GABRIEL VIANA POLETTI

GABRIEL ANGELO PEREZ GASPARINI SABAUDO

GUILHERME HENRIQUE GONÇALVES SILVA

JOSÉ VICTOR ANDREI DALTO MORENO

# CONCORRÊNCIA

## Programação Concorrente

**LONDRINA – PR**

**2020**

GABRIEL VIANA POLETTI

GABRIEL ANGELO PEREZ GASPARINI SABAUDO

GUILHERME HENRIQUE GONÇALVES SILVA

JOSÉ VICTOR ANDREI DALTO MORENO

# CONCORRÊNCIA

Trabalho solicitado pela professora Cinthyan da disciplina Paradigmas de Computação, do curso de Ciência de Computação, turno integral da instituição Universidade Estadual de Londrina.

**LONDRINA – PR**

**2020**

**RESUMO**

Esta é a história de uma das maiores revoluções na programação de computadores: a invenção da programação simultânea. Tom Kilburn e David Howarth foram os pioneiros no uso de interrupções para simular a execução simultânea de vários programas no computador Atlas (Kilburn 1961). Essa técnica de programação ficou conhecida como multiprogramação. Os primeiros sistemas de multiprogramação foram programados em linguagem assembly sem qualquer base conceitual. O menor erro de programação pode fazer com que esses sistemas se comportem de uma maneira completamente errática que tornou o teste do programa quase impossível. É nesta época em que o conceito de programação concorrente começa a aparecer.

**ABSTRACT**

This is the story of one of the major revolutions in computer programming: the invention of concurrent programming. Tom Kilburn and David Howarth pioneered the use of interrupts to simulate concurrent execution of several programs on the Atlas computer (Kilburn, 1961). This programming technique became known as multiprogramming. The early multiprogramming systems were programmed in assembly language without any conceptual foundation. The slightest programming mistake could make these systems behave in a completely erratic manner that made program testing nearly impossible. It was at that time, that the concept of competitive programming began to shows up.

Sumário

## 1 INTRODUÇÃO

Começa com introduções aos vários tipos de concorrência no nível de subprogramas ou de unidades, e no nível de sentenças. É incluída uma breve descrição dos tipos comuns de arquiteturas multiprocessadas de computadores. A seguir, é apresentada uma extensa discussão sobre concorrência no nível de unidade.

Ela começa com uma descrição dos conceitos fundamentais que devem ser entendidos antes de discutirmos os problemas e desafios do suporte linguístico para a concorrência no nível de unidade, especificamente a sincronização de competição e a de cooperação. A seguir, são descritas as questões de projeto para fornecer suporte linguístico para concorrência. Posteriormente, segue uma discussão detalhada, incluindo exemplos de código, das principais abordagens para o suporte linguístico para concorrência: semáforos, monitores.

## 2 PROGRAMAÇÃO CONCORRENTE

Tradicionalmente, a grande maioria dos programas escritos são programas sequenciais, para serem executados em um único computador por um único processo. O problema é dividido em uma série de instruções que são executadas uma após a outra. Nesse caso, existe somente um fluxo de controle (fluxo de execução, linha de execução, thread) no programa. Isso permite, por exemplo que o programador realize uma “execução imaginária” de seu programa apontando com o dedo, a cada instante, o comando que está sendo executado no momento.

De acordo com o D.r Eduardo Alchieri, um programa concorrente (do inglês Concurrent Programming) pode ser visto como se tivesse vários fluxos de execução. Para o programador realizar agora uma “execução imaginaria”, ele vai precisar de vários dedos, um para cada fluxo de controle. Em programação concorrente é definido o uso simultâneo de múltiplos recursos computacionais para resolver um problema. Para ser executado por diversos processos, o problema é quebrado (ou é naturalmente formato por partes) em partes que podem ser executadas (resolvidas concorrentemente. Cada uma destas partes é representada por uma série de instrução, sendo que as instruções de cada parte são executadas concorrentemente em diferentes processos.

Um programa é considerado concorrente quando ele (o próprio programa, durante a sua execução) origina diferentes processos que irão interagir entre si para realizar alguma tarefa.

É comum em sistemas multiusuários que um mesmo programa seja executado simultaneamente por vários usuários. Por exemplo, um editor de texto. Entretanto, ter 10 execuções simultâneas do editor de texto não faz dele um programa concorrente. O que se tem são 10 processos independentes executando o mesmo programa sequencial (compartilhando o mesmo código). Cada processo tem a sua área de dados e ignora a existência das outras execuções do programa.

À primeira vista, a concorrência pode parecer um conceito simples, mas ela apresenta um desafio significativo para o projetista de linguagens de programação.

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

- Nível de instrução (executando duas ou mais instruções de máquina simultaneamente).
- Nível de sentença ou comando (executando duas ou mais sentenças na linguagem fonte simultaneamente).
- Nível de unidade (executando duas ou mais unidades de subprograma simultaneamente).
- Nível de programa (executando dois ou mais programas simultaneamente).

A execução concorrente de subprogramas pode ocorrer ou fisicamente, em processadores separados, ou logicamente, compartilhando-se um único processador. À primeira vista, a concorrência pode parecer um conceito simples, mas ela apresenta um desafio significativo para o projetista de linguagens de programação.

Mecanismos de controle de concorrência aumentam a flexibilidade da programação. Eles foram originalmente inventados para serem usados em problemas particulares oriundos dos sistemas operacionais, mas eles são requeridos por uma variedade de outras aplicações de programação.

Alguns dos programas mais usados atualmente são os navegadores Web, cujo projeto é fortemente baseado em concorrência. Os navegadores devem realizar muitas funções diferentes ao mesmo tempo, dentre elas enviar e receber dados de servidores Web, desenhar texto e imagens na tela e reagir às ações do usuário com o mouse e com o teclado (Sebesta, 2011).

## 2.1 Arquiteturas multiprocessadas

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. Essa arquitetura permitiu a esses computadores, surgidos no final dos anos 1950, executar um programa enquanto realizavam entrada ou saída para outros programas.

Em meados dos anos 1960, apareceram máquinas com diversos processadores parciais idênticos alimentados com certas instruções a partir de um único fluxo de instruções. Por exemplo, algumas máquinas possuíam dois ou mais multiplicadores de ponto flutuante, enquanto outras tinham duas ou mais unidades aritméticas de ponto flutuante completas. Os compiladores para essas máquinas precisavam determinar quais instruções poderiam ser executadas concorrentemente e precisavam agendar essas instruções de forma correta. Sistemas com essa estrutura suportavam concorrência no nível de instruções (Sebesta, 2011).

Em 1966, Michael J. Flynn sugeriu uma categorização das arquiteturas de computadores definida em relação aos fluxos de dados e de instruções, que poderiam ser simples ou múltiplos. Os nomes dessas categorias foram bastante usados desde os anos 1970 até o início dos 2000. As duas categorias que usavam múltiplos fluxos de dados eram definidas como segue: computadores com múltiplos processadores que executam a mesma instrução simultaneamente, cada um com dados diferentes, são chamados de arquiteturas de computadores SIMD – Single-Instruction Multiple Data (Única Instrução Múltiplos Dados). Computadores SIMD (Single Instruction Multiple Data) são utilizados para a resolução de problemas computacionalmente intensivos da área científica e de engenharia, em que existem estruturas de dados regulares como vetores e matrizes. As duas principais formas de máquinas SIMD são os processadores vetoriais. Elas têm grupos de registros

que armazenam os operandos de uma operação vetorial na qual a mesma instrução é executada simultaneamente no conjunto inteiro de operandos. Até pouco tempo, a maioria dos supercomputadores eram processadores vetoriais.

Computadores com múltiplos processadores que operam independentemente, mas cujas operações podem ser sincronizadas, são chamados de MIMD – Multiple-Instruction Multiple-Data (Múltiplas Instruções Múltiplos Dados). Cada processador em um computador MIMD executa seu próprio fluxo de instruções. Computadores MIMD podem aparecer em duas configurações distintas: sistemas de memória compartilhados e distribuídos.

As máquinas MIMD distribuídas, nas quais cada processador tem sua própria memória, podem ser construídas tanto em um chassi quanto distribuídas, talvez em uma grande área. As máquinas MIMD de memória compartilhada obviamente devem fornecer alguma forma de sincronização para prevenir colisões de acesso à memória. Mesmo máquinas MIMD distribuídas requerem sincronização para operarem juntas em programas únicos. Computadores MIMD, mais caros e mais gerais do que computadores SIMD, suportam concorrência no nível de unidade.

Uma das motivações mais fortes para usar máquinas concorrentes é aumentar a velocidade de computação. Entretanto, dois fatores de hardware foram combinados para fornecer computação mais rápida, sem requerer qualquer mudança na arquitetura dos sistemas de software. Primeiro, as taxas de *clock* dos processadores têm se tornado cada vez mais rápida com cada geração de processadores (mais ou menos a cada 18 meses).

Segundo diversos tipos de concorrência vêm pré-definidos nas arquiteturas dos processadores. Dentre eles, estão o *pipelining* de instruções e dados da memória para o processador (as instruções são obtidas e decodificadas enquanto a instrução atual está sendo executada), o uso de linhas separadas para instruções e dados, pré-carregando instruções e dados, e o paralelismo na execução de operações aritméticas. Todos esses tipos são coletivamente chamados de concorrência oculta.

O resultado dos aumentos na velocidade de execução é grande ganhos de produtividade sem requerer que os desenvolvedores produzam sistemas de software concorrentes. Entretanto, a situação está mudando. O fim da sequência de aumentos significativos na velocidade de processadores individuais está próximo. Com a aparição de processadores múltiplos em um único *chip*, como os Intel Core Duo, há mais pressão para desenvolvedores de software usarem mais os processadores múltiplos disponíveis nas máquinas. Se eles não fizerem, a concorrência em hardware será perdida e os ganhos de produtividade diminuirão significativamente (Sebesta, 2011).

## 2.2 Benefícios do uso de programação concorrente

O objetivo de desenvolver softwares concorrente é produzir algoritmos concorrentes escaláveis e portáveis. Um algoritmo concorrente é escalável se a velocidade de sua execução aumenta quando mais processadores estão

disponíveis. Isso é importante porque o número de processadores aumenta com cada nova geração de máquinas. Os algoritmos devem ser portáveis porque o tempo de vida de hardware é relativamente curto. Logo, os sistemas de software não devem depender de uma arquitetura em particular, ou seja, devem rodar eficientemente em máquinas com diferentes arquiteturas.

Existem ao menos duas razões para projetar sistemas de software concorrentes. A primeira é a velocidade de execução dos programas. Hardware concorrente fornece uma maneira efetiva de aumentar a velocidade de execução de programa, desde que os programas sejam projetados para usar a concorrência em hardware. Existe agora muitos computadores instalados com múltiplos processadores, incluindo muitos dos computadores pessoais vendidos nos últimos anos. É uma perda não usar essa capacidade de hardware.

A segunda razão é que a concorrência 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 a solução de alguns problemas. Além disso, muitos programas são escritos para simular entidades e atividades físicas. Em muitos casos, o sistema simulado inclui mais de uma entidade, e elas fazem tudo simultaneamente (Sebesta, 2011).

## 2.3 Categoria de concorrência

Também deve-se considerar que a execução de unidades de programa concorrentes pode ocorrer fisicamente em processadores separados, ou logicamente em fatias de tempo diferentes em um único processador. Existem duas categorias de controle de unidades concorrentes. Na Concorrência física assume-se que mais de um processador está disponível e vários subprogramas de um mesmo programa estão literalmente executadas simultaneamente. Na concorrência lógica tanto o programador como o sistema assumem que existem múltiplos processadores fornecendo concorrência, quando, na verdade, a execução atual do programa é feita em intervalos de tempo diferentes no mesmo processador.

Do ponto de vista do programador e do projetista de linguagem, a concorrência lógica é o mesmo que a concorrência física. É tarefa do implementador da linguagem, usando as capacidades do sistema operacional, mapear a concorrência lógica para o 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 (Sebesta, 2011).

# 3 CONCEITOS FUNDAMENTAIS

Antes de abordarmos os suportes de linguagem de concorrência, é importante conhecer alguns conceitos que fazem parte de concorrência, e os requisitos para que tais suportes sejam úteis.

## 3.1 *Threads*

Tarefa, processo, ou também chamado de *thread*, é um conceito relacionado à uma unidade de programa, e que pode estar em execução concorrentemente com outras unidades do mesmo programa. Uma *thread* pode suportar uma linha de execução de controle (Sebesta, 2011).

Existem dois tipos de tarefas: tarefas leves e tarefas pesadas. As tarefas leves, além de mais fáceis de serem implementadas, possuem menor custo de processamento, e em algumas ocasiões é a melhor opção de uso para um programa ou subprograma.

As tarefas se diferenciam dos subprogramas em alguns pontos, tais como (Sebesta, 2011):

- Uma tarefa é implicitamente chamada, enquanto um subprograma deve ser explicitamente chamado.
- Quando uma unidade de programa invoca uma tarefa, esta unidade não precisa esperar que uma tarefa complete sua execução antes de continuar sua própria.
- Quando a execução de uma tarefa terminar, o controle pode ou não retornar à unidade que iniciou sua execução.

As tarefas podem trabalhar em conjunto, comunicando-se por passagem de mensagens, variáveis compartilhadas, ou parâmetros. Tais tarefas possuem a denominação de disjuntas.

Grande parte das tarefas possuem a característica de trabalharem em conjunto com outras, ou seja, são não disjuntas, e necessitam de uma mesma forma de comunicação, tanto para compartilhar dados, como para sincronizar execuções.

## 3.2 Sincronização

Há também o conceito de sincronização**.** A sincronização é um mecanismo que controle a ordem de como as tarefas serão executadas. Há 2 tipos de sincronização necessários para o compartilhamento de dados, chamados de sincronização de competição, e sincronização de cooperação.

A sincronização de competição acontece quando as tarefas em questão precisam de um recurso que não pode ser usado de maneira simultânea. Já a sincronização de cooperação funciona de maneira diferente, ela será necessária entre as tarefas quando uma delas necessitar que a outra termina uma tarefa específica, não importa qual seja, e só assim, a primeira poderá continuar seu funcionamento.

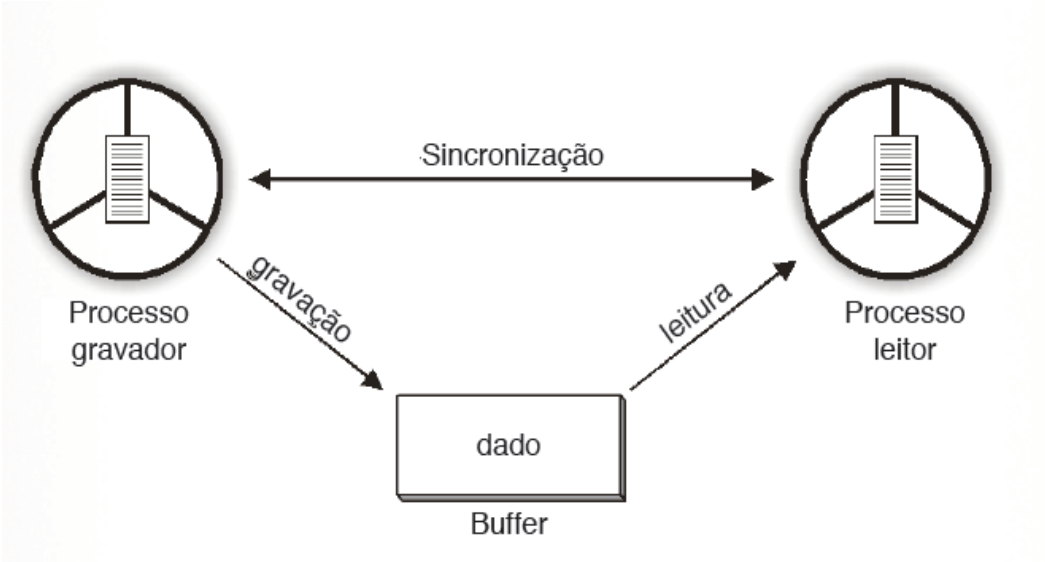



Ilustração de sincronização entre 2 tarefas

(Fonte:https://sites.google.com/site/proffernandosiqueiraso/aulas/7-sincronizacao-e-comunicacao-de-processos)

Resumindo, na sincronização de cooperação, as tarefas podem precisar esperar pelo término de processamento específico do qual sua operação correta depende, enquanto na sincronização de competição, as tarefas vão tomando a liderança independentemente de qual processamento foi finalizado, ambas competem pelo recurso em questão (Sebesta, 2011).

Abaixo segue uma ilustração abstraindo a ideia de como os processos buscam um recurso compartilhado:

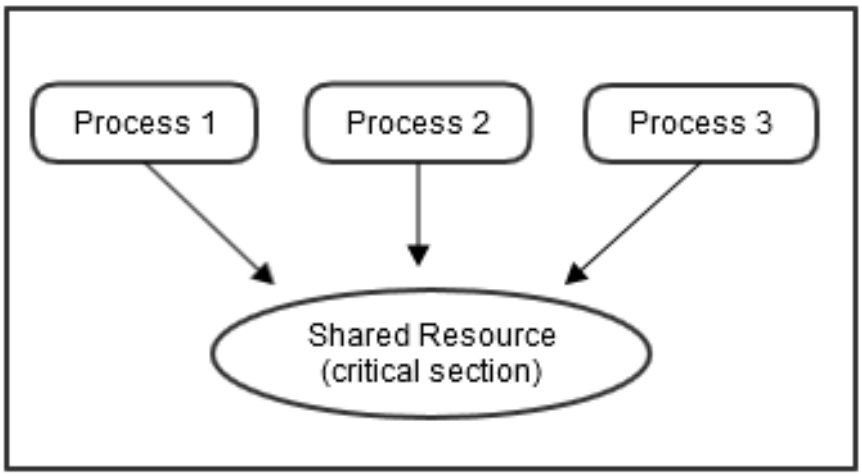



Ilustração de um recurso compartilhado entre tarefas.

(Fonte:https://commons.wikimedia.org/wiki/File:Multiple_Processes_Accessing_the_shared_resource.png)

## 3.3 Eficácia e uso

Abordando agora a eficácia do uso desses mecanismos de sincronização, será mencionado primeiro sobre a sincronização de competição. Ela é útil para prevenir que processos que estejam ativos acessem dados compartilhados ao mesmo tempo, pois caso isso acontecesse, poderia haver um grave problema de integridade dos dados compartilhados.

Uma maneira de exemplificar essa situação é quando um programa ou subprograma necessita que uma ocasião aconteça, se apenas certos eventos acontecerem antes. A sincronização de competição fará com que os processos envolvidos funcionem de forma ordenada, seguindo os eventos descritos, a fim de um resultado único e correto. Se não houvesse essa gestão, o resultado sairia diferente do esperado. Esse problema é chamado de condição de corrida, porque uma ou mais tarefas estão competindo para usar o dado compartilhado, e o comportamento do programa depende de qual tarefa chegar primeiro.

Agora em relação à sincronização de cooperação, podemos ilustrar um problema usando sua utilização, chamado de problema produtor-consumidor**.** Originado no sistema operacional, este problema remete à uma unidade de programa que produz um valor de algum dado ou recurso, e outra unidade o usa. Os dados são armazenados num buffer pela unidade que a produz, e removidos mais tarde por uma unidade consumidora (Sebesta, 2011).

## 3.4 *Deadlock*

Há também um problema chamado de *deadlock**,*** que ocorre quando 2 *threads* diferentes são executados de forma assíncrona. *Deadlock* é um erro que ocorre quando uma *thread* estiver esperando por um evento que nunca acontecerá (Tucker, 2009).

Uma maneira de exemplificar o erro de *deadlock*, é o problema de congestionamento de trânsito em uma intersecção, onde cada veículo que está entrando é bloqueado por outro. Os *deadlocks* podem acontecer se caso estas 4 condições ocorrerem (Tucker, 2009):

- As *threads* garantem direitos exclusivos de um dado.
- As *threads* retém alguns dados enquanto esperam por outros, ou seja, elas recebem estes dados de forma gradativa, e não de uma vez.
- Os dados não podem ser removidos das *threads* em espera
- Existe uma cadeia circular de *threads* na qual cada processo está travando uma ou mais dados necessários para os próximos processos na cadeia.

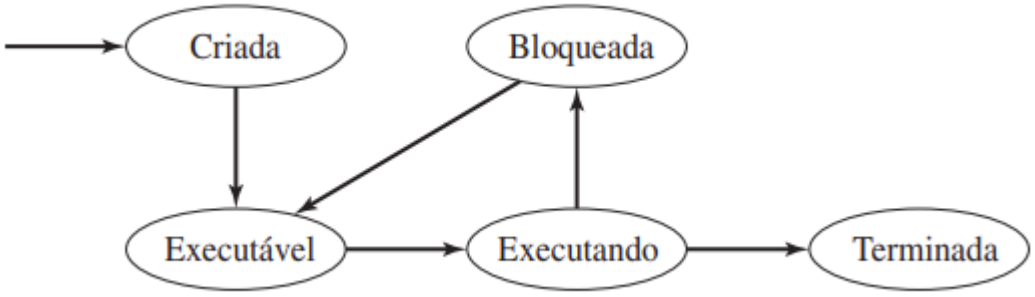
Mais à frente será discutido alguns métodos para acesso de um recurso compartilhado, tais como os semáforos, monitores e passagens de mensagens.

## 3.5 Escalonador

Os mecanismos envolvidos numa sincronização têm por objetivo gerenciar e atrasar tarefas, já que ela impõe uma ordem para execução que garante estes atrasos. Acontece que existe uma relação entre processos e processadores, porém existe a possibilidade de haver mais tarefas do que processadores. Há um programa que é responsável por gerenciar esse compartilhamento entre processadores e tarefas, chamado de escalonador**.**

O escalonador é um subsistema do sistema operacional, que tem por objetivo escolher os processos a serem executados pela CPU. Existem alguns diversos eventos complicadores que podem mudar a maneira de funcionamento de um escalonador, atrasos de sincronização, e espera por operações de entrada e saída. As tarefas podem estar em diversos estados, tais como:

- **Nova,** quando ela acabou de ser criada, porém ainda está inativa.
- **Pronta,** uma tarefa que está pronta para ser executada, mas que ainda não está em funcionamento, ou ainda não deu tempo de processador pelo escalonador.
- **Executando,** quando a tarefa está sendo executada e em funcionamento.
- **Bloqueada,** uma tarefa que estava em funcionamento, porém foi bloqueada por alguma interferência, como por exemplo alguns dos diversos eventos.
- **Morta,** uma tarefa que está morta e inativa novamente. Pode ter sido finalizada ou for morta explicitamente pelo programa.




As mudanças entre os estados dessas threads podem ser visualizadas em códigos escritos em algumas linguagens como C ou Ada, mas principalmente em Java, justamente pela existência dos conceitos de herança, e interface.

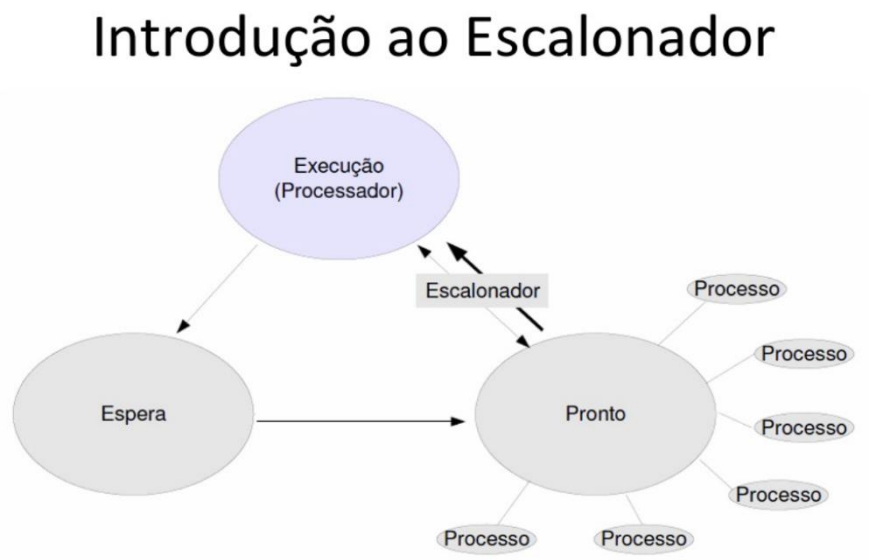



Ilustração de como um escalonador funciona
(Fonte:https://www.slideshare.net/SofiaTrindade6/categorias-de-escalonamento-e-objetivos-do-algortmo-escalonado)

### 3.6 Comportamento das *threads*

É importante ressaltar o comportamento das *threads*, já que elas podem mudar entre estados. Um exemplo seria os estados bloqueado e executando, e podemos tomar como exemplo uma fila de impressão, em que uma *thread* pode enviar vários documentos para uma fila de impressão, porém há a alternância de estados, pois há a espera de um documento ser impresso, para que o próximo seja liberado.

Programas concorrentes requerem comunicação ou interação *interthreads*, e isso acontece pelas seguintes razões de que:

- Em algumas situações, as *threads* precisam de acesso exclusivo a um dado compartilhado, como na situação de fila de impressão citada acima (Tucker, 2009).
- Uma *thread* pode, às vezes, precisar trocar dados com outras *threads*.

Em ambos os casos, as duas threads em comunicação devem sincronizar sua execução para evitar conflito ao adquirir recursos ou para fazer contato ao trocar dados (Tucker, 2009).

Essa comunicação entre *threads* é feita a partir de conceitos já citados: passagem de mensagens, variáveis compartilhadas e parâmetros (Tucker, 2009).

# 4 Semáforos

Um semáforo é uma estrutura de dados que 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.

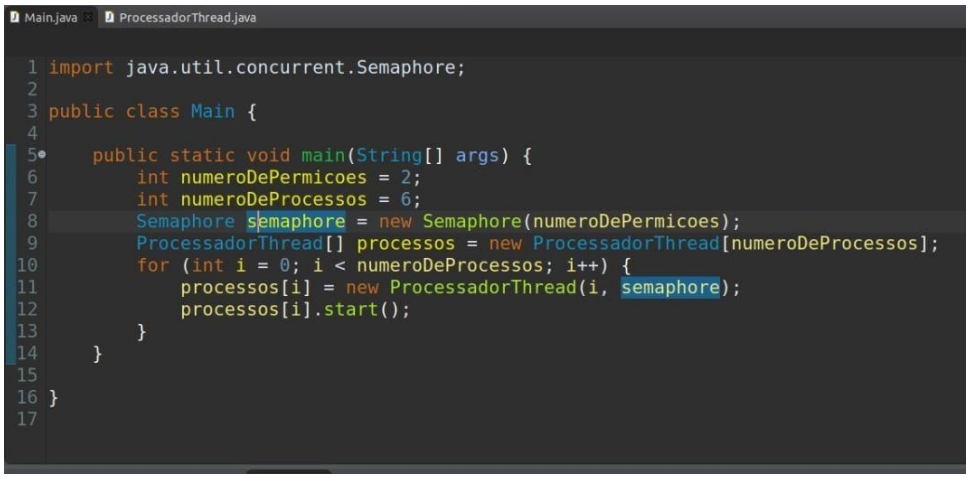
O conceito de um semáforo é que, para fornecer acesso limitado a uma estrutura de dados, guardas são colocadas ao redor do código que acessa a estrutura. Uma guarda é um dispositivo linguístico que permite ao código guardado ser executado apenas quando uma condição específica é verdadeira. Uma guarda pode ser usada para permitir que apenas uma tarefa acesse uma estrutura de dados compartilhada em um dado momento.

Um semáforo é uma implementação de uma guarda. Uma parte integral de um mecanismo de guarda é um procedimento para garantir que todas as execuções tentadas do código de guarda ocorram em algum momento do tempo. O procedimento típico é fazer com que requisições para o acesso ocorridas quando ele não pode ser dado 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. Essa é a razão pela qual um semáforo deve ter tanto um contador quanto uma fila de descritores de tarefa. (Sebesta, 2011).

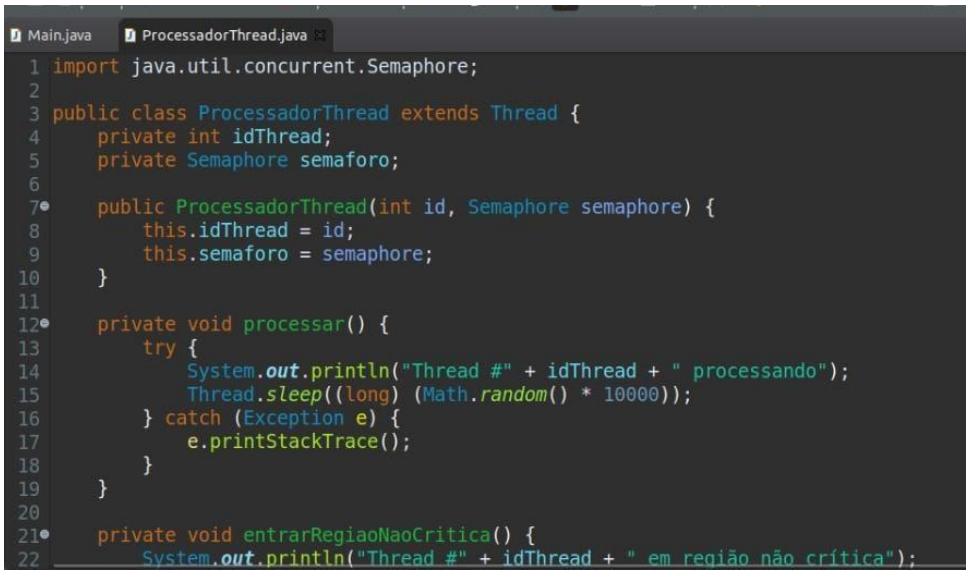
## 4.1 Avaliação

Usar semáforos para sincronização de cooperação cria um ambiente de programação inseguro. Não existe uma maneira de verificar estaticamente a corretude de seu uso, que depende da semântica do programa nos quais eles aparecem. No exemplo do buffer, deixar a sentença wait (emptyspots) de fora da tarefa producer resultaria em um transbordamento do buffer. Deixar wait (fullspots) de fora da tarefa consumer resultaria em um transbordamento negativo do buffer. Deixar de fora qualquer uma das liberações resultaria em um impasse.

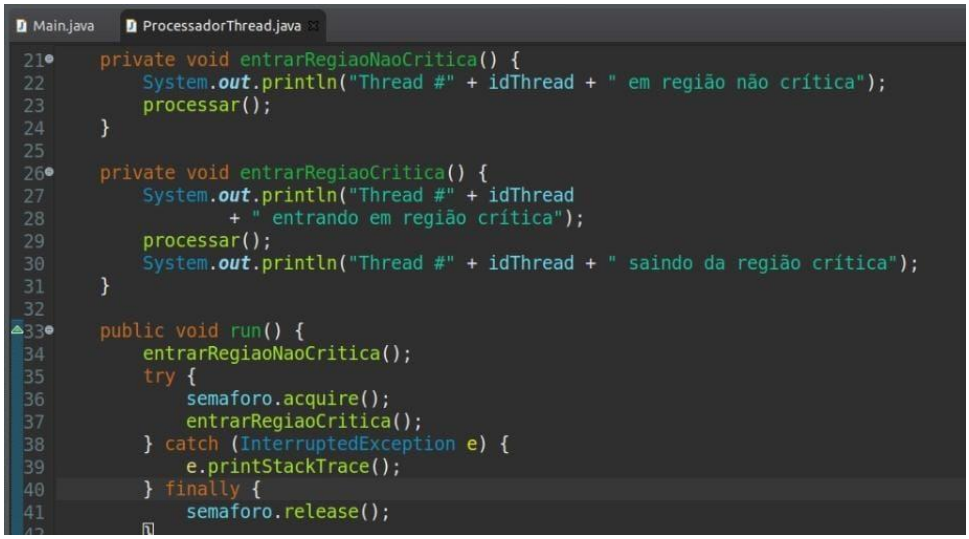
Essas são as falhas de sincronização de cooperação. Os problemas de confiabilidade que os semáforos causam ao fornecer sincronização de cooperação também surgem quando são usados para sincronização de competição. Deixar de fora a sentença wait (access) em qualquer uma das tarefas pode causar acessos inseguros ao buffer. Deixar de fora a sentença release (access) em qualquer das tarefas resultaria em um impasse. Notando o perigo do uso de semáforos, Per Brinch Hansen (1973) escreveu “O semáforo é uma ferramenta de sincronização elegante para um programador ideal que nunca comete erros”. Infelizmente, programadores ideais são raros. (Sebesta, 2011).




Exemplo do código Main da implementação do semáforo.




Funções do semáforo.




Continuação das funções do semáforo.

## 4.2 Funcionamento do código

Repare que semáforos não-cheios/não-vazios são semáforos de contagem, enquanto o semáforo de bloqueio é um semáforo binário.

O produtor primeiro produz informações localmente. Depois ele se certifica de que o buffer está não-cheio, fazendo uma operação P no semáforo nonfull; se o buffer não estiver cheio, um ou mais produtores continuarão. Em seguida, o produtor precisa de acesso exclusivo às variáveis do buffer compartilhado. Para obter esse acesso, o produtor executa uma operação P no semáforo binário lock. Passar além desse ponto garante acesso exclusivo às diversas variáveis do buffer compartilhado. O produtor deposita suas informações e sai da seção crítica executando uma operação V no semáforo de bloqueio. Finalmente, o produtor sinaliza às threads consumidoras que o buffer não está vazio por meio de uma operação V no semáforo nonempty. (Tucker, 2009).

# 5 MONITORES

Quando os conceitos de abstração de dados estavam sendo formulados, as pessoas envolvidas nesse esforço aplicaram os mesmos conceitos a dados compartilhados em ambientes de programação concorrente para produzir monitores. De acordo com Per Brinch Hansen. Essa solução pode fornecer sincronização de competição sem semáforos ao transferir a responsabilidade de sincronização para o sistema de tempo de execução (Sebesta, 2011).

Assim sendo, monitor é uma técnica para sincronizar duas ou mais tarefas que compartilham um recurso em comum, geralmente um dispositivo de hardware ou uma região da memória. Com um modelo de concorrência baseado em monitores, o compilador ou o interpretador podem inserir mecanismos de exclusão mútua transparentemente em vez do programador ter acesso às primitivas para tal, tendo que realizar o bloqueio e desbloqueio de recursos manualmente. O monitor consiste em um conjunto de procedimentos para permitir a manipulação de um recurso compartilhado, uma trava de exclusão mútua, as variáveis associadas ao recurso e uma invariante que define as premissas para evitar disputa de recursos.

A primeira linguagem de programação a incorporar monitores foi o Pascal Concorrente. Dentre as linguagens contemporâneas, eles são suportados por Ada, Java e C#. (Sebesta, 2011).

Em Java, um monitor pode ser implementado em uma classe projetada como um tipo de dados abstrato, com os dados compartilhados sendo o tipo. Acessos a objetos da classe são controlados por meio da adição do modificador synchronized aos métodos de acesso.

## 5.1 Sincronização de Cooperação

Apesar de o acesso mutuamente exclusivo aos dados compartilhados ser intrínseco com um monitor, a cooperação entre processos ainda é tarefa do programador. Em particular, o programador deve garantir que um buffer compartilhado não sofra de transbordamentos positivos ou negativos. Diferentes linguagens fornecem diferentes maneiras de programar a sincronização de cooperação, onde todas são relacionadas aos semáforos. (Sebesta, 2011).

## 5.2 Sincronização de Competição

O acesso ao dado compartilhado no monitor é limitado pela implementação a um simples processo por vez; então, acesso mutuamente exclusivo é inerente da definição semântica do monitor.

O programador não sincroniza o acesso mutuamente exclusivo aos dados compartilhados pelo uso de semáforos ou de outros mecanismos. Como os mecanismos de acesso fazem parte do monitor, a implementação de um pode ser feita de forma a garantir acesso sincronizado. Chamadas a procedimentos do monitor são implicitamente enfileiradas se ele estiver ocupado no momento da chamada. (Sebesta, 2011).

5.3 Avaliação

Monitores são uma forma melhor de fornecer sincronização de competição do que os semáforos, principalmente por causa dos problemas dos semáforos. A sincronização de cooperação ainda é um problema com os monitores. Semáforos e monitores são igualmente poderosos para expressar o controle de concorrência – os semáforos podem ser usados para implementar monitores, e monitores podem ser usados para implementar semáforos. (Sebesta, 2011).

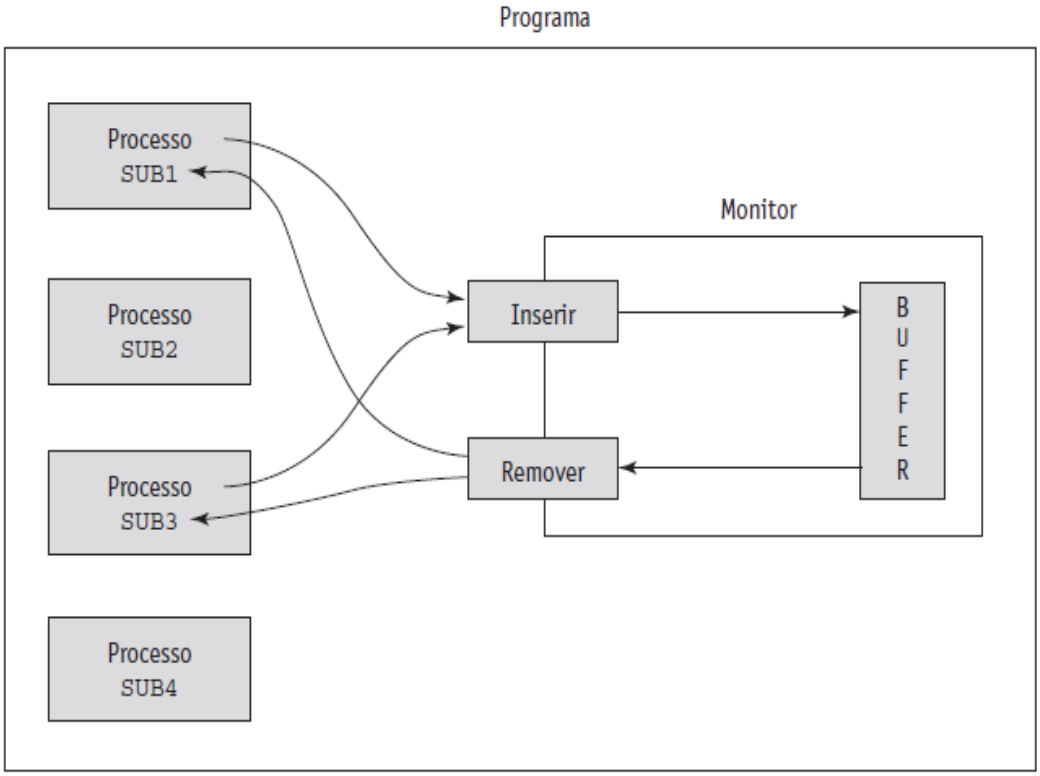



**Figura 5.4-1** Um programa usando um monitor para controlar o acesso a um buffer compartilhado. (Sebesta, 2011).

# 6 PASSAGEM DE MENSAGENS

Os primeiros esforços para projetar linguagens que fornecem a capacidade para passagem de mensagens entre tarefas concorrentes foram os de Brich Hansen (1978) e Hoare (1978). Eles também desenvolveram uma técnica para tratar do problema de o que fazer quando múltiplas requisições simultâneas eram feitas por outras tarefas para se comunicarem com uma dada tarefa.

Construções não determinísticas para controle no nível de sentenças, chamadas de comandos protegidos, foram introduzidas por Dijkstra (1975) e são a base para a construção projetada para controlar a passagem de mensagens. (Sebesta, 2011).

## 6.1 Passagem Síncrona e Assíncrona

A passagem de mensagens pode ser síncrona ou assíncrona. O conceito básico da passagem síncrona de mensagens é que as tarefas estão normalmente ocupadas e não podem ser interrompidas por outras unidades. A alternativa é fornecer um mecanismo linguístico capaz de permitir a uma tarefa especificar para outras quando ela está pronta para receber mensagens. Tanto a sincronização de tarefas de cooperação quanto a de competição podem ser manipuladas pelo modelo de passagem de mensagens. (Sebesta, 2011).

Já na passagem de mensagens assíncronas, a tarefa é capaz de reagir imediatamente a mensagens de outras tarefas. Além da parte do disparo, a cláusula de seleção assíncrona tem uma parte abortável, podendo conter qualquer sequência de sentenças Ada. A semântica de uma cláusula de seleção assíncrona é executar apenas uma de suas duas partes. (Sebesta, 2011).

## 7 CONCLUSÃO

Este trabalho visa analisar as características que diferencial uma programação concorrente de uma estruturada e a apresentação dos 4 níveis na execução do software na programação concorrente. Discutimos os principais pontos positivos que a programação concorrente e o objetivo de desenvolver softwares concorrente.

Foi visto alguns conceitos importantes para o entendimento geral sobre a programação de concorrência, como o significado das threads, sua importância, modificações e ações da threads e seus comportamentos, tais comportamentos como erros de programa, estados das threads e como isso pode influenciar. Foi possível analisar e entender sobre o uso de sincronização, seu significado e o porquê de ser um conceito tão presente num programa concorrente. Aliado a isso, o estudo de um escalonador e seu objetivo também foi possível, já que ambos os conceitos andam juntos uns com os outros.

Um semáforo é uma estrutura de dados que controle o acesso de aplicações aos recursos, baseando-se em um número inteiro, que representa a quantidade de acessos que podem ser feitos. Assim utilizamos semáforos para controlar a quantidade de acesso a determinado recurso.

Como já comentamos, no Java os Thread são acordados aleatoriamente, então a saída não será a mesma. O que é realmente importante notar é que nunca temos mais que duas Thread na região crítica.

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

## 8 BIBLIOGRAFIA

Sebesta, R. Conceitos de Linguagem de Programação. 9.ed, Bookman Company Ed, 2011.

Tucker, A., and R. Noonan. Linguagens de Programação. Princıpios e Paradigmas, 2009.

Alchieri, Eduardo. Programação Concorrente. Cic.unb.br. Disponível em <https://tecnoblog.net/247956/referencia-site-abnt-artigos/>.Acesso em: 01 de set. 2020.

Coura, Débora Pereira. Linguagem de programação: Concorrência. pt.slideshare.net. Disponível em <https://pt.slideshare.net/alexsandro_xpt/cap-08-concorrencia>. Acessado em: 03 de set. 2020.