DISEÑO Y ADMINISTRACIÓN DE SISTEMAS OPERATIVOS **Enero de 2014**

Material permitido: **Calculadora NO programable.** Tiempo: **2 horas.** N1	**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 qué máscara representa un fichero regular donde únicamente el propietario del fichero puede leer, escribir y ejecutar el fichero; y además el bit `S_ISUID` está activado y los bits `S_ISGID` y `S_ISVTX` están activados.

RESPUESTA:

```
-rws --S --T
```

- (Página 66. Apartado 2.6.1. Máscara de modo simbólica). La estructura de la máscara de modo simbólica es:
  s9s8s7s6s5s4s3s2s1s0
- El carácter s9 (-), indica que el tipo de fichero es ordinario.
- El carácter s8 indica, si se encuentra habilitado el permiso de lectura para el propietario del fichero, (r) indica el caso afirmativo.
- El carácter s7 indica, si se encuentra habilitado, el permiso de escritura para el propietario del fichero: (w) indica el caso afirmativo.
- El carácter s6, puede indicar varias cosas: Si vale (s) indica que el bit S_ISUID está activado y que se encuentra habilitado el permiso de ejecución para el propietario del fichero.
- El carácter s5 indica, si se encuentra habilitado, el permiso de lectura para los miembros del grupo al que pertenece el propietario del fichero. El valor (-) indica el caso negativo.
- El carácter s4 indica, si se encuentra habilitado, el permiso de escritura para los miembros del grupo al que pertenece el propietario del fichero. El valor (-) indica caso negativo.
- El carácter s3, puede indicar varias cosas: Si vale (S) indica que el bit S_ISGID está activado y que se encuentra deshabilitado el permiso de ejecución para los miembros del grupo al que pertenece el propietario del fichero.
- El carácter s2 indica, si se encuentra habilitado, el permiso de lectura para el resto de usuarios. El valor (-) indica el caso negativo.
- El carácter s1, indica si se encuentra habilitado el permiso de escritura para el resto de usuarios. El valor (-) indica el caso negativo.
- El carácter s0, puede indicar varias cosas: Si vale (T) indica que el bit S_ISVTX está activado y que se encuentra deshabilitado el permiso de ejecución para el resto de usuarios.

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

i) (*1p*) Una tubería sirve para transmitir datos a múltiples procesos de forma simultánea.
ii) (*1p*) Se tiene un computador con una memoria principal de capacidad C_{MP}= 16 Mbytes y un tamaño de página S_P= 1 Kbyte. El número total de marcos de página es 8192.

RESPUESTA:

i) La afirmación es **falsa** porque una tubería no puede ser utilizada para transmitir datos a múltiples procesos de forma simultánea, puesto que los datos son borrados una vez leídos de la tubería (página 265)

ii) La afirmación es **falsa.** El número de marcos de página se calcula a partir de la fórmula 5 del tema 7 (pág 325):
$NTM=C_{MP}/S_P= 2^4 \cdot 2^{20}$Bytes/$2^{10}$ Bytes/marco =2^{13}=16384 marcos de página.

3. (*2 p*) Conteste **razonadamente** a las siguientes preguntas:
a) (*0.5 p*) ¿Qué es una región de un proceso?
b) (*0.5 p*) ¿Cuáles son las principales regiones en que se descompone un proceso?
c) (*1 p*) ¿Cuál es el contenido y las características de las regiones en que se descompone un proceso?

RESPUESTA:

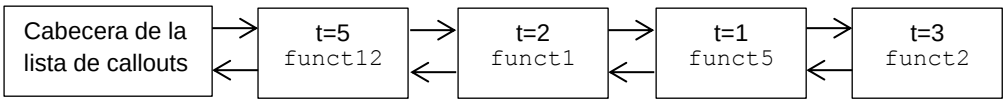
Apartado a) Una *región* de un proceso es un área de direcciones virtuales (o lógicas) contiguas del espacio de direcciones virtuales del proceso.

Apartado b) El espacio de direcciones de memoria virtual de un proceso consiste al menos de tres regiones: *la región de código* (o texto), *la región de datos* y *la región de pila*. Adicionalmente, puede contener *regiones de memoria compartida*, que posibilitan la comunicación de un proceso con otros procesos.

Apartado c) Las *regiones de código y de datos* se corresponden con las secciones de código y datos del fichero ejecutable. La *región de datos inicializados* o *zona estática de la región de datos* es de tamaño fijo. Por el contrario el tamaño de la *región de datos no inicializados* o *zona dinámica de la región de datos* puede variar durante la ejecución de un proceso.

La *región de pila* o *pila de usuario* se crea automáticamente y su tamaño es ajustado dinámicamente en tiempo de ejecución por el núcleo. La ejecución del código del programa irá marcando el crecimiento o decrecimiento de la pila, el núcleo asignará espacio para la pila conforme se vaya necesitando. La pila está constituida por *marcos de pila lógicos*. Un marco se añade a la pila cuando se llama a una función y se extrae cuando se vuelve de la misma. Existe un registro especial de la máquina denominado *puntero de la pila* donde se almacena la dirección, dependiendo de la arquitectura de la máquina, de la próxima entrada libre o a la última utilizada. Análogamente, la máquina indica la dirección de crecimiento de la pila, hacia las direcciones altas o bajas. Un marco de pila contiene usualmente la siguiente información: los parámetros de la función, sus variables locales y las direcciones almacenadas en el instante de la llamada a la función en diferentes registros especiales de la máquina, como por ejemplo, el contador del programa y el puntero de la pila.

4. (*2p*) 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) (*0.75 p*) Determinar el tiempo de disparo (en tics) de `func1`, `func2`, `func5` y `func12`.
c) (*0.75 p*) Supuesto que ha transcurrido un tic, dibujar la lista de *callout*.

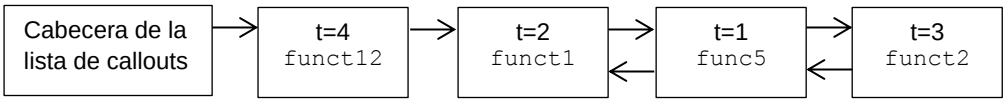
RESPUESTA:

Apartado 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.

Apartado 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 `func12` y en ella también se almacena su tiempo de disparo que es **5 tics**.
- La segunda entrada está asociada al *callout* para la función `func1`. En su entrada de la lista se almacena el tiempo que resta para ser invocada con respecto a `func12`, en este caso es 2 tics. Su tiempo de disparo es la suma de los tiempos almacenados en esta segunda entrada y en la primera entrada, es decir, 5+2= **7 tics**.
- La tercera entrada está asociada al *callout* para la función `func5`. En su entrada de la lista se almacena el tiempo que resta para ser invocada con respecto a `func1`, en este caso es 1 tic. Su tiempo de disparo es la suma de los tiempos almacenados en esta tercera entrada y en las dos entradas anteriores, es decir, 5+2+1= **8 tics**.
- La cuarta entrada está asociada al *callout* para la función `func2`. En su entrada de la lista se almacena el tiempo que resta para ser invocada con respecto a `func5`, en este caso es 3 tics. Su tiempo de disparo es la suma de los tiempos almacenados en esta cuarta entrada y en las tres entradas anteriores, es decir, 5+2+1+3= **11 tics**

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



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 y describir su salida asumiendo que es ejecutado por el administrador del sistema, en el directorio actual existe un directorio *dir1* y dentro de él un único archivo *prueba.txt*, y además que la raíz del sistema de archivos es de tipo *ext4* y se encuentra alojada en el dispositivo *sda1*.

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

     void main(void)
     {
          int res,res2,pid;
[1]       if ((res=mount("/dev/sda1","./dir1","ext4",0,NULL))==0)
               {
[2]            if ((pid=fork())==0)
                    {
                    printf("\nPrimer ls:\n");
[3]                 execl("/bin/ls","ls","dir1",NULL);
[4]                 perror("fallo en execl");}
               else
                    {
[5]                 wait();
[6]                 sleep(1);
[7]                 if (umount("./dir1")==1) perror("fallo en umount");
                    printf("\nSegundo ls:\n");
[8]                 system("ls dir1");
                    printf("Fin.\n");
                    }
               }
          else
               {
               perror("Error en mount");
               }
     }
```

RESPUESTA

Apartado a):

[1] La llamada al sistema `mount` monta el sistema de ficheros existente en `/dev/sda1` de tipo `ext4` en el directorio `dir1` de la ruta actual (`./`). Si la llamada al sistema se ejecuta con éxito devuelve a la variable `res` y la condición del `if` se cumple. En caso contrario se ejecutaría el contenido del `else` final.

[2] La llamada al sistema `fork()` crea un proceso hijo devolviendo 0 al hijo recién creado y el `pid` del proceso hijo al padre. De este modo la condición del `if` se cumplirá para el hijo (que ejecutará el contenido a continuación de él). El padre ejecutará el contenido del `else` siguiente.

[3] La llamada al sistema `execl` substituye el contexto del proceso en ejecución por el archivo ejecutable `/bin/ls` pasándole como argumentos de entrada su nombre "`ls`" y el parámetro `dir1`. El parámetro `NULL` sirve para indicar que no hay parámetros adicionales. El resultado de esta instrucción es que el proceso hijo pasa a ser el proceso que se crearía si en la línea de comandos se teclease `ls dir1`. Esto es, mostrará el contenido del directorio `dir1` por la salida estándar.

[4] La función `perror()` muestra el contenido de la variable global `errno` que almacena el descriptor del error que se ha producido en la llamada al sistema anterior. Si **[3]** se ha ejecutado con éxito el contexto del proceso hijo habrá sido substituido por el nuevo proceso `ls` de tal modo que esta sentencia no llegará a ejecutarse. En caso de que se produzca un error `perror` mostrará la cadena "`fallo en execl`" seguida de "`:`" y del mensaje asociado al error que se ha producido.

[5] 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.

[6] La llamada al sistema `sleep` suspende al proceso durante un segundo.

[7] La llamada al sistema `umount` desmonta el sistema de archivos `/dev/sda1` que previamente se había montado en `dir1`. En este momento, `dir1` vuelve quedar como estaba en un principio. En caso de que se ejecute con éxito devolverá 0. En caso contrario se cumple la condición del `if` y se ejecutará la función `perror` indicando "`fallo en umount:`" seguido del descriptor del error que se ha producido.

[8] La función de librería `system()` permite ejecutar la orden el intérprete de comandos asociado al proceso actual. La orden que se ejecuta es "`ls dir1`" luego el resultado es similar al de la llamada al sistema **[3]**. Hay no obstante, una diferencia notable, mientras que en **[3]** se eliminaba el contexto del proceso actual (substituyéndolo por el proceso nuevo), en **[8]** simplemente se espera a que termine la ejecución de `ls dir1` y después se continúa con la operación normal del programa.

Apartado b):

El funcionamiento del programa sería el siguiente. En primer lugar se montaría el contenido del sistema de ficheros `/dev/sda1` en el directorio `./dir1`, de este modo el contenido del directorio `dir1` quedaría oculto y sólo sería accesible el contenido nuevo (la raíz del sistema de ficheros).

En segundo lugar se crearía un proceso hijo, este al ejecutarse mostrará el texto "`primer ls`" y, a continuación, invocará al programa `ls` con el argumento `dir1`, esto hace que el contexto del proceso hijo sea substituido por el del proceso `ls` recién creado y que la salida de `ls` se muestre por la salida estándar como si la hubiese producido el proceso hijo.

Puesto que se ha montado el sistema de ficheros correspondiente a `/dev/sda1`, esto es, el sistema de ficheros raíz dentro de la carpeta `dir1`, el contenido mostrado por `ls` es el de la raíz del sistema de archivos, quedando oculto el contenido previo de `dir1` (que era el archivo prueba.txt). De este modo la salida del proceso hijo sería similar a la siguiente:

```
Primer ls:
bin     etc             lib         opt     tmp
boot    home            usr
…
```

Por otra parte el proceso padre quedaría a la espera de que el hijo terminase su función. Esperaría un segundo (`sleep(1))` para asegurarse de que el sistema de ficheros no está siendo utilizado y puede desmontarse y finalmente desmontaría el sistema de archivos de `dir1`. De este modo en `dir1` volvería a estar accesible el fichero `prueba.txt`.

A continuación se imprime "`Segundo ls`" y se invoca de nuevo `ls` mediante la función `system`. El resultado es que se ejecuta "`ls dir1`". Una vez concluido se devuelve el control al padre que imprime "`Fin.`" y termina. Por lo tanto durante la ejecución del padre se muestra lo siguiente por pantalla:

```
Segundo ls:
prueba.txt
Fin.
```