Examen Final de Sistemas Operativos
20 de septiembre de 2007

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 **2 horas.**
*** El examen se contesta en las hojas dadas con el enunciado.**
*** No** se pueden utilizar libros **ni** apuntes, ni usar móvil (o similar)
* Será necesario presentar el DNI o carnet universitario para realizar la entrega del examen

---

Nombre: ……..
NIA: ……
Grupo: ……

Ejercicio 1 (2 puntos)

a. ¿Qué es un proceso Zombie?

b. ¿Cuál es la diferencia entre enlace físico y enlace simbólico?

c. ¿Cuál es la utilidad de utilizar un sistema con paginación en dos niveles?

d. ¿Qué decide el planificador cuando no existe ningún proceso en estado "listo"?

Solución:

a. Es un proceso hijo que ha finalizado su ejecución antes de que el proceso padre haga la llamada Wait. Así, no se pueden liberar todos los recursos del proceso hijo, ya que se debe almacenar el valor que se devuelve al padre (en el BCP).
b. En el caso del enlace físico dos nombres apuntan al mismo i-nodo. El archivo sólo se elimina cuando se han borrado todos los enlaces. Este tipo de enlace sólo permite enlazar archivos. En los enlaces simbólicos por su parte existen dos nombres y dos i-nodos que referencian al mismo fichero. El archivo se elimina cuando se borra el enlace físico.
c. De este modo, se pueden considerar varios segmentos de memoria (por ejemplo, datos, código y pila) que no tienen porque ser contiguos.
d. Se ejecuta el proceso nulo (es un proceso que no realiza ninguna tarea importante, aunque en algunos casos se utiliza como monitor del sistema).

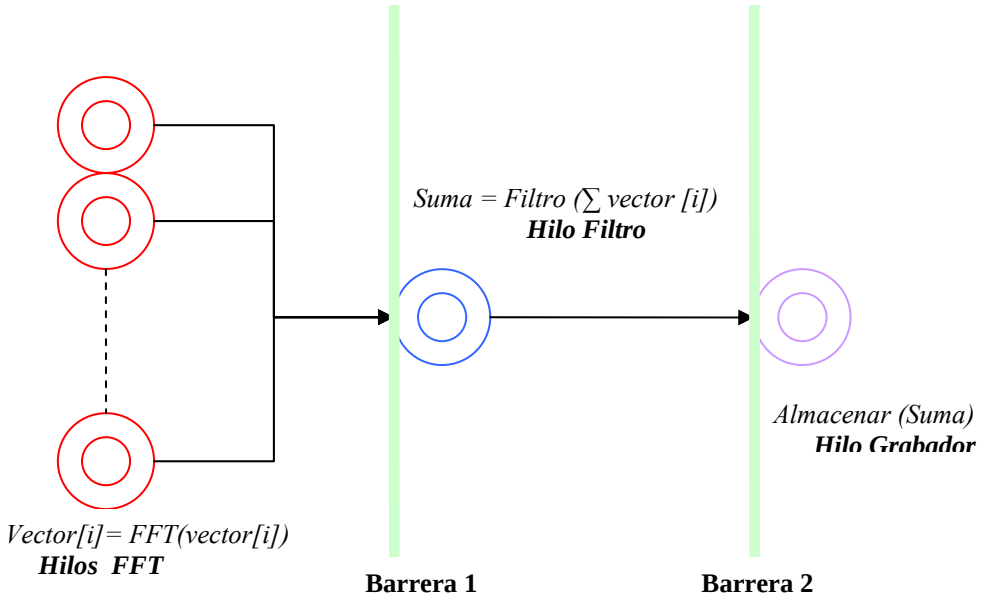
Ejercicio 2 (3 puntos)

Dada la siguiente fórmula usada para el proceso de señales se quiere, usando hilos, paralelizar a nivel de hilos el siguiente cálculo.

```
int vector[10];

resultado = Filtro(FFT(vector[1])+FFT(vector[2])+...+FFT(vector[10]))
```

El objetivo es desarrollar un programa usando hilos, mutex y variables condicionales donde existan 3 tipos de hilos y 2 barreras para cada parte del proceso tal y como se muestra en la figura:



Las Fases del programa se describen a continuación:
1.- Hilo ***FFT*** que se encarga de hacer la Transformada *int FFT(int)* de un elemento del vector. De este hilo deberá haber 10 instancias una por cada elemento del vector a procesar.
2.- Hilo ***Filtro*** que hace el filtro de la suma de todos los elementos de vector y le aplica la función *int Filtro(int)*.
3- Hilo G***rabador*** que almacena el resultado usando la función *void Almacenar(int)*

El hecho de tener barreras implica que todos los hilos que concurren sobre cada una de ellas deben haber finalizado para poder iniciar la ejecución del código del hilo siguiente.

Completar el código que a continuación se muestra usando las llamadas al sistema para el manejo de hilos que se crean necesarias:

NOTA: Asumir que las siguientes funciones *int FFT(int value), int Filtro(int value) , void Almacenar(int value)* están disponibles y pueden ser invocadas desde cualquier punto del código.

;

```
#define NUMERO_FFT 10

/* Variables compartidas entre fases del programa*/
int vector[NUMERO_FFT];
int FFT_terminadas;
int puedeFiltrar;
int puedeGrabar;
int suma;

/* Variables condicionales para la sincronizacion y comunicacion */
pthread_cond_t condFiltro;
pthread_cond_t condGrabador;

/* Mutex comun a todos los procesos */
pthread_mutex_t mutex;

void barrera1 (){
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
}
void barrera2 (){
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
}
void FFT(int * i){
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
            printf ("soy el hilo...%d \n",*i);
            vector[i] = FFT(vector[i])
            FFT_terminadas = FFT_terminadas + 1;
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
};
void Filtro (){
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

```
        //sumar vector(i) de FFT y aplicar filtro()
        printf ("Suma de los resultados parciales y ahora aplicando el Filtro a la suma....\n");
        int i;
        for (i=0; i<NUMERO_FFT;i++)  suma = suma + vector[i];
        Filtro (suma)
};
void Grabacion (){
        printf ("Estoy grabando el resultado...\n");
        Almacenar (suma)
};

int main(int argc, char * argv[]){
        pthread_t * arrayThreadFFT;
        pthread_t threadFiltro;
        pthread_t threadGrabacion;
        // Inicializamos variables compartidas. Estado inicial
        FFT_terminadas = 0;  // Ninguna FFT completada
        puedeFiltrar=0;              // Luego no se puede Filtrar
        puedeGrabar=0;               // Tampoco se puede Grabar el Resultado
        suma = 0;
        int i=0;

        /* Creación de un vector  dinámico de hilos  FFT */

        /* Creación de Procesos ligeros */

        /* Esperar a que finalice el hilo de Grabación */

        /* Liberar la memoria de las estructuras dinámicas */
        free (arrayThreadFFT);
```

```
        return 0;
    }
```

Solución:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <pthread.h>

#define NUMERO_FFT 10

int vector[NUMERO_FFT];
int FFT_terminadas;
int puedeFiltrar;
int puedeGrabar;
int suma;


/* Variables condicionales para la sincronizacion y comunicacion */
pthread_cond_t condFiltro;
pthread_cond_t condGrabador;

/* Mutex comun a todos los procesos */

pthread_mutex_t mutex;

void barrera1 (){
        pthread_mutex_lock(&mutex);
        if (FFT_terminadas == NUMERO_FFT){
                puedeFiltrar = 1;
                pthread_cond_signal (&condFiltro);
        };
        pthread_mutex_unlock(&mutex);
}

void barrera2 (){
        pthread_mutex_lock(&mutex);
        puedeGrabar=1;
        pthread_cond_signal (&condGrabador);
        pthread_mutex_unlock(&mutex);
}

void FFT(int * i){
        pthread_mutex_lock(&mutex);
        printf ("soy el hilo...%d \n",*i);
        //Se implementa el algoritmo que hace FFT(vector[i])
        FFT_terminadas = FFT_terminadas + 1;
        pthread_mutex_unlock(&mutex);
        barrera1();
        pthread_exit(0);
};


void Filtro (){
```

```
        pthread_mutex_lock(&mutex);
        while (!puedeFiltrar){
                printf ("Estoy bloqueao para filtrar\n");
                pthread_cond_wait(&condFiltro,&mutex);
                }
        //sumar vector(i) de FFT y aplicar filtro()
        printf ("Estoy sumando los resultados parciales y ahora aplicando el Filtro a la
suma....\n");
        int i;
        for (i=0; i<NUMERO_FFT;i++) suma = suma + vector[i];
        //Filtro (suma)
        pthread_mutex_unlock(&mutex);
        barrera2();
        pthread_exit(0);
};


void Grabacion (){
        pthread_mutex_lock(&mutex);
        while (!puedeGrabar){
                printf ("Estoy bloqueado para Grabar\n");
                pthread_cond_wait(&condGrabador,&mutex);
        }
        printf ("Estoy grabando el resultado...\n");
        //Almacenar Filtro(suma)
        pthread_mutex_unlock(&mutex);
        pthread_exit(0);
};

int main(int argc, char * argv[]){
        pthread_t * arrayThreadFFT;
        pthread_t threadFiltro;
        pthread_t threadGrabacion;
        // Inicializamos variables compartidas. Estado inicial
        FFT_terminadas = 0;  // Ninguna FFT completada

        puedeFiltrar=0;                 // Luego no se puede Filtrar
        puedeGrabar=0;                  // Tampoco se puede Grabar el Resultado
        suma = 0;
        int i=0;

        /* Array de FFT */
        arrayThreadFFT = (pthread_t *)malloc( sizeof(pthread_t) * NUMERO_FFT);

        /* Creacion de Procesos ligeros */
        pthread_create(&threadGrabacion,NULL,(void *)Grabacion,NULL);

        pthread_create(&threadFiltro,NULL,(void *)Filtro,NULL);
        i=0;
        while (i < NUMERO_FFT){
                pthread_create(&arrayThreadFFT[i],NULL,(void *)FFT,&i);
                i++;
        }
        /* Esperar a que finalize el hilo de Grabación */
        pthread_join(threadGrabacion,NULL);
```

```
        /* Liberar la memoria de las estructuras dinamicas */
        free (arrayThreadFFT);
}
```

Ejercicio 3 (2 puntos)

Atendiendo al siguiente código:

```
int sumando_1;
int_sumando_2=18;

main (int argc, char **argv) {

        int suma;

        exit (0)
}
```

1. ¿De qué está compuesto el bloque de activación de la función "main"?

2. En el momento en el que se lanza la ejecución del programa, ¿a qué segmento o segmentos de memoria van las variables "sumando_1" y "sumando_2"? Indica sus principales características.

3. ¿Qué es una variable de tipo "static"? ¿A qué segmento son asignadas las variables de este tipo?

Solución:

a. **Bloque de activación de la función main: parámetros de la función (argc, argv), variables locales (suma), dirección de retorno y variables de entorno.**

b. **Segmentos de datos sin valor inicial (sumando_1) y con valor inicial (sumando_2).**
   Datos con valor inicial: Privada, RW, T. Fijo, Soporte en Ejecutable.
   Datos sin valor inicial: Privada, RW, T. Fijo, Sin Soporte (rellenar 0)

c. **Son asignadas al segmento de datos, ya que se tratan del mismo modo que una variable global.**

Ejercicio 4 (3 puntos)

Se desea ejecutar un programa en un sistema POSIX (Linux en concreto) y cuyo comportamiento es el que sigue: el programa crea 100 procesos, espera por la terminación del último de los 100 y finaliza su ejecución.

Cada uno de los 100 procesos lanzados pondrá una alarma a un segundo, pedirán 1 MB de memoria usando la llamada al sistema *malloc*, y la rellenará cada byte de dicha región con el carácter 'a'. Cuando termine de rellenarse la memoria o cuando salte la alarma (lo primero que ocurra), el proceso finaliza, enviando un valor 100 al padre (a través de la llamada al sistema *exit*)

Se pide, razonando las respuestas y comentando adecuadamente el código:

a) Rellene la hoja dada para codificar el programa descrito en C, con llamadas al sistema POSIX (y las de biblioteca del estándar C). **(2 puntos)**

b) ¿Cuánta memoria será necesaria para ejecutar el programa descrito? **(0.25 puntos)**

c) Si se dispusiera de un sistema con 10 MB de RAM y sistema operativo con memoria virtual paginada con direccionamiento de 32 bits. ¿Se podría ejecutar este programa? **(0.25 puntos)**

d) Si no se rellena cada byte de la región pedida con *malloc*, ¿Podría ejecutarse todos los 100 procesos estando todos en memoria principal? **(0.25 puntos)**

e) Imagine el programa original propuesto donde **no** se rellena la región pedida con *malloc*. Imagine ahora el programa original propuesto donde en lugar de usar memoria dinámica se define una variable global que es un vector de 1 MB (*char vector[MB]*). ¿Qué podría cambiar con respecto a la ejecución del programa entre ambos casos? **(0.25 puntos)**

NOTA: utilice las llamadas *signal* y *alarm* para la parte del tratamiento de señales.

Solución

a)

```
/* inclusión de librerías */
#include <stdio.h>
#include <stdlib.h>



/* definición de constantes */
#define KB    (1024)
#define MB   (KB*KB)

/* función de tratamiento de la alarma */
```

```
void func_alarm ( int sig )
{
   exit(100) ;
}



/* función principal */
int main ( char *argv[], int argc )
{
   /* declaración de variables */
    int pid, ret ;
    int i, status ;



   /* código del programa */
   signal(SIGALARM, func_alarm) ;



   for (i=0; i<100; i++)
   {
      /* más código… */
      pid = fork() ;
      switch (pid)
      {
          case -1:   perror("fork") ;
                     exit(-1) ;
                     break ;

          default:  /* hijo */
                     alarm(1) ;
                     pch = malloc(MB) ;
                     memset(pch,'a',MB) ;
                     alarm(0) ; exit(100) ;
                     break ;
      }



   }

   /* esperar por el hijo */
   do
   {
        ret = wait(&status) ;
   } while (ret != pid) ;
```

```
   /* fin del programa */
    return (0) ;
}
```

b)

Cada proceso necesitará 1 MB de memoria para el segmento de datos y unos pocos KB para el código y pila. En total se necesitarán más de 100 MB de memoria.

c)

Si, puesto que el límite de la memoria virtual es 2^32 (y no los 10 MB de memoria física)

d)

Sí, puesto que el sistema operativo puede optar por no asignar marcos de página hasta que no se acceda a la memoria virtual, por lo que solo se utilizaría las páginas de código (compartidas con COW, por ejemplo) y las de pila

e)

Dado que es una variable global, estará en el segmento de datos estáticos y no en el segmento de datos dinámicos.
Cuando se crea un proceso hijo, en ese momento se reclama la memoria (y se asigna los marcos de páginas, puesto que dicha memoria se rellena con ceros).
Por esta razón, la creación de procesos necesitará más tiempo y espacio en memoria principal que el primer escenario comentado.