

Universidade Federal De Uberlândia  
Faculdade de Computação  
Sistemas De Informação  
Sistemas Operacionais  
GSI018  
Prof. Rivalino Matias Jr.

BRENO HENRIQUE DE OLIVEIRA FERREIRA, 11821BSI245  
EULLER HENRIQUE BANDEIRA OLIVEIRA , 11821BSI210  
ERICK CRISTIAN DE OLIVEIRA PEREIRA, 11621BSI265  
SAMUEL AUGUSTO MEIRELES DA SILVA, 11821BSI252

# TCD 7

Objetivo: Projetar e programar o gerenciamento de threads no user-space.

# 1. main.c

## Main.c

### Elementos globais

#### 1. Bibliotecas

1. Importa a biblioteca `stdio.h`, tal biblioteca possui as funções responsáveis pela manipulação de entrada/saída

```
#include <stdio.h>
```

2. Importa a biblioteca `fiber.h`, tal biblioteca possui as funções responsáveis pela criação, destruição, saída, espera e obtenção do id de `fibers`

```
#include "fiber.h"
```

## 2. main

main

```
int main(int argc, char const *argv[])  
{
```

1. Cria a variável fid1 do tipo fiber\_t

```
fiber_t fid1 = NULL;
```

2. Chama a função fiber\_create, tal função irá criar uma fiber e tal fiber irá executar a função

3. Verifica se a função fiber\_create retornou um código de erro

```
if(fiber_create(&fid1, &threadFunction, NULL) == -1)  
    perror("cannot create a fiber\n");
```

4. Exibe o id da fiber criada

```
printf("Criou a fiber 1 = %p\n", fid1);
```

threadFunction

## 2. main

5. Cria a variável `fid2` do tipo `fiber_t`

```
fiber_t fid2 = NULL;
```

6. Chama a função `fiber_create`, tal função irá criar uma fiber e tal fiber irá executar a função `threadFunction`

7. Verifica se a função `fiber_create` retornou um código de erro

```
if(fiber_create(&fid2, &threadFunction, NULL) == -1)
    perror("cannot create a fiber\n");
```

8. Exibe o id da fiber criada

```
printf("Criou a fiber 2 = %p\n", fid2);
```

9. Cria a variável `fid3` do tipo `fiber_t`

```
fiber_t fid3 = NULL;
```

10. Chama a função `fiber_create`, tal função irá criar uma fiber e tal fiber irá executar a função `threadFunction`

`threadFunction`

11. Verifica se a função `fiber_create` retornou um código de erro

```
if(fiber_create(&fid3, &threadFunction, NULL) == -1)
    perror("cannot create a fiber\n");
```

12. Exibe o id da fiber criada

```
printf("Criou a fiber 3 = %p\n", fid3);
```

13. Destrói a fiber 1

```
fiber_destroy(fid1);
```

14. Coloca a fiber 3 na fila de espera de execução

```
fiber_join(fid3, NULL);
```

### 3. thread\_function

#### threadFunction

```
void* threadFunction()  
{
```

1. Exibe o id da fiber em execução

```
printf("Rotina da thread %p\n", fiber_self());
```

2. Exibe uma mensagem inicial para teste

```
printf("Olá mundo! :D\n");
```

3. Inicia um loop somente para aumentar o tempo de duração da rotina

```
int i = 0;  
while (++i < 1000000000);
```

4. Exibe uma mensagem final para teste

```
printf("Adeus :C\n\n");
```

5. Sai da fiber

```
fiber_exit(NULL);
```

## 4. fiber.h

### Fiber.h

1. Define o identificador de uma fiber

```
typedef void * fiber_t;
```

2. Define o protótipo da função fiber\_create (tal função pertence à fiber.c)

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

3. Define o protótipo da função fiber\_join (tal função pertence à fiber.c)

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

4. Define o protótipo da função fiber\_destroy (tal função pertence à fiber.c)

```
int fiber_destroy(fiber_t fiber);
```

5. Define o protótipo da função fiber\_self (tal função pertence à fiber.c)

```
fiber_t fiber_self();
```

6. Define o protótipo da função fiber\_exit (tal função pertence à fiber.c)

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

# 5. fiber.c

## Fiber.c

### 1. Elementos globais

#### 1. Bibliotecas

```
#include <ucontext.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/time.h>
```

- `ucontext.h`: Manipulação de contexto
- `stdio.h`: Manipulação de entrada/saída
- `stdlib.h`: Manipulação de alocações e liberações de memória etc...
- `signal.h`: Manipulação de sinal
- `time.h`: Manipulação do tempo

#### 2. Constantes literais

##### 1. Tamanho da pilha que será alocada para o contexto de execução de uma fiber

```
#define FIBER_STACK_SIZE 1024
```

##### 2. Estados de execução de uma fiber

```
#define STATE_READY 0
#define STATE_BLOCKED 1
#define STATE_FINISHED 2
```

- `STATE_READY`: Indica que a Fiber pode ser executada
- `STATE_BLOCKED`: Indica que a execução da Fiber foi interrompida
- `STATE_FINISHED`: Indica que a Fiber foi executada

##### 3. Define os tempos padrões

```
#define TIME_SLICE_SEC 10
#define TIME_SLICE_USEC 35000
```

- `TIME_SLICE_SEC`: Fatia de tempo em segundos
- `TIME_SLICE_USEC`: Fatia de tempo em microssegundos

##### 4. Define o id inicial da fiber pai

```
#define PARENT_ID -1
```

## 5. fiber.c

### 3. fiber\_t

Identificador de uma fiber

```
typedef void * fiber_t;
```

### 4. Waiting

@struct Waiting

@brief Estrutura que armazena as fibers em espera do join de outra fiber.

@param id identificador da fiber que está sendo aguardada .

@param next ponteiro para a próxima fiber na lista de espera.

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

### 5. Fiber

@struct Fiber

@brief Estrutura de uma fiber (thread no espaço do usuário). Implementa a lista circular para o algoritmo de preempção round robin.

@param next ponteiro para outra estrutura na lista.

@param context ponteiro para o contexto de execução da fiber.

@param status estado atual da fiber; STATE\_READY a fiber está pronta para ser executada; STATE\_BLOCKED a fiber está em espera; STATE\_FINISHED fiber finalizada

@param retval ponteiro que armazena o endereço do valor de retorno.

@param join\_rval ponteiro que armazena o endereço do valor de retorno da fiber que está sendo aguardada.

@joinFiber ponteiro para a fiber que essa fiber está esperando

@param waitList lista de fibers que estão aguardando essa fiber.

```
typedef struct Fiber
{
    struct Fiber *next;
    ucontext_t context;
    int status;
    void *retval;
    void *join_rval;
    struct Fiber *joinFiber;
    Waiting *waitlist;
} Fiber;
```



## 5. fiber.c

### 6. Fiber\_List

@struct Fiber\_List

@brief Estrutura que armazena as fibers. Representa uma lista circular para o algoritmo de preempção round robin.

@param head ponteiro para a primeira fiber da lista; para a cabeça.

@param tail ponteiro para a última fiber da lista; para a cauda.

@param running ponteiro para a fiber em execução.

@param size quantidade de elementos inseridos na lista.

```
typedef struct Fiber_List
{
    Fiber *head;
    Fiber *tail;
    Fiber *running;
    int size;
} Fiber_List;
```

### 7. fiber\_list

Lista de fibers

```
Fiber_List *fiber_list = NULL;
```

### 8. scheduler

Contexto do escalonador e da thread principal

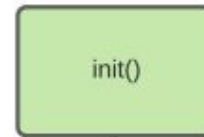
```
ucontext_t scheduler_ctx, parent_ctx;
```

Timer do escalonador

```
struct itimerval timer;
```

## 6. init

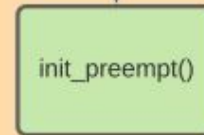
Executada ao carregar a biblioteca no programa principal.



Aloca e inicializa a estrutura global que representa a lista de fibers; e cria o contexto do escalonador.



Inicializa a estrutura de sinais para lidar com sinais do tipo SIGVTALRM.



## 6. init

init

```
__attribute__((constructor)) void init()
```

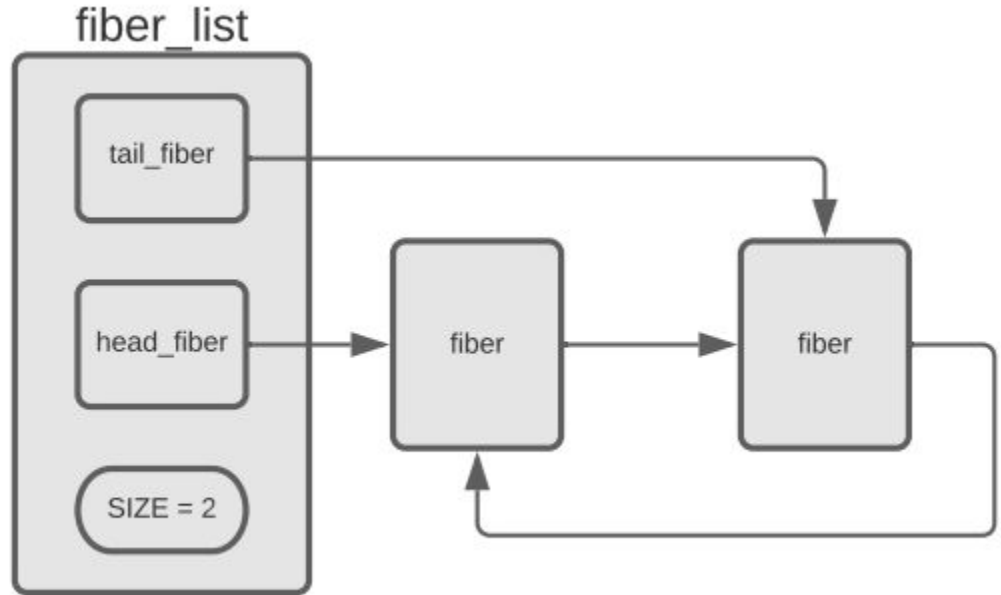
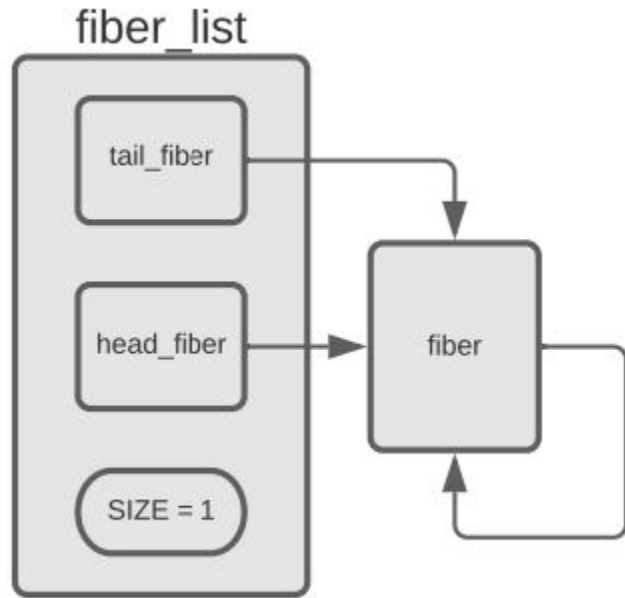
1. Chama a função `init_fiber_list`

```
init_fiber_list();
```

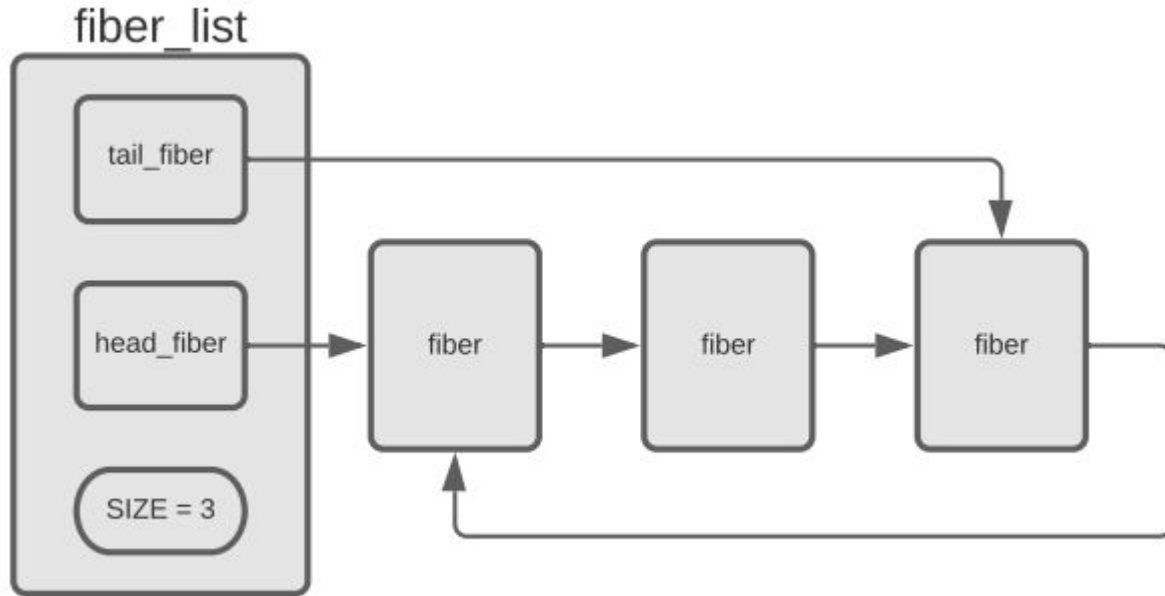
2. Chama a função `init_preempt`

```
init_preempt();
```

## 7. init\_fiber\_list



## 7. init\_fiber\_list



## 7. init\_fiber\_list

init\_fiber\_list

```
int init_fiber_list() {
```

@name init\_fiber\_list()

@brief Inicializa a lista de fibers. Insere a estrutura da thread principal na lista. Inicializa o contexto do escalonador.

1. Aloca uma lista de fibers

```
fiber_list = malloc(sizeof(Fiber_List));
```

2. Verifica se a lista de fibers não foi alocada

```
if (fiber_list == NULL) {  
    perror("list malloc failed at init_fiber_list.");  
    return -1;  
}
```

## 7. init\_fiber\_list

3. Aloca a fiber pai

```
Fiber *parentFiber = malloc(sizeof(Fiber));
```

4. Verifica se a fiber pai não foi alocada

```
if (parentFiber == NULL)
{
    perror("malloc failed at init_fiber_list.");
    return -1;
}
```

5. O campo context da fiber pai recebe o contexto do fiber pai

```
parentFiber->context = parent_ctx;
```

6. O fiber posterior ao fiber pai é o próprio fiber pai

```
parentFiber->next = parentFiber;
```

7. Define qual é o estado do fiber pai

```
parentFiber->status = STATE_READY;
```

8. A fiber fiber pai se torna a cabeça da lista de fibers

```
fiber_list->head = parentFiber;
```

## 7. init\_fiber\_list

9. A fiber fiber se torna a cauda da lista de fibers

```
fiber_list->tail = parentFiber;
```

10. O campo running (fiber que está em execução) pertencente à lista de fibers recebe a fiber pai

```
fiber_list->running = parentFiber;
```

11. O tamanho da lista de fibers é incrementado

```
fiber_list->size = 1;
```

12. Definição do uc\_link

```
scheduler_ctx.uc_link = &parent_ctx;
```

Ponteiro para o contexto que será retomado quando este contexto retornar;

|

13. Alocação da pilha

```
scheduler_ctx.uc_stack.ss_sp = malloc(FIBER_STACK_SIZE);
```

14. Definição do tamanho da pilha

```
scheduler_ctx.uc_stack.ss_size = FIBER_STACK_SIZE;
```



## 7. init\_fiber\_list

15. Definição da flag da pilha

```
scheduler_ctx.uc_stack.ss_flags = 0;
```

16. Verifica se a pilha não foi alocada

```
if (scheduler_ctx.uc_stack.ss_sp == NULL)
{
    perror("stack malloc failed at init_fiber_list.");
    return -1;
}
```

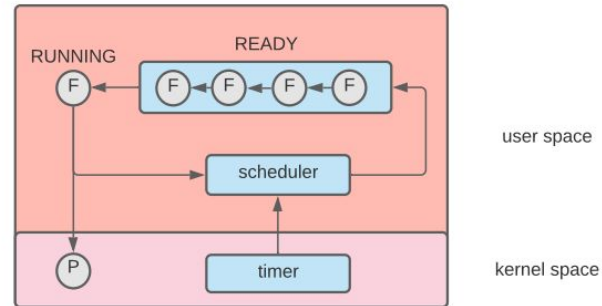
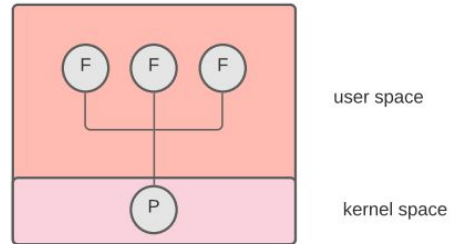
17. Chama a função makecontext

```
makecontext(&scheduler_ctx, scheduler, 0);
```

A função makecontext(ucontext\_t \*ucp, (void \*func)(), int argc, ...) modifica o contexto especificado por ucp, que foi inicializado usando getcontext(). Quando este contexto é retomado usando swapcontext() ou setcontext(), a execução do programa continua chamando func, passando os argumentos que seguem argc na chamada da makecontext().

## 8. scheduler

Uma thread no kernel  
para várias threads no  
espaço do usuário 1:M



Um timer é utilizado para dar início ao fluxo. Quando o timer expira é lançado um sinal. Então o escalonador troca a fiber em execução olhando em uma lista circular de fibers que estão prontas. Uma fiber executando pode ser preemptada ao fim do seu time slice ou terminar antes. Caso ainda exista fibers para serem executadas retorna ao contexto do escalonador.

## 8. scheduler

scheduler

```
void scheduler()  
{
```

@name scheduler()

@brief Função responsável por fazer a preempção das fibers, para implementar o algoritmo round robin a estrutura de lista e fiber tem o comportamento de uma lista circular. Essa função verifica o estado da fiber e age de acordo. Caso o estado da fiber seja STATE\_READY, seu contexto será setado como o contexto de execução. Quando a lista estiver vazia suas estruturas serão desalocadas.

1. Para o timer

```
stop_timer();
```

2. O próximo timer recebe o próximo fiber que será executado

```
Fiber *nextFiber = fiber_list->running->next;
```

3. Enquanto não encontrar uma fiber pronta para ser executada

```
while (nextFiber->status != STATE_READY)  
{
```

3.1. Caso a fiber já tenha terminado de ser executada

```
if (nextFiber->status == STATE_FINISHED)  
{
```

3.1.1. Libera as heads que estão esperando esta (caso existam)

```
release_fibers(nextFiber->waitlist);
```

3.1.2. Destrói essa fiber e obtém a próxima (caso exista)

```
nextFiber = pop(nextFiber);
```

```
...
```

3.1.3 Se a pop retornar NULL

```
if (nextFiber == NULL)  
{
```

3.1.3.1 Caso não haja mais nenhuma fiber na lista

```
if (fiber_list->size == 0)  
{
```

## 8. scheduler

```
3.1.3 Se a pop retornar NULL
    if (nextFiber == NULL)
    {
        3.1.3.1 Caso não haja mais nenhuma fiber na lista
            if (fiber_list->size == 0)
            {
                3.1.3.1.1 Libera a lista de fibers
                    free(fiber_list);

                3.1.3.1.2 Libera a pilha do escalonador
                    free(scheduler_ctx.uc_stack.ss_sp);

                3.1.3.3 Retorna o código de sucesso e fecha o programa
                    exit(0);

                3.1.3.2 Caso contrário, retorna um código de erro e fecha o programa
                    exit(-1);
            }
    }

3.2 Caso a thread atual esteja num join
    if (nextFiber->status == STATE_BLOCKED)
    {
        3.2.1 Caso a thread que ela está esperando não estiver encerrada
            if (nextFiber->joinFiber->status != STATE_FINISHED)
            3.2.1.1 Pula a thread que está esperando
                nextFiber = nextFiber->next;

        3.2.2 Caso a thread que ela está esperando tenha terminado
            else
            3.2.2.1 Define o estado como STATE_READY ( a thread está pronta para ser executada)
                nextFiber->status = STATE_READY;
    }

4. Define a próxima fiber selecionada como a fiber atual
    fiber_list->running = nextFiber;

5. Redefine o timer para o tempo normal
    timer.it_value.tv_sec = TIME_SLICE_SEC;
    timer.it_value.tv_usec = TIME_SLICE_USEC;

6. Inicia o timer
    start_timer();

7. Define o contexto atual como o da próxima fiber

    if (setcontext(&nextFiber->context) == -1)
    {
        perror("setcontext failed at scheduler");
        return;
    }
```

## 9. release\_fibers

release\_fibers

```
void release_fibers(Waiting *waitingList)
```

@name release\_fibers(Waiting \*waitingList)

@brief Libera todas as fibers da lista de espera para que sejam executadas.

@param waitingList - lista de espera das fibers.

1. Enquanto houver head esperando

```
while (waitingList != NULL)
{
```

- 1.1 Recebe o próximo nó da lista

```
Waiting *waitingNode = waitingList->next;
```

- 1.2 Recebe a fiber com o id do nó atual da waitingList

```
Fiber *waitingFiber = waitingList->id;
```

- 1.3 Se a fiber existir e estiver esperando

```
if (waitingFiber != NULL && waitingFiber->status ==  
STATE_BLOCKED)
```

```
{
```

- 1.3.1 Libera a fiber

```
waitingFiber->status = STATE_READY;
```

- 1.3.2 Guarda o retval

```
waitingFiber->join_rval = waitingFiber->joinFiber->retval;
```

- 1.4 Libera o nó no topo

```
free(waitingList);
```

- 1.5 Vai para o próximo nó

```
waitingList = waitingNode;
```

# 10. init\_preempt

init\_preempt

```
int init_preempt() {
```

@name init\_preempt()

@brief Inicializa uma estrutura sigaction que representa um sinal e atribui uma rotina (preempt) para lidar com o sinal do tipo SIGVTALRM.

@return 0 para sucesso; -1 para falha.

1. A variável new\_s do tipo struct sigaction é declarada

```
struct sigaction new_s;
```

2. O campo sa\_handler pertencente à struct new\_s recebe o endereço da rotina preempt

```
new_s.sa_handler = &preempt;
```

3. O campo sa\_flags pertencente à struct new\_s recebe o valor 0

```
new_s.sa_flags = 0;
```

5. Chama a função sigaction

6. Verifica se a função sigaction retornou um código de erro

```
if(sigaction (SIGVTALRM, &new_s, NULL) == -1){  
    perror("sigaction failed at init_preempt.");  
    return -1;  
}
```

```
sigaction(int signum, struct sigaction act, struct sigaction oldact);
```

Examina e troca a ação tomada por um processo quando ocorrer um sinal que tenha o valor do signum, no caso SIGVTALRM. act é ação que deve ser tomada quando o sinal ocorrer. oldact é a ação antiga

# 11. preempt

É chamada sempre que  
ocorre um sinal do tipo  
SIGVTALRM

preempt()

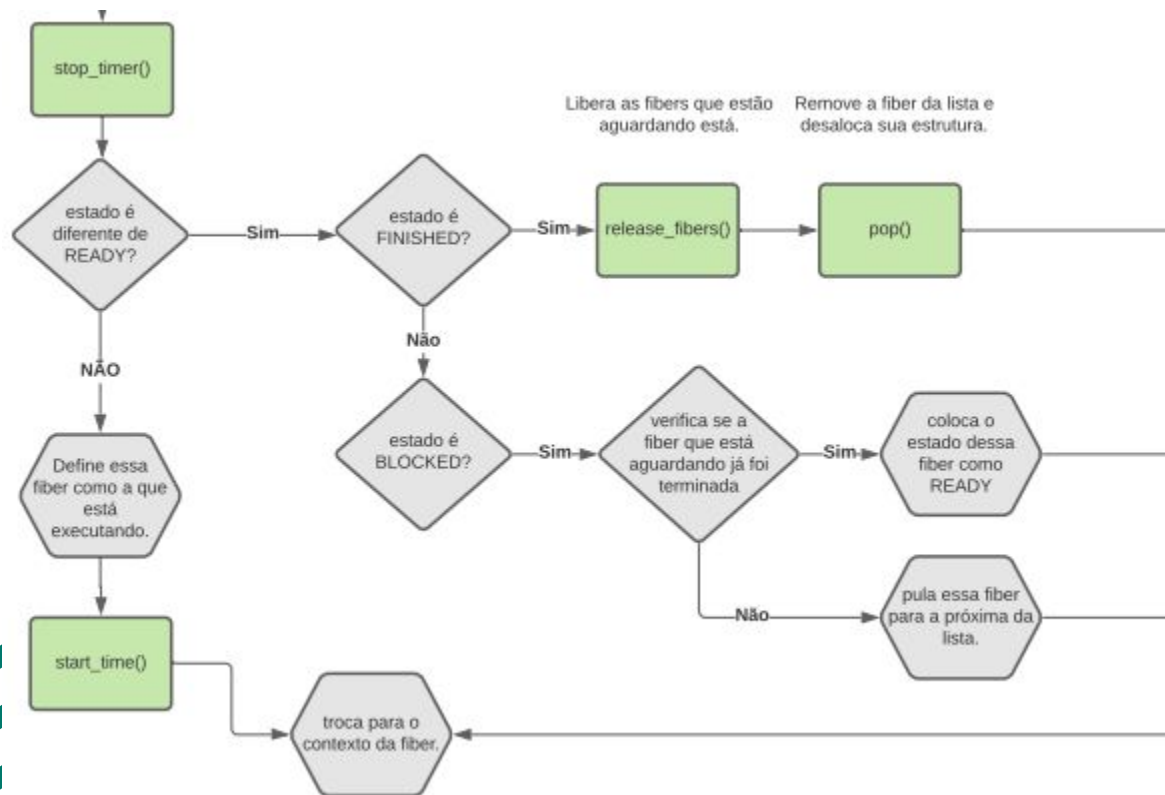
salva o contexto da fiber  
atual e troca para o  
contexto do escalonador

scheduler()

Verifica o estado da fiber e  
lida com ela de acordo.

stop\_timer()

## 11. preempt






## 11. preempt

```
/**
 * @name preempt()
 *
 * @brief Handler do sinal SIGVTALRM lançado pelo timer quando expirado. Salva o
 * contexto da fiber atual e troca para o contexto do escalonador.
 *
 * @return identificador da fiber.
 */
void preempt()
{
    /**
     * swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
     *
     * Salva o contexto atual na variável apontada por oucp; Atribui contexto de
     * execução que é apontado pela variável ucp. Em outras palavras, troca o
     * contexto atual (oucp) pelo contexto em ucp.
     */
    if (swapcontext(&fiber_list->running->context, &scheduler_ctx) == -1)
    {
        perror("swapcontext failed at preempt.");
        return;
    }
}
```

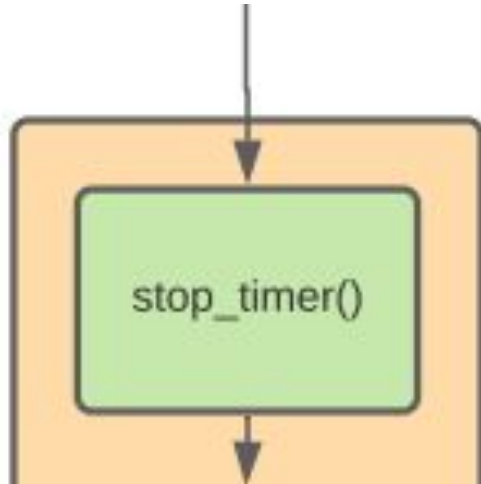
## 12. fiber\_create

Ao ser chamada pelo usuário da biblioteca vai parar o timer; alocar uma estrutura para uma fiber; criar seu contexto; inicializar suas variáveis; chamar a função push para adiciona-la na lista; retornar o identificador; e por fim iniciar o timer.



fiber\_create()

## 12. fiber\_create

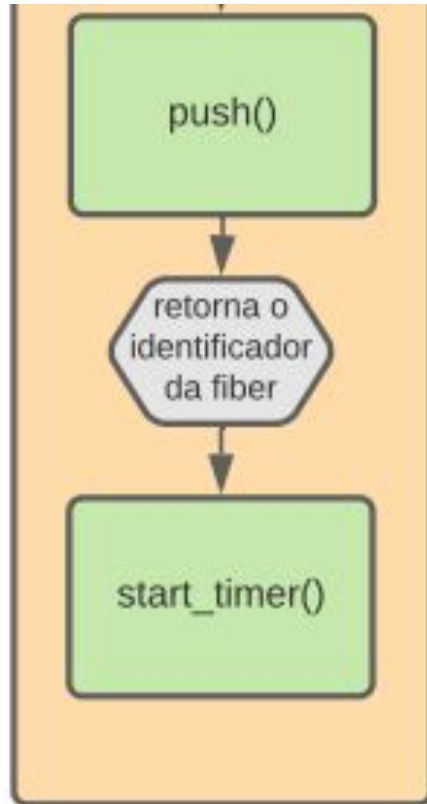


Para a execução do timer.

## 12. fiber\_create



## 12. fiber\_create



Insere uma fiber no final da lista de fibers. E reordena a lista.

Starta a execução do timer.  
Para um timer VIRTUAL, isso é baseado no tempo de execução do processo.

## 12. fiber\_create

fiber\_Create

@name fiber\_create(fiber\_t \*fiber, void \*(\*start\_routine)(void \*), void \*arg)

@brief Cria uma fiber (thread no user-space).

@param fiber identificador que será retornado por referência.

@param start\_routine rotina que será executada.

@param arg argumento que será passados para a rotina.

@return 0 para sucesso; -1 para falha.

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

1. Desativa o timer

```
stop_timer();
```

2. Cria a variável que irá armazenar o contexto da fiber que será criada

```
ucontext_t context;
```

3. Verifique se o ponteiro fiber é igual a null. Se for, a função é encerrada ao retornar um código de erro.

```
if (fiber == NULL)  
| | return -1;
```

## 12. fiber\_create

4. Aloca uma nova fiber

```
Fiber * new_node = (Fiber *)malloc(sizeof(Fiber));
```

5. Verifica se a fiber não foi alocada

```
if (new_node == NULL)
{
    perror("malloc failed at fiber_create.");
    return -1;
}
```

## 12. fiber\_create

### 6. Chama a função getcontext

Argumento: &fiber->context (Endereço do campo context presente no ponteiro para a struct fiber)

A função getcontext(ucontext\_t \*ucp) inicializa a estrutura apontada por ucp para o contexto atual da thread que fez a chamada. O tipo ucontext\_t para o qual ucp aponta define o contexto do usuário e inclui o conteúdo do atual contexto de execução como registradores, a máscara de sinal e a pilha de execução atual.

### 7. Verifica se ocorreu um erro na obtenção do contexto da fiber

```
if (getcontext(&fiber->context) == -1) {  
    perror("getcontext failed at init_fiber_context.");  
    return -1;  
}
```



## 12. fiber\_create

### 8. Definição do `uc_link`

```
context.uc_link = &scheduler_ctx;
```

Ponteiro para o contexto que será retomado quando este contexto retornar;

### 9. Alocação da pilha

```
context.uc_stack.ss_sp = malloc(FIBER_STACK_SIZE);
```

### 10. Definição do tamanho da pilha

```
context.uc_stack.ss_size = FIBER_STACK_SIZE;
```

### 11. Definição da flag da pilha

```
context.uc_stack.ss_flags = 0;
```

### 12. Verifica se a pilha não foi alocada

```
if (context.uc_stack.ss_sp == 0)
{
    perror("stack malloc failed at fiber_create.");
    return -1;
}
```

## 12. fiber\_create

13.. Chama a função makecontext

```
makecontext(&context, (void (*)(void))start_routine, 1, arg);
```

A função makecontext(ucontext\_t \*ucp, (void \*func)(), int argc, ...) modifica o contexto especificado por ucp, que foi inicializado usando getcontext(). Quando este contexto é retomado usando swapcontext() ou setcontext(), a execução do programa continua chamando func, passando os argumentos que seguem argc na chamada da makecontext().

14. O campo context pertencente ao ponteiro new\_node (ponteiro para struct Fiber) recebe o contexto que foi criado

```
new_node->context = context;
```

## 12. fiber\_create

15. Chama a função init\_fiber\_attr

```
init_fiber_attr(new_node);
```

Argumento:

new\_node (Ponteiro para a nova fiber que foi criada)

16. Chama a função push

```
push(new_node);
```

Argumento:

new\_node (Ponteiro para a nova fiber que foi criada)

17. O conteúdo do ponteiro fiber recebe o ponteiro para a fiber que foi criada

```
*fiber = new_node;
```

18. Inicia o timer

```
start_timer();
```

## 13. init\_fiber\_attr

init\_fiber\_attr

```
void init_fiber_attr(Fiber *new_node)
```

@name init\_fiber\_attr(Fiber \*new\_node)

@brief Inicializa as variáveis da fiber.

@param new\_node ponteiro para fiber para inicializar seus valores.

1. Define qual será a próxima fiber

```
new_node->next = NULL;
```

2. Define qual é o estado da fiber atual

```
new_node->status = STATE_READY;
```

3. Define qual é o retorno da fiber atual

```
new_node->retval = NULL;
```

4. Define o retorno da fiber que a fiber atual estava esperando

```
new_node->join_rval = NULL;
```

5. Define qual é a fiber que a fiber atual está esperando

```
new_node->joinFiber = NULL;
```

6. Define a lista de espera da fiber atual

```
new_node->waitList = NULL;
```

## 14. push

.push

```
void push(Fiber *fiber)
```

@name push(Fiber \*fiber)

@brief Insere a fiber no final da lista de fibers. Caso a lista seja nula chama a função `init_fiber_list()`.

@param fiber ponteiro para fiber que será inserida.

1. Verifica se a lista de fibers não foi alocada
2. Se não tiver sido alocada, a função `init_fiber_list()` é chamada

```
if (fiber_list == NULL)
    init_fiber_list();
```

4. A cabeça da lista de fibers se torna o próximo fiber

```
fiber->next = fiber_list->head;
```

5. A fiber atual se torna a fiber posterior à cauda da lista de fibers

```
fiber_list->tail->next = fiber;
```

6. A fiber atual se torna a cauda da lista de fibers

```
fiber_list->tail = fiber;
```

7. A fiber atual se torna a fiber posterior a anterior

```
fiber->prev->next = fiber;
```

8. O tamanho da fiber list é incrementado

```
fiber_list->size++;
```

## 15. start\_timer

start\_timer

```
void start_timer()
```

@name start\_timer()

@brief Inicia o timer com os valores de time slice.

@return sem retorno.

1. Define o tv\_sec do it\_value

```
timer.it_value.tv_sec = TIME_SLICE_SEC;
```

2. Define o tv\_usec do it\_value

```
timer.it_value.tv_usec = TIME_SLICE_USEC;
```

it\_value é uma estrutura que representa o tempo até a expiração do timer;

tv\_sec são os segundos até a expiração do timer;

tv\_usec são os microsegundos até a expiração do timer;

## 15. start\_timer

3. Define o tv\_sec do it\_interval

```
timer.it_interval.tv_sec = TIME_SLICE_SEC;
```

4. Define o tv\_usec do it\_interval

```
timer.it_interval.tv_usec = TIME_SLICE_USEC;
```

it\_interval é uma estrutura que representa o valor que será colocado no it\_value após a expiração;

tv\_sec são os segundos até a expiração do timer;

tv\_usec são os microsegundos até a expiração do timer;

5. Chama a função setitimer

6. Atribui o valor do interval timer;

7. Verifica tal função retornou um código de erro

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

```
setitimer(int which, struct itimerval * new, struct itimerval * old);
```

ITIMER\_VIRTUAL => Especifica que deve decrementar o timer com base no tempo de execução do processo. Sua expiração gera um sinal do tipo SIGVTALRM.



## 16. stop\_timer

stop\_timer

```
void stop_timer()
```

@name stop\_timer()

@brief Para o timer.

1. tv.sec recebe o valor 0

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

2. tv\_usec recebe o valor 0

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

3. Chama a função setitimer

4. Atribui o valor do interval timer;

5. Verifica tal função retornou um código de erro

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

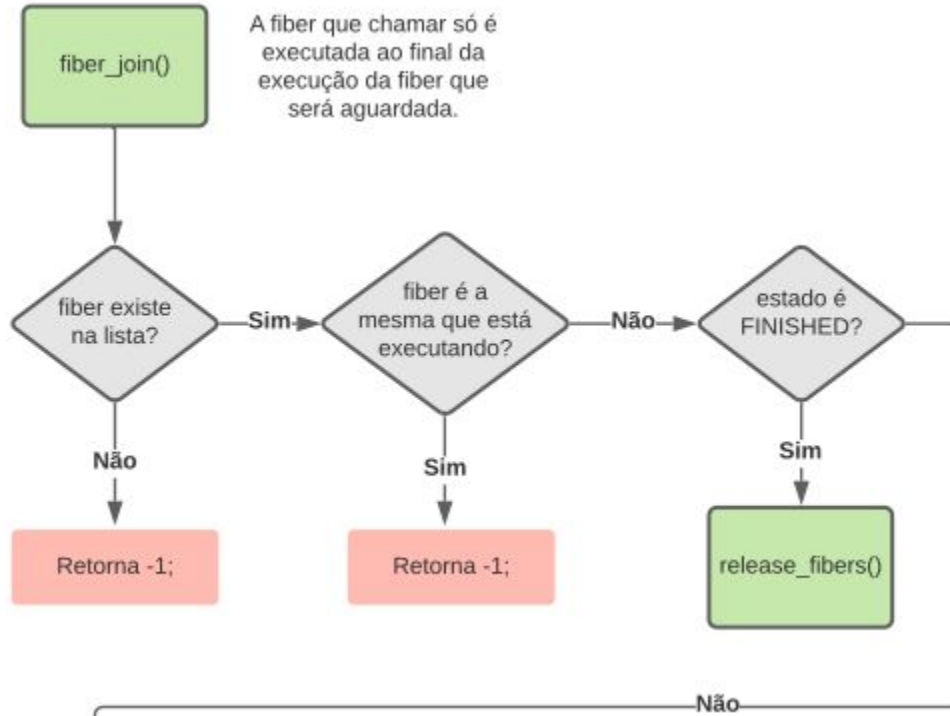
setitimer(int which, struct itimerval \* new, struct itimerval \* old);

Atribui o valor do interval timer

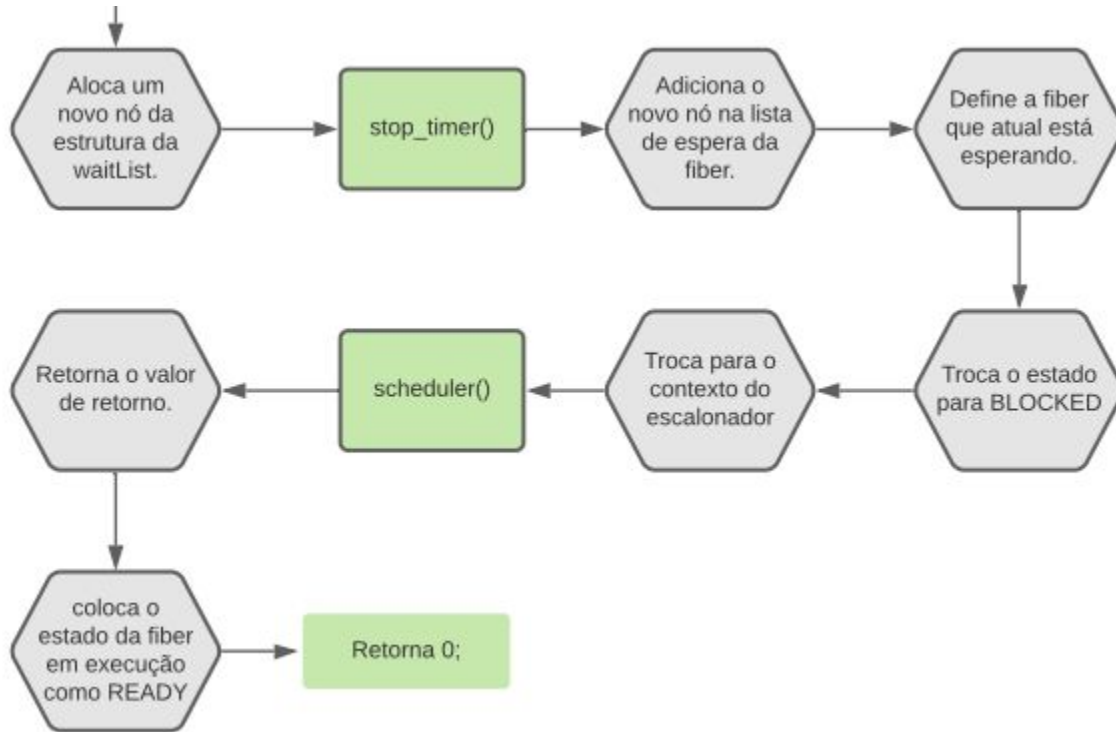
ITIMER\_VIRTUAL => Especifica que deve decrementar o timer com base no tempo de execução do processo. Sua expiração gera um sinal do tipo SIGVTALRM, tal sinal será recebido na função init\_preempt, esse recebimento fará com que a função preempt seja chamada.



## 17. fiber\_join



## 17. fiber\_join



## 17. fiber\_join

fiber\_join

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

@name fiber\_join(fiber\_t fiber, void \*\*retval)

@brief Coloca a fiber atual em espera para execução até o fim de outra fiber.

@param retval endereço para onde será colocado o valor de retorno da fiber.

Caso seja nulo será ignorado.

@return 0 para sucesso; -1 para falha.

1. Cria a variável i

```
int i = 0;
```

2. Cria o ponteiro fiber\_node

```
Fiber *fiber_node = NULL;
```

3. Define uma condição em loop para a função continuar

```
for (fiber_node = fiber_list->head; fiber != fiber_node && ++i <= fiber_list->size; fiber_node = fiber_node->next);
```

3. Se i for maior que o tamanho da lista, a função se encerra ao retornar o código de erro

```
if (i > fiber_list->size)
    return -1;
```

4. Verifica se a fiber a ser esperada é a que está executando

```
if (fiber_node == fiber_list->running)
    return -1;
```

## 17. fiber\_join

5. Verifica se a fiber que deveria terminar antes já terminou

```
if (fiber_node->status == STATE_FINISHED)
{
    release_fibers(fiber_node->waitList);
    return 0;
}
```

6. Aloca o nó de espera

```
Waiting *waitingNode = malloc(sizeof(Waiting));
```

7. Verifica se o nó de espera não foi alocado

```
if (waitingNode == NULL)
{
    perror("malloc failed at fiber_join.");
    return -1;
}
```

8. Define o id do nó de espera

```
waitingNode->id = fiber;
```

9. Define o nó posterior ao nó de espera

```
waitingNode->next = NULL;
```

10. Para o timer, pois o código irá entrar em uma área crítica

```
stop_timer(NULL);
```

## 17. fiber\_join

11. Adiciona um nó na lista de espera da fiber a ser guardada

```
if (fiber_node->waitList == NULL)
{
    fiber_node->waitList = (Waiting *)waitingNode;
}
else
{
    Waiting *waitingTop = (Waiting *)fiber_node->waitList;
    fiber_node->waitList = (Waiting *)waitingNode;
    fiber_node->waitList->next = (Waiting *)waitingTop;
}
```

12. Define a fiber que a fiber atual está esperando

```
fiber_list->running->joinFiber = (Fiber *)fiber_node;
```

13. Define que a fiber atual está esperando

```
fiber_list->running->status = STATE_BLOCKED;
```

14. Troca para o contexto do escalonador

```
if (swapcontext(&fiber_list->running->context, &scheduler_ctx) == -1)
{
    perror("swapcontext failed at fiber_join.");
    return -1;
}
```

## 17. fiber\_join

15. Recupera o valor de retorno da fiber que estava sendo aguardada.

Caso NULL tenha sido passado como argumento para `retval`, nada mais é feito.

Caso a fiber que estava sendo aguardada por esta tenha sido destruída, as rotinas de destruição já distribuíram os valores de `retval` corretamente para os atributos `join_rval` das head que estavam aguardando-a.

```
if (retval != NULL)
{
```

15.1 Caso a `joinFiber` não tenha sido destruída ainda, o `retval` é recuperado diretamente dela

```
if (fiber_list->running->join_rval == NULL && fiber_list->running->joinFiber != NULL)
| | *retval = fiber_list->running->joinFiber->retval;
```

15.2 Caso contrário, o `retval` é recuperado do atributo `join_rval` da própria fiber que chamou `fiber_join()`

```
else
| | *retval = fiber_list->running->join_rval;
```

15.3 Reseta os `retvals` da fiber

```
fiber_list->running->retval = NULL;
fiber_list->running->join_rval = NULL;
```

16. Define o status da fiber atual como pronta para executar

```
fiber_list->running->status = STATE_READY;
```

## 18. fiber\_exit

Coloca o estado da fiber como FINISHED, atribui o valor de retorno e chama o escalonador



## 18. fiber\_exit

### 12. fiber\_exit

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

@name fiber\_exit(void \*retval;

@brief Troca o status da fiber atual para STATE\_FINISHED e atribui o endereço para o valor de retorno. Quando identificada pelo scheduler será desalocada e nunca mais será executada.

1. O campo retval (retorno da fiber) pertencente ao campo running (fiber que está em execução) pertencente à lista fibers recebe o retorno da fiber

```
fiber_list->running->retval = retval;
```

2. O campo state pertencente ao campo running (fiber que está em execução) pertencente à lista de fibers recebe a variável global STATE\_FINISHED

```
fiber_list->running->state = STATE_FINISHED;
```

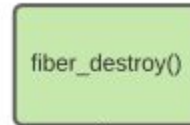
3. Chama a função preempt

```
preempt();
```

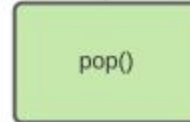


## 19. fiber\_destroy

Função para o usuário  
destruir uma fiber.



Verifica se fiber não é nula. E se seu estado é FINISHED. Remove da lista e desaloca a estrutura.



# 19. fiber\_destroy

## fiber\_destroy

```
int fiber_destroy(fiber_t fiber)
```

@name fiber\_destory(fiber\_t fiber)

@brief Desaloca a fiber.

@param fiber - identificador da fiber que deve ser desalocada.

@return 0 para sucesso; -1 para falha.

1. Cria a variável i

```
int i = 0;
```

2. Cria o ponteiro fiber\_node

```
Fiber *fiber_node = NULL;
```

3. Define uma condição em loop para a função continuar

```
for (fiber_node = fiber_list->head; fiber != fiber_node && ++i <= fiber_list->size; fiber_node = fiber_node->next);
```

4. Se o i for maior que o tamanho da lista, a função se encerra ao retornar o código de erro

```
if (i > fiber_list->size)
    return -1;
```

5. Chama a função pop

6. Se tal função retornar o valor null, a função se encerra ao retornar o código de erro

```
if (pop(fiber_node) == NULL)
    return -1;
```

## 20. pop

pop

```
Fiber *pop(Fiber *fiber)
{
```

@name pop(Fiber \*fiber)

@brief Libera a memória da fiber e remove ela da lista.

@param fiber - fiber que será desalocada.

@return próxima fiber na lista.

1. Verifica se o ponteiro fiber é igual a null. Se for, a função se encerra ao retornar null

```
    if (fiber == NULL)
        return NULL;
```

2. Verifica se o tamanho da lista é igual a 0. Se for, a função se encerra ao retornar null

```
    if (fiber_list->size == 0)
        return NULL;
```

3. Verifica se o estado da fiber é diferente de `STATE_FINISHED` (estado concluído). Se for, a função se encerra ao retornar null.

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

4. A primeira fiber da lista de fibers se torna a fiber anterior

```
    Fiber *prev_fiber = fiber_list->head;
```

5. O ponteiro `next_fiber` recebe a próxima fiber

```
    Fiber *next_fiber = fiber->next;
```

6. Enquanto a fiber não for a fiber posterior à fiber anterior:

```
    while (fiber != prev_fiber->next)
    {
```

- 6.1 A fiber anterior recebe a fiber posterior à fiber anterior

```
        prev_fiber = prev_fiber->next;
```

7. A fiber posterior a anterior recebe a fiber posterior

```
        prev_fiber->next = fiber->next;
```

8. Verifica se a fiber é igual à primeira fiber da lista de fibers

```
        if (fiber == fiber_list->head)
```

- 8.1 A primeira fiber da lista de fibers recebe a próxima fiber

```
        fiber_list->head = fiber->next;
```

## 20. pop

9. Verifica se fiber é igual à cauda da lista de fibers

```
if (fiber == fiber_list->tail)
```

9.1 A cauda da lista de fibers recebe a fiber anterior

```
fiber_list->tail = prev_fiber;
```

10. Libera a pilha da fiber

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

11. Libera a fiber

```
free(fiber);
```

12. O ponteiro fiber recebe o valor null

```
fiber = NULL;
```

---

13. O tamanho da fiber é decrementado

```
fiber_list->size--;
```

14. Verifica se o tamanho da lista de fibers é igual a 0

```
if (fiber_list->size == 0)
{
```

14.1 A fiber anterior recebe o valor null

```
prev_fiber = NULL;
```

14.2 A fiber posterior recebe o valor null

```
next_fiber = NULL;
```

15. Retorna a fiber posterior

```
return next_fiber;
```

## 21. fiber\_self

. fiber\_self

```
fiber_t fiber_self() {
```

@name fiber\_self()

@brief retorna o identificador da [fiber](#) em execução.

@return identificador da fiber.

1. Retorna a fiber (pertencente à lista de fibers) que está em execução

```
  return fiber_list->running;
```

# Execução

1. Comando para gerar o código executável

```
gcc fiber.c fiber.h main.c -o main.exe
```

2. Executa código executável

```
./main.exe
```

- 3.

3.1 Cria a fiber 1, 2 e 3

3.2 Fiber 2 e 3 vai para a fila de espera

3.2 Executa a fiber 1

```
Criou a fiber 1 = 0x555c80328680
Criou a fiber 2 = 0x555c80338e80
Criou a fiber 3 = 0x555c80349270
Rotina da thread 0x555c80328680
Olá mundo! :D
```

- 4.

4.1 Finaliza a execução da fiber 1

4.2 Fiber 2 sai da fila de espera

4.3 Executa a fiber 2

```
Adeus :C
```

```
Rotina da thread 0x555c80338e80
Olá mundo! :D
```

- 6.

6.1 Finaliza a execução da fiber 2

6.2 Fiber 3 sai da fila de espera

6.3 Executa a fiber 3

```
Adeus :C
```

```
Rotina da thread 0x555c80349270
Olá mundo! :D
```

- 8.

8.1 Finaliza a execução da fiber 3

```
Adeus :C
```