Advanced Programming
Home Assignment 6

Carlo Rosso, Chinar Shah

Contents

1 Intro **1**

2 Adding Workers **2**

3 Job Cancellation **3**

4 Timeouts **3**

4.1 Managed by SPC 3

5 Exception **4**

6 Removing workers **4**

7 Questions **4**

7.1 Managed by SPC 5

1 Intro

Overall this assignment was very top heavy, with the first exercise being the most time consuming to implement. This was mostly because it took some time to understand the context of the question and the overall assignment before we had to implement it. But after the first exercise was done - the rest was fairly fast and easy. However, in saying that it was difficult to ascertain what we needed to implement because there

were so many moving parts. The biggest uncertainty was if we had to implement the whole state monad for the workers. Reflecting back on the work, we could probably make the SPCM more complex so that it can handle everything, but it would be much more messier and convoluted.

2 Adding Workers

We implemented `workerAdd` with the following steps:

1. Implement `WorkerM` monad which represents a running worker with its state
    - We emulated `SPCM` monad because they are both stateful monads and they both run a server process to allow communication
2. Add `MsgWorkerAdd` to `SPCMsg` to allow adding workers with the signature `MsgWorkerAdd WorkerName (ReplyChan (Either String Worker))`
3. We implemented `workerAdd`

We also need to define a new message for testing purpose:

1. Add `spcWorkers` to the `SPCState` to keep track of the workers
2. Implement `workerAssignJob` to assign a job to a worker
3. Add `MsgJobDone` to `SPCMsg` to allow workers to notify the SPC when a job is done
4. The same thing goes also for `Worker` therefore we added the messages
    - `WorkerJobNew`
    - `WorkerJobDone`

    to `WorkerMsg` and added the handlers for those in the `workerHandle` function
5. Implement the handler for `MsgWorkerAdd`, which is reported above
6. Implement the handler for `MsgJobDone`, which is reported above
7. Implement `schedule` to assign jobs to workers if possible
8. Add `spcWaiters` to the `SPCState` to keep track of the waiters for jobs
9. Implement handler for `MsgJobWait` to add a waiter to the list of waiters
10. Implement `jobWait` to wait for a job to finish
11. Finally, we added `spcChan` to the `SPCState`, which is the channel used by the SPC server to receive messages

Whilst difficult to implement, the solution is functional and covered in the tests of ‘adding job’

3 Job Cancellation

Since we already defined the monad earlier, implementing the cancellation was very easy, straightforward and similar to the exercise.

Note that if the job is pending, we do not need to interact with any worker, otherwise we need to find the worker that is running the job and tell it to stop. We implemented it, by sending a message to all the workers. The workers have the job id in their state, so they can check if they are running the job with that id, if that is the case they will cancel the job. Overall, the solution for this was functional and tested with the test 'canceling job'.

4 Timeouts

We implemented both the solutions, because we wanted to see the difference, below. Both solutions were functionally correct and were working. From this implementation we have a preferance for the workers to manage the timeouts because it simplifies the SPC, making it more lightweight. Here the SPC can focus on managing the workers while the workers are only focused on the jobs. Hence this method is more modular and clean.

4.1 Managed by SPC

```
checkTimeouts :: SPCM ()
checkTimeouts = do
  now <- io getSeconds
  state <- get
  forM_ (spcJobsRunning state) $ \(jobid, deadline) ->
    when (now >= deadline) $ do
      jobDone jobid DoneTimeout
      io $ send (spcChan state) $ MsgJobCancel jobid
```

The core of the implementation is `checkTimeouts`:

1. Update the state of SPCM so that `spcJobsRunning` contains the job id and the deadline of each job (we do not need the job itself anymore)
2. On the server start we add a thread that sends a `MsgTick` every second just like in the exercise

3. At every tick we call `checkTimeouts` that checks whether a job has exceeded the deadline and it cancels the job

We confirm the functionality in the 'timeout' test.

5 Exception

This was the simplest assignment task as we can just emulate the exercise code. Hence that can attest to its functionality.

6 Removing workers

This was also fairly fast as we had the context of the assignment and the structure of the code understood. This was done by the following code - which is functionally correct.

```haskell
    MsgWorkerStop wname -> do
      state <- get
      case lookup wname $ spcWorkers state of
        Just worker -> do
          io $ workerStop worker
          put $ state {spcWorkers = removeAssoc wname $ spcWorkers state}
        Nothing -> pure ()
```

7 Questions

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 enqueued in an SPC instance with no workers, it will be placed in the pending jobs queue. The job will remain in this state until a worker is added to the SPC instance, at which point the SPC will assign the pending job to the newly added worker.

The relevant test case that demonstrates this behavior is 'adding worker'

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?

7.1 Managed by SPC

Con's

- Not the simplest solution

Pro's:

- More modular
- Simplify the worker and make the SPC more complex

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

The tests that verify if a worker executes a job that is cancelled are found in the test file under

1. 'canceling job'
2. 'timeout'
3. 'crash'

All of these tests confirm that the worker can still be used to execute the tests

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?

The impact of a worker crash on the SPC depends on how the timer is implemented. If the timer is managed by the worker itself, the SPC will remain unaware of the worker's crash until it attempts to communicate with the worker. Consequently, any job being executed by the crashed worker will remain incomplete and stuck in the "Running" state, which could lead to resource contention and hinder overall system performance.

On the other hand, if the timer is managed by the SPC, it will detect when a job exceeds its deadline. In this scenario, the SPC will automatically cancel the job, moving it into the spcJobsDone list with a status of DoneTimeout. This allows the SPC to maintain an accurate representation of job statuses and ensures that resources can be reallocated efficiently. Overall, managing the timer at the SPC level enhances system resilience by enabling timely job management and reducing the potential for unmonitored job failures.