Tuesday 6 February 2024
09:30-11:00 GMT
Duration: 1 hour 30 minutes

DEGREE of BSc Software Engineering (Graduate Apprenticeship)

SYSTEMS PROGRAMMING
COMPSCI 2030

(Answer all 4 questions)

This examination paper is worth a total of 60 marks.

1. Pointers and Abstract Data Types

(a) The following variables are all pointers to the same data type: `start, end, middle`
`start` and `end` point to the first and last elements in an array, respectively.
Now consider the statement: `middle = (start + end) / 2;`

(i) Explain why this statement is invalid in 10 words or fewer. [3]

Solution:
Pointers cannot be added to each other. **OR**
Pointers could only be added to integer values.

(ii) Write an alternative single statement of code that achieves the same objective of pointing to the element in the middle of the array. Assume that the array has an even number of elements. Also, explain why this would work; i.e. how the new code avoids the exact problem of the previous code. [3]

Solution:
`middle = start + (end - start) / 2;` **[2]**
The value of `(end-start)/2` is an integer, not a pointer, so it can legally be added to `start`. **[1]**

(b) Consider the following declaration:

```c
struct node {
    char val;
    int min, max;
};

struct {
    struct node n;
    int count;
} v, *w = &v;
```

(i) The following statement is invalid. Explain why.
`printf("%d", w->n->min);` [2]

Solution:
You cannot access member min with the arrow notation (`->`)

(ii) Re-write the previous statement correctly. [1]

Solution:
`printf("%d", w->n.min);` **[1] OR**
Indicate to use the dot notation (`.`) but without code **[0.5]**

(iii) The following statement is also invalid. Explain why.
`w->max = 999;` [1]

Solution:
There is no member `max` of `w` **OR** You need to access `n` first

(iv) Re-write the previous statement correctly. [1]

Solution:
`w->n.max;`

(c) The following code segment has a bug.

```
int* initaliseInteger(int val) {
    int x = val;
    int* ptr = &x;
    return ptr;
}
```

(i) Name the issue and describe why it occurs (in 30 words or fewer). [2]

Solution:
Dangling pointer **[1]**
The function returns a pointer to a local variable that would expire outside the scope of the `initaliseInteger` function, where it cannot be dereferenced. **[1]**

(ii) Will this bug be discovered at compile time (without any special compile flags) or at run time? Explain why. [2]

Solution:
Run time **[1]**
There is nothing wrong with the syntax **[1]**

2. Memory Management

(a) Using the following allocation: `foo *p = malloc(sizeof(foo));` ,
Where is `p` stored? Where is the value it points to stored?
1 word per question would be sufficient. [2]

Solution:
`p`: Stack **[1]**
`*p`: Heap **[1]**

(b) The following code segment attempts to implement a dynamic array using `malloc()`. A dynamic array is one that can be resized as needed during runtime. Although the code compiles without any errors, it does not achieve what is intended.

```
 1  #include <stdlib.h>
 2
 3  int* allocateArray(int arr[], int len) {
 4      if (arr) {
 5          free(arr);
 6          arr = NULL;
 7      }
 8      arr = (int *)malloc(len * sizeof(int));
 9      return arr;
10  }
11
12  int main() {
13      int size = 8;
14      int *array = allocateArray(NULL, size);
15
16      // store some values in the array
17      for (int i = 0; i < size; i++) {
18          array[i] = i*i;
19      }
20
21      // attempt to change the array size
22      size = 10;
23      array = allocateArray(array, size);
24
25      free(array);
26      array = NULL;
27  }
```

There a few issues with this code. Please read each part **carefully** and answer as instructed.

(i) Executing this code will throw a runtime error. Identify the type of error, the line number that is responsible, and explain why it happens (in 20 words or fewer). [6]

Solution:
Double free **[2]**
Line 25 **[2]**

`array` has already been freed inside `allocateArray()`. **[2]**

(ii) If the above runtime error is fixed, would the array keep the values stored in it before resizing (see lines 16–19)? Explain (in 30 words or fewer). [3]

Solution:
No **[1]**
The space for `array` has been free'd / the original values are no longer accessible **[1]**
`array` now points to a new block on the heap **[1]**

(iii) Again, ignoring the runtime error discussed in (i), suggest how you would modify the code in order to keep the original values stored in the array even after resizing. Please write code and indicate using line numbers where it would be placed and/or if would replace any existing lines. [4]

Solution:
`arr = (int *)realloc(arr, len * sizeof(int));` **[2]**
to replace lines 4–8 **[2]**

3. Concurrent Systems Programming

(a) Why would you use multiple *threads* to implement an application? Name two reasons and explain each briefly. A sentence or two should suffice for explaining each. [4]

Solution: Valid answers, for instance:

- Responsiveness: Allowing multiple threads to be in progress at the same time, so they all can react to new inputs/data, even with just one CPU core
- Speeding up programs: Running parts of programs (tasks) in parallel, so that the overall program finishes faster
- Scaling: Handling more and more load (data/requests/users) with more and more resources, without necessarily a speed-up
- Convenience: It might be easier to construct some applications as a collection of threads, e.g. using threads for different components of a program

[4] : 1 mark for a correct name (up to [2]), 1 mark for a sensible explanation (up to [2])

(b) A programmer has written the following C function in a program that sells tickets:

```c
1 int tickets_sold = 0;
2
3 int sell_a_ticket() {
4     int ticket_id = tickets_sold;
5     tickets_sold = tickets_sold + 1;
6     return ticket_id;
7 }
```

When the `sell_a_ticket` function is called, the program should return a unique identifier for each ticket sold as well as increment the number of tickets sold.

This works well until the programmer creates multiple threads that serve multiple customers and concurrently call the function.

Answer the following two questions briefly:

(i) Referencing specific lines of code, what can happen when multiple threads execute the `sell_a_ticket` function concurrently? [2]

Solution: Any valid scenario, such as:

- Two threads each locally store the previous count (Line 4) before they each increment it and return the `ticket_ID` (Line 5 and 6). While the count of sold tickets might be right, both threads will return the same `ticket_ID`, so it is not unique anymore.
- It is possible that two or more threads concurrently execute Line 5, which is not necessarily atomic, leading to a wrong count of sold tickets.

Marking:

- valid scenario **[1]**
- sensible referencing of lines of code **[1]**

(ii) Which line or lines of code are problematic here and why? [2]

Solution: Lines 4 - 5 **[1]** are a critical section that is not protected **[1]**

(c) Briefly explain the purpose of a mutex and name the two `pthread` library functions that are used for this purpose. [3]

Solution: A mutex provides Mutual Exclusion **[1]**; it ensures only one thread/thread has access to a shared resource / executes a critical section **[1]**

The pthread library functions to acquire and release mutex locks are `pthread_mutex_lock()` and `pthread_mutex_unlock()` **[1]**

(d) The following multiple-choice questions are marked negatively: For each correct answer you gain two marks and for each wrong answer you lose one mark.

There is one correct answer per question.

(i) Which of the following statements is true about Green Threads (such as available in Python)?

1. Green Threads speed up programs that run on parallel hardware.
2. Green Threads run without resources provided by operating systems.
3. Green Threads can be provided by libraries.

[2]

Solution: 3 (or C) **[2]**

(ii) According to Amdahl's Law, which of the following factors primarily limits the possible speed-up of a program?

1. The workload available for the parallel part of program.
2. The proportion of the program that can run in parallel.
3. The speed of the fastest processor in the system.

[2]

Solution: 2 (or B) **[2]**

4. Memory Management for Multiple Processes

(a) The address space of a process is used for its code, static data, heap, and stack memory. Answer the following two questions briefly:

(i) Which parts of the address space of a process are accessible to all its threads? [1]

Solution: The entire address space of a process is accessible to its threads / all four parts listed above can be accessed by the threads of a process **[1]**

(ii) Are stack and heap memory specific to a thread, a process, or both? [2]

Solution: Stack memory is managed for each individual thread **[1]**, heap memory is managed for an entire process**[1]**

(b) A single process uses 10GB of memory on a computer system with 8GB of RAM. Explain briefly how this is possible. One or two sentences should suffice to explain this. [2]

Solution: This becomes possible with "virtual memory" / memory virtualisation / memory address translation **[1]**, which allows to transparently address memory pages that currently reside on disks **[1]**

(c) Operating systems maintain information about running processes in a data structure called Process Table. This table is held in memory, but entries in the table (also known as Process Control Blocks) contain registers. Explain briefly why. [2]

Solution: When a process looses CPU access (e.g. because it waits for I/O, its time slice is up, or it is interrupted), its processor state is stored in memory, in the process table **[1]**, to be able to restore this state whenever the process is scheduled again **[1]**.

(d) Order the following levels of memory/storage by their storage capacity (descending order, so largest first): main memory, caches, disks, registers. [2]

Solution: disks, main memory, caches, registers [0.5 per correctly positioned level]

(e) The following multiple-choice questions are marked negatively: For each correct answer you gain two marks and for each wrong answer you lose one mark.

There is one correct answer per question.

(i) Which of the following statements is true about *caching*?

1. Caching is usually the most space-efficient way to organise data.
2. Caching is often used to ensure data integrity and reliability.
3. Caching is typically done across multiple levels of memory/storage.

[2]

Solution: 3 (or C) **[2]**

(ii) Which of the following statements is true about *internal fragmentation*?
1. Internal fragmentation occurs when a process has more memory allocated to it than it needs.
2. Internal fragmentation refers to small amounts of memory that are not allocated to any process.
3. Internal fragmentation occurs when two processes share memory.

[2]

Solution: 1 (or A) **[2]**

(iii) Which of the following statements is true about the *Translation Lookaside Buffer (TLB)*?
1. The TLB allows the operating system to find memory that is not currently used by any process.
2. The TLB is used to store recently accessed page table entries.
3. The TLB speeds up the retrieval of data from secondary storage, such as hard disks.

[2]

Solution: 2 (or B) **[2]**

Mark table

Question	Points	Score
1	15	
2	15	
3	15	
4	15	
Total:	60	