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

a) (*1 p*) La llamada al sistema `msgrcv` permite que un proceso pueda enviar un mensaje.

b) (*1 p*) `specfs` mantiene un nodo-s común por cada proceso que quiere acceder a un dispositivo.

RESPUESTA:

a) **FALSA.** La llamada al sistema `msgrcv` permite que un proceso pueda extraer (recibir) un mensaje de una determinada cola de mensajes. (*Ver 6.4.3d Recepción de mensajes*).

b) **FALSA**. `specfs` mantiene un único nodo común por cada dispositivo para centralizar todas las operaciones de acceso al mismo, evitando por ejemplo que un proceso cierre un dispositivo mientras otro proceso lo está usando (*Ver 9.5.4 El nodo-s común y en particular la figura 9.4*).

2. (*1.5 p*) Conteste a las siguientes cuestiones:

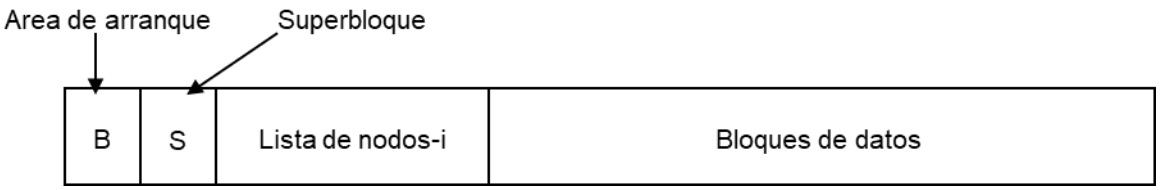
a) (*0.5 p*) Dibuje un esquema adecuadamente rotulado de la estructura de *s5fs* en un disco.

b) (*1 p*) Describa cada uno de los elementos del esquema del punto anterior.

RESPUESTA

(Ver Apartado 8.8.1 Organización en el disco del s5fs)

a)



b) Al comienzo de la partición de disco asociada al sistema de ficheros *s5fs* se encuentra el *área de arranque*, que puede contener el código requerido para arrancar (carga e inicialización) el sistema operativo.

A continuación, encuentra el *superbloque*, que contiene atributos y metadatos del propio sistema de ficheros.

A continuación del superbloque se encuentra la *lista de nodos-i*, que es un array lineal de nodos-i. Hay un nodo-i por cada fichero. Un nodo-i contiene información administrativa o metadatos del fichero. Cada nodo-i puede ser identificado por su *número de nodo-i*, que es igual al índice de la *lista de nodos-i.*

El espacio después de la lista de nodos-i es el *área de datos*, que contiene bloques de datos para ficheros y directorios, así como *bloques indirectos*, que contienen punteros para bloques de datos de ficheros.

3 (*1.5 p*) Describa los tres casos que se pueden presentar cuando el *ladrón de páginas* pretende realizar una transferencia y debe comprobar si existe ya una copia de una página en el área de intercambio.

RESPUESTA

(Ver Apartado 7.6 Transferencia de páginas de memoria principal al área de intercambio)

Cuando el *ladrón de páginas* pretende realizar una transferencia de una página al dispositivo de intercambio debe considerar si ya existe una copia de dicha página en el dispositivo, se pueden presentar tres casos:

1) *No existe una copia de la página en el dispositivo de intercambio*. Entonces el núcleo "planifica" la página para ser transferida, es decir, coloca la página en una lista de páginas que deben ser transferidas. Cuando esta lista alcanza un cierto tamaño (que depende de las capacidades del manejador del disco) el *núcleo copia todas las páginas de esta lista* en el dispositivo de intercambio.

2) *Existe una copia de la página en el dispositivo de intercambio y no se ha modificado el contenido de la página de memoria principal* (el campo *modificada* de la entrada de tabla de páginas asociada a dicha página está *sin activar*). Entonces el núcleo desactiva el campo *válida*, decrementa el contador de referencias en la entrada de la tabla `dmp` y coloca dicha entrada en la lista de marcos de página libres.

3) *Existe una copia de la página en el dispositivo de intercambio y se ha modificado el contenido de la página almacenada en memoria principal*. Entonces el núcleo "planifica" la página para ser transferida y decrementa el contador de entradas de la tabla de intercambio. Cuando el contador llega a cero, libera el espacio que ocupaba la copia de la página en el dispositivo de intercambio. Cuando se vuelva almacenar la página en el dispositivo de intercambio, su copia se almacenará en otra posición distinta.

4. (*2 p*) Supóngase que un proceso A se está ejecutando en modo usuario en un sistema UNIX BSD4.3 y que se produce la siguiente secuencia de sucesos:

1) En el instante de tiempo t_1 el proceso A invoca a una llamada al sistema.
2) En el instante de tiempo t_2 mientras se está ejecutando la llamada al sistema llega una interrupción del disco duro.
3) En el instante de tiempo t_3 mientras se está ejecutando la rutina de servicio del disco llega una interrupción de reloj.

Determinar **razonadamente** el número de cambios de contexto que se producirán en el sistema debido a la aparición de estos sucesos.

RESPUESTA

(Ver Apartado 3.6.4 Cambio de contexto)

Para responder a esta pregunta se debe recordar que:

- Un cambio de contexto es el conjunto de tareas que debe realizar el núcleo para aplazar o finalizar la ejecución del proceso actualmente en ejecución y comenzar o continuar con la ejecución de otro proceso.
- Las interrupciones son atendidas en modo núcleo dentro del contexto del proceso que se encuentra actualmente en ejecución, aunque dicha interrupción no tenga nada que ver con la ejecución de dicho proceso.
- Un sistema UNIX BSD4.3 es no expropiable, es decir, si un proceso se está ejecutando en modo núcleo no se le puede expropiar el uso de la CPU para que la use otro proceso aunque éste sea más prioritario.

Teniendo en cuenta todo lo anterior, si el proceso A se está ejecutando en modo usuario y en el instante de tiempo t_1 invoca a una llamada al sistema, se cambia de modo usuario a modo núcleo y se comienza a atender la llamada al sistema. En el instante t_2 mientras se está ejecutando la llamada al sistema llega una interrupción del disco duro, la cual se atiende en el contexto del proceso A. En el instante de tiempo t_3 mientras se está ejecutando la rutina de servicio del disco llega una interrupción de reloj, que al ser más prioritaria que la interrupción del disco, pasa a ser atendida, también en el contexto del proceso A. En conclusión, la aparición de esta serie de sucesos **no produce** ningún cambio de contexto.

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

a) (2 p) Explicar el significado de las sentencias enumeradas (**[ ]**) del programa que se muestra en la página siguiente.

b) *(1 p)* Explicar el funcionamiento del programa si se compila en el ejecutable `programa`, se invoca con la orden **.**`/programa` y no se produce ningún error.

NOTAS: `fflush(stdout)` vacía el buffer de salida estándar, forzando una impresión inmediata de su contenido. Además, es importante recordar que cuando se duplica un descriptor de fichero se utiliza el primer descriptor libre. Finalmente se recuerda y que los primeros descriptores por defecto que posee de un proceso recién creado son 0 (`stdin`), 1 (`stdout`) y 2 (`stderr`).

La pregunta continua en la página siguiente

```
        #include <stdio.h>
        #include <stdlib.h>
        #include <sys/types.h>
        #include <sys/wait.h>
        #include <unistd.h>

        int main (void){

        int pid1, pid2, salida, status, par;
        int tuberia[2];

[1]     salida=system("echo '5 DyASO' > temporal");
        salida=system("echo '3 error' >> temporal");

[2]     if(pipe(tuberia)==-1) {perror("Error open:");exit(1);}

[3]     if ((pid1=fork())==-1) {perror("error en fork 1: ");exit(2);}
        if (pid1==0)
             {
[4]          close(1);
[5]          dup(tuberia[1]);
             close(tuberia[0]);close(tuberia[1]);
[6]          sleep(1);
[7]          execl("/bin/cat","cat","temporal",NULL);
             perror("Error execl");
             printf("¡Incorecto!\n");
             exit(0);
             }

        if ((pid2=fork())==-1) {perror("error en fork 2: ");exit(3);}
        if (pid2==0)
             {
             close(0);
             dup(tuberia[0]);
             close(tuberia[0]);close(tuberia[1]);
[8]          printf("Ejercicio ");fflush(stdout);
             execl("/bin/grep","grep", "DyASO",NULL);
             perror("Error execl");
             printf("¡Erroneo!\n");
             exit(0);
             }
        close(tuberia[0]);
        close(tuberia[1]);
        par=wait(&status);
[9]     par=wait(&status);
        printf("Correcto! \n\n");
[10]    unlink("temporal");
        }
```

RESPUESTA

APARTADO A):

[1] La llamada `system()` permite ejecutar la orden el intérprete de comandos asociado al proceso actual. La orden que se ejecuta es "`echo '5 DyASO' > temporal`" que redirige la salida de `echo` a un fichero cuyo nombre es `temporal`. De este modo se crea (o sobrescribe si ya existiese) un fichero llamado `temporal` cuyo contenido es `5 DyASO`. Terminada de ejecutar dicha orden se continúa con el programa y se guarda en resultado el status del proceso `echo`.

[2] La llamada al sistema `pipe()` crea una tubería sin nombre. Si se produce un error devuelve -1, ejecutándose el bloque de código a continuación, que muestra el error con `perror` y seguidamente termina el programa con el código de salida 1. Si no hay ningún error, coloca en el array tubería los manejadores que permiten leer y escribir en la tubería recién creada.

[3] La llamada al Sistema `fork()` crea un proceso hijo que es una copia idéntica del proceso padre. Al proceso padre le devuelve el pid del hijo que queda almacenado en la variable `pid1` mientras que al proceso hijo le devuelve 0. En caso de error, `fork` devuelve -1, de tal modo que se ejecuta la cláusula `if`, se imprime "`Error fork`" y se termina el proceso con la llamada al sistema `exit` y el código de retorno 2.

[4] La llamada al sistema `close` cierra el fichero cuyo descriptor es `1`, es decir la salida estándar del proceso que la invoca.

[5] `dup(tuberia[1])` duplica el descriptor del extremo de escritura de la tubería `(tuberia[1])`. Al duplicarse un descriptor se utiliza el primer descriptor libre (el recientemente cerrado descriptor 1 o `stdin`). De este modo la salida estándar del proceso queda redirigida al extremo escritura de la tubería. Cuando el proceso escriba en su "salida estándar" realmente estará escribiendo en la tubería.

[6] `sleep` es una llamada al sistema que suspende la ejecución del proceso que la invoca durante el tiempo indicado como primer argumento (un segundo).

[7] `execl` es una llamada al sistema que carga en la memoria el programa ejecutable indicado en el primer parámetro, con las opciones que se le pasan a continuación. En este caso se ejecutará el programa `cat` del directorio `bin` (`/bin/cat`) al que se le pasan como argumentos su propio nombre (convenio de UNIX) así como el argumento `temporal`. `NULL` indica que ya no hay más argumentos. Si la llamada al sistema tiene éxito, el contexto y la memoria del proceso que la invoca es sustituida por el nuevo ejecutable `cat`.

[8] `printf("Ejercicio "); fflush(stdout)` imprime el texto "`Ejercicio `" por salida estándar. `fflush` hace que el buffer de salida estándar se imprima inmediatamente.

[9] `wait` hace que el proceso padre se suspenda hasta la terminación de algún hijo. Obteniendo en la variable `status` el código de retorno del mismo.

[10] La llamada al sistema `unlink` borra el fichero "temporal".

APARTADO B:

El funcionamiento del programa es el siguiente:

En primer lugar, mediante las llamadas al sistema `system` (**[1]** y la instrucción siguiente) se crea un fichero cuyo nombre es "`temporal`" con el siguiente contenido:

```
5 DyASO
3 error
```

A continuación se crea una tubería sin nombre y se ejecuta `fork` para crear un proceso hijo al que llamaremos **H1**. Puesto que no se producen errores el programa continúa.

El proceso padre recibe en `pid1` el pid de su hijo por lo que no ejecuta el bloque `if` y a continuación crea un segundo proceso hijo **H2**. Tampoco se producen errores.

Hecho esto, el proceso padre queda a la espera de que terminen sus dos procesos hijos con dos llamadas al sistema `wait`.

Por su parte, el proceso **H1** redirige la salida estándar al extremo de escritura de la tubería (ver **[5]**) y cierra los descriptores de la tubería `tuberia[0]` y `tuberia[1]` que no se van a utilizar posteriormente. Es importante destacar que esto no cierra la tubería ya que es accesible a través del descriptor 1 (`stdin`). Hecho esto, el proceso duerme durante un segundo.

Transcurrido dicho segundo, el proceso **H1** realiza una llamada al sistema `execl` que ejecuta la orden "`cat temporal`". Puesto que el contexto de **H1** es reemplazado por el ejecutable `cat` el resto de órdenes nunca llega a ejecutarse, en particular no se imprime "`¡Incorrecto!`".

Nótese asimismo que los objetos de fichero abierto se heredan de padres a hijos, de modo que el proceso cat sigue teniendo redirigida su salida estándar a la tubería. De este modo el nuevo proceso **H1** imprime el contenido del fichero temporal a la tubería y no por pantalla.

El funcionamiento del proceso **H2** es similar, pero en este caso se redirecciona la estrada estándar del proceso (0) al extremo de lectura de la tubería. Seguidamente imprime por salida estándar:

```
Ejercicio
```

Y termina ejecutando "`grep DyASO`" con `execl`. Dicho proceso filtra la entrada estándar (que ahora es la salida de la tubería) en busca de la palabra DyASO e imprime la línea correspondiente, es decir:

```
5 DyASO
```

Este proceso es similar al que realizaría un intérprete de órdenes cuando se ejecuta la orden "`echo temporal | grep DyASO`".

Una vez concluidos los procesos **H1** y **H2**, se despierta el padre que los esperaba mediante sendas llamadas al sistema `wait`. Una vez despierto imprime "`Correcto`" por salida estándar, borra el fichero "temporal" y termina.

La salida completa del programa es por tanto:

```
Ejercicio 5 DyASO
Correcto!
```

NOTA 1: Sea cual sea el orden en el que el SO planifique los procesos padre e hijos, la salida es siempre la misma ya que el hijo **H1** y el hijo **H2** sincronizan su actividad a través de la tubería. En particular, el proceso **H2** queda a la espera de que el proceso **H1** termine de escribir en la tubería. Además, el proceso padre espera la terminación de los dos procesos hijos antes de imprimir el texto “Correcto!”.

NOTA 2: Como no se producen errores en la ejecución en `execl`, no se ejecutan las órdenes posteriores y en particular nunca se imprime “¡Incorrecto!” ni “¡Erroneo!”. Además `grep` filtra la línea que contiene el texto DyASO y por lo tanto tampoco se imprime la línea “3 error”.