FIT2100 Tutorial #3
I/O Management, and Disk Scheduling
Week 5 Semester 2 2019

August 21, 2019

Revision Status:

`$Id: FIT2100-Tutorial-06.tex, Version 2.0 2018/10/02 13:45 Jojo $`

`Adapted to tutorial 3, 2019 by Daniel Kos, Aug 2019`

Acknowledgement

Some exercises presented in this tutorial were adapted from: William Stallings (2017). *Operating Systems: Internals and Design Principles*, Pearson.

Contents

1 Background

This tutorial provides students with the opportunity to explore further on the various concepts of memory and I/O managements 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.

2 Pre-tutorial Reading

You should complete the following two sets of reading:

- Lecture Notes: Weeks 3 and 4
- Stallings' textbook: Chapter 11

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.

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 also known as *heap variables*. The *heap* is an area of the program's memory where data may remain for the whole lifetime of the program.[1]

The `static` keyword is useful for declaring a 'local' variable in the heap 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.

```
1 void some_function() {
2 //even if this function is called many times, the first initialisation
3 //of something = 5 is only done the first time the function is called.
4
5     static int something = 5; //only initialised once
6     something++;
7 }
```

In the above example, 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 heap rather than the stack, so it will continue to exist (and it will keep its value) after the function has returned. 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 5, 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 heap until the program exits. In this respect, it behaves like a global variable.

3.1.1 Pre-class exercise (2 marks)

Consider the following code file (`factorial-series.c`). The code has a problem which may cause a segmentation fault.

(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?

[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.'

```c
//This program has a problem...

#include <stdio.h>

#define SERIES_SIZE 100
//the number of elements to be returned by factorial_series()

int* factorial_series(int);

int main(int argc, char* argv[]) {
    int* fact_series;
    int factorial_result = 1;
    int i;

    fact_series = factorial_series(5);
    //get the array returned by factorial_series

    for(i=0; i<SERIES_SIZE; i++) factorial_result *= fact_series[i];

    printf("5! is %d\n", factorial_result);

    return 0;
}

int* factorial_series(int n) {
//This function returns a pointer to an array containing numbers that can
//be multiplied to produce the factorial series n * (n-1) * (n-2) * ... * 1
//The size of the array is defined in the macro SERIES_SIZE
//For example: factorial_series(4) will return an array containing
//elements {4, 3, 2, 1, 1, 1, ...}.

    int series[SERIES_SIZE];

    int index = 0;
    for(index = 0; index<SERIES_SIZE; index++) {
        series[index] = n;

        if(n>1) n--;
        //once n is already 1, just fill each remaining element with a 1
        //since this will not affect the result of the multiplication.
    }

    return series;
}
```

Something to think about

Even after fixing the code example above, 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.)

4 I/O Management and Disk Scheduling

4.1 Review Questions

Question 1

Define three techniques for performing I/O.

Question 2

What is the difference between *block-oriented* devices and *stream-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

Define the following disk scheduling algorithms: **FIFO**, **SSTF**, **SCAN**, and **C-SCAN**.

Question 6

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

[2] blocks are sometimes called clusters

4.2 Problem-Solving Tasks

4.2.1 Task 1

Assume that the disk head is initially positioned on track 100 and is moving in the direction of decreasing track number. For the following sequence of disk track requests:

27, 129, 110, 186, 147, 41, 10, 64, 120

(a) Describe or trace the order in which these requests are served based on the four disk scheduling algorithms: (i) FIFO, (ii) SSTF, (iii) SCAN, and (iv) C-SCAN.

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

4.2.2 Task 2 (3 marks)

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

Calculate how much disk space (in sectors, tracks, and surfaces) will be required to store such a data file containing 300,000 records if the disk is has 512 bytes per sector, 96 sectors per track, 110 tracks per surface, and 8 usable surfaces.

5 Spoilers!

Answers to 'Something to think about' questions

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` `int`s), 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.