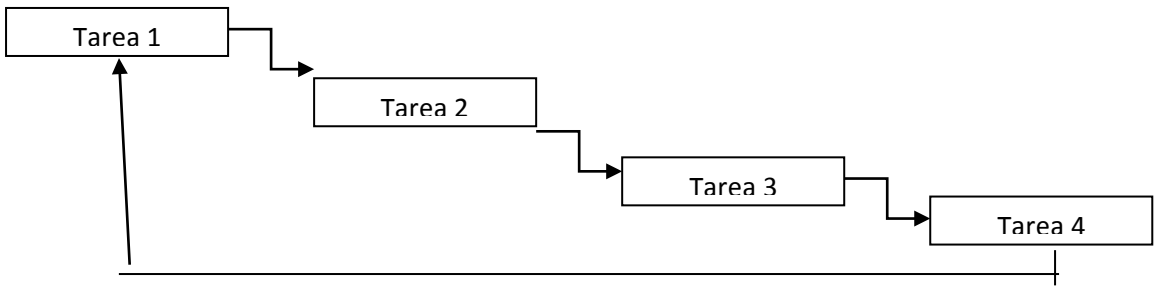
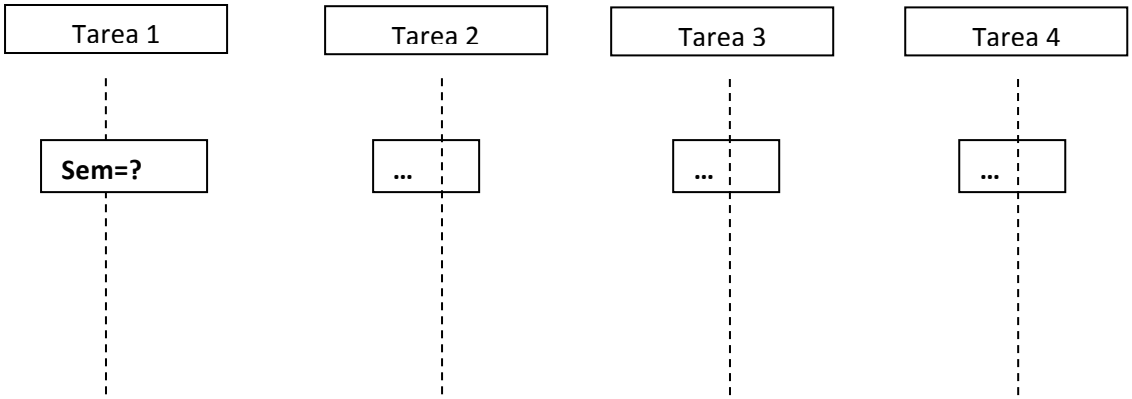
Ejercicio 1.

Se desea implementar una aplicación de *n* procesos concurrentes, donde cada proceso ejecuta una determinada tarea en forma de *pipeline.* Un ejemplo de esta aplicación con n=4 sería la siguiente:



Los procesos ejecutarán de manera ordenada, desde el primero hasta el último. La primera tarea debe comenzar la ejecución mientras que el resto de tareas se suspenden. Cada tarea calcula sus resultados y, antes de finalizar, notifica a la siguiente tarea que puede comenzar. Los resultados de cada tarea son privados, es decir, no necesitan compartirse con otras tareas. Se pide:

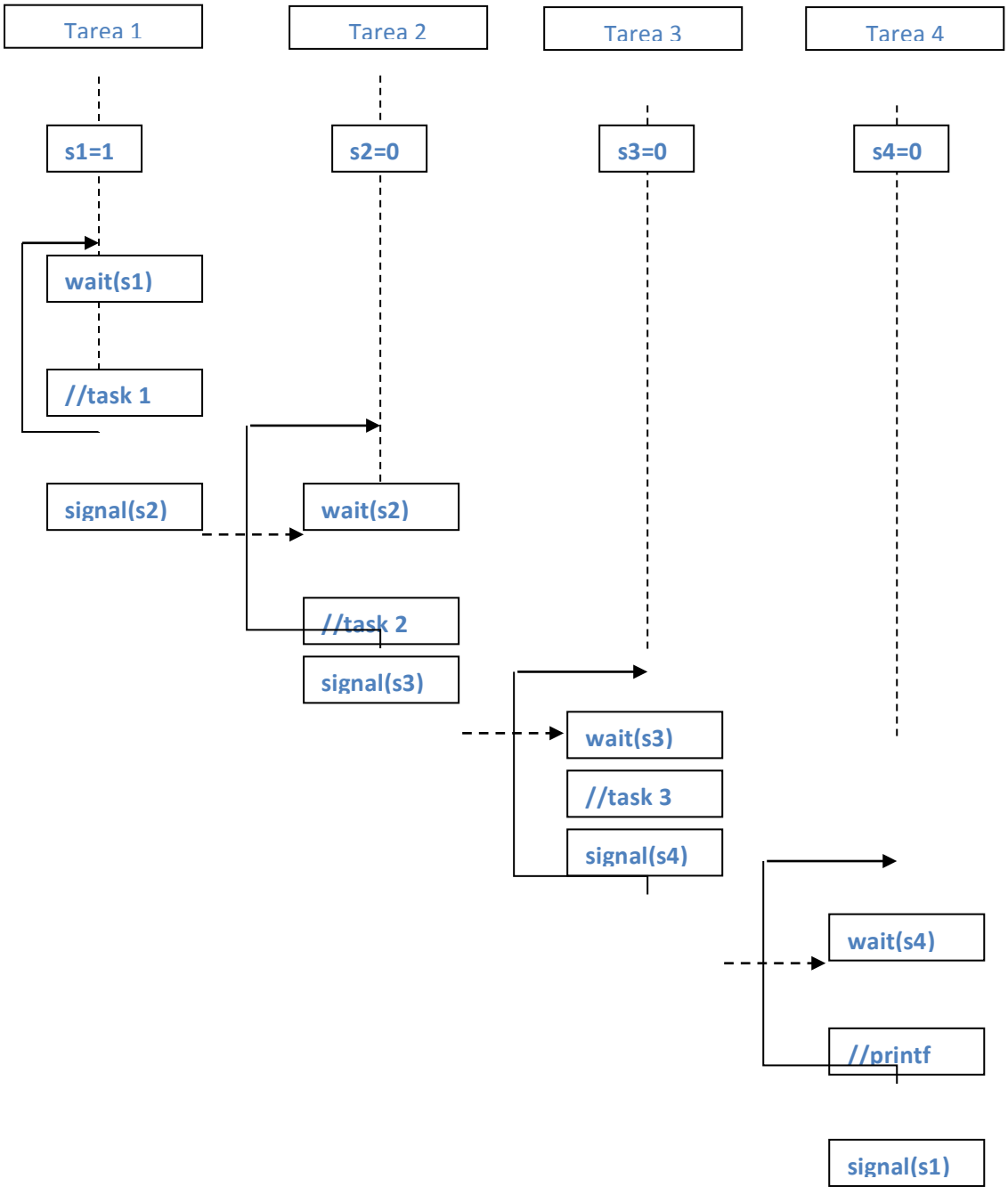
a) Diseñar un programa que permita la sincronización de los procesos para n=4 utilizando procesos ligeros y semáforos. Para ello, complete la siguiente plantilla indicando el valor inicial del semáforo y las primitivas de sincronización genéricas necesarias para llevar a cabo la sincronización (wait y signal).



b) Implementación de la aplicación en el lenguaje de programación C para n=4, de acuerdo al diseño anterior.

Solución:

a) Este problema admite varias soluciones. Una solución podría ser crear cuatro procesos ligeros y sincronizarlos mediante semáforos. Necesitaremos cuatro semáforos, uno para cada proceso, donde el valor de inicialización es 1 en el primer semáforo y 0 para el resto. Un posible diseño sería el siguiente.



b)

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>

#define         CONT    4

/* VARIABLES GLOBALES */
sem_t sem1;
sem_t sem2;
sem_t sem3;
sem_t sem4;

tarea1(){
        int i=0;
        while(i<CONT){
                sem_wait(&sem1);
                printf("Proceso 1...\n");
                sem_post(&sem2);
                i++;
        }
        pthread_exit(0);
}

tarea2(){
        int i=0;
        while(i<CONT){
                sem_wait(&sem2);
                printf("Proceso 2...\n");
                sem_post(&sem3);
                i++;
        }
        pthread_exit(0);
}
tarea3(){
```

```
        int i=0;
        while(i<CONT){
                sem_wait(&sem3);
                printf("Proceso 3...\n");
                sem_post(&sem4);
                i++;
        }
        pthread_exit(0);
}
tarea4(){
        int i=0;
        while(i<CONT){
                sem_wait(&sem4);
                printf("Proceso 4...\n");
                sem_post(&sem1);
                i++;
        }
        pthread_exit(0);
}
int main(){
        pthread_t th1, th2, th3, th4;

        sem_init(&sem1, 0, 1);
        sem_init(&sem2, 0, 0);
        sem_init(&sem3, 0, 0);
        sem_init(&sem4, 0, 0);

        pthread_create(&th1,NULL, (void*)tarea1, NULL);
        pthread_create(&th2,NULL, (void*)tarea2, NULL);
        pthread_create(&th3,NULL, (void*)tarea3, NULL);
        pthread_create(&th4,NULL, (void*)tarea4, NULL);

        pthread_join(th1, NULL);
        pthread_join(th2, NULL);
        pthread_join(th3, NULL);
        pthread_join(th4, NULL);

        sem_destroy(&sem1);
        sem_destroy(&sem2);
        sem_destroy(&sem3);
```

```
        sem_destroy(&sem4);

        exit(0);
}
```

Ejercicio 3.

Dado el sistema de ficheros de la figura, que tiene las siguientes características:

- Tamaño de bloque: 1024 bytes.
- Tamaño de dirección de bloque: 2 bytes.
- Número de sectores por bloque: 2
- Tiempo de lectura de un sector: 1 ms.
- Cada i-nodo ocupa un bloque.
- Campos de un i-nodo:
  - Identificador de i-nodo (ID)
  - Metadatos (atributos del fichero, identificador propietario y grupo, etc.)
  - Tipo de elemento: directorio (dir), fichero (fil) o enlace (lnk).
  - Contador de enlaces (CE)
  - 1 punteros directos (PD),
  - 1 puntero indirecto simple (PIS)
  - 1 puntero indirecto doble (PID).

La siguiente figura muestra una configuración del sistema de ficheros. Un valor en blanco significa que la entrada asociada está vacía (es tipo void/nill).

Bloque 0	Bloque 1	Bloque 2	Bloque 3	Bloque 4	Bloque 5	Bloque 6	Bloque 7	Bloque 8
Superbloque i-nodo raíz: 0	ID: 0 Metadatos Tipo: dir CE: 3 PD: 51 PIS:	ID: 1 Metadatos Tipo: fil CE: 1 PD: 100 PIS:	ID: 2 Metadatos Tipo: fil CE: 1 PD: 103 PIS: 52	ID: 3 Metadatos Tipo: dir CE: 4 PD: 53 PIS:	ID: 4 Metadatos Tipo: dir CE: 2 PD: 54 PIS:	ID: 5 Metadatos Tipo: lnk CE: 1 PD: 55 PIS:	ID: 6 Metadatos Tipo: dir CE: 2 PD: 56 PIS:	ID: 7 Metadatos Tipo: fil CE: 1 PD: 120 PIS: 121

	PID:	PID:	PID:	PID:	PID:	PID:	PID:	PID: 57

Bloque 51	Bloque 52	Bloque 53	Bloque 54	Bloque 55	Bloque 56	Bloque 57	Bloque 58	Bloque 59
. 0	104	. 3	. 4	/Murcia	. 6	130	131	
.. 0	105	.. 0	.. 3		.. 3	58	132	
Madrid 1	106	Norte 4	Ciudad 5				133	
Lugo 2		Sur 6						
Murcia 3		Centro 7						

Se pide:

1. Representar la estructura del árbol de ficheros/directorios. ¿Qué problema puede existir al recorrer la estructura anterior en la búsqueda de un fichero?
2. ¿Cuál es el tamaño máximo que puede tener un fichero?
3. Describa cómo se realizan la siguiente operación y qué cambios se efectuarían en el sistema de ficheros.
   rm /Murcia/Norte/Ciudad
4. Calcule el tiempo necesario para leer el primer byte del fichero /Madrid
5. Calcule el tiempo necesario para leer el último byte del fichero /Murcia/Centro

NOTA: Comando *rm*: borra un fichero.

SOLUCIÓN:

1. Representar la estructura del árbol de ficheros/directorios. ¿Qué problema puede existir al recorrer la estructura anterior en la búsqueda de un fichero? En caso en que hubiera un problema ¿cómo se podría resolver?

/Madrid (Fichero)

/Lugo (Fichero)

/Murcia/Norte/Ciudad (Enlace)

/Murcia/Sur (Directorio vacío)

/Murcia/Centro (Fichero)

El problema es una referencia circular a través del enlace:

/Murcia/Norte/Ciudad que apunta a /Murcia

Dos posibles soluciones es no utilizar enlaces simbólicos en las búsquedas o restringir el número de directorios de las mismas.

2. ¿Cuál es el tamaño máximo que puede tener un fichero?

Un puntero directo direcciona un bloque de 1 KB.

Dado que el tamaño de bloque es 1KB y el tamaño de dirección son 2B, cada punto indirecto simple direcciona un bloque que contiene 1KB/2B=512 punteros directos. Pudiendo direccionar 512 KB en total.

Un puntero indirecto doble direcciona un bloque que contiene 1KB/2B=512 punteros indirectos simples, cada uno de los cuales direcciona 512 KB. Pudiendo direccionar: 512 * 512KB = 256 MB

El tamaño máximo de fichero será: 1KB + 512KB + 256 MB.

3. Describa cómo se realizan cada una de siguiente operación y qué cambios se efectuarían en el sistema de ficheros.
rm /Murcia/Norte/Ciudad

Se accede al bloque 0 y se identifica el i-nodo raíz, se accede al i-nodo 0 (bloque 1) y su contenido (bloque 51). Se accede al i-nodo 3 (bloque 4) y a su contenido (bloque 53). Se accede al i-nodo 4 (bloque 5) y a su contenido (bloque 54). Se accede al i-nodo 5 (bloque 6) y se observa que el contador de enlaces es 1, por lo que puede ser borrado el enlace. Tanto el i-nodo 5 como el bloque 55 quedan libres. El resultado es:

Bloque 0	Bloque 1	Bloque 2	Bloque 3	Bloque 4	Bloque 5	Bloque 6	Bloque 7	Bloque 8
Superbloque i-nodo raíz: 0	ID: 0 Metadatos Tipo: dir CE: 3 PD: 51 PIS: PID:	ID: 1 Metadatos Tipo: fil CE: 1 PD: 100 PIS: PID:	ID: 2 Metadatos Tipo: fil CE: 1 PD: 103 PIS: 52 PID:	ID: 3 Metadatos Tipo: dir CE: 4 PD: 53 PIS: PID:	ID: 4 Metadatos Tipo: dir CE: 2 PD: 54 PIS: PID:	ID: 5 ~~Metadatos~~ ~~Tipo: lnk~~ ~~CE: 1~~ ~~PD: 55~~ ~~PIS:~~ ~~PID:~~	ID: 6 Metadatos Tipo: dir CE: 2 PD: 56 PIS: PID:	ID: 7 Metadatos Tipo: fil CE: 1 PD: 120 PIS: 121 PID: 57

Bloque 51	Bloque 52	Bloque 53	Bloque 54	Bloque 55	Bloque 56	Bloque 57	Bloque 58	Bloque 59
. 0 .. 0 Madrid 1 Lugo 2 Murcia 3	104 105 106	. 3 .. 0 Norte 4 Sur 6 Centro 7	. 4 .. 3 ~~Ciudad~~ 5	~~/Murcia~~	. 6 .. 3	130 58	131 132 133	

4. Calcule el tiempo necesario para leer el primer byte del fichero /Madrid
   El acceso al fichero requiere las siguientes operaciones:

   Se accede al bloque 0 y se identifica el i-nodo raíz, se accede al i-nodo 0 (bloque 1) y su contenido (bloque 51). Se accede al i-nodo 1 (bloque 2) y a su contenido (bloque 100). En total, se acceden a 5 bloques, lo que supone 10 sectores, que tardan 10 ms.

5. Calcule el tiempo necesario para leer el último byte del fichero /Murcia/Centro
   Se accede al bloque 0 y se identifica el i-nodo raíz, se accede al i-nodo 0 (bloque 1) y su contenido (bloque 51). Se accede al i-nodo 3 (bloque 4) y a su contenido (bloque 53). Se accede al i-nodo 7 (bloque 8). El último byte del fichero está asociado al puntero indirecto doble. Por lo que es necesario, acceder al bloque 57 y a través de la última

entrada del mismo al bloque 58. La última entrada del mismo apunta al último bloque de datos que es el 133.

Así es necesario acceder a: 9 bloques de datos que suponen 18 sectores con un tiempo asociado de 18 ms.

Ejercicio 4.

Se desea implementar una interfaz que mejore el acceso al sistema de ficheros de cara a cierto tipo de necesidades. Este sistema estará basado en las llamadas estándar POSIX e incluirá la siguiente interfaz de usuario:

- `int sustituir(char *fichero_in, char *fichero_out, char car_viejo, char car_nuevo)`
  - Recibe la ruta absoluta de un fichero a leer, la ruta absoluta de un fichero a escribir, un carácter a sustituir y el carácter por el que sustituir el parámetro anterior. Abrirá el fichero a leer y copiará los caracteres en el fichero de salida, reemplazando todas las ocurrencias de **car_viejo** por **car_nuevo** en el proceso.
  - Devuelve el número de sustituciones realizadas o -1 si se ha producido algún problema.

Se pide:

a) Describa el pseudocódigo de la función **sustituir**.
b) Implemente la función **sustituir** en lenguaje C en base a la interfaz proporcionada.
c) Implementar una función **main** que reciba como parámetros dos nombres de fichero y dos caracteres (viejo y nuevo) y controle que se han recibido exactamente estos cuatro parámetros. En caso de no recibirlos, mostrará un error y terminará el programa. En caso contrario, llamará a **sustituir** con los mismos e imprimirá el resultado.

SOLUCIÓN:

B)

```
1.  int sustituir(char *fichero_in, char *fichero_out, char car_viejo, char car_nuevo) {
2.    FILE *file_in, file_out;
3.    file_in = fopen(fichero_in, "r");
4.    file_out = fopen(fichero_out, "w");
5.    if(file_in == NULL || file_out == NULL) {
6.      printf("Error opening files\n");
7.      return -1;
8.    }
9.    int total = 0;
10.   char c;
```

```
11.  while((c = getc(file_in)) != EOF) {
12.    if(c == car_viejo) {
13.      total++;
14.      c = car_nuevo;
15.    }
16.  putc(c, file_out);
17.  }
18.  fclose(file_in);
19.  fclose(file_out);
20.  return total;
21. }
```

C)

```
1.  int main(int argc, char *argv[]) {
2.    if(argc != 5) {
3.      printf("Numero de argumentos incorrecto\n");
4.      return -1;
5.    }
6.    int total = sustituir(argv[1], argv[2], argv[3][0], argv[4][0]);
7.    printf("Numero de sustituciones: %d\n", total);
8.    return 0;
9.  }
```

d)

Ejercicio 5.

Se quiere implementar un programa que juegue al juego conocido como “los chinos”. El código inicial es una función *main* y una variable global *monedas_totales*. El programa deberá realizar lo siguiente:

- La función main creará 3 hilos pasándole a cada hilo un ID.
- Cada hilo generará una cantidad aleatoria de monedas entre 0 y 5.
- Cada hilo generará una estimación de monedas totales formado por sus monedas más una cantidad aleatoria entre 0 y 10.
- Sumará su cantidad de monedas a *monedas_totales.*
- Una vez TODOS los hilos hayan sumado sus monedas, cada hilo comprobara si SU estimación es igual al *monedas_totales*. Si lo es, imprimirá por pantalla su ID y la frase “Yo gano”.

Se pide:

a) Implemente el código necesario para que la función *main* cree 3 hilos y espere por ellos, pasando a cada hilo un identificador (int).
Implemente además la función de los hilos. Puede utilizar la función rand()
   - rand() % X; genera un número entre 0 y X - 1

b) Implemente los mecanismos de sincronización necesarios para acceder a la variable *monedas_totales* sin que se den condiciones de carrera.

c) Implemente el mecanismo de sincronización necesario para que ningún hilo compruebe si ha ganado ANTES de que todos los hilos hayan terminado de sumar su cantidad a *monedas_totales*.

SOLUCION

```
int main(){

        pthread_t th1[3];

        pthread_create (&th1[0], NULL, jugar, 1);

        pthread_create (&th1[1], NULL, jugar, 2);

        pthread_create (&th1[2], NULL, jugar, 3);

        pthread_join (th1[0], NULL);

        pthread_join (th1[1], NULL);

        pthread_join (th1[2], NULL);

}


int jugar(void* arg){

        int numHilo = (int)arg;

        int misMonedas = rand() % 6;

        int estimo = misMonedas + rand() % 11;

        pthread_mutex_lock(mutex);

        monedas_totales += misMonedas;

        pthread_mutex_unlock (mutex)

        // BARRERA PUEDE HACERSE CON barrier O CON variables condicionales
```

```
        pthread_mutex_lock(mutex2);
        n_acabados++;
        while(n_acabados != 3)
                pthread_cond_wait(condVar);
        pthread_cond_bcast(condVar);
        pthread_mutex_unlock (mutex2)

        if(estimo == monedas_totales){
                printf("Yo gano %d\n", numHilo);
        }
}
```