University of Glasgow

May 2019
(Duration: 1 hour 30 minutes)

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

System Programming H
Mock Exam
With Answers

(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

(a) Describe how strings are represented in C. How much bytes of memory do the string literal "`Hello World`" require in a C program? **[2]**

Strings are represented as arrays of individual characters, each represented by a value of type char. The end of the string is indicated by the special character \0. The string "Hello World" therefore is represented by 12 characters (including the \0) character and requires 12 bytes of memory.

(b) Implement a *binary search tree of strings* 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 and that you do not leak memory. Provide comments explaining your implementation.

```
// a node in the tree should store a string label
typedef struct node Node;

// create a Node with the given label. Returns NULL if unsuccessful
Node* node_create(const char* label);

// destroys the node and connected nodes in the tree
void node_destroy(Node* n);

// inserts a node in the right place to maintain the order of the tree
void insert(Node* n, const char* label);

// lookup a node in the tree. Returns NULL if the label is not present
Node* lookup(Node* n, const char* label);
```

You should use these string handling functions:

`int strcmp(const char* s1, const char* s2);`
returns 0 is `s1` and `s2` are equal, a value `<0` if `s1` is smaller than `s2`, `>0` if `s1` is bigger than `s2`

`int strlen(const char* s);`
returns the number of printable characters in the string

`char* strcpy(char* destination, const char* source);`
copies the string at `source` to the `destination`

- Define the `struct`. A `Node` should store a string label. **[2]**

```
struct {
  char* label;
  Node* left_child;
  Node* right_child;
};
```

- Implement `node_create` and `node_destroy`.
  Be careful to handle all allocations and pointers properly. **[6]**

```
Node* node_create(const char* label) {
  if (!label) return NULL;
  Node* n = malloc(sizeof(Node));
  if (!n) return NULL;

  n->label = malloc(strlen(label)+1);
  if (!n->label) { free(n); return NULL; }
  strcpy(n->label, label);
  n->left_child = NULL;
  n->right_child = NULL;

  return n;
}

Node* node_destroy(Node* n) {
  if (!n) return;
  node_destroy(n->left_child);
  node_destroy(n->right_child);
  free(n->label);
  free(n);
}
```

- Implement `insert_node`. Ensure that the node is inserted at the right place to maintain the search order of the tree. If a node with the given label exists already in the tree no node should be added. **[4]**

```
void insert(Node* n, const char* label) {
  if (!n) return;
  int cmp = strcmp(n->label, label);
  if (cmp == 0) return;
  if (cmp < 0) {
    if (n->left_child == NULL) {
      n->left_child = node_create(label);
    } else {
      insert(n->left_child, label);
    }
  }
  if (cmp > 0) {
    if (n->right_child == NULL) {
      n->right_child = node_create(label);
    } else {
      insert(n->right_child, label);
    }
  }
}
```

- Implement `lookup`. **[3]**

```
Node* lookup(Node* n, const char* label) {
  if (!n) return NULL;
  int cmp = strcmp(n->label, label);
  if (cmp < 0) { return lookup(n->left_child, label); }
  if (cmp > 0) { return lookup(n->right_child, label); }
  // cmp must be 0
  return n;
}
```

- Implement a `main` function that uses the provided interface to create a binary search tree with three nodes with the labels “Systems”, “Programming”, “2019”. Write a function that prints all values in the tree and show how it is used. The tree should be destroyed at the end of the program and there should be no memory leaks. **[3]**

```
void print(Node* n) {
  if (!n) return;
  print(n->left_child);
  print(n->right_child);
  printf(“%s\n”, n->label);
}

int main() {
  Node* n = node_create(“Systems”);
  insert(n, “Programming”);
  insert(n, “2019”);

  print(n);

  node_destroy(n);
}
```

TOTAL MARKS [20]

2. Memory and Resource Management & Ownership

(a) Explain the purpose of a `void`-pointer in C. Give an example use case. **[2]**

A void pointer allows referring to values of an arbitrary type. The programmer is forced to cast the void pointer into a pointer of a concrete data type before dereferencing it. Void pointers are essential for implementing generic data types that should store values of any arbitrary type.

(b) Explain the difference between the two memory regions *stack* and *heap*. Explain briefly how the programmer manages memory on the heap in C. **[2]**

The stack is automatically managed by the compiler for this the size of every variable must be known without executing the program. The heap is managed explicitly by the programmer by using the malloc function to request memory of a certain size and deallocating it by calling free.

(c) Values are passed to function via *call-by-value*. Explain briefly what this means. Explain how arrays are treated when passed to a function and how this relates to pointers. **[3]**

Call-by-value means that arguments are copied into the parameter of a function. Arrays are not copied when passes into a function, instead the address (i.e. pointer) to the first argument is copied. Therefore, the value of arrays can be modified inside of a function.

(d) Explain what a *segmentation fault* is and describe a strategy for finding out which part of a C program triggered this error. Give a common cause for a segmentation fault. **[3]**

A segmentation fault is raised by the hardware notifying the operating system that your program has attempted to access a restricted area of memory.
A strategy to find the responsible part of the C program is to use a debugger and use the backtrace feature which shows the call stack at the time of the segmentation fault.
The most common cause for a segmentation fault is the dereferencing of a NULL pointer.

(e) Explain the concept of *Ownership* for memory management. Describe the benefits over the direct use of `malloc` and `free`. Explain how in C++ the RAII technique works and how this relates to the lifetime of variables and what role the special *constructor* and *destructor* functions play in this context. **[4]**

Ownership means that we identify a single entity which is responsible for managing a location in memory.
The benefits over malloc and free are that common problems of manual memory management are avoided by construction, for example: double free errors (calling free multiple times) or memory leaks (not calling free at all).

RAII in C++ ties the management of memory to the lifetime of a variable on the stack. For this we create a special struct that defines a *constructor* which allocates the resource (i.e. calls malloc) and a *destructor* that deallocates the resource (i.e. calls free). A variable of this struct type on the stack will allocate the memory when it is constructed and automatically deallocate it when the variable goes out of scope.

(f) C++ provides two main smart pointers: `std::unique_ptr` and `std::shared_ptr`. Explain the difference between the two and discuss in which situation which one should be used. **[2]**

The std::unique_ptr models unique ownership when we can identify a unique owner of the resource that is responsible for managing its lifetime. An example would be the children of a node in a tree – the children will automatically be deallocated whenever the node is deallocated itself.
The std::shared_ptr models shared ownership for situations where it is not possible to find a unique owner of the resource. An example would be a graph without cycles (such as a DAG) where multiple a node can have multiple incoming edges. A node is automatically deallocated when the last incoming edge disappears, but not before.

(g) Give the definition of a `struct` for a Node with a string label in a directed acyclic graph (DAG) [a graph with directed edges and without cycles] that uses C++ containers and C++ smart pointers to model and manage the graph data structure. Explain your solution briefly.
Would a similar implementation be possible for a graph with cycles?
Justify your answer.

Describe (but do not implement) what implementation would be required in C to achieve a correct and leak free implementation. **[4]**

```
struct node {
  std::string label;
  std::vector<std::shared_ptr<struct node>> edges;
};
```

The outgoing edges of a node a stored in a vector (an array that can dynamically grow or shrink). As there is no unique ownership (there can be multiple incoming edges into a node) we use a shared pointer to model the shared ownership.

This design is problematic if there are cycles in the graph, as then the shared_ptr form a cycle as well and no memory is released.

An implementation in C would need to ensure that all nodes in the graph are freed after they are no longer accessible. For this a counter could be maintained in the node that counts the incoming edges. Once that reaches zero the node can be freed.

TOTAL MARKS [20]

3. Concurrent Systems Programming

(a) Describe the term *Thread* and distinguish it from the term *Process*. **[2]**

A thread of execution is an independent sequence of program instructions.
Multiple threads can be executed simultaneously.
Multiple threads share the same address space of a single process.
The operating system ensures that addresses spaces of processes are protected from each other.

(b) Give an example showing the need for mutual exclusion to ensure the correctness of the results of a computation. **[2]**

Consider the simultaneous removal of elements from a linked list.
Starting from:
a -> b > c -> d -> NULL
Two threads simultaneously start removing elements from the list.
The first thread removes b by moving the next pointer of a to c:
a ---> c -> d -> NULL
b-/
The second removes c by moving the next pointer of b to d:
a ---> c -> d -> NULL
b-/
The resulting list still contains node c (even though it was deleted explicitly).

(c) C++ provides *futures* and *promises* as higher-level synchronisation abstractions. Describe how they work together to simplify writing multi-threaded code. **[2]**

A future is a handle representing a value that is not yet computed.
We can pass this handle around or store it in memory.
Once we require the value we call .get() which waits until the value has been computed.

The promise is the "other side" of a future, allowing to provide a value (*fulfilling the promise*) once it has been computed.

Without future and promise the value would have to be explicitly protected by a mutex (or another low level primitive) and a condition variable that could be used to wait for the value to be computed.

(d) Look at the following API definition of the `ts_set` class.

```
struct ts_set {
private:
  std::set<int> set;
  std::mutex m;

public:
  std::set<int>::iterator find(int i) {
    std::unique_lock<std::mutex> lock(m);
    return set.find();
  }
  std::set<int>::iterator begin() {
    std::unique_lock<std::mutex> lock(m);
    return set.begin();
  }
  std::set<int>::iterator end() {
    std::unique_lock<std::mutex> lock(m);
    return set.end();
  }
  void insert(int i) {
    std::unique_lock<std::mutex> lock(m);
    set.insert(i);
  }
  void erase(std::set<int>::iterator pos) {
    std::unique_lock<std::mutex> lock(m);
    set.erase(pos);
  }
};
```

The provided interface and implementation is not safe to use in multithreaded code despite using a mutex. Explain why and give an example describing a problematic scenario using the API with multiple threads.

[4]

The interface is not thread safe. Iterators pointing to an element could be invalidated by another thread by removing (or adding) elements.

Two potential problematic scenarios are:
1. A call to find returns an iterator, but immediately after the call the corresponding element is removed from the set, so that the iterator is now invalid.
2. Calls to find and end test if a value is in the set and only if this is the case a call to insert should add the value. This is impossible to achieve with this interface, as the mutex is released between the individual operations and allow for the tested property (that the value is not in the set) to become false.

(e) Design a thread safe interface for a stack of `int` values with a limited size and implement it in C++ or PThreads.

The stack should allow to *push* new elements onto the stack and to *pop* (i.e. remove) elements from the stack.
When the stack is full *push* should block and wait until there is again space, similarly *pop* should block when the stack is empty until there are elements to be removed.
Use condition variables in your solution to avoid busy waiting. **[6]**

```
#define N 4
struct stack {
private:
  int values[N];
  int index = 0;
  std::mutex m;
  std::condition_variable cv_not_full;
  std::condition_variable cv_not_empty;

public:
  void push(int i) {
    std::unique_lock<std::mutex> lock(m);
    cv_not_full.wait(lock, [this]{ return index < N; });
    values[index] = i;
    index++;
    cv_not_empty.notify_one();
  }

  int pop() {
    std::unique_lock<std::mutex> lock(m);
    cv_not_empty.wait(lock, [this]{ return index > 0; })
    index--;
    int i = values[index];
    cv_not_full.notify_one();
    return i;
  }
};
```

Implement *try_push* and *try_pop* that will not block but instead immediately return when the stack is full/empty. Both functions should indicate if the operation was performed successfully or not.

[4]

```
bool try_push(int i) {
  std::unique_lock<std::mutex> lock(m);
  if (index >= N) {
    return false;
  } else {
    values[index] = i;
    index++;
    return true;
  }
}

std::optional<int> try_pop() {
  std::unique_lock<std::mutex> lock(m);
  if (index <= 0) {
    return std::optional<int>();
  } else {
    index--;
    return values[index];
  }
}
```

TOTAL MARKS [20]