THE UNIVERSITY of EDINBURGH

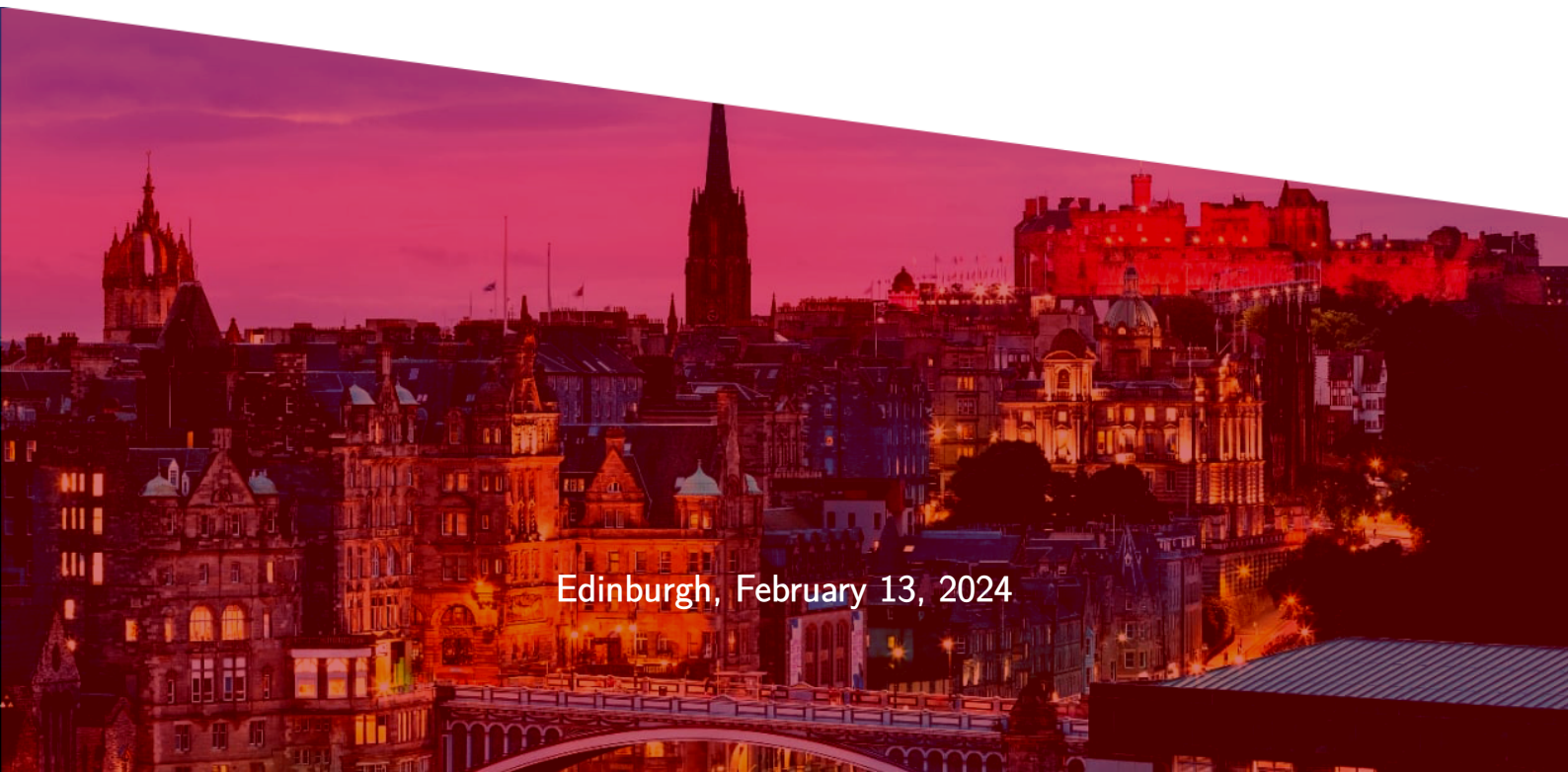
Operating Systems Courseworks 2023-2024

Course Number: INFR100792023

Semester Number: 2

Score Out of 100: 50%

Authors: Amir Noohi, Alan Nair



Edinburgh, February 13, 2024

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	30/01/2024	13/02/2024	N/A
CW1	17%	16 Hours	13/02/2024	05/03/2024	19/03/2024
CW2	17%	16 Hours	05/03/2024	19/03/2024	02/04/2024
CW3	16%	16 Hours	19/03/2024	02/04/2024	16/04/2024

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 Coder." 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.

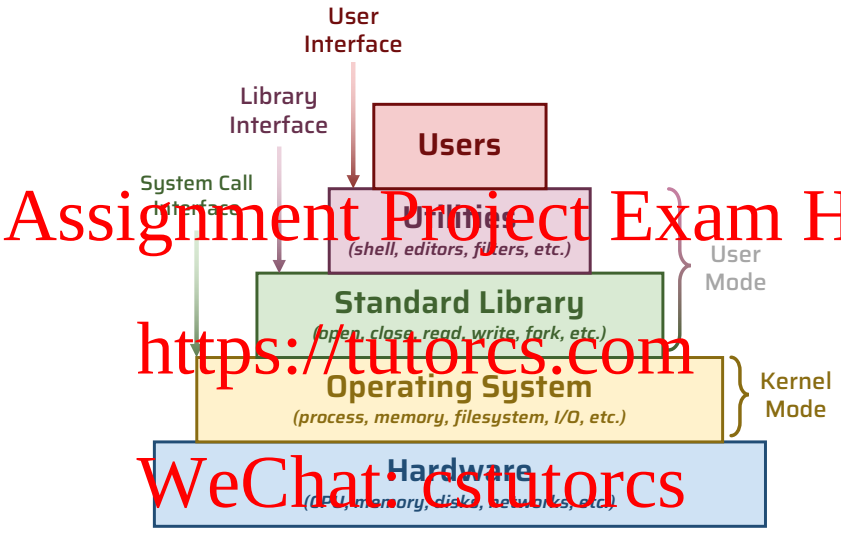


Figure 1.1: Computer System Diagram

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.

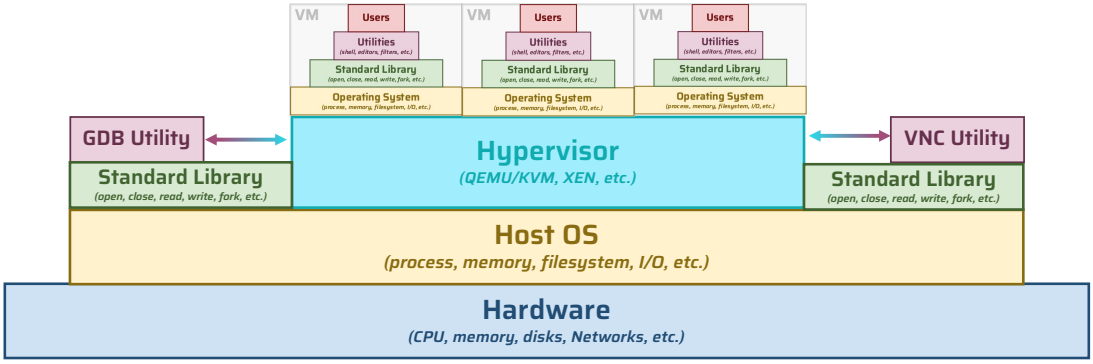


Figure 1.2: Virtualized System Diagram

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 2.

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 3.

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 | Environment Setup

2.1 | Introduction

Welcome to an essential and exciting aspect of your journey in operating system coursework! In this chapter, we're going to delve into the fascinating world of testing environments. Understanding where and how to test your code is not just a step in learning; it's an adventure in problem-solving and creativity.

In the dynamic field of Linux kernel development, there are several approaches to explore. One effective and straightforward method is through virtualization. This section is dedicated to guiding you through this process, focusing on the utilization of University infrastructure.

While this document emphasizes using the university's infrastructure, it's essential to acknowledge the flexibility in your approach to coursework. If you have access to your own infrastructure, you may consider skipping this chapter entirely. Remember, since university resources are shared, running projects on your local infrastructure could be faster and more efficient. You're encouraged to explore and utilize the resources that best suit your learning style and project requirements. Whether you opt for the university's high-performance resources and standardized environment or set up a virtual environment on your personal system, all coursework will be evaluated equally. This ensures that regardless of the method chosen, you have the opportunity to achieve full marks. Despite this, please note that support and guidance will be primarily available for DICE setups.

For students opting to use their personal machines, the process involves a few essential steps. Firstly, it's recommended to install a virtualization tool like VMware or VirtualBox. These tools allow you to create and manage virtual machines (VMs), which provide isolated environments for kernel development without affecting your main operating system. After setting up the virtualization software, the next step is to create a VM and install a Linux distribution on it, such as Ubuntu, Fedora, or Debian, which are popular choices for development work.

Once your Linux environment is ready within the VM, the next task is to install the prerequisites for compiling the Linux kernel. This typically includes development tools such as the GNU Compiler Collection (GCC), make, and the ncurses library, among others. After installing these tools, you can proceed to download the Linux kernel source code and begin the exciting process of compiling and testing. This approach offers the flexibility of a customizable development environment on your machine while ensuring that your primary operating system remains unaffected by the development activities. It's a great way to learn about system administration and kernel development in a controlled and safe environment.

2.2 | Using DICE Servers

DICE, which stands for Distributed Informatics Computing Environment, is the University's specialized computing environment designed to support your journey in Linux kernel development. This platform offers two primary ways for engaging with coursework: using dedicated lab machines or accessing the environment remotely.

If you haven't already created a DICE account, the instruction available at https://computing.help.inf.ed.ac.uk/accounts provides guidance on how to do so. This account is essential for accessing either the physical lab machines or the remote desktop service.

2.2.1 | Lab Machines in Appleton Tower

Appleton Tower is equipped with over 300 lab machines, all set up with DICE. These machines are readily available for in-person use and have been configured with everything you need for your coursework. This option provides a hands-on experience and direct interaction with the tools and resources necessary for Linux kernel development.

2.2.2 | Remote Desktop Access

Despite the availability of lab machines, we recommend using the School of Informatics' remote desktop service for your coursework. Opting for remote access aligns with the coursework instructions and ensures a consistent experience across the student body. Remote access allows you to work flexibly from any location, providing convenience and continuity in your learning process.

For detailed instructions on establishing a connection to the DICE remote desktop server, tailored to various operating systems, please refer to https://computing.help.inf.ed.ac.uk/remote-desktop. Additionally, it's important to note that if you are outside the university's network and unable to connect to the remote desktop server, you must use a VPN. The link provided includes comprehensive instructions on how to set up and use the university's VPN service.

Once you have connected to the DICE remote desktop server, it's important to note that this environment itself does not support virtualization. Therefore, you will need to connect further to specific lab machines or the student compute server for more advanced capabilities. In this guide, we will walk you through using the student compute server for this purpose(https://computing.help.inf.ed.ac.uk/compute-servers).

2.2.3 | DICE File System

Upon connecting to any server on DICE, you'll find yourself within the AFS (Andrew File System), a shared file system used across all servers. It's crucial to be aware of the space limitations associated with using the AFS, as outlined in https://computing.help.inf.ed.ac.uk/afs-quotas. These limitations mean that AFS may not be suitable as the final location to set up your virtual machine.

For more extensive storage needs, such as storing your virtual machine image, we recommend using the scratch space available on DICE desktops or servers. Detailed guidance on how to utilize this scratch space effectively can be found at https://computing.help.inf.ed.ac.uk/scratch-space. This resource will instruct you on how to use disk scratch space to store your virtual machine image and ultimately bring up your virtual machine for your coursework.

2.3 | Getting Ready Your Workspace

Once you've successfully connected to the DICE remote desktop server, the next step is to connect to the student compute server or whatever lab machine you are using. for example for student computer server you can use below command:

```
ssh student.compute
```


After connecting to appropriate machine you need to create a directory in `/disk/scratch`. for doing this first check if `/disk/scratch` exists and then create a folder under your student number like below:


```
mkdir /disk/scratch/sXXXXXXXX
```

Please note that we will not inspect files in your folder on `student.compute`. Your coursework will be evaluated based on what you submit to the LEARN platform. Additionally, rest assured that other users will not have access to your files here.

For a more streamlined setup, a bash script is available at https://os.systems-nuts.com/pre.sh. This automated script ensures that you have all the required files in the right place, allowing you to focus more on your coursework and less on setup logistics. You can run this bash script easily using the below command:

```
bash <(curl -sL https://os.systems-nuts.com/pre.sh)
```

The output of running this command in a DICE remote desktop server is shown below:

```
[*] Checking for /disk/scratch...
[+] Directory /disk/scratch exists.
[*] Checking for your directory /disk/scratch/sXXXXXXX...
[*] Creating your directory /disk/scratch/sXXXXXXX...
[+] Directory /disk/scratch/sXXXXXXX created successfully.
[+] Changed to directory /disk/scratch/sXXXXXXX.
[*] Downloading files...
debian.tar.xz      100%[===================>] 344.92M  44.9MB/s    in 9.0s
[+] Downloaded: debian.tar.xz
stop.sh            100%[===================>]   1.23K  --.-KB/s    in 0s
[+] Downloaded: stop.sh
[*] Extracting debian.tar.xz...
344MiB 0:00:14 [24.4MiB/s] [============================>] 100%
[+] Extracted debian.tar.xz
[+] Removed existing debian.tar.xz
[+] Preparation Completed.
```

As demonstrated in the output, the *pre.sh* script automates the setup of your working environment by doing below steps:

1. Check if the current machine has *qemu* virtualization feature.
2. Check if the directory */disk/scratch/* exists. If it does, it will create a folder for you within this directory.
3. If */disk/scratch* doesn't exist, it will create your folder in the current directory.
4. Download and extract all necessary coursework files into your newly created folder.

2.4 | Running Your VM

Congratulations! At this point, you have successfully set up your working environment. For running your VM, you need multiple files as below:

1. **File System Image:** A virtual hard disk for your VM, containing the operating system and files. which you've already downloaded(called debian.qcow2)
2. **Kernel bzImage:** The compressed Linux kernel image.

The command to run your VM is as follows:

```
qemu-system-x86_64 -m 4G -smp 4 -drive file=path/to/filesystem.qcow2 -kernel path/to/bzImage -append 'root
    =/dev/sda1 console=ttyS0' -nographic
```

Let's break down this command:

- **qemu-system-x86_64:** The QEMU emulator for 64-bit x86 systems.
- **-m 4G:** Allocates 4 GB of memory to the VM.
- **-smp 4:** Assigns 4 virtual CPU cores to the VM.
- **-drive file=path/to/filesystem.qcow2:** Specifies the path to the file system image.
- **-kernel path/to/bzImage:** Points to the bzImage, the compressed Linux kernel image.
- **-append 'root=/dev/sda1 console=ttyS0':** Adds kernel command line parameters.
- **-nographic:** Runs the VM without a graphical interface.

For accessing the provided Debian image, use the following credentials:

- **Username:** root
- **Password:** debian

2.5 | Configuring and Compiling the Linux Kernel

As you may have noticed, you have everything except linux kernel bzImage. For having bzImage, You should follow these steps to configure and compile the Linux kernel. This process will prepare the kernel for use in a virtualized environment, such as QEMU, and is optimized for x86 architecture.

2.5.1 | Downloading Linux Kernel

The following instructions detail the process for downloading and extracting Linux kernel version 6.1.74:

1. Visit the Linux kernel archive at https://www.kernel.org/ and download the *linux-6.1.74.tar.xz* tarball.
2. Use the command *tar -xvf linux-6.1.74.tar.xz* to extract the tarball. This will create a directory named *linux-6.1.74* with the kernel source code.

2.5.2 | Creating the Default Configuration File

Start by creating a default configuration file that is tailored for x86 machines. This can be done using the following command:

```
make x86_64_defconfig
```

This command generates a '.config' file with default settings suitable for x86_64 architectures.

2.5.3 | Adjusting Configuration for a KVM Guest

After creating the default configuration, further refine it for running as a KVM guest:

```
make kvm_guest.config
```

This step adjusts the kernel configuration to optimize it for running in a virtualized environment.

2.5.4 | Disabling Unrelated Device Drivers

Next, disable any unrelated device drivers, such as GPU or sound card drivers, which are not necessary for this coursework. This is done using the 'make menuconfig' interface:

```
make menuconfig
```

Navigate through the menu to disable drivers that are not needed. This helps in reducing the kernel size and compilation time.

2.5.5 | Compiling the Kernel

Once the configuration is complete, compile the kernel. Utilize all available CPU cores to speed up the compilation process:

```
make -j$(nproc)
```

Note: The compilation time can vary significantly, ranging from 1 minute to 5 minutes or more, depending on the specifications of the machine/server you are using.

2.5.6 | Output of Compilation

When you compile the Linux kernel, several output files are generated. Each of these files has a specific role, particularly when it comes to testing and debugging. After running the 'make -j$(nproc)' command, the following are the key output files you should expect:

- **bzImage:** The most significant output for our purposes is the 'bzImage' (Big ZImage), located in the 'arch/x86/boot/' directory. This is the compressed kernel image that you will use with QEMU for booting your VM. The path to this file is typically 'arch/x86/boot/bzImage'.
- **vmlinux:** This is the uncompressed kernel image. It contains the debugging symbols and is crucial for debugging purposes. You'll find it in the root of the build directory.

- **Modules:** If your kernel configuration includes loadable kernel modules, these will be compiled as well. The compiled modules are located in the 'lib/modules/' directory, categorized by their respective kernel version.
- **System Map:** The System.map file, usually found in the root of the build directory, contains the addresses of all the symbols in vmlinux. This file is useful for debugging and understanding the kernel layout.
- **Headers:** The kernel headers, found in the 'include/' directory, are used for building kernel modules and are essential for developers who are writing or compiling external kernel modules.
- **Configuration File:** A copy of the '.config' file used for compilation is often saved along with the kernel, usually in the build directory, for reference and archival purposes.

Note: The exact locations and presence of these files might slightly vary depending on the configuration options you chose during the 'make menuconfig' step. The compilation time for these files can range from a minute to several minutes, depending on the server's specifications and the complexity of your kernel configuration.

After the compilation, you will primarily use 'bzImage' for booting your VM with QEMU and 'vmlinux' for debugging purposes with GDB. Remember to verify the kernel version with 'uname -r' after booting the VM to ensure that your compiled kernel is running.

2.6 | Using QEMU Monitor

The QEMU Monitor, often referred to simply as the "Monitor," is a powerful command-line interface for managing and interacting with your virtual machine (VM) during runtime. It allows you to control various aspects of your VM, such as pausing, resuming, taking snapshots, and changing configurations. In this subsection, we'll explore how to access and utilize the QEMU Monitor. You can access the Monitor interface by pressing `'Ctrl' + 'a'` followed by `'c' (Ctrl-a, c)`. To return to the VM's serial console, do the same.

When you access the serial console in your VM, you can use the following keyboard shortcuts to perform various actions:

- **CTRL + a h**: Print this help
- **CTRL + a x**: Exit the emulator
- **CTRL + a s**: Save disk data back to a file (if using the '-snapshot' option)
- **CTRL + a t**: Toggle console timestamps
- **CTRL + a b**: Send a break (magic sysrq)
- **CTRL + a c**: Switch between the console and Monitor interface
- **CTRL + a CTRL + a**: Sends `'CTRL + a'` to the VM

When working with the Linux kernel using QEMU, the QEMU Monitor provides a set of powerful commands and options to assist in debugging and analyzing the virtual machine's state. The Monitor is an interactive shell that allows users to control the QEMU virtual machine, inspect its state, and perform various management tasks. For those looking to delve deeper into the capabilities of the QEMU Monitor, a comprehensive list of available commands can be found in the official QEMU documentation. To explore these commands and learn more about their functionalities, you can visit the following link: QEMU Monitor - System Documentation.

2.7 | Debugging the Linux Kernel

There are many great tools that are useful for debugging the Linux kernel, including good old-fashioned printk, ftrace, and kgdb. In this section we'll be exploring how to use the kernel debugger (kgdb) to debug a QEMU VM, although some of the techniques below may be applied to debugging via hardware interfaces like JTAG. Using gdb as a front-end for the kernel debugger allows us to debug the kernel in the familiar and powerful debugging interface of gdb.

2.7.1 | Building Kernel

First, configure your kernel for debugging. Enable the following options in the kernel configuration:

- `CONFIG_DEBUG_INFO`: Essential for including debug information.
- `CONFIG_DEBUG_INFO_SPLIT`: Reduces the size of the kernel image by splitting debug info.
- `CONFIG_GDB_SCRIPTS`: Adds useful GDB helper scripts.
- `CONFIG_KGDB`: Enables the built-in kernel debugger for remote debugging.
- `CONFIG_FRAME_POINTER`: Improves reliability of backtraces. Necessary for certain architectures.

If KASLR is enabled (`CONFIG_RANDOMIZE_BASE=y`), it will affect setting breakpoints. To ensure effective debugging, disable KASLR or add 'nokaslr' to the kernel command-line parameters.

2.8 | Running the VM with QEMU for Debugging

Use QEMU with specific parameters for GDB debugging:

- `-gdb tcp::[port]`: Specify the gdbserver port.
- `-S`: Freeze CPU at startup for early kernel debugging.
- `-kernel [path]`: Path to the kernel image.
- Include 'nokaslr' in command-line parameters if KASLR is enabled.

2.9 | Setting Breakpoints and Debugging with GDB

1. Start GDB and load kernel symbols: `gdb ./vmlinux`.
2. Attach GDB to the VM: `(gdb) target remote :[port]`.
3. Set a hardware-assisted breakpoint at `start_kernel`: `(gdb) hbreak start_kernel`.
4. Resume execution: `(gdb) continue`.

2.10 | Stopping Your VM:

In some cases you may want to stop your running QEMU Virtual Machines (VMs) by force. Using the provided bash script called *stop.sh*. This script will detect all running VMs, index them, and allow you to stop a specific VM by entering its index. This can be especially useful when you are using the curses console, as stopping it through a single terminal can be challenging.

Here's sample output of `stop.sh`:

```
[*] Detecting running QEMU Virtual Machines...
[0]  qemu-system-x86_64 -m 4G -smp 4 -drive file=debian.qcow2
-display none -display curses (PID: 546470)
Enter the index of the VM to stop: 0
[+] Sent stop signal to VM with PID 546470.
```

As shown in the example above, *stop.sh* will display a list of running VMs with their corresponding indices. To stop a VM, simply enter the index of the VM you wish to stop, and the script will send the stop signal to that VM's process, effectively halting it.

3 | Coursework0

3.1 | 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.

3.2 | Background

For this coursework, you will work with the Linux kernel version 6.1, the latest long-term support (LTS) version.

Important aspects to consider:

- **Linux Kernel Version:** You should use Linux kernel version 6.1, ensuring you work with the latest LTS release.
- **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.

3.3 | 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.

3.3.1 | 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"
```


3.3.2 | 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:

```c
#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.

3.4 | 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.

4 | Coursework 1: Scheduling Policies

4.1 | Introduction

In this coursework, you will implement some new scheduling policies on Linux. The goal of the coursework is to give you hands-on experience in development inside the Linux kernel. The relevant topics covered are - *Processes*, *Threads*, *Scheduling*, and *Synchronization* (basic).

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 4.2. Chapter 4.3 contains some instructions you must abide by for this coursework. The specifications for this coursework are listed in Chapter 4.4. Your implementation must conform to this list of specifications. Deviations from this list will be penalized. Chapter 4.5 provides an overview of the infrastructure we use to test your code, and Chapter 4.6 has hints and examples that will help you with some aspects of this coursework. Chapter 4.7 lists out some essential reading material that constitutes essential background for this coursework.

4.2 | Overview

4.2.1 | 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 part, 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 parts is detailed in Chapter 4.4.

Table 4.1: Score Division

Component	Weight
Task 1	10
Task 2	7
Maximum Score	17

4.3 | Instructions

4.3.1 | Base Kernel

We provide you a base kernel, upon which you shall make your modifications for the coursework. This kernel is Linux $v6.1$ with some changes. These changes are discussed in detail in Chapter 4.4.1. 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 repository by running the following command within the source folder.

```
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.

4.3.2 | Submission

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

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

The files `core.c` and `fair.c` are the modified files `kernel/sched/core.c` and `kernel/sched/fair.c` from the Linux source code about your submission. You must contain all your work for the coursework within these two files.

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.

4.3.3 | Untamperable Segments

Some lines in the base kernel are marked with a comment "*CW1 - DO NOT TAMPER*". The following is an example of such a code segment in `kernel/sched/fair.c` in the base kernel.

```
/* CW1 - DO NOT TAMPER - SEGMENT START */
struct task_struct *curtask;  /* CW1 - DO NOT TAMPER */
struct sched_entity *curr = cfs_rq->curr; /* CW1 - DO NOT TAMPER */
[...]
if (unlikely((s64)delta_exec <= 0)) /* CW1 - DO NOT TAMPER */
    return; /* CW1 - DO NOT TAMPER */
/* CW1 - DO NOT TAMPER - SEGMENT END */
```

You must not modify the functionalities that exist within this segment. You are forbidden from editing any line that ends with a comment saying "*CW1 - DO NOT TAMPER*". You may add new lines within the segment, only if all they do is declare new variables and initialize them to compile-time constants. Nevertheless, it is best to restrict your modifications to outside such 'untamperable' segments.

Additionally, there are some variables that contain the string "*DO_NOT_TOUCH*" in their names. The following is an example from `kernel/sched/fair.c`.

```

u64 DO_NOT_TOUCH_CW1, delta_exec; /* CW1 - DO NOT TAMPER */
[...]
DO_NOT_TOUCH_CW1 = delta_exec;  /* CW1 - DO NOT TAMPER */
/* CW1 - DO NOT TAMPER - SEGMENT END */
```


You are prohibited from reading, or writing to these variables even outside the 'untamperable' segments. We use these variables to test your code (see Chapter 4.5.3), therefore please do not tamper with these.

All violations of these aforementioned rules shall be penalized ruthlessly.

4.4 | Specification

4.4.1 | Base Kernel

The base kernel we provide you is Linux `v6.1` with some modifications to facilitate your coursework.

We have defined three (unimplemented) scheduling policies, namely `SCHED_DONOR`, `SCHED_RECVR`, and `SCHED_AGING`. These policies correspond to policy numbers 7, 8, and 9. The policies are defined in `include/uapi/linux/sched.h` and `tools/include/uapi/linux/sched.h`. The `include/uapi` folder contains all user-facing APIs and macros of the Linux kernel, while `tools/include/uapi` contains the same for all the tools that are also a part of the Linux kernel [1].

In `kernel/sched/sched.h` the functions are included as instances of CFS scheduling policies in the function `fair_policy`, which returns true if its argument (policy number) is a policy implemented by fair scheduling class. `SCHED_DONOR`, `SCHED_RECVR`, and `SCHED_AGING` are unimplemented in the base kernel. CFS will treat tasks with these policies exactly as though they had the `SCHED_NORMAL` policy.

[1] https://lwn.net/Articles/507794/

Understanding the Linux Kernel

Question: How does the `fair_policy` function ensure that these new policies get associated with the *fair* scheduling class and not, say, the *idle* scheduling class ?

Hint:
The scheduling class for a task must be set when its policy gets changed, or when it starts.
Policy change happens through the `sched_setscheduler` system call. The system call is defined in `kernel/sched/core.c` (look for a line containing the strings "`SYSCALL_DEFINE`" and "`sched_setscheduler`"). Find the `sched_setscheduler` function in the same file, follow the chain of nested wrapper functions to the function `__sched_setscheduler`. Read this function carefully - look out for a function named `__setscheduler_prio`!
A task that starts must also have its scheduling class set before it begins. How is this handled? Check out `sched_fork`, `init_idle`, and all other instances where `sched_class` is set.

In `kernel/sched/fair.c`, we have provided three variables - `max_vruntime_unclaimed_ns`, `stale_request_ns`, and `flush_claimslist_ns`. They are needed for Task 2. These variables are declared as *extern* [2] variables in `kernel/sched/sched.h`. Their values represent time intervals in nanoseconds and are by default initialized to $10ms$, $100ms$, and $5s$ respectively. DebugFS entries have been created for these variables in `kernel/sched/debug.c`. These values can be configured from user-space (described in Chapter 4.5.1).

Additionally, we have modified the tracing function `sched_stat_runtime` in `include/trace/events/sched.h`. Tracing has been described in Chapter 4.5.3.

4.4.2 | Task 1: Aging Scheduler

In Task 1, you are asked to implement the `SCHED_AGING` policy.

A task whose policy is set to `SCHED_AGING` will behave identically to a `SCHED_NORMAL` task in all respects except when it undergoes a fork. When a `SCHED_AGING` forks, the nice value of that task (ie. the parent task after the fork) will increase by 1. In contrast, the child task will inherit the parent's old (pre-fork) nice-value. The child of a `SCHED_AGING` task must also be a SCHED_AGING task. Additionally, if a task's nice value is already the maximum possible nice value, it cannot increase on forking.

Figure 4.1 shows a schematic of how this policy works.

Understanding the Linux Kernel

Question: Where in the kernel code should you make the required modifications on fork for the Aging Scheduler ?

Hint:
In *kernel/fork.c*, find out where fork is declared as a syscall (search for a line that contains the strings "`SYSCALL_DEFINE`" and "`fork`"). Observe that it calls the function `kernel_clone` which is the main fork routine. `kernel_clone` eventually calls `copy_process`, which is a function that creates a new process as a copy of the old one, without starting it. `copy_process` calls `sched_cgroup_fork` which is defined in `kernel/sched/core.c`, and this function initializes the scheduler-related fields on fork. Understand carefully how the aforementioned functions operate. Do you see which is the best place to implement the `SCHED_AGING` functionality?

When you increment the process's nice value, ensure that all the weights used by CFS are updated correctly (see Chapter 4.6.3).

4.4.3 | Task 2: Vruntime Transfer Policy Pair

For Task 2, you have to implement two policies - `SCHED_DONOR` and `SCHED_RECVR`, namely the *Donor* and *Receiver* Scheduling Policies respectively.

A `SCHED_DONOR` task can effectively request CPU execution time from a `SCHED_RECVR` for itself. A `SCHED_DONOR` task will update its runtime statistics with just *half* of its actual CPU time. The remaining

[2] https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/

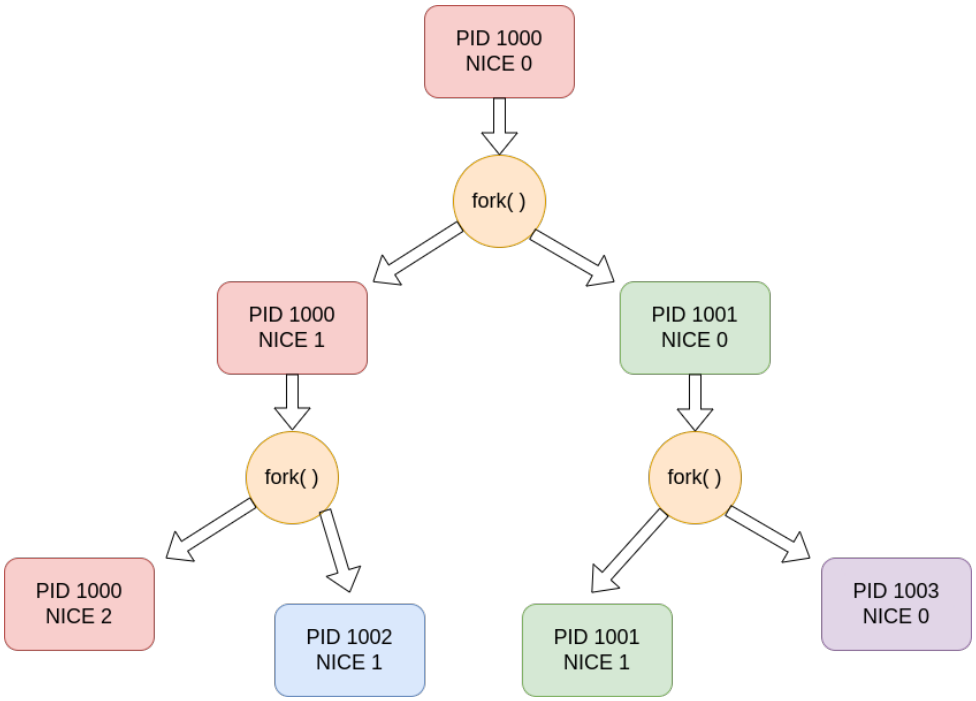


Figure 4.1: Schematic of `SCHED_AGING` functionality. Notice how the NICE value of a process increments on each fork.

CPU time is placed on adoption (a.k.a. 'donation'). A `SCHED_RECVR` task checks for outstanding runtime donations. If it sees one, it will claim the donated runtime (or part of it) for itself, i.e. it will update its own runtime statistics as if it ran for not just its own actual runtime, but also for the donated runtime from the `SCHED_DONOR` task.

For example, consider a `SCHED_DONOR` task A which tries to request CPU time from a `SCHED_RECVR` task B. After $10ms$ of running on the CPU, it experiences a switch to kernel mode. While updating its running statistics, specifically the variables `sum_exec_runtime` and `vruntime` of the field `struct sched_entity se` of A's `task_struct`, it updates the variables as if it only ran for $5ms$ of CPU time. The remaining $5ms$ is given up for adoption. Later, when Task B tries to update its runtime statistics, after running on CPU for $10ms$, it first checks if there are any unclaimed donations from some `SCHED_DONOR` task. It sees the $5ms$ time given up by A. Task B claims the $5ms$, i.e., it will clear the adoption request, and update its on runtime statistics as if it ran for $15ms$. In the end, even though both A and B actually ran for $10ms$ each, it appears as if A ran for $5ms$ and B ran for $15ms$. CFS's future scheduling decisions will try to balance the recorded runtimes of A and B, which will effectively grant A more actual CPU time than B.

The following are some specifications you must meet for this coursework.

1. No more than *HALF* of the actual CPU runtime of a `SCHED_DONOR` task may be donated away. In other words, after switching to `SCHED_DONOR` policy and running for 50 seconds, the recorded runtime of the task since the policy switch must be *at least* 25 seconds.
2. No more than *HALF* of the actual CPU runtime of a `SCHED_RECVR` task may be accepted from donations. In other words, after switching to `SCHED_RECVR` policy and running for 50 seconds, the recorded runtime of the task since the policy switch must be *at most* 75 seconds.
3. Neither donation, nor receiving can happen retrospectively. You must ensure that the act of offering away CPU time for donation, or receiving CPU time, should be done when the statistics are being updated. Once some runtime has been committed to the scheduler statistics (`sum_exec_runtime`), that runtime cannot later be donated/received.
4. A `SCHED_RECVR` task will only accept donations from tasks whose nice values are less than its own nice value.
5. A `SCHED_RECVR` task cannot accept donations from tasks with nice value x if there exist unclaimed donations from tasks with nice value less than x.

6. The global variable `max_vruntime_unclaimed_ns` defined in `kernel/sched/fair.c` is the maximum runtime (in nanoseconds) that can remain unclaimed at any instant. This limit is set at a *per-nice* granularity. This means that all the `SCHED_DONOR` tasks with nice value 0 can together have a combined maximum of `max_vruntime_unclaimed_ns` nanoseconds worth of outstanding (donated but unclaimed yet by any `SCHED_RECVR` task) runtime only. Similarly for all the `SCHED_DONOR` tasks with nice value 1, 2, and so forth.

7. The global variable `stale_request_ns` defined in `kernel/sched/fair.c` is the maximum lifetime that an outstanding donation can remain unclaimed for any nice class. After this, the outstanding donations must be reclaimed by `SCHED_DONOR` tasks of that nice class. For example, consider a scenario where `stale_request_ns` is set to $100ms$. A `SCHED_DONOR` task, namely X, of nice value 5 sees that previous donations from tasks with nice=5 have not been claimed by any `SCHED_RECVR` in the last $100ms$ (this can happen if there are no active `SCHED_RECVR` tasks on the system). Now X must reclaim this donation, exactly as if it were a `SCHED_RECVR` task. The same aforementioned rules related to claiming apply (ie. the reclaimed time cannot be more than half of X's uncommitted runtime, and the already committed runtime of X cannot be updated retrospectively).

8. The global variable `flush_claimslist_ns` defined in `kernel/sched/fair.c` is the period of inactivity (in nanoseconds) after which all unclaimed donations must be flushed. 'Activity' in this context includes any attempt by a `SCHED_DONOR` or `SCHED_RECVR` to update (increment) its runtime statistics (`sum_exec_runtime`). At any instant when it is seen that no activity has transpired in the last `flush_claimslist_ns` period, then it indicates that there may not be any active `SCHED_DONOR` or `SCHED_RECVR` tasks in the system, so we drop all donations for all nice-value classes and reset it all to zero.

9. You do not need to implement any preemption scheme. For this coursework, your sole concern is with how scheduler statistics are maintained, thereby relying on CFS mechanisms to achieve the desired effect on scheduling decisions.

10. Only a privileged user (eg. the *root* user) can successfully call the `sched_setscheduler` system call. If the system call was made by an unprivileged user, return `-EPERM` (ie. *Error: Operation Not Permitted*) [3].

You will have to maintain a shared global data structure that contains the list of unclaimed donations from every nice-value group. Such a data structure is visible to all threads on a system and may be updated concurrently. Therefore you must ensure that this data structure is thread-safe. You will learn about this concept, called Synchronization, in greater detail later in this course. For now, you may refer to Chapter 4.6.1 for tips on how you can make your data structure thread-safe.

Understanding the Linux Kernel

Question: How is CPU time clocked and maintained in the Linux kernel ?

Hint:
Each CPU's runqueue is defined as an instance of `struct rq` (defined in `kernel/sched/sched.h`). Observe the presence of the member variables `clock` and `clock_task`. These variables may be read from by the functions `rq_clock` and `rq_clock_task` defined in `kernel/sched/sched.h`, and written to using the functions `update_rq_clock` and `update_rq_clock_task` defined in `kernel/sched/core.c`. Search for all the instances in the kernel where these functions are called, and try to reason about what role they fulfill in their various instances. Similarly, in `kernel/sched/fair.c`, can you find out if there exists a function that uses `rq_clock_task` to update the task's runtime statistics?

4.5 | Testing and Configuring from User Space

4.5.1 | DebugFS

The Debug Filesystem [4] is mounted at `/sys/kernel/debug`. If you do not find anything in this folder, first ensure that your kernel config has `CONFIG_DEBUG_FS` set. If so, then mount the filesystem at the

[3] All error codes are defined in `/include/uapi/asm-generic/errno-base.h`
[4] https://docs.kernel.org/filesystems/debugfs.html

appropriate location as follows.

```
USER@HOSTNAME:~$ mount -t debugfs none /sys/kernel/debug
```

Debugfs exists as a simple way for kernel developers to make information available to user space. Unlike `/proc`, which is only meant for information about a process, or sysfs, which has strict one-value-per-file rules, debugfs has no rules at all. Developers can put any information they want there.

Chapter 4.4.1 describes how we have configured debugfs entries for three variables used in `kernel/sched/core.c` namely `max_vruntime_unclaimed_ns`, `stale_request_ns`, and `flush_claimslist_ns`. The values of these variables can be read or written in the files of the same name located in `/sys/kernel/debug/sched/`. You may follow the same structure to create your own DebugFS variables (bear in mind, though, that your final submission will not include them, as you can submit only `core.c` and `fair.c`

4.5.2 | System Calls

System Calls are the key method by which your policies will be set. The key system calls relevant to you are - `sched_setscheduler`, `sched_getscheduler`, `getpriority`, and `nice`. Ensure that these system calls function correctly for your policy implementations (including error handling).

4.5.3 | Tracing

The Ftrace infrastructure [5] is an internal tracer designed to help out developers and designers of systems to find what is going on inside the kernel. It can be used for debugging or analyzing latencies and performance issues that take place outside of user-space.

You can view the tracer infrastructure at `/sys/kernel/debug/tracing`. The trace event `sched_stat_runtime` is defined in `include/trace/events/sched.h`. It is invoked as `trace_sched_stat_runtime` from within the `update_curr` function in `fair.c`. You can configure and enable it from user space by the options in the `/sys/kernel/debug/tracing/events/sched/sched_stat_runtime/` folder.

Tracing allows you to debug your code efficiently. We shall be using this to test your code.

4.6 | Hints and Examples

4.6.1 | Spinlocks

One simple way to make a shared data structure thread-safe is to use a spinlock. Linux provides the `raw_spinlock_t` datatype which may be used for this. The following is an example of how it may be declared.

```
struct shared_data {
    int data[100];
    char *name;
    raw_spinlock_t slock;
};
struct shared_data shared_obj;
```

Two functions are associated with spinlocks - *lock* and *unlock*. Any code that operates on the shared data must be enclosed between *lock* and *unlock*. This code is called the *critical section*. When a thread reaches the *lock* function and enters the critical section, that thread is said to *hold* the lock with itself. While the lock-holder is still in the critical section, if another thread tries to enter the critical section, it will not be able to. The second thread will wait (ie. it will *spin*) until the first thread calls *unlock*, at which point the first thread *drops* the lock and exits the critical section. Now the second thread may pick up the lock and enter the critical section.

For `raw_spinlock_t`, the *lock* and *unlock* functionalities are offered by the `raw_spin_lock` and `raw_spin_unlock` functions. These are `void` functions that take a pointer to `raw_spinlock_t` as argument. The following is an example of how to use them.

```
raw_spin_lock(&shared_obj.slock);
data[x]++;
raw_spin_unlock(&shared_obj.slock);
```

[5] https://www.kernel.org/doc/html/v5.0/trace/ftrace.html

Please ensure the following when using spinlocks:

1. Please ensure that no matter what computational path a thread takes after it has picked up a lock, it will eventually drop the lock by calling unlock. Jump statements, and return statements, etc. may cause a lock-holder to exit the critical section without unlocking. This will cause any other thread that tries to enter the critical section to stall, spinning indefinitely.
2. Keep critical sections short, to prevent the time spent spinning by other threads. Any "non-critical" code (i.e. code that does not operate on shared data) should be moved outside the critical section, as far as possible. Be cautious when calling external functions from within critical sections. Conversely when calling external functions, ascertain if they expect the caller to have some lock taken.

4.6.2 | Useful Linux APIs and Macros

- `include/linux/sched/prio.h` contains various macros related to niceness and priority values. These will be very convenient for the coursework.
- The `task_nice` fuunction defined in `include/linux/sched.h` returns the nice value for any `task_struct`. Similarly, many functions defined in `kernel/sched/sched.h` such as `task_of`, `task_cfs_rq`, `cfs_rq_of`, `rq_of` and so on may be used to instantly extract relevant fields from various key structs, instead of chasing nested pointers.
- Several functions in `kernel/sched/fair.c` can be used to update the various weights and loads used by CFS, such as `update_load_add`, `update_load_sub`, `enqueue_load_avg`, `dequeue_load_avg`, `update_load_set`, `update_load_add`, `update_load_sub`, and so on. Understand what they do, and how they are to be used, by studying the various instances where they get used.

4.6.3 | CFS Statistics, Load, and Weights

`struct load_weight` (defined in `include/linux/sched.h`) is a structure that encapsulates the weights that are used in scheduling decisions by CFS. `struct sched_entity` includes a member `load` of type `struct load_weight` to hold the weights associated with the task (or task-group) to which the scheduling entity belongs. The CFS Runqueue (`struct cfs_rq`) also contains a member `load` of type `struct load_weight` to hold the weights associated with the runqueue (ie. the sum of all the weights of the individual scheduling entities on the runqueue).

`struct sched_avg` (defined in `include/linux/sched.h`) encapsulates the load statistics relevant to CFS. Two members of this structure - `load_sum` and `load_avg` convey the load, and must be updated correctly when you implement `SCHED_AGING` in Task 1 of the coursework. `struct sched_entity` includes a member `avg` of type `struct sched_avg` to convey the load associated with the task (or task-group) to which the scheduling entity belongs. The CFS Runqueue (`struct cfs_rq`) also contains a member `avg` of the same type to hold the load associated with the runqueue.

4.7 | Essential Reading

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

1. Completely Fair Scheduler (CFS) Documentation: `Documentation/scheduler/sched-design-cfs.rst`
2. Manual Pages of various system calls (eg. run `man sched_setscheduler` on bash commandline in Linux or find the corresponding entry at https://man7.org/linux/man-pages/man2/).