Programación Concurrente ATIC
Redictado de Programación Concurrente

Clase 7

Facultad de Informática

UNLP

Links al archivo con audio

La teoría con los audios está en formato MP4. Debe descargar los archivos comprimidos de los siguientes links:

- Pasaje de Mensajes Sincrónicos (PMS): https://drive.google.com/uc?id=1xJS1-HKs6FMWRfH6tprV2vZ0Nu-2Hj0R&export=download

- CSP – Lenguaje para PMS: https://drive.google.com/u/1/uc?id=1TgzNTMds7QPzHpLaYY4LjzM9lOVeB9Xx&export=download

Pasaje de Mensajes Sincrónicos (PMS)

Diferencia con PMA

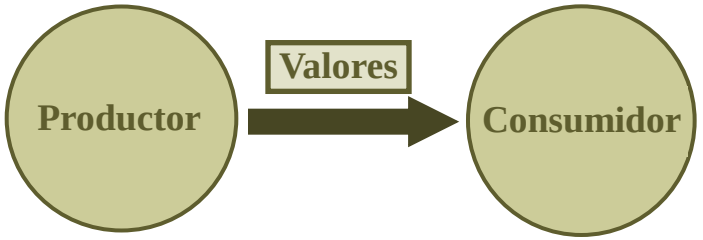
- Los canales son de tipo ***link*** o punto a punto (1 emisor y 1 receptor).
- La diferencia entre *PMA* y *PMS* es la primitiva de trasmisión ***Send***. En *PMS* es bloqueante y la llamaremos (por ahora) ***sync_send***.
  - El trasmisor queda esperando que el mensaje sea recibido por el receptor.
  - La cola de mensajes asociada con un *send* sobre un canal se reduce a 1 mensaje $\Rightarrow$ **menos** memoria.
  - Naturalmente el grado de concurrencia se reduce respecto de la sincronización por PMA *(los emisores se bloquean)*.
- Si bien ***send*** y ***sync_send*** son similares (en algunos casos intercambiables) la semántica es diferente y las posibilidades de ***deadlock*** son mayores en comunicación sincrónica.

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
      }
}
```



Comentarios de PMS

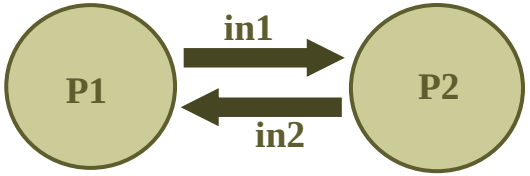
- Si los cálculos del productor se realizan mucho más rápido que los del consumidor en las primeras ***n/2*** operaciones, y luego se realizan mucho más lento durante otras ***n/2*** interacciones:
  - Con PMS los pares send/receive se completarán asumiendo la demora del proceso que más tiempo consuma. Si la relación de tiempo fuera 10 a 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 PMA***. Para lograr el mismo efecto en PMS se debe interponer un proceso ***"buffer"***.
- ¿Que pasa si existe más de un productor/consumidor?.

Comentarios de PMS

- La concurrencia también se reduce en algunas interacciones Cliente/Servidor.
  - 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*** (sync_send) y ***receive*** hagan matching.

Deadlock en PMS

- Dos procesos que intercambian valores.




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

Process P1
{    int valorA = 1, valorB;
     sync_send in2(valorA);
     receive in1(valorB);
}

Process P2
{    int valorA, valorB = 2;
     sync_send in1(valorB);
     receive in2 (valorA);
}
```

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

Process P1
{    int valorA = 1, valorB;
     sync_send in2(valorA);
     receive in1(valorB);
}

Process P2
{    int valorA, valorB = 2;
     receive in2 (valorA);
     sync_send in1(valorB);
}
```

CSP– Lenguaje para PMS

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, SR) 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.

  process A { . . . B ! e; . . . } **process B { . . . A ? x; . . . }**

- Para que se produzca la comunicación, deben ***matchear***, y luego ***se ejecutan simultáneamente***.
- Efecto: sentencia de ***asignación distribuida***.

El lenguaje CSP (Hoare, 1978)

- Formas generales de las sentencias de comunicación:

Destino ! port(e_1, ..., e_n);

Fuente ? port(x_1, ..., 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*** son etiquetas que se usan para 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);**

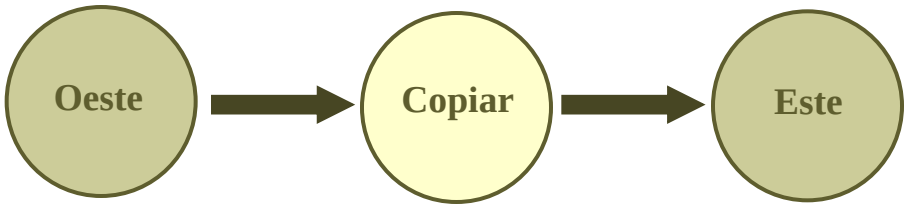
Ejemplos básicos

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

```
Process Oeste
{   char c;
    do true →
            Generar(c);
            Copiar ! (c);
    od
}
```

```
Process Copiar
{   char c;
    do true →
            Oeste ? (c) ;
            Este ! (c);
    od
}
```

```
Process Este
{   char c;
    do true →
            Copiar ? (c) ;
            Usar(c);
    od
}
```



Ejemplos básicos

- Server que calcula el MCD de dos enteros con el 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);
            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);
...
```

Comunicación Guardada

- Limitaciones de **?** y **!** ya que son bloqueantes. Hay problema si un proceso quiere comunicarse con otros (quizás por ≠ 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 operaciones de comunicación (**?** y **!**) pueden ser ***guardadas***, es decir hacer un AWAIT hasta que una condición sea verdadera.
- El **do** e **if** de CSP usan los *comandos guardados* de Dijkstra (B → S).

Comunicación Guardada

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.

Comunicación Guardada

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

if B_1***; comunicación***$_1 \rightarrow S_1$***;***
- B_2***; comunicación***$_2 \rightarrow S_2$***;***

fi

Ejecución:

- *Primero*, se evalúan las guardas.
  - Si todas las guardas fallan, el if termina sin efecto.
  - Si al menos una guarda tiene éxito, se elige una de ellas (no determinísticamente).
  - Si algunas guardas se bloquean, se espera hasta que alguna de ellas tenga éxito.
- *Segundo*, luego de elegir una guarda exitosa, se ejecuta la sentencia de comunicación de la guarda elegida.
- *Tercero*, se ejecuta la sentencia S_i.

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

Comunicación Guardada

Podemos re-programar Copiar para usar comunicación guardada:

```
Process Copiar
    {   char c;
        do 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
    }
```

Ejemplo

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
}
```

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

Ejemplo

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 cada una. Se demora en un mensaje ***acquire*** hasta que haya unidades, y no es necesario salvar los pedidos pendientes.

Ejemplo

Intercambio de Valores

```
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...

Ejemplo

*La Criba de Eratóstenes para la g*eneración de números primos

- **Problema:** generar todos los primos entre 2 y $n \rightarrow 2\ 3\ 4\ 5\ 6\ 7\ 8\ \ldots\ N$
- Comenzando con el primer número (2), recorremos la lista y borramos los múltiplos de ese número. Si *n* es impar: $2\ 3\ 5\ 7\ \ldots\ n$
- 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?*** 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*.

Ejemplo

*La Criba de Eratóstenes para la g*eneración de números primos

```
Process Criba[1]
{    int p = 2;

     for [i = 3 to n by 2]  Criba[2] ! (i);
}

Process Criba[i = 2 to L]
{    int p, proximo;

     Criba[i-1] ? (p);
     do Criba[i-1] ? (proximo) →
                  if  ((proximo MOD p) <> 0 ) and (i < L) →  Criba[i+1] ! (proximo);
     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.

Ejemplo

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[mayor]*** y ***a2[menor]***, hasta que *a1[mayor] < a2[menor]*.

Ejemplo

Ordenación de un Arreglo

```
Process P1
{  int nuevo, a1[1:n/2]; const mayor = n/2;
   ordenar a1 en orden no decreciente
   P2 ! (a1[mayor]);
   P2 ? (nuevo)
   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 nuevo, a2[1:n/2]; const menor = 1;
   ordenar a2 en orden no decreciente
   P1 ? (nuevo);
   P1 ! (a2[menor]);
   do  a2[menor] < nuevo →
       poner nuevo en el lugar correcto en a2, descartando el viejo a2[menor]
       P1 ? (nuevo);
       P1 ? (a2[menor]);
    od
}
```

Ejemplo

Ordenación de un Arreglo

- Notar que en la implementación del intercambio, las sentencias de entrada y salida son bloqueantes → usamos una solución asimétrica.

- Para evitar *deadlock*, *P1* primero ejecuta una salida y luego una entrada, y *P2* ejecuta primero una entrada y luego una salida.

- ***Comunicación guardada para programar una solución simétrica:*** esta solución es más costosa de implementar.

```
P1: … if P2 ? (nuevo) → P2 ! (a1[mayor])
      □ P2 ! (a1[mayor]) → P2 ? (nuevo)
    fi…

P2: … if P1 ? (nuevo) → P1 ! (a2[menor])
      □ P1 ! (a2[menor]) → P1 ? (nuevo)
    fi …
```

- Mejor caso → los procesos intercambian solo un par de valores.

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

Ejemplo

Ordenación de un Arreglo

- Solución con ***b*** procesos ***P[1:b]***, inicialmente con ***n/b*** valores cada uno.
- Cada uno primero ordena sus ***n/b*** 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***.
- **Ejemplo:** *b=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

Ejemplo

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.
    - ✓ ***2b*** mensajes de overhead en cada ronda.
  - Que cada proceso ejecute suficientes rondas para garantizar que la lista estará ordenada (en general, al menos ***b*** rondas).
    - ✓ Cada proceso intercambia hasta ***n/b+1*** mensajes por ronda.
    - ✓ El algoritmo requiere hasta $b^2*(n/b + 1)$ intercambio de mensajes.