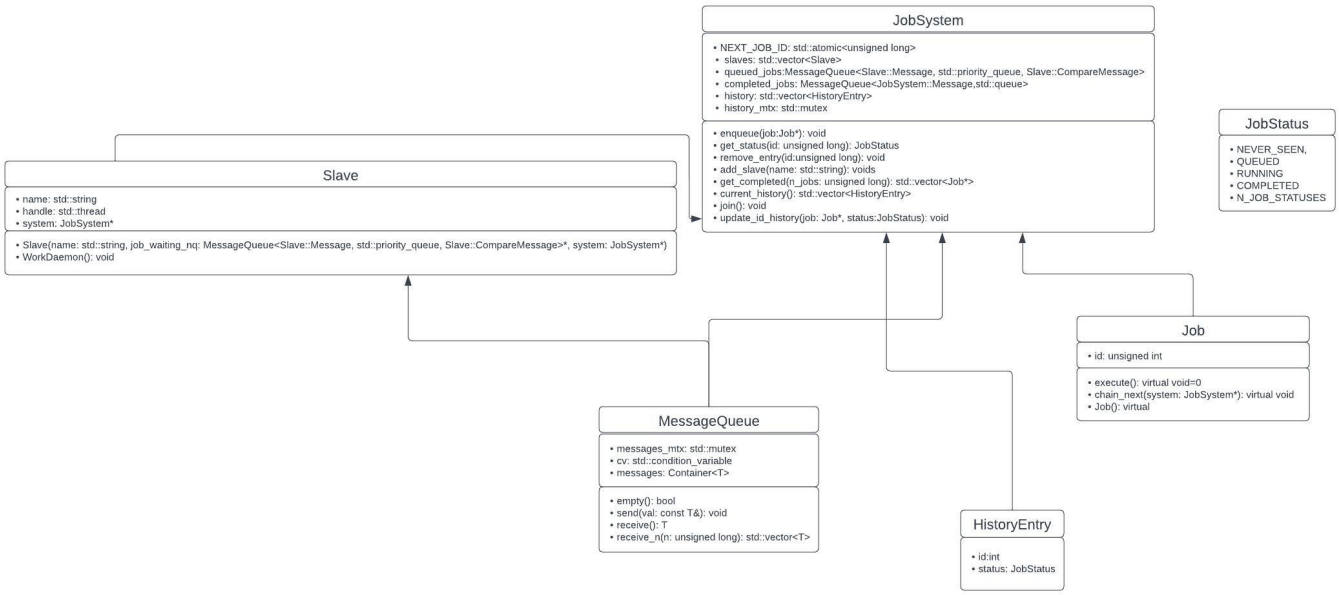
Report

Memory Management and Thread Protection

- Guards
    - I used scope-based lock guards or unique locks instead of relying on raw mutexes. Utilizing these mechanisms offers a valuable advantage: in case of an exception, the locking process takes place automatically when the guard reaches the end of its scope or when explicitly unlocked. This effectively diminishes the likelihood of unintentional deadlocks. While this approach does introduce some additional computational overhead, it remains sufficiently modest to warrant its adoption for enhanced safety considerations.

- Message Passing
    - I also a message passing system that relies on kernel interrupts rather than resorting to busy-waiting. The practice of spinning in userspace is generally discouraged due to its potential to contend for CPU resources with the operating system or introduce undesirable delays. This problem is less pronounced with a smaller number of threads but becomes increasingly impactful as the number of threads significantly surpasses the count of physical CPU cores. To mitigate this issue, I've incorporated a MessageQueue to facilitate data transfer between the Master thread and its Slave threads. This allows threads to efficiently enter a sleep state while awaiting tasks, as opposed to repeatedly waking up at intervals to check for work. Overall, this approach offers several advantages, although it may lead to performance slowdowns in cases where transitioning from kernel mode to user mode incurs significant overhead.

UML:



Job System

Using the JobSystem requires initialization. The JobSystem needs to be initialized first and then the Slaves need to be initialized. This can be achieved by utilizing the 'system.addSlave("thread" + std::to_string(n));' function. In contrast, I/O-bound code may benefit from having more threads than physical cores.
Memory allocation is necessary for a job. To create a new job, you can use the following code: 'MakeJob *mj = new MakeJob(0, "demo");'. The arguments for this particular job include the job's ID and the target make.
To execute the job, you need to enqueue it within the system using 'system.enqueue(mj)'. Once the Job* has been introduced into the system, it will be dispatched to the Slaves in a first-in-first-out (FIFO) manner.
To get completed jobs using 'js.getCompleted(n)', the execution will be paused until 'n' jobs become available and are subsequently returned.

Demonstration

JobSystem is implemented when the names from the makefile are given as arguments.

The Makefile contains the following:

```
compile: ./Code/*.cpp
   clang++ -pipe -Wall -std=c++17 ./Code/*.cpp -o system
```

When make is called, the output will a JSON object for each given target. The error output is as follows:

```
clang++ -pipe -Wall ./Code/*.cpp -o
./Code/main.cpp:2:1: error: unknown type name 'string'; did you mean
'std::string'?
string dsa() {
^~~~~~
std::string
/Library/Developer/CommandLineTools/usr/include/c++/v1/string:61:32: note:
'std::string' declared here
typedef basic_string<char> string;
^ 1 error generated.
make: *** [Makefile:4: makefile] Error 1
```

Parsed JSON Output:


```
"./Code/main.cpp": [
      {
      "chunk": ["#include <iostream>", "string dsa() {"," let d = {"," das
      auto;"],
      "column": 1,
      "line": 2,
      },
}
```


With the parsed JSON output, only 2 lines with the error are given.