Explicación Práctica de Monitores

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/1/uc?id=1OAg5iILucHyegkF4U6z_bYqpsMNafIa9&export=download

Sintaxis – Estructura y comunicación

```
Monitor nombre {
  Variables permanentes del monitor

  Procedure uno ()
     Var locales al Procedure
     { ....
      }

  Código de inicialización del monitor
  {
    Inicialización variables.
   }
}
```

```
Process P[id: 0 .. N-1] {
   ….
   nombre.uno();
}
```

- No existen las variables compartidas.
- Las variables permanentes del monitor sólo se pueden usar dentro del monitor.
- Si el monitor tiene un código de inicialización, hasta que este no termine su ejecución el monitor no atiende llamados a los *procedures*.

- Los procesos interactúan entre ellos y con los recursos compartidos por medio de los monitores haciendo llamados a los *Procedures* de estos.
- En un procedimiento de un monitor TAMBIÉN se puede llamar a procedimientos de OTRO monitor. Pero cuidado porque el monitor desde donde se hizo el llamado se mantiene ocupado (inaccesible por otro proceso) hasta que el procedimiento llamado en el segundo monitor TERMINE POR COMPLETO su ejecución.
- Cuando el monitor está libre TODOS los procesos que están haciendo llamados a sus procedimientos compiten por acceder al monitor, NO acceden de acuerdo al orden de llegada.

Sintaxis - Sincronización

```
Monitor nombre {
  cond vc;

  Procedure uno ()
      { ....
        wait (vc);
        ....
       }

  Procedure dos ()
      { ....
        signal (vc);
        ....
        signal_all (vc);
         ....
       }
}
```

- La Exclusión Mutua es implícita dentro de un monitor al no poder ejecutar más de un llamado a un procedimiento a la vez, hasta que no se termina el *procedure* o no se duerme en una variable *condition* no se libera el monitor para atender otro llamado.
- La Sincronización por Condición es explícita por medio de variables *Conditions* usadas en los monitores. Son variables permanentes del monitor (sólo se pueden usar en el monitor que fueron declaradas):
  - *wait (vc)*: duerme al proceso en la cola asociada a la variable condición (al final de la cola).
  - *Signal (vc)*: despierta al primer proceso dormido en *vc* (al primero que se había dormido) para que compita nuevamente para acceder al monitor, y cuando lo haga continuar con la instrucción después del *wait*.
  - *Signal_all (vc)*: despierta a todos los procesos dormidos en *vc* para que todos pasen a competir por acceder nuevamente al monitor.

EJERCICIO 1

Existen N personas que desean utilizar un cajero automático. En este primer caso no se debe tener en cuenta el orden de llegada de las personas (cuando está libre cualquiera lo puede usar). Suponga que hay una función *UsarCajero()* que simula el uso del cajero.

Lo primero es definir la estructura del programa: que procesos y que monitores se usaran.
En el este caso ¿el monitor representará el Recurso Compartido (el cajero automático) o será el Administrador del Acceso al Recurso Compartido?

En este problema lo único que se debe tener en cuenta es usar el Cajero con Exclusión Mutua, no se requiere otro tipo de sincronización como por ejemplo para respetar un orden. Por lo tanto alcanza con que el monitor represente al Cajero Automático y tenga un procedimiento que simule el uso del mismo.

EJERCICIO 1

```
Monitor Cajero{

  Procedure PasarAlCajero ()
      { UsarCajero ();
       }

}


Process Persona [id: 0..N-1]
{  ….
   Cajero.PasarAlCajero();
   ….
}
```

EJERCICIO 2

Existen N personas que desean utilizar un cajero automático. En este segundo caso se debe tener en cuenta el orden de llegada de las personas. Suponga que hay una función *UsarCajero()* que simula el uso del cajero.

Partimos de la solución anterior.

```
Monitor Cajero{
  Procedure PasarAlCajero ()
      { UsarCajero ();
      }
}

Process Persona [id: 0..N-1]
{  ….
   Cajero.PasarAlCajero();
   ….
}
```

¿Se respeta el orden de llegada de las personas?

NO. Como el monitor representa el cajero, mientras el usuario lo está usando ocupa el monitor sin dejar que otro proceso pueda entrar. Cuando el usuario termina de usarlo, todos los que están esperando compiten por acceder al mismo → Se necesita que el monitor ADMINISTRE EL ACCESO AL CAJERO, y el *UsarCajero()* lo haga el proceso.

EJERCICIO 2

En este caso los clientes deben solicitar el uso del cajero, cuando le llega el turno llama a la función *UsarCajero()* que simule el uso, y luego debe avisar que salió para dejar pasar al siguiente.

```
Process Persona [id: 0..N-1]
{  Cajero.Pasar ();
   UsarCajero();
   Cajero.Salir();
}
```

¿Cómo debe implementar el monitor estos procedimientos?

Se debe tener una variable que indique el estado del recurso (si está libre o alguien lo está ocupando), porque si está libre el usuario no debe esperar, debe pasar a usarlo.

Por otro lado, si está ocupado debe esperar su turno dormido en una variable condición.

EJERCICIO 2

```
Process Persona [id: 0..N-1]
{  Cajero.Pasar ();
   UsarCajero();
   Cajero.Salir();
}
```

```
Monitor Cajero{
  bool libre = true;
  cond cola;

  Procedure Pasar ()
      { if (not libre) → wait (cola);
        libre = false;
       }

  Procedure Salir ()
      { libre = true;
        signal (cola);
       }
}
```

Si marco al cajero como libre podría entrar uno que no estaba en la cola. Y eso puede ocasionar que no se cumpla la EM (entra a usar el cajero ese proceso, y cuando accede el que fue despertado también entra).

Se debe poner como libre el cajero sólo si no hay nadie esperando en la cola

EJERCICIO 2

```
Process Persona [id: 0..N-1]
{  Cajero.Pasar ();
   UsarCajero();
   Cajero.Salir();
}
```

```
Monitor Cajero{
  bool libre = true;
  cond cola;

  Procedure Pasar ()
      { if (not libre) → wait (cola);
        libre = false;
       }

  Procedure Salir ()
      { if (empty (cola) ) → libre = true;
        signal (cola);
       }
}
```

No se puede usar la función *empty* sobre variables *condition*.

Uso un variable entera para contar cuantos procesos están dormidos en la variable *condition*.

EJERCICIO 2

```
Process Persona [id: 0..N-1]
{  Cajero.Pasar ();
   UsarCajero();
   Cajero.Salir();
}
```

```
Monitor Cajero{
  bool libre = true;
  cond cola;
  int esperando = 0;

  Procedure Pasar ()
     { if (not libre) { esperando ++;
                        wait (cola);
                      }
       else  libre = false;
     }

  Procedure Salir ()
     { if (esperando > 0 ) { esperando --;
                             signal (cola);
                           }
       else  libre = true;
      }
}
```

EJERCICIO 3

Partiendo de la solución anterior hacemos una cambio en el enunciado, agregando que si llega una persona anciana tiene prioridad.

```
Process Persona [id: 0..N-1]
{  Cajero.Pasar ();
   UsarCajero();
   Cajero.Salir();
}
```

Que debo modificar?

```
Monitor Cajero{
  bool libre = true;
  cond cola;
  int esperando = 0;

  Procedure Pasar ()
     { if (not libre) { esperando ++;
                        wait (cola);
                      }
       else  libre = false;
     }

  Procedure Salir ()
     { if (esperando > 0 ) { esperando --;
                             signal (cola);
                           }
       else  libre = true;
      }
}
```

EJERCICIO 3

Ahora si necesitamos una estructura de datos "cola ordenada" que nos permita mantener el orden (el insertar agrega al principio de la cola en caso de ser una persona anciana y sino al final). Como parámetro en el procedure Pasar enviamos la edad y el ID.

```
Process Persona [id: 0..N-1]
{  bool edad = ….;
   Cajero.Pasar (id, edad);
   UsarCajero();
   Cajero.Salir();
}
```

```
Monitor Cajero{
  bool libre = true;        cond cola;
  int idAux, esperando = 0;   colaOrdenada fila;

  Procedure Pasar (idP, edad: in int)
     { if (not libre) { insertar(fila, idP, edad);
                        esperando ++;
                        wait (cola);
                      }
       else  libre = false;
     }

  Procedure Salir ()
     { if (esperando > 0 ) {  esperando --;
                              sacar (fila, idAux);
                              signal (cola);
                           }
       else  libre = true;
      }
}
```

EJERCICIO 3

```
Process Persona [id: 0..N-1]
{  bool edad = ….;
   Cajero.Pasar (id, edad);
   UsarCajero();
   Cajero.Salir();
}
```

```
Monitor Cajero{
  bool libre = true;
  cond espera[N];
  int idAux, esperando = 0;   colaOrdenada fila;

  Procedure Pasar (idP, edad: in int)
     { if (not libre) { insertar(fila, idP, edad);
                        esperando ++;
                        wait (espera[idP]);
                      }
       else  libre = false;
     }

  Procedure Salir ()
     { if (esperando > 0 ) { esperando --;
                             sacar (fila, idAux);
                             signal (espera[idAux]);
                           }
       else  libre = true;
      }
}
```

EJERCICIO 4

En un banco hay 3 empleados. Y hay N clientes que deben ser atendidos por uno de ellos (cualquiera) de acuerdo al orden de llegada. Cuando uno de los empleados lo atiende el cliente le entrega los papeles y espera el resultado.

En este caso el cliente debe entrar al banco y esperar a que le llegue su turno, en ese momento se le indica a que empleado debe ir y debe esperar a que este lo atienda.

¿Que monitores y procesos se usaran?

Posible solución:

- **Debemos tener los procesos clientes y los empleados.**
- **Por otro lado debemos tener un monitor para administrar la entrada (ordenar a los clientes y asignarle un empleado).**
- **Para realizar la interacción entre un empleado y un cliente podemos tener un monitor para cada empleado.**

EJERCICIO 4

Primero vamos a ver la parte del problema donde los clientes llegan al banco y esperan su turno de atención. Luego vemos la interacción entre el cliente y el empleado.

El cliente llega al banco y si hay algún empleado libre NO debe esperar y directamente elige a uno de ellos para que lo atienda; sino se duerme y espera a que algún empleado que quede libre lo despierte. Para esto el cliente llama al procedimiento *Llegada* que tiene como parámetro de salida el empleado que va a atender al cliente.

```
Process Cliente[id: 0..N-1]
{  int idE;
   Banco.llegada(idE);
   ….
}
```

El empleado continuamente está atendiendo a los clientes. Cuando está libre avisa al banco que espera un próximo cliente llamando al procedimiento *Próximo* enviando como parámetro su identificador. Si hay algún cliente esperando lo despierta al primero que está en la cola.

```
Process Empleado[id: 0..2]
{  while (true)
      { Banco.próximo(id);
        ….
      }
}
```

EJERCICIO 4

Vamos a usar una cola donde guardaremos los identificadores de los empleados que están libres llamada *elibres*. También usaremos una variable *condition* llamada *esperaC* donde se demoran los clientes cuando no hay empleados libres, y un entero *esperando* para contar la cantidad de clientes que están dormidos (como se vio en el ejercicio 2).

```
Monitor Banco {
  cola elibres;
  cond esperaC;
  int esperando = 0;

  Procedure Llegada(idE: out int)
     { if (empty(elibres)) { esperando ++;
                             wait (esperaC);
                           }
       pop(elibres, idE);
     }

  Procedure Próximo(idE: in int)
     {  push(elibres, idE);
        if (esperando > 0 ) { esperando --;
                              signal (esperaC);
                            }
     }
}
```

Al modificar la cola *elibres* agregando al empleado libre, cualquier cliente que no estaba dormido puede verlo, pasar y sacar a ese empleado de la cola (con la posibilidad de dejarla vacía). Cuando el que fue despertado accede al monitor hace el pop de una cola vacía → NO SE RESPETA EL ORDEN Y DA ERROR

EJERCICIO 4

Para evitar el problema anterior se debe usar *passing the condition*, pero el empleado debe insertar su ID en la cola haya o no clientes dormidos → la condición a chequear por el cliente no debe ser la cola *elibres*.

Agregamos un entero *cantLibres* que llevará la cuenta de los empleados que están "realmente" libres (llamaron al procedimiento *próximo* cuando no había nadie esperando). No siempre su valor será equivalente a la cantidad de elementos de *elibres*.

Esta nueva variable será sobre la que se hace el *passing the condition*.

```
Monitor Banco {
  cola elibres;
  cond esperaC;
  int esperando = 0, cantLibres = 0;

  Procedure Llegada(idE: out int)
     { if (cantLibres == 0) { esperando ++;
                              wait (esperaC);
                            }
       else cantLibres--;
       pop(elibres, idE);
     }

  Procedure Próximo(idE: in int)
     {  push(elibres, idE);
        if (esperando > 0 ) { esperando --;
                              signal (esperaC);
                            }
        else cantLibres++;
     }
}
```

EJERCICIO 4

Ahora vemos la interacción entre el cliente y el empleado que lo debe atender. Para esto tendremos un monitor *escritorio* para cada empleado que es donde interactúan

El cliente, cuando ya sabe a que empleado ir se dirige a su escritorio, le entrega los papeles y espera a que el empleado le retorne los resultados por medio del procedimiento *atención*.

```
Process Cliente[id: 0..N-1]
{  int idE;
   text papel, res;

   Banco.llegada(idE);
   Escritorio[idE].atención(papel, res);
}
```

El empleado, después de avisar que está libre, se dirige a su escritorio a esperar a un cliente llamando al procedimiento *esperarDatos* con un parámetro de salida donde recibe los papeles del cliente.

Resuelve la solicitud y le envía los resultados al cliente por medio del procedimiento *enviarResultado*, y espera a que el cliente tome los resultados y se vaya para poder atender a otro.

```
Process Empleado[id: 0..2]
{  text datos;

   while (true)
      { Banco.próximo(id);
        Escritorio[id].esperarDatos(datos);
        res = resolver solicitud en base a datos
        Escritorio[id].enviarResultado(res);
      }
}
```

EJERCICIO 4

En cada escritorio debemos tener una variable para que el cliente le comunique al empleado los papeles o datos y otra para comunicarse los resultados en el sentido contrario.

Además el cliente debe esperar hasta obtener los resultados en una variable condición *vcCliente*. El empleado debe esperar los datos en otra variable condición *vcEmpleado*, y luego debe esperar a que el cliente haya tomado los resultados, para lo cual podemos usar esa misma variable condición.

```
Monitor Escritorio[id: 0..2] {
  cond vcCliente, vcEmpleado;
  text datos, resultados;

  Procedure Atención(D: in text; R: out text)
     { datos = D;
       signal (vcEmpleado);
       wait (vcCliente);
       R = resultados;
       signal (vcEmpleado);
     }

  Procedure Esperardatos(D: out text)
     {  wait (vcEmpleado);
        D = datos;
     }

  Procedure EnviarResultados(R: in text)
     {  resultados = R;
        signal (vcCliente);
        wait (vcEmpleado);
     }
}
```

Avisa que ya dejo los datos

Debe esperar los resultados

Avisa que ya tomó los datos

Espera que llegue el cliente

Y si el cliente ya había llegado

Avisa que están los resultados

Debe esperar que el cliente tome los resultados

EJERCICIO 4

Si el cliente llega al escritorio antes que el empleado entonces el empleado se duerme y nadie lo despierta (el cliente ya hizo el *signal* y no tiene efecto posterior) → DEADLOCK.

Debo usar una variable booleana *listo* que me indique si el cliente ya llegó, y sólo dormirse si *listo* es falso.

```
Monitor Escritorio[id: 0..2] {
  cond vcCliente, vcEmpleado;
  text datos, resultados;
  boolean listo = false;

  Procedure Atención(D: in text; R: out text)
     { datos = D;
       listo = true;
       signal (vcEmpleado);
       wait (vcCliente);
       R = resultados;
       signal (vcEmpleado);
     }

  Procedure Esperardatos(D: out text)
     { if (not listo) wait (vcEmpleado);
       D = datos;
     }

  Procedure EnviarResultados(R: in text)
     { resultados = R;
       signal (vcCliente);
       wait (vcEmpleado);
       listo = false;
     }
}
```

Marca que llegó el cliente

Sólo se duerme si el cliente no llegó.

Resetea para atender al próximo cliente

EJERCICIO 4

```
Process Cliente[id: 0..N-1]
{  int idE;
   text papel, res;
   Banco.llegada(idE);
   Escritorio[idE].atención(papel, res);
}
```

```
Process Empleado[id: 0..2]
{ text datos;
  while (true)
   { Banco.próximo(id);
     Escritorio[id].esperarDatos(datos);
     res = resolver solicitud en base a datos
     Escritorio[id].enviarResultado(res);
   }
}
```

```
Monitor Banco {
  cola elibres;
  cond esperaC;
  int esperando = 0, cantLibres = 0;

  Procedure Llegada(idE: out int)
     { if (cantLibres == 0)
              { esperando ++;
                wait (esperaC);
              }
       else cantLibres--;
       pop(elibres, idE);
     }

  Procedure Próximo(idE: in int)
     {  push(elibres, idE);
        if (esperando > 0 )
              { esperando --;
                signal (esperaC);
              }
        else cantLibres++;
      }
}
```

```
Monitor Escritorio[id: 0..2] {
  cond vcCliente, vcEmpleado;
  text datos, resultados;
  boolean listo = false;

  Procedure Atención(D: in text; R: out text)
     { datos = D;
       listo = true;
       signal (vcEmpleado);
       wait (vcCliente);
       R = resultados;
       signal (vcEmpleado);
     }

  Procedure Esperardatos(D: out text)
     {  if (not listo) wait (vcEmpleado);
        D = datos;
     }

  Procedure EnviarResultados(R: in text)
     {  resultados = R;
        signal (vcCliente);
        wait (vcEmpleado);
        listo = false;
      }
}
```

EJERCICIO 5

Se debe simular un partido de fútbol 11. Cuando los 22 jugadores llegaron a la cancha juegan durante 90 minutos y luego todos se retiran.

En este caso se debe hacer una barrera hasta que llegan los 22 jugadores, y luego TODOS JUNTOS AL MISMO TIEMPO deben jugar durante 90 minutos.

Tenemos un contador que se incrementa al llegar cada jugador y se duermen en una variable condición hasta que llega el último y los despierta para jugar.

```
Process Jugador[id: 0..21]
{  Cancha.llegada();
   delay (90minutos); //juega el partido
}
```

Cada jugador juega su propio partido posiblemente en diferentes momentos

```
Monitor Cancha
{  int cant = 0;
   cond espera;

   Procedure llegada ()
     { cant ++;
       if (cant < 22) wait (espera)
       else signal_all (espera);
     }
}
```

EJERCICIO 5

El *delay* que representa que todos están jugando al mismo tiempo el partido se debe hacer en un único lugar. Podría ser en el monitor, cuando llega el último jugador (antes del signal_all). O bien usar otro proceso que simula el partido y se duerme hasta que todos llegan; luego hace el *delay* y despierta a todos para que se vayan.

```
Process Jugador[id: 0..21]
{  Cancha.llegada();
}
```

```
Process Partido
{  Cancha. Iniciar();
   delay (90minutos); //se juega el partido
   Cancha.Terminar();
}
```

```
Monitor Cancha
{  int cant = 0;
   cond espera, inicio;
   Procedure llegada ()
     { cant ++;
       if (cant < 22) wait (espera)
       else signal (inicio);
     }
   Procedure Iniciar ()
     { if (cant < 22) wait (inicio);
     }
   Procedure Terminar ()
     { signal_all(espera);
     }
}
```

El último en llegar despierta al *partido*.

Sólo se duerme si no han llegado todos.

Despierta a todos para que se vayan.

EJERCICIO 5

```
Process Jugador[id: 0..21]
{ Cancha.llegada();
}
```

```
Process Partido
{ Cancha. Iniciar();
  delay (90minutos); //se juega el partido
  Cancha.Terminar();
}
```

```
Monitor Cancha
{ int cant = 0;
  cond espera, inicio;
  Procedure llegada ()
    { cant ++;
      if (cant < 22) wait (espera)
      else signal (inicio);
    }
  Procedure Iniciar ()
    { if (cant < 22) wait (inicio);
    }
  Procedure Terminar ()
    { signal_all(espera);
    }
}
```

Ese Jugador se ira sin Jugar

El último jugador en llegar "Inicia" el partido, pero no se duerme (se va sin jugar). Debemos hacer que él también se duerma en *espera,* al igual que el resto de los jugadores.

EJERCICIO 5

```
Process Jugador[id: 0..21]
{  Cancha.llegada();
}
```

```
Process Partido
{  Cancha. Iniciar();
   delay (90minutos); //se juega el partido
   Cancha.Terminar();
}
```

```
Monitor Cancha
{  int cant = 0;
   cond espera, inicio;
   Procedure llegada ()
     { cant ++;
       if (cant == 22) signal (inicio);
       wait (espera);
     }
   Procedure Iniciar ()
     { if (cant < 22) wait (inicio);
     }
   Procedure Terminar ()
     { signal_all(espera);
     }
}
```

EJERCICIO 6

En una empresa de genética hay N clientes que envían secuencias de ADN para que sean analizadas y esperan los resultados para poder continuar. Para resolver estos análisis la empresa cuenta con un servidor que resuelve los pedidos de acuerdo al orden de llegada de los mismos.

Se necesitan los *N* procesos *Cliente* para enviar los pedidos y recibir los resultados, y un *Servidor* para resolverlos

```
Process Cliente [id: 0..N-1]
{ text S, res;
  while (true)
    { --generar secuencia S
      Servidor.Pedido(S, res);
    }
}
```

Mientras el servidos atiende un pedido los clientes no pueden hacer otros pedidos. ¿Cómo se mantiene el orden

```
Monitor Servidor {
    Procedure Pedido(S: in text; R: out text)
      { R = AnalizarSec(S);
      }
}
```

EJERCICIO 6

La resolución del pedido no se debe hacer dentro del monitor para que mientras se resuelve otros clientes puedan hacer nuevos pedidos que se almacenen ordenados.

El *Servidor* debe ser un proceso. Se necesita un monitor *Admin* para almacenar los pedidos y comunicar los resultados

```
Process Cliente [id: 0..N-1]
{ text S, res;
  while (true)
    { --generar secuencia S
      Admin.Pedido(S, res);
    }
}
```

```
Process Servidor
{ text sec, res;
  while (true)
    { Admin.Sig(sec);
      res = AnalizarSec(sec);
      Admin.Resultado(res);
    }
}
```

```
Monitor Admin {
  Cola C;
  Cond espera;
  text res;

  Procedure Pedido (S: in text; R: out text)
    { push (C, S);
      wait (espera);
      R = res;
    }

  Procedure Sig (S: out text)
    { pop (C, S);
    }

  Procedure Resultado (R: in text)
    { res = R;
      signal (espera);
    }
}
```

Se puede sobrescribir *res*.

EJERCICIO 6

Usamos un arreglo donde dejar los resultados para que el servidor no deba esperar a que el resultado sea tomado para poder resolver otro pedido. Para eso se necesita saber quien hizo el pedido.

```
Process Cliente [id: 0..N-1]
{ text S, res;
  while (true)
    { --generar secuencia S
      Admin.Pedido(id, S, res);
    }
}
```

```
Process Servidor
{ text sec, res;
  int aux;
  while (true)
    { Admin.Sig(aux, sec);
      res = AnalizarSec(sec);
      Admin.Resultado(aux, res);
    }
}
```

```
Monitor Admin {
  Cola C;
  Cond espera;
  text res[N];

  Procedure Pedido (IdC: in int; S: in text; R: out text)
    { push (C, (idC,S) );
      wait (espera);
      R = res[idC];
    }

  Procedure Sig (IdC: out int; S: out text)
    { pop (C, (IdC, S));
    }

  Procedure Resultado (IdC: in int; R: in text)
    { res[IdC] = R;
      signal (espera);
    }
}
```

¿Y si no hay pedido pendientes?

EJERCICIO 6

Debo demorar al proceso servidor hasta que haya algún pedido en la cola.

```
Process Cliente [id: 0..N-1]
{ text S, res;
  while (true)
    { --generar secuencia S
      Admin.Pedido(id, S, res);
    }
}
```

```
Process Servidor
{ text sec, res;
  int aux;
  while (true)
    { Admin.Sig(aux, sec);
      res = AnalizarSec(sec);
      Admin.Resultado(aux, res);
    }
}
```

```
Monitor Admin {
  Cola C;
  Cond espera;
  Cond HayPedido;
  text res[N];

  Procedure Pedido (IdC: in int; S: in text; R: out text)
   { push (C, (idC,S) );
     signal (HayPedido);
     wait (espera);
     R = res[idC];
   }

  Procedure Sig (IdC: out int; S: out text)
   { if (empty (C)) wait (HayPedido);
     pop (C, (IdC, S));
   }

  Procedure Resultado (IdC: in int; R: in text)
   { res[IdC] = R;
     signal (espera);
   }
}
```

EJERCICIO 7

Modificamos el enunciado para que haya 2 servidores en lugar de 1. Y partimos de la solución anterior.

```
Process Cliente [id: 0..N-1]
{ text S, res;
  while (true)
    { --generar secuencia S
      Admin.Pedido(id, S, res);
    }
}
```

```
Process Servidor [id: 0..1]
{ text sec, res;
  int aux;
  while (true)
    { Admin.Sig(aux, sec);
      res = AnalizarSec(sec);
      Admin.Resultado(aux, res);
    }
}
```

```
Monitor Admin {
  Cola C;
  Cond espera;
  Cond HayPedido;
  text res[N];

  Procedure Pedido (IdC: in int; S: in text; R: out text)
    { push (C, (idC,S) );
      signal (HayPedido);
      wait (espera);
      R = res[idC];
    }

  Procedure Sig (IdC: out int; S: out text)
    { if (empty (C)) wait (HayPedido);
      pop (C, (IdC, S));
    }

  Procedure Resultado (IdC: in int; R: in text)
    { res[IdC] = R;
      signal (espera);
    }
}
```

Cuando acceda nuevamente al monitor el otro podría haber vaciado nuevamente la cola

EJERCICIO 7

Se debe re chequear la condición.

```
Process Cliente [id: 0..N-1]
{ text S, res;
  while (true)
    { --generar secuencia S
      Admin.Pedido(id, S, res);
    }
}
```

```
Process Servidor [id: 0..1]
{ text sec, res;
  int aux;
  while (true)
    { Admin.Sig(aux, sec);
      res = AnalizarSec(sec);
      Admin.Resultado(aux, res);
    }
}
```

```
Monitor Admin {
  Cola C;
  Cond espera;
  Cond HayPedido;
  text res[N];

  Procedure Pedido (IdC: in int; S: in text; R: out text)
    { push (C, (idC,S) );
      signal (HayPedido);
      wait (espera);
      R = res[idC];
    }

  Procedure Sig (IdC: out int; S: out text)
    { while (empty (C)) wait (HayPedido);
      pop (C, (IdC, S));
    }

  Procedure Resultado (IdC: in int; R: in text)
    { res[IdC] = R;
      signal (espera);
    }
}
```

Al ser dos servidores podría ser que un pedido que llegó antes se termine de resolver antes.

EJERCICIO 7

Se necesita usar variables *condition* privadas.

```
Process Cliente [id: 0..N-1]
{ text S, res;
  while (true)
    { --generar secuencia S
      Admin.Pedido(id, S, res);
    }
}
```

```
Process Servidor [id: 0..1]
{ text sec, res;
  int aux;
  while (true)
    { Admin.Sig(aux, sec);
      res = AnalizarSec(sec);
      Admin.Resultado(aux, res);
    }
}
```

```
Monitor Admin {
  Cola C;
  Cond espera[N];
  Cond HayPedido;
  text res[N];

  Procedure Pedido (IdC: in int; S: in text; R: out text)
    { push (C, (IdC,S) );
      signal (HayPedido);
      wait (espera[IdC]);
      R = res[IdC];
    }

  Procedure Sig (IdC: out int; S: out text)
    { while (empty (C)) wait (HayPedido);
      pop (C, (IdC, S));
    }

  Procedure Resultado (IdC: in int; R: in text)
    { res[IdC] = R;
      signal (espera[IdC]);
    }
}
```