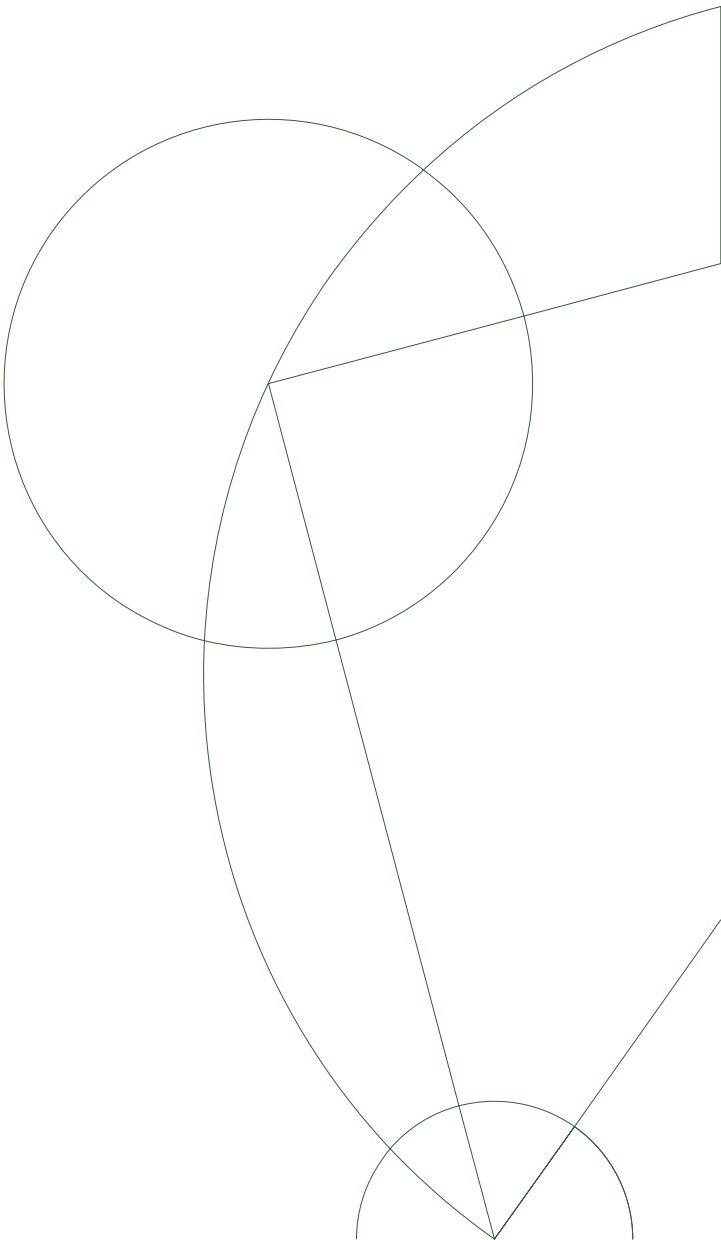
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Assignment 4: Processes

CompSys 2017
Department of Computer Science
University of Copenhagen

Peter, Tobias & Xueying

12. November 2017



1 Compiling and running the code

The `transducers`-library may be compiled and tested via a UNIX-like command line in the following way:

1. Unzip `src.zip` and navigate to `.../src/`.
2. Compile the library by running this command:
   `make transducers.o`
3. Compile all tests and the example by running this command:
   `make all`
4. Run all tests via this command:
   `make test`
5. Run the example via this command:
   `./divisible <n> <first divisor> <second divisor>`

2 Implementing the API

2.1 `transducer_link_source()`

This function creates a stream from a source. Since the function must return immediately, we use `fork()` to create a child process to handle the `transducer_source`-function. The result is then communicated to the parent via file pipes created using `file_pipe()`. It is then the responsibility of the parent to set the file pointer and availability of the stream.

In case either `file_pipe` or `fork()` fails, `transducers_link_source` will return 1, indicating a failure.

If the parent fails to return for unknown reasons, we to return 1 to indicate failure.

2.2 `transducer_link_sink()`

The sink consumes a stream, without creating a new one, by calling the supplied `transducers_sink`-function with the given argument and stream. Before calling the aforementioned function, we make sure to check whether or not the stream has already been used. If it has, the function fails.

2.3 `transducer_link_1()`

The purpose of this function is to read input from a stream and write an output to a new stream. Because of that it can work like a bridge for `transducer_link_source()` and `transducer_link_sink()`. It is similar to the source function, though it takes a stream as input. For this reason, we ensure that the stream is available. If it indeed is, we proceed with it. If not, we indicate failure by returning 1. Similar fail-mechanisms are in place to handle potential failures stemming from the calls to `file_pipe` and `fork()`.

2.4 `transducer_link_2()`

This function takes two streams as input, but creates only one new stream. The process is very similar to that of `transducer_link_1()`, although it takes one more argument: one more input-stream. The usual fail-mechanisms are in place to handle streams that are already in use (and the case where `in1` and `in2` refer to the same stream), as well as failures caused by `file_pipe` and `fork()`.

2.5 `transducers_dup()`

The function takes in a stream as input and creates two new identical streams. As one may expect, it is necessary to allocate memory for both new streams and establish two separate pipes to communicate between parent and child. In order to duplicate the contents of the input-stream, we use a while-loop to read read and write every single char from input and into each pipe. The parent may then read from each pipe and set the contents of the two new streams.

3 Test Programs

3.1 Test0

This test, tests the most basic features of the library: the `transducer_link_source()` and `transducer_link()`. This test was given to us.

3.2 Test1

This test, also test some of the most basic features as the `transducer_link_1()`. This test was also given to us.

3.3 Test2

The purpose of this, is to test the functionality of `transducer_link_sink()`, which is meant to take two streams as input and writes output to a new stream. This test should be similar to the previous one, therefore we added and modified some codes based on it. The `increment_stream` function is rewritten, as a result it can take two inputs and has two while-loops to run through them. In the main function, since two input and one output are needed, there are three streams that we are going to handle. The result after handling should be compared with an expected result which is a char array filled by the two inputs. There will not be any error message if the test is passed.

3.4 Test3

The purpose of the test, is to see if the `transducer_dup()` works as intended. It does this by duplicating a stream using `transducer_dup()` and then comparing the two streams to see if it has copied correctly.

3.5 Test4

In test4 we use several of our library functions together to ensure that they all work as intended when interacting with each other. Although `transducers_link_1` is not included, test4 covers the remaining functions and we consider this to be sufficient.

3.6 Test5

The test is a negative test, that ensures that `transducers_link_sink()` fails when it tries to read from a stream that is already being used.

3.7 Test6

The test is a negative test, that ensures that `transducers_link_1` fails when it tries to write to a stream that is already being used.

3.8 Test7 & Test8

Both test7 and test8 are almost identical. The purpose of either is to ensure that `transducers_link_2` fails when either of the input streams are already in use.

3.9 Test9

The test `transducers_link_2` checks whether or not it fails correctly if it reads from two identical streams. Recall that a stream may only be used once.

3.10 Test10

Test 10 is a negative test and the purpose of the test, is to see if the `transducer_dup()` fails as intended. The `transducer_dup()` should not be able to duplicate a stream that is being used.

4 Memory leaks

We were unable to identify any memory leaks using `valgrind`. Looking at the heap summary however, one notices that not all the memory blocks allocated are being freed. It seems that our library may sill be improved in that regard.

Memory leaks in child processes (zombie children, following our termination of the child processes) are irrelevant to report on, since they use virtually no resources, besides taking up a `pid`. Given more time, we may have implemented asynchronous reaping of the zombie processes.

```
==2014== Memcheck, a memory error detector
==2014== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2014== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2014== Command: ./divisible.c
==2014==
./divisible.c: 1: ./divisible.c: /bin: Permission denied
==2015==
==2015== HEAP SUMMARY:
==2015==     in use at exit: 2,425 bytes in 26 blocks
==2015==   total heap usage: 31 allocs, 5 frees, 38,329 bytes allocated
==2015==
==2015== LEAK SUMMARY:
==2015==    definitely lost: 0 bytes in 0 blocks
==2015==    indirectly lost: 0 bytes in 0 blocks
==2015==      possibly lost: 0 bytes in 0 blocks
==2015==    still reachable: 2,425 bytes in 26 blocks
==2015==         suppressed: 0 bytes in 0 blocks
==2015== Reachable blocks (those to which a pointer was found) are not shown.
==2015== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==2015==
==2015== For counts of detected and suppressed errors, rerun with: -v
==2015== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 1 from 1)
./divisible.c: 2: ./divisible.c: given: not found
: not found.c: 3: ./divisible.c:
: not found.c: 5: ./divisible.c:
: not found.c: 10: ./divisible.c:
./divisible.c: 11: ./divisible.c: Syntax error: "(" unexpected
==2014==
==2014== HEAP SUMMARY:
==2014==     in use at exit: 889 bytes in 23 blocks
==2014==   total heap usage: 46 allocs, 23 frees, 77,865 bytes allocated
==2014==
==2014== LEAK SUMMARY:
==2014==    definitely lost: 0 bytes in 0 blocks
==2014==    indirectly lost: 0 bytes in 0 blocks
==2014==      possibly lost: 0 bytes in 0 blocks
==2014==    still reachable: 889 bytes in 23 blocks
==2014==         suppressed: 0 bytes in 0 blocks
==2014== Reachable blocks (those to which a pointer was found) are not shown.
==2014== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==2014==
==2014== For counts of detected and suppressed errors, rerun with: -v
==2014== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 1 from 1)
```

Figure 1: Example of using valgrind for `divisible.c`.