OO – Engenharia Eletrônica

-

Programação em C/C++

Slides 18: Introdução à programação *multi-thread.*

Thread: linha de execução de um processo.
Multi-thread: execução de múltiplas *threads* em um processo.

Prof. Jean Marcelo SIMÃO,
Aluno monitor: Vagner Vengue.

Thread e *Multi-thread*

Obs.: Material inicial elaborado por Murilo S. Holtman no 2o Semestre de 2007. O Holtman era então aluno da disciplina em questão e estagiário em projeto do Professor da disciplina.

Threads

- São **linhas de processamento** de um programa em um processo no computador.
- A execução de um programa utiliza ao menos uma *thread*.
- Permitem a criação de programas com *multi-thread*.

Exemplo de implementação de *threads*

(utilizando CRuntime Library)

Exemplo 01

- Exemplo de um programa que cria uma *thread* utilizando a biblioteca CRuntime Library e a executa até que o usuário pressione ENTER.

```
#include <iostream>
using namespace std;

#include <process.h>
//biblioteca p usar funções de CRuntime
    Library;

bool thread_ligada;

// função passada como parâmetro para nova
    thread;
void escreveAlgo(void *p)
{
    while ( thread_ligada )
    {
          cout << "Executando thread..."
               << endl;
    }
}
```

tem que ser void

ponteiro para void, pois recebe param.

```
int main()
{
    cout << "Digite ENTER para iniciar e parar
a thread..." << endl;
    cin.get();  // aguarda um ENTER do usuário;
    thread_ligada = true;

    // inicia uma thread, passando como
parâmetro a função que deve ser executada;
    _beginthread(escreveAlgo, NULL, 0);

    cin.get();  // aguarda um ENTER do usuário;
    thread_ligada = false;  // informa a thread
que ela deve parar;
    return 0;
}
```

arg1 - ponteiro para função, nome da função
arg2 - parametros
arg3 - tamanho da pilha

Exemplo 02

```c
#include <process.h> // CRuntime Library
#include <stdio.h>
#include <windows.h>

void MeuThread ( void* i );
char frase[100];
int k = 1;

int main ()
{
    puts ( "Vou ligar o Threads enquanto voce
    me passa caracteres");
    // inicia uma thread, passando como
    // parâmetro a função a ser executada;
    _beginthread ( MeuThread, 0, NULL);

    puts ( "Digite alguns carateres" );
    gets ( frase );

    k = 0;
    puts ( frase );
    system("pause");
    return 0;
}
```

```c
void MeuThread(void *i)
{
    while(k)
    {
        // função da Win32 API que faz com que
        // a thread pare de executar por uma
        // quantidade milissegundos, passada
        // como parâmetro;
        Sleep(1000);
        puts ( " " );
        puts ( " Oi eu sou um Thread." );
        puts ( " " );
    }
}
```

- Exemplo de um programa que cria uma *thread* utilizando a biblioteca CRuntime Library e a executa a enquanto o usuário digita caracteres, até que seja pressionado ENTER.

Exemplo 03

- Exemplo de um programa que cria uma *thread* utilizando a biblioteca CRuntime Library e a executa a cada um segundo aproximadamente.

```
#include <process.h>
#include <stdio.h>
#include <windows.h>

int tempo = 0;

void relogio ( void *i );

int main()
{
    _beginthread ( relogio, NULL , 0 );

    while ( 1 )
    {
        printf ( "Programa aberto faz %i
    segundos. \n ", tempo );
        getchar();
    }

    return 0;
}
```

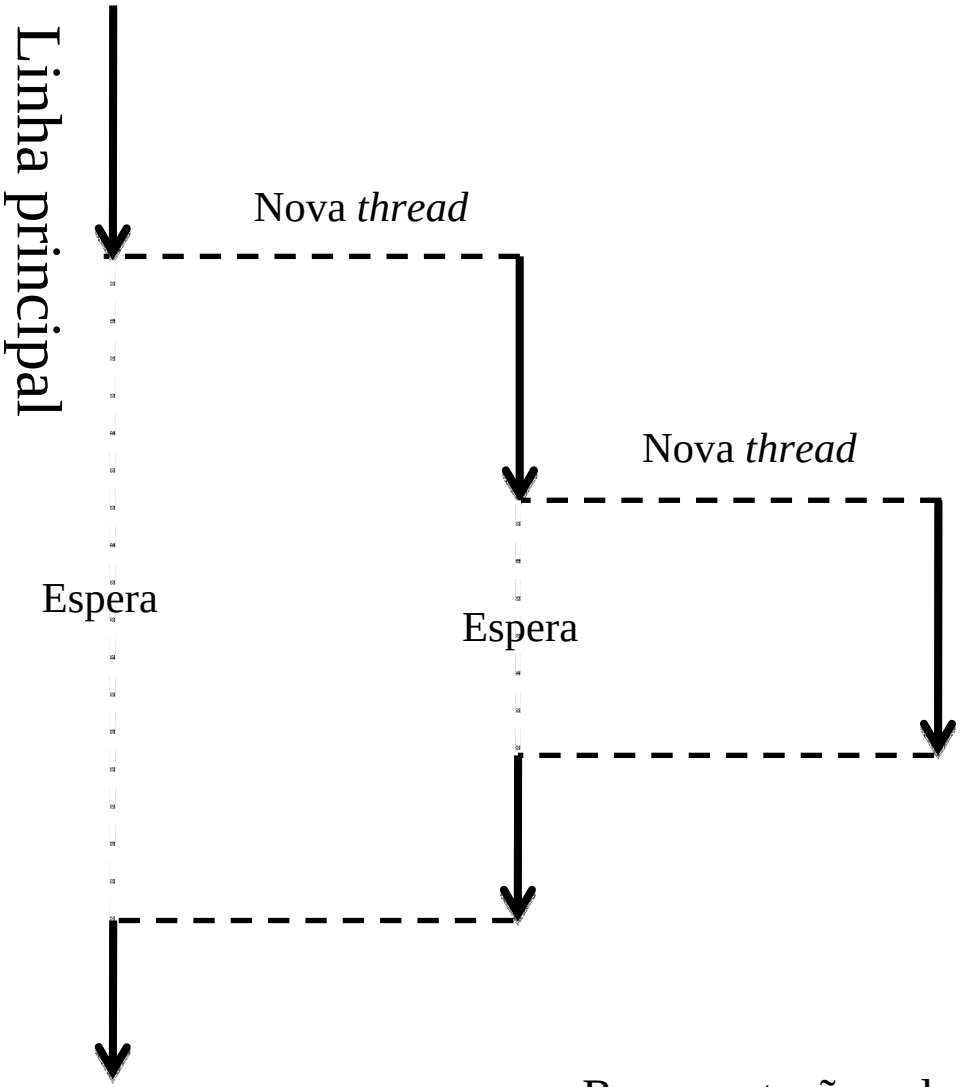
```
void relogio( void *i )
{
    while ( 1 )
    {
        Sleep(1000);
        tempo++;
    }
}
```

Threads x Processos

- *Threads* também são conhecidas como **processos leves** (*lightweight process*), pois têm muitas das características de um processo do computador, porém, com menos recursos.

- Processo: abstração de um programa em execução para o sistema operacional.

- Processos são usados para agrupar recursos, enquanto *threads* são entidades escalonadas (organizadas) para a execução sobre a CPU. (TANENBAUM, 2010)



Representação de um programa em execução com a *thread* principal mais duas *threads*. (3 *threads* e 1 CPU)

Multi-thread

- Situação em que múltiplas *threads* executam em um mesmo processo, ou seja, compartilham os mesmos recursos.

- Em sistema com uma única CPU, as *threads* esperam a vez para executar, porém, com a velocidade de processamento, originam um **pseudo-paralelismo**. Para os seres-humanos é como se elas executassem em paralelo.

- Sistemas com múltiplas CPU's estão se tornando cada vez mais comuns (acessíveis), permitindo algum paralelismo real. No entanto, o número de *threads* em um computador normalmente excede o número de CPU's, gerando um pseudo-paralelismo também.

Multi-thread

- A execução de um programa utiliza ao menos uma *thread*. As demais *threads* devem ser programadas para que executem.

- Amplamente utilizada por *softwares*: *browsers*, jogos, programas que efetuam cálculos e pesquisas demoradas, *softwares* de servidores, entre muitos outros.

Multi-thread - Vantagens

- Permite a **divisão de tarefas** de um programa durante a sua execução, como um editor de textos que pode salvar o texto e verificar os erros de ortografia “ao mesmo tempo” que o texto é editado pelo usuário.

- Permite o **compartilhamento de recursos entre *threads*** de um mesmo programa, como o valor de variáveis globais (na linguagem C++). Isto não seria possível se fosse necessário executar dois programas (processos) para a divisão das tarefas, ao invés de utilizar *multi-thread*.

- Permite um melhor aproveitamento de processamento em sistemas com múltiplas CPU’s.

Multi-thread - Desvantagens

- **Problemas de concorrência**. (comentados a seguir)
- Programas mais difíceis de se depurar (*debugar*).
- A maioria dos programas existentes são *mono-thread*, torná-los *multi-thread* pode ser difícil.
- Exige mais conhecimento dos desenvolvedores.

Problemas em programação *Multi-thread*

Problemas no uso de *threads*

- O principal problema em programas com várias linhas de execução concorrentes diz respeito à sincronização no acesso a recursos compartilhados (blocos de memória, arquivos, dispositivos etc.) por várias tarefas (*threads*).

- Estes problemas são circundados ao se escrever código seguro *(thread-safe)* que inclui a utilização de rotinas reentrantes e objetos de sincronização (como semáforos e mutexes).

Exemplo de conflito entre *threads* que acessam um mesmo recurso

Tarefa da thread 1:

```
{
  ...
  char *s;
  for (s = “ABCDEF”; *s != ‘\0’; s++)
  {
     fputc(*s, stdout);
  }
  ...
}
```

Tarefa da thread 2:

```
{
  ...
  char *s;
  for (s = “123456”; *s != ‘\0’; s++)
  {
     fputc(*s, stdout);
  }
  ...
}
```

Considerando o seguinte contexto: (a) a fatia de tempo destinado a *thread* 1, pelo sistema operacional, se esgota enquanto ela ainda está executando seu laço de repetição; (b) a *thread 2* é então executada de maneira intercalada com a *thread 1*; (c) isto se repete algumas vezes. Assim sendo, a saída padrão (“tela”) poderá conter algo como:

```
ABC1D2E3F456
```

Região Crítica

- Também chamada de Sessão Crítica. É uma **parte do programa em que há acesso a um recurso compartilhado** por *threads*.

- É a parte de código em que se deve utilizar os objetos de sincronização, criando-se códigos seguros (*thread-safe*). Onde apenas uma *thread* por vez acessa a região crítica.

- Exemplo: as partes de código do exemplo anterior onde houve o conflito.

```
for (s = “ABCDEF”; *s != ‘\0’; s++)
{
     fputc(*s, stdout);
}
```

```
for (s = “123456”; *s != ‘\0’; s++)
{
     fputc(*s, stdout);
}
```

Objetos de sincronização em programação *Multi-thread*

Mutexes

- Um mutex (*mutual exclusion*) é uma variável (objeto) que pode estar em dois estados: “impedido” e “desimpedido”.

- São utilizados para **exclusão mutua**, ou seja, permitem que apenas uma *thread* por vez acesse um recurso.

- Se uma *thread* tenta acessar um recurso que outra *thread* está bloqueando, é impedida e libera o processador para que outras *threads* executem. Isso garante que uma *thread* não desperdice processamento porque está aguardando por um recurso bloqueado por outra *thread*.

Semáforos

- São mecanismos que **permitem que um determinado número de *threads* tenham acesso a um recurso**. Agindo como um contador que não deixa ultrapassar um limite.

- No momento em que um objeto de semáforo é criado, é especificada a quantidade máxima de *threads* que ele deve permitir. Então cada *thread* que queira acessar o recurso, deve chamar uma função que decrementa em 1 o semáforo (*down*) e, após utilizar o recurso, chamar uma função que incremente em 1 o semáforo (*up*). Quando o contador do semáforo chega a zero, significa que o número de *threads* chegou ao limite e o recurso ficará bloqueado para as *threads* que chegarem depois, até que pelo menos uma das *threads* que estão utilizando o recurso o libere, incrementado o contador do semáforo.

Mensagens

- Diferentemente de mutex e semáforos, as mensagens não são específicas para o controle de acesso a um recurso. Elas são usadas como uma **forma de comunicação entre *threads***.

- Basta definir um conjunto de mensagens e trocá-las entre as *threads*.

- Pode ser útil quando se tem *threads* executando de forma continua, pois cada uma delas precisa verificar a sua fila de mensagens de tempos em tempos (em laço).

- É a forma de comunicação utilizada pelo sistema operacional Windows para gerenciar os objetos gráficos.

Prioridades

- As *threads* trabalham com a mesma prioridade, no entanto, pode-se alterar a prioridade de algumas *threads* para que executem por mais tempo que outras.

- É preciso tomar cuidado, pois se uma *thread* tiver prioridade muito mais baixa que as demais, talvez nunca execute, ou execute muito pouco, acabando com as vantagens de *multi-thread*. Este tipo de problema é conhecido como condições de corrida (***racing conditions***).

- As prioridades dependem da plataforma e da API utilizada.

Bibliografias relativas a *Threads*.

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

- http://msdn.microsoft.com/en-us/library/ms686364(v=VS.85).aspx – Acesso em 22/10/2010.