UNIVERSITY OF COPENHAGEN

A4 - Computersystemer

Mads Kronborg - `xlq446`
Niklas Lohmann - `jgs746`
Christian Baun - `rbp111`

November 11, 2018

The functionality of the code

The purpose of the transducer API is to contain the different functions needed to create the tree-like shape of processes – known as a transducer-tree.
Such a tree contains the sources, which feed their output to a stream. The transducers can for example take one or more streams of bits and use these to produce one or more new streams of bits. The last stop in the entire process is the sink, which takes an input stream and produces a "real" output, instead of just passing a stream along.

Now all these functions are connected to each other through the streams, which is data being sent through the different transducers in the tree. There can be multiple transducers and streams, but the tree can only have one sink.

Overview of design and reasoning for the way we wrote the code

The structure of our code is rather obvious since the overall architecture and procedure was in the assignment text. We have our struct `stream`, which contains two values, a file pointer and an integer, that is used as a boolean flag.

Instead of going through each and every transducer, we will instead discuss the functions we use the most.

`fork()` is the most important function and its purpose is to create a new child process which can run simultaneously with its parent. This will hypothetically allow us to do two things at once, and is very useful for many things. An example could be `transducers_link_1()` where the child transduces while the parent is allocating the necessary memory. Many of the file handling functions are also present, `fclose`, `fread`, `fwrite` etc., which work with the file pointer in the stream struct.

Testing

Our test suite consists of two positive tests and two negative tests, besides the program `divisible` that tests all linkers and more some.
We used `valgrind -child-silent-after-fork=yes` to continuously test for memory leaks.

Test 0

For test 0 to succeed, both the `transducer_source` and `transducer_sink` needs to return 0, and still generate the expected output.

Test 1

For test 1 to succeed, both `transducer_source`, `transducer_link` and `transducer_sink` needs to return 0 and still generate the expected output.

Negate 1

In this negative test we give `transducer_link_1` a used stream as the input and see if it fails.

Negate 2

In this test we give `transducer_sink` a used stream as the input and see if it fails.

Divisible

The handed out program `divisible` is a great way to test all functions from `transducers.c`.

How to compile our code and test programs

Everything is compiled with the makefile.

```
$ make all
```

Test programs are run using the makefile, but with the `test` argument.

```
$ make test
```

How to run `divisible`.

```
$ ./divisible 10 1 1
```

Task 4: Virtual memory

Abbreviations:
VM = virtual memory
PM = physical memory

1. Is virtual memory useful without a disk? Why?

We understand ‘disk’ as ‘disk memory’ and thus opposed to main memory (DRAM). So the question is actually is virtual memory useful with only DRAM and no disk?
In this case we cannot have any page faults and swapping were data is swapped between disk and main memory. We assume the whole program is read in or cached in main memory at start. Thereby as long as the sum of the working sets are less than the size of main memory VM is still useful.

Virtual memory will still provide its three advantages:

• Effective use of memory space, e.g. with VM sharing physical memory space (for e.g. libraries) and better spacial and temporal locality.
• Applications and processes can work in a standardized and linear memory space instead of a complex shared memory space.
• Protection of physical memory, because it is hidden behind a virtual memory abstraction.

2. Can the virtual memory memory space be larger than the physical memory space? Is this common or useful?

Yes, VM can be larger and it is common. But it is perfectly accepted to have a larger PM than VM. The VM is usually set to a large space, but notice that not all of this space is used. It is just a potential. Therefore it is also a good idea with an advanced datastructure e.g. multi-level page tables.

It is useful because you have a larger address space to use in your VM. For instance Linux processes let address 0x400000 (for 64-bit address spaces) be the lowest address. This seems like a waste, but helps to avoid accidental pointers even if they jump far in the memory.

3. Can the physical memory memory space be larger than the virtual memory space? Is this common or useful?

Yes, PM can be larger than VM, but it it is not common.

This is useful if the system runs a lot of small processes. If the VM is smaller, then each process can only take advantage of a minor part of the PM.

4. Consider a demand-paged system with the following time-measured utilization's:

CPU utilization 10%
Paging disk 97.7%
Other I/O devices 5%

Which of the following would likely improve CPU utilization?

Install a faster CPU?

No, this would not increase CPU utilization. We need more instructions for the CPU to increase utilization.

Install a bigger paging disk?

No, the size of the disk does not affect the CPU utilization.

Install a faster paging disk?
Yes, a faster paging disk would help a little. All the swapping between main memory and disk would process faster.

Install more main memory?

Yes, with more main memory you will increase the number of page hits and decrease the number of page faults. Thus the frequency with which the CPU receives instructions would increase and its utilization would go up.

Increase the degree of multiprogramming?

No, more multiprogramming will give more processes and thus less memory space per process. This will give more page faults and thereby make the CPU wait more.