

DayOfWeek DayOfMonth May 2019

??.?? am/pm – ??.?? am/pm

(Duration: 1 hour 30 minutes)

DEGREES OF MSci, MEng, BEng, BSc, MA and MA (Social Sciences)

System Programming H

(Answer all 3 questions.)

This examination paper is worth a total of 60 marks

The use of a calculator is not permitted in this examination

|  |
| --- |
| INSTRUCTIONS TO INVIGILATORS  **Please collect all exam question papers and exam answer scripts and retain for school to collect. Candidates must not remove exam question papers.** |

1. **C Programming and Data Types**
   1. In the lectures we described the purpose of data types with the phrase  
      *“data types give bits meaning”*.  
      Explain this phrase briefly by explaining how data types help to write meaningful programs and prevent errors. Give an example to support your answer. **[2]**A bit pattern in memory can have many different meanings. Data types assign a fixed meaning to the bit pattern and the type safety of the language ensures that this meaning is not lost when we perform computations. An example might be that the operation +1 will result in different instructions and modification of bit patterns depending if an int or a float is modified.
   2. Implement a *doubly-linked list* in the C programming language.  
        
      Sketch the implementation based on the following interface. Minor syntax errors will not be penalised. Make sure that you handle pointers correctly. Provide comments explaining your implementation.  
        
        
      i) Define the structs. Node should be able to store a value of an arbitrary type. List should store both ends of the list (its *left* and *right* end). **[4]**ii) Implement list\_create and list\_destroy. **[4]**iii) Implement push\_right and pop\_left. **[5]**iv) Implement for\_each\_node. **[2]**v) Implement a main function that uses the provided interface to create a list and push three nodes with string values on the list. The for\_each\_node function should be used to print all values before the list is destroyed. There should be no memory leaks. **[3]**i) 2 mark for node struct declaration; 1 mark for void\*; 1 mark for list struct decl.  
      ii) 2 marks list\_create (1 mark malloc; 1 mark struct init); 2 marks for list\_destroy (1 mark for traversal; 1 mark for free calls)  
      iii) 3 mark push\_right (2 marks for node\_create impl; 1 mark for pointer fiddling) 1 mark for pop\_left; 1 mark for null pointer checks  
      iv) 1 mark for traversal, 1 mark for handling of function pointer  
      v) 1 mark for list\_create call; 1 mark for push\_right calls, 1 mark for for\_each node call and function passing

// a node in the list should be able to store a value of any type  
typedef struct node Node;

// the list should store its left and right end  
typedef struct list List;

// create an empty list; returning NULL if unsuccessful  
List\* list\_create();

// destroys the list and all remaining nodes in it  
void list\_destroy(List\* l);

// allocate a new node and add it to the right side of the list  
void push\_right(List\* l, void\* value);

// remove the left most node from the list and return it  
Node\* pop\_left(List\* l);

// visit each node (from left to right) and call the function  
// for each visited node. The function cannot be called with a  
// NULL pointer.  
void for\_each\_node(List\* l, void(\*fun)(Node\*));

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

struct node {  
 struct node\* prev;  
 struct node\* next;  
 void\* value;   
};  
struct list {  
 struct node\* left;  
 struct node\* right;  
};

List\* list\_create() {

List\* l = malloc(sizeof(List));

if (l) {

l->left = NULL; l->right= NULL;

}

return l;

}

Node\* node\_create(void\* value) {

Node\* n = malloc(sizeof(Node));

if (n) {

n->value = value;

n->next = NULL;

n->prev = NULL;

}

return n;

}

void push\_right(List\* l, void\* value) {

if (l == NULL) return;

Node\* n = node\_create(value);

if (l->right) {

n->prev = l->right;

l->right->next = n;

}

l->right = n;

if (l->left == NULL) l->left = n;

}

Node \* pop\_right(List\* l) {

if (l == NULL || l->right == NULL) return NULL;

Node\* n = l->right;

l->right = l->right->prev;

if (l->left == n) l->left = NULL;

return n;

}

void push\_left(List\* l, void\* value) {

if (!l) return;

Node\* n = node\_create(value);

if (l->left) {

n->next = l->left;

l->left->prev = n;

}

l->left = n;

if (l->right == NULL) l->right = n;

}

Node\* pop\_left(List\* l) {

if (l == NULL || l->left == NULL) return NULL;

Node\* n = l->left;

l->left = l->left->next;

if (l->right == n) l->right = NULL;

return n;

}

void list\_destroy(List\* l) {

Node\* n = pop\_left(l);

while (n) {

free(n);

n = pop\_left(l);

}

free(l);

}

void for\_each\_node(List\* l, void(\*fun)(Node\*)) {

if (l) {

Node\* n = l->left;

while (n) {

(\*fun)(n);

n = n->next;

}

}

}

void print(Node\* n) {

printf("%s\n", n->value);

}

int main() {

List\* l = list\_create();

push\_right(l, "first");

push\_right(l, "second");

push\_right(l, "third");

for\_each\_node(l, print);

list\_destroy(l);

}

**TOTAL MARKS [20]**

1. **Memory and Resource Management & Ownership**
   1. Explain why it is forbidden to dereference a void-pointer. **[2]**A void pointer holds an address to a value of unknown data type. It is unclear what the bits at this memory location mean. The programmer first has to re-establish the meaning of the bits by casting the pointer.
   2. There are two main areas of memory: the stack and the heap.  
      Explain for both how memory is managed and what responsibility the programmer has. **[2]**The stack is the automatically managed area. Variables are put on the stack when a block is entered and removed in reverse order when the block is exited. The heap is managed manually by the programmer who is responsible to call free correctly.
   3. Variables in C can have different *lifetimes*. Explain briefly the three different types of lifetime a variable in C can have. **[3]**variables with *automatic lifetime* (or *stack lifetime*) are deallocated at the end of the block they are declared in.  
      variables with *static lifetime* are deallocated at the end of the program.  
      *allocated variables* are managed manually by the programmer who determines their lifetime
   4. Explain why it is dangerous to return a pointer to a local variable from a function. **[2]**The pointer has a longer lifetime than the variable. Any access to the original variable will be invalid as it has been (potentially) deallocated.
   5. Name and briefly discuss three major challenges with manual memory management using malloc and free. **[3]**1. Double free errors (free is called multiple times)  
      2. Dangling pointers (pointers to variables that have been freed)  
      3. Memory leaks (free is never called)  
      Any other valid challenge is awarded 1 point
   6. Explain the concept and benefits of *Ownership* for resource and memory management. Explain how this is implemented in C++ with the RAII technique. Explain the role of the special *constructor* and *destructor* functions in this context. **[4]**With ownership we identify a single entity which is responsible for managing a resource. In C++ this is enforced via Resource Allocation Is Initialization: resources are allocated in the constructor of an object and deallocated in the destructor of the object. By creating such an object handler on the local stack, the management of the resource is tied to the lifetime of a stack variable.
   7. Implement a structure db\_handle that follows the C++ RAII technique and manages a database connection to ensure that it will be closed at the end of the program. Use in your implementation the following interface for opening and closing the database connection.  
        
        
        
      Provide an implementation of the db\_handle struct and a sketch of a main function using the struct to open a database connection. Indicate in comments in the main function when the connection will be opened and closed. **[4]**DBConnection\* db\_open(const char\* url);

DBConnection\* db\_open(const char\* url);  
void db\_close(DBConnection\* connection);

void db\_close(DBConnection\* connection);

struct db\_handle {

DBConnection \* connection;

db\_handle(const char\* url) {

connection = db\_open(url);

}

~db\_handle() {

db\_close(connection);

}

}

**TOTAL MARKS [20]**

1. **Concurrent Systems Programming**
   1. Explain the difference between *Concurrency* and *Parallelism*, as discussed in the Lecture. **[2]**Concurrency is a programming paradigm; Parallelism is about making programs faster  
      Concurrency is about dealing with lots of things at once; Parallelism is about doing lots of things at once
   2. Describe briefly one mechanism that is used to protect critical sections in multi-threaded code. **[2]**Mutual exclusion which is e.g. ensured via mutexes (but possibly also via other related mechanisms such as semaphores)
   3. Look at the multi-threaded source code sketch below.  
      i) Is this program safe? If not explain why.  
      ii) What will be the size of the buffer at the end of the program?  
      Explain your answer.  
       **[4]**No this program is not safe, as the buffer is unprotected, but two threads simultaneously access it. (2 marks for this answer) It is possible that bufferPtr->pop can be called on an empty buffer. As this program has a race condition, we cannot reasonable expect any single answer. This program might crash or result in a buffer with (almost) arbitrary size.

int main() {  
 std::stack<Stuff> buffer;  
 buffer.push(make\_stuff());  
 buffer.push(make\_stuff());  
  
 auto t1 = std::thread([bufferPtr = &buffer]() {  
 for (int i = 0; i < 1000; i++)  
 bufferPtr->push(make\_stuff());  
 });  
  
 auto t2 = std::thread([bufferPtr = &buffer]() {  
 for (int i = 0; i < 1000; i++)  
 bufferPtr->pop();  
 });  
  
 t1.join(); t2.join();  
 printf(“%d\n”, buffer.size());  
}

* 1. A counting semaphore allows up to *N* many concurrent accesses to a resource.  
     As discussed in the lecture, a semaphore is created with an initial value create(int N) where N > 0. It has two atomic operations (wait and signal) which do logically the following:  
       
       
     Provide an implementation for a counting semaphore that uses condition variables and no busy waiting.   
     i) Provide the implementation of a struct semaphore **[4]**  
     ii) Provide a function void sem\_wait(struct semaphore\*) and **[4]**  
     iii) a function void signal(struct semaphore\*) **[4]**  
     You answer has to be thread safe.  
     You can either use C PThreads or C++ threads in your answer.  
     You do not have to provide any initialisation code.Or an equivalent implementation with C++ Threads  
     4 marks for a definition of a semaphore struct.  
     4 marks for sem\_signal (1 for lock/unlock, 1 for right condition, 1 for count, 1 signal)  
     4 marks for sem\_wait (1 for lock/unlock, 1 for right while, 1 for wait, 1 for signal)  
     sem\_create not necessary

void wait(semaphore S) {  
 while (S <= 0)  
 ; // do nothing  
 S--;  
}

void signal(semaphore S) {  
 S++;  
}

#include <stdlib.h>

#include <pthread.h>

typedef struct semaphore Semaphore;

struct semaphore {

int count;

int size;

pthread\_mutex\_t lock;

pthread\_cond\_t cond;

};

Semaphore \*sem\_create(int N) {

Semaphore \*sem =  
(Semaphore \*)malloc(sizeof(Semaphore));

if (sem != NULL) {

sem->count = N;

sem->size = N;

pthread\_mutex\_init(&(sem->lock),  
 NULL);

pthread\_cond\_init(&(sem->cond),   
 NULL);

}

return sem;

}

void sem\_signal(Semaphore \*sem) {

pthread\_mutex\_lock(&(sem->lock));

if (sem->count < sem->size)

sem->count++;

pthread\_cond\_signal(&(sem->cond));

pthread\_mutex\_unlock(&(sem->lock));

}

void sem\_wait(Semaphore \*sem) {

pthread\_mutex\_lock(&(sem->lock));

while (sem->count <= 0)

pthread\_cond\_wait(&(sem->cond),   
 &(sem->lock));

sem->count--;

pthread\_cond\_signal(&(sem->cond));

pthread\_mutex\_unlock(&(sem->lock));

}

**TOTAL MARKS [20]**