Operating Systems

Laboratory Exercises

Semester 2, 2024-25

Course Organizer
Dr. Antonio Barbalace

Teaching Assistant
Amir Noohi
Karim Manaouil
Alan Nair

THE UNIVERSITY *of* EDINBURGH
informatics

Contents

1 Laboratory Exercises

Welcome to an essential and exciting aspect of your journey in operating system coursework!

1.1 Overview

This laboratory provides a comprehensive introduction to the C programming language and its compilation tools. Through a series of carefully designed exercises, students will master:

- Basic C programming concepts
- Compilation stages and tools
- Variable scoping and memory management
- Control structures and program flow
- Multi-file program organization

Note

Before starting the exercises, ensure you have:

- GCC compiler installed (version 9.0 or higher recommended)
- A text editor or IDE of your choice
- Basic familiarity with terminal/command line interface

The next chapter provides background on how to set up a programming environment for this lab using the University of Edinburgh facilities.

2 Environment Setup

In this chapter, we explore the testing environment for your Linux kernel development labs. Whether you choose to work on physical lab machines or remotely via the University's infrastructure, the steps outlined here will help you set up your environment effectively.

2.1 DICE Infrastructure Overview

The University's **DICE** (Distributed Informatics Computing Environment) setup can be divided into three main categories of machines:

1. **Physical Lab Machines** – Located in Appleton Tower labs.
2. **Gateway Servers** – Provide remote entry points.
   - SSH gateway: `student.ssh.inf.ed.ac.uk`
   - RDP gateway: `s1234567.remote.inf.ed.ac.uk`
3. **Compute Machines** – High-powered servers in the University data centre (e.g., `student.compute.inf.ed.ac.uk`)

How to Connect

- If you are on campus, you can log in directly to physical lab machines in Appleton Tower.
- For off-campus access:
  - Connect to the University VPN.[1]
  - Use SSH or RDP gateways to reach the DICE environment.
- Once on a gateway, you may then connect to a compute machine if you need more resources.

Diagram of DICE Infrastructure

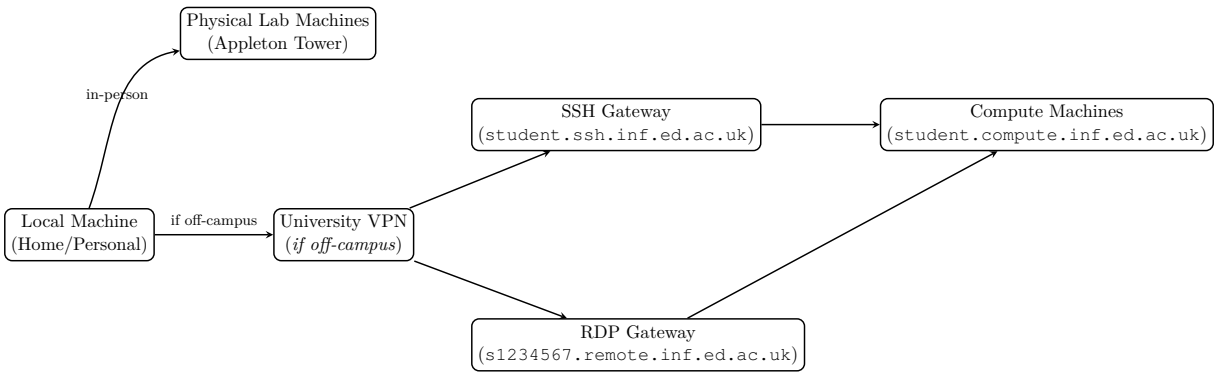

Figure 1: Overview of DICE Infrastructure

[1] https://computing.help.inf.ed.ac.uk/vpn

2.2 File Storage on DICE

DICE provides two main storage areas:

- **AFS (Andrew File System)**
  Available upon login to any DICE machine, but with limited quotas.[2]
  - Good for small files and lab work, such as a folder on your AFS Desktop.
- **`/disk/scratch`**
  Offers more space for large files or virtual machine images.[3]
  - Under `/disk/scratch`, there is a folder called `operating_systems`.
  - Create a subfolder named after your student ID (e.g., `s1234567`) for storing coursework.

2.3 Summary of Recommended Practices

- Use **physical lab machines** in Appleton Tower if you are on campus and want direct access.
- Off-campus users must connect via the **University VPN** before reaching the DICE environment.
- Use **SSH** (`student.ssh.inf.ed.ac.uk`) for command-line work, or **RDP** (`s1234567.remote.inf.ed.ac.uk`) for a full desktop session.[4]
- Move large files or VM images to **`/disk/scratch`** if your AFS quota is insufficient.
- Consult **Computing Support**[5] for any platform-specific configuration or troubleshooting.

[2] https://computing.help.inf.ed.ac.uk/afs-quotas
[3] https://computing.help.inf.ed.ac.uk/scratch-space
[4] https://computing.help.inf.ed.ac.uk/remote-desktop
[5] https://computing.help.inf.ed.ac.uk

EXERCISE

Exercise 1: Hello World Program

Let's begin with the fundamental "Hello World" program, exploring the compilation process step by step.

Step 1: Writing Your First C Program

Create a new file named `hello.c` with the following content:

```c
/* A simple Hello World program */
#include <stdio.h>

int main(int argc, char* argv[]) {
   printf("Hello World!\n");
   return 0;
}
```

Listing 1: hello.c

Step 2: Basic Compilation

Open a terminal in the same directory where hello.c reside. Let's use gcc to compile and execute the hello world program. In the console simply enter the following command:

```
> gcc hello.c
```

This command will generate a file named a.out, this is the executable, you can execute it by entering the following command line:

```
> ./a.out
```

The program will output the string "Hello World!"

Step 3: Preprocessor

In the same terminal used in the previous step, let's execute the following command:

```
> gcc -E hello.c > hello.i
```

This command will generate the file hello.i that contains the preprocessed C file. A lot of information as been added by the C preprocessor atop your program.
The curious student may take the time to investigate why this is happening either by searching on Google, or by using an LLM like ChatGPT.

Step 4: Object File

In the same terminal used in the previous step, let's execute the following command:

```
> gcc -c hello.c
```

This command will generate the object file hello.o that contains your compiled program, however, an object file cannot be executed yet. Being able to create object files is important because the majority of applications are created by lumping multiple objects files into an executable binary.
The curious student may take the time to investigate why object files cannot be executed by the operating system. This can be done by digging the information in GCC related books or online material, indeed LLM can also be used.

Step 5: Linking

In the same terminal used in the previous step, let's execute the following command:

```
> gcc -o hello hello.o
```

This command will generate the executable binary file hello. As an executable, this can be directly executed by the operating system – this is exactly like the a.out generated before. Note that instead of invoking gcc, we could also invoke the linker (ld). However, invoking the compiler driver (gcc) is far more easy, because it helps to resolve dependencies (i.e., mostly, what libraries are required).

Note

The `gcc` command performs preprocessing, compilation, and linking in one step. The `-o` flag specifies the output file name.

EXERCISE

Exercise 2: Working with Multiple Files

In this exercise the student will learn how to manually compile an application composed of multiple C language source files. For the sake of this exercise we assume the application is composed of two files.

Step 1: Source Files

Open your favorite editor (e.g., nano, vim, visual studio code, eclipse) and create two source files:

fun_a.c:

```c
#include <stdio.h>

int fun(int a, int b); // Function declaration

int main(int argc, char* argv[]) {
   int a = fun(123, 456);
   printf("Result: %d\n", a);
   return 0;
}
```

fun_b.c:

```c
int fun(int a, int b) {
   return a * b;
}
```

Step 2: Compilation

Open a terminal in the same directory where fun_a.c and fun_b.c reside. Let's use gcc to compile and execute this sample program. In the console enter the following commands, one after the other:

```
> gcc -c fun_a.c
> gcc -c fun_b.c
```

These commands will generate a file named fun_a.c and fun_b.c, respectively. These two files are not executable but they contain the executable code (as already explained above).

Step 3: Linking

In the same terminal used in the previous step, let's execute the following command:

```
> gcc -o fun fun_a.o fun_b.o
```

This command will create the executable binary file fun, which can be executed on the command line by

```
> ./fun
```

The curious student may take the time to experiment how to create a program that is built by more than two source files. In addition to that, it may also be interesting to explore how use the linker (ld) to blend multiple object files into a single executable binary. Also, try

```
> gcc -o fun_a.o
```

Is the executable binary created? Is there any compilation error? Why?

Note

Using separate compilation helps in:

- Managing large projects efficiently
- Reducing compilation time during development
- Organizing code into logical modules

EXERCISE

Exercise 3: Variable Scoping

This exercise demonstrates variable scoping in C, including global and local variables.

Step 1: Code

Create a file named `scope.c` with the following content:

```c
/* Demonstration of variable scoping */
#include <stdio.h>

int i = 0; // Global variable

int main(int argc, char* argv[]) {
   int i = 1; // Local to main
   {
      int i = 2; // Local to this block
      {
         int i = 3; // Local to innermost block
         printf("Innermost block i = %d\n", i);
      }
      printf("Inner block i = %d\n", i);
   }
   printf("Main function i = %d\n", i);
   return 0;
}
```

Listing 2: scope.c

Step 2: Compile and Run

Open a terminal in the same directory where scope.c reside. Let's use gcc to compile the program. In the console enter the following command:

```
> gcc -o scope scope.c
```

It is now possible to execute the program issuing the command

```
> ./scope
```

What the student concludes from this exercise? The same variable name can be used in different scopes, but the local scope "overrides" any outside definition – hence, you cannot access outside definition. Therefore, it is very important to make sure to reuse the same variable name only if it is really necessary.

Note

This program demonstrates:

- Variable shadowing
- Block-level scope
- The relationship between global and local variables

EXERCISE

Exercise 4: Data Types and Sizes

This exercise explores the sizes of various C data types and demonstrates the use of preprocessor macros.

Step 1: Code

Create a file named `sizes.c`:

```c
#include <stdio.h>

#define CHAR (sizeof(char))
#define SHORT (sizeof(short))
#define INT (sizeof(int))
#define LONG (sizeof(long))
#define LONGLONG (sizeof(long long))
#define FLOAT (sizeof(float))
#define DOUBLE (sizeof(double))
#define LONGDOUBLE (sizeof(long double))

int main(int argc, char* argv[]) {
   printf("Basic Types:\n");
   printf("char: %ld bytes\n", CHAR);
   printf("short: %ld bytes\n", SHORT);
   printf("int: %ld bytes\n", INT);
   printf("long: %ld bytes\n", LONG);
   printf("long long: %ld bytes\n", LONGLONG);
   printf("\nFloating Point Types:\n");
   printf("float: %ld bytes\n", FLOAT);
   printf("double: %ld bytes\n", DOUBLE);
   printf("long double: %ld bytes\n", LONGDOUBLE);
   return 0;
}
```

Listing 3: sizes.c

Step 2: Compile and Run

Open a terminal in the same directory where sizes.c reside. Let's use gcc to compile the program. In the console enter the following command:

```
> gcc -o sizes sizes.c
```

It is now possible to execute the program issuing the command

```
> ./sizes
```

What the student concludes from this exercise? Different variable types have different memory sizes, C let you query the memory size with the `sizeof(...)`.

Note

The actual sizes may vary depending on:

- Your system architecture (32-bit vs 64-bit)
- Your operating system
- Your compiler implementation

EXERCISE

Exercise 5: Type Casting

This exercise demonstrates type casting and potential data loss.

Step 1: Code

Create a file named `cast.c`:

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    char ch;
    short s;
    s = 321;
    ch = s; // Implicit casting from short to char
    printf("Original short: %hi\n", s);
    printf("After casting to char: %hhi\n", ch);
    printf("After casting back to short: %hi\n", (short)ch);
    return 0;
}
```

Listing 4: cast.c

Step 2: Compile and Run

Open a terminal in the same directory where cast.c reside. Let's use gcc to compile the program. In the console enter the following command:

```
> gcc -o cast cast.c
```

It is now possible to execute the program issuing the command

```
> ./cast
```

What the student concludes from this exercise? Because different variable types have different sizes the range of numbers that they can represent variates. Hence, when assigning a short to char – as in this example, the number can be truncated.

Note

Observe what happens when:

- A larger type is assigned to a smaller type
- Data loss occurs due to overflow
- Values are explicitly cast between types

EXERCISE

Exercise 6: Increment Operators

This exercise explores the prefix and postfix increment operators.

Step 1: Code

Create a file named `inc.c`:

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    int i = 42;
    int j;
    printf("Initial value - i: %d\n", i);

    j = (i++ + 10); // Postfix increment
    printf("After i++ - i: %d, j: %d\n", i, j);

    j = (++i + 10); // Prefix increment
    printf("After ++i - i: %d, j: %d\n", i, j);

    return 0;
}
```

Listing 5: inc.c

Step 2: Compile and Run

Open a terminal in the same directory where inc.c reside. Let's use gcc to compile the program. In the console enter the following command:

```
> gcc -o inc inc.c
```

It is now possible to execute the program issuing the command

```
> ./sizes
```

What the student concludes from this exercise? When using post- and pre-fix operators the programmer have to take great care about he/she wants to achieve. The result may be different from the expectation. Avoid using these operators if unsure.

Note

Key points to understand:

- Postfix increment (i++) uses the value before incrementing
- Prefix increment (++i) increments first, then uses the value
- The order of operations can affect the final result

EXERCISE

Exercise 7: Logical Operators

This exercise demonstrates logical operators and their behavior.

Step 1: Code

Create a file named `logic.c`:

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    int i = 42;
    int j = 1;
    printf("Initial values - i: %d, j: %d\n", i, j);

    j = !i; // Logical NOT
    printf("After !i - i: %d, j: %d\n", i, j);

    j = i && j; // Logical AND
    printf("After i && j - i: %d, j: %d\n", i, j);

    j = i || j; // Logical OR
    printf("After i || j - i: %d, j: %d\n", i, j);

    return 0;
}
```

Listing 6: logic.c

Step 2: Compile and Run

Open a terminal in the same directory where logic.c reside. Let's use gcc to compile the program. In the console enter the following command:

```
> gcc -o logic logic.c
```

It is now possible to execute the program issuing the command

```
> ./logic
```

What the student concludes from this exercise? Logical operators can be applied not only to bits but also to integers.

Note

Important concepts:

- Any non-zero value is considered true
- Zero is considered false
- Logical operators return 1 for true and 0 for false

EXERCISE

Exercise 8: Switch Statement

This exercise explores the switch statement and fall-through behavior.

Step 1: Code

Create a file named `switch.c`:

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
   int input = 11;
   int output = 0;

   switch(input) {
      case 1:
         output = 0;
         break;
      case 2:
      case 3:
         output = 1;
         break;
      default:
         output = -1;
   }
   printf("input: %d, output: %d\n", input, output);
   return 0;
}
```

Listing 7: switch.c

Step 2: Compile and Run

Open a terminal in the same directory where switch.c reside. Let's use gcc to compile the program. In the console enter the following command:

```
> gcc -o switch switch.c
```

It is now possible to execute the program issuing the command

```
> ./switch
```

What happen if the input variable is now set to 3? What if the last break is removed, what value the output is going to be?

Note

Key features of switch statements:

- Cases without break statements "fall through"
- The default case handles all unmatched values
- Only constant expressions are allowed in case labels

EXERCISE

Exercise 9: Loop Control

This exercise demonstrates loop control statements: continue and break.

Step 1: CodeLoop Control Statements

Create a file named `loop.c`:

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
   printf("Using continue:\n");
   for (int i = 1; i <= 5; i++) {
      if (i == 3)
         continue;
      printf("%d ", i);
   }
   printf("\n\nUsing break:\n");
   for (int i = 1; i <= 5; i++) {
      if (i == 3)
         break;
      printf("%d ", i);
   }
   printf("\n");
   return 0;
}
```

Listing 8: loop.c

Step 2: Compile and Run

Open a terminal in the same directory where loop.c reside. Let's use gcc to compile the program. In the console enter the following command:

```
> gcc -o loop loop.c
```

It is now possible to execute the program issuing the command

```
> ./loop
```

What the student concludes from this exercise? The continue statement let you skip the rest of the loop, but without exiting the loop. Instead, break exits the loop.

Note

Understanding loop control:

- continue skips the rest of the current iteration
- break exits the loop entirely
- Both can affect program flow significantly

EXERCISE

Exercise 10: Goto Statement

This exercise shows the usage of goto statements in C.

Step 1: CodeGoto Statement Example

Create a file named `goto.c`:

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
   int n = 0;

   if (n == 0)
      goto jump_target;

   printf("This will be skipped if n is 0\n");

jump_target:
   printf("Program continues here\n");
   return 0;
}
```

Listing 9: goto.c

Step 2: Compile and Run

Open a terminal in the same directory where `goto.c` reside. Let's use gcc to compile the program. In the console enter the following command:

```
> gcc -o goto goto.c
```

It is now possible to execute the program issuing the command

```
> ./goto
```

What the student concludes from this exercise? The goto statement let you jump around in the program – this is not common in other programming languages.

Note

Important considerations for goto:

- Generally discouraged in modern programming
- Can make code harder to understand and maintain
- Sometimes useful for error handling in C
- Should be used sparingly and with careful consideration