Examen Práctico – Modulo II

En el presente apunte se repasarán los conceptos fundamentales para entender los temas evaluados en la próxima evaluación de la asignatura. También se citan varios ejemplos básicos y específicos tanto de la explicación como de la práctica.

1 – PMA

Significa "Pasaje de Mensajes Asincrónico", y hace referencia a que el envío de un mensaje por parte de un proceso se realiza de forma *inmediata*[1] sin esperar que el otro proceso confirme su recepción. Por otra parte, cuando un proceso debe recibir un mensaje, se demora su ejecución hasta que se reciba un elemento en el canal especificado.[2]

Para realizar las comunicaciones se deben declarar los canales al principio del programa, indicando cuáles son los parámetros (tipo obligatorio, nombre opcional). Se pide que cada canal tenga al menos un parámetro: "algo se tiene que enviar".

Importante aclarar que los canales para PMA y PMS son unidireccionales, por lo que para enviar y recibir datos entre un mismo par de procesos es necesario usar dos canales diferentes, en otras palabras, no existen los parámetros E/S.

La sintaxis para declarar canales es:	`chan nomCanal(tipo1, tipo2);`
Para declarar arreglos de canales:	`chan nomArreglo[1..N](t1, t2);`
Para enviar un mensaje:	`send nomCanal(valor1, valor2);`
Para recibir un mensaje:	`receive nomCanal(var1, var2);`
Para consultar sin bloquearme:	`empty(nomCanal);`

Se debe evitar usar la función *empty()* cuando hay más de un posible receptor, ya que si hay un único mensaje en el canal puede ocurrir que a ambos se les devuelva verdadero, y ambos podrían intentar recibir el mensaje, pero solo uno tendrá éxito.

Otra aclaración: para recibir, no está permitido usar [*]. Únicamente en PMS.

---

[1] Operación no bloqueante. La única demora que podría ocurrir es que el canal a utilizar está siendo "modificado" por otro proceso (se está encolando o desencolando un mensaje del mismo).

[2] Operación bloqueante si el canal está vacío. Cuando hay al menos un mensaje, se retira el más viejo.

1.1 – Orden de llegada

Los canales funcionan como colas de mensajes. Se garantiza que al recibir un mensaje mediante *receive()* se retira el mensaje que haya sido enviado primero.

Además, podemos quedarnos tranquilos que al enviar un mensaje mediante *send()*, aunque no haya ningún proceso esperando el mismo en dicho instante, el mensaje quedará encolado en el canal, por ende, no existe riesgo que el mismo se pierda.

1.2 – Envío y espera de respuesta

En muchas ocasiones, será necesario esperar una respuesta del otro proceso luego de que hayamos enviado un mensaje, antes de continuar con la ejecución. En otras palabras, se realiza un *send()*, y en la instrucción siguiente un *receive()*.

Hasta ahí todo bien. Supongamos que somos N procesos que le enviamos reportes a un único empleado, y el mismo deberá respondernos luego de analizarlo. ¿Cómo hace para saber a quien de los N procesos responderle? Respuesta: debo pasar mi ID.

Esto implica que existe un único canal para enviar reportes, y N canales para enviar respuestas (cada proceso hará un *receive()* en su canal correspondiente).

```
chan Reportes(int, texto);
chan Respuestas[N](texto);

Process Persona[id: 0..N-1]
{ texto R, Res;
  while (true)
    { R = generarReporteConProblema;
      send Reportes ( id , R );
      receive Respuestas[id] ( Res );
    };
}
```

```
Process Empleado
{ texto Rep, Res;
  int idP;
  while (true)
    { receive Reportes ( idP , Rep );
      Res = resolver (Rep);
      send Respuestas[idP] ( Res );
    };
}
```

1.3 – Variante con más de un empleado

Supongamos ahora que los N procesos necesitan enviar un reporte y recibir una respuesta, pero tenemos 3 empleados. No debemos mandarle a uno en específico y no sabemos si están libres. ¿Debemos usar un *Admin*? ¡NO! Es la misma solución.

Como los empleados no tienen otra cosa que hacer, sólo se dedicarán a hacer *receive()* sobre el único canal de reportes. El primero que lo vea será quien me responda.

1.4 – Uso de empty

Cada uno de los empleados del caso anterior, si no tiene un reporte para procesar, decide volver en 5' a su trabajo. Para facilitar las cosas supongamos que las personas solo dejan los reportes en la mesa y se van (no hay espera de respuesta, no pasan su ID).

Supongamos que hay un único reporte pendiente, y los 3 empleados están libres, entonces todos intentarán recibirlo, pero solo uno tendrá éxito. ¿Cómo hago para que los otros dos que fallaron no se bloqueen? Respuesta: usar un *Admin* (ahora sí).[3]

La idea consiste en que cada empleado, al liberarse, realice un pedido al Admin, para que éste le responda con un reporte (debe procesarlo) o un "vacío" (volver en 5').

Si quisiéramos evitar al Admin, podría existir un canal de reportes por cada empleado, pero esto obligaría a las personas a elegir alguno (no es lo pedido).

```
chan Reportes(texto);
chan Pedido(int);
chan Siguiente[3](texto);

Process Empleado[id: 0..2]
{ texto Rep;
  while (true)
    { send Pedido(id);
      receive Siguiente[id] ( Rep );
      if (Rep <> "VACIO") resolver (Rep)
      else delay (600); //lee 10 minutos
    };
}
```

```
Process Coordinador
{ texto Rep;
  int idE;
  while (true)
    { receive Pedido ( idE);
      if (empty (Reportes) ) Rep = "VACIO";
      else  receive Reportes ( Rep )

      send Siguiente[idE] ( Rep );
    };
}
```

1.5 – Selección de un canal según demanda

Hemos tratado los ejemplos de la explicación práctica. Sin embargo, luego aparecen cosas como "se selecciona la caja con menos personas esperando". Esta situación se resuelve también con un *Admin*, que mantiene información de la cantidad de personas en cada caja (en un vector), para que le responda rápidamente a quien recién llega cuál es donde le conviene esperar (el mínimo del vector, luego el valor de dicha posición se incrementa). Para mantener actualizada la información, a su vez cada empleado de las cajas le avisa al Admin al terminar la atención de una persona (se decrementa el valor de su posición en el vector, sin espera de respuesta).

[3] Si fuese un único empleado receptor de reportes está más que claro que el Admin no es necesario.

1.6 – Manejo de prioridades

Cuando un proceso puede recibir mensajes de dos canales distintos en un momento dado, surge el problema de desconocer en cuál de los dos canales debe realizar un *receive()*. Mediante la función *empty()* se puede prevenir un bloqueo al recibir un mensaje en caso de haber un único receptor. Sin embargo, consultar de manera reiterada cuando está vacío es busy waiting, y debe evitarse.

Una solución es agregar un canal para transmitir "signals" que despierten al receptor cada vez que se haya enviado un mensaje en alguno de los dos canales. Si hay más de un posible receptor, la responsabilidad se delega a un *Admin* (símil caso 1.4).

2 – PMS

Significa "Pasaje de Mensajes Sincrónico", y hace referencia a que tanto el envío de mensajes como la recepción de los mismos son operaciones bloqueantes. A diferencia de PMA, la ejecución de un proceso no continuará hasta que el mensaje sea recibido en el proceso destino. Respecto a la sintaxis, ésta cambia rotundamente, ya que los canales están declarados implícitamente, y en lugar de *send()* y *receive()* usamos "!" y "?".

Para enviar un mensaje:	`procDest!nomCanal(valor1, valor2);`
Para recibir un mensaje:	`procOrig?nomCanal(var1, var2);`
De cualquier proceso de un tipo[4]:	`procOrig[*]?nomCanal (var1);`

Tampoco existe la función *empty()* para consultar si un canal está vacío sin producir un bloqueo, sino que se utiliza un IF o DO no determinístico. Son estructuras de alternativa múltiple; a cada una se la denomina *guarda* y son de la forma $B; C \rightarrow S$.

En criollo, pueden tener una condición booleana **B** (opcional, si no está presente se considera verdadera), una sentencia de comunicación **C** (obligatoria, sólo se permite que sea una recepción mediante "?") y un conjunto de sentencias **S** (obligatorio, se ejecutan si B es verdadera y C realizable[5], puede usarse un "skip" como acción nula). Las guardas tienen 3 estados: fallo ($\bar{B}$), bloqueo ($B \cdot \bar{C}$) y éxito ($B \cdot C$, antes mencionada).

[4] Este "comodín" es un IF no determinístico. Se debe tener cuidado que no respeta orden de llegada.
[5] Entiéndase como que la operación no produce demora si se lleva a cabo: hay mensaje listo para retirar.

2.1 – Ciclos de vida no determinísticos

Únicamente si todas las guardas resultaron en "fallo" se sale de la estructura de selección, continuando con las instrucciones posteriores (no se ejecuta ninguna S).

Si todas las guardas resultaron en "bloqueo", tanto para el IF como el DO se demora el proceso hasta que alguna sea exitosa. La diferencia entre ambas estructuras es que en el IF sólo se ejecutará una alternativa exitosa, en cambio en el DO se repetirá el proceso hasta que todas resulten en fallo (bucle infinito si alguna no tiene B).

Si hay más de 1 guarda exitosa, se elige alguna de ellas de forma no determinística, no importa el orden en el programa o antigüedad de sus mensajes.

2.2 – Envío sin espera de respuesta

En el caso simple, con 1 proceso que envía reportes a un único empleado que le debe responder, con usar las sentencias "!" y "?" en el orden correcto es suficiente. Ahora bien, si se requiere que el primer proceso envíe reportes continuamente sin espera de respuesta, se debe tener en cuenta que el mismo se bloqueará si el empleado no está listo para recibirlos. Rápidamente surge la necesidad de incorporar un *Admin* a la solución.

La segunda guarda debe tener una condición booleana, ya que no se puede procesar el pedido de *Mantenimiento* si no hay reportes pendientes. Por lo tanto debo demorar la recepción de ese mensaje hasta estar en condiciones de entregarle un reporte.

```
Process Admin
{ cola Buffer;
  texto R;
  do Testeo?reporte(R) → push (Buffer, R);
  □ not empty(Buffer); Mantenimiento?pedido() →
                Mantenimiento!reporte (pop (Buffer));
  od
}
```

```
Process Testeo
{ texto R, Res;
  while (true)
    { R = generarReporteConProblema;
      Admin!reporte(R);
    }
}

Process Mantenimiento
{ texto Rep, Res;
  while (true)
    { Admin!pedido();
      Admin?reporte(Rep);
      Res = resolver(Rep);
    }
}
```

2.3 – Envíos múltiples, recepción única

A diferencia de PMA, ahora existe al menos un canal entre cada par de procesos, incluso si hay varios del mismo tipo. Entonces, si hay N procesos enviando reportes, cada uno enviará los mensajes por un canal distinto: el receptor deberá usar el comodín [*].

2.3.1 – Orden de llegada

No hay que olvidar que el [*] es sólo un "atajo de escritura" de guardas para cada uno de los canales del vector, pero no elige al que tiene el mensaje más antiguo. Para respetar el orden de llegada, es necesario retirar los mensajes de manera rápida, ya que una vez acumulados (en diferentes canales) no hay vuelta atrás. Se requiere un *Admin*.

2.3.2 – Variante con respuesta

Si sólo se requiere atender reportes por orden de llegada, con lo descrito alcanza, pero si también se debe responder a la persona debo conocer su ID (para el canal).

```
Process Admin
{ cola Fila;    texto R;    int idT;

  do Testeo[*]?reporte(R, idT) → push (Fila, (R,idT));
  □  not empty(Fila); Mantenimiento?pedido() → pop (Fila, (R, idT))
                                               Mantenimiento!reporte (R, idT);
  od
}
```

```
Process Testeo[id: 0 ..N-1]
{ texto R, Res;
  while (true)
    { R = generarReporteConProblema;
      Admin!reporte(R, id);
      Mantenimiento?respuesta(Res);
    }
}
```

```
Process Mantenimiento
{ texto Rep, Res;       int idT;
  while (true)
    { Admin!pedido();
      Admin?reporte(Rep, idT);
      Res = resolver(Rep);
      Testeo[idT]!respuesta(Res);
    }
}
```

2.3.3 – Variante con más de un empleado

La única diferencia con el anterior es que cada empleado de mantenimiento debe pasar su ID al *Admin* para que luego éste le pueda responder con un reporte a procesar, y si se debe responder a la persona, ésta deberá usar [*], ya que no sabe quién lo procesó.

```
Process Admin
{ cola Fila;    texto R;    int idT, idM;

  do Testeo[*]?reporte(R, idT) → push (Fila, (R,idT));
  □ not empty(Fila); Mantenimiento[*]?pedido(idM)→
                                          pop (Fila, (R, idT))
                                          Mantenimiento[idM]!reporte (R, idT);
  od
}
```

2.4 – Barreras

En muchas ocasiones puede ser necesario esperar a que todos los procesos lleguen a un cierto punto para dar comienzo a otra etapa, como empezar una carrera. La manera recomendada de hacerlo es mediante [*] unas N veces (si son N procesos), ya que no se puede conocer de antemano en qué orden llegarán, y seguro no coincidirá con el ID.

Luego se deberá enviar un aviso de "inicio" a cada uno de los procesos.

3 - ADA

Es un lenguaje de programación concurrente que permite usar procesos (tareas) con canales bidireccionales, por lo que pueden tener parámetros de entrada, salida y E/S, a diferencia de PMA y PMS. Los canales se indican como ENTRY en la declaración de cada TASK o TASK TYPE, únicamente para los casos que se utilice como recepción.

Los arreglos de tareas se inicializan como `arrP : array(1..P) of NomTipo;`

Luego, deben detallarse las instrucciones y variables locales en el cuerpo de cada tarea (TASK BODY). Puede haber operaciones de envío sincrónico, indicando la tarea seguido del llamado al ENTRY. Ejemplo: `NomTarea.NomEntry(valor1, val2);`

Así nomás, produce un bloqueo hasta que el pedido termine de aceptarse. Si quiero evitar eso existe la estructura SELECT + OR DELAY y SELECT + ELSE. El primero establece un tiempo máximo de *aceptación*[6], y el segundo solo realiza un intento para dicho instante. Si falla pasa a ejecutarse las instrucciones del bloque alternativo.

Por otra parte, para la recepción de mensajes se utiliza la función *accept* seguido de la firma del ENTRY como se declaró al principio. Si hay parámetros involucrados, luego puede indicarse **do** + un conjunto de instrucciones que los utilice, mientras el proceso emisor sigue bloqueado, y se agrega un *End NombreEntry*. Una vez terminado el *accept*, el proceso emisor y receptor continúan ejecutando el resto de instrucciones.

Para recibir mensajes de forma no determinística o sin producir bloqueos, existe también una estructura SELECT con guardas (similar al IF no determinístico). Si se desea aceptar mensajes bajo cierta condición B[7], se antepone *when(B) => accept …*

[6] Se espera a que el proceso destino acepte el mensaje, no importa si luego se demora en terminarlo.
[7] Si se desea saber si un canal está vacío o no, en lugar de *empty()* existe el atributo numérico *'count*

3.1 – Identificación de tareas

El ID de cada tarea es desconocido al comienzo de cada una. Si es necesario, debe ser comunicado desde cualquier otro proceso, o bien, desde el programa principal.

3.2 – Envío y espera de respuesta

Mediante la sincronización *Rendezvous*, los canales son bidireccionales. Esto implica que, si ambas partes no deben realizar otro tipo de comunicación en el medio, no se debe utilizar un ENTRY para enviar un reporte y otro para recibir la respuesta, sino que dentro del cuerpo del *accept* se realiza el procesamiento y se asigna la salida.

```
Procedure Banco1 is
  Task empleado is
     Entry Pedido (D: IN texto; R: OUT texto);
  End empleado;
  Task type cliente;
  arrClientes: array (1..N) of Cliente;
  Task Body cliente is
       Resultado: texto;
  Begin
       Empleado.Pedido ("datos", Resultado);
  End cliente;

  Task Body empleado is
  Begin
     loop
        accept Pedido (D: IN texto; R: OUT texto) do
             R := resolverPedido(D);
        end Pedido;
     end loop;
  End empleado;
Begin
   null;
End Banco1;
```

¿Cómo se asegura la atención en orden de llegada?

Los *entry call* a un *entry* se almacenan en la cola implícita del mismo de acuerdo al orden de llegada.

3.3 – Variante con más de un empleado

Cada tarea "*empleado"* tiene sus ENTRY individuales, por lo que no es posible utilizar un canal unificado para los pedidos sin tener un *Admin* como intermediario. Tal cual sucedía en PMS, cada empleado pide el siguiente al liberarse, pero sin necesidad de pasar el ID[8]. Sin embargo, si se debe responder a la persona, ésta sí debe enviar su ID.

3.4 – Manejo de prioridades

Los ENTRY para cada tipo de mensaje deben estar diferenciados, para evitar tener que aceptar todos los mensajes y volverlos a encolar si no correspondía atenderlos. Para implementarlo, se reemplaza el "Signal" por estructuras no determinísticas, aunque se debe agregar la condición "Prioritario'count = 0" a las guardas que no tienen prioridad.[9]

[8] Si no hay límites de espera, el *Admin* puede tener *accept* anidados (Siguiente, y luego PedidoCliente).
[9] Recordar que el orden de escritura de las guardas no influye en absoluto en la elección de las mismas.

3.5 – Demora innecesaria

Si una tarea controlada por un Admin debe repartirse entre N procesos, como la búsqueda en servidores propios a cada uno, se debe tener en cuenta que es probable que no todos los procesos estén listos para comenzar. Incluso alguno de los procesos puede devolver su resultado antes de finalizar la repartición. Entonces hay 2 demoras:

- Procesos listos sin tarea asignada debido a alguno con ID menor demorado.
- Procesos con resultados listos que no pueden terminar hasta enviarlo al Admin.

El primer problema se resuelve reemplazando los llamados por *accept Pedido* en el Admin, de modo que cada proceso envíe una solicitud cuando esté listo. El segundo problema se resuelve mediante la estructura de SELECT con 2 alternativas (OR).

⚠ **Atención:** esto NO aplica para tareas repetitivas. En ese caso, usar 2 bucles "for".[10]

```
Procedure ContadorOcurrencias is
  Task Admin is
       entry Valor (num: out integer);
       entry Resultado (res: in integer);
  End admin;
  Task type Contador;
  ArrC: array (1..N) of Contador;
  Task body Contador is
     vec: array (1..V) of integer := InicializarVector;
     valor, cant: integer :=0;
  Begin
     Admin.valor(valor);
     for i in 1..V loop
        if (vec(i) = valor) then
           cant:=cant+1;
        end if;
     end loop;
     Admin.Resultado(cant);
  End contador;

  Task body Admin is
     numero: integer := elegirNumero; total: integer := 0;
  Begin
     for i in 1..2*N loop
        select
           accept Valor (num: out integer) do
              num := numero;
           end Valor;
        or
           accept Resultado (res: in integer) do
              total:= total + res;
           end Resultado;
        end select;
     end loop;
  End Admin;

Begin
  null;
End ContadorOcurrencias;
```

3.6 – Barreras

Si se debe esperar que varias tareas lleguen a un mismo punto del programa, puede utilizarse un bucle "for" que acepte los pedidos y luego realizar un llamado a cada uno de los procesos para que continúen (esto requiere identificar a los mismos). En caso de ser pocos procesos, no más de 5, se pueden anidar *accept* y terminarlos todos al final, de modo que se evita una segunda comunicación y la identificación de los procesos.

[10] Lo más importante es que se maximice la concurrencia, es decir, no hacer un programa secuencial.