Explicación Práctica de Monitores

Ejercicios

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();
}
```

¿Qué pasa si el orden a utilizar no es el de llegada?

Usar una estructura de datos tipo cola para mantener el ID de los procesos según el orden requerido. Y usar variables *condition* privadas.
Cómo el ejemplo de la teoría de la alocación SJN.

```
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 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 quién 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 pedidos 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 después.

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]);
    }
}
```

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) signal (inicio);
       wait (espera);
      }
   Procedure Iniciar ()
     { if (cant < 22) wait (inicio);
     }
   Procedure Terminar ()
     { signal_all(espera);
     }
}
```

EJERCICIO de la práctica

En un entrenamiento de futbol hay 20 jugadores que forman 4 equipos (cada jugador conoce el equipo al cual pertenece llamando a la función DarEquipo()). Cuando un equipo está listo (han llegado los 5 jugadores que lo componen), debe enfrentarse a otro equipo que también esté listo (los dos primeros equipos en juntarse juegan en la cancha 1, y los otros dos equipos juegan en la cancha 2). Una vez que el equipo conoce la cancha en la que juega, sus jugadores se dirigen a ella. Cuando los 10 jugadores del partido llegaron a la cancha comienza el partido, juegan durante 50 minutos, y al terminar todos los jugadores del partido se retiran (no es necesario que se esperen para salir).

EJERCICIO de la práctica

Process Jugador[id: 0..19]
{ Cancha.llegada();
}

¿Como sabe a que cancha ir?

Process Partido [id: 0..1]
{ Cancha[id].Iniciar();
delay (*90minutos*); *//se juega el partido*
Cancha[id].Terminar();
}

Monitor Cancha [id: 0..1]
{ int cant = 0;
cond espera, inicio;

Procedure llegada ()
{ cant ++;
if (cant == 10) signal (inicio);
wait (espera);
}

Procedure Iniciar ()
{ if (cant < 10) wait (inicio);
}

Procedure Terminar ()
{ signal_all(espera);
}
}

EJERCICIO de la práctica

Primero cada equipo se debe juntar (independientemente del resto), y luego debe ver a que cancha ir. Se necesita un monitor para cada equipo.

```
Process Jugador[id: 0..19]
{  int miEquipo = …;
   int numeroC;

   Equipo[miEquipo].llegada(numeroC);
}
```

```
Monitor Equipo [id: 0..3]
{  int cant = 0;
   cond espera;
   Procedure llegada (cancha: OUT int)
     { cant ++;
       if (cant < 4) wait (espera)
       else  {  cancha = Determinar la cancha
                signal_all (espera);
             }
      }
}
```

¿Cómo se hace, porque no depende sólo de este grupo?

EJERCICIO de la práctica

Se requiere un monitor para administrar las canchas

```
Process Jugador[id: 0..19]
{  int miEquipo = …;
   int numeroC;

   Equipo[miEquipo].llegada(numeroC);
}
```

```
Monitor Admin
{  int cant = 0;
   Procedure DarCancha (cancha: OUT int)
     { cant ++;
       if (cant <= 2) cancha = 0
       else cancha = 1;
     }
}
```

```
Monitor Equipo [id: 0..3]
{  int cant = 0;
   cond espera;
   Procedure llegada (cancha: OUT int)
     { cant ++;
       if (cant < 4) wait (espera)
       else  {  Admin.DarCancha(cancha);
                signal_all (espera);
             }
     }
}
```

Sólo lo conocerá uno de los integrantes del grupo (el último en llegar)

EJERCICIO de la práctica

```
Process Jugador[id: 0..19]
{  int miEquipo = …;
   int numeroC;

   Equipo[miEquipo].llegada(numeroC);
}
```

```
Monitor Admin
{  int cant = 0;
   Procedure DarCancha (cancha: OUT int)
    { cant ++;
      if (cant <= 2) cancha = 0
      else cancha = 1;
     }
}
```

```
Monitor Equipo [id: 0..3]
{  int cant = 0, numCancha;
   cond espera;
   Procedure llegada (cancha: OUT int)
     { cant ++;
       if (cant < 4) wait (espera)
       else  {  Admin.DarCancha(numCancha);
                signal_all (espera);
              };
       cancha = numCancha;
     }
}
```

Ahora si cada persona puede ir a la cancha que le corresponde y jugar el partido.

EJERCICIO de la práctica

```
Process Jugador[id: 0..19]
{  int miEquipo = …;
   int numeroC;

   Equipo[miEquipo].llegada(numeroC);
   Cancha[numeroC].llegada();
}
```

```
Process Partido [id: 0..1]
{  Cancha[id].Iniciar();
   delay (90minutos); //se juega el partido
   Cancha[id].Terminar();
}
```

```
Monitor Cancha [id: 0..1]
{  int cant = 0;
   cond espera, inicio;
   Procedure llegada ()
     { cant ++;
       if (cant == 10) signal (inicio);
       wait (espera);
     }
   Procedure Iniciar ()
     { if (cant < 10) wait (inicio);
     }
   Procedure Terminar ()
     { signal_all(espera);
     }
}
```

EJERCICIO de la práctica

```
Process Jugador[id: 0..19]
{ int miEquipo = …;
  int numeroC;
  Equipo[miEquipo].llegada(numeroC);
  Cancha[numeroC].llegada();
}
```

```
Monitor Equipo [id: 0..3]
{ int cant = 0, numCancha;
  cond espera;
  Procedure llegada (cancha: OUT int)
    { cant ++;
      if (cant < 4) wait (espera)
      else { Admin.DarCancha(numCancha);
             signal_all (espera);
           };
      cancha = numCancha;
    }
}
```

```
Process Partido [id: 0..1]
{ Cancha[id].Iniciar();
  delay (90minutos); //se juega el partido
  Cancha[id].Terminar();
}
```

```
Monitor Admin
{ int cant = 0;
  Procedure DarCancha (num: OUT int)
    { cant ++;
      if (cant <= 2) num = 0
      else num = 1;
    }
}
```

```
Monitor Cancha [id: 0..1]
{ int cant = 0;
  cond espera, inicio;
  Procedure llegada ()
    { cant ++;
      if (cant == 10) signal (inicio);
      wait (espera);
    }
  Procedure Iniciar ()
    { if (cant < 10) wait (inicio);
    }
  Procedure Terminar ()
    { signal_all(espera);
    }
}
```