Problema 1

Considere un sistema donde existen dos flujos de ejecución, tal que el primero, denominado *escritor*, realiza una única escritura en un fichero, mientras que el segundo, llamado *lector*, lleva a cabo una lectura de ese mismo fichero pero sólo después de que el primer flujo haya realizado la escritura.

a) [1 pto] ¿Es posible implementar la sincronización requerida usando **sólo** cerrojos de fichero? Si lo es, describa, sin especificar código, qué acciones de sincronización llevaría a cabo cada flujo de ejecución. En caso contrario, explique por qué motivo no es factible.

b) [1 pto] ¿Es posible implementar la sincronización requerida usando **sólo** *mutex*? Si lo es, describa, sin especificar código, qué acciones de sincronización llevaría a cabo cada flujo de ejecución. En caso contrario, explique por qué motivo no es factible.

c) [1 pto] Se plantea resolver la sincronización requerida con el uso de un semáforo. Describa, sin especificar código, qué valor inicial debería tener el semáforo y qué acciones de sincronización (*sem_wait* o *sem_post*) llevaría a cabo cada flujo de ejecución, especificando en qué momento se realizarían.

d) [1 pto] Considere la siguiente solución a la sincronización requerida basada en el uso de un mutex *m* y una condición *c*, siendo *fd* el descriptor del fichero.

Escritor	*Lector*
write(fd, &dato, sizeof(dato));	pthread_mutex_lock(&m);
pthread_mutex_lock(&m);	pthread_cond_wait(&c, &m);
pthread_cond_signal(&c);	pthread_mutex_unlock(&m);
pthread_mutex_unlock(&m);	read(fd, &dato, sizeof(dato));

Explique por qué motivo es errónea y complétela incluyendo el código necesario para que sea correcta.

e) [1 pto] Se plantea resolver la sincronización requerida con el uso de un FIFO. Suponiendo que ya se ha creado el FIFO y que ambos flujos ya lo han abierto y guardan su descriptor en la variable *pd*, especifique el código requerido en cada flujo para llevar a cabo la sincronización.

f) [1 pto] Suponga que los flujos ejecutan en distintas máquinas, compartiendo el fichero mediante NFS. Explique qué mecanismo se debería usar para llevar a cabo la sincronización y describa, sin especificar código, qué acciones de sincronización llevaría a cabo cada flujo de ejecución, especificando en qué momento se realizarían.

Solución

Antes de pasar a responder las diversas cuestiones que se plantean en el enunciado, se considera conveniente intentar catalogar el escenario de sincronización descrito en el enunciado comparándolo con los clásicos problemas de sincronización estudiados en esta disciplina. Aunque no es estrictamente necesario este paso previo y podría realizarse un estudio específico del problema planteado, se considera que puede resultar didáctico e ilustrativo.

En el escenario propuesto, existe un flujo de ejecución (*lector*) que, antes de llevar a cabo una determinada operación (la lectura de un fichero), tiene que esperar hasta que otro flujo de ejecución (*escritor*) complete una cierta operación (la escritura en el mismo). Se trata, por tanto, de un esquema similar al productor-consumidor pero muy simplificado: por un lado, sólo hay un único productor (el flujo escritor) y un único consumidor (el flujo lector); por otro lado, el productor genera un único dato, por lo que no tiene que esperar hasta que el consumidor haya obtenido un dato para generar el siguiente (no tiene que esperar por huecos), es decir, que, bajo ninguna circunstancia, el productor tendrá que esperar hasta que el consumidor complete ninguna acción. Obsérvese que el esquema propuesto en el enunciado no guarda ninguna relación con el problema de los lectores-escritores, donde hay que arbitrar el acceso a un recurso de manera que pueda haber simultáneamente o bien múltiples lectores o bien un único escritor, pero donde no hay que fijar ningún orden específico en el acceso al recurso.

Por último, antes de entrar a analizar las distintas preguntas propuestas, conviene aclarar que, excepto en el último apartado, en todos los demás se puede suponer cualquier relación entre los dos flujos de ejecución descritos en el enunciado: podrían ser *threads* del mismo proceso, o de dos procesos diferentes ya estén éstos emparentados o sean totalmente independientes.

a) No es posible resolver el problema de sincronización planteado usando **únicamente** cerrojos, puesto que ese mecanismo permite arbitrar el acceso a un fichero de manera que pueda haber simultáneamente o bien múltiples lectores o bien un único escritor, pero no permite fijar ningún orden específico en el acceso al recurso.

b) No es factible resolver el problema de sincronización propuesto utilizando **únicamente** mutex, puesto que ese mecanismo permite crear secciones críticas de manera que se asegure que el código afectado se ejecuta en exclusión mutua, pero no permite controlar el orden de acceso a la sección crítica. Nótese que sólo con mutex no se puede resolver el problema de que un flujo de ejecución se queda esperando hasta que se cumpla una determinada condición. Por ello, en la mayoría de los escenarios de sincronización, es necesario usar los mutex junto con las variables de condición, como se analiza en un apartado posterior.

c) Para resolver el problema de sincronización planteado, bastaría con usar un semáforo iniciado a cero y tal que el flujo escritor ejecutaría una operación *sem_post* después de escribir en el fichero y el flujo lector realizaría una operación *sem_wait* justo antes de leer del mismo.

d) El código planteado es erróneo puesto que si el escritor consigue el mutex antes de que lo haga el lector, la operación *pthread_cond_signal* no tendría ningún efecto al no haber nadie esperando en esa variable de condición. Por tanto, cuando finalmente el lector obtenga el mutex, se quedará bloqueado indefinidamente en la operación *pthread_cond_wait*. Al tratarse de un mecanismo sin estado, a diferencia del semáforo que gestiona un contador interno, cuando se usan variables de condición, es necesario utilizar algunas variables que guarden el estado requerido. En el ejemplo planteado, bastaría con usar una variable booleana inciada a falsa (*bool completada= false;*), que refleje si el flujo lector ya ha completado la escritura.

Escritor

```
write(fd, &dato, sizeof(dato));
pthread_mutex_lock(&m);
completada=true;
pthread_cond_signal(&c);
pthread_mutex_unlock(&m);
```

Lector

```
pthread_mutex_lock(&m);
while (!completada)
        pthread_cond_wait(&c, &m);
pthread_mutex_unlock(&m);
read(fd, &dato, sizeof(dato));
```

Con el uso de esta variable, si el escritor es el primero que entra en la sección crítica controlada por el mutex, aunque no tenga efecto la operación *pthread_cond_signal*, el hecho de que ya ha realizado la escritura queda reflejado en la variable, por lo que cuando el lector ejecute la sección crítica no realizará la operación *pthread_cond_wait*. Nótese que, aunque no sea estrictamente necesario en este caso al haber un único flujo consumidor, se ha optado por usar un *while* en vez de un *if* a la hora de evaluar el estado de la variable para mantener el patrón estándar de este tipo de soluciones.

e) Dado que las tuberías, además de para comunicarse, incluyen también una sincronización (tanto la lectura de una tubería vacía como la escritura en una llena bloquean al proceso que realiza la operación), pueden usarse para resolver el problema de sincronización planteado: el escritor escribirá cualquier información en la tubería (bastaría con un byte con cualquier valor) justo después de modificar el fichero, mientras que el lector leerá de la tubería antes de hacerlo del fichero.

Escritor

```
write(fd, &dato, sizeof(dato));
write(pd, &c, sizeof(char));
```

Lector

```
read(pd, &c, sizeof(char));
read(fd, &dato, sizeof(dato));
```

Nótese que podría plantearse prescindir del fichero y enviar el dato directamente a través de la tubería. Si embargo, esa opción cambiaría el escenario propuesto en el enunciado donde el dato, además de transferirse entre los procesos, queda permanentemente almacenado en el fichero.

f) Al tratarse de dos flujos que ejecutan en dos máquinas diferentes, es necesario usar un mecanismo de paso de mensajes, como son los sockets, y puesto que del enunciado se desprende que no se puede admitir que se pierdan mensajes puesto que esa circunstancia malograría la sincronización requerida, lo razonable sería usar sockets de tipo *stream*, que en el caso de Internet, se implementan sobre el protocolo TCP.

Dado que la llamada de lectura/recepción sobre un socket es bloqueante (es decir, que el proceso que la invoca se queda a la espera de que lleguen datos por el socket), para lograr la sincronización planteada bastaría con que el escritor enviara (*write* o *send*) un byte con cualquier valor por el socket conectado justo después de escribir en el fichero, mientras que el lector debería invocar la llamada para recibir datos del socket conectado (*read* o *recv*) antes de leer del fichero. Evidentemente, de forma previa, ambos procesos han debido establecer la conexión requerida.

Nótese que podría plantearse prescindir del fichero y enviar el dato directamente a través del socket. Si embargo, esa opción cambiaría el escenario propuesto en el enunciado donde el dato, además de transferirse entre los procesos, queda permanentemente almacenado en el fichero.