CS3104 Operating Systems 2025-26

Exercise Class

1 Introduction

This special *one-off* exercise class for CS3104 is designed to help you get familiar with:

1. C++ programming.
2. Using the **StACSOS** environment.
3. Developing OS-level code.

It contains a number of exercises to work through, do as much or as little as you feel you need, and ask questions. If you have C++ programming experience already, you may find it beneficial to skip ahead to the **StACSOS** section (Section 3).

2 C++

In this section of the exercise class, you'll be writing general C++ code, i.e. not OS programming, so you can use things like the standard library, etc. It's designed to get you up to speed with syntax and semantics, and to bridge the gap between C programming (from CS2002) and Java programming (from previous modules).

2.1 Environment

Create a new directory in your network home directory (i.e. under your `$HOME/Documents` directory) for this exercise, and open up VS Code:

```
$ mkdir ~/Documents/CS3104-Exercise
$ cd ~/Documents/CS3104-Exercise
$ code -n .
```

Once in VS Code, pop open a terminal so you can compile and run the code you're going to write.

2.2 Hello World

The classic first program to write is *Hello World*. Create a new file called `hello.cpp` in your directory, and enter the following:

```
#include <iostream>

int main(int argc, const char *argv[])
{
    std::cout << "Hello, world." << std::endl;
    return 0;
}
```

Save the file, then in your terminal, compile and run the program:

```
$ g++ -o hello hello.cpp && ./hello
```

If all goes well, you should see "`Hello World.`" printed out to the screen.

The `iostream` header contains routines that allow us to manipulate, well, I/O streams - such as `stdout`. The double-left-chevron operator (`<<`) is normally a left bitshift, but has been *overloaded* to allow data to be *shifted in* to the stream. You'll find a lot of operator overloading magic in C++ (which arguably can be confusing).

In any case, you are writing the string `"Hello, world."` into `cout` (which is `stdout`), and then writing in a new-line character (`endl`).

C++ supports namespaces (like Java packages), and the double colon operator (`::`) allows us to access definitions within that namespace – here, we're getting `cout` from the `std` namespace. We could also use a `using namespace` statement to bring the entire namespace into scope, but this might pollute the local namespace, and cause complications/confusion.

2.3 Classes and Structs

You may be familiar with structs from C and classes from Java, and in C++ structs and classes are the same thing – but with different visibility restrictions.

Recall that in Java you have `public`, `private`, `protected` – you have the same in C++, but instead of the method name being associated with the access specifier, a *label* is used to denote regions of declarations that should have that accessibility.

Here is a reasonable definition of a `square` class:

```
class square {
public:
  /* This is the constructor */
  square(float side_length) : side_length_(side_length) { }

```

```
  /* Returns the side length of the square. */
  float side_length() const {
    return side_length_;
  }

  /* Returns the area of the square. */
  float area() const {
    return side_length_ * side_length_;
  }

private:
  float side_length_;
};
```

This could also be declared as a struct, by simply replacing the `class` keyword with the `struct` keyword. If you are explicit about accessibility in your definition, there are no differences between classes and structs. However, if you do not specify `public`, `private`, or `protected` labels, then for classes, the default visibility is *private*, and for structs the default visibility is *public*.

Normally, as in Java, you'd write accessor and mutator methods (getters and setters) to deal with the internals of the class.

Activity: Try writing a `person` class, that uses a `std::string` to hold the first name, last name, and email address of a person. Write a test program that instantiates a few person objects.

2.3.1 Inheritance

Like in Java, C++ classes can inherit from (or subclass) other classes. C++ classes support multiple inheritance.

```
class shape {
public:
  virtual float area() const = 0;
};

class square : public shape {
public:
  square(float side_length) : side_length_(side_length) { }

  /* Returns the side length of the square. */
  float side_length() const {
    return side_length_;
  }

  /* Returns the area of the square. */
  float area() const override {
    return side_length_ * side_length_;
  }

private:
```

```
21   float side_length_;
22 };
```

Here, the `square` class inherits from the `shape` class. The `shape` class defines a virtual method called `area`, that does not have an implementation. This is effectively the same as an abstract method in Java, but is called a pure method. If a class contains at least one pure method, then it is an abstract class, and cannot be instantiated:

```
1 int main(int argc, const char *argv[])
2 {
3   shape s;
4
5   return 0;
6 }
```

```
$ g++ -o abstract abstract.cpp
abstract.cpp: In function 'int main(int, const char**)':
abstract.cpp:26:9: error: cannot declare variable 's' to be of abstract type 'shape'
   26 |   shape s;
      |         ^
abstract.cpp:1:7: note:   because the following virtual functions are pure within 'shape':
    1 | class shape {
      |       ^~~~~
abstract.cpp:3:17: note:     'virtual float shape::area() const'
    3 |   virtual float area() const = 0;
      |                 ^~~~
```

Activity: Have a go at writing a class hierarchy that represents animals, e.g. have a base `animal` class, and design a hierarchy that includes things like:

- Dog
- Cat
- Whale
- Bat
- Flying animals
- Mammals
- etc

2.3.2 Templates

Templates are a bit like generics in Java—parameters on types themselves. In C++, they're called templates because they can be primitive values as well as typenames.

You'll have experienced generics in Java when using collection classes, e.g. a list of strings would be strongly-typed as `List<String>`. In C++, templates can be used to specialise implementations of classes.

Activity: Practice this by writing a generic binary tree implementation. It does not need to be a balanced binary tree. There are plenty of online resources to drive inspiration from, but make sure you understand the concepts involved. I'd expect to see methods like "add", "remove", and "get". The implementation should be generic across key-value types.

```
1 template<typename K, typename V>
2 class tree { ... };
```

2.3.3 Online Tutorial

For the remainder of this section, visit https://www.w3schools.com/cpp/ and run through some of those tutorials, to make try out various things you'd like to practice.

3 StACSOS

In this section of the exercise class, you'll be getting used to **StACSOS**, making sure you can run and debug the operating system. Since for your coursework you'll be modifying the code directly, you should make sure you have a working environment.

3.1 Development

3.1.1 Download

To download the source-code, just clone the `git` repository:

```
$ git clone https://github.com/tspink/stacsos
```

3.1.2 Compile

To compile **StACSOS**, just run `make` in the top-level directory:

```
$ make
```

3.1.3 Run

To run **StACSOS** in the QEMU emulator, just run `make run` in the top-level directory:

```
$ make run
```

This has the added benefit of compiling **StACSOS** too. A QEMU window should appear showing the console, and the terminal should fill up with debugging messages.

If you want to include command-line arguments to the kernel, use the `kernel-args` setting on the `make` command-line:

```
$ make run kernel-args="pgalloc=buddy"
```

This will be **super important** for coursework development, where you need to turn your implementation on or off.

3.2 Developing in the Kernel

All core kernel code lives in the `kernel/` directory. In this part of the exercise, you're going to implement the CMOS Real Time Clock device driver, so that the system knows what the current date/time is.

There is already a skeleton device driver in the `kernel/dev/misc/` directory. Open up this file, and see where you have to write your code.

To make this device work, you need to implement the `read_timepoint()` method. Currently, it panics, but you need to send the correct signals to the real time clock, read the values, and fill in the `rtc_timepoint` structure.

Details of the CMOS RTC can be found here: https://wiki.osdev.org/CMOS#Reading_All_RTC_Time_and_Date_Registers.

The actual solution to this problem is around 20 lines of code. You need to:

1. Wait for CMOS updates to finish (see get_update_in_progress() function in example code).
2. Interrogate the RTC for values.
3. Manipulate those values if they're in binary coded decimal format.
4. Fill in the `rtc_timepoint` structure, and return it.

To interact with the CMOS registers, via the I/O address space (usually the `inb` and `outb` x86 instructions), there are helper routines in the `stacsos/kernel/arch/x86/pio.h` header. If you bring this header file into scope, you can use the `cmos_select` and `cmos_data` types to interrogate the CMOS registers.

```
#include <stacsos/kernel/arch/x86/pio.h>

using namespace stacsos::kernel::arch::x86;

...
ioports::cmos_select::write8(xxx);
u8 yyy = ioports::cmos_data::read8();
```

The only way to test this will be to tackle the following section.

3.3 Developing a user-space tool

All user space code lives in the `user/` directory. In this part of the exercise, you're going to write a small tool that prints out the current date and time, read from the RTC device.

The tool is going to be called `date`. To create the new tool, create a new directory hierarchy under the `user/` directory:

- `user/date`
- `user/date/src`
- `user/date/inc`

And modify the `user/Makefile` to include your tool. Modify the following line:

```
apps := init shell sched-test mandelbrot cat poweroff sched-test2
```

And add `date` to it:

```
apps := init shell sched-test mandelbrot cat poweroff sched-test2 date
```

Then, create `main.cpp` in the `user/date/src` directory. Create a main function:

```
#include <stacsos/console.h>

int main(const char *cmdline)
{
    stacsos::console::get().write("date program test\n");
    return 0;
}
```

And check that it compiles and runs:

```
$ make run
```

Then, in the **StACSOS** terminal:

```
> /usr/date
date program test
>
```

To read from the RTC device, you need to write code that:

1. Opens the RTC device file (`/dev/cmos-rtc0`) (hint: use `object::open`)
2. Reads the contents of the device file into a time-of-day (TOD) structure (hint: use the `pread` function on the open file).
3. Prints the contents of the TOD structure to the console (hint: use `console::get().writef()`).

The solution to this exercise should be around 10 lines of code. I'll get you started with the following structure definition (which is used for reading out of the device file):

```
1 struct tod {
2     u16 seconds, minutes, hours, dom, month, year;
3 };
```

Remember: ask for help if you need it! And, ask for more details if you're interested!