SISTEMAS OPERATIVOS II **Mayo 2012**

Material permitido: **Calculadora NO programable** Tiempo: **2 horas**	**Aviso 1**: Todas las respuestas deben estar debidamente **razonadas**. **Aviso 2**: Escriba sus respuestas con una **letra lo más clara posible**. **Aviso 3**: Evite los **tachones**. **Aviso 4**: Notificación de la salida de las calificaciones, solución del examen y fecha de revisión en la página web de la asignatura: **http://www.uned.es/533032/**

ESTE EXAMEN CONSTA DE 5 PREGUNTAS

Preguntas 1 a 4

1. Explique **razonadamente** si las siguientes afirmaciones son verdaderas o falsas:

I) (*1 p*) En UNIX las señales son muy útiles como mecanismo IPC.

II) (*1 p*) En UNIX los atributos de un fichero se encuentran almacenados en la entrada del directorio al que pertenece.

2. 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.

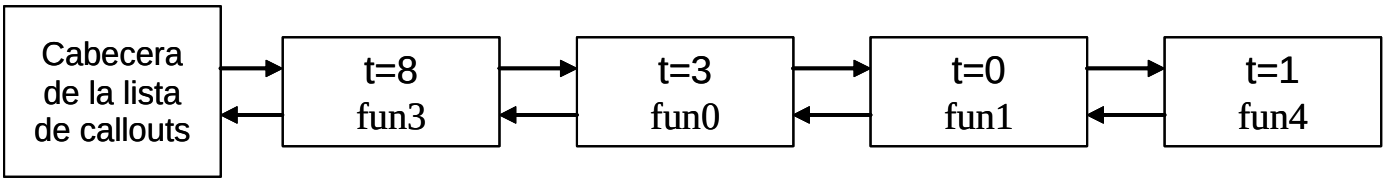
3. (*2 p*) Dibuje un diagrama, **adecuadamente rotulado**, que esquematice las principales acciones que realiza el núcleo durante la ejecución del algoritmo `exec()`.

4. En la figura se muestra la lista de callouts del núcleo del UNIX BSD en un cierto instante de tiempo. Se pide:

a) (*0.5 p*) Explicar brevemente qué es un callout.

b) (*0.5 p*) Determinar el tiempo de disparo (en tics) de fun0, fun1, fun3 y fun4.

c) (*0.5 p*) Supuesto que ha transcurrido un tic, dibujar la lista de callout.



Material permitido: **Calculadora NO programable** Tiempo: **2 horas**	**Aviso 1**: Todas las respuestas deben estar debidamente **razonadas**. **Aviso 2**: Escriba sus respuestas con una **letra lo más clara posible**. **Aviso 3**: Evite los **tachones**. **Aviso 4**: Notificación de la salida de las calificaciones, solución del examen y fecha de revisión en la página web de la asignatura: **http://www.uned.es/533032/**

ESTE EXAMEN CONSTA DE 5 PREGUNTAS

Pregunta 5

5. Al compilar el código C de este programa se crea el ejecutable `j12`. Supóngase que al invocar este programa desde la línea de ordenes del terminal (`$`) se le asocia el *pid* 930 y que la asignación de los *pid* de sus procesos hijos, si se llegaran a crear, se realizaría incrementando en una unidad el *pid* del proceso padre. Suponer además que el intérprete de comandos desde donde se lanza `j12` tiene asociado el *pid* 850. Conteste razonadamente a los siguientes apartados:

a) *(1 p)* Explique el significado de las cinco sentencias enumeradas (**[ ]**) del código del ejecutable `j12`.

b) *(1.5 p)* Explique **detalladamente** el funcionamiento de este programa si se invoca desde el interprete de comandos mediante la orden `$ j12`

```
  #include <signal.h>

  main()
  {

       int a,b;

[1]    if ((a=fork())==0)
       {
            while(1)
            {
                 a=a^7;
[2]              b=raise(SIGCHLD);
[3]              printf("\nMensaje 1[%d]: %d: %d\a\n",getpid(),b, a);
[4]              sleep(3);
            }
       }

       sleep(8);
       kill(a,SIGUSR1);

       sleep(2);
[5]    kill(getppid(),SIGUSR2);
  }
```

SOLUCIÓN EXAMEN MAYO 2012

1. Explique **razonadamente** si las siguientes afirmaciones son verdaderas o falsas:

I) (*1 p*) En UNIX las señales son muy útiles como mecanismo IPC.

II) (*1 p*) En UNIX los atributos de un fichero se encuentran almacenados en la entrada del directorio al que pertenece.

Solución:

I) En UNIX, las señales son útiles para la notificación de eventos pero resultan poco útiles como mecanismo IPC, ya que poseen las siguientes limitaciones:

- Resultan costosas en relación a las tareas que suponen para el sistema.
- Tienen un ancho de banda limitado.
- Una señal puede transportar una cantidad limitada de información

En conclusión, la afirmación es **FALSA**.

II) En UNIX, los *atributos de un fichero* no están almacenados en la entrada del directorio en el que se ubica dicho fichero, sino en una estructura del disco denominada *nodo índice* (nodo-i). Luego la afirmación es **FALSA**.

2. 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.

Solución:

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.

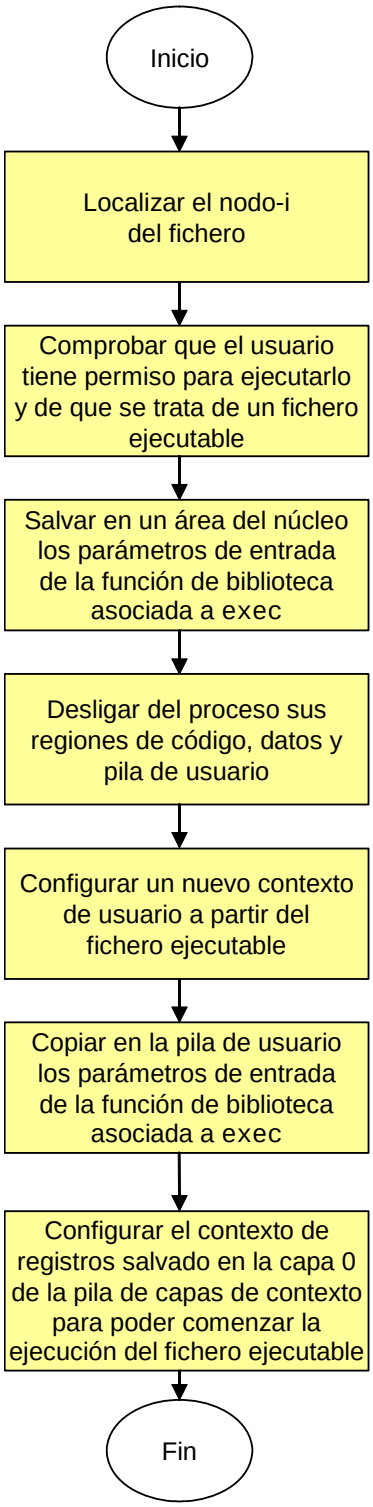
b) De forma general, las órdenes que se pueden ejecutar en un intérprete de comandos pueden ser de dos tipos:

- *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`.

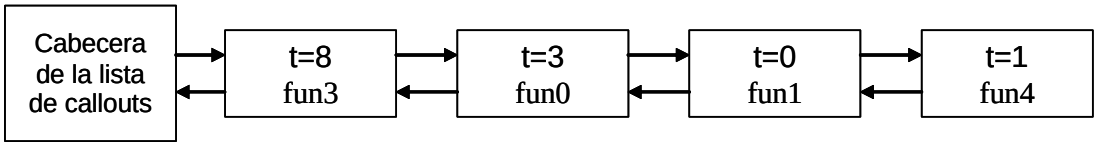
- *Órdenes externas*. Son aquellas órdenes que para poder ser ejecutadas por el intérprete requieren de la búsqueda y ejecución del fichero ejecutable asociado a cada orden. Típicamente son programas ejecutables o shell scripts incluidos en la distribución del sistema o creados por el usuario. La ejecución 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`.

3. (*2 p*) Dibuje un diagrama, **adecuadamente rotulado**, que esquematice las principales acciones que realiza el núcleo durante la ejecución del algoritmo `exec()`.

Solución:



4. En la figura se muestra la lista de callouts del núcleo del UNIX BSD en un cierto instante de tiempo. Se pide:
a) (*0.5 p*) Explicar brevemente qué es un callout.
b) (*0.5 p*) Determinar el tiempo de disparo (en tics) de fun0, fun1, fun3 y fun4.
c) (*0.5 p*) Supuesto que ha transcurrido un tic, dibujar la lista de callout.



Solución:

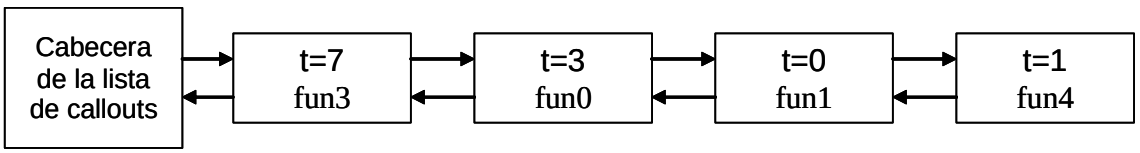
a) Los *callouts* son un mecanismo interno del núcleo que le permite invocar funciones transcurrido un cierto tiempo. Un *callout* típicamente almacena el nombre de la función que debe ser invocada, un argumento para dicha función y el tiempo en tics transcurrido el cual la función debe ser invocada.

b) En la figura del enunciado se muestra la lista de callouts en un cierto instante de tiempo. Se observa que dicha lista contiene cuatro entradas asociados a cuatro callouts. En el UNIX BSD la lista de callouts se ordena en función del tiempo que le resta al callout para ser invocado. A este tiempo comúnmente se le denomina *tiempo de disparo*. Cada entrada de la lista de callouts almacena la diferencia entre el tiempo de disparo de su callout asociado y el tiempo de disparo del callout asociado a la entrada anterior. Por lo tanto la primera entrada de la lista está asociada al callout para la función `fun3` y en ella también se almacena su tiempo de disparo que es **8 tics**. La segunda entrada está asociada al callout para la función `fun0`. En su entrada de la lista se almacena el tiempo que resta para ser invocada con respecto a `fun3`, en este caso es 3 tics. Su tiempo de disparo es la suma de los tiempos almacenados en esta segunda entrada y en la primera entrada, es decir, 8+3= **11 tics**.

   La tercera entrada está asociada al callout para la función `fun1`. En su entrada de la lista se almacena el tiempo que resta para ser invocada con respecto a `fun0`, en este caso es 0 tic. Su tiempo de disparo es la suma de los tiempos almacenados en esta tercera entrada y en las dos entradas anteriores, es decir, 8+3+0= **11 tics**.

   La cuarta entrada está asociada al callout para la función `fun4`. En su entrada de la lista se almacena el tiempo que resta para ser invocada con respecto a `fun1`, en este caso es 1 tics. Su tiempo de disparo es la suma de los tiempos almacenados en esta cuarta entrada y en las tres entradas anteriores, es decir, 8+3+0+1= **12 tics**.

c) En la figura inferior se representa la lista de callouts supuesto que ha transcurrido un tic



5. Al compilar el código C de este programa se crea el ejecutable `j12`. Supóngase que al invocar este programa desde la línea de ordenes del terminal (`$`) se le asocia el *pid* 930 y que la asignación de los *pid* de sus procesos hijos, si se llegaran a crear, se realizaría incrementando en una unidad el *pid* del proceso padre. Suponer además que el intérprete de comandos desde donde se lanza `j12` tiene asociado el *pid* 850. Conteste razonadamente a los siguientes apartados:
a) *(1 p)* Explique el significado de las cinco sentencias enumeradas (**[ ]**) del código del ejecutable `j12`.
b) *(1.5 p)* Explique **detalladamente** el funcionamiento de este programa si se invoca desde el interprete de comandos mediante la orden `$ j12`

```
#include <signal.h>

main()
{

     int a,b;

[1]  if ((a=fork())==0)
     {
          while(1)
          {
               a=a^7;
[2]            b=raise(SIGCHLD);
[3]            printf("\nMensaje 1[%d]: %d: %d\a\n",getpid(),b, a);
[4]            sleep(3);
          }
     }

     sleep(8);
     kill(a,SIGUSR1);

     sleep(2);
[5]  kill(getppid(),SIGUSR2);
 }
```

Solución:

a) El significado de cada una de las sentencias marcadas con **[ ]** de este código es el siguiente:

[1] Llamada al sistema `fork` para crear un proceso hijo. La llamada devuelve al proceso padre el *pid* del proceso hijo y al proceso hijo el valor 0. Dicho valor se almacena en la variable a. Se comprueba dentro de una sentencia condicional if si a es 0, es decir, si se está ejecutando el proceso hijo.

[2] Llamada al sistema `raise` para enviar la señal SIGCHLD al proceso que la invoca. Si se ejecuta con éxito devuelve un 0 en caso contrario devuelve -1. Dicha salida se almacena en la variable b.

[3] Función `printf` para escribir en el dispositivo de salida estándar (típicamente el monitor) el resultado de la llamadas al sistema `getpid()`y los valores de las variables `b` y `a.` La llamada `getpid()` no posee parámetros de entrada, si se ejecuta con éxito devuelve el *pid* del proceso que la invoca y si la llamada falla devuelve el valor -1.

La escritura en el dispositivo de salida estándar se realiza de acuerdo con la siguiente cadena de control

```
"\nMensaje 1[%d]: %d: %d\a\n"
```

Es decir, en pantalla se mostraría lo siguiente: un salto de línea, el mensaje `Mensaje 1`, la apertura de un corchete, el valor expresado en entero decimal del resultado de `getpid()`, el cierre de un corchete, dos puntos, un espacio en blanco, el valor expresado en entero decimal de la variable `b`, dos puntos, un espacio en blanco, el valor expresado en entero decimal de la variable `a` y un salto de línea. Además sonaría un sonido de alerta cuando se muestre este mensaje debido a la secuencia de escape '`\a`'.

[4] Llamada al sistema `sleep` para suspender la ejecución del proceso que la invoca durante 3 segundos.

[5] Llamada al sistema `kill` para enviar una señal SIGUSR2 al proceso padre (ya que `getppid()` devuelve el pid del proceso padre que invoca a esta llamada) del proceso que invoca la llamada.

b) Al escribir la orden `$ j12` se comienza a ejecutar el programa `j12.` Supóngase que a la ejecución de dicho programa se le asocia el proceso A, cuyo *pid*=930 según el enunciado.

El proceso A hace una llamada al sistema `fork` para crear un proceso hijo (proceso B) al cual se le asigna, de acuerdo con el enunciado, *pid*= 931. La llamada al sistema `fork` devuelve 0 al proceso hijo y 931 al proceso padre. Estos valores se almacenan en la variable `a` del padre y del hijo, respectivamente.

Depende del planificador que proceso se ejecuta después del `fork`, el padre `A` o el hijo `B`. Supóngase que se ejecuta primero el proceso hijo `B`. Entonces entra en un bucle `while` infinito, en cada pasada del bucle se realizan las siguientes acciones.

En primer lugar se realiza una operación binaria XOR, entre la variable `a` (que contiene el valor `0` en la primera pasada del bucle) y el número `7`, es decir $a \oplus 7 = \overline{a} \cdot 7 + a \cdot \overline{7}$, almacenando el resultado en `a`.

En segundo lugar se invoca a la llamada al sistema `raise` que envía al proceso `B` la señal SIGCHLD, la acción por defecto asociada a esta señal es ignorarla, por lo que el proceso `B` no

se ve perturbado por dicha señal. Esta llamada al sistema devuelve un 0 si se ejecuta con éxito y -1 en caso contrario. Se va a suponer que se ejecuta con éxito.

En tercer lugar se imprime en pantalla el mensaje

```
Mensaje 1[931]: 0: 7
```

si se trata de la primera, tercera, ... pasada, o el mensaje

```
Mensaje 1[931]: 0: 0
```

si se trata de la segunda, cuarta. ... pasada.

En cuarto lugar invoca a la llamada al sistema `sleep` que suspende la ejecución del proceso B durante 3 segundos.

Cuando se planifica el proceso padre A invoca a la llamada al sistema `sleep` que suspende su ejecución durante 8 segundos.

Cuando termina su suspensión invoca a la llamada al sistema `kill` para enviar la señal SIGUSR1 al proceso hijo B. Esta señal tiene establecida como acción por defecto la terminación del proceso que la recibe.

A continuación el proceso A invoca a la llamada al sistema `sleep` que suspende su ejecución durante 2 segundos.

Finalmente invoca a la llamada al sistema `kill` para enviar la señal SIGUSR2 a su proceso padre, que es el intérprete de comandos (*pid*=850). Esta señal tiene establecida como acción por defecto la terminación del proceso que la recibe. Por lo que el intérprete de comandos desde donde se invoco al programa `j12` termina y se cierra su ventana.

Debe tenerse en cuenta que durante los 8 segundos que duerme el proceso padre A al proceso hijo B le da tiempo a mostrar por pantalla los siguientes tres mensajes antes de que se cierre la ventana del intérprete de comandos:

```
Mensaje 1[931]: 0:7

Mensaje 1[931]: 0: 0

Mensaje 1[931]: 0:7
```