Solución Examen Parcial de Sistemas Operativos - 20 de Marzo de 2013

Grado en Ingeniería Informática

NOTAS:

* La fecha de publicación de las notas, así como de revisión se notificarán por Aula Global.
* Para la realización del presente examen se dispondrá de **1:25 hora.**
* **No** se pueden utilizar libros **ni** apuntes
* Será necesario presentar el DNI o carnet universitario para realizar la entrega del examen

---

Ejercicio 1 (4 puntos).

Responda de forma clara y concisa a las siguientes preguntas. *Espacio máximo una cara por pregunta.*

1.- ¿Cómo se solicita una llamada al sistema operativo? Acompáñelo con un esquema gráfico de los pasos a seguir.

SOLUCION

Se solicita mediante un mecanismo de interrupciones. Cuando un proceso en ejecución la solicita, éste utiliza una instrucción TRAP que genera una interrupción.

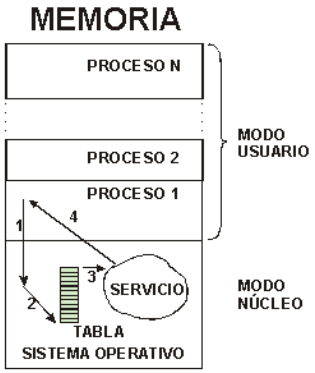
Cuando se programa en un lenguaje de alto nivel, la solicitud de servicios al sistema operativo se hace mediante una llamada a una función determinada, que se encarga de generar la llamada al sistema y el trap correspondiente.

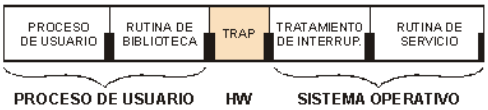
En general estas funciones que solicitan los servicios del sistema operativo se componen de:

- Una parte inicial que prepara los parámetros del servicio de acuerdo con la forma en que los espera el sistema operativo.
- La instrucción TRAP que realiza el paso al sistema operativo.
- Una parte final que recoge los parámetros de contestación del sistema operativo, para devolverlos al programa que hizo la llamada.

Estas funciones se encuentran en una biblioteca del sistema y se incluyen en el código en el momento de su carga en memoria. Para completar la imagen de que se está llamando a una función, el sistema operativo devuelve un valor, como una función real. Al programador le parece, por tanto, que invoca al sistema operativo como a una función. Sin embargo, esto no es así, puesto que lo que hace es invocar una función que realiza la solicitud al sistema operativo.

La Figura muestra todos los pasos involucrados en una llamada al sistema operativo, indicando el código que interviene en cada uno de ellos. Como se muestra en la Figura, la rutina de tratamiento de la interrupción de TRAP usa una tabla interna del S.O. para determinar qué rutina activar dependiendo de cuál es la llamada solicitada.



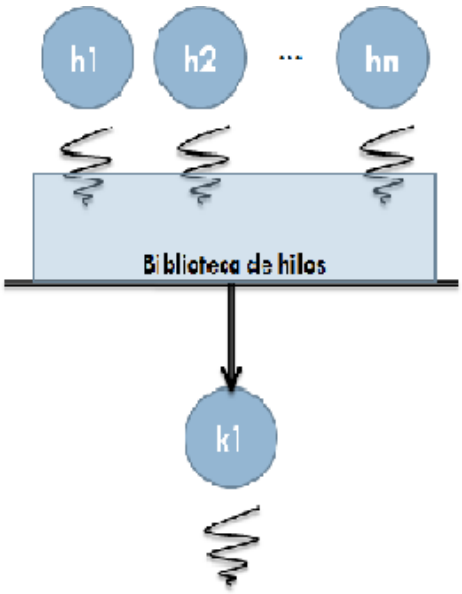


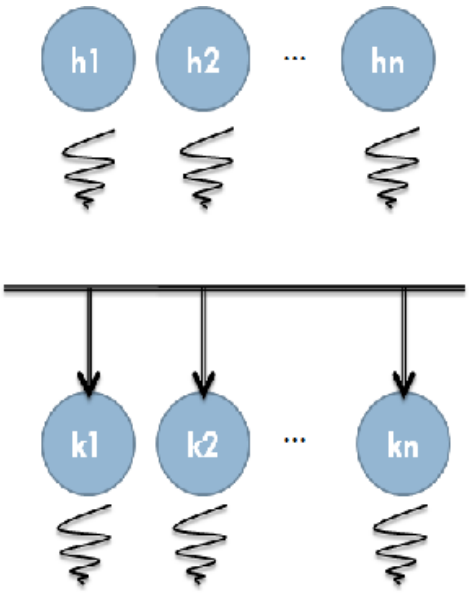
2.- ¿Qué es mejor si un proceso con múltiples threads hace llamadas al sistema bloqueantes, crear threads de biblioteca o de kernel? Ponga un ejemplo con una llamada **write**.

SOLUCION

Los hilos de biblioteca hacen corresponder múltiples hilos de usuario a un único hilo del núcleo, como se ve en la primera figura. Si se produce una llamada bloqueante (read, write, …) se bloquean todos los hilos y el proceso mismo.

Los hilos de kernel hacen corresponder un hilo del kernel a cada hilo de usuario, de forma que el sistema operativo conoce y planifica todos los threads de forma individual. De esta forma, si se produce una llamada bloqueante (read, write, …) solo se bloquea el hilo que hace la llamada y no todo el proceso.





Sea un proceso con 3 hilos en el que th2 hace una llamada **write**. El caso A) muestra lo que ocurre cuando son hilos de biblioteca. El caso b) muestra cuando son hilos de kernel. En este

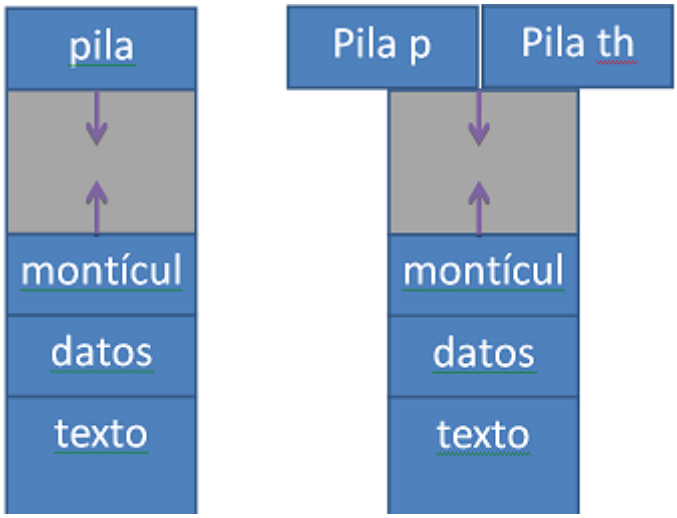
caso se considera el procesos como un thread más. Al bloquearse Th2 se activa de nuevo el proceso.

Proceso	Ejecución		Proceso	Listo	
Th1, Th2, Th3			Th1	Listo	
Th2 -> write	Scheduler activo		Th2	Ejecución	
Proceso	Bloqueado		Th3	Listo	
Th1, Th2, Th3			Th2 -> write	Scheduler activo	
			Th2	Bloqueado	
			Proceso	Ejecución	

3.- Dibuje cómo sería la imagen de memoria de un proceso y la del mismo proceso después de haber creado un thread. Explique las diferencias.

SOLUCION

La imagen de memoria de un proceso está formada por los **espacios de memoria** que un proceso está autorizado a utilizar. Un proceso incluye al menos los siguientes segmentos de memoria: texto, datos estáticos, datos dinámicos (heap) y pila (stack). Cuando el mismo proceso crea un thread, la imagen de memoria incluye una nueva pila para el thread creado. En la figura siguiente se muestra la situación. A) indica el proceso original. B) indica el proceso una vez creado el thread.



4.- ¿Se puede implementar una tubería entre 2 procesos que no son padre e hijo? Expliqué la razón.

SOLUCION

Sí. Se puede crear una tubería entre abuelos y nietos y entre hermanos, no solo entre padre e hijo. La única limitación para conectar con pipes es que los procesos deben estar dentro del árbol de procesos creados por un ancestro común.

Como ejemplo, el programa siguiente muestra una comunicación por pipes entre abuelo, hijo y nieto.

```
#include <unistd.h>
#include <signal.h>
main () {
  int A[2], B[2], C[2];
  int pid, m=-1,n=-1;

  pipe(A); pipe(C);
  if ((pid=fork())!=0) {
    printf("Abuelo\n")
    write(A[1],&pid,sizeof(pid));   /* escribe por pipe A al hijo su pid*/
    read(C[0],&m,sizeof(m)); /*Lee por pipe C la identidad del hijo o el
nieto */
    read(C[0],&n,sizeof(n)); /*Lee por pipe C la identidad del hijo o el
nieto */
    kill(m,SIGTERM); kill(n,SIGTERM);  /* Mata a ambos procesos */
  } else { /* el hijo crea al nieto y se conectan por el pipe B */
    pipe(B);
    if ((pid=fork())!=0) {
      printf("Padre 1\n");
      read(A[0],&m,sizeof(m));  /* lee por el pipe A su pid */
      write(B[1],&pid,sizeof(pid)); /* escribe por pipe B al nieto su
pid*/
      pause(); /* Se bloquea esperando señal de terminación */
    } else {
      printf("Nieto \n");
      read(B[0],&m,sizeof(m)); /* lee por pipe A su pid desde el padre*/
      n==getppid();
      write(C[1],&m,sizeof(m)); write(C[1], &n ,sizeof(n));
      pause();/* Se bloquea esperando señal de terminación */
    }
  }
}
```

Ejercicio 2 (3 puntos)

Se quiere hacer un programa que permita tratar ficheros formados por registros con el siguiente formato:

```
struct persona{
            int nia;          //4 bytes
            char nombre[256]; //256 bytes
            int curso;       // 4 bytes };
```

Se supone que el NIA empieza por 1 y crece consecutivamente.

Se pide:

Escribir un programa que reciba el nombre del fichero como argumento y permita hacer 3 operaciones:

1.- Contar el número de alumnos que hay en total y los que son del curso 2 y mostrar ambos números por pantalla.
2.- Acceder a un alumno en el registro n y mostrar el registro por pantalla.
3.- Buscar un registro de alumno con nombre “Pepe” y mostrarlo por pantalla. Si no hay ninguno se mostrará el mensaje "No hay ninguno".

El programa debe ejecutar un bucle que pida por pantalla el número de operación y llama al procedimiento de la misma. Esto durante tres veces. Al acabar se debe cerrar el fichero.

SOLUCION

```
#include <unistd.h>

#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

struct persona{

    int nia;          //4 bytes

    char nombre[256]; //256 bytes

    int curso;       // 4 bytes

};


int contar_alumnos()

{

    struct persona alumno;

    int contador = 0, fd;
```

```
    int segundo = 0;
        fd = open ((char *)argv[1], O_RDONLY);
    if (fd < 0){
       printf("El archivo no existe \n");
       return(-1);
       }
    while (read(fd, &alumno, sizeof(struct persona)) > 0) {
      contador++;
      if (alumno.curso == 2)
           segundo++;
    }
    printf ("Numero total de alumnos %d \n", contador);
    printf ("Numero de alumnos de segundo curso %d \n", segundo);
    close(fd);
    return(0);
}


int acceder_alumno(int n)
{
    struct persona alumno;
    int fd;
    fd = open ((char *)argv[1], O_RDONLY);
    if (fd < 0){
       printf("El archivo no existe \n");
       return(-1);
    }
    lseek(fd,(n-1)*sizeof(struct persona), SEEK_SET);
    read(fd,&alumno, sizeof(struct persona));
    printf ("NIA %d Nombre %s Curso %d \n", alumno.nia, alumno.nombre,
alumno.curso);
    close(fd);
    return(0);
```

```
}

int mostrar_alumno(char * nombre)

{

    struct persona alumno;

    int existe = 0, fd;

    fd = open ((char *)argv[1], O_RDONLY);

    if (fd < 0){

       printf("El archivo no existe \n");

       return(-1);

    }

    while (read(f,&alumno, sizeof(struct persona)) > 0) {

      if (strcmp(alumno.nombre, nombre)){

           existe = 1;

           printf ("NIA %d Nombre %s Curso %d \n", alumno.nia, alumno.nombre,
alumno.curso);

      }

    }

    if (!existe)

        printf ("No hay ninguno \n");

    close (fd);

    return(0);

}


int main(int argc, char *argv[])

{

    int fd, op, i;  int n = 56;

    for (i=0; i<3; i++) {

        printf("Introduce un numero: ");

        scanf("%d",&op);

        switch (op){

           case 1:

                contar_alumnos();
```

```
                break;
            case 2:
                acceder_alumno(n);
                break;
            case 3:
                mostrar_alumno("Pepe");
                break;
            default:
                printf ("Operacion erronea \n");
        }
    }
}
```

Ejercicio 3 (3 puntos)

Sea el siguiente programa denominado **bucle_procesos**:

```
int main (int argc, char **argv) {

    int contador, i, pid;

    contador = atoi(argv[1]);

    i = 0;

    while (i < contador) {

        pid = fork();

        i++;

    }

}
```

a.) Explique qué hace y dibuje un diagrama con la jerarquía de procesos que se origina cuando se ejecuta el mandato `bucle_procesos 4`.
b.) Explique brevemente qué es un proceso *zombie* y cuándo se producen.
c.) ¿Se generan procesos *zombies* al ejecutar el mandato anterior? Explique su respuesta.
d.) Modifique el programa anterior para que todos los procesos generados compartan una tubería en la que sólo el padre pueda escribir y los demás hijos puedan leer, compitiendo por los datos que escribe el padre. Y que el proceso hijo que lea lo que escribe el padre muestre por pantalla su pid y lo que ha leído

SOLUCION

a Se crea un total de 16 procesos incluido el padre.

- Padre
  - Hijo 0
    - Hijo 0.1
      - Hijo 0.1.2
        - Hijo 0.1.2.3
      - Hijo 0.1.3
    - Hijo 0.2
      - Hijo 0.2.3
    - Hijo 0.3
  - Hijo 1
    - Hijo 1.2
      - Hijo 1.2.3
    - Hijo 1.3
  - Hijo 2
    - Hijo 2.3
  - Hijo 3

Cada hijo directo del padre crea hijos hasta que el contador llega a tomar el valor 3. Y cada hijo indirecto también sigue creando hijos hasta que el contador llega a 3 por lo que se crea una

jerarquía de descendientes asimétrica ya que los primeros hijos tienen más descendencia que los últimos

b.- Un **proceso zombie** es un proceso que ha completado su ejecución pero no puede terminar porque no hay nadie esperando su señal SIGCHLD, lo que significa que su padre a muerto. Como resultado el zombie aun tiene una entrada en la tabla de procesos y ocupa todos sus recursos de memoria, dado que no se pueden desreferenciar para que puedan ser usados por otros procesos.

Los zombies pueden ser identificados ejecutando el comando de Unix "ps –ax" por la presencia de una Z en la columna de estado. Actualmente, los zombies solo existen por cortos períodos de tiempo, hasta que el proceso init se hace cargo de ellos y recibe su señal SIGCHLD.

c.- Sí se crean procesos zombies dado que el padre crea procesos y termina sin esperar a que los hijos lo hagan. Por tanto estos no le pueden enviar la señal SIGCHLD para terminar y deben esperar a que el proceso init les adopte.

d.- A continuación se muestra el programa modificado.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (int argc, char **argv) {
   int contador, i, pid;
  int p[2], dato, leido=1;
  pipe(p);
   contador = atoi(argv[1]);
   i = 0;
   while (i < contador) {
      pid = fork();
      if (pid==0){
            close(p[1]);
            while (leido>0){
              leido=read (p[0], &dato, sizeof(int));
               if (leido>0)
                 printf ("Proceso %d ha leído de tubería el dato %d\n", getpid(), dato);
           }
           exit(0);
      }
```

```
      i++;
   }
   close (p[0]);
   for (i=0; i<10;i++)
     write(p[1], &i, sizeof(int));
}
```