Coroutine prime number sieve

M. Douglas McIlroy

Dartmouth College

For examples in a talk at the Cambridge Computing Laboratory (1968) I cooked up some interesting coroutine-based programs. One, a prime-number sieve, became a classic, spread by word of mouth. As far as I know it didn't appear in print until 1978, in Tony Hoare's influential CSP paper [''Communicating sequential processes'' *CACM* 21 (1978) 666-677]. It's one of those wonderfully intuitive ideas that can be explained over drinks in a bar just as easily as in a classroom lecture—and probably more quickly.

Method

The program implements the sieve of Eratosthenes as a kind of production line, in which each worker culls multiples of one prime from a passing stream of integers, and new workers are recruited as primes are discovered.

Program 1 shows a Unix implementation written in C. A source process writes a sequence of integers, 2,3,4,... into a pipeline of cull filters. Each filter culls multiples of one prime and passes other integers on. The source and filters are trivial programs; the interesting action happens at the end of the pipeline. There, a sink process receives the integers that make it all the way through. These are the primes. Upon receiving each prime, the sink publishes it and inserts a cull filter for it into the pipeline.

Robustness

Notionally the sieve creates pipes and processes forever. But, for every prime it publishes, the sieve allocates another process and another pipe. Eventually some resource limit will be hit and return an error. Since the program ignores errors, it must eventually give bad output or crash—perhaps taking an overstressed system down with it.

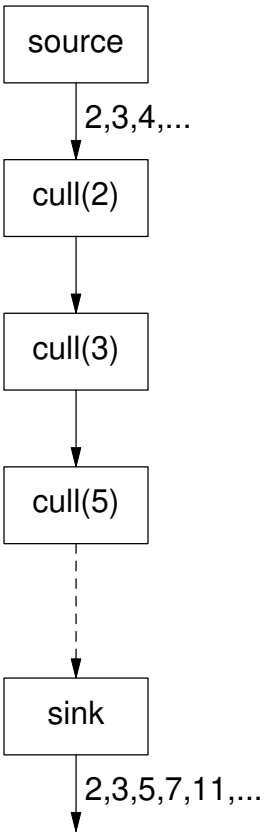
If the program were written to detect errors, it could still continue well beyond the point at which it exhausts resources. Suppose an error occurs when trying to create a filter to cull prime p. If the sink stops creating filters at that point, it could keep on producing primes until it reaches p^2—the smallest composite number that has no prime factor less than p.

The Appendix gives a program that still does not detect errors, but only deploys filters up to $p^{1/2}$ while computing each prime p.

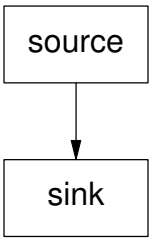
Program 1. Coroutine sieve written in C and realized in Unix processes. **Boldface** highlights the essence of the program.

```
#include <stdio.h>
#include <unistd.h>
void source() {
        int n;
        for(n=2; ; n++)
                write(1, &n, sizeof(n));
}
void cull(int p) {
        int n;
        for( ; ; ) {
                read(0, &n, sizeof(n));
                if(n%p != 0)
                        write(1, &n, sizeof(n));
}       }
/* connect stdin (k=0) or stdout (k=1) to pipe pd */
void redirect(int k, int pd[2]) {
        dup2(pd[k], k);
        close(pd[0]);
        close(pd[1]);
}
void sink() {
        int pd[2];
        int p;                  /* a prime */
        for( ; ; ) {
                read(0, &p, sizeof(p));
                printf("%d\n", p);
                fflush(stdout);
                pipe(pd);
                if(fork()) {
                        redirect(0, pd);
                        continue;
                } else {
                        redirect(1, pd);
                        cull(p);
}       }       }
int main() {
        int pd[2];              /* pipe descriptors */
        pipe(pd);
        if(fork()) {            /* parent process */
                redirect(0, pd);
                sink();
        } else {                /* child process */
                redirect(1, pd);
                source();
}       }
```

Pipeline of Processes



Initial connection



History

When Bob McClure introduced me to Melvin Conway's coroutine concept [''Design of a separable transition-diagram compiler'' *CACM* 6 (1963) 396-408], I was intrigued. Lacking a language that supported coroutines, I did thought experiments with a hypothetical extension of PL/I. That was the framework of my Cambridge talk.

A new language of the time, Simula 67, had a control structure sufficient for coroutines. Its lead designer, Ole-Johann Dahl, loved the sieve scheme when I told him about it (over drinks). I believe he ran the program soon after. He certainly helped spread the word.

When Unix came to be, my fascination with coroutines led me to badger its author, Ken Thompson, to allow writes in one process to go not only to devices but also to matching reads in another process. Ken saw that was possible. As a minimalist, though, he wanted every system feature to carry significant weight. Did direct writes between processes offer a really major advantage over writing to a temporary file in one process and then reading it in the other? Not until I made a specific proposal with a catchy name, ''pipe'', and shell syntax to connect processes via pipes, did Ken finally exclaim, ''I'll do it!''

And he did do it. In one feverish evening Ken modified both kernel and shell, and fixed several standard programs so they could take input from standard input (potentially piped) as well as named files. The next day brought a heady explosion of applications. By the end of the week, department secretaries were using pipes to send text-processor output to the printer spooler. Not long after, Ken replaced the original API and shell syntax for pipes with cleaner conventions that have been used ever since.

The initial pipe exploits did not include the coroutine sieve; a job that spawned processes so furiously would have overwhelmed the small system. Only some time later, more than five years after the Cambridge talk, I made a package that supported coroutines within one single-thread, multi-stack process. Using the package, Dennis Ritchie wrote the first coroutine sieve that I actually saw run.

Ever since Hoare's paper, the coroutine sieve has been a standard demo for languages or systems that support interprocess communication. Implementations using Unix processes typically place the three coroutines—source, cull and sink—in distinct executable files. The fact that the whole program can be written as a single source file, in a language that supports neither concurrency nor IPC, is a tribute not only to Unix's pipes, but also to its clean separation of program initiation into fork for duplicating address spaces and exec for initializing them.

High style

Aside from calling printf, Program 1 is expressed at the kernel API level. Program 2, written at the shell level, expresses the same algorithm more succinctly. Here the actions of setting up pipes and starting processes—more than half of the C program—have been abstracted into the pipe operator "|". The ((...)) clause serves as an arithmetic predicate. The rest of the && compound statement will be eecuted only when the predicate is true.

The logic of Program 2 differs slightly from that of Program 1: (1) To accommodate the pipe operator, sink uses tail recursion instead of a loop. (2) The handy seq command used for the source can produce only finitely many integers, but the limit of a million is more than enough to saturate today's systems.

The sieve is an arch-example of composition of stream-processing operators. Programs 1 and 2 accomplish composition by pipelining. Lazy functional languages provide another approach. A lazy function can take as an argument a conceptually infinite sequence,

Program 2. Sieve as a shell script.*

```
#!/bin/bash
source() {
   seq 2 1000000
}
cull() {
   while true
   do
       read n
      (($n % $1 != 0)) && echo $n
   done
}
sink() {
   read p
   echo $p
   cull $p | sink &
}
source | sink
```

whose elements need not be reified until the function actually accesses them. Composition of operations becomes simple functional composition.

In a functional language the basic logic of the sieve can be laid bare. Pipe connections translate to function applications. Thus, in Haskell, the sieve becomes

```
sink (p:ns) = p : sink (filter ((/= 0).(`mod` p)) ns)
primes = sink [2..]
```

In this code one may recognize in-lined analogs of source and cull in Program 2.

```
source = [2..]
cull p ns = filter ((/= 0).(`mod` p)) ns
```

The standard Haskell function filter selects from a given list those elements that satisfy a given predicate. Here the predicate is the composition of two unary functions, ‘‘not equal to zero’’ and ‘‘mod *p*’’.

Filters as (lazy) functions

In comparison to the concise Haskell version, the C and shell versions seem like quaint historic relics. Nevertheless the C version offers a (prime!) example of the power of the Unix API, and a vivid illustration of the capacity of today’s Unix-like systems to handle computations that comprise hundreds of concurrent processes.

From a functional viewpoint, a Unix filter is a function from byte stream to byte stream. For endless byte streams, functional composition cannot be accomplished sequentially by completing the first function before the second begins. Lazy evaluation is a necessity if

* The identifier source, used for readability here and in Program 3, must be changed to run in the bash shell because it collides with a shell keyword.

one is ever to see any output.†

The late Robert Morris called our attention to this fact at the outset, on "pipe day" itself. Composition by sequential evaluation, he observed, is likewise useless for interactive use. For example, the innocuous cat command (the compositional identity function) is harmless when run in series with the interactive desk calculator dc via a pipe:

```
dc | cat
```

but stymies the interaction when connected via a temporary file:

```
dc >temp; cat <temp
```

Ironically, neither Ken Thompson nor I had taken conscious note of this critical distinction between pipes and intermediate files; otherwise pipes might have made their debut in the first edition of the Unix manual rather than the third.

Try it

Plain-text source is available for the C and (slightly modified) shell versions at

http://www.cs.dartmouth.edu/~doug/sieve/sieve.c
http://www.cs.dartmouth.edu/~doug/sieve/sieve.bash

If your browser doesn't like to get .c or .bash files, try using wget, curl, or lynx.

Appendix. Efficiency: beyond coroutines

The sieves we have shown hog resources. To compute primes up to n, culling filters are deployed for all lesser primes, although only filters for primes up to $n^{1/2}$ are necessary. This profligate use of resources can be avoided by deploying filters only as needed. To do so we must keep a queue, or "waiting list", of primes between $n^{1/2}$ and n.

In all the programs above, such a waiting list is implicit in the part of the chain of culling filters beyond the necessary ones. In the Unix models, the waiting list is heavy: a process and a pipe per prime. The Haskell model, while much lighter, still involves a closure ("thunk") per prime. In every model, each member of the waiting list expends a useless quantum of computing time for every passing prime. (The Unix models impose a further, unrelated burden: buffering allows upstream processes to run ahead of the ultimate consumer, stuffing pipes with numbers that may never be needed.)

Surely the waiting list can be kept simply as a queue of numbers without the overhead of processes or closures. The necessary data is already at hand in the stream of primes. All we need to do is to arrange for it to be read (at different rates) by both the sink and the outside consumer.

The only trickery in the following program is to "prime" the waiting list with 2, so the sink can read from it at the outset. Parameter p' is the modulus for the next filter. Parameter p is a prime, provided it is less than p'^2; otherwise p is discarded and the next filter is started.

---

† Laziness may be achieved by various mechanisms, including call-by-need (as in Haskell) and Landin streams. [P. J. Landin, "A correspondence between ALGOL 60 and Church's lambda notation, Part I" *CACM* **8** (1965) 89-101]

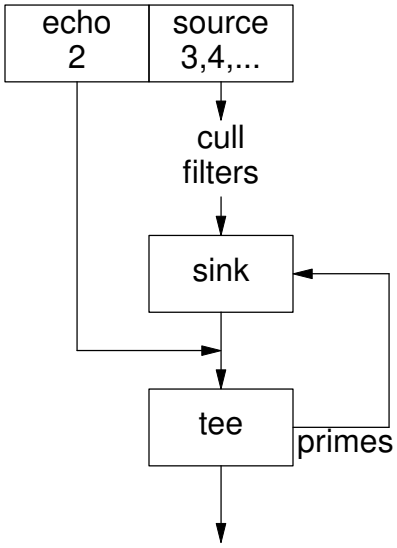
```
primes = 2 : sink [3..] primes
sink (p:ns) ps@(p':ps')
   | p < p'^2 = p : sink ns ps
   | otherwise = sink (filter ((/= 0).(`mod` p')) ns) ps'
```

To make an analogous shell script, we must hand craft some data-flow connections because the pipe operator cannot express certain aspects of the flow graph: (1) sink reads from two input streams; (2) primes are fed back from the output of sink to the input; and (3) output of sink is delivered to two consumers.

In Program 3, feedback is handled by a named pipe, more formally called a fifo, created by mkfifo. The sink output is replicated by tee, which copies its standard input to both standard output and the fifo. The redirection idiom {primes}<fifo opens the fifo for reading in the sink and assigns its file descriptor to primes. To read from this descriptor, sink executes read –u primes. (–u for "unit number" conjures ancient Fortran terminology.)

Program 3. More efficient sieve: tests divisors only up to $p^{1/2}$.

```
#!/bin/bash
source() {
   seq 3 1000000
}
cull() {       # same as in Program 2
   while true
   do  read n
      (($n % $1 != 0)) && echo $n
   done
}
sink() {
   read –u $primes pp
   while
      read p
      (($p < $pp*$pp))
   do
      echo $p
   done
   cull $pp | sink &
}
mkfifo fifo
(echo 2; (source | sink {primes}<fifo)) | tee fifo
```



Program 3 attests to the possibility of using a Unix shell to set up general networks of processes communicating by pipes. However, the code to do so is obscure (and incomplete; it doesn't provide for removing fifo when the program stops). Complex networks are not yet ready for prime(!) time. A tantalizing question arises. Might nonlinear networks become naturalized as design patterns if a perspicuous notation became available?

Written March 2014; tweaked May 2015, February, March, May 2016. Tweaked, added Appendix, converted to PDF November 2016. Replaced close(k);dup(pd[k]) *with* dup2(pd[k], k) *, added footnotes, tweaked text and diagram May 2019. Reworded colophon and inserted one other word September 2019. Avoid implying Fig. 1 can work*

at all after fork fails March 2020.