PROGRAMACION CONCURRENTE

II.1 Comunicación síncrona

J.M. Drake

Notas:

Concurrencia por intercambio de mensajes.

- Modelos de interacción.
- Transmisión síncrona de mensajes.
- Invocación remota de procedimientos.
- Procesos auxiliares de interacción entre procesos.

Notas:

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**: Un proceso tiene acceso al estado de flujo de control que en ese instante tiene otro proceso.
  - **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 la 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 primitiva que permita implementar los tres tipos de interacciones.

Notas:

Modelos de interacción entre procesos

- Existe dos modelos semánticos básico:
  - 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.

Notas:

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 primitiva tienen una doble función: **intercambio de 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.

Notas:

Uno de los mecanismos abstractos de sincronización de procesos en un programa concurrente es el intercambio de mensajes. Básicamente este mecanismo se basa en dos primitivas:

* **send**: es la primitiva por la que un proceso envía un mensaje a otro proceso.
* **wait**: es la primitiva por la que un proceso se suspende a la espera de recibir un mensaje enviado desde otro proceso.

Ambas primitivas tienen una doble función: la de intercambio de datos y la de establecer un sincronismo entre el proceso emisor y el proceso receptor.

Aunque conceptualmente ambas sentencias son muy simples, admiten una gran variedad de modelos que serán el objeto de este tema.

Existen dos aspectos fundamentales que deben establecerse para desarrollar las primitivas de intercambio de mensajes:

* Mecanismo de sincronización que incorpora.
* Método de denominación de los procesos origen y destino de los mensajes.

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.
- En los mecanismos de comunicación síncrona los procesos que interaccionan intercambian información sobre el estado instantáneo de sus líneas de flujo de control.

Notas:

Existen tres alternativas de sincronización, al implementar la entencia SEND:

* **Envío asíncrono**: Tras ejecutar SEND, el proceso emisor continua la ejecución de su código, sin esperar a confirmar que el mensaje sea recibido por el proceso receptor. En este caso no se produce ninguna sincronización entre ambos procesos.

* **Envío síncrono (rendezvous simple):** El proceso que ejecuta la sentencia SEND permanece suspendido hasta confirmar que el proceso destino lo acepta. En este caso se produce una sincronización entre ambos procesos. Las sentencias SEND y WAIT concluyen síncronamente.

* **Invocación remota (rendezvous extendido):** El proceso que ejecuta la sentencia SEND permanece suspendido hasta que el proceso destino acepta la llamada. En este caso, ambos procesos permanecen sincronizados durante la ejecución de un segmento de código común, tras lo que las sentencias Send y Wait concluyen síncronamente.

El mecanismo de comunicación asíncrona se reduce a un proceso de transmisión de un mensaje. El proceso emisor, no obtiene información de si el proceso receptor lo recibe, ni en que estado se encuentra (puede que ya ni siquiera exista). Así mismo, cuando el proceso receptor recibe el mensaje no obtiene información del estado actual del proceso emisor (solo que en cierto instante anterior estuvo en un estado conocido). Este es un mero mecanismo de transferencia de información, y no un mecanismo de sincronización, por lo que no tiene un interés relevante.

En los mecanismos de comunicación síncrona, el aspecto más importante es la acción de sincronización entre los procesos que intervienen. La comunicación puede llevar consigo, o no, una transferencia de información entre los procesos, que a su vez puede ser en una o en las dos direcciones.

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

Notas:

Existen dos aspectos importantes a considerar respecto a la **forma en que se nombran los procesos** que intervienen en el intercambio de mensajes:

* Si los procesos se designan directamente o indirectamente.
* Si los procesos emisor y receptor intervienen de una forma simétrica o asimétrica.

En las primitivas que designan de forma **directa** los procesos que intervienen, utilizan un identificador o nombre que identifica de forma única al proceso dentro del programa. Las que utilizan un método de denominación **indirecta**, en ambas sentencias se hace referencia a un nombre o identificador común, a través del cual el sistema induce que son complementarias. Ese nombre suele recibir la denominación de canal o buzón.

En la forma **simétrica,** el receptor conoce en su llamada a la sentencia WAIT el nombre del proceso emisor de la que espera el mensaje, por el contrario, en la forma **asimétrica** el proceso receptor no necesita conocer el proceso origen de la llamada.

Transmisión síncrona de mensajes.

- Modelos utilizados en OCCAM y CSP.
- 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.

Notas:

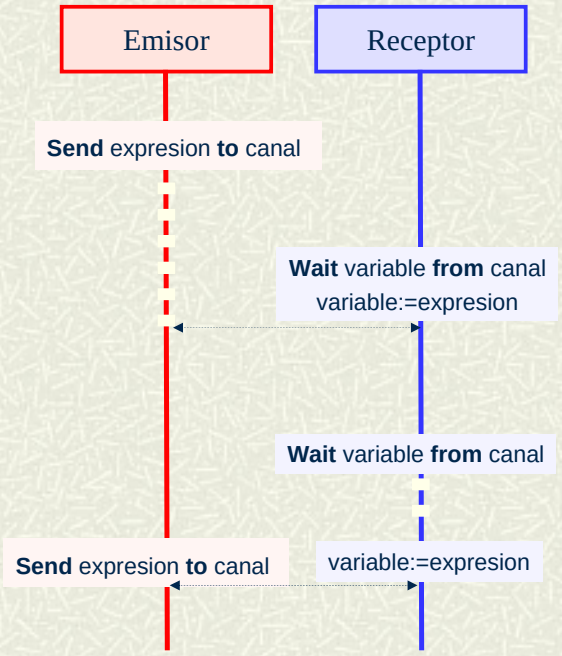
El modelo de comunicación por canal es un mecanismo de envío síncrono de mensajes entre dos procesos en una sola dirección, con denominación indirecta de los procesos emisor y receptor, a través de un identificador que se denomina canal.

Este modelo es el que se utiliza en OCCAM y seguiremos el formalismo CSP de Hoare (1985).

La comunicación por un canal (de ahí su nombre) es entre dos procesos particulares que previamente han accedido a sus terminales.

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.



Notas:

Las operaciones sobre el canal son sincronizadas. El primer proceso que requiere al canal se bloquea hasta que sobre el mismo se ejecuta la operación complementaria. En ese momento se lleva a cabo el rendezvous, con la asignación a la variable del valor resultante de evaluar la expresión, siguiendo a continuación ambos procesos su línea de flujo de control de forma independiente.

Funcionalmente el resultado de la comunicación es un proceso de asignación:

variable := expresión

solo que "variable" es del proceso receptor, mientras que expresión es del proceso emisor.

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.
```


Notas:

Como ejemplo comprobemos la construcción del problema productor-consumidor utilizando la comunicación síncrona a través de un canal.

En este caso, la comunicación a través del canal síncrono, resuelve todos los requerimientos de sincronización entre los procesos Productor y Consumidor.

Espera selectiva.

- Las sentencias send y wait suspende el proceso a la espera de ser atendida,por un **único interactuante**. Esto supone una restricción muy fuerte, ya que un servidor debe quedar a la espera de un único evento.
- La sentencia **select** suspende el proceso a la espera de múltiples eventos:

```
select
    Sentencia_de_aceptación_de_evento_1
    Bloque_de_sentencias_tras_evento_1
or
    Sentencia de aceptación_de_evento_2
    Bloque_de_sentencias_tras_evento_2
or
    .....
else
    Bloque_de_sentencias_de_escape
end;
```


Notas:

La inclusión de solo una sentencia WAIT para recibir eventos por parte de un proceso, constituye una limitación muy grande. Un proceso que ejecuta esta sentencia, queda suspendido a la espera de recibir el correspondiente mensaje, que es el único mecanismo que lo activará. Se necesita disponer de una estructura que permita a un proceso suspendido activarse por múltiples eventos. Estos se consigue con la sentencia **select ... or ... or ... end;**

Select es una sentencia estructurada, que contiene un conjunto de sentencias capaces de ser punto de acceso de evento, y cuya ejecución deja al proceso suspendido y en espera de cualquiera de los eventos alternativos que pueden ser atendidos. Cuando el primero de estos eventos llega, se ejecuta el bloque de programa que comienza con la sentencia que lo acepta.

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.

Notas:

Las sentencias que pueden encabezar cada bloque dentro de una estructura select deben ser de uno de los siguientes tipos:

* **Recepción de un mensaje**: Se activa cuando se recibe un mensaje por el canal del que se recibe.

* **Envío de un mensaje**: Se activa cuando acepta el mensaje el proceso al que se ha enviado.

* **Sentencia de temporización** (timeout): Se activa si transcurre el tiempo de timeout establecido, sin que haya llegado un evento a una de las restantes alternativas de la estructura select..

* **Sentencia de propuesta de terminación** (terminate): Se activa terminando el proceso, si todas las tareas hermanas (del mismo "cobegin") han concluido o se encuentran también en un select con una alternativa "terminate".

* **Alternativa por defecto** (sigue a la palabra de control **else**): Este bloque de programa se ejecuta si al entrar en la sentencia select, ninguna otra alternativa es satisfecha.

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.
```


Notas:

Condición de guarda

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

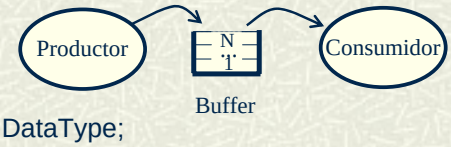
```
select
    when Expresión_Booleana_1 => Sentencia_de_Aceptación_Evento_1:
      Bloque_tras_Evento_1:
or
    when Expresión_Booleana_2=> Sentencia_de_Acepatación_Evento_2;
      Bloque_tras_Evento_1:
.....
else
     ....
end;
```

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

Notas:

Las opciones establecidas en una sentencia select se pueden proteger mediante unas **condiciones booleana de guarda**. Cuando se ejecuta un select se evalúan todas las expresiones de guarda, aquella opciones en las que el valor de la expresión de guarda sea "false", quedan cerradas en esa ejecución y no son sensibles a eventos, son sensibles a eventos las restantes (esto es, en las que la evaluación de la condición de guarda ha resultado "true" y las que no tienen condición de guarda).

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 (N_almacenados < LONG_BUFFER) =>wait buff[tope] from pon;
                tope := (tope +1) mod LONG_BUFFER;
                nAlmacenados:= nAlmacenados +1;
         or
           when (nAlmacenados > 0) =>       wait buff[base] from dame;
                base:= (base +1) mod LONG_BUFFER;
                nAlmacenados:= nAlmacenados -1;
        end;
    forever
end;
```


Notas:

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;
```


Notas:

Cuando un proceso se ha diseñado para que atienda eventos alternativos a través de una estructura "select", la alternativa timeout permite que la suspensión del proceso a la espera este temporizada, esto es, si transcurre un intervalo de tiempo prefijado sin que se reciban alguno de los eventos esperados, concluye la espera y continúa el programa con la alternativa que corresponda según sea el evento.

Como ejemplo de sentencia timeout se muestra la estructura de un proceso de "watchdog", el cual mientras que el proceso cuya actividad vigila está activo recibe un mensaje cada intervalo de tiempo inferior a 2 segundos. Cuando deja de recibirlos, entra en acción y ejecuta una secuencia de recuperación.

"synchronous" es un tipo de canal predefinido que solo transmite evento y no transmite ningún tipo de dato. "any" es una variable predefinida que solo se utiliza con canales de tipo predefinidos synchronous, y que solo sirve para mantener el formato de la sentencia de lectura de mensaje de un canal.

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;
```


Notas:

Cuando un proceso se ha diseñado para que atienda eventos alternativos a través de una estructura "select", la alternativa "else" permite que en el caso de que se entre en la estructura e inmediatamente no pueda ser tomada una de las otras alternativas, concluya el select ejecutándose el bloque de sentencias que siguen a la palabra "else". A todos los efectos una alternativa "else" es equivalente a una alternativa "timeout" con espera nula.

En el ejemplo de sentencia con alternativa "else", se propone una función que constituye una recepción no bloqueante de un mensaje por un canal, esto es, requiere un mensaje por un canal, caso de que el mensaje se haya enviado la función lo retorna, pero en el caso de que no se haya enviado, la fiunción concluye.