| | |
|---|---|
| Student number: | 2467273 |
| Course title: | COMPSCI4081 Systems Programming (H) |
| Questions answered: | ALL |

# Q1.

a)

i. I would use float since it is the smallest necessary data type, which would take 4 bytes, using ‘f’ for a float literal to not be cast to double or something else.
float a = 10.1f;

ii. I would use char, which would take 1 byte.
char b = ‘1’;

iii. I would use float since it is the smallest necessary data type, which would also take up 4 bytes.
float c = 255.2;

iv. Summing up both variables of the struct, we get 12 bytes per visible char in the string, 1 byte for the additional ‘\0’ char at the end of string literal, and 4 bytes for the integer, resulting in 17 bytes.

b)

i. The variable ‘lex’ is not initialised to anything, and the memory is not deallocated at the end, the omission of which would result in memory leaks.

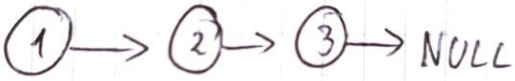
Helper function:

```
freeNodes(struct A* node) {
   if(node == NULL) return;
   freeNodes(node->next);
   free(node);
}
```

20. lex = NULL;
25. freeNodes(lex);

ii. The condition should check whether the node was added correctly and exit the program if not.
12. li == NULL
13. exit();

iii. A linked list is created, first by initialising a struct with NULL, then by adding three nodes in succession to ‘lex’, each pointing to the next one, while the last one points to NULL. In line 24, all three integer values of the list are printed out by accessing each node’s value with successive ‘next’ accesses. Then, in line 25, the whole list is freed by a recursive helper function in b) (i) (freeing each node going backwards from the last one).


1
2
3
NULL

## 2.

a)

i.

Although Count should be as the first variable in the stack, if omitting parameter variables, it would have been allocated to the stack before the for loop (when it is declared with a known size (int = 4 bytes), not initialised), and the question states that the stack is empty at the start of the loop.

In fact, all of the variables encountered would be allocated to the stack before the for loop since they would either be part of the parameter list (added to stack right to left: Node[] of Nodes, the variables of which would be added in an unknown sequence since the struct definition is not given) or initialised before the loop (Count and the four Sums), which is why it does not make sense why the stack would be empty before the loop since all of the variables are just accessed to previously allocated ones in the stack, not declared ones in the loop. Given that it is empty at the start, the stack should, in fact, be empty by the end of the loop since there are no variables declared in the loop.

| | |
|---|---|
| Stack | SumXiByDi |
| | SumYiByDi |
| | SumTiByDi |
| | Sum1ByDi |

ii. The dot notation would be better since it ensures that memory addresses that are close together are accessed (because they are parts of the same struct, the variables of which are stored next to each other). However, with arrow notation, pointers are accessed towards memory addresses that could be stored far away from another arrow pointer access, which means that more data blocks would be needed to be accessed during execution, thus slowing down the program.

b)

i. Double-free. An already freed pointer (memory address) is freed again.

ii. Dangling pointer. The root points to the left child, which is already freed, thus possibly resulting in undefined behaviour if root gives access to its left child, a completely unknown data point.

iii. Memory leak. mem is malloc'd twice, meaning that two memory addresses are allocated to the heap with the same variable name, resulting in the programmer being able to free only the last address, not the first one (since it is inaccessible).

c)

8. db_open(url);
12. db_close(connection);

RAII ties memory management to the lifetime of the variable, which means that programmers do not need to worry about explicitly allocating and deallocating memory to variables, it is automatically done so when the variable is initialised and ends its scope, respectively.

# 3.

a)

i. Line 3 (access of c and modification of it).

ii. If one thread got the value of c from line 3, then another thread got its value from line 3, both would increment, e.g., 0, c by 1 (again, in line 3) and return 1 in line 4, even though they altogether should have computed 2.

iii. *Fewer* than.

iv. The function will actually be called multiple times, but because of the race condition the counting can be intermixed between threads and not updated sequentially, like it should (each thread waiting for the critical region to be free), resulting in fewer counted calls.

b) Condition variables can be used with ‘wait’ to block a calling thread and release the mutex if it is not in use, and ‘signal’ to signal that a thread is done with the critical region and another thread can lock its mutex and continue.

c) Concurrent programming is particularly unpredictable with many threads running, which is why it is sometimes difficult to find why and when one particular thread caused a bug when each execution of the program can be different.

d) Semaphores allow specifying how many threads can access a critical section at a time with an incrementing (called by ‘signal’) and decrementing (called by ‘wait’) counter.

e) A future is a type of data type for representing a variable which has not been returned/computed yet but will be sometime later, the value of which can be forced to be computed (by waiting for its computation) by .get(). A promise, however, is seen as the writing end of a communication channel, i.e., the writer of an asynchronous operation can set the promise’s value, opposite from a future.

f) These asynchronous abstractions allow programmers not to think about mutexes and condition variables as they are automatically taken care of by futures, promises and their respective acquiring/setting functions, which lock and unlock mutexes and condition variables as needed.

g) The if statement in line 3 should be a **while** loop instead, because, otherwise, the code would run critical code after just one waiting call, which could cause more race conditions.