A4: Processes

Computer Systems 2017
Department of Computer Science
University of Copenhagen

Troels Henriksen

Due: Sunday, 12th of November, 23:59
Version 1 (October 30, 2017)

This is the fifth assignment on Computer Systems and the first on the topic of Operating Systems. As before, we encourage pair programming, so please form groups of 2 or 3 students. Groups cannot be larger than 3, and we strongly recommend that you do not work alone.

For this assignment you will receive a mark out of 3 points. You must attain at least half of the possible points to be admitted to the exam. For details, see the Course description in the course page. This assignment belong to the category of Operating Systems. Resubmission is not possible.

Introduction

Get a byte, get a byte, get a byte byte byte
— Dave Cutler, designer of the Windows NT kernel, on the Unix I/O model

This assignment is intended to give you an appreciation of Unix processes, and the use of *pipes* to communicate between processes. You will be implementing a C library for expressing trees of *stream transducers* (or simply *transducers*), where each transducer is intended to be implemented as a single process. The intent behind the library is to easily compose operations that are normally *blocking*, most notable file I/O operations, by exploiting the fact that they only block the calling process. In essence, our library can be seen as a generalisation of shell pipelines.

We will deal with the following concepts:

A *stream* is a potentially infinite sequence of bytes.

A *transducer* reads data from one or more existing streams, and produces output on one or more new streams. For simplicity, the only transducers that produce more than one result stream are built into the library.

A *source* produces a new stream. How this stream is produced can be arbitrary.

A *sink* reads a stream and produces nothing. The sink can perform arbitrary operations, in particular writing to files.

A *transducer tree* consists of one or more *sources*, which produce streams that flow through various *transducers* (or none), that finally end up in a single *sink*.

Source-, sink-, and transducer-*functions* are functions of a particular signature (see below) that can be used to create sources, sinks, and transducers, respectively.

For example, we might have a source that reads lines from a file, a transducer that reads lines from the source and writes those lines that match a specific pattern, and a sink that reads lines from the transducer, and writes them to the screen. This would in essence be the classic Unix program `grep`.

A transducer tree is said to be *finished* once its sink has finished executing.

Task 1: Implementing the API (40%)

The transducer API is already defined for you, in the form of a C header file (`transducers.h`). You are also given a stub of the corresponding implementation file (`transducers.c`), along with a `Makefile` and several test- and example programs that make use of the API. The code handout *compiles*, but it does not *run*. Your main task is to change the implementation file such that it also runs. You are *not* intended or allowed to modify `transducers.h`.

The code handout contains a `Makefile`. With this, you can run

```
$ make all
```

to compile all test- and example programs, and

```
$ make test
```

to run all tests. The list of test- and example-programs is defined at the top of the makefile:

```
TESTS=test0 test1
EXAMPLES=divisible
```

A test program is simply a program that terminates with exit code zero upon success, and nonzero otherwise. The test programs in the handout use `assert()` statements to accomplish this. You are adviced, but not required, to follow the same structure. Even if you modify how testing is done, `make test` should still have the effect of running your test suite (see Task 2).

The C API

A stream is defined as an *opaque type*, via a `typedef`:

```
typedef struct stream stream;
```

The definition of `struct stream` is not contained in the header file, but is instead located in `transducers.c`. All the public functions in `transducers.h` operate on *pointers* to `stream`, which means the user cannot directly create streams, or see their internals. This is very similar to the design of the file system interface in the C standard library, which makes use of an opaque `FILE` type.

Various API functions will produce streams. It is the users responsibility to eventually close them with the following function:

```
void transducers_free_stream(stream *s);
```

This is similar to the requirement for calling `fclose()` for files. A stream may only be freed once the transducer tree in which it is used has *finished* (see above).

Source-, sink-, and transducer-functions are ordinary C functions, which take parameters of type `FILE*` corresponding to the streams with which they interact. Therefore, the API functions accept *function pointers*. For example, a *source function* is defined as follows:

```
typedef void (*transducers_source)(const void *arg, FILE *out);
```

The type `transducers_source` is a pointer to a function that accepts two arguments: an arbitrary pointer, and a `FILE*` pointer representing the output stream of the source. The purpose of the pointer argument is to pass in arbitrary configuration information to the source. A source is created from a source function via the following API function:

```
int transducers_link_source(stream **out,
                            transducers_source s, const void *arg);
```

The source function must be passed the value of the `arg` parameter. The `out` parameter is a *pointer to a pointer*, and is used as follows:

```
stream *s;
transducers_link_source(&s, ..., ...);
```

The `transducers_link_source()` function will create a new `stream` object and write its address to the given pointer variable.

A sink function is defined and used in an API function as follows:

```
typedef void (*transducers_sink)(void *arg, FILE *in);

int transducers_link_sink(transducers_sink s, void *arg,
                          stream *in);
```

The `transducers_link_sink()` function blocks until the sink function returns. A sink must *not* be run in a new process, as it must be able to store its results (presumably via the `void *arg` argument) such that they are visible to the caller.

Two kinds of *transducer functions* are supported: one that takes a single stream input, and one that takes two. The two kinds are defined as:

```
/* A transducer that takes one stream as input.  */
typedef void (*transducers_1)(const void *arg,
                              FILE *out,
                              FILE *in);

/* A transducer that takes two streams as input.  */
typedef void (*transducers_2)(const void *arg,
                              FILE *out, FILE *in1,
                              FILE *in2);
```

Transducers are created from transducer functions via the following API functions:

```
int transducers_link_1(stream **out,
                       transducers_1 t, const void *arg,
                       stream* in);

int transducers_link_2(stream **out,
                       transducers_2 t, const void *arg,
                       stream* in1, stream* in2);
```

As with `transducers_link_source()`, the functions take a pointer to a pointer to store the address of the output stream.

The final API function duplicates a stream:

```
int transducers_dup(stream **out1, stream** out2, stream *in);
```

Details and Error Conditions

All sources and transducers (but not sinks) must be run *asynchronously*. That is, the API functions must return immediately, instead of waiting for the source- and transducer-functions to finish. You are intended to implement this by `fork()`ing a new process for sources and transducers.

All functions return 0 on success, and non-zero on errors. *You* are required to determine what can fail, and check whether it does. You are not required to propagate detailed information about what goes wrong.

A stream may only be connected to *one* transducer (or sink). An attempt to connect a stream that has already been used as the input to another transducer should cause the API function to return an error.

Remember the Unix principle of *silence is golden*. While you are encouraged to use debugging prints *during development*, your final implementation of the transducer library should *not* spew noise on the standard output channels.

Task 2: Write Test Programs (30%)

While sheer fun is reason enough to hack around with the Unix API, for this assignment you will also have to justify that your implementation faithfully implements the required interface. While the code handout contains some test programs, they do not exercise the entire interface. Your task is to determine which parts are missing, and implement tests for these. You are strongly encouraged to follow the style of the already present test programs.

In particular, you are also required to write *negative tests*. A negative test is a program that contains an error, and which succeeds only when the API implementation correctly detects the error.

In particular, the API mentions that a stream may be connected as input to only one transducer or sink. You *must* have a test program that verifies that this error case is correctly detected.

Task 3: Write a Short Report (30%)

Alongside your solution, you should submit a short report. The report should:

- Contain a general description of the problems solved.
- Include an overview of your design, including where memory is allocated (if anywhere), and how it is freed.
- Explain the purpose of every test program you have written.
- Describe how to compile your code and run your tests to reproduce your test results. You need a good justification if this differs from how to run the tests in the handout.
- Discuss the non-trivial parts of your implementation and your design decisions, if any.
- Explain *why* you wrote your code the way you did, not merely *what* it does. *Purpose is more important than mechanism.*
- Disambiguate any ambiguities you might have found in the assignment.

The above is not intended as a table of contents. Do not structure your report as answers to a series of bullet points. In fact, the bullet points above intentionally overlap, so mirroring their structure in the report will yield a repetitive text. Make sure there is a logical and textual *flow*, that you define terms before you refer to them, and that there is a good narrative.
The report is expected to be 2-3 pages and must not exceed 5 pages.

What to Hand In

Your submission must consist of a modification of the handed-out code, retaining the same structure, as well as a report in PDF format. The code handout consists of a directory `src`, which contains the following files:

`.gitignore`: A suitable .gitignore file, should you choose to use Git.

`transducers.h`: The header file for the stream transducer library. **Do not modify this file.**

`transducers.c`: A skeleton implementation of the library. It compiles, but every function fails.

`Makefile`: The file that configures your `make` program. You will have to modify this file when you add more test programs.

`test0.c`, `test1.c`, `divisible.c`: Two test programs and a larger example program.

Hints

- The definition of `struct stream` should contain a `FILE*` pointer and a flag indicating whether the stream already has a reader.

- The API functions `transducers_link_source()`, `transducers_link_1()`, `transducers_link_2()`, and `transducers_dup()` should use `fork()` to create a new process for running the given function. Pipes should be used for communicating between processes.

- Use the `file_pipe()` function, which uses concepts discussed at the lectures, for creating `FILE*` objects from pipes:

```
static int file_pipe(FILE* files[2]) {
  int fds[2];

  int r = pipe(fds);

  if (r == 0) {
    files[0] = fdopen(fds[0], "r");
    files[1] = fdopen(fds[1], "w");

    if (files[0] && files[1]) {
      return 0;
    } else {
      return 1;
    }
  } else {
    return r;
  }
}
```

- Solve the assignment incrementally. Of the two handed-out test programs, `test0.c` requires the fewest working API functions, and `test1.c` requires one more. *Don't* try to solve everything before you test it.

- Some internal error cases are not feasible to test (e.g, it is hard to provoke failures in `malloc()` or `fork()`). However, you *should* `assert()` the success of all operations that can fail and that you do not otherwise handle gracefully. While error recovery or propagation is ideal, merely being aware of *when* errors occur, and crashing immediately, is far superior to silently ignoring errors.

- *When* (not *if*) you use `valgrind` to locate memory leaks, use

  ```
  valgrind --child-silent-after-fork=yes
  ```

  to quiet irrelevant reports about leaks in child processes. The expected solution to this assignment will technically contain memory leaks in child processes, but they are harmless. (Explain why in your report.)

Submission

The submission should contain a file `src.zip` that contains the `src` directory of the handout, with a modified `transducers.c` and `Makefile`. Furthermore, it should include all new test programs you have written.

Alongside the `src.zip` containing your code, submit a `report.pdf`, and a `group.txt`. `group.txt` must list the KU ids of your group members, one per line, and do so using *only* characters from the following set:

$$\{\texttt{0x0A}\} \cup \{\texttt{0x30},\texttt{0x31},\ldots,\texttt{0x39}\} \cup \{\texttt{0x61},\texttt{0x62},\ldots,\texttt{0x7A}\}$$

Please make sure your submission does not contain unnecessary files, including (but not limited to) compiled object files, binaries, or auxiliary files produced by editors or operating systems.