Today: Multithreading Part I and template SST case-study

Multithreading
• std::thread class
• impact of race conditions
• cheap synchronization with std::atomic
• general synchronization with std::mutex
• revisiting wc++

Using templates: A case-study
• SST table design
• Dynamic memory layout with runtime-fixed length column-arrays
• Templated column design
• Figuring out column base addresses and row length using variadic templates
Multithreading

• Threads give us parallelism
• Threads need to coordinate – think of a group discussion
• A single thread starts execution from main
• Can spawn a thread by creating an object of `std::thread`
• When using multiple threads, link your program against the dynamic library `pthread` on Linux
C++ thread support library: std::thread class

Constructors

• template< class Function, class... Args >
  explicit thread( Function&& f, Args&&... args );

• thread() noexcept;

Important member functions

• void join();

• void detach();
Using `std::thread`: My word count example

```cpp
// start all threads and wait for them to finish
std::vector<std::thread> workers;

for(uint32_t i = 0; i < num_threads; ++i) {
    // each thread executes sweep which takes no arguments
    workers.push_back(std::thread(sweep));
}

// waits for each thread to finish
for(auto& worker : workers) {
    worker.join();
}
```
What can go wrong without synchronization?

**Example:** Concurrent increments of a shared integer variable
Each thread
• shares an integer called *count* initialized to 0
• increments it 1 million times concurrently without any synchronization (no optimizations)
What can go wrong without synchronization?

**Example**: Concurrent increments of a shared integer variable

Each thread

- shares an integer called `count` initialized to 0
- increments it 1 million times concurrently without any synchronization (no optimizations)

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>Final value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000000</td>
</tr>
<tr>
<td>2</td>
<td>1059696</td>
</tr>
<tr>
<td>3</td>
<td>1155035</td>
</tr>
<tr>
<td>4</td>
<td>1369165</td>
</tr>
</tbody>
</table>
What can go wrong without synchronization?

• Accessing more complex data structures concurrently will most likely result in segmentation fault
• C++ standard library containers (vector, map, list...) are not thread safe (Why?)
• Example: Concurrent vector updates crash the program
Cheap synchronization with std::atomic<>

- Only available for select data-types: `int`, `bool`, `float` and their variants
- Guarantees atomic access to the variable and atomicity of certain operations in the presence of multiple threads
  - `store`: atomically replaces the value with a non-atomic argument
  - `load`: atomically obtains the value of the atomic object
  - `exchange`: atomically replaces the value with the provided value and returns the old value
  - `operators`: `+= (fetch_add), -= (fetch_sub), ++, --, &, |, ^=`
Cheap synchronization with std::atomic<>

• Not all architectures provide atomic loads and stores of integer variables – even then other operations will not be atomic (increment etc.)

• Demo on https://godbolt.org
  • Different assembly code generated for ARM64 gcc 6.3.0 (linux) and x86-64 gcc 10.2

• Uses intel’s lock signal prefix on x86-64

• Uses instructions such as ldaxr (load-acquire exclusive register) and stlxr (store-release exclusive register) on ARM

• For more about atomics in ARM, read https://stackoverflow.com/questions/11894059/atomic-operations-in-arm
Sequential consistency with std::atomic<>

• `x = x + 7;` is not an atomic operation even if `x` is an atomic integer, but `x += 7;` is!

• `std::atomic<>` guarantees sequential consistency (total global ordering) between all atomic operations

• You can relax the synchronization guarantees with `std::memory_order` ([https://en.cppreference.com/w/cpp/atomic/memory_order](https://en.cppreference.com/w/cpp/atomic/memory_order))

• For example,
  ```
  std::atomic<long> value {0};
  value.fetch_add(1, std::memory_order_relaxed);
  value.fetch_add(5, std::memory_order_release);
  ```

• For more info, read [https://stackoverflow.com/questions/31978324/what-exactly-is-stdatomic](https://stackoverflow.com/questions/31978324/what-exactly-is-stdatomic)
Is `std::atomic<>` enough for all synchronization requirements?
Is std::atomic<> enough for all synchronization requirements?

NO

• Only few primitive types can be atomic
• std::atomic<> applies to just one variable

Example:
• Two integers account1 and account2
• Function transfer: account1 += bal; account2 -= bal;
• Function audit: account1 + account2
Critical section and mutual exclusion

• Instead of thinking about which variables or operations should be atomic, protect areas of code where they are accessed

• Critical section: A segment of the code that only one thread can access at a time

• We want mutual exclusion – No two threads access a critical section at the same time

• In C++, we guarantee mutual exclusion using an std::mutex object
Some important concurrency concepts

• Race condition: When two threads access a critical section at the same time
• Deadlock: When no thread can make any progress
• Livelock: Threads seem to make progress (release/acquire mutexes), but are actually still stuck

• Read
  • https://stackoverflow.com/questions/34510/what-is-a-race-condition
  • https://stackoverflow.com/questions/34512/what-is-a-deadlock
Synchronization in C++: std::mutex class

- **void** lock(); – Locks the mutex if it’s available, blocks otherwise
- **void** unlock(); – Unlocks the mutex if locked by the current thread, otherwise undefined behavior
- **bool** try_lock(); – Non-blocking version of lock, returns false if the mutex is already locked
How to use an std::mutex

• Avoid locking and unlocking directly
  • What if you forget to unlock?
  • What if the thread throws an exception while holding the mutex?
  • Same issues as with releasing memory held by pointers
• Use std::unique_lock<std::mutex> or std::scoped_lock<std::mutex>
  • RAII implementations – guaranteed to release the mutex at destruction
  • Use std::scoped_lock if you never need to release the mutex manually
• Can you answer now: Why are C++ standard containers not thread-safe?
Revisiting my word-count program \texttt{wc++}

• Class wordCounter. Public functions:
  • \texttt{wordCounter(const std::string& dir, uint32_t num_threads)};
  • \texttt{void compute();}
  • \texttt{void display();}

• main thread simply initializes an object of wordCounter and calls compute and display on it
  \begin{verbatim}
  wc::wordCounter word_counter(argv[1], std::stoi(argv[2]));
  word_counter.compute();
  word_counter.display();
  \end{verbatim}
Implementation of `wordCounter::compute`

- Calls helper function `find_all_files(dir, pred)` to gather all `.c` and `.h` file paths
- Spawns the worker threads and waits for all of them to finish
- Each worker executes the sweep function
- Worker threads use an `std::atomic<uint64_t>` variable to get a unique file index
  ```cpp
  uint64_t file_index;
  while((file_index = global_index++) < files_to_sweep.size()) {
    process_file(files_to_sweep[file_index], local_freq);
  }
  ```
- Each thread stores the result in a local map, updates the global map at the end using a mutex
  ```cpp
  std::lock_guard<std::mutex> lock(wc_mtx);
  for(auto [word, cnt] : local_freq) {
    freq[word] += cnt;
  }
  ```
Next time: Exploring various trade-offs in *wc++*

• How much time is spent in various stages (collecting all the files, computing in parallel, sorting results at the end, printing the results)?
• What is the impact on performance with increasing no. of threads?
• What happens if everyone directly modifies `global_freq` instead of maintaining a `local_freq` object?
• What if we use a mutex for finding the next file to process instead of an `std::atomic<uint64_t>`?
• What is the overhead of parsing the files word-by-word instead of all at once?
• What if we divide the files equally among the workers at the start?
Part II: **SST** – A case-study of using templates

- Important data structure used in our distributed systems research
- SST (Shared State Table) is a table (think: a database table) consisting of state variables as the columns and multiple rows

<table>
<thead>
<tr>
<th>Suspected</th>
<th>Proposal</th>
<th>nCommit</th>
<th>Acked</th>
<th>nReceived</th>
<th>Wedged</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Suspected</th>
<th>Proposal</th>
<th>nCommit</th>
<th>Acked</th>
<th>nReceived</th>
<th>Wedged</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Q</td>
<td>4:</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Q</td>
<td>F</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>R</td>
<td>F</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table:**
- **Suspected:** P, Q, R
- **Proposal:** 4: Q
- **nCommit:** 3
- **Acked:** 4
- **nReceived:** 5
- **Wedged:** T, F
SST History

- Conceptualized by Ken for use in RDMA environments
- Version 1 implemented by me in Fall 2015
- Version 2 designed by Matthew Milano in 2017
- Version 2 implemented in 2017 and maintained by me since then

Aside:
- SQL is not type-safe
- What if the column name in the search query is invalid?
Basic requirements

• The user should be able to specify the table layout
• Each row’s data should be stored *contiguously* in memory

First design:
• User defines the row layout as a struct, myRow
  ```
  class myRow {
    int id;
    bool processing;
    int msgs_count;
  }
  ```
• SST is then templated on myRow, contains a vector of myRow objects
• Access row 3’s msgs_count with `sst_obj[3].msgs_count`
Main requirement 1: Support column vectors with fixed runtime size

• For example, a column vector called suspected consisting of 3 columns, suspected[0], suspected[1] and suspected[2]

• Can’t use native array members in myRow

```java
class myRow {
    int id;
    bool processing;
    bool suspected[max_size];
    int msgs_count;
}
```

• max_size must be known at compile time, which is a limitation

• Using a vector of bool in myRow will not store the data contiguously
New design: Allocate memory for the rows at runtime

- When a table entry is accessed using [], do memory translations to find where the entry is stored
- Store all rows contiguously (number of rows fixed after construction)
- Two new classes: SSTField<T> and SSTFieldVector<T>
- Both derive from the common class _SSTField

<table>
<thead>
<tr>
<th>id, base_address = 0</th>
<th>s[0], addr=4</th>
<th>s[1], addr=8</th>
<th>s[2], addr=12</th>
<th>status, base_address = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Access table.id[0] at address 0, table.id[1] at address row_length
- Access table.s[1][1] at address 4 + 1 * 4 + 1 * row_length
User interface

- User defined a mySST class that inherits from the common SST class
- Specifies columns as a sequence of SSTField<T> and SSTFieldVector<T> objects
- Supplies the size of SSTFieldVector<T> objects in the constructor

```cpp
class mySST : public SST {
    SSTFieldVector<message_id_t> seq_num;
    SSTField<int32_t> vid;
    SSTFieldVector<bool> suspected;
    SSTField<int> num_changes;

public:
    mySST(const uint32_t num_rows,
           const uint32_t seq_num_size,
           const uint32_t suspected_size)
        : SST(num_rows),
          seq_num(seq_num_size),
          suspected(suspected_size) {
    }
};
```
Implementation of _SSTField

- Contains the base address of the column
- Contains the length of the field

```cpp
class _SSTField {
public:
    volatile char* base;
    size_t row_length;
    size_t field_length;
    uint32_t num_rows;

    _SSTField(const size_t field_length);

    const char* get_base_address();
};
```
Implementation of SSTField<T>

- Models a single column
- Passed the size of T to _SSTField

```
template<typename T>
class SSTField : public _SSTField {
public:
    using _SSTField::base;
    using _SSTField::field_length;
    using _SSTField::row_length;

    SSTField() : _SSTField(sizeof(T)) {
    }

    // Tracks down the appropriate row
    volatile T& operator[](const size_t row_index) const;
};
```
Implementation of SSTField<T>

- Models a vector of columns
- Passed the number of columns * size of T to _SSTField
Issue: How to find base address and row length?

• The base address of a field depends on the number of fields to its left
• The length of the row depends on the table layout, thus requires knowledge of all the columns

Solution: Use variadic templates!

• Leave out setting base addresses when the SST fields are constructed
• Ask the user to call a function SST::initialize_fields passing all the columns in the constructor of mySST. E.g. initialize_fields(seq_num, vid, suspected, num_changes).
• Compute and set the row length and base addresses in this function using variadic templates
Implementation of SST::initialize_fields<T...>

```cpp
template <typename... Fields>
void SST::initialize_fields(Fields&... fields) {
    compute_row_length(fields...);
    rows = std::make_unique<volatile char[]>(row_length * members.num_nodes);
    volatile char* base = rows.get();
    set_field_params(base, fields...);
}
```
Impl. of SST::compute_row_length<T...>
Impl. of SST::set_field_params<T...>
Now all accesses are well-defined

- User can write
  ```cpp
  MySST sst(num_rows, seq_num_size, suspected_size);
  ```
- And access the fields with
  ```cpp
  sst.vid[0] or sst.suspected[2][3]
  ```