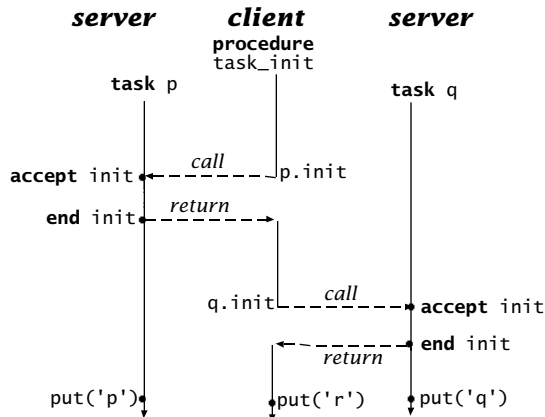


## Concurrency versus Ordering Constraints

Because `put` is **outside** the rendezvous, outputs `p`, `q`, and `r` can appear in any order.



If the `put` statement were included **within** the rendezvous code—

```

accept init(c: character) cb
  me:= c;
  put(me); new_line;
end init;
  
```

then 'p', 'q', and 'r' would be guaranteed to appear in that order—but concurrency (and therefore perhaps performance) would be reduced.

(To maximize concurrency, Ada programmers minimize the code *within* each rendezvous block.)

## Creating tasks *dynamically*

Objects that are created dynamically

- reside in heap store
- are referred to by pointers

(as in Pascal, C, C++, ...).

In Ada, *tasks* are a kind of object.

Terminology:

“pointer to X” = “**access** X” in Ada, so

```

type emitter_ptr is access emitter;
  p, q : emitter_ptr;
  
```

declares `p` and `q` to be variables whose type is pointer-to-task-of-type-emitter.

A task is created dynamically by **new**:

```

new emitter -- like C++
  
```

creates an instance of emitter and returns a pointer to it.

Pointers to tasks can be assigned to pointer variables (of the appropriate type):

```

  p := new emitter;
  q := new emitter;
  
```

As soon as a task is created, it is *eligible* to run—concurrently with its parent task.

Multiple accepts: the **select** construct

A single server can offer several services; for example, a *stack* server might include:

```

loop
  accept push(n: in integer) do
    ...
  end push;
  accept pop(n: out integer) do
    ...
  end pop;
end loop;
  
```

but this server offers only **one** service at a time—

- *first* it waits for a push request
- *then* it waits for a pop request

To make two (or more) accepts active at once:

```

loop
  select
    accept push(n: in integer) do
      ...
    end push;
  or
    accept pop(n: out integer) do
      ...
    end pop;
  end select;
end loop;
  
```

This server accepts calls to push and pop in whatever order they happen to arrive.

A server can specify when an accept is armed:

The **guarded** accept:

```

loop
  select
    when notFull =>
      accept push(n: in integer) cb
        notEmpty := true;
        notFull := ...;
        ...
      end push;
    or
      when notEmpty =>
        accept pop(n: out integer) cb
          notFull := true;
          notEmpty := ...;
          ...
        end pop;
      end select;
  end loop;
  
```

The *guarded* select is like GCN's **if...fi**: all guards false => *exception* (i.e., error).

## Using synchronization to make sharing safe

Our example: *producer* → *buffer* → *consumer*

Implementations (all in Ada):

- 1 Unsynchronized access (*unsafe*)
- 2 Synchronization by Semaphore (low level)
- 3 Synchronization by Task Rendezvous

## Uncontrolled access

global: buf, front, rear, size, notFull, notempty

```

task producer;
task body producer is
  c : character;
begin
  while not end_of_file loop
    if notFull then
      get(c); -- for example, from a file
      buf(rear) := c;
      rear := (rear+1) mod size;
      notFull := front /= (rear+1) mod size;
      notEmpty := true;
    end if;
  end loop;
end producer;

```

```

task consumer;
task body consumer is
  c : character;
begin
  loop
    if notEmpty then
      c := buf(front);
      front := (front+1) mod size;
      notFull := true;
      notEmpty := front /= rear;
      put(c); -- for example, to a file
    end if;
  end loop;
end consumer;

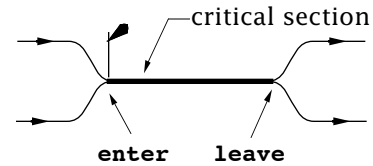
```

If multiple tasks read and write buf, front, and rear, buffer over- and underflow and get/put of the same character twice or more are not prevented.

8 April 2005

**Mutual Exclusion 1: By Semaphore**

*Semaphore*: A signalling device used in railroads to ensure that a given section of track contains at most one train.



Purpose of software semaphores: To limit the number of processes simultaneously executing a given section of code (a "critical section").

**Binary semaphore**: #processes  $\leq 1$ .

```

task type binary_semaphore is
  entry enter; -- entering CS
  entry leave; -- leaving CS
end binary_semaphore;

task body binary_semaphore is
begin
  loop
    accept enter; -- also called p
    accept leave; -- also called v
  end loop;
end binary_semaphore;

```

8 April 2005

**Using binary semaphores**

```

critical : binary_semaphore;
task type one_at_a_time is
  ...
end one_at_a_time;

task body one_at_a_time is
  ...
begin
  ...
  critical.enter;
  ...
  ... -- the task's critical section
  ...
  critical.leave;
  ...
end one_at_a_time;

...
R, S, T: one_at_a_time;
...

```

The first task to execute `critical.enter` proceeds into its critical section.

Any task subsequently executing `critical.enter` waits until the current critical-section task (if any) has executed its `critical.leave`.

Hence the number of tasks executing their critical sections at any time is at most 1.

8 April 2005

**Generalization:  $n$ -ary Semaphores**

An  $n$ -ary semaphore has internal variable  $s$ .

When client performs **leave** operation:  $s := s + 1$

When client performs **enter** operation:

**while**  $s = 0$  **do wait end**;  $s := s - 1$

Semaphore Invariant:  $\#enter - \#leave \leq s_0$ .

```

task type semaphore is
  entry init(n : integer);
  entry enter;
  entry leave;
end semaphore;

task body semaphore is
  s : integer;
begin
  accept init(s0 : integer) do
    s := s0;
  end init;
  loop
    select
      when s >= 1 => -- entering tasks may
        accept enter do -- wait here
          s := s-1;
        end enter;
      or
        accept leave do -- tasks are always free
          s := s+1; -- to leave their CS's
        end leave; -- with no waiting
    end select;
  end loop;
end semaphore;

```

8 April 2005