

Hands On Pthreads

Posix Threads



Objetivo

- Entender o funcionamento de Pthreads

PTHREADS

Processo vs. Thread

- Um processo pode ter uma ou mais threads
- Uma thread compartilha recursos com o seu processo pai

Características das Threads

- Podem criar outras threads
- Existe dentro de um processo e usa seus recursos
- Tem seu fluxo de controle independente
- Só tem as propriedades essenciais para a execução
- Compartilha os recursos com outros processos
- Tem um overhead muito menor

Motivação

- ❑ Intercambiar CPU com I/O
- ❑ Dar prioridades diferentes para problemas diferentes
- ❑ Tratar eventos assíncronos
- ❑ Comunicação entre threads é mais simples do que comunicação entre processos
- ❑ Melhor performance
- ❑ Mais fácil modelar alguns problemas

Programação

- Em geral, a programação com threads se dá através da divisão do programa em funções executadas por cada thread.
- Também pode ser feita através da divisão de dados.
- Podem ser usadas em aplicações seriais para simular execução de forma paralela.

Histórico de Pthreads

- Cada fornecedor de hardware implementou suas própria versão
- Versões eram proprietárias
- Versões eram incompatíveis entre si
- Difícil para programadores portarem suas aplicações
- Para possibilitar criar programas com threads portáteis, o IEEE criou o Padrão IEEE POSIX 1003.1c (1995)
 - Suportado pelo UNIX
- Pthreads são um conjunto de bibliotecas para a linguagem C, que podem ser implementadas como uma biblioteca a parte ou parte da própria biblioteca C.
- O padrão define mais de 60 chamadas de função

Principais chamadas

- ❑ `pthread_create` – cria uma nova thread.
- ❑ `pthread_exit` – conclui a chamada de thread.
- ❑ `pthread_join` – espera que uma thread específica seja concluída.
- ❑ `pthread_yield` – libera a cpu para que outra thread seja executada.
- ❑ `pthread_attr_init` – cria e inicializa uma estrutura de atributos da thread
- ❑ `pthread_attr_destroy` – remove uma estrutura de atributos da thread

```
$ man pthreads  
$ man <chamada>
```

Exemplo:

```
$ man pthread_create
```

Cabeçalho:

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

Arquivos para o Laboratório

□ Baixar os arquivos em

https://github.com/josemacedo/sistemas-operacionais/tree/master/3_Lab_Threads

hello.c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 30

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    /*esta funcao imprime o identificador da thread e sai */
    printf("Ola mundo! Eu sou a thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    /* ... */

    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
    /* sleep nao eh necessario, eh apenas para dar tempo de analisar */
    sleep(1);
    if (rc){
        printf("ERRO; o codigo retornado de pthread_create() eh %d\n", rc);
        exit(-1);
    }
}

/* Ultima coisa que o main() tem que fazer */
pthread_exit(NULL);
}
```

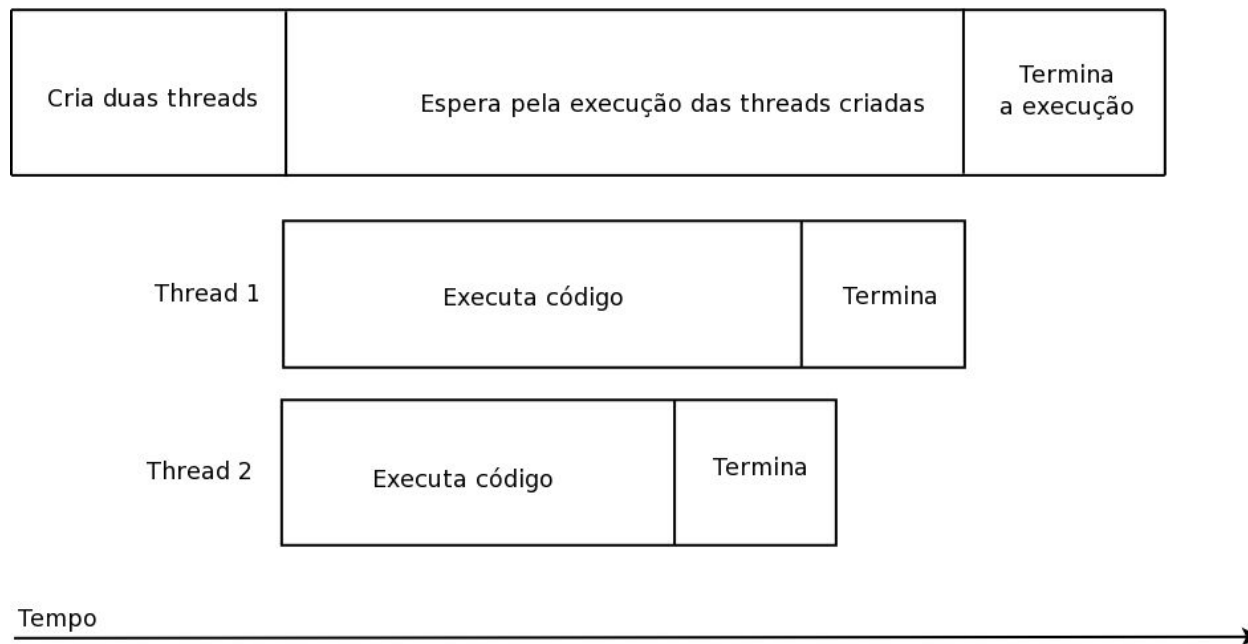
```
$ gcc -o hello hello.c -pthread
$ ./hello
```

Veja o funcionamento do comando sleep(1),
desabilite esse comando, habilite e aumente o
numero entre parenteses

Execução paralela

- O próximo código mostra duas threads executando ao mesmo tempo, realizando cada uma seu trabalho.

paralela.c



paralela.c

```
int main()
{
    pthread_t tid[2];
    thread_arg a[2];
    int i = 0;
    int n_threads = 2;

    /* Cria as threads */
    for(i=0; i<n_threads; i++)
    {
        a[i].id = i;
        pthread_create(&(tid[i]), NULL, thread, (void *)&(a[i]));
    }

    /* Espera que as threads terminem */
    for(i=0; i<n_threads; i++)
    {
        pthread_join(tid[i], NULL);
    }

    pthread_exit((void *)NULL);
}

void *thread(void *vargp)
{
    int i = 0;
    thread_arg *a = (thread_arg *) vargp;

    printf("Começou a thread %d\n", a->id);
    /* Faz um trabalho qualquer */
    for(i = 0; i < 1000000; i++);
    printf("Terminou a thread %d\n", a->id);

    pthread_exit((void *)NULL);
}
```

```
$ gcc -o paralela paralela.c
-pthread
$ ./paralela
```

O exemplo começa com a thread principal, que cria duas outras threads e espera que elas terminem seu trabalho.

Cada uma das threads realiza um trabalho, ao mesmo tempo, e elas terminam o trabalho aproximadamente no mesmo tempo.

Depois, elas retornam, e a thread principal termina.

Exercicio

- ❑ Modifique o código anterior, compile e execute varias vezes, perceba o comportamento das threads
- ❑ Aumente o numero de threads para 10

ANTIGO

```
.....  
  
void *thread(void *vargp)  
{  
    int i = 0;  
    thread_arg *a = (thread_arg *) vargp;  
  
    printf("Comecou a thread %d\n", a->id);  
    /* Faz um trabalho qualquer */  
    for(i = 0; i < 1000000; i++);  
    printf("Terminou a thread %d\n", a->id);  
  
    pthread_exit((void *)NULL);  
}
```

NOVO

```
.....  
  
void *thread(void *vargp)  
{  
    int i = 0;  
    thread_arg *a = (thread_arg *) vargp;  
  
    printf("Comecou a thread %d\n", a->id);  
    /* Faz um trabalho qualquer */  
    int r = rand() % 1000000;  
    for(i = 0; i < r; i++);  
    printf("Terminou a thread %d\n", a->id);  
  
    pthread_exit((void *)NULL);  
}
```

**O que acontece quando
threads compartilham
dados?**



Edite o programa global.c

}

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

typedef struct
{
    int id;
} thread_arg;

void *thread(void *vargp);

int var;

int main()
{
    int n_threads = 50;
    pthread_t tid[n_threads];
    thread_arg a[n_threads];
    int i = 0;

    var = 0;

    //Cria as threads
    for(i=0; i<n_threads; i++)
    {
        a[i].id = i;
        pthread_create(&(tid[i]), NULL, thread, (void *)&(a[i]));
    }

    // Espera que as threads terminem
    for(i=0; i<n_threads; i++)
    {
        pthread_join(tid[i], NULL);
    }

    printf("Valor de var no fim do programa: %d\n", var);

    pthread_exit((void *)NULL);
}
```

```
void *thread(void *vargp)
{
    // Converte a estrutura recebida
    thread_arg *a = (thread_arg *) vargp;

    // Como vamos acessar uma variavel
    printf("Thread %d: valor de var antes da conta: %d\n", a->id+1,
    var);
    int j = 0;
    for(j=0; j<1000; j++)
        var = var + 1;
    printf("Thread %d: valor de var depois da conta: %d\n",
    a->id+1, var);

    pthread_exit((void *)NULL);
}
```


Compile e execute o código anterior

```
$ gcc -o share share.c -pthread
$ ./share
$ ** execute varias vezes e veja o resultado final que será
impresso **
```

- O valor de VAR no fim do programa deve ser 50.000
- O que aconteceu ? Por que ?

Lock e Unlock (mutex)

- ❑ No próximo código duas threads serão criadas, usando a mesma função.
- ❑ No entanto, certa linha dessa função será protegida com o uso de um mutex, já que ela altera o valor de uma variável global (variáveis globais não devem ser usadas, isso é apenas um exemplo!).
- ❑ Essa é uma das técnicas normalmente utilizadas para se proteger zonas críticas do código.
- ❑ Apesar de o uso de variáveis globais ser desaconselhado, normalmente os mutex são declarados globalmente, pois eles devem ser visíveis a todas as threads.

mutex.c

mutex.c

```
typedef struct {
    int id; } thread_arg;

void *thread(void *vargp);
pthread_mutex_t mutex;
int var;
int main()
{
    pthread_t tid[2];
    thread_arg a[2];
    int i = 0;
    int n_threads = 2;
    var = 0;
    /* Cria o mutex */
    pthread_mutex_init(&mutex, NULL);

    /* Cria as threads */
    for(i=0; i<n_threads; i++){
        a[i].id = i;
        pthread_create(&(tid[i]), NULL, thread, (void *)&(a[i]));
    }

    /* Espera que as threads terminem */
    for(i=0; i<n_threads; i++) {
        pthread_join(tid[i], NULL);
    }

    /* Destroi o mutex */
    pthread_mutex_destroy(&mutex);
    pthread_exit((void *)NULL);
}

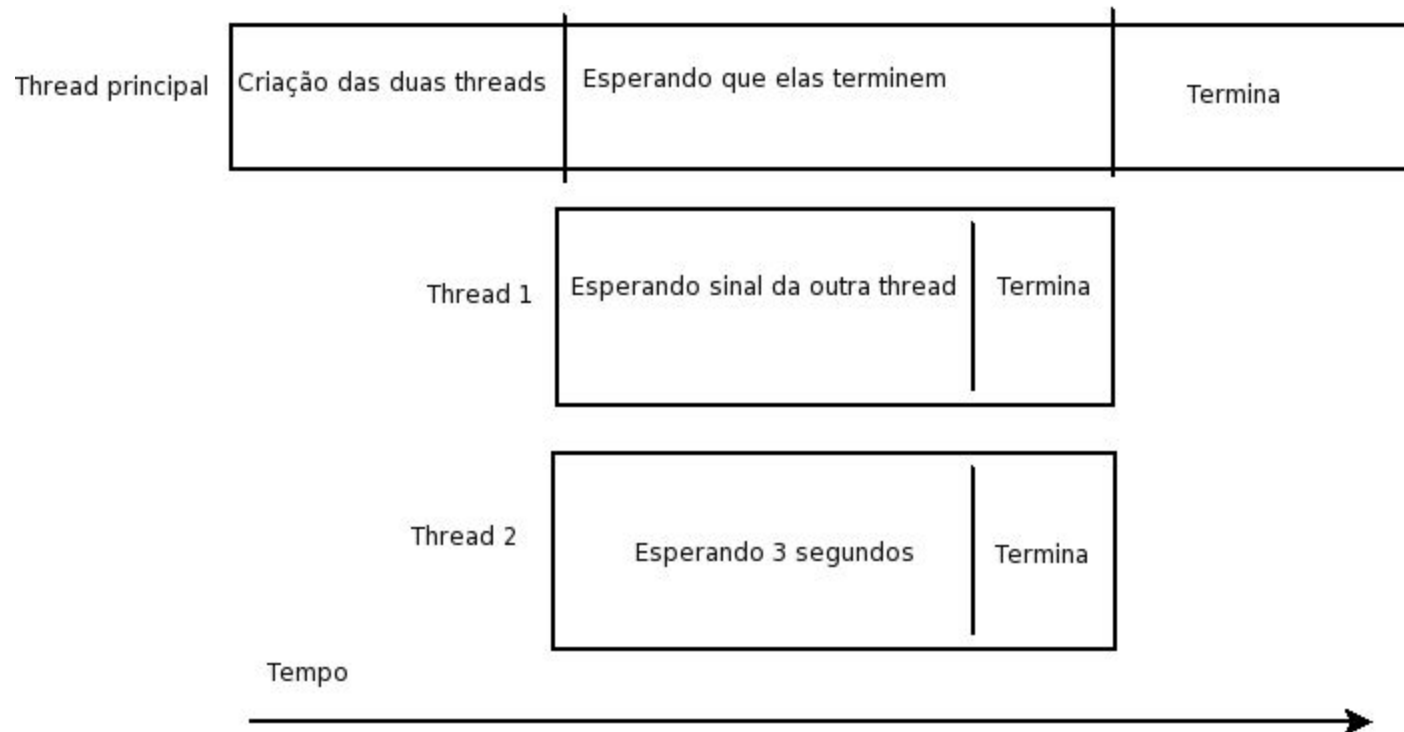
void *thread(void *vargp){
    /* Converte a estrutura recebida */
    thread_arg *a = (thread_arg *) vargp;

    /* Como vamos acessar uma variavel global, deve-se protege-la com uma */fechadura
    pthread_mutex_lock(&mutex);
    printf("Thread %d: valor de var antes da conta: %d\n", a->id+1, var);
    var = var + a->id + 1;
    printf("Thread %d: valor de var depois da conta: %d\n", a->id+1, var);
    pthread_mutex_unlock(&mutex);
    pthread_exit((void *)NULL);
}
```

```
$ gcc -o mutex mutex.c -pthread
$ ./mutex
```

O exemplo começa com a thread principal, que cria outras duas, e espera que elas terminem. Qual das duas threads chegam primeiro ao mutex é indeterminado, mas a que chegar trava o mutex, modifica var, e libera o mutex para que a outra faça o mesmo. Então, ambas terminam, e depois a principal também.

O que acontece no código (mutex.c)?



Fim do Laboratório de Pthreads

