



Nombre:

Curso:

Marcar en la siguiente tabla la respuesta correcta. (4 puntos)

+0.25 respuesta correcta,
-0.08 por error, (Cada tres errores se descuenta un acierto)
0 repuesta no contestada.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	D	A	C	D	C	C	C	B	D	A	B	A	A	B	C

- 1) Un interbloqueo (deadlock) se produce:
 - a) **cuando todos los procesos están esperando que ocurra un evento que nunca se producirá**
 - b) ninguna de las otras respuestas es cierta
 - c) si el resultado de la secuencia depende de la llegada relativa a algún punto crítico en la secuencia
 - d) cuando existe un grupo de procesos que nunca progresan pues no se les otorga tiempo de procesador para avanzar.
- 2)Cuál de las siguientes cuestiones han de resolverse en una llamada a procedimiento remoto:
 - a) La ejecución en espacios de direcciones de memoria diferentes
 - b) El paso de parámetros
 - c) La respuesta ante fallos de una máquina
 - d) **Todas las respuestas son válidas**
- 3) En la comunicación directa entre procesos es necesario::
 - a) **El emisor debe conocer al destinatario y el receptor al remitente**
 - b) No se requiere ningún tipo de identificación
 - c) Conocer el remitente del mensaje
 - d) Conocer el destinatario del mensaje.
- 4) En la comunicación asíncrona entre procesos:
 - a) La primitiva de envío bloqueará al emisor
 - b) Ambas primitivas de envío o recepción bloquearán a los procesos implicados
 - c) **La primitiva de recepción bloqueará al proceso si no hay datos en el buzón**
 - d) Ninguna primitiva de envío o recepción bloqueará a los procesos implicados.
- 5) En la comunicación síncrona entre procesos:
 - a) Ni emisor ni receptor esperan antes de iniciar la transmisión
 - b) El emisor espera siempre al receptor antes de iniciar la transmisión
 - c) El receptor espera siempre al emisor antes de iniciar la transmisión
 - d) **El primero que alcanza la primitiva de comunicación deberá esperar hasta que el otro alcance la suya antes de iniciar la transmisión.**
- 6) La inicialización de la variable de un semáforo:
 - a) Ninguna de las respuestas es correcta
 - b) Puede inicializarse tantas veces como se quiera
 - c) **Sólo puede hacerse una única vez en su ciclo de vida**
 - d) Los semáforos se inician siempre al valor 0.
- 7) En el mecanismo de RPC, el resguardo o sustituto del procedimiento invocado se crea:
 - a) en el lado del servidor
 - b) en el lado del cliente.
 - c) **en el lado del cliente y en el lado del servidor**
 - d) La creación de resguardos o stubs no es una técnica utilizada en RPC
- 8) Las variables de condición en un monitor:
 - a) Son herramientas de más bajo nivel de programación
 - b) No ayudan más que los semáforos
 - c) **Son herramientas de más alto nivel de programación con una estructura que ayuda a la corrección del programa**
 - d) Implícitamente funcionan como pilas para el manejo de los bloqueos/desbloqueos.

- 9) Para un correcto funcionamiento de los procesos concurrentes se debe asegurar:
- La exclusión mutua, la sincronización y evitar el interbloqueo
 - Ninguna de las respuestas incluye todas las condiciones a asegurar**
 - La exclusión mutua y la sincronización
 - Sólo la exclusión mutua.
- 10) El algoritmo de Dekker:
- Presenta situaciones en las que puede no garantizar las propiedades de viveza
 - Es válido para "n" procesos con ligeras modificaciones
 - Está orientado a entornos
 - Está orientado a entornos centralizados.**

11) La siguiente solución al problema de los filósofos:

- Puede generar interbloqueo entre los procesos**
- Puede generar inanición en uno de los filósofos
- No resuelve el problema en ninguna circunstancia
- Resuelve el problema cumpliendo todas las propiedades requeridas

```
var
semaphore palillo[5]={1,1,1,1,1};

process type filosofo (i:integer);
begin
repeat
piensa;
wait (palillo[i]);
wait (palillo[(i+1)mod 5]);
come;
signal (palillo[i]);
signal (palillo[(i+1)mod 5]);
forever
end;
```

12) Dada la siguiente configuración de procesos, determinar la respuesta correcta:

- B se ejecutará siempre después de C
- D se ejecutará después de E y A**
- D se ejecutará siempre después de B y C
- A se ejecutará antes de F

```
process P0
Begin
A;
signal(s);
B;
End;
```

```
process P1
Begin
C;
wait(s);
wait(t);
D;
End;
```

```
process P2
Begin
E;
signal(t);
F;
End;
```

- 13) La asignación de procesadores físicos a hilos se realiza::
- Se hace a dos niveles, un primer nivel para asignar los hilos de usuario a los procesadores lógicos, y un segundo nivel para asignar los procesadores lógicos al procesador o procesadores físicos**
 - Directamente, por parte del planificador del Sistema Operativo
 - Indirectamente, asignando los procesadores lógicos a una CPU
 - Directamente, asignando la CPU al proceso del que forma parte un único hilo
- 14) La característica principal de un monitor es:
- Todas las funciones del monitor se ejecutan en exclusión mutua**
 - A través del monitor se consigue comunicar procesos.
 - Solucionan el problema de la sincronización entre procesos concurrentes
 - Sólo hay un proceso en el monitor en cada momento.
- 15) ¿Cuál de las siguientes afirmaciones es cierta?:
- El paralelismo puede desarrollarse en sistemas monoprocesador
 - El paralelismo es un tipo de concurrencia**
 - El paralelismo y la concurrencia son conceptos que no guardan relación alguna
 - La concurrencia es un tipo de paralelismo
- 16) La técnica de RPC asíncrona
- También es conocida como RPC síncrona extendida
 - La resolución a la RPC es bloqueante en servidor
 - la llamada a procedimiento no es bloqueante en el proceso cliente**
 - la llamada a procedimiento es bloqueante en el proceso cliente

3. Implementemos los procesos Santa, Reno y Duende y el programa principal

Respecto al uso de recursos compartidos que requerirán de exclusión mutua, la principal dificultad está en controlar que se configuran grupos de tres duendes. Para ello es necesario controlar el número de duendes que, en un determinado momento, necesitan la ayuda de Santa para empaquetar. Además, hará falta controlar el número de renos que, están disponibles. El acceso a estas variables ha de hacerse en exclusión mutua, para lo que usaremos un semáforo *mutex*, y otro semáforo *duendes_mutex* para uso exclusivo dentro del proceso *duendes*

Respecto a la sincronización entre los procesos, será necesario disponer de un semáforo para despertar a Santa Claus, otro más para notificar a los renos que se han de enganchar al trineo y un tercero para controlar la espera por parte de los grupos de duendes.

```
Program PoloNorte ()
VAR
    Duendes_para_grupo: integer; // Número de duendes listos para trabajar
    Renos_despiertos: integer; //Número de renos dispuestos a viajar
    Renos_sem: semaphore; // Para bloquear a los renos a la espera de que Santa esté listo
    Santa_sem: semaphore; // Para bloquear a Santa a la espera de ser despertado por renos o duendes
    Duendes_sem: semaphore; // Para bloquear a los duendes en grupos de tres
    Mutex: semaphore; // Para exclusión mutua entre los tres procesos
    Duendes_mutex: semaphore; // Para exclusión mutua entre procesos duendes

BEGIN
    Initial (santa_sem,0); // También se puede inicializar a 1
    Initial (renos_sem,0);
    Initial (duendes_sem,0);
    Initial (mutex,1);
    initial (duendes_mutex,1); // Los semáforos de exclusión mutua se inicializan a 1
    Duendes_para_grupo =0;
    Renos_despiertos=0;
    COBEGIN
        Santa;
        For (i=0; i<N_RENOS; i++) renos (i); // N_RENOS=9
        For (i=0; i<N_DUENDES; i++) renos (i); // N_DUENDES=N
    END;
END.
```

```
Process Santa ()
Var
    I: integer;

BEGIN
    REPEAT
        Wait (santa_sem);
        Wait (mutex); // Exclusión mutua para chequeo de variables
        IF (renos_despiertos == 9) { //Están los renos listos
            // Desbloquear a los renos a los renos...
            for (i = ; i < TOTAL_RENOS; i++) signal (renos_sem);
            // Repartir_regalos()
        }
        ELSE
            IF (duendes_para_grupo == 3) {
                // Desbloquear a los duendes para ayudarlos a fabricar juguetes
                FOR (i = ; i < 3; i++) signal (duendes_sem);
                //Fabricar_juguetes()
            }

            Signal (mutex);
        FOREVER
    END;
```

```
Process Reno (id:integer)
BEGIN
    REPEAT
        // Estar de vacaciones()
        Wait (mutex);
        renos_despiertos ++ ;
        IF (renos_despiertos == 9) signal (santa_sem); // Despertar a Santa Claus
        Signal (mutex);
        Wait (renos_sem); // Esperar a que Santa Claus monte el trineo
        // Salir con Santa a repartir regalos
    FOREVER
END;
```

```
Process Duende (id:integer)
BEGIN
    REPEAT
        Wait (duendes_mutex); // Esperar a que haya grupo disponibles
        Wait (mutex); // Exclusión mutua
        duendes_para_grupo ++;
        IF (duendes_para_grupo == 3) signal (santa_sem); // Despertar a Santa
        ELSE signal (duendes_mutex); // No hay duendes suficientes, despertar al siguiente duende
        Signal (mutex);
        Wait (duendes_sem); // Espero a que Santa esté listo
        // Empaquetar el juguete
        // Cargar el juguete en el trineo

        // Ahora toca salir de forma ordenada
        Wait (mutex); // Para exclusión mutua
        duendes_para_grupo -- ;
        IF (duendes == 0) // Si soy el ultimo duende
            Signal (duendes_mutex); // Acepto nuevas peticiones de grupo
            Signal (mutex);
    FOREVER
}
```

4. El problema es similar al clásico problema del puente. Implementaremos un monitor que gestionará el uso del puente y proporcionará dos métodos exportados: `quieroAcceder` y `hePasado`

Monitor carril

VAR

```
Sentido: {1,2,3}; // Indica el sentido de uso del carril en un momento dado,
                // 1: entrando; 2: saliendo; 3: libre
nCochesEsperando: ARRAY[1..2,] of integer; // N° de coches esperando en cada sentido
                //la primera posición del array para entrar y
                //la segunda posición para conocer cuántos esperan para salir
espera: ARRAY[1..2] of condition; // ARRAY de variables de condición;
                // La primera posición para la espera de coches que desean entrar,
                // la segunda posición para la de salir
```

EXPORT

```
quieroAcceder (), hePasado()
```

```
function quieroAcceder (sentidoAcceso:integer)
```

```
BEGIN
```

```
    IF (sentido=3) THEN // El carril está libre
        sentido=sentidoAcceso; // Ponemos el sentido al sentido del coche
        IF (nCochesEsperando[sentido]>0) THEN resume (espera[sentido])
    }
    ELSE IF (sentido!=sentidoCoche) THEN // La circulación por el carril está en el sentido opuesto
        nCochesEsperando[sentido]++;
        delay (espera[sentido]);
```

```
END;
```

```
function hePasado (sentido_acceso:integer)
```

```
BEGIN
```

```
    IF (nCochesEsperando[sentidoAcceso]>0) THEN
        resume (espera[sentidoAcceso])
        nCochesEsperando[sentidoAcceso]--;
    ELSE
        IF nCoches_Esperando[(sentidoAcceso+1)%2]==0 THEN sentido=3; // carril libre
        ELSE
            sentido = (sentidoAcceso+1)%2; // Cambio el sentido del carril
            nCochesEsperando[(sentidoAcceso+1)%2]--; // Despertar al coche que está esperando
            resume (espera[(sentido+1)%2]);
```

```
END;
```

```
BEGIN // Inicialización del monitor
```

```
    nCochesEsperando[1], nCochesEsperando[2]= 0;
    sentido= 3;
```

```
END;
```

```
Process coche (sentido) // Muestra el comportamiento del coche
```

```
BEGIN
```

```
    Carril.quieroAcceder(sentido);
    // cruzar el carril
    Carril.hePasado(sentido)
```

```
END;
```

5; Para resolver el problema, vamos a utilizar 5 tipos de buzones:

- Un buzón *pedirPasoProf* para almacenar las peticiones de carril de los profesores
- Un buzón *pedirPasoAlum* para almacenar las peticiones de carril de los estudiantes
- Un buzón *liberarPaso* para incluir las liberaciones del recurso (carril)
- $N_PROF + N_ALUM$ buzones *pasoConcedido* para comunicar los permisos de uso del carril por parte del controlador.
- Un buzón *deseoEntrar* que usarán los profesores para indicar su deseo de acceder al carril.

```
Process profesor (id:integer)
BEGIN
    REPEAT
        Send (deseoEntrar,id); // Mostrar interés en pasar
        Send (pedirPasoProf, id); // Solicitar el paso
        Receive (pasoConcedido[id],id);
        // Quedar bloqueado hasta recibir el permiso pasar el carril
        Send (liberarPaso,id); // Liberar el recurso
    FOREVER
END;
```

```
Process alumno (id:integer)
BEGIN
    REPEAT
        Send (pedirPasoAlum, id); // Solicitar el paso
        Receive (pasoConcedido[N_PROF+id],id);
        // Quedar bloqueado hasta recibir el permiso pasar el carril
        Send (liberarPaso,id); // Liberar el recurso
    FOREVER
END;
```

```
Process controlador ()
VAR
    nProfEsperando:integer; // Contador para saber cuántos profesores están esperando
    carrilLibre:boolean; // Almacena el estado del carril
    id:integer;
BEGIN
    nProfEsperando=0; carrilLibre=TRUE;
    REPEAT
        SELECT
            // Retirar peticiones de acceso por parte de los profesores
            Receive (deseoEntrar,id);
            nProfEsperando++;

            // Los profesores solo podrán acceder al carril cuando esté libre
            OR
            WHEN carrilLibre
            Receive (pedirPasoProf,id);
            Send (pasoConcedido[id],id);

            // Los alumnos solo podrán acceder cuando el carril esté libre
            // y no haya peticiones de profesores esperando
            OR
            WHEN (carrilLibre AND (nProfEsperando==0))
            Receive (pedirPasoAlum,id);
            carrilLibre=FALSE;
            Send (pasoConcedido[id],id);

            OR
            Receive (liberarPaso,id);
            carrilLibre=TRUE;
            // Si quien sale del carril es un profesor, decrementar el número de profesores
            IF (id<N_PROF) THEN nProfEsperando--;
        END_SELECT
    FOREVER;
END;
```

```
Program carril ()
VAR
    deseoEntrar, pedirPasoProf, pedirPasoAlum, liberarPaso: mailbox of integer;
    pasoConcedido: ARRAY [1..N_PROF+N_ALUM] of mailbox of integer;
BEGIN
    COBEGIN
        Controlador();
        For (i=0; i<N_PROF;i++) profesor (i);
        FOR (i=0; i<N_ALUM; i++) alumno (i+N_PROF);
    COEND;
END.
```