CS2002 Week 11: SP - Multi-process Pipeline

Jon Lewis (jon.lewis@st-andrews.ac.uk)

Due Wednesday 21st April, 21:00 – 25% of Continual Assessment
MMS is the definitive source for practical weighting and due date/time

Objective

To gain experience with Processes and simple Inter-process Communication (IPC) in C.

Learning Outcomes

By the end of this practical you should be capable of:

- writing a program that creates multiple processes in C
- using pipes for inter-process communication
- developing and debugging programs to check correct operation of your implementations

Overview

In this practical you will design and implement a simple data processing pipeline framework. A data processing pipeline can be visualised as a number of processing stages which are connected in series, conceptually execute in parallel, and in which the output from one stage forms the input to the next pipeline stage. The framework you are to develop should support an arbitrary number of stages, execute each pipeline stage as a separate process and use pipes for inter-process communication between the pipeline stages.

Getting started

To start with, you should download and decompress the `zip` file at

`http://studres.cs.st-andrews.ac.uk/CS2002/Practicals/W11-SP/code.zip`

Please note that the `zip` file contains a number of files in the `src` directory, some of which are blank or only partially implemented. **Take care that you do not accidentally decompress it again once you have worked on your assignment, thereby accidentally overwriting your own implementation.**
Two sample applications are supplied in the zip file and show the functionality expected from your processing pipeline implementation. The first SumSquaresPipeline application demonstrates how it should be possible to compute the sum of squares using three inter-connected pipeline stages. The first stage generates integers and passes them on to the next stage which squares them and passes the squares on to the last stage which sums all the values. The second application supplied shows how it should be possible to use your pipeline implementation to compute prime numbers. The sample SievePipeline application uses a number of inter-connected pipeline stages to create a parallel version of the the Sieve of Eratosthenes.

`http://en.wikipedia.org/w/index.php?title=Sieve_of_Eratosthenes&oldid=544116469`

The Sieve of Eratosthenes can be viewed as a series of filters or pipeline stages. The first stage generates the sequence of integers from 2 up to a specified upper bound and creates a specified number of filter stages. Each filter stage looks at the first number it sees, and treats that as a prime. Then, it removes every multiple of that number it sees by only passing on inputs it receives from previous pipeline stages that are not a multiple of that prime number. With P filter stages you can compute the first P prime numbers within the specified range.

The Pipeline

You are required to design, implement and test a multi-process pipeline framework in accordance with the module interface in `Pipeline.h`. As usual, coding to an interface is important, so please don't change the function signatures that are provided.

You are supplied with a `Pipeline.c` file containing the functions which do not do anything sensible yet and which you will have to implement. You are also provided with `SumSquaresPipeline.c` and `SievePipeline.c` files which show how it should be possible to use your pipeline implementation to compute the sum of squares and prime numbers respectively as indicated above.

As the number of pipeline stages an end-user may desire may not be known at compile time, you are supposed to use dynamic memory allocation and pointers to structs in your implementation. You may find material in C_SP Lectures 7 - 10 useful, including the Person example `L07-08/PersonStructDynamic`. You will have to define a suitable Pipeline struct in `Pipeline.h` which can hold a collection of functions representing the entry points for each of the pipeline stages in a specific pipeline. You can use a dynamically allocated array or a linked list for the collection, it is up to you.

The entry points for each of the three pipeline stages in `SumSquaresPipeline.c` are `generateInts`, `squareInts` and `sumIntsAndPrint`. You will notice that each of these functions takes in two arguments representing the input and output file descriptors which the stages should be able to use for communication. You can assume that the first pipeline stage will not require input from an incoming pipe and that the last stage will not need to write to an outgoing pipe. All other stages will most like make use of both incoming pipes (to read data from the previous stage) and outgoing pipes (to pass data to the next stage).

The entry points for the pipeline stages in `SievePipeline.c` are `generateInts` which generates the number sequence and `sieveInts`. The latter is used for each of the sieve pipeline stages to sieve out multiples of the prime number the stage received as its initial input.

The intention is that complexity of creating a pipeline, the associated processes and pipes for communication should be borne by your Pipeline implementation rather than by the application programmer, i.e. the end-user of your Pipeline. As such, the `main` function in `SumSquaresPipeline.c` is fairly short and shows how it should be possible to use your Pipeline implementation to create a pipeline, add the three stages and execute the pipeline. In the `main` function in `SievePipeline.c` you can see how a number of sieve pipeline stages are added each with `sieveInts` as the specified entry point.

You should examine the function declarations and comments for each function in `Pipeline.h` to give you an insight into how you should implement your Pipeline, what elements you should add to your `struct Pipeline`, and what the expected behaviour of the functions is.

For example, `Pipeline_add(Pipeline* this, Function f)` takes a `Pipeline*` argument (i.e. a pointer to a `Pipeline` struct) representing the pipeline on which to operate, and a pointer to a user-defined entry function (representing the pipeline stage) that should added to the pipeline.

The `Pipeline* new_Pipeline()` "constructor" function should return a pointer to a new, suitably initialised `Pipeline` struct for which memory should have been allocated dynamically.

The `Pipeline_execute` function is undoubtedly the one which will require most thought. The function takes a single `Pipeline*` argument representing the pipeline on which to operate and will have to create processes such that each pipeline stage has its own process. Pipes will have to be created for communication between the stages, and the relevant input and output pipes' file descriptors must be passed to the entry functions for each stage in the correct process. You should study lectures 13 & 14 and may find examples useful, especially the `L14/fork_pipe.c` example to give you an insight into using `fork`, `pipe`, `wait` and `exit`.

When thinking about constructing the chain of inter-connected processes and pipes, it may help to devolve the work into separate helper functions that are executed by parent and/or child process at each stage of creating the chain. In order to create a chain of processes, think about

- whether it makes sense for the child or the parent at each stage to call `fork`
- when `pipe` should be called
- when calls to `fork` should cease
- whether the child or the parent at each stage should call the user-defined entry function
- how you can keep track of the incoming pipe-end from the parent and outgoing pipe-end to the next child at each stage

The first and last stages do not need an incoming and outgoing pipe respectively. Every parent process should `wait` for its child to exit.

As mentioned above, `SumSquaresPipeline.c` and `SievePipeline.c` represent two possible end-user applications of your pipeline and these can be used to test your Pipeline implementation. However, you may during design and implementation decide to create simple sample applications of your own as well, as mentioned in the "General Tips" section below. Instructions on "Compiling Testing and Running" are given in the following section.

Compiling and Running

There are no stacscheck tests for this assignment. However, you are supplied with a `Makefile` to help you build your code and some applications that use your Pipeline. The following lines, executed from a Terminal window and from within the `src` directory, should permit you to build and run the sample SumSquaresPipeline and SievePipeline applications that use your own Pipeline implementation.

```
$ make
clang -g -Wall -Wextra -c SumSquaresPipeline.c
clang -g -Wall -Wextra -c Pipeline.c
...
$ ./SumSquaresPipeline < squares.in
Setting up pipeline to calculate the sum of squares of integers 1 to 10.
generateInts: process 493030, parent 492799
squareInts: process 493031, parent 493030
sumIntsAndPrint: process 493032, parent 493031
sumIntsAndPrint: result = 385

$ ./SievePipeline < sieve1.in
Setting up pipeline to sieve out the first 5 prime numbers up to 15
generateInts: process 493071, parent 492799
sieveInts: process 493072, parent 493071: prime = 2
sieveInts: process 493073, parent 493072: prime = 3
sieveInts: process 493074, parent 493073: prime = 5
sieveInts: process 493075, parent 493074: prime = 7
sieveInts: process 493076, parent 493075: prime = 11
```

Lines pre-fixed with a `$` symbol above show the commands to execute. Note that for the pipelines each stage (process) prints out its process ID and its parent's process ID. The process IDs you see when running your program are not likely to match the ones shown above and will change each time you run the program. However, the final result for sum of squares and the primes should match. Also note that for a correct pipeline setup each stage (process) in the pipeline (apart from the first one) has the previous stage as its parent.

When trying your implementation out, you are advised to find a lab client machine that isn't being used by many people. After connecting to a lab client from the list at

`https://systems.wiki.cs.st-andrews.ac.uk/index.php/Lab_PCs#Remotely_Accessible_Linux_Clients`

you can type `who` in the Terminal window to find out who is connected to the machine and choose a different one if it looks busy. Otherwise your program will most likely run and exit slowly and could cause problems for other users as well.

While you are developing your code and debugging it (with print statements or a debugger) things may go wrong and some processes may not terminate correctly even if you type `ctrl-c`. As such, you should check after running your program whether stale processes with name `SumSquaresPipeline` or `SievePipeline` are still on the system and kill them off. You can do this for example by

```
$ ps ux | grep Pipe
jonl      493720  0.0  0.0   4336    84 pts/0    SN   13:50   0:00 ./SievePipeline
jonl      493721  0.0  0.0   4336    84 pts/0    SN   13:50   0:00 ./SievePipeline
jonl      493722  0.0  0.0   4336    84 pts/0    SN   13:50   0:00 ./SievePipeline
jonl      493723  0.0  0.0   4336    84 pts/0    SN   13:50   0:00 ./SievePipeline
jonl      493724  0.0  0.0   4336    84 pts/0    SN   13:50   0:00 ./SievePipeline
jonl      493725  0.0  0.0   4336    84 pts/0    SN   13:50   0:00 ./SievePipeline
...
```

and if you see such processes still on the system, you can kill them all off in one go by issuing a command such as

```
$ killall SievePipeline
$ killall SumSquaresPipeline
```

If you wish to build your project using `gcc` instead of `clang`, you can alter the first line in the supplied makefile from `CC = clang` to `CC = gcc`.

General Tips

- At the start, you will have to choose a suitable struct to represent your Pipeline and will have to ensure you can allocate memory for the Pipeline and its elements. You should probably write some simple programs to check operation at this stage before trying to execute the pipeline.
- Don't try to develop the whole implementation in one go as this will probably result in more bugs and lengthier, more painful debugging. Instead, it may be sensible to develop your implementation in stages, possibly creating some prototypes along the way. For example, you might write a program which can create two processes where the parent sends an integer over a pipe to its child, which prints it out. Subsequently, you might create three processes where the number is passed from parent to child one, which doubles it, sends it on to child 2, which finally prints it out.
- Once that is working, you might start thinking how you might create a number of processes that communicate using pipes.
- Only *free* things you have *malloc*-ed, or else bad things happen.

- Always check the manual pages for the functions you call.
- Check POSIX function return values – checking the return values is very important and a good way to cut down on debugging time.

Deliverables

Hand in via MMS, by the deadline of 9pm on Wednesday of Week 11, a zip file containing:

- Your assignment directory with all your source code for implementation and tests.
- A PDF report describing your design and implementation, testing, any difficulties you encountered, how you tested your implementation. Take care to explain and justify your design and implementation decisions in clarity and detail.

Marking Guidance

The submission will be marked according to the general mark descriptors at:

`https://studres.cs.st-andrews.ac.uk/CS2002/Assessment/descriptors.pdf`

A very good attempt achieving almost all required functionality, together with a clear report showing a good level of understanding, can achieve a mark of 14 - 16. This means you should produce very good code with very good decomposition and testing and provide clear explanations and justifications of design and implementation decisions in your report. To achieve a mark of 17 or above, you will need to implement all required functionality. Quality and clarity of design, implementation, testing, and your report are key at the top end.

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof): `http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`

Good Academic Practice

As usual, I would remind you to ensure you are following the relevant guidelines on good academic practice as outlined at

`https://www.st-andrews.ac.uk/students/rules/academicpractice/`