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*). En UNIX; para visualizar pantalla a pantalla el contenido del fichero `host.conf` situado dentro del directorio /`etc`, se debe teclear la orden

```
$ cat /etc/host.conf
```

b) (*1 p*) Un proceso puede incurrir en un fallo de validez cuando intenta escribir una página cuyo bit *copiar al escribir* esté activado.

RESPUESTA:

APARTADO a): FALSO
(Ver Ejemplo 2.9)

La orden `cat` haría mostrar la visualización tan rápido que no sería posible leerlo. Para visualizarlo pantalla a pantalla se debe teclear la orden.

```
$ more /etc/host.conf
```

APARTADO b): FALSO
(Ver 7.7 Tratamiento de los fallos de página)

Cuando un proceso intenta escribir una página cuyo bit copiar al escribir está activado, puede incurrir en un fallo de protección, en lugar de fallo de validez.

El sistema puede incurrir en dos tipos de fallos de página: *fallos de validez* y *fallos de protección*.

- Un *fallo de validez* se produce cuando un proceso intenta acceder a una página cuyo bit *válida* (en su entrada asociada en una tabla de páginas) no está activado. El bit *válida* no está activado para aquellas páginas que no pertenecen al espacio de direcciones virtuales del proceso, ni para aquellas páginas que siendo parte del espacio de direcciones virtuales del proceso no tienen asignado actualmente un marco de página.

- Un *fallo de protección* se produce cuando un proceso intenta acceder a una página válida cuyos bits de protección no permiten acceder a la página (por ejemplo si un proceso intenta escribir en su región de código). Asimismo un proceso puede incurrir en un fallo de protección cuando intenta escribir una página cuyo bit *copiar al escribir* esté activado. El núcleo debe determinar la causa del fallo de protección.

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

a) (*1 p*) ¿Qué es el superbloque?

b) (*0.5 p*) Explique cómo procede núcleo cuando la lista parcial de nodos-i libres del superbloque se vacía.

RESPUESTA:

(Ver 8.8.4. El superbloque)

APARTADO a).

El *superbloque* contiene metadatos sobre el propio sistema de ficheros. Hay un único superbloque por cada sistema de ficheros y reside al comienzo del sistema de ficheros en el disco, a continuación del área de arranque. El núcleo lee el superbloque cuando monta el sistema de ficheros y lo almacena en memoria hasta que el sistema de ficheros es desmontado. El superbloque contiene básicamente información administrativa y estadística del sistema de archivos, como por ejemplo:

-Tamaño en bloques del sistema de ficheros.
-Tamaño en bloques de la lista de nodos-i.
-Número de bloques libres y nodos-i libres.
-Comienzo de la lista de bloques libres.
-Lista parcial de nodos-i libres.

APARTADO b).

Puesto que el sistema de ficheros puede contener muchos nodos-i libres o bloques de disco libres, es poco práctico mantener en el superbloque una lista de nodo-i libres completa y una lista de bloques libres completa.

En el caso de los nodos-i, el superbloque mantiene una lista parcial de nodos-i libres. Un nodo-i se considera que está libre cuando su campo di_mode contiene el valor 0.

Cuando la lista se vacía, el núcleo busca en el disco nodos-i libres para rellenar la lista comenzando por el *nodo-i recordado* y en sentido ascendente de número de nodo-i. El *nodo-i recordado* se define como el nodo-i de mayor número de nodo-i que se ha almacenado en la lista parcial de nodos-i libres desde la última vez que ésta fue rellenada.

3. (*1.5 p*) Describa qué es un cerrojo con bucle de espera y explique cómo funciona.

RESPUESTA:

(Ver 6.7.2 Cerrojos con bucle de espera)

Un *cerrojo con bucle de espera* (*spin locks*) es una primitiva muy simple que permite el acceso en exclusiva a un recurso. Si un recurso está protegido por un cerrojo, un proceso intentando acceder a un recurso no disponible estará ejecutando un bucle, lo que se denomina *espera ocupada*, hasta que el recurso esté disponible. Un cerrojo suele ser una variable escalar que vale 0 si el recurso está disponible y que vale 1 si el recurso no está disponible. Poner a 1 el cerrojo significa “cerrar el cerrojo” y ponerlo a 0 significa “abrir el cerrojo”. La variable es manipulada usando un bucle sobre una instrucción atómica del tipo comprobar-configurar.

La característica más importante de un cerrojo es que un proceso retiene una CPU mientras espera a que el cerrojo sea abierto. Es por lo tanto esencial que un cerrojo permanezca cerrado durante periodos de tiempo muy pequeños. En particular, no debe estar cerrado entre operaciones de bloqueo. Asimismo también es deseable bloquear las interrupciones antes de cerrar un cerrojo, para así garantizar que el tiempo de cierre será pequeño.

La premisa básica de un cerrojo es que un proceso realiza una espera ocupada en un procesador mientras que otro proceso está usando el recurso en un procesador diferente. Obviamente esto sólo es posible en sistemas multiprocesador. En un sistema con un único procesador, si el proceso intenta cerrar un cerrojo que ya está cerrado, entonces permanecerá en un bucle infinito. Los algoritmos para sistemas multiprocesador, no obstante, deben operar adecuadamente independientemente del número de procesadores, lo que significa que también deben funcionar adecuadamente en un sistema monoprocesador.

En el caso de los cerrojos, esto requiere el cumplimiento estricto de la siguiente regla: **un proceso nunca le cede el uso de la CPU mientras tiene cerrado un cerrojo**. De esta forma, se asegura en el caso monoprocesador que una hebra nunca tendrá una espera ocupada en un cerrojo.

La mayor ventaja de los cerrojos es que son muy económicos en cuanto a tiempo de ejecución. Cuando no hay disputa por un cerrojo, tanto la operación de cierre como la de apertura típicamente requieren únicamente una instrucción cada una. Resultan ideales para estructuras de datos de uso exclusivo que necesitan ser accedidas rápidamente, como por ejemplo la eliminación de un elemento en una lista doblemente enlazada o mientras se realiza una operación del tipo carga-modificación-almacenamiento de una variable. Por tanto, los cerrojos son utilizados para proteger aquellas estructuras de datos que no necesitaban de protección en un sistema monoprocesador. Los semáforos utilizan un cerrojo para garantizar la atomicidad de sus operaciones.

4. (*2 p*). Supóngase que el ladrón de páginas de un cierto sistema UNIX debe transferir a un dispositivo de intercambio 50 páginas del proceso A, 30 páginas del proceso B, 40 páginas del proceso C, y 60 páginas del proceso D; y que en una operación de escritura puede transferir 56 páginas al dispositivo de intercambio. Determinar la secuencia de operaciones de intercambio de operaciones que tendría lugar si el ladrón de páginas examina las páginas de los procesos en el orden A, B, C y D.

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

La secuencia de operaciones es la siguiente:

1) El ladrón de páginas asigna espacio para 56 páginas y transfiere al dispositivo de intercambio 50 páginas del proceso A y 6 páginas del proceso B. Luego ha transferido todas las páginas del proceso A, le restan por transferir 24 páginas del proceso B, 40 del proceso C y 60 del proceso D.

2) El ladrón de páginas asigna espacio para otras 56 páginas y transfiere al dispositivo de intercambio 24 páginas del proceso B, y 32 páginas del proceso C. Luego ha transferido todas las páginas del proceso B, y le restan por transferir 8 del proceso C y 60 páginas del proceso D.

3) El ladrón de páginas asigna espacio para otras 56 páginas y transfiere al dispositivo de intercambio 8 páginas del proceso C, y 48 páginas del proceso D. Luego ha transferido todas las páginas del proceso C, y le restan por transferir 12 páginas del proceso D.

4) El ladrón de páginas guarda las 12 páginas que restan por intercambiar del proceso D en la lista de páginas de intercambio y no las intercambiará hasta que la lista este llena.

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 `prog`. Explique la ejecución del programa y su salida si se invoca desde la línea de comandos la orden `./prog DyASO`

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

     void main(int argc, char *argv[])
     {
     int i, pid, fd;
     char texto[]="OSAyD 5 oicicrejE";
     int tam=strlen(texto)-1;

     if (argc!=2){printf("Argumentos incorrectos\n"); exit(-1);}
[1]  unlink(argv[1]);
[2]  if (mknod(argv[1],S_IFIFO|0666,0)==-1){perror("Error mknod"); exit(-2);}
[3]  if((fd=open(argv[1],O_RDWR))==-1){perror("Error open"); exit(-3);};
     if((pid=fork())==-1) {perror("error en primer fork"); exit(-4);}
     if(pid==0)
          {
          for(i=0;i<=tam;i++)
               {
[4]            if ((pid=fork())==-1) {perror("error en fork 2"); exit(-3);}

               if(pid==0)
                    {
[5]                 if(i==tam) {close(fd);break;}
                    }

               else
                    {
[6]                 wait();
[7]                 sleep(1);
                    write(fd,&texto[i],1);
                    if (i==0) write(fd,"\n",1);
                    close(fd);
                    break;
                    }
               }
          }
     else
          {
          close(fd);
[8]       execl("/bin/cat","cat",argv[1],NULL);
          printf("Incorrecto\n");
          }
     }
```

RESPUESTA

Apartado a)

[1] `unlink` es la llamada al sistema que permite borrar un fichero. Admite como argumento el nombre del fichero, que a su vez ha sido introducido como segundo argumento de ejecución del programa pincipal (`arrgv[1]`), de este modo, la sentencia borra el fichero `DyASO` (en caso de que dicho fichero exista).

[2] la llamada al sistema `mknod(argv[1],S_IFIFO|0666,0)`, sirve para crear un fichero fifo cuyo nombre fue pasado como argumento a la función principal (esto es el fichero "`DyASO`"). 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 mknod:`" 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] La llamada al sistema `open(argv[1],O_RDWR)` abre el fichero fifo recién creado en modo de lectura y escritura (`O_RDWR`) y devuelve un descriptor de fichero `df`. Si se produce un error, al igual que en la sentencia anterior, es mostrado utilizando `perror` y el programa termina con status -3.

[4] `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` que se encuentra al final del código.

[5] La llamada al sistema `close(fd)` se encarga de cerrar el fichero, cuyo descriptor de fichero es `fd`. 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, el proceso 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 `execl("/bin/cat","cat",argv[1],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 `cat` que se encuentra en el directorio `bin`. Los parámetros que se le pasan al ejecutable como entrada son su propio nombre (primer argumento en la convención de UNIX), y el parámetro `argv[1]` cuyo valor es `DyASO` (segundo parámetro pasado al programa original). El parámetro `NULL` sirve para indicar a `execl` que no hay más parámetros de entrada. El resultado es similar a la invocación por línea de comandos de `$ /bin/cat DyASO`, comando que mostrará el contenido del fichero fifo `DyASO` por la salida estándar.

Apartado b)

En el enunciado original había dos erratas de transición que afectaban a las líneas [5] y [8] del programa que eran respectivamente:

```
[5]  if(i==tam) {close(fd);break}
[8]  execl("/bin/cat","cat","argv[1]",NULL);
```

En lugar de las líneas [5] y [8] corregidas que se muestran en esta solución.

Por este motivo se consideran tres soluciones posibles:

SOLUCIÓN A) Se corrige como si no hubiese ninguna errata en el enunciado original, en cuyo caso el funcionamiento del programa sería el siguiente.

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 agrega 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 `DyASO` en segundo.

Hecho esto, se elimina (si es que existe) el fichero `DyASO` del directorio actual (debido a posibles ejecuciones previas del programa). A continuación, se recrea dicho fichero fifo con `mknod` y se abre en modo de lectura y escritura. Si el fichero existiese cuando se invoca `mknod` se produciría un error, por ese motivo se toma la precaución de borrarlo previamente.

El proceso padre que llamaremos P ejecuta el código de la cláusula `else` final del programa. De este modo cierra el fichero que no va a necesitar y ejecuta la orden `cat DyASO` con `execl`. Puesto que las llamadas al sistema de la familia `exec` (y en particular `execl`) sustituyen el código del proceso original con el del ejecutable invocado, la última orden `printf("Incorrecto\n")` **nunca se llega a ejecutar**. La orden `cat` quedará finalmente a la espera de que alguien escriba en el fichero fifo para mostrar el contendido por salida estándar.

Por otra parte, el proceso hijo H0 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`.
2) El proceso hijo H1, comprueba si se trata de la última ejecución del bulcle (`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.
5) El proceso H2 comprueba que no es la última iteración del bucle y para él `i=2`.
6) El proceso padre H1 queda a la espera de su hijo H2.

…

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 17 caracteres y se ha establecido `tam=strlen(texto)-1`, entonces `tam=16`. En esa iteración el proceso padre será H16 con i=16 y su hijo será H17.

El proceso hijo H17 cumple la condición `i==tam` por lo que cierra el fichero, sale del bucle con break y termina su ejecución sin crear más hijos.

Cuando H17 termine despertará el proceso H16, que a su vez esperará un segundo y escribirá en el fichero fifo el carácter `texto[16]` (la `E`), a continuación cierra el fichero y termina el bucle y con ello su ejecución.

El fin del proceso H16 despierta al proceso H15 que espera otro segundo y escribe `texto[15]`, es decir "`j`" en el fichero fifo, lo cierra y termina.

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

Dicho proceso realiza el proceso anteriormente descrito, pero además, imprime un salto de línea final en el fichero fifo al cumplirse la condición `i==0`.

Por otra parte, recordemos que el proceso padre inicial P había invocado la orden `cat` para leer el fichero fifo. La lectura del fichero fifo es bloqueante mientras no hay datos que leer, siempre y cuando algún proceso tenga abierto el fichero para lectura. Si se intenta leer del mismo y no hay ningún proceso en disposición de escribir se devuelve `EOF`, lo que hace que `cat` termine su ejecución.

De este modo, `cat` (proceso P) quedará bloqueado hasta que H16 escriba en el fichero fifo `DyASO`.

Cuando lo haga mostrará "`E`" y volverá a quedarse bloqueado a la espera de que escriba alguien más. Un segundo después H15 se escribirá la "`j`" y `cat` lo mostrará por pantalla. 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 `cat` (pero en modo lectura). Por tanto, una vez escrito el salto de línea final (proceso H0), `cat` ya no queda bloqueado en la lectura del fichero fifo, recibe `EOF` y termina su ejecución.

NOTA: Sea cual sea la planificación de los procesos por parte del SO la salida ha de ser siempre la misma, ya que cada proceso H_i ha de esperar hasta que acabe su hijo H_{i+1} antes de poder escribir en la tubería. Además el mecanismo del fichero FIFO garantiza que las lecturas realizadas por `cat` se harán en el mismo orden que las escrituras.

NOTA2: Obsérvese que la instrucción `printf("Incorrecto\n")` no se llega a ejecutar nunca por lo que el texto "incorrecto" **no** se mostrará por la salida estándar.

SOLUCIÓN B) Dado que en la línea [5] del examen original faltaba un punto y coma tras la sentencia break, el programa no compila y no puede crearse el ejecutable.

SOLUCIÓN C) Se obvia únicamente la errata de la línea [5]. En ese caso, puesto que en la línea [8] del programa original, el argumento "argv[1]" se encuentra entrecomillado, `execl` recibe como entrada el *texto literal* `"argv[1]"` y no el contenido de la variable `argv[1]`. De este modo, la función invocada por `execl` sería:

```
cat argv[1]
```

en lugar de `cat DyASO`. Si en el directorio actual existiese un archivo cuyo nombre fuese argv[1], se mostraría su contenido por salida estándar. En caso de no existir tal fichero se produciría un error:

```
cat: argv[1]: No existe el archivo o el directorio
```