Sistemas Concurrentes y Distribuidos: Problemas Tema 2

Juan Manuel Rodríguez Gómez

Ejercicio 61

En un pueblo hay una pequeña barbería con una puerta de entrada y otra de salida, dichas puertas son tan estrechas que sólo permiten el paso de una persona cada vez que son utilizadas. En la barbería hay un número indeterminado (nunca se agotan) de sillas de 2 tipos:

a) **Sillas donde los clientes esperan a que entren los barberos.**
b) **Sillas de barbero donde los clientes esperan hasta que termina su corte de pelo.**

No hay espacio suficiente para que más de una persona (barbero o cliente) pueda moverse dentro de la barbería en cada momento, por ejemplo, si los clientes se dan cuenta que ha entrado un barbero, entonces sólo 1 cliente puede levantarse y dirigirse a una silla de tipo (b); un barbero, por su parte, no podría moverse para ir a cortarle el pelo hasta que el cliente se hubiera sentado. Cuando entra un cliente en la barbería, puede hacer una de estas 2 cosas:

- **Aguarda en una silla de tipo (a) y espera a que existan barberos disponibles para ser atendido, cuando sucede esto último, el cliente se levanta y vuelve a sentarse, esta vez en una silla de tipo (b), para esperar hasta que termine su corte de pelo.**
- **Se sienta directamente en una silla de tipo (b), si hay barberos disponibles.**

Un cliente no se levanta de la silla del barbero hasta que este le avisa abriéndole la puerta de salida. Un barbero, cuando entra en la barbería, aguarda a que haya clientes sentados en una silla de tipo (b) esperando su corte de pelo. Después revisa el estado de los clientes, siguiendo un orden numérico establecido, hasta que encuentra un cliente que espera ser atendido, cuando lo encuentra comienza a cortarle el pelo y él mismo pasa a estar ocupado. Cuando termina con un cliente, le abre la puerta de salida y espera a que el cliente le pague, después sale a la calle para refrescarse. El barbero no podrá entrar de nuevo en la barbería hasta que haya cobrado al cliente que justamente acaba de atender. No se admite que un cliente pague a un barbero distinto del que cortó el pelo. Resolver el problema anterior utilizando un solo monitor que defina los siguientes procedimientos:

Monitor Barberia;
procedure corte_de_pelo(i: numero_de_cliente);
// Lo llaman los procesos que simulan los clientes de la barbería, este método incluye la actuación completa del cliente desde que entra hasta que sale; se supone que cuando se espera en alguna silla del tipo que sea, deja que otros procesos puedan moverse (de uno en uno) dentro de la barbería
procedure siguiente_cliente(): numero_de_cliente;
// Es llamado por los procesos que simulan a los barberos; devuelve el número de cliente seleccionado; el criterio de selección consiste en revisar el estado de cada cliente, en orden de menor a mayor índice hasta encontrar uno que espera ser atendido; el índice de este cliente se pasara como argumento al método termina_corte_de_pelo(i: numero_de_cliente), llamado por el mismo barbero
procedure termina_corte_de_pelo(i: numero_de_cliente);
// Es llamado por los barberos para levantar a un cliente de la silla tipo-b y cobrarle antes de que salga de la barbería; el argumento i sirve, por tanto, para que el barbero sepa a que cliente tiene que cobrar

```
const var total_barberos: integer;
Monitor Barberia;
    // Número de barberos que están atendiendo a un cliente
    var barberos_ocupados: integer;
    // TRUE si al cliente del barbero i ya le ha cortado el pelo; FALSE en caso contrario
    var corte_finalizado[0…total_barberos-1]: boolean;
    // TRUE si el cliente del barbero i ya le ha pagado; FALSE en caso contrario
    var pagado[0…total_barberos-1]: boolean;
    // Cola de clientes esperando a sentarse en una silla tipo (b)
    var sentarse: condition;
    // Cola de clientes esperando a que un barbero les corte el pelo
    var corte: condition;
    // Cola de clientes esperando a salir de la barbería
    var salir: condition;
    // Cola de barberos esperando a realizar su servicio
    var realizar_servicio: condition;
```

```
procedure corte_de_pelo(i: numero_de_cliente);
    if (barberos_ocupados == total_barberos) then
        sentarse.wait();
    barberos_ocupados++;
    corte_finalizado[i] = false;

    realizar_servicio.signal();

    while( ! corte_finalizado[i] ) do
        corte.signal();
        corte.wait();

    barberos_ocupados++;
    pagado[i] = true;

    salir.signal();
    sentarse.signal();
```

```
procedure siguiente_cliente() : numero_de_cliente;
    var numero_de_cliente: integer;

    if(barberos_ocupados == 0) then
        realizar_servicio.signal();

    do
        numero_de_cliente++;
    while( ! corte_finalizado[numero_de_cliente] );

    return numero_de_cliente;
```

```
procedure terminar_corte_de_pelo(i: numero_de_cliente);
    corte_finalizado[i] = true;

    corte.signal();

    while( !pagado[i] )
        salir.signal();
        salir.wait();

    pagado[i] = true;
```

```
    begin
        barberos_ocupados = 0;
        corte_finalizado[total_barberos] = true;
        pagado[total_barberos] = false;
    end;
```

Ejercicio 62

Suponer un garaje de lavado de coches con dos zonas: una de espera y otra de lavado con 100 plazas. Un coche entra en la zona de lavado del garaje sólo si hay (al menos) una plaza libre, si no, se queda en la cola de espera. Si un coche entra en la zona de lavado, busca una plaza libre y espera hasta que es atendido por un empleado del garaje. Los coches no pueden volver a entrar al garaje hasta que el empleado que les atendió les cobre el servicio. Suponemos que hay más de 100 empleados que lavan coches, cobran, salen y vuelven a entrar al garaje. Cuando un empleado entra en el garaje comprueba si hay coches esperando ser atendidos (ya situados en su plaza de lavado), si no, aguarda a que haya alguno en esa situación. Si hay al menos un coche esperando, recorre las plazas de la zona de lavado hasta que lo encuentra, entonces lo lava, le avisa de que puede salir y, por último, espera a que le pague. Puede suceder que varios empleados han terminado de lavar sus coches y están todos esperando el pago de sus servicios, no se admite que un empleado cobre a un coche distinto del que ha lavado. También hay que evitar que al entrar 2 ó más empleados a la zona de lavado, habiendo comprobado que hay coches esperando, seleccionen a un mismo coche para lavarlo.

Se pide: programar una simulación de la actuación de los coches y de los empleados del garaje, utilizando un monitor con señales urgentes y los procedimientos que se dan a continuación. Se valorará más una solución al problema anterior que utilice un número menor de signal()'s.

```
Monitor Garaje;
procedure lavado_de_coche(i: 1..100);
// Lo llaman los procesos que simulan los coches, incluye la actuación completa del coche desde que entra en el garaje hasta que sale; se supone que cuando un coche espera, deja a los otros coches que se puedan mover dentro del garaje
procedure siguiente_coche(): positive;
// Es llamado por los procesos que simulan los empleados; devuelve el número de plaza donde hay un coche esperando ser lavado
procedure terminar_y_cobrar(i: 1..100);
// Es llamado por los empleados para avisar a un coche que ha terminado su lavado; terminara cuando se reciba el pago del coche que ocupa la plaza cuyo identificador se pasó como argumento
```

```
Monitor Garaje;
    // Número de plazas libres en la zona de lavado
    var plazas_libres: integer;
    // Número de coches esperando para ser lavados
    var coches_esperando: integer;
    // TRUE si el coche de la plaza i ya ha sido lavado; FALSE en caso contrario
    var lavado_finalizado[100]: boolean;
    // TRUE si el coche de la plaza i ya ha pagado; FALSE en caso contrario
    var pagado[100]: boolean;
    // Cola de coches esperando para entrar a la zona de lavado
    var entrar: condition;
    // Cola de coches esperando para ser lavados
    var lavado: condition;
    // Cola de coches esperando para salir de la zona de lavado
    var salir: condition;
    // Cola de empleados esperando para comenzar su servicio
    var realizar_servicio: condition;
```

```
procedure lavado_de_coche(i: 1…100);
    if(plazas_libres == 0) then
        entrar.wait();
    plazas_libres--;
    coches_esperando++;
    lavado_finalizado[i] = false;

    realizar_servicio.signal();

    while( !lavado_finalizado[i] ) do
        lavado.signal();
        lavado.wait();

    plazas_libres++;
    pagado[i] = true;

    salir.signal();
    entrar.signal();
```

```
procedure siguiente_coche() : positive;
    var j: integer;

    if(coches_esperando == 0) then
        comenzar.wait();

    coches_esperando--;

    do
        j++;
    while( !lavado_finalizado[j] );

    return j;
```

```
procedure terminar_y_cobrar(i: 1…100);
    finalizado[i] = true;

    lavado.signal();

    while( !pagado[i] )
        salir.signal();
        salir.wait();

    pagado[i] = true;
```

```
begin
    plazas_libres = 100;
    esperando = 0;
    terminado[100] = true;
    pagado[100] = false;
end;
```

Ejercicio 63

Demostrar que el monitor "productor-consumidor" es correcto utilizando las reglas de corrección de la operación c.wait() y c.signal() para señales desplazantes. Considerar el siguiente invariante del monitor IM: I == { 0 ≤ n ≤ MAX }.

```
Monitor Buffer;
    var n, frente, atrás: integer;
    var no_vacio, no_lleno: condition;
    var buf: array[1…MAX] of tipo_dato;
```

```
procedure insertar(var d: tipo_dato);
    begin
        { 0 ≤ n ≤ MAX }
        if(atrás mod(MAX) + 1 == frente) then        // El buffer está lleno
            { 0 ≤ n ≤ MAX }
            no_lleno.wait();
            { 0 ≤ n ≤ MAX }
        else                                         // El buffer no está lleno
            { 0 ≤ n ≤ MAX }
            NULL;
            { 0 ≤ n ≤ MAX }
        { 0 ≤ n ≤ MAX }
        buf[atras] = d;
        atrás = atrás mod(MAX) + 1;
       { 0 ≤ n ≤ MAX }
       n++;
       { 0 < n ≤ MAX + 1 }
       no_vacio.signal();
       { 0 ≤ n ≤ MAX } ⇒ { IM }
    end;
```

```
procedure retirar(var x: tipo_dato);
   begin
      { 0 ≤ n  ≤ MAX }
      if(frente == atras) then     // El buffer está vacío
         { 0 ≤ n  ≤ MAX }
         no_vacio.wait();
         { 0 < n ≤ MAX + 1 }
      else                         // El buffer tiene al menos un elemento
         { 0 < n  ≤ MAX }
         NULL;
         { 0 < n  ≤ MAX }
      { 0 < n ≤ MAX }
      x = buf[frente];
      frente = frente mod(MAX) + 1;
      { 0 < n ≤ MAX }
      n--;
      { 0 ≤ n ≤ MAX }
      no_lleno.signal();
      { 0 ≤ n ≤ MAX } ⇒ { IM }
   end;
```

```
   begin
      {V}
      frente = 1;
      atrás = 1;
      n = 0;
      { 0 ≤ n ≤ MAX } ⇒ { IM }
   end;
```

Ejercicio 64

Demostrar la corrección de un monitor que implemente las operaciones de acceso al buffer circular para el problema del productor-consumidor utilizando el siguiente invariante:

$0 \leq n \leq N$; No hay procesos bloqueados

$0 > n$; Hay $|n|$ consumidores bloqueados

$n > N$; Hay $(n - N)$ productores bloqueados

Escribir un monitor que cumpla el invariante anterior, es decir:

- **Se haga no_vacio.signal() solamente cuando haya consumidores bloqueados.**
- **Se haga no_lleno.signal() solamente cuando haya productores bloqueados.**

Se supone que el buffer tiene N posiciones y se utilizan las dos señales mencionadas anteriormente. No está permitido utilizar la operación c.queue() para saber si hay procesos bloqueados en alguna cola de variable condición. Nótese que el invariante del monitor Buffer comprende ahora 3 propiedades. La interpretación de este nuevo invariante es más amplia que la del IM del problema 63: los valores negativos de "n" representan (en valor absoluto) el número de procesos consumidores bloqueados, cuando el buffer está vacío y (n-N) es idénticamente igual al número de productores bloqueados cuando está lleno; cuando el valor de "n" se mantiene entre los límites: 0, N tendríamos, por tanto, y como caso particular, que se cumpliría el IM en las mismas condiciones del problema 63.

```
Monitor Buffer;
    var n, frente, atrás: integer;
    var no_vacio, no_lleno: condition;
    var buf: array[1...N] of tipo_dato;
```

```
procedure insertar(var d: tipo_dato);
    begin
        { IM }
        if(atrás mod(N) + 1 == frente) then          // El buffer está lleno
            { IM }
            n++;
            { (0 < n ≤ N+1) OR (0 ≥ n) OR (n > N+1) } ⇒ { IM }
            no_lleno.wait();
            { IM }
        { IM }
        buf[atras] = d;
        atrás = atrás mod(N) + 1;
        { IM }
        n++;
        { (0 < n ≤ N+1) OR (0 ≥ n) OR (n > N+1) } ⇒ { IM }
        if(n < 0) then
            { (n < 0) AND (IM) } ⇒ { n < 0 } ⇒ { IM }
            n++;
            { (0 < n ≤ N+1) OR (0 ≥ n) OR (n > N+1) } ⇒ { IM }
            no_vacio.signal();
            { IM }
        { IM }
    end;
```

```
procedure retirar(var x: tipo_dato);
   begin
      { IM }
      if(frente == atras) then     // El buffer está vacío
         { IM }
         n--;
         { ( -1 ≤ n < N) OR (-1 > n) OR (n ≥ N) } ⇒ { IM }
         no_vacio.wait();
         { IM }
      { IM }
      x = buf[frente];
      frente = frente mod(N) + 1;
      { IM }
      n--;
      { ( -1 ≤ n < N) OR (-1 > n) OR (n ≥ N) } ⇒ { IM }
      If(n > N) then
         { (n > N) AND (IM) } ⇒ { n > N } ⇒ { IM }
         n--;
         { ( -1 ≤ n < N) OR (-1 > n) OR (n ≥ N) } ⇒ { IM }
         no_lleno.signal();
         { IM }
      { IM }
   end;
```

```
   begin
      {V}
      frente = 1;
      atrás = 1;
      n = 0;
      { IM }
   end;
```

Ejercicio 65

Considerar el programa concurrente mostrado más abajo. En dicho programa hay 2 procesos, denominados P1 y P2, que intentan alternarse en el acceso al monitor M. La intención del programador al escribir este programa era que el proceso P1 esperase bloqueado en la cola de la señal p, hasta que el proceso P2 llamase al procedimiento M.sigue() para desbloquear al proceso P1; después P2 se esperaría hasta que P1 terminase de ejecutar M.stop(), tras realizar algún procesamiento se ejecutaría q.signal() para desbloquear a P2. Sin embargo el programa se bloquea.

```
Monitor M ( ) {
    cond p, q;
    procedure stop {
        begin
            p.wait();
            .......
            q.signal();
        end;
    }
    procedure sigue {
        begin
            .......
            p.signal();
            q.wait();
        end;
    }
    begin
    end;
}
```

```
Proceso P1;
    begin
        while TRUE do
            ...
            M.stop();
            ...
    end;
```

```
Proceso P2;
    begin
        while TRUE do
            ...
            M.sigue();
            ...
    end;
```

a) **Encontrar un escenario en el que se bloquee el programa.**

Supongamos que P2 se ejecuta antes que P1. Entonces P2 llama a la función sigue() del monitor M, de forma que se ejecuta un signal() en la cola p, la cual estará vacía y, a continuación, se ejecutará un wait() en la cola q haciendo que P2 se quede bloqueado en dicha cola. Después se ejecutará P1, el cual llamará a la función stop() del monitor M, de forma que se ejecuta un wait() en la cola p haciendo que P1 se quede bloqueado en dicha cola. De esta manera, P1 se queda bloqueado en la cola 1 y P2 se queda bloqueado en la cola q, produciéndose así un bloqueo del programa.

b) **Modificar el programa para que su comportamiento sea el deseado y se eviten interbloqueos.**

```
Monitor M ( ) {
    cond p, q;
    procedure stop {
        begin
            if ( !q.queue() )
                p.wait();
            .......
            q.signal();
        end;
    }
    procedure sigue {
        begin
            .......
            p.signal();
            if ( !p.queue() )
                q.wait();
        end;
    }
    begin
    end;
}
```

```
Proceso P1;
    begin
        while TRUE do
            ...
            M.stop();
            ...
    end;
```

```
Proceso P2;
    begin
        while TRUE do
            ...
            M.sigue();
            ...
    end;
```

Ejercicio 66

Indicar con qué tipo (o tipos) de señales de los monitores (SC, SW o SU) sería correcto el código de los procedimientos de los siguientes monitores que intentan implementar un semáforo FIFO. Modificar el código de los procedimientos de tal forma que pudieran ser correctos con cualquiera de los tipos de señales anteriormente mencionados.

```
Monitor Semaforo {
    int s;
    cond c;

    void* P(void* arg) {
        if (s == 0)
            c.wait();
        s = s - 1;
    }

    void* V(void* arg) {
        s = s + 1;
        c.signal();
    }
}
begin
    s = 0;
end;
```

Este monitor sería correcto con:

- Señalar y Esperar (SW)
- Señalar y Espera Urgente (SU)

```
Monitor Semaforo {
    int s;
    cond c;

    void* P(void* arg) {
        while (s == 0)
            c.wait();
        s = s - 1;
    }

    void* V(void* arg) {
        notifyAll();
        s = s + 1;
    }
}
begin
    s = 0;
end;
```

Este monitor sería correcto con:

- Señalar y Continuar (SC)

```
Monitor Semaforo {
    int s;
    cond c;

    void* P(void* arg) {
        if (s == 0)
            c.wait();
        else
            s = s - 1;
    }

    void* V(void* arg) {
        if ( c.queue() )
            c.signal();
        else
            s = s + 1;
    }
}
begin
    s = 0;
end;
```

Este monitor sería correcto con:

- Señalar y Continuar (SC)
- Señalar y Esperar (SW)
- Señalar y Espera Urgente (SU)

Como el último monitor funciona con los tres tipos de señales, lo dejamos intacto. Una modificación para el primer y el segundo monitor, con el fin de que funcionen con los tres tipos de señales, sería la siguiente:

```
Monitor Semaforo {
    int s;
    cond c;

    void* P(void* arg) {
        while (s == 0)
            c.wait();
        s = s - 1;
    }

    void* V(void* arg) {
        s = s + 1;
        c.signal();
    }
}
begin
    s = 0;
end;
```

Ejercicio 67

Escribir el código de los procedimientos P() y V() de un monitor que implemente un semáforo general con el siguiente invariante: {s ≥ 0} V {s = s0 + nV – nP } y que sea correcto para cualquier semántica de señales. La implementación ha de asegurar que nunca se puede producir "robo de señal" por parte de las hebras que llamen a las operaciones del monitor semáforo anterior.

```
Monitor Semaforo;
    var s: integer;
    var c: condition;

    procedure P();
        begin
            if (s == 0) then
                c.wait();
            else
                s = s - 1;
        end;

    procedure V();
        begin
            if ( c.queue() ) then
                c.signal();
            else
                s = s + 1;
        end;

begin
    s = 0;
end;
```

Ejercicio 68

Suponer un número desconocido de procesos consumidores y productores de mensajes de una red de comunicaciones muy simple. Los mensajes se envían por los productores llamando a la operación broadcast(int m) (el mensaje se supone que es un entero), para enviar una copia del mensaje m a las hebras consumidoras que previamente hayan solicitado recibirlo, las cuales están bloqueadas esperando. Otra hebra productora no puede enviar el siguiente mensaje hasta que todas las hebras consumidoras no reciban el mensaje anteriormente enviado. Para recibir una copia de un mensaje enviado, las hebras consumidoras llaman a la operación int fetch(). Mientras un mensaje se esté transmitiendo por la red de comunicaciones, nuevas hebras consumidoras que soliciten recibirlo lo reciben inmediatamente sin esperar. La hebra productora, que envió el mensaje a la red, permanecerá bloqueada hasta que todas las hebras consumidoras solicitantes efectivamente lo hayan recibido. Se pide programar un monitor que incluya entre sus métodos las operaciones: broadcast(int m), int fetch(), suponiendo una semántica de señales desplazantes y SU.

```
Monitor RedComunicaciones;
    var mensaje: integer;
    var c, b: condition;
```

```
procedure broadcast(var m: integer);
    begin
        if( !b.queue() ) then
            mensaje = m;
            if (esperando > 0) then
                c.signal();
                b.wait();
    mensaje = -1;
    end;
```

```
procedure fetch(var x: integer);
    begin
        esperando = esperando + 1;
        if (mensaje = -1) then
            c.wait();
        x = mensaje;
    end;
```

```
procedure terminar();
    begin
        esperando = esperando – 1;
        if (esperando > 0) then
            c.signal();
        else if ( b.queue() ) then
            b.signal();
    end;
```

```
begin
    mensaje = -1;
end;
```

Ejercicio 69

Suponer un sistema básico de asignación de páginas de memoria de un sistema operativo que proporciona 2 operaciones: adquirir(int n) y liberar(int n) para que los procesos de usuario puedan obtener las páginas que necesiten y, posteriormente, dejarlas libres para ser utilizadas por otros. Cuando los procesos llaman a la operación adquirir(int n), si no hay memoria disponible para atenderla, la petición quedaría pendiente hasta que exista un número de páginas libres suficiente en memoria. Llamando a la operación liberar(int n) un proceso convierte en disponibles n páginas de la memoria del sistema. Suponemos que los procesos adquieren y devuelven páginas del mismo tamaño a un área de memoria con estructura de cola y en la que suponemos que no existe el problema conocido como fragmentación de páginas de la memoria. Se pide programar un monitor que incluya las operaciones anteriores suponiendo semántica de señales desplazantes y SU.

a) **Resolverlo suponiendo orden FIFO estricto para atender las llamadas a la operación de adquirir páginas por parte de los procesos.**
b) **Relajar la condición anterior y resolverlo atendiendo las llamadas según el orden: petición pendiente con "menor número de páginas primero" (SJF).**

Nota: suponer orden FIFO estricto: suponer que hay una petición pendiente de 30 páginas y la memoria disponible es de 20 páginas; si posteriormente llega una petición de 20 páginas, tendrá que esperar a que exista memoria suficiente para atender la petición de 30 páginas primero.

a)

```
const var memoria_inicial: integer
Monitor PaginasDeMemoriaFIFO;
    var memoria_disponible: integer;
    var peticiones_pendientes: boolean;
    var esperar, obtener: condition;
```

```
procedure solicitar();
    begin
        if( peticiones_pendientes OR obtener.queue() ) then
            esperar.wait();
        peticiones_pendientes = true;
    end;
```

```
procedure adquirir(var n: integer);
    begin
        while(memoria_disponible < n) do
            obtener.wait();
        memoria_disponible = memoria_disponible - n;
        peticiones_pendientes = false;
        if ( esperar.queue() ) then
            esperar.signal();
    end;
```

```
procedure liberar(var n: integer);
    begin
        memoria_disponible = memoria_disponible + n;
        if ( obtener.queue() ) then
            obtener.signal();
    end;
```

```
    begin
        memoria_disponible = memoria_inicial;
        peticiones_pendientes = false;
    end;
```

b)

En este caso, no hace falta el procedimiento "solicitar()", luego, desaparece la variable booleana permanente "peticiones_pendientes". Además, también desaparece la cola condición "esperar".

```
const var memoria_inicial: integer
Monitor PaginasDeMemoriaSJF;
    var memoria_disponible: integer;
    var obtener: condition;
```

```
procedure adquirir(var n: integer);
    begin
        while(memoria_disponible < n) do
            obtener.wait();
        memoria_disponible = memoria_disponible - n;
        if ( obtener.queue() ) then
            obtener.signal();
    end;
```

```
procedure liberar(var n: integer);
    begin
        memoria_disponible = memoria_disponible + n;
        if ( obtener.queue() ) then
            obtener.signal();
    end;
```

```
    begin
        memoria_disponible = memoria_inicial;
    end;
```

Ejercicio 70

Diseñar un controlador para un sistema de riegos que proporcione servicio cada 72 horas. Los usuarios del sistema de riegos obtienen servicio del mismo mientras un depósito de capacidad máxima igual a C litros tenga agua. Si un usuario llama al procedimiento abrir_cerrar_valvula(cantidad: positive) y el depósito está vacío, entonces ha de señalarse al proceso controlador y la hebra que soporta la petición del citado usuario quedará bloqueada hasta que el depósito esté otra vez completamente lleno. Si el depósito no se encontrase lleno o contiene menos agua de la solicitada, entonces el riego se llevará a cabo con el agua que haya disponible en ese momento. La ejecución del procedimiento: control_en_espera() mantiene bloqueado al controlador mientras el depósito no esté vacío. El llenado completo del depósito se produce cuando el controlador llama al procedimiento: control_rellenando, de tal forma que cuando el depósito esté completamente lleno (=C litros) se ha de señalar a las hebras-usuario bloqueadas para que terminen de ejecutar las operaciones de "abrir y cerrar válvula" que estaban interrumpidas.

a) **Programar las 3 operaciones mencionadas en un monitor que asumen semántica de señales desplazantes y SU.**
b) **Demostrar que las hebras que llamen a las operaciones del monitor anterior nunca pueden llegar a entrar en una situación de bloqueo indefinido.**

```
process P(i: 1..N);
   begin
      while true do begin
         Riegos.abrir_cerrar_valvula(litros);
         \\Regar ...
      end;
   end;
```

```
process controlador;
   begin
      while true do begin
        \\El deposito esta inicialmente lleno
         \\Esperar 72 horas
         Riegos.control_en_espera;
         Riegos.control_rellenando;
      end;
   end;
```

a)

IM = (litros_deposito >= 0)

```
const var C: integer
Monitor Riego;
   var litros_deposito: integer;
   var deposito_lleno, control_deposito: condition;
```

```
procedure abrir_cerrar_valvula(var cantidad: positive);
    begin
        { IM }
        if(litros_deposito == 0) then
            { litros_deposito = 0 } ⇒ { IM }
            if( control_deposito.queue() ) then
                { IM }
                control_deposito.signal();
                { IM }
            { IM }
            deposito_lleno.wait();
            { IM }
        { IM }
        litros_deposito = max(litros_deposito – cantidad, 0);
        { IM }
    end;
```

```
procedure control_rellenando();
    begin
        { IM }
        litros_deposito = C;
        { litros_deposito = C } ⇒ { IM }
        deposito_lleno.signal();
        { IM }
    end;
```

```
procedure control_en_espera();
    begin
        { IM }
        control_deposito.wait();
        { IM }
    end;
```

```
begin
    { IM }
    litros_deposito = 0;
    { litros_deposito = 0 } ⇒ { IM }
end;
```

b)

Observamos en el código que nunca se va a poder dar que todas las hebras de nuestro programa estén bloqueadas en las colas de condición (produciéndose así un bloqueo en el programa) ya que antes de hacer "deposito_lleno.wait()" comprobamos si hay alguna hebra esperando en la cola condición "control_deposito" y en el caso de que la hubiera, la señalamos para que salga de dicha cola. También vemos que se mantiene el invariante del monitor, IM = (litros_deposito >= 0).