Kendall Boesch
48231304
Lab 1

Lab I: Multithreaded Job System

I. Introduction

This program is made up of a JobSystem, Compile, Parse, Render, and JSON jobs, JobWorkerThreads, and two structs–JobHistoryEntry and Error. This program is a general-purpose multithreaded job system, capable of compiling and executing C++ code. Additionally, the program outputs a detailed report of any errors and warnings from the compiling and linking process in a JSON file. The system interface allows for thread, system, and Job creation and destruction. Additionally it allows users to track jobs throughout the system and provide details and history on job execution/status.

II. Memory Management & Thread Protection Paradigms

This job system utilizes mutex locks to protect the critical sections of various functions. For example, the JobSystem mutex named m_jobsQueuedMutex is responsible for ensuring only one thread can access and manipulate the m_jobsQueued deque at a time. Without the use of this mutex, it is possible that a race condition would occur. This means that two threads would access the memory location and receive the same information when, in reality, the contents of the memory address being accessed has changed–maybe a new Job was queued that is not reflected, or a job was claimed and no longer exists in the m_jobsQueued vector.

I chose to use mutexes over other thread protection paradigms like semaphores because they ensure mutual exclusion. When using a mutex, if a thread tries to access a protected resource that is already being accessed by a separate thread, the mutex can block the attempting thread. Ultimately mutexes allow only one thread at a time to access the protected resource, ensuring exclusive access and preventing data races.

Another reason I chose to work with mutexes was due to their simplicity and ease of use. That being said, in utilizing mutexes I also had to deal with a few cases of gridlock during the development and debugging stages of this lab. Had I used atomics, I would not have had to worry about gridlock as atomics do not involve blocking. Additionally, with the use of mutexes it was important that I had separate deque objects for queued, running, and completed jobs. Had I kept one job queue and only changed individual job statuses, the program would have been extremely inefficient as only one thread would be able to access the single queue at a time.

III. Usage – From Main & Under the Hood

- **Initialize the JobSystem** – The first step to this program is initializing the JobSystem for use. To initialize the JobSystem, the function JobSystem::createOrGet() is called. Before creating a new JobSystem instance, the createOrGet function checks to see if there is already an initialized JobSystem. If there is not, the function calls the JobSystem constructor and returns a pointer to the new system. If there is already an instance of the system, a pointer to it will be returned. Ultimately this function ensures there is only one instance of the JobSystem. (Singleton design pattern).

- **Create JobWorkerThreads** – The JobSystem calls the function JobSystem::createWorkerThread which takes in a unique name for the thread (const char*) and a worker job channel (unsigned long) as parameters. Within this function, the system creates a new worker thread passing it the same parameters, as well as a pointer to itself. From there, mutex locks are used to ensure no other thread can access the deque object m_workerThreads before the new thread is inserted. After the insertion, the mutex is unlocked and the JobWorkerThread::startUp() function is called on the new thread.
    - **JobWorkerThread::startUp()** – Next, in the newly created JobWorkerThread object, the startUp() function creates the actual std::thread object for execution.
    - **JobWorkerThread::workerThreadMain(void *workerThreadObject)** – The workerThreadMain function then creates a pointer to the thread object passed into it and calls work() on it.
    - **JobWorkerThread::work()** – While the worker thread is not stopping, the work() function gets the job channel from the thread and uses JobSystem::claimJob() to attempt to grab and execute a job. If there is a Job grabbed, the function then calls execute() on the Job.
    - **JobSystem::claimJob(m_workerJobChannels)** – Taking in the job channel on which to search for an unclaimed job, this function iterates through the queued jobs. With each job, the function then performs a bitwise AND operation between m_jobChannels and the job channel assigned to the Job. If the result does not equal 0, the job is then erased from the m_jobsQueued deque and pushed back onto the m_jobsRunning deque. Finally, the function updates the claimed job status to JOB_STATUS_RUNNING and returns the job as a pointer.
- **Create CompileJobs** – Desired CompileJobs are passed to the program as command line arguments. If I wanted to run jobs for automated.cpp, demoError.cpp, and demoWorkingJob.cpp I would simply enter “make compile ARGS=”automated demoError demoWorkingJob” “ into the command line. In main.cpp, a count of how many arguments were passed as well as an array of the arguments passed are passed to the main function as parameters. Main then iterates through the arguments and, for each argument, creates a new CompileJob instance, sets the target attribute to the argument passed, pushes back the new job onto a vector<Job*> jobs, and pushes the job ID of the new job onto a vector<int> jobIds by calling the JobSystem::getUniqueID() function.
    - **JobSystem::getUniqueID(Job*)** – This function takes in a pointer to the job in question and returns an int which holds the unique ID for that Job
- **Queue Jobs –** Next, still in the main function of main.cpp, the vector<Job> jobs is then iterated through. At each job, the system then calls JobSystem::queueJob(Job*) to insert each job into the system.
    - **JobSystem::queueJob(Job*) –** This function first creates a JobHistoryEntry for the job, changing its status to JOB_STATUS_QUEUED, then places the entry in the m_jobHistory deque object. Next, the function pushes the newly queued job to the back of the m_jobsQueued deque to be claimed by a JobWorkerThread. (See bullet above “**JobSystem::claimJob(m_workerJobChannels)**”)
- **Finish Jobs** – Finally, in main.cpp, the vector<int> jobIDs is then iterated through. For each ID in the vector, the system then calls JobSystem::FinishJob(int) function to finish job execution

- **JobSystem::finishJob(int)** – This function gets passed an int holding the job ID of the job to be finished. The function calls JobSystem::isJobComplete(int), passing it the ID to see if the job is still running. While the job is not complete, the function checks to ensure the job status is neither JOB_STATUS_NEVER_SEEN nor JOB_STATUS_RETIRED. If the job status is one of the two, the function will print to the console that it is waiting for the job, but there is no job with that ID in the system and return. If the job is complete, the function then iterates through m_jobsComplete to find the job that was just completed by comparing IDs. Once found, the job is saved to a job pointer thisCompletedJob before it is erased from m_jobsCompleted. Next the function checks to ensure that a job was assigned to thisCompletedJob by comparing it to a nullptr. If the thisCompletedJob is a null pointer, the function prints to console that the job was status completed, but not found. If this is not the case, the function checks to see what the job type is, so it knows how to proceed accordingly. First, it checks if both the job type is 1 (indicating a CompileJob) and the attribute compResults is not an empty string. If both conditions are true, the job just completed is a CompileJob that produced errors or warnings, indicating there is more work to do. In this case, the function creates a new ParseJob and assigns its unparsedText attribute the compilation results of the CompileJob just completed. The new ParseJob is then queued and assigned to the completed CompileJob as childJob. If the conditions for this case fail, however, the function checks if the job just completed is job type 2, indicating it was a ParseJob. If this is the case, the function creates a new JSONJob and assigns the new JSONJob's errorMap with the errorMap from the ParseJob just completed. From there, the JSONJob is queued and assigned as the childJob for the ParseJob just completed. If neither of these cases are true, this means the job that was passed to the finishJob function was either a CompileJob that executed without errors or warnings, or a JSONJob. Regardless of the job type, the finishJob function then calls jobCompleteCallback with is a virtual function overwritten in the different Job's .cpp files. If the function is not overwritten in one of the job type's .cpp file, execution of finishJob just continues. Next, the job status of thisCompletedJob is set to JOB_STATUS_RETIRED. Then, the function checks to see if thisCompletedJob has a child job. If it does, finishJob is then called on the child job. Lastly, thisCompletedJob is deleted.
    - **JobSystem::isJobComplete(int) const –** This function takes in an int which holds a job ID. The function then gets the status of the requested Job and returns true if the Job status is JOB_STATUS_COMPLETED and false otherwise
    - **JobSystem::getJobStatus(int) –** This function takes in an int which holds a job ID. The function then gets the status of the job passed in and returns it as an int.

IV. Job Types

- **Job.h – Abstract Class**
  - **Std::string compResults:** for result of the attempt to compile the requested .cpp file for CompileJob. Referenced outside of CompileJob.cpp, so need in abstract class
  - **std::map<std::string, std::vector<Error>> errorMap:** Map with key is file that produced compile warnings/errors and value as std::vector<Error> for ParseJob & JSONJob. Accessed outside of .cpps, so added to JOb.h
  - **Virtual void execute() –** Virtual function overwritten by different job types implementation
  - **Virtual void jobCompleteCallback() –** Function for post job completion. Overwritten by different job types implementation
  - **Int getUniqueID() const** – returns unique job id
  - **Int m_jobID:** Unique job ID
  - **Int m_jobType:** integer representing jobType. 0 = RenderJob, 1 = CompileJob, 2 = ParseJob, 3 = JSONJob
  - **Unsigned long m_jobChannels:** Channel on which to add the job
  - **Job* childJob:** Pointer to a job that was created from the run of the current job
- **CompileJob**
  - **Int jobType** = 1
  - **Std::string target**: makefile command to compile the requested .cpp file
  - **Std::string compResults**: results of the attempt to compile the requested .cp
  - **Int returnCode**: the code returned by the Jobs attempt to compile the requested .cpp file
  - **CompileJob::execute() override** – this function takes the CompileJob target and makes it a compile command by appending it to a string command which holds “make “ and then appending “ 2>&1” to the end of the concatenation to redirect cerr to cout. Next, the function attempts to open the file and compile the job. If the file is opened, the output is appended to a string res (result). Once there is nothing else to read in, the function closes the pipe and saves the result as returnCode. If the return code is not 0, meaning there were errors or warnings, the compResults attribute is assigned the value of res.
- **ParseJob**
  - **Int jobType** = 2
  - **Std::string unparsedText:** string of all errors and warnings returned from a compile job
  - **std::map<std::string, std::vector<Errors>> errorMap:** Map with key as file that produced errors and value as an array of Error structs
  - **ParseJob::execute() override:** Uses regex to extract warnings and errors from the unparsedText string. Breaks up the errors and warnings into Error structs & inserts them into errorMap.
- **JSONJob**
  - **Int jobType** = 3
  - **Std::string filepath**: path to file that produced the errors in errorMap
  - **ParseJob::execute() override:** Uses regex to extract warnings and errors from the unparsedText string. Breaks up the errors and warnings into Error structs & inserts them into errorMap.

- **JSONJob::execute() override –** This function first creates a Json Document. It iterates through the keys in errorMap and, for each key, it iterates through each Error in the value std::vector<Error>. For each Error, the function utilizes the rapidjson library to create a Value errorObject, and add a member for each of the Error struct attributes. After iterating through the entire Map, it uses a rapidjson PrettyWriter to write the errors nested under the file name of their occurrence.

V. Compile System

- **Before running:** before running the JobSystem any files to be compiled through the system must have their compile commands in the makefile.
  - In this example, I am using the Clang++ compiler to compile a source file "demoError.cpp" and running the compiled executable "error_out" (*figure I.i)*.

```
demoError:
    clang++ -g -std=c++14 ./toCompile/demoError.cpp -o error_out
    ./error_out
```

Figure I.i: Sample target for passing to JobSystem as program argument

- **Running the JobSystem:** To run the JobSystem, the user must enter "make compile" in the terminal followed by ARGS="{target}"where {target} is the named target created in the previous step.
  - In this example, I am passing the demoError target as a command line argument to the JobSystem (*Figure I.ii)*.

```
% make compile ARGS="demoError"
```

Figure I.ii: Command to compile and run the JobSystem Program passing demoError target

- **Program Output:** If any target passed as a command line argument produces any compile errors or warnings, the JobSystem will create a JSON file titled "errors.json" which will contain the path to the file that produced the errors or warnings with details on the errors/warnings nested underneath (*figure III.i)*.
  - In this example, the target demoError was passed as a program argument (*Figure I.ii)*. The

```
{
  "demoError.cpp": [
    {
      "errorMessage": "error: use of undeclared identifier 'cout
      "lineNum": 3,
      "colNum": 5,
      "src": "    cout << \"fail\" << endl; "
    },
    {
      "errorMessage": "error: use of undeclared identifier 'endl
      "lineNum": 3,
      "colNum": 23,
      "src": "    cout << \"fail\" << endl; "
    }
  ]
}
```

Figure III.i: Contents of errors.json file produced by the program

```
1  int main()
2  {
3      cout << "fail" << endl;
4
5      return 0;
6  }
```

target demoError compiles a file called demoError.cpp (*figure II.i)* which contains two errors (*figure II.ii)*.

Figure II.i: Contents of the file to be compiled by the JobSystem

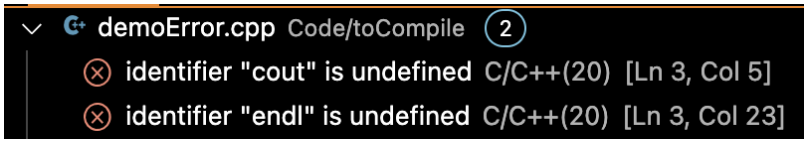

Figure II.iii: Errors detected in demoError.cpp