A continuación se recoge una selección de problemas aparecidos en los exámenes del curso 2006-2007. Tras cada pregunta aparece la caja de respuesta, en el mismo tamaño que tenía en el examen, y dentro de ella la solución. Para leerla debes poner la hoja al revés ante un espejo o ponerla al trasluz mirando por detrás. Seguidamente viene la explicación de cómo se llega a esa respuesta, por si tienes dudas.

❑ Explica brevemente las diferencias fundamentales entre un *cluster* y un sistema de *grid-computing*.

El cluster es un conjunto de computadores, generalmente homogeneo, dedicado una tarea muy específica (ej: computación intensiva), en el que un coordinador central asigna el trabajo
En un grid computing, se intenta utilizar los recursos computacionales poco usados de un conjunto de computadores. No son homogéneos y no hay coordinador central

❑ En criptografía se denomina propiedad de *no-repudio* a la propiedad por la cual un documento firmado digitalmente no puede ser "repudiado" por su firmante. Es decir, él no puede alegar "Yo no firmé eso". ¿Crees que un sistema criptográfico de clave pública como GPG tiene esta propiedad? ¿Por qué?

Efectivamente, los sistemas criptográficos de clave pública en principio tienen esta propiedad, puesto que para efectuar la firma digital de un documento es necesario tener acceso a la clave privada y este acceso sólo es posible para una persona. Por tanto esta persona no puede negar que ha sido ella la firmante. Naturalmente esto depende del grado de celo con que custodie su clave privada. En todo caso, es su responsabilidad, y también lo es el revocar la clave si cree que ha sido comprometida.

❑ El siguiente código muestra parte de un cliente UDP para el protocolo `time`, programado bajo Windows. La función `InicializarBiblioteca()` no se muestra, se asume que inicializa la biblioteca de sockets de Windows. La función `CrearSocketUdp()` devuelve un socket UDP ya listo para enviar y recibir. No se hace comprobación de errores a la hora de enviar o recibir datagramas. El código se muestra incompleto y se preguntará por los huecos.

```
1  ...
2    struct sockaddr_in dir;
3    char *ip=argv[1];
4    char buff[10];
5    int longitud=sizeof(dir);
6    unsigned long int respuesta;
7
8    InicializarBiblioteca(2,0);
9    sock=CrearSocketUDP();
10   dir.sin_family=          ;
11   dir.sin_port=            ;
12   dir.sin_addr.s_addr=                ;
13
14   sendto(                             
15                                       
16                         ;
17
18   recvfrom(                           
19                                       
20                         );
21
22   respuesta=                          
23              - SEGUNDOS_ENTRE_1900_1970;
24
25   printf("%s\n", ctime((time_t *)&respuesta));
26 ...
```

— Completa la inicialización de la estructura `dir`.

```
dir.sin_family=AF_INET;
dir.sin_port=htons(37);
dir.sin_addr.s_addr=inet_addr(ip);
```

Explicación: El campo `sin_port` debe inicializarse con el número de puerto apropiado a este protocolo (37), sin olvidar la conversión al formato de red. El campo `sin_addr.s_addr` debe inicializarse con la IP del servidor, la cual, como se ve en la línea 3, está cargada en la variable `ip`, y debe ser convertida de cadena a 4 bytes en orden de red.

— Completa la llamada a `sendto()`.

```
sendto(sock, buff, 0, 0,
       (struct sockaddr *) &dir,
                longitud);
```

Explicación: El protocolo `time` para UDP requiere que el cliente envíe un datagrama vacío al servidor. Esto es lo que debe hacerse en esta línea. Para enviar un datagrama vacío basta con especificar 0 en el número de bytes a enviar.

— Completa la llamada a `recvfrom()`

```
recvfrom(sock, &respuesta, 4, 0,
         NULL, &longitud);
```

Explicación: La respuesta del servidor es un entero largo sin signo (4 bytes). En esta línea debemos usar la función `recvfrom` para recibir esta respuesta, almacenándola en la variable `respuesta` declarada a tal efecto, y especificando 4 en el número de bytes a recibir. Ya que no estamos interesados en la dirección del remitente, podemos poner `NULL` en este parámetro.

— Completa las líneas 22 y 23

```
respuesta=ntohl(respuesta)
  - SEGUNDOS_ENTRE_1900_y_1970;
```

Explicación: Como se deduce del `printf` que viene después, en esta línea se está preparando la respuesta del servidor para convertirla en una cadena de texto en inglés. La preparación consiste en cambiar el orden de los bytes (que vienen en orden de red y deben dejarse en orden de *host*), y restarle el número de segundos transcurridos entre 1900 (origen de tiempos del protocolo `time`) y 1970 (origen de tiempos en Unix, usado por la función `ctime`). Vemos por tanto que lo que falta en el hueco es una llamada a `ntohl` para corregir el orden de los bytes.

❑ El siguiente fichero `tipos.x` define algunos tipos XDR

```
1  typedef int Enteros<10>;
2
3  union Datos switch(int caso) {
4    case 1: Enteros a;
5    default: int b;
6  }
```

— Escribe cómo será el tipo `Enteros` que aparecerá en el fichero tipos.h una vez ejecutado `rpcgen`

```
struct Enteros {
    int Enteros_len;
    int *Enteros_val;
};
typedef struct Enteros Enteros;
```

Explicación: El tipo `Enteros` es un array de longitud variable, de tamaño máximo 10. En C se convierte en una estructura, uno de cuyos campos almacena cuántos enteros contiene, y el otro es un puntero a los enteros. El tamaño máximo (10) no forma parte del tipo C (pero lo usa el filtro que haga la conversión de este tipo).

Una variable de tipo `Datos` es leida de un fichero, y su valor mostrado en pantalla, usando el siguiente código:

```
1  Datos d;
2  FILE *f;
3  XDR  op;
4
5  f=fopen("Datos.xdr", "r");
6  // Inicializar op
7  [                  ](  [                    ]);
8
9  / Lectura y decodificación de d
10 memset(&d, 0, sizeof(d));
11 [                  ]
12
13 / Imprimir resultado
14 if (d.caso==1) {
15   if ([                          ]) {
16     printf("El array está vacío\n");
17   }
18   else {
19     for(i=0; [                          ]; i++) {
20      printf("a[%d]: %d\n",
21             i, [                          ]);
22   }
23 } else { printf("b=%d\n", d.Datos_u.b) };
```

— Escribe la inicialización de `op`

```
xdrstdio_create(&op, f, XDR_DECODE);
```

Explicación: Esta variable será la que le diga al filtro el sentido de la conversión y el fichero donde se guardará el resultado (o de donde se leerá el dato xdr en binario). En este caso, el fichero es `f` y, ya que estamos leyendo de él, el sentido de la conversión será `XDR_DECODE`.

— Escribe la lectura y decodificación de `d`

```
xdr_Datos(&op, &d);
```

Explicación: Es una mera llamada al filtro `xdr_Datos`, pasándole los parámetros apropiados.

— ¿Qué falta en el if?

```
d.Datos_u.a.Enteros_len==0
```

Explicación: Como se deduce del `printf` que va después, se está comprobando en ese `if` si el array está vacío. Esto es lo mismo que decir que el campo `_len` que almacena la longitud del array vale cero. La dificultad está en la enrevesada sintaxis necesaria para llegar a ese campo. La variable `d` es una estructura con un discriminante (llamado `caso`) y una unión llamada `Datos_u`. Uno de los campos de esta unión, se llama `a` y es a su vez una estructura, dentro de la cual está el campo `Enteros_len` que es el que interesa en este caso.

— ¿Qué falta en el `printf`?

```
d.Datos_u.a.Enteros_val[i]
```

Explicación: El cometido de ese `printf` es imprimir los valores de cada elemento del array. Por tanto hay que acceder al campo `_val` de la estructura correspondiente (véase la explicación de la pregunta anterior)

— Al ejecutar el código anterior aparece en pantalla el mensaje `"a[0]=0"`. ¿Cuáles son los valores en hexadecimal de los bytes que había en el fichero? (escribe un byte por casilla, puedes dejar casillas en blanco al final)

00	00	00	01	00	00	00	01
00	00	00	00				

Explicación: Puesto que lo que se está mostrando por pantalla es el array variable, eso implica que el discriminante valía 1, lo que nos da los cuatro primeros bytes de la respuesta. Por otro lado, aparece un sólo elemento del array en pantalla, lo que indica que la longitud del array es 1, y esto nos da los siguientes cuatro bytes del fichero. Finalmente, los siguientes cuatro bytes (que serán ya los últimos) contendrán el valor del elemento del array, que como se ve por lo que el programa ha impreso, es cero.

❑ Un fichero supuestamente codificado en UTF-8 comienza con la siguiente secuencia de bytes.

```
C2 A1 31 E2 82 AC 21
```

Escribe a continuación el texto que mostrará en pantalla una terminal que "comprenda" UTF-8 al volcar este fchero. Si hubiera algún carácter no válido, escribe un interrogante dentro de un cuadrado en su lugar.

```
¡1€!
```

Explicación: El primer byte comienza por `110`, lo que indica que es el primero de una secuencia de dos. El siguiente comienza por `10`, por tanto es válido y así los dos primeros bytes se decodifican como el carácter `U+00A1`, que al ser menor de `U+00FF` sigue el estándar ISO-8859-1 y corresponde al signo de abrir admiración (como podemos comprobar en las tablas de códigos).

El siguiente byte (31) comienza por `0`, por tanto es ASCII y es el código del carácter '1'.

Los tres siguientes bytes forman una secuencia válida (el primero comienza por `1110` y los tres siguientes por `10`) y se decodifican como el carácter `U+20AC`, que corresponde al símbolo del euro.

Por último, el último byte (21), comienza por `0`, por tanto es ASCII y corresponde al signo de cerrar admiración.

❑ Tenemos un archivo con texto Unicode, pero no sabemos si está codificado en UTF-8 ó UTF-16. Con la herramienta `hd` hacemos un volcado hexadecimal y encontramos los siguientes bytes:

```
FE FF 01 44 00 6F
```

— Utilizando una herramienta de recodificación, le pedimos que transforme el fichero anterior de UTF-16 a UTF-8. Si la herramienta encuentra un carácter imposible de recodificar, emite el error "No es posible". En caso contrario escribe la secuencia de bytes (hexadecimal) resultado de la conversión. ¿Qué imprimirá en este caso?

EF BB BF C5 84 6F

Explicación: La secuencia de bytes almacenada en el fichero es UTF-16 válido, y se interpreta como sigue:

- Los dos primeros bytes deberían ser el Byte Order Mark (BOM), y por tanto el carácter `U+FEFF`. Para que esto sea así, el orden dentro del fichero ha de ser *Big Endian*. Esto nos da la clave para descifrar los restantes datos.
- Los dos bytes siguientes, una vez sabemos que están en formato *Big Endian* son el carácter `U+0144`, que no pertenece a ningún rango surrogado, por tanto se interpretan tal cual están.
- Los dos bytes finales son el carácter `U+006F`, que de hecho es ASCII.

Tenemos por tanto la secuencia de caracteres `U+FEFF`, `U+0144`, `U+006F`. Solo resta codificar cada uno de ellos siguiendo las reglas de UTF-8 para llegar a la respuesta.

❑ Considera la ejecución del siguiente programa:

```
1  main()
2  {
3    int i;
4    for (i=0; i<3; i++)
5    {
6      printf("%d ", i);
7      if (fork()!=0) break;
8    }
9    printf("%d ", i);
10   sleep(30);  // Esperar 30 seg. antes de terminar
11 }
```

— ¿Cuántos procesos habrá en ejecución pasados 20 segundos? (se asume que en se tiempo ya se han ejecutado todos los `fork` que se tengan que ejecutar)

Cuatro

Explicación: Cada vez que se itera el bucle `for`, se ejecuta `fork` y el proceso crea un hijo. En este hijo, la condición `!=0` es cierta, por lo que abandona el bucle. Sólo el padre repite el bucle las tres veces, por lo que al final habrá creado tres hijos, más el propio padre. Los hijos no crean más hijos a su vez.

— ¿Cuántas veces habrá aparecido el número 1 impreso en la pantalla transcurridos 60 segundos? (se asume que todos los procesos han terminado ya)

Dos veces.

Explicación: Como se explicó en la pregunta anterior, el padre es el único que ejecuta el bucle las tres veces, imprimiendo los números del 0 al 2. Para cada número impreso, se crea un hijo, que recibe una copia de la variable `i`, sale del bucle e imprime `i`. El primer hijo creado imprime el `0`, el segundo imprime el `1` y el tercero imprime el `2`. Cada número aparece por tanto dos veces. valor de `i`.

❑ El *fingerprint* de una clave es un "resumen" que se obtiene aplicando un algoritmo *hash*... ¿sobre la clave privada o sobre la pública? Si no supieras la respuesta ¿cómo la deducirías?

Se aplica sobre la clave pública. No puede ser de otro modo porque ese /fingerprint/ se hace llegar a los receptores de nuestra clave pública para que puedan validarla. Estos receptores aplican el mismo /hash/ a la clave recibida y comprueban si les sale el mismo /fingerprint/. No tendría sentido hacerlo sobre la privada, puesto que nuestros interlocutores no la tienen.

❑ Completa el siguiente programa cuyo objetivo es inicializar una estructura `fd_set` con todos los sockets contenidos en el array `asock[]`, para usarlo en un `select()`.

```
int asock[MAX_SOCK], max;
fd_set fd;
...

FD_ZERO(&fd);
max=0;
for (i=0; i<MAX_SOCK; i++)
{
  FD_SET(asock[i], &fd);
  if (asock[i]>max) max=asock[i];
}
select(max+1, &fd, NULL, NULL, NULL);
```

Explicación: Tras poner a cero todos los bits del selector `fd`, hay que poner a 1 todos aquellos presentes en el array `asock[]`, es decir, en cada iteración del bucle `for` se pone a 1 (`FD_SET`) el descriptor `asock[i]`. A la vez, se aprovechan las iteraciones del bucle para actualizar el máximo (`max`) entre todos los valores del array `asock[i]`, puesto que `select()` necesita como primer parámetro este máximo más uno.

❑ Se tienen cuatro máquinas, que llamaremos A, B, C y D cuyos relojes no van sincronizados. La tabla siguiente muestra una serie de eventos que ocurren en estas cuatro máquinas, y para cada evento se muestra el valor que el contador de interrupciones tiene en la máquina en que el evento ha ocurrido.

Maquina	Evento	Contador	Descripción
A	A_1	10	Envía mensaje a D (D_1)
	A_2	15	Finaliza un backup
	A_3	20	Recibe mensaje de B (B_3)
B	B_1	5	Envía mensaje a C (C_1)
	B_2	13	Recibe mensaje de C (C_3)
	B_3	18	Envía mensaje a A (A_3)
C	C_1	3	Recibe mensaje de B (B_1)
	C_2	10	Recibe mensaje de D (D_2)
	C_3	15	Envía mensaje a B (B_2)
D	D_1	12	Recibe mensaje de A (A_1)
	D_2	17	Envía mensaje a C (C_2)

Se pide aplicar el algoritmo de Lamport para sincronizar los relojes lógicos de estas tres máquinas y seguidamente responder a las siguientes cuestiones.

— ¿Cuáles son los nuevos valores de los tres eventos de la máquina A?

$A_1 = 10$, $A_2 = 15$ y $A_3 = 29$ (sólo cambia A_3)

Explicación: Conviene dibujar un diagrama temporal que muestre la ordenación lógica entre los eventos. Después, comenzando por la izquierda, para cada mensaje transmitido entre máquinas, se actualiza el reloj de la máquina que recibe de modo que siempre sea mayor que el reloj de la máquina que envía, sumándole 1 si es necesario siguiendo el algoritmo de Lamport. Hay que tener en cuenta que, una vez se ha actualizado el contador de una máquina, todos los eventos futuros de esa misma máquina quedan incrementados en esa misma cantidad. Si, por ejemplo un contador pasa de 20 a 35, se ha incrementado en 15, y ese incremento se aplica en esa máquina a todos los eventos futuros. Una vez se ha recorrido todo el diagrama de izquierda a derecha realizando estas actualizaciones de relojes, se puede responder ya cuál es el valor final de cualquier contador en cualquier evento.

— Con respecto a los eventos A_1 y C_2, ¿crees que $A_1 \rightarrow C_2$, que $C_2 \rightarrow A_1$ o que $A_1 \,||\, C_2$? Explica tu respuesta.

$A_1 \rightarrow C_2$, lo que se demuestra fácilmente dado que $A_1 \rightarrow D_1$ por ser el envío y recepción de un mensaje, respectivamente; $D_1 \rightarrow D_2$ por ocurrir en la misma máquina y $D_2 \rightarrow C_2$ por ser de nuevo el envío y recepción de un mensaje. Aplicando la propiedad transitiva a estas tres relaciones, se obtiene la conclusión antes afirmada.

❑ ¿Crees que el hecho de que en Unix el tipo de un *socket* sea `int` es una ventaja o una desventaja? ¿Por qué?

Es más bien una desventaja, ya que el programador puede inadvertidamente usar otra variable de tipo `int` en lugar del socket y el compilador no puede detectar este tipo de errores.

❑ Un computador cuyo reloj tiene un ratio máximo de deriva $\rho = 0,01$ se sincroniza a las 10:00:00 con otro que siempre tiene la hora exacta (gracias a un GPS) A las 11:00:00 le pide de nuevo la hora, recibiendo la respuesta a las 11:00:14. La respuesta del servidor contiene las 11:00:17.

— ¿En cuánto tiempo están desfasados los dos relojes? Indica la unidad de tiempo en la respuesta.

10 segundos

Explicación: Entre el momento en que pide la hora y el momento en que la recibe transcurren 14 segundos. El algoritmo de Christian indica que la hipótesis más probable es que el servidor envió su respuesta (la hora que era en el servidor) en el instante intermedio, es decir cuando habían transcurrido 14/2=7 segundos en el cliente. Por tanto, en el cliente eran las 11:00:07 cuando el servidor enviaba su respuesta. Y en el servidor eran las 11:00:17 (pues esa es la hora que figura en su respuesta). Así pues, cuando en el cliente eran las 11:00:07, la hora real era 11:00:17, de modo que el cliente va 10 segundos atrasado.

— ¿Cuál es el máximo desfase que cabría esperar? Indica la unidad de tiempo en la respuesta.

El máximo esperable es 36 (3600ρ)

Explicación: El máximo desfase está relacionado con el ratio de deriva, ρ, que es 0,01. Esto indica que, como máximo, el cliente puede atrasar 0,01 segundos por cada segundo transcurrido. Ya que se había puesto en hora a las 10:00:00, cuando vuelve a hacerlo a las 11:00:00 han pasado 3600 segundos por lo que, como máximo, habrá atrasado (o adelantado) 36 segundos.