Threads, mecanismos

Este tutorial complementa a discussão teórica sobre programação concorrente vista em sala de aula. Vimos que uma thread define um fluxo de execução de instruções em um processo. Processos com múltiplos threads têm portanto múltiplos fluxos de execução.

Threads de um mesmo processo compartilham a memória do processo. Por sua vez, alguns recursos são criados para uso exclusivo das threads, a pilha de execução, por exemplo. Assim, a imagem de uma thread em memória é menor do que um processo. Ainda, a troca de contexto entre threads pode não envolver o sistema operacional (diferente de troca de contexto entre processos). Por esses motivos, é "mais leve" construir sistemas concorrentes com múltiplas threads ao invés de múltiplos processos.

Neste tutorial, mostraremos mecanismos de algumas linguagens de programação (Java, Python e C) que permitem criar threads, definir o código que as threads executarão bem como coordenar e sincronizar a execução de threads.

Java

A plataforma Java tem grande suporte para threads desde suas versões iniciais. Neste tutorial, discutiremos mecanismos mais recentes (lançados na versão 21 da plataforma), incluindo tanto threads *virtuais* (ou *green threads*) e threads de plataforma. A diferença fundamental entre essas duas implementações é que cada thread de plataforma corresponde à uma thread do sistema operacional (*kernel thread*, a unidade básica de escalonamento do sistema operacional) enquanto que várias threads virtuais podem ser mapeadas para a mesma thread do sistema operacional.

Considere um cenário em que as threads de um programa Java bloqueiam com frequência, realizando, por exemplo, operações de entrada e saída. Neste cenário, usar *threads* virtuais é uma boa opção pois permite que a *runtime* Java escalone uma outra *thread* para executar no lugar da que está bloqueada sem a necessidade de chamar o escalonador do sistema operacional; ou seja, sem a necessidade da custo adicional de realizar uma *system call*.

O mecanismo de gerenciamento das threads, seja de plataforma ou virtuais, tem três etapas iniciais: 1) a definição do código a ser executado pela thread; 2) a criação da thread; e, por fim, 3) a inicialização da thread. Estes mecanismos, na maior parte das vezes, são os mesmos para ambos os tipos de threads. Ainda, boa parte das diferenças, principalmente na criação das threads, são simplificadas pela abstração ThreadBuilder. Como sempre, a documentação da plataforma detalha estes e outros aspectos.

Definição do código a ser executado

Em Java, usamos o tipo Runnable para definir o fluxo que será executado por uma Thread. Em outras palavras, escrevemos o código a ser executado pela Thread através do tipo Runnable. Este tipo define um único método, run, que pode executar código arbitrário definido pelo programador:

```
void   run()
```

A thread, ao ser iniciada pelo escalonador, chamará o método run. Este método está limitado a não receber parâmetros e não ter retorno e portanto outras estratégias precisam ser usadas para compartilhar memória entre threads, por exemplo, através de referências passadas por construtores e outros métodos.

De maneira concreta, é possível definir o comportamento de uma thread tanto criando uma classe que implementa a interface definida por Runnable quanto, por herança, criando uma subclasse de Thread. Abaixo, temos duas implementações, uma para cada modo, em que as threads criadas escrevem na saída padrão a mesma mensagem indefinidamente. Na maior parte dos casos, usa-se a primeiro modo (uma numa implementação da interface Runnable) a menos que se queira mudar o comportamento dos métodos implementados na classe Thread (o que é bastante inusual).

```
import java.util.concurrent.*;
```

```java
public class HelloRunnable implements Runnable {

    public static void main(String[] args) {
        HelloRunnable helloRunnable = new HelloRunnable();
        helloRunnable.run();

        HelloThread helloThread = new HelloThread();
        helloThread.start();
    }

    @Override
    public void run() {
        while (true) {
            System.out.println("Hello world!");
        }
    }

    static class HelloThread extends Thread {
        @Override
        public void run() {
            while (true) {
                System.out.println("Hello world!");
            }
        }
    }
}
```

Criação da Thread (de plataforma)

Uma vez implementado o método **run** para definir o comportamento que a thread executará, a thread precisa ser de fato criada. Isso pode ser feito de várias maneiras. Por exemplo, criando um objeto Thread e, por composição, passando uma instância que implementa *Runnable:*

```java
Thread myThread = new Thread(new HelloRunnable(), "myThread-HelloRunnable");
```

Ou criando uma subclasse de thread:

```java
Thread myThread = new HelloThread("myThread-HelloThread");
```

Veja que usei um argumento String para o construtor. Esse parâmetro define o nome para thread criada e pode ser acessado através do método getName. Atribuir nomes únicos para thread é uma excelente prática que ajuda bastante a depurar defeitos em sistemas concorrentes.

Inicialização da thread (de plataforma)

Nos passos anteriores, criamos uma thread e atribuímos a ela um código a ser executado. Ainda é necessário um passo adicional, a inicialização. Para isso, chamamos o método **start** da thread. Isso permite que a thread criada esteja pronta para executar. Eventualmente, o escalonador pode escolher a thread para ser executada.

```
myThread.start();
```

Como dissemos, as threads de um programa são, cada uma, um fluxo de execução distinto dos demais. Em muitos casos, é preciso que as threads sincronizem sua execução. Decidir como sincronizar threads para termos sistemas corretos é algo decidido caso a caso. Aqui, mostraremos somente um mecanismo que permite que uma thread espere que outra termine a sua execução, por exemplo para obter um resultado da computação da thread que terminou. Isso, pode ser feito através do método **join:**

```
myThread.join();
```

Neste caso, a thread que executa a linha acima irá bloquear ao chamar o método **join** até que a thread **myThread** termine sua execução.

Threads virtuais

Os mecanismos acima, permitem criar e executar **threads de plataforma**. Para manipular **threads virtuais**, usamos ThreadBuilder.

Usando este Builder, podemos criar uma thread virtual através das seguintes chamadas:

```
Thread thread = Thread.ofVirtual().unstarted(runnable);
```

e, em seguida:

```
thread.start();
```

Note que o objeto runnable passado como argumento é uma implementação de Runnable tal como fizemos para threads de plataforma.

Ainda, é possível iniciar a thread na mesma cadeia de criação:

```
Thread thread = Thread.ofVirtual().start(runnable);
```

Semáforos

Em java, usamos o tipo java.util.concurrent.Semaphore para manipular Semáforos. Um objeto do tipo Semaphore é criado indicando o valor inicial do semáforo:

new Semaphore(initValue);

Os métodos abaixo, implementam a semântica de wait e signal:

public void acquire()
public void release()

Em particular, o valor do semáforo em java atinge no mínimo zero.

Exemplo

```
import java.util.concurrent.*;

public class FindMinInArray {
   public static void main(String[] args) throws InterruptedException {
       int[] array = {12, 3, 19, 8, 7, 25, 5, 17, 6, 9, 14, 2, 1, 20};

       Task task = new Task(array);
       Thread myThread = new Thread(task, "myThread-Task");
       myThread.start();
       myThread.join();

       int min = task.getMinValue();
       System.out.println("O menor número no vetor é: " + min);
   }

   public static class Task implements Runnable {
       private final int[] array;
       private int minValue;

       public Task(int[] array) {
           this.array = array;
           this.minValue = Integer.MAX_VALUE;
       }

       @Override
       public void run() {
```

```
            for (int num : array) {
                if (num < minValue) {
                    m'inValue = num;
                }
            }
        }

        public int getMinValue() {
            return minValue;
        }
    }
}
```

Explicação dos Componentes

- `public class Task implements Runnable`: Define uma classe chamada `Task` que implementa a interface `Runnable`. Implementar `Runnable` significa que a classe `Task` pode ser executada por uma thread.

- `@Override`: É uma anotação que indica que o método `run()` está sobrescrevendo o método da interface `Runnable`.

- `public void run()`: Este é o método que será executado quando a thread iniciar. Dentro deste método, você define o que a thread deve fazer. Neste caso, imprime "Executando a tarefa." no console.

- `public static void main(String[] args)`: É o ponto de entrada do programa. Este método é chamado pela JVM quando o programa é executado.

- `throws InterruptedException`: Declara que o método `main` pode lançar uma `InterruptedException`. Isso é necessário porque o método `join()` pode lançar essa exceção.

- **`new Thread(new Task(), "my-task")`**: Cria uma nova instância de `Thread`. O construtor `Thread` recebe dois argumentos:
    - `new Task()`: Passa uma instância de `Task` que implementa `Runnable`, definindo o código que a thread deve executar.

    - `"my-task"`: Nomeia a thread como "my-task", o que pode ser útil para depuração e monitoramento.

- **`myThread.start()`**: Inicia a execução da thread. A JVM cria um novo fluxo de execução e chama o método `run()` da instância `Task`. A partir deste ponto, a thread executa seu código de forma independente.

- **`myThread.join()`**: Este método faz com que a thread principal (a que está executando o método `main`) espere até que a thread `myThread` termine sua execução. Sem isso, a thread principal poderia terminar antes da thread `myThread`, encerrando o programa antes que `myThread` complete sua execução.

Links úteis:

https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html
https://docs.oracle.com/javase/tutorial/essential/concurrency/join.html
https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Python

Criação de Threads em Python

O estilo adotado para uso de threads é python é bastante parecido com java (embora mais simples, como de costume).

Uma diferença marcante é que não é necessário criar um tipo para definir o comportamento da thread. Para isso, basta implementar uma função regular (thread_function abaixo).

Em seguida, a thread é criada usando o construtor threading.Thread (https://docs.python.org/3/library/threading.html#threading.Thread) e por fim, a thread é inicializada com a função start

```
import threading
import time

def thread_function():
    time.sleep(2)

if __name__ == "__main__":
    x = threading.Thread(target=thread_function)
    x.start()
```

Passando Argumentos para Funções de Thread

Se a função que a thread vai executar precisar de argumentos, você pode passar esses argumentos utilizando o parâmetro args ao criar a thread:

```
import threading
import time

def thread_function(arg0, arg1):
    print(arg0, arg1)
    time.sleep(2)

if __name__ == "__main__":
    x = threading.Thread(target=thread_function, args=(1,2,))
    x.start()
```

Esperando a Thread Terminar

Por fim, a thread pode demorar mais para finalizar do que a thread mãe. Para isso, utiliza-se o método join():

```
import threading
import time

def thread_function():
    time.sleep(2)
    print("Thread finished")

if __name__ == "__main__":
    x = threading.Thread(target=thread_function)
    x.start()
    x.join()  # Aguarda a conclusão da thread
    print("Main thread continues")
```

Explicação dos Componentes

- `import threading`: Importa o módulo threading, que fornece ferramentas para trabalhar com threads.
- `def thread_function():`: Define a função que a thread executará. Esta função pode realizar qualquer tarefa necessária.
- `threading.Thread(target=thread_function)`: Cria uma nova thread, especificando a função `thread_function` como a tarefa a ser executada pela thread.
- `x.start()`: Inicia a execução da thread.
- `x.join()`: Bloqueia a thread principal até que a thread `x` termine a execução.

Links úteis:

https://docs.python.org/3/library/threading.html#

Semáforos em Python

O semáforo é uma abstração básica para sincronização de threads no modelo de concorrência por compartilhamento de memória. A seguir, mostramos como usar semáforos em python. É importante ter em mente que a API da linguagem python **é ligeiramente diferente** da abstração discutida em sala.

Semáforo Explícitos
É python, o tipo semáforo mantém um contador que é **decrementado através da operação acquire**. Por sua vez, é incrementado **através da operação release**.

Quando o semáforo estiver com valor zero, qualquer thread que chamar **acquire** será bloqueada. Uma thread bloqueada no semáforo será desbloqueada por outra thread que chame **release** no semáforo.

```
import threading
import time
import random

semaphore = threading.Semaphore(2)

def thread_function(name):
    semaphore.acquire()
    try:
        print(f"{name} acquired the semaphore")
        time.sleep(random.randint(1, 10))
    finally:
        print(f"{name} releasing the semaphore")
        semaphore.release()

if __name__ == "__main__":
    threads = []
    for i in range(6):
        thread = threading.Thread(target=thread_function,
args=(f"Thread-{i}",))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()
```

Semáforos com with

Além do modo anterior, no qual era necessário chamar **release** e **acquire** explicitamente, é possível também utilizar um semáforo com a declaração **with** (context manager). Isso simplifica o código e garante que o semáforo será liberado corretamente, mesmo que ocorra uma exceção dentro do bloco **with**.

```
import threading
import time
import random

semaphore = threading.Semaphore(2)

def thread_function(name):
    with semaphore:
        print(f"{name} acquired the semaphore")
        time.sleep(random.randint(1, 10))
        print(f"{name} releasing the semaphore")

if __name__ == "__main__":
    threads = []
    for i in range(6):
        thread = threading.Thread(target=thread_function,
args=(f"Thread-{i}",))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()
```

Barreira

A classe Barrier do módulo threading é usada para sincronizar um ponto em que todas as threads devem esperar umas pelas outras para continuar. Da mesma forma que vimos em sala, uma barreira em python é construída com um tamanho (chamado de **parties** na definição de python) e tem um método **wait** para sinalizar a chegada no ponto da barreira.

```python
import threading
import time
import random

barrier = threading.Barrier(6)

def thread_function(name):
    print(f"{name} is waiting at the barrier")
    time.sleep(random.randint(1, 10))
    barrier.wait()
    print(f"{name} passed the barrier")

if __name__ == "__main__":
    threads = []
    for i in range(6):
        thread = threading.Thread(target=thread_function,
args=(f"Thread-{i}",))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()
```

C

A linguagem C oferece suporte à programação com threads através da biblioteca POSIX threads (pthreads).

Como compilar um programa em C com suporte a threads?
Use o seguinte comando, incluindo a flag `-pthread`:
`gcc -o file file.c -pthread`

E execute com:
`./file`

Criação de threads

Em C, o processo de criação de threads precisa ser muito bem estruturado, para garantir que você tenha controle sobre o funcionamento do seu código. Assim, antes de você criar a sua thread com a função específica de criação de thread e passar os argumentos para que ela comece a executar será necessário declarar uma variável que será responsável por armazenar o identificador da sua thread.

"Como isso? Ainda não entendi…" Antes de pensar em threads pense na simples operação de `int valor = 2;`, essa operação pode ser dividida em duas etapas: a primeira, `int valor;`, aloca um espaço na memória para um inteiro; e a segunda, `valor = 2;`, armazena o valor 2 nessa variável. Com as threads em C o comportamento é o mesmo, antes de criar a sua thread você precisa reservar espaço na memória que será responsável por armazenar o identificador da sua thread.

"Mas por que eu preciso armazenar o identificador da minha thread?"
Para que você consiga utilizar a próxima função que será descrita e também tenha o controle (ainda que parcial) do fluxo de execução dessa thread.

Pronto, com isso em mente é necessário que você saiba duas outras coisas antes de criar a thread (acalme-se, você já já irá conseguir):

1. "O que é essa estrelinha *?" Cuidado com a nomenclatura: o nome correto é asterisco. Ele é um operador usado na declaração de uma variável para indicar que ela é um **ponteiro** e também na desreferenciação de ponteiros, ou seja, para acessar o valor guardado no endereço apontado.
2. "E o que é esse `&`?" Olha a nomenclatura novamente. O símbolo &, chamado de **E comercial**, é um operador que retorna o **endereço de uma variável**. Ele é muito usado para passar variáveis por referência a funções que esperam ponteiros.
3. Exemplo de como funciona:

```c
int x = 5;
int* ptr; // declaro um ponteiro
ptr = &x; // ponteiro tem o endereço de x

printf("endereço ponteiro: %p\n", ptr);
```

```c
    // acessar o que está dentro da memória com ASTERISCO
    printf("valor do ponteiro: %d\n", *ptr);
```

Output:

endereço ponteiro: 0x7ffc24475c7c
valor do ponteiro: 5

Exemplo bônus, se eu fizer isso aqui após o último print, qual será o output? Teste na sua máquina:

```c
    x = 10;
    printf("valor do ponteiro: %d\n", *ptr);
```

Note que eu estou mudando o valor de X e vou imprimir o valor acessado com *ptr.

Mais uma coisa, você pode usar o asterisco com `int* ptr` ou `int * ptr` ou `int *ptr`. Tudo irá produzir a mesma saída.

Compartilhamento de memória entre threads

Em C, todas as threads de um mesmo processo compartilham o mesmo espaço de memória. Isso significa que variáveis globais e ponteiros acessados por diferentes threads se referem aos mesmos dados. Essa característica facilita a comunicação entre threads, mas também exige cuidado. A falta de sincronização pode gerar condições de corrida, por isso é importante utilizar mecanismos como `pthread_mutex_t` ou até mesmo semáforos (sem_t) sempre que houver acesso concorrente a variáveis.

Definindo o comportamento da thread

Em C, o comportamento de uma thread é definido por uma função com a seguinte assinatura:

void* nome_da_funcao(void* arg);

Essa função precisa receber um ponteiro genérico como argumento (void*) e retornar um ponteiro genérico. No exemplo a seguir, usaremos uma função chamada foo que apenas imprime um valor inteiro recebido.

Ufa! Após todo esse background, você está preparado para criar a sua primeira thread em C. Para isso você irá utilizar a função `pthread_create()`, essa função recebe 4 argumentos:

1. Você precisa passar o endereço da variável que armazenará o ID da nova thread, como você é esperto, já declarou isso com `pthread_t thread_id` e irá passar no argumento da função utilizando o **E comercial** para referenciar o endereço da variável que vai armazenar o id da thread quando criada. **(Um ponteiro para a variável que armazenará o ID da nova thread (ex: &thread_id).)**
2. São atributos da thread (stack size, prioridade, etc), se quiser personalizar pode cutucar aqui, caso contrário, deixe como NULL que irá ter o comportamento padrão. **(Os atributos da thread (como tamanho da pilha e prioridade). Use NULL para os padrões.)**

3. É a função que a thread vai executar. Ela precisa receber um ponteiro genérico (void *) e retornar um ponteiro genérico. Assim, a função que você passar precisa ter a seguinte assinatura:
    a. `void* minha_funcao(void* arg)`. **(A função que a thread irá executar. Ela deve ter a assinatura void* func(void*).)**
4. Aqui é onde geralmente gera a dúvida: se sua função precisa receber algum dado (como um número, uma struct, etc.), você passa o endereço desse dado, mas convertido para um void*.
    a. "E o que a thread vai fazer com esse argumento convertido para void*?" Na função você vai recuperar o valor, então ao passar um inteiro como quarto argumento na criação de sua thread, você recupera o valor fazendo um cast de volta para o tipo original: 'int* val = (int*) arg;`. **(Um argumento para essa função, passado como void* (por exemplo, (void*)&valor).)**

"Pronto, criei minha função e minha thread está executando. Posso seguir com minha vida?" Será mesmo? Ela está executando corretamente? Como você pode ter certeza?

se o retorno for igual a 0, parabéns: a thread foi criada corretamente e será executada. Caso contrário, a função retorna um código de erro (um número positivo) que indica que a thread não foi criada.

Por fim, a parte mais simples, a biblioteca pthread também irá fornecer a mesma função que conseguimos ver em linguagens como Python para esperar a thread morrer. Essa função é a `pthread_join()` e deve ser usada para garantir que a thread principal aguarde o término das threads que ela criou. Ela recebe dois argumentos:

1. O ID da thread que você deseja aguardar (o mesmo que foi obtido em `pthread_create`).

2. Um ponteiro para ponteiro (`void**`) onde será armazenado o valor retornado pela função da thread (caso deseje recuperá-lo). Se não quiser capturar esse retorno, passe `NULL`. Isso pode ser útil se você precisar saber o que foi computado pela função chamada, não se assuste, o código para isso é meio estranho, vou fornecer um pequeno exemplo na função `foo2` do código abaixo.

Usar `pthread_join` é uma boa prática porque evita que a thread principal finalize antes que as threads filhas terminem, o que poderia encerrar o programa prematuramente.

O exemplo abaixo ele traz à tona tudo que foi discutido na seção anterior, com isso você já está apto para brincar um pouco mais com as threads em C.

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

void *foo(void *arg)
```

```c
{
    int *valor = (int *)arg;
    printf("Valor recebido: %d\n", *valor);
    return NULL;
}

void *foo2(void *arg)
{
    int *valor = (int *)arg;
    // aloca memória para retornar algo
    int *resultado = malloc(sizeof(int));
    *resultado = (*valor) * 2;
    // função para comunicar que esse é o valor de retorno da função
    pthread_exit((void *)resultado);
}

int main() {
    pthread_t thread_id;
    int valor = 42;

    if (pthread_create(&thread_id, NULL, foo, (void *)&valor) != 0) {
        printf("Erro ao criar a thread\n");
        return 1;
    }
    pthread_join(thread_id, NULL);
    printf("Thread terminou.\n");

    if (pthread_create(&thread_id, NULL, foo2, (void*)&valor) != 0) {
        printf("Erro ao criar a thread\n");
        return 1;
    }
    int *retorno;
    pthread_join(thread_id, (void **)&retorno);
    printf("Thread retornou: %d\n", *retorno);
    // libera a memória alocada
    free(retorno);

    return 0;
}
```

Exemplo com múltiplas threads e mutex:

Neste exemplo, várias threads acessam a variável global contador. Para evitar condições de corrida, usamos pthread_mutex_t, que garante que apenas uma thread por vez execute a região crítica – o trecho de código que modifica dados compartilhados:

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

#define NUM_THREADS 4

int contador = 0;
pthread_mutex_t lock;

void* incrementar(void* arg) {
    pthread_mutex_lock(&lock);
    contador++;
    printf("Thread %ld incrementou. Contador: %d\n", (long)arg,
contador);
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    pthread_mutex_init(&lock, NULL);

    for (long i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, incrementar, (void*)i);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&lock);
    printf("Valor final do contador: %d\n", contador);
    return 0;
}
```

O uso correto de mutex evita que duas threads incrementem contador simultaneamente, garantindo integridade no valor final.

Além de mutex, a biblioteca POSIX também oferece suporte a **semáforos**, definidos na biblioteca semaphore.h. Semáforos (sem_t) permitem controlar o número de threads que podem acessar uma região crítica simultaneamente:

```c
#include <semaphore.h>



sem_t sem;

sem_init(&sem, 0, 1);         // inicializa com valor 1 (binário)

sem_wait(&sem);               // equivalente ao lock

// região crítica

sem_post(&sem);               // equivalente ao unlock
```

Enquanto o mutex só permite um acesso por vez, semáforos podem ser usados para permitir múltiplos acessos.