61.

```
Monitor Barberia;
        var     move: boolean;
                cortes[ ], pagos[ ], movimiento, silla: condition;
                barberos_disp, clientes_disp: integer;
                clientes: <integer> array       // -1 begin, 0 si está esperando, 1 si está atendido
                // barberos: <integer> array    // -1 si el barbero está disponible; i si está atendiendo
al                                              // cliente i
procedure corte_de_pelo(i: numero_de_cliente);
        begin
                if(barbero_disp==0)
                  then
                    if(movimiento)                      //Paso a una silla a)
                      then
                        movimiento.wait();
                    movimiento = true;
                    //tiempo aleatorio de movimiento
                    movimiento = false;
                    if(movimiento.queue())
                        then movimiento.signal();
                    silla.wait();                       //Espera de un barbero disponible
                if(movimiento)                          //Paso a una silla b)
                      then
                        movimiento.wait();
                    movimiento = true;
                    //tiempo aleatorio de movimiento
                    movimiento = false;
                    if(movimiento.queue())
                        then movimiento.signal();
                clientes[i] = 0;
                clientes_disp++;
                barberos.signal();
                cortes[i].wait();
                clientes[i] = 1;                        //Indica que está atendido
                clientes_disp--;
                //Se produce corte de pelo
                cortes[i].wait();                       //espera para salir
                pagos[i].signal();                      //avisa de que ha pagado
                if(movimiento)                          //sale tras pagar
                  then
                    movimiento.wait();
                movimiento = true;
                //tiempo aleatorio de movimiento
                movimiento = false;
                clientes[i] = -1;
                if(movimiento.queue())
                        then movimiento.signal();
        end;
```

//lo llaman los procesos que simulan los clientes de la barberia, este metodo incluye la actuacion 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 barberia

```
procedure siguiente_cliente(): numero_de_cliente;
        barbero_disp++;
        if(silla.queue())
                then silla.signal();
        if(clientes_disp == 0)
                barberos.wait();
        barberos_disp--;
        fin: boolean;
        devolver: integer;
        fin = false;
        for(i=0; i<num_clientes and !fin; i++){
                if(clientes[i] == 0) {
                        then devolver = i;
                                fin = true;
                }
        }
        cortes[i].signal();
        //barberos.wait();
        return devolver;
```

//es llamado por los procesos que simulan a los barberos; devuelve el numero de cliente seleccionado; el criterio de seleccion consiste en revisar el estado de cada cliente, en orden de menor a mayor indice hasta encontrar uno que espera ser atendido; el indice de este cliente se pasara como argumento al metodo termina_corte_de_pelo(i: numero_de_cliente), llamado por el mismo barbero

```
procedure termina_corte_de_pelo(i: numero_de_cliente);
        cortes[i].signal()      //avisa al cliente i de que ha terminado el corte y puede salir.
        pagos[i].wait();        //espera a que el cliente i le pague
        if(movimiento)          //se mueve a la calle
                then
                  movimiento.wait();
                movimiento = true;
                //tiempo aleatorio de movimiento
                movimiento = false;
                clientes[i] = -1;
                if(movimiento.queue())
                        then movimiento.signal();
```

//es llamado por los barberos para levantar a un cliente de la silla tipo-b y cobrarle antes de que salga de la barberia; el argumentoi sirve, por tanto, para que el barbero sepa a que cliente tiene que cobrar

```
begin
        clientes_disp = 0; barberos_disp = 0; for(i=0; i<num_clientes; i++){clientes[i] = -1}
end;
```

*Posibilidad: utilizar un vect. Dinámico de barberos donde por cada barbero que entra se añade una componente y en cada componente del vector se almacena el cliente asociado.

62.

```
Monitor Garaje;
        pagos [100], lavado[100], espera, empleado: cond;
        coches_disp, plazas_disp, plazas[100]: integer;
```

```
procedure lavado_de_coche(i: 1..100);
        if(plazas_disp == 0)
                espera.wait();
        plazas_disp--;
        bool fin = false;
        for(i=0; i<100 and !fin; i++){
                if(plazas[i] == 0)
                        plazas[i] = 1;
                        fin = true;
                        coches_disp++;
                        lavado[i].wait();
        }
        pagos[i].signal();
        plazas_disp++;
//lo llaman los procesos que simulan los coches, incluye la actuacion
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;
        if(coches_disp == 0)
                empleados.wait();
        coche: integer;
        fin :bolean;
        fin = false;
        for(i=0; i<100 and !fin; i++){
                if(plazas[i] == 1)
                        plazas[i] = 2;
                        fin = true;
                        coche = i;
                        coches_disp--;
        }
        return coche;
//es llamado por los procesos que simulan los empleados; devuelve el numero
de plaza donde hay un coche esperando ser lavado

procedure terminar_y_cobrar(i: 1..100);
        lavado[i].signal();
        pagos[i].wait();
//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 paso como argumento

begin
        plazas_disp = 100;  coches_disp = 0;
        for(i=0; i<100; i++) {plazas[i] = 0}
end;
```

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 }.

Condiciones de sincronización: no_lleno → 0<= n < MAX
no_vacio → 0< n <= MAX

Verificamos la inicialización del monitor:

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

Verificamos los procedimientos del monitor:

```
procedure insertar(d: tipo_dato);
    begin
        {IM}
        if (atras mod MAX +1 == frente)
            then
                {atras mod MAX +1 == frente→ n==MAX ( IM ) }
                no_lleno.wait()
                {0<= n <MAX}
        else
            {0<= n <MAX}
            NULL;
            {0<= n <MAX}
        {0<= n <MAX}
        buf[atras]= d;
        {0<= n <MAX}
        atras = atras mod MAX + 1;
        {0<= n <MAX}
        n++;
        {0< n <=MAX}
        no_vacio.signal();
        {0<= n <= MAX → IM}
    end;

procedure retirar(var x: tipo_dato);
    begin
        {IM}
        if ( frente==atras)
            then
                {frente==atras → n==0 ( IM ) }
                no_vacio.wait();
                {0 < n <= MAX}
        else
            {0 < n <= MAX}
            NULL;
            {0 < n <= MAX}
        {0 < n <= MAX}
        x= buf[frente];
        {0 < n <= MAX}
        frente= frente mod MAX 1;
```

```
                {0 < n <= MAX}
                n--;
                {0 <= n < MAX}
                no_lleno.signal();
                {0 <= n <= MAX → IM}
        end;
```

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 ≤ n ≤ 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 novacio.signal() solamente cuando haya consumidores bloqueados.**
- **Se haga nolleno.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 Bufer 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 NUEVO:

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

procedure insertar(d: tipo_dato);
  begin
        if (atras mod N +1 == frente)
                then
                  n++;
                  no_lleno.wait();
        else NULL;
        buf[atras]= d;
        atras = atras mod N + 1;
        n++;
        if (n < 0)
                then
                  no_vacio.signal();
  end;

procedure retirar(var x: tipo_dato);
  begin
        if ( frente==atras)
                then
                  n--;
                  no_vacio.wait();
```

```
        else NULL;
        x= buf[frente];
        frente= frente mod N 1;
        n--;
        if (n > N)
                then
                  no_lleno.signal();
  end;

begin
        frente=1;
        atras= 1;
        n= 0;
end;
```

CORRECCIÓN:

Condiciones de sincronización: no_lleno → n < N
no_vacio → 0 < n

Verificamos la inicialización del monitor:

```
        {V}
        begin
                frente=1;
                atras= 1;
                n= 0;
        end;
        {n = 0 → 0 <= n <= N → IM}
```

Verificamos los procedimientos del monitor:

```
procedure insertar(d: tipo_dato);
  begin
        {IM}
        if (atras mod N +1 == frente)
                then
                  {atras mod N +1 == frente → n >= N → IM}
                  n++;
                  {n > N → IM}
                  no_lleno.wait()
                  {n < N}
        else {n < N} NULL {n < N};
        {n < N} buf[atras]= d; {n < N}
        {n < N} atras = atras mod N + 1; {n < N}
        {n < N} n++; {n <= N}
        if (n < 0)
                then
                  {n < 0}
                  no_vacio.signal();
                  {n<=0 → IM}
  end;
procedure retirar(var x: tipo_dato);
  begin
        if ( frente==atras)
                then
```

```
                {frente==atras → n < 0 ó n==0 → IM}
                n--;
                {n < 0 →  IM}
                no_vacio.wait();
                {0 < n}
        {0 < n} else NULL{0 < n}
        {0 < n} x= buf[frente]; {0 < n}
        {0 < n} frente= frente mod N 1; {0 < n}
        {0 < n} n--; {0 < n}
        if (n > N)
                then
                  {n > N}
                  no_lleno.signal();
                  {n >= N → IM}
  end;
```

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 seãl 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.
(a) Encontrar un escenario en el que se bloquee el programa.
(b) Modificar el programa para que su comportamiento sea el deseado y se eviten interbloqueos.

a)

Si comienza a ejecutarse P2 antes que P1, la ejecución de p.signal() del procedimiento “sigue” no tiene efecto y P2 se bloquea tras ejecutar q.wait(). Entonces P1 ejecuta la primera instrucción del procedimiento “stop” (p.wait()) y se queda también bloqueado en la cola de espera de p. De esta forma, ambos quedan bloqueados sin que ninguno pueda ejecutar una orden signal y se produce un bloqueo del programa.
(Consideramos que el procesamiento que ejecuta P1 en “stop” es suficiente como para que siempre se ejecute q.wait() en “sigue” antes que q.signal() en “stop”, sino, esto generaría otro bloqueo.)

b)

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

```
end;
}
```
En caso de que el procesamiento de P1 pueda terminar antes de que P2 ejecute q.wait() en “sigue”, habría que incluir otro bucle → while(q.empty()) {} antes de q.signal() en stop para solucionarlo.

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.

a) La semántica SC No es correcta para este semáforo FIFO ya que como los procesos señalados vuelven a competir en la cola del monitor, No asegura que el código tras el wait se realice en orden correcto. Las semánticas SW y SU Si son correctas ya que aseguran que el proceso señalado reanuda inmediatamente la ejecución de código del monitor tras la operación wait (No afecta mucho la diferenciación SW ó SU ya que no hay código tras signal).

b) Ninguna semántica es correcta. Dado que usa notifyAll() nos centramos en la tipo SC y vemos que como los procesos despertados vuelven a competir en la cola del monitor, no está asegurada la readquisición del mismo en el orden adecuado.

c) Puesto que no hay código ni después del wait ni después del signal, el funcionamiento correcto de este monitor no depende de la semántica utilizada, todas son correctas. No afecta que no haya bucle para la semántica SC ya que siempre se vuelve a comprobar la condición.

67. Escribir el código de los procedimientos P() y V() de un monitor que implemente un semáforo general con el siguiente invariante: {s ≥ 0} ∨ {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.

Suponemos s0 > 0 siempre por la definición de semáforo.

```
Monitor Semaforo{
        int s, nv, np;
        cond c;
void* P(void* arg){
        np++;
        while (s == 0  || s != s0 + nv – np) {
                c. wait();
        }
        s= s-1;
}
void* V(void* arg){
        nv++;
        s = s+1;
        c.signal();
}
begin
        s = s0;
        nv = 0;
        np = 0;
end;
```

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 envíado. 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.

Prograñación en pseudocódigo:

```
Monitor Mensajes {
        const  var      n_productores = P: integer;
                        n_consumidores = C: integer;
        var     contador_c: integer;
                contador_espera: integer;
                mensaje: integer;
                producir, consumir, transmitir: condition;
                transmitiendo: boolean;
}

procedure broadcast (m: integer) {
        if(transmitiendo)
                then producir.wait();
        transmitiendo = true;
        mensaje = m;
        consumir.signal();
        transmitir.wait();
        transmitiendo = false;
        if(producir.queue())
                then producir.signal();
}

procedure fetch( ): integer {
        var     devolver: integer;
        if(transmitiendo)
                then devolver = mensaje; return devolver;
        else{
                contador_espera++;
                consumir.wait();
                devolver = mensaje;
                contador_c++;
                if(contador_c == contador_espera)
                        then transmitir.signal();
                else
                        consumir.signal();
                return devolver;
        }
```

```
}

begin           //inicialización
        contador_c = 0;
        contador_espera = 0;
        transmitiendo = false;
end;
```

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 posterioremnte llega una petición de 20 páginas, tendrá que esperar a que exista memoria suficiente para atender la petición de 30 páginas primero.

a)

```
Monitor Memoria : public HoareMonitor {
        int const M;
        int ocupada;
        bool peticion;
        cond memoria_disp;          //cond. Sincron. → M – ocupada >= peticion
        cond peticiones;            //cond. Sincron. → peticion == false
}
void adquirir (int n) {
        if(peticion or memoria_disp.queue())
                peticiones.wait();
        peticion = true;
        while (M – ocupada < n)
                memoria_disp.wait();
        ocupada += n;
        peticion = false;
        if(peticiones.queue())
                peticiones.signal();
}
void liberar (int n) {
        ocupada –= n;
        if(memoria_disp.queue())
                memoria_disp.signal();
```

```
}
begin
        ocupada = 0;
        peticion = false;
end;
```

b)

```
Monitor MemoriaSJF : public HoareMonitor {
        int const M;
        int ocupada;
        cond memoria_disp;              //cond. Sincron. → M – ocupada >= peticion
}
void adquirir (int n) {
        while (M – ocupada < n)
                memoria_disp.wait(n);
        ocupada += n;
        if(memoria_disp.queue())
                memoria_disp.signal();
}
void liberar (int n) {
        ocupada –= n;
        if(memoria_disp.queue())
                memoria_disp.signal();
}
```

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.

a)

```
Monitor Riegos :public HoareMonitor {
        double const MAX = C;
        double gastada;
        cond valvula, controller;
}
begin
        gastada = 0;
end;
```

```
void abrir_cerrar_valvula(cantidad: positive) {
	if(gastada == MAX){
		controller.signal();
		valvula.wait();
	}
	if( positive >= MAX – gastada){
		gastada = MAX;
		controller.signal();
	}
	else {
		gastada += positive;
		if(valvula.queue())
			valvula.signal();
	}
}

void control_en_espera(){
	if (gastada < MAX)
		controller.wait();
}

void control_rellenando(){
	gastada = 0;
	valvula.signal();
}
```

b)

La hebra controladora se bloquea por defecto en un momento concreto pero es despertada en el momento que el deposito queda vacío. Por parte de las hebras-usuario, se bloquean solo cuando encuentran el depósito vacío, y previamente a su bloqueo despiertan a la hebra controladora que, justo a continuación, llenará el depósito y despertará a una hebra-usuario sin producirse bloqueo indefinido.

El caso en el que podría producirse bloqueo indefinido es si todas las hebras-usuario “perdieran señal” haciendo un signal inútil y después bloqueándose cuando la hebra controladora está en la espera de 72h y entonces esta también se bloquease tras dicha espera. Pero la condición del procedimiento control_en_espera impide que la hebra controladora se bloquee si el depósito está vacío que es la situación en la que las otras hebras-usuario se han bloqueado. Luego no se bloquea en ese caso y no se produce bloqueo indefinido.