Explicación Práctica de Pasaje de Mensajes Sincrónicos

Ejercicios

Links a los archivos con audio

El archivo en formato MP4 de la explicación con audio se encuentra comprimido en el siguiente link:

- https://drive.google.com/u/0/uc?id=15OVFBf0CJPdj_ADqxkoEv3PfIwGB32he&export=download

Pasaje de Mensajes Sincrónicos (PMS)

- Los programas se componen **SÓLO** de procesos (NO EXISTEN LAS VARIABLES COMPARTIDAS).

- Los canales son de tipo *link* (un único emisor y un único receptor) y sincrónicos. Son estructuras implícitas y no se deben declarar.

- Los procesos interactúan entre ellos ÚNICAMENTE por medio del envío de mensajes (tanto para comunicación como para sincronización por condición).

- No se requiere sincronización por exclusión mutua ya que no existen las variables compartidas.

Sintaxis - sentencias de comunicación

- Sentencias de comunicación: no existen las estructuras de canales, la comunicación se realiza nombrando al proceso con el cual se realiza la comunicación.
  - Envío (!): la operación es bloqueante y sincrónica, se demora hasta que la recepción haya terminado.

    destino!port (mensaje)
    destino[i]!port (mensaje)

  - Recepción (?): la operación es bloqueante y sincrónica.

    origen?port (variable)
    origen[i]?port (variable)

    origen[*]?port (variable)

Sintaxis – comunicación guardada

- Uso de Comunicación Guardada (If y Do): permite seleccionar de forma no determinística entre varias alternativas de comunicación (en el caso de la práctica SÓLO RECEPCIONES) en base a las condiciones del proceso y los mensajes que están listos para ser recibidos.
- Guardas: cada alternativa es una guarda con la forma: $B; C \rightarrow S$
  - ***B***: condición booleana que puede no estar (en ese caso se considera true), e indica si el proceso está o no en condiciones de procesar el mensaje recibido en ***C.***
  - ***C***: sentencia de comunicación (en la práctica sólo RECEPCIÓN) que seguro debe estar**.**
  - ***S***: conjunto de sentencias que se ejecutarán en caso de ser elegida la guarda.

Sintaxis – comunicación guardada

- Evaluación de una guarda:
  - ***Exito***: la condición booleana es Verdadera (o no la tiene) y la comunicación se puede realizar sin producir demora (el emisor está esperando hacer la comunicación).
  - ***Fallo***: la condición booleana es Falsa, sin importar lo que ocurra con la sentencia de comunicación.
  - ***Bloqueo***: la condición booleana es Verdadera (o no la tiene) pero la comunicación NO se puede realizar sin producir demora (el emisor aún no llegó a la sentencia de envío).

Sintaxis – comunicación guardada

- En el ***if "guardado"*** se evalúan todas las guarda y en base a eso:
  - Si una o más son EXITOSAS se selecciona una de ellas en forma NO DETERMINISTICA, se ejecuta la sentencia de recepción (***C***) que forma parte de la guarda, y posteriormente el conjunto de sentencias asociadas a la guarda (***S***).
  - Si todas las guardas FALLAN no se selecciona ninguna y se sale del IF sin realizar ninguna acción.
  - Si no hay ninguna guarda EXITOSA pero hay una o más con estado BLOQUEO entonces el proceso se demora en el IF hasta que haya una guarda exitosa. En ese momento se ejecuta igual que el primer caso.

- El ***do "guardado"*** funciona de la misma manera que el IF, con la única diferencia que en lugar de hacerlo una vez repite el mecanismo hasta que todas las guardas FALLAN.

EJEMPLO DE BUFFER DE LA TEORIA

En la teoría se implementó un proceso para manejar un Buffer Limitado con el siguiente código

```
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
}
Process Oeste
{ while (true)  Copiar!(generar caractér);
}
Process Este
{ char car;
  while (true)  Copiar?(car);
}
```

En la práctica NO se permite una sentencia de ENVÍO como parte de la guarda.

EJEMPLO DE BUFFER DE LA TEORIA

Ambas guardas deben formarse con una sentencia de recepción, por lo que el proceso copiar, en la segunda guarda, debe esperar a que LE PIDAN el siguiente carácter, y recién ahí se lo puede enviar (para evitar hacer el envío cuando el receptor no está listo para hacer la comunicación).

```
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?() → Este!(buffer[front])
                             cantidad := cantidad - 1;
                             front := (front + 1) MOD 80;
   od
}
```

```
Process Este
{ char c;
   while (true)
    {  Copiar!();
       Copiar?(c);
       //usa c
    }
}
```

```
Process Oeste
{ while (true)  Copiar!(generar caractér);
}
```

EJERCICIO 1

En una empresa de software hay un empleado *Testeo* que prueba un nuevo producto para encontrar errores, cuando encuentra uno generan un reporte para que otro empleado *Mantenimiento* corrija el error y le responda. El empleado *Mantenimiento* toma los reportes para evaluarlos, hacer las correcciones necesarias y responderle al empleado *Testeo*.

Lo primero es definir la estructura del programa: que procesos se van a necesitar

En este problema hay dos procesos: Testeo y Mantenimiento.

Testeo le debe enviar el reporte a *Mantenimiento* y esperar a que le responda para continuar trabajando. Por lo tanto no se requiere modelar un buffer donde se vayan almacenando los reportes, ya que nunca habrá mas de uno a la vez.

EJERCICIO 1

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

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

Ambos procesos se comunican directamente entre ellos 2 veces por cada reporte generado, una con el *reporte* y otra con el *resultado*.

EJERCICIO 2

En una empresa de software hay un empleado *Testeo* que prueba un nuevo producto para encontrar errores, cuando encuentra uno generan un reporte para que otro empleado *Mantenimiento* corrija el error **(no requiere una respuesta para seguir trabajando) y continua trabajando**. El empleado *Mantenimiento* toma los reportes para evaluarlos y hacer las correcciones necesarias.

Vamos a partir de la solución anterior suprimiendo la comunicación de la respuesta

Process Testeo

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

Limita la concurrencia.

Process Mantenimiento

```
{ texto Rep, Res;
  while (true)
    { Testeo?reporte(Rep);
      Res = resolver(Rep);
    }
}
```

EJERCICIO 2

El proceso *Testeo* debe esperar a que Mantenimiento haya terminado de procesar el reporte anterior para poder comunicarle el nuevo, durante ese tiempo de demora podría (y debería de acuerdo al enunciado) seguir trabajando. Por esta razón se limita o reduce la concurrencia. → **USAR UN BUFFER**

```
Process Admin
{ cola Buffer;
  texto R;
  do Testeo?reporte(R) → push (Buffer, R);
  □  Mantenimiento!reporte(pop (Buffer));
  od
}
```

No se puede poner en la práctica un envío como parte de la guarda

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

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

EJERCICIO 2

El proceso *Mantenimiento* debe avisar que está listo para recibir un reporte nuevo.
El proceso *Testeo* no sufre modificaciones.

Process Admin

```
{ cola Buffer;
  texto R;
  do Testeo?reporte(R) → push (Buffer, R);
  □  Mantenimiento?pedido() →
            Mantenimiento!reporte (pop (Buffer));
  od
}
```

¿Y si no había reportes pendientes? Se genera un ERROR

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);
    }
}
```

EJERCICIO 2

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);*
    }
}

EJERCICIO 3

En una empresa de software hay ***N* empleados *Testeo*** que prueban un nuevo producto para encontrar errores, cuando encuentra uno generan un reporte para que otro empleado *Mantenimiento* corrija el error y le responda. El empleado *Mantenimiento* toma los reportes para evaluarlos de acuerdo al orden de llegada, hace las correcciones necesarias y le responde al empleado *Testeo* correspondiente (el que hizo el reporte).

Vamos a partir de la solución del ejemplo 1 pero con N procesos *Testeos*

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

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

Debe comunicarse con un *Testeo* particular

¿A quien le responde?

EJERCICIO 3

Cómo no sabemos quien va a enviar un reporte para procesar, el proceso *Mantenimiento* no puede esperar recibir de uno en particular → **DEBEMOS USAR EL COMIDÍN [*] PARA RECIBIR DE CUALQUIERA DE LOS PROCESOS *Testeo***

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

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

¿A quien le responde?

En un envío no se puede poner el comodín, por lo tanto, para enviar la respuesta lo debo hacer a un *Testeo* particular. ¿A cuál? ¿Cómo se quien hizo el reporte?

EJERCICIO 3

Para saber quien hizo el reporte (quien envió el mensaje que recibí con el comodín) debe indicarse como parte del mensaje.

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

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

¿Se mantienen el orden de los pedidos en la atención?

El uso del comodín en la recepción se puede ver como una simplificación de usar el *if "guardado"* con una guarda por cada posición del arreglo de procesos *Testeo*. Cuando se va a ejecutar este tipo de recepción, si hubiese más de un proceso del arreglo que está intentando comunicarse, se selecciona no deterministicamente cuál de todos los mensajes se recibirá (sin ningún tipo de orden o prioridad) → **NO SE RESPETA EL ORDEN**

EJERCICIO 3

Debo agregar un proceso intermedio *Admin* que vaya recibiendo los reportes y los vaya almacenando en forma ordenada mientras el proceso *Mantenimiento* está procesando un reporte. Cuando este último proceso está libre (termino de procesar un reporte) le pide el siguiente al *Admin*.

Este proceso es semejante al del ejemplo 2, aunque el objetivo es diferente: en el 2 es para actuar como un BUFFER para maximizar la concurrencia, mientras que en este es para ORDENAR los pedidos actuando como una COLA.

```
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
}
```

EJERCICIO 3

```
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);
    }
}
```

EJERCICIO 4

En una empresa de software hay *N* empleados *Testeo* que prueban un nuevo producto para encontrar errores, cuando encuentra uno generan un reporte para que uno de los **3 empleados *Mantenimiento*** corrija el error y le responda. Los empleados *Mantenimiento* toma los reportes para evaluarlos de acuerdo al orden de llegada, hace las correcciones necesarias y le responde al empleado *Testeo* correspondiente (el que hizo el reporte).

Vamos a partir de la solución del ejemplo anterior pero con 3 procesos *Mantenimiento* en lugar de sólo 1.

¿Hay que hacer algún cambio?

EJERCICIO 4

```
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
}
```

¿A cuál de los 3 le responde?

¿De quien recibe el pedido?

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

¿De quien recibe la respuesta?

```
Process Mantenimiento[id: 0..2]
{ texto Rep, Res;       int idT;
  while (true)
    { Admin!pedido();
      Admin?reporte(Rep, idT);
      Res = resolver(Rep);
      Testeo[idT]!respuesta(Res);
    }
}
```

EJERCICIO 4

```
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
}
```

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

```
Process Mantenimiento[id: 0..2]
{ texto Rep, Res;       int idT;
  while (true)
    { Admin!pedido(id);
      Admin?reporte(Rep, idT);
      Res = resolver(Rep);
      Testeo[idT]!respuesta(Res);
    }
}
```