SISTEMAS OPERATIVOS - CUESTIONES
11 de septiembre de 2017

Nombre _____ DNI _____

Apellidos _____ Grupo _____

Cuestión 1. (1 punto) Un proceso de usuario crea un fichero e inmediatamente después se posiciona en el byte 700000 empleando la llamada al sistema `lseek()`. A continuación escribe 10 bytes.

a) ¿Cuántos bloques de disco ocupa ahora el fichero (incluyendo bloques de índices)? Justifique la respuesta. Asúmase que se emplea un sistema de ficheros tipo UNIX cuyos nodos-i contienen 10 punteros directos, 1 puntero indirecto simple, 1 puntero indirecto doble y 1 puntero indirecto triple. En este sistema, los bloques tienen un tamaño de 4 Kbytes y el tamaño de los punteros (direcciones de bloques de disco) es de 32 bits. Supóngase que para ahorrar espacio en disco, la implementación del sistema de ficheros UNIX asigna un bloque de datos para el fichero únicamente cuando el bloque lógico correspondiente se modifica por primera vez.

b) Indique cuántos bloques de disco ocuparía el fichero en el caso en el que se hubiera empleado un sistema de ficheros FAT con bloques de 4KB. Razone la respuesta.

Cuestión 2. (1 punto) Indique cuáles de las siguientes afirmaciones son verdaderas y cuáles falsas. En el caso de que considere que la respuesta es falsa, indique el motivo.

a. Si un módulo del kernel accede a una zona de memoria no permitida, el kernel lo detectará y descargará el módulo automáticamente.
b. Dos drivers en Linux (activos simultáneamente) podrían tener asignado el mismo major number. En tal caso gestionarían peticiones emitidas por programas de usuario (p.ej., lectura/escritura) sobre ficheros de dispositivo que tuvieran distintos minor numbers.
c. Desde un módulo del kernel (programado en C) podemos usar cualquier función de la librería estándar de C, como por ejemplo `printf()`.
d. Cada vez que se carga un módulo del kernel Linux, el SO crea un nuevo proceso de usuario en cuyo mapa de memoria residirá el código del módulo. Dicho proceso se activará cuando algún componente del sistema use los servicios que exporta el módulo.

Cuestión 3. (1 punto) La función `system()` de la biblioteca estándar de "C" permite crear un nuevo proceso shell hijo que invoca el comando que se pasa como parámetro a la función, y devuelve el código de salida (*status*) del shell.

```
int system(const char *command);
```

El proceso de usuario que invoca `system()` se queda bloqueado hasta que el comando finalice.

Implemente la función `system()` empleando llamadas al sistema. (**Nota**: Para ejecutar un comando se puede invocar al shell de la siguiente forma: `/bin/bash -c '<comando>'` . Ejemplo: `/bin/bash -c 'ls -l'`).

Cuestión 4. (1 punto) Considere la siguiente solución al problema de los lectores/escritores que hace uso de dos variables condicionales, un cerrojo y otras variables compartidas:

```
mutex mtx;
condvar wrcond;
condvar rdcond;
int nr_readers=0;
int nr_writers=0;
bool writer_in_cs=false;

/* Auxiliary functions */
void ReaderEnter();
void ReaderExit();
void WriterEnter();
void WriterExit()


void ReaderThread() {
    while(true) {
        ReaderEnter();
        <Reader critical section>
        ReaderExit();
    }
}


void WriterThread() {
    while(true) {
        WriterEnter();
        <Writer critical section>
        WriterExit();
    }
}
```

```
void ReaderEnter() {
    lock(mtx);
    while (nr_writers >0)
      cond_wait(rdcond,mtx);
    nr_readers++;
    unlock(mtx);
}

void ReaderExit() {
    lock(mtx);
    nr_readers--;
    if (nr_readers==0)
      cond_signal(wrcond);
    unlock(mtx);
}


void WriterEnter() {
    lock(mtx);
    nr_writers++;
    while (writer_in_cs || nr_readers > 0)
      cond_wait(wrcond,mtx);
    writer_in_cs=true;
    unlock(mtx);
}

void WriterExit() {
    lock(mtx);
    nr_writers--;
    writer_in_cs=false;
    if (nr_writers>0)
      cond_signal(wrcond);
    else
      cond_broadcast(rdcond);
    unlock(mtx);
}
```

a. Con la solución proporcionada, indique cuál de los dos colectivos (lectores o escritores) tiene más prioridad que el otro a la hora de acceder a la sección crítica. Justifique su respuesta.

b. Modifique el código de las funciones `ReaderEnter()`, `WriterEnter()`, `ReaderExit()` y `WriterExit()` para invertir la prioridad, favoreciendo de este modo al colectivo que resultaba perjudicado con la solución anterior.

```
void ReaderEnter() {




}
```

```
void ReaderExit() {




 }
```

```
void WriterEnter() {




}
```

```
void WriterExit() {




}
```

Cuestión 5. (1 punto) Un proceso en UNIX ejecuta el siguiente código:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#define N 6

int A [ N ] = { 0 };
int S = 0;
int P = 1;

void * sum ( void * arg ) {
        int i , my_sum = S;
        for ( i = 0; i < N; i ++)
        my_sum += A [ i ];
        S = my_sum;
        return NULL;
}
void * product ( void * arg ) {
        int i , prod = P;
        for ( i = 0; i < N; i ++)
        prod = prod * A [ i ];
        P = prod;
        return NULL;
}
```

```
int main ( int argc , char * argv [])
{
        int pid , i;
        pthread_t th1 , th2;

        pid = fork ();
        if ( pid == 0 ) {
        for ( i = 0 ; i < N ; i ++)
                A [ i ] = i+1;
        }
        else {
        wait ( NULL );
        }
        pthread_create (& th1 , NULL , sum , NULL );
        pthread_create (& th2 , NULL , product , NULL );
        pthread_join ( th1 , NULL );
        pthread_join ( th2 , NULL );
        printf ( "pid=%d ppid=%d sum: %d product: %d\n" ,
                getpid (), getppid (), S, P);
        return 0;
}
```

Explique razonadamente cuántos hilos se llegan a ejecutar concurrentemente.
Asumiendo que el proceso padre tiene PID 2000 y es hijo de Init y que los PID se van asignando de forma consecutiva, indique qué se mostrará por pantalla al ejecutar el código. Explique si el resultado depende del orden en que se ejecuten los procesos y los hilos.

Cuestión 6. (1 punto) Estudie el siguiente código que se ejecutará en un sistema Linux e indique si cada una de las afirmaciones que se indican a continuación son ciertas o falsas **JUSTIFICANDO SU respuesta para cada una de ellas**. Asuma que, tras alojarse el marco de pila de la función `main()`, el sistema sólo ha asignado una página para pila, una para código y las necesarias para los datos globales (tamaño de página de 1 KB).

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

int len = 1024;

char get_character(FILE *fd, int i) {
char c;
lseek(fd, i, SEEK_SET);
if (c=getc(fd)!=EOF)
    return c;
else
    return “0”;
}
```

```c
int main() {
int i;
FILE *file1;
char M[128];


file1=fopen("test.txt","rw");

 for (i=1; i < len; i++) {
     M[i] = get_character(file1,i);
 }
 return 1;
}
```

a. El código compilará correctamente pero siempre dará una excepción por Violación de segmento al ejecutarlo si el archivo `test.txt` ocupa menos de 1KB.
b. El código compilará correctamente. En ejecución, se corromperán datos de la pila pero la ejecución continuará.
c. El código compilará correctamente. En ejecución, se corromperán datos de la pila y, quizás, de otras regiones de memoria (como el heap, código ...), pero puede que no se produzca una excepción por violación de segmento.
d. Podemos asegurar que se producirá una excepción cuando la función `get_character()` acceda a una posición del fichero posterior a EOF.

SISTEMAS OPERATIVOS - PROBLEMAS
11 de septiembre de 2017

Nombre _____ DNI _____

Apellidos _____ Grupo _____

Problema 1. (2 puntos) Considere un sistema operativo tipo UNIX que emplea un sistema de ficheros basado en i-nodos para el almacenamiento de información en disco con bloques de 2KiB y direcciones de disco de 32 bits. Los i-nodos contienen 8 direcciones de disco para bloques de datos directos, una dirección de bloque índice indirecto simple y una dirección de bloque índice indirecto doble. En dicho sistema hay tres procesos en ejecución cuyas tablas de descriptores de ficheros abiertos (**TDDA**) se muestran a continuación, junto al contenido de las tablas únicas del sistema operativo **TFA** (tabla intermedia de posiciones, que almacena la visión lógica de todos los ficheros abiertos por los procesos de usuario) y **TIN** (tabla intermedia de inodos):

TDDA P1

FD	IDFF
0	23
1	63
2	56
3	4
4	678

TDDA P2

FD	IDFF
0	24
1	64
2	98
3	3

TDDA P3

FD	IDFF
0	24
1	64
2	98
3	3
4	326

TFA

Ind.	Puntero L/E	Número inodo	Perm.	# Refs
...	...	...	...	...
3	10	98	rw	2
4	0	98	r	1
...	...	...	...	...

TIN

inodo	# Refs (núm. entradas en TFA)
...	...
Info inodo 98	2
...	...

Suponiendo que hay una entrada en el directorio `/tmp` con nombre `file.txt` e inodo 98, conteste **razonadamente** a las siguientes preguntas:

a. Si para el **Puntero de L/E** de la **TFA** se utiliza un entero de 4B, ¿Cuál es el tamaño máximo de un fichero en este sistema?, ¿y el de la partición?
b. ¿Tienen los procesos **P1**, **P2** y **P3** alguna relación de parentesco a tenor del contenido de sus **TDDAs**? ¿Se puede saber a ciencia cierta si uno de ellos es el padre de otro?
c. ¿Cómo se modificarán las tablas si el proceso **P1** hace `close(3)`?
d. ¿Cómo quedarían las tablas si a continuación el proceso **P1** invocase `open("/tmp/file.txt", O_RDWR)`?

Problema 2. (2 puntos) En un convento de clausura residen N monjas que mantienen un riguroso protocolo a la hora de comer. Tras acceder al comedor, ninguna monja, por convenio, ha de empezar a comer hasta que el resto de monjas se hayan sentado a la mesa. Adicionalmente, al acabar de comer cada monja debe retomar sus quehaceres en estricto orden de llegada al comedor; es decir, aquella monja que llegó primero al comedor será la que comience antes con sus labores tras la comida. Las monjas dedicarán el resto del día a sus labores, por lo que no volverán a entrar al comedor hasta el día siguiente (esto es, en la codificación no hay que preocuparse de la posibilidad de que una monja trate de comer nuevamente antes de que todas los demás hayan terminado).

Cada monja, modelada como un hilo del programa concurrente, se comporta del siguiente modo:

```
void monja(){
    int turno_salida;
    while(1){
        <<labor_en_cuestion>>
        turno_salida=entrar_comedor_y_sentarse();
        <<comer>>
        salir_comedor(turno_salida);
    }
}
```

Implementar las funciones `entrar_comedor_y_sentarse()` y `salir_comedor()` para imponer las restricciones de sincronización anteriormente citadas empleando mútexes, variables condicionales o semáforos, y variables compartidas. La implementación debe permitir que todas las monjas coman simultáneamente.