Concepts and Models of Parallel and Data-centric Programming

Shared Memory III

Lecture, Summer 2020

Dr. Christian Terboven <terboven@itc.rwth-aachen.de>

Outline

0. Organization
1. Foundations
2. **Shared Memory**
3. GPU Programming
4. Bulk-Synchronous Parallelism
5. Message Passing
6. Distributed Shared Memory
7. Parallel Algorithms
8. Parallel I/O
9. MapReduce
10. Apache Spark

a. Processes and Threads
b. Threading in C++
c. RAII idiom, Move Semantics
d. Mutual Exclusion
e. Condition Variable
f. Example: Queue

Example: Queue (part I: Challenges)

Motivation: Call Center Software

- Lets put us in charge of software design for a call center
  - during peak hours, calls arrive faster than they can be answered
  - when a call arrives, play a recorded announcement
  - when an operator is free and answers a call, remove it from the queue

- What do we need?
  - An implementation for the Queue

- What will be tricky?
  - Making sure the Queue works correctly when used with multiple threads

Queue: ADT design

- Container holding elements of type T
  - Function enq(): enqueue an element
  - Function deq(): de-queue an element
  - Internal representation of data could be an array
    - Variables head and tail point to indexes in that array

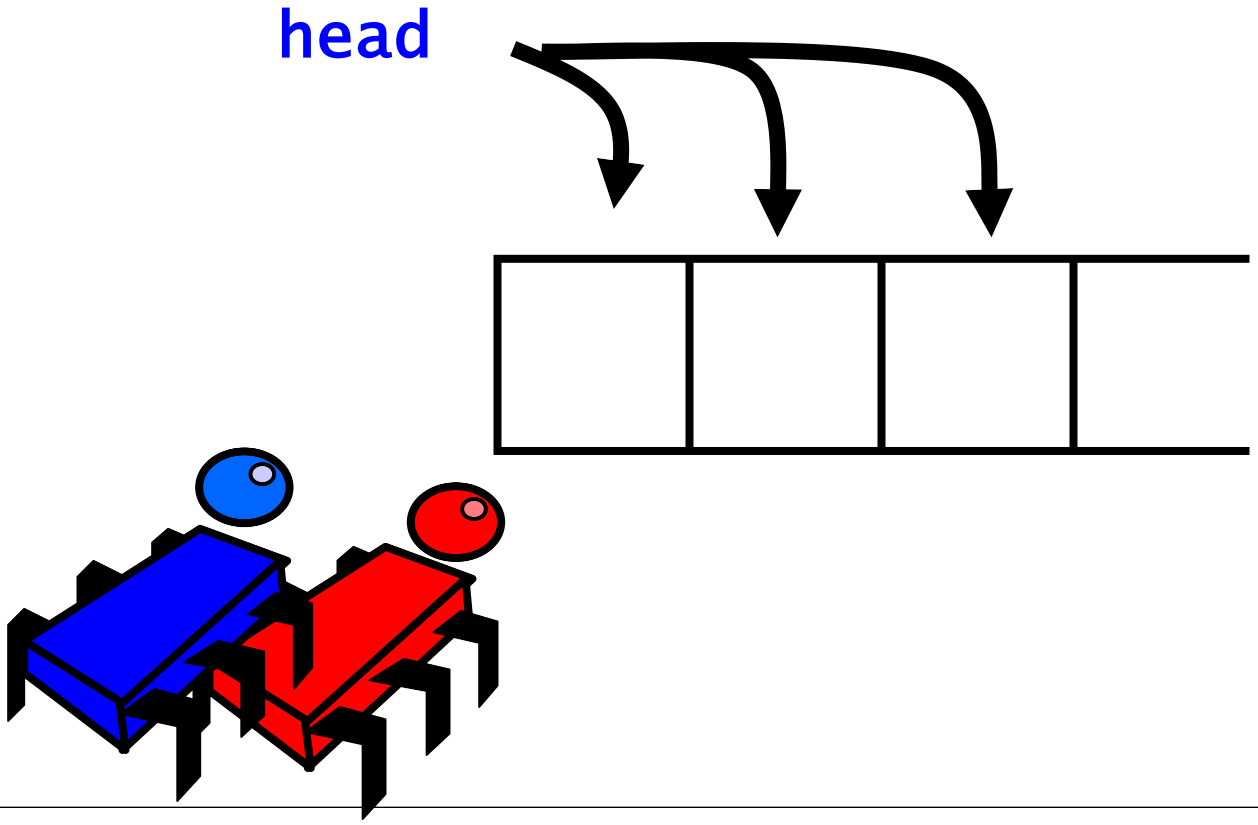
```
template<class T> class NaiveQueue
{
protected:
    T* items; int head, tail;
public:
    NaiveQueue() { ... initialization ... }
    void enq(T element) { ... enqueue element: tail++ % QSIZE ... }
    T deq() { ... return element: head++ % QSIZE ... }
};
```

That was easy?!

- What may go wrong?

That was easy?!

- What may go wrong?



Mutual Exclusion

Race Condition vs. Data Race

Definitions in the sense of C++:

- **Race Condition:** any situation where the outcome depends on the relative ordering of execution of operations on two or more threads
  - In most cases, this is acceptable / not a bug
  - The term is often used to describe problematic race conditions

- **Data Race:** specific case of race condition because of concurrent modification of shared data (i.e., a single object)
  - Result: undefined behavior
  - A **data race** occurs when: two or more threads in a single process access the same memory location concurrently, and at least one of the accesses is for writing, and the threads are not using any exclusive locks to control their accesses to that memory.

Mutexes / 1

- A mutex implements mutual exclusion
  - *Lock* the associated mutex before accessing a shared variable, *unlock* it afterwards
  - Most general data-protection mechanism available
    - Challenge 1: protect the right data (will be covered later)
    - Challenge 2: avoid race conditions in the interface (will be covered later)

- Class `std::mutex`
  - Representation of a mutex for "system threads"
  - Defined in header `<mutex>`
  - Reference: https://en.cppreference.com/w/cpp/thread/mutex

Mutexes / 2

- `std::mutex` is usually not accessed directly
  - RAII-style use!

- Class `std::lock_guard`
  - RAII-style wrapper for `std::mutex`
  - Takes ownership of lock during construction
  - Releases ownership of lock during deconstruction (when scope is left)
  - Defined in header `<mutex>`
  - Reference: https://en.cppreference.com/w/cpp/thread/lock_guard

- Caution: a `std::mutex` may not be locked twice

Mutexes / 3

- `std::mutex` is usually not accessed directly
  - RAII-style use!
- Class `std::unique_lock`
  - RAII-style wrapper for `std::mutex`
  - Allows for deferred locking, time-constrained attempts at locking, recursive locking, transfer of lock ownership, and use with condition variables
  - Defined in header `<mutex>`
  - Reference: https://en.cppreference.com/w/cpp/thread/unique_lock
- Higher functionality comes at a (slightly) higher cost
  - Overhead
  - Memory consumption

Mutual Exclusion vs. Deadlock

- Mutual Exclusion
  - Two threads never in same region simultaneously
  - This is a ***safety*** property
    - safety: something bad will never happen

- No Deadlock
  - if only one wants in, it gets in
  - if both want in, one gets in.
  - This is both ***liveness*** and ***safety*** property
    - liveness: something good will happen (but we do not know when)

Examples

Safe increment of global counter

- To protect the global counter, a mutex is available

```
int g_i = 0;
std::mutex g_i_mutex; // intended to protect g_i

void safe_increment()
{
    std::lock_guard<std::mutex> lock(g_i_mutex);
    ++g_i;
}
```

- `g_i_mutex` is automatically released when lock goes out of scope

Safe locking of two mutexes

- `std::lock` can lock more than one mutex, `std::adopt_lock` „adopts“ the already locked mutex

```
template<class T> class X
{
private:
    std::mutex m;
    T data;
public:
    friend void swap(X& lhs, X& rhs)
    {
        if (&lhs == &rhs) return;
        std::lock(lhs.m, rhs.m);
        std::lock_guard<std::mutex> lock_a(lhs.m, std::adopt_lock);
        std::lock_guard<std::mutex> lock_b(rhs.m, std::adopt_lock);
        swap(lhs.data, rhs.data);
    }
};
```