SOAD - Laboratorio 3

Análisis previo a las implementaciones

Siguiendo con el mismo código y librería que en la práctica anterior, hemos hecho la implementación de las tres funciones que se nos pedían en la librería.

Previo a esta implementación hemos repasado la librería para ver si ya existía, en otro formato, las funciones que se nos pedían. Hemos podido observar que las dos primeras funciones no están en la librería original que se nos ha proporcionado, ya que no existe una cola de prioridad de los threads ni el propio atributo de prioridad de cada thread. Por tanto, es imposible que existan los mecanismos para hacer lo mismo que hacen las funciones sched_nice(int pri, int th_id) ni sched_yield(void) que lo que hacen es modificar la prioridad de un thread y saltar al thread con mayor prioridad (número de prioridad más bajo) respectivamente.

Por otro lado si hemos podido observar que la función sched_handoff(int th_id) ya existe en la librería de la práctica y que por tanto podemos usarla en la implementación de la función. Esta se llama fiberYield() y lo que hace es cambiar el contexto de la fiber actual a la main o al revés según el caso. El hecho de que se llame Yield puede dar confusión con la recién implementada sched_yield(void), por eso es importante aclarar que hacen cada una de ellas ya que vamos a mantener ambas con los mismos nombres. El objetivo de ambas es ceder el control del procesador a otro thread, la diferencia recae en que thread recibe el uso del procesador. Las dos son compatibles y pese a que su nombre parecido puede llevar a confusión es importante aclarar que no hacen lo mismo.

También como una observación previa, las nuevas implementaciones de sched_nice y sched_yield solo son realmente útiles si detrás tenemos un algoritmo que actualice de manera correcta y eficiente las prioridades de los threads que estamos ejecutando en nuestro programa. Esto es importante porque de nada nos sirve poder modificar la prioridad de un thread si no se va a hacer de manera dinámica según los recursos y el momento de la ejecución y de la misma manera no nos servirá de nada ceder el procesador al thread con mayor prioridad si siempre es el mismo y por tanto el programa se ejecuta de manera secuencial y no paralela.

Tampoco hace mucho sentido la implementación de sched_handoff si el algoritmo que asigna las prioridades es suficientemente bueno y nos permite determinar en cada momento que thread necesita el procesador. De manera que si conseguimos determinar de manera correcta la prioridad no nos hará falta usar sched_handoff y con usar sched_yield tendremos suficiente.

Una vez analizada la librería y las funciones que vamos a implementar vamos a ver cómo se han hecho estas implementaciones y como es su funcionamiento con los programas que preparamos en el laboratorio 2.

Análisis de las implementaciones

Hemos tenido que implementar tres llamadas diferentes para hacer las gestiones de la prioridad de la lista de threads además de los cambios de contexto según la prioridad de cada uno de los threads.

Lo primero que hemos modificado respecto el fichero original de la librería ha sido el struct fiber para poder añadir un identificador y el nivel de prioridad de cada thread, con este pequeño cambio ya tenemos definida la base de la cola de prioridad y una manera de identificar cada thread en cada momento.

En relación a la cola de prioridad hemos definido una serie de funciones básicas que nos permiten gestionar la cola con facilidad y ordenarla en cada momento.

Con la implementación base de la cola de prioridad hecha hemos empezado con las funciones que requería la práctica.

La primera **schd_nice(int pri, int th_id)** que es la función que modifica la prioridad de un thread concreto.

```
int sched_nice(int pri, int th_id) {
    int last_priority = fiberList[th_id].priority;

    int ret = enqueue(th_id, pri);

    return ret == 0 ? last_priority : -1;
}
```

Este es el código que hemos implementado. Una vez inicializamos todos los threads su prioridad es 0, pero cuando hacemos spawnFiber() definimos un número aleatorio entre 1 y 20 y esa es la prioridad asignada a ese thread. De la misma manera cuando saltamos de contexto también modificamos la prioridad del thread otra vez para reiniciar su prioridad y que no sea siempre la misma.

Después de implementar el mecanismo de cambiar la prioridad de los threads del programa hemos implementado las funciones de **schd_yield()** y **sched_handoff(int th_id)**, que realizan básicamente la misma función, la única diferencia reside en que schd_yield salta siempre al thread con mayor prioridad mientras que handoff saltará siempre al thread que le indiquemos con el id del thread.

```
int sched_handoff(int th_id) {
    currentFiber = fiberList[th_id].id;

    if (th_id < 0 && th_id > MAX_FIBERS) return -1;

    LF_DEBUG_OUT1( "Switching to fiber %d", currentFiber );
    inFiber = 1;
    longjmp( fiberList[currentFiber].context, 1 );

    return 0;
}
```

Este es el código implementado para schd_yield, donde definimos la fibra actual con el id que recibe como parámetro y luego saltamos al contexto de esa fibra.

El código de schd_yield es el mismo que había en la librería original en FiberYield, pero con alguna modificación.

```
int sched_yield() {
    /* If we are in a fiber, switch to the main context */
    if ( inFiber )
    {
        /* Store the current state */
        LF_DEBUG_OUT1( "SAVING CONTEXT FOR %d", currentFiber );
        if ( setjmp( fiberList[ currentFiber ].context ) )
        {
            /* Returning via longjmp (resume) */
            LF_DEBUG_OUT1( "Fiber %d resuming...", currentFiber );
        }
        else
        {
            LF_DEBUG_OUT1( "Fiber %d yielding the processor...", currentFiber );
            /* Saved the state: Let's switch back to the main state */
            int r = 1 + (rand() % 21);

            sched_nice(r, fiberList[currentFiber].id);

            longjmp( mainContext, 1 );
        }
    }
```

Hemos añadido el cambio de prioridad de la fibra actual en el salto para evitar saltar otra vez a la misma fibra si no hay ningúna con una prioridad mayor a la de la fibra actual.

```
        }
        else
        {
            /* Saved the state so call the next fiber */

            int thread = peak();
            currentFiber = fiberList[thread].id;
            LF_DEBUG_OUT1( "Switching to fiber %d", currentFiber );
            inFiber = 1;
            LF_DEBUG_OUT1( "NULL? %d", fiberList[currentFiber].context == NULL );
            longjmp( fiberList[currentFiber].context, 1 );
            LF_DEBUG_OUT1( "AFTER JUMPING TO %d", currentFiber );
        }
    }

    return queueIdx;
}
```

Finalmente hemos añadido el mismo código que en schd_handoff pero esta vez saltamos al peak de la cola, que es el thread con mayor prioridad en ese momento.

También hemos modificado la función FiberYield() para que use el código definido en schd_yield() de manera que no hace falta modificar ningún código que usase FiberYield previamente a esta nueva versión de la biblioteca y aseguramos la retrocompatibilidad para los programas antiguos usando nuestra librería.

Testing

Para el juego de pruebas hemos utilizado el código de ejemplo de la sesión anterior donde generamos dos fibras distintas una con la ejecución de la función searchMax() y otra con la función printSum(). Gracias a la retrocompatibilidad que hemos comentado anteriormente no ha sido necesario modificar los archivos del juego de pruebas y hemos podido comprara los resultados con los obtenidos anteriormente.

Los resultados de ejecutar el juego de pruebas de la práctica anterior eran los siguientes.

```
alumne@a5s103pc13:~/Desktop/lab2-ex2$ ./example2-sjlj
Computing sum, current: 0
Search Max It: 0, Current Max=4
Computing sum, current: 4
Search Max It: 1, Current Max=4
Computing sum, current: 9
Search Max It: 2, Current Max=5
Computing sum, current: 11
Search Max It: 3, Current Max=5
Computing sum, current: 19
Search Max It: 4, Current Max=8
Computing sum, current: 21
Search Max It: 5, Current Max=8
Computing sum, current: 72
Search Max It: 6, Current Max=51
Computing sum, current: 114
Search Max It: 7, Current Max=51
Computing sum, current: 192
Search Max It: 8, Current Max=78
Computing sum, current: 290
Search Max It: 9, Current Max=98
291
98
alumne@a5s103pc13:~/Desktop/lab2-ex2$
```

Y los resultados obtenidos con el juego de pruebas en la librería actual son los siguientes.

```
Computing sum, current: 0
Search Max It: 0, Current Max=4
Computing sum, current: 4
Search Max It: 1, Current Max=4
Computing sum, current: 9
Search Max It: 2, Current Max=5
Computing sum, current: 11
Search Max It: 3, Current Max=5
Computing sum, current: 19
Computing sum, current: 21
Computing sum, current: 72
Computing sum, current: 114
Computing sum, current: 192
Computing sum, current: 290
291
Search Max It: 4, Current Max=8
```

Podemos ver cómo cambiando la prioridad de cada thread de manera aleatoria hemos cambiado el output, que ya no es intercalado sino que ahora salta según las prioridades asignadas.

Conclusión

Gracias a esta práctica podemos ver la importancia de usar la retrocompatibilidad con nuestras librerías pues de no haberse usado en este caso tendríamos que haber modificado el código del juego de pruebas y cualquier otro usuario usando nuestra librería se hubiera visto afectado, pues sus programas anteriores hubieran funcionado de manera diferente.

También es importante remarcar que el algoritmo que decide qué prioridad recibe cada thread en cada momento para controlar la ejecución a nuestra conveniencia y evitar que se ejecuten partes del código en momentos que no tocan.