

SOAD - Laboratorio 2

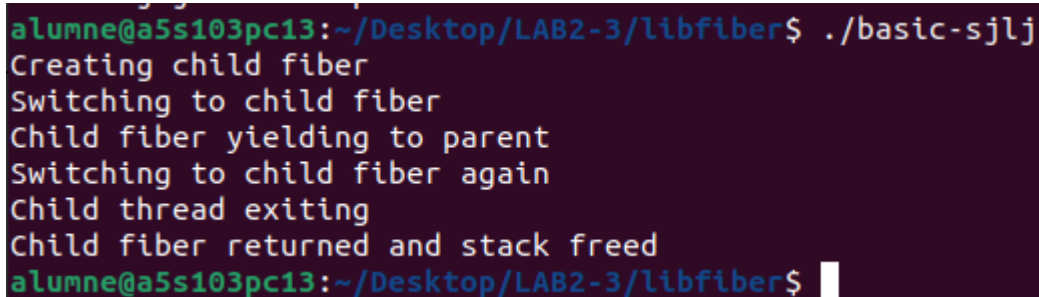
Hemos elegido analizar la versión sjlj de la librería libfiber. Esta librería define distintas operaciones que podemos usar para generar y gestionar las fibers del programa. Estas operaciones són la implementación de las funciones que tenemos en el documento libfiber.h, la diferencia con libfiber-uc.c es que libfiber-sjlj.c implementa las funciones con las instrucciones setjmp() y longjmp(). En libfiber-uc.c usamos swapcontext() en su lugar.

Para ver mejor cómo funciona esta librería implementada con setjmp y longjmp hemos analizado los dos programas que vienen adjuntos a la práctica 2-3, basic-sjlj.c y example.c.

basic-sjlj.c

En este primer programa generamos una fiber hijo y saltamos entre la ejecución de esta y la ejecución del padre.

Una vez ejecutamos el programa podemos ver el siguiente output.

A terminal window with a dark purple background and light green text. The prompt is 'alumne@a5s103pc13:~/Desktop/LAB2-3/libfiber\$'. The command './basic-sjlj' has been executed, resulting in the following output: 'Creating child fiber', 'Switching to child fiber', 'Child fiber yielding to parent', 'Switching to child fiber again', 'Child thread exiting', 'Child fiber returned and stack freed'. The prompt is shown again at the end.

```
alumne@a5s103pc13:~/Desktop/LAB2-3/libfiber$ ./basic-sjlj
Creating child fiber
Switching to child fiber
Child fiber yielding to parent
Switching to child fiber again
Child thread exiting
Child fiber returned and stack freed
alumne@a5s103pc13:~/Desktop/LAB2-3/libfiber$
```

Podemos observar que una vez creada la fiber hijo saltamos a ejecutar con esta y luego devolvemos a la ejecución al padre, para volver al hijo y terminar con la ejecución de este. Antes de terminar su ejecución la fiber hijo libera la pila y acaba la ejecución del programa.

En este programa redefinimos el signal de usuario 1 para saltar al fiber hijo y que este ejecute una función sencilla que escribe por pantalla y luego salta otra vez al padre.

Básicamente este pequeño programa nos permite saltar de la ejecución del padre a la ejecución del hijo y viceversa varias veces. Una vez creamos la fibra hijo en las primeras líneas de la función main() y luego llamamos al signal SIGUSR1 para hacer el cambio de contexto a la fibra hijo. Todo esto lo hacemos con las instrucciones setjmp() y longjmp().

Con la instrucción setjmp(struct) lo que hacemos es guardar el contexto actual del programa en el struct y devolvemos 0. De esta manera podemos saber si estamos en el hijo o no, si el valor es 0 estamos en el padre y si retorna diferente de 0 podemos suponer que estamos en algún hijo, funciona de la misma manera que lo hace el fork() que se usa en los laboratorios de Sistemas operativos.

Una vez hemos guardado el contexto del padre podemos pasar a ejecutar la fibra hijo y una vez finalice su parte volver al contexto del padre con un `longjmp(struct ret)` donde el struct es el contexto al que queremos volver y ret el valor que queremos que retorne esta función. `Longjmp` vuelve al punto donde se encontraba el contexto que hemos guardado y devuelve el valor como si fuera un `setjmp`, de la misma manera que en `setjmp` gracias al valor de vuelta de esta función podemos saber en qué contexto nos encontramos y de dónde venimos.

Gracias al programa `basic-sjlj.c` podemos ver como funciona exactamente `setjmp` y `longjmp` y que nos permite hacer.

example.c

El programa `example.c` nos permite ejecutar las funciones que hemos visto definidas en la librería `libfiber`. Podemos ejecutar este programa con las diferentes versiones de las librerías que se nos han facilitado en la práctica. En este caso probaremos de ejecutar el ejemplo con la librería implementando `sjlj` y luego con `uc` y veremos si hay alguna diferencia.

Antes de las ejecuciones vamos a analizar el código que nos ofrece `example.c`.

El programa implementa tres funciones muy sencillas, `squares`, `fibonacci` y `fiber1`. Estas funciones calculan el cuadrado del 1 al 10, el número de fibonacci des de 2 fins a 15 y escribir por pantalla.

Cada una de estas funciones será ejecutada por una fiber distinta y se irá alternando la ejecución gracias a la función `fiberYield()` de la librería.

```
46 int main()
47 {
48     /* Initialize the fiber library */
49     initFibers();
50
51     /* Go fibers! */
52     spawnFiber( &fiber1 );
53     spawnFiber( &fibonacci );
54     spawnFiber( &squares );
55
56     /* Since these are nonpre-emptive, we must allow them to run */
57     waitForAllFibers();
58
59     /* The program quits */
60     return 0;
61 }
```

Antes de generar las fibers concretas inicializamos todas las fibers que tendrá nuestro programa, las definidas en la librería. Una vez hemos generado todas las fibers podemos hacer el spawn con cada una de las funciones que necesitamos que ejecuten.

Después de hacer el spawn de las fibers usamos la función `waitForAllFibers()` para asegurar que acaban todas las fibers antes de finalizar el programa.

Entonces las funciones principales que usamos en example.c para ser concurrentes són `fiberYield()` y `waitForAllFibers()`.

El resultado de ejecutar example.c con la librería `sjlj` es.

```
alumine@a5s103pc13:~/Desktop/LAB2-3/libfiber$ ./example-sjlj
Hey, I'm fiber #1: 0
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
0*0 = 0
Hey, I'm fiber #1: 1
fibonacci(3) = 2
1*1 = 1
Hey, I'm fiber #1: 2
fibonacci(4) = 3
2*2 = 4
Hey, I'm fiber #1: 3
fibonacci(5) = 5
3*3 = 9
Hey, I'm fiber #1: 4
fibonacci(6) = 8
4*4 = 16
fibonacci(7) = 13
5*5 = 25
fibonacci(8) = 21
6*6 = 36
fibonacci(9) = 34
7*7 = 49
fibonacci(10) = 55
8*8 = 64
fibonacci(11) = 89
9*9 = 81
fibonacci(12) = 144
fibonacci(13) = 233
fibonacci(14) = 377
```

Y el resultado de ejecutar example.c con la librería uc es.

```
alunne@a5s103pc13:~/Desktop/LAB2-3/libfiber$ ./example-uc
Hey, I'm fiber #1: 0
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
0*0 = 0
Hey, I'm fiber #1: 1
fibonacci(3) = 2
1*1 = 1
Hey, I'm fiber #1: 2
fibonacci(4) = 3
2*2 = 4
Hey, I'm fiber #1: 3
fibonacci(5) = 5
3*3 = 9
Hey, I'm fiber #1: 4
fibonacci(6) = 8
4*4 = 16
fibonacci(7) = 13
5*5 = 25
fibonacci(8) = 21
6*6 = 36
fibonacci(9) = 34
7*7 = 49
fibonacci(10) = 55
8*8 = 64
fibonacci(11) = 89
9*9 = 81
fibonacci(12) = 144
fibonacci(13) = 233
fibonacci(14) = 377
```

El resultado es exactamente el mismo, esto se debe a que en la librería de sjlj usamos las instrucciones setjmp y longjmp para salvar y restaurar el contexto de nuestro programa y en la librería uc se usa directamente swapcontext para cambiar entre los contexto que necesitamos.

Gracias a example.c podemos ver como se usan las funciones fiberYield y waitForAllFibers para tener concurrencia entre fibers y que impacto tienen en nuestro programa.

example2.c

De la misma manera que en example.c hemos definido dos funciones distintas, la primera searchMax() que calcula el máximo de una array y la segunda printSum() que escribe por pantalla la suma de todos los elementos del array.

Usamos la misma estructura que en example.c para ejecutar dos fibers cada una con las funciones que hemos definido y usamos el `waitForAllFibers()` para esperar a que finalicen la ejecución todas las fibers antes de terminar el programa.

Vamos a ver los resultados de ejecutar example2.c con la libreria `sjlj` y luego con la libreria `uc`.

El resultado de ejecutar example2.c con la libreria `sjlj` es.

```
alumine@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
alumine@a5s103pc13:~/Desktop/lab2-ex2$
```

El resultado de ejecutar example2.c con la librería uc es.

```
alunne@a5s103pc13:~/Desktop/lab2-ex2$ ./example2-uc
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
alunne@a5s103pc13:~/Desktop/lab2-ex2$
```

De la misma manera que en example.c podemos ver que la ejecución con ambas librerías son idénticas. El motivo de esto sigue siendo el mismo las instrucciones setjmp y longjmp son equivalentes a swapcontext y nos permiten salvar y cambiar el contexto de ejecución en nuestros programas.

Conclusión

La conclusión a la que llegamos con esta práctica es que las librerías sjlj y uc son equivalentes para programas de un tamaño tan pequeño como los que hemos probado.

Gracias a las funciones que nos proporciona la librería somos capaces de generar programas que usan y gestionan fibers y podemos cambiar los contextos entre las distintas fibers para obtener la ejecución concurrente de estos fibers.

Las librerías usadas (sjlj, uc) són equivalentes en cuanto a las instrucciones de cambio de contexto que hemos analizado, pero no en el resto de elementos que las forman, ya que no los hemos analizado.

En definitiva son dos librerías muy útiles que nos ayudan a la hora de hacer programas distribuidos donde parte del código no está en los propios ficheros fuente.