

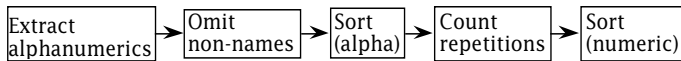
2. Find names' frequency in (program) text.

```
tr -cs A-Za-z0-9 '\012' |      replace nonalphanumeric
                              characters with new-lines
grep "^[A-Za-z]"             |      output only the lines
                              that begin with letters
sort                          |      sort the lines
                              alphabetically
uniq -c                       |      eliminate repeated
                              lines, and precede
                              each output line with
                              a count of its
                              occurrences
sort -rn                      |      sort the lines in
                              reverse order, on the
                              basis of the first
                              number in each line
```

In Haskell,

```
> infixl 9 >.>
> (>.>) :: (a->b)->(b->c)->(a->c)
> (f >.> g) x = g(f x)

> tr "-cs" "A-Za-z0-9" '\012' >.>
> grep "^[A-Za-z]" >.>
> sort "" >.>
> uniq "-c" >.>
> sort "-rn"
```



Explicit Synchronization

Programmers pay explicit attention to **events** in a computation.

event = an atomic (=indivisible, =uninterruptible) action

Examples of events:

- assignment
- function application
- procedure call
- expression evaluation
- ... etc. (varies with language)

A *sequence* of events in a single process is called a **thread**.

Concurrency is achieved by **interleaving** threads from distinct processes.

Suppose $P = a b$, $Q = x y z$

Then the interleavings of P and Q include

$a b x y z$
 $a x b y z$
 $a x y b z$

but not

$a y b z x$

Concurrent tasks in Ada

process ~ "task" in Ada
 module ~ "package" in Ada

Example:

```
with text_io; use text_io;
procedure id is
    task P;
    task body P is
    begin
        put("a");
        put("b");
    end P;

    task Q;
    task body Q is
    begin
        put("x");
        put("y");
        put("z");
    end Q;

begin
end id;
```

This specifies a procedure `id` which has two offspring tasks `P` and `Q`.

Tasks `P` and `Q` (and `id`) execute concurrently.
 Output: Any interleaving of "ab" and "xyz".

Shared Data: ensuring safe access

Sometimes we allow processes to be *nondeterministic*:

Several outcomes possible, depending on timing, and all of them are OK

Examples: flight reservation system
 printer access
 file access

Sometimes only *one* result is acceptable.

Example: either of two banking transactions may take place ahead of the other:

{ $a = 1000$ }

`deposit(a,200)`

`deposit(a,300)`

{ $a = 1500$ }

Operation `deposit` must be *atomic*; if not, its operations might be interleaved arbitrarily ...

{ $a = 1000$ }

LOAD a	LOAD a
ADD 200	ADD 300
STORE a	STORE a

{ $a = 1200 \vee a = 1300 \vee a = 1500$ }

⇒ Multiprocess systems need ways to make a sequence of events into a *single atomic event* (for purposes of interleaving) —i.e., a **critical section**.

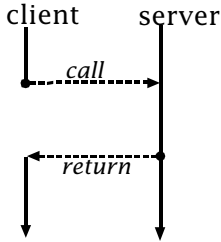
Atomicity's scope is important— it's sufficient to treat a **cs** as an atomic operation **only** with respect to other **cs**'s that deal with the **same** shared **resources**.

This allows a system to have multiple independent shared resources —e.g., printer and disk drive— each accessible independently of the other.

Synchronization: constraining the order of events in two distinct processes.

Ada's synchronization feature:

the **Rendezvous mechanism**



A **client** process **issues** a **call**.

The **server** process **accepts** the call.

If the server reaches the **accept** statement before the client issues the call, the server waits.

After a server accepts a call—

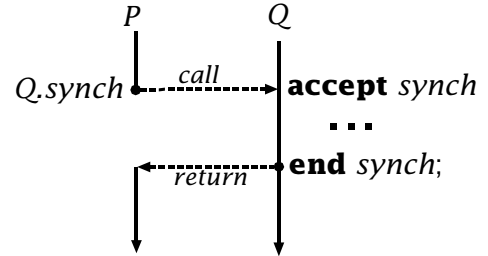
- the *client* remains suspended
- the *server* executes alone

until the server reaches **end** of the rendezvous code (i.e., of the **accept** block). At that point, the client is released to resume execution.

A single rendezvous (**accept**) may be called by several clients, but each call names a specific rendezvous (**accept**) in a specific server.

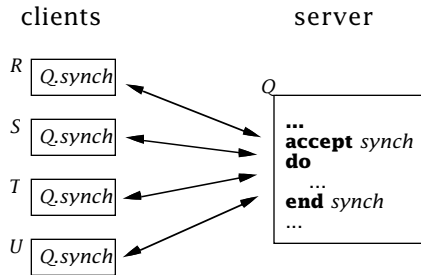
Call-accept **matching**

Call: $\langle server-name \rangle . \langle entry-name \rangle$,
 where $\langle entry-name \rangle$ identifies one of the named server's **accepts**.



Note the asymmetry:

- Servers' names must be **known** to clients
- Clients' names may be **unknown** to server.



Each execution of an **accept** serves a *single* client; other clients *wait* for the **accept**'s next execution.

In addition to synchronization, the rendezvous also provides **communication** via parameters and arguments:

- **call**: **in** parameters client → server
- **return**: **out** parameters server → client

Example:

<pre> task P a,b : integer; begin a := 0; Q.inc(a, b); -- now b = 1 end P; </pre>	<pre> task Q begin accept inc(x : in integer, y : out integer) do y := x + 1; end inc; end Q; </pre>
---	--

Task types enable declarations of multiple instances.

Example:

```

with text_io; use text_io;
procedure task_init is

```

```

task type emitter is -- declares a task type
entry init(c: character);
end emitter;

```

```

task body emitter is
me: character;
begin
accept init(c: character) do
me := c;
end init;
put(me); new_line; -- order not constrained
end emitter;

```

```

p, q: emitter; -- declares two tasks of type emitter;
-- they start running at once

```

```

begin
p.init('p'); -- passes 'p' to task p
q.init('q'); -- passes 'q' to task q
put('r'); new_line;
end task_init;

```