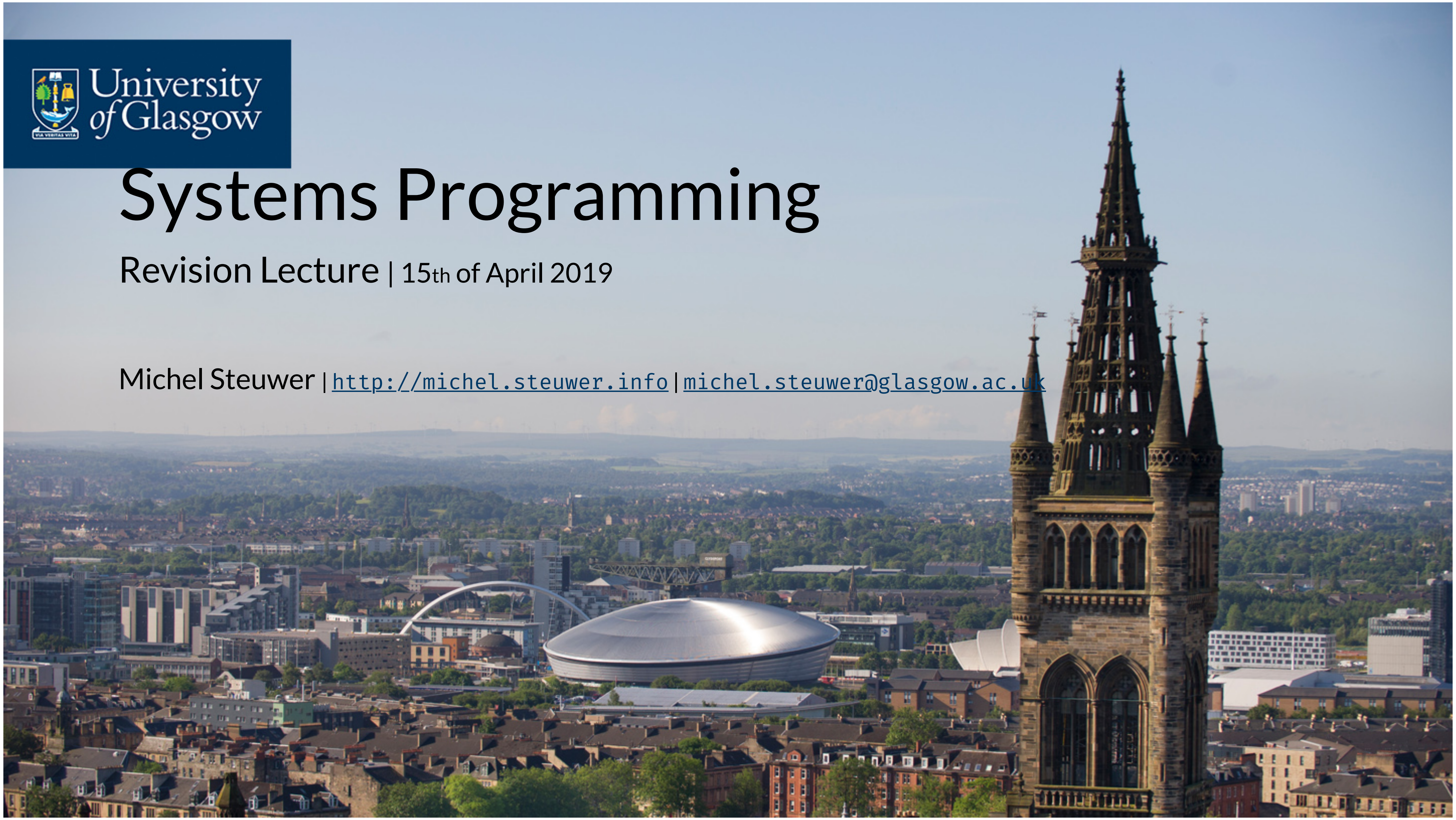

Systems Programming

Revision Lecture | 15th of April 2019

Michel Steuwer | http://michel.steuwer.info | michel.steuwer@glasgow.ac.uk

System Programming - Part 1

- In this course we have studied systems programming. We have learned, that:
    - **data types** give bit representations in memory a meaning
    - **structs** allow us to implement custom data structures (like link lists or trees)
    - every variable is stored at a fix location **memory**
    - a **pointer** is a variable storing the address of a memory location
    - memory on the **stack** is automatically managed but memory on the **heap** must be managed manually
    - to organise resource and memory management we should think about **ownership**
    - in C++ ownership is implemented by tying the resource management to the **lifetime** of a stack variable
    - there exist a set of **smart pointers** and **containers** for easy and non-leaking memory management

System Programming - Part 2

- In the second part of the course, we talked about programming concurrent systems. We learned, that:
    - **concurrency** is a programming paradigm to *deal with lots of things at once*
    - **parallelism** is about *doing lots of things at once* and this way making a program run faster
    - multiple **threads** share the same address space of a single **process**
    - `Pthreads` is a threading implementation in `C`, `C++` provides threads in its standard library
    - **mutual exclusion** ensures that two threads don't simultaneously enter their **critical section**
    - **condition variables** are a synchronisation mechanism to avoid busy waiting for a condition
    - the bounded buffer is an example of a **monitor**, an object that encapsulates low level synchronisation

Exam marking

Question:

> *Looking over the mock exam, it is the same number of total marks and duration as exams for AP from previous years, but the answers award much less marks for the same work as for previous exams and we have to do much more to get the same grade.*
> *To give a concrete example, both questions 1 in the mock exam and in exam 2016 are worth 20/60 marks and are about implementing BSTs. However, in the 2016 paper we are only supposed to implement: structures, create, lookup, and main.*
> *In the mock exam we are to implement structures, create, destroy, insert, lookup, print tree, main.*
> *I only use this question as an example. Pretty much all the questions follow this pattern of giving much less marks for the same amount of effort than for previous years.*
> *I am posting this, because I wanted to know whether we should expect this kind of marks weighting for the final exam or the sort of weighting like in the previous years?*
> *Also, on a another note, in the mock exam, if I used strdup() in node_create instead of strcpy() and strlen(), would I be marked down?*
> *Thanks*

Exam - 9th May 2019

- The mock exam on moodle is a good preparation for the exam!
- Questions in the exam will be in a similar style
- Questions in the exam will be awarded similar amount of points
- The exam might cover all topics discussed in all 8 lectures (without the guest lecture)
- Programming skills will be required - in C and C++
- Minor syntax errors will not be penalised! (e.g. missing a semicolon)
  But, syntax around pointers (`*`, `&`, `->`) is important and mistakes will be penalised.
- Necessary C/C++ APIs (e.g. for string handling) will be given in the exam.
  But, you must know how to use `malloc`, `free`, and C++ smart pointers

Semaphores

- Question:

  > *Just to request for you to cover semaphores in the revision lecture.*
  > *I would find it quite helpful to have it went over and shown in practice.*

- Semaphores have been covered in Lecture 6
- A *semaphore* is a synchronisation primitive that counts how many items of a resource are used and blocks access if no resources are available
- Example: ensure that concurrently used array does not overflow

```
int array[5];
sem_t count; // counting how many elements are free in array

int main() {
  count = sem_create(5, 5); // 5 elements in total, all are free
  /* ... */ }

void produceItem() { sem_wait(count);   addItem(array);     }
void consumeItem() { removeItem(array); sem_signal(count); }
```

Lectures 7

- Question:

    I'm just revising your course material and every lecture is very well structured with clear ideas however I struggle to grasp all the ideas from Lecture 7 and Lecture 8 (especially from launching tasks with std::async to std::future in Lecture 7). I listened to the lecture recordings and it also seemed to be quite rushed.
    Would you mind summarising the main concepts from these lectures that are important to understand for the exam? I also struggle to image what sort of understanding of these concepts is required (implementation, theory etc.).

- Main concepts from Lecture 7:
    - Concurrency in C++ with **`std::thread`**, **`std::mutex`** and **`std::condition_variable`**
    - **Tasks** are a higher level abstraction of threads allowing us to focus on *communication* between tasks rather than low-level synchronisation of threads. Tasks are launched with `std::async`.
    - A tasks communicates it's computed results via a **`std::future`**.
      A future object represents a value that might not yet be computed.

Lecture 8

- Main concepts from Lecture 8:
    - **Thread safe interfaces** (discussed separately)
    - A **task system** allows to efficiently use hardware threads
    - Using multiple queues and **work-stealing** is essential to achieve good performance

Thread safe interfaces

Question:

> *I had this question for a while but didn't get round to email you. From your feedback on the second assessed coursework I quote: "TheTable uses a mutex to protect is internal state, but its usage is not thread safe. The usage at the end of process is problematic. The experimentation is sound, the anaylsis of the results is ok. (70/90) ", I realised that in actual fact many other students got the exact same feedback, but none of them got an explanation on what about TheTable is not thread safe or why the usage is problematic at the end of process.*
> *As far as I am aware, we all had a very similar implementation, and applied what we were taught in the lectures and I don't see where we should have made it more thread safe, could you please explain?*

- We discussed the importance of thread safe interfaces in Lecture 8 with an example of `std::list`

Thread safe interfaces - Answer

- So, what is problematic about the usage at the end of process?

```
static void process(const char *file, std::list<std::string> *ll) {
  // ...
    // 2bii. if file name not already in table ...
    if (theTable.find(name) != theTable.end()) { continue; }
    // ... insert mapping from file name to empty list in table ...
    theTable.insert(name, {});
    // ... append file name to workQ
    workQ.push_back(name); } /* ... */ }
```

- Even if all methods of `theTable` are individually thread safe (i.e. using a mutex to protect the internal state) the following scenario is possible:
    - Thread A and B execute `process` concurrently
    - Thread A calls `theTable.find(name)` and `theTable.end()`
    - Thread B calls `theTable.find(name)` and `theTable.end()`
    - Both threads check if they are equal (this is not part of the critical section) and they are
    - Both threads call `theTable.insert()` and both push the filename on the `workQ`
- The API has to change to protect the entire check if `name` is in the table!

Questions? | What to revise next?

- Good luck with all your exams!

Systems Programming

Revision Lecture | 15th of April 2019

Michel Steuwer | http://michel.steuwer.info | michel.steuwer@glasgow.ac.uk