SOLUTION: DO NOT DISTRIBUTE

Anytime before the actual exam
~~**Your real exam is 8/9 Feb!**~~
Expected Duration: 1 hour 30 minutes (expected in 2022, to confirm)
Time Allowed: 3 hours (expected in 2022, to confirm)
~~**Timed exam within 24 hours**~~

DEGREE of BSc Software Engineering (Graduate Apprenticeship)

SYSTEMS PROGRAMMING
COMPSCI 2030

(Answer all 4 questions. This is a mock exam, v20220123)

~~**This examination paper is an open book, online assessment and is worth a total of 90 marks.**~~

1. These multiple-choice questions are negatively marked. For each correct answer you gain three marks, and for each wrong answer, you lose one. There is one correct answer per question.

There are 30 total marks in this question.

(a) Which of the following would require the most memory in standard C?

1. `float`
2. `long int`
3. `char`
4. `double`

[3]

Solution:
4 (or D) **[3]**

(b) How are new types defined in C?

1. We use the typedef keyword
2. Types cannot be defined, only imported
3. We use the deftype keyword
4. C doesn't allow new types

[3]

Solution:
1 (or A) **[3]**

(c) Which of the following is true about C pointers?

1. C pointers only come from `malloc()`, `calloc()`, and `realloc()`
2. Functions cannot return pointers
3. Pointer size depends on the system running our code
4. Any pointer returned by a function must be `void`

[3]

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

(d) We cannot allocate heap memory using:

1. `malloc()`
2. `calloc()`
3. `realloc()`
4. `free()`

[3]

Solution:
4 (or D) **[3]**

(e) Stack memory is:

1. Unsafe to return the address of from a function
2. Allocated using `malloc()`
3. A method used by the operating system to allocate memory in pages
4. A modern memory management technique for deallocating space automatically, which C does not have

[3]

Solution:
1 (or A) **[3]**

(f) A use-after-free error is caused by:

1. `free()`-ing memory twice
2. Accessing a value at a previously `free()`-d address
3. Losing reference to a pointer to heap-allocated memory
4. Running out of space to store values in memory

[3]

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

(g) Assume `arr` is an array of **`int`**s, and `index` is a valid integer index in `arr`. One of the following snippets has a different value to the others. Which one?

1. `index[arr]`
2. `arr[index]`
3. `(index + arr)`
4. `*(index + arr)`

[3]

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

(h) A race condition can occur when:

1. Two or more threads wait indefinitely on a lock held by another
2. Two or more threads access the same memory location at the same time

3. Two or more threads return a value at the same time
4. Two or more threads process copies of the same values

[3]

Solution:
1 (or A) **[3]**

=> No, it's "2 (or B)"

(i) Which of the following is true of threads?:

1. A thread is created and managed by an operating system
2. A thread is independent of the process which started it
3. A thread has heap memory separate to that of other threads
4. A thread is a language construct managed by a runtime, rather than the operating system itself

[3]

Solution:
1 (or A) **[3]**

(j) `pthread_join()` is a function used to:

1. Signal that a thread should terminate
2. Signal the most important thread to the operating system, for scheduling purposes
3. Ensure safe access in a critical region
4. Wait for a thread to finish executing

[3]

Solution:
4 (or D) **[3]**

2. These questions will center around code comprehension. Each snippet of code is completely self-contained, meaning that what you read should compile without warnings or errors without additions. However, in each question there will be an error regarding the use of C, an error pertaining to memory, or an error caused by mismanaged concurrency. For each question, you should:

1. Identify the bug in the code, and
2. Suggest a fix.

There are 21 total marks in this question.

(a)

```
1  #include <stdio.h>
2
3  typedef struct _book {
4      char title[50];
5      int timesRead;
6  } book;
7
8  void printBook(book *toPrint) {
9      printf("You␣have␣read␣%s␣%d␣times.\n",
10            toPrint.title, toPrint.timesRead);
11 }
12
13 int main() {
14     book goodBook = {"House␣of␣Leaves", 0};
15     goodBook->timesRead++;
16     goodBook->timesRead++;
17     printBook(&goodBook);
18     return 0;
19 }
```

[7]

Solution:

Two marks for identifying that the struct's fields are being allocated the wrong way (should be . for direct fields, -> when struct pointer needs resolving) on lines 10, 15, or 16. **[3]**

Three marks for explaining in any way that they need to be switched around. **[4]**

(b)

```
1  #include <stdio.h>
2
3  void calc_next_collatz(int *num) {
4      if (num % 2 == 0) {
5          num /= 2;
6      } else {
7          num = num * 3 + 1;
8      }
```

```
}

int main() {
    int x = 6;
    while (x != 1) {
        calc_next_collatz(&x);
        printf("␣->␣%d", x);
    }
    printf("\n");

}
```

[5]

Solution: ... calc_next_collatz's argument: it's a pointer, not dereferenced

Two points for identifying that there's an issue with **[2]**

Three points for either suggesting dereferencing `x` in the collatz calculation function or having that function return the value it **[3]**

(c)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


void * printInfo(void * arg) {
    char * user_input = (char *) arg;

    long int len = strlen(user_input);
    printf("Argument␣%s␣has␣length␣%ld\n", user_input, len);

    pthread_exit(&len);
}

int main(int argc, char * argv[]) {

    pthread_t * threads = malloc(argc * sizeof(pthread_t));
    int i;
    for (i = 0; i < argc; i++) {
        pthread_create(&threads[i], NULL, printInfo, argv[i]);
    }

    long int ** lengths = malloc(argc * sizeof(int *));

    for (i = 0; i < argc; i++) {
        pthread_join(threads[i], (void *)&lengths[i]);
    }

    long int total = 0;
```

```
31      for (i = 0; i < argc; i++) {
32          total += *lengths[i];
33      }
34      printf("\n\nTotal␣length␣is␣%ld\n", total);
35
36      return 0;
37  }
```

[7]

Solution:

Three marks for spotting that `len` is being returned as a stack variable. **[3]**

Four marks for suggesting heap-allocating with `malloc()` or `calloc()` to fix. **[4]**

3. These questions require longform answers. There are 23 total marks available.

(a) (i) Briefly explain the difference between stack and heap memory. Explain whether each one is specific to a thread, a process, both, or neither.

[4]

Solution:
Stack memory is tied to the lifetime of functions, and is thread-specific. Heap memory is process-specific and is managed manually by a program's interaction with the operating system.
This or anything suffuciently similar is worth four marks. **[4]**

(ii) Can a memory leak happen to stack-allocated memory? Why, or why not?

[2]

Solution:
The following would be worth two marks, or any other reasonable explanation.
A memory leak can't happen in stack-allocated memory, because as soon as we are finished referencing the stack (when a function returns), all associated memory is effectively freed. As a result, all memory is automatically freed at the end of its lifetime. **[2]**

(iii) One of the design decisions in C++ which diverges from C's design is its use of a technique called RAII. What does this acronym stand for, and in what way does it help memory management in C++?

Ignore, not covered!

[4]

Solution:
RAII stands for Resource Aquisition is Initialisation. **[2]**
Any of the following, or anything sufficiently similar, is worth an additional two marks. **[2]**

- C++ programmers use RAII to tie the lifetime of a resource to the lifetime of a variable or data. Heap-allocated memory is one of these resources, making it easier to reason about.
- Memory management is made easier by using RAII to release a resource, like heap-allocated memory, when an object is destroyed. This means that program logic and resource management are largely separated.
- When a resource such as heap-allocated memory is aquired under RAII, it is aquired on initialising a related variable. When that variable is destroyed, it controls the logic concerning the release of its aquired resources. This means that the program using the resource does not need to concern itself with aquisition and release, making issues such as memory leaks (as well as deadlocks and file leaks) easier to avoid.

(b) When freeing memory, fragmentation can occur, where small chunks of memory are left

unused, but cannot be allocated because of their size and proximity to allocated blocks of memory. What operating system feature mitigates memory fragmentation?

[2]

Solution:
Paging or virtual memory get two marks. **[2]**

(c) Briefly explain what POSIX is, why one might write POSIX-compliant code, and an example of when an operating system being non-POSIX-compliant impacts code portability.

Ignore, not covered!

[6]

Solution:
Award marks for anything sufficiently similar to this:
POSIX is a standard operating systems can adhere to, which provides many things to code we write, including a standard model for threading and processes, file IO, and standard C headers. **[2]**
We might write POSIX-compliant code in settings where we would want to write a program which made use of operating system features, such as starting a thread, but wanted that code to be portable (i.e. to run on many platforms without modification or a runtime). **[2]**
An operating system being non-POSIX-compliant means that some (but not necessarily all) reliances on POSIX-provided libraries and APIs may break. An example of this is Windows' management of threads, which is not posix compliant, and means that C programs taking advantage of threading on Windows must make different OS calls to threaded code in a POSIX-compliant environment. **[2]**

(d) (i) What causes a race condition?

[2]

Solution:
Two threads entering a critical region at the same time, or anything sufficiently similar. **[2]**

(ii) Race conditions can be defended against using a concurrency primitive which, when used incorrectly, causes a deadlock. What is the name of this concurrency primitive, and how can its misuse cause a deadlock?

[3]

Solution:
Mutex mismanagement causes a deadlock. **[1]**
Deadlocks are caused by at least two threads each waiting to aquire a mutex / resource held by another, or anything sufficiently similar. **[2]**

4. Explain the significance of each line of the program on the following page commented with `???`. What do `i` and `c` do? What will this code print when run? This question is worth 16 marks.

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct _n {
    int val;
    struct _n * l;
    struct _n * r;
} n;

void c(n * p, int v, int r) {
    n * new = malloc(sizeof(n)); // ??? 1
    new->l = NULL;
    new->r = NULL;
    new->val = v;
    if (r) { // ??? 2
        p->r = new;
    } else {
        p->l = new;
    }
}

int i(n * r, int e) {
    if (e == r->val) { // ??? 3
        return 0;
    } else if (e < r->val) {
        if (r->l == NULL) {
            c(r, e, 0); // ??? 4
            return 1;
        } else {
            return i(r->l, e); // ??? 5
        }
    } else {
        if (r->r == NULL) {
            c(r, e, 1);
            return 1;
        } else {
            return i(r->r, e);
        }
    }
}

int main() {
    n r = {5, NULL, NULL};
    i(&r, 2);
    i(&r, 5);
    i(&r, 8);
    i(&r, 3);
}
```

[16]

Solution:

The snippet of code provided defines a BST and provides an insert operation on it — the logic is effectively the same as the insert logic from the assessed exercise.

Two marks for noting that `c` creates a new node on the tree, and `i` inserts a value onto the tree. **[4]**

We avoid inserting the same value twice, so the data structure contains the values `2, 3, 5, 8` when the program terminates. **[2]**

Two marks for each comment explanation.

1. We need to heap-allocate new nodes on the tree, because nodes reference each other across different function scopes. **[2]**
2. We provide an **`int`**, treated as a `bool`, which represents whether we are inserting to the right or the left of our parent node. The if statement controls this. **[2]**
3. We avoid inserting the same value twice, so if we have found the value we are interested in, we return `false`. **[2]**
4. We need to create a new node to the left of our current node, so we make a call to `c` with the final parameter set as `0` signalling an insert to the left. **[2]**
5. We need to find an appropriate place to insert. Make a recursive call, inserting on the left subtree. **[2]**

CONTINUED OVERLEAF

Mark table

Question	Points	Score
1	30	
2	19	
3	23	
4	16	
Total:	88	