www.sapientia.uji.es | 30

UNIVERSITAT JAUME•I

100 Ejercicios resueltos de Sistemas Operativos

José Ribelles Miguel
José Martínez Sotoca
Pedro García Sevilla

100 Ejercicios resueltos de Sistemas Operativos

José Ribelles Miguel
José Martínez Sotoca
Pedro García Sevilla

Departament de llenguatges i sistemes informàtics

■ Codi d'assignatura IG11

Edita: Publicacions de la Universitat Jaume I. Servei de Comunicació i Publicacions
Campus del Riu Sec. Edifici Rectorat i Serveis Centrals. 12071 Castelló de la Plana
http://www.tenda.uji.es e-mail: publicacions@uji.es

Col·lecció Sapientia, 30
www.sapientia.uji.es

ISBN: 978-84-693-0148-7

Aquest text està subjecte a una llicència Reconeixement-NoComercial-CompartirIgual de Creative Commons, que permet copiar, distribuir i comunicar públicament l'obra sempre que especifique l'autor i el nom de la publicació i sense objectius comercials, i també permet crear obres derivades, sempre que siguen distribuïdes amb aquesta mateixa llicència.
http://creativecommons.org/licenses/by-nc-sa/2.5/es/deed.ca

Índice general

Prefacio 4

1. Procesos e Hilos 6
1.1. Procesos 6
1.2. Planificación 13
1.3. Hilos 18

2. Comunicación y Sincronización de Procesos 20
2.1. Tuberías 20
2.2. Semáforos 24

3. Gestión de Archivos y Directorios 33
3.1. Sistemas de Archivos 33
3.2. Archivos y Directorios 35

4. Gestión de Memoria 38
4.1. Paginación 38
4.2. Políticas de Reemplazo 39

5. Ejercicios Generales 41

6. Soluciones 58

Prefacio

Los Sistemas Operativos han constituido tradicionalmente una materia troncal en los planes de estudio de todas las titulaciones de Informática. Las asignaturas que desarrollan estos contenidos incluyen aspectos teóricos fundamentales como procesos e hilos, gestión de memoria, comunicación, sincronización y sistemas de archivos. Además, es frecuente incluir una parte práctica que permite que el alumno conozca no sólo los principios teóricos, sino también cómo se aplican en sistemas operativos reales.

El objetivo de este libro es proporcionar suficiente material práctico para apoyar la docencia, tanto presencial, desarrollada en clases de problemas o en laboratorio, como no presencial, proporcionando al estudiante un material de apoyo al estudio de un nivel y contenido adecuado a una asignatura real.

En concreto, las cuestiones, ejercicios y problemas que se recogen en este libro son el resultado de su recopilación a lo largo de cuatro cursos, desde el año 2004, del material utilizado en la asignatura de Sistemas Operativos de la Ingeniería Técnica en Informática de Gestión de la Universitat Jaume I de Castellón. Dicha asignatura se estructura en 3 créditos de teoría, 1,5 créditos de problemas y 1,5 créditos de laboratorio. No obstante, el material incluido es bastante genérico y puede ser empleado en cualquier asignatura básica de Sistemas Operativos.

El contenido de este libro se divide en 6 capítulos cuya descripción se indica a continuación:

1. Gestión de Procesos e Hilos: planificación de procesos, jerarquía de procesos y uso de las llamadas al sistema para la gestión de procesos e hilos.

2. Comunicación y Sincronización de Procesos: problemas clásicos de la sección crítica, productor-consumidor y lector-escritor; y llamadas al sistema para el manejo de semáforos y tuberías.

3. Gestión de Archivos y Directorios: sistemas de archivos tipo FAT y nodo-i, llamadas al sistema para la gestión de archivos y directorios.

4. Gestión de Memoria: memoria virtual, paginación y políticas de reemplazo.

5. Problemas generales: problemas cuya resolución incluya conceptos tratados en varios de los capítulos anteriores.

6. Soluciones: en este capítulo se encuentran las soluciones a los ejercicios planteados en todos los capítulos anteriores.

Se ha creado la página Web `http://ig11.uji.es` como apoyo a este material, para mantenerlo actualizado incluyendo más ejercicios, páginas de ayuda, fe de erratas, etc.

Por último, no queremos dejar de expresar nuestro agradecimiento a los profesores Gustavo Casañ, Isabel Gracia y Antonio Castellanos, todos ellos del Departamento de Lenguajes y Sistemas Informáticos de la Universitat Jaume I, que también han participado en la impartición de la asignatura durante otros cursos y, como no, en la elaboración de algunos de los ejercicios propuestos de este libro.

Marzo, 2010

Capítulo 1

Procesos e Hilos

1.1. Procesos

1. Observa el siguiente código y escribe la jerarquía de procesos resultante.

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

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

  int num;
  pid_t pid;

  for (num= 0; num< 3; num++) {
    pid= fork();
    printf ("Soy el proceso de PID %d y mi padre tiene %d de PID.\n",
            getpid(), getppid());
    if (pid!= 0)
      break;
    srandom(getpid());
    sleep (random() %3);
  }
  if (pid!= 0)
    printf ("Fin del proceso de PID %d.\n", wait (NULL));

  return 0;
}
```

Ahora compila y ejecuta el código para comprobarlo. Contesta a las siguientes preguntas:

- ¿Por qué aparecen mensajes repetidos?
- Presta atención al orden de terminación de los procesos,
  - ¿qué observas?
  - ¿por qué?

2. Observa el siguiente código y escribe la jerarquía de procesos resultante.

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

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

  int num;
  pid_t pid;

  srandom(getpid());
  for (num= 0; num< 3; num++) {
    pid= fork();
    printf ("Soy el proceso de PID %d y mi padre tiene %d de PID.\n",
             getpid(), getppid());
    if (pid== 0)
      break;
  }
  if (pid== 0)
    sleep(random()%5);
  else
    for (num= 0; num< 3; num++)
      printf ("Fin del proceso de PID %d.\n", wait (NULL));

  return 0;
}
```

Ahora compila y ejecuta el código para comprobarlo. Presta atención al orden de terminación de los procesos, ¿qué observas? ¿por qué?

3. Dibuja la estructura del árbol de procesos que obtendríamos al ejecutar el siguiente fragmento de código:

```
for (num= 0; num< 2; num++) {
  nuevo= fork();  /* 1 */
  if (nuevo== 0)
    break;
}
nuevo= fork();    /* 2 */
nuevo= fork();    /* 3 */
printf("Soy el proceso %d y mi padre es %d\n", getpid(), getppid());
```

4. Considerando el siguiente fragmento de código:

```
for (num= 1; num<= n; num++){
  nuevo= fork();
  if ((num== n) && (nuevo== 0))
    execlp ("ls", "ls", "-l", NULL);
}
```

   a) Dibuja la jerarquía de procesos generada cuando se ejecuta y n es 3.

   b) Indica en qué procesos se ha cambiado la imagen del proceso usando la función *execlp*.

5. Dibuja la jerarquía de procesos que resulta de la ejecución del siguiente código. Indica para cada nuevo proceso el valor de las variables i y j en el momento de su creación.

```
for (i= 0; i< 2; i++) {
  pid= getpid();
  for (j= 0; j< i+2; j++) {
    nuevo= fork();    /* 1 */
    if (nuevo!= 0) {
      nuevo= fork();  /* 2 */
      break;
    }
  }
  if (pid!= getpid())
    break;
}
```

6. Estudia el siguiente código y escribe la jerarquía de procesos resultante. Después, compila y ejecuta el código para comprobarlo (deberás añadir llamadas al sistema *getpid*, *getppid* y *wait* para conseguirlo).

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

#define L1 2
#define L2 3

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

  int cont1, cont2;
  pid_t pid;

  for (cont2= 0; cont2< L2; cont2++) {
    for (cont1= 0; cont1< L1; cont1++) {
      pid= fork();
      if (pid== 0)
        break;
    }
    if (pid!= 0)
      break;
  }
  return 0;
}
```

7. Dibuja la jerarquía de procesos que resulta de la ejecución del siguiente código. Introduce las llamadas al sistema *wait* para que una vez generado el árbol de procesos los hijos sean esperados por sus respectivos padres. Además, haz que se informe de los tiempos de ejecución de las aplicaciones *xload* y *kcalc* que se generen así como del tiempo total de ejecución. Para calcular el tiempo transcurrido, puedes utilizar la función *time()* de la librería estándar *time.h*. La llamada *time(NULL)* devuelve los segundos transcurridos desde las 00:00:00 del 1/1/1970 hasta el instante de la llamada.

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

  int i, j;
  pid_t pid, nuevo, nuevo1;
  time_t ini, fin;

  for (i= 0; i< 2; i++){
    pid= getpid();
    for (j= 0; j< i+2; j++){
      nuevo= fork();
      if(nuevo== 0){
        break;
        nuevo1= fork();
        if(nuevo1== 0)
          execlp ("xload", "xload", NULL);
      }
    }
    if (pid!= getpid())
      execlp ("kcalc", "kcalc", NULL);
  }
  return 0;
}
```

8. Escribe un programa que genere un árbol de procesos similar al que aparece en la figura 1.1. Los valores de profundidad y anchura del árbol serán dados como parámetros de entrada por el usuario en cada ejecución. En el ejemplo, se considera que la profundidad es 5 y la anchura 3. Tu programa podrá empezar, por ejemplo, de la siguiente manera:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

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

  int i;
  int prof, anch;

  if (argc!= 3) exit(0);

  profundidad= atoi(argv[1]); /* profundidad */
  anchura=     atoi(argv[2]); /* anchura     */

  /* completar aquí */

  printf("Soy el proceso %d y mi padre es %d\n", getpid(), getppid());
  sleep (2);
  return 0;
}
```

Modifica el programa anterior para que la expansión en anchura del árbol se produzca sólo en aquellos niveles de profundidad par (y distinta de cero). En la figura 1.2 se muestra un ejemplo.

9. Escribe el fragmento de código que genera la jerarquía de procesos que se muestra en la figura 1.3 para profundidad n.

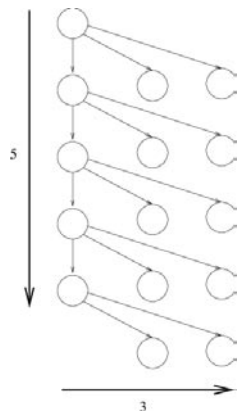

Figura 1.1: Árbol de profundidad 5 y anchura 3.

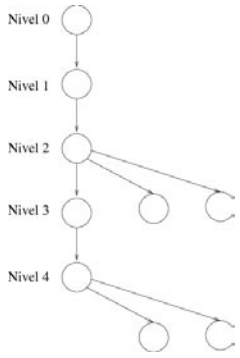

Figura 1.2: Árbol de profundidad 5 y anchura 3, que sólo se expande en anchura para los niveles pares distintos de cero.

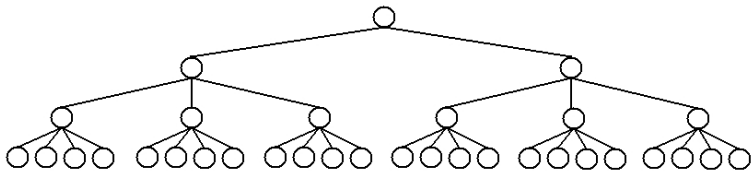

Figura 1.3: Jerarquía de procesos con profundidad $n = 3$.

10. Observa el siguiente fragmento de código que trata de medir el tiempo de ejecución del programa *prueba.exe*. Indica qué problemas se producen, por qué y cómo resolverlos.

```
time_t inicio= 0, fin= 0;

if (fork()!= 0) {
  wait (NULL);
  fin= time (NULL);
  printf ("Tiempo empleado: %ld\n", fin-inicio);
} else {
  inicio= time (NULL);
  execlp ("prueba.exe", "prueba.exe", NULL);
}
```

11. El programa siguiente pretende lanzar a ejecución una calculadora, *kcalc*, y otra aplicación, *xload*, utilizando dos llamadas al sistema *execlp* consecutivas. Antes de compilar y ejecutar el programa, piensa qué va a ocurrir.

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

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

  execlp ("kcalc", "kcalc", NULL);
  printf ("¿Se imprimira este mensaje?\n");
  execlp ("xload", "xload", NULL);
  printf ("¿Y este otro?\n");
  return 0;
}
```

Ahora, compílalo y ejecutálo y observa qué ocurre. ¿Has acertado? ¿Sabes por qué? Modifícalo para que el usuario vea las dos aplicaciones al mismo tiempo. Haz además que el proceso principal espere a la finalización de ambas aplicaciones e informe de la finalización de cada una especificando si terminó *kcalc* o *xload*.

12. Añade al programa resultado del problema 11 el cálculo del tiempo que cada uno de los procesos ha estado en ejecución, incluido el proceso padre, y sea el proceso padre el que informe de ellos antes de finalizar.

13. Escribe un programa en C que pida por teclado dos cadenas de caracteres y después escriba cada cadena por pantalla carácter a carácter. La escritura de cada cadena deberá hacerla un proceso diferente. Recuerda utilizar la función *fflush* después de escribir cada carácter. El proceso padre deberá esperar a que termine el proceso hijo. Obtendrás resultados interesantes si después de escribir cada carácter introduces un retardo aleatorio para simular que su escritura consume un cierto tiempo.

14. El siguiente programa en C lee repetidamente por teclado el nombre de un programa a ejecutar y pregunta si se debe esperar a que termine la ejecución del mismo. El programa termina de ejecutarse cuando el usuario introduce como programa a ejecutar *salir*.

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

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

  bool fin= false;
  char nombre_prog[50], esperar[5];
  while (!fin) {
    printf ("Programa: "); scanf ("%s", nombre_prog);
    if (strcmp(nombre_prog, "salir")!=0) {
      printf ("Esperar? "); scanf ("%s", esperar);

      /* completar aquí */

    }
    else
      fin= true;
  }

  return 0;
}
```

Realiza las modificaciones oportunas para que el programa cree un nuevo proceso que se encargue de ejecutar el programa indicado. Se debe utilizar la variable *PATH* para buscar el programa a ejecutar. Si dicho programa no se pudiese ejecutar por cualquier motivo, se deberá mostrar un mensaje de error e informar al usuario del valor actual de la variable *PATH*. En cualquier caso, el proceso inicial esperará o no la finalización del programa dado en función de lo que el usuario haya indicado. Cuando no se espere la finalización del programa dado, se debe indicar el identificador del proceso creado. Para comprobar el correcto funcionamiento de este programa se aconseja escribir otro programa que muestre por pantalla los números del 1 al 20 con un intervalo de 1 segundo entre cada número. Pide que sea éste el programa que se ejecute, unas veces esperando su finalización y otras no.

15. El siguiente programa recibe como parámetro de entrada un número entero y muestra como resultado su factorial.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

long long int factorial (int n) {

  long long int resultado= 1;
  int num;

  for (num= 2; num<= n; num++)  {
    resultado= resultado* num;
    printf ("Factorial de %d, resultado parcial %lld\n", n, resultado);
    sleep (random()\%3);
  }
  return resultado;
}

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

  if (argc== 2)
    printf ("El factorial de %s es %lld\n",
             argv[1], factorial (atoi (argv[1])));
  return 0;
}
```

   a) Escríbelo, compílalo y ejecútalo para comprobar su funcionamiento.

   b) Escribe un nuevo programa que reciba dos números enteros como parámetros de entrada y cree dos procesos de manera que cada uno calcule el factorial de uno de los números, de forma concurrente, y utilizando el fichero ejecutable obtenido en el apartado anterior.

   c) Haz que el proceso padre sea el último en terminar, es decir, que espere a la terminación de sus procesos hijos.

16. Generaliza la solución del problema 15 de manera que no esté limitado a 2 el número de factoriales a calcular. Procede de la siguiente manera:

a) Crea un proceso por cada factorial a calcular, y que todos los procesos se ejecuten de forma concurrente.

b) El proceso padre deberá esperar a todos los procesos hijos y mostrar un mensaje a medida que vayan terminando indicando el PID del proceso finalizado.

c) Modifícalo para que no se imprima mensaje cuando el primer proceso hijo finalice, pero si para los demás.

17. Escribe un programa, al que llamarás *tiempo.c*, cuyo objetivo es lanzar a ejecución un segundo programa que será indicado en la línea de órdenes (junto con sus argumentos) como por ejemplo: *$ tiempo ls -R -l /tmp*. Además, haz que se contabilice de forma aproximada el tiempo que tarda en ejecutarse el segundo programa.

1.2. Planificación

Para la realización de los siguientes ejercicios ten en cuenta que:

- La simulación comienza siempre con una interrupción de reloj.
- Cuando un proceso cambia su estado de bloqueado a listo, y si en el enunciado no se indica nada al respecto, el proceso se situará siempre al final de la cola de espera.

18. Considera un sistema con las siguientes características:

    - Se utiliza el algoritmo de planificación Round-Robin con un *quantum* de dos interrupciones de reloj.
    - La interrupción de reloj se produce cada 4 unidades de tiempo.
    - Las rutinas de tratamiento de las interrupciones hardware y de la interrupción de reloj consumen 1 unidad de tiempo. La rutina de tratamiento de la interrupción software consume 2 unidades de tiempo.
    - Existen dos dispositivos de entrada/salida sobre los que se pueden realizar operaciones en paralelo.
    - Los niveles de prioridad de las interrupciones son:
        - Nivel 0 (más prioritario): Interrupción hardware del dispositivo 1 (IH1)
        - Nivel 1: Interrupción de reloj (IR)
        - Nivel 2: Interrupción hardware del dispositivo 2 (IH2)
        - Nivel 3 (menos prioritario): Interrupción software (IS)

    Existen dos procesos A y B que están listos para ejecutar, de modo que inicialmente A será atendido antes que B. El modelo que siguen estos dos procesos es el siguiente:

Proc A	CPU (2ut.)	E/S D1 (2ut.)	CPU (3ut.)	E/S D2 (5ut.)	CPU (1ut)
Proc B	CPU (5ut.)	E/S D1 (1ut.)	CPU (1ut.)		

Indica qué proceso o interrupción está atendiendo la CPU en cada unidad de tiempo. Indica también aquellos instantes en los que los dispositivos de E/S están siendo utilizados.

19. Considera un sistema con las siguientes características:

    - Se utiliza el algoritmo de planificación Round-Robin con un *quantum* de dos interrupciones de reloj.
    - La interrupción de reloj se produce cada 5 unidades de tiempo.
    - Las rutinas de tratamiento de la interrupción hardware y de la interrupción de reloj consumen 1 unidad de tiempo. La rutina de tratamiento de la interrupción software consume 2 unidades de tiempo.
    - Existe un dispositivo de entrada/salida sobre el que se pueden realizar operaciones en paralelo con la CPU.
    - Los niveles de prioridad de las interrupciones son:
      - Nivel 0 (más prioritario): Interrupción hardware del dispositivo de E/S (IH)
      - Nivel 1: Interrupción de reloj (IR)
      - Nivel 2 (menos prioritario): Interrupción software (IS)

    Existen dos procesos A y B que están listos para ejecutar, de modo que inicialmente A será atendido antes que B. El modelo que siguen estos dos procesos es el siguiente:

Proc A	CPU (3ut.)	E/S (3ut.)	CPU (2ut.)	E/S (1ut.)	CPU (1ut.)
Proc B	CPU (8ut.)	E/S (1ut.)	CPU (1ut.)		

    Indica qué proceso o interrupción está atendiendo la CPU en cada unidad de tiempo. Indica también aquellos instantes en los que los dispositivos de E/S están siendo utilizados.

20. Considera un sistema con las siguientes características:

    - Se utiliza el algoritmo de planificación Round-Robin con un *quantum* de dos interrupciones de reloj.
    - La interrupción de reloj se produce cada 4 unidades de tiempo.
    - Las rutinas de tratamiento de las interrupciones hardware y de la interrupción de reloj consumen 1 unidad de tiempo. La rutina de tratamiento de la interrupción software consume 2 unidades de tiempo.
    - Existen un dispositivo de entrada/salida sobre el que se pueden realizar operaciones en paralelo con la CPU.

- Los niveles de prioridad de las interrupciones son:
  - Nivel 0 (más prioritario): Interrupción hardware del dispositivo de E/S (IH)
  - Nivel 1: Interrupción de reloj (IR)
  - Nivel 2 (menos prioritario): Interrupción software (IS)

Existen dos procesos A y B que están listos para ejecutar, de modo que inicialmente A será atendido antes que B. El modelo que siguen estos dos procesos es el siguiente:

Proc A	CPU (6ut.)	E/S (1ut.)	CPU (2ut.)
Proc B	CPU (2ut.)	E/S (6ut.)	CPU (3ut.)

Indica qué proceso o interrupción está atendiendo la CPU en cada unidad de tiempo. Indica también aquellos instantes en los que los dispositivos de E/S están siendo utilizados.

21. Considera un sistema con las siguientes características:

- Se utiliza el algoritmo de planificación Round-Robin con un *quantum* de dos interrupciones de reloj.
- La interrupción de reloj se produce cada 4 unidades de tiempo.
- Las rutinas de tratamiento de la interrupción hardware y de la interrupción de reloj consumen 1 unidad de tiempo. La rutina de tratamiento de la interrupción software consume 1 unidad de tiempo.
- Existe un dispositivo de entrada/salida sobre el que se pueden realizar operaciones en paralelo con la CPU.
- Los niveles de prioridad de las interrupciones son:
  - Nivel 0 (más prioritario): Interrupción hardware del dispositivo de E/S (IH)
  - Nivel 1: Interrupción de reloj (IR)
  - Nivel 2 (menos prioritario): Interrupción software (IS)

Existen tres procesos A, B y C que están listos para ejecutar, y que, inicialmente, serán atendidos en ese orden. El modelo que siguen estos tres procesos es el siguiente:

Proc A	CPU (3ut.)	E/S (10ut.)	CPU (4ut.)
Proc B	CPU (6ut.)		
Proc C	CPU (3ut.)	E/S (4ut.)	CPU (1ut.)

Indica qué proceso o interrupción está atendiendo la CPU en cada unidad de tiempo teniendo en cuenta que cuando un proceso pasa de bloqueado a listo se sitúa al principio de la cola de espera. Indica también aquellos instantes en los que los dispositivos de E/S están siendo utilizados.

22. Considera un sistema con las siguientes características:

    - Se utiliza el algoritmo de planificación Round-Robin con un *quantum* de dos interrupciones de reloj.
    - La interrupción de reloj se produce cada 4 unidades de tiempo.
    - Las rutinas de tratamiento de la interrupción hardware y de la interrupción de reloj consumen 1 unidad de tiempo. La rutina de tratamiento de la interrupción software consume 2 unidades de tiempo.
    - Existe un dispositivo de entrada/salida sobre el que se pueden realizar operaciones en paralelo con la CPU.
    - Los niveles de prioridad de las interrupciones son:
        - Nivel 0 (más prioritario): Interrupción hardware del dispositivo de E/S (IH)
        - Nivel 1: Interrupción de reloj (IR)
        - Nivel 2 (menos prioritario): Interrupción software (IS)

    Existen dos procesos A y B que están listos para ejecutar, de modo que inicialmente A será atendido antes que B. El modelo que siguen estos dos procesos es el siguiente:

Proc A	CPU (3ut.)	E/S (1ut.)	CPU (1ut.)	E/S (1ut.)	CPU (1ut.)
Proc B	CPU (4ut.)				

    Indica qué proceso o interrupción está atendiendo la CPU en cada unidad de tiempo. Indica también aquellos instantes en los que los dispositivos de E/S están siendo utilizados.

23. Considera un sistema con las siguientes características:

    - Se utiliza el algoritmo de planificación Round-Robin con un *quantum* de dos interrupciones de reloj.
    - La interrupción de reloj se produce cada 4 unidades de tiempo.
    - Las rutinas de tratamiento de la interrupción hardware y de la interrupción de reloj consumen 1 unidad de tiempo. La rutina de tratamiento de la interrupción software consume 1 unidad de tiempo.
    - Existe un dispositivo de entrada/salida sobre el que se pueden realizar operaciones en paralelo con la CPU.
    - Los niveles de prioridad de las interrupciones son:
        - Nivel 0 (más prioritario): Interrupción hardware del dispositivo de E/S (IH)
        - Nivel 1: Interrupción de reloj (IR)
        - Nivel 2 (menos prioritario): Interrupción software (IS)

Existen dos procesos A y B que están listos para ejecutar, de modo que inicialmente A será atendido antes que B. El modelo que siguen estos dos procesos es el siguiente:

Proc A	CPU (14ut.)				
Proc B	CPU (2ut.)	E/S (1ut.)	CPU (1ut.)	E/S (1ut.)	CPU (1ut.)

Indica qué proceso o interrupción está atendiendo la CPU en cada unidad de tiempo. Indica también aquellos instantes en los que los dispositivos de E/S están siendo utilizados.

24. Considera un sistema con las siguientes características:

    - Se utiliza el algoritmo de planificación Round-Robin con un *quantum* de dos interrupciones de reloj.
    - La interrupción de reloj se produce cada 4 unidades de tiempo.
    - Las rutinas de tratamiento de las interrupciones hardware y de la interrupción de reloj consumen 1 unidad de tiempo. La rutina de tratamiento de la interrupción software consume 2 unidades de tiempo.
    - Existen dos dispositivos de entrada/salida sobre los que se pueden realizar operaciones en paralelo.
    - Los niveles de prioridad de las interrupciones son:
        - Nivel 0 (más prioritario): Interrupción hardware del dispositivo 1 (IH1)
        - Nivel 1: Interrupción hardware del dispositivo 2 (IH2)
        - Nivel 2: Interrupción de reloj (IR)
        - Nivel 3 (menos prioritario): Interrupción software (IS)

    Existen tres procesos A, B y C que están listos para ejecutar en ese mismo orden. El modelo que siguen estos dos procesos es el siguiente:

Proc A	CPU (1ut.)	E/S D1 (8ut.)	CPU (1ut.)
Proc B	CPU (2ut.)	E/S D2 (2ut.)	CPU (7ut.)
Proc C	CPU (4ut.)		

    Indica qué proceso o interrupción está atendiendo la CPU en cada unidad de tiempo teniendo en cuenta además que cuando los procesos pasan de bloqueados a listos se sitúan al principio de la cola de espera. Indica también aquellos instantes en los que los dispositivos de E/S están siendo utilizados.

25. Considera un sistema con las siguientes características:

    - Se utiliza el algoritmo de planificación Round-Robin con un *quantum* de dos interrupciones de reloj.

- La interrupción de reloj se produce cada 5 unidades de tiempo.
- Las rutinas de tratamiento de la interrupción hardware y de la interrupción de reloj consumen 1 unidad de tiempo. La rutina de tratamiento de la interrupción software consume 1 unidad de tiempo.
- Existe un dispositivo de entrada/salida sobre el que se pueden realizar operaciones en paralelo con la CPU.
- Los niveles de prioridad de las interrupciones son:
  - Nivel 0 (más prioritario): Interrupción de reloj (IR)
  - Nivel 1: Interrupción hardware del dispositivo de E/S (IH)
  - Nivel 2 (menos prioritario): Interrupción software (IS)

Existen tres procesos A, B y C que están listos para ejecutar en ese mismo orden. El modelo que siguen estos tres procesos es el siguiente:

Proc A	CPU (15ut.)		
Proc B	CPU (2ut.)	E/S (1ut.)	CPU (3ut.)
Proc C	CPU (1ut.)	E/S (6ut.)	CPU (4ut.)

Indica qué proceso o interrupción está atendiendo la CPU en cada unidad de tiempo. Indica también aquellos instantes en los que los dispositivos de E/S están siendo utilizados.

1.3. Hilos

26. Modifica el programa que se ilustra en el enunciado del problema 15 de manera que reciba dos números enteros como parámetros de entrada y calcule sus factoriales de forma concurrente utilizando dos hilos que se ejecutan en paralelo con el hilo principal. El hilo principal deberá esperar a que terminen los otros dos hilos. Recuerda que para compilarlo se debe añadir *-lpthread* a la orden *gcc*.

27. Modifica el programa resultado del problema 26 de manera que no esté limitado a 2 el número de factoriales a calcular. Haz que se creen tantos hilos como parámetros de entrada y que todos se ejecuten de forma concurrente. El hilo principal debe esperar a que terminen el resto de hilos y, a medida que vayan terminando, muestre un mensaje que indique un identificador del hilo finalizado.

28. Modifica el programa solución del problema 13 para que la escritura de cada cadena la haga un hilo diferente que se ejecutan en paralelo con el hilo principal. El hilo principal debe esperar a que terminen los otros dos hilos.

29. El siguiente programa cuenta el número de veces que el carácter 'a' o 'A' aparece en el fichero indicado como parámetro de entrada. Modifícalo para que ahora se cree un hilo y sea éste el que ejecute la función cuenta.

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

#define MAXLON 1000

void cuenta (char *nombre) {

  int pos, cont= 0, leidos;
  char cadena[MAXLON];
  int fd;

  fd= open (nombre, O_RDONLY);
  while ((leidos= read (mf, cadena, MAXLON))!= 0)
    for (pos= 0; pos< leidos; pos++)
      if ((cadena[pos]== 'a') || (cadena[pos]== 'A'))
        cont++;
  printf ("Fichero %s: %d caracteres 'a' o 'A' encontrados\n", nombre, cont);
  close (fd);
}

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

  if (argc!= 2) {
    printf ("Indica el nombre de un fichero.\n");
    exit(0);
  }
  cuenta (argv[1]);
  return 0;
}
```

30. Modifica el programa resultado del problema 29 para que se creen tantos hilos como ficheros especificados como parámetros de entrada, y que todos los hilos creados se ejecuten de forma concurrente.

31. Modifica el programa resultado del problema 30 para que el resultado de la búsqueda lo informe el hilo principal cuando hayan terminado el resto de hilos. Haz uso del paso de parámetros a un hilo para que cada hilo pueda devolver el resultado de la búsqueda al hilo principal.

32. Modifica el programa solución del problema 30 para obtener el número de espacios en blanco que hay en cada uno de los ficheros dados como parámetros. Inicialmente, se deben crear tantos procesos como ficheros especificados. Cada proceso creará los hilos necesarios, que son los que realizarán la búsqueda, atendiendo a la restricción de que un hilo procesará como máximo K caracteres, donde K es una constante predefinida. Todos los hilos y procesos se han de ejecutar de forma concurrente. Cada proceso esperará a que terminen sus hilos y mostrará el total de espacios encontrados.

Capítulo 2

Comunicación y Sincronización de Procesos

2.1. Tuberías

33. Se desea informar del tiempo invertido en ejecutar las órdenes *ls | wc -l*. Para ello se escribe el siguiente programa. Sin embargo, no informa de forma correcta. Modifícalo para que lo haga sin cambiar el número de procesos que se están generando.

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

int main (int argc, char *argv[]){
  int tubo[2];
  time_t ini, fin;
  pipe(tubo);
  if(fork()==0){
    if(fork()==0){
      dup2(tubo[1],STDOUT_FILENO);
      close(tubo[0]);
      close(tubo[1]);
      execlp("ls","ls",NULL);
    }else{
      dup2(tubo[0],STDIN_FILENO);
      close(tubo[0]);
      close(tubo[1]);
      execlp("wc","wc","-l",NULL);
    }
  }
  else{
    printf("Tiempo invertido: %ld segundos\n", fin-ini);
  }
  return 0;
}
```

34. Observa el siguiente fragmento de código que trata de realizar lo mismo que si un usuario escribiera *ls | sort* en la línea de comandos de un sistema UNIX. Indica qué problemas se producen, por qué y cómo resolverlos.

```
int tubo[2];
pipe(tubo);
if (fork()!=0) {
  dup2(tubo[1], STDIN_FILENO);
  execlp("sort", "sort", NULL);
  close(tubo[0]);
  close(tubo[1]);
} else {
  dup2(tubo[0], STDOUT_FILENO);
  close(tubo[1]);
  close(tubo[0]);
  execlp("ls", "ls", NULL);
}
```

35. Al ejecutar el siguiente programa, el proceso no termina. Explica por qué. Da una solución que no cambie el número de procesos que se generan.

```
int main(int argc, char *argv[]) {
  int tubo[2];
  pipe(tubo);
  if (fork()==0) {
    if (fork()== 0) {
      dup2 (tubo[1], STDOUT_FILENO);
      close(tubo[0]);
      close(tubo[1]);
      execlp("ls", "ls", NULL);
    } else {
      dup2 (tubo[0], STDIN_FILENO);
      close(tubo[0]);
      close(tubo[1]);
      execlp("wc", "wc", "-l", NULL);
    }
  } else {
    wait(NULL);
    printf ("Fin del proceso\n");
  }
}
```

36. Describe todas las situaciones que se producen o podrían producirse al ejecutar el siguiente programa:

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

  int  tubo[2];
  FILE *fichero;
  char linea[MAX];

  pipe(tubo);

  if(!(fichero=fopen(argv[1],"r"))){
    printf("Error al abrir el fichero %s\n",argv[1]);
    exit(2);
  }
```

```
  while(fgets(linea,MAX,fichero))
    write(tubo[1],linea,strlen(linea));
  fclose(fichero);

  close (tubo[1]);
  dup2  (tubo[0],STDIN_FILENO);
  close (tubo[0]);
  execlp("sort","sort",NULL);

  exit(0);
}
```

37. Describe todas las situaciones que se producen o podrían producirse al ejecutar el siguiente programa:

```
int main(int argc, char *argv[]) {
  int tubo[2];
  pipe(tubo);
  if (fork()==0) {
    close(tubo[0]);
    dup2(tubo[1], STDOUT_FILENO);
    close(tubo[1]);
    execlp("ls", "ls", NULL);
  } else {
    dup2(tubo[0], STDIN_FILENO);
    close(tubo[0]);
    close(tubo[1]);
    wait(NULL);
    execlp("wc", "wc", "-l", NULL);
  }
  exit(0);
}
```

38. Escribe un programa que ejecute la siguiente línea de órdenes igual que lo haría un intérprete de comandos: *paste fich1 fich2 | sort | nl > fich3*. Debes considerar que *fich1*, *fich2* y *fich3* serán parámetros dados a tu programa en la línea de comandos.

39. Escribe un programa que genere tres procesos en paralelo que colaboran para realizar las siguientes tareas:

    - El primer proceso, utilizando la orden *grep*, encontrará las líneas de un fichero (fich1) que contienen una palabra (ambos dados como parámetros en la línea de comandos) y las escribirá en una tubería.
    - El segundo proceso, utilizando la orden *grep*, encontrará las líneas de un fichero (fich2) que contienen la misma palabra (ambos dados como parámetros en la línea de comandos) y las escribirá en la misma tubería.
    - El tercer proceso, utilizando la orden *wc*, leerá de la tubería las líneas producidas por los otros dos, las contará y escribirá el resultado en un nuevo fichero (fich3) pasado como parámetro en la línea de comandos.

    Así, el programa se utilizará en la línea de comandos de la siguiente forma:

$ programa palabra fich1 fich2 fich3. La figura 2.1 muestra gráficamente la comunicación requerida entre los procesos.

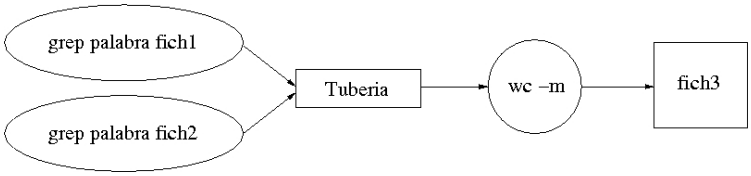

Figura 2.1: Esquema de funcionamiento.

40. Escribe un programa que genere tres procesos en paralelo que colaboran para realizar las siguientes tareas:

    - El primer proceso leerá líneas de un fichero de texto dado como parámetro y escribirá alternativamente en dos tuberías las líneas pares e impares del mismo.
    - El segundo proceso, utilizando la orden *grep*, leerá de la tubería que contiene las líneas pares y seleccionará aquellas líneas que contengan una palabra dada como parámetro en la línea de comandos. El resultado se almacenara en un fichero cuyo nombre estará formado por la palabra dada seguido de *.txt*.
    - El tercer proceso realiza una función similar sobre la tubería que contiene las líneas impares, pero utilizando otra palabra también dada como parámetro.

    La figura 2.2 muestra gráficamente los procesos y cómo se comunican estos cuando el programa se ejecuta con los siguientes parámetros: *$ programa fichero.txt uno dos*

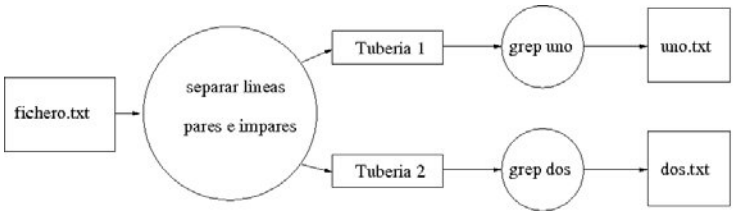

Figura 2.2: Esquema de funcionamiento.

41. Escribe un programa que genere los procesos necesarios para que colaboren en realizar las siguientes tareas:

    - Tarea 1: leer líneas de un fichero de texto dado como parámetro de entrada y escribir alternativamente en dos tuberías (tubería 1 y tubería 2) las líneas pares e impares del mismo.

- Tarea 2: utilizando el comando *grep*, leer de la tubería que contiene las líneas pares y seleccionar aquellas líneas que contengan una palabra dada como parámetro en la línea de comandos. El resultado se enviará a través de la tubería 3.
- Tarea 3: realizar una función similar a la tarea 2 pero sobre la tubería que contiene las líneas impares y utilizando otra palabra diferente también dada como parámetro de entrada.
- Tarea 4: ejecutar el comando *sort* sobre la información que se recoja por la tubería 3 de manera que se muestren de forma ordenada las líneas recogidas.

Observa la siguiente figura 2.3. En ella se representa de forma gráfica una propuesta de los procesos que se deben generar y de cómo se comunican estos cuando el programa se ejecute con los siguientes parámetros: *$ programa fichero.txt uno dos*. Antes de comenzar a escribir la solución, determina si estás de acuerdo o no con el esquema de funcionamiento propuesto. Si no lo estás explica porqué.

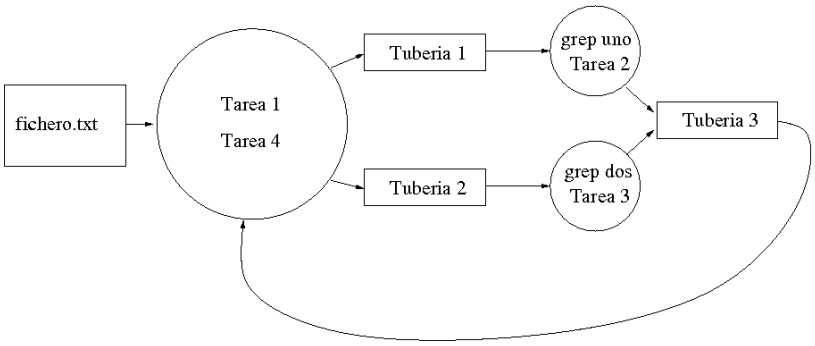

Figura 2.3: Esquema de funcionamiento.

2.2. Semáforos

42. Dados los siguientes procesos y sus respectivas secuencias de código, indica si existiría o no situación de interbloqueo y explica por qué. En cualquier caso, indica también la salida por pantalla y el valor final de los semáforos. Supón que inicialmente todos los semáforos tienen valor cero.

```
Proceso 1          Proceso 2          Proceso 3
---------          ---------          ---------
printf("3");       sem_wait(&s1);     sem_wait(&s2);
sem_post(&s3);     printf("1");       sem_wait(&s4);
printf("4");       sem_wait(&s3);     printf("2");
sem_post(&s2);     sem_post(&s4);     printf("5");
sem_post(&s1);     sem_wait(&s3);     sem_post(&s3);
```

43. Considera que los siguientes fragmentos de código se ejecutan en paralelo:

```
Código A:                  Código B:
---------                  ---------
  printf("A1");              printf("B1");
  sem_post(&s1);             sem_wait(&s1);
  sem_wait(&s2);             printf("B2");
  printf("A2");              sem_post(&s3);
  sem_wait(&s2);             sem_wait(&s3);
  sem_post(&s1);             printf("B3");
  printf("A3");              sem_post(&s2);
                             sem_wait(&s1);
                             sem_post(&s2);
                             printf("B4");
```

Sabiendo que todos los semáforos están inicializados a 0, indica todas las posibles salidas que puede proporcionar su ejecución y si se produce o no interbloqueo para cada una de ellas.

44. Modifica el programa resultado del problema 31 para que, utilizando una variable global a la cual acceden todos los hilos (llámala *cuenta_blancos*), estos acumulen el total de blancos encontrados. Utiliza un semáforo para asegurar que los accesos a dicha variable se realizan de forma adecuada. Haz que el programa principal informe también del resultado.

45. Escribe un programa que ejecute tres hilos en paralelo a los que llamaremos A, B y C. El hilo A consta de tres bloques de código (a1, a2 y a3), el hilo B de otros cuatro (b1, b2, b3 y b4) y el C de 3 (c1, c2 y c3). Haz que el código de cada uno de estos bloques consista en repetir cinco veces los siguientes pasos: escribir un mensaje que lo identifique y realizar un retardo aleatorio. Ejecuta el programa para comprobar que los hilos A, B y C se ejecutan en paralelo y que sus bloques de código pueden alternarse de varias formas distintas. Ahora, modifica el programa anterior para que los tres hilos se sincronicen de la siguiente forma:

    - c1 no debe comenzar hasta que acabe a1
    - a2 no debe comenzar hasta que acabe b1
    - c2 no debe comenzar hasta que acabe b2
    - b3 no debe comenzar hasta que acabe a2
    - a3 no debe comenzar hasta que acabe c2
    - b4 debe acabar el último

46. Se crean tres hilos de manera que uno ejecuta *escribirA*, otro *escribirB* y el tercero *escribirC*. Introduce los semáforos oportunos para que la salida sea ABCABCABCABCABCABC.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#include <unistd.h>
#include <pthread.h>

#define MAX 6

void *escribirA (void *nada){
  int num;
  for(num=0;num<MAX;num++){
    printf("A");
    fflush(NULL);
    sleep(random() %3);
  }
  pthread_exit(NULL);
}

void *escribirB (void *nada){
  int num;
  for(num=0;num<MAX;num++){
    printf("B");
    fflush(NULL);
    sleep(random() %2);
  }
  pthread_exit(NULL);
}

void *escribirC (void *nada){
  int num;
  for(num=0;num<MAX;num++){
    printf("C");
    fflush(NULL);
    sleep(random() %2);
  }
  pthread_exit(NULL);
}

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

  pthread_t th1, th2, th3;
  srandom(time(NULL));
  pthread_create(&th1, NULL, escribirA, NULL);
  pthread_create(&th2, NULL, escribirB, NULL);
  pthread_create(&th3, NULL, escribirC, NULL);
  pthread_join(th1, NULL);
  pthread_join(th2, NULL);
  pthread_join(th3, NULL);
  return 0;
}
```

47. Observa el siguiente fragmento de código donde los semáforos *sem1* y *sem2* están inicializados a cero, un hilo ejecuta la función incrementa y otro la función decrementa. Describe los valores que, durante la ejecución, puede adoptar la variable *num* así como las posibles situaciones de interbloqueo que pudieran darse.

```
int num=10;

void * incrementa(void *nada) {
  int i;
  for (i=0;i<3;i++){
    sem_wait(&sem1);
    num++;
    printf("Inc. Número = %d\n",num);
    sem_post(&sem1);
  }
  sem_post(&sem2);
  sleep(random() %3);
  sem_wait(&sem2);
  pthread_exit(NULL);
}
void * decrementa(void *nada){
  int i;
  for (i=0;i<3;i++){
    sem_post(&sem1);
    sleep(random() %3);
    sem_wait(&sem2);
    num--;
    printf("Dec. Número = %d\n",num);
    sem_post(&sem2);
    sem_wait(&sem1);
  }
  sem_wait(&sem1);
  pthread_exit(NULL);
}
```

48. Se crean dos hilos de manera que uno ejecuta *escribirA* y el otro *escribirB*. Introduce los semáforos oportunos para que la salida sea BABABABABA. No olvides indicar los valores iniciales de los semáforos que utilices.

```
void *escribirA (void *p) {
  int i;
  for (i= 0; i< 5; i++) {
    printf ("A");
    fflush(NULL);
    sleep(random() %2);
  }
  pthread_exit(NULL);
}

void *escribirB (void *p) {
  int i;
  for (i= 0;i< 5; i++) {
    printf ("B");
    fflush(NULL);
    sleep(random() %2);
  }
  pthread_exit(NULL);
}
```

49. Dado el siguiente código indica si existe o no interbloqueo. En el caso de existir, indica claramente para cada hilo en qué línea de código se queda bloqueado y en qué iteración del bucle ocurre (valores de las variables i, j, k). Observa los valores de inicialización de los semáforos.

```
sem_t s1, s2, s3;

void *escribirA (void *p){
  int i;
  srandom (pthread_self ());
  for (i= 0; i< MAX; i++){
    printf ("A");
    sem_post (&s2);
    sem_wait (&s1);
    fflush (NULL);
    sleep (random() %3);
  }
  pthread_exit (NULL);
}

void *escribirB (void *p){
  int j;
  srandom (pthread_self ());
  for (j= 0; j< MAX; j++){
    sem_wait (&s2);
    printf ("B");
    sem_post (&s3);
    sem_wait (&s2);
    fflush (NULL);
    sleep (random() %2);
  }
  pthread_exit (NULL);
}

void *escribirC (void *p){
  int k;
  srandom (pthread_self ());
  for (k= 0; k< MAX; k++){
    sem_wait (&s3);
    printf ("C");
    sem_post (&s1);
    sem_wait (&s3);
    fflush (NULL);
    sleep (random() %2);
  }
  pthread_exit (NULL);
}

int main (int argc, char *argv[]) {
  pthread_t th1, th2, th3;
  sem_init (&s1, 0, 1);
  sem_init (&s2, 0, 1);
  sem_init (&s3, 0, 0);
  pthread_create (&th1, NULL, escribirA, NULL);
  pthread_create (&th2, NULL, escribirB, NULL);
  pthread_create (&th3, NULL, escribirC, NULL);
  pthread_join (th1, NULL);
  pthread_join (th2, NULL);
  pthread_join (th3, NULL);
  return 0;
}
```

50. Considera el siguiente trozo de código del problema productor-consumidor:

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

```
#include <pthread.h>
#include <unistd.h>
#include <stdbool.h>

#define MAX 10
#define FIN -1

int buffer[MAX];
sem_t huecos, elementos;

int generar_dato (void) { return random() %256;}
int numero_aleatorio(void) { return random() %100;}

void *productor (void *p) {

  int pos_productor= 0;
  int num, dato, n;

  n= numero_aleatorio();
  printf ("Productor con %d datos\n", n);
  for(num= 0; num< n; num++) {
    dato= generar_dato();
    sem_wait (&huecos);
    buffer[pos_productor]= dato;
    pos_productor= (pos_productor+ 1) %MAX;
    sem_post (&elementos);
  }
  buffer[pos_productor]= FIN;
  pthread_exit (NULL);
}

void *consumidor(void *p){

  int pos_consumidor, dato;
  bool continuar= true;

  while (continuar) {
    sem_wait (&elementos);
    dato= buffer[pos_consumidor];
    pos_consumidor= (pos_consumidor+1) %MAX;
    if (dato== FIN)
      continuar= false;
    sem_post (&huecos);
    printf ("Numero aleatorio: %d\n", dato);
  }
  pthread_exit (NULL);
}
```

Este código contiene errores y está incompleto. Corrige y completa. Haz que el hilo consumidor muestre en pantalla todos los datos producidos por el hilo productor.

51. Realiza las modificaciones oportunas al código resultado del problema 50 para que en lugar de un hilo productor hayan tres que se ejecuten en paralelo. El consumidor terminará cuando haya consumido todos los datos producidos.

52. Escribe un programa que ejecute dos hilos en paralelo para realizar las siguientes tareas:

- El primer hilo calculará los números primos que hay entre dos números que el programa recibe como parámetros. Para enviar estos datos al segundo hilo, los almacenará en un buffer intermedio de 5 elementos de capacidad.
- El segundo hilo tomará los datos que aparezcan en el buffer intermedio y calculará la suma de los dígitos de cada número. Para cada número primo se mostrará en la salida estándar una línea que contendrá dicho número y la suma de sus dígitos.

Deberás utilizar semáforos para que los dos hilos se sincronicen en el acceso al buffer intermedio. Estas son las funciones para comprobar si un número es primo y para calcular la suma de los dígitos de un número.

```
/* Devuelve la suma de los dígitos del número dado */
int suma_digitos (int numero) {

  int suma= 0;

  while (numero> 0) {
    suma+= numero%10;
    numero/= 10;
  }
  return suma;
}

/* Indica si el número dado es primo */
bool es_primo (int numero) {

  int divisor;

  for (divisor= 2; divisor<= sqrt(numero); divisor++)
    if (numero%divisor== 0)
      return false;

  return true;
}
```

53. Escribe un programa que realice una simulación de la evolución del estado de las reservas en una aula de libre acceso. Para simplificar, supón que el aula tiene sólo un ordenador que se puede reservar en períodos de 1 hora, desde las 9:00 hasta las 21:00. 25 alumnos pueden reservar períodos individuales de 1 hora, cancelar reservas y consultar el estado de las reservas.

    La simulación debe consistir en que cada alumno realice cuatro operaciones, cada una de las cuales podrá ser una reserva, cancelación o consulta. La elección de la operación será aleatoria, pero con mayor probabilidad para la realización de reservas (50 %) que para la realización de consultas y cancelaciones (25 % cada una). Cuando la operación a realizar sea una reserva, debe elegirse aleatoriamente la hora que el alumno va a reservar.

    El programa debe implementar un hilo principal que lanza 25 hilos en paralelo, uno por cada alumno. Mediante el empleo de semáforos debe garantizarse

que los accesos a la tabla de reservas sean correctos. En concreto, cuando se realice una reserva o cancelación no puede realizarse ninguna otra operación en paralelo sobre la tabla, pero una consulta puede simultanear su acceso a la tabla con otras consultas.

En la implementación de la consulta de la tabla, muestra por pantalla el estado de cada período de una hora en una línea distinta. Para realizar una reserva, comprueba que el período que se solicita reservar esté libre. Al realizar una cancelación, deben quedar libres todas las horas que tuviese reservadas el estudiante que la solicita. Si ese estudiante no tuviese ninguna reserva se debe mostrar un mensaje de error.

Como todas las operaciones están controladas por semáforos, escribe mensajes que indiquen el estado de las operaciones diferenciando cuándo se solicita realizar una acción (aún no se tiene autorización), y cuándo se realiza efectivamente (ya se tiene autorización). Así, se puede seguir la evolución de las operaciones y los efectos que producen en el estado de la tabla de reservas. A continuación tienes un ejemplo del tipo de traza que se espera obtener.

```
Solicitud de reserva de al012: 9-10
Reserva de al012: 9-10
Solicitud de consulta de al004
Solicitud de cancelación de al006
Solicitud de reserva de al000: 15-16
Solicitud de consulta de al019
Consulta de al019: 9-10 <=> al012
Consulta de al019: 10-11 <=> LIBRE
Consulta de al019: 11-12 <=> LIBRE
Consulta de al019: 12-13 <=> LIBRE
Consulta de al019: 13-14 <=> LIBRE
Consulta de al019: 14-15 <=> LIBRE
Consulta de al019: 15-16 <=> LIBRE
Consulta de al019: 16-17 <=> LIBRE
Consulta de al019: 17-18 <=> LIBRE
Consulta de al004: 9-10 <=> al012
Consulta de al004: 10-11 <=> LIBRE
Consulta de al004: 11-12 <=> LIBRE
Consulta de al004: 12-13 <=> LIBRE
Consulta de al004: 13-14 <=> LIBRE
Consulta de al004: 14-15 <=> LIBRE
Consulta de al004: 15-16 <=> LIBRE
Consulta de al004: 16-17 <=> LIBRE
Consulta de al004: 17-18 <=> LIBRE
Consulta de al004: 18-19 <=> LIBRE
Consulta de al004: 19-20 <=> LIBRE
Consulta de al004: 20-21 <=> LIBRE
Consulta de al019: 18-19 <=> LIBRE
Consulta de al019: 19-20 <=> LIBRE
Consulta de al019: 20-21 <=> LIBRE
Denegada cancelación de al006: No tiene reservas
Reserva de al000: 15-16
```

```
Solicitud de reserva de al010: 9-10
Denegada reserva de al010: 9-10 está ocupada
Solicitud de cancelación de al012
Cancelación de al012: 9-10
...
```

54. Un puente es estrecho y sólo permite pasar vehículos en un único sentido al mismo tiempo. Si pasa un coche en un sentido y hay coches en el mismo sentido que quieren pasar, entonces estos tienen prioridad frente a los del otro sentido (si hubiera alguno esperando para entrar en el puente). No hay límite al número de vehículos que pueden haber en el puente al mismo tiempo.

    Simula el sistema suponiendo que los coches son hilos y el puente el recurso compartido. Utiliza semáforos para garantizar que se cumplen las condiciones de acceso al puente. Cada hilo debe mostrar por pantalla cuándo entra en el puente y cuándo lo abandona.Se generarán un total de 100 vehículos, 50 en un sentido y 50 en el otro. Tras un tiempo de espera al azar (utilizar *sleep(random() %20*) o algo similar) los vehículos intentan entrar en el puente y, si lo consiguen, permanecerán en él durante un segundo (*sleep(1)*) antes de abandonarlo. Se apreciará más el comportamiento del sistema si se alterna la creación de hilos en un sentido u otro.

Capítulo 3

Gestión de Archivos y Directorios

3.1. Sistemas de Archivos

55. Disponemos de un disco duro de 20 GB de capacidad. Hay establecida sobre él una única partición que contiene un sistema de ficheros del tipo FAT32 en el que cada agrupamiento (cluster) consta de 16 sectores de 512 bytes cada uno. ¿Cuántos sectores del disco se necesitarán para almacenar cada copia de la FAT? Razona tu respuesta.

56. La policía ha arrestado al sospechoso de un delito. Al analizar el contenido de su ordenador piensan que pueden inculparle pues el contenido del mismo es el siguiente:

Núm de bloque de datos	Contenido
10	he
11	sido
12	yo
13	no
14	sigan
15	buscando

Como experto informático, pides consultar el contenido de la FAT, que es el siguiente:

Núm de entrada en la FAT	Contenido
10	11
11	EOF
12	13
13	10
14	15
15	12

¿Apoyarías la opinión de la policía? Razona tu respuesta.

57. Tenemos un sistema de ficheros tipo FAT sobre el que hay almacenado un fichero de 160 Kbytes. Sabemos que para dicho fichero se emplean 10 entradas de la FAT y que cada sector del disco contiene 512 bytes. ¿Cuántos sectores como mínimo forman cada bloque o agrupamiento en dicho sistema? Razona tu respuesta.

58. Se dispone de una partición de disco con sistema de ficheros basado en FAT16. Si el tamaño de bloque es de 1KB, ¿cuántos KB de dicha partición podrán direccionarse como máximo? Si la partición resulta tener un tamaño de 2GB, ¿qué tamaño debería como mínimo tener el bloque para poder direccionar la partición por completo?

59. Se dispone de una partición de disco con sistema de ficheros basado en FAT16. A la hora de ponerle formato el usuario especifica que los bloques sean de tamaño 4Kbytes ¿Cuántos Kbytes teóricamente podrán direccionarse como máximo? Si la partición resulta tener un tamaño de 8Gbytes, ¿consideras adecuado el tamaño de bloque escogido por el usuario? Justifica la respuesta. En caso de que no estés de acuerdo propón un tamaño de bloque e indica en cuántos de esos bloques se almacena la FAT.

60. Para una partición de 8GB y tamaño de bloque de 1 KB,

    - Si se utiliza un sistema de ficheros basado en FAT16, ¿qué cantidad de espacio en disco queda inutilizable?
    - Si se utiliza un sistema de ficheros basado en nodos-i, donde cada nodo-i consta de dos índices directos, dos indirectos simples y dos indirectos dobles, y para referenciar un bloque se utilizan 128 bits, ¿qué cantidad de datos de un fichero que en concreto ocupa 131 KB puede ser irrecuperable en el caso de que un bloque de la partición resultara ilegible? Analiza todos los casos posibles.

61. Considera un sistema de ficheros basado en nodos-i, en el que cada nodo-i contiene cinco índices directos, tres indirectos simples, dos indirectos dobles y uno indirecto triple. Si el tamaño de un bloque de datos es de 2 Kbytes y para referenciar a un bloque se utilizan 64 bits, ¿cuántos bloques de disco almacenarán enlaces para un fichero que contiene 1548 Kbytes de datos? Razona tu respuesta.

62. Sea una partición de disco donde el tamaño de bloque es de 4KB. Se utiliza un sistema de ficheros basado en nodos-i, donde cada nodo-i consta de dos índices directos, dos indirectos simples y uno indirecto doble. Si para referenciar a un bloque se utilizan 32 bits, ¿cuál es el número de bloques que contendrán enlaces si el fichero ocupa el máximo tamaño posible?

3.2. Archivos y Directorios

63. La siguiente función muestra el nombre de todas las entradas del directorio que se le pasa como parámetro:

```
void listado(char nomdir[]) {
  DIR    *d;
  struct dirent *entrada;
  char   *ruta;

  d= opendir(nomdir);
  if (d== NULL)
    printf("Error al abrir el directorio\n");
  else {
    entrada= readdir(d);
    while (entrada!= NULL) {
      ruta= malloc(strlen(nomdir)+strlen(entrada->d_name)+2);
      sprintf(ruta,"%s/%s", nomdir, entrada->d_name);
      printf("%s\n", ruta);
      free(ruta);
      entrada= readdir(d);
    }
    closedir(d);
  }
}
```

Modifícala para que únicamente muestre aquellas entradas que se correspondan con enlaces simbólicos a directorios.

64. La siguiente función muestra el nombre de todas las entradas del directorio que se le pasa como parámetro:

```
void listado(char nomdir[]) {
  DIR    *d;
  struct dirent *entrada;
  char   *ruta;

  d= opendir(nomdir);
  if (d== NULL)
    printf("Error al abrir el directorio\n");
  else {
    entrada= readdir(d);
    while (entrada!= NULL) {
      ruta= malloc(strlen(nomdir)+strlen(entrada->d_name)+2);
      sprintf(ruta,"%s/%s", nomdir, entrada->d_name);
      printf("%s\n", ruta);
      free(ruta);
      entrada= readdir(d);
    }
    closedir(d);
  }
}
```

Modifícala para que por cada fichero regular que haya en el directorio se cree un archivo *zip* en el directorio */tmp* y se muestre la diferencia de espacio entre el archivo original y el nuevo archivo *zip*. Crea el archivo *zip* con la orden *zip ruta.zip ruta*.

65. Escribe un programa que dado un directorio como argumento de entrada elimine los ficheros, enlaces y directorios (sólo los vacíos) que hubiesen en él. El programa debe informar de si el directorio dado como parámetro existe así como de cada uno de los elementos eliminados. No hagas un recorrido recursivo del árbol de directorios.

66. Escribe un programa que borre una serie de nombres de ficheros y/o directorios recibidos como parámetros. Además, el programa debe mostrar para cada parámetro dado:

    - si se trata de un fichero regular, un enlace simbólico, un directorio, o bien si el nombre dado no es válido
    - si dicho nombre pudo borrarse correctamente, así como el número de bloques de disco que serán liberados en tal caso. Ten en cuenta que, en el caso de ficheros regulares, los bloques de disco sólo serán liberados si se trata del último enlace físico sobre el fichero.

67. Escribe un programa que, a partir de un directorio dado como parámetro, informe de los ficheros regulares que encuentre a partir de dicho directorio (deberá recorrer el árbol de directorios a partir de dicho directorio) tales que pertenezcan al usuario que ejecuta el programa, y se hayan accedido desde una hora antes del comienzo de la ejecución del programa. Durante el recorrido recursivo del directorio dado, se deben ignorar aquellos directorios que no puedan ser abiertos por no tener los permisos necesarios.

68. Escribe un programa que calcule la suma de los bytes ocupados por todos los ficheros y directorios que estén contenidos a partir de un directorio dado como parámetro. ¿Qué ocurre cuando hay dos enlaces duros que hacen referencia al mismo fichero? Haz que en estos casos el espacio ocupado se considere sólo una vez. Ten en cuenta que la estructura *stat* contiene el número de nodo-i asignado al fichero.

69. Escribe un programa que reciba como argumentos un fichero regular y un directorio, y cambie por enlaces simbólicos al fichero regular todos los enlaces duros referidos a él que encuentre a partir del directorio dado. Para simplificar, considera que tanto el fichero como el directorio se pasan al programa como caminos absolutos que no contienen el nombre ., ni el nombre .., ni ningún enlace simbólico.

70. Un reproductor multimedia recorre de forma recursiva el directorio */media* (y sus subdirectorios) a la búsqueda de ficheros con extensión *jpeg*, *avi* y *mp3*. En la ruta */resultado* hay tres carpetas de nombres *jpeg*, *avi* y *mp3*. Por cada fichero regular que encuentra con la extensión adecuada en el proceso de búsqueda, crea un enlace simbólico en la carpeta correspondiente a la extensión del archivo. Por ejemplo, si encuentra un fichero con extensión *avi*

crea el enlace simbólico a dicho fichero en el directorio */resultado/avi*. Escribe un programa en C que funcione de acuerdo al enunciado del problema. Además, haz que no haya que crear el enlace al archivo si éste ya existe.

Capítulo 4

Gestión de Memoria

4.1. Paginación

71. Considera un sistema de paginación en el que se puede direccionar como máximo 1 Gbyte de memoria, el tamaño de página es de 16 Kbytes y cada byte se direcciona independientemente ¿Cuántas páginas podrá tener asignadas como máximo un proceso en este sistema? Si empleamos una tabla de páginas con dos niveles, en el que la tabla de primer nivel contiene 1024 entradas, ¿cuántas tablas de segundo nivel son necesarias para un proceso que requiere 6401 páginas? Razona tu respuesta.

72. Considera un sistema de paginación en el que las direcciones lógicas son de 22 bits y el tamaño de página es de 2 Kbytes. Sabiendo que cada byte se direcciona independientemente, calcula el ahorro de memoria que obtendríamos para representar la tabla de páginas de un proceso que está utilizando 90 Kbytes de memoria, cuando empleamos una tabla de páginas con dos niveles en lugar de tener una tabla de un solo nivel. En el sistema con dos niveles, debes considerar que se emplean 5 bits de la dirección para el segundo nivel. Además, cada entrada de las tablas de páginas precisa 8 bytes. Razona la respuesta.

73. Considera un sistema de paginación en el que las direcciones lógicas son de 20 bits y el tamaño de página de 4 Kbytes. Sabiendo que cada byte se direcciona independientemente, calcula el ahorro de memoria que obtendríamos para representar la tabla de páginas de un proceso que está utilizando 192 Kbytes de memoria, cuando empleamos un tabla de páginas con dos niveles en lugar de tener una tabla de un solo nivel. En el sistema con dos niveles debes considerar que se emplea el mismo número de bits de la dirección para cada nivel. Cada entrada de las tablas de páginas precisa 16 bytes. Razona tu respuesta.

74. Considera un sistema de paginación en el que las direcciones lógicas son de 22 bits y el tamaño de página de 2 Kbytes, y que cada byte se direcciona in-

dependientemente ¿Cuántas páginas podrá tener asignadas como máximo un proceso en este sistema? Si empleamos una tabla de páginas de dos niveles, en la que la tabla de primer nivel contiene únicamente 8 entradas, ¿cuántas tablas de segundo nivel son necesarias para un proceso que requiere 1000 páginas? Razona la respuesta.

75. Lee las siguientes afirmaciones y razona si estás de acuerdo o no. Pon un ejemplo que apoye tu respuesta.

    - En un sistema de paginación, utilizar una tabla de páginas de dos niveles suele producir un ahorro en el consumo de memoria en comparación con el uso de una tabla de páginas de un nivel.
    - En un sistema de paginación, el número de páginas que como máximo se le puede asignar a un proceso es mayor en el caso de utilizar una tabla de páginas de dos niveles que en el caso de utilizar una única tabla de páginas.

76. Considera un sistema de paginación en donde se puede puede direccionar un máximo de 1GB y el tamaño de página es de 32KB. Sabiendo que cada palabra de 16 bits se direcciona independientemente, calcula el ahorro de memoria que obtendriamos de representar la tabla de páginas de un proceso que está utilizando 90MB de memoria, cuando empleamos una tabla de páginas de dos niveles en lugar de tener una tabla de un solo nivel. En el sistema de dos niveles, debes considerar que se empleará el mismo número de bits para cada nivel. Además, cada entrada en la tabla de páginas precisa de 16 bytes. Razona la respuesta.

77. Se considera un sistema de paginación en donde se puede direccionar un máximo de 1GB y el tamaño de página es de 32KB. Sabiendo que el tamaño de la palabra es de 64 bits y que cada palabra se direcciona independientemente, calcula el ahorro de memoria que obtendríamos de representar la tabla de páginas de un proceso que está utilizando 256 MB de memoria cuando se emplea una tabla de páginas de dos niveles en lugar de tener una tabla de un solo nivel. En el de dos niveles, debes considerar que se empleará el mismo número de bits para cada nivel. Además cada entrada de páginas precisa de 8 bytes. Razona la respuesta.

4.2. Políticas de Reemplazo

Para realizar los siguientes ejercicios ten en cuenta que:

- Inicialmente los marcos están libres.
- La política de reemplazo sólo se utiliza a partir del momento en que no hayan marcos libres.

78. Considera un sistema de paginación bajo demanda en el que un proceso que tiene asignados 3 marcos de página genera la siguiente secuencia de referencias a páginas:

```
2,3,1,2,4,5,2,3,1,5,6,1
```

Indica qué accesos producirían un fallo de página cuando se utilizan las políticas de reemplazo local FIFO y LRU. Sabemos que este proceso se va a ejecutar muy a menudo en el sistema y nos interesa tener el mejor sistema de paginación para él. ¿Valdría la pena aumentar el número de marcos de página asignados al proceso hasta 4 para alguna de estas dos políticas? Indica el número de fallos de página que se producirían en esta nueva situación para cada algoritmo.

79. Considera un sistema de paginación bajo demanda en el que un proceso que tiene asignados 4 marcos de página genera la siguiente secuencia de referencias a páginas:

```
4,2,4,1,6,3,2,5,6,4,1,3,5,3
```

Indica qué accesos producirían un fallo de página cuando se utiliza cada una de las políticas de reemplazo local FIFO, LRU y óptima.

80. Se ha de diseñar un sistema de paginación bajo demanda en el que se utiliza una política de reemplazo local con tres marcos de página asignados para cada proceso. Para la siguiente secuencia de referencias a páginas: 1, 2, 3, 2, 1, 5, 6, 3, 2, 1 ¿qué política de reemplazo produciría un resultado más cercano a la óptima, FIFO o LRU? Demuéstralo indicando los accesos que producirían fallo de página para cada uno de los métodos.

Capítulo 5

Ejercicios Generales

81. Escribe un programa para limpiar el directorio /tmp. Recorriendo todos los subdirectorios que se encuentren a partir de /tmp, el programa debe eliminar todos los ficheros (de cualquier tipo) que pertenezcan al usuario que ejecuta el programa para los que haga más de 2 semanas que no se acceden. En el caso concreto de los directorios, se deben eliminar los que ya estaban vacíos o queden vacíos al eliminar los ficheros que contenían, independientemente del tiempo transcurrido desde su último acceso. Además, debe identificar los ficheros para los que haga más de 1 semana pero menos de 2 que no se acceden e informar al usuario de que serán borrados próximamente.

    El programa recibirá como parámetro una dirección de correo electrónico a la que enviará dos mensajes. El primer mensaje indicará en el asunto *Ficheros borrados de /tmp* y contendrá el nombre de cada fichero o directorio que se haya eliminado. El segundo mensaje indicará en el asunto *Ficheros que se borrarán en breve* y contendrá los nombres de los ficheros para los que haga más de 1 semana pero menos de 2 que no se acceden. Los nombres de los ficheros en los dos mensajes deberán estar ordenados alfabéticamente. Para el envío de los mensajes de correo electrónico se empleará la orden: *mail < dirección_correo > -s <asunto>* Para realizar las acciones requeridas, el programa deberá crear procesos y tuberías de acuerdo al esquema que se muestra en la figura 5.1.

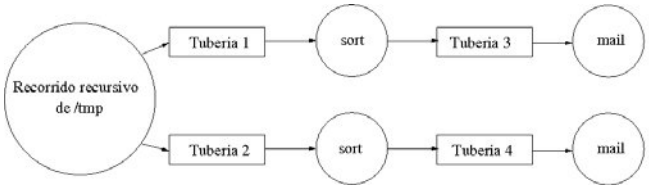

Figura 5.1: Esquema de funcionamiento.

Fíjate en que un único proceso recorre el directorio /tmp y escribe en una tubería los nombres de los ficheros y directorios que borra y en otra tubería

distinta los nombres de los ficheros que se borrarán próximamente. Otros dos procesos independientes son los encargados de la ordenación de los nombres almacenados en las tuberías y, por último, dos procesos más enviarán los mensajes de correo electrónico. En total, debes manejar 5 procesos y cuatro tuberías como se muestra en la figura 5.1.

82. Escribe un programa que cree dos procesos de manera que, a partir de dos directorios dados como parámetros de entrada, uno de los procesos realice una búsqueda de los ficheros con terminación .c y .h que se encuentren a partir del primer directorio dado (deberás recorrer el árbol de directorios) y envíe su ruta a través de una tubería a un segundo proceso que creará una copia en el segundo directorio. La copia sólo se debe realizar si se dan las dos siguientes condiciones:

    - El fichero encontrado no existe en el segundo directorio.
    - El fichero existe en el segundo directorio, pero el tiempo de última modificación de la copia es anterior al del original.

    Para realizar la copia de cada fichero otro proceso deberá ejecutar el siguiente comando: *cp <ruta_fichero> segundo_directorio*

83. Escribe un programa en C que convierta ficheros en formato *postscript* a formato *pdf*. El programa recibirá como parámetros una serie de nombres de directorios y, para cada uno de ellos, deberá crear un proceso que lo recorra recursivamente. Todos estos procesos deben ejecutarse en paralelo. Cada vez que se encuentre un fichero cuyo nombre termine en *.ps*, se considerará que se trata de un fichero en formato *postscript* y se obtendrá un fichero equivalente en formato *pdf* mediante el comando:

    $$ps2pdf < ruta_fichero.ps > < ruta_fichero.pdf >$$

    Una vez generado el fichero *pdf*, se debe borrar el fichero *postscript* correspondiente. Puedes suponer que el número de enlaces duros de cada fichero *postscript* será siempre uno.

    Antes de la finalización de cada proceso, se deberá informar de los siguientes aspectos:

    - Número de ficheros transformados.
    - Ahorro de espacio en disco medido en número de bloques.
    - Tiempo total en segundos que se ha empleado en procesar el directorio correspondiente.

84. Escribe un programa en C que realice una simulación de la asignación de grupos de prácticas a los estudiantes matriculados en una determinada asignatura. El programa recibirá como parámetros en la línea de comandos:

- el número de estudiantes matriculados en la asignatura,
- la cantidad de grupos de prácticas que existen, y
- el número de plazas en cada grupo de prácticas.

Puedes suponer que el número de estudiantes siempre será menor o igual que el número total de plazas de prácticas disponibles, es decir, que

$$numero_estudiantes \leq cantidad_de_grupos \times plazas_por_grupo$$

El programa deberá crear una serie de hilos que se ejecuten en paralelo, de modo que haya un hilo que simule el comportamiento de cada estudiante. Además, habrá otro hilo gestor que será el encargado de realizar la asignación de estudiantes a grupos de prácticas, teniendo en cuenta las preferencias de cada estudiante. Cada hilo correspondiente a un estudiante debe realizar las siguientes acciones:

a) Decidir el orden de preferencia de los grupos de prácticas.

b) Enviar de una en una al gestor una serie de peticiones de inclusión en los grupos de prácticas según el orden decidido en el punto anterior. La petición que se envía al gestor debe contener dos datos:
   - número de estudiante que hace la petición, y
   - número de grupo que se solicita.

Para cada petición recibida, el hilo gestor debe mostrar en la pantalla un mensaje que indique:

- Si la acepta porque hay plazas libres en el grupo y el estudiante aún no tiene ningún grupo asignado.
- Si la rechaza porque el grupo solicitado no tiene plazas disponibles.
- Si la rechaza porque a ese alumno ya se le había asignado otro grupo de prácticas (es decir, se le había aceptado una petición anterior).

Fíjate en que los estudiantes envían peticiones para todos los grupos de prácticas ordenadas según sus preferencias. El gestor procesará todas las solicitudes y rechazará las correspondientes a estudiantes que ya tienen asignado un grupo de prácticas o a grupos que están llenos.

La comunicación entre los hilos correspondientes a los estudiantes y el hilo gestor debe realizarse a través de un *buffer* intermedio en el que se podrán almacenar como máximo 20 peticiones. Debes garantizar que todos los accesos al *buffer* sean correctos.

Para simplificar la solución del problema, considera que ya están definidas las siguientes variables globales, estructuras y funciones:

```
/* Variables globales.
   IMPORTANTE. Debes inicializarlas en el programa principal según los
               parámetros de la línea de comandos.                    */
int num_estudiantes, cantidad_grupos, plazas_por_grupo;

/* Tipos de datos */
typedef struct {
  int num_estudiante;
  int num_grupo;
} TipoPeticion;

/* Funciones */

int *decide_preferencias(int num_estudiante);
  /* Devuelve un vector de enteros en el que los grupos de prácticas
     están ordenados según las preferencias del estudiante */

int grupo_asignado(int num_estudiante);
  /* Devuelve el número de grupo de prácticas asignado a un estudiante.
     Si todavía no se le ha asignado ningún grupo devuelve -1 */

int hay_plazas_libres(int num_grupo);
  /* Devuelve 1 si hay plazas libres en el grupo dado o 0 en caso contrario */

void asignar_grupo(int num_estudiante, int num_grupo);
  /* Asigna el estudiante al grupo indicado */
```

En concreto, se te pide que proporciones:

- La función que simula el comportamiento de un estudiante.
- La función que simula el gestor de peticiones.
- El programa principal, que debe incluir todas las inicializaciones que sean necesarias, la creación de todos los hilos y la espera hasta que todos ellos terminen.
- Todos los tipos, variables, etc. que puedas necesitar y que no aparezcan ya definidos en el enunciado.

85. Escribe un programa en C que imprima los ficheros regulares que encuentre en una serie de directorios dados como parámetros. Para cada directorio se deberá crear un proceso que lo recorra recursivamente de modo que todos ellos se ejecuten en paralelo. Los ficheros cuyo tamaño sea menor de 2 Mbytes se enviarán a la cola de impresión *ps1* y el resto se enviará a la cola de impresión *ps2*. Para enviar un fichero a una cola de impresión se deberá ejecutar el siguiente comando: *lpr -P <cola> <fichero>*. Antes de la finalización del programa se deberá indicar el tiempo total empleado en la ejecución del mismo.

    NOTA: Se incluye una versión de la función de recorrido recursivo de un directorio:

```
void recorre(char *nombredir) {
  DIR *d;
  struct dirent *entrada;
  struct stat datos;
  char *ruta;

  printf("empiezo a recorrer %s\n", nombredir);
  d= opendir(nombredir);
  if (d== NULL) {
    printf("Error al abrir el directorio\n");
    return;
  }

  entrada= readdir(d);
  while (entrada!= NULL) {
    if (strcmp(entrada->d_name, ".") &&
        strcmp(entrada->d_name, "..")) {
      ruta= malloc(strlen(nombredir)+strlen(entrada->d_name)+2);
      sprintf(ruta,"%s/%s", nombredir, entrada->d_name);
      lstat (ruta, &datos);
      if (S_ISDIR (datos.st_mode))
        recorre (ruta);
      printf ("Procesando %s\n", ruta);
      free (ruta);
    }
    entrada= readdir(d);
  }
  closedir(d);
}
```

86. Se desea realizar una simulación de una granja de gallos en la que hay dos tipos de gallos: domésticos y salvajes. En la granja, la vida de un gallo cualquiera consiste básicamente en comer, beber y dormir en este orden. Para realizar la acción de beber se accede a una fuente de agua común para todos los gallos y se evita que un gallo salvaje se junte con gallos domésticos o con otros gallos salvajes mientras dure la acción. Por contra, un gallo doméstico si que puede realizar la acción de beber junto a otros gallos domésticos. Respecto a las acciones de comer y dormir no se establece ninguna condición. La granja cuenta con un total de 30 gallos, 25 de ellos domésticos y el resto salvajes. Utiliza el programa que se acompaña para resolver las siguientes cuestiones:

    a) En primer lugar, se desea mejorar la función dormir de manera que ahora tendrá dos parámetros de entrada: *void dormir (int id, int tiempo);*. La variable *id* es un identificador del gallo y *tiempo* será el tiempo que va a dormir (un valor entero aleatorio entre 7 y 10). Modifica el programa principal de manera que, para cada gallo de la simulación, se envíe ambos parámetros a las funciones correspondientes *galloDomestico* y *galloSalvaje*, y modifica éstas también de forma conveniente.

    b) Realiza las modificaciones oportunas de manera que se simule la vida de los gallos domésticos y salvajes tal y como se explica en el enunciado.

c) Limita el número de gallos domésticos realizando la acción de beber al mismo tiempo a 6.

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#define N 30;

/* simula la vida de un gallo salvaje */
void *galloSalvaje (void *nada){

  while (1) {
    comer();  /* función que simula la acción de comer */
    beber();  /* función que simula la acción de beber */
    dormir(); /* función que simula la acción de dormir */
  }
  pthread_exit(NULL);
}

/* simula la vida de un gallo doméstico */
void *galloDomestico (void *nada){

  while (1) {
    comer();   /* función que simula la acción de comer */
    beber();   /* función que simula la acción de beber */
    dormir();  /* función que simula la acción de dormir */
  }
  pthread_exit(NULL);
}

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

  pthread_t th[N];
  int i;

  for (i= 0; i< 25; i++) /* crea los gallos domésticos */
    pthread_create (&th[i], NULL, galloDomestico, NULL);
  for (i= 25; i< N; i++) /* crea los gallos salvajes */
    pthread_create (&th[i], NULL, galloSalvaje, NULL);

  for (i= 0; i< N; i++)
    pthread_join(th[i], NULL);

  exit(0);
}
```

87. Escribe un programa en C que ejecute en paralelo cuatro hilos. De ellos, tres serán productores de datos y uno consumidor. Los hilos se comunicarán a través de un *buffer* intermedio en el que se podrán almacenar como máximo 5 datos.

    Cada hilo productor se encarga de generar como máximo 1000 números aleatorios. Para cada número generado, comprueba si es un número primo y, de ser así, lo almacena en el *buffer*. El hilo consumidor extrae números del *buffer* y escribe en pantalla aquellos números para los que la suma de sus dígitos

sea impar.

El programa terminará cuando el hilo consumidor haya escrito cincuenta números en pantalla o cuando cada productor haya generado sus 1000 números aleatorios. Antes de terminar, el programa informará de la cantidad de números primos procesados y la cantidad de números aleatorios generados.

Puedes considerar que las siguientes funciones ya están definidas:

```
int numero_aleatorio(void);
int es_primo(int numero);
int suma_digitos(int numero);
```

88. Escribe un programa para *limpiar* el directorio */tmp*. Recorriendo todos los subdirectorios que se encuentren a partir de */tmp*, el programa debe eliminar todos los ficheros regulares que pertenezcan al usuario que ejecuta el programa y cuyo tamaño sea superior a 2 Mbytes. Además, se deben eliminar los directorios que ya estaban vacíos o queden vacíos al eliminar los ficheros que contenían. También se deben identificar los ficheros regulares que ocupen más de 1 Mbyte pero menos de 2 Mbytes para informar al usuario de que esos ficheros están ocupando una gran cantidad de espacio en disco.

    El programa recibirá como parámetro una dirección de correo electrónico a la que enviará dos mensajes. El primer mensaje indicará en el asunto *Ficheros borrados de /tmp* y contendrá el nombre de cada fichero o directorio que se haya eliminado. El segundo mensaje indicará en el asunto *Ficheros que ocupan mucho espacio en /tmp* y contendrá el nombre y tamaño en bytes de cada fichero cuyo tamaño esté entre 1 y 2 Mbytes.

    Para el envío de los mensajes de correo electrónico se empleará la orden:

    mail <dirección_correo> -s <asunto>

    Para realizar las acciones requeridas, el programa deberá crear procesos y tuberías de acuerdo al esquema que se muestra en la figura 5.2.

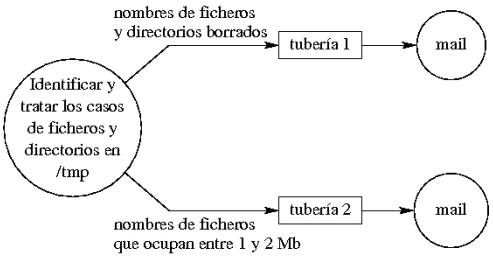

Figura 5.2: Esquema de funcionamiento.

Fíjate en que un único proceso recorre el directorio */tmp* y escribe en una tubería los nombres de los ficheros y directorios que borra y en otra tubería distinta los nombres de los ficheros que ocupan más de 1 Mbyte. Otros dos

procesos independientes son los encargados de enviar los mensajes de correo electrónico. En total, debes manejar 3 procesos y 2 tuberías como se muestra en la figura.

89. Un programador quiere simular el movimiento de camiones y barcos en una estación petroquímica. Los barcos llegan a la estación para descargar de sus bodegas el producto crudo que posteriormente se refina y se carga en los tanques de los camiones que van llegando a la estación. Las condiciones de funcionamiento de la estación que ha de tener en cuenta el programador son:

    a) La estación tiene capacidad para atender a tantos camiones como lleguen.
    b) La estación tiene capacidad para atender a los barcos de uno en uno.
    c) Mientras se atiende a un barco no se puede atender a nuevos camiones pero sí a los camiones que hubiesen ya en la estación.

    En base a estas condiciones, el programador escribe el siguiente código:

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

#define N_vehiculos 200
sem_t estacion;

void *camion (void * nada) {

  long int n= (long int) pthread_self();

  printf ("Llega camion: %ld\n", n);
  sem_wait(&estacion);
  printf ("Se atiende camion: %ld\n", n);
  sleep(random() %3+2); /* tiempo invertido en atender al camion */
  printf ("Sale camion: %ld\n", n);
  pthread_exit(NULL);
}

void *barco (void * nada) {

  long int n= (long int) pthread_self();

  printf ("Llega barco: %ld\n", n);
  sem_wait(&estacion);
  printf("Se atiende barco: %ld\n", n);
  sleep(random() %5+5); /* tiempo invertido en atender al barco */
  printf ("Sale barco: %ld\n", n);
  sem_post(&estacion);
  pthread_exit(NULL);
}

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

  int i;
  pthread_t th;
```

```
  for (i= 0; i< N_vehiculos; i++) {
    sleep(random()%3);
    if (random()%100 < 95)
      pthread_create(&th, NULL, camion, NULL);
    else
      pthread_create(&th, NULL, barco, NULL);
  }
  for (i= 0; i< N_vehiculos; i++)
    pthread_join(th, NULL);

  return 0;
}
```

Se pide:

a) Puesto que el código anterior contiene errores y no satisface las tres condiciones descritas en el enunciado, corrige los errores e indica las modificaciones necesarias para que se satisfagan.

b) Haz las modificaciones oportunas (al programa resultado del apartado anterior) para que, sin dejar de cumplirse las condiciones de funcionamiento a) y b), se cumpla también la siguiente nueva condición:

1) Mientras se atiende a un barco no se puede atender a nuevos camiones y el barco sólo puede ser atendido cuando no hayan camiones utilizando la estación.

90. Escribe un programa que ejecute la siguiente línea de órdenes igual que lo haría un intérprete de comandos:

grep palabra file1 | sort -r | uniq > file_salida

y que además nos diga cuánto tiempo (en segundos) ha tardado en ejecutarse completamente. Tanto *file1*, *file_salida* como *palabra* serán parámetros dados por tu programa en la línea de comandos en ese orden.

91. Simula el movimiento de personas que entran y salen de una sucursal bancaria teniendo en cuenta las siguientes condiciones. En dicha sucursal, durante el horario de atención al público, un furgón blindado puede llegar para la carga y descarga de dinero. Para que se realice esta operación, es necesario que no haya clientes en el interior de la sucursal, por lo que los guardias de seguridad del furgón deben esperar a que la sucursal se encuentre vacía antes de iniciar la operación. La sucursal puede atender a tantos clientes como lleguen, no hay límite de capacidad. Utiliza el programa que se acompaña, que contiene algún error, y da dos soluciones al problema, una por cada una de las siguientes condiciones respecto al momento de llegada del furgón a la sucursal:

a) Que no se impida la entrada de nuevos clientes a la sucursal mientras existan clientes en su interior.

b) Que sí se impida la entrada de nuevos clientes a la sucursal.

(Recuerda que en ambos casos, la operación no puede empezar hasta que la sucursal esté sin clientes, como ya se deja claro en el enunciado)

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

#define N_hilos 200

void *cliente (void *n){

  int id= *(int *)n;

  sleep  (random() %100);
  printf ("Llega el cliente %d\n", id);
  sleep  (random() %3+2); /* tiempo de espera en la cola */
  printf ("Se atiende al cliente: %d\n", id);
  sleep  (random() %3+2); /* tiempo invertido en la operacion */
  printf ("Sale el cliente %d\n",id);

  pthread_exit(NULL);
}



void *furgon (void *n){

  int id= *(int *)n;

  sleep  (random() %100);
  printf ("LLega el furgon %d\n", id);
  printf ("Se atiende a los guardias %d\n", id);
  sleep  (random() %5 + 5); /* tiempo invertido en la operacion */
  printf ("Se va el furgon  %d\n",id);

  pthread_exit(NULL);
}

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

  pthread_t hilo;
  int i, furgon_creado= 0;

  for (i= 0; i< N_hilos; i++)
    if (furgon_creado== 1)
      pthread_create(&hilo, NULL, cliente, (void *) &i);
    else
      if (random() %100 < 95)
        pthread_create(&hilo, NULL, cliente, (void *) &i);
      else {
        pthread_create(&hilo, NULL, furgon, (void *) &i);
        furgon_creado= 1;
      }
  return 0;
}
```

92. Se desea realizar una implementación en C del problema clásico del productor-consumidor. Las características propias de esta implementación son:

- Los hilos se comunican a través de un búffer intermedio de capacidad 10.
- El hilo productor produce un número aleatorio de datos.
- El hilo consumidor escribe el número consumido en la pantalla.
- El programa termina cuando se hayan consumido los datos producidos.

En concreto, se pide:

a) Escribe el programa principal que: crea, inicializa y destruye los semáforos; crea dos hilos, uno productor y otro consumidor; espera a que se cumpla la condición de terminación del programa.

b) Escribe el código del hilo productor y del consumidor.

c) Escribe las modificaciones oportunas para que en lugar de un hilo productor hayan tres que se ejecuten en paralelo y garantizando que se consumen todos los datos.

d) Escribe las modificaciones oportunas para que en lugar de un hilo consumidor hayan dos que se ejecuten en paralelo (es decir, tendremos tres productores y dos consumidores).

NOTA: Se incluye una versión incompleta del código.

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

#define MAX 10

int buffer[MAX];

int pos_productor= 0, pos_consumidor= 0;
int menos_uno= 0;

int genera_dato(void);
int numero_aleatorio(void);

void *productor (void *nada) {
  int i, dato, n= numero_aleatorio();
  for(i=0; i< n; i++){
    dato= generar_dato();
    sleep(random()%2);
    buffer[pos_productor]= dato;
    pos_productor= (pos_productor+1)%MAX;
  }
  buffer[pos_productor]= -1;
  pos_productor= (pos_productor+1)%MAX;
  pthread_exit(NULL);
}
```

```
void *consumidor (void *nada) {
  int i,dato;
  while(menos_uno!= 1){
    dato= buffer[pos_consumidor];
    pos_consumidor= (pos_consumidor+1) %MAX;
    if (dato== -1)
      menos_uno++;
    printf ("%d", dato);
    sleep(random(%2));
  }
  pthread_exit(NULL);
}
```

93. Escribe un programa en C que, a partir de una serie de directorios que se proporcionan como parámetros, los recorra recursivamente y calcule cuántos enlaces simbólicos hay dentro de ellos. El programa debe crear un hilo por cada directorio dado, de modo que el recorrido de cada uno de ellos se ejecute en paralelo con el resto. Por simplicidad, se considerará que nunca habrá más de 10 directorios como parámetros. Tanto para los directorios dados como parámetros como para los que aparezcan en el recorrido recursivo de los mismos, sólo se deben tener en cuenta aquellos directorios que pertenezcan al usuario que ejecuta el programa. El resto de directorios simplementen se ignorarán.

    Nota. No se pide el número de enlaces que hay en cada uno de los directorios dados, sino la suma de todos ellos. Es decir, el resultado del programa debe ser un único número.

94. Se desea escribir un programa que implemente una primera aproximación a la generación de un cierto tipo de claves criptográficas. Para ello, se deben encontrar pares de números p y q tales que:

    - p y q sean primos;
    - $(p-1)/2$ y $(q-1)/2$ también sean primos.

    Si se cumplen estas dos condiciones, entonces el producto $p * q$ nos proporciona una clave criptográfica. Para resolver el problema planteado se utiliza una estrategia productor-consumidor mediante tres procesos que se ejecutan en paralelo de la siguiente forma:

    - Dos procesos productores se encargan de analizar secuencialmente todos los números de un rango dado para averiguar cuáles de ellos son primos.
    - Un proceso consumidor que, repetidamente, toma un par cualquiera de números primos de los encontrados por los productores y comprueba si dicho par de números cumplen los requisitos necesarios para generar una clave criptográfica. En caso afirmativo, muestra por pantalla los dos números utilizados y el valor de la clave criptográfica generada. En

cualquier caso, estos dos números utilizados serán descartados y no se volverán a considerar.

Al acabar la ejecución, el programa debe indicar la cantidad total de números primos que han encontrado los productores, así como el número de claves criptográficas generadas.

Escribe un programa en C que proporcione una solución válida al problema utilizando tuberías.

95. Resuelve el ejercicio 94 utilizando 3 hilos (2 productores y 1 consumidor) que ahora se comunican a través de un buffer intermedio de capacidad 20 elementos en lugar de utilizar tuberías. La sincronización en la ejecución de los hilos se debe realizar mediante semáforos.

96. Escribe un programa que, a partir de un directorio dado como parámetro comprima y añada al fichero */tmp/comprimido.zip* todos los ficheros regulares que encuentre a partir de dicho directorio (deberá recorrer el árbol de directorios a partir de dicho directorio) tales que:

    - Pertenezcan al usuario que ejecuta el programa.
    - Se hayan modificado desde la última hora (medida desde el inicio de la ejecución del programa).

    Para comprimir un fichero *fich* y añadirlo al fichero */tmp/comprimido.zip* utiliza el programa *zip* de la siguiente forma: *$ zip /tmp/comprimido.zip fich*.

    Ten en cuenta que no se pueden hacer varias acciones *zip* sobre el mismo archivo de manera simultánea. Además, por cada fichero regular que encuentres que cumpla las condiciones anteriores deberás enviar su ruta a través de una tubería a otro proceso que creará un fichero llamado *usuario.log* que contendrá un informe ordenado, utiliza el comando *sort*, de las rutas de los ficheros regulares añadidos al archivo *zip*.

97. En una reunión de atletismo, hay tres tipos de atletas participantes. Por un lado los lanzadores de jabalina, por otro los lanzadores de martillo y por último los corredores. A primera hora del día se les da acceso libre a las instalaciones para entrenar. Por seguridad, se han establecido unas restricciones de uso de las instalaciones:

    a) Un atleta lanzador de martillo o de jabalina debe esperar para entrar en las instalaciones a que no haya ningún atleta, sea del tipo que sea, utilizándolas.

    b) Si hay un atleta lanzador de jabalina o de martillo utilizando las instalaciones, un nuevo atleta que llegue, sea del tipo que sea, debe esperar para entrar en las instalaciones a que el que ya hay la abandone.

c) Si en las instalaciones hay algún atleta corredor y llega un nuevo atleta corredor, este entrará en las instalaciones incluso aunque hubiesen atletas lanzadores esperando para entrar.

Se pide:

- Completa el código que se te da a continuación de forma que se simule el comportamiento descrito en el enunciado. Las órdenes *printf* deben permanecer en tu solución. Sitúalas donde debas.
- Modifica el código solución del apartado anterior para que se respete el orden de llegada de los atletas a las instalaciones. De esta forma, si llega un nuevo atleta corredor habiendo atletas lanzadores esperando, el atleta corredor deberá esperar a que entren y salgan de las instalaciones los lanzadores que llegaron antes que él, incluso aunque en el momento de la llegada hubiesen atletas corredores en las instalaciones.

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>

#define N_atletas 30

void *corredor (void * nada) {

  int n= (int) syscall(SYS_gettid);
  printf ("Llega corredor: %d\n", n);
  printf ("Entra en las instalaciones el corredor %d\n", n);
  sleep  (random() %3); /* tiempo invertido en usarlas */
  printf ("Sale de las instalaciones el corredor: %d\n", n);

  pthread_exit(NULL);
}

void *jabalino (void * nada) {

  int n= (int) syscall(SYS_gettid);
  printf ("Llega lanzador de jabalina: %d\n", n);
  printf ("Entra en las instalaciones el jabalino %d\n", n);
  sleep  (random() %3+2); /* tiempo invertido en usarlas */
  printf ("Sale de las instalaciones el jabalino: %d\n", n);

  pthread_exit(NULL);
}

void *martillo (void * nada) {

  int n= (int) syscall(SYS_gettid);
  printf ("Llega lanzador de martillo: %d\n", n);
  printf ("Entra en las instalaciones el martillo %d\n", n);
  sleep  (random() %3+2); /* tiempo invertido en usarlas */
  printf ("Sale de las instalaciones el martillo: %d\n", n);

  pthread_exit(NULL);
}
```

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

  int num;
  pthread_t th[N_atletas];

  for (num= 0; num< N_atletas; num++) {
    sleep(random()%3);
    if (random()%100 < 66)
      pthread_create(&th[num], NULL, corredor, NULL);
    else
      if (random()%100 < 82)
    pthread_create(&th[num], NULL, jabalino, NULL);
      else
    pthread_create(&th[num], NULL, martillo, NULL);
  }

  for (num= 0; num< N_atletas; num++)
    pthread_join(th[num], NULL);

  return 0;
}
```

98. En un concurso televisivo de la canción denominado Egovision participan grupos musicales en representación de diferentes países. Después de las actuaciones, comienza la fase de la votación. En esta fase, cada persona del público vota enviando un SMS indicando el país y la puntuación entre 0 y 10. Durante la fase de votación, cada país participante puede consultar el estado de las votaciones cuando quiera y cuantas veces desee. Escribe un programa en C que realice una simulación de este proceso y que además satisfaga las siguientes condiciones:

    - Utiliza un vector de enteros, de dimensión el número de países, para almacenar los puntos que cada país recibe y llámalo *puntuaciones*.
    - La simulación genera un total de 100 hilos.
    - Cada hilo generado será de tipo voto con una probabilidad del 90 % o de tipo consulta con una probabilidad del 10 %.
    - Cada hilo de tipo voto recibe dos datos generados de forma aleatoria, el país votado (representado por un identificador de tipo entero) y una puntuación, y acumula la nueva puntuación del país indicado a la que ya hay en el vector puntuaciones.
    - Cada hilo de tipo consulta recibe un dato generado de forma aleatoria: el país que realiza la consulta. Se debe permitir que varias consultas se puedan realizar de forma simultánea garantizando la coherencia de los datos mostrados, es decir, que si varios países realizan consultas al mismo tiempo, entonces todos deben mostrar el mismo resultado.
    - Para comprobar visualmente la simulación sitúa en el código para cada tipo de hilo la impresión de los mensajes, que se facilitan a continuación, de forma correcta.

```
#define NHILOS 100
#define PAISES 10

char paises[PAISES][10]={"España", "Portugal", "Francia", "Ucrania",
"Letonia", "Servia", "Israel", "Alemania", "Italia", "Holanda"};

int puntuaciones[PAISES]={0,0,0,0,0,0,0,0,0,0};

funcion voto {

  imprime("Voto al pais %s %d puntos\n",paises[pais],puntos);

  imprime("Ya he votado al pais %s %d puntos\n",paises[pais],puntos);

}

funcion consulta {

  imprime ("El pais %s realiza la consulta\n",paises[pais]);

  Para (int i=0; i< PAISES;i++)
    imprime("(%s,%d)\n",paises[i],puntuaciones[i]);

  imprime("El pais %s termina la consulta\n",paises[pais]);
}
```

99. En un concurso televisivo de la canción denominado Egovision participan grupos musicales en representación de diferentes países. Después de las actuaciones, comienza la fase de la votación. En esta fase, cada persona del público vota enviando un SMS indicando el país y la puntuación entre 0 y 10. Escribe un programa en C que realice una simulación de este proceso y que además satisfaga las siguientes condiciones:

    - Utiliza un vector de enteros, de dimensión el número de países, para almacenar los puntos que cada país recibe y llámalo puntuaciones.
    - La simulación genera N procesos nuevos que en paralelo emiten un voto cada uno de ellos.
    - Una tubería comunica los N procesos nuevos con el proceso principal.
    - Cada proceso que emite un voto genera dos datos de forma aleatoria: el país al que vota (un identificador de tipo entero) y la puntuación correspondiente. Ambos son enviados por la tubería al proceso principal.
    - El proceso principal actualiza el resultado de la votación a medida que le llegan los votos por la tubería. Una vez recogidos todos los votos, el resultado final se muestra ordenado (ver ejemplo a continuación). Para conseguir la ordenación se envía una línea de texto por país, con el formato del ejemplo, mediante una nueva tubería a un proceso que hace la ordenación de los datos recibidos mediante la orden *sort -r*.

```
156 puntos, España
150 puntos, Alemania
135 puntos, Holanda
.
.
```

- No deben aparecer ni procesos huérfanos ni procesos zombies.

```
#define N 100
#define PAISES 10

char paises[PAISES][10]={"España", "Portugal", "Francia", "Ucrania",
"Letonia", "Servia", "Israel", "Alemania", "Italia", "Holanda"};

int puntuaciones[PAISES]={0,0,0,0,0,0,0,0,0,0};
```

100. Se desea conocer el número de nodo-i de cada uno de los ficheros que en un sistema dado tengan un tamaño superior a 1024 bloques. También se quiere saber cuántos ficheros satisfacen esta condición. Escribe un programa en C que, para conseguir ambos objetivos, actúe de la siguiente manera:

    - La búsqueda de ficheros comienza siempre a partir de un directorio dado como parámetro de entrada.
    - Se podrán especificar varios directorios como parámetros de entrada.
    - El proceso inicial deberá crear tantos procesos de búsqueda como directorios especificados como parámetros de entrada y deberán ejecutarse en paralelo tanto entre ellos como con él mismo. De esta manera, cada proceso creado realizará la búsqueda recursiva de ficheros en el árbol de directorios de uno de los directorios dados como parámetros de entrada.

    Cuando un proceso de búsqueda encuentre un fichero que satisfaga la condición dada (tamaño superior a 1024 bloques), enviará su número de nodo-i a través de una tubería a un proceso que realizará una ordenación de todos los números recibidos utilizando el comando del sistema *sort*. Cuando un proceso de búsqueda haya terminado de recorrer el árbol de directorios, enviará al proceso inicial, y a través de otra tubería, el número de ficheros encontrados que han cumplido la condición dada. Una vez todos los procesos de búsqueda creados hayan terminado, el proceso inicial informará del total de ficheros encontrados.

Capítulo 6

Soluciones

1. La jerarquía resultante se puede observar en la figura 6.1.

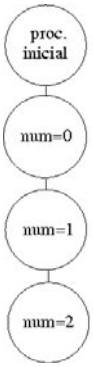

Figura 6.1: Jerarquía en profundidad donde para cada proceso se indica la iteración del bucle en la que se crea.

   Aparecen mensajes repetidos porque la orden *printf*, la que hay dentro del bucle, la ejecuta cada proceso hijo una primera vez tras ser creado como resultado de la llamada al sistema *fork* que hay justo en la línea anterior del *printf*, y una segunda vez en la siguiente iteración del bucle (excepto en el caso de encontrarse en su última iteración).

   Se observa que los procesos terminan en el orden contrario al que se han creado, es decir, primero termina el último proceso creado y el último proceso en terminar es el inicial. Esto ocurre gracias a la llamada al sistema *wait* que cambia el estado del proceso padre a bloqueado hasta que el proceso hijo termine.

2. La jerarquía resultante se puede observar en la figura 6.2.

   Se observa que el proceso padre es siempre el último en terminar. Esto es debido a que la llamada al sistema *wait*, que hace que el proceso padre pase a estado bloqueado hasta que un proceso hijo termine, se encuentra dentro de un bucle que va a hacer que se llame tantas veces como procesos hijos creó.

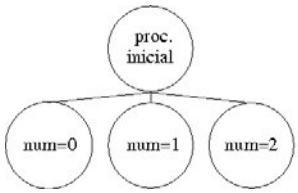

Figura 6.2: Jerarquía en anchura donde para cada proceso se indica la iteración del bucle en la que se crea.

Sin embargo, los procesos hijos acaban en cualquier orden dependiendo del retardo que cada uno realice con la orden *sleep*.

3. La jerarquía resultante se puede observar en la figura 6.3.

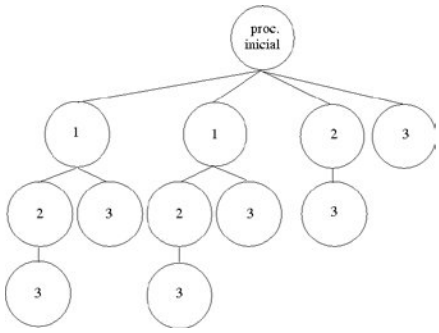

Figura 6.3: Árbol de procesos donde para cada proceso se indica la llamada a *fork* responsable de su creación.

4. La jerarquía resultante se puede observar en la figura 6.4.

5. La jerarquía resultante se puede observar en la figura 6.5.

6. La jerarquía resultante se puede observar en la figura 6.6. A continuación se muestra un ejemplo del código con las llamadas correspondientes.

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

#define L1 2
#define L2 3

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

  int cont1, cont2;
  pid_t pid;

  for (cont2= 0; cont2< L2; cont2++) {
    for (cont1= 0; cont1< L1; cont1++) {
      pid= fork();
      if (pid== 0) break;
```

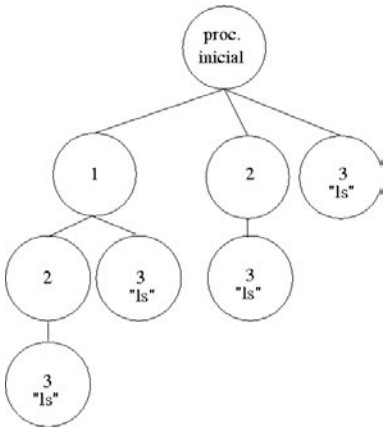

Figura 6.4: Árbol de procesos donde para cada proceso se indica la iteración en la que se crea y si realiza el cambio de imagen (aquellos que contienen ”ls”).

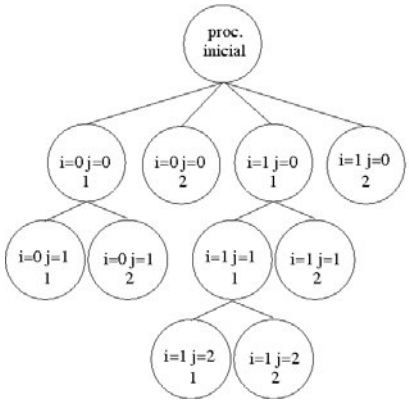

Figura 6.5: Árbol de procesos donde para cada uno se indica la iteración en la que se crea y la llamada a *fork* responsable de su creación.

```
    }
    if (pid!= 0) break;
  }

  printf ("Soy el proceso de PID %d y mi padre tiene %d de PID.\n",
          getpid(), getppid());
  if (pid!= 0)
    for (cont1= 0; cont1< L1; cont1++)
      printf ("Fin del proceso de PID %d.\n", wait (NULL));

  return 0;
}
```

7. La jerarquía resultante se puede observar en la figura 6.7. A continuación se muestra el código para que se produzcan las esperas solicitadas así como se informe de los tiempos correspondientes. Señalar que todos los procesos hijos ejecutan la aplicación *kcalc* y que la ejecución de *xload* nunca se produce.

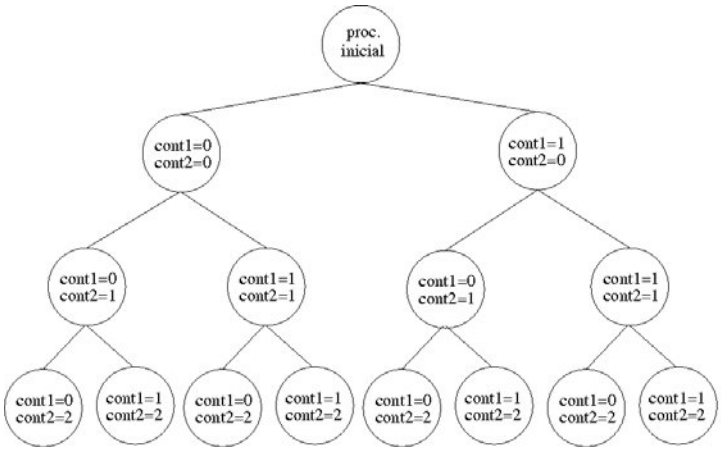

Figura 6.6: Árbol de procesos donde para cada uno se indica la iteración en la que se crea.

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

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

  int i, j;
  pid_t pid, nuevo, nuevo1;
  time_t ini, fin;

  ini= time(NULL);
  for (i= 0; i< 2; i++){
    pid= getpid();
    for (j= 0; j< i+2; j++){
      nuevo= fork();
      if(nuevo== 0){
        break;
        nuevo1= fork();
        if(nuevo1== 0)
          execlp ("xload", "xload", NULL);
      }
    }
    if (pid!= getpid())
      execlp ("kcalc", "kcalc", NULL);
  }

  for (i= 0; i< 2; i++)
    for (j= 0; j< i+2; j++){
      wait(NULL);
      printf ("Tiempo en ejecución de kcalc: %ld\n", time(NULL)-ini);
    }

  printf ("Tiempo total: %ld\n", time(NULL)-ini);
  return 0;
}
```

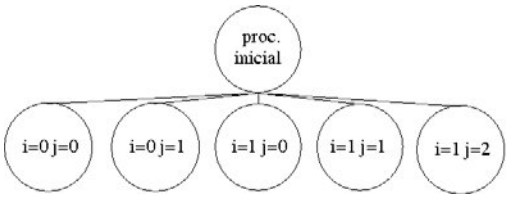

Figura 6.7: Jerarquía donde para cada proceso hijo se indica la iteración del bucle en la que se crea.

8. A continuación se muestra el trozo de código que habría que insertar en el programa del enunciado donde figura el comentario de */* completar aquí */*.

```
for (prof= 0; prof< profundidad-1; prof++)
  if (fork()!= 0) /* jerarquía en profundidad */
    break;

/* if ((prof!= 0) && (prof%2== 0)) */ /* ampliación */
for (anch= 0; anch< anchura-1; anch++)
  if (fork()== 0) /* jerarquía en anchura */
    break;
```

9.
```
for (cont1= 0; cont1< n; cont1++){
  pid= getpid();
  for(cont2= 0;cont2< cont1+2; cont2++){
    nuevo= fork();
    if (nuevo == 0)
      break;
  }
  npid= getpid();
  if (npid== pid)
    break;
}
```

10. Para medir el tiempo habrá que realizar dos llamadas a *time()*, una antes de ejecutar *prueba.exe* y otra después. Restando los valores obtenidos calculamos el tiempo que ha transcurrido. Ya que el proceso hijo es quien va a realizar el cambio de imagen, la medición de tiempo la debe realizar el proceso padre. El proceso padre no toma el valor inicial del tiempo correcto, que está inicializado a cero, ya que la linea *inicio= time();* la ejecuta el proceso hijo. La solución es que dicha línea la ejecute el proceso padre poniéndola, por ejemplo, justo antes de la llamada a *wait(NULL);*.

11. Si la aplicación *kcalc* está en alguna de las rutas almacenadas por la variable de entorno *PATH*, se producirá el cambio de imagen en el proceso pasando a ejecutarse la aplicación *kcalc*. En el caso de no encontrarse, la ejecución del proceso continua imprimiendo el mensaje e intentará realizar el cambio de imagen con la aplicación *xload*. En el caso de tampoco encontrarla, el proceso continua su ejecución imprimiendo un nuevo mensaje y finalizando. A continuación se muestra el programa modificado tal y como se solicita en el enunciado.

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


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

  pid_t pid_kcalc, pid_xload, pidfin;
  int num;

  pid_kcalc= fork();
  if (pid_kcalc== 0) {
    execlp ("kcalc", "kcalc", NULL);
    printf ("No se encuentra la aplicación kcalc\n");
    exit (0);
  }

  pid_xload= fork();
  if (pid_xload== 0) {
    execlp ("xload", "xload", NULL);
    printf ("No se encuentra la aplicación xload\n");
    exit (0);
  }

  for (num= 0; num< 2; num++) {
    pidfin= wait (NULL);
    if (pidfin== pid_kcalc)
      printf ("Ha terminado kcalc\n");
    else
      printf ("Ha terminado xload\n");
  }

  return 0;
}
```

12.

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


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

  pid_t pid_kcalc, pid_xload, pidfin;
  int num;
  time_t ini, fin;

  ini= time (NULL);

  pid_kcalc= fork();
  if (pid_kcalc== 0) {
    execlp ("kcalc", "kcalc", NULL);
    printf ("No se encuentra la aplicación kcalc\n");
    exit (0);
  }

  pid_xload= fork();
```

```
  if (pid_xload== 0) {
    execlp ("xload", "xload", NULL);
    printf ("No se encuentra la aplicación xload\n");
    exit (0);
  }

  for (num= 0; num< 2; num++) {
    pidfin= wait (NULL);
    fin= time (NULL);
    if (pidfin== pid_kcalc)
      printf ("Ha terminado kcalc, %ld segundos\n", fin-ini);
    else
      printf ("Ha terminado xload, %ld segundos\n", fin-ini);
  }
  fin= time (NULL);
  printf ("Proceso padre, %ld segundos\n", fin-ini);

  return 0;
}
```

13.

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

#define MAXLON 100

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

  char cadena1[MAXLON], cadena2[MAXLON];
  int pos;
  pid_t pid;

  printf ("Introduce la cadena 1\n"); gets (cadena1);
  printf ("Introduce la cadena 2\n"); gets (cadena2);

  pid= fork();
  if (pid!= 0) { /* proceso padre */
    for (pos= 0; cadena1[pos]!= '\0'; pos++) {
      printf ("%c", cadena1[pos]);
      fflush (stdout);
      sleep (random() %3);
    }
    wait (NULL); /* espera a que termine el hijo */
    printf ("\n");
  }
  else /* proceso hijo */
    for (pos= 0; cadena2[pos]!= '\0'; pos++) {
      printf ("%c", cadena2[pos]);
      fflush (stdout);
      sleep (random() %3);
    }

  return 0;
}
```

14.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdbool.h>

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

  pid_t nuevo;
  bool fin= false;
  char nombre[50], esperar[5];

  while (!fin) {
    printf ("Nombre: "); scanf ("%s", nombre);
    if (strcmp(nombre, "salir")!= 0) {
      printf ("Esperar? "); scanf ("%s", esperar);
      nuevo= fork ();
      if (nuevo!= 0) { /* proceso padre */
        if (strcmp (esperar, "si")== 0)
          while (nuevo!= wait (NULL));
        else
          printf ("PID del proceso hijo %d\n", nuevo);
      } else { /* proceso hijo */
        if (execlp (nombre, nombre, NULL)== -1)
          printf ("No se ha podido ejecutar. Variable PATH: %s\n",
                  getenv ("PATH"));
        exit(0);
      }
    }
    else
      fin= true;
  }
  return 0;
}
```

15.

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

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

  pid_t nuevo;

  if (argc!= 3) {
    printf ("Especfica dos numeros enteros.\n");
    exit (0);
  }

  nuevo= fork ();
  if (nuevo!= 0) {
    nuevo= fork ();
    if (nuevo!= 0) {
      wait (NULL);
      wait (NULL);
      printf ("Factoriales calculados.\n");
    }
    else {
      execl ("./factorial", "./factorial", argv[2], NULL);
      printf ("No se encuentra la aplicacion factorial\n");
      exit(0);
```

```
    }
  }
  else {
    execl ("./factorial", "./factorial", argv[1], NULL);
      printf ("No se encuentra la aplicacion factorial\n");
      exit(0);
    }

  return 0;
}
```

16.
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

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

  pid_t proceso, proc1;
  int param;

  if (argc< 2) {
    printf ("Indica los numeros de los que quieres calcular su factorial.\n");
    exit(0);
  }

  for (param= 1; param< argc; param++) {
    proceso= fork();
    if (param== 1)
      proc1= proceso;
    if (proceso== 0) {
      execl ("./factorial", "./factorial", argv[param], NULL);
      printf ("No se encuentra la aplicacion factorial\n");
      exit(0);
    }
  }

  for (param= 1; param< argc; param++) {
    proceso= wait(NULL);
    if (proceso!= proc1)
      printf ("Proceso de PID %d finalizado\n", proceso);
  }

  printf ("Factoriales calculados.\n");
  return 0;
}
```

17.
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>


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

  pid_t pid;
  time_t inicio, fin;
```

```
  if (argc< 2) {
    printf ("Indica un programa a ejecutar.\n");
    exit(0);
  }

  pid= fork();
  if (pid== 0) {
    execvp (argv[1], &argv[1]);
    printf ("No se encuentra la aplicacion %s\n", argv[1]);
    exit(0);
  }
  inicio= time (NULL);
  wait (NULL);
  fin= time (NULL);

  printf ("Duración: %ld segundos\n", fin-inicio);
  return 0;
}
```

18.

E/S D2												
E/S D1							A1	A2				
CPU	IR	A1	A2	IS	IR	IS	B1	B2	IH1	IR	B3	B4

E/S D2								A1	A2	A3	A4	A5
E/S D1												B1
CPU	IR	A1	A2	A3	IR	IS	IS	B5	IR	IS	IS	N

E/S D2												
E/S D1												
CPU	IH1	IR	IH2	B1	IR	IS	IS	A1	IR	IS	IS	

19.

E/S								A1	A2	A3		
CPU	IR	A1	A2	A3	IS	IR	IS	B1	B2	B3	IH	IR

E/S									A1			
CPU	B4	B5	B6	IR	A1	A2	IS	IS	IR	IH	B7	B8

E/S				B1								
CPU	IS	IR	IS	A1	IH	IS	IR	IS	B1	IS	IS	

20.

E/S												A1
CPU	IR	A1	A2	A3	IR	A4	A5	A6	IR	IS	IS	B1

E/S							B1	B2	B3	B4	B5	B6
CPU	IH	IR	B2	IS	IR	IS	A1	A2	IR	IS	IS	N

E/S												
CPU	IH	IR	B1	B2	IR	B3	IS	IS				

E/S							A1	A2	A3	A4	A5	A6
CPU	IR	A1	A2	A3	IR	IS	B1	B2	IR	B3	B4	B5

21.

E/S	A7	A8	A9	A10				C1	C2	C3	C4	
CPU	IR	C1	C2	C3	IH	IR	IS	A1	IR	A2	A3	IH

E/S												
CPU	IR	C1	IS	B6	IR	IS	A4	IS				

E/S								A1				
CPU	IR	A1	A2	A3	IR	IS	IS	B1	IH	IR	B2	B3

22.

E/S					A1							
CPU	IR	A1	IS	IS	IR	IH	B4	IS	IR	IS	A1	IS

E/S												
CPU	IR	IS										

E/S												
CPU	IR	A1	A2	A3	IR	A4	A5	A6	IR	B1	B2	IS

23.

E/S	B1											B1
CPU	IR	IH	A7	A8	IR	A9	A10	A11	IR	B1	IS	A12

E/S												
CPU	IH	IR	A13	A14	IR	IS	B1	IS				

E/S D2											B1	B2
E/S D1					A1	A2	A3	A4	A5	A6	A7	A8
CPU	IR	A1	IS	IS	IR	B1	B2	IS	IR	IS	C1	C2

24.

E/S D2												
E/S D1												
CPU	IH1	IH2	IR	C3	IR	B1	B2	B3	IR	B4	B5	B6

E/S D2												
E/S D1												
CPU	IR	A1	IS	IS	IR	C4	IS	IS	IR	B7	IS	IS

E/S												
CPU	IR	A1	A2	A3	A4	IR	A5	A6	A7	A8	IR	B1

25.

E/S			B1				C1	C2	C3	C4	C5	C6
CPU	B2	IS	C1	IR	IH	IS	A9	A10	IR	A11	A12	A13

E/S												
CPU	IH	IR	B1	B2	B3	IS	IR	C1	C2	C3	C4	IR

E/S												
CPU	IS	A14	A15	IS								

26.
```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
```

```
void * factorial (void * dato) {

  long long int resultado= 1;
  int num, n= atoi((char *) dato);

  for (num= 2; num<= n; num++)  {
    resultado= resultado* num;
    printf ("Factorial de %d, resultado parcial %lld\n", n, resultado);
    sleep (random() %3);
  }
  printf ("El factorial de %d es %lld\n", n, resultado);
  pthread_exit(NULL);
}

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

  pthread_t hilo1, hilo2;

  if (argc!= 3) {
    printf ("Indica dos numeros enteros.\n");
    exit(0);
  }

  pthread_create (&hilo1, NULL, factorial, (void *) argv[1]);
  pthread_create (&hilo2, NULL, factorial, (void *) argv[2]);

  pthread_join (hilo1, NULL);
  pthread_join (hilo2, NULL);

  printf ("Factoriales calculados\n");
  return 0;
}
```

27.

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

void * factorial (void * dato) {

  long long int resultado= 1;
  int num, n= atoi((char *) dato);

  for (num= 2; num<= n; num++)  {
    resultado= resultado* num;
    printf ("Factorial de %d, resultado parcial %lld\n", n, resultado);
    sleep (random() %3);
  }
  printf ("El factorial de %d es %lld\n", n, resultado);
  pthread_exit (NULL);
}

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

  pthread_t hilo[argc-1];
  int param;

  if (argc< 2) {
    printf ("Indica los numeros de los que quieres obtener su factorial.\n");
    exit (0);
```

```
  }

  for (param= 0; param< argc-1; param++)
    pthread_create (&hilo[param], NULL, factorial,
                    (void *) argv[param+1]);

  for (param= 0; param< argc-1; param++) {
    pthread_join (hilo[param], NULL);
    printf ("Hilo %d finalizado\n", param+1);
  }

  printf ("Factoriales calculados\n");
  return 0;
}
```

28.
```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#define MAXLON 100

void *imprimecadena (void *dato) {

  int pos;
  char *cadena= (char *) dato;

  for (pos= 0; cadena[pos]!= '\0'; pos++) {
    printf ("%c", cadena[pos]);
    fflush (stdout);
    sleep  (random() %3);
  }
  pthread_exit(NULL);
}

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

  pthread_t hilo1, hilo2;
  char cadena1[MAXLON], cadena2[MAXLON];

  printf ("Introduce la cadena 1\n"); gets (cadena1);
  printf ("Introduce la cadena 2\n"); gets (cadena2);

  pthread_create (&hilo1, NULL, imprimecadena, (void *) cadena1);
  pthread_create (&hilo2, NULL, imprimecadena, (void *) cadena2);

  pthread_join (hilo1, NULL);
  pthread_join (hilo2, NULL);

  printf ("\n");
  return 0;
}
```

29.
```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
#define MAXLON 1000

void *cuenta (void *dato) {

  int pos, cont= 0, leidos;
  char *nombre= (char *) dato, cadena[MAXLON];
  int fd;

  fd= open (nombre, O_RDONLY);
  while ((leidos= read (fd, cadena, MAXLON))!= 0)
    for (pos= 0; pos< leidos; pos++)
      if ((cadena[pos]== 'a') || (cadena[pos]== 'A'))
        cont++;
  printf ("Fichero %s: %d caracteres 'a' o 'A' encontrados\n", nombre, cont);
  close (fd);
  pthread_exit (NULL);
}

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

  pthread_t hilo;

  if (argc!= 2) {
    printf ("Indica el nombre de un fichero.\n");
    exit(0);
  }
  pthread_create (&hilo, NULL, cuenta, (void *) argv[1]);
  pthread_join   (hilo,  NULL);

  return 0;
}
```

30.

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

#define MAXLON 1000

void *cuenta (void *dato) {

  int pos, cont= 0, leidos;
  char *nombre= (char *) dato, cadena[MAXLON];
  int fd;

  fd= open (nombre, O_RDONLY);
  while ((leidos= read (fd, cadena, MAXLON))!= 0)
    for (pos= 0; pos< leidos; pos++)
      if ((cadena[pos]== 'a') || (cadena[pos]== 'A'))
        cont++;
  printf ("Fichero %s: %d caracteres 'a' o 'A' encontrados\n", nombre, cont);
  close (fd);
  pthread_exit (NULL);
}
```

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

  pthread_t hilo[argc-1];
  int param;

  if (argc< 2) {
    printf ("Indica el nombre de uno o varios ficheros.\n");
    exit(0);
  }

  for (param= 0; param< argc-1; param++)
    pthread_create (&hilo[param], NULL, cuenta,
                    (void *) argv[param+1]);

  for (param= 0; param< argc-1; param++)
    pthread_join(hilo[param], NULL);

  return 0;
}
```

31.

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

#define MAXLON 1000

typedef struct {
  char *nombre;
  int cont;
} info;

void *cuenta (void *dato) {

  int pos, cont= 0, leidos;
  info *misdatos= (info *) dato;
  char cadena[MAXLON];
  int fd;

  fd= open (misdatos->nombre, O_RDONLY);
  while ((leidos= read (fd, cadena, MAXLON))!= 0)
    for (pos= 0; pos< leidos; pos++)
      if ((cadena[pos]== 'a') || (cadena[pos]== 'A')) cont++;
  misdatos->cont= cont;
  close (fd);
  pthread_exit (NULL);
}


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

  pthread_t vectorhilos[argc-1];
  info vectordatos[argc-1];
  int param;

  if (argc< 2) {
    printf ("Indica el nombre de uno o varios ficheros.\n"); exit(0);
  }
```

```
  for (param= 0; param< argc-1; param++) {
    vectordatos[param].nombre= argv[param+1];
    vectordatos[param].cont=   0;
    pthread_create (&vectorhilos[param], NULL, cuenta,
                    (void *) &vectordatos[param]);
  }

  for (param= 0; param< argc-1; param++)
    pthread_join (vectorhilos[param], NULL);

  for (param= 0; param< argc-1; param++)
    printf ("Fichero %s: %d caracteres 'a' o 'A' encontrados\n",
            vectordatos[param].nombre, vectordatos[param].cont);

  return 0;
}
```

32.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>

#define K 50

typedef struct {
  char *nombre;
  int  inicio;
  int  encontrados;
} info;

void *busca (void *datos) {

  int pos, cont= 0, leidos, fd;
  info* datoshilo= (info *) datos;
  char cadena[K];

  sleep (random() %3);
  fd= open (datoshilo->nombre, O_RDONLY);
  lseek (fd, datoshilo->inicio, SEEK_SET);
  leidos= read (fd, cadena, K);
  for (pos= 0; pos< leidos; pos++)
    if (cadena[pos]== ' ')
      cont++;
  close (fd);
  datoshilo->encontrados= cont;
  pthread_exit (NULL);
}

void cuentaEspacios (char *nombre){

  int longitud, pos, nHilos, fd, total= 0;
  pthread_t *vectorhilos;
  info *vectordatos;

  /* averigua la  longitud del fichero */
  fd= open (nombre, O_RDONLY);
```

```
  longitud= lseek (fd, 0, SEEK_END);
  close (fd);

  /* determina el numero de hilos a crear */
  nHilos= longitud/K;
  if ((longitud% K)!= 0)
    nHilos++;
  vectorhilos= (pthread_t *) malloc (nHilos*sizeof(pthread_t));
  vectordatos=      (info *) malloc (nHilos*sizeof(info));

  for (pos= 0; pos< nHilos; pos++) {
    vectordatos[pos].nombre= nombre;
    vectordatos[pos].inicio= pos* K;
    vectordatos[pos].encontrados= 0;
    pthread_create (&vectorhilos[pos], NULL, busca,
                    (void *) &vectordatos[pos]);
  }

  for (pos= 0; pos< nHilos; pos++) {
    pthread_join (vectorhilos[pos], NULL);
    total+= vectordatos[pos].encontrados;
  }

  printf ("Fichero %s: %d espacios en blanco encontrados\n", nombre, total);
  free (vectorhilos);
  free (vectordatos);
}

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

  int numfich;

  for (numfich= 1; numfich< argc; numfich++)
    if (fork()== 0) {
      cuentaEspacios (argv[numfich]);
      exit(0);
    }

  for (numfich= 1; numfich< argc; numfich++)
    wait(NULL);

  return 0;
}
```

33.

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

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

  int tubo[2];
  time_t ini= time(NULL), fin;

  if (fork()== 0){
    pipe (tubo);
    if (fork()== 0){
      dup2   (tubo[1],STDOUT_FILENO);
      close  (tubo[0]);
```

```
      close  (tubo[1]);
      execlp ("ls","ls",NULL);
    }else{
      dup2   (tubo[0],STDIN_FILENO);
      close  (tubo[0]);
      close  (tubo[1]);
      execlp ("wc","wc","-l",NULL);
    }
  } else {
    wait (NULL);
    fin= time (NULL);
    printf ("Tiempo invertido: %ld segundos\n", fin-ini);
  }

  return 0;
}
```

34. El proceso que ejecuta el comando *ls* redirige la salida estandar utilizando el descriptor de lectura de la tubería, en lugar del de escritura. Esto produce un error. A su vez, el proceso que ejecuta la orden *sort* redirige el descriptor de lectura estandard utilizando el descriptor de escritura de la tubería. También produce un error.

    Solución, cambiar las líneas siguientes:

    ```
    dup2(tubo[1], STDIN_FILENO); por dup2(tubo[0], STDIN_FILENO);
    dup2(tubo[0], STDOUT_FILENO); por dup2(tubo[1], STDOUT_FILENO);
    ```

    Y además, las dos llamadas a *close* del proceso que va a cambiar su imagen con el comando *sort*, ponerlas entre *dup2* y *execlp* para evitar que, cuando vacíe la tubería habiendo terminado el *ls*, se bloquee por tener él mismo el descriptor de escritura de la tubería abierto cuando se produce el cambio de imagen.

35. El problema que se produce es debido a que el proceso inicial tiene acceso a la tubería y no cierra el descriptor de escritura. En consecuencia, el proceso que cambia su imagen al comando *wc* (que lee de la tubería) pasa a bloqueado al vaciarla, aún cuando haya terminado el que cambia su imagen al comando *ls* (que es el que realmente escribe en la tubería).

    Para solucionarlo, por ejemplo, y ya que el proceso inicial no va a utilizar la tubería, mover la orden *pipe* justo después del primer *if* manteniendo así el acceso a los otros dos procesos pero evitando que el proceso inicial tenga acceso a la tubería.

36. El programa pretende realizar el *sort* de un fichero. Para ello escribe, en primer lugar, el contenido del fichero en una tubería. Es en este punto donde el proceso puede pasar a estado de bloqueo si se llena la tubería, no saliendo de este estado al no haber ningún proceso que lea de ella y, en consecuencia, la vacíe. En el caso de que la tubería tuviese capacidad para almacenar el

contenido completo del fichero, la ejecución continuaría y se mostraría el resultado de ejecutar la orden *sort* sobre el contenido del fichero.

37. El flujo de información de la tubería se realiza desde el proceso *ls*, que actúa como proceso hijo y escribe en la tubería, hacia el proceso *wc* que actúa como proceso padre y lee de la tubería.

    Si la cantidad total de información escrita en la tubería, resultado de la ejecución del comando *ls*, es inferior al tamaño de la tubería, y sólo en este caso, no habrá problema alguno y se podrá observar el resultado de la ejecución de *ls | wc -l*.

    Sin embargo, en cuanto la cantidad de información superase la capacidad de la tubería, entonces el proceso que escribe pasaría al estado de bloqueado, del cual no saldría ya que el único proceso que podría leer datos de la tubería, el proceso padre, está también en estado de bloqueo como consecuencia de la ejecución de la orden *wait(NULL)*.

38.
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <wait.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

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

  int tubo1[2], tubo2[2], mifichero;

  pipe(tubo1);
  pipe(tubo2);

  if (fork()== 0) {
    if (fork()== 0) {
      dup2  (tubo1[1], STDOUT_FILENO);
      close (tubo1[0]);
      close (tubo2[0]);
      close (tubo2[1]);
      close (tubo1[1]);
      execlp ("paste", "paste", argv[1], argv[2], NULL);
    } else {
      dup2  (tubo1[0], STDIN_FILENO);
      dup2  (tubo2[1], STDOUT_FILENO);
      close (tubo1[1]);
      close (tubo2[0]);
      close (tubo1[0]);
      close (tubo2[1]);
      execlp ("sort", "sort", NULL);
    }
  } else {
    dup2  (tubo2[0], STDIN_FILENO);
    close (tubo1[0]);
    close (tubo1[1]);
    close (tubo2[1]);
    close (tubo2[0]);
    mifichero= open(argv[3], O_WRONLY|O_CREAT);
    dup2 (mifichero, STDOUT_FILENO);
```

```
    close (mifichero);
    execlp ("nl", "nl", NULL);
  }

  return 0;
}
```

39.
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <wait.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

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

  int tubo1[2], mifichero;

  pipe(tubo1);

  if (fork()== 0) {
    dup2  (tubo1[1], STDOUT_FILENO);
    close (tubo1[0]);
    close (tubo1[1]);
    execlp ("grep", "grep", argv[1], argv[2], NULL);
  } else
    if (fork()== 0) {
      dup2  (tubo1[1], STDOUT_FILENO);
      close (tubo1[0]);
      close (tubo1[1]);
      execlp ("grep", "grep", argv[1], argv[3], NULL);
    } else {
      dup2  (tubo1[0], STDIN_FILENO);
      close (tubo1[0]);
      close (tubo1[1]);
      mifichero= creat(argv[4], 00644);
      dup2 (mifichero, STDOUT_FILENO);
      close (mifichero);
      execlp ("wc", "wc", "-m", NULL);
    }

  return 0;
}
```

40.
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <wait.h>
#include <unistd.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>

#define MAXLON 1000

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

  int tubo1[2], tubo2[2], fd;
  int nlinea;
```

```
  FILE *fich;
  char linea[MAXLON], cad1[MAXLON], cad2[MAXLON];

  pipe(tubo1);
  pipe(tubo2);

  if (fork()!= 0) {
    if (fork()!= 0) { /* Separar lineas pares e impares */
      close (tubo1[0]);
      close (tubo2[0]);
      if (!(fich= fopen (argv[1],"r"))) {
        printf ("Error al abrir el fichero %s\n",argv[1]);
        exit(0);
      }
      nlinea= 1;
      while (fgets (linea, MAXLON, fich)) {
        if (nlinea%2== 1)  /* Impar */
          write (tubo1[1], linea, strlen (linea));
        else  /* Par */
          write (tubo2[1], linea, strlen (linea));
        nlinea++;
      }
      fclose (fich);
      close (tubo1[1]);
      close (tubo2[1]);
      wait (NULL);
      wait (NULL);
    } else {  /* impares */
      dup2  (tubo1[0], STDIN_FILENO);
      close (tubo1[1]);
      close (tubo2[1]);
      close (tubo1[0]);
      close (tubo2[0]);
      sprintf (cad1, "%s.txt", argv[2]);
      fd= creat(cad1,00644);
      dup2  (fd, STDOUT_FILENO);
      close (fd);
      execlp ("grep", "grep", argv[2], NULL);
    }
  } else { /* pares */
      dup2  (tubo2[0], STDIN_FILENO);
      close (tubo1[1]);
      close (tubo2[1]);
      close (tubo1[0]);
      close (tubo2[0]);
      sprintf (cad2, "%s.txt", argv[3]);
      fd= creat (cad2,00644);
      dup2 (fd, STDOUT_FILENO);
      close (fd);
      execlp ("grep", "grep", argv[3], NULL);
  }
  return 0;
}
```

41. Es necesario que las tareas 1 y 4 las hagan procesos distintos que se ejecuten en paralelo, ya que, si las realiza el mismo proceso, la tarea 4, que es la que vaciaría la tubería 3, nunca empezará mientras no se termine la tarea 1, lo que puede producir que la tubería 3 se llene y, en consecuencia, los procesos que ejecutan el comando *grep* se bloqueen al escribir sobre la tubería llena,

dejando de leer de las tuberías 1 y 2, y produciendo a su vez que las tuberías 1 y 2 puedan llenarse y también el bloqueo del proceso que ejecuta la tarea 1 al intentar escribir sobre una tubería llena.

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

#define MAX 1000

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

  int  nlinea;
  FILE *fichero;
  char linea[MAX];
  int  tubo1[2], tubo2[2], tubo3[2];

  if (argc!=4) {
    printf("Error en el número de parametros\n");
    exit(1);
  }

  pipe(tubo1);
  pipe(tubo2);
  pipe(tubo3);

  if (fork()!=0) {
    if (fork()!= 0) {
      if (fork()!= 0) { // tarea 4
        dup2(tubo3[0], STDIN_FILENO);
        close (tubo1[0]);
        close (tubo1[1]);
        close (tubo2[0]);
        close (tubo2[1]);
        close (tubo3[0]);
        close (tubo3[1]);
        execlp ("sort", "sort", NULL);
      } else { // tarea 3
        dup2(tubo2[0], STDIN_FILENO);
        dup2(tubo3[1], STDOUT_FILENO);
        close (tubo1[0]);
        close (tubo1[1]);
        close (tubo2[0]);
        close (tubo2[1]);
        close (tubo3[0]);
        close (tubo3[1]);
        execlp ("grep", "grep", argv[3], NULL);
      }
    } else { //tarea 2
      dup2(tubo1[0], STDIN_FILENO);
      dup2(tubo3[1], STDOUT_FILENO);
      close (tubo1[0]);
      close (tubo1[1]);
      close (tubo2[0]);
      close (tubo2[1]);
      close (tubo3[0]);
      close (tubo3[1]);
      execlp ("grep", "grep", argv[2], NULL);
    }
```

```
  } else { //tarea 1
    close (tubo1[0]);
    close (tubo2[0]);
    close (tubo3[0]);
    close (tubo3[1]);
    if (!(fichero= fopen(argv[1],"r"))) {
      printf("Error al abrir el fichero %s\n",argv[1]);
      exit(2);
    }
    nlinea=0;
    while (fgets(linea,MAX,fichero)) {
      if (nlinea%2==0)
        write(tubo1[1],linea,strlen(linea));
      else
        write(tubo2[1],linea,strlen(linea));
      nlinea++;
    }
    fclose(fichero);
    close (tubo1[1]);
    close (tubo2[1]);
  }

  return 0;
}
```

42. No se da situación de interbloqueo, todos los procesos acabarán las secuencias de código. La salida por pantalla es la siguiente: 34125. Valor final de los semáforos: los cuatro semáforos están a cero.

43. Ocurre situación de interbloqueo siempre. En el código A, el proceso se bloquea en la llamada al segundo *sem_wait(&s2)* mientras que, en el código B, el proceso se bloquea en la llamada al segundo *sem_wait(&s1)*.

    Hasta alcanzar el punto de interbloqueo, las posibles salidas son 2. Inicialmente aparecen las cadenas A1 y B1, que pueden salir combinadas de cualquier manera, es decir, A1B1 ó B1A1 ó AB11 ó BA11. Después, siempre saldrá lo mismo: B2B3A2.

44.
```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <semaphore.h>

#define MAXLON 1000
int cuenta_blancos= 0;

sem_t blancos;

typedef struct {
  char* nombre;
  int cont;
} info;
```

```
void *cuenta (void *dato) {

  int pos, cont= 0, leidos;
  info *misdatos= (info *) dato;
  char cadena[MAXLON];
  int fd;

  fd= open (misdatos->nombre, O_RDONLY);
  while ((leidos= read (fd, cadena, MAXLON))!= 0)
    for (pos= 0; pos< leidos; pos++)
      if (cadena[pos]==' ') cont++;
  misdatos->cont= cont;
  sem_wait (&blancos);
  cuenta_blancos+= cont;
  sem_post (&blancos);
  close (fd);
  pthread_exit (NULL);
}

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

  pthread_t hilo[argc-1];
  info datos[argc-1];
  int numfich;

  if (argc< 2) {
    printf ("Indica el nombre de uno o varios ficheros.\n");
    exit(0);
  }

  sem_init (&blancos, 0, 1);

  for (numfich= 0; numfich< argc-1; numfich++) {
    datos[numfich].nombre= argv[numfich+1];
    datos[numfich].cont=   0;
    pthread_create (&hilo[numfich], NULL, cuenta,
                    (void *) &datos[numfich]);
  }

  for (numfich= 0; numfich< argc-1; numfich++)
    pthread_join (hilo[numfich], NULL);

  for (numfich= 0; numfich< argc-1; numfich++)
    printf ("Fichero %s: %d espacios en blanco encontrados\n",
            datos[numfich].nombre, datos[numfich].cont);

  printf ("Total de espacios en blanco: %d\n", cuenta_blancos);
  sem_destroy (&blancos);

  return 0;
}
```

45.

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

sem_t s1, s2, s3, s4, s5, s6;
```

```
void escribe (char *dato) {

  int num;

  for(num= 0; num< 5; num++){
    printf ("%s\n",dato);
    fflush (stdout);
    sleep  (random() %3);
  }
}

void *A (void *p){

  escribe  ("a1");
  sem_post (&s1);
  sem_wait (&s2);
  escribe  ("a2");
  sem_post (&s4);
  sem_wait (&s5);
  escribe  ("a3");
  sem_post (&s6);
  pthread_exit (NULL);
}

void *B (void *p){

  escribe  (" b1");
  sem_post (&s2);
  escribe  (" b2");
  sem_post (&s3);
  sem_wait (&s4);
  escribe  (" b3");
  sem_wait (&s6);
  sem_wait (&s6);
  escribe  (" b4");
  pthread_exit (NULL);
}

void *C (void *p){

  sem_wait (&s1);
  escribe  ("  c1");
  sem_wait (&s3);
  escribe  ("  c2");
  sem_post (&s5);
  escribe  ("  c3");
  sem_post (&s6);
  pthread_exit (NULL);
}

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

  pthread_t th1, th2, th3;

  sem_init (&s1, 0, 0);
  sem_init (&s2, 0, 0);
  sem_init (&s3, 0, 0);
  sem_init (&s4, 0, 0);
  sem_init (&s5, 0, 0);
  sem_init (&s6, 0, 0);
  pthread_create (&th1, NULL, A, NULL);
```

```
  pthread_create (&th2, NULL, B, NULL);
  pthread_create (&th3, NULL, C, NULL);
  pthread_join (th1, NULL);
  pthread_join (th2, NULL);
  pthread_join (th3, NULL);
  sem_destroy (&s1);
  sem_destroy (&s2);
  sem_destroy (&s3);
  sem_destroy (&s4);
  sem_destroy (&s5);
  sem_destroy (&s6);

  return 0;
}
```

46.

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

#define MAX 6

sem_t A, B, C;

void *escribirA (void *nada){

  int num;

  for (num= 0; num< MAX; num++){
    sem_wait (&A);
    printf   ("A");
    fflush   (stdout);
    sem_post (&B);
    sleep    (random() %3);
  }
  pthread_exit (NULL);
}

void *escribirB (void *nada){

  int num;

  for (num= 0; num< MAX; num++){
    sem_wait (&B);
    printf   ("B");
    fflush   (stdout);
    sem_post (&C);
    sleep    (random() %2);
  }
  pthread_exit (NULL);
}

void *escribirC (void *nada){

  int num;

  for (num= 0; num< MAX; num++){
    sem_wait (&C);
    printf   ("C");
```

```
    fflush   (stdout);
    sem_post (&A);
    sleep    (random()%2);
  }
  pthread_exit (NULL);
}

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

  pthread_t th1, th2, th3;
  srandom(time(NULL));
  sem_init (&A, 0, 1); /* habilitado */
  sem_init (&B, 0, 0); /* no habilitado */
  sem_init (&C, 0, 0); /* no habilitado */
  pthread_create (&th1, NULL, escribirA, NULL);
  pthread_create (&th2, NULL, escribirB, NULL);
  pthread_create (&th3, NULL, escribirC, NULL);
  pthread_join (th1, NULL);
  pthread_join (th2, NULL);
  pthread_join (th3, NULL);
  sem_destroy (&A);
  sem_destroy (&B);
  sem_destroy (&C);

  return 0;
}
```

47. Estos son los posibles valores de num así como los diferentes estados de los hilos:

    - num es 13, *incementa* termina, *decrementa* se bloquea en *sem_wait(&sem2)* con i a cero.
    - num es 12, *incrementa* termina, *decrementa* se bloquea en *sem_wait(&sem2)* con i a uno.
    - num es 11, *incrementa* termina, *decrementa* se bloquea en *sem_wait(&sem2)* con i a dos.
    - num es 10, *incrementa* termina, *decrementa* se bloquea en *sem_wait(&sem1)* a continuación del bucle.

48. Utilizando dos semáforos, sA y sB, con sA a cero y sB a uno, esta es la solución:

```
void *escribirA (void *p) {
  int i;

  for (i= 0; i< 5; i++) {
    sem_wait(&sA);
    printf ("A");
    fflush(NULL);
    sleep(random()%2);
    sem_post(&sB);
  }
  pthread_exit(NULL);
}
```

```
void *escribirB (void *p) {
  int i;

  for (i= 0;i< 5; i++) {
    sem_wait(&sB);
    printf ("B");
    fflush(NULL);
    sleep(random()%2);
    sem_post(&sA);
  }
  pthread_exit(NULL);
}
```

49. Los tres hilos se van a quedar bloqueados:

- *escribirA* queda bloqueado en el *sem_wait(&s1)* con $i = 2$.
- *escribirB* queda bloqueado en el primer *sem_wait(&s2)* con $j = 2$.
- *escribirC* queda bloqueado en el primer *sem_wait(&s3)* con $k = 1$.

50.
```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>
#include <stdbool.h>

#define MAX 10
#define FIN -1

int buffer[MAX];
sem_t huecos, elementos;

int generar_dato (void) { return random() %256;}
int numero_aleatorio(void) { return random() %100;}

void *productor (void *p) {

  int pos_productor= 0;
  int num, dato, n;

  n= numero_aleatorio();
  printf ("Productor con %d datos\n", n);
  for(num= 0; num< n; num++) {
    dato= generar_dato();
    sem_wait (&huecos);
    buffer[pos_productor]= dato;
    pos_productor= (pos_productor+ 1) %MAX;
    sem_post (&elementos);
  }
  sem_wait (&huecos);
  buffer[pos_productor]= FIN;
  sem_post (&elementos);

  pthread_exit (NULL);
}

void *consumidor(void *p){

  int pos_consumidor= 0, dato;
  bool continuar= true;

  while (continuar) {
    sem_wait (&elementos);
    dato= buffer[pos_consumidor];
    sem_post (&huecos);
    if (dato== FIN)
      continuar= false;
    else {
      printf ("Numero aleatorio: %d\n", dato);
      pos_consumidor= (pos_consumidor+1) %MAX;
    }
  }

  pthread_exit (NULL);
}
```

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

  pthread_t hiloproductor, hiloconsumidor;

  sem_init (&elementos, 0, 0);
  sem_init (&huecos,    0, MAX);

  pthread_create (&hiloproductor, NULL, productor,  NULL);
  pthread_create (&hiloconsumidor, NULL, consumidor,  NULL);

  pthread_join (hiloproductor, NULL);
  pthread_join (hiloconsumidor, NULL);

  sem_destroy (&huecos);
  sem_destroy (&elementos);

  return 0;
}
```

51.

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

#define MAX 10
#define FIN -1
#define NUMPROD 3

int buffer[MAX];
int pos_productor= 0;
sem_t huecos, elementos, pos_prod;

int generar_dato     (void) { return random() %256;}
int numero_aleatorio (void) { return random() %100;}

void *productor (void *p) {

  int num, dato, n;

  n= numero_aleatorio();
  for(num= 0; num< n; num++) {
    dato= generar_dato();
    sem_wait (&huecos);
    sem_wait (&pos_prod);
    buffer[pos_productor]= dato;
    pos_productor= (pos_productor+ 1) %MAX;
    sem_post (&pos_prod);
    sem_post (&elementos);
  }

  sem_wait (&huecos);
  sem_wait (&pos_prod);
  buffer[pos_productor]= FIN;
  pos_productor= (pos_productor+ 1) %MAX;
  sem_post (&pos_prod);
  sem_post (&elementos);

  pthread_exit (NULL);
}
```

```
void *consumidor(void *p){

  int pos_consumidor= 0, dato, cont_fin= 0;
  bool continuar= true;

  while (continuar) {
    sem_wait (&elementos);
    dato= buffer[pos_consumidor];
    sem_post (&huecos);
    pos_consumidor= (pos_consumidor+1) %MAX;
    if (dato== FIN) {
      cont_fin++;
      if (cont_fin== NUMPROD)
        continuar= false;
    } else
      printf ("Numero aleatorio: %d\n", dato);
  }

  pthread_exit (NULL);
}

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

  pthread_t hiloconsumidor, hiloproductor[NUMPROD];
  int num;

  sem_init (&elementos, 0, 0);
  sem_init (&huecos,    0, MAX);
  sem_init (&pos_prod,  0, 1);

  for (num= 0; num< NUMPROD; num++)
    pthread_create (&hiloproductor[num], NULL, productor,  NULL);
  pthread_create (&hiloconsumidor, NULL, consumidor, NULL);

  for (num= 0; num< NUMPROD; num++)
    pthread_join (hiloproductor[num], NULL);
  pthread_join (hiloconsumidor, NULL);

  sem_destroy (&huecos);
  sem_destroy (&elementos);
  sem_destroy (&pos_prod);

  return 0;
}
```

52.

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>
#include <math.h>
#include <stdbool.h>

#define MAX  5
#define FIN -1

int buffer[MAX];
sem_t huecos,elementos;
int inicio, fin;
```

```
/* Devuelve la suma de los dígitos del número dado */
int suma_digitos (int numero) {

  int suma= 0;

  while (numero> 0) {
    suma+= numero%10;
    numero/= 10;
  }

  return suma;
}

/* Indica si el número dado es primo */
bool es_primo (int numero) {

  int divisor;

  for (divisor= 2; divisor<= sqrt(numero); divisor++)
    if (numero%divisor== 0)
      return false;

  return true;
}

void *productor (void *param) {

  int pos_productor=0;
  int num;

  for (num= inicio; num< fin; num++)
    if (es_primo (num)) {
      sem_wait (&huecos);
      buffer[pos_productor]= num;
      pos_productor= (pos_productor+1)%MAX;
      sem_post (&elementos);
    }

  sem_wait (&huecos);
  buffer[pos_productor]= FIN;
  sem_post (&elementos);

  pthread_exit(NULL);
}

void *consumidor (void *param) {

  int pos_consumidor=0;
  int dato;
  bool fin= false;

  while (!fin) {
    sem_wait (&elementos);
    dato= buffer[pos_consumidor];
    pos_consumidor= (pos_consumidor+1)%MAX;
    sem_post (&huecos);
    if (dato== FIN)
      fin= true;
    else
      printf ("Número primo: %d; Suma de los digitos: %d\n",
               dato, suma_digitos(dato));
  }
```

```
  pthread_exit (NULL);
}

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

  pthread_t hiloproductor, hiloconsumidor;

  if (argc!= 3) {
    printf ("Indica los dos números enteros.\n");
    exit (0);
  }

  inicio= atoi (argv[1]);
  fin=    atoi (argv[2]);

  sem_init (&huecos, 0, MAX);
  sem_init (&elementos, 0, 0);

  pthread_create (&hiloproductor, NULL, productor, NULL);
  pthread_create (&hiloconsumidor, NULL, consumidor, NULL);

  pthread_join (hiloproductor, NULL);
  pthread_join (hiloconsumidor, NULL);

  sem_destroy (&huecos);
  sem_destroy (&elementos);

  return 0;
}
```

53.

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

#define MAXALU 25
#define MAXRES  4
#define RANGO  12
#define LIBRE -1
#define HORAINICIO 9

sem_t sem_tabla, sem_consultas;
int tabla[RANGO], consultas=0;

void consulta (int id) {

  int num;

  sem_wait (&sem_consultas);
  consultas++;
  if (consultas== 1)
    sem_wait(&sem_tabla);
  sem_post (&sem_consultas);

  for (num= 0; num< RANGO; num++)
    if (tabla[num]!= LIBRE)
      printf ("Consulta de %d: %d- %d ¡=¿ %d\n",
              id, num+HORAINICIO, num+HORAINICIO+1, tabla[num]);
    else
```

```
      printf ("Consulta de %d: %d- %d ¡=¿ LIBRE\n",
              id, num+HORAINICIO, num+HORAINICIO+1);

  sem_wait (&sem_consultas);
  consultas--;
  if (consultas== 0)
    sem_post (&sem_tabla);
  sem_post (&sem_consultas);

  pthread_exit (NULL);
}

void reserva (int hora, int id) {

  printf ("Solicitud de reserva de %d; %d- %d\n", id, hora, hora+1);
  sem_wait (&sem_tabla);
  if (tabla[hora-HORAINICIO]!= LIBRE)
    printf ("Denegada reserva de %d: %d- %d está ocupada\n", id, hora, hora+1);
  else {
    tabla[hora-HORAINICIO]= id;
    printf ("Reserva de %d: %d- %d\n", id, hora, hora+1);
  }
  sem_post (&sem_tabla);
}

void cancela (int id) {

  int num;
  bool hay= false;

  printf ("Solicitud de cancelación de %d\n", id);
  sem_wait (&sem_tabla);
  for (num= 0; num< RANGO; num++) {
    if (tabla[num]== id) {
      hay= true;
      tabla[num]= LIBRE;
      printf ("Cancelación de %d: %d- %d\n", id,
               num+HORAINICIO, num+HORAINICIO+1);
    }
  }
  if (hay== false)
    printf ("Denegada cancelación de %d: No tiene reservas\n", id);
  sem_post (&sem_tabla);
}

void *alumno (void *dato) {

  int id= *(int *)dato, num, prob;

  for (num= 0; num< MAXRES; num++) {
    prob= random() %100;
    if (prob< 50)
      reserva (random() %RANGO+HORAINICIO, id);
    else
      if (prob< 75)
        cancela (id);
      else
        consulta (id);
  }
  pthread_exit (NULL);
}
```

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

  pthread_t hilos[MAXALU];
  int num, dato[MAXALU];

  sem_init (&sem_tabla, 0, 1);
  sem_init (&sem_consultas, 0, 1);

  for (num= 0; num< RANGO; num++)
    tabla[num]= LIBRE;

  for (num= 0; num< MAXALU; num++) {
    sleep (random() %3); /* simula una espera aleatoria */
    dato[num]= num;
    pthread_create (&hilos[num], NULL, alumno, (void *)&dato[num]);
  }

  for (num= 0; num< MAXALU; num++)
    pthread_join (hilos[num], NULL);

  sem_destroy (&sem_tabla);
  sem_destroy (&sem_consultas);

  return 0;
}
```

54.

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

#define MAX 100

int ncochesDirB= 0, ncochesDirA= 0;
sem_t puente, dirB, dirA;

void *direccionB (void *dato) {

  int id= (int) dato;

  sleep (random() %20);

  sem_wait (&dirB);
  ncochesDirB++;
  if (ncochesDirB== 1)
    sem_wait (&puente);
  printf ("Entra en puente Dir B coche %d\n", id);
  sem_post (&dirB);

  sleep (1); /* cruzar el puente */

  sem_wait (&dirB);
  ncochesDirB--;
  printf ("Sale del puente Dir B coche %d\n", id);
  if (ncochesDirB== 0)
    sem_post (&puente);
  sem_post (&dirB);

  pthread_exit (NULL);
}
```

```
void *direccionA (void *dato) {

  int id= (int) dato;

  sleep (random() %20);

  sem_wait (&dirA);
  ncochesDirA++;
  if (ncochesDirA== 1)
      sem_wait (&puente);
  printf ("Entra en puente Dir A coche %d\n", id);
  sem_post (&dirA);

  sleep (1); /* cruzar el puente */

  sem_wait (&dirA);
  ncochesDirA--;
  printf ("Sale del puente Dir A coche %d\n", id);
  if (ncochesDirA== 0)
    sem_post (&puente);
  sem_post (&dirA);

  pthread_exit (NULL);
}

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

  pthread_t hilos[MAX];
  int num;

  sem_init (&dirA, 0, 1);
  sem_init (&dirB, 0, 1);
  sem_init (&puente, 0, 1);

  for (num= 0; num< MAX; num+= 2) {
    sleep (random() %3);
    pthread_create (&hilos[num],  NULL,direccionB,(void*)num);
    pthread_create (&hilos[num+1],NULL,direccionA,(void*)(num+1));
  }

  for (num= 0; num< MAX; num++)
    pthread_join (hilos[num], NULL);

  sem_destroy (&dirA);
  sem_destroy (&dirB);
  sem_destroy (&puente);

  return 0;
}
```

55. En primer lugar se calcula lo que ocupa la FAT, que es el tamaño del enlace (32 bits) por el número de entradas de la tabla que, a su vez, es el tamaño del disco dividido por el tamaño del agrupamiento y que en este problema son $20GB/(16*512bytes) = 20*2^{17}$ entradas. Luego la tabla ocupa $20*2^{17}*32bits = 20*2^{19}$ bytes.

    Si se divide lo que ocupa la tabla por el tamaño del agrupamiento se obtiene el número de agrupamientos que ocupa la tabla: $20*2^{19}/(16*512) = 20*$

$2^6 = 1280$ agrupamientos, que multiplicado por 16, que es el número de sectores por agrupamiento, se obtiene el número total de sectores que es 20480.

56. En un sistema de archivos FAT, los bloques se asignan como una lista enlazada que finaliza con la posición fin de lista EOF. Es posible recuperar datos utilizando los enlaces partiendo desde esa posición EOF hacia atrás. La reconstrucción de la lista de bloques será:

$$14 \rightarrow 15 \rightarrow 12 \rightarrow 13 \rightarrow 10 \rightarrow 11 \rightarrow EOF$$

La información de esa lista de bloques será:

$$sigan \rightarrow buscando \rightarrow yo \rightarrow no \rightarrow he \rightarrow sido \rightarrow EOF$$

57. Se trata de un sistema de archivos FAT en el que un fichero de 160KB ocupa 10 entradas. Como cada sector de disco contiene 512 bytes, el fichero ocupa 160 KB / 512 bytes = 320 sectores. Como dicho fichero ocupa 10 entradas, entonces cada agrupamiento estará compuesto por 32 sectores como mínimo.

58. Como el sistema es FAT16 puede direccionar un total de 2^{16} bloques * 1KB = 64MB.

    Si la partición es de 2GB, el tamaño de bloque debería ser como mínimo el resultado de dividir el tamaño de la partición entre el número máximo de bloques, es decir, $2GB/2^{16}bloques = 32KB$ por bloque.

59. El máximo será el número de entradas de la FAT por el tamaño del bloque, es decir, $2^{16} * 4KB = 256MB$.

    Si la partición es de 8GB, no es adecuado el tamaño de bloque ya que sólo utilizaría 256MB de esos 8GB.

    El tamaño de bloque adecuado se puede obtener dividiendo el tamaño de la partición entre el número de índices de la FAT, es decir, $8GB/2^{16} = 128KB$ por bloque.

    Para saber cuántos bloques ocupa la FAT, calculamos el tamaño de esta que es $2^{16}*2bytes = 128KB$, el cual coincide con el tamaño de justo un bloque.

60. En un sistema FAT16 tenemos 2^{16} posibles índices. Por lo tanto, el espacio máximo es el número de índices por el tamaño del bloque, es decir, $2^{16} * 1KB = 64MB$. Como la partición es de 8GB, entonces el espacio inutilizado es la diferencia, es decir, $8GB - 64MB$.

    En el caso del sistema de ficheros basado en nodos-i, el número de enlaces que caben en un bloque se calcula dividiendo el tamaño del bloque por el

tamaño del enlace, es decir, $1KB/16Bytes = 64$ enlaces por bloque. Como el fichero ocupa 131KB, se utilizarán los dos índices directos (con los que se direcciona un total de 2KB), los dos indirectos simples (con los que se direcciona un total de 128KB) y el primer índice indirecto doble (para direccionar el bloque de datos que falta).

Si el bloque no legible es un bloque de datos sólo se perdería dicho bloque, es decir, 1KB. Sin embargo, en el caso de no poder leer un bloque de enlaces, si se trata del bloque apuntado por el índice indirecto simple, se perderían 64 bloques de datos. Si se trata de uno de los dos bloques de índices utilizados por el índice indirecto doble, se perderá un sólo bloque de datos.

61. Si el fichero tiene 1548 KB, y en cada bloque caben 2KB, utiliza 1548KB / 2KB = 774 bloques de datos.

    Los cinco índices directos del nodo-i permiten acceder a cinco bloques de datos, y no se utilizan bloques para almacenar enlaces, por lo que quedan por indexar 769 bloques de datos.

    Cada índice indirecto simple permite indexar 256 bloques de datos ya que si el tamaño de bloque son 2KB y cada índice ocupa 64 bits, es decir, 8 bytes, tenemos que 2KB / 8bytes = 256 índices. Al mismo tiempo, cada índice indirecto simple consume un bloque para almacenar enlaces.

    Con los tres enlaces indirectos simples que tiene el nodo-i, indexamos 3*256= 768 bloques de datos del fichero, consume 3 bloques para almacenar enlaces, y aún queda un bloque de datos del fichero por indexar.

    Para indexar el bloque de datos que falta, se utilizará el primer índice indirecto doble, que utilizará 2 bloques para almacenar enlaces (el mínimo para indexar un bloque).

    Así, en total, se utilizan 3 + 2 = 5 bloques para almacenar enlaces.

62. El número de índices que caben en un bloque se obtiene dividiendo el tamaño del bloque entre el tamaño del índice, es decir, 4KB/32bits= 1024 índices.

    Los índices directos del nodo-i no consumen bloques de índices. Cada índice indirecto simple consume un bloque de índices, y cada índice indirecto doble consume 1+1024 bloques de índices. Si el fichero ocupa el máximo tamaño posible, el número de bloques que contendrán índices para ese fichero será: $2 + 1 + 1024$ bloques.

63.
```
void listado(char nomdir[]) {
  DIR    *d;
  struct dirent *entrada;
  char   *ruta;
  struct stat datos, datos_l;

  d= opendir(nomdir);
  if (d== NULL)
    printf("Error al abrir el directorio\n");
```

```
  else {
    entrada= readdir(d);
    while (entrada!= NULL) {
      ruta= malloc(strlen(nomdir)+strlen(entrada->d_name)+2);
      sprintf(ruta,"%s/%s", nomdir, entrada->d_name);
      stat (ruta,&datos);
      lstat(ruta,&datos_l);
      if(S_ISDIR(datos.st_mode) && S_ISLNK(datos_l.st_mode))
        printf("%s\n", ruta);
      free(ruta);
      entrada= readdir(d);
    }
    closedir(d);
  }
}
```

64.
```
void listado(char nomdir[]) {
  DIR    *d;
  struct dirent *entrada;
  char   *ruta, *ruta_zip;
  struct stat datos, datos_zip;

  d= opendir(nomdir);
  if (d== NULL)
    printf("Error al abrir el directorio\n");
  else {
    entrada= readdir(d);
    while (entrada!= NULL) {
      ruta= malloc(strlen(nomdir)+strlen(entrada->d_name)+2);
      sprintf(ruta,"%s/%s", nomdir, entrada->d_name);
      lstat(ruta,&datos);
      if(S_ISREG(datos.st_mode)){
        ruta_zip= malloc(strlen(entrada->d_name)+10);
        sprintf(ruta_zip,"/tmp/%s.zip", entrada->d_name);
        if (fork()== 0)
          execlp("zip", "zip", ruta_zip, ruta, NULL);
        else {
          wait(NULL);
          stat(ruta_zip, &datos_zip);
          printf("Datos %ld\n", datos.st_size - datos_zip.st_size);
        }
        free(ruta_zip);
      }
      free(ruta);
      entrada= readdir(d);
    }
    closedir(d);
  }
}
```

65.
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>
#include <string.h>

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

  DIR *dir;
```

```
  struct dirent *entrada;
  struct stat datos;
  char * ruta;

  if (argc!= 2) {
    printf ("Debe especificar un directorio\n");
    exit (1);
  } else
    printf ("El directorio especificado existe y es accesible\n");

  dir= opendir(argv[1]);
  if (dir== NULL) {
    printf ("El directorio especificado no existe o no tienes permisos.\n");
    exit (1);
  }

  entrada= readdir(dir);
  while (entrada!= NULL) {
    ruta= malloc(strlen (argv[1])+ strlen (entrada->d_name)+ 2);
    sprintf (ruta," %s/ %s", argv[1], entrada->d_name);
    lstat (ruta, &datos);
    if (S_ISDIR (datos.st_mode))
       if (rmdir (ruta)== 0)
         printf (" %s se ha borrado\n", ruta);
       else
         printf (" %s NO se ha borrado\n", ruta);
    else
      if (unlink (ruta)== 0)
        printf (" %s se ha borrado\n", ruta);
      else
        printf (" %s NO se ha borrado\n", ruta);
    free (ruta);
    entrada= readdir(dir);
  }
  closedir(dir);

  return 0;
}
```

66.

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

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

  struct stat datos;
  int num;

  if (argc< 2) {
    printf ("Debe especificar al menos un fichero o directorio\n");
    exit (1);
  }

  for (num= 1; num< argc; num++) {
    if (lstat (argv[num], &datos)!= 0) {
       printf ("Error al ejecutar stat, %s no existe\n", argv[num]);
    } else {
      if (S_ISREG (datos.st_mode)) {
        printf (" %s es un fichero\n", argv[num]);
```

```
        if (unlink (argv[num])== 0) {
          printf ("%s se ha borrado\n", argv[num]);
          if (datos.st_nlink> 1)
            printf ("Pero no se han liberado bloques\n");
          else
            printf ("Se han liberado %d bloques\n", (int)datos.st_blocks);
       } else
          printf ("%s NO se ha borrado\n", argv[num]);
      } else {
          if (S_ISLNK (datos.st_mode)) {
            printf ("%s es un enlace simbolico\n", argv[num]);
            if (unlink (argv[num])== 0)
              printf ("%s se ha borrado y se han liberado %d bloques\n",
                      argv[num], (int)datos.st_blocks);
            else
              printf ("%s NO se ha borrado\n", argv[num]);
          } else {
              if (S_ISDIR (datos.st_mode)) {
                printf ("%s es un directorio\n", argv[num]);
                if (rmdir (argv[num])== 0)
                  printf ("%s se ha borrado y se han liberado %d bloques\n",
                          argv[num], (int)datos.st_blocks);
                else
                  printf ("%s NO se ha borrado\n", argv[num]);
              }
          }
       }
    }
  }
  return 0;
}
```

67.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

void recorre (char *nombredir) {

  DIR *dir;
  struct dirent *entrada;
  struct stat datos;
  char *ruta;

  dir= opendir(nombredir);
  if (dir== NULL) {
    printf ("Error al abrir el directorio %s\n", nombredir);
    return;
  }

  entrada= readdir(dir);
  while (entrada!= NULL) {
    if (strcmp (entrada->d_name,".") &&
        strcmp (entrada->d_name,"..")) {
      ruta= malloc (strlen (nombredir)+ strlen (entrada->d_name)+ 2);
      sprintf (ruta, "%s/%s", nombredir, entrada->d_name);
      lstat (ruta, &datos);
      if (S_ISDIR (datos.st_mode))
```

```
        recorre (ruta);  /* recorrido recursivo */
      else
        if (S_ISREG (datos.st_mode) && (datos.st_uid== getuid()) &&
            (time(NULL)-datos.st_atime> 60*60))
          printf ("%s\n",ruta);
      free (ruta);
    }
    entrada= readdir(dir);
  }
  closedir (dir);
}

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

  if (argc!= 2)
    printf("Debe especificar un directorio\n");
  else
    recorre (argv[1]);

  return 0;
}
```

68.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

int total= 0, longitud= 0, *nodos_i= NULL;

void anyade (int nodo_i) {

  if (longitud%10== 0)
    nodos_i= (int *) realloc (nodos_i, (longitud+10)* sizeof (int));
  nodos_i[longitud]= nodo_i;
  longitud++;
}

bool esta(int nodo_i) {

  int num;

  for (num= 0; num< longitud; num++)
    if (nodos_i[num]== nodo_i)
      return true;

  return false;
}

void recorre (char *nombredir) {

  DIR *dir;
  struct dirent *entrada;
  struct stat datos;
  char *ruta;

  dir= opendir(nombredir);
```

```
  if (dir== NULL) {
    printf ("Error al abrir el directorio %s\n", nombredir);
    return;
  }

  entrada= readdir(dir);
  while (entrada!= NULL) {
    if (strcmp(entrada->d_name,".") && strcmp(entrada->d_name,"..")) {
      ruta= malloc(strlen (nombredir)+ strlen(entrada->d_name)+ 2);
      sprintf (ruta,"%s/%s", nombredir, entrada->d_name);
      lstat (ruta, &datos);
      if (S_ISDIR (datos.st_mode))
        recorre (ruta);  /* llamada recursiva */
      total+= datos.st_size;
      if (datos.st_nlink> 1) {
        if (esta (datos.st_ino)== false)
         anyade (datos.st_ino);
        else
          total-= datos.st_size;  /* repetido */
      }
      free (ruta);
    }
    entrada= readdir (dir);
  }
  closedir (dir);
}

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

  if (argc!= 2) {
    printf ("Debe especificar un directorio\n");
    exit (1);
  }

  recorre (argv[1]);
  printf ("Total en bytes: %d\n", total);

  return 0;
}
```

69.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

void recorre (char *nombredir, char *nombrefile, int nodo_i) {

  DIR *dir;
  struct dirent *entrada;
  struct stat datos;
  char *ruta;

  dir= opendir(nombredir);
  if (dir== NULL) {
    printf ("Error al abrir el directorio %s\n", nombredir);
    return;
  }
```

```
  entrada= readdir(dir);
  while (entrada!= NULL) {
    if (strcmp (entrada->d_name,".") &&
        strcmp (entrada->d_name,"..")) {
      ruta= malloc (strlen (nombredir)+ strlen (entrada->d_name)+ 2);
      sprintf (ruta, "%s/%s", nombredir, entrada->d_name);
      lstat (ruta, &datos);
      if (S_ISDIR (datos.st_mode))
        recorre (ruta, nombrefile, nodo_i);  /* llamada recursiva */
      else
        if (S_ISREG (datos.st_mode) &&
            (strcmp (ruta, nombrefile)!= 0) &&
            (datos.st_ino== nodo_i)) {
          unlink  (ruta);
          symlink (nombrefile, ruta);
        }
      free (ruta);
    }
    entrada= readdir (dir);
  }
  closedir (dir);
}

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

  struct stat datos;
  if (argc!= 3)
    printf("Debe especificar un fichero y un directorio\n");
  else {
    lstat (argv[1], &datos);
    if (datos.st_nlink> 1)
      recorre (argv[2], argv[1], datos.st_ino);
  }
  return 0;
}
```

70.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <string.h>
#include <stdbool.h>

bool tiene_extension (char ext[], char nombre[]) {
  return (!strcmp(ext, &nombre[strlen(nombre)-strlen(ext)]));
}

void crear_enlace (char ruta[], char nombre_entrada[], char ext[]) {

  char *ruta2;
  struct stat datos;

  ruta2= (char *) malloc(strlen("/resultado/")+ strlen(ext)+
                         strlen(nombre_entrada)+2);
  sprintf (ruta2, "/resultado/%s/%s", ext, nombre_entrada);
  if (lstat (ruta2, &datos)!= 0)  /* no existe enlace simbolico */
    symlink (ruta, ruta2);
  free (ruta2);
}
```

```
void recorre(char *nombredir) {

  DIR *dir;
  struct dirent *entrada;
  struct stat datos;
  char *ruta;

  dir= opendir(nombredir);
  if (dir== NULL) {
    printf ("Error al abrir el directorio %s\n", nombredir);
    return;
  }

  entrada= readdir(dir);
  while (entrada!= NULL) {
    if (strcmp (entrada->d_name,".") &&
        strcmp (entrada->d_name,"..")) {
      ruta= (char*) malloc (strlen (nombredir)+
                            strlen (entrada->d_name)+ 2);
      sprintf (ruta, "%s/%s", nombredir, entrada->d_name);
      lstat (ruta, &datos);
      if (S_ISDIR (datos.st_mode))
        recorre (ruta);
      else
        if (S_ISREG (datos.st_mode)) {
          if (tiene_extension (".jpeg", entrada->d_name))
            crear_enlace (ruta, entrada->d_name, "jpeg");
          else
            if (tiene_extension (".mp3", entrada->d_name))
              crear_enlace (ruta, entrada->d_name, "mp3");
            else
              if (tiene_extension (".avi", entrada->d_name))
                crear_enlace (ruta, entrada->d_name, "avi");
        }
      free (ruta);
    }
    entrada= readdir (dir);
  }
  closedir (dir);
}

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

  recorre("/media");
  return 0;
}
```

71. Si 1GB es el máximo de memoria que se puede direccionar, la dirección lógica consta de 30 bits. Como el tamaño de página es de 16KB, de los 30 bits de la dirección lógica, 14 bits se utilizan para direccionar el contenido de la página, y el resto, 16, para direccionar las páginas, por lo que el número total de páginas que un proceso puede tener asignadas es de 2^{16}.

    Si la tabla de primer nivel contiene 1024 entradas, esto implica que su codificación consume 10 bits en la dirección lógica, por lo que quedan 6 bits para la tabla de segundo nivel, lo que permite direccionar 2^6 páginas, es decir, 64 páginas. Como el proceso requiere 6401 páginas, se obtiene que necesita 101 tablas de segundo nivel.

72. Si el tamaño de página son 2KB, esto implica que de los 22 bits de la dirección lógica, 11 bits se utilizan para direccionar la página, por lo que quedan 11 bits para codificar la tabla de páginas.

    Si se utiliza una tabla de páginas de un sólo nivel, su tamaño es el número de entradas de la tabla por el tamaño de la entrada, es decir, $2^{11} * 8 = 2^{14}$ bytes.

    Si se utiliza una tabla de páginas de dos niveles, 6 bits de la dirección lógica se emplean para el primer nivel ya que el enunciado dice que 5 bits son para el segundo nivel. Entonces, cada tabla de segundo nivel direcciona hasta 2^5 páginas, es decir 32 páginas de 2KB cada una (total 64KB). Como el proceso requiere 90KB, hacen falta 2 tablas de segundo nivel. Así, el consumo es el de una tabla de primer nivel más el de dos de segundo nivel, esto es, $2^6 * 8 + 2 * 2^5 * 8 = 2^{10}$ bytes.

    Por lo tanto, el ahorro al utilizar una tabla de páginas de dos niveles es de $2^{14} - 2^{10}$ bytes.

73. Como el tamaño de la página es de 4 KB, de los 20 bits de dirección lógica, 12 bits son para direccionar el contenido de la página y, por lo tanto, 8 bits son para direccionar las páginas.

    Si el sistema tiene una tabla de un solo nivel, la tabla tiene 256 entradas, y su coste de almacenamiento es de $256 * 16 = 2^{12}$ bytes.

    Si el sistema tiene una tabla de dos niveles, la tabla de primer nivel tiene 2^4 entradas, 16 entradas, al igual que la de segundo nivel. Si un proceso utiliza 192 KB, se obtiene que necesita $192KB/4KB = 48$ páginas, y se emplean 3 tablas de segundo nivel. Ahora el consumo es de una tabla de primer nivel y 3 de segundo, es decir, $2^4 * 16 + 3 * 2^4 * 16 = 2^{10}$ bytes.

    Por lo tanto, el ahorro es $2^{12} - 2^{10}$ bytes.

74. Si el tamaño de página son 2KB, esto implica que de los 22 bits de la dirección lógica, 11 bits se utilizan para direccionar la página, por lo que quedan 11 bits para codificar la tabla de páginas. Por lo tanto, el número máximo de páginas que puede tener asignado un proceso es 2^{11} páginas.

    Con una tabla de páginas de dos niveles, la tabla de primer nivel tiene 2^3 entradas, y la de segundo nivel tiene 2^8 entradas, es decir, 256 entradas. Por lo que si un proceso utiliza 1000 páginas, entonces son necesarias 4 tablas de segundo nivel.

75. En general, utilizar tablas de páginas de dos niveles suele producir un ahorro en el consumo de memoria comparado con utilizar una tabla de un solo nivel. Sin embargo, en el caso de que el proceso requiriera, por ejemplo, la totalidad del espacio de memoria, ya no sería cierto.

    Por ejemplo, suponer que se disponen de 4 bits para direccionar páginas, donde en un sistema de paginación de 2 niveles, 2 bits son para el primer

nivel y 2 bits son para el segundo nivel. En una tabla de un solo nivel hay entonces 16 entradas. En una tabla de dos niveles, hay 4 entradas para la tabla de primer nivel y, por lo tanto, cuatro tablas de segundo nivel donde cada una de estas tablas consta de 4 entradas. Si cada entrada a la tabla de páginas requiere n bytes, el coste de almacenar la tabla de páginas de un nivel es $16 * n$ bytes, que es menor que el coste de almacenar la tabla de páginas de dos niveles que es $(4 + 4 * 4) * n$ bytes. Por lo tanto, en este caso la tabla de páginas de dos niveles consume más memoria que la de un nivel.

Respecto al segundo punto, la afirmación no es correcta. El número de páginas que se pueden asignar a un proceso es independiente de si el sistema de paginación utiliza un nivel o más, ya que en verdad depende del número de bits de la dirección lógica que se utilizan para direccionar las páginas. Siguiendo con el ejemplo anterior, ese número de bits es 4, y el número máximo de páginas es por lo tanto de 16.

76. Si se direcciona 1GB (2^{30} bytes), entonces la dirección lógica es de 30 bits. Si la página es de 32KB (2^{15} bytes), y cada palabra son 2 bytes, harán falta 14 bits para direccionar el contenido de una página. En consecuencia, quedan 16 bits de la dirección lógica para la tabla de páginas. Si el proceso requiere 90MB, necesita 90MB/32KB = 2880 páginas. Si se utiliza una única tabla de páginas, el consumo de memoria será el tamaño de la tabla por el tamaño de la entrada, es decir, $2^{16} * 16 = 2^{20}$ bytes. Si se utiliza una tabla de páginas de dos niveles, donde 8 bits son para el primer nivel y otros 8 bits para el segundo, el proceso utilizará la tabla de primer nivel más 12 de segundo, ya que cada tabla de segundo nivel apunta a 256 páginas y el proceso requiere un total de 2880. El consumo en este caso es de $2^8 * 16 + 12 * 2^8 * 16 = 13 * 2^{12}$ bytes, y el ahorro es la diferencia de los costes obtenidos, es decir, $2^{20} - 13 * 2^{12}$ bytes.

77. El sistema requiere 12 bits para direccionar el contenido de una página ya que la página es de 32KB (2^{15} bytes) y el tamaño de la palabra es de 64 bits (2^3 bytes). Quedan 18 bits para la tabla de páginas.

    Si el proceso requiere 256MB, entonces $256MB/32KB = 2^{13}$ es el número de páginas. En el caso de la tabla de páginas de dos niveles, se utilizan 9 bits para el primer nivel y otros tantos para el segundo. Por lo que el proceso requiere $2^{13}/2^9 = 2^4$ tablas de segundo nivel. Así, el ahorro de memoria será la diferencia entre los consumos de ambas representaciones, es decir, $2^{18} * 8 - (2^9 + 2^4 * 2^9) * 8$ bytes.

78. Con 3 marcos:

    - FIFO: 10 fallos de página
        - Página 0: 2 → 4 → 3 → 6
        - Página 1: 3 → 5 → 1

    - Página 2: 1 → 2 → 5
  - LRU: 9 fallos de página
    - Página 0: 2 → 5
    - Página 1: 3 → 4 → 3 → 6
    - Página 2: 1 → 5 → 1

Con 4 marcos:

  - FIFO: 9 fallos de página
    - Página 0: 2 → 5 → 6
    - Página 1: 3 → 2
    - Página 2: 1 → 3
    - Página 3: 4 → 1
  - LRU: 8 fallos de página
    - Página 0: 2 → 6
    - Página 1: 3 → 5
    - Página 2: 1 → 3
    - Página 3: 4 → 1

Para ambas políticas se reduce el número de fallos de página al aumentar el número de marcos a 4.

79.
  - FIFO: 8 fallos de página
    - Página 0: 4 → 3
    - Página 1: 2 → 5
    - Página 2: 1 → 4
    - Página 3: 6 → 1
  - LRU: 11 fallos de página
    - Página 0: 4 → 2 → 1
    - Página 1: 2 → 3 → 4
    - Página 2: 1 → 5 → 3
    - Página 3: 6 → 5
  - Óptima: 7 fallos de página
    - Página 0: 4 → 1
    - Página 1: 2 → 5
    - Página 2: 1 → 3
    - Página 3: 6

80.
  - FIFO: 7 fallos de página
    - Página 0: 1 → 5 → 1

- Página 1: 2 → 6
- Página 2: 3 → 2

- LRU: 8 fallos de página
  - Página 0: 1 → 3
  - Página 1: 2 → 6 → 1
  - Página 2: 3 → 5 → 2
- Óptima: 6 fallos de página
  - Página 0: 1 → 5 → 6 → 1
  - Página 1: 2
  - Página 2: 3

81.
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include <time.h>

int tubo1[2], tubo2[2], tubo3[2], tubo4[2];

void cierraTuberias (void) {
  close (tubo1[0]);  close (tubo1[1]);
  close (tubo2[0]);  close (tubo2[1]);
  close (tubo3[0]);  close (tubo3[1]);
  close (tubo4[0]);  close (tubo4[1]);
}

void recorre (char *nombredir) {

  DIR *dir;
  struct dirent *entrada;
  struct stat datos;
  char *ruta;

  dir= opendir(nombredir);
  if (dir==NULL) {
    printf("Error al abrir el directorio %s\n", nombredir);
    return;
  }

  entrada= readdir(dir);
  while (entrada!=NULL) {
    if (strcmp(entrada->d_name,".") && strcmp(entrada->d_name,"..")) {
      ruta= malloc(strlen(nombredir)+strlen(entrada->d_name)+2);
      sprintf (ruta,"%s/%s",nombredir,entrada->d_name);
      lstat (ruta,&datos);
      if (S_ISDIR(datos.st_mode))  {
        recorre (ruta);  /* llamada recursiva */
        if (rmdir(ruta)== 0) {  /* directorio borrado */
          write (tubo1[1], entrada->d_name, strlen(entrada->d_name));
          write (tubo1[1], "\n", 1);
        }
```

```
      } else
        if (S_ISREG (datos.st_mode) && (datos.st_uid== getuid())) {
          if (time(NULL)- datos.st_atime > 2*7*24*60*60) {
            if (unlink (ruta)== 0) {  /* fichero borrado */
              write (tubo1[1], entrada->d_name,
                     strlen(entrada->d_name));
              write (tubo1[1], "\n", 1);
            }
          } else /* el aviso */
          if (time(NULL)- datos.st_atime > 1*7*24*60*60) {
            write (tubo2[1], entrada->d_name,
                   strlen(entrada->d_name));
            write (tubo2[1], "\n", 1);
         }
        }
      free (ruta);
    }
    entrada= readdir(dir);
  }
  closedir (dir);
}

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

  int numproc;

  if (argc!= 2) {
    printf("Debes especificar una direccion de correo electronico.\n");
    exit(1);
  }

  pipe (tubo1);
  pipe (tubo2);
  pipe (tubo3);
  pipe (tubo4);

  for (numproc= 0; numproc< 4; numproc++)
    if (fork()== 0) break;

  switch (numproc) {
    case 0:
      dup2 (tubo1[0], STDIN_FILENO);
      dup2 (tubo3[1], STDOUT_FILENO);
      cierraTuberias ();
      execlp ("sort", "sort", NULL);
      break;
    case 1:
      dup2(tubo2 [0], STDIN_FILENO);
      dup2(tubo4 [1], STDOUT_FILENO);
      cierraTuberias ();
      execlp ("sort", "sort", NULL);
      break;
    case 2:
      dup2 (tubo3[0], STDIN_FILENO);
      cierraTuberias ();
      execlp ("mail", "mail", argv[1], "-s",
              "Ficheros y directorios borrados", NULL);
      break;
    case 3:
      dup2 (tubo4[0], STDIN_FILENO);
      cierraTuberias ();
      execlp ("mail", "mail", argv[1], "-s",
```

```
                "Ficheros que se borraran en breve", NULL);
        break;
      case 4:
        recorre ("/tmp");
        cierraTuberias ();
        wait (NULL);
        wait (NULL);
        wait (NULL);
        wait (NULL);
    }

    return 0;
  }
```

82.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

#define MAXLON 2000

int tubo[2];

void recorre(char *nombredir) {

  DIR *dir;
  struct dirent *entrada;
  struct stat datos;
  char *ruta;
  int pos;

  dir= opendir(nombredir);
  if (dir== NULL) {
    printf ("Error al abrir el directorio %s\n", nombredir);
    return;
  }

  entrada= readdir(dir);
  while (entrada!= NULL) {
    if (strcmp (entrada->d_name,".") &&
        strcmp (entrada->d_name,"..")) {
      ruta= (char*) malloc (strlen (nombredir)+
                            strlen (entrada->d_name)+ 2);
      sprintf (ruta, "%s/%s", nombredir, entrada->d_name);
      lstat (ruta, &datos);
      if (S_ISDIR (datos.st_mode))
        recorre (ruta);
      else
        if (S_ISREG (datos.st_mode)) {
          pos= strlen(entrada->d_name)-2;
          if (strcmp (&entrada->d_name[pos],".c")== 0 ||
              strcmp (&entrada->d_name[pos],".h")== 0) {
            write (tubo[1], ruta, strlen(ruta));
            write (tubo[1], "\n", 1);
          }
        }
      free (ruta);
```

```
    }
    entrada= readdir (dir);
  }
  closedir (dir);
}

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

  char linea[MAXLON], *nombre;
  struct stat datos1, datos2;
  int pos;

  if (argc!= 3) {
    printf ("Debe especificar dos directorios\n");
    exit (1);
  }

  pipe (tubo);

  if (fork()== 0) {
    close   (tubo[0]);
    recorre (argv[1]);
    close   (tubo[1]);
  } else {
    dup2  (tubo[0], STDIN_FILENO);
    close (tubo[0]);
    close (tubo[1]);
    while (gets(linea)!= NULL) {
      pos= strlen(linea)-1;
      while (linea[pos]!= '/')
        pos--;
      nombre= (char*) malloc (strlen (argv[2])+
                              strlen(&linea[pos+1])+ 2);
      sprintf (nombre, "%s/%s", argv[2], &linea[pos+1]);
      lstat (linea, &datos2);
      if ((lstat (nombre, &datos1)!= 0) ||
          (datos1.st_mtime< datos2.st_mtime)) {
        if (fork()== 0)
          execlp ("cp", "cp", linea, nombre, NULL);
        else
          wait (NULL);
      }
      free (nombre);
    }
    wait (NULL);
  }
  return 0;
}
```

83.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <dirent.h>
#include <unistd.h>
#include <time.h>
#include <string.h>

int n_fich= 0;
int ahorro= 0;
```

```
void recorre (char *nombredir) {

  DIR *d;
  struct dirent *entrada;
  struct stat datosPs, datosPdf;
  char *ruta, *rutapdf;

  d= opendir(nombredir);
  if (d== NULL) {
    printf("Error al abrir el directorio\n");
    return;
  }

  entrada= readdir(d);
  while (entrada!=NULL) {
    if (strcmp(entrada->d_name,".") &&
        strcmp(entrada->d_name,"..")) {
      ruta= malloc(strlen(nombredir)+strlen(entrada->d_name)+2);
      sprintf(ruta,"%s/%s",nombredir,entrada->d_name);
      lstat (ruta,&datosPs);
      if (S_ISDIR(datosPs.st_mode))
        recorre(ruta);
      else
        if (S_ISREG(datosPs.st_mode)) {
          if (strcmp(&ruta[strlen(ruta)-3], ".ps")== 0) {/  * .ps? */
            rutapdf= malloc(strlen(ruta)+1);
            sprintf(rutapdf,"%s",ruta);
            rutapdf[strlen(ruta)-3]='\0';
            strcat (rutapdf, ".pdf");
            if (fork()== 0)
              execlp ("ps2pdf", "ps2pdf", ruta, rutapdf, NULL);
            wait(NULL);
            n_fich++;
            lstat(rutapdf,&datosPdf);
            ahorro+= (datosPs.st_blocks - datosPdf.st_blocks);
            unlink(ruta);
            free(rutapdf);
          }
        }
      free(ruta);
    }
    entrada= readdir(d);
  }
  closedir(d);
}

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

  int i;
  time_t inicio;

  if (argc== 1) {
    printf("Debe especificar al menos un directorio\n");
    exit(1);
  }

  for (i= 1; i< argc; i++) {
    if (fork()== 0) {
      inicio= time(NULL);
      recorre (argv[i]);
      printf ("Numero de ficheros transformados: %d\n", n_fich);
      printf ("Ahorro de espacio en bloques: %d\n", ahorro);
```

```
      printf ("Tiempo en procesar el directorio %s: %ld\n", argv[i],
                (time(NULL)- inicio));
      exit(0);
    }
  }

  for (i= 1; i< argc; i++)
    wait(NULL);

  return 0;
}
```

84.
```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

#define MAX 20

struct param {
  int num_estudiante;
  int num_grupo;
};

struct param buffer[MAX];

sem_t huecos,elementos,mutex;

int pos_productor= 0;
int num_estudiantes, cantidad_grupos, plazas_por_grupo;
int *grupo_de_alumno;
int *plazas_libres;

void init_datos() {

  int i;

  grupo_de_alumno= (int*)malloc(sizeof(int)*num_estudiantes);
  for (i= 0; i< num_estudiantes; i++)
    grupo_de_alumno[i]= -1;
  plazas_libres= (int*)malloc(sizeof(int)*cantidad_grupos);
  for (i= 0; i< cantidad_grupos; i++)
    plazas_libres[i]= plazas_por_grupo;
}

int *decide_preferencias (void) {

  int *peticiones;
  int i,pos1,pos2,aux;

  peticiones= (int *) malloc(sizeof(int)*cantidad_grupos);
  for (i= 0; i< cantidad_grupos; i++)
    peticiones[i]= i;
  for (i= 0; i< cantidad_grupos; i++) {
    pos1= random()%cantidad_grupos;
    pos2= random()%cantidad_grupos;
    aux= peticiones[pos1];
    peticiones[pos1]= peticiones[pos2];
    peticiones[pos2]= aux;
```

```
  }
  return peticiones;
}

int grupo_asignado (int num_estudiante) {
  return (grupo_de_alumno[num_estudiante]);
}

int hay_plazas_libres (int num_grupo) {
  return (plazas_libres[num_grupo]>0);
}

void asignar_grupo (int estudiante, int grupo) {

  if (grupo_de_alumno[estudiante]==-1) {
    grupo_de_alumno[estudiante]= grupo;
    plazas_libres[grupo]--;
  }
  else {
    printf("Error inesperado\n");
    pthread_exit(NULL);
  }
}

void *productor(void *p) {

  int num_estudiante= *(int *)p;
  int i;
  int *peticiones;
  struct param dato;

  peticiones= decide_preferencias ();
  for (i= 0; i< cantidad_grupos; i++) {
    dato.num_estudiante= num_estudiante;
    dato.num_grupo= peticiones[i];
    printf ("El estudiante %d pide el grupo %d\n", num_estudiante,
peticiones[i]);
    sem_wait(&huecos);
    sem_wait(&mutex);
    buffer[pos_productor]= dato;
    pos_productor= (pos_productor+1) %MAX;
    sem_post(&mutex);
    sem_post(&elementos);
    if (grupo_asignado(num_estudiante)!=-1)
      pthread_exit(NULL);
  }
  pthread_exit(NULL);
}

void *consumidor (void *p) {

  int pos_consumidor= 0;
  int asignados= 0;
  struct param dato;

  while (asignados< num_estudiantes) {
    sem_wait(&elementos);
    dato= buffer[pos_consumidor];
    pos_consumidor= (pos_consumidor+1) %MAX;
    sem_post(&huecos);

    if (hay_plazas_libres(dato.num_grupo)) {
```

```
      if (grupo_asignado(dato.num_estudiante)!=-1)
        printf("--¿ Peticion rechazada. Estudiante %d, grupo %d. El estudiante ya tiene grupo
asignado\n",
                dato.num_estudiante,dato.num_grupo);
      else {
        asignar_grupo(dato.num_estudiante, dato.num_grupo);
        printf("---¿ Al estudiante %d se le asigna el
grupo %d\n",dato.num_estudiante, dato.num_grupo);
        asignados++;
      }
    } else
        printf("--¿ Peticion rechazada. Estudiante %d, grupo %d. Grupo lleno\n",
                dato.num_estudiante,dato.num_grupo);

  }
  pthread_exit(NULL);
}


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

  pthread_t *estudiantes, gestor;
  int i, *datos;

  if (argc!= 4) {
    printf("Error. Debe proporcionar tres parametros: Num. estudiantes, Num. de grupos y
Tamaño de grupo\n");
    exit(1);
  }

  num_estudiantes= atoi(argv[1]);
  cantidad_grupos= atoi(argv[2]);
  plazas_por_grupo= atoi(argv[3]);

  if (num_estudiantes> cantidad_grupos*plazas_por_grupo) {
    printf("Error. No hay plazas para todos los estudiantes\n");
    exit(1);
  }

  init_datos();
  datos= (int*) malloc(sizeof(int)*num_estudiantes);
  estudiantes= (pthread_t*)
malloc(sizeof(pthread_t)*num_estudiantes);

  sem_init(&huecos,    0, MAX);
  sem_init(&elementos, 0, 0);
  sem_init(&mutex,     0, 1);

  pthread_create (&gestor, NULL, consumidor, NULL);
  for (i= 0; i< num_estudiantes; i++) {
    datos[i]= i;
    pthread_create (&estudiantes[i], NULL, productor, &datos[i]);
  }

  pthread_join(gestor, NULL);

  return 0;
}
```

85.
```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>

void recorre(char *nombredir) {

  DIR *d;
  struct dirent *entrada;
  struct stat datos;
  char *ruta;

  d= opendir(nombredir);
  if(d==NULL){
    printf("Error al abrir el directorio %s\n",nombredir);
    return;
  }

  entrada= readdir(d);
  while (entrada!= NULL) {
    if(strcmp(entrada->d_name,".") &&
       strcmp(entrada->d_name,"..")) {
       ruta=malloc(strlen(nombredir)+strlen(entrada->d_name)+2);
       sprintf(ruta,"%s/%s",nombredir,entrada->d_name);
       lstat(ruta,&datos);
       if (S_ISDIR(datos.st_mode))
         recorre(ruta);
       else
         if(fork()== 0)
           if (datos.st_size<2*1024*1024)
             execlp("lpr","lpr","-P","ps1",ruta,NULL);
           else
             execlp("lpr","lpr","-P","ps2",ruta,NULL);
         else
           wait(NULL);
       free(ruta);
     }
    entrada= readdir(d);
  }
  closedir(d);
}

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

  int num;
  time_t inicio,fin;

  inicio= time(NULL);

  for (num= 1; num< argc; num++)
    if (fork()== 0) {
      recorre(argv[num]);
      exit(0);
    }

  for (num= 1; num< argc; num++)
    wait(NULL);

  fin=time(NULL);
```

```
  printf("Duracion: %ld segundos\n",fin-inicio);

  return 0;
}
```

86.
```
#include <stdlib.h>
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

#define N 30

typedef struct {
  int id;
  int tiempo;
} param;

sem_t svar, slect, stam;
int ngallo= 0;


void comer(){}
void beber(){}
void dormir (int id, int tiempo){
  sleep(tiempo);
  printf("El gallo %d ha dormido %d horas\n",id,tiempo);
}

void *galloSalvaje (void *datos) {

  param *mi_dato= (param *)datos;
  int iden= mi_dato->id;
  int temp= mi_dato->tiempo;

  while(1){
    comer();

    sem_wait(&svar);
    printf("El gallo salvaje %d esta bebiendo\n",iden);
    beber();
    printf("El gallo salvaje %d ha terminado de beber\n",iden);
    sem_post(&svar);

    dormir(iden,temp);
  }
  pthread_exit(NULL);
}

void *galloDomestico (void * datos) {

  param *mi_dato= (param *)datos;
  int iden= mi_dato->id;
  int temp= mi_dato->tiempo;

  while(1){
    comer();

    sem_wait(&slect);
    ngallo++;
```

```
    if(ngallo==1) sem_wait(&svar);
    sem_post(&slect);

    sem_wait(&stam);
    printf("El gallo domestico %d esta bebiendo\n",iden);
    beber();
    printf("El gallo domestico %d ha terminado de beber\n",iden);
    sem_post(&stam);

    sem_wait(&slect);
    ngallo--;
    if(ngallo==0) sem_post(&svar);
    sem_post(&slect);

    dormir(iden,temp);
  }
  pthread_exit(NULL);
}

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

  pthread_t th[N];
  int i;
  param vector[N];

  sem_init(&svar,0,1);
  sem_init(&slect,0,1);
  sem_init(&stam,0,6);

  for(i=0;i<25;i++){ /* crea los gallos domesticos */
    vector[i].tiempo=7+random()%4;
    vector[i].id=i;
    pthread_create(&th[i],NULL,galloDomestico,(void *) &vector[i]);
  }
  for(i=25;i<N;i++){ /* crea los gallos salvajes */
    vector[i].tiempo=7+random()%4;
    vector[i].id=i;
    pthread_create(&th[i],NULL,galloSalvaje,(void *) &vector[i]);
  }

  for(i=0;i<N;i++)
    pthread_join(th[i],NULL);

  sem_destroy(&svar);
  sem_destroy(&slect);
  sem_destroy(&stam);

  exit(0);
}
```

87.

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>
#include <math.h>
#include <stdbool.h>

#define MAX 5

int buffer[MAX];
```

```
sem_t huecos, elementos, pos_prod, prod;
int pos_productor=0, cont_prod= 0, cont_primos= 0;

int suma_digitos_es_impar (int numero) {

  int suma= 0;
  while (numero>0) {
    suma+= numero%10;
    numero/= 10;
  }
  return (suma%2==1);
}

int es_primo(int numero) {

  int divisor;
  for (divisor=2; divisor<= sqrt(numero); divisor++)
    if (numero%divisor==0)
      return 0;
  return 1;
}

void *productor(void *p) {

  int i, dato;

  for (i= 0;i< 1000; i++) {
    dato= random()%1000;
    sem_wait(&prod);
    cont_prod++;
    sem_post(&prod);
    if (es_primo(dato)) {
      sem_wait(&huecos);
      sem_wait(&pos_prod);
      buffer[pos_productor]= dato;
      pos_productor= (pos_productor+1)%MAX;
      sem_post(&pos_prod);
      sem_post(&elementos);
    }
  }

  sem_wait(&huecos);
  sem_wait(&pos_prod);
  buffer[pos_productor]= -1;
  pos_productor= (pos_productor+1)%MAX;
  sem_post(&pos_prod);
  sem_post(&elementos);

  pthread_exit(NULL);
}

void *consumidor(void *p) {

  int pos_consumidor=0;
  int dato, cont= 0, term= 0;
  bool fin= false;

  while (!fin) {
    sem_wait(&elementos);
    dato= buffer[pos_consumidor];
    pos_consumidor= (pos_consumidor+1)%MAX;
    sem_post(&huecos);
```

```
    if (dato== -1) {
      term++;
      if (term== 3)
        fin= true;
    } else {
      cont_primos++;
      if (suma_digitos_es_impar(dato)) {
        printf(" %d ",dato);
        fflush(stdout);
        cont++;
        if (cont== 50)
          fin= true;
      }
    }
  }
  pthread_exit(NULL);
}


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

  pthread_t th[4];
  int i;

  sem_init(&huecos, 0, MAX);
  sem_init(&elementos, 0, 0);
  sem_init(&pos_prod, 0, 1);
  sem_init(&prod, 0, 1);

  for (i= 0; i< 3; i++)
    pthread_create (&th[i], NULL, productor, NULL);
  pthread_create (&th[3], NULL, consumidor, NULL);

  pthread_join(th[3], NULL);

  printf ("\n");
  printf ("Numeros primos procesados: %d\n", cont_primos);
  printf ("Numeros alatorios generados: %d\n", cont_prod);

  sem_destroy(&huecos);
  sem_destroy(&elementos);
  sem_destroy(&pos_prod);
  sem_destroy(&prod);

  return 0;
}
```

88.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include <time.h>

int tubo1[2], tubo2[2];

void cierraTuberias (void) {
  close (tubo1[0]);  close (tubo1[1]);
```

```
  close (tubo2[0]);  close (tubo2[1]);
}

void recorre (char *nombredir) {

  DIR *d;
  struct dirent *entrada;
  struct stat datos;
  char *ruta;

  d= opendir(nombredir);
  if (d==NULL) {
    printf("Error al abrir el directorio %s\n", nombredir);
    return;
  }

  entrada= readdir(d);
  while (entrada!=NULL) {
    if (strcmp(entrada->d_name,".") &&
        strcmp(entrada->d_name,"..")) {
      ruta= malloc(strlen(nombredir)+strlen(entrada->d_name)+2);
      sprintf (ruta,"%s/%s",nombredir,entrada->d_name);
      lstat (ruta,&datos);
      if (S_ISDIR(datos.st_mode))  {
        recorre (ruta);
        if (rmdir(ruta)== 0) {
          write(tubo1[1], entrada->d_name,
                          strlen(entrada->d_name));
          write(tubo1[1], "\n", 1);
        }
      } else
        if (S_ISREG (datos.st_mode) &&
           (datos.st_uid == getuid())) {
          if (datos.st_size> 2*1024*1024) {
            if (unlink(ruta)== 0) {
              write(tubo1[1], entrada->d_name,
                              strlen(entrada->d_name));
              write(tubo1[1], "\n", 1);
            }
          } else
          if (datos.st_size > 1*1024*1024) {
            write(tubo2[1], entrada->d_name,
                            strlen(entrada->d_name));
            write(tubo2[1], "\n", 1);
          }
        }
      free(ruta);
    }
    entrada= readdir(d);
  }
  closedir(d);
}

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

  int i;

  if (argc!=2) {
    printf("Debe especificar una direccion de correo electronico\n");
    exit(1);
  }
```

```
  pipe (tubo1);
  pipe (tubo2);

  for (i= 0; i< 2; i++)
    if (fork()== 0) break;

  switch (i) {
    case 0:
      dup2(tubo1[0], STDIN_FILENO);
      cierraTuberias();
      execlp ("mail", "mail", argv[1], "-s",
              "Ficheros y directorios borrados de tmp", NULL);
      break;
    case 1:
      dup2(tubo2[0], STDIN_FILENO);
      cierraTuberias();
      execlp ("mail", "mail", argv[1], "-s",
              "Ficheros que ocupan mucho espacio en tmp", NULL);
      break;
    case 2:
      recorre("/tmp");
      cierraTuberias();
      wait (NULL);
      wait (NULL);
      break;
  }

  return 0;
}
```

89. Solución al apartado a):

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

#define N_vehiculos 200
sem_t estacion;

void *camion (void * nada) {

  long int n= (long int) pthread_self();

  printf ("Llega camion: %ld\n", n);
  sem_wait(&estacion);
  sem_post(&estacion);
  printf ("Se atiende camion: %ld\n", n);
  sleep(random()%3+2); /* tiempo invertido en atender al camion */
  printf ("Sale camion: %ld\n", n);
  pthread_exit(NULL);
}

void *barco (void * nada) {

  long int n= (long int) pthread_self();

  printf ("Llega barco: %ld\n", n);
  sem_wait(&estacion);
  printf("Se atiende barco: %ld\n", n);
  sleep(random()%5+5); /* tiempo invertido en atender al barco */
```

```
  printf ("Sale barco: %ld\n", n);
  sem_post(&estacion);
  pthread_exit(NULL);
}

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

  int i;
  pthread_t th[N_vehiculos];

  sem_init(&estacion, 0, 1);
  for (i= 0; i< N_vehiculos; i++) {
    sleep(random()%3);
    if (random()%100 < 95)
      pthread_create(&th[i], NULL, camion, NULL);
    else
      pthread_create(&th[i], NULL, barco, NULL);
  }

  for (i= 0; i< N_vehiculos; i++)
    pthread_join(th[i], NULL);

  sem_destroy(&estacion);
  return 0;
}
```

Solución al apartado b):

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

#define N_vehiculos 200
sem_t estacion, camiones;
int ncamiones= 0;

void *camion (void * nada) {

  long int n= (long int) pthread_self();

  printf ("Llega camion: %ld\n", n);
  sem_wait(&camiones);
  ncamiones++;
  if (ncamiones== 1)
    sem_wait(&estacion);
  sem_post(&camiones);
  printf ("Se atiende camion: %ld\n", n);
  sleep(random()%3+2); /* tiempo invertido en atender al camion */
  printf ("Sale camion: %ld\n", n);
  sem_wait(&camiones);
  ncamiones--;
  if (ncamiones== 0)
    sem_post(&estacion);
  sem_post(&camiones);

  pthread_exit(NULL);
}
```

```
void *barco (void * nada) {

  long int n= (long int) pthread_self();

  printf ("Llega barco: %ld\n", n);
  sem_wait(&estacion);
  printf("Se atiende barco: %ld\n", n);
  sleep(random() %5+5); /* tiempo invertido en atender al barco */
  printf ("Sale barco: %ld\n", n);
  sem_post(&estacion);
  pthread_exit(NULL);
}

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

  int i;
  pthread_t th[N_vehiculos];

  sem_init(&estacion, 0, 1);
  sem_init(&camiones, 0, 1);
  for (i= 0; i< N_vehiculos; i++) {
    sleep(random() %3);
    if (random() %100 < 95)
      pthread_create(&th[i], NULL, camion, NULL);
    else
      pthread_create(&th[i], NULL, barco, NULL);
  }

  for (i= 0; i< N_vehiculos; i++)
    pthread_join(th[i], NULL);

  sem_destroy(&estacion);
  sem_destroy(&camiones);
  return 0;
}
```

90.

```
#include <stdio.h>
#include <sys/types.h>
#include <wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>

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

  int tub1[2],tub2[2];
  int fich;
  time_t ini, fin;

  if (fork()!= 0) {
     ini=time(NULL);
     wait(NULL);
     fin= time(NULL);
     printf ("Tiempo total: %ld segundos\n", fin-ini);
     return 0;
  }
  pipe (tub1);
  if (fork()!= 0){
    dup2  (tub1[0], STDIN_FILENO);
```

```
    close (tub1[0]);
    close (tub1[1]);
    fich= creat (argv[3], 00644);
    dup2 (fich, STDOUT_FILENO);
    close(fich);
    execlp ("uniq", "uniq", NULL);
  } else {
    pipe (tub2);
    if (fork()!= 0){
      dup2  (tub1[1],STDOUT_FILENO);
      dup2  (tub2[0],STDIN_FILENO);
      close (tub1[1]);
      close (tub1[0]);
      close (tub2[1]);
      close (tub2[0]);
      execlp ("sort", "sort", "-r", NULL);
    } else {
      dup2  (tub2[1], STDOUT_FILENO);
      close (tub1[0]);
      close (tub1[1]);
      close (tub2[0]);
      close (tub2[1]);
      execlp ("grep", "grep", argv[1], argv[2], NULL);
    }
  }
}
```

91. Solución al apartado a):

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

#define N_hilos 200

sem_t sbanco, scliente;
int ncliente= 0;

void *cliente(void *n){

  int id= *(int *) n;
  sleep(random()%100);
  printf ("Llega el cliente %d\n",id);

  sem_wait (&scliente);
  ncliente++;
  if (ncliente== 1)
    sem_wait (&sbanco);
  sem_post (&scliente);

  sleep(random()%3+2); /*/ tiempo de espera en la cola  */
  printf ("Se atiende al cliente: %d\n", id);
  sleep(random()%3+2); /* tiempo invertido en atenderle */

  sem_wait (&scliente);
  ncliente--;
  printf ("Sale el cliente %d\n", id);
  if (ncliente== 0)
    sem_post (&sbanco);
```

```
  sem_post (&scliente);

  pthread_exit(NULL);
}

void *furgon (void *n) {

  int id= *(int *) n;

  sleep  (random()%100);
  printf ("Llega el furgon %d\n",id);

  sem_wait (&sbanco);
  printf   ("Se atiende a los guardias %d\n", id);
  sleep (random()%5 + 5); /* tiempo invertido en atenderlos */
  printf   ("Se va el furgon %d\n", id);
  sem_post (&sbanco);

  pthread_exit(NULL);
}

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

  pthread_t hilo[N_hilos];
  int id[N_hilos];
  int i, furgon_creado= 0;

  sem_init(&sbanco, 0, 1);
  sem_init(&scliente, 0, 1);

  for (i= 0; i< N_hilos; i++){
    id[i]= i;
    if(furgon_creado== 1)
      pthread_create (&hilo[i], NULL, cliente, (void *)&id[i]);
    else
      if (random()%100< 95)
        pthread_create (&hilo[i], NULL, cliente, (void *)&id[i]);
      else {
        pthread_create (&hilo[i], NULL, furgon, (void *)&id[i]);
        furgon_creado= 1;
      }
  }

  for (i= 0; i< N_hilos; i++)
    pthread_join (hilo[i],NULL);

  sem_destroy(&sbanco);
  sem_destroy(&scliente);

  return 0;
}
```

Solución al apartado b):

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

#define N_hilos 200
```

```
sem_t sbanco, scliente, sorden;
int ncliente= 0;

void *cliente(void *n){

  int id= *(int *) n;
  sleep(random() %100);
  printf ("Llega el cliente %d\n",id);

  sem_wait (&sorden);
  sem_post (&sorden);

  sem_wait (&scliente);
  ncliente++;
  if (ncliente== 1)
    sem_wait (&sbanco);
  sem_post (&scliente);

  sleep(random() %3+2); /*/ tiempo de espera en la cola  */
  printf ("Se atiende al cliente: %d\n", id);
  sleep(random() %3+2); /* tiempo invertido en atenderle */

  sem_wait (&scliente);
  ncliente--;
  printf ("Sale el cliente %d\n", id);
  if (ncliente== 0)
    sem_post (&sbanco);
  sem_post (&scliente);

  pthread_exit(NULL);
}

void *furgon (void *n) {

  int id= *(int *) n;

  sleep  (random() %100);
  printf ("Llega el furgon %d\n",id);

  sem_wait (&sorden);
  sem_wait (&sbanco);
  printf    ("Se atiende a los guardias %d\n", id);
  sleep (random() %5 + 5); /* tiempo invertido en atenderlos */
  printf    ("Se va el furgon  %d\n", id);
  sem_post (&sbanco);
  sem_post (&sorden);

  pthread_exit(NULL);
}

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

  pthread_t hilo[N_hilos];
  int id[N_hilos];
  int i, furgon_creado= 0;

  sem_init(&sbanco, 0, 1);
  sem_init(&scliente, 0, 1);
  sem_init(&sorden, 0, 1);
```

```
  for (i= 0; i< N_hilos; i++){
    id[i]= i;
    if(furgon_creado== 1)
      pthread_create (&hilo[i], NULL, cliente, (void *)&id[i]);
    else
      if (random()%100< 95)
        pthread_create (&hilo[i], NULL, cliente, (void *)&id[i]);
      else {
        pthread_create (&hilo[i], NULL, furgon, (void *)&id[i]);
        furgon_creado= 1;
      }
  }

  for (i= 0; i< N_hilos; i++)
    pthread_join (hilo[i],NULL);

  sem_destroy(&sbanco);
  sem_destroy(&scliente);
  sem_destroy(&sorden);

  return 0;
}
```

92.

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

#define MAX 10
#define N_PROD 3
#define N_CONS 2

int buffer[MAX];

sem_t huecos, elementos, sprod, scon, sfin;
int pos_productor= 0, pos_consumidor= 0;
int cont= 0;

int generar_dato(void)    {return random()%256;}
int numero_aleatorio(void){return random()%100;}

void *productor (void *nada) {

  int i, dato, n= numero_aleatorio();

  for (i= 0; i< n; i++){
    dato= generar_dato();
    sleep(random()%2);
    sem_wait (&huecos);
    sem_wait (&sprod);
    buffer[pos_productor]= dato;
    pos_productor= (pos_productor+ 1)%MAX;
    sem_post (&sprod);
    sem_post (&elementos);
  }

  sem_wait (&sfin);
  cont++;
```

```
  if (cont== N_PROD) /* si soy el ultimo productor */
    for (i= 0; i< N_CONS ; i++){ /* por cada consumidor */
      sem_wait(&huecos);
      buffer[pos_productor]= -1;
      pos_productor= (pos_productor+ 1)%MAX;
      sem_post(&elementos);
    }
  sem_post (&sfin);

  pthread_exit(NULL);
}


void *consumidor(void *nada){

  int dato= 0;

  while (dato!= -1) {
    sem_wait(&elementos);
    sem_wait(&scon);
    dato= buffer[pos_consumidor];
    pos_consumidor= (pos_consumidor+ 1)%MAX;
    sem_post(&scon);
    sem_post(&huecos);
    if (dato!= -1) {
      printf("%d",dato);
      fflush(stdout);
    }
    sleep(random()%2);
  }

  pthread_exit(NULL);
}

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

  pthread_t th[N_PROD+N_CONS];
  int i=0;

  sem_init (&huecos,0,MAX);
  sem_init (&elementos,0,0);
  sem_init (&sprod,0,1);
  sem_init (&scon,0,1);
  sem_init (&sfin,0,1);

  for (i=0; i< N_PROD; i++)
    pthread_create (&th[i], NULL, productor, NULL);
  for (i=0; i< N_CONS; i++)
    pthread_create (&th[i+N_PROD], NULL, consumidor, NULL);
  for (i=0; i< N_PROD + N_CONS; i++)
    pthread_join (th[i], NULL);

  sem_destroy (&huecos);
  sem_destroy (&elementos);
  sem_destroy (&sprod);
  sem_destroy (&scon);
  sem_destroy (&sfin);

  return 0;
}
```

93.
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

int cont_enlaces= 0;
sem_t critico;

void recorre(char *nombredir) {

  DIR *d;
  struct dirent *entrada;
  struct stat datos;
  char *ruta;

  d= opendir(nombredir);
  if (d== NULL) {
    printf("Error al abrir el directorio\n");
    return;
  }

  entrada= readdir(d);
  while (entrada!= NULL) {
    if (strcmp(entrada->d_name,".") &&
        strcmp(entrada->d_name,"..")) {
      ruta= (char *)malloc (strlen(nombredir)+
                            strlen(entrada->d_name)+2);
      sprintf(ruta,"%s/%s",nombredir,entrada->d_name);
      lstat(ruta,&datos);
      if (S_ISDIR(datos.st_mode)) { /* es un directorio? */
        if (datos.st_uid== getuid()) /* pertenece al usuario? */
          recorre(ruta);
        else
          printf ("%s no pertenece al usuario\n", ruta);
      } else
      if (S_ISLNK(datos.st_mode)) { /* es un enlace blando? */
        sem_wait(&critico);
        cont_enlaces++;
        sem_post(&critico);
      }
      free(ruta);
    }
    entrada= readdir(d);
  }
  closedir(d);
}

void * busca (void *dato) {

  struct stat datos;
  char *nombredir= (char *)dato;

  lstat (nombredir, &datos);
  if (S_ISDIR(datos.st_mode))
    if (datos.st_uid== getuid())
      recorre(nombredir);
    else
      printf ("%s no pertenece al usuario\n", nombredir);
```

```
  else
    printf (" %s no es un directorio\n", nombredir);

  pthread_exit(NULL);
}

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

  pthread_t hilo[argc-1];
  int i;

  if (argc== 1) {
    printf("Debe especificar al menos un directorio\n");
    exit(1);
  }

  sem_init (&critico, 0, 1);
  for (i= 0; i< argc-1; i++)
    pthread_create (&hilo[i], NULL, busca, (void *) argv[i+1]);
  for (i= 0; i< argc-1; i++)
    pthread_join (hilo[i], NULL);

  printf ("Enlaces encontrados: %d\n", cont_enlaces);
  sem_destroy (&critico);
  return 0;
}
```

94.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdbool.h>
#include <math.h>

int es_primo(int numero) {

  int divisor;
  for (divisor=2; divisor<= sqrt(numero); divisor++)
    if (numero %divisor==0)
      return 0;
  return 1;
}

void productor (int inicio, int fin, int tubo[2], int tubo_fin[2]) {

  int numero;
  int contador_primos= 0;

  close (tubo[0]);
  close (tubo_fin[0]);

  for (numero= inicio; numero<= fin; numero++)
    if (es_primo(numero)) {
      write (tubo[1], &numero, sizeof(int));
      contador_primos++;
    }

  close (tubo[1]);
  write (tubo_fin[1], &contador_primos, sizeof(int));
```

```
  close (tubo_fin[1]);
}


void consumidor (int tubo[2], int tubo_fin[2]) {

  int datos[2];
  int leidos;
  int contador_claves= 0;
  bool fin= false;

  close (tubo[1]);
  close (tubo_fin[1]);

  while (!fin) {
    leidos= read(tubo[0], datos, 2*sizeof(int));
    if (leidos== 0) /* ya no hay mas datos en la tuberia */
      fin= true;
    else
      if (es_primo((datos[0]-1)/2) && es_primo((datos[1]-1)/2)) {
        printf ("Los numeros %d y %d forman la clave %d\n",
          datos[0], datos[1], datos[0]*datos[1]);
        contador_claves++;
      }
  }

  close (tubo[0]);
  read  (tubo_fin[0], datos, 2*sizeof(int));
  close (tubo_fin[0]);

  printf ("Resumen final\n");
  printf ("Se han encontrado %d numeros primos y %d claves criptograficas\n",
    datos[0]+datos[1], contador_claves);
}

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

  int desde, mitad, hasta;
  int tubo[2], tubo_fin[2];

  if (argc!=3) {
    printf ("Error, debe especificar el rango de numeros a analizar\n");
    exit(1);
  }

  desde= atoi(argv[1]); /* se asume que son numeros validos */
  hasta= atoi(argv[2]);
  mitad=(desde+hasta)/2;

  pipe(tubo);
  pipe(tubo_fin);

  if (fork()!= 0)
    if (fork()!= 0) {
      consumidor(tubo, tubo_fin);
      wait(NULL);
      wait(NULL);
    }
    else
      productor (desde, mitad, tubo, tubo_fin);
  else
    productor(mitad+1, hasta, tubo, tubo_fin);
```

```
  return 0;
}
```

95.
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <math.h>
#include <stdbool.h>

#define MAX    20
#define N_PROD  2
#define FIN    -1

int contador_primos= 0;

sem_t critico, elementos, huecos, primos;
int buffer[MAX];
int pos_productor= 0;

int es_primo(int numero) {

  int divisor;
  for (divisor=2; divisor<= sqrt(numero); divisor++)
    if (numero%divisor==0)
      return 0;
  return 1;
}

void * productor (void * dato){

  int *data = (int *)dato;
  int numero;
  int inicio = data[0];
  int fin =    data[1];

  for (numero= inicio; numero<= fin; numero++)
    if (es_primo(numero)){
      sem_wait(&huecos);
      sem_wait(&critico);
      buffer[pos_productor]= numero;
      pos_productor=(pos_productor+1)%MAX;
      sem_post(&critico);
      sem_post(&elementos);
      sem_wait(&primos);
      contador_primos++;
      sem_post(&primos);
    }

  sem_wait(&huecos);
  sem_wait(&critico);
  buffer[pos_productor]= FIN;
  pos_productor=(pos_productor+1)%MAX;
  sem_post(&critico);
  sem_post(&elementos);

  pthread_exit(NULL);
}
```

```
void *consumidor (void * nada){

  int datos[2];
  int dato,pos_consumidor= 0;
  int par= 0;
  bool fin= false;
  int contador_claves= 0, cont= 0;

  while (!fin) {
    sem_wait(&elementos);
    dato=buffer[pos_consumidor];
    pos_consumidor=(pos_consumidor+1) %MAX;
    sem_post(&huecos);
    if (dato== FIN) {
      cont++;
      if (cont== N_PROD)
        fin= true;
    } else {
      if (par %2== 0)
        datos[0]= dato;
      else {
        datos[1]= dato;
        if (es_primo((datos[0]-1)/2) && es_primo((datos[1]-1)/2)) {
          printf("Los numeros %d y %d forma la clave %d\n",
                  datos[0], datos[1], datos[0]*datos[1]);
          contador_claves++;
        }
      }
      par++;
    }
  }

  printf("Resumen final\n");
  printf("Se han encontrado %d numeros primos y %d claves criptograficas\n",
          contador_primos, contador_claves);

  pthread_exit(NULL);
}

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

  int desde, mitad, hasta;
  pthread_t th1,th2,th3;
  int data1[2], data2[2];

  if (argc!= 3){
    printf ("Error, debe especificar el rango de numeros a analizar\n");
    exit(1);
  }

  desde= atoi(argv[1]); /* se asume que los numeros son validos */
  hasta= atoi(argv[2]);
  mitad= (desde+hasta)/2;

  sem_init (&critico,   0, 1);
  sem_init (&primos,    0, 1);
  sem_init (&elementos, 0, 0);
  sem_init (&huecos,    0, MAX);

  data1[0]= desde;
  data1[1]= mitad;
  data2[0]= mitad+1;
```

```
  data2[1]= hasta;
  pthread_create (&th1, NULL, productor, (void *)data1);
  pthread_create (&th2, NULL, productor, (void *)data2);

  pthread_create (&th3, NULL, consumidor, NULL);

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

  sem_destroy (&critico);
  sem_destroy (&primos);
  sem_destroy (&elementos);
  sem_destroy (&huecos);

  return 0;
}
```

96.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <dirent.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <time.h>
#include <fcntl.h>

time_t inicio;
int tubo[2];

void recorre (char *nombredir) {

  DIR *d;
  struct dirent *entrada;
  struct stat datos;
  char *ruta;

  d= opendir (nombredir);
  if (d== NULL) {
    printf("Error al abrir el directorio %s\n", nombredir);
    return;
  }

  entrada= readdir(d);
  while (entrada!= NULL) {
    if (strcmp (entrada->d_name,".") &&
        strcmp (entrada->d_name,"..")) {
      ruta= malloc (strlen (nombredir)+
                    strlen (entrada->d_name)+2);
      sprintf (ruta, "%s/%s", nombredir, entrada->d_name);
      lstat (ruta,&datos);
      if (S_ISDIR (datos.st_mode))
        recorre(ruta);
      else
        if (S_ISREG (datos.st_mode)) {
          if ((inicio- datos.st_mtime> 60*60) &&
              (datos.st_uid== getuid())) {
            write (tubo[1], ruta, strlen(ruta));
            write (tubo[1], "\n", 1);
```

```
            if (fork()== 0)
              execlp ("zip", "zip", "/tmp/comprimido.zip", ruta, NULL);
            else
              wait(NULL);
          }
        }
      free(ruta);
    }
    entrada=readdir(d);
  }
  closedir(d);
}

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

  int fichero;
  inicio= time(NULL);

  if (argc!= 2) {
    printf ("Indica un directorio como parametro\n");
    exit(0);
  }

  pipe(tubo);
  if (fork()== 0){
    close   (tubo[0]);
    recorre (argv[1]);
    close   (tubo[1]);
  } else {
    fichero= creat ("usuario.log", 00644);
    dup2  (fichero, STDOUT_FILENO);
    dup2  (tubo[0], STDIN_FILENO);
    close (fichero);
    close (tubo[0]);
    close (tubo[1]);
    execlp ("sort", "sort", NULL);
  }

  exit(0);
}
```

97. La solución que se presenta a continuación incluye los dos puntos indicados en el enunciado donde el semáforo *orden* se ha utilizado para resolver el segundo punto.

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>

#define N_atletas 30

sem_t instalaciones, corredores, orden;
int cont= 0;

void *corredor (void * nada) {

  int n= (int) syscall(SYS_gettid);
```

```
  sem_wait(&orden);
  printf ("Llega corredor: %d\n", n);
  sem_wait(&corredores);
  cont++;
  if (cont== 1)
    sem_wait(&instalaciones);
  printf ("Entra en las instalaciones el corredor %d\n", n);
  sem_post(&corredores);
  sem_post(&orden);

  sleep(random() %3); /* tiempo invertido en usarlas */

  sem_wait(&corredores);
  cont--;
  printf ("Sale de las instalaciones el corredor: %d\n", n);
  if (cont== 0)
    sem_post(&instalaciones);
  sem_post(&corredores);

  pthread_exit(NULL);
}

void *jabalino (void * nada) {

  int n= (int) syscall(SYS_gettid);

  sem_wait(&orden);
  printf ("Llega lanzador de jabalina: %d\n", n);
  sem_wait(&instalaciones);
  printf ("Entra en las instalaciones el jabalino %d\n", n);
  sleep(random() %3+2); /* tiempo invertido en usarlas */
  printf ("Sale de las instalaciones el jabalino: %d\n", n);
  sem_post(&instalaciones);
  sem_post(&orden);

  pthread_exit(NULL);
}

void *martillo (void * nada) {

  int n= (int) syscall(SYS_gettid);

  sem_wait(&orden);
  printf ("Llega lanzador de martillo: %d\n", n);
  sem_wait(&instalaciones);
  printf ("Entra en las instalaciones el martillo %d\n", n);
  sleep(random() %3+2); /* tiempo invertido en usarlas */
  printf ("Sale de las instalaciones el martillo: %d\n", n);
  sem_post(&instalaciones);
  sem_post(&orden);

  pthread_exit(NULL);
}

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

  int num;
  pthread_t th[N_atletas];

  sem_init (&instalaciones, 0, 1);
  sem_init (&corredores, 0, 1);
  sem_init (&orden, 0, 1);
```

```
  for (num= 0; num< N_atletas; num++) {
    sleep(random()%3);
    if (random()%100 < 66)
      pthread_create (&th[num], NULL, corredor, NULL);
    else
      if (random()%100 < 82)
        pthread_create (&th[num], NULL, jabalino, NULL);
      else
        pthread_create (&th[num], NULL, martillo, NULL);
  }

  for (num= 0; num< N_atletas; num++)
    pthread_join(th[num], NULL);

  sem_destroy (&instalaciones);
  sem_destroy (&corredores);
  sem_destroy (&orden);

  return 0;
}
```

98.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <pthread.h>
#include <string.h>
#include <semaphore.h>

#define NHILOS 100
#define PAISES 10

typedef struct {
  int pais;
  int valor;
} params;

char paises[PAISES][30]={ "España", "Portugal", "Francia",
"Ucrania", "Letonia", "Servia", "Israel", "Alemania", "Italia", "Holanda"};
int puntuaciones[PAISES]={0,0,0,0,0,0,0,0,0,0};
sem_t srecurso[PAISES],sjurado;
int njurados= 0;

void *voto (void *dato) {

  params *mi_dato= (params*) dato;
  int pais=  mi_dato->pais;
  int puntos=mi_dato->valor;

  sleep(random()%2);

  sem_wait(&srecurso[pais]);
  printf ("Voto al pais %s %d puntos\n", paises[pais],puntos);
  puntuaciones[pais]+= puntos;
  printf ("Ya he votado al pais %s %d puntos\n", paises[pais],puntos);
  sem_post(&srecurso[pais]);

  pthread_exit(NULL);
}
```

```
void *consulta (void *dato) {

  int pais= *(int*)dato, num;

  sleep(random() %2);

  sem_wait(&sjurado);
  njurados++;
  if (njurados== 1)
    for (num= 0; num< PAISES; num++)
      sem_wait(&srecurso[num]);
  printf("El pais %d realiza la consulta\n", pais);
  sem_post(&sjurado);

  for (num= 0; num< PAISES; num++)
    printf ("(%s, %d)\n", paises[num], puntuaciones[num]);

  sem_wait(&sjurado);
  njurados--;
  printf ("El país %d termina la consulta\n", pais);
  if (njurados== 0)
    for (num= 0; num< PAISES; num++)
      sem_post (&srecurso[num]);
  sem_post(&sjurado);

  pthread_exit(NULL);
}

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

  pthread_t hilo[NHILOS];
  int num, pais, idPais[NHILOS];
  params unvoto[NHILOS];

  for (num= 0; num< PAISES; num++)
    sem_init (&srecurso[num], 0, 1);
  sem_init (&sjurado, 0, 1);

  for (num= 0; num< NHILOS; num++){
    pais= random() %PAISES;
    if (random() %100 < 90) {
      unvoto[num].pais=  pais;
      unvoto[num].valor= random() %10;
      pthread_create (&hilo[num], NULL, voto, (void *)&unvoto[num]);
    } else {
      idPais[num]= pais;
      pthread_create (&hilo[num], NULL,
                      consulta, (void *)&idPais[num]);
    }
  }

  for (num= 0; num< NHILOS; num++)
    pthread_join(hilo[num],NULL);

  for (num=0; num< PAISES; num++)
    sem_destroy(&srecurso[num]);
  sem_destroy(&sjurado);

  return 0;
}
```

99.
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

#define N 100
#define PAISES 10

char paises[PAISES][10]={"España", "Portugal", "Francia", "Ucrania",
"Letonia", "Servia", "Israel", "Alemania", "Italia", "Holanda"};
int puntuaciones[PAISES]={0,0,0,0,0,0,0,0,0,0};

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

  int num, dato[2];
  int tubA[2],tubB[2];
  char *linea;

  pipe(tubA);

  for (num= 0; num< N; num++){
    if (fork()== 0){
      srandom (getpid());
      dato[0]= random() %PAISES;
      dato[1]= random() %10;
      printf ("Soy el votante %d, mi voto es para el pais %d %d puntos\n",
               num, dato[0], dato[1]);
      close (tubA[0]);
      write (tubA[1], dato, 2*sizeof(int));
      close (tubA[1]);
      exit (0);
    }
  }

  close (tubA[1]);
  while (read (tubA[0], dato, 2*sizeof(int)))
    puntuaciones[dato[0]]+= dato[1];
  close (tubA[0]);

  printf("La votacion resultante es:\n");
  printf("(pais,numero de votos)\n");
  for (num= 0; num< PAISES; num++)
    printf ("(%s, %d)\n", paises[num], puntuaciones[num]);

  pipe(tubB);

  if (fork()!= 0) {
    close (tubB[0]);
    for (num= 0; num< PAISES; num++){
      linea= (char*) malloc (strlen (paises[num])+ 15);
      sprintf (linea, " %3d puntos, %s\n",
               puntuaciones[num], paises[num]);
      write (tubB[1], linea, strlen(linea));
      free  (linea);
    }
    close (tubB[1]);
  } else {
     dup2   (tubB[0],STDIN_FILENO);
     close  (tubB[1]);
     close  (tubB[0]);
```

```
      execlp ("sort", "sort", "-r", NULL);
    }

    for (num= 0; num< N+1; num++)
      printf ("Fin del proceso de PID %d.\n", wait(NULL));

    return 0;
  }
```

100.
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

int cuenta= 0, tubo1[2], tubo2[2];

void recorre (char *nombredir) {

  DIR *d;
  struct dirent *entrada;
  struct stat datos;
  char *ruta;
  char dato[10];

  d= opendir(nombredir);
  if (d== NULL) return;

  entrada= readdir(d);
  while (entrada!=NULL) {
    if (strcmp(entrada->d_name, ".") &&
        strcmp(entrada->d_name, "..")) {
      ruta= malloc (strlen(nombredir)+ strlen(entrada->d_name)+2);
      sprintf (ruta, "%s/%s", nombredir, entrada->d_name);
      lstat (ruta, &datos);
      if (S_ISDIR(datos.st_mode))
        recorre (ruta);
      else
        if (datos.st_blocks> 1024) {
          sprintf (dato, "%ld\n", (long int)datos.st_ino);
          write (tubo1[1], dato, strlen(dato));
          cuenta++;
        }
      free(ruta);
    }
    entrada= readdir(d);
  }
  closedir (d);
}

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

  int num, dato, total= 0;

  pipe(tubo1);
  pipe(tubo2);
```

```
  for (num= 1; num< argc; num++)
    if (fork()== 0) { /* procesos de busqueda */
      close (tubo1[0]);
      close (tubo2[0]);
      recorre (argv[num]);
      write (tubo2[1], &cuenta, sizeof(int));
      close (tubo1[1]);
      close (tubo2[1]);
      exit(0);
    }
  if (fork()== 0) { /* proceso de ordenacion */
    dup2 (tubo1[0], STDIN_FILENO);
    close (tubo1[0]);
    close (tubo1[1]);
    close (tubo2[0]);
    close (tubo2[1]);
    execlp ("sort", "sort", NULL);
  }

  close (tubo1[0]); /* proceso principal */
  close (tubo1[1]);
  close (tubo2[1]);
  while (read (tubo2[0], &dato, sizeof(int))!= 0)
    total+= dato;
  close(tubo2[0]);

  for (num= 0; num< argc; num++)
    wait(NULL);

  printf ("Total de ficheros encontrados %d\n", total);

  return 0;
}
```