Ejercicios
Tema 3. Comunicación y sincronización entre procesos

Ejercicio 1. ¿Cuál es el número máximo de procesos que pueden ejecutar una operación `wait` sobre un semáforo que se inicializó con un valor de 4? ¿Cuál es el número máximo de procesos que pueden bloquearse?

Ejercicio 2. Escriba un programa que cree 100 procesos ligeros, creados como independientes, es decir, cuando acaban su ejecución liberan sus recursos y nadie puede esperar por ellos. Una vez creados todos los procesos se deberá esperar a la terminación de todos ellos.

Ejercicio 3. Escriba el código de dos procesos ligeros que deben alternar de forma estricta su ejecución. Resuelva el problema utilizando semáforos.

Ejercicio 4. Resuelva el ejercicio anterior utilizando mutex y variables condicionales.

Ejercicio 5. Se desea desarrollar una aplicación que debe realizar dos tareas que se pueden ejecutar de forma independiente. Los códigos de estas dos tareas se encuentran definidos en dos funciones cuyos prototipos en lenguaje de programación C, son los siguientes:

```
void tarea_A(void);
void tarea_B(void);
```

Se pide:

a) Programar la aplicación anterior utilizando tres modelos distintos: un programa secuencial ejecutado por un único proceso, un programa que crea procesos para desarrollar cada una de las tareas, y un programa que realiza las tareas anteriores utilizando procesos ligeros. En cualquiera de los tres casos la aplicación debe terminar cuando todas las tareas hayan acabado.

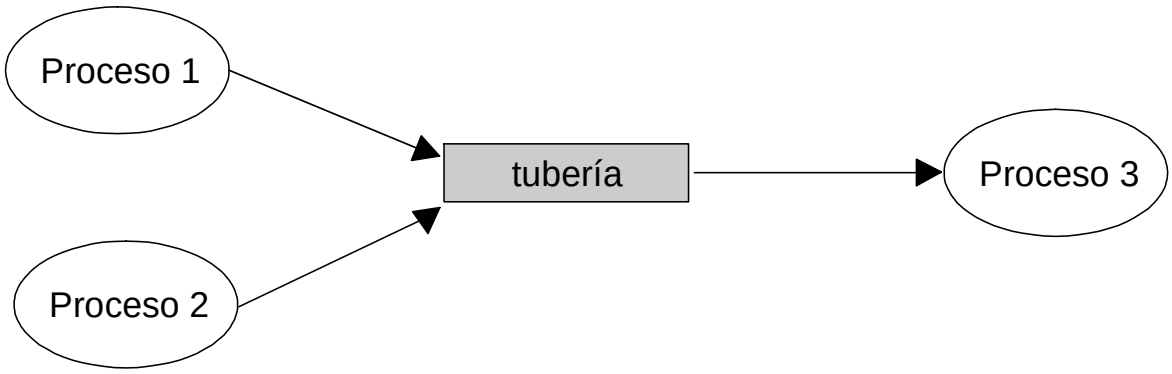
Ejercicio 6. Se desea usar un semáforo para asegurar que el inicio de una determinada actividad del proceso P2 comienza después de que finalice una actividad del proceso P1. ¿Qué primitiva de semáforos debe usar cada proceso y cuál debe ser el valor inicial del semáforo?

Ejercicio 7. Resuelva el ejercicio anterior utilizando mutex y variables condicionales.

Ejercicio 8. Se desea comparar el rendimiento de un servidor de ficheros secuencial con uno multiflujo que utiliza procesos ligeros (threads). En ambos casos el servicio de una petición implica la ejecución con 10 ms de tiempo de cómputo y en el caso de que los datos pedidos no esté en la cache de bloques del servidor, otros 40 ms de acceso al disco. En el servidor secuencial, durante el acceso al disco se bloquea el proceso servidor. Sin embargo, en el segundo caso sólo se bloquea el thread correspondiente. Teniendo en cuenta que el disco sólo puede llevar a cabo una operación en cada momento y suponiendo que el servidor se ejecuta en un sistema monoprocesador se pide:

a) Suponiendo que no se producen aciertos en la cache, ¿Cuántas peticiones por segundo puede procesar el servidor secuencial? ¿y un servidor multiflujo?
b) Igual que el anterior apartado pero con un 100% de aciertos en la cache de bloques.
c) Igual para un 50% de aciertos
d) Considerar cómo afectaría a los resultados de los tres apartados anteriores el uso de un multiprocesador con 4 procesadores para ejecutar el servidor.

Ejercicio 9. Escriba un programa que cree tres procesos que se conecten entre ellos utilizando una tubería tal y como se muestra en la siguiente figura.



Ejercicio 10. Se dispone de dos procesos ligeros para la ejecución de un programa para escuchar música a través de *streaming*:

1. Un proceso ligero **A** que lee datos de un dispositivo de red y almacena los datos en un *buffer* compartido.
2. Un proceso ligero **B** que toma los datos de un *buffer* compartido y escribe esto en un dispositivo de audio.

Ambos procesos utilizan dos primitivas para acceder a los datos:

- cantidad = Leer (dispositivo)
  - En caso de que cantidad sea igual a 0, eso significa que no hay más datos.
- Escribir (dispositivo, cantidad)

Siendo el parámetro: red, audio, o buffer

Codifique los procesos ligeros A y B, e indique que estructuras de datos son necesarias teniendo en cuenta los siguientes supuestos:

- La aplicación parte con el *buffer* vacio
- Cuando el *buffer* contiene datos, el proceso A debe avisar al proceso B de que existen datos suficientes en el buffer.
- Si el *buffer* se encuentra lleno al 100 %, el proceso A para automáticamente.
- Si el *buffer* se encuentra vacio, el proceso B debe esperar a que existan datos.
- Si no hay más datos que leer del dispositivo de red, el proceso A debe avisar al proceso B de que no hay más datos y que debe terminar cuando los datos del *buffer* sean procesados.

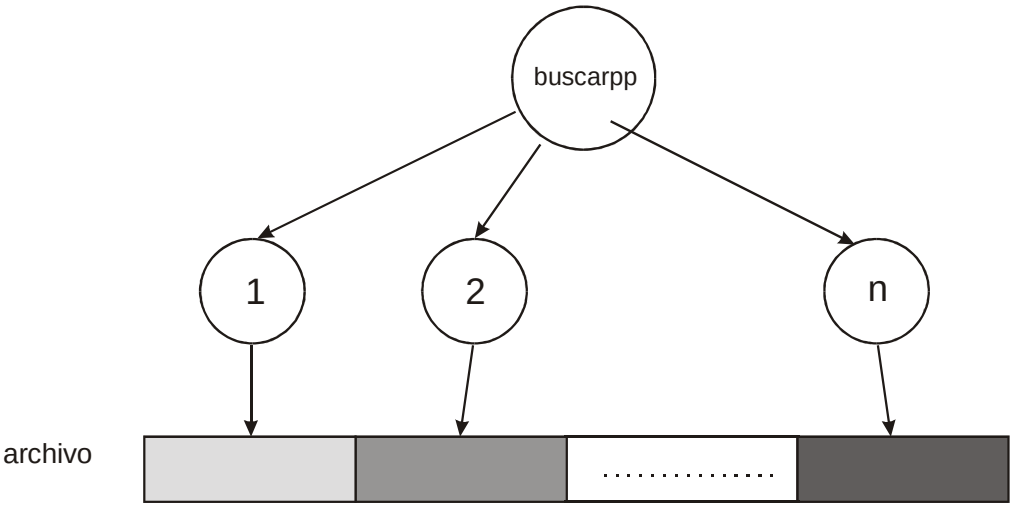
Para la resolución del problema utilice mutex y variables condicionales.

Ejercicio 11. Se desea desarrollar un programa en C, utilizando los servicios del estándar POSIX, que busque cuantas veces aparece un determinado carácter en un archivo. La sintaxis de este programa será la siguiente:

```
buscar  c file
```

donde `c` es el carácter a buscar y `file` el nombre del archivo. El programa debe imprimir el número de veces que aparece el carácter `c` en el fichero. En caso de que el carácter no se encuentre en el archivo el programa dará como resultado 0. El programa deberá realizar un correcto tratamiento de errores.
La versión a desarrollar debe utilizar procesos ligeros, de forma que la búsqueda en el fichero se haga en paralelo (ver figura adjunta). Este programa se encargará de:

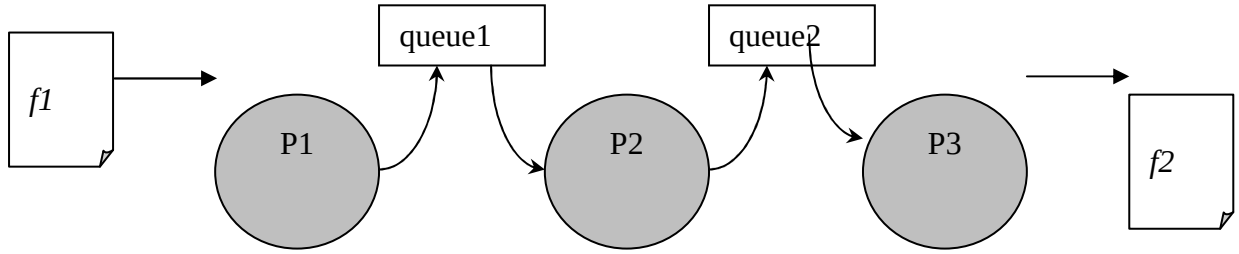
1. Crear los procesos hijos encargados de realizar la búsqueda. Cada uno de estos procesos realizará la búsqueda en una porción del archivo (tal y como se muestra en la figura). Haga una versión para n= 4 procesos ligeros.
2. El programa debe finalizar cuando todos los procesos hayan acabado de procesar su parte. Utilice para el desarrollo de este programa procesos ligeros, mutex y variables condicionales.



Ejercicio 12. Se pretende diseñar e implementar una cadena de montaje usando colas de mensajes de UNIX. La cadena de montaje está compuesta por tres procesos, cada uno especializado en una función. La comunicación entre cada par de procesos (es decir el proceso i y el proceso i+1) se realiza a través de una cola de mensajes de UNIX. En esta cadena de montaje, cada proceso realiza una función bien diferenciada:

- El primer proceso lee un fichero *f1* y escribe en la primera cola trozos del fichero de longitud máxima 4KB.
- El proceso intermedio *i* lee de la cola *i-1* cada trozo del fichero y realiza una simple función de conversión, consistente en reemplazar las letras minúsculas por letras mayúsculas. Una vez realizada esta transformación, escribe el contenido en la cola *i*.
- El último proceso lee de la cola *i-1* y lo vuelca al fichero *f2*.

Gráficamente:



El programa principal acepta dos argumentos de entrada, correspondientes al nombre del fichero origen (*f1)* y destino (*f2).* Debe ejecutarse de la siguiente manera:

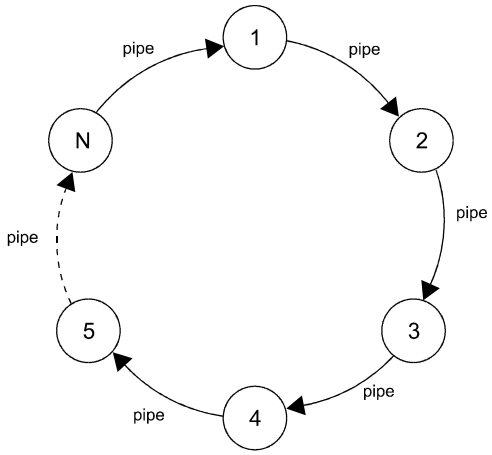
cadena_montaje <f1> <f2>

Se pide:

a) Decisiones de diseño necesarias para la implementación del programa *cadena_montaje.* Se entrega un archivo *cadena_montaje.doc.*
b) Código fuente en el lenguaje de programación C del programa cadena_montaje.c

Ejercicio 13. Se quiere realizar un programa que cree un conjunto de procesos que acceden en exclusión mutua a un fichero compartido por todos ellos. Para ello, se deben seguir los siguientes pasos:

a) Escribir un programa que **cree** N procesos hijos. Estos procesos deben formar un anillo como el que se muestra en la figura. Cada proceso en el anillo se enlaza de forma unidireccional con su antecesor y su sucesor mediante un pipe. Los procesos **no** deben redirigir su entrada y salida estándar. El valor de N se recibirá como argumento en la línea de mandatos. Este programa debe crear además, el fichero a compartir por todos los procesos y que se denomina anillo.txt

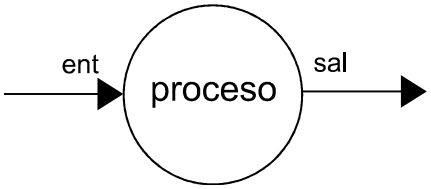


b) El proceso que crea el anillo inserta en el mismo un único carácter que hará de testigo, escribiendo en el pipe de entrada al proceso 1. Este testigo recorrerá el anillo indefinidamente de la siguiente forma: cada proceso en el anillo espera la recepción del testigo; cuando un

proceso recibe el testigo lo conserva durante 5 segundos; una vez transcurridos estos 5 segundos lo envía al siguiente proceso en el anillo. Codifique la función que realiza la tarea anteriormente descrita. El prototipo de esta función es:

```
void tratar_testigo(int ent, int sal);
```

donde ent es el descriptor de lectura del pipe y sal el descriptor de escritura.



c) Escribir una función que lea de la entrada estándar un carácter y escriba ese carácter en un fichero cuyo descriptor se pasa como argumento a la misma. Una vez escrito en el fichero el carácter leído, la función escribirá por la salida estándar el identificador del proceso que ejecuta la función.

d) Cada proceso del anillo crea dos procesos ligeros que ejecutan indefinidamente los códigos de las funciones desarrolladas en los apartados b y c respectivamente. Para asegurar que los procesos escriben en el fichero en exclusión mutua se utilizará el paso del testigo por el anillo. Para que el proceso pueda escribir en el fichero debe estar en posesión del testigo. Si el proceso no tiene el testigo esperará a que le llegue éste. Nótese que el testigo se ha de conservar en el proceso mientras dure la escritura al fichero. Modificar las funciones desarrolladas en los apartados b y c para que se sincronicen correctamente utilizando semáforos.