Advanced Programming 2024

Assignment Six

Concurrent Programming

Due: 2024-10-27

Synopsis: Implementing a concurrent job scheduling system with support for timeouts and error handling.

1 Introduction

The main limitation of the Stateful Planning Committee (SPC), developed during the exercises, is that it supports only a single job at any given time, meaning there is no true concurrency. In this assignment we will extend SPC with a notion of *workers*, which are persistent threads that receive work orders from the central SPC thread. The number of workers can vary during the lifetime of an SPC session. Workers are explicitly created and destroyed through API functions.

The starting point of your work is the SPC developed during the exercises, except that the existing scheduling/execution machinery has been removed. You will be gradually reconstructing the removed functionality throughout the assignment, including support for timeouts and exceptions. The main challenge in this assignment is designing the protocol used for communication between SPC and its workers.

1.1 Concepts

A worker can be either *idle*, if it is not currently executing a job, or *busy*, which means it is executing a job. Each worker is represented by a *worker thread*, which communicates with SPC through messages. A worker may launch further threads as necessary.

Initially, an SPC instance has no workers, and any submitted jobs will be pending. At any time, if there is at least one pending job and one idle worker, a pending job and an idle worker will be picked, and that worker asked to execute that job. The worker is now considered busy.

When a worker finishes with a job, it notifies SPC. The worker is now considered idle. SPC then notifies anyone waiting for completion of the job and performs other necessary bookkeeping.

All workers have a name of type `String`. These have no semantic meaning, but can be used for debugging and logging. One invariant that must hold is that no two workers may have the same name.

An SPC instance with k workers can at any given time be executing at most k jobs concurrently.

2 Task: Adding Workers

A fresh SPC instance has zero workers. A worker can be added with the following function:

```
workerAdd :: SPC -> WorkerName -> IO (Either String Worker)
```

The return value is `Left` with an error message if anything goes wrong. One thing that can go wrong is that a worker with the specified name already exists. Otherwise, the return value is `Right` alongside a handle to a worker, of type `Worker`. This handle is used by API clients for interacting with the worker in later tasks.

A worker starts out idle and can immediately be used by SPC for executing jobs.

You should also add tests.

2.1 Hints

A worker is essentially a kind of server, just like SPC itself. A worker will need to have access to the `Chan SPCMsg` used for sending messages to SPC. The handout contains a message type, `WorkerMsg`, that you should add messages to, just as you did with `SPCMsg`. Similarly, you will need to add new messages to `SPCMsg` for the messages that workers will send to `SPC`.

A minimal solution will involve extending `WorkerMsg` with a message that contains the job to be done (of type `IO ()`), and `SPCMsg` with a message that a worker sends when it is done with a job. This message should contain the worker name so SPC can know from which worker it received it.

You do not need to worry about timeouts or exceptions for now—we will return to that in later tasks.

It is probably overkill to use `GenServer` for the workers—particularly as they are purely an implementation detail internal to the SPC module, so there are no "users".

3 Task: Job Cancellation

In this task you will restore support for job cancellation:

```
jobCancel :: SPC -> JobId -> IO ()
```

The semantics are identical to the version that you implemented in the exercises, but now SPC must communicate the cancellation to the worker (if any) executing the job.

4 Task: Timeouts

In this task you will restore support for timeouts (the `jobMaxSeconds` field of the `Job` type). The semantics are the same as in the exercises: if a job exceeds its time limit, it will be terminated. Further, the worker executing the job must survive this ordeal, and become idle again.

As before, when a job is terminated due to timeout, its final status will be `JobDone DoneTimeout`.

4.1 Hints

The major design decision is to decide whether to enforce the timeouts centrally in the SPC thread, or decentrally in the worker threads. Specifically, either SPC can tell a busy worker that it should abort its current job, or a worker can itself abort after a timeout and inform SPC that the job took too long. Which of these approaches you implement is up to you.

The operational way you notice the timeouts is similar to in the exercises: have some sort of thread deliver regular "ping" messages, and check whether some job has exceeded its deadline.

Depending on your design, you will probably need to extend both `WorkerMsg` and `SPCMsg`.

5 Task: Exceptions

In this task you will restore support for robust handling of exceptions. The most important property is that a job that crashes should not harm the worker. Instead, SPC should be notified, and the worker once again become idle.

As before, when a job terminates with an exception, its final status will be `JobDone DoneCrashed`.

5.1 Hints

You will need to catch exceptions in the workers, just as in the exercises. The logic is essentially the same.

6 Task: Removing Workers

In this task you will add support for this function:

```
workerStop :: Worker -> IO ()
```

The function `workerStop` stops (i.e., shuts down) a worker. It has no effect if the worker is already stopped. If the worker is busy with a job when the worker is stopped, then it is equivalent to first cancelling the job, then stopping the worker.

7 Code handout

The code handout consists of the following nontrivial files.

- `a6.cabal`: Cabal build file. **Do not modify this file.**
- `runtests.hs`: Test execution program. **Do not modify this file.**
- `src/GenServer.hs`: The GenServer library. **Do not modify this file.**
- `src/SPC.hs`: Implementation of the glorious SPC.
- `src/SPC_Tests.hs`: Test suite, where you will add plentiful tests.

8 Your Report

You are expected to comment on the *interesting* details of your implementation. You are *not* expected to give a line-by-line walkthrough of your code. Most importantly, you are expected to reflect on the *quality* of your code:

- Do you think it is functionally correct? Why or why not?
- Is there some improvement you'd have liked to make, but didn't have the time?

It is more important to be aware of the strengths or shortcomings of your solution, than it is to have a complete solution.

8.1 The structure of your report

Your report must be structured exactly as follows:

Introduction: Briefly mention very general concerns, your own estimation of the quality of your solution, and possibly how to run your tests.

A section for each task: Mention whether your solution is functional, test cases cover its functionality, which test cases it fails for (if any), and what you think might be wrong.

A section answering the following numbered questions:

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?
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?
3. 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?

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

All else being equal, **a short report is a good report**.

9 Deliverables for This Assignment

You must submit the following items:

- A single PDF file, A4 size, no more than 5 pages, describing each item from report section above.
- A single zip/tar.gz file with all code relevant to the implementation, including at least all the files from the handout. For this assignment it is not necessary to add additional files.

Remember to follow the general assignment rules listed on the course homepage.

10 Assessment

You will get written qualitative feedback, and points from zero to four. There are no resubmissions, so please hand in what you managed to develop, even if you have not solved the assignment completely.