Tema 3: Sincronización y Comunicación de Procesos

1. Introducción
2. Exclusión mutua
3. Semáforos
4. Monitores
5. Mensajes

Bibliografía

- W. Stalling. **Sistemas Operativos**. 4ª Edición. Capítulo 5.
- A.S. Tanenmbaum. **Sistemas Operativos Modernos**. Temas 2, 11 y 12.

Introducción

- Multiprogramación, multiprocesamiento y procesamiento distribuido
- Necesidad de sincronizar y comunicar procesos
- Métodos básicos de comunicar procesos
  - Compartición de datos
  - Intercambio de información

Exclusión mutua

- Se denomina **Sección Crítica (SC)** de un proceso a aquellas partes de su código que no pueden ejecutarse de forma concurrente
- Agrupación en clases de secciones críticas
- **Protocolo**: Código dedicado a asegurar que la sección crítica se ejecuta de forma exclusiva
- Requisitos resolver para la exclusión mutua:
  - Sólo un proceso debe tener permiso para entrar en la SC en un momento dado.
  - Cuando se interrumpe un proceso en una región no crítica no debe interferir el resto de procesos
  - No puede demorarse un proceso indefinidamente en una sección crítica.
  - Cuando ningún proceso está en su SC, cualquier proceso que solicite entrar debe hacerlo sin dilación
  - No se deben hacer suposiciones sobre la velocidad relativa de los procesos ni el número de procesadores
  - Un proceso permanece en su SC sólo por un tiempo finito.

Exclusión mutua: Soluciones por Software

- La responsabilidad de mantener la Exclusión Mutua recae sobre los procesos.
- Es necesaria una memoria principal compartida accesible a todos los procesos.
- Existe una exclusión mutua elemental en el acceso a la memoria.
- Algoritmo de Peterson
- Algoritmo de Dekker
- Inconvenientes:
  - La espera de acceso a un recurso se realiza de forma *ocupada*.
  - Presentan dificultades antes una cantidad elevada de procesos concurrentes.

Soluciones por Software: Algoritmo de Peterson

```
booleano señal[2];
int turno;
```

```
void P0()
{
  while (true)
  {
    . . .
    señal[0] = true;
    turno = 1;
    while (señal[1] && turno==1);
    /*Sección Crítica*/
    señal[0] = false;
    . . .
  }
}
```

```
void P1()
{
  while (true)
  {
    . . .
    señal[1] = true;
    turno = 0;
    while (señal[0] && turno==0);
    /*Sección Crítica*/
    señal[1] = false;
    . . .
  }
}
```

```
void main()
{
    señal[0] = false;
    señal[1] = false;
    cobegin
      P0();P1();
    coend;
}
```

Soluciones por Software: Algoritmo de Dekker

```
booleano señal[2];
int turno;
```

```
void P0()
{
  while (true)
  {
    . . .
    señal[0] = true;
    while (señal[1])
        if (turno ==1)
        {
           señal[0] = false;
           while (turno == 1);
           señal[0] = true;
         }
      /*Sección Crítica*/
     turno = 1;
     señal[0] = falso;
    . . .
  }
}
```

```
void P1()
{
  while (true)
  {
    . . .
    señal[1] = true;
    while (señal[0])
        if (turno ==0)
        {
           señal[1] = false;
           while (turno == 0);
           señal[1] = true;
         }
      /*Sección Crítica*/
     turno = 0;
     señal[1] = falso;
    . . .
  }
}
```

```
void main()
{
   señal[0] = false;  señal[1] = false; turno = 1;
   cobegin
     P0();P1();
   coend;
}
```

Exclusión mutua: Soluciones por Hardware

- Inhabilitación de interrupciones
  - Sistemas monoprocesador. Sólo aplicable a nivel de núcleo

```
while (true)
{
    . . .
    Inhabilitar interrupciones;
    /*Sección crítica*/
    Habilitar interrupciones;
    . . .
}
```

  - Se degrada la eficiencia del procesador
- Instrucciones especiales de máquina
  - Se realizan varias acciones atómicamente: leer y escribir, leer y examinar, …
  - No están sujetas a interferencias de otras instrucciones

Semáforos

- Tipo Abstracto de Datos
- Datos:
  - Contador entero
  - Cola de procesos en espera
- Operaciones:
  - **Inicializar**: Inicia el contador a un valor no negativo
  - **P()**: Disminuye en una unidad el valor del contador. Si el contador se hace negativo, el proceso que ejecuta P se bloquea.
  - **V()**: Aumenta en una unidad el valor del contador. Si el valor del contador no es positivo, se desbloquea un proceso bloqueado por una operación P.
- Las operaciones son atómicas a nivel hardware
- Se denomina **semáforo binario** aquel en el que el contador sólo toma valor 0 ó 1.
- El proceso que espera entrar en la SC no usa el procesador, está bloqueado.

Semáforo general: definición de primitivas

```
struct TSemáforo
{
    int contador;
    TColaProcesos Cola;
}

void inicializar(TSemáforo s, int n)
{
  s.contador=n;
}

void P(TSemáforo s)
{
  s.contador--;
  if (s.contador<0)
  {
    poner este proceso en s.cola;
    bloquear este proceso;
  }
}

void V(TSemáforo s)
{
  s.contador++;
  if (s.contador<=0)
  {
     quitar un proceso p de s.cola;
     poner el proceso p en la cola de listos;
  }
}
```

Semáforo binario: definición de primitivas

```
struct TSemáforo_bin
{
    int contador;
    TColaProcesos cola;
}

void inicializar_B(TSemáforo_bin s, int n)
{
  s.contador=n;
}

void P_B(TSemáforo_bin s)
{
  if (s.contador == 1) s.contador = 0;
  else
  {
    poner este proceso en s.cola;
    bloquear este proceso;
  }
}

void V_B(TSemáforo_bin s)
{
  if (s.cola.esvacia()) s.contador = 1;
  else
  {
     quitar un proceso p de s.cola;
     poner el proceso p en la cola de listos;
  }
}
```

Semáforos: exclusión mutua

- El valor asignado al contador indicará la cantidad de procesos que pueden ejecutar concurrentemente la sección crítica
- Los semáforos se deben inicializar antes de comenzar la ejecución concurrente de los procesos.

```
TSemáforo s;
void Pi();
{
   while (true)
   {
      . . .
      P(s);
      sección crítica;
      V(s)
      . . .
   }
}

void main
{
   inicializar(s, 1)
   cobegin
      P1(); P2(); … ; Pn();
   coend
}
```

Semáforos: sincronización entre procesos

- El uso de semáforos permite la sincronización entre procesos
- Problema del **productor - consumidor**

Uno o más productores generan cierto tipo de datos y los sitúan en una zona de memoria o buffer. Un único consumidor saca elementos del buffer de uno en uno. El sistema debe impedir la superposición de operaciones sobre el buffer.

Solución: Tamaño de buffer ilimitado

```
TSemáforo s, n;

void productor()
{
   while (true)
   {
      producir();
      P(s);
      añadir_buffer();
      V(s);
      V(n);
   }
}

void consumidor()
{
   while (true)
   {
      P(n);
      P(s);
      coger_ buffer();
      V(s);
      consumir();
   }
}

void main()
{
   inicializar(s, 1);   inicializar(n, 0);
   cobegin
      productor();
      consumidor();
   coend;
}
```

Semáforos: sincronización entre procesos

Solución: Tamaño de buffer limitado

```
#define tamaño_buffer N
TSemáforo e, s, n;

void productor()
{
   while (true)
   {
      producir();
      P(e);
      P(s);
      añadir_buffer();
      V(s);
      V(n);
   }
}

void consumidor()
{
   while (true)
   {
      P(n);
      P(s);
      coger_ buffer();
      V(s);
      V(e);
      consumir();
   }
}

void main()
{
   inicializar(s, 1);
   inicializar(n, 0);
   inicializar(e, tamaño_buffer);
   cobegin
      productor();
      consumidor();
   coend;
}
```

Semáforos: sincronización entre procesos

Problema de los **lectores y escritores**

Se dispone de una zona de memoria o fichero a la que acceden unos procesos (lectores) en modo lectura y otros procesos en modo escritura (escritores).

- Los lectores pueden acceder al fichero de forma concurrente.
- Los escritores deben acceder al fichero de manera exclusiva entre ellos y con los lectores.

El sistema debe coordinar el acceso a la memoria o al fichero para que se cumplan las restricciones.

Solución: Prioridad a los lectores

```
TSemáforo mutex, w;
int lectores;

void escritor_i()
{
    . . .
    P(w);
    escribir();
    V(w);
    . . .
}

void lector_j()
{
    . . .
    P(mutex);
    lectores++;
    if (lectores==1) P(w);
    V(mutex)
    leer();
    P(mutex);
    lectores--;
    if (lectores==0) V(w);
    V(mutex)
    . . .
}

void main()
{
   inicializar(mutex, 1);
   inicializar(w, 1);
   lectores=0;
   cobegin
      escritor_1();…; escritor_n(); lector_1(); … ; lector_m();
   coend;
}
```

Semáforos: sincronización entre procesos

Solución: Prioridad a los escritores

```
TSemáforo mutex1, mutex2, w, r;
int lectores, escritores;
```

```
void lector_i()
{
    . . .
    P(r);
    P(mutex1);
    lectores++;
    if (lectores==1) P(w);
    V(mutex1);
    V(r);
    leer();
    P(mutex1);
    lectores--;
    if (lectores==0) V(w);
    V(mutex1);
    . . .
}
```

```
void escritor_j()
{
    . . .
    P(mutex2);
    escritores++;
    if (escritores==1) P(r);
    V(mutex2);
    P(w);
    escribir();
    V(w);
    P(mutex2);
    escritores--;
    if (escritores==0) V(r);
    V(mutex2)
    . . .
}
```

```
void main()
{
  inicializar(mutex1, 1); inicializar(mutex2, 1);
  inicializar(w, 1); inicializar(r, 1);
  lectores = 0; escritores = 0;
  cobegin
    escritor_1();…; escritor_n(); lector_1(); … ; lector_m();
  coend;
}
```

Semáforos: sincronización entre procesos

Solución: Acceso según orden de llegada

```
TSemáforo mutex, fifo,  w;
int lectores;
```

```
void lector_i()
{
    . . .
    P(fifo);
    P(mutex);
    lectores++;
    if (lectores==1) P(w);
    V(mutex);
    V(fifo);
    leer();
    P(mutex);
    lectores--;
    if (lectores==0) V(w);
    V(mutex);
    . . .
}
```

```
void escritor_j()
{
    . . .
    P(fifo);
    P(w);
    V(fifo);
    escribir();
    V(w);
     . . .
}
```

```
void main()
{
   inicializar(mutex, 1); inicializar(fifo, 1); inicializar(w, 1);
   lectores = 0;
   cobegin
      escritor_1();…; escritor_n(); lector_1(); … ; lector_m();
   coend;
}
```

Semáforos: sincronización entre procesos

- Problema del **barbero dormilón**

Una barbería tiene una sala de espera con **n** sillas, y una habitación con un sillón donde se atiende a los clientes. Si no hay clientes el barbero se duerme. Si un cliente entra en la barbería y todas las sillas están ocupadas, entonces se va, sino, se sienta en una de las sillas disponibles .Si el barbero está dormido, el cliente lo despertará.

El sistema debe coordinar el barbero y los clientes.

```
#define sillas n
TSemáforo mutex, clientes, barbero;
int espera;
```

```
void barbero()
{
   while (true)
   {
      P(clientes);
      P(mutex);
      espera=espera-1;
      V(barbero);
      V(mutex);
      cortar_pelo();
   }
}
```

```
void cliente_i()
{
      P(mutex);
      if (espera<sillas)
      {
        espera=espera+1;
        V(clientes);
        V(mutex);
        P(barbero);
        se_corta_pelo();
      }
      else V(mutex);
}
```

```
void main()
{  inicializar(mutex, 1); inicializar(clientes, 0);
   inicializar(barbero, 0);  espera=0;
    cobegin
      barbero();
      cliente_1(); cliente_2(); … cliente_m();
   coend;
}
```

Semáforos: sincronización entre procesos

Problema de **los cinco filósofos**

Cinco filósofos se dedican a pensar y a comer en una mesa circular. En el centro de la mesa hay un cuenco con arroz, y la mesa está puesta con cinco platos y cinco palillos, uno por cada filósofo. Cuando un filósofo tiene hambre se sienta en la mesa a comer en su sitio. El filósofo sólo puede coger un palillo cada vez y no le puede quitar un palillo a un compañero que lo tenga en la mano. Cuando un filósofo tiene los dos palillos come sin soltarlos hasta que termine y vuelve a pensar.

El sistema debe coordinar los filósofos para evitar la espera indefinida y no se mueran de hambre.

Solución que mantiene la exclusión mutua

```
TSemáforo palillo[5];
void filósofo(int i)
{
    while (true)
    {
       pensar();
       P(palillo[i]);
       P(palillo[(i+1)%5]);
       comer();
       V(palillo[i]);
       V(palillo[(i+1)%5]);
    }
}
void main()
{  int i;
   for (i=0; i<5; i++) inicializar(palillo[i], 1);
   cobegin
      filósofo(0); filósofo(1); … filósofo(4);
   coend;
}
```

Se produce interbloqueo cuando acuden a comer todos a la vez

Semáforos: sincronización entre procesos

- Problema de **los cinco filósofos**

Solución que evita los interbloqueos y mantiene la exclusión mutua

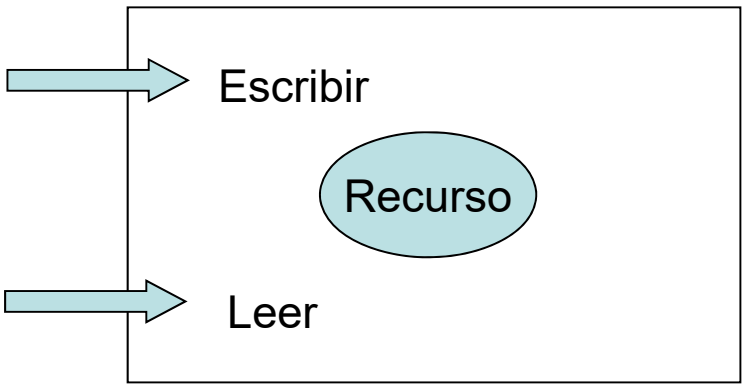
```
TSemáforo palillo[5], silla;

void filósofo(int i)
{
    while (true)
    {
       pensar();
       P(silla);
       P(palillo[i]);
       P(palillo[(i+1)%5]);
       V(silla);
       comer();
       V(palillo[i]);
       V(palillo[(i+1)%5]);
    }
}
void main()
{
   int i;
   for (i=0; i<5; i++) inicializar(palillo[i], 1);
   inicializar(silla, 4);
   cobegin
      filósofo(0); filósofo(1); … filósofo(4);
   coend;
}
```

Semáforos: Limitaciones

- Resulta difícil construir un programa correcto mediante semáforos. No es sencillo recordar qué semáforo está asociado a cada recurso o variable.
- Las operaciones P y V se distribuyen por todo el programa y no es fácil advertir el efecto global que provocan.
- El usuario es responsable tanto de la gestión de la exclusión mutua como de la sincronización entre los procesos.
- Cuando se examina un recurso y este está ocupado el proceso siempre se bloquea.

Monitores

- Tipo Abstracto de Datos: Datos locales, procedimientos y una secuencia de inicio.
- Los datos locales sólo están accesibles desde los procedimientos del monitor.
- A un monitor sólo puede entrar un proceso en un instante dado, de modo que si un proceso quiere usar un monitor y existe otro proceso que ya lo está usando, entonces el proceso que quiere entrar se suspende hasta que salga el que está dentro.
- Si los datos del monitor representan a algún recurso, el monitor ofrecerá un servicio de exclusión mutua en el acceso a ese recurso.



Monitor sencillo

Monitores: Sincronización

- El monitor proporciona sincronización por medio de variables de condición.
- Procedimientos para operar con las variables de condición:
  - **Espera(condición)**: Suspende la ejecución del proceso que llama bajo la condición. Se dispone de una cola de procesos a cada variable de condición.
  - **Señal(condición)**: Reanuda la ejecución de algún proceso suspendido en el procedimiento anterior. Si no hay procesos suspendidos no hace nada.
- Cuando un proceso se bloquea en una cola de una variable condición, sale del monitor, permitiendo que otro proceso pueda entrar en él.
- La propia naturaleza del monitor garantiza la exclusión mutua, sin embargo, la sincronización entre los procesos es responsabilidad del programador.

Monitores: sincronización entre procesos

Productor y consumidor: Tamaño de buffer limitado

```
struct TMonitor
{
   TElementos buffer[N];
   int sigent, sigsal;
   int contador;
   condition no_lleno, no_vacio;
}
{   sigent = 0; sigsal = 0; contador = 0;  }
```

Procedimientos del monitor:

```
void añadir(TElemento x)
{
   if (contador==N)
      espera(no_lleno);
   buffer[sigent] = x;
   sigent = (sigent + 1) %N;
   contador++;
   señal(no_vacío);
}
```

```
void coger(TElemento x)
{
   if (contador==0)
      espera(no_vacio);
   x = buffer[sigsalt];
   sigsal = (sigsal + 1) %N;
   contador--;
   señal(no_lleno);
}
```

Procesos:

```
void productor()
{
   TElemento x;
   while (true)
   {
      x = producir();
      añadir(x);
   }
}
```

```
void consumidor()
{
   TElemento x;
   while (true)
   {
      coger(x);
      consumir(x);
   }
}
```

```
void main()
{
   cobegin
   {
      productor();
      consumidor();
   }
   coend;
}
```

Monitores: sincronización entre procesos

Lectores y escritores: Prioridad a los escritores

```
struct TMonitor
{
    int lectores, escritores;
    condition leer, escribir;
}
{   lectores = 0; escritores = 0  }
```

Procedimientos del monitor:

```
void pre_leer()
{
    if ((escritores > 0) || (escribir.n_cola>0)) espera(leer);
    lectores++;
    señal(leer);
}
void post_leer()
{
    lectores--;
    if (lectores == 0) señal(escribir);
}
void pre_escribir()
{
    if ((lectores>0) || (escritores > 0)) espera(escribir);
    escritores++;
}
void post_escribir()
{
    escritores--;
    if (escribir.n_cola>0) señal(escribir);
    else señal(leer);
}
```

Monitores: sincronización entre procesos

Lectores y escritores: Prioridad a los escritores

Procesos:

```
void lector_i()
{
        . .
        pre_leer();
        leer();
        post_leer();
        . . .
}
```

```
void escritor_j()
{
        . .
        pre_escribir();
        escribir();
        post_escribir();
        . . .
}
```

```
void main()
{
  cobegin
    escritor_1();…; escritor_n(); lector_1(); … ; lector_m();
  coend;
}
```

Monitores: sincronización entre procesos

- Problema de **los cinco filósofos (I)**

```
struct TMonitor
{
    int estado[5];
    condition silla[5];
}
{
    for (i=0; i<5; i++) inicializar(estado[i], 0);
}
```

estado == 0: pensando
estado == 1: hambriento
estado == 2: comiendo

Procedimientos del monitor:

```
void coge_palillos(int i)
{
    estado[i]=1;
    prueba(i);
    if (estado[i]==1) espera(silla[i]);
}
void deja_palillos(int i)
{
    estado[i]=0;
    prueba((i+1)%5); //le da el palillo al filósofo de la derecha si lo necesita
    prueba((i+4)%5); //le da el palillo al filósofo de la izquierda si lo necesita
}
void prueba(int i)
{
    if ((estado[(i+1)%5]!=2) && (estado[(i+4)%5]!=2)
       && estado[i] ==1))
    {
        estado[i]=2;
        señal(silla[i]);
    }
}
```

Monitores: sincronización entre procesos

- Problema de **los cinco filósofos (II)**

Procesos:

```
void filosofo (int i)
{
    while (true)
    {
        pensar();
        coge_palillos(i);
        comer();
        deja_palillos(i);
    }
}

void main()
{
   int i;
   cobegin
      filósofo(0); filósofo(2); … filósofo(4);
   coend;
}
```

Esta solución mantiene la exclusión mutua y evita interbloqueos pero puede provocar que algún filósofo se muera de hambre.

Mensajes

- El paso de mensajes resuelve la comunicación y la sincronización de procesos. Adecuado para sistemas centralizados y distribuidos.
- Primitivas:
  - Enviar(destino, mensaje)
  - Recibir(origen, mensaje)
- Las primitivas son atómicas a nivel hardware.
- Direccionamiento
  - **Directo**: Se nombra de forma explícita en la primitiva el proceso al que se refieren.

    Enviar (Proceso$_i$, mensaje)
  - **Indirecto**: Los mensajes se envían y se reciben a través de una entidad intermedia llamada *buzón*.

    Enviar (buzón, mensaje)
    - Se desacopla el emisor y el receptor
    - Los conjuntos de emisores y receptores no tienen porqué tener la misma cardinalidad.
    - La asociación de procesos a buzones puede ser estática o dinámica.

Mensajes: sincronización

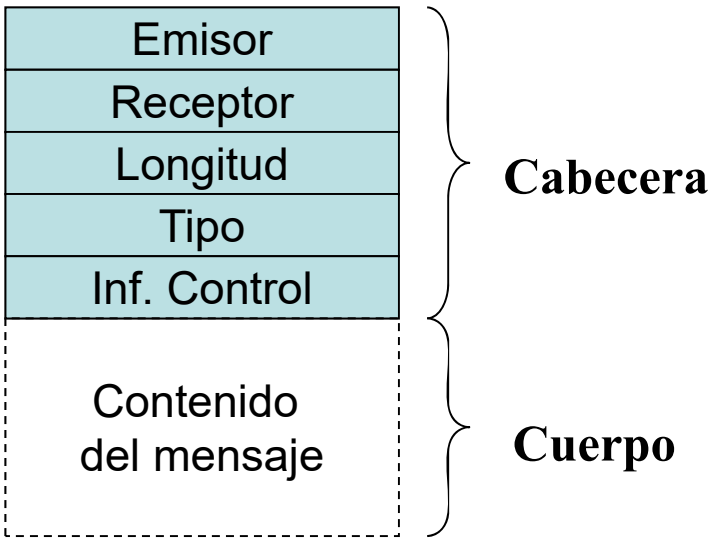
- Modelos de sincronización: *Enviar*
  - **Bloqueante**: El proceso que envía sólo prosigue su tarea cuando el mensaje ha sido recibido
  - **No Bloqueante**: El proceso que envía un mensaje sigue su ejecución sin preocuparse de si el mensaje se recibe o no.
  - **Invocación remota**: El proceso que envía el mensaje sólo prosigue su ejecución cuando ha recibido una respuesta explícita del receptor.
- Modelos de sincronización: *Recibir*
  - **Bloqueante**: El proceso que realiza *recibir* un mensaje lo recoge si éste existe o bien se bloquea si el mensaje no está.
  - **No Bloqueante:** El proceso que realiza *recibir* un mensaje especifica un tiempo máximo de espera del mensaje.

    Recibir(buzón, mensaje, tiempo_espera)

Mensajes: estructura de los mensajes

- Intercambio de información:
  - **Por valor**: Se realiza una copia del mensaje desde el espacio de direcciones del receptor.
  - **Por referencia**: Se transmite sólo un puntero al mensaje.
- Clasificación
  - **Longitud fija**
  - **Longitud variable**
  - **De tipo definido**

Los mensajes constan de una cabecera y de un cuerpo



Mensajes: exclusión mutua

- **Ejemplo**: Utilizar un mensaje como testigo entre los procesos.
  - Enviar no bloqueante y Recibir bloqueante
  - Direccionamiento indirecto. Existencia de un único buzón: *exmut*
  - Mensajes de contenido *nulo*.

Exclusión mutua por medio de mensajes

```
#define N_procesos xxx
TBuzón exmut;

void P(int i)
{
  mensaje msj;
  while (true)
  { . . .
    recibir(exmut, msj);
    /*acceso a la sección crítica*/
    enviar(exmut, msj)
    . . .
  }
}
void main()
{
  crear_buzón(exmut);
  enviar(exmut, NULL);
  cobegin
    P(1); P(2); …, P(N_procesos);
  coend;
}
```

Mensajes: sincronización

Productor y consumidor: buffer limitado

```
#define tamaño xxx
int i;
TBuzon puede_producir, puede_consumir;
void productor()
{
   mensaje msjp,aux;
   while(true)
   {
      msjp = producir();
      recibir(puede_producir, aux);
      enviar(puede_consumir, msjp);
   }
}
void consumidor()
{
   mensaje msjc;
   while(true)
   {
      recibir(puede_consumir, msjc);
      enviar(puede_producir, NULL);
      consumir(msjc);
   }
}
void main()
{
   crear_buzón(puede_producir);
   crear_buzón(puede_consumir);
   for(i=0; i<capacidad; i++) enviar(puede_producir,NULL);
   cobegin
      productor();
      consumidor();
   coend;
}
```