Assignment 3: Processes and Pipes

Extremely Important!

You must not leave processes lying around on any of the machines you work on, or the machines will eventually become unusable. **Always**, before you log out, and frequently while you are working, use `ps aux | grep <username>` or `top` to check if you have any stray processes still running. Use `kill` to kill them. You may find `killall pfact` on Linux to be useful.

Be extra careful in the early stages of developing your program. Do not test large values of *n* until you are confident that your program is working.

If you notice that other students have processes that have been around for a long time, please post a note to the discussion board with their user names, and ask them politely to clean up their processes.

Overview

Your task in this assignment is to write a program that determines whether a number N can be factored into exactly two primes, and if it can be factored, identifies the values of the two factors. This task of finding the two prime factors for a number is interesting and important for public key crytography but your program certainly won't find factors for the large RSA numbers that are interesting from a cryptogrphy perspective. In fact, the algorithm you are asked to implement is not at all efficient. However, it lends itself to parallelism and will allow you to practice using unix pipes and processes.

The algorithm you will implement is based on the Sieve of Eratosthenes, which itself is an algorithm to identify prime numbers.

The Sieve of Eratosthenes

The idea behind the Sieve of Eratosthenes is to use a set of consecutive filters. Each filter takes a list of numbers as input and removes (or filters out) some of them, so that the output is a smaller list. Each filter is related to a *prime number*, which is a natural number greater than 1 that can be divided without remainder only by itself and 1. The job of the filter related to some prime *m*, is to remove all multiples of *m*. For example, the filter for m=3 would remove any of 3,6,9,12,15, ... that appear in the input. The filters in the sieve are related to the prime numbers in increasing order. So they are m=2, m=3, m=5 etc. If the sieve's input is the list of integers from 2 to `n`, the basic sieve will have filters for all the prime numbers between 2 and `n`. See below for an example sieve for `n = 14`.

```
2 3 4 5 6     m=2             m=3              m=5             m=7          m=11        m=13
7 8 9 10      ==>   3 5 7 9   ==>  5 7 11 13   ==>  7 11 13   ==>   11 13  ==>   13   ==>
11 12 13 14           11 13
```

Notice that the filter for m=3, removes all the multiples of 3 that occurred in the input. Although 6 is a multiple of 3, it was already removed by the filter for m=2.

How does this help you find the prime numbers? Didn't you have to know the primes already in order to know what filter to create next? The very cool observation that makes this all work is that once the list of numbers have passed through some filters, the smallest number remaining in the input list will be the next smallest prime. Convince yourself that this is true. Perhaps even prove it on a scrap of paper.

Now, notice something else important. As soon as the filter is equal to or larger than the square root of `n`, the remaining input numbers must all be primes. Again, convince yourself of why this must be true.

So to find all the prime numbers less than or equal to 14 using the sieve, we would start with a list of integers from 2 to 14, see that 2 is prime, make a filter for 2 and remove multiples of 2, see that 3 is the smallest number remaining, make a filter for 3, see that 5 is the smallest remaining and notice that it is larger than the square root of 14. In order to find all the primes less than or equal to 14, we don't create the filter for 5, we simply return the values we made filters for (2 and 3) and append the outputs from filter 3 [5,7,11,13].

Using the Sieve to Find Prime Factors

Make sure you fully understand the algorithm of the sieve above before reading on. In the assignment, however, we won't simply implement the sieve. Instead we will use the **idea** of the sieve to determine if a number is prime, if it has exactly two prime factors (that might be the same if it is a square of a prime), or if it is composed of more than two prime factors. Here is the algorithm you must implement:

- Create filters starting with `m=2` and use the list of integers from 2 to `n` as the initial input.
- Each time **before** you create a filter for value `m`, check to see if `m` is a factor of `n`. Keep track of the factors you have found.
- Stop creating filters when `m` would be greater than or equal to the square root of `n`. Don't create a filter for the square root if `n` is a perfect square.
- If you find no factors, then `n` must be prime.
- As soon as you discover two primes that are factors, and both are less than the square root of `n`, you can conclude that `n` is not the product of two primes. In this case, you should stop creating filters immediately.
- If you find exactly one factor between 2 and square root of `n` (inclusive), then `n` may be the product of two primes. Be careful here. What happens if `n` is a cube? Consider the case of `n=8` or `n=52`.

Your Program

To implement the above algorithm, you must use Unix processes and pipes in a C program called `pfact.c` (with executable `pfact`) that takes the value for `n` as a command-line argument. If the program is run with an argument that is not a positive integer or with the wrong number of arguments, use the following statement to print a message to `stderr` and exit with a termination code of 1:

```
fprintf(stderr, "Usage:\n\tpfact n\n");
```

Each filter will be implemented as its own unix process and the numbers will pass from filter to filter using a pipe. Although we talk about a filter receiving a "list" of numbers, the input will not come as an array. Instead, a process that wants to "pass" a number to the next filter will write it to the pipe connected to that process.

You must follow the algorithm given above. Pay particular attention to the fact that you must check if `m` is a factor (and possibly exit) before you fork the process to filter for `m`.

Additionally, the termination code (exit code) of each process will be one plus the number of processes (filters) that follow it. On successful completion, the **original** `pfact` process will print the total number of processes that were forked and exit with a zero termination code.

There are three possible outcomes, and each case listed below includes a `printf` statement that you **must** use to print the results. You are welcome to use different variable names but the formatting must be exactly what is given.

1. n is the product of exactly two primes :
   `printf("%d %d %d\n", n, factor1, factor2);`

(where `n` is the number to be factored, and `factor1` and `factor2` hold the two factors of n and `factor1` ≤ `factor2`)

2. n itself is prime:
   `printf("%d is prime\n", n);`
3. n is not the product of exactly two primes:
   `printf("%d is not the product of two primes\n", n);`

The master process collects the exit status of its child which is expected to be the number of filter processes created, and uses the following statement to print a message:

```
printf("Number of filters = %d\n", result);
```

Here are some sample runs of the program:

```
$ pfact 15
15 3 5
Number of filters = 2
$ pfact 311
311 is prime
Number of filters = 7
$ pfact 25
25 5 5
Number of filters = 2
$ pfact 72
72 is not the product of two primes
Number of filters = 1
$ pfact 125
125 is not the product of two primes
Number of filters = 5
```

One ~~Required~~ Optional Helper Function

`int filter(int m, int read_fd, int write_fd)`

You ~~must~~ may write (and make use of) a function named `filter` which takes three arguments: the filter value `m`, a file descriptor `read_fd` from which integers are received, and a file descriptor `write_fd` to which integers are written. Its purpose is to remove (filter) any integers from the data stream that are multiples of `m`. The function returns 0 if it completes without encountering an error and 1 otherwise.

`filter` should be the only function in `filter.c`. Any other helper functions that you write for this assignment should be in pfact.c

IMPORTANT: It is important that filter not `fork()` or `close()` any file descriptors. The creating of any child processess and the cleaning up of any file descriptors should happen in the code that calls `filter`.

Details and Tips

If including `filter`, make sure that your code compiles using the Makefile provided. It links with the `math` library, in case you want to use the `sqrt` function.

If omitting `filter`, revise the provided Makefile accordingly to compile your code. This Makefile must depend only on pfact.c and not on any other files. Notice that the original Makefile links with the `math` library, in case you want to use the `sqrt` function.

Remember to close the ends of pipes you don't need, and remember to close the pipes when they are no longer needed. If your program appears to be stuck or hung, chances are that a processes is waiting to read from a pipe that will never be written to.

There are many cases where a process may be ready to stop before it has read all the numbers that are being sent on the pipe. If a process closes the read end of the pipe when a process is still writing to the write end of the pipe, the writing process will receive a `SIGPIPE` signal and terminate. This is not the behaviour we want, because it could happen when the program is operating as expected. The solution is to ignore the `SIGPIPE` signal, and then check *errno* if the *write* call fails. (See "`man 2 write`" to see error codes for write). When the signal is ignored, and write is called after the read end of the pipe has terminated. You can use the code snippet below to ignore signals (see "`man signal`" to determine which header file to include).

```
if (signal(SIGPIPE, SIG_IGN) == SIG_ERR) {
    perror("signal");
    exit(1);
}
```

Remember to check the return values of system calls.

You will be writing integers to a pipe rather than strings, so you will be using the `read` and `write` system calls.

Submission

First, login to MarkUs to ensure that the proper subdirectory gets created in your repository and to get the `Makefile` provided.

Commit `pfact.c` and either `filter.c` (if you chose to implement the `filter` function) or the revised Makefile (if you chose to omit the `filter` function) to your repository. No other files will be accepted. Please commit to your repository often, so that you don't lose your work. ~~Do not modify the provided `Makefile` or `filter.h` header file.~~

Remember to also test your code on teach.cs before your final submission. Your program must compile cleanly (without any warning or error messages) on teach.cs using ~~our original Makefile~~ the Makefile in your repo and may not crash (seg fault, bus error, abort trap, etc.) on our tests. Programs that do not compile or which crash will be assigned a 0.

Public Key cryptography and Prime Number Factorization

Why would we want to find the two prime factors of an integer anyway? Is there a practical reason to do this or is it just a toy example? The answer is related to an important area of computer science called public key cryptography.

If Alice wants to send Bob an encrypted message, Bob can send Alice his public key, and Alice can use the public key to encrypt her message, and send it to Bob. Knowing the public key doesn't help decrypt the message, but Bob's private key can be used to decrypt the message. The RSA public-key cryptosystem is one of the oldest and most widely used public-key cryptosystems.

The core idea behind the generation of public and private keys is that factoring a number (RSA number) which is the product of two large primes, is computationally hard. The product of two large primes can be used to generate the public key, and the two factors are used to generate the private key.

Wikipedia describes the RSA problem succinctly:

RSA Laboratories states that: for each RSA number n, there exist prime numbers p and q such that n = p x q. The problem is to find these two primes, given only n.

Much research in number theory and computer science has been carried out over the years to find efficient algorithms to factor large RSA numbers. RSA numbers that are 1024 digits long are still considered relatively secure, but with increasing computer speeds, that probably won't be the case too much longer.