Universidad de Huelva

Universidad de Huelva

Grado en Ingeniería Informática

Tema 5. Soluciones basadas en el Paso de Mensajes

Resumen

Autor: Alberto Fernández Merchán
Asignatura: Programación Concurrente y Distribuida

1. Introducción

Cuando no existe la posibilidad de utilizar mecanismos de memoria compartida se necesitan otras herramientas de control de concurrencia, como el paso de mensajes (sistemas distribuidos).

Las primitivas básicas son *send* y *receive*. Son como una extensión de los semáforos donde hay intercambios de información. También es posible hacerlos en sistemas con memoria compartida.

2. Aspectos de diseño

Para definir el paso de mensajes necesitamos definir los siguientes aspectos:

- Identificación de procesos: Forma en la que el emisor indica a quién va dirigido el mensaje. Existen dos variantes:
  - **Denominación Indirecta**: Mensajes enviados de forma anónima (a un depósito intermedio). Esto es conocido como buzón.
    Se permite la comunicación 1:1, 1:N, N:1 y N:N
  - Denominación Directa: Los mensajes se envían/reciben a procesos concretos. Existen dos tipos:
    - **Denominación Directa Simétrica**: Ambos procesos se conocen y se nombran.
    - **Denominación Directa Asimétrica**: El emisor identifica al receptor pero no al revés.
- **Sincronización**: El proceso receptor cuando requiere un mensaje esperará hasta que llegue. Pero el emisor puede comportarse de forma distinta. Existen dos alternativas:
  - **Asíncrona**: El emisor envía el mensaje y continúa, sin preocuparse de si el mensaje será recibido.
  - **Síncrona**: El emisor espera a que el receptor esté dispuesto para recibir el mensaje. (*rendez-vous*)
- **Características del canal**:
  - **Flujo de Datos**: Según la circulación de la información:
    - **Unidireccional**: Emisor $\rightarrow$ Receptor
    - **Bidireccional**: Emisor $\leftrightarrow$ Receptor
  - **Capacidad del canal**: Cantidad de mensajes pendientes de entregar que puede haber en el canal.
    - **Capacidad cero**: Comunicación síncrona.
    - **Capacidad finita**: Según el tamaño del buffer de almacenamiento.
  - **Tamaño de los mensajes**: Podrán ser fijos o variable.
  - **Canales con tipo o sin tipo**: Algunos canales solo permiten enviar mensajes del tipo dle que se han definido.
  - **Envío de copia o referencia**: Mismas características que el paso de parámetros.
  - **Errores en las comunicaciones**: Detectarlos y operar en consecuencia. Consideraremos que llegan sin errores.

2.1. Espera selectiva

Cuando se emplea un mecanismo de comunicación cliente/servidor, donde un proceso recibe peticiones de varios procesos, el proceso servidor debería ser capaz de atender peticiones a través de varios canales a la vez.

Como la operación receive espera hasta recibir el mensaje, puede ocurrir que llegue a un canal sobre el que no se está haciendo la espera. Para permitirlo, se dispone de la sentencia *select*.

Al llegar a la sentencia select, el proceso selecciona de forma aleatoria alguna de las ramas sobre las que se haya realizado una operación de envío. De no haber ninguna, el proceso se bloquea en espera de que llegue

alguna de ellas.

Una variante más potente es el select con cláusulas *when*. Al llegar a *select*, solo se consideran abiertas aquellas cláusulas con condición cierta. Esta condición solo se evalúa al comenzar la sentencia *select*, no tras realizar el bloqueo.

3. Paso de mensajes asíncrono

El emisor envía el mensaje y continúa sin esperar a que sea leído. Tenemos dos alternativas:

- Cuando no hay mensajes **se espera** la llegada de uno.
- Cuando no hay mensajes se **continúa** (devuelve un valor que representa la ausencia de mensajes).

Es necesaria la existencia de un *buffer*, donde se almacenen los mensajes. (**buzón**).
Vamos a establecer comunicaciones N:N. Necesitaremos poder definir buzones con la siguiente sintaxis:

```
var
  nombre_buzon: mailbox of tipo;
  nombre_buzon: mailbox [1..N] of tipo;
```

La primera definición es un buzón sin límite y la segunda es un buzon con tamaño limitado. En este último, la operación de envío provocará el bloqueo del proceso emisor.

Las colas asociadas al buzón serán *FIFO*.

Las operaciones permitidas en un buzón son:

- **send(buzon,mensaje)**: El emisor deja el mensaje y continúa. El tipo de mensaje debe de ser el mismo que el del buzón. Si le buzón es limitado y está lleno, el proceso espera a poder poner el mensaje cuando haya sitio.

- **receive(buzon,mensaje)**: El receptor saca el primer mensaje de la cola y continúa. Si el buzón está vacío, espera hasta que haya un mensaje.
- **empty(buzon)**: Devuelve un booleano que representa si el buzón está vacío o no.

Todas estas operaciones se consideran atómicas.

3.1. Problemas clásicos con buzones

3.1.1. El problema del productor-consumidor

```
program prodcon;

var
        buzon: mailbox of char;

process producer;
var
        data: char;
begin
        for data := 'a' to 'z' do
        begin
                send(buzon,data);
        end
end;

process consumer;
var
        data: char;
begin
        repeat
                receive(buzon,data);
                writeln(data);
        until data = 'z';
        writeln
end;

begin
        cobegin
                producer;
                consumer
        coend
end.
```

3.1.2. El problema de los lectores-escritores

```
process type Lectores(id:integer);
var
        testigo:integer;
begin
        repeat

                send(abrir_lectura, id);
                receive(buzon[id], testigo)

                (*LECTURA DEL RECURSO*)

                send(cerrar_lectura, testigo);

        forever
end;

process type Escritores(id:integer);
var
        testigo:integer;
begin
        repeat

                send(abrir_escritura,id);
                receive(buzon[id],testigo):

                (*LECTURA DEL RECURSO*)

                send(cerrar_escritura,testigo):

        forever
end;
```

```
process Controlador;
var
        id:integer;
        testigo:integer;
        nl:integer;
        escribiendo:boolean;

begin
        nl:=0;
        escribiendo:=false;
        repeat
                select
                        when empty(abrir_escritura) and escribiendo=false =>
                                receive(abrir_lectura,id);
                                nl:=nl+1;
                                send(buzon[id], testigo);
                or
                        receive(cerrar_lectura,testigo);
                        nl:=nl-1;
                or
                        when (nl=0) and (escribiendo=false) =>
                                receive(abrir_escritura,id);
                                escribiendo:=true;
                                send(buzon[id], testigo);
                or
                        receive(cerrar_escritura,testigo);
                        escribiendo:=false;
                end select;

        forever
end;
```

3.1.3. El problema de los filósofos

```
program Filosofos;

type
        buzon_filosofo: mailbox of integer;

const
        N = 5;

var
        pido_palillos: array[0..N-1] of buzon_filosofo;
        palillos_concedidos: array[0..N-1] of buzon_filosofo;
        suelto_palillos: array[0..N-1] of buzon_filosofo;

process type Filosofo(id:integer);
var
        testigo:integer;
begin
        repeat
                (*PENSANDO*)
                send(pido_palillos[id], testigo);
                receive(palillos_concedidos[id], testigo)

                (*COMIENDO*)

                send(suelto_palillos[id], testigo);

        forever
end;


process Controlador;
var
        palillos: array[0..N-1] of integer;
        testigo:integer;
        i:integer;

begin
        for i:=0 to N-1 do palillos[i]:=1;
        repeat
                select
                        for i:=0 to N-1 replicate (*EQUIVALE A TANTOS OR COMO ITERACIONES TENGA EL BUCLE*)
                                when palillos[cont]=1 and palillos[(cont+1) mod N]=1 =>
                                        receive(pido_palillos[i], testigo);
                                        palillos[i]:=0;
                                        palillos[(i+1) mod N]:=0;
                                        send(palillos_concedidos[i], testigo);
                or
                        for i:=0 to N-1 replicate (*EQUIVALE A TANTOS OR COMO ITERACIONES TENGA EL BUCLE*)
                                receive(suelto_palillos[i],testigo);
                                palillos[i]:=1;
                                palillos[(i+1) mod N]:=1;

                end select;

        forever
end;
```

4. Paso de mensajes síncrono

Tanto send como receive son bloqueantes, es decir, el emisor se asegura de que el receptor ha recibido el mensaje antes de continuar. Al no almacenar mensajes, no es necesario disponer de un buffer de mensajes. El mecanismo que vamos a estudiar son los canales.

Un canal permite enlazar dos procesos. La relación que se establece es de 1:1 entre emisor y receptor, estableciendo un flujo de datos unidireccional. Estos canales suelen tener un tipo que es el de los datos que se envían.

Se definen como:

```
var
  ch: channel of tipo;

ch ! s; //Envía el valor de s al canal ch
ch ? r; //Recibe del canal ch un valor que asigna a r
```

Si se desea usar un canal sin tipo (el valor no es relevante) se declarará de tipo *synchronous*. La variable *any*

queda automáticamente definida:

```
var
  ch: channel of synchronous;

ch ! any; //Envía mensaje de sincronización al canal ch
ch ? any; //Recibe mensaje de sincronización al canal ch
```

La espera selectiva se puede realizar mediante la sentencia *select*:

```
select
  ch[1] ? mensaje[1];
  sentencias;
or
  ch[2] ? mensaje[2];
  sentencias;
or
 . . . .
or
  ch[N] ? mensaje[N];
  sentencias;
end;
```

```
select
  for cont=1 to N replicate
    ch[cont] ? mensaje[cont];
    sentencias;
end;
```

También se pueden añadir cláusulas *when*. En ese caso, la condición solo se evalúa al comienzo de la sentencia select y no vuelve a hacerse hasta que se vuelva a ejecutar dicha sentencia.

Con la sentencia de prioridad, si varias alternativas son posibles se elgirá la primera por orden secuencial.

```
pri select
  ch1 ? mensaje1;
  sentencias;
or
  ch2 ? mensaje2;
  sentencias;


end;
```

4.1. Problemas clásicos con canales

4.1.1. El problema del productor-consumidor

```
program prodcon;

const
        MAX=4; {SON 5 DE 0..4}
        N=5;
        NUMPROD=2;
        NUMCONS=2;
var
        insertar: array[1..NUMPROD] of channel of integer;
        extraer: array[1..NUMCONS] of channel of integer;

process GestorBuffer;
var
        buffer: array[0..MAX] of integer;
        head, tail, numelementos, cont1, cont2: integer;
begin
        head:=0;
        tail:=0;
        numelementos:=0;
        repeat
                select
                        for cont1:=1 to NUMPROD replicate
                                when numelementos < MAX+1 =>
                                        insertar[cont1] ? buffer[tail];
                                        tail:=(tail+1) mod (MAX+1);
                                        numelementos:=numelementos+1;
                or
                        for cont2:=1 to NUMCONS replicate
                                when numelementos > 0 =>
                                        extraer[cont2] ! buffer[head];
                                        head:=(head+1) mod (MAX+1);
                                        numelementos:=numelementos-1;
                or
                        terminate

                end

        forever
end;
```

```
process type Productor(id:integer); {PRODUCTOR}
var
        i:integer;
begin
        for i := 1 to N do
        begin
                insertar[id] ! i;
                writeln(id,' produce ',i);
        end
end;

process type Consumidor(id:integer); {PRODUCTOR}
var
        i,dat:integer;
begin
        for i := 1 to N do
        begin
                extraer[id] ? dat;
                writeln(id,' consume ',dat);
        end
end;

var
        Productores: array[1..NUMPROD] of Productor;
        Consumidores: array[1..NUMCONS] of Consumidor;
        cont:integer;

begin {MAIN}
        cobegin
                for cont:=1 to NUMPROD do Productores[cont](cont);
                for cont:=1 to NUMCONS do Consumidores[cont](cont);
                GestorBuffer;
        coend
end.
```

4.1.2. El problema de los lectores-escritores

```
{Problema de los Lectores/Escritores}
(* Prioridad en la lectura *)
program LectEscr;

const
        NLEC = 5;
        NESC = 2;
        TODOS = 7;
var

        peticion: array[1..TODOS] of channel of char;
        abrir_lectura: array[1..NLEC] of channel of synchronous;
        cerrar_lectura: array[1..NLEC] of channel of synchronous;
        abrir_escritura: array[1..NESC] of channel of synchronous;
        cerrar_escritura: array[1..NESC] of channel of synchronous;

process type Lector(id:integer);
var
        i, nlecturas:integer;
begin
        nlecturas:=10;
        for i:=1 to nlecturas do
        begin
                peticion[id] ! 'L';
                abrir_lectura[id] ! any;

                (*LECTURA DEL RECURSO*)

                cerrar_lectura[id] ! any;

        end
end;

process type Escritor(id:integer);
var
        i, nescrituras:integer;
begin
        nescrituras:=3;
        for i:=1 to nescrituras do
        begin
                peticion[id+NLEC] ! 'E';
                abrir_escritura[id] ! any;

                (*ESCRIBIR EL RECURSO*)

                cerrar_escritura[id] ! any;

        end
end;
```

```
process controlador;
var
        nl, c0, c1, c2, c3, c4:integer;
        escribiendo:boolean;
        tipopeticion: char;
        npeticionesE:integer;

begin
        nl:=0;
        escribiendo:=false;
        npeticionesE:=0;
        repeat
                select
                        for c0:=1 to NLEC+NESC replicate
                                peticion[c0] ? tipopeticion;
                                if tipopeticion='E' then npeticionesE:=npeticionesE+1;
                or
                        for c1:=1 to NLEC replicate
                                when npeticionesE=0 =>
                                abrir_lectura[c1] ? any;
                                nl:=nl+1;
                or
                        for c2:=1 to NLEC replicate
                                cerrar_lectura[c2] ? any;
                                nl:=nl-1;
                or
                        for c3:=1 to NESC replicate
                                when (nl=0) and (escribiendo=false) =>
                                abrir_escritura[c3] ? any;
                                escribiendo:=true;
                or
                        for c4:=1 to NESC replicate
                                cerrar_escritura[c4] ? any;
                                escribiendo:=false;
                                npeticionesE:=npeticionesE-1;
                or
                        terminate
                end

        forever
end;
```

4.1.3. El problema de los filósofos

```
{Problema de los Filosofos}
program Filosofos;

const
	N = 5;

var
	pido_palillos, suelto_palillos: array[0..N-1] of channel of synchronous;

process type Filosofo(id:integer);
var
	veces: integer;
begin
	veces:=0;
	repeat
		(*PENSANDO*)
		pido_palillos[id] ! any;

		(*COMIENDO*)
		writeln(id,' COMIENDO');

		suelto_palillos[id] ! any;

		veces:=veces+1;
	until veces=5;
end;


process Controlador;
var
	palillos: array[0..N] of integer;
	i:integer;

begin
	for i:=0 to N-1 do palillos[i]:=1;
	repeat
		select
			for i:=0 to N-1 replicate (*EQUIVALE A TANTOS OR COMO ITERACIONES TENGA EL BUCLE*)
				when (palillos[i]=1) and (palillos[(i+1) mod N]=1) =>
					pido_palillos[i] ? any;
					palillos[i]:=0;
					palillos[(i+1) mod N]:=0;
		or
			for i:=0 to N-1 replicate (*EQUIVALE A TANTOS OR COMO ITERACIONES TENGA EL BUCLE*)
				suelto_palillos[i] ? any;
				palillos[i]:=1;
				palillos[(i+1) mod N]:=1;
		or
			terminate

		end

	forever
end;
```