Rendezvous

Rendezvous provides synchronization and two-way communication between two threads, a client and a server.

- From the client's point of view the rendezvous is a procedure call (remote or local)
- The client blocks until the server executes an in statement (MPDP) or accept statement (Ada).
- Server blocks until a call is made that it is prepared to accept.
- Once both are ready
  - ∗ arguments are copied to parameters
  - ∗ code is executed by the server
  - ∗ results are returned and "copy out" parameters are copied to arguments.
  - ∗ Both threads proceed independently.

Example

```
module TicketServer
    op getNext returns int ;
body
    process daemon {
        int next := 0 ;
        while( true) {
            in getNext() returns val ->
                val := next ;
            ni
            next := next + 1 ;} }
   end TicketServer
```

Note that mutual exclusion is implicit, since the server thread can be handling but 1 request at a time.

Choice

The server thread can offer a choice of requests that it will accept and can specify the conditions under which it will accept a choice.

```
module Bounded_buffer
    op deposit(char data);
    op fetch(result char data);
body

    process Buffer {
        char buf[n]; # buffer
        int front = 0; # first full slot
        int count = 0; # number of full slots

        while (true) {
            in deposit(data) and count < n ->
                buf[(front+count)%n] = data ;
                count := count+1 ;
            [] fetch(data) and count > 0 ->
                data := buf[front] ;
                front := (front+1)% n ;
                count := count - 1 ;
            ni } }
end Bounded_buffer
```

Rendezvous vs. monitors

The previous example is highly reminiscent of the monitor solution.

Both provide “structured” approaches to mutual exclusion.

Mutual exclusion is implicit in both monitors and rendezvous.

Passive and active objects

Monitors are passive objects.

- Only client threads exist.
- The client thread performs the service for itself inside the monitor.
- Server state does not change spontaneously.

Modules containing server threads are active objects.

- The server thread acts on behalf of the client thread within the module for the duration of the rendezvous.
- Server threads may change the module state in between calls from clients.

Data state vs. control state

With active objects, control state as well as data state can regulate operations that can proceed.

```
module Buffer
    op deposit( char data) ;
    op fetch( result char data) ;
body
    char buffer ;

    process daemon {
        while (true) {
            in deposit(data) -> buffer := data ; ni
            in fetch(data) -> data := buffer ; ni } }
end Buffer
```

- Acceptance of deposit and fetch strictly alternate.
- 2 states are represented by the program counter.
- Use of control state is sometimes clearer than use of data state.

Wait/signal vs. nested “in”

In monitors: a wait can happen at any point service is suspended until it can resume later.

With rendezvous: once a service has started it can only be suspended by a nested “in”. Consider:

```
module Barrier {
    op done ;
body

    process daemon {
        while (true) {
            in done() -> in done() -> skip ni
            ni ; } }
end Sync2
```

How would you write an N process barrier?

Rendezvous vs. (Remote) Procedure Call

Rendezvous provides synchronization and mutual exclusion.

- RPC is handled by a new thread. Mutual exclusion must be made explicit.
- Rendezvous is handled by a single thread. Hence implicit mutual exclusion.

Rendezvous vs. Synchronous Message Passing

As with synchronous message passing the client (sender) is delayed until the server (receiver) is ready to accept the communication.

In a degenerate form

in MessType(param) -> local := param **ni**

rendezvous is a synchronous receive. We abbreviate the above by

receive MessType(local) ;

Rendezvous adds the ability for the server to

- delay the client until further processing is done and
- to send information back to client after that processing.

Rendezvous in Ada

The rendezvous is strongly associated with Ada since

- Ada introduced the concept (early '80s)
- No other major language has supported it.

In Ada

- in is called accept.
- Operations are called entries.
- Choice requires use of select statement.

Conditional acceptance can not depend on parameter values.

An Ada Example

```
loop
    select when count < n =>
        accept deposit( data : in char ) do
            buf((front+count) mod n) := data ;
        end deposit ;
        count := count+1 ;
    or when count > 0 =>
        accept fetch( data : out char ) do
            data := buf(front) ;
        end fetch ;
        front := (front+1) mod n ;
        count := count - 1 ;
    end select ;
end loop ;
```

Non-blocking servers and clients (Ada)

The server thread can opt not to block.

```
loop
    select
        accept calibration( v : real ) do
            scale := v ;
        end calibration
    else
        null ; - - do nothing
    end select
    sensorOut := scale * sensorIn ;
end loop
```

Likewise, the client can make a conditional call depending on whether the server is currently prepared to accept it.

```
select
    Queue.deposit( packet ) ;
else droppedPacketCount := droppedPacketCount + 1 ;
end select
```

Time outs

The server thread may time-out if it blocks too long.

A watch-dog thread

```
loop
    select
        accept AllIsWell ;
    or delay 10.0 ;
        RaiseAlarm ;
    end select
end loop
```

Likewise, the client may time out if service is not sufficiently fast

```
select Queue.deposit( packet ) ;
or delay 20.0 ;
    droppedPacketCount := droppedPacketCount + 1 ;
end select
```

Multiple primitives

The SR language (and MPDP) combines RPC, AMP, and rendezvous into a unified framework.

Client may invoke operation by

- *call* — synchronous, waits for return
- *send* —asynchronous, does not wait for reply.

Server may implement operation by

- *procedure*
  - ∗ New thread is created to handle invocation.
  - ∗ No implicit blocking of client.
- *in statement*
  - ∗ Existing server thread handles invocation.
  - ∗ Client is blocked until server is ready to accept

Considering all combinations

	call	send
procedure	RPC	new thread created
in	rendezvous	async. mess. pass.

Consider the Bounded_buffer module above.

- A client could either call `deposit` or send a `deposit` message depending on whether it needs to wait for completion or not.

- But sending a fetch message would be a problem since it has an output.
  - ∗ We could change fetch to have a semaphore parameter.
  - ∗ The semaphore is Veed when the output is ready.
  - ∗ This idea assumes reference (rather than copy in/copy out) parameters.
- We could implement deposit and fetch with procedures rather than an in statement.

```
module Bounded_buffer {
    op deposit(char data);
    op fetch(result char data) ;
body

    monitor BufferMon {
        ...
    }

    procedure deposit(data) {
        BufferMon.deposit( data ) ; }

    procedure fetch(data) {
        BufferMon.fetch( data ) ; }
end Bounded_buffer
```

See MPDP for other interesting examples.

While multiple primitives are a neat idea and provide valuable insight into the relationships of the various interprocess communication mechanisms, few languages provide direct support.