Lab Three

1. Document API for library

I have three classes in the library: Company, Employee and Task

1.1 Company

Attribute:

- vector<Employee *> employee;
- vector<thread> threads;
- int employeeNum;

Method:

- Company(int employeeNum);
- void Company::hire()
- void Company::startWork(Task &task)
- void Company::stopWork()
- void Company::fire()
- void Company::report()

void Company::hire()

For this method, N(employeeNum , which is defined by the number of concurrent threads supported by the available hardware implementation) Employee objects will be added to employees vector, and method will output "N employees hired".

void Company::startWork(Task &task)

This method will take a pointer to a Task object as its parameter, and create N threads running Employee::work(), and method will output "N employees start working".

void Company::stopWork()

This method will terminate and delete all the threads that are created, and output "N employees stop working"

void Company::fire()

This method will remove all Employee objects from the employees vector.

void Company::report()

This method will output each employees workload.

1.2 Employee

Attribute:

- double workload;
- atomic<bool> workStatus{false};

Method:

- void hire();
- void fire();
- void work(Task &task);

void Employee::hire()
This method will set employee's work status to true, which means this employees is able to work.

void Employee:fire()
This method will set employee's work status to false, which means this employees is not able to work.

void work(Tasl &task)
This method will check employee's work status, if status is true, employee starts to work. The method will add up to employee's own workload.

1.3 Task

Attribute:

- double totalWork;
- mutex mu;

Method:

- Task (double totalWork);
- void work();

void work()
This method will add up total workload of all employees.

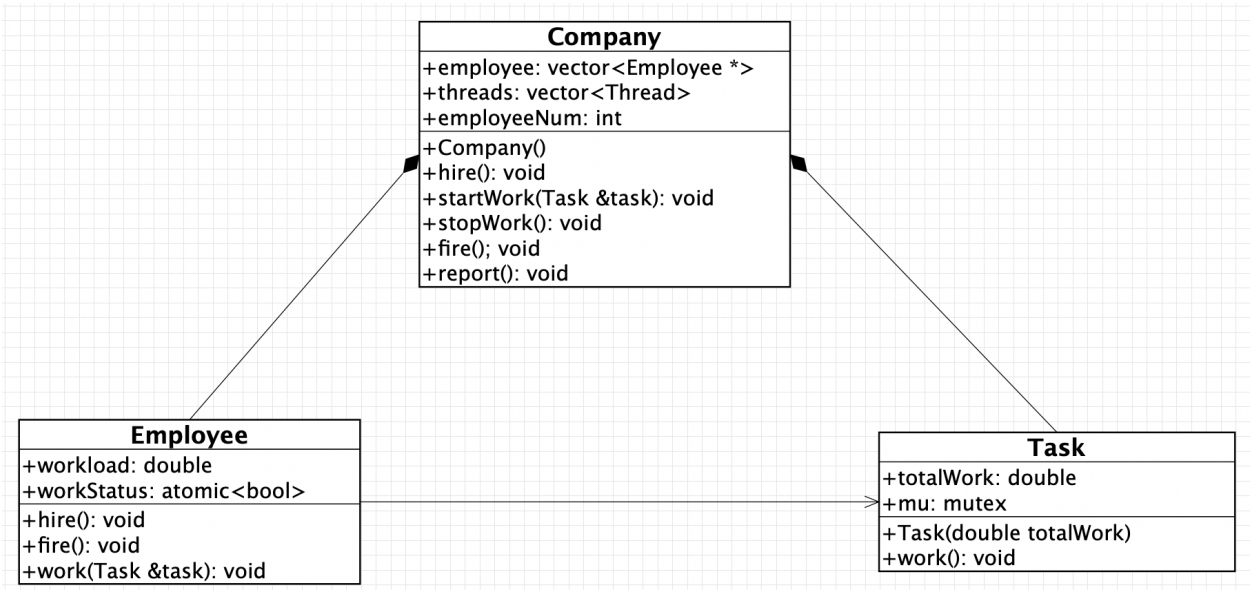
1.4 Main

void test()
This method will execute each function in the library sequentially. Screenshot of execution of this method is shown below.

```
7 employees hired
7 employees start working
7 employees stop working
Employee ID: 0   Workload: 1383.21
Employee ID: 1   Workload: 1822.82
Employee ID: 2   Workload: 1402.29
Employee ID: 3   Workload: 1192.34
Employee ID: 4   Workload: 1483.48
Employee ID: 5   Workload: 1361.36
Employee ID: 6   Workload: 1354.52
Company total workload: 10000
7 employees fired
```

2. Design Concept

In this library, we have three classes: Company, Employee and Task. Singleton design pattern is used in this library. A company has multiple employees. Employee is a component of company. Therefore, relationship between class Company and class Employee is composition. A company has multiple tasks. Task is a component of company. Therefore, relationship between class Company and class Task in also composition. A task can only be assigned to one employee, and a employee can complete multiple tasks. Therefore, relationship between class Employee and class Task is association. Relationship between classes is shown in the UML below.



3. Compare Execution Time of the Library Code between Native Multithreading and WASM Multithreading

1. Paste 100 execution runtime of native multithreading and WASM multithreading to one excel file, and perform basic statistic measure on them.

Native Mutithreading	WASM Multithreading
4.13519	7.053
2.88982	5.812
5.29948	5.958
6.34297	4.851
11.1402	4.968
7.68805	5.155
10.0794	6.159
7.26768	6.146
3.97623	6.362
9.49453	6.7

We can tell from the statistic results in the form below that execution time of native multithreading is in the range of 1.4444 seconds to 11.1402 seconds, and that of WASM multithreading is in the range of 4.799 seconds to 7.075 seconds. What is more, the average execution time of native multithreading is 5.6523334 seconds, while the average execution time of WASM multithreading is 6.38522 seconds.

	Native Multithreading	WASM Multithreading
mean	5.6523334	6.38522
max	11.1402	7.075
min	1.44444	4.799

2. Use Excel to calculate 95% confidence intervals. We can see from the form below: 95% confidence interval of native multithreading is 5.485198377 to 5.819468423, 95% confidence interval of WASM multithreading is 6.351193938 to 6.419246062.

	Native Multithreading	WASM Multithreading
mean	5.6523334	6.38522
max	11.1402	7.075
min	1.44444	4.799
standard_dev	2.696617668	0.548988944
95% confidence	0.167135023	0.034026062
mean - confidence	5.485198377	6.351193938
mean + confidence	5.819468423	6.419246062

From the following graph we can tell that lower bound of confidence interval of WASM multithreading is higher than the upper bound of confidence interval of native multithreading, which means there is no overlap between confidence interval of WASM multithreading and native multithreading. Therefore, we can conclude that these results are of statistically significant, performance of native multithreading is better than WASM multithreading.