**Introdução da aula**



**Qual é o foco da aula?**

Nessa aula, vamos estudar sobre a comunicação entre processos e *threads*.

**Objetivos gerais de aprendizagem**

Ao longo desta aula, você irá:

* definir a execução cooperativa de processos e *threads*, que solicitam o uso de recursos como memória, arquivos, dispositivos de entrada/saída e registros;
* explicar os métodos que impedem que um processo invada outro quando um deles está em sua região crítica;
* analisar os tipos de exclusão mútua: dormir e acordar, semáforos, monitores e troca de mensagens.

**Situação-problema**

Em um sistema operacional, os processos e *threads* trocam informações entre si ou solicitam a utilização de recursos simultaneamente, como arquivos, dispositivos de entrada/saída e memória. Um exemplo de comunicação interprocessos é a transferência de dados entre processos. Se um processo deseja imprimir um arquivo, ele o insere em um diretório de impressão com um nome para identificá-lo e outro processo é responsável por verificar periodicamente se existem arquivos a serem impressos.

Nessa aula você saberá como é feita a comunicação entre processos e *threads*. Veremos alguns pontos sobre essa comunicação, como condições de disputa, regiões críticas e exclusão mútua com espera ociosa. Além disso, estudaremos os mecanismos de sincronização que resolvem a exclusão mútua: dormir e acordar, semáforos, monitores e troca de mensagens.

Relembrando nosso contexto, Lucas acabou de ser contratado como gerente da área de Tecnologia da Informação de uma empresa prestadora de serviços hospitalares e, durante o acompanhamento do atendimento realizado por um estagiário, em relação aos constantes travamentos do *software* de edição de arquivos ao salvar ou realizar a formatação, eles observam que outro usuário está fazendo o *download* de um arquivo.

Durante o *download*, um processo (A) é responsável por baixar o arquivo e outro processo (B) é responsável por gravar estas informações em disco. Caso tenha outro processo acessando o disco, o processo (B) aguarda a sua vez para poder acessar o disco e gravar as informações. Caso contrário, o processo (B) grava os dados em disco e libera o recurso compartilhado para outro processo.

Dessa forma, o estagiário faz os seguintes questionamentos a Lucas:

* quais são os desafios da comunicação entre os processos?
* quais os problemas clássicos da comunicação entre processos?
* e se fosse utilizado um algoritmo de escalonamento de processos por meio de semáforos?
* qual a importância e os benefícios de sua utilização?

Para que você consiga responder esse e outros questionamentos sobre a comunicação entre processos e os mecanismos de sincronização que resolvem a exclusão mútua, nesta aula vamos conhecer mais sobre os conteúdos pertinentes a este tema.

Bons estudos!

**Execução cooperativa de processos e threads**



Em uma aplicação concorrente (execução cooperativa de processos e *threads*), os processos precisam se comunicar entre eles, então solicitam o uso de recursos como memória, arquivos, dispositivos de entrada/saída e registros.

Por exemplo, uma região de memória é compartilhada entre vários processos. O sistema operacional deve garantir que esta comunicação seja sincronizada para manter o bom funcionamento do sistema e a execução correta das aplicações.

Segundo Tanenbaum (2003), é necessário levar em consideração três tópicos:

1. como um processo passa a informação para outro processo.
2. garantir que dois ou mais processos não invadam uns aos outros quando estão em regiões críticas (será detalhada no decorrer da aula). Quando um processo estiver usando uma região de memória, o outro processo deve aguardar a sua vez.
3. é necessário existir uma hierarquia quando houver dependências. Se o processo A produz dados e o processo B os imprime, B deve esperar até que A produza dados para serem impressos.

Estas questões serão discutidas ao longo desta aula e também se aplicam a *threads*.

**Condições de disputa ou condições de corrida**

Condições de disputa ou condições de corrida acontecem quando dois ou mais processos estão compartilhando alguma região da memória (lendo ou escrevendo dados) e o resultado final depende das informações de quem executa e quando.

Como exemplo, podemos citar o problema da Conta\_Corrente relatado por Machado e Maia (2007). Nesta situação, o saldo bancário de um cliente é atualizado por meio de um programa após o lançamento de um débito ou crédito no arquivo de contas correntes (neste arquivo são armazenadas informações sobre o saldo dos correntistas do banco).

O registro do cliente e o valor depositado ou sacado são lidos por um programa e o saldo do cliente é atualizado. Suponha que dois funcionários do banco atualizem o saldo do mesmo cliente simultaneamente.

O processo do primeiro funcionário lê o registro do cliente e soma ao saldo o valor sacado pelo cliente. Porém, antes de gravar o novo saldo no arquivo, o segundo funcionário lê o registro do mesmo cliente que está sendo atualizado e lança um crédito a ser somado ao saldo.

Independentemente do processo que atualizar primeiro, o dado gravado no arquivo referente ao saldo está inconsistente.

**Regiões críticas**

Para impedir as condições de disputa, é necessário definir maneiras que impeçam que mais de um processo leia e escreva ao mesmo tempo na memória compartilhada. Esses métodos são chamados de exclusão mútua, ou seja, quando um processo estiver lendo ou gravando dados, sua região crítica ou processo deve esperar.

A parte do programa em que o processo acessa a memória compartilhada é chamada de região crítica ou seção crítica.

Segundo Tanenbaum (2003), para termos uma boa solução, é necessário satisfazer quatro itens:

1. dois ou mais processos jamais estarão ao mesmo tempo em suas regiões críticas.
2. não se pode afirmar nada sobre o número e a velocidade de CPUs.
3. nenhum processo que esteja executando fora de sua região crítica pode bloquear outros processos.
4. nenhum processo deve esperar sem ter uma previsão para entrar em sua região crítica.

A seguir veremos as soluções propostas para realizar a exclusão mútua: exclusão mútua com espera ociosa, dormir e acordar, semáforos, monitores e troca de mensagens.

**Exclusão mútua: com espera ociosa**



Segundo Tanenbaum (2003), existem alguns métodos que impedem que um processo invada outro quando um deles está em sua região crítica. Confira quais são eles, a seguir.

**Desabilitando interrupções**

Nesta solução, as interrupções são desabilitadas por cada processo (qualquer parada que pode ocorrer por um evento) assim que entra em sua região crítica e ativadas novamente antes de sair dela. Desta forma, a CPU não será disponibilizada para outro processo.

Esta solução não é prudente, uma vez que, ao dar autonomia para processos, a multiprogramação fica comprometida. Se um processo, ao entrar em sua região crítica, desabilitasse as interrupções e se esquecesse de habilitá-las novamente ao sair, o sistema estaria comprometido.

Em sistemas com múltiplos processadores, a interrupção acontece em apenas um processador e os outros acessariam normalmente a memória compartilhada, comprometendo essa solução.

\_\_\_\_\_\_

**🔁 Assimile**

A solução de exclusão mútua com espera ociosa, como desabilitando interrupções e instrução TSL são mecanismos implementados no *hardware* do computador. As demais soluções são implementadas via *software*.

\_\_\_\_\_\_

**📝 Exemplificando**

Suponha que um processo de atualização da base de dados de uma empresa de vendas de seguros de vida desabilite as interrupções durante a noite para fazer uma nova carga da base de dados. Esta interrupção é necessária, uma vez que a carga demorará aproximadamente dez horas. Se o processo finalizar e não reabilitar as interrupções, a base de dados no dia seguinte não poderá ser usada e a consequência pode ser a perda financeira e de fidelização de clientes.

\_\_\_\_\_\_

**Variáveis de impedimento**

Esta solução contém uma variável chamada lock, inicializada com o valor 0. Segundo Tanenbaum (2003), o processo testa e verifica o valor dessa variável antes de entrar na região crítica e, caso o valor seja 0, o processo o altera para 1 e entra na região crítica. Caso o valor da variável seja 1, o processo deve aguardar até que seja alterado para 0.

Variáveis de impedimento não resolvem o problema de exclusão mútua e ainda mantêm a condição de disputa. Quando o processo 1 vê o valor da variável 0 e vai para alterar o valor para entrar na região crítica, chega o processo 2 e altera o valor da variável para 1, antes de o processo 1 ter alterado. Logo, os dois processos entram, ao mesmo tempo, na região crítica.

**Alternância obrigatória**

Segundo Tanenbaum (2003), essa solução utiliza uma variável *turn* compartilhada que informa qual processo poderá entrar na região crítica (ordem). Esta variável deve ser alterada para o processo seguinte, antes de deixar a região crítica.

Suponha que dois processos desejam entrar em sua região crítica. O processo A verifica a variável *turn* que contém o valor 0 e entra em sua região crítica. O processo B também encontra a variável *turn* com o valor 0 e fica testando continuamente para verificar quando ela terá o valor 1.

O teste continuo é chamado de espera ociosa, ou seja, quando um processo deseja entrar em sua região crítica, ele examina se sua entrada é permitida e, caso não seja, o processo fica esperando até que consiga entrar. Isso ocasiona um grande consumo de CPU, podendo impactar na performance do sistema.

Ainda segundo Tanenbaum (2003), assim que o processo A deixa sua região crítica, a variável *turn* é atualizada para 1 e permite que o processo B entre em sua região crítica.

Vamos supor que o Processo B é mais ágil e deixa a região crítica. Os processos A e B estão fora da região crítica e *turn* possui o valor 0. O processo A finaliza antes de ser executado em sua região não crítica. Como o valor de *turn* é 0, o processo A entra de novo na região crítica, e o processo B ainda permanece na região não crítica. Ao deixar a região crítica, o processo A atualiza a variável *turn* com o valor 1 e entra em sua região não crítica.

Os processos A e B estão executando na região não crítica e o valor da variável *turn* é 1. Se o processo A tentar entrar de novo na região crítica, não conseguirá, pois o valor de *turn* é 1. Desta forma, o processo A fica impedido pelo processo B, que **NÃO** está na sua região crítica.

Esta situação viola a seguinte condição: nenhum processo que esteja executando fora de sua região crítica pode bloquear outros processos.

**Solução de Peterson**

Segundo Tanenbaum (2003), essa solução foi implementada por meio de um algoritmo que consiste em dois procedimentos escritos em C, baseado na definição de duas primitivas (*enter\_region e leave\_region*) utilizadas pelos processos que desejam utilizar sua região crítica. Antes de entrar na região crítica, todo processo chama enter\_region com os valores 0 ou 1.

Este apontamento faz com que o processo aguarde até que seja seguro entrar. Depois de finalizar a utilização da região crítica, o processo chama *leave\_region* e permiti que outro entre. Como a solução de Alternância Obrigatória, a Solução de Peterson precisa da espera ociosa.

**Instrução TSL**

Segundo Tanenbaum (2003), a instrução TSL (*test and set lock*, ou seja, teste e atualize a variável de impedimento) conta com a ajuda do *hardware*.

A instrução TSL RX, LOCK faz uma cópia do valor do registrador RX para LOCK. Um processo somente pode entrar em sua região crítica se o valor de LOCK for 0. A verificação do valor de LOCK e sua alteração para 0 são realizadas por instruções ordinárias.

A solução de Alternância Obrigatória, a Solução de Peterson e a instrução TSL utilizam a espera ociosa.

**Exclusão mútua: dormir e acordar e semáforos**



**Dormir e acordar**

As soluções apresentadas até aqui utilizam a espera ociosa (os processos ficam em um laço ocioso até que possam entrar na região crítica). Para resolver este problema, são realizadas chamadas *sleep* (dormir) e *wakeup* (acordar) ao sistema, que bloqueiam/ desbloqueiam o processo, ao invés de gastar tempo de CPU com a espera ociosa.

A chamada *sleep* faz com que o processo que a chamou durma até que outro processo o desperte, e a chamada *wakeup* acorda um processo.

Para exemplificar o uso dessas chamadas, vamos apresentar o problema do produtor/consumidor.

**Problema do produtor/consumidor**

De acordo com Tanenbaum (2003), o problema do produtor/consumidor normalmente acontece em programas concorrentes em que um processo gera informações (produtor) para uso de outro processo (consumidor).

Imagine que dois processos compartilham um *buffer* (memória) de tamanho fixo. O problema acontece quando o produtor quer inserir um novo item, porém o *buffer* está cheio ou o consumidor deseja remover um item e o *buffer* está vazio.

A solução é colocar o processo, impedido pela capacidade do *buffer*, para dormir (através da chamada *sleep*) até que o outro modifique o *buffer* e acorde o anterior (por meio da chamada *wakeup*). Para controlar a quantidade de itens no *buffer*, é utilizada uma variável *count*.

O problema da condição de disputa acontece quando o *buffer* está vazio e o consumidor verifica o valor da variável *count*. O escalonador decide parar a execução do consumidor e executa o produtor, que inclui itens no *buffer* enviando um sinal de acordar para o consumidor. Como o consumidor ainda não está logicamente adormecido, o sinal para acordar é perdido e tanto o consumidor, quanto o produtor ficarão eternamente adormecidos.

**Semáforos**

Segundo Machado e Maia (2007), a utilização de semáforos é um dos mecanismos utilizados em projetos de sistemas operacionais e em aplicações concorrentes. Hoje, grande parte das linguagens de programação disponibiliza procedimentos para que semáforos sejam utilizados.

Um semáforo é uma variável inteira que realiza duas operações: *DOWN* (decrementa uma unidade ao valor do semáforo) e UP (incrementa uma unidade ao valor do semáforo).

As rotinas *DOWN*e*UP* são indivisíveis e executadas no processador. Um semáforo com o valor 0 indica que nenhum sinal de acordar foi salvo e um valor maior que 0 indica que um ou mais sinais de acordar estão pendentes (TANENBAUM, 2003).

De acordo com Machado e Maia (2007), os semáforos são classificados como:

* binários, também conhecidos como *mutexes* (*mutual exclusion semaphores*), que recebem os valores 0 ou 1.
* contadores, que recebem qualquer valor inteiro positivo, além do 0.

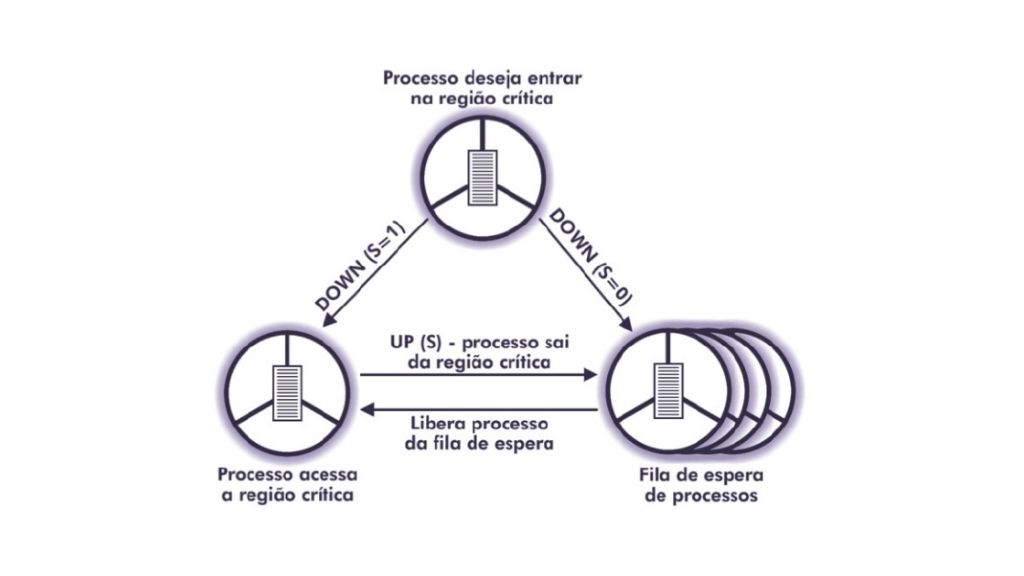
Para compreendermos melhor o uso de semáforos, utilizaremos um exemplo de semáforos binários.

Segundo Machado e Maia (2007), o semáforo com o valor igual a 1 significa que nenhum recurso está utilizando o processo e valor igual a 0 significa que o recurso está em uso.

A figura “Semáforo binário” apresenta o uso do semáforo binário na exclusão mútua. Quando um processo deseja entrar em sua região crítica, é executada a instrução *DOWN*.

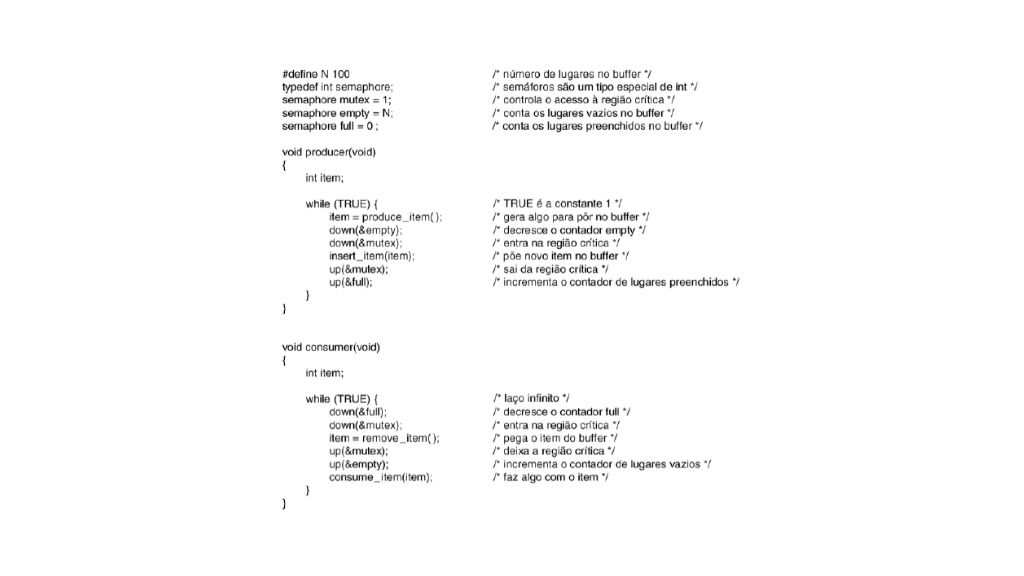
Caso o valor do semáforo seja igual a 1, o valor é decrementado e o processo pode entrar em sua região crítica. Caso o valor seja 0 e a operação *DOWN* seja executada, o processo é impedido de entrar em sua região crítica, permanecendo em fila no estado de espera.

O processo que utiliza o recurso executa a instrução *UP* ao deixar a região crítica incrementa o valor do semáforo e libera o acesso ao recurso. Caso existam processos aguardando na fila para serem executados, o sistema selecionará um e alterará o seu estado para pronto.

Semáforo binário. Fonte: Machado e Maia (2007, p. 108).

O problema do produtor/consumidor (perda de sinal de acordar) apresentado acima pode ser resolvido através de semáforos, conforme apresentado na figura “Problema do produtor/consumidor usando semáforos”, em que três semáforos são usados: um chamado *mutex*, que controla o acesso à região crítica e é inicializado com o valor 1, outro chamado *full*, que conta os valores preenchidos no *buffer*, sendo o valor inicial 0 e o terceiro chamado de *empty*, que conta os lugares vazios no *buffer* (representa o número de lugares do *buffer*).

Segundo Tanenbaum (2003), um processo, ao executar a operação *DOWN*, decrementa o seu valor inteiro. Caso o valor do semáforo seja negativo, o processo é impedido e inserido ao fim da fila. Quando um processo executa a operação UP, o seu valor inteiro é incrementado. Se existir algum processo impedido na fila, o primeiro processo é liberado.

Problema do produtor/consumidor usando semáforos. Fonte: Tanenbaum (2003, p. 82).

**Exclusão mútua: monitores e troca de mensagens**



**Monitores**

Segundo Tanenbaum (2003), um monitor é uma coleção de rotinas, estrutura de dados e variáveis que ficam juntos em um módulo ou pacote. Também pode ser definido como uma unidade de sincronização de processos de alto nível.

Um processo, ao chamar uma rotina de um monitor, verifica se existe outro processo ativo. Caso esteja, o processo que chamou é bloqueado até que o outro deixe o monitor, senão, o processo que o chamou poderá entrar.

A utilização de monitores garante a exclusão mútua, uma vez que só um processo pode estar ativo no monitor em um determinado momento e os demais processos ficam suspensos até poderem estar ativos. O compilador é o responsável por definir a exclusão mútua nas entradas do monitor.

É preciso definir métodos para suspenderem os processos caso não possam prosseguir, mesmo que a implementação da exclusão mútua em monitores seja fácil.

Assim, é necessário introduzir variáveis condicionais, com duas operações: *wait*e*signal*. Se um método do monitor verifica que não pode prosseguir, um sinal *wait*é emitido (bloqueando o processo), permitindo que outro processo que estava bloqueado acesse o monitor. A linguagem de programação Java suporta monitores.

\_\_\_\_\_\_

**💭 Reflita**

Os monitores e semáforos são soluções para CPUs que utilizam memória compartilhada. E como seria esse processo em sistemas distribuídos, em que existem troca de informações entre processos que estão em máquinas diferentes?

\_\_\_\_\_\_

**Troca de Mensagens**

Segundo Tanenbaum (2003), esse método utiliza duas chamadas ao sistema:

* *send (destination, &message)* - envia uma mensagem para um determinado destino.
* *receive (source, &message)* - recebe uma mensagem de uma determinada origem.

Caso nenhuma mensagem esteja disponível, o receptor poderá ficar suspenso até chegar alguma.

A troca de mensagens possui problemas como sua perda pela rede. Para evitar este problema, assim que uma mensagem é recebida, o receptor enviará uma mensagem de confirmação de recebimento. Caso o receptor receba e não confirme o recebimento, não será problema, uma vez que as mensagens originais são numeradas de forma sequencial.

Uma questão importante refere-se à autenticação, pois é necessário saber se a fonte é real. Além disso, as mensagens enviadas e recebidas não podem ser ambíguas.

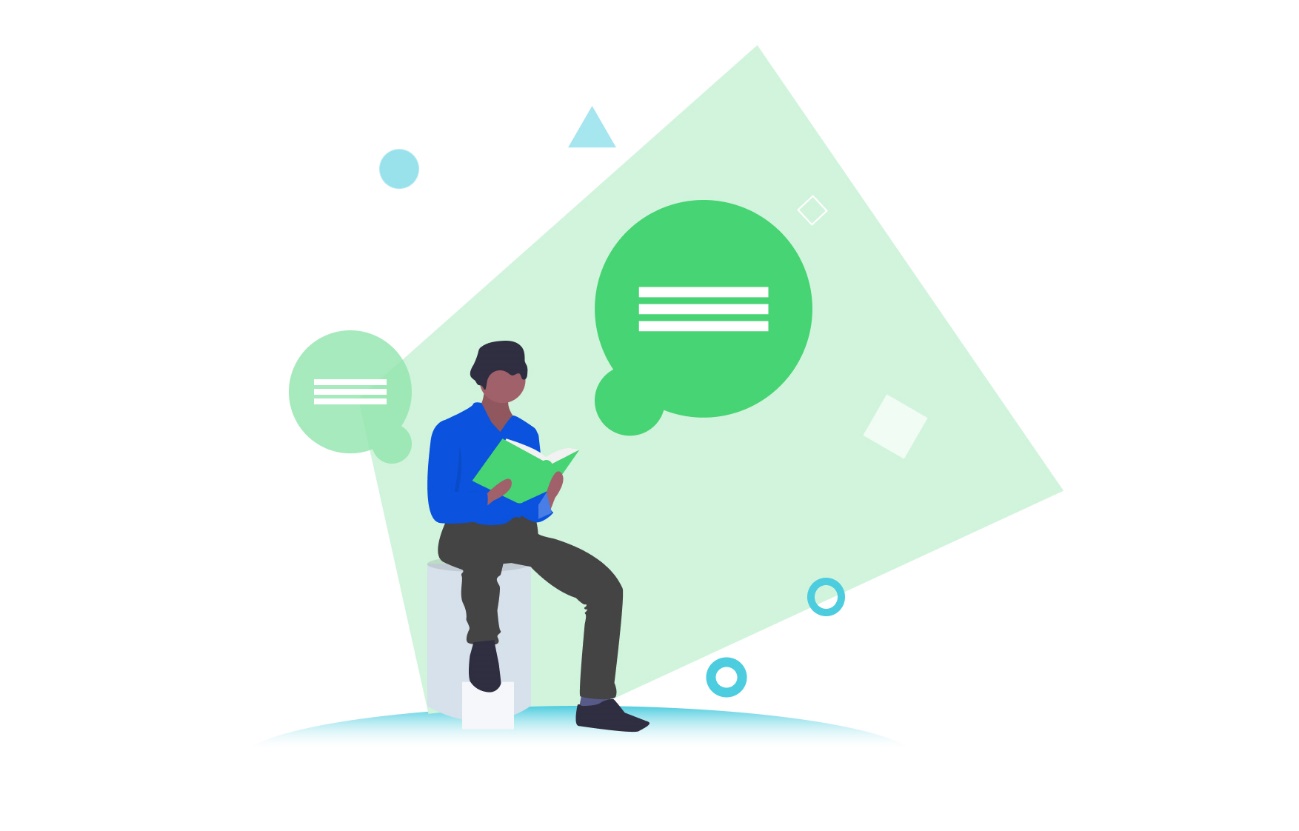
Quanto ao desempenho, copiar mensagens é um procedimento mais lento do que realizar operações sobre semáforos ou monitores. Uma solução seria realizar a troca de mensagens através de registradores.

\_\_\_\_\_\_

**🔁 Assimile**

Vimos nesta aula que, para resolver o problema de exclusão mútua, existem as seguintes soluções: exclusão mútua com espera ociosa, dormir e acordar, semáforos, monitores e troca de mensagens. Estes métodos são equivalentes quando implementados em um único processador.

**Conclusão**



Agora que você já aprendeu como os processos e *threads* fazem a comunicação, conheceu as questões sobre a comunicação entre processos como condições de disputa, regiões críticas e exclusão mútua com espera ociosa e estudou sobre os mecanismos de sincronização que resolvem a exclusão mútua dormir e acordar, semáforos, monitores e troca de mensagens, vamos voltar ao nosso contexto.

Durante o atendimento do chamado referente aos constantes travamentos do *software* de edição de arquivos ao salvar ou realizar a formatação, eles observam que outro usuário está fazendo o *download* de um arquivo. Como os sistemas operacionais precisam manter a comunicação entre processos para um bom funcionamento do sistema e para não comprometer a execução das aplicações, os desafios são grandes.

Os processos que concorrem por recursos do computador devem ter suas execuções sincronizadas, garantindo o correto processamento das aplicações. Para isso, o sistema operacional deve implementar mecanismos de controle e sincronismo, como a exclusão mútua que consiste em não permitir que dois processos acessem o mesmo recurso ao mesmo tempo, garantindo assim que nenhum processo invada a região crítica de outro.

Neste caso, foram desenvolvidas soluções para evitar a exclusão mútua, como: **dormir e acordar**, que faz com que um processo durma enquanto outro esteja acessando o recurso, até que seja acordado pelo outro processo; **semáforo**, uma variável responsável por contar a quantidade de sinais de dormir e acordar, sendo vantajoso por não existir a espera ocupada que desperdiça o processamento de máquina; entre outras soluções.

Analisando esta situação, um dos mecanismos de sincronização de processos chama-se Semáforos. Os Semáforos garantem que processos não acessem um mesmo recurso simultaneamente através de duas variáveis *Down*e*Up*, que são verificadas e alteradas sem a possibilidade de interrupções. A variável*Down* é decrementada quando o processo grava os dados e libera o disco para outro processo. A variável Up é incrementada quando outro processo já está usando a CPU.

Existem os problemas clássicos da comunicação entre processos e existem vários métodos para resolver estes problemas. Podemos citar o problema do jantar dos filósofos e o problema do barbeiro adormecido, conforme apresentado em Tanenbaum (2003).

O problema do jantar dos filósofos foi proposto e resolvido assim: cinco filósofos estão sentados em uma mesa circular e cada um tem um prato de espaguete e garfos. Porém, para comer são necessários dois garfos, pois o espaguete está escorregadio. Cada filósofo alterna entre comer e pensar e, quando algum deles fica com fome, tenta pegar os garfos da sua esquerda e direita, um de cada vez, para poder comer o espaguete. Caso consiga pegar os dois garfos, ele come por um tempo, liberando os garfos ao final e voltando à sua atividade de pensar. O problema é não gerar o bloqueio dos filósofos implementando soluções para resolver a questão do acesso exclusivo a um número restrito de recursos.

O problema do barbeiro adormecido acontece em uma barbearia. A barbearia tem um barbeiro, uma cadeira de barbeiro e várias cadeiras para que os clientes esperem a sua vez. Se não tiver clientes, o barbeiro adormece e quando um cliente chega, o barbeiro acorda e corta seu cabelo. Se chegarem outros clientes, eles verificam se há cadeiras de espera disponíveis e, caso não tenha, os clientes vão embora. O problema principal é programar o barbeiro e os clientes para não cairem em condição de disputa.