Programación concurrente

Master de Computación

I Conceptos y recursos para la programación concurrente:

I.5 Sincronización basada en intercambio de mensajes.

J.M. Drake

M. Aldea

Concurrencia por intercambio de mensajes.

- Modelos de interacción.
- Transmisión síncrona de mensajes.
- Invocación remota de procedimientos.

Interacción entre procesos

- Históricamente los lenguajes de programación concurrente y las APIs de los sistemas operativos ofrecen un conjunto de primitivas que facilitan la interacción entre procesos de forma sencilla y eficiente.
- Estas primitivas deben hacer posible:

  Sincronización: Permite sincronizar los flujos de control de dos procesos.

  Exclusión mutua: Garantiza que mientras que un proceso accede a un recurso o actualiza una variables compartida, ningún otro proceso accede al mismo recurso o a variable compartida.

  Sincronización condicional: Garantiza que un recurso sólo es accedido cuando se encuentra en un determinado estado interno.
- Cada lenguaje o cada API debe ofrecer un conjunto completo de primitivas que permitan implementar los tres tipos de interacciones.

Modelos de interacción entre procesos

- Existe dos modelos semánticos básicos:

  Los procesos interaccionan intercambiando mensajes entre ellos.

  Los procesos interaccionan entre ellos accediendo a variables o regiones de memoria compartida.

- En el modelo basado en intercambio de mensajes la interacción de sincronización es explícita, y por el contrario, la exclusión mutua debe ser construida de forma indirecta.
- En el modelo basado en memoria compartida la interacción de exclusión mutua es directa y por el contrario, la sincronización debe ser implementada de forma indirecta.
- El modelo semántico de las primitivas hace referencia a la formulación de la primitiva de sincronización, y no al mecanismo físico o lógico con el que se implementa.

Sincronización por intercambio de mensajes.

- Se basa en la introducción de dos primitivas:

  send: Primitiva a través de la que un proceso envía un mensaje a otro proceso.

  wait: Primitiva a través de la que un proceso se suspende a la espera de recibir un mensaje enviado desde otro proceso.

- Ambas primitivas tienen una doble función: **intercambiar datos** y establecer una **sincronización** entre el proceso emisor y receptor.
- Admiten una gran variedad de modelos. Estos se diferencian en dos aspectos:

  Mecanismo de sincronización.

  Modo de designar los procesos fuente y destino.

Mecanismos de sincronización

- Existen tres protocolos básicos para implementar las primitivas Send-Wait:

  Envío Asíncrono: El proceso emisor continua con independencia del estado del receptor.

  Envío Síncrono (Rendezvous simple): El proceso emisor permanece suspendido hasta confirmar que el mensaje ha sido recibido. Las sentencias Send y Wait terminan síncronamente.

  Invocación Remota (Rendezvous completo): El proceso emisor permanece suspendido hasta confirmar que el mensaje ha sido aceptado. Emisor y receptor ejecutan síncronamente un segmento de código. Las sentencias Send y Wait terminan síncronamente.

- Todas las interacciones pueden llevar consigo una **transferencia de datos** que puede ser en un único sentido, o en ambos.

Aspectos sobre designación de los procesos.

- En función de como un proceso **hace referencia** al otro:

  Designación directa: Cada proceso utiliza el identificador del otro proceso.

  ```
  send Mensaje to Nombre_Proceso_Destino
  wait Mensaje from Nombre_Proceso_Emisor
  ```

  Designación indirecta: Ambos procesos hacen referencia a un identificador común.

  ```
  send Mensaje to Nobre_de_Canal
  wait Mensaje from Nombre_de_Canal
  ```

- **Simetría** de la comunicación:

  Simétrica: Ambos procesos conocen al otro con el que comunican.

  ```
  send Mensaje to Nombre_Proceso_Destino
  wait Mensaje from Nombre_Proceso_Emisor
  ```

  Asimétrica: El receptor no conoce al emisor.

  ```
  send Mensaje to Nombre_Proceso_Destino
  wait Mensaje
  ```

Transmisión síncrona de mensajes.

- Es un mecanismo síncrono de comunicación con denominación simétrica e indirecta a través de un identificador que denominamos canal.
- Sintaxis de las primitivas:

  var Canal : **channel of** Type_de_Mensaje;

  send Expresión **to** Canal;

  wait Variable **from** Canal;

- Semántica de las primitivas:

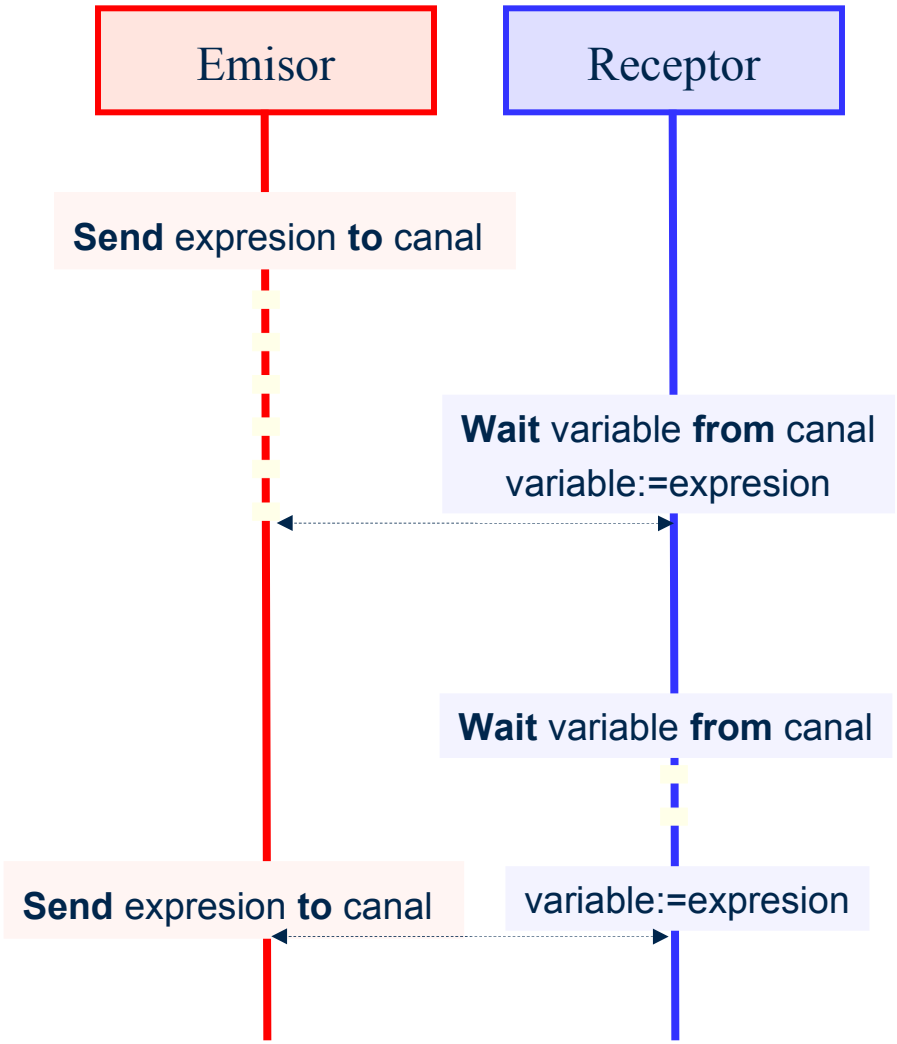
  El primer proceso que requiere el canal se suspende hasta que el segundo ejecuta la operación complementaria.

  Cuando ambos han accedido se realiza la asignación: Variable:= Expresión

  Las sentencias send y wait finalizan de forma síncrona.

Semántica de la comunicación sincrona

- Semántica de las primitivas:
  - El primer proceso que requiere el canal se suspende hasta que el segundo ejecuta la operación complementaria.
  - Cuando ambos han accedido se realiza la asignación:
    variable:= expresión
  - Las sentencias send y wait finalizan de forma síncrona.



Productor-Consumidor por comunicación síncrona.

```
program Productor_consumidor;
  type Canal_de_enteros = channel of Integer;
  var canal : Canal_de_enteros;

    process productor;
      var dato : integer;
    begin
      ..... -- El dato es producido
      send dato to canal;
      .....
    end;

                                    process consumidor;
                                      var dato : Integer;
                                    begin
                                      .....
                                      wait dato from canal;
                                      ..... -- El dato es consumido
                                    end;

begin
  cobegin
    productor; consumidor;
  coend
end.
```

Tipos de sentencias de aceptación de eventos en un select.

- Los tipos de sentencias que pueden encabezar las alternativas de una sentencia select, son:

  Recepción de un mensaje: Se activa cuando se recibe un mensaje.

  Envío de un mensaje: Se activa cuando se acepta el envío de un mensaje.

  Sentencia de temporización (timeout): Se activa si el select permanece en espera el tiempo especificado.

  Sentencia de finalización coordinada (terminate): Se activa, concluyendo el proceso, si todos los procesos hermanos (del mismo cobegin) han concluido o se encuentran en un select con una alternativa "terminate".

  Alternativa por defecto (else): Se ejecuta si al entrar en el select no se acepta alguna alternativa que esté dispuesta.

Ejemplo: Control de parque público.

```
program Control_Parque;
var entrada_1, entrada_2: channel of Integer;
      cuenta: Integer;

process torno_1;
   var n : Integer;
begin
  for n:=1 to 20 do
    send 1 to entrada_1;
end;
```

```
process torno_2;
  var n: Integer;
begin
  for n:=1 to 20 do
    send 1 to entrada_2;
end;
```

```
process contador;
  var  temp:Integer;
begin
   repeat
    select
      wait temp from entrada_1;
    or
      wait temp from entrada_2;
    or
      terminate;
    end select;
    cuenta:=cuenta+temp;
  forever;
end;
```

```
begin
    cuenta:=0;
    cobegin contador; torno_1; torno_2; coend;
    writeln(cuenta);
end.
```

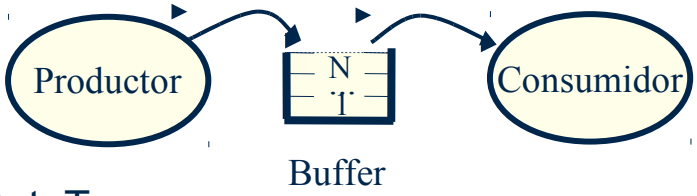
Condición de guarda

- Las sentencias de una alternativa select pueden estar protegidas por una **condición de guarda**.
- Sintaxis:

```
select
    when Expresión_Booleana_1 => Sentencia_de_Aceptación_Evento_1:
       Bloque_tras_Evento_1:
or
    when Expresión_Booleana_2=> Sentencia_de_Acepatación_Evento_2;
       Bloque_tras_Evento_1:
.....
else
     ....
end;
```

- Semántica: Al ejecutarse la sentencia select, se evalúan las condiciones de guarda. Aquellas que resultan ciertas quedan abiertas como alternativas, las otras cerradas.

Ejemplo: Buffer limitado a N elementos




```
type Acc_buffer = channel of DataType;
process Buffer (var pon, dame : Acc_buffer);
    const LONG_BUFFER = 31;
    var buff : array [0..LONG_BUFFER - 1] of DataType;
        tope, base, nAlmacenados : integer;
begin
    tope:= 0;  base:=0; nAlmacenados := 0;
    repeat
        select
           when (nAlmacenados < LONG_BUFFER) =>wait buff[tope] from pon;
            tope := (tope +1) mod LONG_BUFFER;
            nAlmacenados:= nAlmacenados +1;
         or
            when (nAlmacenados > 0) => send buff[base] to dame;
            base:= (base +1) mod LONG_BUFFER;
            nAlmacenados:= nAlmacenados -1;
        end;
    forever
end;
```

Alternativa timeout.

- La alternativa timeout establece que la espera a múltiples eventos esté temporizada. Si transcurre el tiempo establecido en ella, se ejecutan las sentencias que la siguen y finaliza la sentencia select.

- Ejemplo: proceso watchdog.

```
type Canal_WD = Synchronous;     -- Synchronous es  Channel sin datos
process Watchdog(var llamada: Canal_WD);
begin
   repeat
      select
        wait any from llamada;
      or
        timeout 2000;
        ....      -- Acciones de recuperación
      end;
   forever;
end;
```

Alternativa else.

- La alternativa else hace que la sentencia select sea no bloqueante. Si, cuando se ejecuta la sentencia select, ninguna alternativa puede ser aceptada, se ejecuta el bloque else y se termina la sentencia.
- Ejemplo: Recepción de mensaje no bloqueante.

```
function Rec_no_bloqueante(var dato:Integer;
                           var canal:CanalData): Boolean;
begin
   select
      wait dato from canal;
      Rec_no_bloqueante:=True;
   else
      Rec_no_bloqueante:=False;
   end;
end;
```

Invocación Remota

- Implementa el tercer mecanismo de sincronización de las primitivas Send-Wait:

  Envío Asíncrono: El proceso emisor continua con independencia del estado del receptor.

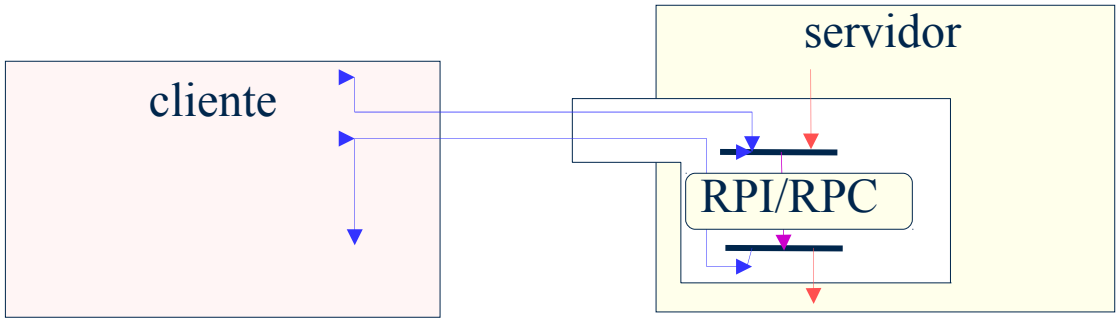
  Envío Síncrono (Rendezvous simple): El proceso emisor permanece suspendido hasta confirmar que el mensaje ha sido recibido. Las sentencias Send y Wait terminan síncronamente.

  Invocación Remota (Rendezvous completo): El proceso emisor permanece suspendido hasta confirmar que el mensaje ha sido aceptado. Emisor y receptor ejecutan síncronamente un segmento de código. Las sentencias Send y Wait terminan síncronamente.

- La invocación remota de procedimiento es un mecanismo de comunicación **síncrona**, **con denominación directa** y **asimétrica**, que sigue un formalismo semejante a la declaración e invocación de procedimientos remotos (RPC).

Formalismo semejante a los RPC

- En una invocación remota de procedimiento RPI el servidor ofrece una operación (entry) para que el cliente ejecute en el espacio del servidor con el thread del cliente y del servidor sincronizados.



Componentes de la invocación remota.

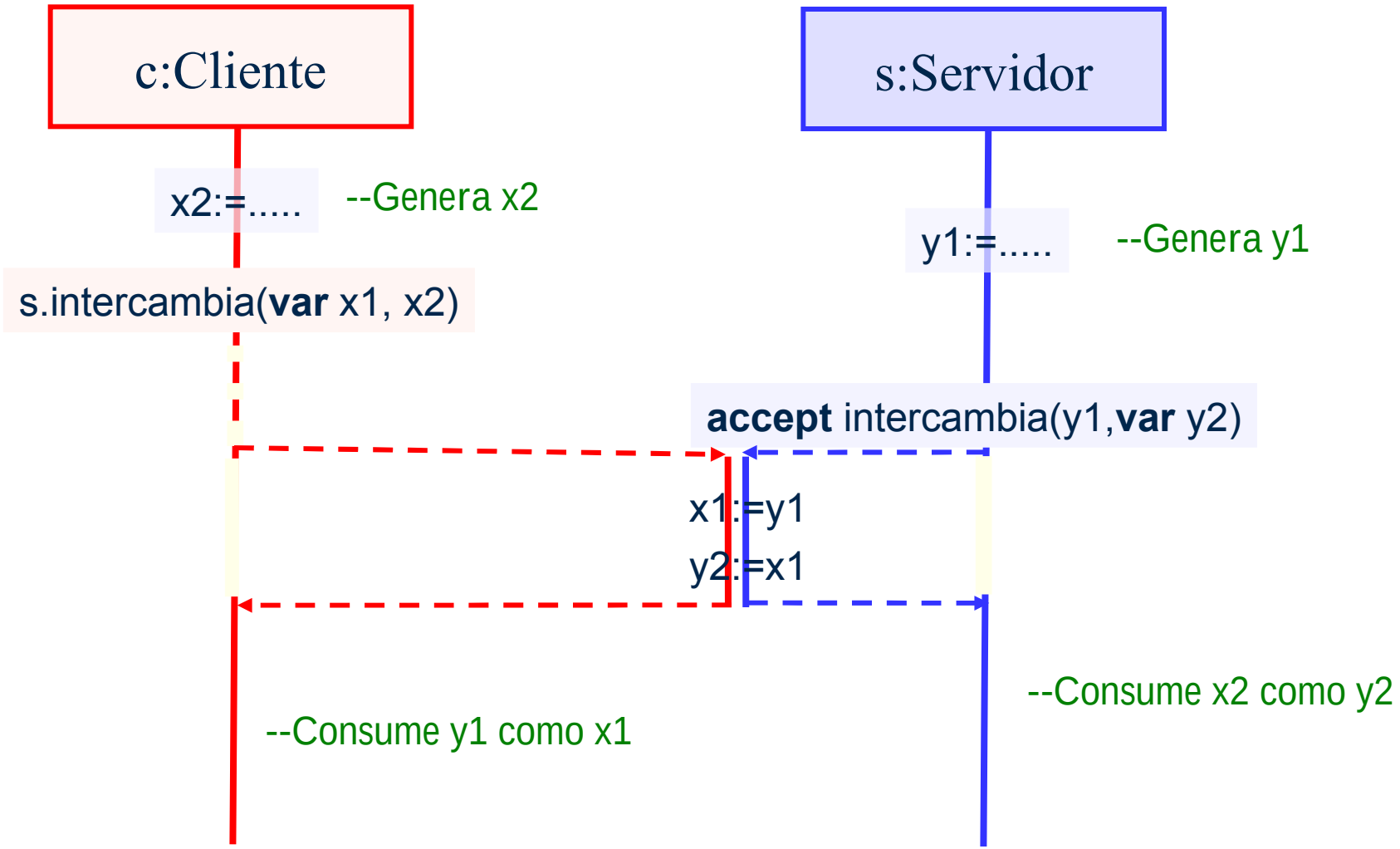
- En el **proceso propietario (Servicio)** del procedimiento remoto existen sentencias para declarar el procedimiento remoto y para aceptar su invocación:

```
process propietario;
    entry procedimientoRemoto(dato:TipoDato);
begin
    ....
    accept procedimientoRemoto(dato:TipoDato) do
        -- Bloque de sentencias
    ....
end;
```

- En el **proceso invocante (Cliente)** existe sentencias de invocación del procedimiento:

```
    ....
    propietario.procedimientoRemoto(nuevoDato);
    .....
```

Semántica de invocación de procedimiento remoto (1)



Ejemplo: Control de acceso a variable compartida.

```
Program AccesoConcurrente;
  var dato:VariableCompartida;

  process escritor;
     i:Integer;
  begin
     ...
     dato.write(i);
     ...
  end;

  process lector;
     n: integer;
  begin
     ...
     dato.read(n);
     ...
  end;

begin
   dato.write(0);
   cobegin
      escritor;
      lector;
   coend;
end;
```

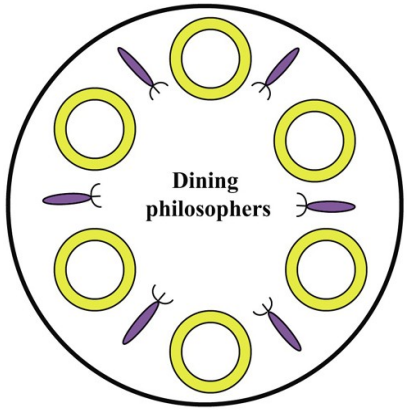
```
process type VariableCompartida;
   entry read(var v: integer);
   entry write(v: integer);

    var variable ; Integer;

begin
   accept write(v : Integer) do variable:=  v;
   repeat
        select
           accept write(v : Integer) do variable:=  v;
        or
            accept read(var v: Integer) do v:= variable;
        or
           terminate;
        end;
    forever;
end;
```

Ejemplo: Cena de los filósofos chinos

- Varios filósofos sentados en una mesa circular
- Cada filósofo sólo hace dos cosas:

  Comer: coge los cubiertos a izquierda y derecha y come durante un tiempo indeterminado

  Pensar: suelta los cubiertos y piensa durante un tiempo indeterminado



- Típico problema planteado en cursos de concurrencia

  Permite reproducir muchas situaciones de interés: interbloqueo, inanición, esperas activas, ...

  Puede implementarse de muy diversas formas: intercambio de mensajes, memoria compartida, ...

- Lo resolveremos de una manera especial:

  Introduciendo una silla menos que filósofos para evitar bloqueos

Ejemplo: Cena de los filósofos chinos (1).

```
program CenaFilosofosChino;
const N=5;
var i: Integer;

process type TipoPalillo:
   entry toma;
   entry deja;
begin .. end;

process type TipoFilosofo(nombre:Integer);
begin .. end;

process gestorSillas;
  entry seSienta;
  entry seLevanta;
begin ... end;

var filosofo:array [1..N] of TipoFilosofo;
      palillo: array [1..N] of TipoPalillo;

begin
    cobegin
        for i:=1 to N do begin palillo[i]; filosofo[i] end;
        gestorSillas;
   coend;
end.
```

Ejemplo: Cena de los filósofos chinos (2)

```
process type TipoPalillo;
  entry toma;
  entry deja;
  var tomado : Boolean;
begin
    tomado := false;
    repeat
       select
          when not tomado =>
              accept toma do tomado := true;
       or
           when tomado =>
              accept deja do  tomado := false;
       end;
    forever;
end;
```

Ejemplo: Cena de los filósofos chinos (3)

```
process gestorSillas;
  entry seSienta;
  entry seLevanta;

  var sillasLibres : Integer;

begin
    sillasLibres:= N-1;
    repeat
      select
          when sillasLibres >0 => accept seSienta do
            sillasLibres:= sillasLibres -1;
       or
          accept seLevanta do
            sillasLibres:= sillasLibres +1;
        end;
    forever;
end;
```

Ejemplo: Cena de los filósofos chinos (4)

```
process type
var

begin

                              mod
   for        to  !   do begin
            "    #                    -- Está pensando
          $   " %  $    "
       "    &        '%   "
       "    &(     '%   "
            )"    #                    -- Está comiendo
       "    &(     '%  *"
       "    &        '%  *"
          $   " %  + ,"  "
   end;
end;
```