other processes. Moreover, we can readily connect filters into networks that perform larger computations. All that required is that each filter produces output that meets the input assumptions of the filter(s) that consume that output. Many of the user-level commands in the UNIX operating system are filers, e.g., the text formatting programs tbl, eqn, and $troff$.

A *client* is a triggering process; a *server* is a reactive process. Clients make requests that trigger reactions from servers. A client thus initiates activity, at times of its choosing; it often then delays until request has been served. A server waits for requests to be made, then reacts to them. The specific actions a server takes can depend on the kind of the requests, parameters in the request messages, and the server's state; the server might be able to respond to a request immediately, or it might have to save the request and respond later. A server is often a non-terminating process and often provides service to more than one client. For example, a file server in a distributed systems typically manages a collection of files and services requests from any client that wants to access those files.

6.6.1 An network of filters – Prime number generation

The sieve of Eratosthenes - named after the Greek mathematician who developed it - is a classic algorithm for determining which numbers in a given range are prime. Suppose we want to generate all the primes between 2 and n:

1. First, write down a list with all the numbers:

$$2,\ 3,\ 4,\ 5,\ 6,\ \ldots,\ n$$

2. Starting with the first uncrossed-out number in the list, 2, go through the list and cross out multiples of that number. If n is odd, this yields the list:

$$2,\ 3,\ \not{4},\ 5,\ \not{6},\ \ldots,\ n$$

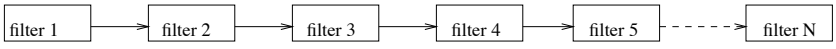
   At this point, crossed-out numbers are not prime; uncrossed-out numbers are still candidates for being prime.

3. Now moving to the next uncrossed-out number in the list, 3, and repeat the above process by crossing out multiples of 3.
4. Continue this process until every number has been considered, the uncrossed-out numbers in the final list will be all the primes between 2 and n.

The essence of this algorithm is that the primes form a sieve that prevents their multiples from falling through.

You can easily write a sequential program for this algorithm. Now consider how we might parallelize this algorithm. One possibility is to assign a different process to each possible prime p and to have each in parallel cross out multiples of p. However, if we can know each p is prime, we do not have to solve this problem anymore.

Now we employ a *pipeline* of filter processes as shown in the following configuration graph:



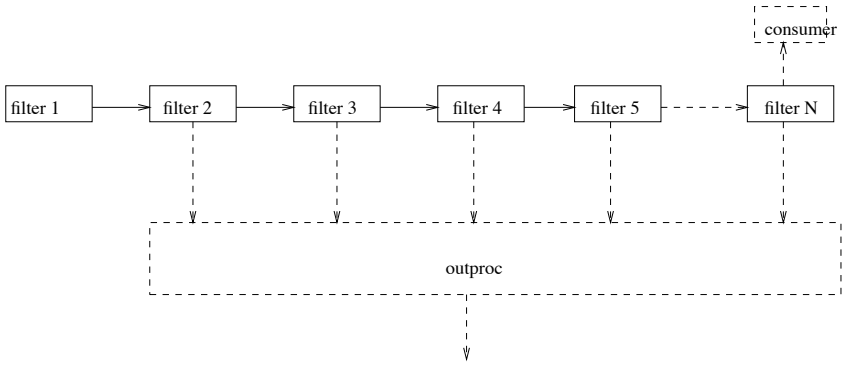
- The first filter process, $filter[1]$, sends the stream of integers starting at 2 (i.e. $2, 3, 4, 5, 6, \ldots$).

- Every other filter process receives a stream of numbers from its left neighbour.
- The first number p that process $filter[i]$ ($i > 1$) receives is the $(i-1)$th prime.
- Each $filter[i]$ subsequently passes on all other numbers it receives that are *not* multiples of its prime p (discards all the multiples of p).
- These N filters generates the first $N-1$ primes.

We can now write a program as follows:

```
PROGRAMM sieve1; CONST N= ... (*N-1 = number of primes to
generate*) TYPE chan = CHANNEL Of integer; VAR pipeline :
ARRAY[1..N] OF chan;
    ploop:integer
PROCESS filter[1];
VAR i: integer;
BEGIN                    write(2);
   i:=2;                 i:=3;
   REPEAT
     ch[1] ! i;
     i:=i+1              i:=i+2;
   FOREVER
END;
PROCESS TYPE filters(i:integer);
VAR p,next: integer;
BEGIN
   ch[i-1] ? p;          write(p);
   REPEAT
     ch[i-1] ? next;
     IF (next MOD p) <> 0 THEN ch[i] ! next
   FOREVER
END;
VAR
  filter: ARRAY[2..N] OF filter;
BEGIN
   COBEGIN
      filter[1]; FOR ploop:=2 TO N DO filter[ploop](ploop)
   COEND
END.
```

The above program terminates in deadlock. The $filter[N]$ will be blocked on $ch[N]!next$ since no process is ready to consume its output; this in turn will block filter[N-1]. The blocked filter[N-1] will block filter[N-2], and so on. The program does not print out the primes generated either. To solve these two problems, we add two processes, $consumer$ which consumes the integers passing through filter[N], and $outproc$ which receives the prime from each filter and print it out:




```
PROGRAM sieve; CONST N= ... (*N-1 = number
of primes to generate*) TYPE chan = CHANNEL Of integer; VAR
pipeline : ARRAY[1..N] OF chan;
    output: ARRAY[1..N] OF chan;     (*added*)
    ploop:integer
```

```
PROCESS filter[1];
VAR i: integer;
BEGIN
   i:=2;
   REPEAT
     ch[1] ! i;
     i:=i+1
   FOREVER
END;
```

```
PROCESS TYPE filters(i:integer);
VAR p,next: integer;
BEGIN
   ch[i-1] ? p;
   output[i] ! p;     (*added: send p to outproc*)
   REPEAT
     ch[i-1] ? next;
     IF (next MOD p) <> 0 THEN ch[i] ! next
   FOREVER
END;
```

```
PROCESS consumer;
VAR local: integer;
BEGIN
  REPEAT ch[N] ? local FOREVER
END;
```

```
PROCESS outproc;
VAR I, Num: integer;
BEGIN
  FOR I:=2 TO N DO
    BEGIN
     output[I] ? num;
     writeln(num)
    END
END;
```

```
VAR
  filter: ARRAY[2..N] OF filter;
BEGIN
   COBEGIN
      filter[1]; consumer; outproc;
      FOR ploop:=2 TO N DO filter[ploop](ploop)
   COEND
END.
```

Then above program will not deadlock. However, it will not terminate. It is certainly desirable that the program should terminate normally after all N-1 primes have been printed. We will come back later to discuss the termination problem in general.

6.7 Synchronous channels

As mentioned earlier, with message passing processes synchronize while communicating. The sending process and receiving process synchronize at the communication point. Exchange of message and synchronization are carried out by the sending and receiving operations which are executed simultaneously. However, sometimes two processes need to synchronize without the need of exchange of a message. In this case a dummy piece of data would have to be communicated. This can lead to confusion for someone reading the program at a later date. Pascal-FC allows the intension of the programmer to be clearly expressed by introducing a special base type for such *contentless* communication.

The type **synchronous** is predefined. There are no values associated with this type. A variable called **any**, of type **synchronous**, is automatically declared by the compiler for every program. The following sketch code illustrates how one process (**starter**) can be used to delay and then release two (**worker**) processes.

```
PROGRAM starters;
TYPE syn = CHANNEL OF synchronous;
     barriers = ARRAY[1..2] OF syn;
VAR barrier: barriers;
PROCESS starter;
VAR I: integer;
BEGIN
....
  FOR I:=1 TO 2 DO
     barrier[I] ! any;
...
END;
PROCESS TYPE worker(num:integer);
BEGIN
 ....
 barrier[num] ? any;
 ...
END;
VAR workers: ARRAY[1..2] of worker;
...
```