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 is by using `lock_guard`s or `unique_lock`s, when necessary. The benefit to using these is that in the event of an exception, locking will occuring automatically when the locks go out of scope (or when explicitly unlocked), reducing the chances of unintended deadlocks. These types of mutex come at the disadvantage that they provide some overhead, however, it is small enough that it is work the additional safety that they provide.

0.1.2 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. To avoid this, a `MessageQueue` system 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 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. All memory that is publicly accessible, including memory allocated by jobs themselves, is managed under the responsibility of the programmer. 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. The effect of this is that complexity that might've existed within the system is now a burden to the programmer.

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, `Slave`s 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 of vary based on the nature of the work being done. Compute-bound, branchless code may prefer to limit the number of `Slaves`s 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, we will `enqueue` the job into the system for it to be executed, 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 guranteed to be completed in a FIFO manner).

0.4 Demonstration of `JobSystem`