FIT2100 Tutorial #3
I/O Management
&
Disk Scheduling
Week 5 Semester 2 2021

August 16, 2021

Revision Status:

```
$Id: FIT2100-Tutorial-06.tex, Version 2.0 2018/10/02 13:45 Jojo $
Updated by Dr. Charith Jayasekara, Aug 2021.
```

Contents

1 Background

In this tutorial, you will be provided with the opportunity to explore further on the various concepts of memory and I/O management as discussed in the lectures. There will also be a chance to reinforce some C programming concepts during the tutorial session itself.

You should complete the suggested reading in Section 2, also read through Section 3 and complete the pre-class exercise which relates to C programming before attending the tutorial.

Before attending the tutorial, you should:

- Complete your pre-class readings (Sections 2 and 3)
- Complete your pre-class quiz
- Prepare a list of questions on C programming concepts that you would like to be clarified during the tutorial
- Attempt the review questions and problem-solving tasks (Section 4)

2 Pre-tutorial Reading

You should complete the following two sets of readings:

- Lecture Notes: Week 3 and 4
- Tanenbaums' textbook: Chapter 5

3 Clarification of C programming concepts

Now is a good time to look back through the reading notes in the first two tutorials, and nominate difficult topics for group discussion during the first hour of this week's tutorial. Refer back to the first two tutorials and take note of any concepts you need to reinforce your understanding on.

3.1 Variable scope and lifetime in C

Variables in C are usually *local* if they are declared inside a function, and *global* if they are defined outside functions. Computer memory is split into several segments such as the *stack*, the *heap*, and the *data* segment. The *heap*[1] is an area of the program's memory where data may be dynamically allocated to, and remain here until they are de-allocated. The *data* segment is an area of the program's memory where data may remain for the whole lifetime of the program.

Local variables (and arguments passed to a function), are commonly known as *stack variables*. A running program has a stack in memory that keeps track of the current state of the program's execution including where functions have been called from, so that the program can eventually return back to the `main` function after all function calls have returned. Local variables and arguments are also stored on this stack, and are removed from the stack when the function returns. (It is pointless to return a pointer to a stack variable, from the function it was declared in. The variable simply won't exist in memory anymore once the function returns.)

Global variables (defined outside functions) are stored in the *data* segment. The `static` keyword is useful for declaring a 'local' variable in the *data* segment instead of the stack. In the following example, we create a variable named `something` which acts partly like a local variable and partly like a global variable at the same time.

```c
/* static_keyword.c */

/**
 * Even if this function is called many times, the first initialisation
 * of something = 41 is only done the first time the function is called.
 **/
void some_function() {
  static int something = 41; // only initialised once
  something++;

  printf("Value of something is: %d\n", something);
}
```

In the code above, the variable `something` is still a locally-scoped variable (in that you cannot directly access the variable named `something` outside the function it was declared in), but it is allocated to the *data* segment rather than the stack, so it will continue to exist (and it will keep its value) after the function has returned.

[1] The heap also allows for allocation of very large blocks of memory. Very large variables, e.g. arrays containing millions of elements, must be allocated in the heap. There is not enough room in the stack for very large amounts of data and you could get a 'stack overflow'. You might recall from the previous tutorial that dynamic memory allocation on the heap can be done via `malloc()`.

The second time you call the function, the initialisation will not be repeated (it was already initialised the first time); so rather than being re-initialised to 41, it will keep the same value as it had last time the function was called. The lifetime of this variable is the same as the lifetime of the running program itself.

If you have a pointer to the memory address of this variable, you can access its value from any part of your program via its memory address, since it remains in the *data* segment until the program exits. In this respect, it behaves like a global variable.

Try using the keyword `static` yourself:

Download the code for the following program, `factorial-series.c`, then compile and run it. The code has a problem which may cause a segmentation fault. Answer the following questions:

(a) Before compiling this code, try to identify the cause of the problem.

(b) The problem can be fixed by inserting the keyword `static` into the code in one place. Where should this keyword be inserted?

Something to think about

Even after fixing the `factorial-series.c` code in part (b), the function is not ideal. For example, what happens if we call the function twice with two different `n` values and then try to compare the two arrays that have been returned? (What happens to the returned array from the first call, when we call the function twice?) What happens to the result if `n > SERIES_SIZE` ?

How could this function be rewritten to allow it to return a result in a different array each time it is called? (If you are stuck, read the spoilers at the end of this document.)

3.2 Pre-class Quiz Checkpoint (2 marks)

Now that you have completed reading this section, please attempt the pre-class quiz for Tutorial 3 on Moodle before you attend the class.

The pre-class quiz is worth 2 out of 10 raw marks for the in-class assessment of each module. The tasks that follow are worth 8 out of 10 raw marks. In total, this class's assessment tasks are worth 1% of your total final mark for the unit.

4 I/O Management and Disk Scheduling

4.1 Review Questions

After finishing the pre-tutorial reading, please try to solve the following review questions:

Question 1

Define three techniques for performing I/O.

Question 2

What is the difference between *block-oriented* devices and *character-oriented* devices?

Question 3

Discuss how could the system performance be improved by using *double buffering* rather than a *single buffer* for I/O operations?

Question 4

What *delay elements* are involved in a disk read or write?

Question 5

Compare the difference between the following disk scheduling algorithms: **FCFS**, **SSF**, **Elevator**, and **C-SCAN**.

Question 6

What is the difference between *internal* and *external* fragmentation in terms of allocation of blocks[2] on disk?

[2]blocks are sometimes called clusters.

4.2 Problem-Solving Tasks (8 marks)

4.2.1 Task 1 (4 marks)

Imagine a program that works with data files, where each file consists of fixed-size data records of 128 bytes each.

A total of 2,500,000 such records needs to be stored on disk. Assuming the disk has 4096 bytes per sector, 63 sectors per track, 90 tracks per surface, and 4 usable surfaces, how many of the following will be required to store all records?

(a) sectors

(b) tracks

(c) surfaces

(d) disks

4.2.2 Task 2 (4 marks)

Assume that the disk head is initially positioned on track 89 and is moving in the direction of decreasing track number. For the following sequence of disk track requests:

125, 112, 15, 190, 137, 56, 12, 89, 38, 164, 133

(a) Describe or trace the order in which these requests are served based on the following four disk scheduling algorithms:

  (i) FCFS

  (ii) SSF

  (iii) Elevator

  (iv) C-SCAN

(b) Calculate the average seek length (in terms of the number of tracks traversed) for each of the disk scheduling algorithms.

(c) Discuss your attempts for (a) and (b) with your tutor.

5 Spoilers!

Answers to the 'Something to think about' question

Ideally, the size of the array returned by `factorial_series` should not be fixed, it should be the same size as the argument `n`. It should also be possible to place the result in a different array each time, rather than replacing what is in the same static array each time the function is called. Here are two possible ways to achieve that:

We could rewrite the `factorial_series` function to take an additional argument where the caller must also provide an array to place the result in (passed as an argument). This is similar to how the `scanf` function works when used with the `%s` specifier.

Another way would be to call `malloc(n * sizeof(int));` inside the `factorial_series` function to manually allocate the correct amount of memory for the result array (array of `n` `ints`), and return a pointer to that array at the end of the function. However, using `malloc` creates an additional requirement: the caller would need to call the `free()` function on the array when they are finished with the result, since manually-allocated memory must always be freed from the heap when it is no longer needed.