SOAD-Laboratorio 3

En este laboratorio he seguido trabajando con la librería libfiber.h/libfiber-sjlj.c. Concretamente hemos tenido que añadir a la librería una planificación de threads basada en prioridades fijas y cesión voluntaria del procesador. Para ello, se nos pedía implementar las siguientes llamadas a librería, la declaración de las cuales he añadido en libfiber.h y cuya implementación he añadido en libfiber-sjlj.c:

- **int sched_nice(int pri, int th_id):** el thread master cambia la prioridad del thread th_id por la prioridad pri y se reordena la cola de threads en consecuencia. Si el parámetro pri pasado es 0, devuelve la prioridad actual del thread y no hace nada. Si th_id es cero, el thread al que se le va a cambiar la prioridad es el actual. Si se le pasa un valor de prioridad no válido (he considerado el rango 1-100), un valor de th_id no válido (he considerado el rango 1-numFibers) o el thread que llama la función no es el master devuelve -1. En cambio, si todo ha ido bien, se retorna la prioridad antigua del thread th_id.

- **int sched_yield(void):** cede el procesador al thread más prioritario (aquel que está primero en la cola de threads) a menos de que el más prioritario ya sea el que lo tiene. Si el actual es el poseedor o estamos en el contexto del master y no existen threads en la cola, no se hace nada y se retorna 0 (ya que no lo he considerado como un error, sino que no se tiene que hacer nada).

- **int sched_handoff(int th_id):** cede el procesador al thread th_id a menos de que ya sea el que lo tiene y aunque tenga menor prioridad que otros. Como antes, si el th_id pasado no es válido, no se encuentra entre 1 y numFibers, se produce un error y retorna -1. En el caso de que todo vaya bien o no se haga nada, retorna 0.

Además, he añadido una función que me calcula el índice de la cola dónde se encuentra un thread determinado para poder saber en las otras funciones qué posición del vector fiberList he de tratar:

- **int search(int th_id):** devuelve en qué índice de la cola se encuentra el thread th_id. Si no lo encuentra (algo que no se va a dar) retorna -1.

Estudiando las funcionalidades de planificación que ofrece ya la librería, he visto que para la implementación de sched_nice debería programar yo un algoritmo para colocar donde toque los threads cuando se les cambia la prioridad, pero que en cambio, para las dos otras funciones (sched_yield y sched_handoff) podría reutilizar bastante código de fiberYield, ya que implementa cambios de contexto que es lo que queremos nosotros pero personalizados.

La diferencia que hay entre fiberYield, sched_yield y sched_handoff es que fiber_yield lo que hace es que si se llama desde una fibra se devuelve el control al contexto principal, mientras que si se llama desde el contexto principal se devuelve el control a la siguiente fibra activa de la cola de fibras, y en cambio sched_yield y sched_handoff permiten cambiar de contexto tanto desde una fibra como desde el contexto principal a otra fibra, en el primer caso a la fibra más prioritaria, y en el segundo a la que se quiera. Por lo tanto, considero que son funcionalidades lo suficientemente útiles y diferentes como para no incluir ninguna dentro de otra, y por ello yo las mantendría todas.

A continuación, procedo ya a explicar cómo he implementado las funciones y qué atributos he añadido para mantener la coherencia:

sched_nice(int pri, int th_id)

Primero de todo, he identificado que necesitaría añadir dos parámetros al struct fiber; un parámetro de id y otro de prioridad, para poder guardar el id de cada thread y su prioridad.

```
typedef struct
{
        jmp_buf context;
        void (*function)(void);
        int active;
        void* stack;
        int priority;
        int id;
#ifdef VALGRIND
        int stackId;
#endif
} fiber;
```

De este modo, podremos identificar threads y ordenarlos por prioridad (las más altas son las de menor valor). En cuanto a la inicialización de las fibras, he optado por asignarles a todas inicialmente una prioridad de 20 y un id igual a su posición + 1, pues el vector de fibras fiberList es el que nos las almacenará de más prioritarias a menos prioritarias, funcionando como una cola.

```
void initFibers()
{
        int i;
        for ( i = 0; i < MAX_FIBERS; ++ i )
        {
                fiberList[i].stack = 0;
                fiberList[i].function = 0;
                fiberList[i].active = 0;
                fiberList[i].priority = 20;
                fiberList[i].id = i+1;
        }

        return;
}
```

Pasando ya a la función de esta sección, lo primero que hago es comprobar que el thread id y la prioridad proporcionados sean válidos y que no estemos llamando la función desde una fibra .

```
int sched_nice(int pri, int th_id) {

        if (th_id >= numFibers || th_id < 0 || pri < 0 || pri > 100 || inFiber) {
                printf("El thread especificado no existe o la prioridad especificada no es valida o no estamos en el thread master");
                return -1;
        }

        int index;
        if (th_id == 0) index = currentFiber;
        else index = seach(th_id);
        if (pri == 0) return fiberList[index].priority;

        int anterior = fiberList[index].priority;
        fiberList[index].priority = pri;
        if (index < numFibers - 1 && pri >= fiberList[index+1].priority) {
                for (int i = index; (i < numFibers - 1) && (fiberList[i].priority >= fiberList[i+1].priority); i++) {

                        fiber aux = fiberList[i+1];
                        fiberList[i+1] = fiberList[i];
                        fiberList[i] = aux;
                }
        }
        else if (index > 0 && pri < fiberList[index-1].priority) {
                for (int i = index; (i > 0) && (fiberList[i].priority < fiberList[i-1].priority); i--) {

                        fiber aux = fiberList[i-1];
                        fiberList[i-1] = fiberList[i];
                        fiberList[i] = aux;
                }
        }
        return anterior;
}
```

A continuación, si el parámetro th_id es 0 asigno como thread a tratar el actual, y si no, hago uso de search

```
int search(int th_id) {

        for (int i = 0; i < numFibers; i++) {
                if (fiberList[i].id == th_id) return i;
        }
        return -1;
}
```

para encontrar en qué índice de fiberList se encuentra el thread con el id proporcionado, y si el parámetro pri es 0 se devuelve la prioridad de este thread y se acaba la ejecución. Contrariamente, guardo la prioridad anterior que tenía el thread al que se la vamos a cambiar y finalmente lo coloco en su nueva posición, moviendo su información hacia la derecha mientras los threads que hasta el momento eran menos prioritarios ahora lo sean más o tengan la misma prioridad o moviéndola a la izquierda mientras los threads que hasta el momento eran más prioritarios ahora lo sean menos. Finalmente, devuelvo la prioridad guardada.

Cabe mencionar que si los threads no estuviesen ordenados desde el primer momento o si se pudieran insertar threads con cualquier prioridad en cualquier posición esta implementación no serviría dado que solo ordena bien si todos los

elementos menos el que se trata ya están ordenados, pero no es el caso. De todos modos, con un algoritmo de ordenación como es el caso por ejemplo del selection sort que no es nada complicado, este problema quedaría solucionado.

sched_yield()

En este caso lo primero que hago es comprobar que la función no se esté llamando desde una fibra y que sea la más prioritaria, si esto sucede, retorno 0 y se acaba la ejecución. De lo contrario, me guardo el id del thread más prioritario (luego habrá que retornarlo) y entonces nos aparecen dos casos; que la función se haya llamado desde una fibra que no sea la más prioritaria y que la función se haya llamado desde el contexto principal. Para ambos casos lo que hacemos es guardar el contexto, pasar el procesador al thread más prioritario, y cuando éste nos lo devuelve, comprobar si ha acabado su ejecución y si es así liberar el espacio que tenía y su entrada, reducir el numero de threads actuales y mover los restantes una posición a la izquierda, y si no simplemente continuar con la ejecución del contexto en el que nos encontrábamos. Cabe mencionar que para el segundo caso, si no hay ningún thread disponible se retorna un 0 y no se acaba haciendo nada. Finalmente, una vez ya ha acabado todo el proceso de pasarle el procesador al thread más prioritario (el de la posición 0 de fiberList) se devuelve su id que nos habíamos guardado y finaliza la función.

```
int sched_yield(void) {

        if (inFiber && currentFiber == 0) return 0;
        int result = fiberList[0].id;

        /* Fiber case */
        if ( inFiber)
        {
                /* Store the current state */
                if ( setjmp( fiberList[ currentFiber ].context ) )
                {

                        if ( ! fiberList[currentFiber].active )
                        {
                                /* If we get here, the fiber returned and is done! */
                                LF_DEBUG_OUT1( "Fiber %d returned, cleaning up.", currentFiber );

                                free( fiberList[currentFiber].stack );
#ifdef VALGRIND
                                VALGRIND_STACK_DEREGISTER(fiberList[currentFiber].stackId);
#endif
                                -- numFibers;

                                if ( currentFiber != numFibers )
                                {
                                        for (int i = numFibers; i > 0; i--) { fiberList[i-1] = fiberList[i]; }
                                }

                                /* Clean up the entry */
                                fiberList[numFibers].stack = 0;
                                fiberList[numFibers].function = 0;
                                fiberList[numFibers].active = 0;

                        }
                        else
                        {
                                LF_DEBUG_OUT1( "Fiber %d yielded execution.", currentFiber );
                        }
                }
                else
                {
                        LF_DEBUG_OUT1( "Fiber %d yielding the processor...", currentFiber );
                        /* Saved the state: Let's switch back to first fiber in queue */
                        currentFiber = 0;
                        longjmp(fiberList[currentFiber].context, 1 );
                }
        }
```

```
        else
        {
                if ( numFibers == 0 ) return 0;

                /* Save the current state */
                if ( setjmp( mainContext ) )
                {
                        /* The fiber yielded the context to us */
                        inFiber = 0;
                        if ( ! fiberList[currentFiber].active )
                        {
                                /* If we get here, the fiber returned and is done! */
                                LF_DEBUG_OUT1( "Fiber %d returned, cleaning up.", currentFiber );

                                free( fiberList[currentFiber].stack );
#ifdef VALGRIND
                                VALGRIND_STACK_DEREGISTER(fiberList[currentFiber].stackId);
#endif


                                -- numFibers;
                                if ( currentFiber != numFibers )
                                {
                                        for (int i = numFibers; i > 0; i--) { fiberList[i-1] = fiberList[i]; }
                                }

                                /* Clean up the entry */
                                fiberList[numFibers].stack = 0;
                                fiberList[numFibers].function = 0;
                                fiberList[numFibers].active = 0;
                        }
                        else
                        {
                                LF_DEBUG_OUT1( "Fiber %d yielded execution.", currentFiber );
                        }
                }
                else
                {
                        currentFiber = 0;
                        LF_DEBUG_OUT1( "Switching to fiber %d", currentFiber );
                        inFiber = 1;
                        longjmp( fiberList[ currentFiber ].context, 1 );
                }
        }

        return result;
}
```

Como podemos observar, la implementación es bastante parecida a la de fiberYield, y algún detallito que quizás faltaba por comentar es que cuando llamamos la función desde el main, tenemos que tratar también el parámetro inFiber, indicando cuándo estamos en un thread con un 1 y cuándo no lo estamos con un 0.

sched_handoff(int th_id)

El código de esta función es idéntico al de la anterior, lo único que tenemos que comprobar que el th_id proporcionado sea válido, es decir, que esté entre 1 y numFibers y luego proceder a buscar la posición del thread th_id en fiberList para a continuación hacer exactamente lo mismo que en la función anterior pero

cediéndole el procesador a este thread y no al más prioritario, con la única excepción de que al final retornamos un 0 en lugar de su id.

Para acabar con este laboratorio, voy a hacer un programa que incluiré en la entrega para comprobar el funcionamiento de las funciones pero también diseñaré un caso de uso en que sea útil disponer de estas llamadas para mejorar el rendimiento de su aplicación.

```
int main()
{

        /* Initialize the fiber library */
        initFibers();
        for (int i = 0; i < 10; i++) {

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

        int prioritat_anterior1 = sched_nice(1, 2);
        int prioritat_anterior2 = sched_nice(21, 3);
        printf("Primer cambio hecho al thread 2, antes tenia prioridad: %d\n", prioritat_anterior1);
        printf("Primer cambio hecho al thread 3, antes tenia prioridad: %d\n", prioritat_anterior2);
        int prioritat_actual1 = sched_nice(0, 2);
        printf("Prioridad del thread 2: %d\n", prioritat_actual1);
        int prioritat_actual2 = sched_nice(0, 3);
        printf("Prioridad del thread 3: %d\n", prioritat_actual2);

        int error = sched_nice(-1,1);
        printf("Error por prioridad no valida: %d\n", error);
        getinfo();
        sched_handoff(1);
        sched_yield();

        /* Since these are nonpre-emptive, we must allow them to run */


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

He hecho este pequeño programa para probar las funcionalidades implementadas. Concretamente, creo 10 threads asignándole una función a cada uno y luego le pruebo de cambiar la prioridad al thread 2 por una prioridad alta y al thread 3 por una prioridad baja y compruebo con el valor retornado por sched_nice que la prioridad anterior es correcta. A continuación compruebo también la prioridad actual de los threads 2 y 3 y verifico que si le paso a la función una prioridad negativa da error y retorna -1. Después, he creado una función getinfo en la librería que te saca el orden de threads por prioridad, para ver que según los cambios que he hecho se han ordenado correctamente. Finalmente, he querido probar sched_yield y

sched_handoff (evidentemente aquí quedarían threads pendientes de ejecutar aún, únicamente era para hacer la prueba), y he tenido problemas. Cuando quiero hacer varias llamadas a estas funciones, ya sea únicamente a una o a las dos combinadas, como mucho solo me funcionan 2 iteraciones y luego ya se produce un segmentation fault. Tras haberlo estado analizando, finalmente no he conseguido solucionarlo. Creo que el problema está en que no consigo establir bien la relación entre el setjmp de usr1handlerCreateStack y el longjmp de sched_yield y el de sched_handoff para realizar el cambio de contexto al contexto del thread que se quiere ejecutar, pero no estoy seguro de ello. No obstante, en esta prueba donde solo llamamos una vez a cada una, los resultados han sido correctos:

```
xavi@xavi-VirtualBox:~/Escritorio/SOAD/LAB3_MartiLlull_Xavier$ example-Xavi
Primer cambio hecho al thread 2, antes tenia prioridad: 20
Primer cambio hecho al thread 3, antes tenia prioridad: 20
Prioridad del thread 2: 1
Prioridad del thread 3: 21
El thread especificado no existe o la prioridad especificada no es valida o no estamos en el thread master
Error por prioridad no valida: -1
2
1
4
5
6
7
8
9
10
3
Residuo 3 doy 0
Residuo 3 doy 2
```

Como podemos ver, los threads 2 y 3, antes de realizar el cambio de prioridad tenían prioridad 20 como hemos visto en initFibers(), y después del cambio efectivamente se ha cambiado a 1 y 21 respectivamente. Luego vemos como nos sale un mensaje de error por intentar asignar una prioridad negativa y que la función nos devuelve un -1. Para acabar de comprobar que el funcionamiento de sched_nice es correcto, usamos getinfo y vemos si los threads están ordenados correctamente. Y así es, el 2 tiene prioridad 1, luego los que aparecen no han sido modificados hasta el 3, que le hemos puesto la prioridad más baja y se ha colocado después del 10. Finalmente, vemos que sched_handoff da paso al thread con id 1, que efectivamente lleva asociada la función que dice residuo 3 doy 0 porque fue el primer thread creado y sched_yield da paso a la ejecución del thread 2, que es el más prioritario y podemos comprobar que es así por el mismo motivo.

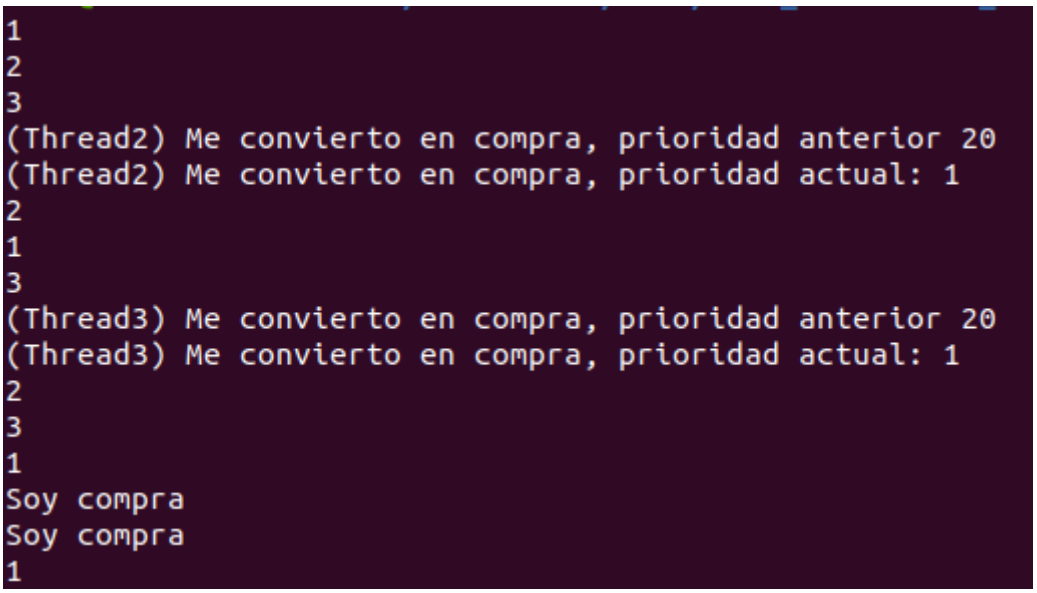
A pesar de no poder hacer pruebas como dios manda por no tener en correcto funcionamiento sched_yield y sched_handoff, intentaré describir un caso de uso donde se podrían utilizar de manera positiva para el trabajo a realizar y subiré el código aunque no funcione.

Caso de uso

Un ejemplo de caso de uso podría ser por ejemplo uno de los casos de uso que tendría la aplicación que nuestro grupo describirá en el midterm; efectuar una compra online. Al realizar esta acción, sería positivo poder asignarle una prioridad mayor al thread de usuario que maneja este y otros casos de uso como iniciar sesión en la aplicación que serían menos prioritarios, y situarlo el primero en la cola, para a continuación cederle el procesador con sched_yield o seleccionar con sched_handoff cualquiera de los procesos igual de prioritarios. Esto mejoraría el rendimiento de la aplicación ya que no se irían cargando threads y ejecutándolos por orden de llegada sino que los más críticos podían ejecutarse cuanto antes. El programa que he creado para representar este caso de uso es:

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

#include <unistd.h>

void inicioSesion()
{
        printf("Soy inicio de sesion\n");
}

void compra()
{
        printf("Soy compra\n");
}

int main()
{

        /* Initialize the fiber library */
        initFibers();
        spawnFiber(&inicioSesion);
        spawnFiber(&inicioSesion);
        spawnFiber(&inicioSesion);

        getinfo();
        int prioritat_anterior = sched_nice(1, 2);
        printf("(Thread2) Me convierto en compra, prioridad anterior %d\n", prioritat_anterior);
        int prioritat_actual = sched_nice(0, 2);
        printf("(Thread2) Me convierto en compra, prioridad actual: %d\n", prioritat_actual);
        getinfo();

        int prioritat_anterior1 = sched_nice(1, 3);
        printf("(Thread3) Me convierto en compra, prioridad anterior %d\n", prioritat_anterior1);
        int prioritat_actual1 = sched_nice(0, 3);
        printf("(Thread3) Me convierto en compra, prioridad actual: %d\n", prioritat_actual1);
        getinfo();

        changeFunc(1,&compra);
        changeFunc(2,&compra);
        sched_handoff(3);
        sched_yield();
        getinfo();

        /* Since these are nonpre-emptive, we must allow them to run */


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

Lo que hago es crear tres threads que representan un inicio de sesión, y entonces al segundo y el tercero les cambio la prioridad como si se convirtiesen en una compra. De hecho, he añadido otra función a la librería que permite cambiar la función asociada al thread pasándole el índice que tiene en la cola de threads y la función que se quiere adoptar. Por lo tanto, el segundo thread y el tercero se han convertido en una compra y el segundo era el más prioritario por haberse convertido primero, y lo que hacemos es escoger el tercero con sched_handoff(3) y luego como el segundo sigue siendo el más prioritario usamos sched_yield y finalmente comprobamos que solo queda el 1.



Como podemos ver en la salida, al principio el orden es 1, 2, 3, luego se cambia la prioridad del 2 y pasa a ser el más prioritario, luego se cambia la prioridad del 3 y pasa a ser el segundo más prioritario por llevar menos tiempo que el 2 y finalmente se ejecuta el tercero y luego el segundo, escribiendo ambos por salida estándar “soy compra” para después comprobar que solo queda el 1.