Home Assignment 6
Advanced Programming

qhz731

October 23, 2024

1 Introduction

In this assignment, I implemented a concurrent job scheduling system that supports multiple workers for the Stateful Planning Committee (SPC). The main functions of the system include adding and removing workers, job scheduling, job cancellation, timeout handling, and exception handling. The code uses the channel and GenServer patterns to implement message passing between SPC and workers, and each worker runs in an independent thread. For the convenience of testing, I have written a series of test cases that cover the main functions of the system. To run the tests, simply execute the `cabal test --test-show-details=always` command in the project root directory. Based on existing test cases, my solution is functionally correct, but there is still room for improvement in the robustness and error handling of the code in some boundary cases of worker management and concurrent job execution.

2 Task: Adding Workers

In this task, I implemented an SPC worker system based on message passing and state management. When SPC receives the MsgWorkerAdd message, it first checks whether the worker name already exists. If not, it creates a new worker process through spawn and manages the worker state with WorkerM monad. WorkerM tracks the currently executed job (currentJob) through WorkerState, achieving the encapsulation and management of the worker state. In terms of scheduling, SPC's schedule function identifies idle workers (by checking the records in spcJobsRunning) and assigns pending jobs to idle workers through the workerIsIdle function. During the job allocation process, the SPC status is updated (moving the job from pending to running), and a message to execute the job is sent to the worker. SPC has an independent jobDoneChan channel, which is used by workers to report job completion status to SPC. After receiving the job completion message, SPC will send MsgJobDone to SPCServer to call jobDone to update the job status, and the worker will idle again for subsequent job scheduling. The entire implementation adopts a hierarchical state management architecture. SPC is responsible for global scheduling and state maintenance, while the worker manages its own execution state through an independent state monad. The two realize interaction through the message passing mechanism.

3 Task: Job Cancellation

In this task, I implemented the job cancellation function. When SPC receives MsgJobCancel, it will first look for the target job and its corresponding worker in spcJobsRunning. If a worker is found to be executing the job, a cancellation request is sent to the worker through MsgCancelJob in WorkerMsg. When a worker executes a new job, it will create an independent execution thread through forkIO to run the job, and store the thread ID (ThreadId) together with the JobId in the currentJob field of WorkerState. This design makes it easy to terminate the execution thread of a specific job through killThread without affecting the operation of the worker's main thread. When processing the cancellation request, the worker will terminate the corresponding job thread, update its own status (set currentJob to Nothing), and report to SPC through jobDoneChan that the job ends in the DoneCancelled state. After receiving this message, SPC handles the subsequent process through the jobDone function.

4 Task: Timeouts

In this task, I adopted a centralized timeout management strategy to implement job timeout control. When the system assigns a job to a worker, it calculates the job deadline based on the jobMaxSeconds field of the job and records the information together with the job in spcJobsRunning. The SPC main thread triggers the checkTimeouts function through the MsgTick message received regularly (every second), and the checkTimeouts function obtains the current time and compares it with the deadlines of all running jobs. For jobs that exceed the time limit, SPC sends a MsgCancelJob message to the corresponding worker with the end reason DoneTimeout. This design reuses the same infrastructure as the job cancellation mechanism: after receiving the timeout cancellation request, the worker terminates the job execution thread (by calling killThread with the ThreadId stored in WorkerState.currentJob), updates its own status, and reports the job end to SPC through jobDoneChan. By centrally managing timeout detection at the SPC level, the system implements unified time management, avoids the complexity of distributed clock synchronization, and simplifies the implementation of workers, allowing them to focus on job execution without maintaining their own timers.

5 Task: Exceptions

In this task, when a worker receives a new job, it runs the job in an independent execution thread and wraps the job execution in an exception handling block. The Haskell exception handling mechanism is used to catch any exceptions (SomeException) that may occur when the worker executes the job. If the job is completed successfully, the worker reports the Done status to SPC through jobDoneChan; if an exception is caught, the DoneCrashed status is reported. This design ensures that the exception is confined to the job execution thread and does not spread to the worker's main thread or other parts of the SPC system, ensuring that the worker can recover after a job crashes and continue to process new jobs.

6 Task: Removing Workers

In this task, I implemented the worker's exit mechanism. The main challenge was in solving the state synchronization problem during the exit process. The implementation adopts a multilevel state processing strategy: First, the workerStop function sends the MsgWorkerStop message to the worker to trigger the exit process. After receiving the stop message, the worker checks the currentJob state in WorkerState. If there is a running job, it calls killThread to terminate the job thread and reports to SPC through jobDoneChan that the job ends with the DoneCancelled state.

On the worker side, the message processing function returns a bool value to indicate if the worker should stop. I used workerLoop instead of forever. When a stop message is received, it returns false, no longer calls worker Loop to terminate the loop, and then sends an exit notification to SPC through workerGoneChan. However, due to the asynchronous nature of the communication channel, the SPC may have a short state inconsistency time window after the worker is actually terminated, resulting in new jobs being incorrectly assigned to the terminated worker. To solve this problem, I implemented the checkIfJobIsRunningOnDeadWorkers function to check and reset the job status assigned to the terminated worker before each message processing.

Although this design cannot completely avoid the existence of state inconsistency time windows, it ensures the final state consistency of the system through state checking and compensation mechanisms.

7 Q&A

7.1 What happens when a job is enqueued in an SPC instance with no workers, and a worker is added later? Which of your tests demonstrate this?

When a job is queued in an SPC instance with no workers, the job is added to the spcJobsPending list and remains in a waiting state. After adding the worker, the job will be assigned to the worker and executed. My test case "add-job" demonstrates this scenario: first, a new SPC instance is created, and a job is added. The job status is verified as JobPending at this time through jobStatus. Then a worker "worker1" is added. The system's schedule mechanism detects the idle worker and assigns the pending job to the worker for execution through workerIsIdle. At this time, check again that the job status has changed to JobRunning. After the job is executed, the status changes to Done, and the value of ref is successfully modified, verifying that the job was indeed executed. In order to further verify that the worker can continue to process jobs, the test case adds a second job and confirms its successful execution.

```
testCase "add-job" $ do
  spc <- startSPC
  ref <- newIORef False
  j <- jobAdd spc $
    Job (threadDelay 1000000 >> writeIORef ref True) 2
  r1 <- jobStatus spc j
```

```
  r1 @?= JobPending
  -- Add a worker
  _ <- workerAdd spc "worker1"
  r2 <- jobStatus spc j
  r2 @?= JobRunning
  r3 <- jobWait spc j
  r3 @?= Done
  v <- readIORef ref
  v @?= True
  -- Check if worker can execute other jobs
  j2 <- jobAdd spc $ Job (writeIORef ref False) 1
  r4 <- jobWait spc j2
  r4 @?= Done
  v2 <- readIORef ref
  v2 @?= False,
```

7.2 Did you decide to implement timeouts centrally in SPC, or decentrally in the worker threads? What are the pros and cons of your approach? Is there any observable effect?

I chose to implement a centralized timeout management mechanism in SPC. When assigning jobs, SPC calculates the deadline of the job (current time plus jobMaxSeconds) and stores it in spcJobsRunning together with the job information. The checkTimeouts check is triggered by regular MsgTick messages, and a cancellation request is sent to the worker to cancel the timeout jobs.

Pros: 1. Centralized time management avoids the problem of distributed clock synchronization and ensures the accuracy of timeout determination. 2. Simplifies the implementation of workers. When implementing the worker I only need to focus on job execution without maintaining their own timers.

Cons: 1. Due to periodic checks (once per second), there may be a timeout determination delay of up to 1 second. 2. All timeout checks are performed in the SPC main thread, which may cause performance bottlenecks when the number of jobs is large. 3. If the SPC thread is blocked, it may affect the timeout detection of all jobs.

7.3 Which of your tests verify that if a worker executes a job that is canceled, times out, or crashes, then that worker can still be used to execute further jobs?

I wrote three test cases ("cancel-job", "timeout", "crash") to verify the availability of the worker after handling an abnormal job:

```
testCase "cancel-job" $ do
  spc <- startSPC
  ref <- newIORef False
  j <- jobAdd spc $
```

```haskell
      Job (threadDelay 2000000 >> writeIORef ref True) 3
    _ <- workerAdd spc "worker1"
    jobCancel spc j
    r <- jobWait spc j
    r @?= DoneCancelled
    v <- readIORef ref
    v @?= False
    -- Ensure the worker is still working
    j2 <- jobAdd spc $ Job (writeIORef ref True) 1
    r2 <- jobWait spc j2
    r2 @?= Done
    v2 <- readIORef ref
    v2 @?= True,
```

These test cases cover three situations where the job is terminated abnormally, and verify the continued availability of the worker by submitting a new job after the abnormal job. The test results show that the worker is able to maintain normal working status after handling an abnormal job.

7.4 If a worker thread were somehow to crash (either due to a bug in the worker thread logic, or because a scoundrel killThreads it), how would that impact the functionality of the rest of SPC?

In my implementation, if the worker thread crashes unexpectedly, it will cause the SPC system state to be inconsistent. Since the worker fails to send the MsgWorkerGone message normally, the record of the worker in the spcWorkers list will not be removed, and the job being executed on the worker will continue to remain in the spcJobsRunning state until it is canceled due to timeout. Since SPC does not know that the worker has crashed, after the timed-out job is canceled, the SPC system will think that the worker is idle and continue to try to assign new jobs to the crashed worker.

This is indeed a defect in the implementation. Possible improvements include implementing a worker heartbeat mechanism and regularly checking whether the worker is alive.

8 Reference

I used the AI tool Cursor to generate part of the test cases. The generated tests are carefully checked to ensure the reliability.