OO – Engenharia Eletrônica

-

Programação em C/C++

Slides 18B: Introdução à *Multithreading*.

Exemplos: Programação OO *Multithreading* com pthreads.

Aluno: Vagner Vengue

Threads Orientadas a Objeto

Thread Orientada a Objeto

- O padrão POSIX de threads torna o código fonte mais portável, uma vez que este padrão é aceito pelos principais sistemas operacionais atuais, como Linux, Unix e Windows.

- As pthreads são preparadas para trabalhar com funções, o que é uma vantagem para linguagem estruturadas, como a linguagem C, porém não é tão interessante para linguagens orientadas a objetos, com a linguagem C++.

- Além de ser mais conveniente trabalhar com as threads de maneira OO, torna-se mais fácil a implementação e compreensão do código e dispõem-se de todas as vantagens desse estilo de programação.

Primeira tentativa

(contra-exemplo)

```cpp
#include <iostream>
#include <stdlib.h>

#include <pthread.h>                        // header para pthreads;

using namespace std;

class Thread
{
    private:
        pthread_t _threadID;                // pthread;
        // atributo dat thread, pode conter
        // informações sobre o funcionamento da thread;
        pthread_attr_t _tAttribute;

        void* runThread(void* pThread); // inicia a thread;
        virtual void run();                 // método que executará;
    public:
        Thread();
        ~Thread();

        void start();
        void join();                        // espera a thread acabar;
        void yield();                       // libera o processador;
};
```

```
Thread::Thread()    { }
Thread::~Thread()   { }
void Thread::run()  { }                    // deve ser redefinido;

void Thread::start()
{
    // inicia o atributo;
    int status = pthread_attr_init( &_tAttribute );
    status = pthread_attr_setscope( &_tAttribute,
                                    PTHREAD_SCOPE_SYSTEM );
    // cria uma thread
    status = pthread_create( &_threadID,
                             &_tAttribute,
                             Thread::runThread,
                             (void*)this );
    // destrói o atributo;
    status = pthread_attr_destroy( &_tAttribute );
}

void Thread::yield()
{
    sched_yield();
}
```

Por definição das pthreads, deve ser usado um ponteiro de função e esta deve receber como parâmetro um ponteiro do tipo void.

```
void Thread::join()
{
    pthread_join( _threadID, NULL );
}

void* Thread::runThread(void* pThread)
{
    Thread* sThread = static_cast<Thread*> (pThread);
    sThread->run();                    // executa a thread;
}

class MinhaT : public Thread       // classe derivada de Thread;
{
    public:
        MinhaT() {}
        ~MinhaT(){}

        void run()                 // define o método virtual para
        {                          // executar o código desejado;
            cout << "Oi" << endl;
        }
};
```

```
int main ()
{
    MinhaT *t = new MinhaT;      // cria um objeto da classe derivada;
    t->start();                  // inicia a thread;
    t->join();                   // comando join, para que a thread
                                 // main espere a 't' acabar;
    cout << endl << "Hi! I am the main thread." << endl;

    cout << endl << "pressione ENTER para continuar ...";
    cin.get();
    return 0;
}
```

```
Message
In member function `void Thread::start()':
no matches converting function `runThread' to type `
candidates are: void* Thread::runThread(void*)
[Build Error] [main.o] Error 1
```

Mensagem de erro do compilador Dev C++, que indica que o ponteiro de função passado como parâmetro não é compatível.

Para entender veja o Exemplo 01.

Exemplo 01

(Como criar uma pthread de maneira orientada a objeto)

```cpp
#include <iostream>
#include <stdlib.h>

#include <pthread.h>

using namespace std;

class Thread
{
    private:
        pthread_t _threadID;                    // pthread;
        // atributo da thread, pode conter
        // informações sobre o funcionamento da thread;
        pthread_attr_t _tAttribute;
        // método que inicia as thread;
        static void* runThread(void* pThread);
        virtual void run();                     // código para execução;
        void printError(const string msg) const;
    public:
        Thread();
        ~Thread();

        void start();                           // cria a thread;
        void join();                            // espera a thread acabar;
        void yield();                           // libera o processador;
};
```

A mensagem de erro da primeira tentativa pode ser solucionada com um método estático.

```
Thread::Thread()    { }
Thread::~Thread()   { }
void Thread::run() { }

void Thread::yield()
{
    sched_yield();
}

void Thread::start()
{
    // inicia o atributo;
    int status = pthread_attr_init( &_tAttribute );
    status = pthread_attr_setscope( &_tAttribute,
                                    PTHREAD_SCOPE_SYSTEM );
    if ( status != 0)
        printError("falha ao iniciar atributo da thread.");

    // cria uma thread;
    status = pthread_create( &_threadID, &_tAttribute,
                             Thread::runThread, (void*)this);
    if ( status != 0)
        printError("falha ao iniciar a thread.");

    // destrói o atributo;
    status = pthread_attr_destroy( &_tAttribute );
    if ( status != 0)
        printError("falha ao destruir atributo da thread.");
}
```

```
void Thread::join()
{
    int status = pthread_join( _threadID, NULL );
    if ( status != 0)
        printError("comando join falhou.");
}

void* Thread::runThread(void* pThread)
{
    Thread* sThread = static_cast<Thread*> (pThread);

    if (NULL == sThread){ cout << "thread falhou." <<endl;     }
    else {    sThread->run();  /* executa a thread; */        }
}

void Thread::printError(const string msg) const
{
    cout << "Erro: " << msg << endl;
}
```

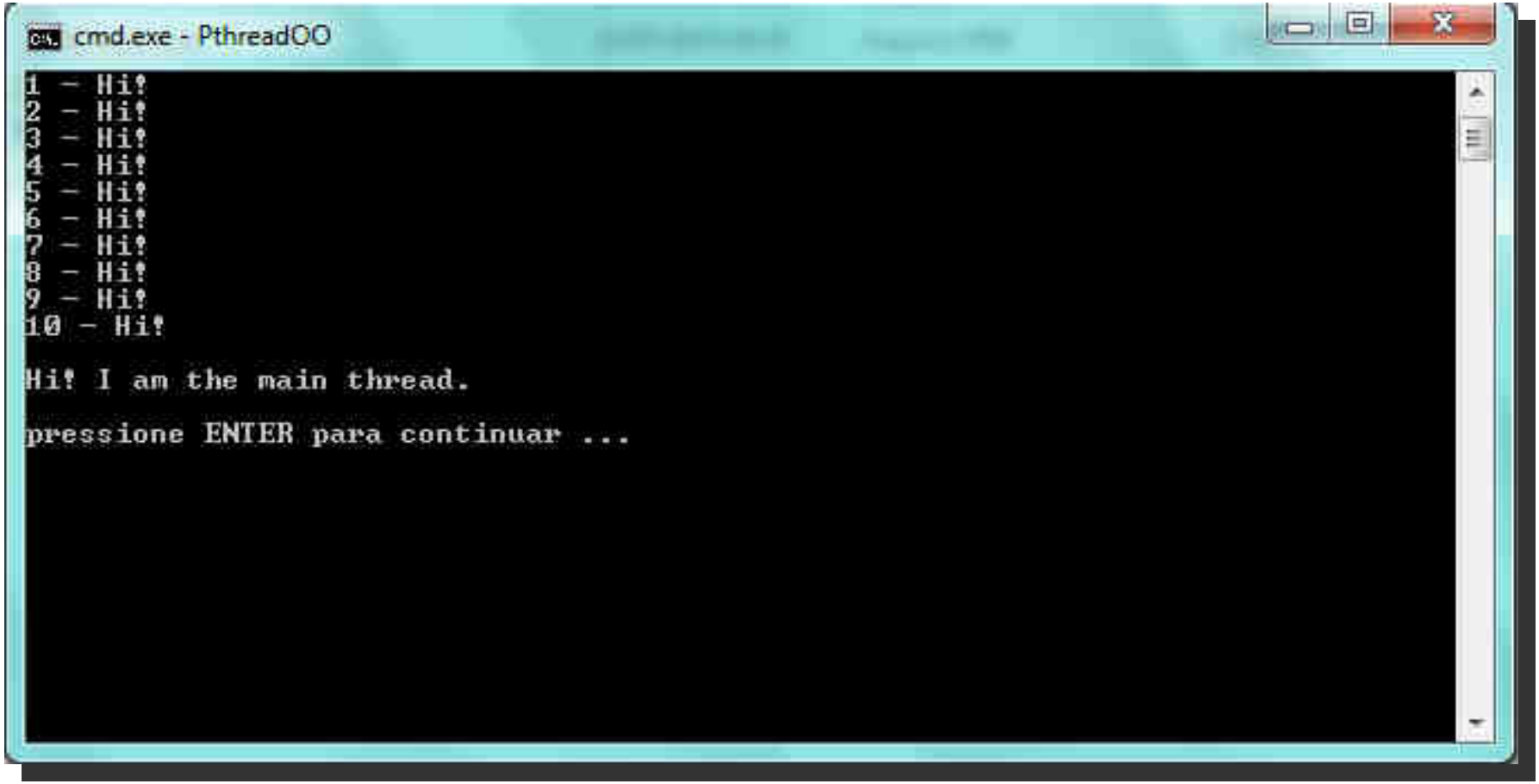
- Quando um método é declarado, implicitamente também é declarado um ponteiro da classe, o ponteiro ‘this’. Desta forma, um método que retorna um ponteiro dentro de uma classe, possui **dois** ponteiros, como o método feito no contra-exemplo primeira tentativa. Isso não ocorre com as funções simples de programação estruturada (modularizada).

-As pthreads recebem como parâmetro apenas **um** ponteiro de função e isso gerou o erro do contra-exemplo. Para contornar esse problema, deve-se utilizar um método estático, que não possui o ponteiro ‘this’.

```
// qualquer classe derivada de Thread é uma classe de thread;
class MinhaT : public Thread
{
    public:
        MinhaT() { }
        ~MinhaT(){ }

        void run()
        {
            for (int i = 0; i < 10; i++)
               cout << i+1 << " - Hi! " << endl;
        }
};

int main ()
{
    MinhaT *t;

    t = new MinhaT;

    t->start();
    t->join();

    cout << endl << "Hi! I am the main thread." << endl;

    cout << endl << "pressione ENTER para continuar ...";
    cin.get();
    return 0;
}
```

Após definida a classe Thread, todas as threads do programa podem ser derivadas dela, como pode ser visto ao lado.

Resultado do programa Exemplo 01.



Exemplo 02

(Um exemplo de uso da pthreads OO)

Thread Orientada a Objeto

- No mundo de competições da Formula 1, os dois irmãos Michael Schumacher e Ralf Schumacher, ambos pilotos, conseguem ter uma plena relação de harmonia. Umas das razões é o fato de não conversarem sobre as corridas quando saem juntos. Vamos supor que em uma corrida de 100 voltas, cada um deles pare de acelerar a cada 10 voltas para que o outro irmão não fique para trás. Construa um programa que simule esta situação da seguinte forma:

  - Cada um dos dois pilotos deve ser uma thread.
  - A cada dez voltas eles param de acelerar (processar).
  - A thread principal deve esperar os dois pilotos acabarem a corrida antes de terminar o programa.
  - Cada piloto deve informar no Console quando acabar as 100 voltas.
  - Desenvolva o programa usando programação orientada a objeto.

```
#ifndef _CORREDOR_H_
#define _CORREDOR_H_

#include <string.h>
#include "Thread.h"

using std::string;
using std::cout;
using std::endl;

// classe Corredor derivada de Thread:
class Corredor : public Thread
{
    private:
        string _nome;
        int _nVoltas;
        void run();
    public:
        Corredor(const int voltas = 0);
        ~Corredor();

        void setNome(const string nome);
        string getNome() const;
        void setVoltas(const int voltas);
        int getVoltas() const;
};
#endif
```

- Neste exemplo foi utilizada a classe Thread do Exemplo 1, que serve de base para a classe Corredor (thread), vista ao lado.

```
#include "Corredor.h"
#include "Principal.h"

Corredor::Corredor(const int voltas)
{
    setVoltas( voltas );
}

Corredor::~Corredor(){ }

// redefinição do método virtual de Thread;
void Corredor::run()
{
    for(int i = 0; i < _nVoltas; i++)
    {
        if ( i % 10 == 0 )
            yield();      // irmãos sendo generosos;
    }

    while( true )
    {
        if ( ! Principal::consoleOcupado )
        {
            Principal::consoleOcupado = true;
            cout << getNome() << " chegou!" << endl;
            Principal::consoleOcupado = false;
            break;
        }
    }
}
```

- Na classe Corredor é preciso apenas redefinir o método virtual 'run' com o código que se deseja executar.

- A cada 10 voltas o "corredor" chama o método yield, que informa ao programa que naquele momento ele está "oferecendo" o recurso processamento para outra thread.

- Para evitar a concorrência das threads pelo Console, foi criada uma variável estática que informa se o Console está ocupado.

- Está solução ameniza o problema de concorrência, mas não o resolve, uma vez que a thread pode ser interrompida exatamente entre o comando 'if' e a linha seguinte, onde a variável deveria mudar para o estado ocupado. Além de desperdiçar processamento.

Thread Orientada a Objeto com Mutex

- Duas ou mais threads podem existir no mesmo processo sem a interferir uma no trabalho da outra, porém, na maioria das vezes elas precisam compartilhar recursos, uma vez que esse compartilhamento é a principal vantagem no uso de múltiplas threads em um programa.
- No exemplo anterior foi utilizada uma técnica chamada *Busy Waiting* para gerenciar o uso do Console pelas duas threads. Essa técnica além de não eliminar o problema de concorrência, ainda consome processamento.
- O Mutex é uma técnica desenvolvida pelos fabricantes de sistemas operacionais, que utiliza uma técnica parecida com o *Busy Waiting*, porém as chances de apresentar erro são quase zero. Essa técnica também tem como principal vantagem o melhor aproveitamento do processador. No momento em que a thread não consegue acessar a região crítica, ela chama o comando Yield automaticamente, liberando o processador.
- Nos sistemas operacionais de computadores pessoais atuais, as threads não sofrem escalonamento e sim os processos, assim como o Mutex tem controle apenas sobre as threads do mesmo processo. Isso se deve ao fato do sistema operacional não poder deixar uma thread influenciar no funcionamento das outras, como em uma situação onde o programador esquece de chamar o método pthread_mutex_unlock.

Exemplo 03

(Um exemplo de uso do Mutex)

Thread Orientada a Objeto com Mutex

- No exemplo anterior foi desenvolvido um programa que simulava uma situação de corrida entre os irmãos Michael e Ralf Schumacher, adapte-o para funcionar com mais corredores, utilizando um Mutex e com as seguintes condições:
  - Cada um dos pilotos deve ser uma thread.
  - Todos devem correr até que seja pressionado ENTER.
  - Ao final deve ser informado o número de voltas que cada um fez e o vencedor.
  - Desenvolva o programa usando programação orientada a objeto.

```cpp
#include <iostream>
#include <pthread.h>
using namespace std;

class Thread
{
    private:
        pthread_t _threadID;                    // pthread;
        static pthread_mutex_t _mutex;          // mutex estático;
        // atributo da thread, pode conter
        // informações sobre o funcionamento da thread;
        pthread_attr_t _tAttribute;
        // método que inicia as thread;
        static void* runThread(void* pThread);
        virtual void run();                     // código para execução;
        void printError(const string msg);
    public:
        Thread();
        ~Thread();

        void start();                           // cria a thread;
        void join();                            // espera a thread acabar;
        void yield();                           // libera o processador;
        void lock();
        void unlock();
};
```

- Mutex estático, comum a todas as threads.

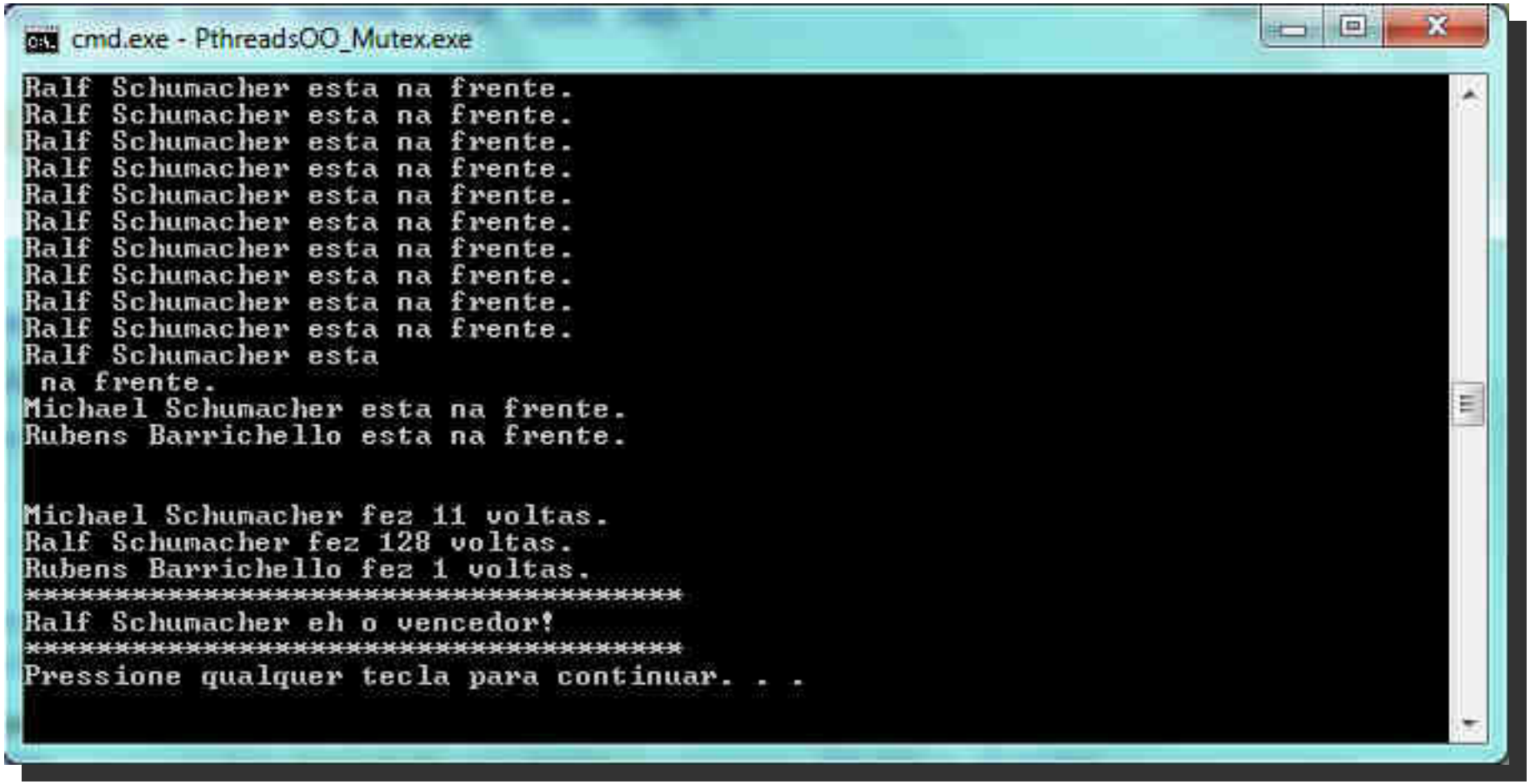
```
void Thread::lock()
{
    if ( NULL == Thread::_mutex )
        pthread_mutex_init( &Thread::_mutex, NULL );

    pthread_mutex_lock( &Thread::_mutex );
}

void Thread::unlock()
{
    if ( NULL != Thread::_mutex )
        pthread_mutex_unlock( &Thread::_mutex );
}

void Thread::printError(const string msg)
{
    lock();      // pára para a mensagem;
    cout << "Erro: " << msg << endl;
    unlock();
}
```

```
Corredor::Corredor(const string nome)
{
    _nome = nome;
    _voltas = 0;
}

Corredor::~Corredor(){}

void Corredor::run()
{   // enquanto o semáforo estiver aberto;
    while ( Principal::_semaforoAberto )
    {
        lock();
        cout << _nome << " esta na frente." << endl;
        unlock();
        _voltas ++;      // incrementas as voltas

        // a cada 10 voltas avisa o programa que está
        // oferecerecendo o processador para outra thread;
        if ( _voltas % 10 == 0)
            yield();
    }
}

int Corredor::getVoltas()
{
    return _voltas;
}
```

- Cada vez que é preciso imprimir uma mensagem, a thread chama a função lock.

- Como as threads não sofrem escalonamento do sistema operacional, mudando apenas quando ocorrem eventos no processo. É responsabilidade do programador gerenciar o uso das threads, com os comando Yield ou Sleep.

Resultado do programa Exemplo 03.



Atividades – Exercícios

- Estudar os programas dos exemplos procurando entender as diferenças entre cada um.

- Desenvolva outros programas para testar a classe Thread criada.

Bibliografias relativas a *Threads*.

- TANENBAUM, Andrew Stuart: **Sistemas Operacionais Modernos.** Prentice Hall. 3ª Edição 2010.