Concepts and Models of Parallel and Data-centric Programming

Shared Memory IV

Lecture, Summer 2020

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

i12 High Performance Computing | RWTH AACHEN UNIVERSITY

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 II: Challenges)

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

```
1  template<class T> class NaiveQueue
2  {
3  protected:
4     T* items; int head, tail;
5  public:
6     NaiveQueue() { ... initialization ... }
7     void enq(T element) { ... enqueue element: tail++ % QSIZE ... }
8     T deq() { ... return element: head++ % QSIZE ... }
9  };
```

Anything else?

- What may go wrong?

  - What if
    - Enqueuer finds a full array?
    - Dequeuer finds an empty array?
  - Wait for something to happen

Condition Variable

Motivation

- A thread could “actively“ wait for an event to happen

```
 1  bool flag;
 2  std::mutex m;
 3  void wait()
 4  {
 5     std::unique_lock<std::mutex> lk(m);
 6     while (!flag)
 7     {
 8        lk.unlock();
 9        std::this_thread::sleep_for(std::chrono::milliseconds(100));
10        lk.lock();
11     }
12  }
```

- What is “tricky“ here?

Motivation

- A thread could “actively“ wait for an event to happen

```
 1  bool flag;
 2  std::mutex m;
 3  void wait()
 4  {
 5     std::unique_lock<std::mutex> lk(m);
 6     while (!flag)
 7     {
 8        lk.unlock();
 9        std::this_thread::sleep_for(std::chrono::milliseconds(100));
10        lk.lock();
11     }
12  }
```

Unlock

Lock (again)

- What is “tricky“ here?

Condition / 1

- A condition variable implements notification
  - Block one (or multiple) thread(s) until another thread modified the condition variable and performs the notification

- Waiting thread(s):
  - Acquire the mutex (must be `std::unique_lock`)
  - Execute `wait()`, `wait_for()` or `wait_until()` (releases the mutex)
  - Wakeup occurs on the condition or after timeout, mutex is re-acquired

- Notifying thread:
  - Acquire the mutex () (usually via `std::lock_guard`)
  - Perform the modification, release the mutex
  - Execute `notify_one()` or `notify_all()`

Condition / 2

- Class `std::condition_variable`
  - Representation of a condition variable for “system threads”
  - Defined in header `<condition_variable>`
  - Reference: https://en.cppreference.com/w/cpp/thread/condition_variable

- Fundamentally, a condition variable is just an optimization over a busy-wait
  - Any code should work with a sub-optimal implementation
    - Repeated `unlock()` – `lock()` sequence

Example: Queue (part III: Implementation)

Thread-safe Queue / 1

- Thread-safe queue make use of `std::queue` internally
- Allows two (or more) threads to call `push()` and `wait_and_pop()` simultaneously
- Mutex, Condition variable, and implementation details are all hidden from the „user“

```
 1  template<typename T> class threadsafe_queue
 2  {
 3  private:
 4     std::queue<T> data;
 5     std::mutex mut;
 6     std::condition_variable cond;
 7  public:
 8     threadsafe_queue() {}
 9
10  // continued on next slide
```

Thread-safe Queue / 2

```
 1  public:
 2     threadsafe_queue(threadsafe_queue const& other)
 3     {
 4        std::lock_guard<std::mutex> lk(other.mut);
 5        data = other.data;
 6     }
 7     void push(T new_val)
 8     {
 9        std::lock_guard<std::mutex> lk(mut);
10        data.push(new_value);
11        cond.notify_one();
12     }
13     void wait_and_pop(T& value)
14     {
15        std::unique_lock<std::mutex> lk(mut);
16        cond.wait(lk, [this]{return ! data.empty() ;} );
17        value = data.front();
18        data.pop();
19     }
```

What you have learnt

Correct wording & Definitions / 1

- Critical Region:
  - A **section** of code that may only be executed by one process at any one time.

- Mutual Exclusion:
  - A **property**, the requirement that one thread of execution never enter its critical section at the same time that another concurrent thread of execution enters its own critical section.

- Starvation:
  - The **problem** encountered in concurrent computing where a process / thread is perpetually denied necessary resources to process its work.

Correct wording & Definitions / 2

- Deadlock:
  - A **program state** in which each member of a group (here: each thread) is waiting for some other member to take action.

- Race Condition:
  - Any situation where the outcome depends on the relative ordering of execution of operations on two or more threads

- Data Race: happens when there are two memory accesses in a program where both:
  - target the same location
  - are performed concurrently by two threads
  - are not reads
  - are not synchronization operations