UNIVERSIDADE DE ITAÚNA

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

Conceitos de Linguagem de Programação

Concorrência

ITAÚNA
2022

SUMÁRIO

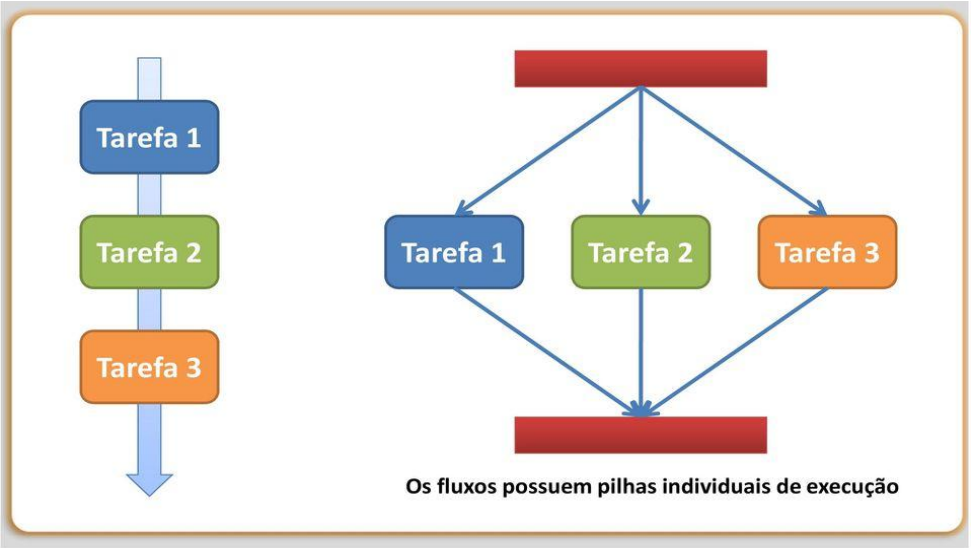
1. INTRODUÇÃO

A programação concorrente foi usada inicialmente na construção de sistemas operacionais. Atualmente, ela é usada para desenvolver aplicações em todas as áreas da computação. Este tipo de programação 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, mas apresenta desafios significativos para o programador, para o projetista de linguagem de programação e para o projetista de sistema operacional.

A concorrência na execução de software pode ocorrer em quatro níveis: em nível de instrução (executar duas ou mais instruções de máquina simultaneamente), em nível de sentença (executar duas ou mais sentenças de linguagem de alto nível simultaneamente), em nível de unidade (executar duas ou mais unidades de subprograma simultaneamente) e em nível de programa (executar dois ou mais programas simultaneamente). Como nenhuma questão de projeto de linguagem está envolvida nelas, as concorrências em nível de instrução e em nível de programa não serão discutidas. As concorrências em nível de subprograma e de sentença são discutidas, especialmente a primeira.

- 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:



Exemplo do fluxo de uma requisição e resposta para um servidor.
Fonte: PASQUINI, Gibson (2020).

Diversas arquiteturas de computadores têm mais de um processador e podem suportar alguma forma de execução concorrente. Os primeiros computadores com múltiplos processadores apresentavam um processador de propósito geral e um ou mais processadores, chamados de periféricos, usados apenas para operações de entrada e saída.

Entretanto, a situação tem mudado bastante. O fim da sequência de aumentos significativos na velocidade de processadores individuais está próximo. Aumentos significativos no poder da computação resultam agora do aumento no número de processadores, por exemplo, grandes sistemas de servidor, como aquele executado pelo Google e pela Amazon e por aplicações de pesquisa científica. Muitas outras tarefas de computação volumosas são agora executadas em máquinas com grande número de processadores relativamente pequenos.

Outro avanço recente no hardware de computação foi o desenvolvimento de vários processadores em um único chip, como os Intel Core Duo e Core Quad, que estão pressionando os desenvolvedores de software a usar mais os múltiplos processadores disponíveis nas máquinas. Se eles não fizerem isso, a concorrência em hardware será perdida e os ganhos de produtividade diminuirão significativamente.

- Categorias de concorrência

Existem duas categorias de controle de unidades concorrentes. A mais natural é a concorrência física, assumindo a disponibilidade de mais de um processador, diversas unidades do mesmo programa são executadas simultaneamente. Um ligeiro relaxamento desse conceito de concorrência 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.

Do ponto de vista do programador e do projetista de linguagem, concorrência lógica é o mesmo que concorrência física. É tarefa do implementador da linguagem, por meio das capacidades do sistema operacional, mapear a concorrência lógica no sistema de hardware hospedeiro. Tanto a concorrência lógica quanto a física permitem que o conceito de concorrência seja usado como uma metodologia de projeto de programas.

- Motivações para o uso da concorrência

Existem ao menos quatro razões para se implementar a programação concorrente em seus projetos:

1. **Velocidade de execução de programas em máquinas com múltiplos processadores -** essas máquinas fornecem uma maneira efetiva de aumentar a velocidade de execução dos programas, 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, um programa escrito para usar execução concorrente pode ser mais rápido que o mesmo programa escrito para execução sequencial.

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. Em muitos casos, o sistema simulado inclui mais de uma entidade, e elas fazem tudo simultaneamente, por exemplo, aeronaves voando em um espaço aéreo controlado. Software que usa concorrência deve ser utilizado para simular tais sistemas de forma precisa.

4. **Aplicações são distribuídas por várias máquinas, localmente ou pela Internet -** muitas máquinas, por exemplo, os carros, têm mais de um computador embarcado, cada um dedicado a alguma tarefa específica, isso implica na sincronização para execução desses programas. Agora a concorrência é usada em numerosas tarefas de computação diárias. 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: Introdução à concorrência em nível de subprograma

Uma **tarefa** (também chamadas de **processos**) é uma unidade de um programa, similar a um subprograma, que pode estar em execução concorrente com outras unidades do mesmo programa. Cada tarefa pode suportar uma linha de

execução de controle. Em algumas linguagens, por exemplo, Java e C#, certos métodos servem como tarefas. Tais métodos são executados em objetos chamados **linhas de execução**. Três características das tarefas as distinguem dos subprogramas:

1. Uma tarefa pode ser implicitamente iniciada, enquanto um subprograma precisa ser explicitamente chamado.

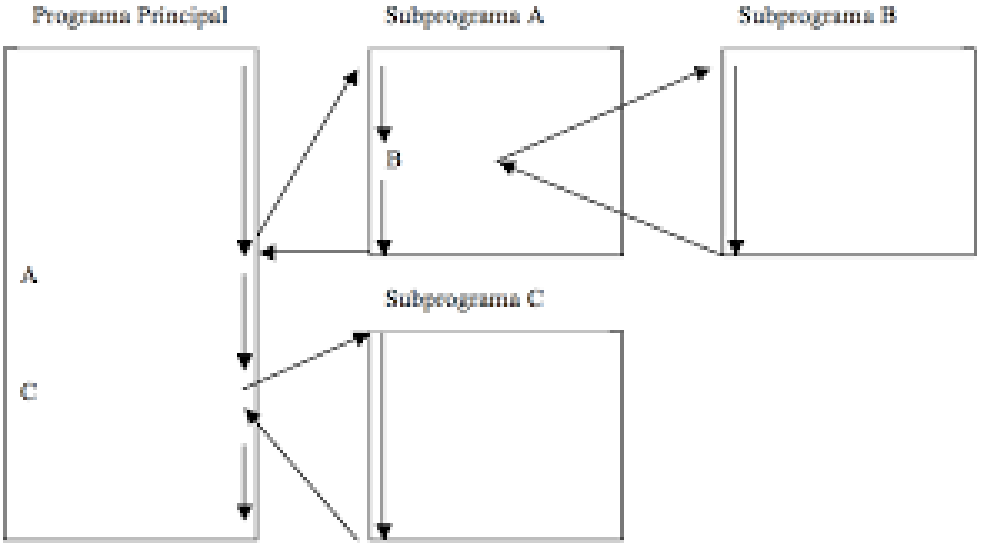
2. Quando uma unidade de programa invoca uma tarefa, em alguns casos ela não precisa esperar que a tarefa conclua sua execução antes de continuar a sua própria.

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

As tarefas são enquadradas em duas categorias gerais: **pesadas (heavyweight)** e **leves (lightweight)**. Simplificando, uma tarefa pesada executa em seu próprio espaço de endereçamento. Já as tarefas leves são todas executadas no mesmo espaço de endereçamento. É mais fácil implementar tarefas leves do que tarefas pesadas. Além disso, elas podem ser mais eficientes do que as pesadas, porque menos esforço é necessário para gerenciar sua execução.

Uma tarefa pode se comunicar com outras por meio de variáveis não locais compartilhadas, passagem de mensagens ou parâmetros. Se uma tarefa não se comunica com outra ou não afeta a execução de qualquer outra no programa, é chamada de **disjunta**.

A **sincronização** é um mecanismo que controla a ordem na qual as tarefas são executadas. Dois tipos de sincronização são necessários quando as tarefas compartilham dados: **cooperação e competição**. A sincronização de 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, sendo assim, talvez as tarefas precisem esperar o término do processamento específico do qual sua operação correta depende. A sincronização de competição é necessária entre duas tarefas quando ambas exigem o uso de algum recurso que não pode ser simultaneamente utilizado, as tarefas talvez precisem esperar o término de qualquer outro processamento, feito por qualquer outra tarefa, que atualmente esteja ocorrendo em dados compartilhados específicos.



Exemplo do fluxo de Subprogramas.
Fonte: GRESTA, Leonardo (2018).

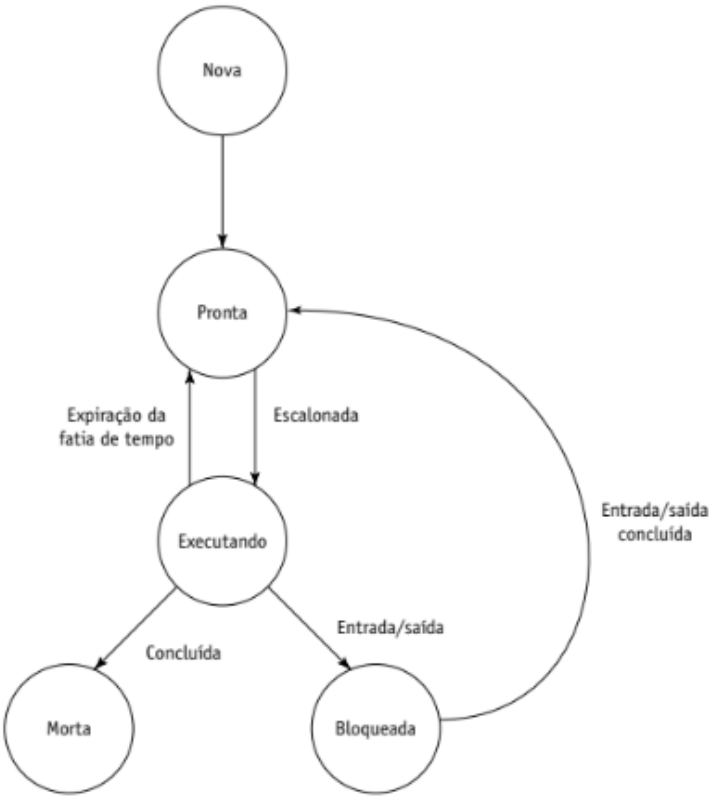
Um programa de sistema de tempo de execução, chamado **escalonador**, gerencia o compartilhamento de processadores entre as tarefas. Se nunca existissem interrupções e todas as tarefas tivessem a mesma prioridade, o escalonador poderia simplesmente dar a cada tarefa uma fatia de tempo, como 0,1 segundo, e quando o turno de uma tarefa chegasse, o escalonador poderia deixá-la ser executada por essa quantidade de tempo em um processador. É claro, existem diversos eventos complicadores, atrasos de tarefas para sincronização e para operações de entrada e saída. Como as operações de entrada e saída são muito lentas em relação à velocidade do processador, uma tarefa não pode manter um processador enquanto espera pela conclusão de uma delas.

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. Ou ainda não foi dado a ela tempo de processador pelo escalonador, ou ela já executou anteriormente, mas foi bloqueada. As tarefas prontas para serem executadas são armazenadas em uma fila chamada de fila de tarefas prontas.
3. **Executando:** uma tarefa que está sendo executada, ou seja, 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). Além de entrada e saída, algumas linguagens oferecem operações para que um programa de usuário especifique que uma tarefa deve ser bloqueada.

5. **Morta:** uma tarefa morta não está mais ativa em nenhum sentido, ou seja, sua execução está concluída, ou é morta explicitamente pelo programa.

Uma questão importante na execução de tarefas é a seguinte: como uma tarefa pronta é escolhida para ser movida para o estado “executando” quando a tarefa que está atualmente sendo executada se tornou bloqueada ou a fatia de tempo expirou? Diversos algoritmos diferentes têm sido usados para fazer essa escolha, alguns baseados em níveis especificados de prioridade. O algoritmo que faz a escolha é implementado no escalonador.



Exemplo do fluxo de Subprogramas.
Fonte: SEBESTA, Robert (2018).

No ambiente de programas sequenciais, um programa tem a característica de **vivacidade** se continua a executar, eventualmente sendo concluído. Em termos mais gerais, a vivacidade significa que, se um evento digamos, o término do programa supostamente ocorreria, ele ocorreria em um momento ou outro. Ou seja, o progresso é continuamente feito. Em um ambiente concorrente e com recursos compartilhados, a vivacidade de uma tarefa pode deixar de existir, isto é, o programa pode não continuar e, dessa forma, nunca terminará. Esse tipo de perda de vivacidade é chamado de impasse (deadlock). **Deadlock** é uma ameaça séria para a confiabilidade de um programa, dessa forma, evitá-lo demanda sérias considerações tanto no projeto de linguagens quanto no de programas.

- Projeto de linguagem para concorrência

Em alguns casos, a concorrência é implementada por meio de bibliotecas. Entre elas está a OpenMP, uma interface de programação de aplicações usada para suportar programação multiprocessada de memória compartilhada em C, C++ e Fortran, em uma variedade de plataformas. Diversas linguagens foram projetadas para suportar a concorrência, começando com PL/I, em meados dos anos 1960, e incluindo as linguagens contemporâneas como Java, C#Python e Ruby.

2. SEMÁFOROS

Um **semáforo** é um mecanismo simples que pode ser usado para fornecer sincronização de tarefas. Embora os semáforos sejam uma estratégia primitiva para fornecer sincronização, eles ainda são utilizados tanto em linguagens contemporâneas como em sistemas com suporte à concorrência baseado em biblioteca.

Os semáforos também podem ser usados para prover sincronização de cooperação. 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. Assim, uma guarda pode ser usada para permitir que apenas uma tarefa acesse uma estrutura de dados específica compartilhada em dado momento. 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. Um descritor de tarefa é uma estrutura de dados que armazena todas as informações relevantes acerca do estado de execução de uma tarefa. Uma parte integrante de um mecanismo de guarda é um procedimento usado para garantir que todas as execuções tentadas do código de guarda ocorram em algum momento. 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 deve ter alguma maneira de gravar tanto o número de posições vazias quanto o de posições preenchidas. O componente de contagem de um semáforo 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 usado por produtores e consumidores, e outra pode usar seu contador para

manter o número de posições preenchidas no buffer. As filas desses semáforos podem armazenar os descritores de tarefas forçadas a esperar pelo acesso ao buffer. 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.

Nosso buffer de exemplo é projetado como um tipo de dados abstrato no qual todos os dados entram por meio do subprograma **DEPOSIT** e todos deixam o buffer por meio do subprograma **FETCH**. O subprograma DEPOSIT precisa apenas verificar o semáforo emptyspots para ver se existe alguma posição vazia. Se existir ao menos uma, ele pode continuar com DEPOSIT, o que deve ter o efeito colateral de decrementar o contador de emptyspots. Se o buffer estiver cheio, o chamador de DEPOSIT deve esperar na fila de emptyspots até que uma posição vaga seja disponibilizada. Quando DEPOSIT tiver concluído sua tarefa, o subprograma DEPOSIT incrementa o contador do semáforo fullspots para indicar que existe mais uma posição preenchida no buffer.

O subprograma FETCH tem a sequência oposta de DEPOSIT. Ele verifica o semáforo fullspot para ver se o buffer contém ao menos um item. Se contém, um item é removido e o semáforo emptyspots tem seu contador incrementado por 1. Se o buffer estiver vazio, a tarefa que chama é colocada na fila de fullspots para esperar até que um item apareça. Quando o FETCH termina, ele precisa incrementar o contador de emptyspots.

As operações em tipos semáforos geralmente não são diretas – elas são feitas por meio dos subprogramas wait e release. Logo, a operação DEPOSIT descrita anteriormente é, na verdade, realizada em parte por chamadas a wait e a release. Wait e release devem ser capazes de acessar a fila de tarefas prontas.

O subprograma de semáforo **wait** é usado para testar o contador de uma variável semáforo. Se o valor for maior que zero, o chamador pode continuar sua operação. Nesse caso, o valor do contador da variável semáforo é decrementado para indicar que agora há um item a menos daquilo que o semáforo estiver contando. Se o valor do contador é zero, o chamador deve ser colocado na fila de espera da variável semáforo e outra tarefa pronta deve ser dada ao processador.

O subprograma de semáforo release é usado por uma tarefa para permitir a outra ter um item do contador da variável de semáforo especificado, independentemente do item que tal variável estiver contando. Se a fila da variável semáforo especificada estiver vazia, ou seja, se nenhuma tarefa está esperando, release incrementa seu contador (para indicar a existência de mais um item sendo controlado e agora disponível). Se uma ou mais tarefas estão esperando, release move uma delas da fila do semáforo para a fila de tarefas prontas.

- 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. No exemplo do buffer, omitir a sentença wait (emptyspots) da tarefa producer resultaria em um transbordamento do buffer. Omitir wait (fullspots) da tarefa consumer resultaria em um transbordamento negativo do buffer. Omitir qualquer uma das liberações resultaria em um impasse. Essas são 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.

3. MONITORES

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.

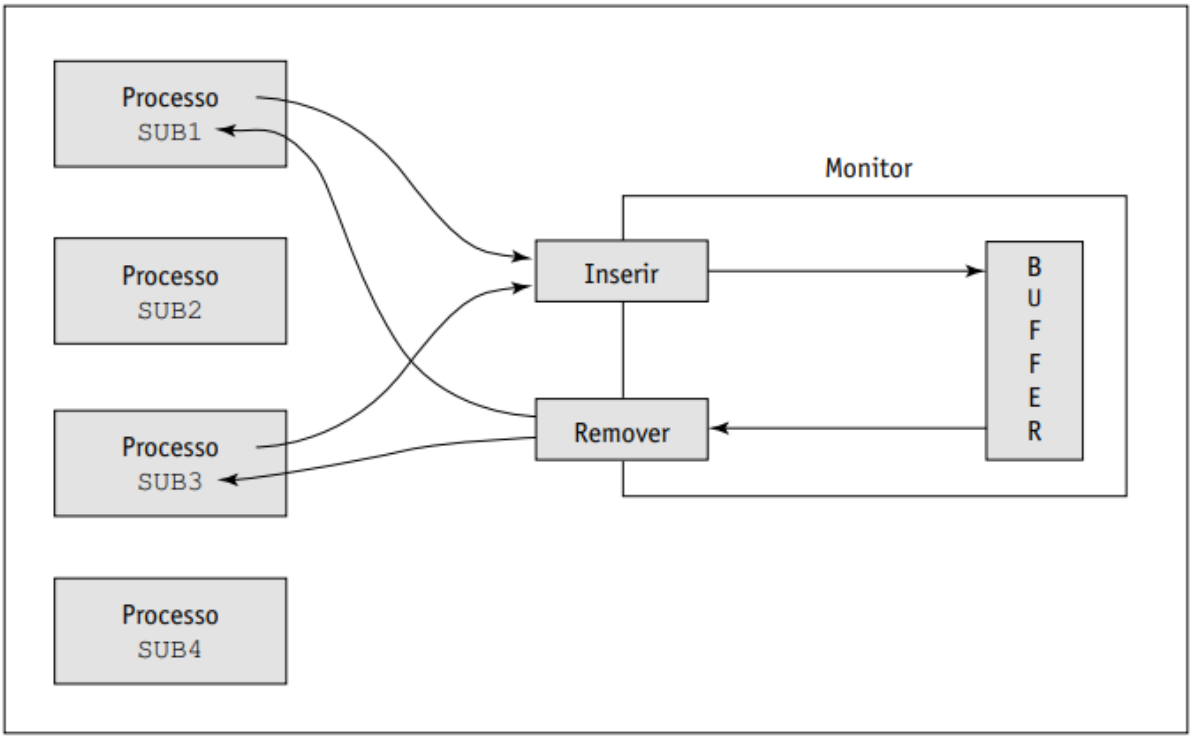
A primeira linguagem de programação a incorporar monitores foi Pascal Concorrente (Brinch Hansen, 1975). Modula (Wirth, 1977), CSP/k (Holt et al., 1978) e Mesa (Mitchell et al., 1979) também fornecem monitores. Entre as linguagens contemporâneas, eles são suportados por Ada, Java e C#.

- 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 se ele estiver ocupado no momento da chamada.

- Sincronização de Cooperação

Apesar de o 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.



Exemplo de programa usando monitor para controlar acesso a um buffer compartilhado.
Fonte: SEBESTA, Robert (2018).

- 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, o que ficará claro quando as implementações de monitores em Ada e em Java forem discutidas nas seções seguintes.

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 passagem de mensagens 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.

4. PASSAGEM DE MENSAGENS

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.

- O conceito de 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. Suponha que as tarefas A e B estejam em execução e que A deseja enviar uma mensagem para B. Se B estiver ocupada, não é desejável permitir que outra tarefa a interrompa. Isso impediria seu processamento atual. Além disso, as mensagens normalmente causam processamento associado no receptor, o que não seria desejado se outro processamento estivesse incompleto. A alternativa é fornecer um mecanismo linguístico capaz de permitir a uma tarefa especificar para outras quando ela está pronta para receber mensagens. Essa estratégia é, de alguma forma, parecida com a de um executivo que instrui sua secretária a colocar em espera todas as chamadas até que outra atividade, talvez uma conversa importante, tenha terminado. Posteriormente, quando a conversa atual estiver concluída, o executivo dirá à secretária que aceita conversar com uma das pessoas colocadas em espera.

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, se uma tarefa A está esperando por uma mensagem no momento em que B a envia, a mensagem pode ser transmitida. Essa transmissão real da mensagem é chamada de rendezvous. Note que um rendezvous pode ocorrer apenas se tanto o remetente quanto o destinatário quiserem que ele aconteça. 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.

5. SUPORTE DE ADA PARA CONCORRÊNCIA

As tarefas Ada podem ser mais ativas que os monitores. Estas são entidades passivas que fornecem serviços de gerenciamento para os dados compartilhados que armazenam. Eles fornecem seus serviços, apesar de o fazerem apenas quando são solicitados. Quando usadas para gerenciar dados compartilhados, as tarefas Ada funcionam como gerentes que podem residir com os recursos que gerenciam. Elas têm diversos mecanismos, alguns determinísticos e outros não determinísticos, 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 –, ambas com o mesmo nome. A interface de uma tarefa são seus pontos de entrada, ou posições onde ela pode receber mensagens de outras tarefas. Como esses pontos de entrada integram sua interface, é natural que sejam listados na parte de especificação de uma tarefa. Como um rendezvous pode envolver uma troca de informações, as mensagens podem ter parâmetros; logo, os pontos de entradas de tarefas ainda devem permitir parâmetros, os quais também devem ser descritos na parte de especificação. Na aparência, uma especificação de tarefa é similar ao pacote de especificação para um tipo de dados abstrato.

O corpo de uma tarefa deve incluir alguma forma sintática dos pontos de entrada que correspondem às cláusulas entry na parte de especificação da tarefa. Em Ada, esses pontos de entrada no corpo de tarefas são especificados por cláusulas introduzidas pela palavra reservada accept. Uma cláusula accept é definida na faixa de sentenças que começam com a palavra reservada accept e terminam com a palavra reservada end correspondente. As cláusulas accept em si são relativamente simples, mas outras construções nas quais elas podem ser incorporadas podem tornar sua semântica complexa.

O nome de entrada em accept casa com o nome em uma cláusula entry na parte de especificação da tarefa associada. Os parâmetros opcionais fornecem as maneiras de comunicação de dados entre a tarefa chamadora e a chamada. As sentenças entre o do e o end definem as operações que ocorrem durante o rendezvous. Juntas, essas sentenças são chamadas de corpo da cláusula accept. Durante o rendezvous real, a tarefa remetente é suspensa.

Sempre que uma cláusula accept receber uma mensagem que não está pronta para aceitar por qualquer razão, a tarefa remetente deve ser suspensa até que a cláusula accept na tarefa receptora esteja pronta para aceitar a mensagem. É claro, a cláusula accept deve também lembrar das tarefas que enviaram mensagens não aceitas. Para esse propósito, cada cláusula accept em uma tarefa possui uma fila associada para armazenar uma lista de outras tarefas que tentaram se comunicar com ela sem sucesso.

Uma tarefa Ada que envia uma mensagem para outra deve conhecer o nome de entrada na tarefa. Contudo, o oposto não é verdade: uma entrada de tarefa não precisa conhecer o nome das tarefas das quais ela receberá mensagens. Essa assimetria está em contraste com o projeto da linguagem conhecida como Processos Sequenciais Comunicantes (CSP - Communicating Sequential Processes) (Hoare, 1978). Nela, que também usa o modelo de concorrência de passagem de mensagens, as tarefas aceitam mensagens apenas de outras explicitamente nomeadas. A desvantagem desse projeto é que não podem ser construídas bibliotecas de tarefas para uso geral.

As tarefas são declaradas na parte de declaração de um pacote, subprograma ou bloco. Tarefas criadas estaticamente começam sua execução no mesmo momento das sentenças no código ao qual a parte declarativa está anexada. Por exemplo, uma tarefa declarada em um programa principal inicia sua execução no mesmo momento em que a primeira sentença no corpo de código do programa principal. O término de uma tarefa, que é uma questão complexa, é discutido posteriormente nesta seção.

As tarefas podem ter qualquer quantidade de entradas. A ordem na qual as cláusulas accept associadas aparecem na tarefa determina a ordem na qual as mensagens podem ser aceitas. Se uma tarefa tem mais de um ponto de entrada e requer que eles sejam capazes de receber mensagens em qualquer ordem, ela usa uma sentença select para envolver as entradas

- Sincronização de Cooperação

Cada cláusula accept pode ter uma guarda anexada, na forma de uma cláusula when, que pode postergar um rendezvous. Uma cláusula accept com uma when pode estar aberta ou fechada. Se a expressão booleana da cláusula when é verdadeira, a accept é chamada de aberta (open); se a expressão booleana é falsa, a accept é dita fechada (closed). Uma cláusula accept que não tem uma guarda é sempre aberta. Uma cláusula aberta está disponível para rendezvous; uma fechada não pode realizar um rendezvous

Suponha que existam diversas cláusulas accept guardadas em uma cláusula select. Tal cláusula select normalmente é colocada em um laço infinito. O laço faz com que a cláusula select seja executada repetidamente, com cada cláusula when avaliada em cada repetição. Cada repetição faz uma lista de cláusulas accept abertas ser construída. Se precisamente uma das cláusulas abertas tiver uma fila não vazia, uma mensagem dessa fila é retirada e um rendezvous ocorre. Se mais de uma das cláusulas accept possuem filas não vazias, uma das filas é escolhida não deterministicamente, uma mensagem é retirada dela e um rendezvous ocorre. Se as filas de todas as cláusulas abertas estiverem vazias, a tarefa espera pela chegada de uma mensagem em uma das cláusulas accept, fazendo um rendezvous ocorrer no momento da chegada. Se um select for executado e todas as cláusulas accept estiverem fechadas, ocorrerá um erro ou exceção em tempo de execução. Essa possibilidade pode ser evitada certificando-se de que uma das cláusulas when é sempre verdadeira ou adicionando-se uma cláusula else no select. Uma cláusula else pode incluir qualquer sequência de sentenças, exceto uma cláusula accept.

Uma cláusula select pode ter uma sentença especial, terminate, selecionada apenas quando ela está aberta e nenhuma outra cláusula accept está. Uma cláusula terminate, quando selecionada, significa que o trabalho da tarefa está finalizado, mas ainda não terminado

- Sincronização de Compteição

Se o acesso a uma estrutura de dados deve ser controlado por uma tarefa, o acesso mutuamente exclusivo pode ser obtido declarando-se a estrutura de dados dentro de uma tarefa. A semântica da execução de tarefa normalmente garante acesso mutuamente exclusivo à estrutura, porque apenas uma cláusula accept na tarefa pode estar ativa em dado momento. As únicas exceções para isso ocorrem quando as tarefas são aninhadas em procedimentos ou em outras tarefas. Por exemplo, se uma tarefa que define uma estrutura de dados compartilhada tem uma tarefa aninhada, esta também pode acessar a

estrutura compartilhada, o que poderia destruir a integridade dos dados. Logo, as tarefas que supostamente devem controlar o acesso a uma estrutura de dados compartilhada não devem definir tarefas.

- 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 de forma eficiente. Os objetos protegidos de Ada 95 oferecem um método alternativo de fornecer sincronização de competição que não envolve o mecanismo de rendezvous.

Objetos protegidos podem ser acessados por subprogramas protegidos ou por entradas sintaticamente semelhantes às cláusulas accept nas tarefas. Os subprogramas protegidos podem ser procedimentos protegidos, os quais fornecem acesso de leitura e escrita mutuamente exclusivo para os dados do objeto protegido, ou funções protegidas, as quais fornecem acesso concorrente somente à leitura para esses dados. As entradas diferem dos subprogramas protegidos porque podem ter guardas.

Dentro do corpo de um procedimento protegido, a instância atual da unidade protegida envolvida é definida como uma variável; dentro do corpo de uma função protegida, a instância atual da unidade protegida envolvida é definida como uma constante, permitindo acessos concorrentes do tipo somente leitura.

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

Usar o modelo geral de passagem de mensagens de concorrência para construir monitores é como usar os pacotes Ada para suportar tipos de dados abstratos, ambos são ferramentas mais gerais do que o necessário. Objetos protegidos são uma maneira melhor de fornecer acesso sincronizado a dados compartilhados.

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.

6. LINHAS DE EXECUÇÃO EM JAVA

A linguagem Java utiliza a Interface Runnable e a classe Thread para possibilitar a execução de mais de um fluxo de controle sobre o código de programa. Aqui temos um exemplo simples para ilustrar um dos métodos de criação de múltiplos fluxos de execução (o outro método consiste em estender a classe Thread).

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. Resumindo:

```
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(). */

  }

}
```

No programa abaixo é utilizada uma interface padrão denominada Runnable. Uma classe que implementa a interface Runnable deve implementar um método com a seguinte assinatura: public void run(). O programa utiliza ainda uma classe denominada Thread. Um

dos construtores da classe Thread recebe como parâmetro uma referência para um objeto cuja classe implementa a interface Runnable. Quando invocamos o método start() de uma instância de Thread o fluxo de controle é duplicado e um dos fluxos de controle segue pelo método run() do objeto Runnable associado e o outro fluxo de controle retorna para o ponto onde o método start() foi invocado. A menos dos aspectos denominados de *sincronização*, não existem garantias sobre a velocidade relativa de execução dos fluxos de controle. Enquanto existirem fluxos de controle a execução de um programa Java contínua, ou seja, o fluxo de controle que percorre o método main pode terminar e o programa continua em execução desde que exista um ou mais fluxos de controle. No programa abaixo após o término do fluxo de controle sobre o método main permanecerão dois fluxos de controle sobre as duas instâncias de Thread correspondentes a duas instâncias de tarefa:

```
class Tarefa implements Runnable{

  private String s;

  Tarefa(String id){

        s=id;

  }

  public void run(){

        for(;;) System.out.print(s+" ");

  }

}

class linhaSimples{

  public static void main(String[] args){

        Runnable tarefa1=new Tarefa("1");

        Runnable tarefa2=new Tarefa("2");

        Thread trabalhador1=new Thread(tarefa1);

        Thread trabalhador2=new Thread(tarefa2);

        trabalhador1.start();

        trabalhador2.start();

  }

}
```

No java, os semáforos são implementados através da classe Semaphore, disponível no pacote java.util.concurrent.Semaphore. O construtor básico de Semaphore recebe um parâmetro inteiro, o qual inicializa o contador do semáforo. Por exemplo, o seguinte poderia ser usado para inicializar os semáforos fullspots e emptyspots para o exemplo de buffer:

```
fullspots = new Semaphore(0);

emptyspots = new Semaphore(BUFLEN);
```

A operação deposit do método producer apareceria como segue:

```
emptyspots.acquire();

deposit(value); fullspots.release();
```

Do mesmo modo, a operação fetch do método consumer apareceria como segue:

```
fullspots.acquire();

fetch(value);

emptyspots.release();
```

7. LINHAS DE EXECUÇÃO EM C#

Em vez de apenas métodos chamados run, como em Java, qualquer método C# pode executar em sua própria linha de execução. Quando são criadas linhas de execução C#, elas 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. Assim, a execução de uma linha é controlada por meio de seu representante associado. Uma linha de execução C# é criada por meio de um objeto Thread. O construtor de Thread deve receber uma instanciação de ThreadStart, à qual deve ser enviado o nome do método a ser executado na linha de execução. Por exemplo, podemos ter:

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

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

Nesse exemplo, criamos 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.

Assim como em Java, em C# existem duas categorias de linhas de execução: atrizes e servidoras. As linhas de execução atrizes não são chamadas especificamente; em

vez disso, são iniciadas. Além disso, os métodos que executam não recebem parâmetros nem valores de retorno. Como em Java, criar uma linha de execução não inicia sua execução concorrente. Para linhas de execução atrizes, a execução deve ser solicitada por meio de um método da classe Thread, nesse caso chamado Start, como em myThread.Start();

Como em Java, é possível fazer uma linha de execução esperar pelo término da execução de outra linha de execução antes de continuar usando-se o método similarmente chamado de Join. Por exemplo, suponha que a linha de execução A tenha a seguinte chamada:

B.Join();

A linha de execução A será bloqueada até que a linha de execução B termine. O método Join pode receber um parâmetro **int**, o qual especifica um limite de tempo, em milissegundos, que o chamador esperará para que a linha de execução termine. Uma linha de execução pode ser suspensa por uma quantidade de tempo específica por meio do Sleep, um método estático público de Thread. O parâmetro para Sleep é um número inteiro de milissegundos.

Uma linha de execução pode ser finalizada com o método Abort, apesar de ele não a matar. Em vez disso, ele lança a exceção ThreadAbortException, que a linha de execução pode capturar. Quando ocorre a captura, a linha de execução normalmente libera qualquer recurso alocado e então termina (ao chegar ao final de seu código).

Uma linha de execução servidora só pode ser executada quando chamada por meio de seu representante. Essas linhas de execução são denominadas servidoras porque fornecem algum serviço quando requisitadas. Tais chamadas podem ser feitas tratando-se o objeto representante como se fosse o nome do método. Na verdade, essa era uma abreviação para uma chamada a um método representante denominado Invoke. Assim, se o nome de um objeto representante fosse chgfun1 e o método referenciado por ele recebesse um parâmetro **int**, poderíamos chamar esse método com uma das seguintes sentenças:

chgfun1(7);

chgfun1.Invoke(7);

Essas chamadas são síncronas, isto é, quando o método é chamado, o chamador é bloqueado até que o método finalize sua execução. C# também suporta chamadas assíncronas a métodos que executam em linhas de execução. Quando uma linha de execução é chamada assincronamente, ela e a linha de execução chamadora executam de forma concorrente, pois a chamadora não é bloqueada durante a execução da linha. Uma linha de execução é chamada de forma assíncrona por meio do método de instância representante BeginInvoke, ao qual são enviados os parâmetros para o método do representante, junto com dois parâmetros adicionais, um de tipo AsyncCallback e outro de tipo **object**. BeginInvoke retorna um objeto que implementa a interface IAsyncResult. A classe representante também define o método de instância EndInvoke, o qual recebe um parâmetro de tipo IAsyncResult e retorna o mesmo tipo retornado pelo método encapsulado

no objeto representante. Para chamar uma linha de execução de forma assíncrona, a chamamos com BeginInvoke.

```
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 (de tipo IAsyncResult) retornado por BeginInvoke. EndInvoke retorna o valor de retorno da linha de execução chamada.

Se o chamador precisa continuar algum trabalho enquanto a linha de execução é chamada executa, ele deve ter um meio de determinar quando ela terminou. Para isso, a interface IAsyncResult define a propriedade IsCompleted. Enquanto a linha de execução chamada executa, o chamador pode incluir código que ela pode executar em um laço **while** que dependa de IsCompleted.

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

8. CONCORRÊNCIA EM LINGUAGENS FUNCIONAIS

Multi-LISP

Multi-LISP é uma extensão de Scheme que permite ao programador especificar partes de um programa que podem ser executadas concorrentemente. Essas formas de concorrência são implícitas; o programador simplesmente informa ao compilador sobre algumas partes do programa que podem ser executadas concorrentemente. O Programador indica através da construção pcall. Se uma chamada de função é incorporada a uma construção pcall, os parâmetros da função podem ser avaliados concorrentemente. Por exemplo, considere a seguinte construção pcall:

```
(pcall f a b c d)
```

A função é f, com parâmetros a, b, c e d. O pcall faz com que os parâmetros da função possam ser avaliados concorrentemente. O programador é responsável por fazer com que esse processo seja usado com segurança, sem afetar a semântica da avaliação da função. Na verdade, isso é uma questão simples se a linguagem não permite efeitos colaterais, ou se o programador projetou a função de modo a não tê-los, ou, pelo menos, a tê-los de forma limitada. Contudo, Multi-LISP permite alguns efeitos colaterais. Se a função não foi escrita para evitá-los, pode ser difícil para o programador determinar se pcall pode ser usada com segurança.

A construção future de Multi-LISP é uma fonte de concorrência mais interessante e potencialmente mais produtiva. Como em pcall, uma chamada de função é empacotada em uma construção future. Tal função é avaliada em uma linha de execução separada, com a linha de execução pai continuando sua execução até que precise usar o valor de retorno da função.

Se a função não tiver concluído sua execução quando seu resultado for necessário, a linha de execução pai esperará até que ela termine, antes de continuar. Se uma função tem dois ou mais parâmetros, eles também podem ser empacotados em construções future, no caso em que suas avaliações poderão ser feitas concorrentemente em linhas de execução separadas.

F#

Como o F# faz parte da plataforma .NET, a concorrência em F# é baseada nas mesmas classes utilizadas por C#, especificamente System.Threading.Thread. Por exemplo, 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()
```

Em C#, para criar uma instância de um representante predefinido, ThreadStart, é necessário enviar o nome do subprograma ao seu construtor e enviar a nova instância do representante como parâmetro para o construtor de Thread. Em F#, se uma função espera um representante como parâmetro, uma expressão ou função lambda pode ser enviada e o compilador se comportará como se você tivesse enviado o representante. Assim, no código acima, a função myConMethod é enviada como parâmetro para o construtor de Thread, mas o que é realmente enviado é uma nova instância de ThreadStart (para a qual myConMethod foi enviada). Além disso, a classe Thread também define o método Sleep, o qual faz dormir, pelo número de milissegundos enviado a ele como parâmetro, a linha de execução a partir da qual foi chamado.

Dados imutáveis compartilhados não exigem sincronização entre as linhas de execução que os acessam. No entanto, se os dados compartilhados forem mutáveis, o que é possível em F#, será exigido um cadeado para evitar sua corrupção por parte das várias

linhas de execução que tentam alterá-los. Uma variável mutável pode ser trancada enquanto uma função opera nela, a fim de fornecer acesso sincronizado ao objeto com a função lock. Essa função recebe dois parâmetros, o primeiro dos quais é a variável a ser alterada. O segundo é a expressão lambda que altera a variável.

Uma variável mutável alocada no monte é de tipo ref. Por exemplo, a seguinte declaração cria uma variável assim, chamada sum, com o valor inicial 0:

```
let sum = ref 0
```

Uma variável tipo ref pode ser alterada em uma expressão lambda que utilize o operador de atribuição ALGOL/Pascal/Ada, :=. A variável ref deve ser prefixada com um ponto de exclamação (!) para receber seu valor. No seguinte, a variável mutável sum é trancada enquanto a expressão lambda adiciona o valor de x a ela:

```
lock(sum) (fun () -> sum := !sum + x)
```

É possível executar instruções de forma assíncrona, por meio dos metodos, BeginInvoke e EndInvoke, assim como a interface IAsyncResult, ambos presentes em C#, para facilitar a determinação da conclusão da execução da linha chamada de forma assíncrona.

9. CONCORRÊNCIA EM NÍVEL DE SENTENÇA

Fortran de alto desempenho

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 principais sentenças de especificação de HPF 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.

A sentença DISTRIBUTE especifica quais dados serão distribuídos e o tipo de distribuição a ser usada. Sua forma é a seguinte:

!HPF$ DISTRIBUTE (tipo) ONTO procs :: lista_de_identificadores

Na instrução, o tipo pode ser bloco (BLOCK) ou cíclica (CYCLIC) e a lista de identificadores são os nomes das variáveis de matrizes que serão distribuídas. Uma variável especificada para ser distribuída em bloco (BLOCK) é dividida em *n* grupos iguais; cada um consiste em coleções contíguas de elementos de matriz igualmente distribuídos nas memórias de todos os processadores. Por exemplo, se uma matriz com 200 elementos chamada MATRIZ é distribuída em bloco em cinco processadores, seus primeiros 40 elementos serão armazenados na memória do primeiro processador, outros 40 na do segundo e assim consecutivamente. Uma distribuição cíclica (CYCLIC) especifica quais elementos individuais da matriz são ciclicamente armazenados nas memórias dos processadores. Nesse caso, se MATRIZ possui 10 processadores, o primeiro elemento será armazenado na memória do primeiro processador, o segundo elemento na memória do segundo processador e sucessivamente.

A forma da sentença ALIGN é

ALIGN elemento_matriz1 WITH elemento_matriz2

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

Essa sentença é usada para relacionar a distribuição de uma matriz com a de outra. Desse modo, especifica que o elemento index de lista1 será armazenado na memória do mesmo processador que o elemento index+1 de lista2 para todos os valores de index.

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
```

No exemplo, a instrução especifica a atribuição dos elementos de list_2 aos elementos correspondentes de list_1. Porém, deve se avaliar o lado direito de todas as 1.000 atribuições primeiro, antes que quaisquer atribuições ocorram e esse processo permite a execução concorrente de todas as sentenças de atribuição.

Essas foram apenas algumas das sentenças do HPF, existem várias outras que são usadas para as mais diversas finalidades.

10. 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, aceleração de aplicativos essenciais para 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.

O presente trabalho foi realmente uma introdução sobre o assunto. Há muito conhecimento ainda para ser estudado quando falamos em concorrência com múltiplos threads ou concorrência com múltiplos processos. Esses assuntos são extensos e poderiam render bem mais conteúdo, porém acreditamos que a melhor forma de entendê-los, é realmente aplicá-los usando linguagens e frameworks.

11. 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.