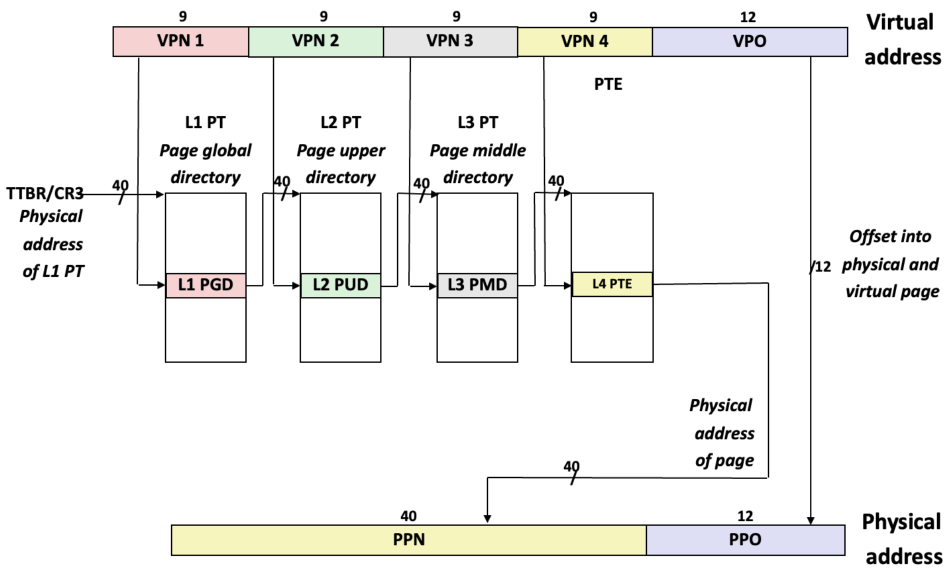

9
VPN 1
9
VPN 2
9
VPN 3
9
VPN 4
12
VPO
Virtual address
PTE
L1 PT Page global directory
L2 PT Page upper directory
L3 PT Page middle directory
TTBR/CR3 Physical address of L1 PT
40
L1 PGD
L2 PUD
L3 PMD
L4 PTE
12
Offset into physical and virtual page
Physical address of page
40
PPN
12
PPO
Physical address

Figure 4.1: Virtual to physical address translation.

Your task is to collect the number of times a page table action has been performed for each pid. The page actions which can be performed are allocate, free, and set, and you need to collect the number of times each of these tasks has been performed for each level of the page table hierarchy (PGD, PUD, PMD, PTE) resulting in $3 * 4 = 12$ values total. To achieve this, you will need to modify the following files:

1. Define the action counter variables inside the `task_struct` struct in file `include/linux/sched.h`. You should have a separate counter for each of the 12 values (3 actions × 4 page table levels).
2. Identify and modify the appropriate kernel functions to collect statistics by incrementing the counters you defined. You should look for functions that:
   - Allocate page table entries at each level (PGD, PUD, PMD, PTE)
   - Free page table entries at each level
   - Set(update) page table entries at each level

   These functions are typically found in various memory management files within the kernel source, such as files in the `mm/` directory, architecture-specific memory management code in `arch/x86/mm/`, and relevant header files in `include/`.
3. Create a new kernel module in `fs/proc/pg_stats.c` that creates a `/proc/pg_stats` entry. You will need to implement a complete procfs module including the necessary initialization function, file operations structure, and a show function that displays the statistics for each process. Your module should use the `seq_file` interface and iterate through all processes to display their page table operation counters in the specified format.

After you kernel compiles, the output can be viewed by running `cat /proc/pg_stats`, which will print a series of lines. Each line represents a pid, followed by the 12 collected counters in the following order:

```
[pid]: [[pgd_alloc],[pgd_free],[pgd_set]], [[pud_alloc],[pud_free],[pud_set]], [[pmd_alloc],
       [pmd_free],[pmd_set]], [[pte_alloc],[pte_free],[pte_set]]
```

Your log should display all the currently active processes in the system. Each line should contain the process ID followed by the 12 counter values as specified in the output format.

To test if your implementation works correctly, you can run a simple test program that performs memory operations. For example, create a C program that allocates and accesses memory in various ways.

Compile and run this program, then examine the output of `cat /proc/pg_stats` while it's running. You should see your test process listed with non-zero counter values for various page table operations. After the program exits, run `cat /proc/pg_stats` again to verify that your process is no longer listed, confirming that only currently active processes are shown.

4.3.3 | Task 2: Fault Statistics

In this task, you will capture per-process statistics pertaining to page faults and expose it to the user via the `/proc/<PID>` interface. There are 5 fault types that are of interest to us.

1. **Write** faults happen on an illegal store to the faulting address.
2. **User** faults are triggered by a memory access in the userspace.
3. **Instruction** faults are triggered by an instruction fetch.
4. **COW** or Copy-On-Write fault happens when a Copy-On-Write page is written to. The fault arising in this case is handled by copying the contents of the original page into a new page with updates (a.k.a. the pending writes).
5. **Mlocked** faults are faults that happen on a page that has been locked using the `mlock` system call.

You must add a file entry named `/proc/<PID>/fault_stats` that contains a list of fault-types and the number of faults of each type encountered by the process since its creation.

The following is an example of how it should look like

```
root@systems-nuts:/proc/1234# cat fault_stats
write 137651
user 200010
instruction 13009
cow 0
mlocked 0
```

To complete this task, you are asked to modify the following files.

1. In `fs/proc/base.c`, implement the new `/proc/<PID>/fault_stats` file.
2. In `include/linux/sched.h`, implement the counters that hold the per-task counts of the different fault types.
3. In `mm/memory.c`, you will find the default page fault handling methods. You may modify these handlers so that they update the counts of each fault type when encountered.

You must restrict your modifications to these three files only.

4.4 | Submission Structure

The submission for this coursework should be organized in a specific structure to ensure correct compilation and ease of evaluation. The submission must be a compressed zip file named `CW2_STUDENTNUMBER.zip` (for example, `CW2_S1234567.zip`). The 'S' in the student number should be capitalized. The zip file should contain two folders, each corresponding to one of the tasks. The folder names are case-sensitive.

- **Task 1 Folder:** In the folder named `Task1`, you should include all modified files while preserving their original path structure. For example:
  1. If you modified `include/linux/sched.h`, include it as `Task1/include/linux/sched.h`
  2. If you modified `arch/x86/mm/pgtable.c`, include it as `Task1/arch/x86/mm/pgtable.c`
  3. Include your new file `fs/proc/pg_stats.c` as `Task1/fs/proc/pg_stats.c`
  4. Include any other files you modified with their full path structure preserved
- **Task 2 Folder:** In the folder named `Task2`, you should include all modified files while preserving their original path structure:

1. Your modified `fs/proc/base.c` file as `Task2/fs/proc/base.c`
2. Your modified `include/linux/sched.h` file as `Task2/include/linux/sched.h`
3. Your modified `mm/memory.c` file as `Task2/mm/memory.c`

Please ensure that the directory structure and file names are exactly as specified. Failure to adhere to this structure will result in an automatic score of 0 for the respective task. It is crucial to follow these guidelines for your code to be compiled and assessed correctly.

Note: Only include the files you have modified. Do not include the entire kernel source tree or any binary files.