Giovanni Boscan 09/29/2023

Lab 1 Report

The Program

The Job System is a multi-threaded program that takes *make* commands as inputs, compiles the code, and outputs a detailed description of the errors, warnings, and notes (if any) in *JSON* format plus a job history file that contains a record of each Job state. Moreover, the system provides a reusable interface for developers who may want to use the code for other purposes, providing them with the ability to easily modify the code by creating different Job classes, inherited from a single Job interface, and executing any task they need. The program prints to the console basic information about the Jobs being executed and their respective outputs.

Job System Class

The Job System class is instantiated by calling the `CreateOrGet()` function, which returns a static pointer. If the pointer is null, it will create an instance of the object and reserve memory for the Job History vector so pushing back is faster. In that way, there is only one instance of that Job System throughout the whole program. Then, the `CreateWorkerThread()` function is called to create an instance of a worker thread. This function uses a mutex to protect the `workerThread` vector since it has to push back and create a new thread. With the `QueueJobs()` function, a specific `JobID` is passed in order to add it to the queue of running jobs. It also updates the `jobHistory` and the `jobStatus` for that Job. Afterward, when JobWorkerThread claims a Job and completes the actual Job, the `OnJobCompleted()` function is called to let the Job System know that it has to move the Job from the running deque to the completed deque. For this reason, we need mutexes that protect the different deques while moving the Jobs since two different JobWorkerThreads may reach that at the same time. At the end of the function, we unlock the mutex to avoid deadlocks, however, the use of mutex slows down the execution. In that order, the function `FinishCompletedJobs()` checks the deque of completed Jobs and calls the `CompleteCallback()` function of each Job present in that

deque, marks them as retired, and deletes the Job from the deque. Again, since this deque is shared among the JobWorkingThreads, these operations have to be carefully protected by mutexes since they can reach this line at the same time and cause problems and even corruption of the data. Also, the function `FinishJob()` accomplishes the same task as the previous one but in this case, terminates a specific Job by its ID. It also checks if the Job exists in the queue and will print an error if not found. Finally, the JobSystem class provides a destructor that shouts down all the JobWorkerThread and deletes them. In the same way, there is a function to destroy a single JobWorkerThread by ID called `DestroyWorkerThreads()` which will mark the thread as `isStopping` but will finish its current job if any. Lastly, the `Destroy()` function will delete the static pointer to the JobSystem.

Job Worker Thread Class

When a JobWorkerThread is instantiated, the constructor sets the unique name, Job channels, and the pointer to the JobSytem (static). As soon as JobSystem creates a JobWorkerThread from the function `CreateWorkerThread()`, the `StartUp()` function is called, which creates a new thread that contains a function pointer to `WorkerThreadMain()`. After that, `WorkerThreadMain()` makes a pointer to the current JobWorkerThread object and calls the `Work()` function. In that order, the `Work()` function will actively search for a Job until the `IsStopping` flag is set to true. If a Job is claimed, it will execute that Job and then call the `OnCompletedJob()` function when it finishes to let the JobSystem know that the Job has been completed. If the `ShutDown()` function is called, then the flag `IsStopping` will be set to true so the JobWorkerThread will not look further for any more Jobs. Lastly, when the destructor is called, the `ShutDown()` function will be executed to stop looking for Jobs. Then, the current thread will get blocked until it's finished and finally will get deleted.

Job Class

This abstract class serves as a template for any type of Job, therefore it can be reused by the programmer to accomplish any task. It contains a constructor which accepts a Job channel and a

Job type. The constructor will also create/increment a static variable to increment the Job ID so it is unique for every new Job that gets created. Also, a virtual destructor, as well as a `JobCompleteCallback()`, are declared to let the programmer know that it should be defined in the child class. In that order, a virtual `Execute()` function is set to 0 to let the programmer know that the function must be defined on the child class since the `Execute()` function in the child is obligatory for it to make sense and do something.

Compile Job Class

The CompileJob class is a children class of Job. Its task is to compile code (provided a single *make* command) and parse any errors/warnings/notes to a *JSON* output file. The constructor accepts a Job channel and a Job type so it can set the members of the parent class to the arguments passed. The `Execute()` function opens a terminal buffer, executes the make command, grabs the output, and stores it in the `output` member variable. Then, the `parseFile()` function gets called, which grabs the stored console output (already in a *JSON* format) and splits each token into a separate *JSON* object. At this point, the `generateJson()` function is called, which is in charge of parsing that single *JSON* console output, extracting the necessary data from it, and creating a new *JSON* object that contains the error/warning/note details in the format specified below. Finally, when this function finishes executing, the `parseFile()` function outputs the JSON object to the respective output file.

Job System As a Reusable Library

The Job System was thoughtfully designed to be modified by the programmer as desired to meet their specific needs. Classes, as described above, were purposefully built to support compatibility with any Job type. The programmer just has to create a new Job class, implement the `Execute()` function with the desired algorithm, and edit `main()` as needed and the Job System will take care of the multithreaded execution. For example, the `main()` function provided by default shows how to create the JobSystem, set the JobWorkerThreads to the

maximum number allowed by the system, create the CompileJobs, queue them, and query the statuses of every Job.
The programmer also doesn't need to worry about the memory management or safety of the threads, thanks to the provided abstraction layer and robustness of the system. Further, the programmer has the option to change the channel that determines which thread can run which job. In that way, the programmer has a lot more control over the execution of different types of Jobs in specific threads. Moreover, the provided interface allows the programmer to query specific information about the JobSystem, Jobs, and JobWorkerThreads thanks to the multiple setters and getters the different classes contain, features that can help to debug or extend the functionality of the program.

Compile System Examples

The program produces a *JSON* output file per each *make* command. This output file will be located in the Data folder of the workspace and will be named after its respective *make* command label. For example, if the command is `make project1`, the output file will be named "output_project1.json". The format of the *JSON* output can be generally described as the following: `[[File 1], [File 2], …, [File n]]` where `[File x] = [{Error 1}, {Error 2}, …, {Error n}]` and `{Error x} = {"code": code_snippet, "column": column_number, "error"|"warning"|"note": message, "file": file_name, "line": line_number}`. The `code_snippet` will be a single line string containing the error line plus two lines above and below from the source file, the `column_number` will be a number that indicates the column number where the error/warning/note happened, the `message` will be the message thrown by the compiler that gives details about the error/warning/note, `file_name` will be a relative path to the source file where the error/warning/note occurred, and `column_number` will be a number that indicates the line number where the error/warning/note happened. If a file does not have any compiling errors, it will not be shown in the final *JSON* output. If the entire set of files gets compiled with no errors, the output file will only contain `null`. For example:

Given code:

`Dog.h`

```
#pragma once
#include <iostream>

class Dog
{
public:
    Dog();
    void bark();
} // Notice there is a missing semicolon here!
```

`Dog.cpp`

```
#include "Dog.h"

Dog::Dog() {}

void Dog::bark()
{
    int woof = 1 // Notice there is a missing semicolon here!
}
```

`hello_world.cpp`

```
#pragma once // Pragma once produces compiler warning in files with main function
#include <iostream>

int main()
{
    float *ptr1, val = 3.14;
    char *ptr2;
    ptr1 = &val;
    ptr2 = &val; // float* can't be assigned to char*

    std::cout << "Hello World!!!" << std::endl // Missing semicolon here
    return 0;
}
```

Output JSON:

`output_project1.json`


```
{
    "./compilecode/Project1/Dog.cpp": [
        {
            "code": "{\n    int woof = 1\n}\n",
            "column": 1,
            "error": "expected ',' or ';' before '}' token",
            "file": "./compilecode/Project1/Dog.cpp",
            "line": 8
        }
    ],
    "./compilecode/Project1/Dog.h": [
        {
```

```
            "code": "#include <iostream>\n\nclass Dog\n{\npublic:\n",
            "column": 1,
            "error": "new types may not be defined in a return type",
            "file": "./compilecode/Project1/Dog.h",
            "line": 4
        },
        {
            "code": "#include <iostream>\n\nclass Dog\n{\npublic:\n",
            "column": 1,
            "file": "./compilecode/Project1/Dog.h",
            "line": 4,
            "note": "(perhaps a semicolon is missing after the definition of 'Dog')"
        },
        {
            "code": "#include <iostream>\n\nclass Dog\n{\npublic:\n",
            "column": 1,
            "error": "return type specification for constructor invalid",
            "file": "./compilecode/Project1/Dog.h",
            "line": 4
        }
    ],
    "./compilecode/Project1/hello_world.cpp": [
        {
            "code": "#pragma once\n#include <iostream>\n\n",
            "column": 9,
            "file": "./compilecode/Project1/hello_world.cpp",
            "line": 1,
            "warning": "#pragma once in main file"
        },
        {
            "code": "    char *ptr2;\n    ptr1 = &val;\n    ptr2 = &val;\n\n    std::cout << \"Hello
World!!!\" << std::endl\n",
            "column": 12,
            "error": "cannot convert 'float*' to 'char*' in assignment",
            "file": "./compilecode/Project1/hello_world.cpp",
            "line": 9
        },
        {
            "code": "    ptr2 = &val;\n\n    std::cout << \"Hello World!!!\" << std::endl\n    return
0;\n}\n",
            "column": 47,
            "error": "expected ';' before 'return'",
            "file": "./compilecode/Project1/hello_world.cpp",
            "line": 11
        }
    ]
}
```


Given code:

`Dog.h`

```
#pragma once
#include <iostream>

class Dog
{
```

```
public:
    Dog();
    void bark();
} // Notice there is a missing semicolon here!
```

Dog.cpp

```
#include "Dog.h"

Dog::Dog() {}

void Dog::bark()
{
    int woof = 1; // Semicolon has been added now (no compilation errors in this file)
}
```

hello_world.cpp

```
#pragma once // Pragma once produces compiler warning in files with main function
#include <iostream>

int main()
{
    float *ptr1, val = 3.14;
    char *ptr2;
    ptr1 = &val;
    ptr2 = &val; // float* can't be assigned to char*

    std::cout << "Hello World!!!" << std::endl // Missing semicolon here
    return 0;
}
```

Output JSON:

output_project2.json


```
{
    // Notice that "./compilecode/Project1/Dog.h" does not appear since it doesn't contain any errors
    "./compilecode/Project1/Dog.h": [
        {
            "code": "#include <iostream>\n\nclass Dog\n{\npublic:\n",
            "column": 1,
            "error": "new types may not be defined in a return type",
            "file": "./compilecode/Project1/Dog.h",
            "line": 4
        },
        {
            "code": "#include <iostream>\n\nclass Dog\n{\npublic:\n",
            "column": 1,
            "file": "./compilecode/Project1/Dog.h",
            "line": 4,
            "note": "(perhaps a semicolon is missing after the definition of 'Dog')"
        },
        {
            "code": "#include <iostream>\n\nclass Dog\n{\npublic:\n",
            "column": 1,
            "error": "return type specification for constructor invalid",
```

```
            "file": "./compilecode/Project1/Dog.h",
            "line": 4
        }
    ],
    "./compilecode/Project1/hello_world.cpp": [
        {
            "code": "#pragma once\n#include <iostream>\n\n",
            "column": 9,
            "file": "./compilecode/Project1/hello_world.cpp",
            "line": 1,
            "warning": "#pragma once in main file"
        },
        {
            "code": "    char *ptr2;\n    ptr1 = &val;\n    ptr2 = &val;\n\n    std::cout << \"Hello
World!!!\" << std::endl\n",
            "column": 12,
            "error": "cannot convert 'float*' to 'char*' in assignment",
            "file": "./compilecode/Project1/hello_world.cpp",
            "line": 9
        },
        {
            "code": "    ptr2 = &val;\n\n    std::cout << \"Hello World!!!\" << std::endl\n    return
0;\n}\n",
            "column": 47,
            "error": "expected ';' before 'return'",
            "file": "./compilecode/Project1/hello_world.cpp",
            "line": 11
        }
    ]
}
```


Given code:

`pointer.cpp`

```cpp
// Reference: https://www.w3schools.com/cpp/trycpp.asp?filename=demo_pointer_change

// This program does not contain any compilation errors

#include <iostream>
#include <string>
using namespace std;

int main() {
  string food = "Pizza";
  string* ptr = &food;

  // Output the value of food
  cout << food << "\n";

  // Output the memory address of food
  cout << &food << "\n";

  // Access the memory address of food and output its value
  cout << *ptr << "\n";

  // Change the value of the pointer
```

```
  *ptr = "Hamburger";

  // Output the new value of the pointer
  cout << *ptr << "\n";

  // Output the new value of the food variable
  cout << food << "\n";
  return 0;
}
```

Output JSON:

`output_project4.json`

```
null
```

Diagrams