Operating Systems 2022-23
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 **50**.

The coursework is split into separate tasks, each of which must be submitted individually:

- **Task 0 (formative):** Write a `"Hello, world!"` program in user space. `[0 marks]`
  (submission closes Thursday 9th of February 2023 at 12pm GMT)
- **Task 1:** Implement a priority task scheduler in the kernel. `[17 marks]`
  (due Thursday 2nd of March 2023 at 12pm GMT)
- **Task 2:** Implement a buddy memory allocator in the kernel. `[17 marks]`
  (due Thursday 16th of March 2023 at 12pm GMT)
- **Task 3:** Implement a block layer cache for devices. `[16 marks]`
  (due Thursday 30th of March 2022 at 12pm GMT)

You may start any of the tasks as early as you wish, but you **must** submit each task before the respective deadline. Coursework submitted after the deadline will attract a score of **zero**. However, if you require an extension, you must contact the ITO **before** the deadline. The policy is **Rule 1:**Extensions are permitted (7 days) and Extra Time Adjustments (ETA) are permitted and can be combined (more details on the Assessment Learn page).

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 **Marking and Deliverables** section for each 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. You will be marked on correctness, coding style and efficiency. You are expected to thoroughly test your code before submission, and coursework that does not compile will also attract a score of **zero**.

1.1 Background

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

x86 operating system that has been written from scratch in C++ (mostly).

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 adhere to the POSIX standards. It is modular in the sense that functionality can be plugged in/out at compile time, but **InfOS** 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 broaden 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 the first two tasks, you will be implementing a piece of core operating system functionality to be loaded into the **InfOS** kernel, and for the third task, you will be implementing a program in user space. To ensure that the first two tasks are mutually exclusive, and that your implementations do not prejudice each other, **InfOS** is shipped with basic scheduling and memory management 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 the other.

1.3.1 Development Environment

InfOS has been tested on DICE, so you are encouraged to develop your coursework on DICE. 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`

You should also be able to develop locally on any Linux machine, if you have a modern C++ compiler and the QEMU emulator installed. Unfortunately, you cannot use Virtual DICE for this coursework. This is because the Virtual DICE software repository is only a subset of the DICE software repository, and does not contain QEMU. If you want to work in a virtual machine, perhaps because of poor internet connection, it is recommended that you set up a Ubuntu virtual machine instead. Once you have acquired the **InfOS** source-code, you can load it into an IDE of your choice.

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. These scripts have been tested on DICE, but should also work on any Linux distribution if QEMU is installed in a standard location (e.g. if you installed it using 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.

[1] https://tcs6.host.cs.st-andrews.ac.uk/

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
```

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.

IMPORTANT NOTE If you are using Remote DICE, you will need to use slightly different scripts, because Remote DICE makes people on the same remote system share VNC sessions. In your compilation terminal, use the build script and the run script separately:

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

Then, in your viewer terminal, use the view script:

```
$ ./view-rdice.sh
```

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-mq.cpp
```

Warning! This will permanently erase **any** changes you have made to the `sched-mq.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/infos/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 base 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.

2.9 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.

3 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!)

4 *Task 0:* "Hello, world!"

This **formative** task is designed to make sure you are comfortable with the **InfOS** development environment, and so is very straightforward to implement. You are strongly encouraged to attempt this task, but it **will not** contribute towards your final grade. You should take this as an opportunity to become familiar with the **InfOS** source code, and C++ programming in general.

4.1 Introduction

`"Hello, world!"` programs are often used to help a student become familiar with a new programming environment, and **InfOS** is no exception. Your task is to print `"Hello, world!"` to the **InfOS** console.

4.2 Skeleton

You are provided with a skeleton user space program, in which you must write your code. You will find the skeleton in `infos-user/src/hello-world/main.cpp`. Do not copy or move this file out of this directory, just open it up in your IDE of choice and edit it.

4.3 Testing

Your source code will automatically be compiled into the **InfOS** user space. To compile and run **InfOS**, issue the `build-and-run.sh` command from your `infos-coursework` directory like so:

```
$ ./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. If the system boots up to the **InfOS** shell, you can then issue the command `/usr/hello-world`, at which point, you should see your output in the **InfOS** console.

4.4 Marking and Deliverables

As mentioned, this part of the coursework is purely formative, and **will not** be marked. However, if you wish to familiarise yourself with the submission process, you may submit your implementation by uploading the source code file `main.cpp` to the Learn box **Coursework 0 - Hello, world!**. The submission box will close at 12:00 GMT on **Thursday 9th February 2023**.

5 *Task 1:* The Priority Task Scheduler

Your **first summative** coursework task is to implement the multiple queue priority scheduling algorithm discussed in lectures.

It is due by **12:00 GMT** on **Thursday 2nd March 2023**, and is worth **17 marks out of 50**.

5.1 Introduction

The task scheduler of an operating system is the component that decides which tasks get to run on the processor. When there is only a single physical processor in a system, that processor must 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 processor.

RUNNING The task is currently running on a processor.

SLEEPING The task is waiting for an event to occur before it can become runnable.

The version of **InfOS** that you will be working with is unicore, so you only need to think about scheduling on one processor. Only **runnable** tasks can be scheduled onto a CPU. Ideally, a scheduler will make sure that every task gets a chance to run.

The multiple queue priority scheduling algorithm partitions the runqueue into several separate queues, and each task is permanently assigned to a queue based on its priority. A process can have one of four priorities, ordered here from highest to lowest priority:

REALTIME The highest priority in the system, reserved for time critical tasks.

INTERACTIVE Intended for interactive tasks which are not time critical, but should be very responsive to user interaction.

NORMAL The default priority level for a standard task. Tasks created without an explicit priority level are designated as normal tasks.

DAEMON The lowest priority in the system. Inspired by the imaginary agent in physics, Maxwell's demon, daemons are background tasks that work constantly to perform system chores.

For a task in a particular queue to be scheduled, all the higher priority queues must be empty at the point when the scheduling event occurs. That is, on every timer tick, the runnable task with the highest priority should be scheduled to execute. If multiple tasks of the same priority are waiting to execute, they should be scheduled in a round robin fashion until all of those tasks are completed.

5.2 Important Files

The generic scheduler core exists in the `infos/kernel/sched.cpp` source code file, and this is where you can find the majority of the scheduling subsystem. You are **not** required to change this code, but it is useful to know where this is so that you can understand how scheduling works within **InfOS**. When a scheduling event occurs, such as a preemptive timer tick, the scheduler core calls into a **scheduling algorithm** to ask for a task to run. The **scheduling algorithm** is what you must implement.

The generic scheduler core will automatically detect your algorithm when you build **InfOS**, but you **must** tell **InfOS** to use your algorithm, otherwise it will use the built-in scheduler. See Section 5.4 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 processor (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.

InfOS has the notion of a `SchedulingEntity`, which is an abstraction that allows easily switching between scheduling entire processes or scheduling individual threads. Currently, **InfOS** schedules individual threads.

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 `infos/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. However, you should be cautious: locking is expensive and may impede the rest of the system, so only use it where necessary.

5.3 Skeleton

You are provided with a skeleton scheduling algorithm interface, in which you must write your code to implement the **multiple queue priority scheduler**. The skeleton is commented to indicate where you should write your code. You will find the skeleton in `coursework/sched-mq.cpp`. Do not copy or move this file out of the coursework directory, just open it up in your IDE of choice and edit it.

5.4 Testing

The algorithm can be chosen with a command-line parameter, like so:

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

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

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

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

```
notice: *** USING SCHEDULER ALGORITHM: mq
```

If your algorithm does not work at all, then the system will likely not boot or crash. You can debug your code by adding logging to the scheduling algorithm and using the `sched.debug=1` option to print that debugging output:

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

However, you should be aware that adding **lots** of logging during a scheduling event will increase the chance of a race condition happening, which may cause the system to crash. This is nothing to worry about, since logging is just for debugging purposes. You should remove the logging before your final submission, and if you're curious, you can inspect the logging code to find out why this happens.

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

```
> /usr/prio-sched-test
```

You can view the source code for the scheduler test within `infos-user/src/prio-sched-test` and use this to determine the correct ordering of task execution. You should then inspect the scheduler test console output to decide whether your implementation is prioritising tasks correctly. This scheduler test is provided as an example only, and you should also write your own tests to verify that your solution is correct.

5.5 Advanced Task

1. The multiple queue priority algorithm allows for greater flexibility during scheduling, but it is not perfect. Describe a common problem with the algorithm, and suggest a potential solution, making sure to explain how your suggestion addresses the problem. Your solution should **not** be any of the algorithms already discussed in lectures, but instead, the result of your own independent research. If your new solution has any drawbacks, be sure to acknowledge those. You should write your answer in a text file called `cw1.txt` or a PDF file called `cw1.pdf`, making sure you cite all sources appropriately.
2. Implement your proposed solution. To do this, you will need to create a new class called `coursework/sched-adv.cpp` that implements the `SchedulingAlgorithm` interface and implements the three methods listed above. The friendly name of your algorithm should be `adv`. Marks will be awarded for both correctness and creativity.

5.6 Marking and Deliverables

This part of the coursework attracts **17 marks**. Marks will be given for correctness, efficiency, coding style and the inclusion of error checking. You must submit your implementation by

uploading the source code file `sched-mq.cpp` to the Learn box **Coursework 1 - The Scheduler before** 12:00 GMT on **Thursday 2nd March 2023**. No other form of submission will be accepted, and late submissions will be subject to the policy detailed here under rule three. If you have attempted the advanced tasks, you should also upload `cw1.txt`/`cw1.pdf` and `sched-adv.cpp` to the same Learn box before the deadline.

6 *Task 2:* The Buddy Page Allocator

Your **second summative** coursework task is to implement a physical memory allocator based on the buddy allocation algorithm.

It is due by **12:00 GMT** on **Thursday 16th March 2023**, and is worth **17 marks out of 50**.

6.1 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. However, 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].

6.2 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 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

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

definition when implementing your algorithm. 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 per-order free lists. There is also an additional pointer, which you may wish to use in your implementation to improve efficiency. However, you are not required to use this for full correctness marks. This pointer is the `prev_free` pointer, which can also be found in the page descriptor. 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.

6.3 Important Files

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, and this allocator 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, otherwise it will use the built-in page allocator. See Section 6.7 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 **at least** 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*

- `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.
This method is a *helper method*

- `allocate_pages`**:** This is called by the memory management 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.

- `remove_page_range`**:** This is called by the memory management core during initialisation to mark a range of pages as unavailable for allocation. For example, this would be called to mark the pages in memory that contain the kernel code as unavailable.

- `insert_page_range`**:** This is called by the memory management core to mark a number of contiguous pages as available for allocation. This may be done during initialisation, or when the system no longer needs pages that were previously marked as unavailable.

- `init`**:** This is called by the memory management core during start-up, so that the algorithm can initialise its internal state. Note that you **cannot** assume all memory is free initially, so you must only insert pages into the free lists when the function `insert_page_range` is called.

Note: Three of the above methods are helper methods, and you should use them in your implementation of the other methods. You will probably also find it useful to define some additional helper functions of your own.

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.

6.4 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 `allocate_pages`

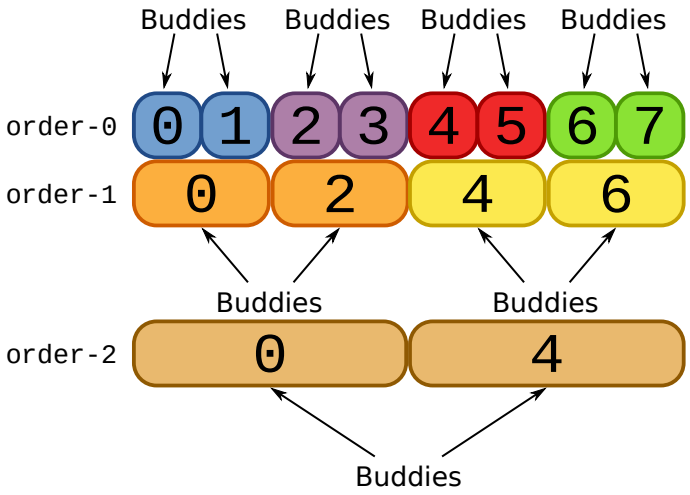

Figure 1: An illustration of block buddy relationships, in different allocation orders.

2. Calling `remove_page_range`

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

6.4.1 `PageDescriptor *allocate_pages(int order) override`

The `allocate_pages` method is called when a contiguous number of pages need 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.

6.4.2 `void remove_page_range(PageDescriptor *start, uint64_t count)`

This method is called when a **particular** set of pages must be made unavailable. The start page and the number of pages to remove are passed in by the caller. It is likely that the pages being reserved exist in a higher order allocation block, therefore your implementation must split the allocation blocks down (as per the buddy allocation algorithm) until only the pages being reserved are allocated.

6.5 Freeing Pages

Again, there are two ways to free a page (or pages). You must implement both the `free_pages` method used by the memory management core and the `insert_page_range` method used by the kernel. Free pages should be put back into the free lists, coalescing buddies back up to the maximum order as per the buddy allocation algorithm.

6.6 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 to indicate 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 to edit the file.

6.7 Testing

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

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 to 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_state()` 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_state()` 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`. You can also use the `pgalloc.debug=1` option to produce more debugging output from the memory management core, like so:

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

6.8 Advanced Task

The buddy allocator you have implemented is very famous and is widely adopted in the real world (e.g., Linux [3] and FreeBSD [4]). However, it suffers from internal fragmentation [5] and some memory can go wasted, if not used very carefully (e.g., by layering on top of it other, more efficient allocators, such as slab allocators[6]). For example, for an allocation of 20KiB, the buddy algorithm will return a memory block of 32KiB which results in an internal fragmentation of 12KiB.

A Fibonacci allocator, on the other hand, might help to overcome this problem. The Fibonnaci allocator work in a similar way to the Buddy allocator, but instead of page sizes which are powers of two, it uses page sizes following a Fibonacci sequence [7]. That is, page sizes of 1x4KiB, 2x4KiB, 3x4KiB, 5x4KiB, 8x4KiB, 13x4KiB, 21x4KiB, etc. Considering the example given above, with a Fibonacci allocator, the request will return exactly 20KiB, wasting no memory on internal fragmentation.

For the advanced task, we ask you to implement the Fibonacci allocator in **InfOS** and then compare the memory fragmentation with the Buddy allocator.

1. First, refer to the various sources on the Internet and write a correct implementation of the Fibonacci allocator. **InfOS** might *not* be stable with the Fibonacci allocator so we would suggest that for the advanced task you divide the memory into two halves each of which is managed by a different allocator. The first half is managed by the buddy allocator (from which **InfOS** allocations should take place as before). The other half is managed by the Fibonnaci allocator. You may implement Fibonacci allocator in the same namespace/file as the buddy alloc (but with different API).

2. Second, compare the estimated internal fragmentation between the Fibonacci allocator and the Buddy allocator you implemented in the previous section. For this part, you have to define a global counter in InfOS that accumulates the amount of internal fragmentation (following the same logic of the example given above, if an allocation of 20KiB is requested and you return 32KiB then you add 12KiB of internal fragmentation to the counter). Run experiments under both the algorithms, gather the statistics and make a comparison in your final report (`cw2.pdf`) which you need to submit along with your code. We do not require any specific experimentation protocol, design what you see fits the best the purpose of the task.

6.9 Marking and Deliverables

This part of the coursework attracts **17 marks**. Marks will be given for correctness, efficiency, coding style and the inclusion of error checking, which is particularly important for this task.

[3] https://www.kernel.org/doc/gorman/html/understand/understand009.html

[4] McKusick, M.K., Neville-Neil, G.V. and Watson, R.N., 2014. The design and implementation of the FreeBSD operating system. Pearson Education.

[5] https://epubs.siam.org/doi/pdf/10.1137/0206044

[6] https://www.kernel.org/doc/gorman/html/understand/understand011.html

[7] https://en.wikipedia.org/wiki/Fibonacci$_n umber$

You must submit your implementation by uploading the source code file `coursework/buddy.cpp` to the Learn box **Coursework 2 - The Buddy Allocator before** 12:00 GMT on **Thursday 16th March 2023**. No other form of submission will be accepted, and late submissions will be subject to the policy detailed here under rule three. If you have attempted the advanced tasks, you should also upload `cw2.txt`/`cw2.pdf` and `mem-adv.cpp` to the same Learn box before the deadline.

7 *Task 3:* The file buffer cache (block cache) over ATA-device

Your **final summative** coursework task is to implement a block cache for **InfOS** that caches the recently accessed (file) blocks.

It is due by **12:00 GMT** on **Thursday 30th March 2023**, and is worth **16 marks out of 50**.

7.1 Introduction

Block devices (e.g., HDDs, CD-ROM drives, SSDs) organize data as a linear collection of fixed-size blocks. To speed up access to the physical block devices, an OS usually maintains an in-memory (kernel) buffer cache which stores some (or all) recently accessed blocks.

Reading and writing data to a block device first passes through into the buffer cache. Upon a file access, if a block is available in the buffer cache, the cache will save the system an access to the secondary storage, and thus, optimize performance.

To avoid growing indefinitely over time, buffer caches might evict some cached blocks based on some replacement policy. In this task we are primarily interested in an LRU (Least recently used) replacement policy according to which the least recently accessed (older) block is evicted.

For this task you would need to implement a **read-only** buffer cache for the ATA-device. Reads should first search for the block into the cache before performing an I/O. Recall from section 2.8 that **InfOS** cannot use the standard C++ library, so you must write any `libc` functions that you require.

7.2 Skeleton

In InfOS the block layer cache should be placed on top of the ATA-block-device. That said, you would probably need to extend the body of `read_blocks()` functions in `drivers/ata/ata-device.cpp`. In this task you are not provided with skeleton files—you should come up with the design by yourself!

Your code solution should be in `page-cache.h` and/or `page-cache.cpp`. For our testing, these files should be placed under the `infos/include/infos/drivers/ata/` and `infos/drivers/ata/` directories respectively. The cache needs to have a fixed size of 64 blocks and for us to be able to test it with various cache sizes this should be a constant variable.

IMPORTANT Note: For the final submission, upon a block access your code should print the offset of the block and whether it is a cache miss or cache hit (e.g., `debug: cache: [CACHE HIT] offset=686`).

In addition, you might want to modify the `include/infos/drivers/ata/ata-device.h` and `drivers/ata/ata-device.cpp` files but no other code files should be altered. If you need to update the `Makefiles`, you should also include them in the submission (explain in `cw3.txt` what are your modifications there). Lastly, you would need to write a small `cw3.txt` file that *(i)* explains the high-level design (e.g., the API, what data structures did you use and why), *(ii)* the performance complexities of the read and the eviction policy and lastly, *(iii)* how do

you ensure correctness/consistency in your cache. Also, can you briefly discuss any limitations (performance, space) of your code (if any)?

This file should directly answers the questions and be brief and concrete—at most two/three paragraphs.

7.3 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, the operating system might not load. Unlike the other tasks, this task **does not** require special command-line arguments.

To check for correctness of your implementation you can use the `/usr/cat` command to print files of the filesystem (*read path*). For example, you can create custom files, e.g., `/usr/testdir/test` and either use `cat` command or write a userspace program that access the secondary storage in a specific pattern. For example, assuming a test file `/usr/testdir/test` that has content (`1 2 3 4 5 6 7 8 9 10 11 12`), you can execute the `/usr/cat /usr/testdir/test` twice. The first time you should observe some missed blocks (the number of the misses depends on whether the `/usr/cat` has been invoked before).

```
debug:  cache:  [CACHE MISS] offset=1061
```

The second time you should expect only cache hits as everything should be present in the buffer cache due to previous command.

```
debug:  cache:  [CACHE HIT] offset=1061
```

7.4 Advanced Task

1. For the advanced part you would need to implement another eviction/replacement policy for your read-only cache. You might get ideas from various sources in the internet. Feel free to create a second fork of the `page-cache-2.h/cpp`.
2. By running simple commands in **InfOS** can you compare the two policies you have implemented (e.g., by measuring the cache misses) ? Explain the behavior of these two caches and their performance briefly in the report.

7.5 Marking and Deliverables

This part of the coursework attracts **16 marks**. Marks will be given for correctness, efficiency, design and coding style and the inclusion of error checking. You must submit your implementation by uploading the following files (if modified) `include/infos/drivers/ata/ata-device.h`, `drivers/ata/ata-device.cpp`, `page-cache.h` and/or `page-cache.cpp` code files to the Learn box **Coursework 3 - The file buffer cache before** 12:00 GMT on **Thursday 30th March 2023**. If you have attempted the advanced tasks, you should also upload the respective files to the same Learn box before the deadline. No other form of submission will be accepted, and late

submissions will be subject to the policy detailed here under rule three. If you have attempted the advanced tasks, you should also upload `cw3.txt/cw3.pdf` to the same Learn box before the deadline.

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 18), 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), most blocks are contained in the highest order. You may see slight 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, when 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] 7 9e 3901
 debug: mm: [1] 9c 3902
 debug: mm: [2] 98 3904
 debug: mm: [3] 8 90 3908
 debug: mm: [4] 10 80 3910
 debug: mm: [5] 20 3920 bffc0
 debug: mm: [6] 40 3940 bff80
 debug: mm: [7] 3980 bff00
 debug: mm: [8] bfe00
 debug: mm: [9] 3a00 bfc00
 debug: mm: [10] 3c00 bf800
 debug: mm: [11] bf000
 debug: mm: [12] be000
 debug: mm: [13] bc000
 debug: mm: [14] 4000 b8000
 debug: mm: [15] 8000 b0000
 debug: mm: [16] 10000 a0000
 debug: mm: [17] 20000 80000
 debug: mm: [18] 40000 100000 140000 180000
  info: mm: ------------------------
  info: mm: (1) ALLOCATING ONE PAGE
  info: mm: ALLOCATED PFN: 0x7
 debug: mm: BUDDY STATE:
 debug: mm: [0] 9e 3901
 debug: mm: [1] 9c 3902
 debug: mm: [2] 98 3904
 debug: mm: [3] 8 90 3908
```

```
debug: mm: [4] 10 80 3910
debug: mm: [5] 20 3920 bffc0
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm: ------------------------
 info: mm: (2) FREEING ONE PAGE
debug: mm: BUDDY STATE:
debug: mm: [0] 7 9e 3901
debug: mm: [1] 9c 3902
debug: mm: [2] 98 3904
debug: mm: [3] 8 90 3908
debug: mm: [4] 10 80 3910
debug: mm: [5] 20 3920 bffc0
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm: ------------------------
 info: mm: (3) ALLOCATING TWO CONTIGUOUS PAGES
 info: mm: ALLOCATED PFN: 0x9c
debug: mm: BUDDY STATE:
debug: mm: [0] 7 9e 3901
debug: mm: [1] 3902
debug: mm: [2] 98 3904
debug: mm: [3] 8 90 3908
```

```
debug: mm: [4] 10 80 3910
debug: mm: [5] 20 3920 bffc0
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm: ------------------------
 info: mm: (4) FREEING TWO CONTIGUOUS PAGES
debug: mm: BUDDY STATE:
debug: mm: [0] 7 9e 3901
debug: mm: [1] 9c 3902
debug: mm: [2] 98 3904
debug: mm: [3] 8 90 3908
debug: mm: [4] 10 80 3910
debug: mm: [5] 20 3920 bffc0
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm: ------------------------
 info: mm: (5) OVERLAPPING ALLOCATIONS
debug: mm: BUDDY STATE:
debug: mm: [0] 7 9e 3901
debug: mm: [1] 9c 3902
debug: mm: [2] 98 3904
debug: mm: [3] 8 90 3908
debug: mm: [4] 10 80 3910
```

```
debug: mm: [5] 20 3920 bffc0
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm: ------------------------
 info: mm: (6) MULTIPLE ALLOCATIONS, RANDOM ORDER FREE
 info: mm: * AFTER ALLOCATION
debug: mm: BUDDY STATE:
debug: mm: [0]
debug: mm: [1]
debug: mm: [2] 3904
debug: mm: [3] 8 90 3908
debug: mm: [4] 10 80 3910
debug: mm: [5] 20 3920 bffc0
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm:   FREE 0x9d
debug: mm: BUDDY STATE:
debug: mm: [0] 9d
debug: mm: [1]
debug: mm: [2] 3904
debug: mm: [3] 8 90 3908
debug: mm: [4] 10 80 3910
debug: mm: [5] 20 3920 bffc0
```

```
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm:   FREE 0x98
debug: mm: BUDDY STATE:
debug: mm: [0] 9d
debug: mm: [1]
debug: mm: [2] 98 3904
debug: mm: [3] 8 90 3908
debug: mm: [4] 10 80 3910
debug: mm: [5] 20 3920 bffc0
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm:   FREE 0x3902
debug: mm: BUDDY STATE:
debug: mm: [0] 9d 3902
debug: mm: [1]
debug: mm: [2] 98 3904
debug: mm: [3] 8 90 3908
debug: mm: [4] 10 80 3910
debug: mm: [5] 20 3920 bffc0
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
```

```
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm:   FREE 0x3903
debug: mm: BUDDY STATE:
debug: mm: [0] 9d
debug: mm: [1] 3902
debug: mm: [2] 98 3904
debug: mm: [3] 8 90 3908
debug: mm: [4] 10 80 3910
debug: mm: [5] 20 3920 bffc0
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm:   FREE 0x3901
debug: mm: BUDDY STATE:
debug: mm: [0] 9d 3901
debug: mm: [1] 3902
debug: mm: [2] 98 3904
debug: mm: [3] 8 90 3908
debug: mm: [4] 10 80 3910
debug: mm: [5] 20 3920 bffc0
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
```

```
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm:   FREE 0x9e
debug: mm: BUDDY STATE:
debug: mm: [0] 9d 9e 3901
debug: mm: [1] 3902
debug: mm: [2] 98 3904
debug: mm: [3] 8 90 3908
debug: mm: [4] 10 80 3910
debug: mm: [5] 20 3920 bffc0
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm:   FREE 0x7
debug: mm: BUDDY STATE:
debug: mm: [0] 7 9d 9e 3901
debug: mm: [1] 3902
debug: mm: [2] 98 3904
debug: mm: [3] 8 90 3908
debug: mm: [4] 10 80 3910
debug: mm: [5] 20 3920 bffc0
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
```

```
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm:   FREE 0x9c
 info: mm: * AFTER RANDOM ORDER FREEING
debug: mm: BUDDY STATE:
debug: mm: [0] 7 9e 3901
debug: mm: [1] 9c 3902
debug: mm: [2] 98 3904
debug: mm: [3] 8 90 3908
debug: mm: [4] 10 80 3910
debug: mm: [5] 20 3920 bffc0
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm: ------------------------
 info: mm: (7) RESERVING PAGE 0x4d80 and 0x4d84
debug: mm: BUDDY STATE:
debug: mm: [0] 7 9e 3901 4d81 4d85
debug: mm: [1] 9c 3902 4d82 4d86
debug: mm: [2] 98 3904
debug: mm: [3] 8 90 3908 4d88
debug: mm: [4] 10 80 3910 4d90
debug: mm: [5] 20 3920 4da0 bffc0
debug: mm: [6] 40 3940 4dc0 bff80
debug: mm: [7] 3980 4d00 bff00
debug: mm: [8] 4c00 bfe00
debug: mm: [9] 3a00 4e00 bfc00
debug: mm: [10] 3c00 4800 bf800
debug: mm: [11] 4000 bf000
debug: mm: [12] 5000 be000
debug: mm: [13] 6000 bc000
debug: mm: [14] b8000
debug: mm: [15] 8000 b0000
```

```
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm: ------------------------
 info: mm: (8) FREEING RESERVED PAGE 0x4d84
debug: mm: BUDDY STATE:
debug: mm: [0] 7 9e 3901 4d81
debug: mm: [1] 9c 3902 4d82
debug: mm: [2] 98 3904 4d84
debug: mm: [3] 8 90 3908 4d88
debug: mm: [4] 10 80 3910 4d90
debug: mm: [5] 20 3920 4da0 bffc0
debug: mm: [6] 40 3940 4dc0 bff80
debug: mm: [7] 3980 4d00 bff00
debug: mm: [8] 4c00 bfe00
debug: mm: [9] 3a00 4e00 bfc00
debug: mm: [10] 3c00 4800 bf800
debug: mm: [11] 4000 bf000
debug: mm: [12] 5000 be000
debug: mm: [13] 6000 bc000
debug: mm: [14] b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm: ------------------------
 info: mm: (9) FREEING RESERVED PAGE 0x4d80
debug: mm: BUDDY STATE:
debug: mm: [0] 7 9e 3901
debug: mm: [1] 9c 3902
debug: mm: [2] 98 3904
debug: mm: [3] 8 90 3908
debug: mm: [4] 10 80 3910
debug: mm: [5] 20 3920 bffc0
debug: mm: [6] 40 3940 bff80
debug: mm: [7] 3980 bff00
debug: mm: [8] bfe00
debug: mm: [9] 3a00 bfc00
debug: mm: [10] 3c00 bf800
debug: mm: [11] bf000
debug: mm: [12] be000
debug: mm: [13] bc000
debug: mm: [14] 4000 b8000
debug: mm: [15] 8000 b0000
debug: mm: [16] 10000 a0000
```

```
debug: mm: [17] 20000 80000
debug: mm: [18] 40000 100000 140000 180000
 info: mm: ------------------------
 info: mm: PAGE ALLOCATOR SELF TEST - COMPLETE
```