Report on the Reference Solution to the A4 CompSys Assignment

Troels Henriksen (athas@sigkill.dk)

November 19, 2017

1 Introduction

This report documents the reference solution of the A4 assignment. It is intended not just to explain the reference solution, but also as an example of a report with good structure and content. You may use it as inspiration for reports you write later during this and other courses. However, do not follow it slavishly: different problems call for different solutions, and different people have different style. If you have another style you prefer, then stick to it. An idiosyncratic report executed well is superior to a conventional report executed poorly.

The problem to be solved is to finish the implementation of a C library of stream transducers. The library API was defined in the handed-out header files, and I have not found reason to stray from it. However, Section 4 notes some deficiencies that arise from limitations in the API itself.

2 Implementation

Due to the requirement for multiple stream transducers (and sources) to execute concurrently, I have chosen to implement both as subprocesses. The process structure of a program using the transducers API will thus contain a single main process, with one child process for every active source or transducer. Sinks are executed in the main process itself, as specified in the assignment text.

I have chosen to implement a stream as a pointer to a `FILE` object, along with a flag indicating whether the stream already has an associated reader (either a transducer or sink):

```
struct stream {
  FILE *f;
  int linked;
};
```

The purpose of the flag is to signal an error if we try to read the same stream from multiple transducers, as required by the assignment text. Memory

is allocated for the `stream` object whenever streams are created, and freed when `transducers_free_stream()` is called. This is the only place where dynamic allocation is used.

Most of the functions for creating transducers work in approximately the same way, and I will not include the code here. The general approach for creating a transducer, given the input stream(s), is as follows:

1. Create a pipe object (or two, in the case of `transducers_dup()`).
2. Fork a new child process.
3. In the child, close the *read* end of the pipe.
4. In the parent (the main process), close the *write* end of the pipe.
5. Produce a `stream` object containing the *read* end of the pipe.

This has the effect that the only process that can write to the pipe is the child, which means that when the child terminates, we may get an end-of-file condition when trying to read from the pipe. We can then write transducers (and sinks) that loop until end-of-file is reached on the input stream.

3 Testing

Apart from the handed-out test programs, several more were written that test various other properties of the API, including error detection. All tests can be run with `make test`. The properties tested are:

Program	**Property**
`test0.c`	The functions `transducers_link_source()` and `transducers_link_sink()` can be connected and reproduces the output of the source function.
`test1.c`	A simple single-input transducer works.
`test1.c`	A simple two-input transducer works.
`test1.c`	`transducers_dup()` works.
`test_error0.c`	A sink cannot be connected to a stream that has already been connected.
`test_error1.c`	A single-input transducer cannot be connected to a stream that has already been connected.
`test_error2.c`	A two-input transducer cannot be connected to a stream that has already been connected.

3.1 Test Limitations

The developed tests do not contain transducers that produce more or fewer output elements than they receive as input. The `divisible.c` demo program does contain such a transducer, but it is not a test program per se, as it is not checked whether its output valid.

We also do not have tests that check the absence of memory leaks in the library. Instead, this has been verified manually via `valgrind`.

4 Limitations and Potential Problems

The transducers library as implemented has some issues:

- An error causing a transducer to crash may be silently ignored. Since the termination of a transducer process will implicitly cause its streams to be closed (by the kernel), this is not distinguishable from a transducer terminating normally. The transducers API contains no mechanism for propagating out-of-band diagnostic information.
- The child processes implementing the transducers are not reaped by the main process. This may cause zombies to pile up, unless the main process manually calls `wait()` an appropriate number of times. Conceptually, all the child processes should be reaped once a transducer network finishes, but this is not possible in the current implementation, as the child PIDs are not stored anywhere, and the API design does leave us any obvious place to store them. A global variable would be inappropriate, as it would mean only a single transducer network could be constructed at a time.

5 Conclusions

I have implemented every part of the transducers library as specified in the assignment text. I judge the implementatio to be of adequate quality. The tests verify the correct operation of all API functions, as well as the salient error cases. The main limitations in the implementation are caused by the specified API itself, which we are unfortunately required to implement as given.