Sistemas Operativos Avanzados (MUII)
Ejercicio sobre interbloqueos (28 de abril de 2021)

Planteamiento del trabajo

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 (sección 7.10.1 del segundo volumen del libro recomendado).
- El desarrollo de aplicaciones concurrentes libres de interbloqueos (sección 7.11.2 del segundo volumen del libro recomendado).

Primera parte: Tratamiento de los interbloqueos internos del sistema operativo

Para esta primera parte del ejercicio, que corresponde al problema 54 del libro recomendado, retomamos el mecanismo de *spinlocks* de Linux.

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 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. En ese fragmento, se han identificado con un número las sentencias de cada función, considerando como una única sentencia las líneas coloreadas en azul. Analice qué ocurriría en cada uno de los siguientes escenarios de reordenamiento de sentencias identificando qué trazas de ejecución causarían interbloqueos:

1. Orden de ejecución 2, 3 y 1.
2. Orden de ejecución 1, 3 y 2.
3. Orden de ejecución 5, 4 y 6.
4. Orden de ejecución 6, 4 y 5.

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();                                  // 1
      preempt_disable();                                    // 2
      spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);         // 3
      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_);               // 4
      do_raw_spin_unlock(lock);
      local_irq_enable();                                   // 5
      preempt_enable();                                     // 6
}
```

Segunda parte: Desarrollo de aplicaciones concurrentes libres de interbloqueos

Esta segunda parte del ejercicio, que corresponde al problema 59 del libro recomendado, pretende aplicar algunas pautas estudiadas de cara a desarrollar aplicaciones concurrentes libres de interbloqueos. Concretamente, nos vamos a centrar en aplicaciones *multithread* que usan cerrojos de tipo *mutex*.

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 proporcionan como material de apoyo el fichero `interbloqueos.tgz` con el siguiente contenido:

- `arbol.h`: contiene la declaración del nodo.
- `arbol.c`: incluye la implementación de las funciones anteriormente descritas.
- `main.c`: contiene código con un ejemplo que usa las funciones de acceso a la estructura de árbol.

Se pide responder a las siguientes cuestiones:

5. 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.

6. Incluya en los puntos identificados del fichero `main.c` (marcados con el comentario "operación conflictiva") `sendas llamadas a las funciones del árbol de manera que se produzca un interbloqueo, tal como se identificó en el punto anterior. Para asegurarse de que se produzca el interbloqueo incluya un `sleep` en la ubicación adecuada del fichero `arbol.c`. Debe entregar esta versión como los ficheros `arbol1.c` y `main1.c`.

7. Extienda los casos de interbloqueo describiendo un escenario con más de dos *threads*.

8. Describa cómo se podrían evitar los interbloqueos.

9. Incluya en `arbol.c` esa estrategia, que entregará como el fichero `arbol2.c`.

10. 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).

Entrega del ejercicio

El plazo de entrega del trabajo es el 8 de junio de 2021.

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

```
entrega.soa interbloqueos.2021
```

Este mandato recogerá del directorio `~/DATSI/SOA/interbloqueos.2021` los siguientes ficheros:

- `autor.txt`: con los datos del alumno.
- `memoria.pdf`: que debe incluir la solución del ejercicio, del directorio
- `main1.c`: con las incorporaciones pedidas.
- `arbol1.c`: que incluya las incorporaciones solicitadas.
- `arbol2.c`: que incluya las incorporaciones planteadas.