

Programación Concurrente 2016

Clase 8

Facultad de Informática
UNLP



Resumen de la clase anterior

Programación Distribuida

4 Mecanismos equivalentes de PM

Patrones de interacción entre procesos

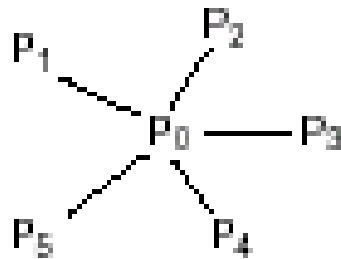
Mensajes Asíncronos

- Canales, send, receive, empty
- Filtros: Carac_a_Linea. Red de Ordenación
- Clientes y servidores: Monitores Activos. Continuidad conversacional.

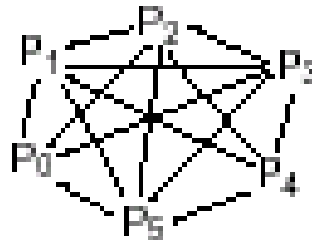
PMA. Pares (peers) interactuantes: Intercambio de valores

Ejemplo donde los procesadores están conectados por tres modelos de arquitectura: **centralizado, simétrico y en anillo circular**.

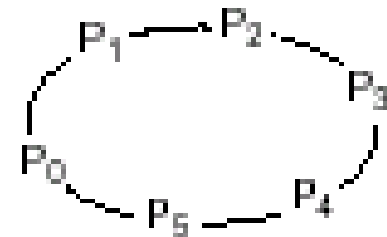
Problema: c/ proceso tiene un dato local **V** y los N procesos deben saber cuál es el menor y cuál el mayor de los valores



(a) Centralized solution



(b) Symmetric solution



(c) Ring solution

La arq. centralizada es apta para una solución en que todos envían su dato local **V** al procesador central, éste ordena los N datos y reenvía la información del mayor y menor a todos los procesos.

⇒ $2(N-1)$ mensajes.

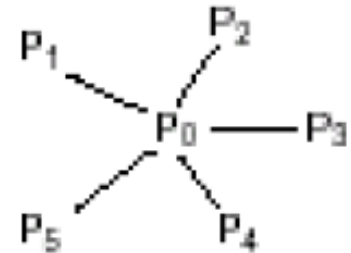
Si $p[0]$ dispone de una primitiva broadcast se reduce a N mensajes.

PMA. Pares (peers) interactuantes: Intercambio de valores (Solución centralizada)

chan vaores(INT), resultados[n] (INT minimo, INT maximo);
Process P[0] { #Proceso coordinador. v ya está inicializado.
INT v; INT nuevo, minimo = v, máximo = v;

```
FOR [i=1 to n-1] {  
    receive valores (nuevo);  
    IF (nuevo < minimo)  
        minimo = nuevo;  
    IF (nuevo > maximo)  
        maximo = nuevo;  
}  
FOR [i=1 to n-1]  
    send resultados [i] (minimo, maximo);
```

```
}  
Process P[i=1 to n-1] { # Proceso cliente. v ya está inicializado.  
INT v; INT minimo, máximo;  
    send valores (v);  
    receive resultados [i] (minimo, maximo);  
}
```



PMA. Pares (peers) interactuantes: Intercambio de valores (Solución simétrica)

En la arquitectura simétrica o “full connected” hay un canal entre cada par de procesos. Todos los procesos ejecutan el mismo algoritmo.

Cada proceso transmite su dato local v a los $n-1$ restantes procesos. Luego recibe y procesa los $n-1$ datos que le faltan, de modo que **en paralelo** toda la arquitectura está calculando el mínimo y el máximo y toda la arquitectura tiene acceso a los n datos.

Ejemplo de solución *SPMD*: cada proceso ejecuta el mismo programa pero trabaja sobre datos distintos

⇒ $n(n-1)$ mensajes

Si disponemos de una primitiva de broadcast, serán nuevamente n mensajes.

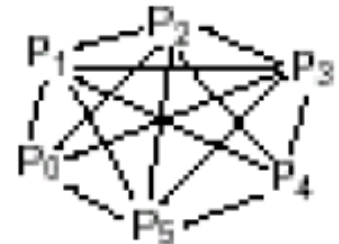
PMA. Pares (peers) interactuantes: Intercambio de valores (Solución simétrica)

Chan valores[n] (INT);

```
Process P[i=0 to n-1] {                                # Todos los procesos idénticos
  INT v;                                           # asumimos que v fue inicializado
  INT nuevo, minimo = v, máximo=v; # estado inicial,

  FOR [k=0 to n-1 st k <> i] # envío del dato local
    send valores[k] (v);

  FOR [k=0 to n-1 st k <> i] { # recibo y proceso los datos remotos
    receive valores[k] (nuevo);
    IF (nuevo < minimo)
      minimo = nuevo;
    IF (nuevo > maximo)
      maximo = nuevo;
  }
}
```



PMA. Pares (peers) interactuantes: Intercambio de valores (Solución en anillo circular)

Un tercer modo de organizar la solución es tener un anillo donde $P[i]$ recibe mensajes de $P[i-1]$ y envía mensajes a $P[i+1]$.

$P[n-1]$ tiene como sucesor a $P[0]$

Esquema de 2 etapas. En la 1ra c/ proceso recibe dos valores y los compara con su valor local, transmitiendo un máximo local y un mínimo local a su sucesor. En la 2da etapa todos deben recibir la circulación del máximo y el mínimo global.

***$P[0]$ deberá ser algo diferente para “arrancar” el procesamiento.
Se requerirán 2 $(n-1)$ mensajes.***

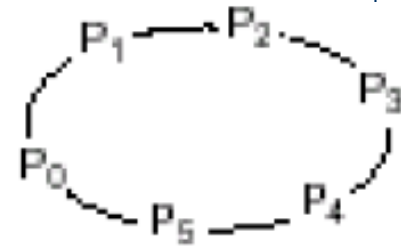
Notar que si bien el número de mensajes es lineal (igual que en la centralizada) los tiempos pueden ser muy diferentes.

Por qué???

PMA. Pares (peers) interactuantes: Intercambio de valores (Solución en anillo circular)

```
chan valores[n] (INT minimo, INT maximo);  
Process P[0] { # Proceso que inicia los intercambios.  
  INT v; INT minimo = v, máximo=v;  
  # Enviar v a P[1]  
  send valores[1] (minimo, maximo);  
  # Recibir los valores minimo y maximo globales de P[n-1] y pasarlos  
  receive valores[0] (minimo, maximo);  
  send valores[1] (minimo, maximo); }
```

```
Process P[i = 1 to n-1] { # Procesos del anillo.  
  INT v; INT minimo, máximo;  
  # Recibe los valores minimo y maximo hasta P[i-1]  
  receive valores[i] (minimo, maximo);  
  IF (v < minimo) minimo = v;  
  IF (v > maximo) maximo = v;  
  # Enviar el minimo y maximo al proceso i+1  
  send valores[i+1 MOD n] (minimo, maximo);  
  # Esperar el minimo y maximo global  
  receive valores[i] (minimo, maximo);  
  IF (i < n-1) send valores[i+1] (minimo, maximo); }
```



PMA. Pares (peers) interactuantes: Intercambio de valores. Comentarios sobre las soluciones

Simétrica es la más corta y sencilla de programar, pero usa el mayor número de mensajes (si no hay broadcast)

Pueden transmitirse en paralelo si la red soporta transmisiones concurrentes, pero el overhead de comunicación acota el speedup.

Centralizada y **anillo** usan n° lineal de mensajes, pero tienen distintos patrones de comunicación que llevan a distinta performance

En **centralizada**, los msgs al coordinador se envían casi al mismo tiempo \Rightarrow sólo el 1er *receive* del coordinador demora mucho

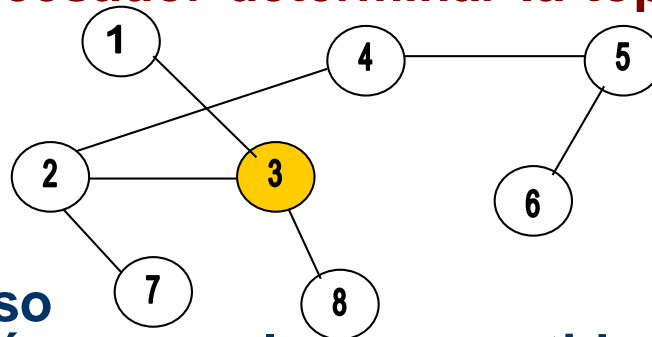
En **anillo**, todos los procesos son *productores* y *consumidores*. El último tiene que esperar a que todos los otros (uno por vez) reciban un msg, hacer poco cómputo, y enviar su resultado.

Los msg circulan 2 veces completas por el anillo \Rightarrow Solución inherentemente lineal y lenta para este problema, pero puede funcionar si cada proceso tiene mucho cómputo.

PMA. Algoritmo Heartbeat para el cálculo de la topología de una red

Procesadores conectados por canales bidireccionales
C/ uno se comunica sólo con sus vecinos y conoce esos links

Cómo puede cada procesador determinar la topología completa de la red?



Modelización:

Procesador \Rightarrow proceso

Links de comunicación \Rightarrow canales compartidos.

Soluciones posibles:

- los procesos tienen acceso a MC
- distribuida: los vecinos interactúan para intercambiar info local

Algoritmo Heartbeat \rightarrow se expande, enviando información; luego se contrae, incorporando nueva información

PMA. Algoritmo Heartbeat para el cálculo de la topología de una red

Solución con Variables Compartidas

Procesos *Nodo*[*p*:1..*n*]

Vecinos de *p*:

vecinos[1:*n*], t.q *vecinos*[*q*] true en *Nodo*[*p*] si *q* vecino de *p*

Problema: computar *top* (matriz de adyacencia), donde *top*[*p*,*q*] true si *p* y *q* son vecinos

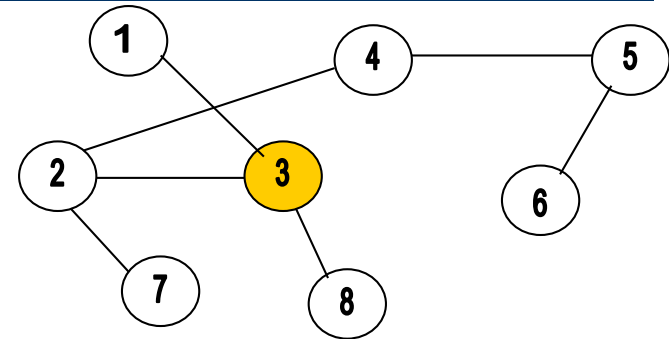
```
var top[1:n,1:n] : bool := ([n*n] false)
```

```
Process Nodo[p:1..n] { bool vecinos[1:n];
```

```
  # vecinos[q] es true si q es un vecino de Nodo[p]
```

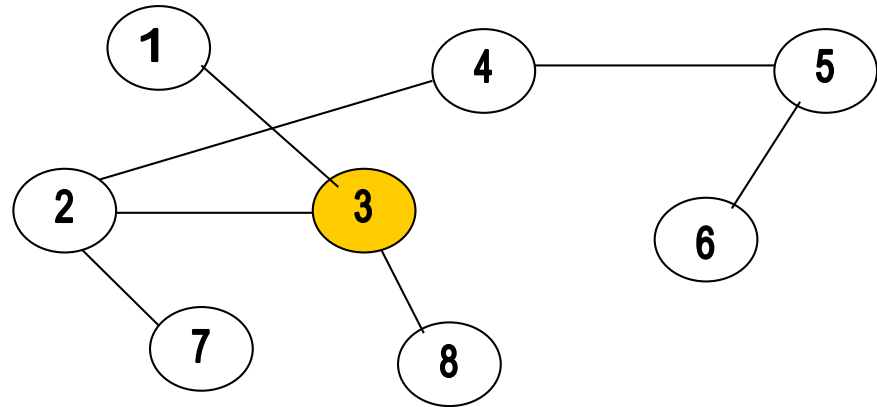
```
  for [q = 1 to n st vecinos[q] ] top[p,q] = true;
```

```
  { top[p,1:n] = vecinos[1:n] }
```



PMA. Algoritmo Heartbeat para el cálculo de la topología de una red

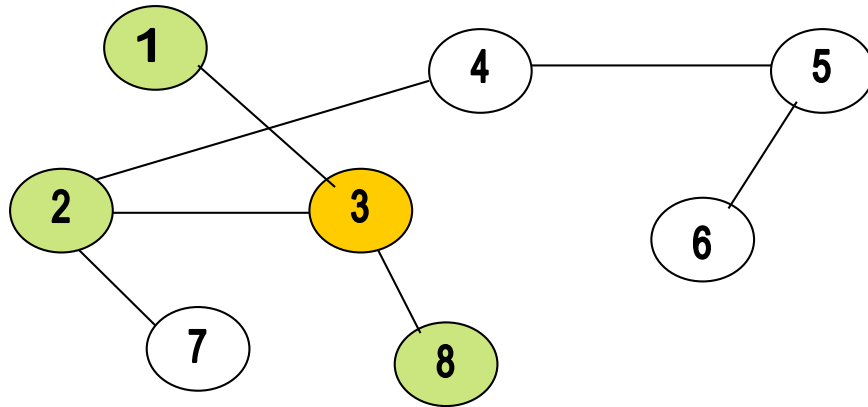
Solución Distribuida



Inicialmente en el nodo 3:

	1	2	3	4	5	6	7	8
1								
2								
3	T	T	T					T
4								
5								
6								
7								
8								

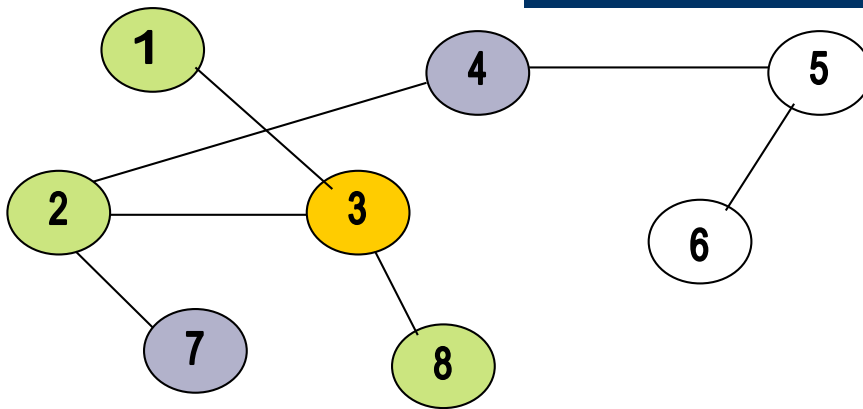
PMA. Algoritmo Heartbeat para el cálculo de la topología de una red



Después de una ronda, en nodo 3:

	1	2	3	4	5	6	7	8
1	T		T					
2		T	T	T			T	
3	T	T	T					T
4								
5								
6								
7								
8			T					T

PMA. Algoritmo Heartbeat para el cálculo de la topología de una red



Después de dos rondas, en nodo 3:

	1	2	3	4	5	6	7	8
1	T		T					
2		T	T	T			T	
3	T	T	T					T
4		T		T	T			
5								
6								
7		T					T	
8			T					T

PMA. Algoritmo Heartbeat para el cálculo de la topología de una red

Después de r rondas lo siguiente es true $\forall p$:

RONDA: ($\forall q: 1 \leq q \leq n: (\text{dist}(p,q) \leq r \Rightarrow \text{top}[q,*] \text{ lleno})$)

C/ nodo debe ejecutar un n° de rondas p/ conocer la top completa

Si el *diámetro D* de la red es conocido \Rightarrow

chan topologia[1:n]([1:n,1:n] bool) # un canal privado por nodo

Process Nodo[p:1..n] { var vecinos[1:n] : bool

bool top[1:n,1:n] = ([n*n] false)

top[p,1..n] = vecinos # inicialmente vecinos[q] true si q es vecino de Nodo[p]

int r = 0; bool nuevatop[1:n,1:n];

while (r < D) {

 # envía conocimiento local a sus vecinos

 for [q = 1 to n st vecinos[q]] send topologia[q](top);

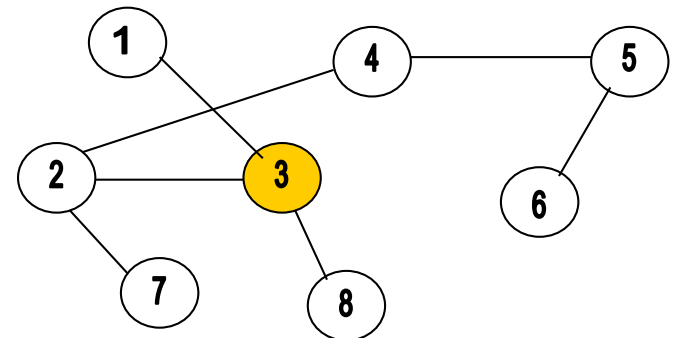
 # recibe las topologías y hace or con su top

 for [q = 1 to n st vecinos[q]] {
 receive topologia[p](nuevatop)
 top = top or nuevatop }

 r = r + 1

 }

}



PMA. Algoritmo Heartbeat para el cálculo de la topología de una red

- rara vez se conoce el valor de D
- excesivo intercambio de mensajes \Rightarrow los procesos cercanos al “centro” conocen la topología más pronto y no aprenden nada nuevo en los intercambios
- el tema de la terminación \Rightarrow local o distribuida?

Cómo se pueden solucionar estos problemas?

PMA. Algoritmo Heartbeat para el cálculo de la topología de una red

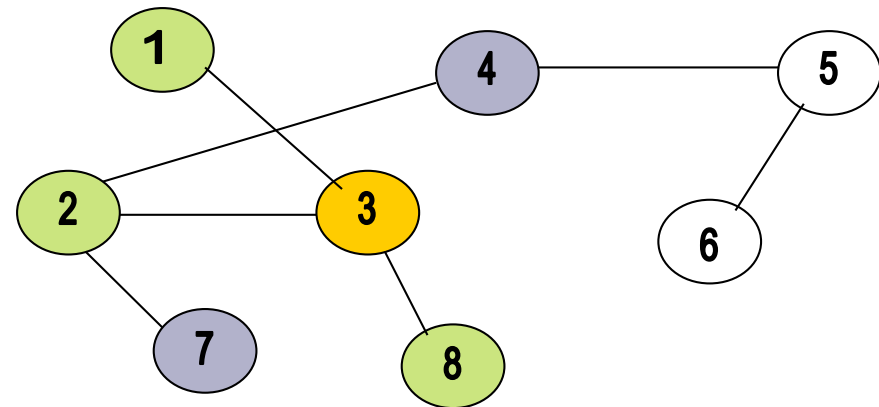
Después de r rondas, p conoce la topología a distancia r de él.

Para cada nodo q dentro de la distancia r de p , los vecinos de q estarán almacenados en la fila q de top

$\Rightarrow p$ ejecutó las rondas suficientes tan pronto como cada fila de top tiene algún valor true

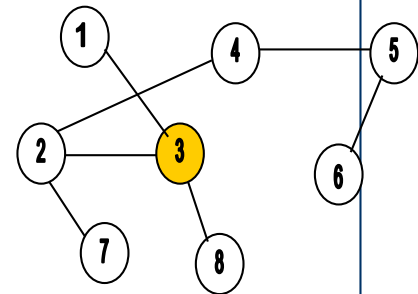
Luego necesita ejecutar una última ronda para intercambiar la topología con sus vecinos

No siempre la terminación se puede determinar localmente



PMA. Algoritmo Heartbeat para el cálculo de la topología de una red

```
chan topologia[1:n](emisor : int; listo : bool; top : [1:n,1:n] bool)
Process Nodo[p:1..n] { bool vecinos[1:n];
    # inicialmente vecinos[q] true si q es vecino de Nodo[p]
    bool activo[1:n] = vecinos          # vecinos aún activos
    bool top[1:n,1:n] = ([n*n]false)    # vecinos conocidos
    int r = 0; bool listo = false;
    int emisor; bool qlisto; bool nuevatop[1:n,1:n];
    top[p,1..n] = vecinos;              # llena la fila para los vecinos
    while (not listo) {
        # envía conocimiento local de la topología a sus vecinos
        for [q = 1 to n st activo[q] ] send topologia[q](p,false,top);
        # recibe las topologías y hace or con su top
        for [q = 1 to n st activo[q] ] {
            receive topologia[p](emisor,qlisto,nuevatop);
            top = top or nuevatop;
            if (qlisto) activo[emisor] = false; }
        if (todas las filas de top tiene 1 entry true) listo=true;
        r := r + 1 }
    # envía topología a todos sus vecinos aún activos
    for [q = 1 to n st activo[q] ] send topologia[q](p,listo,top);
    # recibe un mensaje de cada uno para limpiar el canal
    for [q=1 to n st activo[q]] receive topologia[p](emisor,d,nuevatop) }
```



Extensión de lenguajes secuenciales con bibliotecas específicas

Una técnica muy utilizada es el desarrollo de bibliotecas de funciones que permiten comunicar/sincronizar procesos, no dependientes de un lenguaje de programación determinado.

Las soluciones basadas en bibliotecas pueden ser menos eficientes que los lenguajes “reales” de programación concurrente, aunque permiten “agregarse” al código secuencial con bajo costo de desarrollo.

Las arquitecturas distribuidas han potenciado las soluciones basadas en PVM o MPI, que son básicamente bibliotecas de comunicaciones. Un esquema anterior (y original) es el de LINDA.

Los pgms MPI usan un estilo SPMD. C/ proceso ejecuta una copia del mismo programa, y puede tomar distintas acciones de acuerdo a su “identidad”. Las instancias interactúan llamando a funciones MPI, que soportan comunicación proceso-a-proceso y grupales

La biblioteca MPI. Un ejemplo simple

Dos procesos intercambian valores (14 y 25). Solución empleando MPI:

```
# include <mpi.h>
main (INT argc, CHAR *argv [ ] ) {
    INT myid, otherid, size;
    INT length=1, tag=1;
    INT myvalue, othervalue;
    MPI_status status;
    MPI_Init (&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_Rank (MPI_COMM_WORLD, &myid);
    IF (myid == 0) {
        otherid = 1; myvalue=14;
    } ELSE {
        otherid=0; myvalue=25;
    }
    MPI_send (&myvalue, length, MPI_INT, otherid, tag,MPI_COMM_WORLD);
    MPI_recv (&othervalue, length, MPI_INT, MPI_any_source, tag,
              MPI_COMM_WORLD, &status);
    printf ("process %d received a %d\n", myid, othervalue);
    MPI_Finalize ( ); }
```

La biblioteca MPI

MPI_Init inicializa la biblioteca MPI y obtiene una copia de los comandos pasados al programa. La variable **MPI_COMM_WORLD** es inicializada con el conjunto de procesos que se arrancan.

MPI_COMM_Size determina el número de procesos arrancados (2)

MPI_COMM_Rank determina la id del proceso (0 a size-1)

MPI_Finalize Termina este proceso y libera la biblioteca MPI.

MPI_Send envía un mensaje a otro proceso. Los argumentos son: el buffer que contiene el mensaje, el nro de elementos a enviar, el tipo de datos del mensaje, la identidad del proceso destino, un tag del usuario para identificar el tipo de mensaje y la variable **MPI_COMM_WORLD**.

MPI_Recv recibe un mensaje de otro proceso. Los argumentos son: el buffer en el cual poner el mensaje, el número de elementos del mensaje, el tipo de datos del mensaje, la identidad del proceso que envía o “no importa”= **MPI_any_source**, un tag del usuario para identificar el tipo de mensaje, el grupo de comunicación y el status de retorno.

Pasaje de Mensajes Sincrónicos

La principal diferencia con PMA es que la primitiva de transmisión (llamémosla `sync_send`) es bloqueante.

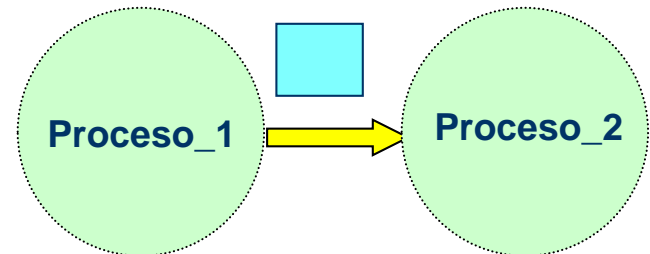
⇒ El trasmisor queda esperando que el mensaje sea recibido.

⇒ *La cola de mensajes asociada con un send sobre un canal se reduce a 1 mensaje ⇒ MENOS memoria.*

⇒ Naturalmente el grado de concurrencia se reduce respecto de la sincronización por PMA (*siempre un proceso se bloquea*)

Los canales son punto a punto (1 emisor – 1 receptor)

Si bien `send` y `sync_send` son similares (en algunos casos intercambiables) la semántica es diferente y las posibilidades de deadlock mayores en comunicación sincrónica.



Pasaje de Mensajes Sincrónicos

Ejemplo: Productor - consumidor

```
chan valores(INT);
Process Productor {
    INT datos[n];
    FOR [i=0 to n-1] {
        # Hacer cálculos productor
        sync_send valores (datos[i]);
    }
}
Process Consumidor {
    INT resultados[n];
    FOR [i=0 to n-1] {
        receive valores (resultados[i]);
        # Hacer cálculos consumidor
    }
}
```



Pasaje de Mensajes Sincrónicos

Comentarios.

Si los cálculos del productor se realizan mucho más rápido que los del consumidor en las primeras n_1 operaciones, y luego se realizan mucho más lento durante otras n_1 interacciones:

✓ Con PMS los pares send/receive se completarán asumiendo la demora del proceso que más tiempo consuma. Si la relación de tiempo fuera 10-1 significaría multiplicar por 10 los tiempos totales.

✓ Con PMA, al principio el productor es más rápido y sus mensajes se encolan. Luego el consumidor es más rápido y “descuenta” tiempo consumiendo la cola de mensajes.

⇒ **Mayor concurrencia en AMP.** Para lograr el mismo efecto en PMS se debe interponer un proceso “buffer”.

Pasaje de Mensajes Sincrónicos

Comentarios.

La concurrencia también se reduce en algunas interacciones C/S:

- Cuando un cliente está liberando un recurso, no habría motivos para demorarlo hasta que el servidor reciba el mensaje, pero con PMS se tiene que demorar.
- Otro ejemplo se da cuando un cliente quiere escribir en un display gráfico, un archivo u otro dispositivo manejado por un proceso servidor. Normalmente el cliente quiere seguir inmediatamente después de un pedido de *write*.

Otra desventaja del PMS es la mayor probabilidad de deadlock. El programador debe ser cuidadoso de que todas las sentencias send y receive hagan matching.

Pasaje de Mensajes Sincrónicos.

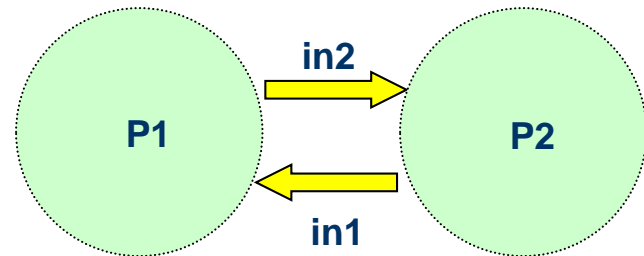
El tema del deadlock...

Dos procesos que intercambian valores:

```
chan in1(INT), in2(INT);
```

```
Process P1 {  
  INT value1 = 1, value2;  
  sync_send in2(value1);  
  receive in1 (value2);  
}
```

```
Process P2 {  
  INT value1, value2=2;  
  sync_send in1(value2);  
  receive in2 (value1);  
}
```



Con mensajes sincrónicos esta solución entra en deadlock.

Por qué??

Con AMP esta resolución es válida.

(Además, al ser simétrica es escalable fácilmente)

Pasaje de Mensajes Sincrónicos. El lenguaje CSP (Hoare, 1978)

CSP (Communicating Sequential Processes, Hoare 1978) fue uno de los desarrollos fundamentales en Programación Concurrente. Muchos lenguajes reales (OCCAM, ADA, MPD) se basan en CSP.

Las ideas básicas introducidas por Hoare fueron PMS y **comunicación guardada**: PM con waiting selectivo

Canal: link directo entre dos procesos en lugar de mailbox global. Son half-duplex y nominados.

Las sentencias de Entrada (? o **query**) y Salida (! o **shriek** o **bang**) son el único medio por el cual los procesos se comunican.

Para que se produzca la comunicación, deben **matchear**, y luego **se ejecutan simultáneamente**.

Efecto: sentencia de asignación distribuida.

Pasaje de Mensajes Sincrónicos. El lenguaje CSP (Hoare, 1978)

Formas generales de las sentencias de comunicación:

Destino ! port(e_1, \dots, e_n);

Fuente ? port(x_1, \dots, x_n);

Destino y *Fuente* nombran un proceso simple, o un elemento de un arreglo de procesos.

Fuente puede nombrar *cualquier* elemento de un arreglo (*Fuente*[*]).

port es un canal de comunicación simple en el proceso destino o un elemento de un arreglo de ports en el proceso destino.

Los ports se usan p/ distinguir entre distintas clases de mensajes que un proceso podría recibir (puede omitirse si es sólo uno)

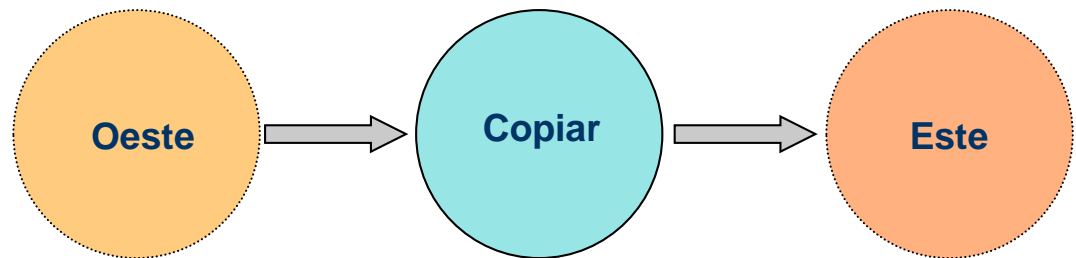
Dos procesos se comunican cuando ejecutan sentencias de comunicación que hacen matching.

A ! canaluno(dato); B ? canaluno(resultado);

CSP. Ejemplos

Proceso filtro que copia caracteres recibidos del proceso Oeste al proceso Este:

```
Process Copiar {  
  CHAR c;  
  do true →  
    Oeste ? c;  
    Este ! c ;  
  od  
}
```



Las operaciones de comunicación (? y !) pueden ser ***guardadas*** (equivalente a un AWAIT hasta que una condición sea verdadera).

El **do** e **if** de CSP usan los *comandos guardados* de Dijkstra ($B \rightarrow S$)

CSP. Ejemplos

Server que calcula el MCD de dos enteros con algoritmo de Euclides.
MCD espera recibir entrada en su port **args** desde un cliente.
Envía la respuesta al port **resultado** del cliente.

```
Process MCD {  
  INT Id, x, y;  
  do true →  
    Cliente[*] ? args(id, x,y);  
    # Lo que sigue se repite hasta que x==y  
    do x > y → x := x - y;  
    □ x < y → y := y - x;  
    od  
    Cliente[Id] ! resultado(x);  
  od  
}
```

Cliente[i] se comunica con **MCD** ejecutando:
... **MCD** ! args(i, v1, v2); **MCD** ? resultado(r) ...

CSP. Comunicación Guardada

Limitaciones de ? y ! ya que son bloqueantes.

Problema si un proceso quiere comunicarse con otros (quizás por \neq ports) sin conocer el orden en que los otros quieren hacerlo con él.

Por ejemplo, el proceso Copiar podría extenderse para hacer *buffering* de k caracteres: si hay más de 1 pero menos de k caracteres en el buffer, Copiar podría recibir otro carácter o sacar 1.

Las sentencias de comunicación guardada soportan comunicación no determinística:

B; C \rightarrow S;

B puede omitirse y se asume true.

B y C forman la **guarda**.

La guarda *tiene éxito* si B es true y ejecutar C no causa demora.

La guarda *falla* si B es falsa. La guarda se *bloquea* si B es true pero C no puede ejecutarse inmediatamente.

CSP. Comunicación Guardada

Las sentencias de comunicación guardadas aparecen en **if** y **do**.

```
if B1; comunicación1 → S1;  
  □ B2; comunicación2 → S2;  
fi
```

Ejecución:

Primero, se evalúan las expresiones booleanas

- Si ambas guardas fallan, el **if** termina sin efecto
- Si al menos una guarda tiene éxito, se elige una (no determinísticamente)
- Si ambas guardas se bloquean, se espera hasta que una tenga éxito

Segundo, luego de elegir una guarda exitosa, se ejecuta la sentencia de comunicación en la guarda elegida.

Tercero, se ejecuta la sentencia S_i

La ejecución del **do** es similar (se repite hasta que todas las guardas fallen)

CSP. Comunicación Guardada

Podemos re-programar Copiar p/ usar comunicación guardada:

```
Process Copiar {  
    CHAR c;  
    do Oeste ? c → Este ! c;  od  
}
```

```
Process Copiar {  
    CHAR c;  
    do true →  
        Oeste ? c;  
        Este ! c ;  
    od  
}
```

Extendemos *Copiar* para manejar un buffer de tamaño 2. Luego de ingresar un carácter, el proceso que copia puede estar recibiendo un segundo carácter de Oeste o enviando uno a Este.

```
Process Copiar2 {  
    CHAR c1, c2;  
    Oeste ? c1;  
    do Oeste ? c2 → Este ! c1 ; c1=c2;  
    □ Este ! c1 → Oeste? c1;  
    od
```

CSP. *Copiar* con un buffer limitado

```
Process Copiar {  
    CHAR buffer[80];  
    INT front = 0, rear = 0, cantidad = 0;  
    do cantidad < 80; Oeste ? buffer[rear] →  
        cantidad = cantidad + 1;  
        rear = (rear + 1) MOD 80;  
    □ cantidad > 0; Este ! buffer[front] →  
        cantidad := cantidad - 1;  
        front := (front + 1) MOD 80;  
    od  
}
```

- a) con PMA, procesos como *Oeste* e *Este* ejecutan a su propia velocidad pues hay buffering implícito.
- b) con PMS, es necesario programar un proceso adicional para implementar buffering si es necesario.

CSP. *Asignación de recursos*

```
Process Alocador {  
  INT disponible = MaxUnidades;  
  SET unidades = valores iniciales;  
  INT indice, idUnidad;  
  do disponible > 0; cliente[*] ? acquire(indice) →  
    disponible = disponible - 1;  
    remove (unidades, idUnidad);  
    cliente[indice] ! reply(idUnidad);  
  □ cliente[*] ? release(indice, idUnidad) →  
    disponible = disponible + 1;  
    insert (unidades, idUnidad);  
  od  
}
```

La solución es concisa. Usa múltiples ports y un brazo del *do* para atender c/u. Se demora en un mensaje *acquire* hasta que haya unidades, y no es necesario salvar los pedidos pendientes.

CSP. *Intercambio de valores entre dos procesos*

```
Process P1 {  
  INT valor1 = 1, valor2;  
  if P2 ! valor1 → P2 ? valor2;  
  □ P2 ? valor2 → P2 ! valor1;  
  fi  
}  
Process P2 {  
  INT valor1 , valor2 = 2;  
  if P1 ! valor2 → P1 ? valor1;  
  □ P1 ? valor1 → P1 ! valor2;  
  fi  
}
```

Esta solución simétrica NO tiene deadlock porque el no determinismo en ambos procesos hace que se acoplen las comunicaciones correctamente.

Si bien es simétrica, es más compleja que la de PMA...

CSP. Generación de números primos: *La Criba de Eratóstenes*

Problema: generar todos los primos entre 2 y n

2 3 4 5 6 7 8 ... n

Comenzando con el primer número (2), recorremos la lista y borramos los múltiplos de ese número. Si n es impar:

2 3 5 7 ... n

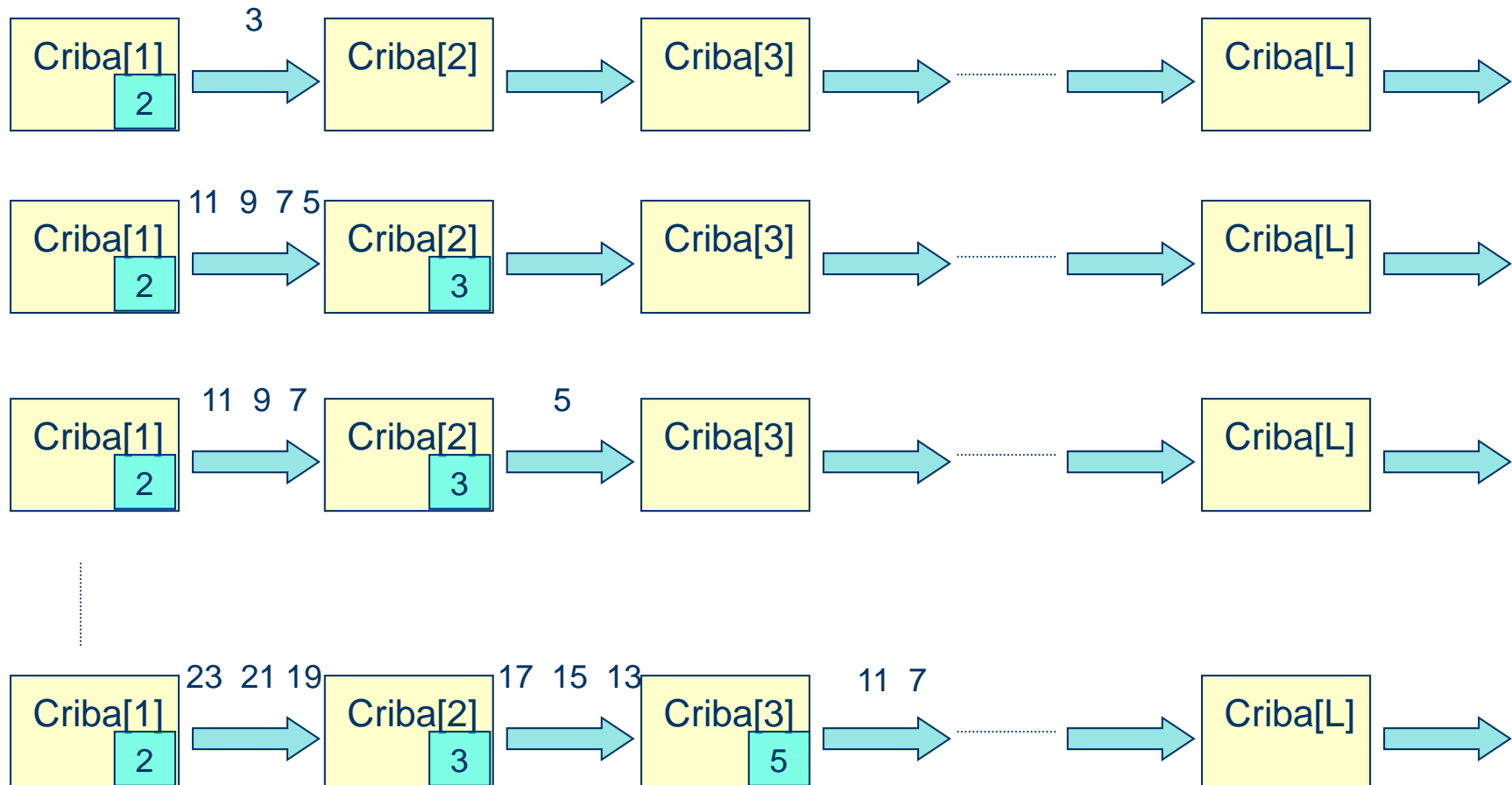
Pasamos al próximo número, 3, y borramos sus múltiplos. Siguiendo hasta que todo número fue considerado, los que quedan son todos los primos entre 2 y n .

La criba *captura* primos y *deja caer* múltiplos de los primos.

Cómo paralelizar??

- un proceso por cada número (mala solución...)
- pipe de procesos filtro: cada uno recibe un stream de números de su predecesor y envía un stream a su sucesor. El primer número que recibe es el próximo primo, y pasa los *no múltiplos*

CSP. Generación de números primos: *La Criba de Eratóstenes*



CSP. Generación de números primos: *La Criba de Eratóstenes*

```
Process Criba[1] {  
    INT p=2;  
    for [i = 3 to n by 2] Criba[2] ! i # pasa impares a Criba[2]  
}  
Process Criba[i = 2 TO L] {  
    INT p, proximo;  
    Criba[i-1] ? p # p es primo  
    do Criba[i-1] ? proximo → # recibe próximo candidato  
        if (proximo MOD p) <> 0 → # si es primo  
            Criba[i+1] ! proximo; # entonces lo pasa  
        fi  
    od  
}
```

El número total de procesos *Criba* (L) debe ser lo suficientemente grande para garantizar que se generan todos los primos hasta n . Excepto *Criba*[1], los procesos terminan bloqueados esperando un mensaje de su predecesor. Cuando el programa para, los valores de p en los procesos son los primos. Puede modificarse con centinelas.

Ordenación de un arreglo

Problema: ordenar un arreglo de n valores en paralelo (n par, orden no decreciente)

Dos procesos P1 y P2, cada uno inicialmente con $n/2$ valores (arreglos a1 y a2 respectivamente)

Los $n/2$ valores de cada proceso se encuentran ordenados inicialmente.

Idea: realizar una serie de intercambios. En cada uno P1 y P2 intercambian $a1[\text{mayor}]$ y $a2[\text{menor}]$, hasta que $a1[\text{mayor}] < a2[\text{menor}]$

Ordenación de un arreglo

```
Process P1 { INT a1[1:n/2]           # inicializado con n/2 valores
    const mayor := n/2; INT nuevo
    ordenar a1 en orden no decreciente
    P2 ! a1[mayor]; P2 ? nuevo
    # intercambia valores con P2
    do a1[mayor] > nuevo →
        poner nuevo en el lugar correcto en a1, descartando el viejo a1[mayor]
        P2 ! a1[mayor]; P2 ? nuevo
    od
}

Process P2 { INT a2[1:n/2]           # inicializado con n/2 valores
    const menor := 1; INT nuevo
    ordenar a2 en orden no decreciente
    P1 ? nuevo; P1 ! a2[menor];
    # intercambia valores con P1
    do a2[menor] < nuevo →
        poner nuevo en el lugar correcto en a2, descartando el viejo a2[menor]
        P1 ? nuevo; P1 ! a2[menor];
    od
}
```

Ordenación de un arreglo

Notar que en la implementación del intercambio, las sentencias de E/ y S/ son bloqueantes \Rightarrow usamos una solución asimétrica

Para evitar deadlock, *P1* primero ejecuta una S/ y luego una E/, y *P2* ejecuta primero una E/ y luego una S/

Comunicación guardada para programar una solución simétrica:

P1: ... if *P2* ? nuevo \rightarrow *P2* ! *a1*[mayor] \square *P2* ! *a1*[mayor] \rightarrow *P2* ? nuevo fi ...

P2: ... if *P1* ? nuevo \rightarrow *P1* ! *a2*[menor] \square *P1* ! *a2*[menor] \rightarrow *P1* ? nuevo fi ...

Esta solución es más costosa de implementar

Mejor caso \rightarrow los procesos intercambian solo un par de valores.

Peor caso \rightarrow intercambian $n/2 + 1$ valores: $n/2$ para tener cada valor en el proceso correcto y uno para detectar terminación.

Ordenación de un arreglo

Solución con k procesos $P[1:k]$, inicialmente con n/k valores c/u
Cada uno primero ordena sus n/k valores. Luego ordenamos los n elementos usando aplicaciones paralelas repetidas del algoritmo compare-and-exchange

Cada proceso ejecuta una serie de rondas:

En las impares, cada proceso con número impar juega el rol de $P1$, y cada proceso con número par el de $P2$.

En las rondas pares, cada proceso numerado par juega el rol de $P1$, y cada proceso impar el rol de $P2$.

Ej: $k=4$. Algoritmo *odd/even exchange sort*

ronda	P[1]	P[2]	P[3]	P[4]
0	8	7	6	5
1	7	8	5	6
2	7	5	8	6
3	5	7	6	8
4	5	6	7	8

Ordenación de un arreglo

Cada intercambio progresa hacia una lista totalmente ordenada.

Cómo pueden detectar los procesos si toda la lista está ordenada?

Un proceso individual no puede detectar que la lista entera está ordenada después de una ronda pues conoce solo dos porciones

- Se puede usar un coordinador separado. Después de cada ronda, los procesos le dicen a éste si hicieron algún cambio a su porción.

2k mensajes de overhead en cada ronda.

- Que cada proceso ejecute suficientes rondas para garantizar que la lista estará ordenada (en general, al menos k rondas).

En el k -proceso, cada uno intercambia hasta $n/k+1$ msg por ronda.
El algoritmo requiere hasta $k^2(n/k + 1)$ intercambio de mensajes.

El lenguaje OCCAM

Hoare introdujo CSP como lenguaje formal que sigue el modelo de PMS, pero nunca fue implementado en forma completa.

OCCAM es un lenguaje real, que implementa lo esencial de CSP sobre la arquitectura “modelo” de los transputers.

Transputers + OCCAM = “sistema multiprocesador p/ procesamiento concurrente”, donde tanto la arquitectura como el lenguaje son simples y adecuadas a la programación concurrente PMS.

OCCAM es un lenguaje simple con una sintaxis rígida

Los procesos y los caminos de comunicación e/ ellos son estáticos (cantidades fijas definidas en compilación).

Modelo de comunicación sincrónico por canales half duplex.

El lenguaje OCCAM

Las sentencias básicas (asignación y comunicación) son vistas como procesos primitivos.

No soporta recursión, ni creación o nombrado dinámico \Rightarrow algunos algoritmos son difíciles de programar, aunque el compilador puede determinar cuántos procesos tiene un programa y cómo se comunican. Permite mapear procesos a procesadores del transputer

Unidades básicas de programa \Rightarrow declaraciones y 3 "procesos" primitivos: *asignación*, *input (receive)*, *output (sync_send)*

Canales declarados globales a los procesos, pero cada canal debe tener exactamente un emisor y un receptor.

Los procesos primitivos se combinan en procesos convencionales usando "*constructores*": secuencial (SEQ), paralelo (PAR, similar a la sentencia co), y sentencia de comunicación guardada.

El lenguaje OCCAM

En la mayoría de los lenguajes, el default es ejecutar sentencias secuencialmente; el programador tiene que decir explícitamente cuándo ejecutar sentencias concurrentemente.

Occam toma un enfoque distinto: no hay default.

```
INT x, y :  
SEQ  
  x := x + 1  
  y := y + 1
```

Dado que las dos sentencias acceden variables distintas, pueden ser ejecutadas concurrentemente. Esto se expresa por:

```
INT x,y :  
PAR  
  x := x + 1  
  y := y + 1
```

Los procesos especificados con PAR ejecutan con igual prioridad y sobre cualquier procesador.

El lenguaje OCCAM. Constructores

Otros constructores:

- **PRI PAR** para especificar que el primer proceso tiene la prioridad más alta, el segundo la siguiente, etc.
- **PLACED PAR** para especificar explícitamente dónde es ubicado y ejecutado cada proceso
- **IF:** para alternativas. Similar a la sentencia **IF** guardada, pero las guardas se evalúan en el orden en que están listadas, y al menos una debe ser TRUE.
- **CASE:** variante de **IF** para alternativas múltiples.
- **WHILE:** uno de los mecanismos de iteración; simil while de Pascal.
- **ALT** soporta comunicación guardada. C/ guarda consta de un input process, una expresión booleana y un input process, o una expresión booleana y un SKIP. Guardas evaluadas en orden no determinístico.
- **PRI ALT** puede usarse para forzar que las guardas sean evaluadas en el orden deseado.

El lenguaje OCCAM .

Replicadores y canales

Replicador (similar a un cuantificador). Por ej., lo siguiente declara un arreglo **pepe** de 21 elementos e iterativamente asigna un valor a cada elemento:

```
[20] INT pepe :  
SEQ i = 0 FOR 10  
  pepe[i] := i
```

Las sentencias en distintos constructores no pueden compartir variables. Para comunicarse y sincronizar, deben usar canales:

CHAN OF protocol *name* :

Los canales se acceden por procesos primitivos input (?) y output (!). A lo sumo un proceso puede emitir por un canal, y a lo sumo uno puede recibir por un canal.

Nombrado estático \Rightarrow el compilador puede forzar este requerimiento.

El lenguaje OCCAM .

Comunicación - Ejemplo

keyboard y *screen* canales que se asumen conectados a periféricos.

(Occam no define mecanismos de E/S como parte del lenguaje, pero provee mecanismos para ligar canales de E/S a dispositivos).

CHAN OF BYTE comm :

PAR

WHILE TRUE

BYTE ch :

SEQ

keyboard ? ch

comm ! ch

WHILE TRUE

BYTE ch :

SEQ

comm ? ch

screen! ch

El lenguaje OCCAM .

El constructor ALT

Puede usarse un replicador como abreviatura (por ej. para esperar recibir entrada de uno de un arreglo de canales).

Ej: alocador de recursos simple:

```
ALT i = 0 FOR n
  avail > 0 & acquire[i] ? unitid
  SEQ
    avail := avail - 1  -- and select unit to allocate
    reply[i] ! Unitid
  release[i] ? unitid
  avail := avail + 1  -- and return unit
```

Acquire, reply y release arreglos de canales (un elemento x cliente)
Son arreglos pues un canal puede ser usado por sólo 2 procesos.

No se permiten output processes en guardas de un ALT \Rightarrow algunos algoritmos duros de programar, pero simplifica la implementación.

El lenguaje OCCAM . Relojes.

Occam también contiene una facilidad de **timer**.

Un timer es una clase especial de canal.

Un proceso hardware está siempre listo para enviar hacia algún canal timer declarado en un programa.

Si un proceso declara el canal timer *myclock*, puede leer la hora ejecutando:

myclock ? time

El proceso también puede demorarse hasta que el reloj alcance un cierto valor ejecutando:

myclock ? time AFTER value

Aquí, *value* es un tiempo absoluto, no un intervalo.

Un timer también puede usarse en una guarda en un constructor ALT; esto sirve para programar un interval timer.