Material permitido: **Calculadora NO programable.** Tiempo: **2 horas.** N	**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*) Las entradas de un directorio FFS son invariables en cuanto a longitud.

b) (*1 p*) Un proceso en segundo plano puede enviar las salidas al monitor.

RESPUESTA:

Apartado a). Falso.

Las entradas de un directorio FFS varían en longitud. La parte fija de la entrada consiste del número de nodo-i, el tamaño asignado y el tamaño del nombre del fichero en la entrada. Éste está seguido por un nombre de fichero terminado en un carácter nulo con espacio extra de 4 bytes. El tamaño máximo de un nombre de fichero es de 255 caracteres. Cuando se borra un nombre de fichero, FFS fusiona el espacio liberado con una entrada previa. (*Aptdo. 8.11.3 Mejoras en la funcionalidad de un sistema de ficheros FFS)*

Apartado b) Cierto.

Un proceso puede estar en *primer plano* o en *segundo plano*. Solo puede haber un proceso en primer plano al mismo tiempo. El proceso que está en primer plano es el que interactúa con el usuario, recibe entradas de teclado y envía las salidas al monitor. El proceso en segundo plano no recibe ninguna señal desde el teclado **por lo general, se ejecuta en silencio** sin necesidad de interacción. Para ejecutar una tarea en segundo plano basta con añadir el carácter '`&`' al final de la orden (Aptdo. *2.8.2 Primer plano y segundo plano).*

Que un proceso "por lo general" se ejecute en silencio **no significa** que no pueda enviar su salida al monitor, sirva como ejemplo la orden:

```
$yes &
```

Que se ejecuta en segundo plano pero sin dejar de escribir "y" en la pantalla. Es fácil comprobar que se encuentra en segundo plano ya que no es posible detenerla con Ctrl-C (no recibe las señales del teclado).

NOTA: dado que la filosofía de los procesos en segundo plano es no molestar e interrumpir la acción del proceso en primer plano, esta pregunta puede dar lugar a interpretaciones erróneas. Por ese motivo también se dará por válidas respuestas alternativas (aunque en rigor no sean correctas) siempre que estén bien razonadas y demuestren que se comprende la diferencia entre primer y segundo plano, por ejemplo:

"Falso: los procesos en segundo plano no reciben señales desde el teclado y se ejecutan normalmente en silencio".

2. (*1.5 p*) Responda a las siguientes cuestiones razonadamente: a) (*0.5 p*) ¿Qué es un intérprete de comandos? b) (*1 p*) ¿Qué diferentes tipos de intérpretes de comandos existen en función de su forma de invocación?

RESPUESTA:

a) Un intérprete de comandos es un programa ejecutable que puede ser invocado con diferentes opciones y argumentos. Su principal tarea es interpretar y ejecutar las órdenes que el usuario da al Sistema Operativo.

b) De forma general en función de su forma de invocación se distinguen tres tipos de intérpretes de comandos:

- *Intérprete de entrada* (*login shell*). Es aquél cuyo primer carácter del argumento cero es un `'-'`, o uno que ha sido llamado con la opción `--login`. Cuando un usuario entra en el sistema se suele invocar a un intérprete de entrada que entre otras funciones configura el valor de ciertas variables de entorno e invoca a un intérprete interactivo.

- *Intérprete interactivo* (*interactive shell*). Es uno cuya entrada y salida estándares están conectadas a terminales, es decir el usuario puede interactuar con ellos, o uno que ha sido llamado con la opción `-i`.

- *Intérprete no interactivo* (*non-interactive shell*). Es uno que no cumplen ninguna de las condiciones para ser un intérprete interactivo. Un ejemplo típico de intérprete no interactivo es un subintérprete invocado para ejecutar un shell script. (Aptdo. *2.11 Ficheros de arranque de un intérprete de comandos)*

3. (*1.5 p*) Responda razonadamente las siguientes cuestiones: a) (*0.5 p*) ¿Qué son las señales? b) (*1 p*) ¿Cuáles son las principales fuentes de generación de señales?

RESPUESTA:

a) Las señales proporcionan un mecanismo para notificar a los procesos los eventos que se producen en el sistema. Los eventos se identifican mediante números enteros, aunque también tienen asignados constantes simbólicas que facilitan su identificación al programador.

Algunos de estos eventos son notificaciones asíncronas (por ejemplo, cuando un usuario envía una señal de interrupción a un proceso pulsando simultáneamente las teclas `[control+c]` en el terminal), mientras que otros son errores síncronos o excepciones (por ejemplo, acceder a una dirección ilegal).

Las señales también se pueden utilizar como un mecanismo de comunicación y sincronización entre procesos.

En el mecanismo de señalización se distinguen dos fases principalmente: *generación* y *recepción* o *tratamiento*. Una señal es generada cuando ocurre un evento que debe ser notificado a un proceso. La señal es recibida o tratada cuando el proceso para el cual fue enviada la señal reconoce su llegada y toma las acciones apropiadas. Asimismo, se dice que una señal está *pendiente* para el proceso si ha sido generada pero no ha sido tratada todavía. (Aptdo. 4.4 Señales)

b) El núcleo genera señales para los procesos en respuesta a distintos eventos que pueden ser causados por: el propio proceso receptor, otro proceso, interrupciones o acciones externas. Así, las principales fuentes de generación de señales son:

- *Excepciones*. Cuando durante la ejecución de un proceso se produce una excepción (por ejemplo, un intento de ejecutar una instrucción ilegal), el núcleo se lo notifica al proceso mediante el envío de una señal.

- *Otros procesos*. Un proceso puede enviar una señal a otro proceso, o a un conjunto de procesos, mediante el uso de las llamadas al sistema `kill` o `sigsend`. Un proceso también puede enviarse una señal a si mismo usando la llamada al sistema `raise`.

- *Interrupciones del terminal*. La pulsación simultánea por parte de un usuario de teclas, como `[control+c]` o `[control+\]`, produce el envío de señales a los procesos que se encuentran ejecutándose en el primer plano de un terminal.

- *Control de tareas*. Los interpretes de comandos generan señales para manipular tanto a los procesos que se encuentran ejecutándose en primer plano como a los que se encuentran ejecutándose en segundo plano. Cuando un proceso termina o es suspendido, el núcleo se lo notifica a su padre mediante el envío de una señal.

- *Cuotas*. Cuando un proceso excede su tiempo de uso de la CPU o el tamaño máximo de un fichero, el núcleo envía una señal a dicho proceso.

- *Notificaciones*. Un proceso puede requerir la notificación de ciertos eventos, como por ejemplo que un dispositivo se encuentra listo para realizar una operación de E/S. El núcleo informa al proceso de este evento enviándole una señal.

- *Alarmas*. Un proceso puede configurar una alarma para se active transcurrido un cierto tiempo. Cuando éste expira, el núcleo se lo notifica enviándole una señal. (*Aptdo. 4.4 Señales*)

4. (*2 p*). Supóngase que la lista parcial de i-nodos libres del superbloque está vacía, su i-nodo recordado es 750 y su variable índice puede tomar como máximo el valor 5. Además, existen los siguientes i-nodos libres en la tabla de i-nodos: 875, 765, 782, 773, 810, 793, 850 y 825. Dibuje la lista parcial de i-nodos libres del superbloque una vez que ha sido rellenada por el núcleo. ¿Cuál sería ahora el i-nodo recordado?

RESPUESTA:

El núcleo comenzará a buscar i-nodos libres a partir del i-nodo recordado (750) en orden ascendente, por lo tanto conviene ordenar los números de i-nodo libres del enunciado en esta forma

765 773 782 793 810 825 850 875

Por otra parte, puesto que la variable índice puede tomar como máximo el valor 5, eso implica que la lista parcial de i-nodos del superbloque va a poder contener como máximo 6 números de i-nodos libres. En consecuencia, el núcleo seleccionará para rellenar la lista de i-nodos libres a los seis primeros que encuentre, estos son 765, 773, 782, 793, 810 y 825.

Finalmente, la lista parcial de i-nodos libres del superbloque es rellenada por el núcleo colocando los i-nodos en orden descendente, con lo que se tiene:

i-nodo recordado ->	**825**	810	793	782	773	765
índice	**0**	1	2	3	4	5

Por lo tanto, el i-nodo recordado es el **825.** *(Tema 8)*

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

a) *(1.5 p)* Explique el significado de las sentencias enumeradas (**[ ]**) del programa mostrado en la página siguiente.

b) *(1.5 p)* El programa es compilado produciendo el fichero ejecutable `programa`.
Explique la ejecución del programa y su salida si se invoca desde la línea de comandos la orden "`./programa correcto`", el SO asigna al proceso creado el pid 1000 y no se produce ningún error.
Nota: `fflush` sirve para vaciar el buffer de salida estándar haciendo que la salida de `printf` se imprima inmediatamente.

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <fcntl.h>
    #include <unistd.h>

    int main(int argc, char *argv[]){
    char buffer, texto[]=" OSAyD 5 oicicrejE";
    int i, pid, tub[2], tam=strlen(texto)-1;
[1] if (argc!=2){printf("Argumentos incorrectos\n"); exit(-1);}
[2] if(pipe(tub)==-1) {perror("Error open:");exit(-2);}
[3] if((pid=fork())==-1) {perror("error en primer fork:"); exit(-3);}
    if(pid==0) {
[4]     printf("Hijo 0, PID=%d\n",getpid()); fflush(stdout);
        close(tub[0]);
        for(i=0;i<=tam;i++) {
          if ((pid=fork())==-1) {perror("error en siguente fork:"); exit(-4);}

            if(pid==0) {
                 printf("Hijo %i, PID=%d\n",i+1,getpid()); fflush(stdout);
[5]              if(i==tam) {close(tub[1]); break;}
                 }
            else{
[6]              wait();
[7]              sleep(1);
[8]              write(tub[1],&texto[i],1);
                 if (i==0) write(tub[1],argv[1],strlen(argv[1]));
                 close(tub[1]);
                 break;}
            }
        }
    else {
         printf("Padre 0, PID=%d\n",getpid()); fflush(stdout);
         close(tub[1]);
[9]      while((read(tub[0],&buffer,1))>0) {
              printf("%c",buffer); fflush(stdout);
              }
         printf(".\n");
         close(tub[0]);
         return 0;
         printf("Incorrecto\n");
         }
    }
```

RESPUESTA

Apartado a)

[1] La variable `argc` indica el número de argumentos de entada de la función principal. Por lo tanto, esta sentencia comprueba que hay dos argumentos de entrada (el nombre de la función que se pasa como primer argumento según la convención de UNIX y un segundo argumento adicional). En caso contrario, la llamada al sistema `exit` termina el programa con un código de terminación -1.

[2] la llamada al sistema `pipe()` crea una tubería sin nombre y devuelve un array con dos descriptores. El primero de ellos (`tub[0]`) sirve para leer, mientras que el segundo (`tub[1]`) se utiliza para escribir. Si la llamada al sistema transcurre con éxito, devolverá 0, en caso contrario, devolverá -1 y se ejecuta la condición del `if`. Si esto ocurre, `perror()` imprimirá "Error open:" seguido del descriptor del error que se ha producido en la llamada al sistema anterior (almacenado en la variable global `errno`). A continuación la llamada al sistema `exit(-2)` terminaría el proceso devolviendo un código de terminación `-2`.

[3] `fork()` es la llamada al sistema que se utiliza para crear un nuevo proceso hijo duplicando el código del proceso padre. Dicha llamada se encuentra dentro de un `if`, que, en caso de producirse un error, lo muestra con `perror` y termina el proceso con status -3. `fork()` devuelve el *pid* del proceso hijo al padre y 0 al proceso hijo. De este modo, el hijo ejecutará el bloque de código a continuación del `if` de la línea siguiente, mientras que el padre ejecutará el código correspondiente a la cláusula `else` final.

[4] `printf` Se encarga de imprimir el texto "`Hijo 0, PID=`" seguido del *pid* del proceso y un salto de línea. El *pid* del proceso se obtiene mediante la llamada al sistema `getpid()`.

[5] La llamada al sistema `close(fd)` se encarga de cerrar un descriptor de fichero. En este caso se cierra `tub[1]` que es el descriptor de escritura de la tubería. Dicha llamada es invocada cuando la variable `i` del bucle es igual a `tam`, esto es, en la última iteración del mismo. Llegado a este punto se sale del bucle con `break`.

[6] `wait()` es una llamada al sistema que sincroniza la ejecución del proceso padre y del proceso hijo. Cuando el padre la invoca queda a la espera a que termine su hijo antes de continuar.

[7] `sleep(1)` es una llamada al sistema que suspende el proceso en durante un determinado tiempo. En este caso, el proceso que la invoque dormirá durante 1 segundo.

[8] La llamada al sistema `write` se utiliza para escribir de forma genérica en un fichero. En este caso particular, se escribe en el extremo de escritura de una tubería (`tub[1]`). El texto se toma de la posición i-ésima de la cadena texto (`&texto[i]`) y se escribe un solo byte.

[9] La llamada al sistema `read` se utiliza para leer del extremo de lectura de la tubería (`tub[1]`). Esta llamada lee un único byte y lo deposita en la posición de memoria donde se encuentra la variable `buffer` (`&buffer`). Dicha llamada devuelve el número de bytes leídos, que actúa como condición de un bucle `while`. De este modo el bucle se ejecutará mientras `write` sea capaz de leer algo de la tubería.

Apartado b)

En primer lugar se definen las constantes y variables que se usan en el programa y se calcula la longitud de la cadena "`OSAyD 5 oicicrejE`". A dicha longitud se le resta 1 para tener en cuenta el carácter nulo que existe a final de la misma.

A continuación, se comprueba que el número de argumentos de entrada `argv` es correcto. En este caso no se produce ningún error, ya que se reciben dos argumentos: el nombre del ejecutable en primer lugar y el argumento "`correcto`" en segundo.

Hecho esto, se crea una tubería sin nombre. Como no se producen errores el programa continúa

El proceso padre, que llamaremos P0, tiene *pid*=1000 y ejecuta el código de la cláusula `else` final del programa. En primer lugar imprime "`Padre 0, PID=1000`". A continuación, cierra el extremo de la tubería que no va a necesitar y entra en un bucle.

Dicho bucle va leyendo carácter a carácter de la tubería e imprimiéndolos por salida estándar. Finalmente ejecuta `return` y devuelve el valor de retorno 0.

Por otra parte, el proceso hijo H0 imprime "`Hijo 0, PID=1001`" Y a continuación entra en un bucle `for` que se encarga de crear más hijos. Analicemos el bucle pasó a paso:

1) Se genera un hijo nuevo H1 con `fork` para el cuál `i=0`, su pid es 1002 y que imprime `Hijo 1, PID=1002.`
2) El proceso hijo H1, comprueba si se trata de la última ejecución del bucle (i==tam). En el caso del proceso H1 no se cumple la condición, de modo pasaría a la siguiente iteración i=1.
3) El proceso H0 (que es el padre de H1) queda espera a la terminación de su hijo con `wait`, (para él la variable i vale 0).
4) En la siguiente iteración del bucle `i=1` se crea un nuevo proceso H2, que imprime `Hijo 2, PID=1003`
…

Este proceso continua creando una cadena de procesos hijos hasta que se llegue a la última iteración del bucle `i==tam`. Puesto que el tamaño de la cadena de texto es de 18+1 caracteres (ya que `strlen` incluye el carácter de fin de línea), entonces `tam=18`. En esa iteración, el proceso padre será H18 con i=18 y su hijo será H19.

El proceso hijo H19 cumple la condición `i==tam` por lo que cierra la tubería, sale del bucle con break y termina su ejecución sin crear más hijos.

Cuando H19 termine despertará su padre H18, que a su vez esperará un segundo y escribirá en la tubería el carácter `texto[18]` (la "`E`"), a continuación cierra el fichero y termina el bucle y con ello su ejecución.

El fin del proceso H18 despierta al proceso H17 que espera otro segundo y escribe `texto[17]`, es decir "`j`" en la tubería, lo cierra y termina.

El proceso continúa **escribiendo una letra por segundo en el fichero fifo** hasta llegar al proceso H0.

Recordemos que mientras los ficheros hijos escriben en la tubería, el proceso padre P0 está leyendo en un bucle. Además, el proceso lector P0 quedará bloqueado cada vez que se vacíe la tubería y haya al menos un proceso que la tenga abierta para escritura.

De este modo, el proceso P quedara a la espera de que H18 escriba "`E`" en la tubería, lo leerá, lo mostrará inmediatamente por la salida estándar. En la siguiente iteración del bucle, volverá a quedarse bloqueado a la espera de que escriba alguien más. Un segundo después, H17 escribirá la "`j`" y así sucesivamente...

Esta secuencia seguirá **mostrando una letra por segundo** hasta que se haya escrito el mensaje siguiente:

```
Ejercicio 5 DyASO
```

Llegado este punto, todos los procesos han cerrado el fichero para lectura y el único proceso que tiene el fichero abierto es P (pero en modo lectura). Por tanto, una vez que el hijo H0 termine, P ya no queda bloqueado en la lectura del fichero `fifo` y finaliza el bulce `while`. A continuación, escribe por la salida estándar "correcto", que es el argumento que se pasó al lanzar el programa y por último termina su ejecución. De este modo la salida completa del programa podría ser:

```
Padre 0, PID=1000
Hijo 0, PID=1001
Hijo 1, PID=1002
Hijo 2, PID=1003
Hijo 3, PID=1004
Hijo 4, PID=1005
Hijo 5, PID=1006
Hijo 6, PID=1007
Hijo 7, PID=1008
Hijo 8, PID=1009
Hijo 9, PID=1010
Hijo 10, PID=1011
Hijo 11, PID=1012
Hijo 12, PID=1013
Hijo 13, PID=1014
Hijo 14, PID=1015
Hijo 15, PID=1016
Hijo 16, PID=1017
Hijo 17, PID=1018
Hijo 18, PID=1019
Ejercicio 5 DyASO correcto.
```

NOTA 1: Dependiendo de la planificación del SO el mensaje "`Padre 0, PID=1000`" puede imprimirse en cualquier orden respecto de los mensajes de los procesos hijos. Lo que siempre ocurre es que los mensajes de los procesos hijos se imprimen en el orden indicado y que lo último en imprimirse es el mensaje "`Ejercicio 5 DyASO correcto.`"

Esto es así ya que cada proceso H_i ha de esperar hasta que acabe su hijo H_{i+1} antes de poder escribir en la tubería y el mecanismo de la tubería garantiza que las lecturas se realizan en el mismo orden que las escrituras.

NOTA 2: Obsérvese también que la instrucción `printf("Incorrecto\n")` no se llega a ejecutar nunca **al ser posterior a la instrucción** `return` que finaliza el programa.