III
Programación Concurrente
Clases 7 a 9: Memoria distribuida

Ramiro Martínez D'Elía

2021

Índice general

Capítulo 1

Programa distribuido

1.1. Definición

Los mecanismos de sincronización vistos hasta ahora, están pensados para programas concurrentes que ejecutan en hardware con procesadores que comparten memoria (arquitecturas de memoria compartida).

Sin embargo, hay arquitecturas donde los CPU solo comparten una red de comunicación. Bajo este contexto, los procesos ya no comparten variables sino que comparten ***canales***.

Los ***canales*** son abstracciones de una red de comunicación física y al ser el único recurso compartido entre procesos, toda variable es local a un solo proceso y solo accedida por ese proceso. Esto descarta la necesidad de exclusión mutua.

La ausencia de variables compartidas, permite que los procesos puedan ejecutarse en procesadores distribuidos. Por esta razón, los programas concurrentes que utilizan pasaje de mensajes son llamados ***programas distribuidos***.

1.2. Canales

Los canales son ***estructuras FIFO*** (First In First Out) que contienen mensajes pendientes. Estas estructuras soportan 2 (dos) **operaciones atómicas *send* y *receive***. Para iniciar una comunicación, un proceso envía un mensaje por el canal; y otro lo adquiere recibiendolo desde el mismo canal.

La operación ***send*** puede ser **asincrónica** (no demora al proceso que la invoca) y **sincrónica** (demora al proceso que la invoca, hasta que su mensaje fuese recibido). La operación ***receive*** **siempre demora** al proceso que la invoca.

La estructura FIFO y la atomicidad de las primitivas send y receive aseguran que; los mensajes eventualmente serán recibidos, los mensajes no serán corrompidos y serán entregados en el orden en que fueron encolados.

```
1  channel ch1(string),
2          ch2(string);
3
4  Program Uno
5      string respuesta;
6      send ch2("Hola");
7      receive ch1(respuesta);
8  End.
9
10 Program Dos
11     string saludo;
12     receive ch2(saludo);
13     send ch1("Aca esta tu respuesta ...);
14 End.
```

Según la forma en que se utilicen, los canales, pueden clasificarse en los siguientes tipos:

- **Mailbox**: Cualquier proceso puede enviar y recibir datos, por alguno de los canales declarados. Relación *n:n*.
- **Input Port**: El canal tiene un único receptor y *(*n) emisores. Relación *1:n*.

- **Link**: El canal tiene un único receptor y un único emisor. Relación *1:1*.

Capítulo 2

Patrones de resolución

Los tipos de procesos que podemos encontrar en los programas distribuidos, son los mismos que en un programa concurrente con memoria compartida (peers, filters, clientes y servidores). Lo mismo ocurre, con los patrones de resolución.

A continuación, abordaremos un patrón en particular: ***interacting peers***, donde se describirá la forma de interconexión entre procesos. Todos los ejemplos, implementan ***pasaje de mensajes asincrónicos (PMA)***.

> 💎 Soluciones para ordenar números: Merged Network, Odd/Even Exchange Sort, Pipeline. ¿Cantidad de procesos? ¿Cantidad de mensajes? (Ver apunte de gemas).

2.1. Iteracting peers: intercambiando valores

Supongamos el problema donde existen n procesos, cada uno con un valor local v. El objetivo es, lograr que todos los procesos conozcan el máximo y mínimo valor de v de entre todos.

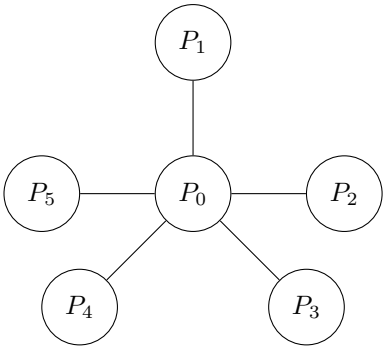
En este caso, los procesos son identicos. Por lo cual, podemos diseñar una solución basada en el esquema de pares que interactúan.

A contuniación se detallan soluciones, empleando 3 (tres) formas distintas de conexión entre procesos: ***centralizada***, ***simétrica*** y ***circular***.

2.1.1. Solución centralizada

En esta solución, utilizaremos un proceso central que; recibirá los valores de los demás. Computará los máximos y mínimos e informará al resto.

```
1  channel values(int),
2          results[n](int, int);
3
4  Process Peer[i = 1 to n-1]
5      int v, min, max;
6      send values(v);
7      receive results[i](min, max);
8  End.
9
10 Process CentralPeer[0]
11     int v, min, max, temp;
12
13     for(i=1 to n-1)
14         receive values(temp);
15         # Compara y actualiza ...
16     end;
17
18     for(i=1 to n-1)
19         send results[i](min, max);
20 End.
```

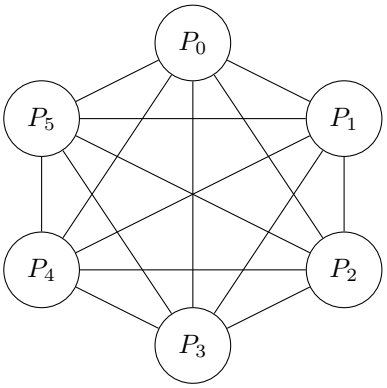


2.1.2. Solución simétrica

En esta solución, todos los procesos realizan el mismo algoritmo. Cada par de procesos, se encuentran conectados mediante un canal.

Cada proceso, envía los datos al resto, y así cada uno puede conocer todos los valores y efecturar el cálculo de máximos y mínimos en paralelo.

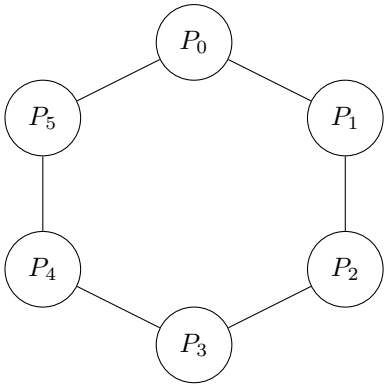
```
1  channel values[n](int, int);
2
3  Process Peer[i = 0 to n]
4      int v, min, max, temp;
5
6      for(j=0 to n st j != i)
7          send values[j](v);
8
9      for(j=0 to n st j != i)
10         receive receive[i](temp)
11         # Compara y actualiza.
12 End.
```



2.1.3. Solución circular

En esta solución, cada proceso recibe mensajes de su antecesor y envía mensajes a su sucesor. Con este enfoque, harán falta 2 (dos) iteraciones al anillo. La primera, para obtener el máximo y el mínimo. Y la segunda, para notificar a todos los procesos.

```
1  channel values[n](int, int);
2
3  Process Initial[0]
4      int v, min = v, max = v;
5
6      # Inicia primera iteracion ...
7      send values[1](min, max);
8      receive values[0](min, max);
9
10     # Inicia segunda iteracion ...
11     send values[1](min, max);
12 End.
13
14 Process Peer[i = 1 to n-1]
15     int v, min, max;
16
17     # Primera iteracion ...
18     receive values[i](min, max);
19     # Compara y actualiza valores ...
20     send values[(i+1) mod n](min, max);
21
22     # Segunda iteracion
23     receive values[i](min, max);
24     send values[(i+1) mod n](min, max);
25 End.
```



2.1.4. Conclusiones

Desde el punto de vista de la cantida de mensajes enviados y la eficiencia, podemos obtener las siguientes conclusiones para las soluciones descriptas.

Solución	**# Mensajes**	**Orden**	**Conclusión**
Centralizada	$2(n-1)$	Lineal $O(n)$	No hay paralelismo, toda la carga del cómputo recae en CentralPeer. Menor overhead en el pasaje de mensajes. Menor eficiencia a mayor n.
Simétrica	$n(n-1)$	Cuadrático $O(n^2)$	Cómputo paralelo pero, mayor overhead en el pasaje de mensjaes.
Circular	$2(n-1)$	Lineal $O(n)$	Cómputo paralelo. Mayor eficiencia para n muy grandes.

Nota: Con la primitiva ***broadcast***, la cantidad de mensajes se reduce a n para todos los casos. Lo que hace, dicha primitiva es envíar un mensaje (de forma concurrente) a todos los procesos que escuchen el canal utilizado.

Capítulo 3

PMS - Pasaje de Mensajes Asincrónico

3.1. CSP y Comunicación guardada

Con CSP (***Communicating Sequential Processes***) se introduce el concepto de ***PMS y comunicación guardada***. Los procesos, se enviarán mensajes. entre ellos, mediante links directos. La forma general de comunicación, entre procesos, en CSP es de la siguiente forma:

```
1  DestinationProc!portName(...);
2  SourceProc?portName(...);
```

Siendo los operadores ***!*** y ***?*** para el envío y recepción de mensajes respectivamente. Dos procesos, entablabran comunicación cuando ejecuten sentencias de comunicación que hagan match entre sí.

El siguiente ejemplo, muestra un proceso (Copy) que copia caracteres entre 2 (dos) procesos East y West.

```
1  Process Copy
2
3  Char c;
4
5  while (true)
6      west?(c)
7      east!(c);
8  end.
9
10 End.
```

La principal limitación de ***!*** y ***?*** es que, son primitivas bloqueantes. Usualmente, se desea, que un proceso pueda comunicarse con los demás; sin importar el orden en que los otros quieran comunicarse con el.

Por ejemplo, si modificamos el ejemplo de Copy de forma tal que; ahora se deben copiar de a 2 caracteres. El proceso Copy, entonces, debe esperar a recibir los 2 caracteres desde West, antes de retransmitirlos. Esto provoca que, East quede bloqueado.

```
1  Process Copy
2
3  Char c1, c2;
4
5  while (true)
6      west?(c1); west?(c2);
7      east!(c1); east!(c1);
8  end.
9
10 End.
```

La comunicación guardada, ofrece un tipo de ***comunicación no determinística*** que resuelve la limitación planteada. ***Combinando sentencias condicionales y de comunicación*** de la siguiente forma $B; C \rightarrow S$, donde:

- ***B*** es una sentencia condicional, que de no existir se asume *verdadera*.
- ***C*** es una sentencia de comunicación.
- ***D*** es un conjunto de sentencias a ejecutar.
- Juntas ***B*** y ***C*** forman una ***guarda*** que "protegen"la ejecución de ***S***

Con respecto al funcionamiento de las ***guardas***, se debe tener en cuenta lo siguiente:

- La guarda tiene ***éxito*** si, ***B*** es verdadera y ***C*** no causa demora (tiene mensajes → ejecución inmediata).
- La guarda ***falla*** si, ***B*** es falsa.
- La guarda se ***bloquea*** si, ***B*** es verdadera y ***C*** causa demora.

Podemos aplicar comunicación guardada sobre 2 (dos) tipos de estructuras del control: el ***if*** y el ***do***. Cada una, con la siguiente semántica de ejecución:

- Si ***todas las guardas fallan*** (en simultáneo); la estructura de control finaliza sin efectos.
- Si al menos una guarda tiene éxito; se elige una de manera no deterministica, se ejecuta su ***B*** y luego su ***C***. En el caso del ***if***, la estructura finaliza.
- Si todas las ***guardas están bloqueadas***; se espera hasta que alguna tenga éxito.

La versión del proceso Copy, para 2 caracteres, se puede mejorar utilizando comunicación guardada del siguiente modo:

```
1  Process Copy
2
3  char c1, c2;
4  west?(c1);
5
6  do  true; west?(c2) -> east!(c1); c1=c2;
7  []  true; east!(c1) -> west?(c1);
8  end;
9
10 End.
```

Aplicando comunicación guardada, el proceso *Copy*, una vez recibido el primer caracter podrá (de forma no determinística) esperar al segundo caracter o enviar el primero a *East*. De esta forma, logramos reducir el bloqueo en la comunicación entre *Copy* y *East*.

Capítulo 4

RPC, Rendezvous y Notaciones primitivas

4.1. Introducción

La técnica de pasaje de mensajes, resulta óptima para resolver problemas del tipo productores/consumidores y pares que interactúan. Ya que, estos plantean un tipo de comunicación unidireccional entre procesos.

No obstante, el uso de esta técnica para problemas del tipo cliente/servidor; no resulta del todo óptima. Ese tipo de problemas supone una comunicación bidireccional. Es decir, un cliente realiza una petición y, posteriormente, un servidor le otorga una respuesta. Esto nos obliga a definir un gran número de canales:

- Al menos 1 (un) canal por el cual, el servidor reciba peticiones.
- Un canal de respuesta para cada cliente.

Los mecanismos de ***RPC*** y ***Rendezvous***, resuelven de forma más adecuada los problemas del tipo cliente/servidor. Combinando PMS con aspectos de monitores.

Como en ***monitores***, un módulo (o proceso) expone operaciones públicas. Dichas operaciones, son invocadas por otro módulo (o proceso) utilizando la sentencia ***call***.

Como en ***PMS***, la eejcución de una sentencia ***call*** demora al llamador hasta obtener una respuesta.

Las ***operaciones***, resultan ***canales de comunicación bidireccional***. Desde el llamador hacia el proceso que sirve la llamada, y luego, de vuelta al llamador.

4.2. Remote Procedure Calls (RPC)

Los programas se descomponen en módulos (con procesos y procedimientos). Los procesos, de un módulo, son llamados ***background*** para diferenciarlos de los procesos exportados (***servers***). El siguiente ejemplo, muestra de forma trivial la especificación de un módulo.

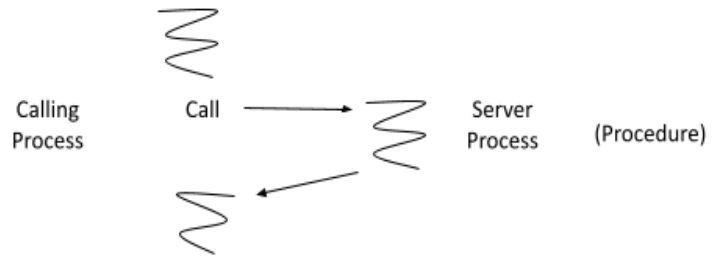
```
1  Module module_name
2      op opname(params)[returns type];        # (1) Interfaz publica
3  Body
4      int x;                                  # (2) Definicion de variables locales
5      x = 0;                                  # e inicializacion
6
7      proc opname(params)[returns type]       # (3) Definicion de procedimientos exportados
8      begin
9          # Cuerpo del procedimiento.
10     end;
11
12     proc local_proc(params)[returns type]   # (4) Definicion de procedimientos locales
```

```
13      begin
14          # Cuerpo del procedimiento.
15      end;
16
17      Process Main                            # (5) Definicion de procesos locales
18          # Cuerpo del proceso bakground
19      end;
20  End module_name;
```

Los procesos background tienen acceso a las variables y pueden invocar procedimientos del mismo módulo. A su vez, como sucede en monitores, pueden invocar procedimientos definidos en otros módulos. Siempre y cuando, estos últimos, sean exportados por dicho módulo.

La ejecución de una llama intermódulo, difiere con respecto a una local.

- Un nuevo proceso atiende la llamada y los argumentos son pasados como mensajes entre el llamador y el proceso servidor.
- El proceso llamador que demorado mientras se ejecuta el cuerpo del procedimiento invocado.
- Cuando el proceso servidor finaliza su ejecución del procedimiento invocado, retorna los resultados y finaliza.
- El proceso llamador, continua su ejecución.



4.2.1. Sincronización

Con RPC, la ***sincronización entre proceso llamador y servidor está implícita***. No obstante, necesitamos mecanismos para que procesos server y background sincronicen dentro de un módulo. Existen 2 (dos) enfoques para proveer sincronización dentro de un módulo.

A lo sumo, un proceso activo	***Ejecución concurrente***
Apropidado para entornos monoprocesador ya que, reduce el context switch.	Apropidado, de forma natural, para entornos multriprocesador.
▪ Mutex → Implícito.	▪ Mutex → Explícito (semáforos o monitores).
▪ Condición → Explícito (semáforos o monitores).	▪ Condición → Explícito (semáforos o monitores).

4.2.2. Time Server

El siguiente módulo es una adaptación del ejemplo del libro.

```
1  module TimeServer
2
3  op get_time(): int;
4  op delay(int interval);
5
6  body
7
8  int time = 0;                  # Tiempo inicial del TimeServer (en ms).
```

```
9  Queue delayed(waketime, id);    # Cola de procesos dormidos, por la operacion delay.
10 Sem m = 1;                      # Semaforo para garantizar mutex, en el acceso a delayed.
11 Sem d = ([n] 0);                # Arreglo de semaforos para demorar procesos.
12
13 proc get_time(): int            # Retorna el tiempo actual del servidor en ms.
14     return time;
15 end;
16
17 proc delay(int interval)        # Demora un proceso tantos ms, como interval lo indique.
18     P(m);
19     waketime = time + interval;
20     delayed.push(waketime, id);
21     V(m);
22     P(d[id]);
23 end;
24
25 process Clock                   # Proceso background. Mantiene actualizada la variable time
26     while (true)                # y despierta procesos demorados por la operacion delay.
27         time++;
28         P(m);
29         while (time >= menor waketime en delayed)
30             delayed.pop(waketime, id);
31             V(d[id]);
32         end;
33         V(m);
34     end;
35 end;
36 End.
```