SOAD-Laboratorio 2

Estudio de las librerías de threads

(versión clone descartada porque no compila)

libfiber.h/libfiber-sjlj.c

Tras haber analizado, compilado y ejecutado los programas que contenía el paquete libfiber, me he decidido por estudiar la opción de la librería de threads libfiber.h/libfiber-sjlj.c. Inicialmente, vamos a analizar el código de libfiber-sjlj.c, que implementa las funciones de libfiber.h y comentaremos las diferencias que supone lo que es el código equivalente sin estar organizado en una librería.

libfiber-sjlj.c es un programa que implementa el uso de hilos.

El código define una estructura llamada fiber que contiene los siguientes campos:

- context: un objeto de tipo jmp_buf, que almacena el contexto de ejecución de la fibra.
- function: un puntero a una función que será ejecutada cuando la fibra comience a ejecutarse.
- active: un entero que indica si la fibra está activa o no.
- stack: un puntero a la pila de la fibra.
- stackId: un identificador de pila.

También hay una serie de variables globales que se utilizan para administrar las fibras:

- fiberList: un arreglo de fibras que representa la "cola" de fibras.
- currentFiber: un entero que indica el índice de la fibra actualmente en ejecución.
- inFiber: un booleano que indica si estamos en la fibra principal o en una fibra secundaria.
- numFibers: un entero que indica el número de fibras activas.

El código también define varias funciones:

- initFibers: inicializa el arreglo de fibras.
- spawnFiber: crea una nueva fibra y la agrega a la cola de fibras.
- fiberYield: hace que la fibra actual ceda el control a otra fibra en la cola (si hay alguna) o vuelva a la fibra principal.

En general, la idea es que cada fibra tenga su propia pila y contexto de ejecución. Cuando una fibra llama a fiberYield, la función almacena su contexto actual en su objeto jmp_buf y luego invoca longjmp en el objeto jmp_buf de la siguiente fibra en la cola. Cuando la siguiente fibra comienza a ejecutarse, restaura su contexto almacenado previamente. De esta manera, la ejecución se alterna entre las diferentes fibras.

Por otro lado, el código equivalente que no está organizado como una librería se diferencia principalmente por el hecho de tener que escribir el código para todas las funciones, algo más complejo, largo y peor estructurado. No obstante, el programador tiene un mayor control; si se escribe todo el código en lugar de utilizar una librería, se tiene un mayor control sobre el funcionamiento del programa y se pueden realizar ajustes personalizados más fácilmente. Además, al no usar una librería se puede ahorrar espacio; si se tiene un proyecto pequeño, no se necesitarán muchas funciones, por lo que una librería puede no ser necesaria y no utilizarla podría reducir el espacio de almacenamiento y en cuanto a gestión de memoria se refiere, el programador también puede llevar a cabo la que sea más óptima por su cuenta. Finalmente, no organizar el código como una librería implica menor dependencia del código con la librería en particular y no hay la necesidad de actualizarla en el futuro.

Código de prueba de la librería

A continuación, probaré la librería que he escogido para poner a prueba sus funcionalidades y familiarizarme más con ella con un senzillo programa en C:

```c
#include "libfiber.h"
#include <stdio.h>
#include <stdlib.h>

#include <unistd.h>

int suma = 0;

void uno()
{
        printf("Residuo 3 doy 0\n");
        suma = suma + 1;
}

void dos()
{
        printf("Residuo 3 doy 1\n");
        suma = suma + 1;
}

void tres()
{
        printf("Residuo 3 doy 2\n");
        suma = suma + 1;
}

void ultimo()
{
        printf("Soy el que han creado tras la muerte de los demas y vengo a decir la suma total que han realizado: %d", suma);
}

int main(int argc, char* argv[])
{
        if (argc != 2) {
        printf("Usage: %s <number of threads>\n", argv[0]);
        return 1;
        }

        if (atoi(argv[1]) > 10) {
        printf("Number of threads cannot be greater than 10\n");
        return 1;
        }

        int num_threads = atoi(argv[1]);
        /* Initialize the fiber library */
        initFibers();

        for (int i = 0; i < num_threads; i++) {

           if (i % 3 == 0) {spawnFiber( &uno );}
           else if (i % 3 == 1) {spawnFiber(&dos);}
           else if (i % 3 == 2) {spawnFiber(&tres);}
        }

        /* Since these are nonpre-emptive, we must allow them to run */
        waitForAllFibers();
        spawnFiber( &ultimo);
        waitForAllFibers();


        /* The program quits */
        return 0;
}
```

Fig. 1: programa de prueba example-Xavi.c

Concretamente el código que he creado recibe un parámetro que es el número de hilos a crear, que no puede ser mayor a 10 porque initFibers() inicializa un array para 10 hilos. A continuación, los crea y cada uno, según si su número de iteración residuo 3 da 0, 1 o 2, tiene una función asociada que muestra este resultado y además actualiza una variable global suma. Al acabar la creación, introduzco un waitForAllFibers() para esperar a que acabe la ejecución de todos ellos y finalmente

creo otro que dice que es el último y saca el valor de la suma que es igual al valor de hilos creados anteriormente.

Pienso que el programa es interesante por varios motivos: cuando obtenemos la salida vemos que el orden de ejecución de los hilos no sigue el orden del bucle, sino que pueden tener un orden diferente en la cola para la ejecución, y todos ellos contribuyen a actualizar una variable global que se encuentra en el main, lo que refleja que se pueden repartir el trabajo. Además, demuestra que podemos usar waitForAllFibers() como herramienta de sincronización, si queremos que un hilo se ejecute estrictamente después de que haya acabado la ejecución de otro hilo o conjunto de hilos.

Estudio del código de ejemplo

Vamos a analizar más a fondo las funciones que llama el código de ejemplo con la implementación de la librería que escogida. Concretamente analizaremos en qué funciones se llevan a cabo los cambios de contexto y qué ocurre entre las llamadas de cambio, la parte del código donde se producen y las estructuras de datos y variables relacionadas para mantener la información necesaria:

Las únicas funciones donde se llama a setjmp y longjmp son fiberYield (función llamada desde fiber1, fibonacci y squares de example.c y waitForAllFibers de libfiber-sjlj.c), y usr1handlerCreateStack de libfiber-sjlj.c, que es el manejador de señales de las fibras.

La estructura fiber contiene un salto (jmp_buf) que guarda el contexto de ejecución de una fibra, un puntero a una función que se ejecutará en la fibra, un indicador de si la fibra está activa o no, y un puntero al stack de la fibra, información clave y suficiente para llevar a cabo el control de los cambios de contexto.

Lo primero que se hace es llamar a initFibers(), que inicializa la estructura fiber. Luego, se crean y se agregan a la lista de fibras nuevas fibras con la función

spawnFiber, concretamente 3: una con la función de entrada fiber1, otra con la función de entrada square y otra con la función fibonacci.
En la creación de cada una de ellas, la función sigaction() se utiliza para instalar un manejador de señales que se activa cuando salta un SIGUSR1. Cuando se levanta la señal, el manejador cambia al stack de la fibra y guarda el contexto actual en la estructura fiberList[numFibers].context. Luego, la función func se ejecuta en la fibra y, cuando se completa, se retorna al contexto guardado y se devuelve el control al hilo principal.

Por último cabe mencionar la función fiberYield, que es la que se encarga de, si se llama desde una fibra, salvar el contexto actual en la estructura fiberList[currentFiber].context y retornar al hilo principal, y si se llama desde el hilo principal, cambiar al contexto de la siguiente fibra activa en la lista de fibras.

¿Cómo hacer que el flujo de acciones a realizar cuando hay un cambio de contexto queden localizadas en funciones específicas de la librería?

Para que los cambios de contexto pudieran ser gestionados desde una librería, se podría diseñar una librería que proporcionase funciones que encapsulasen toda la complejidad de la gestión de contexto y que permitiesen al programador hacer uso de ellas de manera transparente. La librería ocultaría los detalles de la implementación de los cambios de contexto y proporcionaría una interfaz simple para el programador que vendría a ser como una API.

Esta API podría incluir funciones para crear nuevos contextos, cambiar de contexto y liberar recursos asociados con los contextos con funciones del tipo:

- context_t *create_context(void (*func)(void *), void *arg, size_t stack_size): Esta función crea un nuevo contexto, donde func es un puntero a una función que se ejecutará en el contexto creado y arg es un puntero a los argumentos de la función. stack_size es el tamaño de la pila que se asignará al contexto.

- void switch_context(context_t *from, context_t *to): Esta función cambia de contexto, desde from a to.
- void destroy_context(context_t *context): Esta función libera los recursos asociados con un contexto creado anteriormente.

En pocas palabras, hemos cambiado la utilización de setjmp y longjmp por la utilización que ya habíamos visto de la estructura context_t que nos proporciona C ya que a mi modo de ver permite crear una API más práctica para el usuario. Básicamente, lo que el programador podría hacer es crear contextos, y cuando quiera hacer un cambio y ejecutar otro, usar switch_context para realizar el cambio y que se ejecute la función especificada en el create_context. Finalmente, cuando quisiera liberar los recursos asociados a los contextos podría hacer uso de destroy_context.