Material permitido: **Calculadora NO programable.** Tiempo: **2 horas.** N1	**Aviso 1:** Todas las respuestas deben estar razonadas. **Aviso 2:** Escriba sus respuestas con una letra **lo más clara posible.** **Aviso 3: No use *Tipp-ex*** o similares (atasca el escáner).

ESTE EXAMEN CONSTA DE 5 PREGUNTAS

1. (*2 p*) Conteste razonadamente si las siguientes afirmaciones son verdaderas o falsas:

a) (*1 p*). La llamada al sistema `lseek` tiene tres parámetros de entrada. Uno de ellos indica el número de bytes que se va a desplaza el puntero de lectura/escritura. Por tanto, dicho parámetro solo puede ser un número entero positivo.

b) (*1 p*) Para transferir la abstracción de hebra enteramente al nivel de usuario, debe intervenir el núcleo.

RESPUESTA:

Apartado a). FALSO.
(Ver 1.8.5.c. Acceso aleatorio a un fichero)

Este parámetro puede tomar un valor entero positivo o negativo.

La llamada al sistema `lseek` permite realizar accesos aleatorios mediante la configuración del puntero de lectura/escritura a un valor específico. Su sintaxis es:

```
resultado=lseek(fd,offset,origen);
```

donde `fd` es el descriptor del fichero, offset es el número de bytes que se va desplazar el puntero y origen es la posición desde donde se va desplazar el puntero, que puede tomar los siguientes valores constantes definidos en el fichero de cabecera <`stdio.h`>:

`SEEK_SET`. El puntero avanza `offset` bytes con respecto al inicio del fichero. El valor es 0.
`SEEK_CUR`. El puntero avanza `offset` bytes con respecto a su posición actual. El valor es 1.
`SEEK_END`. El puntero avanza `offset` bytes con respecto al final del fichero. El valor es 2.

Si `offset` es un número positivo, los avances deben entenderse en su sentido natural; es decir, desde el inicio del fichero hacia el final del mismo. Sin embargo, también se puede conseguir que el puntero retroceda pasándole a l`seek` un desplazamiento negativo.

Si la llamada se ejecuta con éxito, en `resultado` se almacena la posición que ha tomado el puntero de lectura/escritura, medida en bytes, con respecto al inicio del fichero. En caso de error en `resultado` se almacena el valor -1.

Apartado b). FALSO.
(Ver 4.9.4. Hebras de usuario)

Es posible transferir la abstracción de hebra enteramente al nivel de usuario, sin que el núcleo intervenga para nada. Esto se consigue a través de paquetes de librería tales como *C-threats* de Mach y *pthreads* de POSIX.

2. (*1.5 p*). Describe qué son las variables de condición y para qué se utilizan.

RESPUESTA:
Ver 6.7.3. Variables de condición.

Una *variable de condición* es un mecanismo complejo asociado con un *predicado* (una expresión lógica para evaluar si es verdadera o falsa) basado en algún dato compartido. Permite a un proceso bloquearse en función de su valor y suministra los servicios para despertar a uno o todos los procesos bloqueados cuando el resultado del predicado cambia. En general, resulta más útil para implementar la espera por un evento que para asegurar el acceso exclusivo a un recurso.

Supóngase, por ejemplo, uno o más procesos de un servidor que están esperando por la llegada de peticiones de clientes. Las peticiones entrantes son pasadas a los procesos que esperan o colocadas en una cola si no hay ningún proceso listo para atenderlas. Cuando un proceso del servidor está listo para atender la siguiente petición, primero comprueba la cola. Si hay una petición pendiente, el proceso la elimina de la cola y la atiende. Si la cola está vacía, el proceso se bloquea hasta que llega una petición.

Este escenario de funcionamiento se puede implementar asociando una variable de condición con la cola. El dato compartido es la propia cola de peticiones y el predicado es que la cola no este vacía. Puede suceder que una petición llegue después de comprobar la cola pero antes de bloquear el proceso. El proceso se bloqueará aunque exista una petición pendiente. En consecuencia se requiere una operación atómica para comprobar el predicado y bloquear al proceso si fuese necesario.

Las variables de condición suministran esta atomicidad usando un cerrojo. El cerrojo protege el dato compartido y evita el problema de no atención de peticiones anteriormente comentado. Se implementa para cada variable de condición una función llamada `wait()` que recibe el cerrojo como argumento y atómicamente bloquea al proceso y abre el cerrojo. Cuando se produce el evento `wait()` despierta al proceso y vuelve a cerrar el cerrojo antes de retornar.

3. (*1.5 p*). Conteste razonadamente las siguientes cuestiones. a) *(0.5 p)* ¿Para qué usa el núcleo el algoritmo `syscall()`?; b) (*1 p*) Dibuje un diagrama adecuadamente rotulado que resuma las principales acciones que realiza este algoritmo.

RESPUESTA.
(Ver 3.8. Interfaz de las llamadas al sistema)

Nota: Este apartado corrige una errata del enunciado original publicado. En particular, la puntuación el apatado b), de modo que la suma de ambos apartados a y b sea la puntuación total correcta (1.5 puntos).

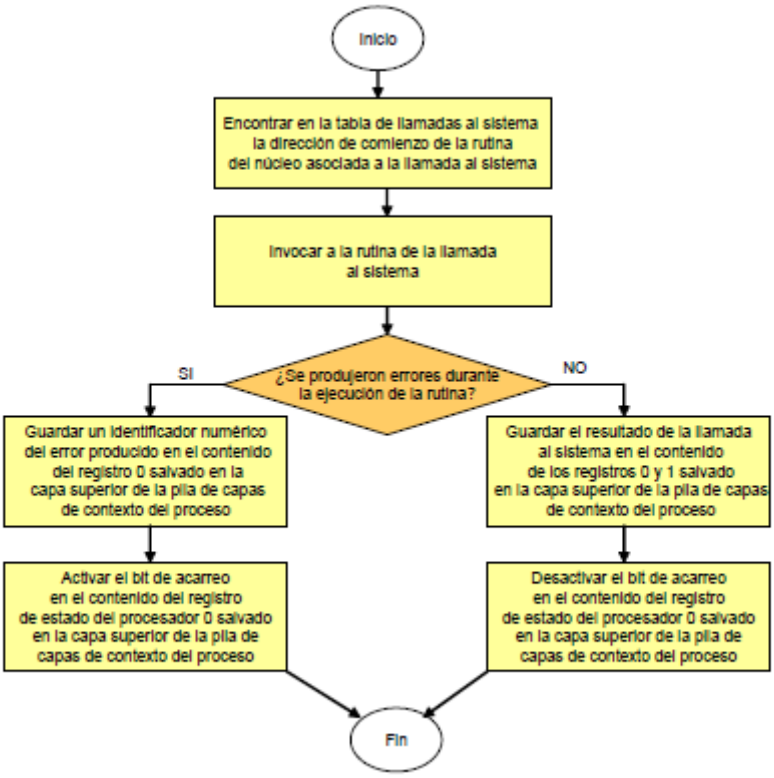
APARTADO a)

Las *llamadas al sistema* son el mecanismo que los procesos de usuario utilizan para solicitar al núcleo el uso de los recursos del sistema. Un proceso invoca a una llamada al sistema como si se tratase de una función de librería cualquiera. El compilador de C utiliza una librería predefinida de funciones, denominada *librería C* que contiene los nombres de las llamadas al sistema y que es enlazada, por defecto, con todos los programas de usuario. Estas funciones de librería, entre otras instrucciones, ejecutan una instrucción que provoca una interrupción software especial denominada *trap del sistema operativo*.

El tratamiento del *trap* por parte del núcleo provoca el cambio de modo de ejecución a modo núcleo, salvar el contexto del proceso actual y la invocación del algoritmo del núcleo que trata las llamadas al sistema, típicamente denominado syscall().

APARTADO b)

El diagrama que resume las principales acciones es el siguiente:



4 *(2 p)***.** Considérese una cache de buffers de bloques en un determinado sistema de ficheros que utiliza una sola partición. Esta cache consta de cuatro colas de dispersión. En un cierto instante de tiempo t, esta cache contiene una copia de 12 bloques de disco, cuyos números son: 11, 74, 85, 32, 8, 13, 2, 31, 41, 56, 47, 82. Dibuje un posible esquema adecuadamente rotulado de la cache de buffers de bloques en dicho instante t.

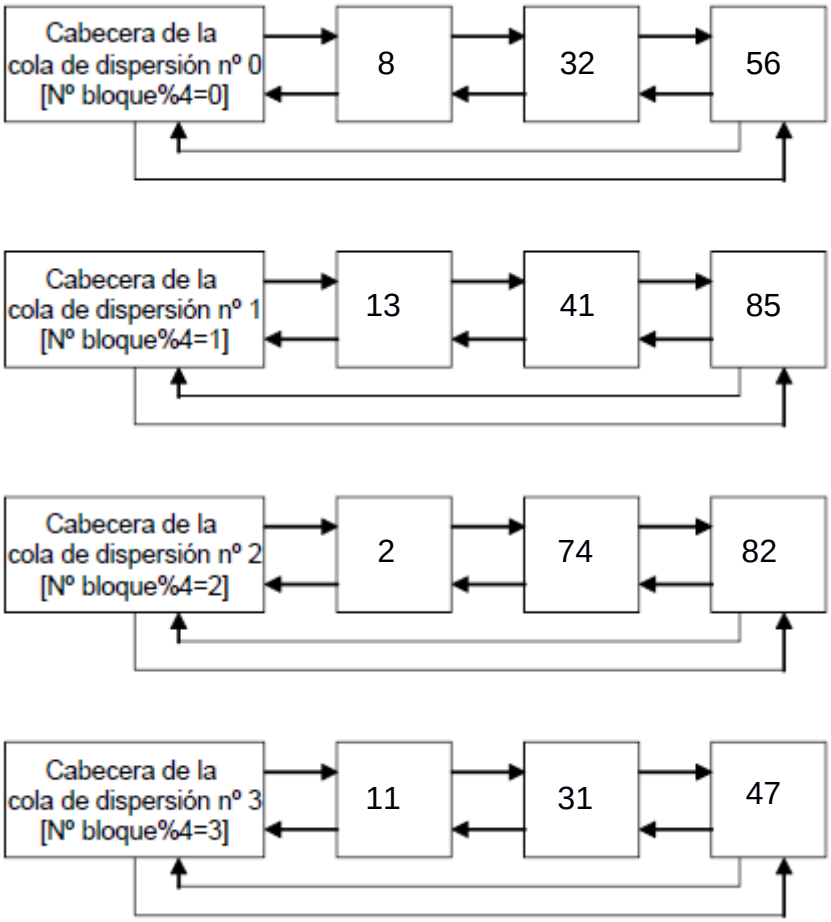
RESPUESTA:
(Ver 8.6. La caché de buffers de bloques)

De acuerdo con el enunciado, la cache de buffers de bloques consta de cuatro colas de dispersión: cola nº 0, cola nº 1, cola nº 2 y cola nº 3. Para averiguar a qué cola de dispersión pertenece cada una de las copias de bloques de disco, hay que realizar la operación:

Nº bloque%4=0

Es decir, obtener el resto de la división del número del bloque por 4, que es el número de colas de dispersión.

Un posible esquema es el siguiente:



5. *(3 p)* Conteste razonadamente a los siguientes apartados:

a) (1.5 p) Explicar el significado de las sentencias enumeradas (**[ ]**) del programa que se muestra en la página siguiente.

b) *(1.5 p)* Si el programa anterior se compila produciendo el ejecutable DyASO y se ejecuta con la orden `./DyASO`. Explique detalladamente el funcionamiento del programa así como su salida suponiendo que el primer proceso creado tiene pid=1000, los pids de los procesos se asignan consecutivamente y no se produce ningún error.

NOTA: la función `snprintf (buffer_de_salida, tamaño_del_buffer, cadena_de_control, Arg1, … ArgN )` formatea una cadena de texto usando una cadena de control y un conjunto de argumentos del mismo modo que `printf`. Pero, en lugar de escribirlo en la salida estándar, lo hace en el buffer que se le pasa como primer argumento y limitando la salida (si fuese necesario) al tamaño especificado en el segundo argumento.
NOTA2: La función `fflush(stdout)` obliga a que la escritura por salida estándar se realice inmediatamente impidiendo condiciones de carrera en la impresión del buffer de salida estándar `stdout`.

La pregunta continua en la página siguiente

```
        #include <stdio.h>
        #include <stdlib.h>
        #include <string.h>
        #include <unistd.h>
        #include <sys/msg.h>
[1]     void main(int argc, char *argv[])
        {
        key_t llave;
        struct{ long tipo;
             char cadena[2];
             } mensaje;
        int i, pid, msqid, N, longitud=sizeof(mensaje)-sizeof(mensaje.tipo);
        char Ntexto[10], texto[]="Ejercicio 5 DyASO ";
[2]     if ( (llave=ftok(argv[0],'M'))==-1) {perror("Error ftok"); exit(-1); }
[3]     if ( (msqid=msgget(llave, IPC_CREAT | 0600))==-1 )
                             {perror("Error msgget"); exit(-2); }

        if (argc>2) {printf("Argumentos incorrectos\n"); exit(-3); }
        if (argc==1) {
             N=strlen(texto);
             for(i=0; i<=N; i++) {
                  mensaje.tipo=1;
                  mensaje.cadena[0]=texto[i];
                  mensaje.cadena[1]=0;
[4]               msgsnd(msqid, &mensaje,longitud,0);
                  }
             }
        else      N=atoi(argv[1]);

[5]     if ( (pid=fork())==-1 ) {perror("Error fork"); exit(-4);}
        if (pid==0) {
             if (N>0) {
                  printf("Proceso hijo %d\n",getpid());
                  fflush(stdout);
                  snprintf(Ntexto, 10 , "%d",N-1);
[6]               if(execl(argv[0],argv[0],Ntexto,NULL)==-1)
                        {perror("Error en execl"); exit(-4);}
                  }
             printf("Fin ultimo hijo %d\n",getpid());
             fflush(stdout);
[7]          sleep(1);
             }
        else {
             wait();
[8]          msgrcv(msqid, &mensaje,longitud,1,0);
[9]          printf("%s",mensaje.cadena);
             fflush(stdout);
             if (argc==1) printf("correcto\n");
             }
        }
```

RESPUESTA
Apartado A:

[1] Se declara la función principal `main`. El parámetro `argc` es el número de argumentos pasado como entradas a la función `main`, mientras que `argv` es un array de punteros a cadenas de texto que contiene los argumentos con los que ha sido invocado el programa.

[2] La llamada al sistema `ftok` crea una llave asociada al fichero que recibe como argumento y a una letra. Puesto que la el nombre del fichero (`argv[0]`) es el primer argumento de la función `main` (eso es, nombre del ejecutable que se ha invocado). Se creara una clave asociada al propio ejecutable `DyASO` y la letra "`M`". Si se produce un error, `ftok` devolverá -1 cumpliéndose entonces la condición del `if`. En este caso `perror()` imprimiría "`Error ftok:`" seguido del descriptor del error que se ha producido en la llamada al sistema `ftok` (almacenado en la variable global `errno`). Finalmente la llamada al sistema `exit(-1)` terminaría el proceso devolviendo un código de terminación -1.

[3] La llamada al sistema `msgget` crea una cola de mensajes asociada a la `llave` anterior y devuelve el identificador de la cola de mensajes `msquid`. Al igual que en el caso anterior, si se produjese un error se mostraría mediante `perror` y se saldría del programa con una condición de error -2.

[4] La llamada al sistema `msgsnd` envía a través de la cola de mensajes cuyo identificador es `msquid` el contenido de la estructura `mensaje`.

[5] `fork()` es una llamada al sistema cuyo efecto es crear un proceso hijo del proceso actual duplicando el proceso inicial o padre. `fork()` devuelve al proceso padre el pid del proceso hijo, y devuelve 0 al proceso hijo, dicho valor de retorno se almacena en la variable `pid`. En caso de error al crear el proceso nuevo, se ejecutaría el bloque de código correspondiente al `if` (esto es, se mostraría un mensaje de error y finalizaría el programa con la condición de error -5).

[6] La llamada al sistema `execl(argv[0],argv[0],Ntexto,NULL)` se encarga de ejecutar un fichero ejecutable en el contexto del proceso actual. La memoria del proceso que la invoca pasará a ser sustituida por el código del programa ejecutable.

En este caso, el programa está llamando a si mismo (recordemos que la convención en UNIX, es que el primer argumento `argv[0]` sea el nombre del fichero ejecutable) y pasándose como parámetro su propio nombre, así como la cadena de texto `Ntexto`. El parámetro `NULL` sirve para indicar a `execl` que no hay más parámetros de entrada.

[7] `sleep(1)` es una llamada al sistema que pone a dormir al proceso que la invoca durante 1 segundo.

[8] La llamada al sistema `msgrcv` permite recibir un mensaje de la cola de mensajes `msquid` creada en la sentencia [3]. La llamada es bloqueante y queda a la espera de que haya mensajes del tipo especificado como cuarto argumento (mensajes de tipo 1). Cuando llegue un mensaje del tipo apropiado a la cola de lo extrae copiando su contenido en la estructura "`mensaje`". Los mensajes son leídos por orden de llegada.

[9] La sentencia `printf` escribe por salida estándar el texto almacenado en el campo `cadena` de la estructura `mensaje`.

Apartado B:

El funcionamiento del programa es el siguiente:

Cuando el programa es invocado desde la línea de órdenes con `./DyASO` se crea un proceso P1 con pid=1000.

Este proceso crea las variables que necesitará posteriormente, entre las que se incluye una estructura `mensaje` que consta de dos campos: el primero de tipo entero largo y una carga útil en forma de cadena de texto. Además, se determina el tamaño de la carga útil descontando el tamaño del tipo del mensaje del tamaño de la estructura completa.

A continuación, el proceso P1 crea una clave asociada a su propio ejecutable y la letra M sin producirse ningún error. Además, se cumple la condición `argc==1` (Puesto que no hay argumentos adicionales de entrada salvo el nombre del propio ejecutable "`./DyASO`" que se encuentra almacenado en `argv[0]`).

Al cumplirse dicha condición, se entra en un bucle que envía N mensajes de tipo 1. Cada uno de ellos contiene una cadena que consta de la letra i-esima de la cadena `"Ejercicio 5 DyASO "` y un carácter nulo que marca el fin del string. Puesto que N se establece al tamaño de la cadena, se envían una a una, las 18 letras del mensaje en orden.

Seguidamente, se ejecutará la instrucción [5] dando lugar a la aparición de un hijo H1 con pid=1001. El proceso padre, con pid=1000, quedará pausado con `wait` hasta que el proceso hijo termine. **Una vez que termine el hijo** con pid=1001, el padre despertará y leerá un mensaje de la cadena de mensajes, extrayendo una letra del mensaje y la imprimirá por pantalla. Finalmente como se cumple la condición `argc==1`, imprimirá también el texto `"correcto"` y terminará su ejecución.

Por su parte el proceso H1 cumple la condición `N=18>0` por lo que imprimirá "Proceso hijo" seguido de su pid, esto es "`Proceso hijo 1001`". A continuación, creará una cadena `Ntexto` que contiene el valor de N decrementado en una unidad (17). Seguidamente invoca de nuevo el ejecutable `./DyASO`, pasándole como argumento dicha cadena, esto es invocará `./DyASO 17`.

Puesto que el nuevo ejecutable sustituye al antiguo proceso la siguiente línea no llega a ejecutarse y no se imprime ningún mensaje.

La ejecución de `./DyASO 17` producirá de nuevo un proceso padre (que continua siendo H1 con pid=1001) y un proceso hijo H2 con pid=1002. A diferencia del caso anterior no se cumple `argc==1` por lo que se establece `N=17` (el valor del segundo argumento de entrada `argv[1]`) que es convertido a un entero mediante `atoi`).

El proceso H1 queda a la espera de que termine su hijo H2, cuando lo haga, recibirá un mensaje e imprimirá la letra que se encuentra en la carga útil, pero no imprimirá el texto `"correcto"` ya que argc=2.

El proceso H2 cumple la condición `N=17>0` de modo que imprimirá "`Proceso hijo 1001`" y volverá a invocar `./DyASO 16`.

Este patrón continuará creando más procesos H2, H3… a medida que N va disminuyendo. Estos procesos quedarán a la espera del fin de sus respectivos hijos. Cuando por fin se cree el proceso H19 recibirá el valor N=0 como argumento de entrada, de modo que no se cumple la condición `N>0`.

Esto significa **que el proceso H19 no ejecutará `execl`** sino que imprimirá por pantalla "`Fin ultimo hijo 1019`", esperará un segundo y terminará su ejecución.

Cuando termine H19, se despierta a su padre H18, que leerá el primer mensaje de la cola de mensajes imprimiendo la letra "`E`". A continuación, despertará H17 e imprimirá la letra "`j`" y así sucesivamente hasta que terminen el proceso H1 que imprimirá la letra "`O`". Entonces se desertará P1 que, como se indicó previamente, imprimirá un espacio (último carácter de la cadena `"Ejercicio 5 DyASO "`) seguido del texto "`correcto`".

De este modo la salida del programa es:

```
Proceso hijo 1001
Proceso hijo 1002
Proceso hijo 1003
Proceso hijo 1004
Proceso hijo 1005
Proceso hijo 1006
Proceso hijo 1007
Proceso hijo 1008
Proceso hijo 1009
Proceso hijo 1010
Proceso hijo 1011
Proceso hijo 1012
Proceso hijo 1013
Proceso hijo 1014
Proceso hijo 1015
Proceso hijo 1016
Proceso hijo 1017
Proceso hijo 1018
Fin ultimo hijo 1019
Ejercicio 5 DyASO correcto
```

NOTA: Sea cual sea el orden de planificación de los procesos padres e hijos **el resultado es siempre el mismo** ya que: 1) cada proceso hijo H1… H19 imprime su pid antes de que se creen procesos nuevos. 2) Además, cada proceso padre espera por su hijo antes de terminar y por lo tanto el orden de las terminaciones está fijado H19 H18…H1, P1. 3) Finalmente, dado que la cola de mensajes preserva el orden en el que los mensajes han sido enviados las letras se leerán y mostrarán por pantalla en el mismo orden en el que fueron enviadas.