Programación concurrente – Tres Arroyos

Programación distribuida – PMA – PMS

Conceptos:

Modelo: Procesos que se comunican mediante el envió y recepción de mensajes.
Cada proceso tiene memoria local (no compartida).
Los procesos envían y reciben mensajes hacia y desde canales, cuando se recibe un mensaje este se elimina del canal.

Exclusión mutua y sincronización:

Cada proceso trabaja en un espacio de direcciones distinto al resto por lo tanto no existe la necesidad de exclusión mutua.
Si existe información a compartir entre procesos debería encapsularse en un proceso.
La sincronización se realiza con el envío y recepción de mensajes.

PMA (Pasaje de mensajes Asincrónicos)

Canales:

Los canales son colas de mensajes de tamaño potencialmente infinito.

Definición:

```
Chan miCanal;
Chan misCanales[1..N]
```

Operaciones:

Send: Un proceso envía un mensaje y no espera a que este sea recibido, sigue su ejecución. Uso:

```
Send miCanal(mensaje);
Send misCanales[i](mensaje);
```

Receive: Un proceso espera por un mensaje, mientras el canal este vacío el proceso se demora hasta que haya algún mensaje. Si dos procesos quieren recibir de un mismo canal y hay un solo mensaje, solo un proceso recibirá el mensaje mientras el otro proceso se demorara hasta que llegue algún mensaje. Uso:

```
Receive miCanal(mensaje);
Receive misCanales[i](mensaje);
```

Empty: Los procesos pueden preguntar si existe un mensaje en algún canal. Uso:

```
Empty(miCanal);
Empty(misCanales[i]);
```

Guardas:

Alternativas multiples:

```
If Condicion Booleana
        Sentencia
*
  Condicion Booleana
        Sentencia
…
End if;
```

Las guardas se evalúan en algún orden arbitrario, se elige alguna verdadera no deterministicamente, si ninguna guarda es verdadera el if no tiene efecto.

Alternativa iterativa:

```
do Condicion Booleana
        Sentencia
*
  Condicion Booleana
        Sentencia
…
End do;
```

Las sentencias en las guardas son evaluadas y ejecutadas hasta que todas las condiciones sean falsas.

Uso común:

```
If not empty(miCanal)
        Receive miCanal(mensaje);
*
  Not empty (misCanales[i])
        Receive misCanales[i](mensaje)
…
End if;
```

Problemas del uso de Empty:

Si dos procesos al mismo tiempo preguntan por el mismo canal y este tiene solo un mensaje ambos aceptaran la condición como verdadera pero solo uno de los procesos podrá leer el mensaje, el otro procesos quedara bloqueado.

```
Process A
    …
    If not empty(canal)
            Receive canal();
    …
End process A;

Process B
```

```
        …
        If not empty(canal)
                Receive canal();
        …
    End process B;
```

Productor – Consumidor (Buffer implicito)

Ejemplo: existe un procesador de texto que necesita usar una impresora este le envía un texto que luego la impresora imprime.

```
Chan c_texto;
```

```
Process procesadorDeTexto
        While (true)
                "Genera el texto"
                Send c_texto(texto);
        End while;
End procesadorDeTexto;
```

```
Process impresora
        While (true)
                Receive c_texto(texto);
                "Imprime el texto"
        End while;
End impresora;
```

El canal funciona como buffer de manera implícita, en este caso el procesador de textos no espera que la impresora este lista pudiendo generar y enviar "infinitos" mensajes antes que la impresora este disponible, lo que no sera real.

Ejemplo: existe un procesador de texto que necesita usar una impresora, cuando la impresora esta libre le envía un texto que luego la impresora imprime.

```
Chan c_impresoraLibre;
Chan c_texto;
```

```
Process procesadorDeTexto
    While (true)
        "Genera el texto"
        Receive c_impresoraLibre(m);
        Send c_texto(texto);
    End while;
End procesadorDeTexto;
```

```
Process impresora
        While (true)
                Send c_impresoraLibre(m);
                Receive c_texto(texto);
                "Imprime el texto"
        End while;
End impresora;
```

En este caso hay una sincronización mas para que el procesador de textos espere que la impresora este disponible.
Notar que el código no cambiaria si existen mas de un procesador de texto y/o mas impresoras.
Usar el buffer implícito implica que las prioridades estarán dadas por la política del canal, si se tuvieran distintos procesadores de texto y se quiere dar prioridad no se podría manejar.

Prioridades

Ejemplo: existen dos tipos de procesadores de texto, los que generan texto a color y los que generan texto solo blanco y negro, existen además dos impresoras, una a color y otra blanco y negro.

Los procesadores de texto color solo imprimen en la impresora color, mientras que los de texto blanco y negro pueden imprimir en cualquiera de las dos impresoras, pero solo podrán imprimir en la impresora color si no hay ningún procesador de texto color usándola.

Error común: seguir la idea anterior para la implementación usando guardas en el procesador de texto blanco y negro. Se agrega una nueva idea, una impresora cuando esta libre envía un mensaje a un canal, los procesos verificaran cuando una impresora este libre haciendo empty del canal.

```
Chan c_impresoraColorLibre;
Chan c_impresoraByNLibre;
Chan c_ColorTexto;
Chan c_ByNTexto;

Process impresoraByN
        While (true)
                Send c_impresoraByNLibre(m);
                Receive c_ByNTexto(texto);
                "Imprime el texto"
        End while;
End impresoraByN;

Process procesadorDeTextoByN[p:1..N]
Ok:boolean = false;
        "Genera el texto"
        While not ok
                If not empty(c_impresoraColorLibre);
                        Receive c_impresoraColorLibre(m);
                        Send c_ColorTexto(texto);
                        Ok=true;
                * not empty(c_impresoraByNLibre);
                        Receive c_impresoraByNLibre(m);
                        Send c_ByNTexto(texto);
                        Ok=true;
                End if;
        End;
End procesadorDeTextoByN;

Process procesadorDeTextoColor[p:1..N]
        "Genera el texto"
        Receive c_impresoraColorLibre(m);
        Send c_ColorTexto(texto);
End procesadorDeTextoColor;

Process impresoraColor
        While (true)
                Send c_impresoraColorLibre(m);
                Receive c_ColorTexto(texto);
                "Imprime el texto"
        End while;
End impresoraColor;
```

Esto tiene varios problemas:

Los procesadores de blanco y negro podrían imprimir siempre en una impresora color aun cuando la impresora blanco y negro este libre.

Si la impresora color envia un mensaje avisando que esta libre y hay 2 procesadores de texto blanco y negro en las guardas, ambos podrían evaluar el if not empty(c_impresoraColorLibre), ambos podrían ingresar por ser esta guarda verdadera y solo uno podría hacer el receive correspondiente, mientras que el otro quedaría bloqueado esperando por la impresora color, no seria un inconveniente porque en algún momento la usaría pero no maximizaría la concurrencia en caso que la impresora blanco y negro estelibre.

Solucion: Agregar un proceso administrador que se encargue de las impresoras manejando además las prioridades.

```
Process procesadorDeTextoByN[p:1..N]
        "Genera el texto"
        Send c_pedirImpresoraByN(p);
        Receive c_obtenerImpresoraByN[p](tipo);
        If tipo="color"
                Send c_ColorTexto(texto);
        Else if tipo="ByN"
                Send c_ByNTexto(texto);
        End if;
End procesadorDeTextoByN;

Process procesadorDeTextoColor[p:1..N]
        "Genera el texto"
        Send c_pedirImpresoraColor(p);
        Receive c_obtenerImpresoraColor[p]();
        Send c_ColorTexto(texto);
End procesadorDeTextoColor;

Process impresoraColor
        While (true)
                Send c_impresoraColorLibre(m);
                Receive c_ColorTexto(texto);
                "Imprime el texto"
        End while;
End impresoraColor;

Process impresoraByN
        While (true)
                Send c_impresoraByNLibre(m);
                Receive c_ByNTexto(texto);
                "Imprime el texto"
        End while;
End impresoraByN;

Process administrador
        While true()
                If not empty(c_pedirImpresoraColor) and not empty(c_impresoraLibreColor)
                        Receive c_pedirImpresoraColor(id);
```

```
                    Receive c_impresoraLibreColor(m);
                    Send c_obtenerImpresoraColor[id](m);
            * not empty(c_pedirImpresoraByN) and not empty(c_impresoraLibreByN)
                    Receive c_pedirImpresoraByN(id);
                    Receive c_impresoraLibreByN("ByN");
                    Send c_obtenerImpresoraByN[id](m);
            * not empty(c_pedirImpresoraByN) and empty(c_impresoraLibreByN)  and not
            empty(c_impresoraLibreColor) and empty(c_pedirImpresoraColor)
                    Receive c_pedirImpresoraByN(id);
                    Receive c_impresoraLibreByN(m);
                    Send c_obtenerImpresoraByN[id]("color");
            End if;
        End while;
End administrador;
```

El hecho de tener un arreglo de canales para darle la impresora a los procesadores es útil para no delegar el orden a la política de recepción del canal. De esta forma se realiza una comunicación directa con el proceso que solicito una impresora.

Barrera distribuida

Una barrera de N procesos, la forma más simple es haciendo que N-1 envien un mensaje que recibira un proceso, cuando este recibe los N-1 mensajes envía a cada uno de los procesos.

```
Process P[p:1..N]
    If p==1
            For i=1 to N-1
                    Receive llegue();
            For i=1 to N-1
                    send seguir();
    Else
            send llegue();
            receive seguir();
    End if;
End P;
```

Relojes – Estados (Vaciado del buffer)

Se tiene un banco con una sola caja, los clientes llegan se encolan y esperan ser atendidos, si pasados 15 minutos un cliente no fue atendido se retira.

El estado del cliente se encapsula en un proceso, tanto el reloj como el empleado deben interactuar con este proceso que deberá recibir el mensaje de los ambos obligatoriamente para que no queden mensajes encolados que nadie vaya a recibir, además, en caso que el cliente se haya ido, deberá avisarle al empleado que ya no debe atenderlo.

```
Process Cliente[c:1..N]
    Send cola(c);
    Send contar[c]();
    Receive estado[c](estado);
```

```
    If estado == “atender”
            “Se atiende”
    Else
            “se va”
    End if;
End Cliente;

Process Estado[e:1..N]
    Receive evento[e](estado);
    If estado = “atender”
            Send estado[e](“atender”);  //Envia al cliente el estado
            Send atender(“atender”);  //Avisa al empleado que lo tiene que atender
            Receive evento[e](reloj);  //Espera por el mensaje del reloj
    Else
            Send estado[e](“irse”);  //envía al cliente el estado
            Receive evento[e](empleado);  //Espera por el mensaje del empleado
            Send atender(“se fue”);  //Avisa al empleado que NO lo tiene que atender
End Estado;

Process Empleado
    While true
            Receive cola(cliente);
            Send evento[cliente](“atender”);
            Receive atender(estado);
            If estado==”atender”
                    //Atiende al cliente
            End if;
    End while

End Empleado;

Process Reloj[r:1..N]
    Receive contar[r]();
    Delay(15);
    Send evento[r](“irse”);
End Reloj;
```

PMS (Pasaje de mensajes sincrónicos)

Canales:

Los canales son unidireccionales y con capacidad para un mensaje, existen uno por cada par de procesos aunque puede aceptarse que haya más por cada par de procesos por una cuestión semantica.
Por un mismo canal no podrá enviar o recibir mensajes más de un proceso, solo aquellos a los que el canal pertenece.
No se puede preguntar por si un canal esta Empty.

Definición:

Canales:

Chan canal;
Chan canales[1..N];

Operaciones:

Envío(!): Un proceso envía un mensaje y espera a que este sea recibido, sigue su ejecución. Uso:

procesoReceptor!canal(datos)

Es posible que haya varios procesos del mismo tipo por lo tanto se consideran arreglos de procesos y si se quiere enviar al procesoReceptor i-esimo seria:

procesoReceptor[i]!canal(datos)

donde se supone que los procesos receptores se definen:

```
Process procesoReceptor[p:1..n]
…
        procesoEmisor?canal(datos);
…
End process;
```

Recepción(?): Un proceso espera por un mensaje, mientras el canal este vacío el proceso se demora hasta que haya algún mensaje. Uso:

procesoEmisor?canal(datos)

En caso de recibir del proceso i-esimo si hay varios del mismo tipo:

procesosEmisores[i]?canales(datos)

NOTA: como existe un canal por cada par de procesos, si exiten dos procesos con solo una instancia de estos se podría obviar el nombre del canal:

Process A	**Process B**
B!(mensaje);	**A?(mensaje);**
End;	**End;**

De todos modos a fines semánticos es útil nombrar el canal para saber que es lo que se quiere transmitir en este; si la comunicación entre A yB involucrara mas de un envio y recepcion ocurriría lo siguiente:.

Process A	**Process B**
B!(mensaje);	**A?(mensaje);**
B?(mensaje);	**A!(mensaje);**
End;	**End;**

En este caso no se sabe que es lo que se hace en cada envio y recepción , suponiendo que A le solicita algo a B y que B responde se podrían agregar los nombres de los canales para indicar el propósito de la comunicación.

Process A	**B!solicitud(mensaje);**

```
        B?respuesta(mensaje);                    A?solicitud(mensaje);
End;                                             A!respuesta(mensaje);
Process B                                    End;
```

Guardas:

Alternativas multiples:

```
If Condicion Booleana -> cuantificador, sentencia de recepcion
        Sentencia
*
  Condicion Booleana -> cuantificador, sentencia de recepcion
        Sentencia
…
End if;
```

Solo se aceptan sentencias de recepción.
Las guardas se evalúan en algún orden arbitrario, se elige alguna verdadera no deterministicamente, una guarda es verdadera si se cumple la condición booleana y existe un mensaje en el canal de recepción, la condición booleana puede no existir en cuyo caso se asume true, si ninguna guarda es verdadera el if no tiene efecto.

Alternativa iterativa:

```
do Condicion Booleana -> cuantificador, sentencia de recepcion
        Sentencia
*
  Condicion Booleana -> cuantificador, sentencia de recepcion
        Sentencia
…
End do;
```

Las sentencias en las guardas son evaluadas y ejecutadas hasta que todas las condiciones sean falsas.

Uso común:

```
If condicion -> (*) procesoEmisor1[i]?Canal(mensaje);
        Sentencias;
*
  ProcesoEmisor2?Canal(mensaje)
        Sentencias;
*
  Condición->ProcesoEmisor3?Canal(mensaje)
        Sentencias;
…
End if;
```

Cuando se tiene varios procesos de los que se puede recibir y estos son procesos iguales es útil tener guardas con cuantificadores para simplificar el hecho de tener que poner una guarda por cada proceso, es el caso de la primera guarda, se recibe por cualquiera de los canales del arreglo de canales, para esto se requiere el cuantificador (*) que indica la posibilidad de recibir por cualquiera de los canales involucrados en la guarda, e identifica con [i] porque canal se recibió.

Posibilidad de deadlock:

Dado que un proceso cuando envía un mensaje espera a que este sea recibido existe mayor posibilidad de bloqueos.

```
Process A
    ProcesoB!Canal1(mensaje);
    ProcesoB?Canal2(mensaje)
End A

Process B
    ProcesoA!Canal2(mensaje);
    ProcesoA?Canal1(mensaje)
End B
```

Ninguno de los procesos podrá avanzar.

Maximizacion de la concurrencia:

Para maximizar la concurrencia muchas veces se requiere de un proceso intermedio.
Productor – consumidor:

Sin maximizar la concurrencia: el problema es que el productor deberá esperar a que el consumidor reciba el ítem para poder producir otro.

```
Process Productor
  While true
        //Produce un ítem
        Consumidor!c_item(item);
  End while;
End Productor;

Process Consumidor
  While true
        Productor?c_item(item);
        //Consume un item
  End while;
End Consumidor;
```

Maximizando concurrencia: Se agrega un proceso buffer intermedio permitiendo que el productor siga trabajando aunque el consumidor no haya recibido aun el mensaje.

```
Process Buffer
Cola q;
        While true
                Do Productor?c_productor(item);
```

```
                    Push(q,item);
                *   not empty(q), Consumidor?c_consumidor()
                    item=pop(q);
                    Consumidor!c_item(item);
        End While;
End Buffer;

Process Productor
  While true
        //Produce ítem
        Buffer!c_productor(item);
  End while;
End Productor;

Process Consumidor
  While true
        Buffer!c_consumidor();
        Buffer?c_item(item);
        //Consume item
  End while;
End Consumidor;
```

Mas de un productor: En caso de que haya mas de un productor se requiere de un arreglo de canales para estos, además modifica el proceso buffer agregando un cuantificador a la guarda.

```
Process Buffer
Cola q;
        While true
                Do (*) Productor[i]?c_productor(item);
                    Push(q,item);
                *   not empty(q), Consumidor?c_consumidor()
                    item=pop(q);
                    Consumidor!c_item(item);
        End While;
End Buffer;

Process Productor[p:1..N]
  While true
        //Produce ítem
        Buffer!c_productor(item);
  End while;
End Productor;

Process Consumidor
  While true
        Buffer!c_consumidor();
        Buffer?c_item(item);
        //Consume item
  End while;
End Consumidor;
```