Insper

24 - POSIX Threads I

Igor Montagner

Parte 1 - criando tarefas e esperando elas acabarem

O exemplo abaixo cria uma thread que roda a função `primeira_thread`, espera por seu fim e mostra a mensagem *Fim do programa*.

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

void *minha_thread(void *arg) {
    printf("Hello thread!\n");
}


int main() {
    pthread_t tid;

    int error = pthread_create(&tid, NULL, minha_thread, NULL);

    printf("Hello main\n");

    pthread_join(tid, NULL);

    return 0;
}
```

Ele deve ser compilado com a flag especial `-pthread`:

```
> gcc exemplo1.c -o exemplo1 -pthread
```

Vamos dissecar a chamada da função `pthread_create`:

```c
int error = pthread_create(
                           &tid, // variável para guardar ID da nova thread
                           NULL, // opções de criação. NULL = opções padrão
                           minha_thread, // função a ser executada
                           NULL); // parâmetro passado para a função acima
);
```

Toda thread que rodarmos terá a seguinte assinatura (mudando, é claro, o nome da função).

```c
void *minha_thread(void *arg);
```

Uma variável do tipo `void *` representa um endereço de memória cujo conteúdo é desconhecido. Ou seja, ele diz somente onde encontrar os dados, mas não diz o que está guardado na memória naquele lugar. Este tipo de variável é usada quando queremos passar blocos de memória entre funções mas não queremos fixar um tipo de dados. Veremos com mais detalhes como isto funciona na parte 2.

Exercício: O manual contém entradas muito bem escritas de todas as chamadas de POSIX threads que usaremos. Abra as seguintes e se familiarize com seu conteúdo.

```
> man 7 pthreads
> man 3 pthread_create
> man 3 pthread_join
```

Assim como processos, threads são escalonadas pelo kernel. Isto significa que não controlamos a ordem em que elas rodam no nosso programa. Ou seja, ao executar `pthread_create` não sabemos se a thread principal (aquela que roda o `main`) continuará rodando ou se o controle passará instantaneamente para a nova thread. A primitiva de **sincronização** mais simples que dispomos é `pthread_join`, que garante que uma thread só prossegue quando outra acabar.

Exercício: Retire o `pthread_join` do programa exemplo e o execute. O quê acontece?

Exercício: É possível que duas threads chamem `pthread_join` na mesma thread destino? Consulte o manual para saber esta resposta.

A resposta acima indica que precisaremos de outras primitivas de **sincronização** mais sofisticadas no futuro. Veremos isso nas próximas aulas.

Exercício: Crie quatro threads, cada uma executando uma função que faz um print diferente. Compile e execute seu programa várias vezes. A saída muda?

Parte 2 - passando argumentos para threads

Nossas threads ainda são muito limitadas: elas não recebem nenhum argumento nem devolvem resultados. Vamos consertar isso nesta seção.

Vimos na parte 1 que o último argumento de `pthread_create` é um ponteiro `void` para os dados que nossa função deverá receber. Neste sequência de exercícios iremos aprender a usar este argumento para passar dados para nossas threads.

Exercício: Nosso primeiro exercício será feito passo a passo. Siga cada um dos passos a risca e depois responda as questões. Vamos trabalhar a partir de um arquivo vazio.

1. Crie um programa simples com uma função `main` que aloca (usando malloc) um vetor `vi` com 4 `int`s e um vetor `tids` com 4 `pthread_t`s.
2. Adicione ao seu programa um `for` que cria 4 threads (colocando seus ids no vetor `tids`). Passe como último argumento o endereço do elemento correspondente de `vi`.
3. Espere pelo fim desta thread.
4. Crie uma função `void *tarefa_print_i(void *arg)` que declara uma variável `int *i` e dá print em seu conteúdo. Inicialize a variável `i` como mostrado abaixo:

   int i = *(int*) arg;

Se seu programa estiver correto você deverá ver no terminal 4 prints com números de 0 a 3, cada um vindo de um thread.

> **!** Se tiver problemas valide seu código com algum colega que já tenha sido validado pelo professor. Se não tiver ninguém por perto já validado me chame ;)

Exercício: Explique como é feita a passagem do argumento para a thread.

Exercício: Passamos para a thread um valor alocado dinamicamente. Por que isso é necessário?

Vamos explorar a resposta da pergunta acima nos próximos exercícios. Para cada exercício, encontre seu problema, descreva-o usando suas próprias palavras e mostre um exemplo de saída possível. Somente depois de escrever sua resposta rode o programa.

> **!** Cada exercício foca em um problema diferente. A resposta não é a mesma para ambas.

Exercício: Código 1 (arquivo *parte2-1.c*)

```c
void *minha_thread(void *arg) {
    int *i = (int *) arg;
    printf("Hello thread! %d\n", *i);
}

// dentro do main

for (int i = 0; i < 4; i++) {
    pthread_create(&tid[i], NULL, minha_thread, &i);
}
```

Exercício: Código 2 (arquivo *parte2-2.c*)

```c
void *minha_thread(void *arg) {
    int *i = (int *) arg;
    printf("Hello thread! %d\n", *i);
}

pthread_t *criar_threads(int n) {
    pthread_t *tids = malloc(sizeof(pthread_t) * n);

    for (int i = 0; i < 4; i++) {
        pthread_create(&tids[i], NULL, minha_thread, &i);
    }

    return tids;
}

// dentro do main
    pthread_t *tids = criar_threads(4);
```

> **!** Valide sua solução com o professor.

Agora que já entendemos como passar um argumento e que devemos sempre colocá-lo no *heap*, passar vários é muito simples: alocamos um `struct` com todos os dados que queremos enviar e passamos seu endereço no último argumento. Ao recebê-lo, a função faz um *cast* de `void *` para um ponteiro para o `struct`.

Exercício: Modifique seu exercício do começo desta parte para receber dois argumentos do tipo inteiro e imprimir ambos valores.

Parte 3 - retornando valores

Na prática, ao passar `struct`s para threads como argumentos já sabemos como retornar valores: basta adicionar um campo que própria thread deve preencher com o resultado de sua execução. Isso é equivalente a criar uma função que

Exercício: Modifique seu exercício da parte anterior para que as threads retornem a multiplicação dos dois inteiros passados. Faça o print deste valor no `main`.

Parte 4 - acelerando programas

Esta parte final junta todos os exercícios anteriores. Vamos acelerar um programa simples que faz a soma de todos os valores de um vetor. Seu trabalho será

1. Criar um programa que recebe um valor n e depois lê n números fracionários da entrada.
2. Criar duas threads, cada uma fazendo a soma de uma metade do vetor.
   - Como você faria para usar uma só função para ambas threads?
   - Que argumentos essa função receberia?
3. Esperar o fim das duas threads criadas e realizar a soma final.
4. Mostrar essa soma no terminal.

Não esqueça de checar se seu programa realmente mostra o valor certo.