Using *n*-ary semaphores

Data invariant for the producer-consumer problem:

$0 \le \#in - \#out \le \#buffer$

This invariant can be maintained with two semaphores, each of which enforces the invariant

$\#enter - \#leave \le s_0$.

Filling the buffer: $\#in - \#out \le \#buffer$

$s_0 = \#buffer$; producer: ***enter***, consumer: ***leave***.

Emptying the buffer: $\#out - \#in \le 0$

$s_0 = 0$; consumer: ***enter***, producer: ***leave***.

In Ada:

```
...
filling,  emptying : semaphore;
critCons, critProd : binary_semaphore;
...
begin
  filling.init(size);
   -- provides notFull service
  emptying.init(0);
   -- provides notEmpty service
end
```


The semaphores' clients

```
task producer;
task body producer is
  c : character;
begin
  while not end_of_file loop
      get(c);
        filling.enter;
          critProd.enter;
            buf(rear) := c;
            rear := (rear+1) mod size;
          critProd.leave;
        emptying.leave;
  end loop;
end producer;
```

```
task consumer;
task body consumer is
  c : character;
begin
  loop
        emptying.enter;
          critCons.enter;
            c := buf(front);
            front := (front+1) mod size;
          critCons.leave;
        filling.leave;
    put(c);
  end loop;
end consumer;
```

Safety is assured, *but* the synchronization code is distributed among the various clients, and an error in any client can invalidate the entire system.

Synchronization by Rendezvous **centralizes the synchronization**

A *buffer* server-task type:

```
task type buffer is
  entry insert( c : in  character );
  entry remove( c : out character);
end buffer;
```

```
  task B : buffer;
```

The buffer task's clients:

```
task producer;
task body producer is
  c : character;
begin
  while not end_of_file
  loop
     get(c);
     B.insert(c);
  end loop;
end producer;
```

```
task consumer;
task body consumer is
  c : character;
begin
  loop
    B.remove(c);
    put(c);
  end loop;
end consumer;
```



```
task body buffer is
   ...declare variables...
begin
   ...initialize variables...
  loop
    select
      when notFull =>
        accept insert(c: in character) do
          buf(rear) := c;
          rear := (rear+1) mod size;
          notFull := front /= succ(rear);
          notEmpty := true;
        end insert;
    or
      when notEmpty =>
        accept remove(c: out character) do
          c := buf(front);
          front := (front+1) mod size;
          notFull := true;
          notEmpty := front /= rear;
        end remove;
    end select;
  end loop;
end buffer;
```

The buffer task's clients are free of all responsibility for synchronization. An error in one of them cannot bring down the entire system.

Another example: a Print-server task

Assumptions:

Each client calls

```
procedure printDoc( theDoc : in array (1..pMax) of Page )
```

The print server calls

```
procedure printOnePage( aPage : in Page )
```

which sends a **single page** to the printer.

Implementations:

```
procedure printDoc( theDoc : in array (1..pMax) of Page )

   pn : integer;
   pages : integer;

begin

   pn := 1;
   PrintServer.StartJob;

   loop
      PrintServer.PrintPage( theDoc( pn ) );
      exit when pn = pMax;
      pn := pn + 1;
   end loop;

   PrintServer.EndJob( pages );

   if pn /= pages then
      error( "a page was not printed" );
   end if;

end printDoc;
```



```
task PrintServer is
   entry StartJob;
   entry PrintPage( thePage : in Page );
   entry EndJob( pages : out integer );
   busy : boolean;
   pageCount : integer;
begin
   busy := false;
   loop
     select
       when not busy =>
       accept StartJob do
         busy := true;
         pageCount :=0;
       end StartJob;
     or
       when busy =>
       accept PrintPage( thePage : in Page ) do
         pageCount := pageCount + 1;
         printOnePage( thePage );
       end PrintPage;
     or
       when busy =>
       accept EndJob( page : out integer ) do
         page := pageCount;
         busy := false;
       end EndJob;
     end select;
   end loop;
end PrintServer
```

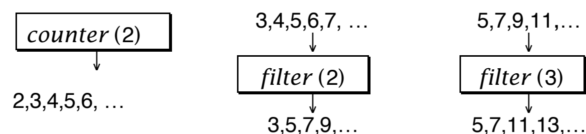

Example: a prime-number generator

Method: "Sieve of Eratosthenes" (c. 200 B.C.)

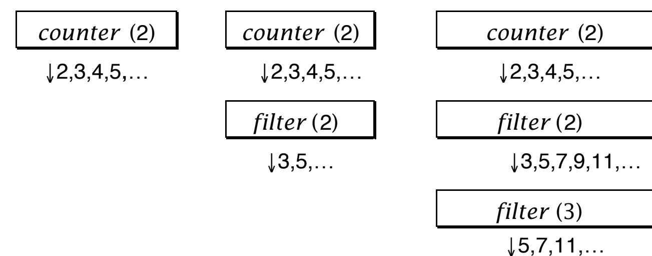
[2] 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 …

[3] 5 7 9 11 13 15 17 19 21 23 25 27 29 31 …

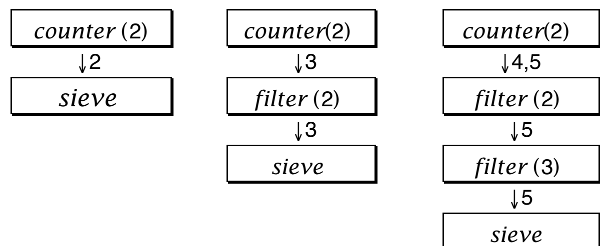
[5] 7 11 13 17 19 23 25 29 31 35 37 41 …



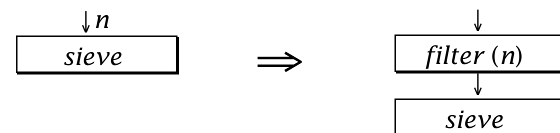
Evolution of the *primes* pipeline:



The evolution's agent: task *sieve*



What does *sieve* do?



In Haskell:

```
> primes = sieve [2..]

> sieve (p:ns)
>   = p : sieve [ n | n<-ns, n `mod` p > 0 ]
```

In Ada…

```
with Ada.Integer_Text_IO, ada.text_io;
use Ada.Integer_Text_IO;
```