Multithreading Job System Report

0.1 Memory Management and Thread Protection Paradigms

The architecture of the job system is similar to the one detailed in class, however it makes several different design decisions to reduce complexity.

0.1.1 Guards over raw mutexes

One of the first changes is by using scope-based `lock_guard`s or `unique_lock`s, rather than raw mutexes. The benefit to using these is that in the event of an exception, locking will occur automatically when the guard goes out of scope (or when explicitly unlocked), reducing the chances of unintended deadlocks. This approach adds some overhead, but it is minimal enough to justify the increased safety.

0.1.2 Efficient Message Passing

One of the other notable changes is the use of a message passing system backed by kernel interrupts, rather than busy-waiting. In general, spinning in userspace is a bad idea, as it may compete for CPU time with the OS, or introduce delays. This issue is less noticeable with fewer threads but becomes more impactful when the number of threads significantly exceeds the number of physical cores. To avoid this, a `MessageQueue` is used for passing data between the Master thread and its Slave threads. This allows threads to efficiently sleep while waiting for work, rather than waking up at an interval to check for work. In general, this solution is advantageous, but may cause slowdowns when context-switching from kernelmode to usermode is expensive.

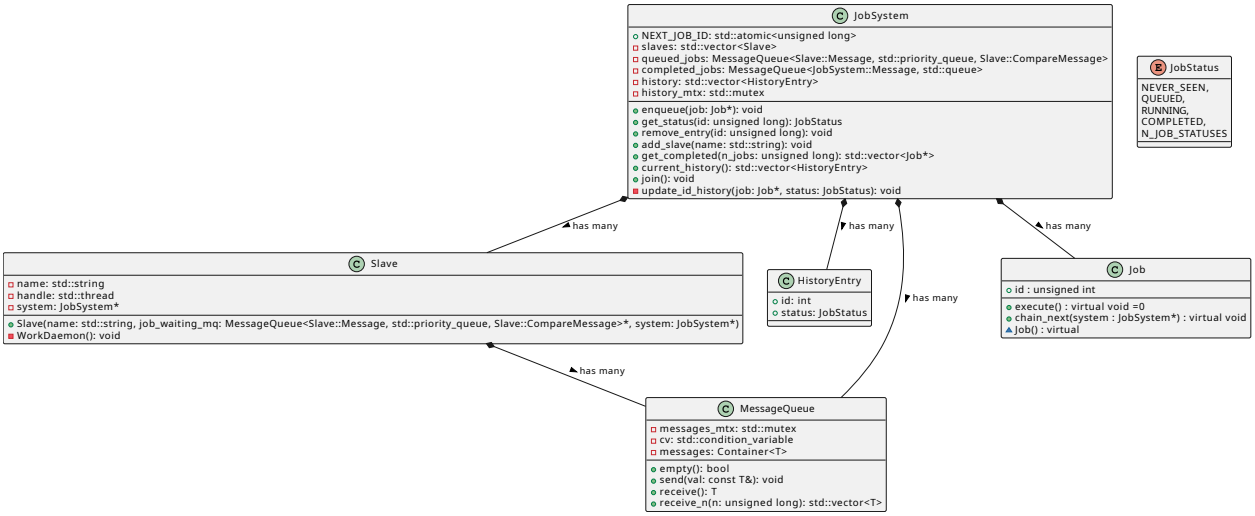
0.1.3 No public system-scoped memory

The `JobSystem` itself is designed such that it will not carry any publicly accessible memory. The programmer is responsible for managing all publicly accessible memory, including memory allocated by the jobs. All jobs that are handled by the system will have memory that may outlive the scope of the `JobSystem` and can be freed by the programmer at his free-will. In effect, this reduces complexity within the system, however it adds it as an additional burden on the programmer.

0.1.4 Job chaining

While circular dependencies are generally regarded as an antipattern, it is applicable in this scenario. Because make jobs and parsing jobs always lead to a single outcome (to generate JSON output of compiler messages), it is acceptable to allow jobs to automatically call/chain other jobs upon completion. This allows the inclusion of support for different compilers or parsing mechanisms without needing to rewrite or handle cases for function along the job pipeline. When a make-job (in this case, the root job) is called, it is always expected to end with a JSON-job output. Additionally, this also relieves the main thread of additional job management.

0.2 UML Overview



0.3 Usage of `JobSystem`

0.3.1 Preamble

Using the JobSystem is straightforward, however it will require bookkeeping in terms of memory usage of jobs.

0.3.2 Intializing the `JobSystem`

First, the programmer must initialize the `JobSystem`. Here we will call it `JobSystem system;`

0.3.3 Initializing `Slaves`

Additionally, in order to do work, `Slaves` must be initialized to do work. This can be done with `system.add_slave("thread" + std::to_string(n));` This function may be called multiple times, however the appropriate amount varies based on the nature of the work being done. Compute-bound, branchless code may prefer to limit the number of `Slavess` to the number of physical cores, whereas I/O bound code may opt for many more threads than physical cores.

0.3.4 `Job` Allocation

Next, memory must be allocated for a job. Note: this memory, must outlive the lifetime of the JobSystem. Not doing so will result in undefined behavior. Next we will create a new MakeJob using `MakeJob *mj = new MakeJob(0, "demo");`. The arguments for this specific job are the `id` and the make `target`.

0.3.5 Submitting `Jobs` into the `JobSystem`

Next, to execute the job, we will enqueue it into the system using `system.enqueue(mj)`. The `enqueue()` function will accept any pointer, which inherits from the `Job` API. Once the `Job*` has entered the system, it will be sent to the `Slaves` in a FIFO manner, (however it is not guaranteed to be completed in a FIFO manner).

0.3.6 Receiving completed `Jobs` from the `JobSystem`

The `JobSystem` offers a basic interface for recieving completed jobs. `js.get_completed(n);` will block execution until `n` jobs are available to be recieved, and returned. Note: there is no logical checking that is done by the function itself. It is up to the programmer to determine whether it is possible for `n` jobs to be available, otherwise the program may hang if an `n` that is too large.

0.4 `JobSystem` Function Information

0.4.1 void enqueue(Job *job)

The enqueue function of the job system accepts a pointer to a job. The system does not take any preventative measures to ensure that the Job* is not modified during execution. Modification of the Job* is up to the programmer's discretion.

0.4.2 JobStatus get_status(unsigned long id) const

This function will query the job system of all jobs that have been enqueued (either directly by the programmer, or indirectly by other jobs). Querying for a job with an id that is in the system multiple times will lead to undefined behavior.

0.4.3 void remove_entry(unsigned long id)

This function will query the job system's current job history and remove the `HistoryEntry` with the associated Job ID. Similar to get_status, this function leads to undefined behavior if there are multiple history entries with the same ID in the system.

0.4.4 void add_slave(std::string name)

This function will spawn a new slave thread, which may execute work, on the job system. It may be assigned a name, however it is only used for debugging purposes within the system and its information cannot be passed onto the job itself.

0.4.5 std::vector¡Job *¿ get_completed(unsigned long n_jobs)

This function will block the thread it was called on and await for `n_jobs` to be available from any of the slave threads. This function does not perform any checks to ensure the number of jobs to be returned is valid, nor does it gurantee any ordering of the returned jobs. This is at the discretion of the programmer.

0.4.6 std::vector¡HistoryEntry¿ current_history() const

This function returns a stale copy of the current job history of the job system.

0.4.7 void join()

This function should always be called when all the desired jobs have been enqueued. This will block execution on the thread it was called from until all the jobs have been completed. It is undefined behavior to continue sending jobs to the system after `join()` has been called.

0.5 Demonstration of `JobSystem`

0.5.1 Overview

The `JobSystem` implementation being demonstrated is being used to build targets from a Makefile in parallel. It is provided arguments, which are the names of the targets from a Makefile in the current directory. When the `JobSystem` is built into a binary, say named `system`.

0.5.2 Makefile targets

Given a Makefile with the contents:

```
demo: ./Code/Demo/main.cpp
clang++ -pipe -Wall ./Code/Demo/*.cpp -o demo
```

0.5.3 Internal operations

When running `./system demo`, the system will internally call the makefile and call parsing and JSON generating jobs after. The JSON output is held in a queue until all make targets are completed. Since there is only a single target, it will be printed to stdout immediately.

0.5.4 Output

The output of the program will be a single line JSON object for each built target. Errors and warnings are not differentiated in JSON format, unless it is explicitly noted by the compiler. For example, the following error output:

```
clang++ -pipe -Wall ./Code/Demo/*.cpp -o demo
./Code/Demo/main.cpp:2:1: error: unknown type name 'string'; did you mean 'std::string'?
string dsa() {
^~~~~~
std::string
/usr/bin/../lib64/gcc/x86_64-pc-linux-gnu/13.2.1/../../../../include/c++/13.2.1/bits/stringfwd.h:77:33:
  typedef basic_string<char>    string;
                                ^
1 error generated.

make: *** [Makefile:5: demo] Error 1
```

The parsed JSON will be:

```
{
  "./Code/Demo/main.cpp": [
    {
      "chunk": [
        "#include <iostream>",
        "string dsa() {",
        "  let d = {",
        "    das auto;"
      ],
      "column": 1,
      "line": 2,
      "message": "unknown type name 'string'; did you mean 'std::string'?"
    },
}
```

In the parsed output, at most, 2 lines surrounding the line containing the error are provided, along with information about the line and column # where the error occurred, as well as the error message emitted by the clang compiler.
In events where no errors are emitted by the compiler, the program will simply return a null JSON value.