Operating Systems 2020-21
Practical Coursework

1 Introduction

The goal of the **Operating Systems** practical coursework is to implement important functionality in an existing *research operating system* called **InfOS**. The coursework counts for **50%** of the total course mark, and is marked out of a total of **100**.

The coursework is split into **four** main tasks, each of which must be submitted individually:

- **Task 1:** Implement a device driver for a real-time clock. `[15 marks]`
  (due Thursday 4th February, 2021 at 4pm GMT—Week 4)
- **Task 2:** Implement process scheduling. `[20 marks]`
  (due Thursday 25th February, 2021 at 4pm GMT—Week 6)
- **Task 3:** Implement a page-based memory allocator. `[40 marks]`
  (due Thursday 11th March, 2021 at 4pm GMT—Week 8)
- **Task 4:** Implement a file system driver. `[25 marks]`
  (due Thursday 25th March, 2021 at 4pm GMT—Week 10)

You may start any of the tasks as early as you wish, but you **MUST** submit each task before their respective deadline. Coursework submitted after these deadlines *may* still be marked (at the discretion of the marker), but will attract a score of **zero**. If you require an extension, you must contact the ITO **before** the deadline.

The submission mechanism will be electronic, using an online form that can be accessed from Learn. Details for submitting each assignment are given in the respective **Marking and Deliverables** section, for each coursework task.

Read this document in full, and pay attention to the number of marks for each task, so that you can get an idea of the amount of effort required.

1.1 Background

The research operating system that is the subject of this coursework is called **InfOS** and has been developed specifically for this course. It is a 64-bit x86 operating that has been written from scratch in C++, and is not based on any other particular OS.

InfOS is structured as a monolithic kernel, similar to Linux, but as it is written in C++, it employs object-oriented principles in its implementation. It is **not** a Unix-like system, and does not subscribe to the POSIX standards. It is modular in the sense that functionality can be plugged in/out at compile time, but does not (currently) support dynamically loadable kernel modules.

1.2 Necessary Skills

This coursework involves programming in C++, so familiarity with object oriented programming and the C++ programming language will be very helpful. You should also take this as an opportunity to expand your C++ programming skills.

The coursework tasks generally follow the syllabus, so it is essential that you keep up-to-date with the course content.

1.3 Overview

For each task, you will be implementing a piece of core operating system functionality to be loaded into the **InfOS** kernel. To ensure that each task is mutually exclusive, and that your implementations do not prejudice each other, **InfOS** is shipped with basic implementations already built-in.

This means that **InfOS** will boot unmodified, and you can work on each task separately, without worrying that one implementation might affect another.

1.3.1 Development Environment

You are encouraged to develop your coursework on DICE, as we know this platform works. However, you should also be able to develop locally, on a Linux machine (or even in a VM) if you have a modern C++ compiler, and the QEMU emulator installed. Once you have acquired the **InfOS** source-code, you can load it into an IDE of your choice.

If you want to work on DICE remotely, you can use the *Informatics Remote Desktop Service* to access a remote DICE desktop. See the following website for details:

`http://computing.help.inf.ed.ac.uk/remote-desktop`

1.3.2 Testing

To test the operating system, QEMU will be used as an emulator. QEMU is a virtualisation environment that can be used to boot real operating systems in a virtual machine. It is installed on DICE and has been tested with the version of **InfOS** that you will be using. QEMU is also available in all of the popular Linux distributions' package repositories. You can even compile it from source, if you wish, although this is not usually necessary.

Helper scripts are provided to quickly compile and run **InfOS** in QEMU, which will work if QEMU is installed in a standard location (e.g. on DICE or if you installed it from your package manager). See the individual tasks for more details.

2 InfOS

2.1 Introduction

InfOS is a research operating system, developed from scratch by *Tom Spink*[1]. It is a 64-bit x86 operating system, designed in C++ to promote readability, and use of familiar object-oriented programming paradigms.

InfOS was developed because modern versions of the Linux kernel are incredibly complex, and contain highly optimised implementations of core operating system operations, such as the *process scheduler* and *memory allocator*. It is not feasible to understand the entirety of the Linux kernel, nor is it feasible to re-implement core functionality without a significant understanding of the kernel architecture. **InfOS** tackles this problem by providing well-defined interfaces for these subsystems, and providing a "pluggable" architecture that enables swapping different algorithms in and out.

2.2 Source Code

A fully booting version of the **InfOS** kernel, along with the associated user-space is available in a few public `git` repositories, which live here - *pull requests are welcome!*:

- `https://github.com/tspink/infos` - the core kernel
- `https://github.com/tspink/infos-user` - the userspace

However, for your convenience, you will be using the following `git` repository:

`https://github.com/tspink/infos-coursework`

Which contains helper scripts, coursework skeletons, and the core kernel and userspace repositories as *git submodules*.

2.2.1 Pre-requisites

Everything you need to compile and run **InfOS** on DICE is available, but if you plan to do this yourself in your own environment, you'll need:

1. An up-to-date version of the GNU C++ compiler, which supports at least the `-std=gnu++17` command-line option.
2. The `make` build system.
3. An up-to-date version of Qemu.

2.2.2 Getting the source-code

To get the source-code, from a suitable location in your home directory issue the following commands in your terminal:

```
$ git clone --recurse-submodules https://github.com/tspink/infos-coursework
```

[1] https://homepages.inf.ed.ac.uk/tspink

This will checkout the coursework repsitory into a new directory `infos-coursework`, which has been created in the *current directory*.

In the `infos-coursework` directory, you'll find some scripts for building and running, the **InfOS** core respository, the **InfOS** user repository, and a place for you to put your coursework answers, which will be automatically picked up by the build system.

2.2.3 Compiling and Running

The `infos-coursework` repository contains some scripts to help compile and run the kernel, including compiling the answers to your coursework. The most interesting scripts are:

- `build.sh`: compiles the **InfOS** kernel, and the **InfOS** user-space. Also compiles your coursework.
- `run.sh`: runs the compiled kernel in QEMU. Also allows you to specify command-line options for passing to the kernel.
- `build-and-run.sh`: does both of the above.

Try this out *now* - issue the following command in the `infos-coursework` directory:

```
$ ./build-and-run.sh
```

You should see the kernel compile, followed by the user-space, and then QEMU should start and the kernel should boot. If you have any problems - take a look at Piazza and ask for help.

IMPORTANT NOTE If you are using DICE, you will need to use a VNC client to interact with the user-space. The version of QEMU that is installed on DICE does not support the GTK interface, so after you've started the kernel, you'll need to run the following command **in another terminal**:

```
$ vncviewer localhost
```

This will start a VNC viewer, and connect to the VNC server that QEMU creates. You should only need to do this on DICE, as most packaged versions of QEMU come with built-in graphics support.

If you get an error message, such as *connection refused*, check that the kernel compiled correctly and has started running. You will see any compilation failures on your main terminal. Make sure you don't run multiple instances of the kernel, otherwise the `vncviewer` command will connect to the first one.

PRO-TIP Press `Ctrl+C` in the terminal window to quit QEMU.

2.2.4 Coursework Skeletons

You are provided with skeleton implementations for each coursework task, which are automatically compiled by the **InfOS** build system when you use the `build.sh` or `build-and-run.sh` scripts. This means that to complete your coursework, you simply need make modifications to the skeletons in the `infos-coursework/coursework` directory.

These files are part of the repository, so if you find that you've made a serious mistake and wish to start again, you can restore them using a git command, e.g. from inside the `coursework` directory:

```
$ git checkout sched-rr.cpp
```

Warning! This will permanently erase **any** changes you have made to the `sched-rr.cpp` file, and you will not be able to recover them, unless you have made a backup.

2.2.5 Structure and Implementation

The top-level directory structure (which you will find in the `infos-coursework/infos` directory) has the following directories, each loosely representing a major subsystem of the **InfOS** kernel:

- `arch/` Architecture-specific code.
- `drivers/` Device drivers.
- `fs/` File-system drivers and VFS subsystem.
- `kernel/` Core kernel routines.
- `mm/` Memory management subsystem.
- `util/` Utility functions.

There are also some directories that contain support files:

- `build/` Build system support files.
- `include/` C++ header files (following the source-code structure)
- `out/` Kernel build output directory.

InfOS is written in C++, and so all source-code files have the extension `.cpp`. However, due to the low-level nature of operating system development, some code is written in x86 assembly language. Being architecture specific, these files primarily live in the `arch/x86` directory, and have the extension `.S`.

Nearly every C++ *source*-code file has an associated *header* file (although there are some exceptions), which exists under the `include/` directory. The structure of the `include` directory follows that of the top-level source-code directory, except the header files have the extension `.h`. Normally, there is one class declaration per header file, and then one corresponding source-code file that implements that class. For strongly related classes, occasionally there will be multiple declarations in the header file.

To better organise the class hierarchy, and promote readability, nested directories typically correspond to C++ namespaces, with the root namespace being `infos`.

For example, if you are interested in looking at the ATA storage device driver, you will be interested in looking at the following files:

- `drivers/ata/ata-device.cpp`

- `include/drivers/ata/ata-device.h`

A class called `ATADevice` is declared in the header file `ata-device.h`, and it is implemented in the source-code file `ata-device.cpp`. The class is declared within the following namespace hierarchy:

$$\texttt{infos} \mapsto \texttt{drivers} \mapsto \texttt{ata} \mapsto \texttt{ATADevice}$$

It should be clear that each level of the namespace hierarchy corresponds to the directory in which the source-code and header file live.

2.2.6 Modules

InfOS is built on modules, but it currently does not support dynamically loading them. Instead, the architecture of the OS is built around object-oriented principles, and as such producing a different implementation for a particular feature simply requires subclassing the appropriate type, and implementing the interface.

2.3 Start-up

InfOS uses the *multiboot* protocol to boot, which is a very convenient way of starting an operating system. This protocol is supported by many boot loaders, and Qemu supports booting multiboot enabled kernels natively.

Execution starts in 32-bit mode, in the `multiboot_start` assembly language function. This function lives in the `arch/x86/multiboot.S` source-code file. After this, execution transitions into the `start32` assembly language function, in the `arch/x86/start32.S` source-code file.

This function initialises 64-bit mode, and jumps to `start64` in the `arch/x86/start64.S` source-code file. Finally, control transfers to the `x86_init_top` function, which is the first executed C++ function, in the `arch/x86/start.cpp` source-code file. From there, you can follow the sequence of events that bring up the operating system.

2.4 Memory Manager

The memory manager is responsible for providing memory to programs/subsystems that request it. The majority of requests will be for memory that can be used to store objects, (e.g. via the C++ `new` operator), but some requests may be for entire *pages* of memory (e.g. for allocating page tables).

A *page* of memory is a block of memory that is the most fundamental unit dealt with by the underlying architecture. Pages are always aligned to their size boundaries (i.e. the address of a page is always a multiple of their size) and on x86 (and therefore in **InfOS**) the page size is 4096 bytes.

InfOS has two memory allocators: a physical memory allocator, and an object allocator. The physical memory allocator deals with allocating physical pages of memory, whilst the object allocator deals with satisfying arbitrarily sized amounts of memory for storing objects. The

object allocator calls into the physical memory allocator to request large blocks of memory, which are then used to store smaller objects.

The object allocator used is the open-source `dlmalloc` allocator. The built-in physical memory allocator is an inefficient linear scan allocator.

2.4.1 Physical Memory

Fundamentally, a computer system has an amount of *physical memory* (RAM). This memory is what is actually available for the storage of data, and is exposed in the *physical memory space*. In modern systems, it is possible to *address* up to 52-bits (4096 Tb) of physical memory, but a normal desktop system may have between 2-16Gb of memory installed.

The physical memory space consists of various regions of usable and unusable pages. It also contains memory mapped devices that are not *real* memory, but allow the configuration and operation of those devices by reading and writing to the associated memory address. To work out what pages are available, an operating system needs some support from the bootloader and the architecture.

2.4.2 Virtual Memory

Virtual memory is a flat view of memory as seen by the programs that are running. Each program has its own virtual memory area (or VMA), which maps virtual addresses to physical addresses. This mapping also includes protections, to prevent programs from reading and writing to pages that it should not have access to, e.g. kernel pages.

The mapping specifies which *virtual* memory address corresponds to which *physical* memory address, at the granularity of a *page* (i.e. 4096 bytes). A physical address may have multiple virtual addresses pointing to it.

2.5 Process Scheduler

The **InfOS** process scheduler is responsible for sharing out execution time on the CPU for each process that is on the ready queue. **InfOS** uses a timer, ticking at a frequency of 100Hz to interrupt and pre-empt processes to determine if they need to be re-scheduled.

2.5.1 Scheduling Algorithms

There are many scheduling algorithms for process scheduling, and the built-in scheduler implements an inefficient version of the Linux CFS scheduler. **InfOS** does not support process priorities, greatly simplifying the scheduler implementation.

2.6 Device Manager

The **InfOS** device manager is responsible for detecting the devices that exist on the system, and creating an abstraction that allows them to be accessed by programs that require them. For example, it interrogates the system's PCI bus to detect storage devices, and allows that storage device to be accessed by file-system drivers.

2.7 Virtual Filesystem Switch (VFS)

The Virtual Filesystem Switch (or VFS) subsystem presents an abstract interface for accessing files. Within a virtual file-system, *physical* file-systems (for example those that exist on a disk) are *mounted* and can be accessed. Multiple *physical* file-systems can be mounted within the virtual file-system, and they appear as normal directories within the VFS tree. Physical file-systems could be local, on-disk file-systems, or they could be remote network file-systems. Some file-systems could be dynamically created, e.g. **InfOS** creates a *device* file-system which contains files that represent each registered device in the system. You can see this by entering:

```
> /usr/ls /dev
```

At the **InfOS** shell, to list the contents of the `/dev` directory.

2.8 InfOS API

As **InfOS** is a bare-metal operating system, it cannot use the standard C++ library, and hence standard C++ routines and objects (such as strings, lists and maps) are not available.

Since these containers can be quite useful, **InfOS** implements its own versions of some of these containers and exposes them for use by operating system code. They do not directly correspond to the standard C++ implementations, but they provide all the methods you would expect these containers to have. This section will describe some of these containers, particularly those which you will find useful for the coursework, and how to use them.

You can see many examples of their use throughout the **InfOS** source-code.

2.8.1 `List<T>`

The templated `List<T>` class is a container for objects that is implemented as a linked-list. It can be used by declaring a variable of type `List<T>`, where `T` is the type of object contained within the list. To use it, you must `#include` the `infos/util/list.h` header file.

As an example, to create a list that contains integers, one would write:

```
List<int> my_list;
```

The list object can be used as a stack or a queue, and can be iterated over with a C++ iterator statement:

```
int sum = 0;
for (auto& elem : my_list) {
    sum += elem;
}
```

The `List<T>` class exposes the following methods:

`void append(T elem)` Appends `elem` to the **end** of the list.

`void remove(T elem)` Removes any element that equals `elem` from the list.

`void enqueue(T elem)` Appends `elem` to the **end** of the list.

`T dequeue()` Removes the element at the **start** of the list, and returns it.

`void push(T elem)` Inserts `elem` at the **start** of the list.

`T pop()` Removes the element at the **start** of the list, and returns it.

`T first()` Returns the element at the **start** of the list.

`T last()` Returns the element at the **end** of the list.

`T at(int index)` Returns the element at the given index in the list.

`unsigned int count()` Returns the number of elements in the list.

`bool empty()` Returns `true` if the list is empty.

`void clear()` Removes all items from the list.

2.8.2 `Map<TKey, TValue>`

The templated `Map<TKey, TValue>` class is a tree-based implementation of an associative array. It is implemented as a red-black tree, so it is reasonably efficient, but the implementation details are not important.

To use it, you must `#include` the `infos/util/map.h` header file.

As an example, to create a map that associates integers to integers, you would declare it as follows:

```
Map<int, int> my_map;
```

You could then insert elements into the map, and look them up:

```
my_map.add(1, 10);
my_map.add(2, 20);
my_map.add(3, 30);

int value;
if (my_map.try_get_value(2, value)) {
  // Element with key 2 was found, variable 'value'
  // now contains the value.
} else {
  // Element with key 2 was not found
}
```

The `Map<>` class exposes the following methods:

`void add(TKey key, TElem elem)` Inserts `elem` into the map with the given `key`.

`void remove(TKey key)` Removes the element in the map associated with `key`.

`bool try_get_value(TKey key, TElem& elem)` Looks up the value associated with `key`. Returns `true` if the key exists, and updates the contents of `elem` with the value. Returns `false` if the key does not exist, and `elem` is undefined.

`void clear()` Removes all elements in the map.

2.8.3 Use of Containers

Important Note: The `List<>` and `Map<>` containers both use *dynamic memory allocation* to create the internal structures that represent the respective data-structure, and as such are not suitable for use in code that executes before the memory allocator has been fully initialised—this applies to memory allocation code itself.

2.8.4 Logging and Debugging

InfOS emits a significant amount of debugging information by default, which can be turned on and off via command-line arguments. The provided helper script that launches **InfOS** in QEMU allows you to append command-line arguments, and details about which arguments are relevant to a particular subsystem are provided in later sections.

Being a bare-metal operating system, it is difficult to debug **InfOS** in a debugger such as GDB, and so you must rely on logging output to discover problems. The launch script will direct **InfOS** debugging output to the terminal, so you can scroll through the output to read the log.

All of the skeleton files are set-up for logging, and you can use a `printf`-style function to write out to the log:

```
int var = 5;
syslog.messagef(LogLevel::DEBUG, "A log message without formatting");
syslog.messagef(LogLevel::DEBUG, "A log message with formatting value=%d", var);
```

The first parameter indicates the *level* of the message, the second parameter is the message to be displayed, and the optional following parameters are the values for the `printf`-style format string in the message text. You do not need to include a new-line in the message text, as the logging system will do this for you.

Again, there are many examples throughout the source-code that use the logging infrastructure. Some subsystems use their own logging object, e.g. the VFS subsystem has a `vfs_log` object, but the `syslog` object is always available for logging.

3 Building and Testing

Throughout this coursework, you will be required to repeatedly build and test your implementations. After using the `prepare-coursework.sh` script to create the development environment in your home directory, you can issue the `build-and-run.sh` command to compile and test the operating system.

As previously mentioned, this will launch **InfOS** inside QEMU, as it is a virtualisation environment that supports running native operating systems.

3.1 User Space

An operating system kernel on its own does not do anything useful for the user. In order for you to interact with the kernel, an example user-space is provided. This user-space contains a very basic *shell* program that allows you to execute commands, along with some other commands that you can use for testing purposes.

When you launch **InfOS** the shell will automatically load, and instructions are provided on what commands are available. Use the `/usr/ls` command to list a directory and see the available files.

The **InfOS** user space is built automatically if you use the `build-and-run.sh` script, and a disk image is automatically created and loaded into QEMU at run time.

4 General Note on Writing Solutions

In general, you are provided with skeleton files to help you get started with your implementation, and you are free to make whatever modifications to the skeleton that you see fit—including adding helper functions and modifying class variables. In fact, to improve readability you are encouraged to do so.

However, bear in mind that you *cannot* change the **InfOS** API, i.e. you cannot modify the **InfOS** source-code for your solution. Your implementation must be wholly contained within the skeleton file for that particular task. For each task you may only submit the named files as specified in the respective **Marking and Deliverables** section, which means your implementation must be restricted to those files.

When designing a solution, pay careful attention to potential sources of errors. Marks are available for including appropriate error handling code. Marks are also available for use of efficient algorithms, and the readability of your solution.

Use logging facilities to help you produce your solution, but don't go overboard—logging is expensive and can slow your implementation down. Remember to remove any logging you don't need before final submission (and remember to test your implementation again after you've removed the logging!)

5 *Task 1:* Device driver for Real-time Clock

Your first coursework task is to implement a device driver for a generic **real-time clock**.

It is due by **4PM GMT** on **Thursday 4th February, 2021**, and is worth **15 marks out of 100**.

5.1 Introduction

A real-time clock (RTC) is tasked with keeping track of the current date and time. RTCs are generally powered separately, usually by a battery, and keep ticking even when the main power to the computer has been removed. Without an internet connection, this is the only way for the computer to know what the current date and time is, although if the battery runs out (or the data becomes corrupt), then the time will be wrong.

5.2 Background

The CMOS RTC has been around since the 90s, and is accessed in quite a straightforward way. There is a tiny area of static memory that is accessed a byte at a time, and certain bytes correspond to certain components of the current date and time, as managed by the RTC.

Each second, the RTC updates these bytes, incrementing seconds, minutes, hours, etc accordingly.

5.3 Accessing the RTC

In order to access the RTC, interrupts must first be disabled (you can use a scoped `UniqueIRQLock` to do this). Then, you must wait for an update cycle to begin, and for an update cycle to complete before reading CMOS data. Once this situation has occurred, the best course of action is to read a block of bytes from CMOS memory into a buffer, then process those bytes individually. The idea is to read CMOS memory quickly, before the next update cycle occurs (which, if you've followed the procedure above, should be in about a second).

CMOS memory is accessed through the I/O space, and requires the use of the low-level `__outb` (for writing a byte) and `__inb` (for reading a byte) instructions. To access a byte of CMOS memory, the memory offset for the byte you are interested in is written to port `0x70`, then the value is read from port `0x71`. These low-level access routines are found in the `arch/x86/pio.h` header file.

For example, to read the byte at offset 3 in CMOS memory, you would issue the following instructions:

```
__outb(0x70, 3);         // Activate offset 3
uint8_t v = __inb(0x71); // Read data
```

`v` now contains the value in offset 3 of CMOS memory.

5.3.1 The Update Cycle

The RTC has two status registers (`A` and `B`), and status register `A` contains a bit that indicates whether or not an update is in progress. Status register `A` lives at offset `0xA`, and the update-in-progress flag is stored in bit `7`. This bit is **set** when an update is in progress, and **cleared** otherwise.

To read from the RTC, you should wait for an update to begin, then wait for the update to complete, **before** reading any CMOS registers.

5.3.2 CMOS Registers

The following table lists the interesting CMOS registers for the real-time clock:

Offset	Description
`0x00`	Seconds
`0x02`	Minutes
`0x04`	Hours
`0x06`	Weekday
`0x07`	Day of Month
`0x08`	Month
`0x09`	Year
`0x0A`	Status Register `A`
`0x0B`	Status Register `B`

5.3.3 Register Format

The values contained within the registers may be in either **binary** or **binary coded decimal**, and you must figure out which this is by interrogating status register `B`. If bit 2 is **set** in status register `B`, then the register values are stored in binary. If bit 2 is **cleared**, then the register values are in binary coded decimal, and you must convert them into binary before returning them out of the function.

Your code must handle both of these situations, although you have no influence on whether or not the RTC operates in binary or BCD mode.

5.4 Returning the Date and Time

Once you have read and parsed the real-time clock, you must return the necessary in the appropriate fields of the structure supplied in the `read_timepoint` function (parameter `tp`). The structure contains the following fields:

```
struct RTCTimePoint {
    unsigned short seconds, minutes, hours;
    unsigned short day_of_month, month, year;
};
```

These fields must be filled in from the (possibly converted) values read from the RTC, before the function returns. You can ignore the `weekday` register from the RTC.

5.5 Resources

There is detailed information about accessing the CMOS RTC on the osdev website:

`https://wiki.osdev.org/CMOS`

This website provides all the information necessary to implement this task, although feel free to search for further information elsewhere, if it helps.

5.6 Important Files and Interaction with InfOS

InfOS uses the RTC to set it's internal representation of the date and time, which it increments locally on each clock tick. This is because it's quite expensive to read the RTC, since you must wait for an update cycle to complete. Therefore, the two clocks can become out-of-sync and drift. In this task, we're not concerned with drift, so you can safely ignore any minor deviation between the two times.

Your job is to implement the `read_timepoint` function in the `cmos-rtc.cpp` file, to interrogate the real-time clock, and store the values in the provided structure.

5.7 Skeleton

You are provided with a skeleton module, in which you must write your code to implement the device driver. The skeleton is commented, and indicates where you should write your code. You are free to modify this skeleton in any way you see fit, and you are encouraged to write helper functions to make your code more readable.

You will find the skeleton in `infos-coursework/coursework/cmos-rtc.cpp`. Do not copy or move this file anywhere, just open it up in your IDE of choice, and edit the file.

5.8 Testing

To compile and run **InfOS**, issue the `build-and-run.sh` command from your `infos-coursework` directory:

```
$ ./build-and-run.sh
```

Your source-code will be built, and if there are any errors, these will be displayed to you and the operating system will not load. Unlike the other tasks, this task **does not** require special command-line arguments to get the device to operate.

In the **InfOS** shell, run the `/usr/date` command, to see what the kernel's view of the current date is:

```
> /usr/date
the current system date & time is: 01/01/20 10:13:22
the current hardware date & time is: 01/01/20 10:13:25
```

This program prints out *two* times: (1) what the kernel thinks the date and time is, based on its (lousy) internal counter, and (2) the date and time as reported by the hardware real-time clock.

The internal counter drifts quite significantly from the hardware clock, as there is no drift correction code implemented, but this is not important for this task. What is important is that the **hardware date and time** is close to the current real date and time. If your implementation works, then the hardware date and time output should be correct (± a few minutes is fine). This assumes that the clock on the **host** machine is correct.

If you have made any errors, then you'll likely see very strange date and times being displayed.

5.9 Marking and Deliverables

This part of the coursework attracts **15 marks**.

Marks will be given for correctly interrogating the CMOS real-time clock (as per the specification), and correctly parsing and returning the values from the clock. Marks will also be given for readability/coding style, and for inserting appropriate error checking.

You must submit your implementation, by uploading your source-code to **Coursework 1 - The Real time Clock** in Learn **BEFORE** 4PM GMT on **Thursday 4th February, 2021**. No other form of submission will be accepted, and late submission will attract a score of zero.

6 *Task 2:* Process Scheduling

Your second coursework task is to implement two different processing scheduling algorithms:

1. A round-robin scheduler
2. A FIFO scheduler

It is due by **4PM GMT** on **Thursday 25th February, 2021**, and is worth **20 marks out of 100**.

6.1 Introduction

The process scheduler of an operating system is the component that decides which tasks get to run on the CPU. When there is only a single physical processor in a system, each task that is running needs to be given an opportunity to actually run on that processor, so that the processor can be shared amongst all *runnable* tasks.

A task can be in a number of states:

STOPPED The task is not running.

RUNNABLE The task wants to run on a CPU.

RUNNING The task is currently running on a CPU.

SLEEPING The task is waiting for an event to occur, before it becomes runnable.

Only *runnable* tasks can be *scheduled* onto a CPU, and scheduling is the act of determining which task should run next. Ideally, a scheduler will make sure that every task gets a chance to run.

The Linux kernel scheduler implements a so-called **completely fair scheduler**. It orders tasks by *virtual runtime*, and allows the runnable tasks with the lowest *virtual runtime* to run first. As the Linux kernel supports process priorities, the *virtual runtime* is weighted, depending on the process' priority.

A round-robin scheduling algorithm can be implemented as a simple list of tasks. When a new task is to be picked for execution, it is removed from the front of the list, and placed at the back. Then, this task is allowed to run for its timeslice.

A FIFO scheduling algorithm can also be implemented as a simple list of tasks. Tasks are added to the list, and execute until their completion (i.e. they are removed from the runqueue).

6.2 Important Files and Interaction with InfOS

The generic scheduler core exists in the `kernel/sched.cpp` source-code file. This is where the majority of the scheduling subsystem lives. You will not be required to change this code, but it is useful to know where this is so that you can refer back to it. In **InfOS**, the generic scheduler calls into a *scheduling algorithm* when a scheduling event occurs. The *scheduling algorithm* is what you must implement.

The generic scheduler core will automatically detect your algorithms when you compile them into the source-code, but you **MUST** tell **InfOS** to use them when running and debugging, otherwise it will be using the built-in scheduler. See Section 6.4 for more details.

Implementing the scheduling algorithm interface requires implementing three methods:

- `add_to_runqueue`: This is called by the scheduler core when a task becomes eligible to run on the CPU (e.g. it has started, or has woken up from sleep).
- `remove_from_runqueue`: This is called by the scheduler core when a task is no longer eligible to run (e.g. it has terminated, or is going to sleep).
- `pick_next_task`: This is called by the scheduler core when it is time for a new task to run. This is where you will implement the majority of the algorithm.

You can take a look at the built-in scheduler, which is *loosely* based on the Linux CFS scheduler (although it is much less efficient). This code lives in `kernel/sched-cfs.cpp`.

You **MUST** make sure that interrupts are disabled when manipulating the run queue. You can use a scoped `UniqueIRQLock` for this.

6.3 Skeletons

You are provided with two skeleton scheduling algorithm interfaces, in which you must write your code to implement both the *round-robin scheduler* and the *FIFO scheduler*. The skeletons are commented, and indicate what methods you should fill in, and where you should write your code.

You will find the skeletons in `coursework/sched-rr.cpp` for the round-robin algorithm, and `coursework/sched-fifo.cpp` for the FIFO algorithm. Do not copy or move these files out of the coursework directory, just open them up in your IDE of choice to edit them.

6.4 Testing

Your source-code will be automatically compiled into the **InfOS** kernel, and the algorithm can be chosen with a command-line parameter.

To compile and run **InfOS**, issue the `build-and-run.sh` command from your `infos-coursework` directory, **and tell InfOS to use your scheduler** with the `sched.algorithm=rr` (for round-robin) and `sched.algorithm=fifo` (for FIFO) option e.g.:

```
$ ./build-and-run.sh sched.algorithm=rr
$ ./build-and-run.sh sched.algorithm=fifo
```

It is important that you put the `sched.algorithm` option on the command-line, otherwise the built-in scheduler will be used instead.

Your source-code will be built, and if there are any errors, these will be displayed to you and the operating system will not load.

To double-check that **InfOS** is using your scheduler, scroll back in the log window and look for the line:

```
notice: *** USING SCHEDULER ALGORITHM: rr
```

or

```
notice: *** USING SCHEDULER ALGORITHM: fifo
```

If your algorithm does not work at all, then the system will likely not boot and may crash. You can use the `sched.debug=1` option to produce more debugging output from the scheduler core:

```
$ ./build-and-run.sh sched.algorithm=rr sched.debug=1
```

If the system boots up to the **InfOS** shell, then you can try running some test programs that will exercise the scheduler:

```
> /usr/sched-test1
> /usr/sched-test2
```

VERY IMPORTANT NOTE If you've implemented the FIFO algorithm correctly, you'll find that the `sched-test2` actually causes the system to stop responding to further commands (the system will continue to run, but only one thread will be executing). For a bonus point, in your solution write a comment that explains why this behaviour occurs.

6.5 Marking and Deliverables

This part of the coursework attracts **20 marks**. Marks will be given for correctly implementing the algorithms, for readability/coding style, and for the inclusion of basic error checking.

You must submit your implementation, by uploading your source-code to **Coursework 2 - The Scheduler** in Learn **BEFORE** 4PM GMT on **Thursday 25th February, 2021**. No other form of submission will be accepted, and late submission will attract a score of zero.

Make sure you submit both `sched-rr.cpp` and `sched-fifo.h`.

7 *Task 3:* Physical Memory Buddy Allocator

Your third coursework task is to implement a physical memory allocator, based on the buddy allocation algorithm.

It is due by **4PM GMT** on **Thursday 11th March 2021** (week 8), and is worth **40 marks out of 100**.

7.1 Introduction

Normally, when a program requests memory it will simply ask for a particular amount. It expects the memory allocator to find space for this, and it does not care where that memory is. But, memory allocators need to put this memory somewhere, and at a very low-level, this memory has to exist in *physical* form.

Physical memory allocation is the act of allocating real physical memory pages to places that require them. Typically, higher-level memory allocators (such as the **InfOS** object allocator) request physical pages, which are used for storage of smaller objects.

A particular algorithm for managing this physical memory is the **buddy allocation algorithm**, and this is what you are required to implement for **InfOS**.

Like the scheduler, **InfOS** comes with a built-in physical memory allocation algorithm, but it is very simple and inefficient. This task requires you to implement the buddy allocation algorithm as described in the lectures and from various resources online[2].

7.2 Background

A page of memory is the most fundamental unit of memory that can be allocated by the page allocator. In **InfOS**, the page size is 4096 bytes (`0x1000` in hex). Pages are always aligned to their size, and can be referred to with either:

1. Their *page frame number* (PFN), or
2. their *base address*.

PFNs are zero-indexed, so for example, the second page in the system has a PFN of `1`. Since pages are aligned, the base address of PFN `1` is `0x1000`. Likewise, given a page base address of `0x20000`, a simple division by `4096` (or right-shift by `12`) yields a PFN of `32`.

For every physical page of memory available in the system, a *page descriptor* object exists which holds information about that page. These pages descriptors are held as a **contiguous** array, and so can be efficiently indexed, given the physical address or PFN of a page. This property will become important in your implementation, as it means that given a **pointer** to a particular *page descriptor*, you can look at the adjacent *page descriptor* by simply incrementing the pointer.

The most interesting field in the page descriptor for you is the `next_free` pointer, which you can use to build a linked-list of page descriptors. This will be very useful when building the

[2]For example, `https://en.wikipedia.org/wiki/Buddy_memory_allocation`

per-order free lists. You **MUST NOT** modify any other fields in the page descriptors, as the memory management core is responsible for these.

The physical page allocator does not allocate by memory size, or even by number of pages. Instead it allocates by *order*. The *order* is the power-of-2 number of pages to allocate. So, an allocation of order `zero` is an allocation of $2^0 = 1$ page. An allocation of order `four` is an allocation of $2^4 = 16$ pages. Allocating by *order* makes it significantly easier to implement the buddy allocator.

The buddy allocator maintains a list of free areas for each order, up to a maximum order. The maximum order should be configurable, and is `#define`d for you already in the skeleton. Use this definition when implementing your algorithm.

7.3 Important Files and Interaction with InfOS

The memory allocation subsystem is quite complex, and so has its own top-level directory (`mm/`) in the **InfOS** source code. Under this directory lives the following files:

`dlmalloc.cpp` An import of the dlmalloc memory allocator, used for allocating objects within physical pages. This is the allocator that is called when memory is requested for use. dlmalloc itself calls into the physical page allocator when it needs more physical pages.

`mm.cpp` The generic memory manager core.

`object-allocator.cpp` The generic allocation routines for allocating objects (e.g. those allocated with the `new` keyword).

`page-allocator.cpp` The generic allocation routines for allocating physical pages. This calls into the page allocation algorithm that you will be implementing.

`simple-page-alloc.cpp` The built-in page allocation algorithm that does a linear scan for free ranges—hence it is *highly* inefficient.

`vma.cpp` A virtual memory area (or VMA) is the view of virtual memory seen by a particular task. This file contains the routines that manipulate the page tables that map virtual addresses to physical addresses.

The memory management core will automatically detect your algorithm when you compile it into the source-code, but you **MUST** tell **InfOS** to use it when running and debugging, otherwise it will be using the built-in page allocator. See Section 7.7 for more details.

Some of the ground-work for buddy allocation has already been done for you, and placed in the skeleton file you are provided with. Implementing the page algorithm interface requires implementing these methods:

- `split_block`**:** Given a particular page, being correctly aligned for the given order, this function splits the block in half and inserts each half into the order below. For example, calling `split_block` on page four in order one will remove the block starting at page four (in order one) and insert two blocks into order zero, starting at page four and page five. **This method is a *helper method***

- `merge_block`: Given a particular page, being correctly aligned for the given order, this function will merge this page and its buddy into the order above. For example, calling `merge_block` on page five in order zero will remove page four and page five from order zero, and insert a new block into order one, starting at page four.
  This method is a *helper method*
- `alloc_pages`: This is called by the memory mangement core to allocate a number of contiguous pages.
- `free_pages`: This is called by the memory management core to release a number of contiguous pages.
- `reserve_page`: This is called by the memory management core during initialisation to mark a *particular* page as unavailable for allocation. For example, this would be called to mark the pages that contain the kernel code in memory as unavailable.
- `init`: This is called by the memory management core during start-up, so that the algorithm can initialise its internal state.

Note: Two of the above methods are helper methods, and you should use them in your implementation of the other methods.

Remember: When implementing your algorithm, you cannot use dynamic memory allocation in your memory allocator! This means you cannot use the `new` operator, and you cannot use containers such as `List<T>` and `Map<T, U>`, since these rely on dynamic memory allocation. As mentioned previously, you can use the `next_free` field in the page descriptor structures to build linked lists.

The skeleton file contains the following helper methods, which you should use in your implementation. You should also use these functions as a guide:

- `pages_per_block`: This function returns the number of pages that make up a block of memory for a particular *order*. For example, in order 1, the number of pages that make up a block is 2.
- `is_correct_alignment_for_order`: This function determines if the supplied page is correctly aligned for a particular *order*. For example, page zero is correctly aligned in order 1, but page one is not.
- `buddy_of`: Given a particular page, and the order in which this page lies, this function returns this page's buddy. The buddy is the page either in front or behind the given page, depending on the alignment of the supplied page. See Figure 1. For example, in order zero, the buddy of page three is page two and the buddy of page two is page three.
- `insert_block`: Given a particular page, being correctly aligned for the given order, this function inserts that block into the free list.
- `remove_block`: Given a particular page, being correctly aligned for the given order, this function removes that block from the free list.

You don't necessarily need to use all of these functions for your implementation, but some may come in handy during debugging.

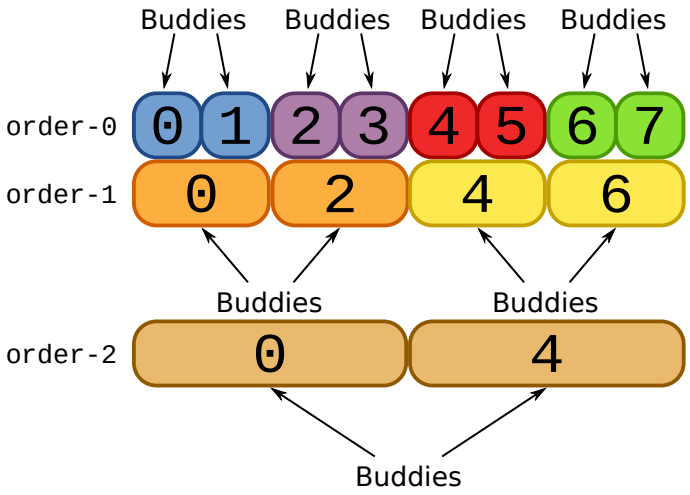

Figure 1: An illustration of block buddy relationships, in different allocation orders.

7.4 Allocating Pages

There are two ways to allocate a page (or pages) with the algorithm interface, and you will be implementing both of these:

1. Calling `alloc_pages`
2. Calling `reserve_page`

Each of these will now be described in turn, as they behave similarly but have different semantics.

7.4.1 `PageDescriptor *alloc_pages(int order) override`

The `alloc_pages` method is called when a contiguous number of pages needs to be allocated. The caller does not care *where* in memory these pages are, just that the pages returned are contiguous. Because of this guarantee, if the caller asks (for example) for an order 1 allocation (i.e. two pages), the routine simply needs to return the **first** page descriptor of a sequence of **two** page descriptors that are available for allocation (by following the buddy allocation algorithm).

This works because the pages are contiguous, and because the page descriptor array is contiguous.

7.4.2 `bool reserve_page(PageDescriptor *pgd)`

The `reserve_page` method is called when a **particular** page must be made unavailable. This page is passed in by the caller as a page descriptor object.

It is likely that the page being reserved exists in a higher order allocation block, therefore your implementation must split the allocation blocks down (as per the buddy allocation algorithm) until only the page being reserved is allocated.

7.5 Freeing Pages

You must implement the `free_pages` method, which takes in a page descriptor previously allocated by your system, and puts the block back into the free list, coalescing buddies back up to the maxium order as per the buddy allocation algorithm.

7.6 Skeleton

You are provided with a skeleton page allocation algorithm interface, in which you must write your code to implement the *buddy allocator*. The skeleton is commented, and indicates what methods you should fill in, and where you should write your code.

You will find the skeleton in `coursework/buddy.cpp`. Do not copy or move this file anywhere, just open it up in your IDE of choice and edit the file.

7.7 Testing

Because memory allocation is such a fundamental operation, it is quite likely that during the course of you implementing your algorithm the system will either:

1. Not boot at all.
2. Triple fault, and continually restart.
3. Behave very strangely.

Therefore, a good test is: *does the system boot to the shell?* and *can I run programs?*

However, in order to more accurately quantify the success of your implementation, a *self-test* mode is available that will test the memory allocator during start-up. This self-test mode will make a series of allocations and check that standard conditions hold, and will also use the `dump()` method of the allocation algorithm to print out the state of the buddy system. You should use this output to make sure your buddy system is behaving correctly when allocating and freeing pages. The `dump()` method will display the free list for each order. See Appendix A for example output of the self-test mode, which you should use to make sure your own implementation is behaving in a similar fashion.

You can activate this self-test mode by adding `pgalloc.self-test=1` to the command-line, see below for an example, making sure you still specify `pgalloc.algorithm=buddy`.

To compile and run **InfOS**, issue the `build-and-run.sh` command from your `os-coursework` directory, **and tell InfOS to use your page allocator** with the `pgalloc.algorithm=buddy` option e.g.:

```
$ ./build-and-run.sh pgalloc.algorithm=buddy
```

It is important that you put the `pgalloc.algorithm=buddy` option on the command-line, otherwise the built-in allocator will be used instead.

Your source-code will be built, and if there are any errors, these will be displayed to you and the operating system will not load.

To double-check that **InfOS** is using your algorithm, scroll back in the log window and look for the line:

```
notice: *** USING PAGE ALLOCATION ALGORITHM: buddy
```

If your algorithm does not work at all, then the system will likely not boot and may crash. You can use the `pgalloc.debug=1` and `pgalloc.self-test=1` options to produce more debugging output from the memory management core, and to run the self tests.

```
$ ./build-and-run.sh pgalloc.algorithm=buddy pgalloc.debug=1 pgalloc.self-test=1
```

7.8 Marking and Deliverables

This part of the coursework attracts **40 marks**. Marks will be given for correctly implementing the algorithm, for readability/coding style, for use of efficient algorithms, and for inserting appropriate error checking.

You must submit your implementation, by uploading your source-code to **Coursework 3 - The Memory Allocator** in Learn **BEFORE** 4PM GMT on **Thursday 11th March, 2021**. No other form of submission will be accepted, and late submission will attract a score of zero.

8 *Task 4:* TAR File-system Driver

Your fourth coursework task is to implement a file-system driver, that presents TAR archive files as a *virtual* file-system.

It is due by **4PM BST** on **Thursday 25th March 2021** (week 10), and is worth **25 marks out of 100**.

8.1 Introduction

Your final task is to implement a file-system driver that can read TAR files and present them as a virtual file-system.

A TAR file is an uncompressed archive file that contains other files. It was originally created for when cassette tapes[3] were used for data storage. It is a very simple archive format that simply concatenates the contents of each file into one long file, with a header describing each file in the sequence. Quite often, TAR files will be compressed after creation, e.g. by `gzip` to produce a compressed archive. You will recognise such a compressed archive as a file with the extension `.tar.gz`.

This task does **not** deal with compressed TAR archives, only the raw uncompressed archive.

Normally, file-systems are highly optimised on-disk structures that layout files as efficiently as possible. Then, file-system drivers interrogate the on-disk structure, and provide the necessary operations for reading, writing and looking up files to the operating system.

However, creating or even implementing such a file-system driver is non-trivial and so not feasible as a coursework exercise. But, due to the simplicity of the TAR file format, and the ease of which you can create your own archives it is a suitable candidate for this task.

You will be creating a **read-only** file system driver, writing will **not** be supported.

8.2 Background

Existing file-systems, such as `ext4` and `btrfs` are designed to store files in an efficient manner, making full use of the available storage medium and providing efficient access to the file data. Operating systems use file-system drivers to access this on-disk structure, and present the file-system hierarchy to the program/user.

TAR files are simple archives that store files sequentially. Each file in the archive is prefixed with a header that contains information about the file being stored. TAR files are split into blocks of 512 bytes. The first block before a file contains the header, and subsequent blocks contain the file data – up to the end of the file. Then, the following block contains the header for another file. The archive ends with two blocks containing all zeroes.

8.3 Resources

See `https://www.gnu.org/software/tar/manual/html_node/Standard.html` for a description of the TAR file format.

[3] `https://en.wikipedia.org/wiki/Compact_Cassette`

The header block contains numeric fields that are stored as ASCII octal strings. Because of this, you are provided with a helper method (`octal2ui`) that converts an ASCII octal string into a normal integer representation.

8.4 Important Files and Interaction with InfOS

This piece of coursework is spread between two files: `tarfs.cpp` and `tarfs.h`. It is possible to complete the task by only modifying `tarfs.cpp`, but you may find it easier to add helper functions, which may require you to modify the class definitions. Make sure you submit both `tarfs.cpp` and `tarfs.h`, when you submit your work.

This particular task is straightforward, but requires interacting with a number of different **InfOS** subsystems, as described in the following sections.

8.4.1 Block Devices

A block device is a device driver that provides access to a storage medium by allowing the caller to *read* and *write* entire blocks at a time. Your TARFS driver will be interrogating a block device that contains a TAR file. The block size of the block device has been set-up to match the block size of the TAR file: 512 bytes.

To read from a block device, you can use the `read_blocks(buffer, block_nr, count)` method of the block device object, e.g.:

```
int block_size = bdev.block_size();

uint8_t *buffer = new uint8_t[block_size * 2];
bdev.read_blocks(buffer, 0, 2);
delete buffer;
```

This reads two blocks, starting from block zero of the block device into the buffer `buffer`. You will not need to write to the block device in this exercise.

In the `TarFS` class, there is a helper method `block_device()` that returns a reference to the current block device in use for the file system. You will use this in your `build_tree()` implementation (see Section 8.5.1).

8.4.2 File-system Nodes

A file-system presents a hierarchical, tree-based view of files, with directories being the nodes in the tree and files being the leaves. Because **InfOS** employs the virtual filesystem model, there is effectively one master virtual file system tree (for the entire system), with multiple physical file-system trees *mounted* (or *overlayed*) within it. File-system drivers are only concerned with the *physical* file-system tree. This *physical* tree represents the tree of the underlying file-system on the disk, and it's the job of the VFS core to overlay it on top of the *virtual* tree.

To accomplish this, **InfOS** defines two types of file-system nodes:

1. A **Virtual** File-system Node (`VFSNode`), representing a directory or a file in the virtual file-system tree.

2. A **Physical** File-system Node (`PFSNode`), representing a directory or a file in the physical file-system tree.

`VFSNode`s and `PFSNode`s are similar in that they each contain a list of children. There is a *root* node that represents the root of the file-system tree.

When your driver is loaded, it will read the **entire** TAR file, and build the physical file-system tree, by creating `PFSNode`s that represent all of the files and directories in the archive. The TarFS skeleton contains a subclassed `PFSNode` class called `TarFSNode`. This class provides methods for adding child `TarFSNode`s. See the header file (`tarfs.h`) for further details.

8.4.3 Files and Directories

The `PFSNode`s represent the files and directories in the file-system tree, but to access the data within files (or a listing of the children in a directory), these nodes must be *opened* for access. Calling `open()` or `opendir()` on a `PFSNode` returns a `File` or `Directory` object that represents a file or directory that can be read from.

The TarFS skeleton subclasses the `File` and `Directory` classes with `TarFSFile` and `TarFSDirectory` respectively. These classes provide the ability to read and write to files, and to get a listing of children in directories.

8.4.4 Mounting

Mounting a file-system simply refers to overlaying the physical file-system tree onto the virtual file-system tree. The VFS core takes care of this by creating VFSNodes that are attached to corresponding PFSNodes.

8.5 Skeleton

You are provided with a skeleton file-system driver, in which you must write your code to implement the TARFS driver. The skeleton is commented, and indicates what methods you should fill in, and where you should write your code. Some of the ground-work for the driver has already been done for you, and you should refer to this code for examples of usage of certain functions.

The first thing you should tackle is the definition of the `posix_header` structure, which describes the layout of a header block in a TAR file. You'll find descriptions of this structure online, and you should make sure that your structure **exactly** matches this format. This structure contains all the information you need to implement the rest of the driver.

Note that this structure is defined as `packed`, which means that it will be layed out in memory **exactly** as you define it, i.e. the compiler will not insert padding.

After defining this structure, implementing the driver interface requires implementing these methods:

- `build_tree`**:** This function is used during mount time to build the tree representation of the file system. Normally, a file system wouldn't read its entire directory layout into memory, but for simplicity this is what should be done here.

- `pread:` This function is used for reading the data associated with a file. It reads from a particular offset in the file, for a particular length into the supplied buffer.
- `size:` This is a very simple method (a one liner!) that returns the size of the file represented by the TarFSFile object. You can get the size by interrogating the header block of the file.

If you have used the `prepare-coursework.sh` script, you will find the skeleton in `os-coursework/coursework/tarfs.cpp`, and a corresponding header file in `os-coursework/coursework/tarfs.h`. Do not copy or move these files anywhere, just open them up in your IDE of choice and edit the files.

8.5.1 `TarFSNode *TarFS::build_tree()`

The `build_tree` method lives inside the `TarFS` class, which is the main class associated with a file-system. Its primary function is to return a `TarFSNode` (which is a subclass of a `PFSNode`) that represents the root of the physical file-system tree.

In this routine, you will interrogate the block device to read the blocks in the TAR file. You can access the block device with the `block_device()` helper method, e.g.:

```
size_t nr_blocks = block_device().block_count();
syslog.messagef(LogLevel::DEBUG, "Block Device nr-blocks=%lu", nr_blocks);
```

8.5.2 `int TarFSFile::pread(void *buffer, size_t size, off_t off)`

The `pread` method lives inside the `TarFSFile` class, which is the class that represents an *open* file (i.e. one that is ready for reading). When this is called, the file data, starting at offset `off`, and for a length of `size` must be read from the archive and placed in the buffer.

Note that this method is about reading files—not reading the archive. Therefore, the offset refers to the offset within the **current file**. The skeleton already remembers which block within the TAR file the file data starts at, and keeps it in the field called `_file_start_block`.

In the `TarFSFile` class you can access the block device by using the `_owner` field to get at the owning file-system object, e.g.:

```
size_t nr_blocks = _owner.block_device().block_count();
```

You will need to take particular care when reading files that span multiple blocks.

8.5.3 `unsigned int TarFSFile::size() const`

The `size` method again lives inside the `TarFSFile` class, and simply exists to return the size of the file. This information is available in the header block for the file, which has been conveniently stored for you in the `_hdr` field.

8.6 Testing

To compile and run **InfOS**, issue the `build-and-run.sh` command from your `os-coursework` directory, **and tell InfOS to use your TARFS driver** with the `boot-fs-type=tarfs` option

e.g.:

```
$ ./build-and-run.sh boot-fs-type=tarfs
```

It is important that you put the `boot-fs-type=tarfs` option on the command-line, otherwise the built-in file-system driver will be used instead.

Your source-code will be built, and if there are any errors, these will be displayed to you and the operating system will not load.

To double-check that **InfOS** is using your implementation, scroll back in the log window and look for the line:

```
notice: *** USING FILE-SYSTEM DRIVER: tarfs
```

If your implementation does not work, then the system will likely not boot and may crash. If it partially works, you may get to a command prompt, but you should try running some programs, such as `/usr/cat /usr/docs/README` to verify behaviour.

You can also use the `/usr/ls` command to list directories, e.g.:

```
> /usr/ls /
> /usr/ls /usr
> /usr/ls /usr/docs
> /usr/ls /dev
```

8.7 Marking and Deliverables

This part of the coursework attracts **25 marks**. Marks will be given for correctly reading TAR files, exposing the underlying data, for readability/coding style, and for inserting appropriate error checking. There are **two** files associated with this task, both of which should be submitted.

You must submit your implementation, by uploading your source-code to **Coursework 4 - The File System** in Learn **BEFORE** 4PM GMT on **Thursday 25th March, 2021**. No other form of submission will be accepted, and late submission will attract a score of zero.

Make sure you submit both `tarfs.cpp` and `tarfs.h`—especially if you have made any changes to the class definitions.

A Page Allocator Self Test Output

The following listing shows what output you should expect to see from **InfOS** when using the self-test mode for the page allocator. A number of different tests are performed, and the `dump()` method is used to print out the state of the buddy system.

This dump iterates over each allocation order, zero up to the maximum (which is 16), and then follows the chain of blocks in the linked list, printing out their page frame numbers (PFNs). For example, in the initial state (when no allocations have been made), only the highest order contains blocks. You may see slightly variations in the PFNs allocated, depending on how you split blocks, but this output should give you an indication of the expected behaviour.

What is **most** important is that the ending state of the self test matches the initial state, that is no allocations have been made.

```
 notice: mm: PAGE ALLOCATOR SELF TEST - BEGIN
 notice: mm: ------------------------
   info: mm: * INITIAL STATE
  debug: mm: BUDDY STATE:
  debug: mm: [0]
  debug: mm: [1]
  debug: mm: [2]
  debug: mm: [3]
  debug: mm: [4]
  debug: mm: [5]
  debug: mm: [6]
  debug: mm: [7]
  debug: mm: [8]
  debug: mm: [9]
  debug: mm: [10]
  debug: mm: [11]
  debug: mm: [12]
  debug: mm: [13]
  debug: mm: [14]
  debug: mm: [15]
  debug: mm: [16] 0 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm: ------------------------
   info: mm: (1) ALLOCATING ONE PAGE
   info: mm: ALLOCATED PFN: 0x0
  debug: mm: BUDDY STATE:
  debug: mm: [0] 1
  debug: mm: [1] 2
  debug: mm: [2] 4
  debug: mm: [3] 8
  debug: mm: [4] 10
```

```
  debug: mm: [5] 20
  debug: mm: [6] 40
  debug: mm: [7] 80
  debug: mm: [8] 100
  debug: mm: [9] 200
  debug: mm: [10] 400
  debug: mm: [11] 800
  debug: mm: [12] 1000
  debug: mm: [13] 2000
  debug: mm: [14] 4000
  debug: mm: [15] 8000
  debug: mm: [16] 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm: ------------------------
   info: mm: (2) FREEING ONE PAGE
  debug: mm: BUDDY STATE:
  debug: mm: [0]
  debug: mm: [1]
  debug: mm: [2]
  debug: mm: [3]
  debug: mm: [4]
  debug: mm: [5]
  debug: mm: [6]
  debug: mm: [7]
  debug: mm: [8]
  debug: mm: [9]
  debug: mm: [10]
  debug: mm: [11]
  debug: mm: [12]
  debug: mm: [13]
  debug: mm: [14]
  debug: mm: [15]
  debug: mm: [16] 0 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm: ------------------------
   info: mm: (3) ALLOCATING TWO CONTIGUOUS PAGES
   info: mm: ALLOCATED PFN: 0x0
  debug: mm: BUDDY STATE:
  debug: mm: [0]
  debug: mm: [1] 2
  debug: mm: [2] 4
  debug: mm: [3] 8
  debug: mm: [4] 10
  debug: mm: [5] 20
  debug: mm: [6] 40
```

```
  debug: mm: [7] 80
  debug: mm: [8] 100
  debug: mm: [9] 200
  debug: mm: [10] 400
  debug: mm: [11] 800
  debug: mm: [12] 1000
  debug: mm: [13] 2000
  debug: mm: [14] 4000
  debug: mm: [15] 8000
  debug: mm: [16] 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm: ------------------------
   info: mm: (4) FREEING TWO CONTIGUOUS PAGES
  debug: mm: BUDDY STATE:
  debug: mm: [0]
  debug: mm: [1]
  debug: mm: [2]
  debug: mm: [3]
  debug: mm: [4]
  debug: mm: [5]
  debug: mm: [6]
  debug: mm: [7]
  debug: mm: [8]
  debug: mm: [9]
  debug: mm: [10]
  debug: mm: [11]
  debug: mm: [12]
  debug: mm: [13]
  debug: mm: [14]
  debug: mm: [15]
  debug: mm: [16] 0 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm: ------------------------
   info: mm: (5) OVERLAPPING ALLOCATIONS
  debug: mm: BUDDY STATE:
  debug: mm: [0]
  debug: mm: [1]
  debug: mm: [2]
  debug: mm: [3]
  debug: mm: [4]
  debug: mm: [5]
  debug: mm: [6]
  debug: mm: [7]
  debug: mm: [8]
  debug: mm: [9]
```

```
  debug: mm: [10]
  debug: mm: [11]
  debug: mm: [12]
  debug: mm: [13]
  debug: mm: [14]
  debug: mm: [15]
  debug: mm: [16] 0 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm: ------------------------
   info: mm: (6) MULTIPLE ALLOCATIONS, RANDOM ORDER FREE
   info: mm: * AFTER ALLOCATION
  debug: mm: BUDDY STATE:
  debug: mm: [0] b
  debug: mm: [1]
  debug: mm: [2] c
  debug: mm: [3]
  debug: mm: [4] 10
  debug: mm: [5] 20
  debug: mm: [6] 40
  debug: mm: [7] 80
  debug: mm: [8] 100
  debug: mm: [9] 200
  debug: mm: [10] 400
  debug: mm: [11] 800
  debug: mm: [12] 1000
  debug: mm: [13] 2000
  debug: mm: [14] 4000
  debug: mm: [15] 8000
  debug: mm: [16] 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm:   FREE 0x8
  debug: mm: BUDDY STATE:
  debug: mm: [0] 8 b
  debug: mm: [1]
  debug: mm: [2] c
  debug: mm: [3]
  debug: mm: [4] 10
  debug: mm: [5] 20
  debug: mm: [6] 40
  debug: mm: [7] 80
  debug: mm: [8] 100
  debug: mm: [9] 200
  debug: mm: [10] 400
  debug: mm: [11] 800
  debug: mm: [12] 1000
```

```
  debug: mm: [13] 2000
  debug: mm: [14] 4000
  debug: mm: [15] 8000
  debug: mm: [16] 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm:   FREE 0x4
  debug: mm: BUDDY STATE:
  debug: mm: [0] 8 b
  debug: mm: [1]
  debug: mm: [2] 4 c
  debug: mm: [3]
  debug: mm: [4] 10
  debug: mm: [5] 20
  debug: mm: [6] 40
  debug: mm: [7] 80
  debug: mm: [8] 100
  debug: mm: [9] 200
  debug: mm: [10] 400
  debug: mm: [11] 800
  debug: mm: [12] 1000
  debug: mm: [13] 2000
  debug: mm: [14] 4000
  debug: mm: [15] 8000
  debug: mm: [16] 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm:   FREE 0x9
  debug: mm: BUDDY STATE:
  debug: mm: [0] b
  debug: mm: [1] 8
  debug: mm: [2] 4 c
  debug: mm: [3]
  debug: mm: [4] 10
  debug: mm: [5] 20
  debug: mm: [6] 40
  debug: mm: [7] 80
  debug: mm: [8] 100
  debug: mm: [9] 200
  debug: mm: [10] 400
  debug: mm: [11] 800
  debug: mm: [12] 1000
  debug: mm: [13] 2000
  debug: mm: [14] 4000
  debug: mm: [15] 8000
  debug: mm: [16] 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
```

```
   info: mm:   FREE 0xa
  debug: mm: BUDDY STATE:
  debug: mm: [0]
  debug: mm: [1]
  debug: mm: [2] 4
  debug: mm: [3] 8
  debug: mm: [4] 10
  debug: mm: [5] 20
  debug: mm: [6] 40
  debug: mm: [7] 80
  debug: mm: [8] 100
  debug: mm: [9] 200
  debug: mm: [10] 400
  debug: mm: [11] 800
  debug: mm: [12] 1000
  debug: mm: [13] 2000
  debug: mm: [14] 4000
  debug: mm: [15] 8000
  debug: mm: [16] 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm:   FREE 0x2
  debug: mm: BUDDY STATE:
  debug: mm: [0] 2
  debug: mm: [1]
  debug: mm: [2] 4
  debug: mm: [3] 8
  debug: mm: [4] 10
  debug: mm: [5] 20
  debug: mm: [6] 40
  debug: mm: [7] 80
  debug: mm: [8] 100
  debug: mm: [9] 200
  debug: mm: [10] 400
  debug: mm: [11] 800
  debug: mm: [12] 1000
  debug: mm: [13] 2000
  debug: mm: [14] 4000
  debug: mm: [15] 8000
  debug: mm: [16] 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm:   FREE 0x1
  debug: mm: BUDDY STATE:
  debug: mm: [0] 1 2
  debug: mm: [1]
  debug: mm: [2] 4
```

```
  debug: mm: [3] 8
  debug: mm: [4] 10
  debug: mm: [5] 20
  debug: mm: [6] 40
  debug: mm: [7] 80
  debug: mm: [8] 100
  debug: mm: [9] 200
  debug: mm: [10] 400
  debug: mm: [11] 800
  debug: mm: [12] 1000
  debug: mm: [13] 2000
  debug: mm: [14] 4000
  debug: mm: [15] 8000
  debug: mm: [16] 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm:   FREE 0x0
  debug: mm: BUDDY STATE:
  debug: mm: [0] 2
  debug: mm: [1] 0
  debug: mm: [2] 4
  debug: mm: [3] 8
  debug: mm: [4] 10
  debug: mm: [5] 20
  debug: mm: [6] 40
  debug: mm: [7] 80
  debug: mm: [8] 100
  debug: mm: [9] 200
  debug: mm: [10] 400
  debug: mm: [11] 800
  debug: mm: [12] 1000
  debug: mm: [13] 2000
  debug: mm: [14] 4000
  debug: mm: [15] 8000
  debug: mm: [16] 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm:   FREE 0x3
   info: mm: * AFTER RANDOM ORDER FREEING
  debug: mm: BUDDY STATE:
  debug: mm: [0]
  debug: mm: [1]
  debug: mm: [2]
  debug: mm: [3]
  debug: mm: [4]
  debug: mm: [5]
  debug: mm: [6]
```

```
  debug: mm: [7]
  debug: mm: [8]
  debug: mm: [9]
  debug: mm: [10]
  debug: mm: [11]
  debug: mm: [12]
  debug: mm: [13]
  debug: mm: [14]
  debug: mm: [15]
  debug: mm: [16] 0 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm: ------------------------
   info: mm: (7) RESERVING PAGE 0x14e000 and 0x14f000
  debug: mm: BUDDY STATE:
  debug: mm: [0]
  debug: mm: [1] 14c
  debug: mm: [2] 148
  debug: mm: [3] 140
  debug: mm: [4] 150
  debug: mm: [5] 160
  debug: mm: [6] 100
  debug: mm: [7] 180
  debug: mm: [8] 0
  debug: mm: [9] 200
  debug: mm: [10] 400
  debug: mm: [11] 800
  debug: mm: [12] 1000
  debug: mm: [13] 2000
  debug: mm: [14] 4000
  debug: mm: [15] 8000
  debug: mm: [16] 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm: ------------------------
   info: mm: (8) FREEING RESERVED PAGE 0x14f000
  debug: mm: BUDDY STATE:
  debug: mm: [0] 14f
  debug: mm: [1] 14c
  debug: mm: [2] 148
  debug: mm: [3] 140
  debug: mm: [4] 150
  debug: mm: [5] 160
  debug: mm: [6] 100
  debug: mm: [7] 180
  debug: mm: [8] 0
  debug: mm: [9] 200
```

```
  debug: mm: [10] 400
  debug: mm: [11] 800
  debug: mm: [12] 1000
  debug: mm: [13] 2000
  debug: mm: [14] 4000
  debug: mm: [15] 8000
  debug: mm: [16] 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm: ------------------------
   info: mm: (9) FREEING RESERVED PAGE 0x14e000
  debug: mm: BUDDY STATE:
  debug: mm: [0]
  debug: mm: [1]
  debug: mm: [2]
  debug: mm: [3]
  debug: mm: [4]
  debug: mm: [5]
  debug: mm: [6]
  debug: mm: [7]
  debug: mm: [8]
  debug: mm: [9]
  debug: mm: [10]
  debug: mm: [11]
  debug: mm: [12]
  debug: mm: [13]
  debug: mm: [14]
  debug: mm: [15]
  debug: mm: [16] 0 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000
c0000 d0000 e0000 f0000 100000 110000 120000 130000 140000
   info: mm: ------------------------
   info: mm: PAGE ALLOCATOR SELF TEST - COMPLETE
```