Operating Systems

Laboratory Exercises

Semester 2, 2024-25

Course Organizer
Dr. Antonio Barbalace

Teaching Assistant
Amir Noohi
Karim Manaouil
Alan Nair

Contents

1 Laboratory Exercises 2

Welcome to the second session of the Operating Systems Lab! In these exercises, you will:

- Practice core C programming
- Understand compilation and linking
- Explore variable scoping and memory management
- Experiment with control structures and function pointers
- Learn multi-file program organization

Note

Requirements to get started:

- A GCC compiler (version 9.0 or higher recommended)
- A text editor or IDE (vim, nano, VSCode, Eclipse, etc.)
- Basic familiarity with the terminal/command line

The next section describes how to set up the recommended environment via the University of Edinburgh's DICE infrastructure.

2 Environment Setup

Whether you work on physical lab machines or remotely, follow these steps to set up your environment for both C programming and kernel labs.

2.1 DICE Infrastructure Overview

The University's **DICE** system includes:

1. **Physical Lab Machines** in Appleton Tower
2. **Gateway Servers** for remote access:
   - SSH Gateway: `student.ssh.inf.ed.ac.uk`
   - RDP Gateway: `s1234567.remote.inf.ed.ac.uk`
3. **Compute Machines** in the University data centre: `student.compute.inf.ed.ac.uk`

Connection Methods:

- **On Campus:** Log in directly to Appleton Tower lab machines.
- **Off Campus:**
  - Use the University VPN.[1]
  - Connect via SSH or RDP gateways.
- **High Resource Needs:** From a gateway, SSH or RDP into a compute machine.

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

Diagram of DICE Infrastructure

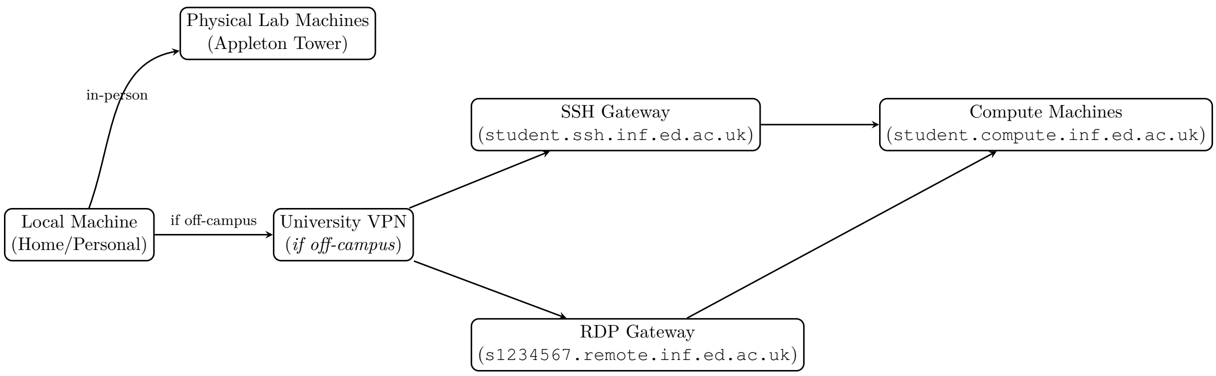

Figure 1: Overview of DICE Infrastructure

2.2 File Storage on DICE

- **AFS (Andrew File System)** Automatically available on all DICE machines but has limited quota.[2]
- **`/disk/scratch`** Larger space for bulky files or VMs.[3] Create your subdirectory under `operating_systems`.

2.3 Recommended Practices

- Use the **physical lab machines** in Appleton Tower if you are on campus.
- If off campus, **VPN + SSH/RDP** into DICE.
- Store large files on `/disk/scratch`.
- Consult **Computing Support**[4] for assistance.

[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

EXERCISE

Exercise 1: Struct vs Union

Goal

Explore the memory layout and behavior of `struct` vs. `union` in C.

Step 1: Create the Source File

Create `fract.c`:

```c
#include <stdio.h>

struct fraction {
  int num;
  int denom;
}; // don't forget the semicolon!

union ufraction {
  int num;
  int denom;
}; // union shares memory among members

int main() {
  struct fraction f1, f2;
  union ufraction uf1, uf2;

  printf("struct fraction size: %ld, union ufraction size: %ld\n",
         sizeof(f1), sizeof(uf1));

  f1.num = 22;
  f1.denom = 7;

  uf1.num = 22;
  uf1.denom = 7; // overwrites uf1.num

  printf("f1 = %d / %d; uf1 = %d / %d\n",
         f1.num, f1.denom, uf1.num, uf1.denom);

  f2 = f1;
  uf2 = uf1;

  printf("f2 = %d / %d; uf2 = %d / %d\n",
         f2.num, f2.denom, uf2.num, uf2.denom);

  return 0;
}
```

Listing 1: fract.c

Step 2: Compile and Run

Open a terminal where `fract.c` is located and run:

```
gcc -o fract fract.c
./fract
```

Solution Explanation

- **Struct Layout:** Each member (`num` and `denom`) has its own memory. Hence `sizeof(struct fraction)` is typically `8` bytes on a 64-bit system (4 bytes for each int).
- **Union Layout:** All members share the same memory region, so `sizeof(union ufraction)` matches the size of its largest member (4 bytes on many systems).
- **Assignment Effects:** Writing to `uf1.denom` overwrites the same memory used by `uf1.num`, revealing union's unique "shared memory" property.

Want to Know More?

Search for "common uses of union in C" or "union pitfalls" to learn best practices and potential dangers.

EXERCISE

Exercise 2: Arrays and Out-of-Bounds Access

Goal

Review array creation and illustrate the dangers of out-of-bounds access in C.

Step 1: Create the Source File

outbound.c:

```c
#include <stdio.h>
#define ELEMENTS 16

int main() {
  int idx;
  int numbers[ELEMENTS];

  // Populate the array
  for (int i = 0; i < ELEMENTS; i++) {
    numbers[i] = i;
  }

  // Access in-bound
  idx = 1;
  printf("numbers[%d] = %d\n", idx, numbers[idx]);

  // Access out-of-bounds
  idx = ELEMENTS + 4;
  printf("numbers[%d] = %d\n", idx, numbers[idx]);

  return 0;
}
```

Listing 2: outbound.c

Step 2: Compile and Run

```
gcc -o outbound outbound.c
./outbound
```

Solution Explanation

- **In-Bounds Access:** `numbers[1]` is valid, so it correctly prints `1`.
- **Out-of-Bounds Access:** `numbers[ELEMENTS + 4]` is undefined behavior. It may print some garbage value, crash, or appear "fine" but potentially overwrite nearby memory.

Want to Know More?

Look up "undefined behavior in C array indexing" to see real-world bugs and security vulnerabilities caused by out-of-bounds reads/writes.

EXERCISE

Exercise 3: Pointers and Pointer Dereferencing

Goal

Practice using pointers (`&` for address-of, `*` for dereference).

Step 1: Create the Source File

`ptr.c`:

```c
#include <stdio.h>

int main(){
  int *p, x;
  p = &x;  // p points to x
  *p = 3;  // write 3 into x

  printf("&p = %p - &x = %p\n", (void*)&p, (void*)&x);
  printf("p = %p - x = %d\n", (void*)p, x);
  printf("*p = %d\n", *p);

  // Uncomment below to see compiler warnings or odd behavior:
  // printf("*p = %d - *x = %d\n", *p, *x);
  // printf("*p = %d - *x = %d\n", *p, *(int*)(unsigned long)x);

  return 0;
}
```

Listing 3: ptr.c

Step 2: Compile and Run

```
gcc -o ptr ptr.c
./ptr
```

Solution Explanation

- **Pointer Setup:** `p = &x` means `p` holds the address of `x`.
- **Dereference:** `*p` assigns a value to the memory pointed to by `p`, effectively modifying `x`.
- **Printing Addresses:** Casting to `(void*)` is common to avoid warnings when printing pointer addresses with `%p`.

Want to Know More?

Search for “pointer arithmetic in C” and “memory layout of local variables” for a deeper dive into pointer mechanics.

EXERCISE

Exercise 4: Complex Data Types

Goal

Define a `struct` for complex numbers and calculate a sum.

Step 1: Create the Source File

`cmplx.c`:

```c
#include <stdio.h>

struct complex {
  double real;
  double imag;
};
typedef struct complex Complex;

#define ELEMENTS 4

int main(){
  Complex cmplx[ELEMENTS];
  Complex sum = {0.0, 0.0};

  // Initialize each complex number
  for (int i = 0; i < ELEMENTS; i++){
    cmplx[i].real = i;
    cmplx[i].imag = i;
  }

  // Compute the sum
  for (int i = 0; i < ELEMENTS; i++){
    sum.real += cmplx[i].real;
    sum.imag += cmplx[i].imag;
  }

  printf("Sum = %lf + %lfi\n", sum.real, sum.imag);
  return 0;
}
```

Listing 4: cmplx.c

Step 2: Compile and Run

```
gcc -o cmplx cmplx.c
./cmplx
```

Solution Explanation

- **Struct Definition:** `struct complex` with two `double` members is straightforward to handle.
- **Initialization Loop:** We assign each array element so that the real part and the imaginary part are the same (just `i`).

- **Sum Accumulation:** Summing each element's `real` and `imag` individually yields a final `sum`.

Want to Know More?

Explore "typedef in C" vs. using `struct` directly, and investigate complex math libraries in C for advanced operations.

EXERCISE

Exercise 5: More Pointers: Reasoning

Goal

Predict pointer-based changes before compiling.

Step 1: Code Snippet to Analyze

```c
int a, b, *p, *q;
a = 10;
b = 20;
p = &a;
q = &b;

*q = a + b; // line 6
a = a + *q; // line 7
q = p;     // line 8
*q = a + b; // line 9

printf("a=%d b=%d *p=%d *q=%d\n", a, b, *p, *q);
```

Listing 5: pointer reasoning

Questions:

- Where is `a + b` stored at line 6?
- How does line 7 alter `a`?
- What happens after line 8, when `q = p`? How does line 9 then affect `a`?

Step 2: Compile, Run, and Compare

Wrap the snippet in a `main()` function, compile, and see if your predictions match.

Solution Explanation

- **Line 6:** `*q = a + b` effectively does `b = 10 + 20 = 30`.
- **Line 7:** `a = a + *q` means `a = 10 + 30 = 40`.
- **Line 8:** `q = p` sets `q` to point to `a`.
- **Line 9:** `*q = a + b` is now `a = 40 + 30 = 70`.

Hence the final output should reflect `a = 70`, `b = 30`.

Want to Know More?

Look for “debugging pointers in C with gdb” or “visualizing pointer operations in memory.”

EXERCISE

Exercise 6: Functions: Passing by Value vs. Pointer

Goal

Demonstrate how C always passes arguments by value, but pointers can modify the caller's variables.

Step 1: Passing by Value

`swap.c`:

```
#include <stdio.h>

void Swap(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}

int main(){
  int a = 1;
  int b = 2;
  Swap(a, b);
  printf("After Swap(a, b): a = %d, b = %d\n", a, b);
  return 0;
}
```

Listing 6: swap.c

Step 2: Compile and Run

```
gcc -o swap swap.c
./swap
```

Solution Explanation (Passing by Value)

- **Parameter Copy:** `x` and `y` receive copies of `a` and `b`, not the originals.
- **No Lasting Effect:** `a` and `b` in `main()` remain unchanged after `Swap(a, b)`.

Step 3: Using Pointers

`swapPtr.c`:

```
#include <stdio.h>

void Swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

int main(){
  int a = 1;
```

```c
    int b = 2;
    Swap(&a, &b);
    printf("After Swap(&a, &b): a = %d, b = %d\n", a, b);
    return 0;
}
```

Listing 7: swapPtr.c

Solution Explanation (Pointers)

- **Direct Access:** `Swap(&a, &b)` means `*x` and `*y` refer to the real variables `a` and `b` in `main()`.
- **Effect on Callers:** Modifying `*x` and `*y` changes `a` and `b` directly.

Want to Know More?

Search "pointer parameters in C" or "emulating pass-by-reference in C" for advanced usage patterns.

EXERCISE

Exercise 7: Function Pointers

Goal

Use function pointers to dynamically select which function to call at runtime.

Step 1: Create the Source File

`funPtr.c`:

```c
#include <stdio.h>

int max(int a, int b) { return (a > b) ? a : b; }
int min(int a, int b) { return (a < b) ? a : b; }

// Global function pointer
int (*func_ptr)(int, int) = 0;

int main() {
  int x = 45, y = 87, z = 103;
  int value;

  if (z <= 128)
    func_ptr = max;
  else
    func_ptr = min;

  value = func_ptr(x, y);
  printf("z = %d, value = %d\n", z, value);
  return 0;
}
```

Listing 8: funPtr.c

Step 2: Compile and Run

```
gcc -o funPtr funPtr.c
./funPtr
```

Solution Explanation

- **Runtime Function Choice:** By setting `func_ptr` to either `max` or `min`, we decide at runtime which function to invoke.
- **Callbacks in C:** Function pointers are fundamental to event-driven architectures, jump tables, and plugin systems.

Want to Know More?

Look up "function pointers in C usage" or "callback patterns in C" for scenarios like sorting with custom comparators.

EXERCISE

Exercise 8: `printf` Tips and Tricks

Goal

Learn various formatting options for `printf`.

Step 1: Create the Source File

`printf.c`:

```c
#include <stdio.h>

int main(){
  printf("Hello %d World %d!\n", 11, 13+4);
  printf("Hello %s World %c%d!\n", "great", 'A', 3);
  printf("number: %4u\n", 15); // width without leading zeros
  printf("number: %04u\n", 15); // width with leading zeros
  printf("number: %.3f\n", 0.123456789f); // float precision
  return 0;
}
```

Listing 9: printf.c

Step 2: Compile and Run

```
gcc -o printf printf.c
./printf
```

Solution Explanation

- **%d, %s, %c, %f:** Common specifiers for printing integers, strings, characters, and floats.
- **Width/Precision:** `%4u` means at least 4 characters wide; `%04u` pads with zeros. `%.3f` prints 3 digits after the decimal point.

Want to Know More?

Look up “printf format specifiers cheat sheet” for advanced conversions, including `%x`, `%p`, and length modifiers.

EXERCISE

Exercise 9: Character Counting from Slides

Goal

Count occurrences of specific characters using a dedicated function.

Task

Write a program to count how many times `'a'`, `'b'`, and `'e'` appear in this paragraph:

> "The overwhelming majority of program bugs and computer crashes stem from problems of memory access, allocation, or deallocation. Such memory-related errors are also notoriously difficult to debug."

Hint:

```
int countChars(const char *str, char c) {
    // returns the number of times 'c' occurs in 'str'
}
```

Then call it for each character you want to count.

Solution Explanation

- **Loop Over String:** Typically, you iterate until you reach a null terminator (`'\0'`).
- **Case Sensitivity:** Remember that `'a'` differs from `'A'`, so decide if you need to handle both or just the lowercase version.

Want to Know More?

Try "implementing custom string functions in C" or "comparison of standard library vs. custom string routines."

EXERCISE

Exercise 10: Basic Input/Output with `scanf` and `printf`

Goal

Demonstrate reading from standard input and writing to standard output.

Step 1: Create the Source File

`inout.c`:

```c
#include <stdio.h>

int main(void) {
  int x;
  printf("Enter a number: ");
  scanf("%d", &x);
  printf("You entered: %d\n", x);
  return 0;
}
```

Listing 10: inout.c

Step 2: Compile and Run

```
gcc -o inout inout.c
./inout
```

Solution Explanation

- **Basic I/O:** `scanf(" %d", &x)` reads an integer from the user, storing it in `x`.
- **Format String Dangers:** If you mismatch the specifier (e.g., use `%f` instead of `%d`), you get undefined behavior or runtime errors.

Want to Know More?

Look up “scanf pitfalls” or “C input validation best practices” to see how to handle invalid user inputs gracefully.