

Conceitos de Linguagem de Programação

Concorrência

DAVI VENTURA CARDOSO PERDIGÃO
EDMILSON LINO CORDEIRO
ERIC HENRIQUE DE CASTRO CHAVES

Sumário



Introdução

Arquitetura, Categorias,
Motivações e Conceitos

...



Semáforos

Sincronização e Avaliação

...



Monitores

Sincronização e Avaliação

...



Passagem de Mensagens

Conceito

...



Suporte de ADA

Objetos e Avaliação

...



JAVA e C#

Exemplos de códigos

...



Linguagens Funcionais e Nível de Sentença

Multi-LISP, F# e Fortran

...



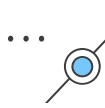
Conclusão

...

01

Introdução

**Arquitetura, Categorias, Motivações
e Conceitos**



- Usada inicialmente na construção de **sistemas operacionais**. Atualmente, usada para desenvolver **aplicações** em **todas** as áreas da computação.
- Tornou-se ainda mais importante com o advento dos sistemas distribuídos e das máquinas com **arquitetura paralela**.
- À **primeira vista**, a concorrência pode parecer um conceito **simples**.

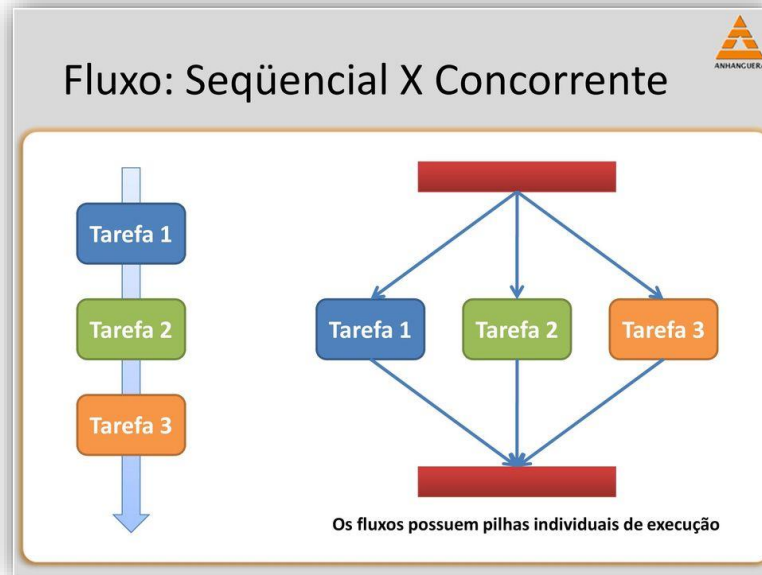
A concorrência na execução de software pode ocorrer em **quatro níveis**:

1. **Nível de instrução** - executar duas ou mais instruções de máquina simultaneamente;
2. **Nível de sentença** - executar duas ou mais sentenças de linguagem de alto nível simultaneamente;
3. **Nível de unidade** - executar duas ou mais unidades de subprograma simultaneamente;
4. **Nível de programa** - executar dois ou mais programas simultaneamente.



Arquiteturas Multiprocessadas

Um programa **sequencial** é composto por um conjunto de instruções que são executadas sequencialmente, sendo que a execução dessas instruções é denominada **processo**. Um programa **concorrente** especifica **dois ou mais** programas sequenciais que podem ser executados **concorrentemente** como processos paralelos:



...
Diversas arquiteturas possuem **mais de um processador** e **suportam** execuções concorrentes. Os **primeiros** computadores com múltiplos processadores apresentavam um processador **geral** e um ou mais processadores **"periféricos"**, para operações de **entrada e saída**.

Entretanto, o **fim** da sequência de **aumentos** significativos na **velocidade** de processadores **individuais** está próximo. **Aumentos** significativos no poder da computação resultam agora o aumento no **número** de **processadores** (grandes sistemas de servidor, como os executados pelo Google e pela Amazon). Muitas outras tarefas de computação volumosas são agora executadas em máquinas com grande número de **processadores pequenos**.

Outro avanço no hardware de computação foi o desenvolvimento de **vários processadores** em um **único chip** (Core Quad), que estão **pressionando** os desenvolvedores a usar os múltiplos processadores. Caso contrário, a **concorrência** será e os ganhos de **produtividade diminuirão**.
...

Categorias de Controle de Unidades



Física

A mais natural, assumindo a disponibilidade de mais de um processador, diversas unidades do mesmo programa são executadas simultaneamente.

...



Lógica

Um ligeiro relaxamento implica na concorrência lógica, que permite ao programador e ao aplicativo de software assumirem a existência de múltiplos processadores fornecendo concorrência, quando a execução real dos programas está ocorrendo de maneira intercalada em um processador.



Motivações

1. **Velocidade de execução de programas em máquinas com múltiplos processadores** - máquinas mais efetivas, desde que sejam projetados para usar a concorrência em hardware. É um desperdício não usar essa capacidade de hardware;
2. **Velocidade de execução quando comparado à sequencial** - mesmo quando uma máquina tem apenas um processador;
3. **Fornece um método diferente de conceituar soluções de programas para problemas** - muitos domínios de problema se prestam naturalmente à concorrência, da mesma forma que a recursão é uma maneira natural de projetar soluções para alguns problemas, por exemplo, aeronaves voando em um espaço aéreo controlado;
4. **Aplicações são distribuídas por várias máquinas, localmente ou pela Internet** - em todo sistema operacional existem muitos processos concorrentes sendo executados o tempo todo, gerenciando recursos, obtendo entrada de teclados, exibindo saída de programas e lendo e escrevendo em dispositivos de memória externos. Em resumo, a concorrência se tornou um aspecto onipresente da computação.





Conceitos Fundamentais: Tarefas

Uma **tarefa (processo)** é uma unidade de um programa, similar a um subprograma, que pode estar em execução concorrente com outras unidades do mesmo programa. Em algumas linguagens, por exemplo, Java e C#, certos métodos são executados em objetos chamados **linhas de execução**. Três características das tarefas as distinguem dos subprogramas:

01 Pode ser implicitamente iniciada, enquanto um subprograma precisa ser explicitamente chamado.

02 Quando uma unidade de programa invoca uma tarefa, em alguns casos ela não precisa esperar que a tarefa conclua sua execução antes de continuar a sua própria.

03 Quando a execução de uma tarefa estiver concluída, o controle pode ou não retornar à unidade que iniciou sua execução.

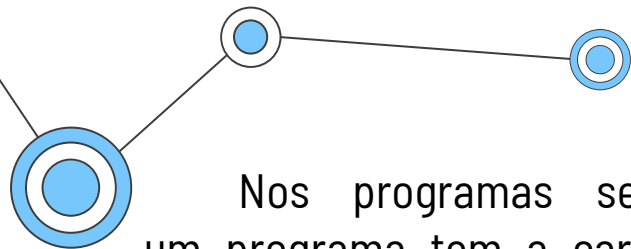


Conceitos Fundamentais: Tarefas

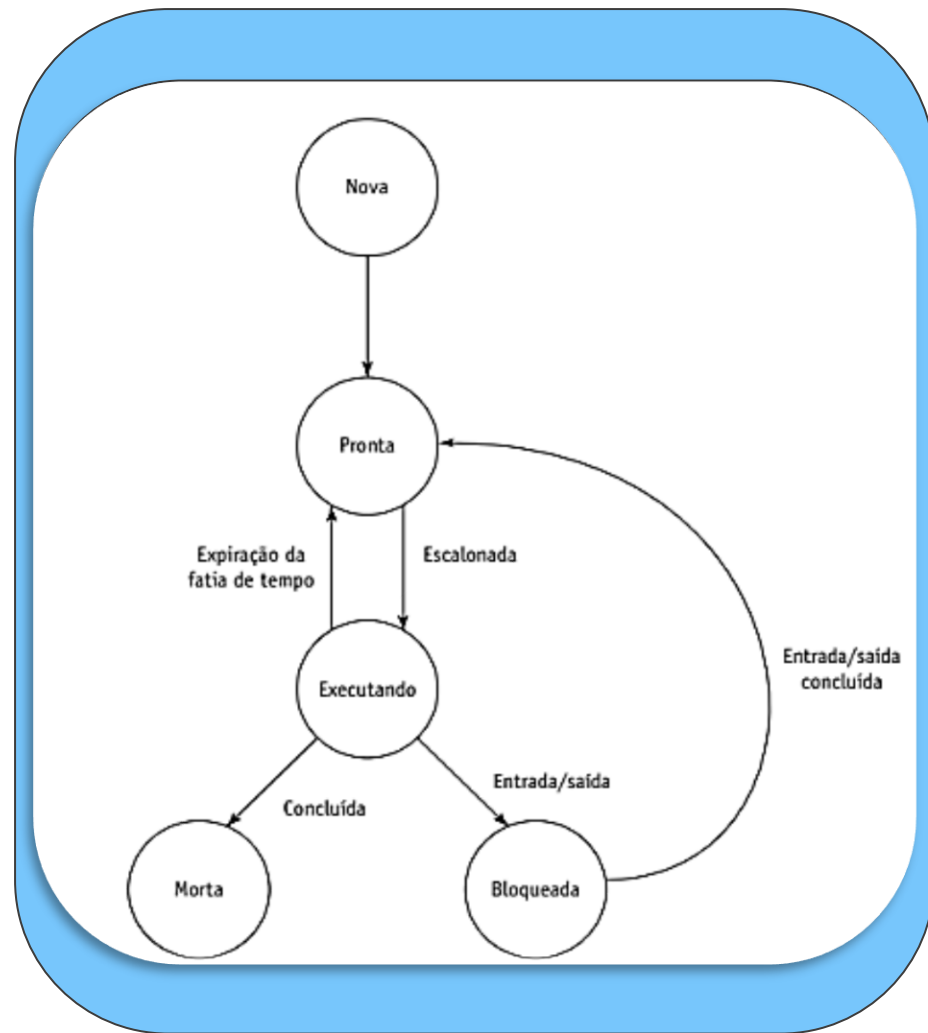


As tarefas são enquadradas em duas categorias: **pesadas (heavyweight)** e **leves (lightweight)**. Uma tarefa pesada executa em seu próprio espaço de endereçamento. Já as tarefas leves são todas executadas no mesmo espaço de endereçamento. As tarefas podem estar em diversos **estados**:

1. **Nova**: quando foi criada, mas ainda não iniciou sua execução;
2. **Pronta**: uma tarefa pronta para ser executada, mas não executada atualmente. As tarefas prontas para serem executadas são armazenadas em uma fila chamada de fila de tarefas prontas;
3. **Executando**: tem um processador e seu código está sendo executado;
4. **Bloqueada**: uma tarefa que estava executando, mas foi interrompida por um entre diversos eventos (o mais comum é uma operação de entrada ou de saída);
5. **Morta**: uma tarefa morta não está mais ativa em nenhum sentido, ou seja, sua execução está concluída.



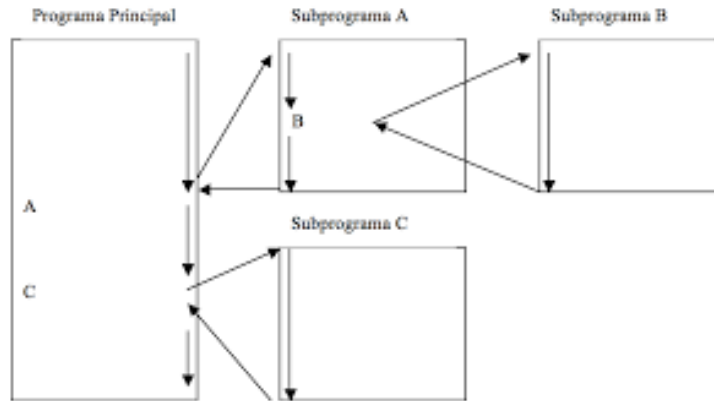
Nos programas sequenciais, um programa tem a característica de **vivacidade** se continua a executar, eventualmente sendo concluído. Isso significa que seu progresso é **continuamente** feito. Em um ambiente com recursos compartilhados, a vivacidade de uma tarefa pode deixar de existir e o programa pode não continuar, dessa forma **nunca terminará (Deadlock)**.



Conceitos Fundamentais: Sincronização

Mecanismo que controla a **ordem** na qual as tarefas são executadas. Dois tipos de sincronização são necessários quando as tarefas compartilham dados:

- A **cooperação** é necessária entre as tarefas A e B quando A deve esperar que B conclua alguma tarefa específica antes que possa começar ou continuar sua execução;
- A **competição** é necessária entre duas tarefas quando ambas exigem o uso de algum recurso que não pode ser simultaneamente utilizado;





02

Semáforos

Sincronização e Avaliação




Um **semáforo** é um mecanismo de **sincronização** de **tarefas**. Embora sejam uma estratégia primitiva, ainda são utilizados tanto linguagens atuais e em sistemas com suporte à concorrência baseado em biblioteca.

Para fornecer acesso limitado a uma estrutura de dados, **guardas** podem ser colocados em torno do código que acessa a estrutura. Uma guarda é um dispositivo linguístico que permite ao código guardado ser executado apenas quando uma **condição** específica for **verdadeira**. Um semáforo é uma implementação de uma guarda. Especificamente, um semáforo é uma estrutura de dados que consiste em um inteiro e em uma fila que armazena **descritores de tarefas**. A estratégia típica é ter **requisições** para o acesso que ocorram quando ele não pode ser dado, ou que sejam armazenadas na fila de descritores de tarefas, da qual elas podem ser obtidas posteriormente, de forma a ser permitido deixar e executar o código guardado. ...



Sincronização de Cooperação

Para a sincronização de **cooperação**, o buffer necessita gravar tanto o número de **posições vazias** quanto o de **posições preenchidas**. O componente de **contagem** pode ser usado para esse propósito.



...

- Uma **variável** de semáforo pode usar seu contador para manter o número de posições vazias em um buffer compartilhado entre produtores e consumidores, e outra pode usar o contador para o número de posições preenchidas. As **filas** desses semáforos armazenam os descritores de tarefas em **espera**;
- A fila **emptyspots** pode armazenar tarefas de produtor à espera por posições disponíveis no buffer;
- A fila **fullspots** pode armazenar tarefas de consumidor que estão aguardando os valores serem colocados no buffer.



Sincronização de Competição

O acesso à **estrutura** pode ser controlado com um **semáforo adicional**. Esse semáforo não precisa contar nada, mas pode simplesmente indicar com seu contador se o **buffer** está em **uso**. A sentença **wait** permite o acesso apenas se o contador do semáforo tiver o **valor 1**, indicando que o buffer compartilhado não é acessado. Se o contador do semáforo tem o valor 0, existe um acesso atual ocorrendo e a tarefa é colocada na fila do semáforo. O contador do semáforo deve ser **inicializado com 1** e as filas dos semáforos devem ser sempre **inicializadas como vazias** antes que seu uso possa começar.



Avaliação

Usar semáforos para sincronização de cooperação cria um ambiente de programação **inseguro**. **Não** há como verificar estaticamente a **exatidão** de seu uso, o qual depende da semântica do programa em que aparecem, podendo haver, por exemplo, **falhas de sincronização de cooperação**.

Os problemas de **confiabilidade** que os semáforos causam ao fornecer sincronização de cooperação também surgem quando eles são usados para **sincronização de competição**. Omitir a sentença **wait(access)** em qualquer uma das tarefas pode causar **acessos inseguros ao buffer**. Omitir a sentença **release(access)** em qualquer das tarefas resultaria em um **impasse**. Essas são **falhas de sincronização de competição**.

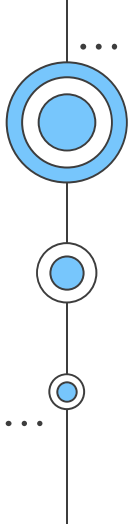
.



03

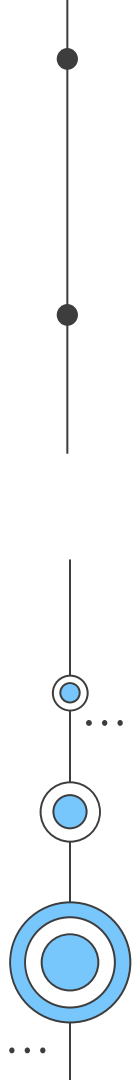
Monitores

Sincronização e Avaliação



Uma **solução** para alguns dos problemas dos semáforos em um ambiente concorrente é **encapsular** as estruturas de dados compartilhadas com suas operações e **ocultar** suas representações – ou seja, fazer com que elas sejam tipos de **dados abstratos**, com algumas restrições especiais. Essa solução pode fornecer sincronização de competição sem semáforos, transferindo a responsabilidade de sincronização para o sistema de tempo de execução.

A primeira linguagem de programação a incorporar monitores foi **Pascal Concorrente** (Hansen, 1975). **Modula** (Wirth, 1977), **CSP/k** (Holt, 1978) e **Mesa** (Mitchell, 1979) também fornecem monitores. Entre as linguagens contemporâneas, eles são suportados por **Ada**, **Java** e **C#**.



...

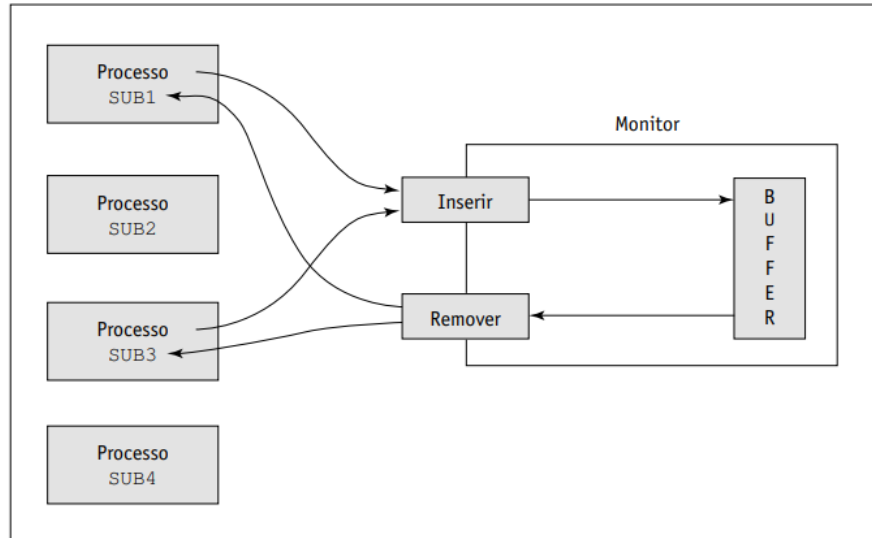


Sincronização de Competição

Um dos recursos mais **importantes** dos monitores é o fato de que os **dados compartilhados residem** no monitor, em vez de em uma das unidades clientes. O programador **não** sincroniza o acesso mutuamente exclusivo aos dados compartilhados por meio de semáforos ou outros mecanismos. Como os mecanismos de acesso fazem parte do monitor, a implementação de um pode ser feita de forma a **garantir acesso sincronizado**, permitindo apenas um acesso de cada vez. **Chamadas** a procedimentos do monitor são implicitamente **bloqueadas** e **armazenadas** em uma fila, caso ele estiver ocupado no momento da chamada.

Sincronização de Cooperação

Apesar do acesso mutuamente exclusivo a dados compartilhados ser **particular** a um **monitor**, a **cooperação entre processos** ainda é tarefa do **programador**. Em particular, o programador deve **garantir** que um buffer compartilhado **não** sofra transbordamentos positivos ou negativos. Diferentes linguagens fornecem maneiras distintas de programar a sincronização de cooperação, todas relacionadas aos **semáforos**.





Avaliação

Monitores são uma forma **melhor** de fornecer **sincronização de competição** do que os **semáforos**. A sincronização de cooperação ainda é um **problema** dos monitores. Semáforos e monitores são igualmente **poderosos** para expressar o controle de concorrência – os semáforos podem ser usados para implementar monitores e os monitores podem ser usados para implementar semáforos.

Ada fornece **duas maneiras** de **implementar** monitores. Ada 83 inclui um modelo **geral** de tarefas que pode ser usado para suportá-los. Ada 95 adicionou uma maneira mais **limpa** e **eficiente** de construí-los, chamada de objetos protegidos. Ambas as estratégias usam como o modelo básico para suportar concorrência. O modelo de **passagem de mensagens** permite que as unidades concorrentes sejam **distribuídas**, enquanto os monitores **não** permitem.

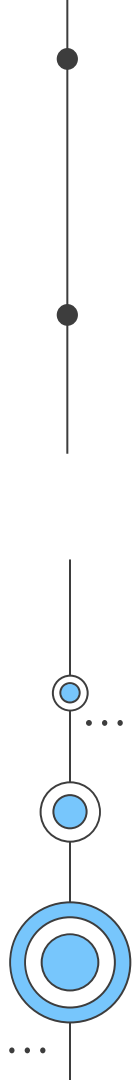



04

Passagem de Mensagens

Conceitos





A passagem de mensagens refere-se ao **envio** de uma **mensagem** a um **processo** que pode ser um **objeto**, **processo paralelo**, **sub-rotina**, **função** ou **thread**. Esta mensagem pode ser usada para **chamar** outro **processo**, direta ou indiretamente. A passagem de mensagens é especialmente **útil** na programação orientada a objetos e na programação paralela quando uma única mensagem (na forma de um sinal, pacote de dados ou função) é enviada a um destinatário.

...



Passagem Síncrona de Mensagens

A passagem de mensagens pode ser **síncrona** ou **assíncrona**. O conceito básico da passagem síncrona decorre do fato de que as tarefas estão normalmente **ocupadas** e não podem ser interrompidas por outras unidades.

Uma tarefa pode ser projetada de forma a **suspender** sua execução em determinado momento, seja porque está desocupada ou porque precisa de informações de outra unidade para poder continuar. Isso é como uma pessoa esperando uma chamada importante. Em alguns casos, não há nada a fazer além de sentar e esperar. Entretanto, durante um **rendezvous**, a informação da mensagem pode ser transmitida em qualquer das direções (ou em ambas). Tanto a sincronização de tarefas de cooperação quanto a de competição podem ser **manipuladas** pelo modelo de passagem de mensagens.

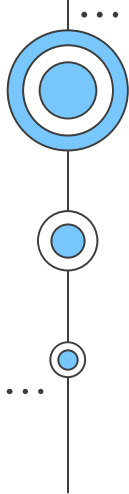


05

Suporte de ADA

Objetos e Avaliação

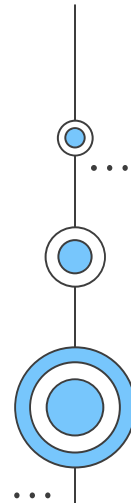




As tarefas Ada podem ser **mais ativas** que os monitores, pois são **entidades passivas** que fornecem serviços de gerenciamento para os dados compartilhados que armazenam. Quando usadas para gerenciar dados compartilhados, as tarefas Ada funcionam como **gerentes** que podem residir com os recursos que gerenciam. Elas têm diversos mecanismos que as permitem escolher entre requisições que competem pelo acesso aos seus recursos.

Existem duas partes em uma tarefa Ada – uma de **especificação** e uma de **corpo**. A interface de uma tarefa são seus pontos de entrada, ou posições onde ela pode receber mensagens de outras tarefas.

O corpo de uma tarefa deve incluir uma forma sintática dos pontos de entrada que correspondem às cláusulas **entry** na parte de especificação da tarefa. Esses pontos de entrada são especificados pela palavra reservada **accept**. A cláusula accept é definida na faixa de sentenças que começam com a palavra reservada accept e terminam com a palavra reservada **end**.





Objetos Protegidos



O **acesso** a dados compartilhados pode ser **controlado** envolvendo-se os dados em uma tarefa e permitindo-se o acesso apenas por meio de entradas de **tarefas**, as quais fornecem **sincronização de competição** implicitamente. Um problema desse método é a **dificuldade** de implementar o mecanismo de rendezvous. Os objetos protegidos de Ada 95 oferecem um **método alternativo** de fornecer sincronização de competição que não envolve o rendezvous.

Objetos protegidos podem ser **acessados** por subprogramas protegidos ou por entradas sintaticamente **semelhantes** às cláusulas **accept** nas tarefas. Chamadas de entrada para um objeto protegido fornecem **comunicação síncrona** com uma ou mais tarefas por meio do mesmo objeto protegido. Essas chamadas de entrada permitem acesso **similar** àquele fornecido para os **dados** envoltos em uma **tarefa**.



Avaliação

Na **ausência** de **processadores distribuídos** com **memórias independentes**, a escolha entre monitores e tarefas com passagem de mensagens como uma forma de implementar acesso sincronizado a dados compartilhados em um ambiente concorrente é uma questão de **preferência** pessoal. Entretanto, no caso de **Ada**, os objetos protegidos são claramente **melhores** que as tarefas para suportar acesso concorrente a dados compartilhados. O código não só é mais **simples**, mas também muito mais **eficiente**.

Para sistemas distribuídos, a **passagem de mensagens** é um modelo **melhor** para a concorrência, porque suporta o conceito de **processos separados** executando em **paralelo** em processadores separados.



06

Java e C#

Exemplos de Códigos



JAVA

Utilização de Interface **Runnable** e a classe **Thread** para possibilitar a execução de mais de um fluxo de controle sobre o código de programa. A **metáfora** é que as instâncias da classe Thread são abstrações de **trabalhadores**:

```
Thread operario= new Thread();
```

```
Thread trabalhador= new Thread();
```

Mas os trabalhadores precisam de **tarefas**. As instâncias de classes que implementam a interface Runnable devem definir as tarefas:

```
class trabalho implements Runnable{...}
```

```
class tarefa implements Runnable{...}
```

Podemos **associar** uma tarefa a um trabalhador utilizando um construtor de Thread. Uma instância de Thread invoca o método **run()** da instância de Runnable. o método **start()** de uma instância de Thread inicia o fluxo de controle. Só se pode chamar start() **uma vez**:

```
class Tarefa implements Runnable{
```

```
... public void run() /* código para realizar a tarefa */
```

```
}
```

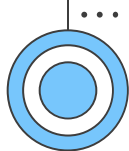
```
public static void main(String[] args){
```

```
    Runnable tarefa=new Tarefa();
```

```
    Thread trabalhador=new Thread(tarefa);
```

```
    trabalhador.start(); /* o fluxo de controle irá retornar, mas um novo fluxo permanecerá em start(). */
```

```
}
```

Os semáforos são implementados através da classe **Semaphore**, disponível no pacote **java.util.concurrent.Semaphore**. Semaphore recebe um parâmetro inteiro, o qual inicializa o contador do semáforo:

```
fullspots = new Semaphore(0);
```

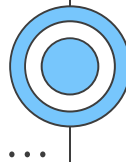
```
emptyspots = new Semaphore(BUFLEN);
```

Do mesmo modo, a operação fetch do método consumer:

```
fullspots.acquire();
```

```
fetch(value);
```

```
emptyspots.release();
```



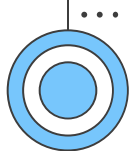
C#

Em vez de apenas métodos chamados **run**, como em Java, **qualquer método** C# pode executar em sua própria linha de execução, que são associadas a uma instância de um representante predefinido, **ThreadStart**. Quando a execução de uma linha inicia, seu representante tem o **endereço** do método que deve executar. Uma linha de execução é criada por meio de um **Thread**, que deve receber uma instanciação de ThreadStart, à qual deve ser enviado o nome do método a ser executado:

```
public void MyRun1() { ... }
```

```
Thread myThread = new Thread(new ThreadStart(MyRun1));
```

Exemplo de uma linha de execução chamada **myThread**, cujo representante aponta para o método **MyRun1**. Assim, quando a linha de execução começa, ela chama o método cujo endereço está em seu representante. Nesse exemplo, myThread é o **representante** e MyRun1 é o **método**.



```
public float MyMethod1(int x);
```

```
Thread myThread = new Thread(new ThreadStart(MyMethod1));
```

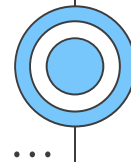
A seguinte sentença chama **MyMethod** de forma assíncrona:

```
IAsyncResult result = myThread.BeginInvoke(10, null, null);
```

O valor de retorno da linha de execução chamada é obtido com o método **EndInvoke**, que recebe como parâmetro o objeto (**IAsyncResult**) retornando **BeginInvoke**. EndInvoke retorna o valor de retorno da linha de execução chamada.

Se o chamador precisa continuar algum trabalho, ele deve ter um meio de determinar quando. Para isso, a interface **IAsyncResult** define a propriedade **IsCompleted**. Ela pode executar em um laço **while**.

Essa é a maneira **eficiente** de fazer algo na linha de execução chamadora enquanto se espera que a linha de execução conclua seu trabalho.





07

Linguagens Funcionais e Nível de Sentença

Multi-LISP, F# e Fortran





Linguagens Funcionais



01

Multi-LISP, uma extensão de **Scheme** que permite especificar partes de um programa para serem executadas concorrentemente. Essas formas são implícitas, o programador simplesmente informa (através da construção **pcall**) ao compilador sobre algumas partes do programa que podem ser executadas concorrentemente. Se uma chamada de função é incorporada a uma construção pcall, os parâmetros da função podem ser avaliados concorrentemente: **(pcall f a b c d)**

02

F#, parte da plataforma **.NET** e concorrência baseada nas mesmas classes utilizadas por C# (**System.Threading.Thread**). Suponha que queiramos executar a função **myConMethod** em sua própria linha de execução. A função a seguir, quando chamada, criará a linha de execução e iniciará a execução da função nela:

```
let createThread() =  
    let newThread = new Thread(myConMethod)  
    newThread.Start()
```



Nível de Sentença



Fortran de alto desempenho, também chamado de **HPF** (High-Performance Fortran) é uma coleção de extensões a Fortran 90 feitas para permitir que os programadores especifiquem informações ao compilador a fim de ajudá-lo a otimizar a execução de programas em computadores multiprocessados.

As suas principais sentenças de especificação são usadas para **descrever o número de processadores**, a **distribuição dos dados** nas memórias desses processadores e o **alinhamento dos dados** com outros em termos de localização de **memória**. Essas sentenças aparecem como comentários especiais em um programa Fortran e são introduzidos pelo prefixo **!HPF\$**. A especificação **PROCESSORS** tem a seguinte forma: **!HPF\$ PROCESSORS procs (n)**. Essa sentença é usada para especificar ao compilador o número de processadores que podem ser usados pelo código gerado para esse programa.

As especificações **DISTRIBUTE** e **ALIGN** são usadas para fornecer informações ao compilador sobre máquinas que não compartilham memória, isto é, cada processador tem sua própria memória.

DISTRIBUTE especifica quais dados serão distribuídos e o tipo de distribuição a ser usada: **!HPF\$ DISTRIBUTE (tipo) ONTO procs :: lista_de_identificadores**

A forma de **ALIGN** é: **ALIGN elemento_matriz1 WITH elemento_matriz2**

ALIGN lista1(index) WITH lista2(index+1)

A sentença **FORALL** especifica uma sequência de sentenças de atribuição que podem ser executadas concorrentemente:

FORALL (index = 1:1000)

list_1(index) = list_2(index)

END FORALL



08

Conclusão



Computação paralela aplicada de forma eficiente em conjuntos de computadores multiprocessados pode resultar em **ganhos consideráveis** em termos de tempos de execução, tornando possível, por exemplo, a monitoração de diversos tipos de sistemas em **tempo real**. No entanto, como a maioria dos algoritmos utilizados para tais tarefas foram inicialmente desenvolvidos com arquiteturas **computacionais sequenciais**, ou seja, com um único processador, estes não fazem uso completo destes novos ambientes paralelos de computação. Desta forma, há a necessidade de se **adaptar antigas metodologias** ou desenvolver **novas técnicas**, que sejam capazes de aproveitar a capacidade computacional.





Referências Bibliográficas



- CASTOR, Fernando. **Threads - Programação Concorrente**. Disponível em: <https://www.cin.ufpe.br/~if686/aulas/13_Threads_PC.pdf>. Acesso em: 19 de novembro de 2022.
- COURTOIS, Parnas. **Concurrent Control with "Readers" and "Writers"**. Comm. Publicado em: 10 de novembro de 1971.
- ROCHA, Rodrigo. **Aula 15 - Concorrência**. Disponível em: <<https://rodrigorgs.github.io/aulas/mata56/aula15-concorrencia>>. Acesso em: 16 de novembro de 2022.
- SEBESTA, Robert. **Conceitos de Linguagem de Programação**. Ed.Bookman. Publicado em: 17 de janeiro de 2011.
- TAFT, Duff. **ADA 95: Reference Manual language and standard libraries**. Springer-Verlag. Publicado em: 1995.
- TOMÉ, Maiqui. **Olá Mundo da Programação Concorrente**. Disponível em: <<https://dev.to/maiquitome/ola-mundo-da-programacao-concorrente-57mp>>. Acesso em: 14 de novembro de 2022.
- TOSCANI, Carissimi. **Sistemas Operacionais e Programação Concorrente**. Série didática do II-UFRGS. Publicado em: 2003.

A decorative graphic on the left side of the slide. It consists of a vertical line with several circular nodes. The nodes are blue with white outlines. The top node is a small circle. Below it is a larger circle. Below that is a small circle. At the bottom is a large circle. There are three dots above the top node and three dots to the right of the bottom node. The line connects the nodes in a slightly curved path.

Obrigado!

Alguma pergunta?

A decorative graphic on the right side of the slide. It consists of a vertical line with several circular nodes. The nodes are blue with white outlines. The top node is a small circle. Below it is a larger circle. Below that is a small circle. At the bottom is a small circle. There are three dots above the top node and three dots to the right of the bottom node. The line connects the nodes in a slightly curved path.