Programming Test for Engineering Candidates

MatrixOrigin Technologies.

April 11, 2022

Contents

1 Notice 3

2 Evaluation 3

3 Submission 3

4 Task description 4

4.1 Precondition 4

4.1.1 Task 1.1 4

4.1.2 Task 1.2 5

4.1.3 Task 2.1 5

4.1.4 Task 2.2 6

4.1.5 Task 2.3 6

4.2 Task 3(Optional) 6

4.2.1 Task 3.1 6

4.2.2 Task 3.2 7

4.2.3 Task 3.3 7

1 Notice

- Please use Go programming language to solve the problems
- Please write your code with standard libraries of the language. You may use 3rd-party libraries only for logging and worker-server communications.
- Please tackle the task by yourself. Do not share or discuss this coding task with anyone including other applicants. Especially, do not upload your solution and/or problem description to public repository of GitHub. If we find any evidence of leakage, the applicant will be disqualified. If one applicant allows another applicant to copy answers, both applicants will be disqualified.
- We expect you to spend up to less than two days for this task. You can submit your work without solving all of the problems. Please do your best without neglecting your coursework, solving optional tasks(Task 3) would be a remarkable differentiation for candidates.

2 Evaluation

We will evaluate your submission based on the following criteria.

- Whether it outputs correct answers
- Processing time
- Memory consumption
- Readability of the source code
- Coverage with unit tests to ensure the correct behavior of the source code 3rd-party libraries are used properly (if used)
- How easy it is to reproduce the experimental results
- The submitted report is concise and easy to follow.

3 Submission

Please use Linux tar command to archive your source codes and Makefile. No executables should be included in the archive. An archive name should be firstname.lastname.tar.

Please submit your solution (via email to qinshuqi@matrixorigin.io or yingfeng.zhang@gmail.com). Please note that you can submit your solution only once. Any subsequent submissions will be ignored.

4 Task description

Implement a job execution system which consists of a server and a worker with the following core functionality:

1. A server gets a timestamp and returns information about jobs related to this timestamp.
2. A worker sequentially executes job tasks in the jobs received from the server.

Use the *job_list.zip* file as a job list for all tasks included in problem directory. The dataset has *xxxx.job* files and the job files have the contents as shown below.

```
data/
 00001.job
 00002.job
 00003.job
 ...
```

1. *JobID*: Job ID
2. *Created*: Timestamp when the job was created
3. *Priority*: "High" or "Low" priority
4. *Tasks*: Array of job task points (there are multiple tasks per job, and the points correspond to the "duration" of each task)

4.1 Precondition

- Job tasks with the same Job ID must be executed sequentially and in the specified order. A worker cannot execute them concurrently.
- Job tasks with different Job IDs can be executed concurrently.
- A worker can insert sleep between two tasks, but cannot insert into a task. For example, for Tasks=[2,2], the worker can insert sleep between the first 2-point task and the second 2-point task. However, the worker cannot split the tasks into 1-point tasks and insert sleep in between.
- You don't have to consider time spent in communication between server and worker. Also, server and worker are assumed to be synchronous.
- Each Job ID is unique in the whole system.
- The minimum timestamp granularity is "second".

4.1.1 Task 1.1

Implement a web server that takes a timestamp and returns all registered jobs where the timestamp is as in the request. Think of an appropriate server API. For example, assume the 2 jobs below are registered to a server.

```
00001.job
  1. JobID: 1
  2. Created: 00:00:01
  3. Priority: Low
  4. Tasks: 5, 6, 7

00002.job
  1. JobID: 2
  2. Created: 00:00:03
  3. Priority: High
  4. Tasks: 3,5
```

timestamp	JobID-Task No (Remain Point)		Executing Point
00 : 00 : 01	1 – 1(5)		5
00 : 00 : 02	1 – 1(4)		4
00 : 00 : 03	1 – 1(3)	2 – 1(3)	6
00 : 00 : 04	1 – 1(2)	2 – 1(2)	4
00 : 00 : 05	1 – 1(1)	2 – 1(1)	2
00 : 00 : 06	1 – 2(6)	2 – 2(5)	11
00 : 00 : 07	1 – 2(5)	2 – 2(4)	9
00 : 00 : 08	1 – 2(4)	2 – 2(3)	7
00 : 00 : 09	1 – 2(3)	2 – 2(2)	5
00 : 00 : 10	1 – 2(2)	2 – 2(1)	3
00 : 00 : 11	1 – 2(1)		1
00 : 00 : 12	1 – 3(7)		7
...			
00 : 00 : 18	1 – 3(1)		1

Table 1: Table of Task 1.2

In this case, the returned value for a given timestamp is as shown in the following examples.

- $time = 00:00:01- >$ return contents of 00001.job
- $time = 00:00:02- >$ return nothing (does not mean no response)
- $time = 00:00:03- >$ return contents of 00002.job

4.1.2 Task 1.2

Implement a worker that gets jobs from the server implemented in Task 1.1 and executes the tasks contained in the jobs. Additionally, make a chart, plotting remaining executing point every timestamp and include it in the submission material. An array of points of job tasks represents how long to execute, and 1 point corresponds to 1 second. A worker consumes 1 point every 1 second after starting the tasks. In the implementation you don't have to actually sleep for 1 second. For the example from Task 1.1, the result of job execution will be the table 1. "Executing point" columns is to be plotted to a chart.

4.1.3 Task 2.1

Add a "capacity" function to the worker implemented in Task 1.2, "capacity" is a limit of the sum of the remaining task points. Submit the result of executing point history chart with capacity 15. A task point cannot be higher than the worker capacity. For the example from Task 1, the result of job execution will be the table 2.

timestamp	JobID-Task No (Remain Point)		Executing Point
$00:00:01$	$1-1(5)$		5
$00:00:02$	$1-1(4)$		4
$00:00:03$	$1-1(3)$	$2-1(3)$	6
$00:00:04$	$1-1(2)$	$2-1(2)$	4
$00:00:05$	$1-1(1)$	$2-1(1)$	2
$00:00:06$	$1-2(6)$		6
$00:00:07$	$1-2(5)$	$2-2(5)$	10
$00:00:08$	$1-2(4)$	$2-2(4)$	8
$00:00:09$	$1-2(3)$	$2-2(3)$	6
$00:00:10$	$1-2(2)$	$2-2(2)$	4
$00:00:11$	$1-2(1)$	$2-1(1)$	2
$00:00:12$	$1-3(7)$		7
...			
$00:00:18$	$1-3(1)$		1

Table 2: Table of Task 2.1

timestamp	JobID-Task No (Remain Point)		Executing Point
$00:00:01$	$1-1(5)$		5
$00:00:02$	$1-1(4)$		4
$00:00:03$	$1-1(3)$	$2-1(3)$	6
$00:00:04$	$1-1(2)$	$2-1(2)$	4
$00:00:05$	$1-1(1)$	$2-1(1)$	2
$00:00:06$			5
$00:00:07$	$1-2(6)$	$2-2(5)$	10
$00:00:08$	$1-2(5)$	$2-2(4)$	8
$00:00:09$	$1-2(4)$	$2-2(3)$	6
$00:00:10$	$1-2(3)$	$2-2(2)$	4
$00:00:11$	$1-2(2)$	$2-2(1)$	2
$00:00:12$	$1-3(1)$		1
$00:00:13$	$1-3(7)$		7
...			
$00:00:18$	$1-3(1)$		1

Table 3: Table of Task 2.2

4.1.4 Task 2.2

Add a "priority" function to the worker implemented in Task 2-1. Submit the result of executing with capacity 15 as a point history chart, too.

4.1.5 Task 2.3

Implement a smarter worker considering "capacity" and "priority" with your own efficiency indicator. Submit a report with how to decide the indicator and shape the scheduler.

4.2 Task 3(Optional)

This task 3 is optional, please do your best without neglecting your coursework. Choose any one from the task 3.1–3.3 and submit a report.

4.2.1 Task 3.1

Change priority from binary value "High", "Low" to a numerical value [0-100] . In this case, 100 means the highest priority job. Please use the file $job\_list\_3-1.zip$ included in problem directory.

4.2.2 Task 3.2

Modify the system to run more than one worker. Compare execution efficiency between several workers with capacity 10 and a single worker with capacity 15.

4.2.3 Task 3.3

Assume that if the task point is an even number, a worker can execute the task at twice the speed. For example, a 5-point task takes 5 seconds and a 6-point task takes 3 seconds. Implement a worker which runs more efficiently under that condition.