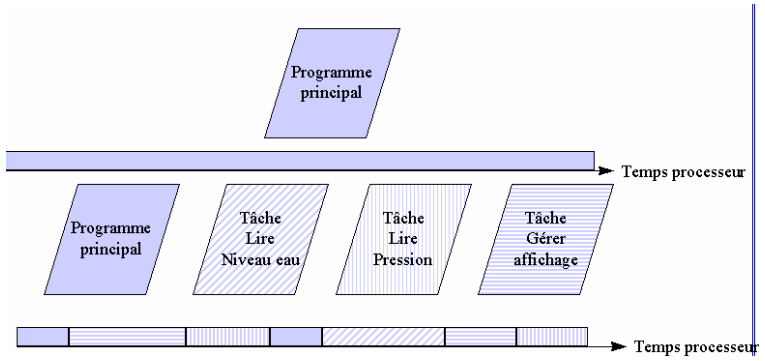
Entradas protegidas e sincronização condicional



A programação concorrente permite manobrar diferentes entidades activas em pseudo-paralelismo.

Necessidade de:

Sincronização

Exclusão mútua

Partilha de recursos

Troca de informação → RENDEZ-VOUS

Entradas protegidas
Sincronização condicional

Entrada é uma operação com um interface semelhante ao de um procedimento: **ENTRY E (...);**

No corpo pode associar-se um guarda booleano: **Entry E (...) when B is ...**

Guarda falso → tarefa que chama a entrada é suspensa numa fila de espera

Guarda verdadeiro → tarefa que chama pode prosseguir

```
protected MAILBOX is

   entry POST (I : in INTEGER);
   entry COLLECT (I : out INTEGER);

private

   BOX  : INTEGER;
   FREE : BOOLEAN := TRUE;

end MAILBOX;
```

Uma entry deve ser usada quando se aplica um guarda.

Cada entry tem uma fila associada a si

```
protected body MAILBOX is

   entry POST (I : in INTEGER) when FREE is
   begin
      BOX := I;
      FREE := FALSE;
   end POST;

   entry COLLECT (I : out INTEGER) when not FREE is
   begin
      I := BOX;
      FREE := TRUE;
   end COLLECT;

end MAILBOX;
```

Contrariamente ao que sucede com as entry das tarefas, numa unidade protegida, as entry têm uma zona declarativa, bem como algumas instruções executáveis.

O guarda é uma simples condição booleana que não pode incluir referências aos parâmetros da entrada.

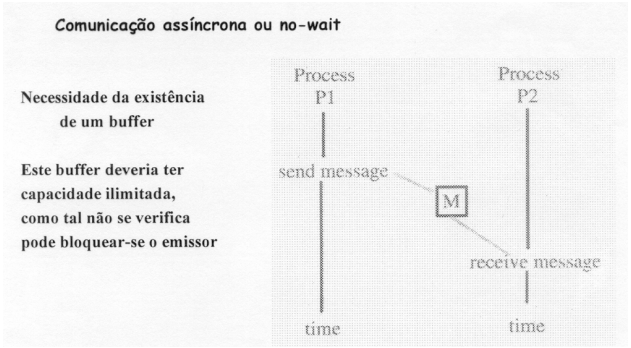
Time	Producer Task	Consumer Task
1	**MAILBOX.POST (1);**	**....**
2		**MAILBOX.COLLECT (ITEM)**
3	**(enter MAILBOX)**	
4		**(suspended on protected unit's queue)**
5	**when FREE ...**	
6	**(suspended on POST queue)**	
7	**(release MAILBOX)**	
8		**(enter MAILBOX)**
9		**when not FREE ...**
10		**I := BOX;**
11		**FREE := TRUE;**
12		**(leave MAILBOX)**
13	**(re-enter MAILBOX)**	
14	**BOX := I;**	
15	**FREE := FALSE;**	
16	**(leave MAILBOX)**	**....**

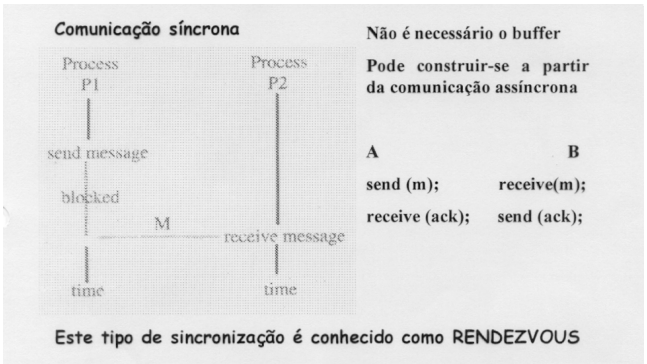
Comunicação entre processos mediante mensagens

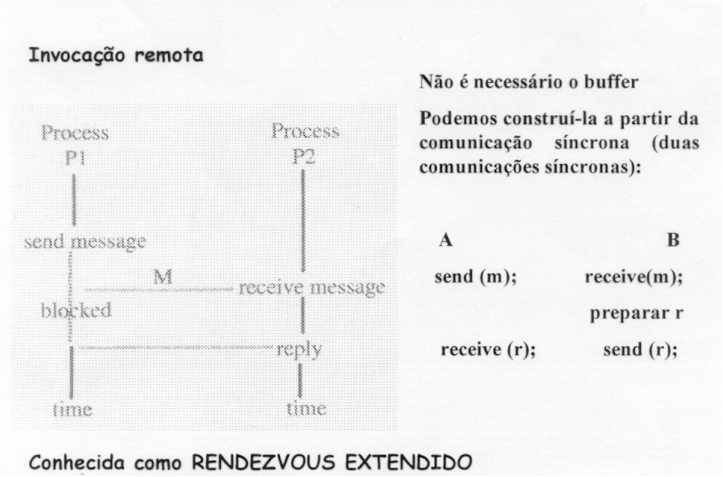
• Comunicação assíncrona → o emissor continua a sua execução após ter enviado uma mensagem.

• Comunicação síncrona → o emissor espera que o receptor receba a sua mensagem.

• Invocação Remota → o emissor espera que o receptor receba a sua mensagem e ainda uma resposta deste.







DESVANTAGENS DO ENVIO ASSÍNCRONO:

Necessidade de buffer com dimensão potencialmente infinita

A maioria dos envios são programados para esperarem um sinal de reconhecimento → COMUNICAÇÃO SÍNCRONA

São necessárias mais comunicações com o modelo assíncrono →os programas tornam-se mais complexos

Maiores dificuldades para provar o estado correcto de todo o sistema

IDENTIFICAÇÃO DO EMISSOR E DO RECEPTOR

Identificação directa → o emissor identifica explicitamente o receptor

send mensagem to processo

Identificação indirecta → quando se usa um intermediário como por exemplo, um canal, uma mailbox, um link, ...

send mensagem to mailbox

SIMETRIA

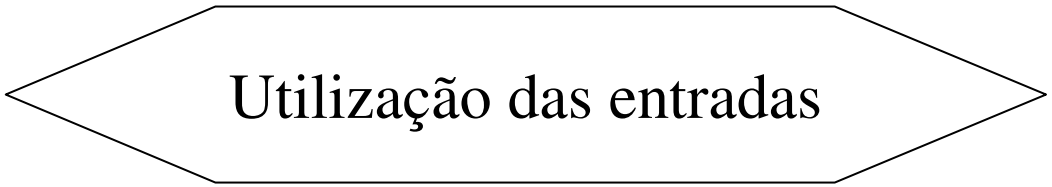
A comunicação é simétrica quando o emissor identifica o receptor e vice-versa.

send mensagem to processo

receive mensagem from processo

A comunicação é assimétrica quando o receptor aceita mensagens de qualquer emissor → comunicação adequada para realizar servidores

COMO ESTABELECER COMUNICAÇÃO ENTRE TAREFAS EM ADA



Exemplo:

```
task type Screen is
    entry Put (Char:Character; X,Y : Coordinate);
end Screen;

Display : Screen;
```

Podemos ter entradas homónimas, desde que tenhamos parâmetros distintos na sua declaração.

Podemos definir famílias de entrada com um discriminante discreto:

```
type Chan_Number is new Integer range 1..7;
task Multiplexor is
    entry Channels (Chan_Number) (Data: Input_Data);
end Multiplexor;
```

PODEMOS CRIAR ENTRADAS PRIVADAS

```
task type Telephone_ Operator is
   entry Directory_Enquiry (Person : in Name; Phone: out Number);
   entry Report_Fault (Phone: in Number);
private
   entry Allocate_Repair_Worker (Id: out Worker_Id);
end Telephone_Operator;
```

A CHAMADA DAS ENTRADAS

Não se dispõem neste caso da clausula use, e portanto para se chamar uma entrada temos identificar a tarefa receptora e a entrada desejada.

```
Display.Put (C,10,20) -- onde C é um caracter

Multiplexor.Channel (3) (D);

Operator.Directory_Enquiry ("Juan Pérez", No_de_Juan);
```

Se se efectuar uma chamada a uma entrada de uma tarefa que não está activa, eleva-se a excepção *Tasking_Error.*

Para protegermos o SW contra eventuais erros fatais, temos de fazer:

```
begin
    Display.Call(J);
exception
 when Tasking_Error =>
                        Manipulação do erro e continuação
 end;
```

A RECEPÇÃO DAS MENSAGENS

Receber uma mensagem envolve o aceitar de uma chamada para a entrada apropriada.

```
type Chan_Number is new Integer range 1..7;
task Multiplexor is
    entry Channels (Chan_Number) (Data: Input_Data);
end Multiplexor;
```

```
accept Channels (3) (Data: Input_Data) do
   -- armazenar os dados do terceiro canal
end Channels;
```

UM TRATAMENTO accept PODE SER COLOCADO EM QUALQUER PONTO DO CORPO DE UMA TAREFA.

NÃO SE PODE COLOCAR NUM PROCEDIMENTO

O TRATAMENTO accept PODE MESMO SER COLOCADO DENTRO DE OUTRO TRATAMENTO accept, MAS NÃO PARA A MESMA ENTRADA.

TODAS AS ENTRADAS (E MEMBROS DE FAMILIAS DE ENTRADAS) DEVEM TER accepts ASSOCIADOS.

DEVE EXISTIR PELO MENOS UM ACCEPT POR CADA ENTRADA, PODENDO HAVER MAIS.

O CORPO DO accept CONTÉM AS INSTRUÇÕES QUE SE EXECUTAM QUANDO SE ACEITA A CHAMADA.

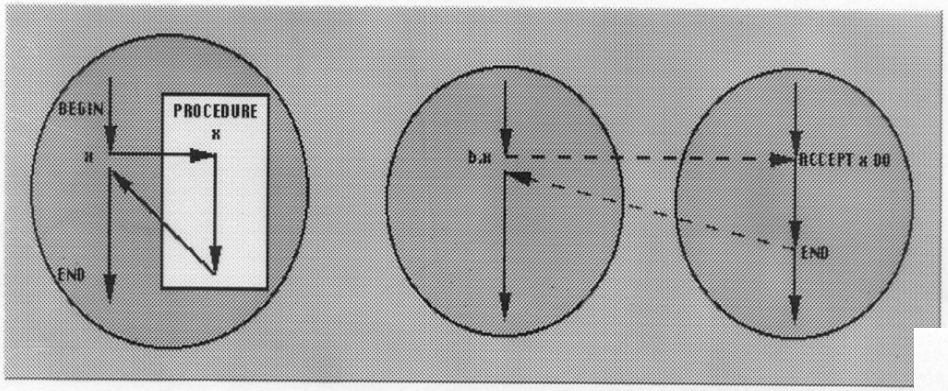
A SEQUÊNCIA DE INSTRUÇÕES PODE INCLUIR MANEJADORES DE EXCEPÇÕES

SE O CORPO É NULO, PODE-SE USAR UMA FORMA SIMPLIFICADA:

accept E;

Uma nota interessante:

Note-se que embora uma chamada para uma entrada se pareça com uma chamada para um procedimento, os dois diferem na sua implementação, como se pode constatar pela figura que se segue:



Quando um procedimento é chamado, mesmo que seja um procedimento situado no interior de um monitor ou unidade protegida, é a sequência de controle do processo que chama que é usada para executar a operação.

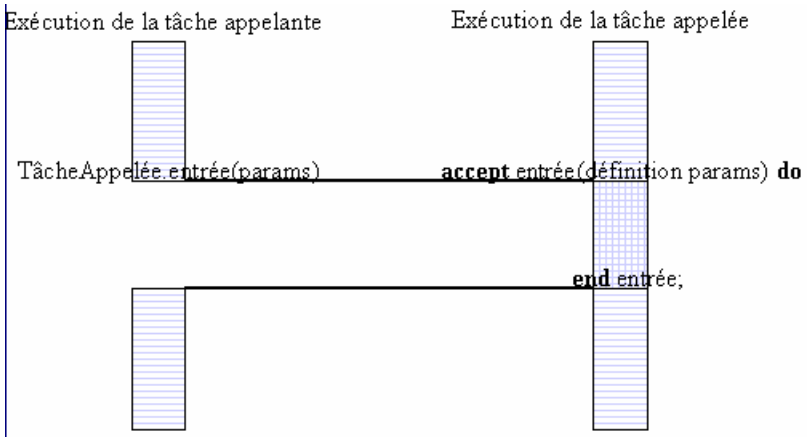
Quando uma entrada é chamada, o processo que chama é suspenso, e é a sequência de controle da tarefa chamada que executa a correspondente declaração de aceitação.

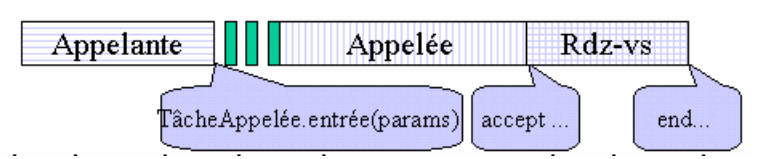
Isto significa que a tarefa chamada não precisa de residir obrigatoriamente no mesmo processador que o processo que a chama. Pode ser distribuída a outro processador num dado Sistema.

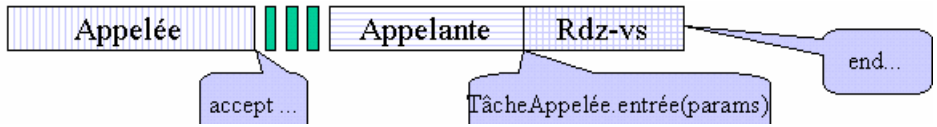
Infelizmente, a execução de um *accept* é muito mais dispendiosa em tempo que a execução equivalente de uma chamada a um procedimento quando temos uma implementação num sistema monoprocessador.

De cada vez que é chamada uma entrada e que as correspondentes declarações de *accept* são executadas, temos de suspender a tarefa que chamou e levar ao processador a tarefa chamada, ou seja temos de efectuar uma mudança de contexto.

Tal situação, como é obvio implica "perdas de tempo", e existe tempo do processador que é "gasto" de forma inútil.


Exécution de la tâche appelante
Exécution de la tâche appelée
TâcheAppelée.entrée(params)
accept entrée(définition params) do
end entrée;


Appelante
Appelée
Rdz-vs
TâcheAppelée.entrée(params)
accept ...
end...


Appelée
Appelante
Rdz-vs
end...
accept ...
TâcheAppelée.entrée(params)

```
with Ada.Text_Io, Ada.Integer_Text_Io;
use Ada.Text_Io, Ada.Integer_Text_Io;

procedure Cachorro_Quente is

   task Cozinheiro is
      entry Faz_Cachorro(Numero : Integer);
   end Cozinheiro;

   procedure Pedido(Numero:Integer) renames
      Cozinheiro.Faz_Cachorro;

   task body Cozinheiro is
   begin
      Put_Line("Estou pronto para servir um cachorro");

      --mas só o serve se alguem o pedir.... não faz cachorros antes de serem pedidos

      accept Faz_Cachorro(Numero : Integer) do
         Put_Line("vai sair o primeiro cachorro");
         Put("estou a confeccionar o cachorro ");
         Put_Line("um pouco de mostarda");
         delay 2.0;
      end Faz_Cachorro;

      --espera pedidos de mais 4 cachorros
      for Index in 1..4 loop
         accept Faz_Cachorro(Numero : Integer) do
            Put("Vai sair o cachorro numero");
            Put(Numero, 3);
            New_Line;
            Put("estou a confeccionar o cachorro ");
            Put_Line("um pouco de mostarda");
            delay 2.0;
         end Faz_Cachorro;
      end loop;

      accept Faz_Cachorro(Numero : Integer) do
         Put("Vai sair o ultimo cachorro");
         New_Line;
         Put("estou a confeccionar o cachorro ");
         Put_Line("um pouco de mostarda");
         delay 2.0;
      end Faz_Cachorro;

      Put_Line("Acabaram-se os cachorros");
   end Cozinheiro;
```

```
begin
   for Index in 1..6 loop
      Pedido(Index);
      Put_Line("o cliente esta a comer o cachorro");
      New_Line;
   end loop;
   Put_Line("Não tenho mais fome");
end Cachorro_Quente;
```

Nº de accepts e de chamadas às entradas...

```
with Ada.Text_Io, Ada.Integer_Text_Io;
use Ada.Text_Io, Ada.Integer_Text_Io;

procedure Armazem is

   Numero_Cachorros : Integer := 0;

   -- neste caso existe um local onde são colocados os cachorros quentes
   -- e de onde são retirados quando um cliente pede cachorros quentes

   task Cinco_Cachorros;        -- esta tarefa adiciona cinco cachorros ao stock
   task Retira_Cinco_Cachorros; -- esta tarefa retira cinco cachorros ao stock

   task Armazenamento_Cachorros is
      entry Aumentar_Stock;        -- adiciona um cachorro ao stock
      entry Diminuir_stock;        -- apaga um cachorro do stock
   end Armazenamento_Cachorros;

   task body Armazenamento_Cachorros is
   begin
      for Index in 1..18 loop
         Put("estamos no loop ");
         put(index);
         select
            when Numero_Cachorros < 5 =>
            accept Aumentar_Stock do
               Numero_Cachorros:= Numero_Cachorros + 1;
               Put("adicionamos um cachorro ao stock. Temos agora ");
               Put(Numero_Cachorros, 3);
               New_Line;
            end Aumentar_Stock;
         or
            when Numero_Cachorros > 0 =>
            accept Diminuir_stock do
               Put_Line("retirar um cachorro do stock");
               Numero_Cachorros := Numero_Cachorros - 1;
            end Diminuir_stock;
         end select;
      end loop;
   end Armazenamento_Cachorros;

   task body Cinco_Cachorros is
   begin
      for Index in 1..5 loop
         delay 0.1;
         Armazenamento_Cachorros.Aumentar_Stock;
      end loop;
   end Cinco_Cachorros;

   task body Retira_Cinco_Cachorros is
   begin
      for Index in 1..5 loop
         delay 0.6;
         Armazenamento_Cachorros.Diminuir_Stock;
      end loop;
   end Retira_Cinco_Cachorros;

begin
   for Index in 1..4 loop
      delay 0.9;
      Armazenamento_Cachorros.Aumentar_Stock;
      Armazenamento_Cachorros.Diminuir_Stock;
   end loop;
end armazem;
```