Material permitido: **Calculadora NO programable.** Tiempo: **2 horas.** N2	**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. (*1p*). Señala cuál de las siguientes afirmaciones es verdadera y **razona** la respuesta. El algoritmo de planificación usado en BSD4.3:

a) Permite que un proceso de baja prioridad quede abandonado mientras existan procesos de mayor prioridad.

b) Establece la prioridad de un proceso mediante la llamada al sistema `priocntl`.

c) Permite que cualquier proceso aumente su prioridad mediante el uso de la llamada al sistema `nice`.

d) Es estrictamente no expropiable lo que hace que pueda aparecer inversión de prioridades.

RESPUESTA D):

Ver apartado 5.3.4 del libro base. El resto de opciones son falsas ya que a) el cálculo dinámico de prioridades evita el abandono de procesos (ver apartado 5.3.5), b) `priocntl` es un mecanismo de SVR4, no de BSD4.3 (sección 5.4.2) c) ya que el uso de `nice` con valores negativos está restringido al superusuario (sección 5.3.1).

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

i) (*1p*) Si en el directorio actual existe un archivo `uno.txt` con número de nodo-i 4096 que posee una sola referencia y se invocan consecutivamente los comandos "`ln uno.txt dos.txt`" y "`ln -s uno.txt tres.txt`"; el contador de referencias del archivo `tres.txt` vale 1.

ii) (*1p*) Una mejora de FFS respecto a s5fs es que Intenta asignar bloques secuenciales de un fichero en posiciones rotacionalmente óptimas.

RESPUESTA:

i) La afirmación es **verdadera.** El primer comando crea un enlace duro de nombre `dos.txt` que apunta al mismo archivo 4096 por lo que el contador de referencias del mismo pasa a valer 2. El segundo comando crea un enlace simbólico, que es un archivo con otro número de nodo-i y cuyo contador de referencias vale 1, el contador de referencias del archivo 4096 no se modifica. Si se borran los nombres `uno.txt` y `dos.txt` el contador de referencias del archivo 4096 valdrá cero y el archivo se borra (quedando roto el enlace simbólico `tres.txt`). Páginas 346 y siguientes del libro base.

ii) La afirmación es **verdadera.** FFS intenta mantener los directorios padre e hijo en grupos de cilindros separados (pag 396) y asigna bloques secuenciales en posiciones rotacionalmente óptimas si es posible (pag 379). FFS implementa además los enlaces simbólicos.

3. (*2 p*) Conteste a las siguientes cuestiones:
a) (*0.5 p*) Explique **brevemente** el funcionamiento de un intérprete de comandos de UNIX.
b) (*1.5 p*) Enumere y explique los tipos de órdenes que de forma general se pueden ejecutar en un intérprete de comandos.

RESPUESTA:

Apartado a)
Un *intérprete de comandos* es un fichero ejecutable. El proceso que se crea asociado a la ejecución de dicho fichero en primer lugar lee y ejecuta las órdenes establecidas en los diferentes ficheros de arranque del intérprete para configurar sus variables internas, a continuación muestra el marcador en pantalla y se queda a la espera de recibir órdenes del usuario. Cuando se teclea una orden, el intérprete de comandos en primer lugar busca el nombre de la orden y comprueba si es una orden interna. En caso afirmativo la ejecuta. En caso contrario considera que es una orden externa por lo que debe buscar su programa ejecutable asociado. Si lo encuentra lo ejecuta. En el caso de que no se pueda encontrarlo mostrará un mensaje de aviso por la pantalla.

Apartado b)
De forma general, las órdenes que se pueden ejecutar en un intérprete de comandos pueden ser de dos tipos:
- *Órdenes internas* (*builtin commands*). Son aquellas órdenes cuyo código de ejecución se encuentra incluido dentro del propio código del intérprete. Así la ejecución de una orden interna no supone la creación de un nuevo proceso. Ejemplos de órdenes internas son `cd` y `pwd`.
- *Órdenes externas*. Son aquellas órdenes que para poder ser ejecutadas por el intérprete requieren de la búsqueda y ejecución del fichero ejecutable asociado a cada orden. Típicamente son programas ejecutables o *shell scripts* incluidos en la distribución del sistema o creados por el usuario. La ejecución de una orden externa supone la creación de al menos un nuevo proceso con la llamada al sistema `fork` y la invocación del programa ejecutable con la llamada al sistema `exec`. Ejemplos de órdenes externas son `ls` y `mkdir`.

4. (*2p*) En un sistema UNIX la máscara de modo del fichero ordinario resultados es 6644.
a) (*1p*) Explique razonadamente el significado de todos los bits de la máscara de modo.
b) (*0.5 p*) Escriba la máscara de modo simbólica asociada a este fichero.
c) (*0.5 p*) ¿Qué orden se debe teclear desde la línea de comandos (`$`) para que la máscara simbólica del fichero pase a ser `-rwx r-- --T`?

RESPUESTA:

Apartado a)
La máscara de modo 6644 en octal equivale al número de 12 bits:

[b_{11} b_{10} b_9	b_8 b_7 b_6	b_5 b_4 b_3	b_2 b_1 b_0]
1 1 0	1 1 0	1 0 0	1 0 0

Los tres bits más significativos b_{11} b_{10} b_9 están asociados a los bits `S_ISUID, S_ISGID` y `S_ISVTX` respectivamente.
Los bits b_8 b_7 b_6 establecen los permisos de lectura, escritura y ejecución, respectivamente, del propietario del fichero.
Los bits b_5 b_4 b_3 establecen los permisos de lectura, escritura y ejecución, respectivamente, del resto de usuarios.
Por tanto, esta máscara indica que los bits `S_ISUID, S_ISGID` están activados, que el propietario tiene permiso de lectura y escritura y que todos los demás usuarios poseen únicamente permiso de lectura sobre el fichero.

Apartado b)

La cadena de caracteres asociada a esta máscara de modo 6644 de este fichero regular es

```
- rwS     r-S  r--
```

Apartado c)

La cadena de caracteres `-rwx r-- --T` (s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0) está dando la siguiente información sobre el fichero. El carácter s_9 indica que es un fichero de tipo regular. Los caracteres (s_8 s_7 s_6) indican que el propietario del fichero posee permiso de lectura, de escritura y de ejecución. Además el bit `S_ISUID` está desactivado. Los caracteres (s_5 s_4 s_3) indican que los usuarios pertenecientes al mismo grupo que el propietario del fichero solamente poseen permiso de lectura, y que el bit `S_ISGID` está desactivado. Finalmente los caracteres (s_2 s_1 s_0) indican que los restantes usuarios no poseen ningún permiso, y que el bit `S_ISVTX` está activado.

En conclusión, dicha cadena de caracteres es equivalente a la máscara de modo

001 111 100 000

Que en octal es 1740. Luego un posible comando a utilizar es

```
$ chmod  1740 resultados
```

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

a) *(1.5 p)* Explicar el significado de las sentencias enumeradas (**[ ]**) de este programa.

b) *(1.5 p)* Explicar el funcionamiento del programa ejecutable `prog` resultante de compilar el código siguiente cuando se invoca desde la ventana de comandos: `./prog SO`.

```
     #include <stdio.h>
     #include <string.h>
     #include <stdlib.h>
     #include <sys/types.h>
     #include <unistd.h>
     #define MAX 256
[1]  void main(int argc,char *args[])
          {
          int tuberia[2];
          int pid;
          char mensaje[MAX];
          if (argc!=2)
               {printf("Argumentos incorrectos\n");
               exit(-1);}
[2]       if (pipe(tuberia)==-1)
               {
[3]            perror("Error en pipe");
[4]            exit(-1);
               }
[5]       if ((pid=fork())==-1)
               {
               perror("Error en fork");
               exit(-1);
               }
          else if (pid==0)
               {
[6]            if (read(tuberia[0], mensaje, MAX)==0)
                    {perror("Error de lectura"); exit(-2);}
               printf("Examen %s ",mensaje);
[7]            if (write(tuberia[1],args[1],strlen(args[1])+1)==0)
                    {perror("Error de escritura"); exit(-2);}
```

```
[8]             close(tuberia[0]);
                close(tuberia[1]);
                exit(0);
                }
          else
                {
                if (write(tuberia[1],"DyA",4)==0)
                     {perror("Error de escritura"); exit(-2);}
[9]             wait();
                if (read(tuberia[0], mensaje,MAX)==0)
                     {perror("Error de lectura"); exit(-2);}
                printf("%s\n",mensaje);
                close(tuberia[0]);
                close(tuberia[1]);
                exit(0);
                }
          }
```

RESPUESTA

Apartado a)

[1] Se declara la función principal `main` que admite un número de parámetros definido por el entero `argc` y un vector de cadenas de caracteres que contiene los parámetros. Siendo el primero de ellos el nombre del archivo ejecutable.

[2] La llamada al sistema `pipe()` crea una tubería sin nombre. Si se produce un error devuelve cero, ejecutándose por tanto el bloque de código a continuación, si no hay ningún error coloca en el array `tuberia` los manejadores que permiten leer y escribir en la tubería recién creada.

[3] La función `perror()` muestra por la salida estándar el contenido de la variable global `errno` que almacena el descriptor del error que se ha producido en la llamada al sistema anterior. En caso de que se haya producido un error `perror` mostrará la cadena "`Error en pipe`" seguida de "`:`" y del mensaje asociado al error que se ha producido.

[4] La llamada al sistema `exit()` en caso de llegar a ejecutarse finaliza la ejecución del proceso devolviendo al proceso padre un estatus -1.

[5] La llamada al sistema `fork()` crea un proceso hijo al que devuelve 0, mientras que al padre le devuelve el `pid` del hijo. Este resultado se almacena en la variable `pid`. En caso de producirse un error devolverá -1, de tal modo que la se verificará la condición del `if` ejecutándose el bloque de código que se encuentra a continuación.

[6] La llamada al sistema `read` se utiliza para leer el contenido de la tubería, admite como parámetros el manejador lectura de la tubería `tuberia[0]` que se creó en **[2]**, un puntero al buffer dónde se almacenará la información leída `mensaje` y el número de bytes a leer (como máximo) `MAX`. Cuando se realiza una lectura en una tubería vacía, el proceso quedará bloqueado a la espera de que se escriban datos en la tubería o hasta que la tubería se cierre. Cuando finaliza su ejecución devuelve el número de bytes leídos, en caso de ser cero se cumpliría la condición del `if`.

[7] La llamada el sistema `write` funciona de forma complementaria a `read`, admite como entrada el descriptor de lectura de la tubería `tuberia[1]`, el buffer con los datos a leer, que procede del segundo argumento de entrada a la función principal (`args[1]`) y la longitud de los datos. Es importante destacar que un *string* tiene siempre un carácter más que su longitud (el carácter nulo que marca el fin del *string*), por ello el tamaño a escribir es el tamaño del *string* obtenido mediante la función de librería `strllen` e incrementado en una unidad.

[8] La instrucción `close()` cierra el manejador de la tubería, en este caso el manejador de lectrura `tubería[0]`. Es importante recalcar que la tubería sique existiendo hasta que se cierren todos los manejadores a la misma.

[9] La llamada al sistema `wait()` hace que el proceso padre espere a la finalización del proceso hijo (o a la recepción de otra señal) antes de continuar. Esto garantiza que el hijo se ejecutará antes que el padre termine la instrucción **[9].**

Apartado b)

El funcionamiento del programa es el siguiente:
En primer lugar la función es invocada mediante `./prog SO` de tal manera que recibe dos argumentos, su nombre `prog` y el segundo argumento `SO`.

A continuación se comprueba si el número de parámetros es correcto, en nuestro caso lo es, si fuera de otra forma se mostraría un mensaje y se finalizaría la ejecución del programa. Hecho esto, se crea una tubería con la llamada al sistema `pipe`. Si se produce un error se muestra otro mensaje y se finaliza. En caso contrario se continúa.

Llegados a **[5]** el proceso se divide en dos, creándose un proceso hijo que es una copia del padre, `fork()` devuelve entonces 0 al hijo que ejecuta el contenido del `else if` y el `pid` del hijo al padre que ejecuta el contenido del último `else`.

Puesto que la instrucción `read` es bloqueante, sea cual sea la planificación de los procesos, el `write` del padre terminará de ejecutarse antes que el `read` del hijo, ya que aunque el hijo se planificara en primer lugar, al llegar a **[6]** quedará bloqueado hasta que haya algo que leer en la tubería.

El proceso padre, por su parte, escribe en la tubería la cadena "`DyA`" y a continuación se queda esperando (wait) a que termine el proceso hijo. Esta espera es fundamental ya que de lo contrario el padre podría leer de la tubería lo que él mismo acaba de escribir ("`DyA`").

A continuación el hijo lee el contenido de la tubería ("`DyA`") mostrando por salida estándar "`Examen DyA`". Hecho esto, el hijo escribe el primer argumento de entrada a la función ("`SO`") en la tubería, cierra ambos extremos de la tubería y finaliza su ejecución.

Al finalizar el proceso hijo, el padre recibe una señal que le despierta del `wait()`, a continuación lee el contenido de la tubería y lo muestra por la salida estándar. Finalmente cierra los extremos de la tubería y termina.

La salida del programa será por lo tanto:

```
Examen DyA SO
```