Assignment 4

Computer Systems
Department of Computer Science
University of Copenhagen

Pelle Rubin Galløe <hqr241@alumni.ku.dk>,
Mads B. Christensen <rsv422@alumni.ku.dk>,
Julian S. W. Wulff <bkx655@alumni.ku.dk>

Version 1;
Due: Monday, November 17

1 Introduction

This is a technical report for Computer Systems B5-6, 2019. In this project we have been assigned to implement a C library for expressing trees of stream transducers. Furthermore the report contains a section about virtual memory where we answer the questions asked in the assignment. The target audience for this report is the teachers assistants of the course. The solution is developed in `C`, with a set of restrictions defined in the assignment.

As part of the deliverables is a `Makefile` containing macros for compiling the C library, the test programs, and running the test programs.

Compiling the C library can be done by running the following command in the terminal:

```
$ make transducers.o
```

Compiling the C library and all of the test programs can be done running the command:

```
$ make all
```

Having compiled the code the test programs can be run with the following command

```
$ make test
```

2 Implementation

In order to achieve the desired functionality of the proposed stream transducer tree, our C library contains half a dozen of different functions that in combination can be used to produce such a tree.

Since our implementation works of streams potentially containing an infinite sequence of bytes, streams are defined as `struct` type containing a `FILE*` pointer and a flag indicating the stream has a reader, in order to protect the location of the read pointer. We have chosen to implement this flag as an `int` when set to 0 indicating that it does not have a reader and when $! = 0$ indicating it has a reader.
As the backbone in our implementation we use the function `file_pipe`, discussed at the lectures. The purpose of this functions is to create pipes that are used for communicating between processes and create the file object used for reading the data of a stream.

We have implemented the header file in the `transducers.c` file. We have written the non-implemented functions and struct. We have written 3 tests that each test different functionality of our code. We allocate memory in the functions `transducers_link_source`, `transducers_link_1`, `transducers_link_2` and `transducers_dup`. We free this memory in the `transducers_free_stream` function. When we run our tests with the `valgrind` command, we detect no memory leaks.

2.1 transducers.c

The file `transducers.c` is where we implement the structure. The header file `transducers.h` is our template for the different functions. To implement the most basic functionality we define our `stream struct` and implement the functions: `transducers_free_stream`, `transducers_link_source` and `tranducers_link_sink` [1]. The three functions: `transducers_link_1`, `transducers_link_2` and `transducers_dup` is what enables the tree structure functionality.

The `stream struct` has (as described above) a `FILE*` and an `int` such that a stream contains a stream of bytes and a flag that indicates if it is in use.

The `transducers_link_source` function should create a source and a stream for its output (as described in the header file). We start by creating a pipe using two file pointers. If this fails we return 1. If the pipe is successful we fork our process. The child process creates the source and writes to the "w" end of our pipe. The parent process reads from the "r" end of our pipe, allocates memory for a stream, and initialize the stream. We then return 0.

The `transducers_free_stream` function must handle terminating a stream and freeing the allocated memory. We do so by closing the `FILE*` and setting the flag to 0 and then freeing the allocated memory using `free(s)`

[1]This allows us to run test0 which tests for this exact basic functionality.

The `tranducers_link_sink` should read input from a given stream. If the stream is already in use (ie. flag == 1) we return 1. Otherwise we set the flag to 1 and pass the arguments and the FILE* to the sink. And then return 0.

The `transducers_link_1` function reads input from one stream, parses it through a transducer and produces a new stream for its output. We check if the input stream is available and if we successfully created a pipe. If the stream is not available or we are unable to create the pipe, we return 1. Otherwise we fork our process. In the child process we parse the input from our stream through our transducer and write it to the "w" end of our pipe. In the parent process we allocate space for a new stream. Initialize its parameters, and set the output to be the pointer to this new stream. Lastly we return 0.

The `transducers_dup` function reads input from one stream and copies every byte read onto two new output streams. Again we check that our input stream is available, if not we return 1. We then try to create two pipes. If either pipe fails we return 1. If not we fork our process. In the child process we loop while as long as there is bytes to read in our input stream. In each loop, we read a byte and write it to the "w" end of both our pipes. In the parent process we allocate memory for each of the output streams. Initialize them with the correct pointer and value. We set our outputs to the newly created streams, set the flag of the input stream to 1 and return 0.

The `transducer_link_2` function reads input from two streams, parses them through a transducer and writes the output to an output stream. We check if both input streams are available, if not we then return 1. If they are both free, we try to create a pipe, if unsuccessful we return 1. Otherwise we fork our process. In the child process we parse the input through our transducer and write the output to the "w" end of our pipe, and then exit. In the parent process we allocate memory for our stream. Initialize its parameters and set our output to be the pointer to this new stream. Lastly we return 0.

2.2 Testing

In order to check to correctness of our implementation we have a couple of test programs. All of the test can be run as described in the introduction by running the command `make test`. In order to make sure that all of our functions in the library is working correctly we have made sure to test all of the functions in at least a couple of times in different settings. Furthermore we made sure not only to assert that the functions return 0 for correct behavior, we also made sure to make negative tests asserting they return 1 for example when we try to read from a stream that already has a reader.

The two test `test0` and `test1` is both given with the assignment. `test0` test for the basic functionality of a source, stream and sink [2]. The most fundamental functionality needed. It creates a stream and a source and sinks said stream. `test1` tests for a very basic tree-like structure, with only one branch, using the `transducers_link_1` function.

`test2` tests a tree structure with two branches, using `transducers_dub` and sinking each branch.

`test3` tests the functionality of our stream flag, to assert if it is being correctly flipped. This is done by calling the `transducers_link_1` function twice in a row with the same stream.

`test4` tests a larger tree structure with multiple branches and the `transducers_link_2` function (in addition to the other functions previously tested).

All of our test run and return as expected and therefore we a pretty confident that our implementation is working as supposed. We have also tried running the `divisible` program with alot of different arguments which also gave us correct result. Furthermore we also used `valgrind` on all our test programs in order to check for memory leaks, and all our test programs return with "All heap blocks were freed – no leaks are possible".

3 Virtual memory

Virtual memory is an abstraction of the main memory used in modern computers. This abstraction acts as a tool to manage the memory more efficiently. Most importantly virtual memory provides three important capabilities:

1. Tool for caching.
   As a tool for caching, virtual memory makes it possible to use the main memory more efficiently by treating it as a cache for an address space stored on a disk. Hereby only keeping the active parts of the active process in the main memory. Since virtual memory makes use of the disk, virtual memory makes it possible to utilize more memory than is actually in the physical memory.

2. Tool for memory management.
   By providing each process with a uniform address space it makes memory management easier. Furthermore it also makes it easier for process'

[2]It also uses the `transducers_free_stream` but even though it is important, it is not critical to the functionality.

to share data in conjunction with capability to use virtual memory as a tool for memory protection.

3. Tool for memory protection.
   With the use of virtual memory it provides the ability to protect different address spaces of different processes in a number of ways, making some parts readable, writable or executable by other process. Hereby making it easier and more secure for processes to share data.

Since virtual memory works by paging memory from the main memory to a secondary storage (typical a disk) and vice versa, you could ask yourself, is virtual memory is useful with out a disk?

Looking over the capabilities that virtual memory provides it is obvious that virtual memory is still useful, since virtual memory still works even without a disk. Without a disk you would still have all the benefits and capabilities mentioned above, that virtual memory provides. Without a disk though you would be limited by the amount o physical memory available since it is not possible to page to a disk for later use.

As mentioned earlier virtual memory is used as a tool for caching and uses a secondary storage for this mechanisms to allow for a greater memory space than the physical memory allows for. It is actually pretty common for the virtual memory space to be large than the virtual memory space. One of the reason for this that physical memory is more expensive than disks. With the use of virtual memory and locality it is actually not even that ineffective to have virtual memory that is bigger than the physical memory. For this reason it is very useful to allow for virtual memory space that are bigger than the physical memory space such that you are not limited by the physical memory space.
Vice versa it is not very common for the physical memory space to be large than the virtual. In practice it is possible that the virtual memory space can be smaller than the physical. If that case it means that all the virtual memory can be keped in the physical memory. This can be useful if efficiency is important. Since it very slow when data is transferred between the disk and main memory you make sure to never use the disk with a virtual memory space smaller than the physical hereby running the processes faster.

Considering a demand-paged system with the following time-measured utilization:

CPU utilisation 10%

Paging disk 97.7%

Other I/O devices 5%

We see that the bottle neck in this system is the utilization of the paging disk. In order to improve the CPU utilization it is necessary to make the CPU receive more data from the main memory to work on. Since we the 97.7% utilization of the paging disk it would seem the CPU utilization is stalling from data being ready in the main memory. It would therefor not make sense to install a faster CPU since this is not the bottle neck and we don not even utilize the full CPU. Installing a bigger paging disk would neither improve the CPU utilization since it having the data ready in the main memory that would improve the CPU utilization. Instead of installing a bigger paging disk, installing a faster paging disk would properly improve the situation. Installing a faster paging disk would mean that the main memory would be able to retrieve data from the paging disk faster, hereby serving it quicker to the CPU and improving its utilization. But a bigger improvement would probably be to install more memory. By installing more main memory it would reduce the load on the paging disk and increasing the amount page hits making the CPU able to work longer without having a to wait on page faults. Lastly, depending on the reason for the high utilization of the paging disk increasing the degree of multiprogramming could also improve the CPU utilization. If the low degree of CPU utilization is due to a lot of page faults increasing the degree of multiprogramming could improve the CPU utilization making the CPU able to be utilized in other process that are not stalling.