Universidade Federal de Uberlândia

Faculdade de Computação

FiberLib

Implementação de uma biblioteca de threads em user-space

Curso: Bacharelado em Sistemas de Informação

Disciplina: Sistemas Operacionais

Professor: Rivalino Matias Júnior

Alunos: Vitor Manoel Gonçalves Teixeira, Diego Batistuta Ribeiro de Andrade e Guilherme Bartasson Naves Junker

Data de entrega: 13/12/2019

Sumário

1. **Descrição**……..……..…...3

2. **Código**……...4

   2.1. Elementos Globais……..……..4

   2.2. Estruturas de dados……...6

   2.3. Rotinas abertas da biblioteca……...8

   2.4. Escalonador……...……..……19

   2.5. Rotinas auxiliares gerais……..25

3. **Referências**……..……29

1.Descrição

Este grupo foi responsável por projetar e programar rotinas de criação e gerenciamento de threads em user-space, comumente referidas como “fibers”, que fazem parte de uma biblioteca denominada **FiberLib**. A FiberLib possui estruturas e rotinas que permitem a criação de threads em user level, seguindo o modelo de M threads no user level mapeadas para uma thread no kernel level **(M:1)**, num escalonamento preemptivo baseado em time-slice, utilizando o algoritmo **round-robin**. A criação, escalonamento e armazenamento de contexto das threads foi feita utilizando system calls de manipulação de contextos, sendo elas: **getcontext()**, **setcontext()**, **makecontext()** e **swapcontext()**, além de system calls de manipulação de timers e sinais, sendo elas: **setitimer(), getitimer()** e **sigaction()**.

Essa biblioteca foi implementada com base em várias bibliotecas com o mesmo intuito(implementação de threads) e nos conhecimentos prévios de estruturas de dados dos integrantes do grupo. Apesar de isso ter exigido bastante esforço e dedicação, o resultado foi eficiente e satisfatório até onde pôde ser analisado, no que se refere aos requisitos do trabalho. A FiberLib foi criada em formato de **shared library** e pode ser utilizada, de forma amadora, em programas que necessitem de multi-threading simples. Pelo fato de ela não possuir nenhuma estrutura para semáforos e outras formas de evitar bloqueios do processo e acesso de áreas críticas por múltiplas threads, evitar que problemas relacionados a isso aconteçam é **de total responsabilidade de seus usuários**.

As rotinas **fiber_create()**, **fiber_exit()** e **fiber_join()** foram implementadas com base na funcionalidade das rotinas equivalentes da biblioteca pthread, sendo elas, respectivamente: **pthread_create()**, **pthread_exit()** e **pthread_join()**. Há um arquivo de cabeçalho “**fiber.h**” que possui os símbolos necessários para o uso das rotinas da biblioteca e um arquivo “**fiber.c**” com os códigos das rotinas e declarações das estruturas e variáveis globais utilizadas por elas.

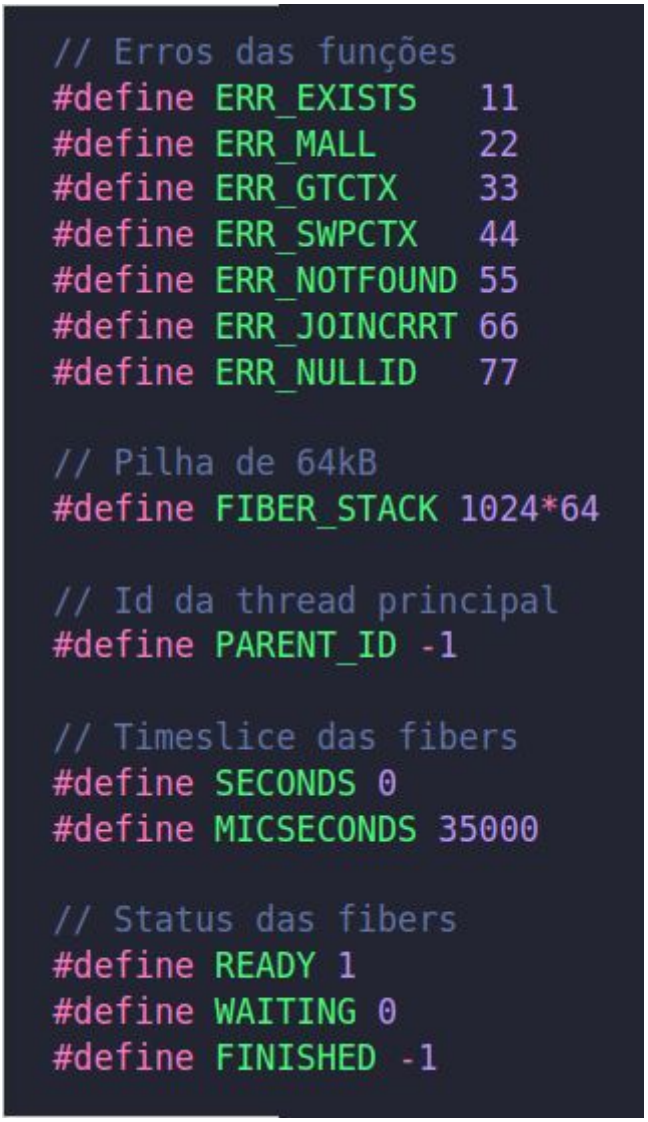
Pelo fato de o escalonamento ser baseado no algoritmo round-robin, uma maneira intuitiva e simples de implementá-lo é por meio de uma lista circular, e por isso a FiberLib armazena as fibers em uma lista desse tipo, com sua cauda apontando para a cabeça, assim simplificando o trabalho do escalonador.

2. Código

Nesta seção serão descritas as estruturas(definições, variáveis globais, definições de tipo e estruturas de dados) e rotinas da biblioteca.

2.1. Elementos globais

- **Constantes literais**



```
// Erros das funções
#define ERR_EXISTS   11
#define ERR_MALL     22
#define ERR_GTCTX    33
#define ERR_SWPCTX   44
#define ERR_NOTFOUND 55
#define ERR_JOINCRRT 66
#define ERR_NULLID   77

// Pilha de 64kB
#define FIBER_STACK 1024*64

// Id da thread principal
#define PARENT_ID -1

// Timeslice das fibers
#define SECONDS 0
#define MICSECONDS 35000

// Status das fibers
#define READY 1
#define WAITING 0
#define FINISHED -1
```

- **ERR_EXISTS:** Retorno de erro da rotina fiber_create(), quando a mesma recebe um id que já existe na lista atual de fibers, ou seja, quando se tenta criar uma Fiber que já existe;
- **ERR_MALL:** Retorno de erro padrão para quando um malloc() falha em uma das rotinas;

- **ERR_GTCTX:** Retorno de erro padrão para quando uma system call getcontext() falha;
- **ERR_SWPCTX:** Retorno de erro padrão para quando uma system call swapcontext() falha;
- **ERR_NOTFOUND:** Retorno de erro da fiber_join(), caso o id passado para a função não exista na lista de fibers;
- **ERR_JOINCRRT:** Retorno de erro da fiber_join(), caso uma Fiber tentar executar um join em si mesma;
- **ERR_NULLID:** Retorno de erro da fiber_create(), caso o ponteiro para o id da Fiber receba NULL na chamada da função;
- **FIBER_STACK:** É a definição do tamanho das pilhas das Fibers (64kb);
- **PARENT_ID:** É o id da thread principal, de valor -1;
- **SECONDS:** Tempo, em segundos, do temporizador da biblioteca;
- **MICSECONDS:** Tempo, em microsegundos, do temporizador da biblioteca (35 milissegundos);
- **READY:** Indica que a Fiber está pronta para executar;
- **WAITING:** Indica que a Fiber está esperando outra Fiber;
- **FINISHED:** Indica que a Fiber terminou;

- **f_List**

```
FiberList * f_list = NULL;
```

A f_List é a lista global de Fibers, uma struct do tipo **FiberList** que armazena as Fibers criadas pela biblioteca. Mais informações sobre o tipo FiberList são fornecidas posteriormente.

- **timer**

```
struct itimerval timer;
```

Struct do tipo itimerval utilizada como temporizador da biblioteca para realizar o escalonamento preemptivo das Fibers.

- **schedulerContext e parentContext**

```
ucontext_t schedulerContext, parentContext;
```

São ambas structs do tipo ucontex_t, responsáveis por armazenar o contexto do escalonador de Fibers (schedulerContext) e da thread principal (parentContext).

- **fiber_t**

```
typedef int fiber_t;
```

Typedef de int, o fiber_t é utilizado na biblioteca para ids de Fibers. Foi decidido que o tipo seria um inteiro para que o Id da thred principal fosse -1.

2.2. Estruturas de dados

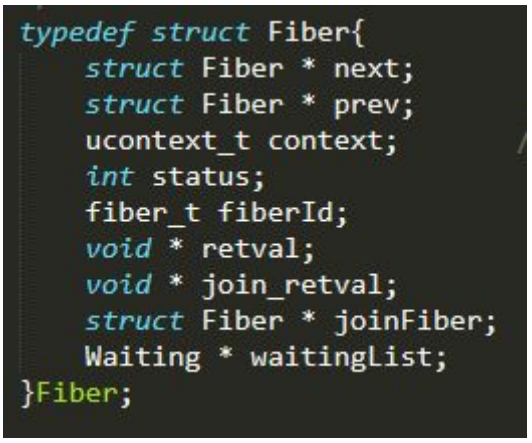
- **Waiting**

```
typedef struct Waiting{
    fiber_t waitingId;
    struct Waiting * next;
}Waiting;
```

Waiting é uma struct que representa um nodo de uma lista de id's de Fibers que estão esperando uma Fiber ser encerrada. A rotina **releaseFibers()** percorre a waitingList de uma Fiber quando ela termina, e altera o status de todas as fibers que estão esperando para **READY**, caso existam.

- **waitingId:** fiber_t que representa o id da fiber do nodo atual;
- **next:** Ponteiro para uma struct do tipo waiting que aponta para o próximo nodo;

- **Fiber**



```
typedef struct Fiber{
    struct Fiber * next;
    struct Fiber * prev;
    ucontext_t context;
    int status;
    fiber_t fiberId;
    void * retval;
    void * join_retval;
    struct Fiber * joinFiber;
    Waiting * waitingList;
}Fiber;
```

Fiber é a struct de uma fiber, que possui como atributos:

- **next e prev**: Ponteiros para outras structs de Fiber, eles são necessários para se formar a lista circular;
- **context**: Ponteiro para uma struct ucontext_t, que guarda o contexto da fiber;
- **status**: Inteiro que representa o estado atual da fiber, ou seja, se ela está ativa/pronta para ser executada (representado pela constante literal **READY**), esperando outra fiber (representado pela constante **WAITING**) ou se ela terminou (representado pela constante **FINISHED**);
- **fiberId**: Inteiro que representa o id da fiber. Ele nunca é negativo e é necessário para identificar a Fiber. O Id da thread principal é representado pela constante **PARENT_ID**, que vale -1;
- **retval:** Ponteiro do tipo void que aponta para o valor de retorno da Fiber;
- **join_retval:** Ponteiro do tipo void que aponta para o valor de retorno de uma fiber que esta Fiber está esperando;
- **joinFiber:** Ponteiro para uma struct do tipo Fiber que aponta para a Fiber que essa fiber está esperando;
- **waitingList:** Ponteiro para um struct do tipo Waiting que aponta para uma lista de fibers que estão esperando essa fiber.

- **FiberList**

```
typedef struct FiberList{
    Fiber * fibers;
    Fiber * currentFiber;
    int nFibers;
    int started;
}FiberList;
```

FiberList é a struct que representa a lista de fibers criadas pela biblioteca, e possui como atributos:

- **fibers:** Ponteiro para uma struct do tipo Fiber que aponta para a "cabeça" da lista circular de fibers. Até que **fiber_exit()** seja chamada na main, a thread principal estará na cabeça da lista;
- **currentFiber**: Ponteiro para uma struct do tipo Fiber que aponta para a fiber que está sendo executada no momento;
- **started:** Inteiro que indica se o timer e o escalonamento das fibers já estão funcionando;
- **nFibers**: Inteiro que representa a quantidade de Fibers na lista de Fibers;

2.3. - Rotinas abertas da biblioteca

Nesta seção serão detalhadas as três rotinas abertas da biblioteca, ou seja, as funções que os usuários da biblioteca têm acesso. Além disso, para cada uma dessas rotinas, também serão abordadas as rotinas internas auxiliares da biblioteca diretamente relacionadas a elas.

Fiber_Create

```
int fiber_create(fiber_t *fiber, void *(*start_routine) (void *), void *arg);
```

A fiber_create é a função responsável por criar as fibers, e na primeira criação bem-sucedida de uma Fiber, também chama rotinas responsáveis por inicializar a lista de fibers e iniciar o timer e o envio de sinais para que o processo execute o escalonador. Para a alocação da pilha da Fiber, foi utilizado um método em

- **Parâmetros:**
    - fiber: Id de uma fiber;

- start_routine: Rotina (função) que a fiber executará;
- arg: Parâmetro que a start_routine receberá;

- **Retorno:**
  - int: Pode retornar um código de erro (ver constantes literais de erro), ou zero, indicando que a função executou normalmente;
- **Funcionamento:**
  1. Cria uma struct do tipo ucontext_t(fiberContext) e um ponteiro para struct do tipo Fiber (fiberNode);
  2. Verifica se o fiber (ponteiro para fiber_t recebido pela função) é null e se há alguma Fiber com o id igual a ele:

```
// Se o ponteiro apontar para NULL
if(fiber == NULL)
    return ERR_NULLID;

// Verificando se já existe uma fiber com esse id
if(findFiber(* fiber) != NULL){
    printf("Essa fiber já existe\n");
    return ERR_EXISTS;
}
```

  3. Pega o contexto atual e o armazena na fiberContext, atribui o endereço do schedulerContext ao uc_link, além de alocar e configurar corretamente o uc_stack(pilha da thread) e verifica se o alocamento de memória foi bem sucedido:

```
// Obtendo o contexto atual e armazenando-o na variável fiberContext
if(getcontext(&fiberContext) == -1){
    perror("Ocorreu um erro no getcontext da fiber_create");
    return ERR_GTCTX;
}

// Modificando o contexto para uma nova pilha
fiberContext.uc_link = &schedulerContext;
fiberContext.uc_stack.ss_sp = malloc(FIBER_STACK);
fiberContext.uc_stack.ss_size = FIBER_STACK;
fiberContext.uc_stack.ss_flags = 0;

// Caso a alocação da pilha falhe
if (fiberContext.uc_stack.ss_sp == 0 ) {
    perror("erro malloc na criação da pilha na fiber_create");
    return ERR_MALL;
}
```

4. Cria a fiber usando o fiberContext, start_routine e arg, inicializa a struct que armazenará a fiber recém criada, e chama a rotina interna auxiliar **pushfiber()**, que insere a fiber no final da lista de fibers e atribui o seu id (equivalente ao número de fibers na lista, não considerando a thread principal), que logo após é repassado ao endereço para o qual o ponteiro fiber aponta:

```
// Criando a fiber propriamente dita
makecontext(&fiberContext, (void (*)(void )) start_routine, 1, arg);

// Inicializando a struct recém-criada que armazena a fiber
fiberNode->context = fiberContext;
fiberNode->prev = NULL;
fiberNode->next = NULL;
fiberNode->status = READY;
fiberNode->retval = NULL;
fiberNode->join_retval = NULL;
fiberNode->joinFiber = NULL;
fiberNode->waitingList = NULL;

// Inserindo a nova fiber na lista de fibers
pushFiber(fiberNode);

// Atribuindo o id da fiber adequadamente
* fiber = fiberNode->fiberId;
```

5. Caso essa seja a primeira chamada da fiber_create(), as rotinas internas **initFiberList()** e **startFibers()** serão chamadas. A primeira cria e instancia a lista de fibers, insere a thread principal como cabeça da lista e cria a “fiber” do escalonador. A segunda configura o timer e o sinal de alarme para que o escalonador seja chamado periodicamente, executando o escalonador a cada ciclo. Note que a rotina initFiberList() não é chamada no código da fiber_create(), mas sim, no código da pushFiber(). Além disso, o contexto da thread principal é capturado e repassado para o nodo-cabeça da lista de fibers. Caso tudo tenha ocorrido como esperado, a função retorna 0:

```c
// Verificando se o escalonador já começou a rodar.
// Caso não tenha, startFibers() é chamada e o contexto
// da thread principal é capturado.
if (f_list->started == 0) {
    f_list->started = 1;
    startFibers();

    // Obtendo o contexto da thread atual e o transferindo para o currentContext
    if(getcontext(&f_list->fibers->context) == -1){
        perror("Ocorreu um erro no getcontext da fiber_create");
        return ERR_GTCTX;
    }
}

return 0;
```

PushFiber

```c
void pushFiber(Fiber * fiber)
```

pushFiber() é uma função auxiliar para a fiber_create(), utilizada para inserir uma fiber no “fim” da lista circular de fibers, assim como chamar a rotina interna initFiberList(), que cria a lista de fibers, caso ela não exista.

- **Parâmetros:**
  - fiber: ponteiro para struct do tipo fiber que representa a fiber a ser inserida;
- **Retorno:**
  - void;
- **Funcionamento:**
  1. Cria uma struct itimerval que será usada para guardar o tempo restante da fiber atual ao ser chamada a rotina interna **stopTimer()**, para evitar erros durante a inserção da fiber na lista. Também verifica se a f_list está inicializada, e caso não esteja, chama initFiberList() para inicializá-la corretamente:

```c
struct itimerval restored;

// Iniciando a lista de fibers, caso seja null
if(f_list == NULL)
    initFiberList();
```

2. Para o timer com stopTimer() e salva o tempo restante da fiber atual em restored, pois essa parte da rotina é crítica e deve ser atômica:

```
// Para o timer para evitar troca de fibers em região crítica
stopTimer(&restored);
// Caso só haja a fiber da thread principal na lista
if (f_list->nFibers == 1) {
    fiber->prev = (Fiber *) f_list->fibers;
    fiber->next = f_list->fibers;
    f_list->fibers->prev = (Fiber *) fiber;
    f_list->fibers->next = (Fiber *) fiber;

}
else { // Caso haja outras fibers na lista
    fiber->next = (Fiber *) f_list->fibers;
    fiber->prev = (Fiber *) f_list->fibers->prev;
    fiber->prev->next = (Fiber *) fiber;
    f_list->fibers->prev = (Fiber *) fiber;
}
```

3. Incrementa o atributo nFibers de f_list, que representa o número de fibers, define o id da Fiber como esse número subtraído de 1(não considera a thread principal, que possui id igual a -1), e restaura o timer de acordo com o tempo restante anterior por meio da rotina interna **restoreTimer()**:

```
// Incrementando o número de fibers
f_list->nFibers++;

// Definindo o id da fiber
fiber->fiberId = f_list->nFibers - 1;

// restaura o timer para o restante do timeslice que a fiber atual possuía
restoreTimer(&restored);
```

InitFiberList

```
int initFiberList()
```

initFiberList() é uma função auxiliar da pushFiber e é a responsável por criar e inicializar a lista de fibers de maneira correta, e só é chamada quando a biblioteca identifica que a lista ainda não foi criada.

- **Parâmetros:**
  - Nenhum;
- **Retorno:**
  - int: podendo ser um código de erro (ver DEFINES de erro), ou zero, indicando que a função executou normalmente;
- **Funcionamento:**
  1. Aloca espaço na memória para a lista de fibers f_list e caso tudo ocorra como esperado, Inicializa os atributos de f_list:

```c
// Criando a estrutura da lista
f_list = (FiberList *) malloc(sizeof(FiberList));
if (f_list == NULL) {
    perror("erro malloc na criação da lista na initFiberList");
    return ERR_MALL;
}
// Inicializando a lista de fibers
f_list->fibers = NULL;
f_list->nFibers = 0;
f_list->currentFiber = NULL;
f_list->started = 0;
```

  2. Cria uma struct do tipo Fiber para a thread principal, a inicializa com a variável global parentContext, que no momento está vazia, e atribui o id. Após isso, incrementa o atributo nFibers da lista:

```c
// Criando a estrutura de fiber para a thread principal
Fiber * parentFiber = (Fiber *) malloc(sizeof(Fiber));
if (parentFiber == NULL) {
    perror("erro malloc na criação da parentFiber da initFiberList");
    return ERR_MALL;
}

// Inicializando a estrutura da thread principal
parentFiber->context = parentContext;
parentFiber->fiberId = PARENT_ID;
parentFiber->prev = NULL;
parentFiber->next = NULL;
parentFiber->status = READY;

// Adicionado a estrutura da thread principal como o primeiro elemento da lista
f_list->fibers = (Fiber *) parentFiber;

// Incrementando o número de fibers da lista
f_list->nFibers++;
```

3. Define a fiber atual como a thread principal, instancia a variável global schedulerContext, aloca sua pilha e cria a “fiber” do escalonador, que executa a rotina **fiberScheduler()**. Caso tudo tenha ocorrido como esperado, a função retorna 0.

```c
// Obtendo um contexto para o escalonador
if(getcontext(&schedulerContext) == -1){
    perror("Ocorreu um erro no getcontext da initFiberList");
    return ERR_GTCTX;
}

// Limpando o contexto recebido para o escalonador
schedulerContext.uc_link = &parentContext;
schedulerContext.uc_stack.ss_sp = malloc(FIBER_STACK);
schedulerContext.uc_stack.ss_size = FIBER_STACK;
schedulerContext.uc_stack.ss_flags = 0;
if (schedulerContext.uc_stack.ss_sp == NULL) {
    perror("erro malloc na criação da pilha na initFiberList");
    return ERR_MALL;
}

// Criando o contexto do escalonador
makecontext(&schedulerContext, fiberScheduler, 1, NULL);

return 0;
```

StartFibers

```
void startFibers()
```

A startFibers() é uma rotina auxiliar da fiber_create e é a responsável por inicializar o timer virtual e configurar o tratador de sinal, para que a cada ciclo do timer o sinal **SIGVTALRM** seja enviado para o processo, cujo tratador é a rotina **timeHandler()**, que salva o contexto atual e troca para o contexto do escalonador.

- **Parâmetros:**
  - Nenhum;
- **Retorno:**
  - void;
- **Funcionamento:**
  1. Cria uma struct do tipo sigaction(sa);
  2. Atribui o endereço da timerHandler() ao sa_handler de sa, chama **sigaction()**, usando SIGVTALRM e sa, configura o timer utilizando as constantes literais SECONDS e MICSECONDS, e chama **restoreTimer()**, que, nesse caso, inicia o timer chamando **setitimer()**:

```
// Atribuindo a rotina timeHandler() como tratador do sinal
sa.sa_handler = &timeHandler;

// Chamada de sistema para configurar o tratador do sinal SIGVTALRM no processo
if(sigaction (SIGVTALRM, &sa, NULL) == -1){
    perror("Ocorreu um erro no sigaction da startFibers");
    return;
}

// Inicializando o timer e seus intervalos
timer.it_value.tv_sec = SECONDS;
timer.it_value.tv_usec = MICSECONDS;

timer.it_interval.tv_sec = SECONDS;
timer.it_interval.tv_usec = MICSECONDS;

//começando o timer
restoreTimer(&timer);
```

Fiber_Join

```
int fiber_join(fiber_t fiber, void **retval)
```

A fiber_join() é a função responsável por fazer com que a fiber que a chamou espere a fiber cujo id foi passado como argumento(fiber_t fiber) terminar antes de continuar sua execução, e transferir o endereço do valor de retorno da fiber a ser aguardada para retval.

- **Parâmetros:**
  - fiber: É do tipo fiber_t, ou seja, representa o id de uma fiber;
  - retval: Ponteiro previamente alocado pelo usuário da biblioteca, necessário para receber o endereço do valor de retorno da fiber a ser aguardada. Caso o usuário da biblioteca passe NULL como argumento, a atribuição do valor de retorno será ignorada;
- **Retorno:**
  - int: Pode ser um código de erro (ver DEFINES de erro), ou zero, indicando que a função executou normalmente;
- **Funcionamento:**
  1. Cria um ponteiro para Fiber(fiberNode) que recebe o ponteiro para fiber(ou **NULL**) retornado pela rotina interna **findFiber()**, uma rotina simples que procura uma fiber na lista de fibers cujo id é igual ao id recebido por ela. Feito isso, algumas verificações são feitas. Se a fiber não for encontrada na lista, ou se a fiber atual tentar chamar fiber_join() em si mesma, os respectivos códigos de erro são retornados. Caso a fiber a ser aguardada já terminou antes mesmo da chamada da fiber_join(), a função retorna 0 normalmente, pois o propósito foi cumprido:

```
// Tentando encontrar a fiber com o id fiber
Fiber * fiberNode = findFiber(fiber);

// Se a fiber não foi encontrada na lista
if(fiberNode == NULL)
    return ERR_NOTFOUND;

// Se a fiber a ser esperada é a que está executando
if(fiberNode->fiberId == f_list->currentFiber->fiberId)
    return ERR_JOINCRRT;

// Se a fiber que deveria terminar antes já terminou
if(fiberNode->status == FINISHED)
    return 0;
```

2. Cria um ponteiro do tipo **Waiting**(waitingNode), aloca memória a ele, verifica se a memória foi alocada corretamente, atribui o valor de fiber ao waitingId do waitingNode e atribui NULL ao next do waitingNode:

```c
// Criando um nodo para a lista de espera da fiber que será aguardada
Waiting * waitingNode = (Waiting *) malloc(sizeof(Waiting));

// Caso a alocação falhe
if (waitingNode == NULL) {
    perror("erro malloc na fiber_join");
    return ERR_MALL;
}

// Atribuindo o id do nodo e inicializando seu ponteiro next
waitingNode->waitingId = fiber;
waitingNode->next = NULL;
```

3. Chama **stopTimer()** para parar o timer, pois os próximos passos são críticos, e verifica se há alguma Fiber na lista de espera da fiberNode. Se não houver, apenas transfere o waitingNode para a cabeça da waitingList do fiberNode. Se houver, trata a WaitingList como uma pilha, e a reorganiza de forma que o waitingNode seja a nova cabeça da lista, e seu next aponte para a antiga cabeça:

```c
// Parar o timer, área crítica
stopTimer(NULL);

// Adicionando um nodo na lista de espera da fiber a ser aguardada
if(fiberNode->waitingList == NULL){
    fiberNode->waitingList = (Waiting *) waitingNode;
}
else{
    Waiting * waitingTop = (Waiting *) fiberNode->waitingList;
    fiberNode->waitingList = (Waiting *) waitingNode;
    fiberNode->waitingList->next = (Waiting *) waitingTop;
}
```

4. Coloca o endereço do fiberNode no ponteiro joinFiber da Fiber que está executando no momento, altera o status dela para WAITING e chama a system call **swapContext()**, salvando o contexto da fiber atual corretamente e transferindo a execução para a “fiber” do escalonador:

```
// Definindo a fiber que a fiber atual está esperando
f_list->currentFiber->joinFiber = (Fiber *) fiberNode;

// Marcando a fiber atual como esperando
f_list->currentFiber->status = WAITING;

// Trocando para o contexto do escalonador
if(swapcontext(&f_list->currentFiber->context, &schedulerContext) == -1){
    perror("Ocorreu um erro no swapcontext da fiber_join");
    return ERR_SWPCTX;
}
```

5. Verifica se o ponteiro retval passado para a join_fiber() é diferente de NULL, e se for, recupera o valor de retorno da Fiber que está sendo aguardada(caso exista) e reseta os retvals da Fiber atual para evitar problemas:

```
if(retval != NULL){
    // Caso a joinFiber não tenha sido destruída ainda, o retval é recuperado diretamente dela
    if(f_list->currentFiber->join_retval == NULL && f_list->currentFiber->joinFiber != NULL)
        *retval = f_list->currentFiber->joinFiber->retval;
    // Caso contrário, o retval é recuperado do atributo join_retval da própria fiber que chamou
    // fiber_join()
    else
        *retval = f_list->currentFiber->join_retval;

    // Resetando os retvals da fiber
    f_list->currentFiber->retval = NULL;
    f_list->currentFiber->join_retval = NULL;
}
```

6. Muda o status da Fiber atual para READY e retorna 0.

```
// Definindo o status da fiber atual como pronta para executar
f_list->currentFiber->status = READY;

return 0;
```

Fiber_Exit

```
void fiber_exit(void *retval)
```

A fiber_exit() é a função responsável por mudar o status de uma fiber para FINISHED, ou seja, indicar que uma fiber acabou sua execução e precisa ser destruída. Além disso, ela transfere o endereço do valor de retorno da fiber que terminou(ou NULL) por meio do ponteiro retval.

- **Parâmetros:**
  - retval: Ponteiro para void;
- **Retorno:**
  - void;
- **Funcionamento:**
  1. Atribui o retval recebido como parâmetro ao retval da Fiber que está sendo executada, muda o status da Fiber que está sendo executada para FINISHED e chama o timeHandler:

```
    // Instanciando o valor de retorno da fiber
    f_list->currentFiber->retval = retval;
    // Definindo status da fiber atual como terminada
    f_list->currentFiber->status = FINISHED;

    // Chamando o escalonador corretamente
    timeHandler();
}
```

2.4. Escalonador

Esta seção explicará o funcionamento do escalonador e suas rotinas auxiliares. O escalonador se baseia no algoritmo round-robin, e é chamado toda vez que um intervalo do timer dispara um sinal para a processo.

FiberScheduler

```
void fiberScheduler()
```

A fiberscheduler() é a rotina do escalonador, e é responsável executar as Fibers da lista que estão com status **READY** e **WAITING**(caso a Fiber que ela está esperando esteja encerrada), retirar as Fibers com status **FINISHED** da Lista, além

de liberar a memória alocada para elas e reconfigurar a lista de fibers. Antes de liberar a memória alocada pela fiber terminada por meio da rotina auxiliar **fiber_destroy()**, outra rotina é chamada, a **releaseFibers()**.

A responsabilidade da releaseFibers() é alterar o status de espera das Fibers da **waitingList** dessa Fiber para **READY**, além de transferir o valor de retorno dela para os ponteiros **join_retval** dessas Fibers e liberar a memória alocada para os nodos de tipo Waiting. Caso o escalonador identifique que todas as fibers da lista, incluindo a thread principal, foram encerradas por meio de fiber_exit() e destruídas, o escalonador libera a memória alocada por todas as estruturas da biblioteca restantes, incluindo a pilha de seu próprio contexto, e chama exit(0) caso tudo ocorra como esperado, ou exit(-1) caso algum comportamento não identificado ocorra na fiber_destroy().

- **Parâmetros:**
  - Nenhum;
- **Retorno:**
  - void;
- **Funcionamento:**
  1. Chama stopTimer para parar o timer, pois os próximos passos são críticos, e cria a estrutura que armazenará a próxima Fiber a ser executada(nextFiber):

```
stopTimer(NULL);

// Estrutura que armazenará a próxima fiber a ser executa
Fiber * nextFiber = (Fiber *) f_list->currentFiber->next;
```

  2. Verifica se o status da fiber apontada por nextFiber é diferente de READY, e enquanto ele for, ou seja, enquanto uma Fiber pronta para ser executada não for encontrada:
     a. Verifica se o status é FINISHED:
        i. Se for, chama a releaseFibers() para liberar as Fibers que estão esperando esta;
        ii. Chama a fiber_destroy(), passa o ponteiro nextFiber para ela, e atribui o resultado ao mesmo ponteiro(a rotina fiber_destroy() retorna um ponteiro para Fiber, que será um ponteiro para a Fiber logo à frente da destruída, ou NULL, caso todas as Fibers tenham sido destruídas);
        iii. Se nextFiber for null, verifica se há mais Fibers na lista de Fibers:

1. Se não houver, libera a memória alocada para f_list e a pilha da fiber do escalonador, e chama exit(0);
2. Se houverem, chama exit(-1), algo inesperado ocorreu;

b. Verifica se o status é WAITING:
   i. Se for, verifica se o status da Fiber que ela está esperando é diferente de FINISHED:
      1. Se for, passa o endereço da fiber à frente da apontada por nextFiber para nextFiber;
      2. Se não, muda o status da Fiber que está esperando para READY;

```
// Enquanto não encontrar uma fiber pronta para ser executada
while(nextFiber->status != READY) {
    // Caso a fiber já tenha terminado
    if(nextFiber->status == FINISHED){
        // Liberando as fibers esperando esta(caso existam)
        releaseFibers(nextFiber->waitingList);
        // Destruindo essa fiber e obtendo a próxima(caso exis
        nextFiber = fiber_destroy(nextFiber);
        // Se a fiber_destroy retornar NULL
        if(nextFiber == NULL){
            // Caso não haja mais nenhuma fiber na lista
            if(f_list->nFibers == 0){
                free(f_list); // Liberando a lista de fibers
                free(schedulerContext.uc_stack.ss_sp); // Libe
                exit(0); // Terminando o programa
            }
            // Caso contrário, algum erro ocorreu
            exit(-1);
        }
    }
    // Caso a thread atual esteja num join
    if(nextFiber->status == WAITING) {
        // Caso a thread que ela está esperando não estiver en
        if(nextFiber->joinFiber->status != FINISHED)
            nextFiber = (Fiber *) nextFiber->next; // Pula a t
        // Caso a thread que ela está esperando tenha terminad
        else
            nextFiber->status = READY; // Definir o status com

    }
}
```

3. Atribui a Fiber apontada por nextFiber, que saiu do laço por ter status READY, ao ponteiro da Fiber atual. Feito isso, restaura o timer para o timeslice padrão definido pela biblioteca por meio da rotina **restoreTimer()**, e define o contexto atual como o contexto da nextFiber:

```
f_list->currentFiber = (Fiber *) nextFiber;

// Redefinindo o timer para o tempo normal
timer.it_value.tv_sec = SECONDS;
timer.it_value.tv_usec = MICSECONDS;

// Resetando o timer
restoreTimer(&timer);

// Definindo o contexto atual como o da próxima fiber
if(setcontext(&nextFiber->context) == -1){
    perror("Ocorreu um erro no setcontext da fiberScheduler");
    return;
}
```

ReleaseFibers

```
void releaseFibers(Waiting *waitingList){
```

A releaseFibers() é uma função auxiliar da fiberScheduler() que é responsável por liberar todas as fibers da waitingList num join para que sejam executadas no escalonador, além liberar a memória alocada para os nodos da waitingList e também instanciar o join_retval de cada uma das fibers corretamente.

- **Parâmetros:**
  - waitingList: Ponteiro do tipo Waiting para a lista de Fibers que estão esperando;
- **Retorno:**
  - void;
- **Funcionamento:**
  1. Enquanto houver nodos na waitingList, a função obtém o próximo nodo da lista, procura a fiber com o id do nodo atual da waitingList e se a Fiber com esse id existir, muda o status dela para READY, e armazena o valor de retorno da Fiber que ela estava aguardando no seu atributo

join_retval. Feito isso, libera a memória alocada pelo nodo do topo da waitingList e define o próximo nodo como o topo da lista:

```
// Enquanto houver fibers esperando
while(waitingList != NULL){
    // Recebe o próximo nodo da lista
    Waiting * waitingNode = waitingList->next;
    // Procura a fiber com o id do nodo atual da waitingList
    Fiber * waitingFiber = findFiber(waitingList->waitingId);
    // Se a fiber existir e estiver esperando
    if(waitingFiber != NULL && waitingFiber->status == WAITING){
        // Libera a fiber
        waitingFiber->status = READY;
        // Guarda o retval
        waitingFiber->join_retval = waitingFiber->joinFiber->retval;
    }
    // Libera o nodo no topo
    free(waitingList);
    // Vai para o próximo nodo
    waitingList = (Waiting *) waitingNode;
}
```

Fiber_destroy

```
Fiber * fiber_destroy(Fiber * fiber)
```

A fiber_destroy() é uma função auxiliar do fiberScheduler que é responsável por remover a Fiber apontada pelo parâmetro fiber da lista de Fibers, liberar a memória que foi alocada para sua estrutura e a a pilha de seu contexto, além de reconfigurar a lista de Fibers e diminuir o seu atributo nFibers em 1, já que uma das fibers foi removida.

- **Parâmetros:**
  - fiber: Ponteiro para struct do tipo Fiber que representa a Fiber a ser destruída;
- **Retorno:**
  - Ponteiro para um struct tipo Fiber;
- **Funcionamento:**
  1. Cria um inteiro(id) e passa o fiberId da Fiber para ele;
  2. Verifica se o parâmetro recebido é NULL, e caso seja, retorna NULL;
  3. Verifica se o status da Fiber é FINISHED, ou seja, se a ela terminou de executar. Caso não seja, retorna null, pois apenas fibers encerradas devem ser destruídas:

```
if(fiber->status != FINISHED)
    return NULL;
```

4. Obtém os endereços das Fibers anterior e posterior à Fiber que será destruída:

```
// Obtendo o endereço da fiber anterior à
Fiber * prevFiber = (Fiber *) fiber->prev;

// Obtendo o endereço da fiber que sucede
Fiber * nextFiber = (Fiber *) fiber->next;
```

5. Verifica se a Fiber que será removida não possui suas Fibers anterior e posterior, e caso possua, efetivamente a remove da lista ao alterar os ponteiros dessas Fibers:

```
// Caso os ponteiros next e prev da fiber na
if(nextFiber != NULL && prevFiber != NULL){
    // Removendo a fiber da lista circular
    nextFiber->prev = (Fiber *) prevFiber;
    prevFiber->next = (Fiber *) nextFiber;
}
```

6. Verifica se o id é o id da thread principal, e se for, modifica a cabeça da lista para a próxima Fiber:

```
if(id == PARENT_ID)
    f_list->fibers = nextFiber;
```

7. Destrói a fiber, ou seja, libera sua pilha e estrutura, e atribui NULL ao ponteiro para evitar acesso indevido de memória:

```
free(fiber->context.uc_stack.ss_sp);
free(fiber);
fiber = NULL;
```

8. Decrementa o atributo nFibers da f_list, verifica se a lista de fibers está vazia, e se estiver, preenche os ponteiros das Fibers anterior e posterior com NULL para evitar acesso indevido de memória e retorna a Fiber posterior:

```
f_list->nFibers--;

// Caso a lista tinha sid
// de memória
if(f_list->nFibers == 0){
    nextFiber = NULL;
    prevFiber = NULL;
}

return nextFiber;
```

2.5. Rotinas auxiliares gerais

Nesta seção serão detalhadas as rotinas auxiliares da biblioteca que são usadas por mais de uma função, ou seja, não estão diretamente ligadas a uma rotina específica.

StopTimer

```
void stopTimer(struct itimerval * restored)
```

A stopTimer() é a função responsável por parar o temporizador e salvar o tempo restante do timer no seu parâmetro **restored**, normalmente usada em áreas críticas para garantir o correto funcionamento da biblioteca. Caso restored receba NULL, o tempo restante não é salvo, e a função apenas para o timer.

- **Parâmetros:**
  - restored: Ponteiro para uma struct do tipo itimerval que representa um temporizador;
- **Retorno:**
  - void;
- **Funcionamento:**
  1. Verifica se o ponteiro restored é diferente de null, e caso seja, obtém o tempo restante do timer e armazena e o armazena nele, utilizando a system call **getitimer()**:

```
if(restored != NULL) // se restored for NULL, não é
    if (getitimer(ITIMER_VIRTUAL, restored) == -1) {
        perror("erro na getitimer() da stopTimer");
        exit(1);
    }
```

2. Atribui 0 como tempo do timer:

```
timer.it_value.tv_sec = 0;
timer.it_value.tv_usec = 0;
```

3. Para o timer (executando-o com o tempo zerado), por meio da system call **setitimer()**:

```
if(setitimer (ITIMER_VIRTUAL, &timer, NULL) == -1){
    perror("Ocorreu um erro no setitimer() da stopTimer");
    return;
}
```

RestoreTimer

```
void restoreTimer(struct itimerval * restored)
```

A restoreTimer() é a função responsável por restaurar o temporizador que foi parado, ou simplesmente configurar o timer com os valores da estrutura apontada por **restored**, utilizando a system call **setitimer()**.

- **Parâmetros:**
  - restored: é um ponteiro para uma struct do tipo itimerval que representa um temporizador;
- **Retorno:**
  - void;
- **Funcionamento:**
  1. Reinicia o timer com restored e verifica se ocorreu algum erro na setitimer():

```
if(setitimer (ITIMER_VIRTUAL, restored, NULL) == -1){
    perror("Ocorreu um erro no setitimer da restoreTimer");
    return;
}
```

FindFiber

```
Fiber * findFiber(fiber_t fiberId){
```

A findFiber() é a função responsável por encontrar e retornar uma Fiber por meio de seu id, caso ela exista na lista de Fibers.

- **Parâmetros:**
  - FiberID: fiber_t, ou seja, um inteiro que representa o id de uma fiber;
- **Retorno:**
  - void;
- **Funcionamento:**
  1. Cria um inteiro(i) e um ponteiro para uma struct do tipo Fiber(fiber), e verifica se a lista de fibers foi inicializada. Caso não tenha sido, retorna NULL:

```
int i;
Fiber * fiber;

// Caso não haja lista de fibers ainda
if(f_list == NULL)
    return NULL;
```

  2. Obtém a primeira fiber da lista de fibers e verifica se o fiberId foi inicializado. Caso ele não tenha sido inicializado, retorna NULL:

```
// Obtendo a primeira fiber da li
fiber = (Fiber *) f_list->fibers;

// Se não estiver inicializado
if(fiberId == 0)
    return NULL;
```

  3. Verifica se o fiberId que foi passado é igual ao fiberId da thread principal, e se for o caso, retorna a estrutura da thread principal:

```
if(f_list->fibers->fiberId == fiberId)
    return f_list->fibers;
```

  4. Verifica se o fiberId que foi passado é igual ao fiberId da Fiber atual, e caso seja, retorna a Fiber atual:

```
if(f_list->currentFiber->fiberId == fiberId)
    return f_list->currentFiber;
```

5. Procura na lista de Fibers uma fiber com o fiberId igual ao que foi passado:

```
for(i = 0; i < f_list->nFibers; i++){
    fiber = (Fiber *) fiber->next;
    if(fiber->fiberId == fiberId)
        break;
}
```

6. Caso não haja uma Fiber correspondente ao fiberId recebido, retorna NULL. Caso tenha encontrado a Fiber, retorna seu endereço:

```
if(i == f_list->nFibers)
    return NULL;
else return fiber; // Ca
```

TimeHandler

```
void timeHandler(){
```

A timeHandler() é o tratador do sinal **SIGVTALRM**, sendo responsável por salvar o contexto da fiber atual e trocar para o contexto da Fiber do escalonador. É chamada periodicamente a cada vez que o timeslice de uma Fiber acaba. Também é chamada diretamente no final da rotina **fiber_exit()**, para que fibers encerradas não utilizem seu timeslice restante.

- **Parâmetros:**
  - nenhum;
- **Retorno:**
  - void;
- **Funcionamento:**
  1. Por meio da system call **swapcontext()**, salva o contexto atual e troca para o contexto da Fiber do escalonador.

```
if(swapcontext(&f_list->currentFiber->context, &schedulerContext) == -1){
    perror("Ocorreu um erro no swapcontext da timeHandler");
    return;
}
```

3. Referências

- TANENBAUM, Andrew S. Sistemas Operacionais Modernos: 4 ed. São Paulo: Pearson Education do Brasil, 2016
- https://www.evanjones.ca/software/threading.html
- https://tinycthread.github.io
- https://stackoverflow.com/questions/7332755/setitimer-question/7340778#7340778
- https://www.gnu.org/software/pth/pth-manual.html#threading_background
- https://stackoverflow.com/questions/7578318/implementing-join-function-in-a-user-level-thread-library
- http://man7.org/linux/man-pages/man3/pthread_create.3.html
- http://man7.org/linux/man-pages/man3/pthread_join.3.html
- http://man7.org/linux/man-pages/man3/pthread_exit.3.html
- http://man7.org/linux/man-pages/man3/getcontext.3.html
- http://man7.org/linux/man-pages/man3/makecontext.3.html
- http://man7.org/linux/man-pages/man3/swapcontext.3.html
- http://man7.org/linux/man-pages/man2/sigaction.2.html
- http://man7.org/linux/man-pages/man2/setitimer.2.html
- http://man7.org/linux/man-pages/man2/getitimer.2.html