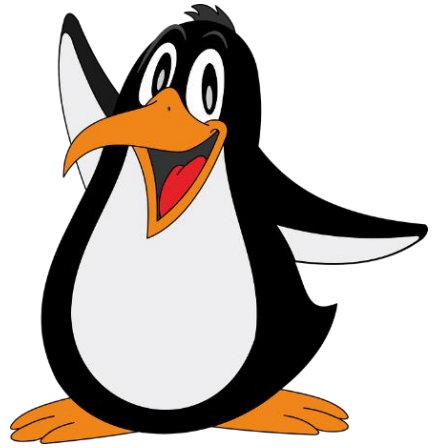
Sistemas Operacionais II

Threads – Parte 1

Fabricio Breve
fabricio.breve@unesp.br
https://www.fabriciobreve.com

Agenda



- Semelhanças e Diferenças entre *Threads* e Processos
- *Pthreads*: API de threads padrão POSIX
- Criação de *Threads*
- Saindo de *Threads*
- Passando dados para *Threads*
- Juntando *Threads*
- Valor de Retorno de *Threads*
- *Threads IDs*
- Atributos de *Threads*
- Cancelamento de *Threads*
- *Threads* síncronas e assíncronas
- Seções Críticas Não Canceláveis
- Dados Específicos de *Thread*
- Manipuladores de Limpeza

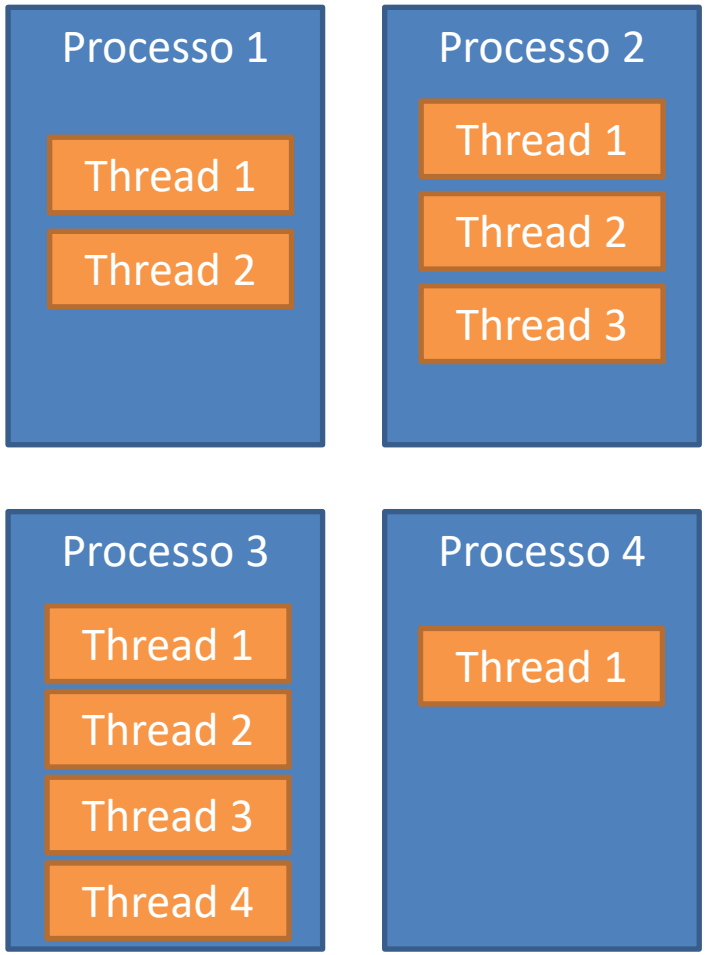
Threads

- Semelhanças com processos:
  - Permitem ao programa fazer mais de uma tarefa ao mesmo tempo.
  - O núcleo do Linux faz o escalonamento das mesmas assincronamente, interrompendo cada *thread* periodicamente para que as demais possam executar.

https://kidscraftroom.com/penguin-craft-penguin-ornaments/

Threads

- *Threads* existem dentro de processos.
  - Quando um programa é chamado, o Linux cria um novo processo e dentro daquele processo cria uma única *thread*, que executa o programa sequencialmente.
  - Uma *thread* pode criar *threads* adicionais.
    - Todas as *threads* de um mesmo processo rodam o mesmo programa.
    - Porém podem executar diferentes partes do programa em qualquer instante de tempo.



Diferenças entre Threads e Processos

- Processos:
  - Processo filho inicialmente roda o mesmo programa que o pai (mesma memória virtual, descritores, etc.).
    - Mas processo filho pode modificar sua memória, fechar descritores, etc. sem afetar seu pai e vice-versa.
- *Threads*:
  - Nada é copiado, mas sim compartilhado (memória, descritores de arquivos, etc.).
    - Se uma *thread* muda o valor de uma variável, outras threads acessarão o valor modificado.
    - Se uma *thread* fecha um descritor de arquivo, outras *threads* não poderão mais ler ou escrever para tal descritor.

Threads

- O GNU/Linux implementa a API de *threads* do padrão POSIX (*pthreads*).
- Todas as funções e tipos de dados de threads estão declarados no arquivo de cabeçalho `<pthread.h>`
- As funções de *thread* **não** estão incluídas na biblioteca C padrão, mas sim em **libpthread**
  - Adicione **`-lpthread`** na linha de comando quando for ligar seu programa.

Criação de *Threads*

- Cada *thread* em um processo é identificada por um *thread ID*.
  - Em C ou C++, use o tipo **`pthread_t`**
  - Ao ser criada, uma *thread* executa a *função de thread*.
    - Função comum com o código que a *thread* deve executar.
    - Recebe um único argumento do tipo **`void*`**
      - Usado para passar dados para a nova *thread*.
    - Também tem **`void*`** como tipo de retorno.
      - Usado para retornar dados ao criador.

Criação de *Threads*

- A função **`pthread_create`** cria uma nova *thread*.
  - Ela recebe os seguintes argumentos:
    1. Ponteiro para uma variável **`pthread_t`**, na qual o *thread ID* da nova variável será armazenado.
    2. Ponteiro para um objeto *atributos de thread* que fornece detalhes de como a *thread* interage com o restante do programa (será discutido mais adiante), ou **`NULL`** para criar a thread com os atributos padrão.
    3. Ponteiro para a *função de thread*.
    4. Um argumento atributo para a *thread* do tipo **`void*`** que será passado como argumento para a *função de thread*.

Criação de *Threads*

- Uma chamada para **`pthread_create`** retorna imediatamente, e a *thread* original continua executando as instruções após a chamada.
- Ao mesmo tempo, a nova *thread* começa a execução da *função de thread*.

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

/* Imprime x's para stderr.  O parâmetro não é usado. Não retorna.  */

void* print_xs (void* unused)
{
  while (1)
    fputc ('x', stderr);
  return NULL;
}

/* O programa principal.  */

int main ()
{
  pthread_t thread_id;
  /* Cria uma nova thread.  A nova thread executará a função print_xs.  */
  pthread_create (&thread_id, NULL, &print_xs, NULL);
  /* Imprime o's continuamente para stderr.  */
  while (1)
    fputc ('o', stderr);
  return 0;
}
```

thread-create.c

Veja em execução:
https://youtu.be/V9KI5sySraM

- Compile e ligue com:
  - `gcc -o thread-create thread-create.c -lpthread`

Saindo de uma *thread*

- Em circunstâncias normais uma *thread* termina quando:
  - Retorna de uma *função de thread*, como no exemplo anterior.
  - Chama explicitamente **`pthread_exit`**
    - Pode ser chamada dentro da *função de thread* ou de outras funções chamadas diretamente ou indiretamente pela *função de thread.*
    - O argumento é o valor de retorno da *thread*.

Passando dados para *threads*

- O argumento da *thread* é uma maneira conveniente de passar dados para *threads*.
  - Passe um ponteiro para uma estrutura ou *array* de dados.
  - Uma estratégia comum é criar uma estrutura para cada *função de thread*, que contém os “parâmetros” que a função espera.
  - Dessa forma a mesma função de *thread* pode ser reusada para muitas *threads.*
    - Mesmo código, dados diferentes.

thread-create2.c

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

/* Parametros para a função de imprimir.  */

struct char_print_parms
{
  /* O caracter a ser impresso.  */
  char character;
  /* A quantidade de vezes para imprimí-lo.  */
  int count;
};

/* Imprime uma quantidade de caracteres para stderr, conforme dado por PARAMETERS,
   que é um ponteiro para uma estrutura char_print_parms.  */

void* char_print (void* parameters)
{
  /* Cast do ponteiro para o tipo correto.  */
  struct char_print_parms* p = (struct char_print_parms*) parameters;
  int i;

  for (i = 0; i < p->count; ++i)
    fputc (p->character, stderr);
  return NULL;
}
```

```c
/* O programa principal.  */

int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;

  /* Cria um novo thread para imprimir 30000 x's.  */
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

  /* Cria um novo thread para imprimir 20000 o's.  */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

  /* Certifique-se de que a primeira thread terminou.  */
  pthread_join (thread1_id, NULL);
  /* Certifique-se de que a segunda thread terminou.  */
  pthread_join (thread2_id, NULL);

  /* Agora podemos retornar de forma segura.  */
  return 0;
}
```

Veja em execução:
https://youtu.be/9paoSXIlqnI

Juntando *Threads*

- Note as seguintes linhas do exemplo anterior:

```
/* Certifique-se de que a primeira thread terminou.  */
pthread_join (thread1_id, NULL);
/* Certifique-se de que a segunda thread terminou.  */
pthread_join (thread2_id, NULL);
```

- Por que elas são necessárias?
  - Quando **`main`** termina de executar, a memória referente às variáveis locais é liberada.
    - Poderia acontecer de **`main`** terminar e desalocar as estruturas de opções enquanto as *threads* ainda estivessem acessando-as.

Juntando *Threads*

- A função **`pthread_join`** espera que threads terminem, tal qual **`wait`** faz para processos.
- **`pthread_join`** recebe dois argumentos:
  - *Thread ID* da thread a ser esperada.
  - Ponteiro para uma variável **`void*`** que receberá o valor de retorno da *thread*.
    - Use **`NULL`** se não se importa com o retorno da *thread*.

Valores de Retorno de *Threads*

- Se o segundo argumento de **`pthread_join`** não for nulo, o valor retornado será colocado no local apontado por tal argumento.
  - Se quiser passar um único valor inteiro, pode fazê-lo usando *cast* do valor para **`void*`** e então fazer um *cast* de volta para **`int`** após chamar **`pthread_join`**

primes.c

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

/* Computa sucessivos números primos (bastante ineficientemente). Retorna o
   N-ésimo número primo, onde N é o valor apontado por *ARG.  */

void* compute_prime (void* arg)
{
  int candidate = 2;
  int n = *((int*) arg);

  while (1) {
    int factor;
    int is_prime = 1;

    /* Testa se é primo por sucessivas divisões.  */
    for (factor = 2; factor < candidate; ++factor)
      if (candidate % factor == 0) {
        is_prime = 0;
        break;
      }
    /* Este é o primo que estavamos procurando?  */
    if (is_prime) {
      if (--n == 0)
        /* Retorna o número primo desejado como o valor de retorno da thread.  */
        return (void*) candidate;
    }
    ++candidate;
  }
  return NULL;
}
```

Observação: o aviso “cast to pointer from integer of different size” ocorre em sistemas de 64 bits porque em tais sistemas **`int`** utiliza 32 bits e **`void*`** utiliza 64 bits. Para tornar o programa portável utilize **`intptr_t`** em vez de **`int`** (inclua **`<stdint.h>`** para utilizá-lo).

```c
int main ()
{
  pthread_t thread;
  int which_prime = 5000;
  int prime;

  /* Inicie a computação da thread, até o 5000º número primo.  */
  pthread_create (&thread, NULL, &compute_prime, &which_prime);
  /* Faça algum outro trabalho aqui...  */
  /* Espere a thread do número primo completar e pegue o resultado.  */
  pthread_join (thread, (void*) &prime);
  /* Imprima o maior número primo computado.  */
  printf("O %dº número primo é %d.\n", which_prime, prime);
  return 0;
}
```

primes.c

com **intptr_t** em vez de **int**

```c
#include <pthread.h>
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

/* Computa sucessivos números primos (bastante ineficientemente). Retorna o
   N-ésimo número primo, onde N é o valor apontado por *ARG.  */

void* compute_prime (void* arg)
{
  intptr_t candidate = 2;
  intptr_t n = *((intptr_t*) arg);

  while (1) {
    int factor;
    int is_prime = 1;

    /* Testa se é primo por sucessivas divisões.  */
    for (factor = 2; factor < candidate; ++factor)
      if (candidate % factor == 0) {
        is_prime = 0;
        break;
      }
    /* Este é o primo que estávamos procurando?  */
    if (is_prime) {
      if (--n == 0)
        /* Retorna o número primo desejado como o valor de retorno da thread.  */
        return (void*) candidate;
    }
    ++candidate;
  }
  return NULL;
}
```

```c
int main ()
{
  pthread_t thread;
  intptr_t which_prime = 5000;
  intptr_t prime;

  /* Inicie a computação da thread, até o 5000º número primo.  */
  pthread_create (&thread, NULL, &compute_prime, &which_prime);
  /* Faça algum outro trabalho aqui...  */
  /* Espere a thread do número primo completar e pegue o resultado.  */
  pthread_join (thread, (void*) &prime);
  /* Imprima o maior número primo computado.  */
  printf("O %" PRIdPTR "º número primo é %" PRIdPTR ".\n", which_prime, prime);
  return 0;
}
```

Veja em execução:
https://youtu.be/tMmpG82zmjo

Mais sobre *Thread IDs*

- A função **`pthread_self`** retorna o *thread ID* da thread na qual é chamada.
- Um *thread ID* pode ser comparado com outro *thread ID* utilizando-se a função **`pthread_equal`**

Atributos de *Thread*

- Utilizados para mudar o comportamento de uma *thread* individual.
  - Se for usado um ponteiro nulo no atributo de **`pthread_create`**, o comportamento padrão é assumido.
  - Se quiser criar uma *thread* com comportamento personalizado, crie um objeto *atributos de thread* para especificar outros valores de atributos.

Atributos de *Thread*

- Para especificar atributos de *threads* personalizados, siga os seguintes passos:
  1. Crie um objeto **`pthread_attr_t`**. A maneira mais fácil é declarar uma variável automática deste tipo.
  2. Chame **`pthread_attr_init`**, passando um ponteiro para este objeto. Isto inicializará os atributos com os valores padrões.
  3. Modifique o objeto de atributos para conter os valores de atributos desejados.
  4. Passe um ponteiro para o objeto de atributos ao chamar a função **`pthread_create`**.
  5. Chame **`pthread_attr_destroy`** para liberar o objeto de atributos. A variável **`pthread_attr_t`** não será desalocada e poderá ser reinicializada com **`pthread_attr_init`**.

Atributos de *Thread*

- Um mesmo objeto de atributos pode ser utilizado para inicializar várias *threads*.
- Após criar a *thread*, o objeto de atributos não é mais necessário.

Atributos de *Thread*

- Na maioria dos programas GNU/Linux, o único atributo de interesse é o *detach state*.
  - Os demais são mais utilizados em programação de tarefas de tempo real.
  - Por padrão, quando uma *thread* termina, seus recursos não são liberados, ela espera no sistema (da mesma forma que um processo zumbi) até que uma chamada **`pthread_join`** seja feita.
  - Uma *thread* destacada (com o atributo *detach* ativado) tem seus recursos liberados automaticamente quando termina.
    - Não pode ser sincronizada ou ter seu valor de retorno lido com **`pthread_join`**.

Atributos de *Thread*

- Para ativar o estado *detach* em um objeto de atributos de *thread*, use **`pthread_attr_stdetachstate`**.
  - O primeiro argumento é um ponteiro para o objeto de atributos de *thread*.
  - O segundo argumento é o estado de *detach* desejado.
    - Passe `PTHREAD_CREATE_DETACHED` como segundo argumento.
- Também é possível destacar uma *thread* já criada utilizando **`pthread_detach`**, mas o inverso não é possível.

```c
#include <pthread.h>

void* thread_function (void* thread_arg)
{
  /* Faça algum trabalho aqui...  */
  return NULL;
}

int main ()
{
  pthread_attr_t attr;
  pthread_t thread;

  pthread_attr_init (&attr);
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
  pthread_create (&thread, &attr, &thread_function, NULL);
  pthread_attr_destroy (&attr);

  /*  Faça trabalho aqui...  */

  /* Não é necessário juntar a segunda thread.  */
  return 0;
}
```

detached.c

Cancelamento de *Thread*

- Em circunstâncias normais uma *thread* termina retornando da *função de thread* ou chamando **`pthread_exit`**
- Porém, é possível que uma *thread* peça a outra para terminar.
  - Isto é chamado *cancelar* uma *thread*.
  - **`pthread_cancel`** é usado para cancelar uma *thread*.
    - O argumento é o ID da *thread.*
  - *Threads* canceladas que não são destacadas ainda precisam ser juntadas para liberar seus recursos.
  - O valor de retorno de uma *thread* cancelada é `PTHREAD_CANCELED`.

Cancelamento de *Thread*

- Frequentemente uma *thread* está em um estado em que não pode ser interrompida.
  - Exemplo:
    - Uma *thread* pode alocar alguns recursos, usá-los e então desalocá-los.
    - Se ela for interrompida, não desalocará os recursos e os mesmos vazarão.
  - Para evitar tal problema, uma *thread* pode controlar se e quando ela pode ser cancelada.

Cancelamento de *Thread*

- Com relação ao cancelamento, uma *thread* pode estar em um dos três seguintes estados:
  - **Assincronamente cancelável**: pode ser cancelada em qualquer ponto de sua execução.
  - **Sincronamente cancelável**: pode ser cancelada, mas não em qualquer ponto de sua execução. Requisições de cancelamento são colocadas em uma fila e a thread é cancelada somente quando atinge pontos específicos em sua execução.
  - **Incancelável**: tentativas de cancelá-la são silenciosamente ignoradas.
- Quando criada, uma *thread* é sincronamente cancelável.

Threads síncronas e assíncronas

- **`pthread_setcanceltype`** pode ser usado para mudar o estado de cancelamento de uma *thread*.
  - Afeta a *thread* que chama a função.
  - O primeiro argumento deve ser:
    - **`PTHREAD_CANCEL_ASYNCHRONOUS`** para tornar a thread assincronamente cancelável.
    - **`PTHREAD_CANCEL_DEFERRED`** para retorná-la ao estado de sincronamente cancelável.
  - O segundo argumento, se não for NULL, receberá um ponteiro para uma variável que receberá o estado de cancelamento anterior da *thread*.

Threads síncronas e assíncronas

- Exemplo:
  - A chamada a seguir, torna a *thread* que a chama assincronamente cancelável:
    - `pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS, NULL);`

Threads síncronas e assíncronas

- **`pthread_testcancel`** pode ser usado para testar se há um pedido de cancelamento pendente em uma *thread* sincronamente cancelável.
  - Caso exista, a *thread* é cancelada.
  - Deve ser chamado periodicamente durante computações longas, em pontos onde a *thread* pode ser cancelada sem vazar recursos ou produzir outros efeitos nocivos.
- Algumas outras funções são pontos de cancelamento implícitos, veja quais são elas na seção 7 do manual de **`pthreads`**
  - **`man 7 pthreads`**
  - Observe que algumas outras funções podem usar tais funções e assim tornarem-se indiretamente pontos de cancelamento.

Seções Críticas Não Canceláveis

- Uma *thread* pode desabilitar o cancelamento dela própria com a função **`pthread_setcancelstate`**
  - O primeiro argumento é `PTHREAD_CANCEL_DISABLE` para desativar o cancelamento, ou `PTHREAD_CANCEL_ENABLE` para reabilitar o cancelamento.
  - O segundo argumento, se não nulo, aponta para uma variável que receberá o estado de cancelamento anterior.
  - Exemplo:
    - **`pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,NULL)`**

Seções Críticas Não Canceláveis

- Com o uso de **`pthread_setcancelstate`** é possível implementar seções críticas.
  - Seção crítica: trecho de código que deve ser executado completamente ou não ser executado.
    - Não pode ser cancelado no meio da execução.
    - Exemplo:
      - Rotina de sistema bancário que transfere dinheiro de uma conta para outra:
        - » Adicione o valor na conta de destino.
        - » Diminua o mesmo valor na conta de origem.

```c
#include <pthread.h>
#include <stdio.h>
#include <string.h>

/* Um array de saldos em contas, indexado pelos números das contas.  */

float* account_balances;

/* Transfere DOLLARS da conta FROM_ACCT para a conta TO_ACCT.  Retorna
   0 se a transação foi bem sucedida, ou 1 se o saldo de FROM_ACCT é insuficiente. */

int process_transaction (int from_acct, int to_acct, float dollars)
{
  int old_cancel_state;

  /* Checar o saldo em FROM_ACCT.  */
  if (account_balances[from_acct] < dollars)
    return 1;

  /* Iniciar a seção crítica.  */
  pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, &old_cancel_state);
  /* Movimentar o dinheiro.  */
  account_balances[to_acct] += dollars;
  account_balances[from_acct] -= dollars;
  /* Final da seção crítica.  */
  pthread_setcancelstate (old_cancel_state, NULL);

  return 0;
}
```

critical-section.c

Seções Críticas Não Canceláveis

- É importante restaurar o estado de cancelamento antigo no final da seção crítica em vez de configurá-la incondicionalmente para PTHREAD_CANCEL_ENABLE
  - Isto permite chamar a função **`process_transaction`** de maneira segura estando dentro de outra seção crítica.
    - Neste caso sua função deixará o estado de cancelamento da maneira que o encontrou.

Dados Específicos de *Thread*

- Ao contrário de processos, *threads* compartilham o mesmo espaço de endereços.
  - Se uma *thread* modifica uma localização na memória (por exemplo, uma variável global), a mudança é visível para todas as outras *threads*.
    - Possibilidade de trabalhar com os mesmos dados sem necessidade de utilizar comunicação entre processos.

Dados Específicos de *Thread*

- Cada *thread* tem sua própria pilha.
  - Cada uma executa código diferente.
  - Cada chamada a uma função ou sub-rotina tem seu próprio conjunto de variáveis locais.
- Porém é possível duplicar certas variáveis para que cada *thread* tenha uma cópia separada.
  - O GNU/Linux oferece isso através de uma área de dados específicos de thread.
    - Variáveis em tal área são duplicadas para cada *thread*.
    - Não podem ser acessadas usando referências à variáveis da maneira tradicional.
    - Há funções específicas para atribuir e ler seus valores.

Dados Específicos de *Thread*

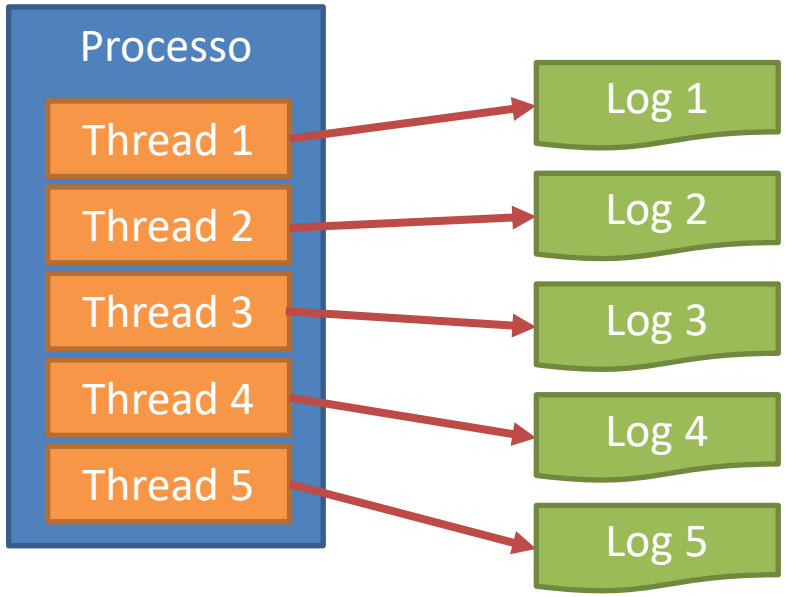
- Você pode criar quantos itens de dados específicos de *thread* quiser, todos do tipo `void*`
  - Cada item é referenciado por uma chave.
    - O valor da chave é usado por cada *thread* para acessar sua própria cópia da variável.
    - Utilize **`pthread_key_create`** para criar uma nova chave e, portanto, um novo item de dados.
      - Primeiro argumento é um ponteiro para uma variável **`pthread_key_t`**
      - O segundo argumento é uma função de limpeza.
        - » Passando um ponteiro de função aqui, tal função será chamada sempre que a *thread* sair, passando o valor específico de *thread* correspondente à chave como parâmetro.
          - Se o valor específico de *thread* for `NULL` a função não é chamada.
        - » A função de limpeza é chamada mesmo que a *thread* seja cancelada.
        - » Configure como `NULL` se não precisar de função de limpeza.

Dados Específicos de *Thread*

- Depois de criar uma chave, cada thread pode configurar seu valor específico usando **`pthread_setspecific`**
  - Primeiro argumento é a chave.
  - Segundo argumento é o valor `void*` específico a ser armazenado.
- Para recuperar um dado específico de *thread* chame **`pthread_getspecific`** passando a chave como argumento.

Dados Específicos de *Thread*

- Suponha que sua aplicação divide uma tarefa em múltiplas *threads* e que cada *thread* tem um arquivo de *log* separado.
  - A área de dados específica de *thread* é um bom lugar para armazenar o ponteiro para o arquivo de *log* de cada *thread* individual.



```c
#include <malloc.h>
#include <pthread.h>
#include <stdio.h>

/* A chave usada para associar um ponteiro de arquivo de log com cada thread.  */
static pthread_key_t thread_log_key;

/* Escreva MESSAGE para o arquivo de log correspondente à thread.  */

void write_to_thread_log (const char* message)
{
  FILE* thread_log = (FILE*) pthread_getspecific (thread_log_key);
  fprintf (thread_log, "%s\n", message);
}

/* Feche o ponteiro para arquivo de log THREAD_LOG.  */

void close_thread_log (void* thread_log)
{
  fclose ((FILE*) thread_log);
}

void* thread_function (void* args)
{
  char thread_log_filename[40];
  FILE* thread_log;

  /* Gerar o nome de arquivo para o arquivo de log desta thread.  */
  sprintf (thread_log_filename, "thread%d.log", (int) pthread_self ());
  /* Abrir o arquivo de log.  */
  thread_log = fopen (thread_log_filename, "w");
  /* Armazenar o ponteiro de arquivo em um dado específico de thread sob thread_log_key.  */
  pthread_setspecific (thread_log_key, thread_log);

  write_to_thread_log ("Thread iniciando.");
  /* Faça trabalho aqui... */

  return NULL;
}
```

Aumente o valor dessa *string* para 40 (em vez de 20 original) para evitar eventuais problemas com estouro de pilha

```c
int main ()
{
  int i;
  pthread_t threads[5];

  /* Cria uma chave para associar ao ponteiro de arquivo de log em
     dados específicos de thread.  Use close_thread_log para limpar o
     pointeiro de arquivo.  */
  pthread_key_create (&thread_log_key, close_thread_log);
  /* Criar threads para fazer o trabalho.  */
  for (i = 0; i < 5; ++i)
    pthread_create (&(threads[i]), NULL, thread_function, NULL);
  /* Esperar todas as threads terminarem.  */
  for (i = 0; i < 5; ++i)
    pthread_join (threads[i], NULL);
  return 0;
}
```

tsd.c

Veja em execução:
https://youtu.be/1GRFVJgQf_A

Manipuladores de Limpeza

- Também é possível especificar funções de limpeza sem criar novos itens de dados específicos de *thread*.
- O GNU/Linux oferece para isso os manipuladores de limpeza (*cleanup handlers*).

Manipuladores de Limpeza

- Manipulador de limpeza é uma função que deve ser chamada quando uma *thread* termina.
  - Usado para desalocar recursos quando a *thread* sai ou é cancelada em vez de terminar a execução de uma região de código.
  - Em circunstâncias normais, quando a *thread* não sai e não é cancelada, o recurso deve ser desalocado explicitamente e o manipulador de limpeza deve ser removido.

Manipuladores de Limpeza

- Para registrar um manipulador de limpeza, chame **`pthread_cleanup_push`**, passando um ponteiro para a função de limpeza e o valor de seu argumento `void*`
- A chamada a **`pthread_cleanup_push`** deve ser balanceada com uma chamada a **`pthread_cleanup_pop`**, que desregistra o manipulador de limpeza.
  - **`pthread_cleanup_pop`** recebe um inteiro como argumento.
    - Se for diferente de zero, a ação de limpeza é executada enquanto o manipulador é desregistrado.

cleanup.c

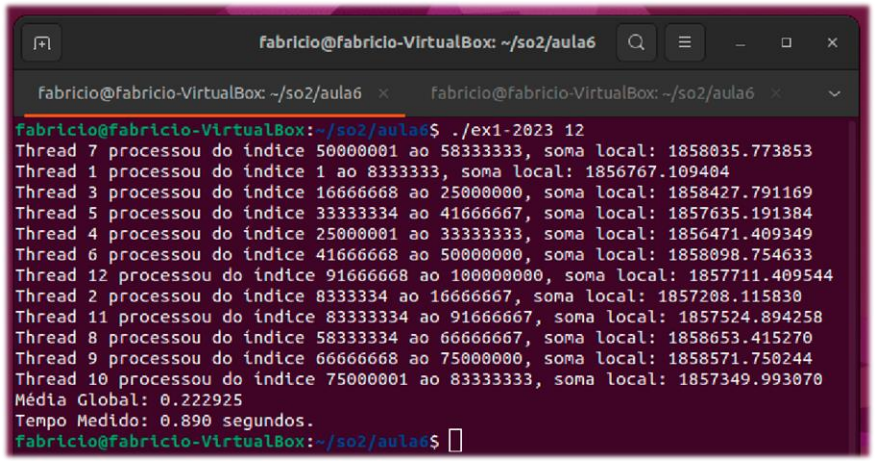
```c
#include <malloc.h>
#include <pthread.h>

/* Alocar um buffer temporário.  */

void* allocate_buffer (size_t size)
{
  return malloc (size);
}

/* Desalocar um buffer temporário.  */

void deallocate_buffer (void* buffer)
{
  free (buffer);
}

void do_some_work ()
{
  /* Alocar um buffer temporário.  */
  void* temp_buffer = allocate_buffer (1024);
  /* Registrar um manipulador de limpeza para este buffer, para desalocá-lo caso
     caso a thread saia ou seja cancelada.  */
  pthread_cleanup_push (deallocate_buffer, temp_buffer);

  /* Faça algum trabalho aqui que possa chamar pthread_exit ou possa ser e
     cancelado...  */

  /* Desregistrar manipulador de limpeza.  Como passamos um valor não-zero,
     isto na verdade faz a limpeza chamando deallocate_buffer.  */
  pthread_cleanup_pop (1);
}
```

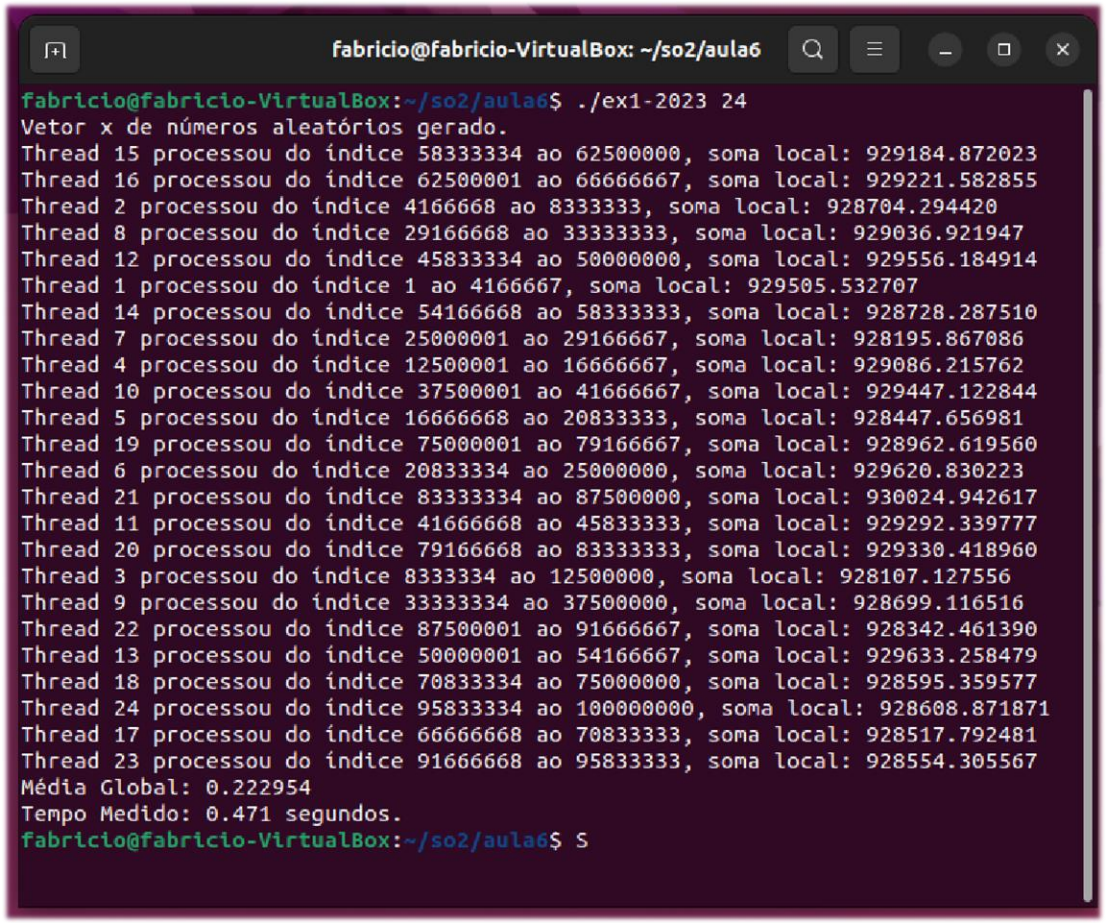
Exercício

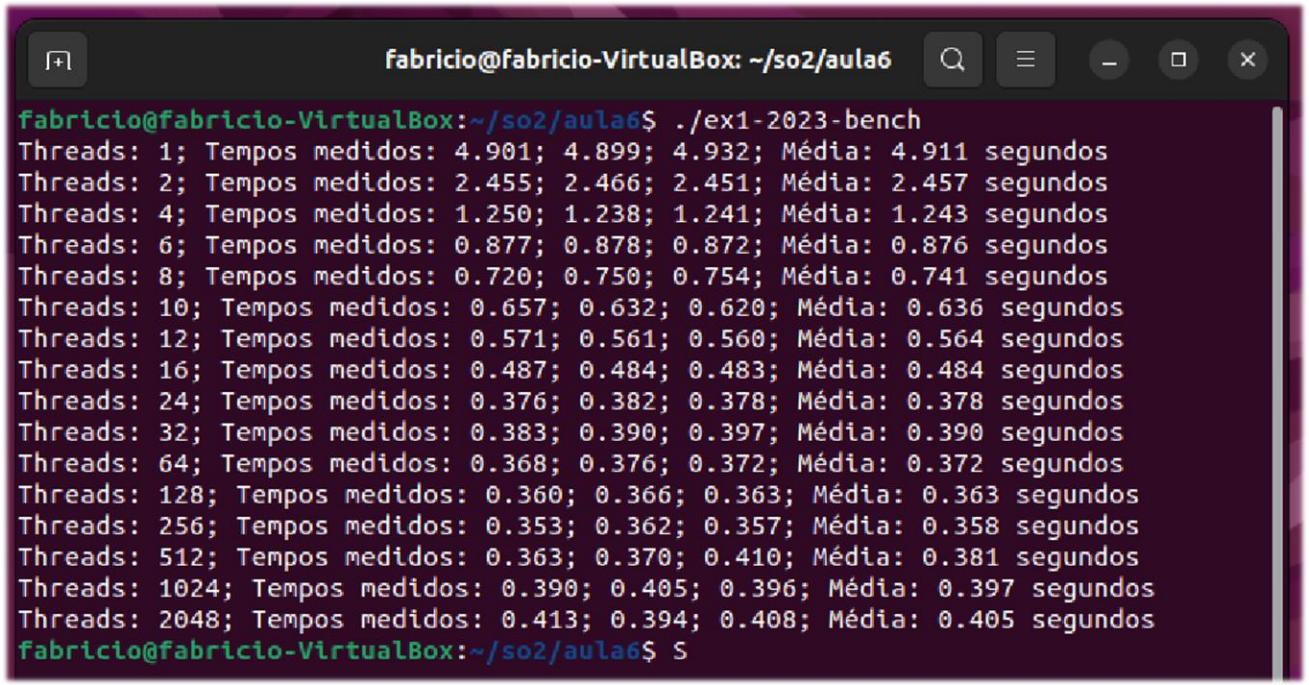
Veja em execução:
https://youtu.be/J7Fw18D0H-E

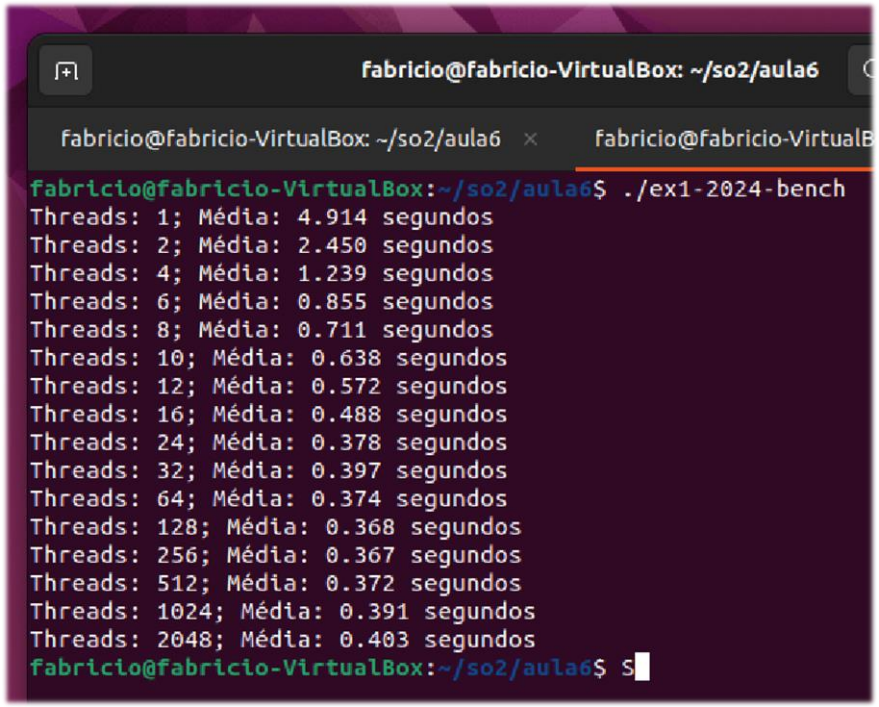
- Faça um programa que:
  - Receba como argumento a quantidade de *threads* que devem ser criadas para a fase de processamento.
  - Crie um vetor $\mathbf{x}$ de 100.000.000 de elementos, onde cada elemento é um número real com valor aleatório entre 0 e 1.
  - Para cada elemento x_i do vetor, calcule $f(x_i)$ tal que:
    - $f(x_i) = 2^{-2\left(\frac{x_i - 0,1}{0,9}\right)^2} (\sin(5\pi x_i))^6$
    - Cada *thread* deve ficar responsável por uma faixa determinada do vetor $\mathbf{x}$.
      - Divida igualmente o trabalho entre as *threads* criadas.
  - Ao final, o programa deve exibir:
    - O valor médio de $f(x_i)$ para todos os $x_i \in \mathbf{x}$:
      - $\frac{1}{100.000.000} \sum_{i=1}^{100.000.000} f(x_i)$
    - O tempo de execução em *wall time* (tempo do relógio).
      - Não é necessário contar o tempo de execução da geração do vetor $\mathbf{x}$

- Compare o tempo de execução com diferentes quantidades de *threads*:
  - 1, 2, 4, 6, 8, 10, 12, 16, 24, 32, 64, 128, 256, ...
    - Execute múltiplas vezes com cada quantidade para obter uma média mais confiável.
- Responda as seguintes questões:
  1. Qual o valor médio obtido para $f(x_i)$?
  2. Qual foi o menor tempo de execução obtido?
  3. Com quantas *threads* o tempo de execução foi o menor?
  4. Com quantas *threads* o tempo de execução deixa de diminuir?
  5. Quantos núcleos tem o processador da máquina onde você executou os experimentos? Existe alguma relação entre a quantidade de núcleos do processador e as quantidades de *threads* observadas nas questões anteriores?


fabricio@fabricio-VirtualBox: ~/so2/aula6
fabricio@fabricio-VirtualBox: ~/so2/aula6
fabricio@fabricio-VirtualBox: ~/so2/aula6
fabricio@fabricio-VirtualBox:~/so2/aula6$ ./ex1-2023 12
Thread 7 processou do índice 50000001 ao 58333333, soma local: 1858035.773853
Thread 1 processou do índice 1 ao 8333333, soma local: 1856767.109404
Thread 3 processou do índice 16666668 ao 25000000, soma local: 1858427.791169
Thread 5 processou do índice 33333334 ao 41666667, soma local: 1857635.191384
Thread 4 processou do índice 25000001 ao 33333333, soma local: 1856471.409349
Thread 6 processou do índice 41666668 ao 50000000, soma local: 1858098.754633
Thread 12 processou do índice 91666668 ao 100000000, soma local: 1857711.409544
Thread 2 processou do índice 8333334 ao 16666667, soma local: 1857208.115830
Thread 11 processou do índice 83333334 ao 91666667, soma local: 1857524.894258
Thread 8 processou do índice 58333334 ao 66666667, soma local: 1858653.415270
Thread 9 processou do índice 66666668 ao 75000000, soma local: 1858571.750244
Thread 10 processou do índice 75000001 ao 83333333, soma local: 1857349.993070
Média Global: 0.222925
Tempo Medido: 0.890 segundos.
fabricio@fabricio-VirtualBox:~/so2/aula6$


fabricio@fabricio-VirtualBox: ~/so2/aula6
fabricio@fabricio-VirtualBox:~/so2/aula6$ ./ex1-2023 24
Vetor x de números aleatórios gerado.
Thread 15 processou do índice 58333334 ao 62500000, soma local: 929184.872023
Thread 16 processou do índice 62500001 ao 66666667, soma local: 929221.582855
Thread 2 processou do índice 4166668 ao 8333333, soma local: 928704.294420
Thread 8 processou do índice 29166668 ao 33333333, soma local: 929036.921947
Thread 12 processou do índice 45833334 ao 50000000, soma local: 929556.184914
Thread 1 processou do índice 1 ao 4166667, soma local: 929505.532707
Thread 14 processou do índice 54166668 ao 58333333, soma local: 928728.287510
Thread 7 processou do índice 25000001 ao 29166667, soma local: 928195.867086
Thread 4 processou do índice 12500001 ao 16666667, soma local: 929086.215762
Thread 10 processou do índice 37500001 ao 41666667, soma local: 929447.122844
Thread 5 processou do índice 16666668 ao 20833333, soma local: 928447.656981
Thread 19 processou do índice 75000001 ao 79166667, soma local: 928962.619560
Thread 6 processou do índice 20833334 ao 25000000, soma local: 929620.830223
Thread 21 processou do índice 83333334 ao 87500000, soma local: 930024.942617
Thread 11 processou do índice 41666668 ao 45833333, soma local: 929292.339777
Thread 20 processou do índice 79166668 ao 83333333, soma local: 929330.418960
Thread 3 processou do índice 8333334 ao 12500000, soma local: 928107.127556
Thread 9 processou do índice 33333334 ao 37500000, soma local: 928699.116516
Thread 22 processou do índice 87500001 ao 91666667, soma local: 928342.461390
Thread 13 processou do índice 50000001 ao 54166667, soma local: 929633.258479
Thread 18 processou do índice 70833334 ao 75000000, soma local: 928595.359577
Thread 24 processou do índice 95833334 ao 100000000, soma local: 928608.871871
Thread 17 processou do índice 66666668 ao 70833333, soma local: 928517.792481
Thread 23 processou do índice 91666668 ao 95833333, soma local: 928554.305567
Média Global: 0.222954
Tempo Medido: 0.471 segundos.
fabricio@fabricio-VirtualBox:~/so2/aula6$ S


fabricio@fabricio-VirtualBox: ~/so2/aula6
fabricio@fabricio-VirtualBox:~/so2/aula6$ ./ex1-2023-bench
Threads: 1; Tempos medidos: 4.901; 4.899; 4.932; Média: 4.911 segundos
Threads: 2; Tempos medidos: 2.455; 2.466; 2.451; Média: 2.457 segundos
Threads: 4; Tempos medidos: 1.250; 1.238; 1.241; Média: 1.243 segundos
Threads: 6; Tempos medidos: 0.877; 0.878; 0.872; Média: 0.876 segundos
Threads: 8; Tempos medidos: 0.720; 0.750; 0.754; Média: 0.741 segundos
Threads: 10; Tempos medidos: 0.657; 0.632; 0.620; Média: 0.636 segundos
Threads: 12; Tempos medidos: 0.571; 0.561; 0.560; Média: 0.564 segundos
Threads: 16; Tempos medidos: 0.487; 0.484; 0.483; Média: 0.484 segundos
Threads: 24; Tempos medidos: 0.376; 0.382; 0.378; Média: 0.378 segundos
Threads: 32; Tempos medidos: 0.383; 0.390; 0.397; Média: 0.390 segundos
Threads: 64; Tempos medidos: 0.368; 0.376; 0.372; Média: 0.372 segundos
Threads: 128; Tempos medidos: 0.360; 0.366; 0.363; Média: 0.363 segundos
Threads: 256; Tempos medidos: 0.353; 0.362; 0.357; Média: 0.358 segundos
Threads: 512; Tempos medidos: 0.363; 0.370; 0.410; Média: 0.381 segundos
Threads: 1024; Tempos medidos: 0.390; 0.405; 0.396; Média: 0.397 segundos
Threads: 2048; Tempos medidos: 0.413; 0.394; 0.408; Média: 0.405 segundos
fabricio@fabricio-VirtualBox:~/so2/aula6$ S


fabricio@fabricio-VirtualBox: ~/so2/aula6
fabricio@fabricio-VirtualBox: ~/so2/aula6
fabricio@fabricio-VirtualB
fabricio@fabricio-VirtualBox:~/so2/aula6$ ./ex1-2024-bench
Threads: 1; Média: 4.914 segundos
Threads: 2; Média: 2.450 segundos
Threads: 4; Média: 1.239 segundos
Threads: 6; Média: 0.855 segundos
Threads: 8; Média: 0.711 segundos
Threads: 10; Média: 0.638 segundos
Threads: 12; Média: 0.572 segundos
Threads: 16; Média: 0.488 segundos
Threads: 24; Média: 0.378 segundos
Threads: 32; Média: 0.397 segundos
Threads: 64; Média: 0.374 segundos
Threads: 128; Média: 0.368 segundos
Threads: 256; Média: 0.367 segundos
Threads: 512; Média: 0.372 segundos
Threads: 1024; Média: 0.391 segundos
Threads: 2048; Média: 0.403 segundos
fabricio@fabricio-VirtualBox:~/so2/aula6$ S

Referências Bibliográficas

1. NEMETH, Evi.; SNYDER, Garth; HEIN, Trent R.;. *Manual Completo do Linux: Guia do Administrador*. São Paulo: Pearson Prentice Hall, 2007. Cap. 4.
2. DEITEL, H. M.; DEITEL, P. J.; CHOFFNES, D. R.; *Sistemas Operacionais: terceira edição*. São Paulo: Pearson Prentice Hall, 2005. Cap. 20.
3. MITCHELL, Mark; OLDHAM, Jeffrey; SAMUEL, Alex; *Advanced Linux Programming*. New Riders Publishing: 2001. Cap. 4.
4. TANENBAUM, Andrew S.; *Sistemas Operacionais Modernos*. 3ed. São Paulo: Pearson Prentice Hall, 2010. Cap. 10.