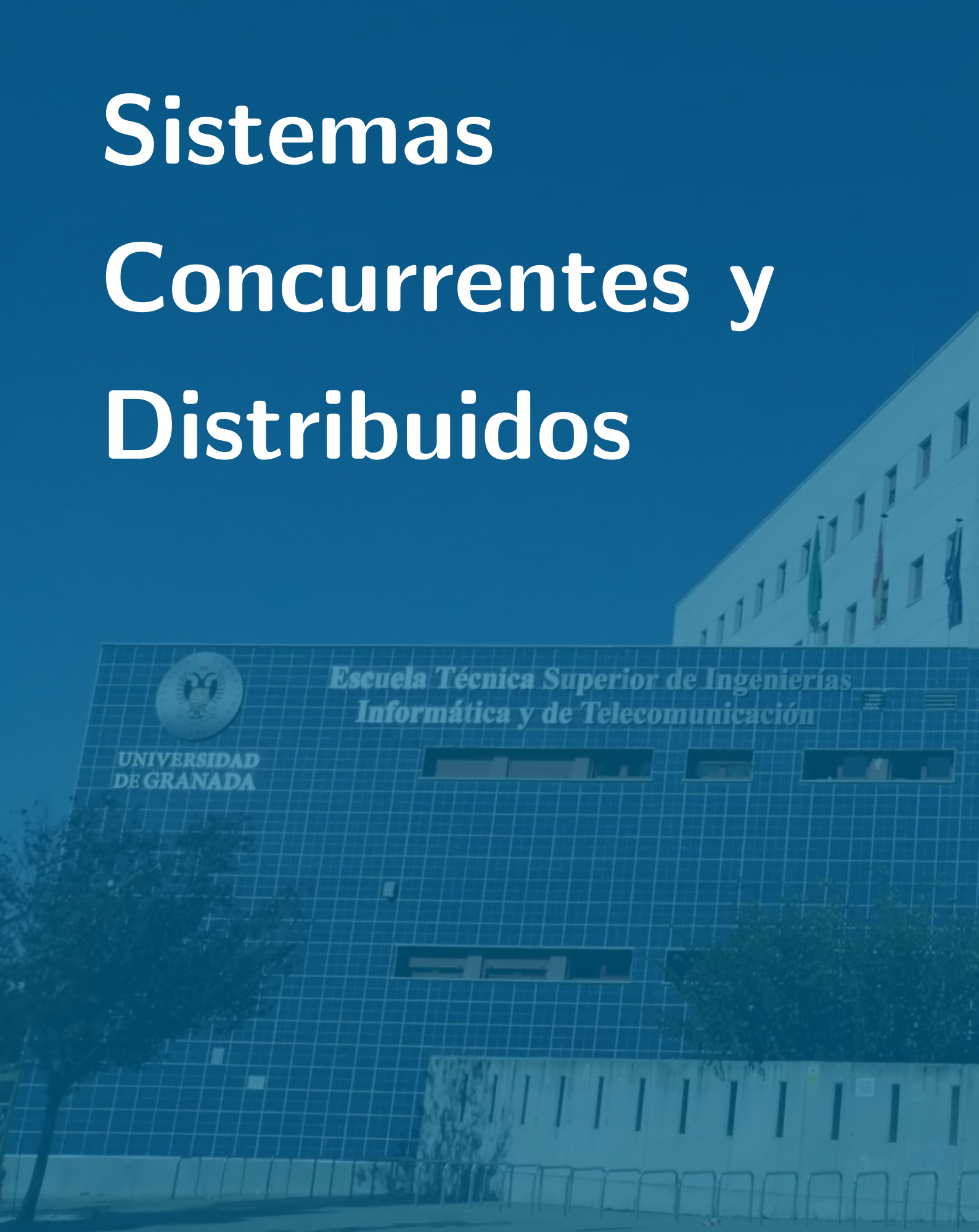
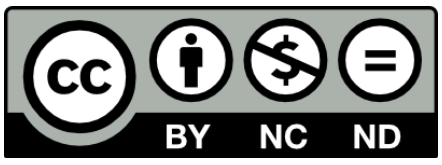
Sistemas Concurrentes y Distribuidos



Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Sistemas Concurrentes y Distribuidos

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán
Arturo Olivares Martos

Granada, 2024-2025

Índice general

1. Relaciones de problemas

1.1. Sincronización en Memoria Compartida

1.1.1. Exclusión mutua

Ejercicio 1.1.1. Un algoritmo para el cual sólo pudiésemos demostrar que cumple las 4 condiciones de Dijkstra, ¿qué tipo de propiedades concurrentes satisfacería?:

(a) seguridad.

(b) vivacidad.

(c) equidad.

Justificar las respuestas.

Ejercicio 1.1.2. En algunas aplicaciones es necesario tener exclusión mutua entre procesos con la particularidad de que puede haber como mucho n procesos en una sección crítica, con n arbitrario y fijo, pero no necesariamente igual a la unidad, sino posiblemente mayor. Diseña una solución para este problema basada en el uso de espera ocupada y cerrojos. Estructura dicha solución como un par de subrutinas (usando una misma estructura de datos en memoria compartida), una para el protocolo de entrada y otro el de salida, e incluye el pseudocódigo de las mismas.

Ejercicio 1.1.3. ¿Podría pensarse que una posible solución al problema de la exclusión mutua, sería el siguiente algoritmo (de la Figura 1.1) que no necesita compartir una variable `turno` entre los 2 procesos? Demostrar (sí o no) se satisfacen las siguientes propiedades:

(a) ¿la exclusión mutua? (propiedad de seguridad)

(b) ¿la ausencia de interbloqueo? (propiedad de alcanzabilidad)

Ejercicio 1.1.4. Al siguiente algoritmo (de la Figura 1.2) se le conoce como solución de Hyman al problema de la exclusión mutua (fue publicado en una revista de impacto en 1966). ¿Es correcta dicha solución?

Ejercicio 1.1.5. Supongamos el algoritmo de exclusión mutua que expresamos a continuación. Tenemos los procesos: $0, 1, \ldots, n-1$. Cada proceso i tiene una variable $s[i]$ inicializada a 0, que puede tomar los valores 0 o 1. El proceso i puede entrar en la sección crítica si:

$$s[i] \neq s[i-1] \text{ para } i > 0$$
$$s[0] = s[n-1] \text{ para } i = 0$$

Tras ejecutar su sección crítica, el proceso i deberá hacer:

$$s[i] = s[i-1] \text{ para } i > 0$$
$$s[0] = (s[0] + 1) \mod 2 \text{ para } i = 0$$

```
{variables compartidas y valores iniciales}
var b0 : boolean := false; {true si P0 quiere acceder o esta en SC}
    b1 : boolean := false; {true si P1 quiere acceder o esta en SC}
```

```
Process P0;
begin
while true do begin
   {Protocolo de entrada}

   {indica que quiere entrar}
   b0 := true;
   {si el otro también}
   while b1 do begin
      {cede temporalmente}
      b0 := false;
      {espera}
      while b1 do begin end
      {vuelve a cerrar el paso}
      b0 := true;
   end

   {Sección crítica}
   {Protocolo de salida}
   b0 := false;
   {Resto de sentencias}
end {while}
end
```

```
Process P1;
begin
while true do begin
   {Protocolo de entrada}

   {indica que quiere entrar}
   b1 := true;
   {si el otro también}
   while b0 do begin
      {cede temporalmente}
      b1 := false;
      {espera}
      while b0 do begin end
      {vuelve a cerrar el paso}
      b1 := true;
   end

   {Sección crítica}
   {Protocolo de salida}
   b1 := false;
   {Resto de sentencias}
end {while}
end
```

Figura 1.1: Código para el Ejercicio 1.1.3.

```
{variables compartidas y valores iniciales}
var c0 : integer := 1;
    c1 : integer := 1;
    turno : integer := 1;
```

```
process P0;
begin
while true do begin
   c0 := 0;
   while turno <> 0 do begin
      while c1 = 0 do begin end
      turno := 0;
   end

   {Sección crítica}
   c0 := 1;
   {Resto de sentencias}
end
end
```

```
process P1;
begin
while true do begin
   c1 := 0;
   while turno <> 1 do begin
      while c0 = 0 do begin end
      turno := 1;
   end

   {Sección crítica}
   c1 := 1;
   {Resto de sentencias}
end
end
```

Figura 1.2: Código para el Ejercicio 1.1.4.

Ejercicio 1.1.6. Se tienen 2 procesos concurrentes que representan 2 máquinas expendedoras de tickets (en la Figura 1.3) (señalan el turno en que ha de ser atendido el cliente), los números de los tickets se representan por dos variables `n1` y `n2` que valen inicialmente 0. El proceso con el número de ticket más bajo entra en su sección crítica. En caso de tener 2 números iguales se procesa primero el proceso número 1.

(a) Demostrar que se verifica la ausencia de interbloqueo (propiedad de *alcanzabilidad* de la sección crítica), la ausencia de inanición (propiedad de *vivacidad*) y la exclusión mutua (una propiedad de *seguridad*).

(b) Demostrar que las asignaciones `n1:=1` y `n2:=1` son ambas necesarias.

Ejercicio 1.1.7. El siguiente programa (en la Figura 1.4) es una solución al problema de la exclusión mutua para 2 procesos. Discutir la corrección de esta solución: si es correcta, entonces probarlo. Si no fuese correcta, escribir escenarios que demuestren que la solución es incorrecta.

Ejercicio 1.1.8. Con respecto al algoritmo de Peterson para N procesos: ¿sería posible que llegaran 2 procesos a la etapa $N-2$, 0 procesos a la etapa $N-3$ y en todas las etapas anteriores existiera al menos 1 proceso? Justificar la respuesta.

Ejercicio 1.1.9. En el *algoritmo de Peterson* para N procesos y considerando cualquier escenario de ejecución de dicho algoritmo, el número máximo de turnos que tiene que esperar cualquier proceso para entrar en sección crítica es $N-1$ turnos.

```
{variables compartidas y valores iniciales}
var n1 : integer := 0;
    n2 : integer := 0;
```

```
Process P1;
begin
while true do begin
   n1 := 1; {E1.1}
   n1 := n2 + 1; {L1.1; E2.1}
   while n2 <> 0 and {L2.1}
         n2 < n1 do begin end;
         {L3.1}

   {Sección crítica, SC.1}
   n1 := 0; {E3.1}
   {Resto de senetencias, RS.1}
end
end
```

```
Process P2;
begin
while true do begin
   n2 := 1; {E1.2}
   n2 := n1 + 1; {L1.2; E2.2}
   while n1 <> 0 and {L2.2}
         n1 < n2 do begin end;
         {L3.2}

   {Sección crítica, SC.2}
   n2 := 0; {E3.2}
   {Resto de senetencias, RS.2}
end
end
```

Figura 1.3: Código para el Ejercicio 1.1.6.

```
{variables compartidas y valores iniciales}
var c0 : integer := 1;
    c1 : integer := 1;
```

```
Process P0;
begin
while true do begin
   repeat
      c0 := 1 - c1;
   until c1 <> 0;

   {Sección crítica}
   c0 := 1;
   {Resto de sentencias}
end
end
```

```
Process P1;
begin
while true do begin
   repeat
      c1 := 1 - c0;
   until c0 <> 0;

   {Sección crítica}
   c1 := 1;
   {Resto de sentencias}
end
end
```

Figura 1.4: Código para el Ejercicio 1.1.7.

```
1  var turno : 0..N-1;
       flag : array[0..N-1] of (pasivo, solicitando, enSC);
   flag := pasivo:

5  Process P(i);
   begin
      {Resto de instrucciones}
      repeat
         flag[i] := solicitando;
10       j := turno;
         while turno <> i do begin
            if flag[turno] = pasivo then
               turno := i;
         end
15
         flag[i] := enSC;
         j := 0;

         while j < N and ((j=i) or flag[j] != enSC) do begin
20          j := j + 1;
         end
      until (j >= N);

      {Sección crítica}
25    flag[i] := pasivo;
   end
```

Figura 1.5: Código para el Ejercicio 1.1.10.

Ejercicio 1.1.10. Con respecto al algoritmo de la Figura 1.5 (algoritmo de Dijkstra para N procesos), demostrar la falsedad de la siguiente proposición: *si un conjunto de procesos está intentando pasar simultáneamente el primer bucle* (el de la línea 11), *y el proceso que tiene el turno está pasivo, entonces siempre conseguirá entrar primero en sección crítica el proceso de dicho grupo que consiga asignar la variable turno en último lugar.*

Ejercicio 1.1.11. El algoritmo de la Figura 1.6 (algoritmo de Knuth para N procesos) resuelve el problema de la exclusión mutua para N procesos, pra lo cual utiliza N variables booleanas `flag`, una variable `turn` y la variable local `j`.

(a) Demostrar que el algoritmo de Knuth verifica todas las propiedades exigibles a un programa concurrente, incluyendo la de equidad.

(b) Escribir un escenario en el que 2 procesos consiguen pasar el bucle de la instrucción de la línea 13, suponiendo que el turno lo tiene inicialmente el proceso `p(0)`.

Ejercicio 1.1.12. Si en el algoritmo de Dijkstra de la Figura 1.5 se cambia la instrucción de la línea 12 por esta otra: `if (flag[turno] <> enSC)`, entonces el algoritmos dejaría de ser correcto. Indicar qué propiedad(es) de corrección faltaría(n) y justificar por qué.

```
var flag : array[0..N-1] of (pasivo, solicitando, enSC);
    turn := 0..N-1;
flag := pasivo;
turn := 0;

Process P(i);
var j : integer;
begin
   {Resto de instrucciones}
   repeat
      flag[i] := solicitando;
      j := turn;
      while j <> i do begin
         if flag[j] <> pasivo then
            j := turn;
         else
            j := (j - 1) mod N;
         endif;
      end

      flag[i] := enSC;
      j := 0;

      while (j < N) and ((j=i) or flag[j] != enSC) do begin
         j := j + 1;
      end
   until (j >= N);

   turn := i;
   {Seccion crítica}
   j := (turn + 1) mod N;
   turn := j;
   flag[i] := pasivo;
end
```

Figura 1.6: Código para el Ejercicio 1.1.11

Ejercicio 1.1.13. Si en el algoritmo de Knuth de la Figura 1.6 se hacen las siguientes sustituciones:

- La condición de la instrucción `until` de la línea 27 por la condición `(j >= N) and (turno = i or flag[turno] = pasivo)`.
- Se inserta el siguiente bucle después de la instrucción de la línea 31:

```
1  while (j <> turn) and (flag[j] = pasivo) do begin
       j := j + 1;
   end
```

(a) Verificar las propiedades de exclusión mutua, alcanzabilidad de la sección crítica, vivacidad y equidad del algoritmo.

(b) Calcular el número de turnos máximo que puede llegar a tener que esperar un proceso que quiera entrar en su sección crítica con el algoritmo anterior.

Ejercicio 1.1.14. Demostrar que las instrucciones entre las líneas 18 y 28 del algoritmo de exclusión mutua distribuido de Ricart-Agrawala (de la Figura 1.7) no necesitan ser protegidas dentro de la sección crítica definida por las operaciones `wait()`, `signal()` del semáforo `s`.

```
var token_presente : boolean := false;
    enSC : boolean := false;
    peticion : array[1..n] of boolean := false;
```

```
Process P(i);
begin
   wait(s);
   if not token_presente then begin
      broadcast(pet, i);
      receive(acceso);
      token_presente := true;
   end

   enSC := true;
   signal(s);

   {Sección crítica}

   enSC := false;
   wait(s);

   for j := i+1 to n, 1 to i-1 do
      if peticion[j] and
         token_presente then begin

         token_presente := false;
         send(j,acceso);
         peticion[j] := false;
      end
   end

   signal(s);
end
```

```
Process Pet(i);
begin
   receive(pet, j);
   wait(s);
   peticion[j] := true;
   if token_presente and not enSC
   then
      {Repetir líneas 18-28}
end
```

Figura 1.7: Código para el Ejercicio 1.1.14

1.1.2. Monitores

Ejercicio 1.1.15. Sean los procesos P1, P2, y P3, cuyas secuencias de instrucciones son las que se muestran en el cuadro siguiente:

```
{variables globales}
Process P1;
begin
  while true do
  begin
    a;
    b;
    c;
  end
end
```

```
Process P2;
begin
  while true do
  begin
    d;
    e;
    f;
  end
end
```

```
Process P3;
begin
  while true do
  begin
    g;
    h;
    i;
  end
end
```

Se pide resolver los siguientes problemas de sincronización, considerando que son independientes unos de otros, con semáforos. Las casuísticas son las siguientes:

1. P2 podrá pasar a ejecutar `e` sólo si P1 ha ejecutado `a` o P3 ha ejecutado `g`.

   En este caso, se trata de una espera única en la que P2 debe esperar bien a P1 o bien a P3. Podemos resolverlo usando sólo un semáforo con valor inicial 0, con lo que debemos hacer las siguientes modificaciones en los códigos superiores:

   P1:

   ```
   a;
   sem_signal(s);
   b;
   c;
   ```

   P2:

   ```
   d;
   sem_wait(s);
   e;
   f;
   ```

   P3:

   ```
   g;
   sem_signal(s);
   h;
   i;
   ```

2. P2 podrá pasar a ejecutar `e` sólo si P1 ha ejecutado `a` y P3 ha ejecutado `g`.

   Ahora, debemos resolver dos esperas únicas, ya que P2 ha de esperar tanto a P1 como ha P3. Como hay dos motivos por los que P2 ha de esperar, usaremos dos semáforos, `s1` y `s2`, ambos inicializados a 0. Las modificaciones a realizar son:

   P1:

   ```
   a;
   sem_signal(s1);
   b;
   c;
   ```

   P2:

   ```
   d;
   sem_wait(s1);
   sem_wait(s2);
   e;
   f;
   ```

   P3:

   ```
   g;
   sem_signal(s2);
   h;
   i;
   ```

3. Sólo cuando P1 haya ejecutado `b`, podrá pasar P2 a ejecutar `e` y P3 a ejecutar `h`.
   En este caso, tenemos dos procesos esperando a un mismo proceso. En esta

ocasión, tenemos que hacer uso de dos semáforos, `s1` y `s2`, ambos inicializados a 0. Notemos que no podemos hacer uso de un semáforo de forma que P2 y P3 usen `wait` y que P1 haga un solo `signal`, ya que si los dos procesos llegan al `wait` a la vez, el `signal` de P1 solo despertará a un proceso, quedándose el otro bloqueado por siempre.

Por tanto, las modificaciones a realizar son:

P1:

```
a;
b;
sem_signal(s1);
sem_signal(s2);
c;
```

P2:

```
d;
sem_wait(s1);
e;
f;
```

P3:

```
g;
sem_wait(s2);
h;
i;
```

También podríamos haber usado un sólo semáforo y que el proceso P1 realizara dos `signal` después de la instrucción `b`.

4. Sincroniza los procesos de forma que las sentencias `b` en P1, `f` en P2, y `h` en P3, sean ejecutadas como mucho por 2 procesos simultáneamente.
Se trata de una generalización de la exclusión mutua, donde en vez de ejecutar a la vez las instrucciones `b`, `f` y `h` por un solo proceso, deben ejecutarse por como mucho dos procesos a la vez. Usaremos por tanto un solo semáforo inicializado con el valor 2. Las modificaciones a realizar son:

P1:

```
a;
sem_wait(s);
b;
sem_signal(s);
c;
```

P2:

```
d;
e;
sem_wait(s);
f;
sem_signal(s);
```

P3:

```
g;
sem_wait(s);
h;
sem_signal(s);
i;
```

Ejercicio 1.1.16. El cuadro que sigue nos muestra dos procesos concurrentes, P1 y P2, que comparten una variable global `x` y las restantes variables son locales a los procesos.

```
{variables globales}
Process P1;
var m: integer;
begin
  while true do
  begin
    m:= 2*x - n;
    print(m);
  end
end
```

```

Process P2;
var d: integer;
begin
  while true do
  begin
    d:= leer_teclado();
    x:= d - c*5;
  end
end
```

Se pide:

1. Sincronizar los procesos para que P1 use todos los valores `x` suministrados por P2.

   Estamos ante un problema del estilo productor/consumidor usando como buffer intermedio una variable. Este problema ya lo aprendimos a solucionar en las prácticas, y nos basta con usar dos semáforos:

   - `sem_prod` inicializado a 1.
   - `sem_cons` inicializado a 0.

   Las modificaciones a realizar en los códigos serían las siguientes:

   P1:

```
while true do begin
  sem_wait(sem_cons);
  m:= 2*x - n;
  sem_signal(sem_prod);
  print(m);
end
```

   P2:

```
while true do begin
  d:= leer_teclado();
  sem_wait(sem_prod);
  x:= d - c*5;
  sem_signal(sem_cons);
end
```

2. Sincronizar los procesos para que P1 utilice un valor sí y otro no de la variable `x`, es decir, utilice los valores primero, tercero, quinto, etc. que vaya alcanzando dicha variable.

   Para ello, la traza del programa que nos interesa obtener es:
   `E, L, E, E, L, E, E, L, ...`

   Para ello, podemos añadir otro par de instrucciones `wait`, `signal` en el consumidor:

   P1:

```
while true do begin
  sem_wait(sem_cons);
  m:= 2*x - n;
  sem_signal(sem_prod);
  print(m);
  sem_wait(sem_cons);
  sem_signal(sem_prod);
end
```

   P2:

```
while true do begin
  d:= leer_teclado();
  sem_wait(sem_prod);
  x:= d - c*5;
  sem_signal(sem_cons);
end
```

   De esta forma, el consumidor desperdicia los valores producidos en posición par por el productor.

Ejercicio 1.1.17. Supongamos que estamos en una discoteca y resulta que está estropeado el servicio de chicas y todos tienen que compartir el de chicos. Se pretende establecer un protocolo de entrada al servicio usando semáforos que asegure siempre el cumplimiento de las siguientes restricciones:

- Chicas: sólo puede estar 1 dentro del servicio.
- Chicos: pueden entrar más de 1, pero como máximo se admitirán a 5 dentro del servicio.
- Versión machista del protocolo: los chicos tienen preferencia sobre las chicas. Esto quiere decir que si una chica está esperando entrar al servicio y llega un chico, este puede pasar y ella sigue esperando. Incluso si el chico que ha llegado no pudiera entrar inmediatamente porque ya hay 5 chicos dentro del servicio, sin embargo, pasará antes que la chica cuando salga algún chico del servicio.
- Versión feminista del protocolo: las chicas tienen preferencia sobre los chicos. Esto quiere decir que si un chico está esperando y llega una chica, ésta debe pasar antes. Incluso si la chica que ha llegado no puede entrar inmediatamente al servicio porque ya hay una chica dentro, pasará antes que el chico cuando salga la chica que está dentro.

Se pide implementar las 2 versiones del protocolo anterior utilizando semáforos POSIX. Las cabeceras que estos han de tener los semáforos no nombrados de POSIX 1003 son las siguientes:

```
// Inicialización
int sem_init(sem_t* semaforo, int pcompartido, unsigned int contador);

// Destrucción
int sem_destroy(sem_t* semaforo);

// Sincronización-espera
int sem_wait(sem_t* semaforo);

// Sincronización-señala
int sem_post(sem_t* semaforo);
```

Observación. Se han de tener en cuenta los siguientes aspectos:

1. El valor inicial del semáforo se le asigna a `contador`. Si `pcompartido` es distinto de cero, entonces el semáforo puede ser utilizado por hilos que residen en procesos diferentes; si no, sólo puede ser utilizado por hilos dentro del espacio de direcciones de un único proceso.
2. Para que se pueda destruir, el semáforo ha debido ser explícitamente inicializado mediante la operación `sem_init(...)`. La operación anterior no debe ser utilizada con semáforos nombrados.
3. Los hilos llamarán a la función `int sem_wait(sem_t* semaforo)`, pasándole un identificador de semáforo inicializado con el valor '0', para sincronizarse con una condición. Si el valor del semáforo fuera distinto de '0', entonces el valor de s se decrementa en una unidad y no bloquea.
4. La operación `int sem_post(sem_t* semaforo)` sirve para señalar a los hilos bloqueadas en un semáforo y hacer que uno pase a estar preparado para ejecutarse. Si no hay hilos bloqueados en este semáforo, entonces la ejecución

de esta operación simplemente incrementa el valor de la variable protegida (s) del semáforo. Hay que tener en cuenta que no existe ningún orden de desbloqueo definido si hay varios hilos esperando en la cola asociada a un semáforo, ya que la implementación a nivel de sistema de la operación anterior supone que el planificador puede escoger para desbloquear a cualquiera de los hilos que esperan. En particular, podría darse el siguiente escenario, otro hilo ejecutándose puede decrementar el valor del semáforo antes que cualquier hilo que vaya a ser desbloqueado como resultado de `sem_post(...)` lo pueda hacer y, posteriormente, se volvería a bloquear el hilo despertado.

Versión machista.

Para plantear la solución en código, hemos creado 4 funciones: `entra_chico`, `entra_chica`, `sale_chico` y `sale_chica`, con el fin de simular el problema planteado.

En estas funciones, debemos usar unas variables compartidas que nos vayan indicando cuántos chicas y chicos hay esperando, así como si el baño está ocupado por chicos o por una chica.

Mostramos ahora las variables compartidas a declarar, junto con su código de inicialización y el código que debemos usar tras el cierre de los servicios para destruir los semáforos:

```
1  const int MAX = 5;
   int chicos_esperando, chicas_esperando, chicos_dentro;
   bool chica_dentro;
   sem_t* mutex, max, cola_chicos, cola_chicas;
5
   void inicializacion(){
      chicos_esperando = chicas_esperando = chicos_dentro = 0;
      chica_dentro = false;
      sem_init(mutex, 1, 1);
10    sem_init(max, 1, MAX);
      sem_init(cola_chicos, 1, 0);
      sem_init(cola_chicas, 1, 0);
   }

15 void destruccion(){
      sem_destroy(mutex);
      sem_destroy(max);
      sem_destroy(cola_chicos);
      sem_destroy(cola_chicas);
20 }
```

A continuación, las funciones de entrada y salida del baño para los chicos:

```
1  void entra_chico(){
      sem_wait(max);   // Espera a que salga uno
      sem_wait(mutex); // Adquiere ex. mutua (em)
```

```
    if(chica_dentro){    // Baño ocupado
        chicos_esperando++;
        sem_post(mutex);    // Libera em
        sem_wait(cola_chicos);
        chicos_esperando--;
    }

    chicos_dentro++;
    // Si puede entrar otro
    if(chicos_dentro < MAX && chicos_esperando > 0){
        sem_post(cola_chicos);
    }else{
        sem_post(mutex);
    }
}

void sale_chico(){
    sem_wait(mutex);     // Adquiere em
    chicos_dentro--;

    // Si se queda el baño libre, hay chica esperando pero no chicos
    if(chicos_esperando == 0 && chicas_esperando > 0 && chicos_dentro ==
    0){
        sem_post(cola_chicas);
    }else{
        sem_post(mutex);
    }
    sem_post(max);
}
```

Ahora, las funciones de entrada y salida de las chicas serían:

```
void entra_chica(){
    sem_wait(mutex);     // Adquiere em

    // Si el baño está ocupado o hay un chico esperando
    if(chicos_dentro > 0 || chica_dentro || chicos_esperando > 0){
        chicas_esperando++;
        sem_post(mutex);
        sem_wait(cola_chicas);
        chicas_esperando--;
    }

    chica_dentro = true;
    sem_post(mutex);
}

void sale_chica(){
    sem_wait(mutex);     // Adquiere em
    chica_dentro = false;

    if(chicos_esperando > 0){    // Prioridad a chicos
        sem_post(cola_chicos);
    }else if(chicas_esperando > 0){
        sem_post(cola_chicas);
    }else{
```

```
25          sem_post(mutex);
        }
    }
```

Versión feminista.

Para esta versión, usaremos las mismas variables compartidas con las mismas funciones `inicializacion` y `destruccion` de la versión anterior.

A continuación, las funciones de entrada y salida del baño para los chicos:

```
1   void entra_chico(){
        sem_wait(max);   // Espera a que salga un chico del baño
        sem_wait(mutex); // Para ex. mutua (em)

5       if(chicas_esperando > 0 || chica_dentro){    // Prioridad a chicas
            chicos_esperando++;   // Un nuevo chico esperando
            sem_post(mutex);    // Libera em antes de bloquearse
            sem_wait(cola_chicos);

10          chicos_esperando--;
        }

        // Ya no hay chicas en cola o dentro del baño

15      chicos_dentro++;
        if(chicos_cola > 0 && chicos_dentro < MAX){  // Si puede entrar otro
            sem_post(cola_chicos);
        }else{
            sem_post(mutex);
20      }
    }

    void sale_chico(){
        sem_wait(mutex);     // Espera para em
25      chicos_dentro--;

        // Si hay una chica y puede entrar
        if(chicos_dentro == 0 && chicas_esperando > 0){
            sem_post(cola_chicas);
30      }else{
            sem_post(mutex);  // Libera em
        }

        sem_post(max);       // Un chico menos
35  }
```

Ahora, mostramos el código de entrada y salida para las chicas:

```
1   void entra_chica(){
        sem_wait(mutex);     // Adquiere em

        if(chicos_dentro > 0 || chica_dentro){   // Si baño ocupado
5           chicas_esperando++;
            sem_post(mutex);
```

```
        sem_wait(cola_chicas);

        chicas_esperando--;
    }

    chica_dentro = true;
    sem_post(mutex);
}

void sale_chica(){
    sem_wait(mutex);      // Adquiere em
    chica_dentro = false;

    if(chicas_esperando > 0){    // Desbloquea a chica
        sem_post(cola_chicas);
    }else if(chicos_esperando > 0){  // Desbloquea a chico
        sem_post(cola_chicos);
    }else{
        sem_signal(mutex);
    }
}
```

Ejercicio 1.1.18. Aunque un monitor garantiza la exclusión mutua, los procedimientos tienen que ser reentrantes. Explicar por qué.

Los procedimientos han de ser reentrantes porque queremos tener procesos que ejecuten procedimientos del monitor, que estos puedan bloquearse durante la ejecución de los mismos y que puedan desbloquearse tras unas condiciones y que puedan seguir ejecutando la función por donde iban.

Es necesario que los procedimientos sean reentrantes para que los procesos que ejecutan los procedimientos y que se bloquean sean capaces de seguir la ejecución de la función una vez desbloqueados por la instrucción por la que se quedaron, manteniendo intactos los valores de las variables locales usadas en el procedimiento.

Ejercicio 1.1.19. Se consideran dos tipos de recursos accesibles por varios procesos concurrentes (denominamos a los recursos como recursos de tipo 1 y de tipo 2). Existen N_1 ejemplares de recursos de tipo 1 y N_2 ejemplares de recursos de tipo 2. Para la gestión de estos ejemplares, queremos diseñar un monitor (con semántica SU) que exporta un procedimiento (`pedir_recurso`), para pedir un ejemplar de uno de los dos tipos de recursos. Este procedimiento incluye un parámetro entero (`tipo`), que valdrá 1 ó 2 indicando el tipo del ejemplar que se desea usar, así mismo, el monitor incorpora otro procedimiento (`liberar_recurso`) para indicar que se deja de usar un ejemplar de un recurso previamente solicitado (este procedimiento también admite un entero que puede valer 1 ó 2, según el tipo de ejemplar que se quiera liberar). En ningún momento puede haber un ejemplar de un tipo de recurso en uso por más de un proceso.

En este contexto, responde a estas cuestiones:

1. Implementa el monitor con los dos procedimientos citados, suponiendo que N_1 y N_2 son dos constantes arbitrarias, mayores que cero.

2. El uso de este monitor puede dar lugar a interbloqueo. Esto ocurre cuando más de un proceso, en algún punto en su código, tiene la necesidad de usar dos ejemplares de recursos de distinto tipo a la vez. Describe la secuencia de peticiones (llamadas al procedimiento correspondiente del monitor) que da lugar a interbloqueo.

3. Una posible solución al problema anterior es obligar a que si un proceso necesita dos recursos de distinto tipo a la vez, deba de llamar a `pedir_recurso`, dando un parámetro con valor 0, para indicar que necesita los dos ejemplares. En esta solución, cuando un ejemplar quede libre, se dará prioridad a los posibles procesos esperando usar dos ejemplares, frente a los que esperan usar solo uno de ellos.

1. Mostramos la implementación usando pseudocódigo:

```
monitor Recursos (N1, N2 : integer);
   var libres : array[1..2] of integer;
       colas : array[1..2] of condition;

   begin
      libres[1] = N1;
      libres[2] = N2;
   end

   procedure pedir_recurso(tipo : 1..2);
   begin
      if libres[tipo] = 0 then
         colas[tipo].wait();
      end;
      libres[tipo]--;
   end

   procedure liberar_recurso(tipo : 1..2);
   begin
      libres[tipo]++;
      cola[tipo].signal();
   end
end
```

2. Para mostrar una situación en la que el uso de este monitor puede dar lugar a un interbloqueo, mostramos los siguientes códigos:

```
var monitor : Recursos(1,1);

process P1;
begin
   monitor.pedir_recurso(1);
   monitor.pedir_recurso(2);
   {Uso de los recursos}
   monitor.liberar_recurso(1);
   monitor.liberar_recurso(2);
end
```

```
process P2;
begin
   monitor.pedir_recurso(2);
   monitor.pedir_recurso(1);
   {Uso de los recursos}
   monitor.liberar_recurso(2);
   monitor.liberar_recurso(1);
end

begin
   cobegin P1 || P2 coend
end
```

Si ahora se ejecutan los códigos y se sucede un entrelazamiento en las instrucciones del programa concurrente obteniendo la traza:

P1:	pedir_recurso(1)
P2:	pedir_recurso(2)
P1:	pedir_recurso(2)
P2:	pedir_recurso(1)

Después de las dos primeras instrucciones, no quedarán recursos de ningún tipo libres, pero P1 no podrá ejecutarse ya que está esperando a un recurso de tipo 2 y P2 tampoco podrá hacerlo, al estar esperando a un recurso de tipo 1.

Notemos que con monitores podemos demostrar la corrección parcial de los programas, pero no que estos terminen (esto es, que estén libres de interbloqueos o situaciones similares). Es responsabilidad del programador evitar este tipo de situaciones.

3. Aunque no podemos crear un monitor que evite el interbloqueo en un mal uso del monitor, sí que podemos ofrecer una solución de compromiso que ayude al programador a evitar este tipo de situaciones, tal y como se describe en el enunciado y mostramos con el siguiente código:

```
    monitor Recursos (N1, N2 : integer);
    var libres : array[1..2] of integer;
        colas : array[0..2] of condition;

    begin
       libres[1] := N1;
       libres[2] := N2;
    end

    procedure pedir_recurso(tipo : 0..2);
    begin
       if tipo = 0 then
          if(libres[1] = 0 or libres[2] = 0) then
             colas[0].wait();
          end
          libres[1]--;
          libres[2]--;
```

```
        else begin
           if(libres[tipo] = 0) then
              colas[tipo].wait();
           end
           libres[tipo]--;
        end
     end

     procedure liberar_recurso(tipo : 1..2);
     var otro_tipo = tipo mod 2 + 1;
     begin
        libres[tipo]++;

        {Si hay otro tipo}
        if(libres[otro_tipo] > 0 and cola[0].queue()) then
           cola[0].signal();
        else
           cola[tipo].signal();
        end
     end
end
```

Ejercicio 1.1.20. Escribir una solución al problema de lectores-escritores con monitores:

1. Con prioridad a los lectores: quiere decir que, si en un momento puede acceder al recurso, tanto un lector como un escritor, se da paso preferentemente al lector.

2. Con prioridad a los escritores: quiere decir que, si en un momento puede acceder tanto un lector como un escritor, se da paso preferentemente al escritor.

3. Con prioridades iguales: en este caso, los procesos acceden al recurso estrictamente en orden de llegada, lo cual implica, en particular, que si hay lectores leyendo y un escritor esperando, los lectores que intenten acceder después del escritor no podrán hacerlo hasta que no lo haga dicho escritor.

En este problema, contamos con procesos de dos tipos, lectores y escritores. Si un escritor hace uso del recurso compartido, este debe hacerlo en exclusión mutua. Sin embargo, si un lector hace uso del recurso, puede haber más lectores que también lo estén usando al mismo tiempo. Planteamos ahora la solución a cada uno de los puntos, usando para ello 2 procedimientos por cada tipo de proceso que interviene en el problema (un procedimiento de entrada al recurso y otro de fin de uso):

1. Solución para dar prioridad a los lectores:

```
Monitor LecEsc;
   var lec_dentro : integer;
       esc_dentro : boolean;
       cola_lec, cola_esc : condition;

   begin
```

```
      lec_dentro = 0;
      esc_dentro = false;
   end

   procedure entra_lector();
   begin
      if esc_dentro then
         cola_lec.wait();
      end

      lec_dentro++;

      if cola_lec.queue() then  {por eficiencia}
         cola_lec.signal();
      end
   end

   procedure sale_lector();
   begin
      lec_dentro--;

      if lec_dentro = 0 then
         cola_esc.signal();
      end
   end

   procedure entra_escritor();
   begin
      if lec_dentro > o OR esc_dentro then
         cola_esc.wait();
      end

      esc_dentro = true;
   end

   procedure sale_escritor();
   begin
      esc_dentro = false;

      if cola_lec.queue() then
         cola_lec.signal();
      else
         cola_esc.signal();
      end
   end
end
```

2. Solución para dar prioridad a los esctirores. En este caso, el código es idéntico, salvo en el procedimiento `sale_escritor`, que quedaría de la siguiente forma:

```
procedure sale_escritor();
begin
   esc_dentro = false;
```

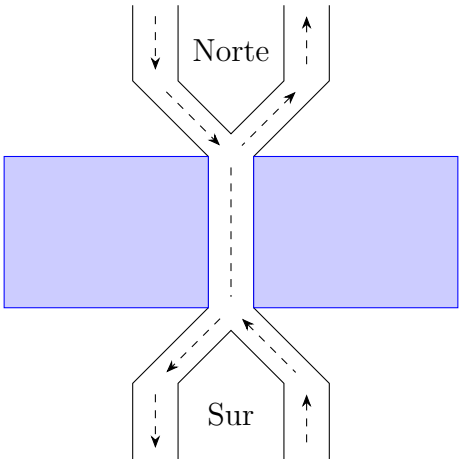


Figura 1.8: Problema de exclusión mutua en el acceso de coches desde 2 sentidos opuestos a un puente de un solo carril.

```
5     if cola_esc.queue() then
          cola_esc.signal();
      else
          cola_lec.signal();
      end
10 end
```

3. Finalmente, para dar una solución en la que lectores y escritores tienen prioridades iguales.

Ejercicio 1.1.21. Varios coches que vienen del norte y del sur pretenden cruzar un puente sobre un río (ver Figura **??**). Sólo existe un carril sobre dicho puente. Por lo tanto, en un momento dado, el puente solo puede ser cruzado por uno o más coches en la misma dirección (pero no en direcciones opuestas).

1. Completar el código del siguiente monitor que resuelve el problema del acceso al puente suponiendo que llega un coche del norte (sur) y cruza el puente si no hay otro coche del sur (norte) cruzando el puente en ese momento.

```
1  Monitor Puente
   var ... ;
   procedure EntrarCocheDelNorte()
   begin
5  ...
   end
   procedure SalirCocheDelNorte()
   begin
   ....
10 end
   procedure EntrarCocheDelSur()
   begin
   ....
   end
```

```
15  procedure SalirCocheDelSur()
    begin
    ...
    end
    { Inicializacion }
20  begin
    ....
    end
```

2. Mejorar el monitor anterior, de forma que la dirección del tráfico a través del puente cambie cada vez que lo hayan cruzado 10 coches en una dirección, mientras 1 ó más coches estuviesen esperando cruzar el puente en dirección opuesta.

1. Este problema es una simplificación del problema de los lectores/escritores. Para el mismo, proponemos el siguiente monitor como solución al problema:

```
1   Monitor Puente;
       var coches_N, coches_S : integer;
           cola_N, cola_S : condition;

5      procedure EntrarCocheDelNorte();
       begin
          { Hay coches en otro sentido }
          if coches_S > 0 then
             cola_N.wait();
10
          coches_N := coches_N + 1;

          { Por temas de eficiencia }
          if coches_N.queue() then
15           coches_N.signal();
       end

       procedure SalirCocheDelNorte();
       begin
20        coches_N := coches_N - 1;

          { El puente se queda vacío }
          if coches_N = 0 then
             coches_S.signal();
25     end
```

```
1      procedure EntrarCocheDelSur();
       begin
          { Hay coches en otro sentido }
          if coches_N > 0 then
5            cola_S.wait();

          coches_S := coches_S + 1;

          { Por temas de eficiencia }
          if coches_S.queue() then
10           coches_S.signal();
       end

       procedure SalirCocheDelSur();
       begin
15        coches_S := coches_S - 1;

          { El puente se queda vacío }
          if coches_S = 0 then
             coches_N.signal();
20     end

       begin
          coches_N := 0; coches_S := 0;
       end
25  end
```

2. Ahora, una vez que haya algún coche en sentido opuesto, contabilizaremos el número de coches que pasan en nuestro sentido mientras que hay esperando en sentido opuesto, con la finalidad de invertir el sentido una vez hayan pasado `umbral` coches:

```
Monitor Puente;
   var coches_N, coches_S, N_pueden, S_pueden : integer;
       cola_N, cola_S : condition;

   procedure EntrarCocheDelNorte();
   begin
      { Hay coches en otro sentido }
      if (coches_S > 0 or N_pueden = 0) then cola_N.wait();

      coches_N := coches_N + 1;

      { Si hay esperando en el sur }
      if cola_S.queue() then N_pueden := N_pueden - 1;

      if (coches_N.queue() and N_pueden > 0) then coches_N.signal();
   end

   procedure SalirCocheDelNorte();
   begin
      coches_N := coches_N - 1;

      { El puente se queda vacío }
      if coches_N = 0 then
      begin
         S_pueden := UMBRAL;
         coches_S.signal();
      end
   end

   procedure EntrarCocheDelSur();
   begin
      { Hay coches en otro sentido }
      if (coches_N > 0 or S_pueden = 0) then cola_S.wait();

      coches_S := coches_S + 1;

      { Si hay esperando en el norte }
      if cola_N.queue() then S_pueden := S_pueden - 1;

      if (coches_S.queue() and S_pueden > 0) then coches_S.signal();
   end

   procedure SalirCocheDelSur();
   begin
      coches_S := coches_S - 1;

      { El puente se queda vacío }
      if coches_S = 0 then
      begin
         N_pueden = UMBRAL;
         coches_N.signal();
      end
   end

   begin
      coches_N := 0; coches_S := 0; N_pueden := UMBRAL; S_pueden := UMBRAL;
   end
end
```

Ejercicio 1.1.22. Una tribu de antropófagos comparte una olla en la que caben M misioneros. Cuando algún salvaje quiere comer, se sirve directamente de la olla, a no ser que ésta esté vacía. Si la olla está vacía, el salvaje despertará al cocinero y esperará a que éste haya rellenado la olla con otros M misioneros. Para solucionar la sincronización usamos un monitor llamado `Olla`, que se puede usar así:

```
1  monitor Olla ;
   ....
   begin
   ....
5  end;
   proceso ProcSalvaje[ i:1..N ] ;
   begin
   while true do begin
           Olla.Servirse_1_misionero();
10         Comer(); { es un retraso aleatorio }
       end
   end;
   proceso ProcCocinero ;
   begin
15 while true do begin
           Olla.Dormir();
           Olla.Rellenar_Olla();
       end
   end;
```

Se pide diseñar el código del monitor `Olla` para que se satisfaga la sincronización requerida en el enunciado del problema, teniendo en cuenta que:

- La solución propuesta no debe producir interbloqueos.
- Los salvajes podrán comer siempre que haya comida en la olla.
- Sólo se ha de despertar al proceso cocinero cuando la olla esté vacía.

La solución al problema viene dada gracias al monitor (donde `M` es una constante que fija el número de misioneros a rellenar cada vez):

```
Monitor Olla;
   var num_misioneros : integer;
       comer, dormir : condition;

   procedure Servirse_1_misionero();
   begin
      { Si no quedan misioneros }
      if num_misioneros = 0 then
      begin
         dormir.signal();  { Despierta al cocinero }
         comer.wait();
      end

      num_misioneros := num_misioneros - 1;

      if num_misioneros > 0 then
         comer.signal();
   end

   procedure Dormir();
   begin
      if num_misioneros > 0 then
         dormir.wait();
   end

   procedure Rellenar_Olla();
   begin
      num_misioneros := M;

      { Avisa a un salvaje }
      comer.signal();
   end

   begin
      num_misioneros := M;
   end
end
```

Ejercicio 1.1.23. Una cuenta de ahorros es compartida por varias personas (procesos). Cada persona puede depositar o retirar fondos de la cuenta. El saldo actual de la cuenta es la suma de todos los depósitos menos la suma de todos los reintegros. El saldo nunca puede ser negativo. Queremos usar un monitor para resolver el problema.

El monitor debe tener 2 procedimientos: `depositar(c)` y `retirar(c)`. Suponer que los argumentos de las 2 operaciones son siempre positivos, e indican las cantidades a depositar o retirar. El monitor usará la semántica señalar y espera urgente (SU). Se deben de escribir varias versiones de la solución, según las variaciones de los requerimientos que se describen a continuación:

1. Todo proceso puede retirar fondos mientras la cantidad solicitada `c` sea menor o igual que el saldo disponible en la cuenta en ese momento. Si un proceso intenta retirar una cantidad `c` mayor que el saldo, debe quedar bloqueado hasta que el saldo se incremente lo suficiente (como consecuencia de que otros

procesos depositen fondos en la cuenta) para que se pueda atender la petición. Hacer dos versiones del monitor:

a) Colas normales FIFO sin prioridad.

b) Con colas de prioridad.

2. El reintegro de fondos a los clientes se hace únicamente según el orden de llegada, si hay más de un cliente esperando, sólo el primero que llegó puede optar a retirar la cantidad que desea, mientras esto no sea posible, esperarán todos los clientes, independientemente de cuanto quieran retirar los demás. Por ejemplo, suponer que el saldo es 200 unidades y un cliente está esperando un reintegro de 300 unidades, entonces si llega otro cliente debe esperarse, incluso si quiere retirar 200 unidades. De nuevo, resolverlo utilizando dos versiones:

   a) Colas normales FIFO sin prioridad.

   b) Con colas de prioridad.

1. Para la primera versión:

   a) Con colas normales FIFO sin prioridad en las variables condición:

```
1  Monitor Cuenta;
   var saldo : integer;
       cola : condition;

5     procedure depositar(c : integer);
      begin
         saldo = saldo + c;
         cola.signal();
      end
10
      procedure retirar(c : integer);
      begin
         while c > saldo do begin
            cola.signal();
15          cola.wait();
         end do
         saldo = saldo - c;
         cola.signal();
      end
20
      begin
         saldo = 0;
      end
   end
```

   b) Con colas de prioridad en las variables condición, el código del monitor queda más simple, teniendo solo que modificar el procedimiento `retirar`:

```
procedure retirar(c : integer);
begin
   while c > saldo do begin
      cola.wait(c);
   end do
   saldo = saldo - c;
   cola.signal();
end
```

2. Ahora, con orden en las retiradas, es necesrio disponer de una variable condición más, que controle si hay alguien esperando a retirar dinero.

   a) Con colas normales FIFO en las variables condición:

```
Monitor Cuenta;
var saldo : integer;
    cola, ventanilla : condition;

   procedure depositar(c : integer);
   begin
      saldo = saldo + c;
      cola.signal();
   end

   procedure retirar(c : integer);
   begin
      if ventanilla.queue() then
         cola.wait();

      while c > saldo do begin
         ventanilla.wait();
      end do

      saldo = saldo - c;
      cola.signal();
   end

   begin
      saldo = 0;
   end
end
```

   b) Con colas con prioridad en las variables condición, lo que hacemos es introducir una nueva variable `contador`, con la finalidad de bloquear a los procesos según su orden de llegada.

```
Monitor Cuenta;
var saldo : integer;
    contador : integer := 0;
    cola : condition;

   procedure depositar(c : integer);
   begin
```

```
      saldo = saldo + c;
      cola.signal();
   end

   procedure retirar(c : integer);
   var ticket : integer;
   begin
      ticket = contador;
      contador = contador + 1;

      if cola.queue() then
         cola.wait(ticket);

      while c > saldo do
         cola.wait(ticket);
      end

      saldo = saldo - c;
      cola.signal();
   end

   begin
      saldo = 0;
   end
end
```

Ejercicio 1.1.24. Los procesos P1, P2, ..., Pn comparten un único recurso R, pero sólo un proceso puede utilizarlo cada vez. Un proceso Pi puede comenzar a utilizar R si está libre; en caso contrario, el proceso debe esperar a que el recurso sea liberado por otro proceso. Si hay varios procesos esperando a que quede libre R, se concederá al proceso que tenga mayor prioridad. La regla de prioridad de los procesos es la siguiente: el proceso Pi tiene prioridad i, (con $1 \leqslant i \leqslant n$), donde los números menores implican mayor prioridad (es decir, si $i < j$, entonces Pi pasa por delante de Pj). Implementar un monitor que implemente los procedimientos `Pedir(...)` y `Liberar()` con un monitor que garantice la exclusión mutua y el acceso prioritario del procesos al recurso R.

Suponiendo que podemos usar variables condición con colas prioritarias, la solución al ejercicio es:

```
Monitor Recurso;
var libre : boolean;
    cola : condition;

   begin
      libre = true;
   end

   procedure Liberar();
   begin
      libre = true;
      cola.signal();
   end
```

```
   procedure Pedir(id : integer); {Nº de proceso}
   begin
      if not libre then
         cola.wait(id);
      end

      libre = false;
   end
end
```

Ejercicio 1.1.25. El siguiente monitor (`Barrera2`) proporciona un único procedimiento de nombre `entrada()`, que provoca que el primer proceso que lo llama sea suspendido y el segundo que lo llama despierte al primero que lo llamó (a continuación ambos continúan), y así actúa cíclicamente. Obtener una implementación de este monitor usando semáforos.

```
Monitor Barrera2 ;
var n : integer; { num. de proc. que han llegado desde el signal }
s : condicion ; { cola donde espera el segundo }
procedure entrada() ;
    begin
        n := n+1 ; { ha llegado un proceso mas }
        if n < 2 then { si es el primero: }
            s.wait() { esperar al segundo }
        else begin { si es el segundo: }
            n := 0; { inicializa el contador }
            s.signal() { despertar al primero }
        end
    end

    { Inicializacion }
    begin
        n := 0 ;
    end
end
```

Para realizar esta implementación, haremos uso de las siguientes variables compartidas:

```
var n : integer;
    mutex, sem : semaphore;
```

donde inicializaremos los semáforos `mutex` y `sem` a 0. Usaremos la siguiente función, que hará las veces del procedimiento `entrada` del monitor:

```
procedure entrada();
begin
   sem_wait(mutex);  {Adquiere em}
   n := n + 1;

   if n < 2 then begin {Si era el primero}
      sem_signal(mutex);  {Libera em antes de bloquearse}
      sem_wait(sem);
```

```
   else then begin {Si era el segundo}
      n := 0;
      sem_signal(sem);
      sem_signal(mutex);
   end
end
```

Ejercicio 1.1.26. Este es un ejemplo clásico que ilustra el problema del interbloqueo, y aparece en la literatura informática con el nombre de el problema de los filósofos-comensales. Se puede enunciar como se indica a continuación: sentados a una mesa están cinco filósofos, la actividad de cada filósofo es un ciclo sin fin de las operaciones de pensar y comer; entre cada dos filósofos hay un tenedor y para poder comer, un filósofo necesita obligatoriamente dos tenedores: el de su derecha y el de su izquierda. Se han definido cinco procesos concurrentes, cada uno de ellos describe la actividad de un filósofo. Los procesos usan un monitor, llamado `MonFilo`. Antes de comer cada filósofo debe disponer de su tenedor de la derecha y el de la izquierda, y cuando termina la actividad de comer, libera ambos tenedores. El filósofo i alude al tenedor de su derecha como el número i, y al de su izquierda como el número $i + 1 \mathrm{mod} 5$. El monitor `MonFilo` exportará dos procedimentos: `coge_tenedor(num_tenedor,num_proceso)` y `libera_tenedor(num_tenedor)` para indicar que un proceso filósofo desea coger un tenedor determinado. El código del programa (sin incluir la implementación del monitor) es el siguiente:

```
monitor MonFilo ;
....
procedure coge_tenedor( num_ten, num_proc : integer );
....
procedure libera_tenedor( num_ten : integer );
....
begin
....
end
proceso Filosofo[ i: 0..4 ] ;
    begin
        while true do begin
            MonFilo.coge_tenedor(i,i);          {argumento 1=codigo tenedor}
            MonFilo.coge_tenedor(i+1 mod 5,i);  {argumento 2=numero de proceso}
        comer();
        MonFilo.libera_tenedor(i);
        MonFilo.libera_tenedor(i+1 mod 5);
        pensar();
    end
end
```

Con este interfaz para el monitor, responde a las siguientes cuestiones:

1. Diseña una solución para el monitor `MonFilo`.

2. Describe la situación de interbloqueo que puede ocurrir con la solución que has escrito antes.

3. Diseña una nueva solución, en la cual se evite el interbloqueo descrito, para ello, esta solución no debe permitir que haya más de cuatro filósofos simultáneamente intentado coger su primer tenedor.

1. Hemos diseñado el siguiente monitor, donde `tenedores[i]` indica si el tenedor i-ésimo ha sido (`true`) cogido o si no (`false`).

```
monitor MonFilo;
   var tenedores : array[0..4] of boolean;
       filosofo : array[0..4] of condition;

   procedure coge_tenedor(num_tenedor, num_proceso : integer);
   begin
      { Si ya han cogido el tenedor }
      if tenedores[num_tenedor] then
         filosofo[num_proceso].wait();

      tenedores[num_tenedor] := true;
   end

   procedure libera_tenedor(num_tenedor);
   begin
      tenedores[num_tenedor] := false;

      {Si el filósofo de la izqda estaba esperando se despierta}
      if filosofo[num_tenedor].queue() then
         filosofo[num_tenedor].signal();
      else then  {Si no, despertamos al filósofo de la dcha}
         filosofo[(num_tenedor - 1) mod 5].signal();
   end

   begin
      for i = 0 to 4 do
         tenedores[i] := false;
      end
   end
end
```

2. Ante el código proporcionado para los filósofos anteriormente, puede producirse una situación de interbloqueo si se sucede la siguiente traza de ejecución:

	Nº Proceso	Llamada a procedimiento
1	0	`MonFilo.coge_tenedor(0,0);`
2	1	`MonFilo.coge_tenedor(1,1);`
3	2	`MonFilo.coge_tenedor(2,2);`
4	3	`MonFilo.coge_tenedor(3,3);`
5	4	`MonFilo.coge_tenedor(4,4);`

Tabla 1.1: Traza de ejecución que lleva a una situación de interbloqueo.

En este instante, todos los filósofos han cogido el tenedor de su derecha, el cual no soltarán hasta comer, para lo que necesitan el tenedor de su izquierda.

Sin embargo, como son 5 filósofos y solo hay disponibles 5 tenedores, ningún filósofo conseguirá nunca su tenedor de la izquierda para poder comer. De esta forma, todos los filósofos quedarán bloqueados por siempre.