Sistemas Operativos Avanzados (MUII)
Ejercicio de interbloqueos (14-abril-2020)

Este ejercicio se centra en el tema de los interbloqueos (*deadlocks*) e intenta mostrar de forma aplicada algunas de las técnicas que se estudian en la parte teórica de la asignatura para tratar este problema, centrándose en dos de los escenarios que se revisan en la misma:

- El tratamiento de los interbloqueos internos del sistema operativo.
- El desarrollo de aplicaciones concurrentes libres de interbloqueos.

Para afrontar este trabajo se proporciona como material de apoyo un documento que realiza un estudio en profundidad del problema de los interbloqueos en el ámbito de los sistemas operativos. Asimismo, están publicadas en la página web de la asignatura unas transparencias asociadas a dicho documento, que no cubren los aspectos teóricos avanzados presentados en el mismo, los cuales no son necesarios para acometer este ejercicio. Dentro de ese documento, son especialmente relevantes para este ejercicio las secciones 1.10.1 y 1.11.2, que se centran, respectivamente, en el tratamiento del interbloqueo en los dos escenarios planteados en el mismo.
En cualquier caso, para entender todos los conceptos necesarios para abordar este trabajo, se puede requerir revisar otras partes del documento. Para facilitar esta labor teniendo en cuenta la longitud apreciable del documento, antes de presentar el ejercicio en sí, se va a realizar un breve recorrido por las distintas secciones del documento.

Breve guía del documento

En la primera sección se plantea el problema del interbloqueo de una manera informal, mostrando varios ejemplos de la vida cotidiana, presentando la diferencia entre el *deadlock* y el *livelock* e identificando ya las tres estrategias para afrontar este problema: la detección y recuperación, la prevención y la predicción.
La segunda sección plantea diversos escenarios de interbloqueos, ya en el ámbito de la informática.
La tercera sección presenta una definición del interbloqueo especificando las cuatro condiciones necesarias para que se produzca (condiciones de Coffman).
A continuación, se plantea un modelo formal para poder tratar el interbloqueo, pero se realiza de forma incremental, lo que permite adaptarse a las necesidades de cada tipo de lector.
Así, en la cuarta sección se especifica un modelo básico que limita tanto las características de los recursos involucrados como el tipo de operaciones disponibles para manejarlos, pero que recoge todos los requisitos que se presentan en un sistema real. Este modelo representa el sistema con un grafo de asignación de recursos, siendo condición necesaria y suficiente la presencia de un ciclo en el grafo para que exista un interbloqueo.
La quinta sección expone en detalle (más de 20 páginas) un modelo extendido, que contempla un sistema más rico, con una mayor variedad en los tipos de recursos tratados y en las operaciones de gestión de los mismos. Esta extensión tiene un interés teórico, pero no es especialmente relevante para abordar este ejercicio. De hecho, las transparencias publicadas para este tema no recogen el detalle de este modelo extendido.
La sexta sección retoma las tres estrategias usadas para tratar los interbloqueos (la detección y recuperación, la prevención y la predicción), que se estudiarán en las siguientes secciones.
Por lo que se refiere a las técnicas de detección y recuperación, presentadas en la séptima sección, se basan en detectar el interbloqueo usando alguna de las soluciones descritas en las secciones previas y deshacerlo para recuperarse del mismo. Este tipo de técnicas no suelen aplicarse a los escenarios en los que se centra este ejercicio, aunque sí se usan en otros contextos. Así, dentro de la asignatura se utilizarán para afrontar la práctica optativa de tratamiento de interbloqueos en el *minikernel*.
En la octava sección, se exponen las técnicas de prevención, que impiden *a priori* la existencia de un interbloqueo, y que son las que se usarán en este ejercicio.
La novena sección describe las técnicas de prevención, entre las que se incluye el clásico algoritmo del banquero, pero que son solo de interés teórico dada su difícil aplicación en un sistema real, no siendo en absoluto relevantes para acometer el ejercicio.
La décima sección explica cómo los sistemas operativos reales tratan el problema del interbloqueo tanto en lo que se refiere a los recursos internos del sistema operativo como a los que utilizan los usuarios. La subsección 1.10.1 trata sobre estos primeros y es la que proporciona principalmente la información necesaria para abordar el primer escenario.
Con respecto a la undécima sección, presenta una serie de pautas a tener en cuenta a la hora de diseñar aplicaciones concurrentes que estén libre de interbloqueos, siendo especialmente de interés para este ejercicio la sección 1.11.2 que se ocupa de estudiar este problema en las aplicaciones concurrentes basadas en *mutex*, tal como corresponde al segundo escenario.

Primera parte: Tratamiento de los interbloqueos internos del sistema operativo

Para esta primera parte del ejercicio, retomamos el mecanismo de *spinlocks* de Linux analizado en el ejercicio de sincronización. Este primer ejercicio consiste en resolver el problema 54 del documento proporcionado a los alumnos.

Por comodidad, se incluya a continuación una copia del mismo. Para afrontar este ejercicio, son especialmente relevantes las secciones “Sincronización mediante operaciones no bloqueantes” y “Sincronización en el tratamiento de interrupciones”. A continuación, se incluye el enunciado de este primer ejercicio.

El sistema operativo Linux usa el mecanismo de *spinlocks* para resolver los problemas de sincronización internos en sistemas multiprocesador. Para resolver la sincronización entre la ejecución de la rutina de tratamiento de una interrupción y cualquier otra actividad del sistema operativo, proporciona la función `spin_lock_irq` que, además de intentar obtener el cerrojo no bloqueante solicitado, inhabilita las interrupciones (`local_irq_disable`) y la expulsión de los procesos (`preempt_disable`). De forma complementaria, la función `spin_unlock_irq`, junto con la liberación del cerrojo no bloqueante especificado, rehabilita las interrupciones (`local_irq_enable`) y la expulsión de los procesos (`preempt_enable`). El ejercicio plantea analizar cómo influiría el orden en que se realizan las tres acciones requeridas por cada función en la posible aparición de interbloqueos.
A continuación, se muestra la implementación interna real de estas operaciones en una determinada versión de Linux. Analice qué ocurriría desde el punto de vista de los interbloqueos si en la función `spin_lock_irq` las dos últimas sentencias, que realizan la operación sobre el *spinlock* propiamente dicha, se ubican al principio de la función y qué sucedería si se colocan entre las operaciones de inhabilitar las interrupciones y la expulsión de procesos. Repita el análisis para la función `spin_unlock_irq`.

https://elixir.bootlin.com/linux/v5.6.3/source/include/linux/spinlock_api_smp.h#L124

```
static inline void __raw_spin_lock_irq(raw_spinlock_t *lock)
{
	local_irq_disable();
	preempt_disable();
	spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
	LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
}

static inline void __raw_spin_unlock_irq(raw_spinlock_t *lock)
{
	spin_release(&lock->dep_map, _RET_IP_);
	do_raw_spin_unlock(lock);
	local_irq_enable();
	preempt_enable();
}
```

Segunda parte: Desarrollo de aplicaciones concurrentes libres de interbloqueos

Esta segunda parte pretende aplicar algunas pautas que se exponen en el documento proporcionado de cara a desarrollar aplicaciones concurrentes libres de interbloqueos. Concretamente, nos vamos a centrar en aplicaciones *multithread* que usan cerrojos de tipo *mutex*, por lo que será especialmente relevante la subsección del documento titulada “Ejemplo con *threads* y cerrojos de tipo *mutex*” dentro de la sección 1.11.2. Este ejercicio corresponde al problema 59 del documento proporcionado a los alumnos. Por comodidad, se incluye a continuación una copia del mismo.

Considere una aplicación *multithread* que usa una estructura en árbol tal que cada nodo del árbol tiene el siguiente contenido:

```
struct nodo {
        pthread_mutex_t mutex; // para actualizarlo en exclusión mutua
        struct nodo *padre;
        int nhijos;
        struct nodo **hijo;
        int valor;
};
```

Esta aplicación implementa dos funciones (`donar_hijos` y `pedir_padre`) que permiten, respectivamente, que un nodo reparta entre los hijos de forma proporcional una cierta cantidad del valor que almacena y que un nodo tome prestado del padre una cierta cantidad del valor que tiene este guardado.

```
void donar_hijos(struct nodo *padre, int cantidad) {
        pthread_mutex_lock(&padre->mutex);
        if (padre->nhijos && cantidad<=padre->valor) {
                for (int i=0; i<padre->nhijos; i++)
                        pthread_mutex_lock(&padre->hijo[i]->mutex);
                for (int i=0; i<padre->nhijos; i++)
```

```
                        padre->hijo[i]->valor+=cantidad/padre->nhijos;
                padre->valor-=cantidad;

                for (int i=0; i<padre->nhijos; i++)
                        pthread_mutex_unlock(&padre->hijo[i]->mutex);
        }
        pthread_mutex_unlock(&padre->mutex);
}
void pedir_padre(struct nodo *hijo, int cantidad) {
        if (hijo->padre)  {
                pthread_mutex_lock(&hijo->mutex);
                pthread_mutex_lock(&hijo->padre->mutex);
                if (hijo->padre->valor>=cantidad) {
                        hijo->padre->valor-=cantidad;
                        hijo->valor+=cantidad;
                }
                pthread_mutex_unlock(&hijo->padre->mutex);
                pthread_mutex_unlock(&hijo->mutex);
        }
}
```

Se pide responder a las siguientes cuestiones:

a) Analice si pueden producirse interbloqueos entre las siguientes operaciones y, en caso afirmativo, describa un ejemplo que implique a dos *threads* especificando la traza de ejecución conflictiva: (i) dos operaciones de donación, (ii) dos operaciones de préstamo y (iii) una operación de préstamo y una donación.

b) Extienda los casos de interbloqueo a escenarios con más de dos niveles.

c) Describa cómo se podrían evitar los interbloqueos.

d) Proponga una pauta para evitar los interbloqueos en escenarios de acceso concurrente a estructuras en árbol (el sistema de ficheros sería un ejemplo de este tipo de escenarios).

La respuesta a esas cuestiones se debe incluir en la memoria de la práctica, pero, además, algunas de ellas se van a reflejar en fragmentos de código que se deben añadir a los ficheros que se proporcionan como material de apoyo en el fichero `interbloqueos.tgz`:

- `arbol.h`: contiene la declaración del nodo.
- `arbol.c`: incluye la implementación de las funciones y en el mismo **debe incorporar** en la ubicación adecuada una operación `sleep` que permita apreciar los interbloqueos y, finalmente, la solución planteada para resolver el problema.
- `main.c`: contiene código que usa las funciones de acceso a la estructura de árbol y en el mismo **se debe incluir** uno de los ejemplos de interbloqueos con dos *threads* (el programa principal y un *thread* creado) identificados en su respuesta al ejercicio.

Entrega del ejercicio

El plazo de entrega del trabajo es el 25 de mayo de 2020.

La entrega se realiza en *triqui* ejecutando el mandato:

```
entrega.soa interbloqueos.2020
```

Este mandato recogerá del directorio `~/DATSI/SOA/interbloqueos.2020` los siguientes ficheros:

- `autor.txt`: con los datos del alumno.
- `memoria.pdf`: que debe incluir la solución del ejercicio, del directorio
- `arbol.c`: que incluya las incorporaciones solicitadas.
- `main.c`: con las incorporaciones pedidas.