Advanced Programming
Home Assignment 6

Carlo Rosso, Chinar Shah

Contents

1 Intro

As a general comment, I would say it took a huge amout of time to implement solve the first exercise, but after that the rest was easy and fast and very similar to the exercise. It was quite confusing to understand what needed to be implemented, because there were many things to do so it was easy to loose track of the current task and it was even harder to see what the next step was.

Also I was uncertain whether I had to implement the whole state monad for the workers. Probably you could also make the SPCM way more complex to handle everything, but I think it would have been a mess.

2 Adding Workers

```haskell
handleMsg :: Chan SPCMsg -> SPCM ()
handleMsg c = do
  schedule
  msg <- io $ receive c
  case msg of
    MsgWorkerAdd wname rsvp -> do
      state <- get
      case lookup wname $ spcWorkers state of
        Just _ -> io $ reply rsvp $ Left "Worker already exists"
        Nothing -> do
          worker <- io $ startWorker c wname
          put $ state {spcWorkers = (wname, worker) : spcWorkers state}
          workerAssignJob worker
          io $ reply rsvp $ Right worker
    MsgJobDone jobid reason wname -> do
      jobDone jobid reason
      state <- get
      case lookup wname $ spcWorkers state of
        Just worker -> do
          workerAssignJob worker
        Nothing -> pure ()
    ...
```

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`

Defining a new message we also need to (otherwise we cannot test it):

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

3 Job Cancellation

```haskell
MsgJobCancel jobid -> do
  state <- get
  case ( lookup jobid $ spcJobsPending state,
         lookup jobid $ spcJobsRunning state
       ) of
    (Just _, _) -> do
      jobDone jobid DoneCancelled
    (_, Just _) -> do
      forM_ (spcWorkers state) $ \(_, Worker worker) -> do
        io $ sendTo worker $ WorkerCancelJob jobid
    _ -> pure ()
```

Since we already defined the monad and everything, implementing the cancellation was very easy, straightforward and similar to the exercise.
Here the steps:

1. implement `WorkerCanelJob` in `WorkerMsg` to allow workers to cancel a job
2. implement the handler for the `WorkerCancelJob` in `workerHandle`
3. implement the handler for `MsgJobCancel` in `handleMsg`
4. implement `jobCancel` to cancel a job

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.

4 Timeouts

I implemented both the two solutions, because I wanted to see the difference, below in the question I sum up the workflow of both solutions. I am not too sure about the pros and cons. Probably I prefer the workers to manage the timeouts, because it simplify the SPC, making it more lightweight and thinking just about how to manage the workers, while the workers are focused only on managing the jobs. It looks to me more modular and clean (I am not sure though).

4.1 Managed by SPC

```haskell
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

4.2 Managed by workers

```haskell
workerHandle :: Chan WorkerMsg -> WorkerM ()
workerHandle c = do
  msg <- ioW $ receive c
  state <- getW
  case msg of
    WorkerJobNew jid job -> do
      case exec state of
        Just _ -> pure ()
        Nothing -> do
          tid <- ioW $ forkIO $ do
            let val = do
                  jobAction job
                  send c $ WorkerJobDone jid Done
                onException :: SomeException -> IO ()
                onException _ = do
                  send c $ WorkerJobDone jid DoneCrashed
            catch val onException
          _ <- ioW $ forkIO $ do
            threadDelay $ jobMaxSeconds job * 1000000
            killThread tid
            send c $ WorkerJobDone jid DoneTimeout
          putW $ state {exec = Just (jid, tid)}
    ...
```

Here we have also the exception handling, which is not yet required. Either way, when a job is started we also start a thread that waits for the timeout and then kills the job and notifies the worker that the job has timed out. Probably this is the simplest way to implement timeouts.

5 Exception

Code above, did not do much, I just copied the exercise code.

6 Questions

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?

6.1 Managed by workers

1. job started: also Timer started
2. Timer waits jobMaxTime seconds
3. Timer interrupts job's thread
4. Timer notifies Worker: job DoneTimeout
5. Worker notifies SPC: job DoneTimeout

6. SPC moves job: Running -> Completed

6.2 Managed by SPC

1. SPC started: also Timer started
2. Timer sends MsgTick to SPC every second
3. SPC receives MsgTick: job exceeds jobMaxTime:
    - SPC moves job: Running -> Completed
    - SPC notifies Worker: job Cancel
4. Worker kills job
5. Worker notifies SPC: job Cancel

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?

Depends by the implementation of the timer. If the timer is managed by the Worker, the SPC would never know of the crash of the worker until the SPC actually tries to communicate with the worker. Therefore the job which was being executed by the crashed worker would never be completed and would be stuck in the Running state.
If the timer is managed by the SPC, the SPC would notice that the job has exceeded the deadline and would cancel the job. The job would be moved into the `spcJobsDone` list with the status `DoneTimeout` and that would be it.