

=> Programação Concorrente

Mungoi, Belmiro
Cassamo, Kayron
Firme, Alfoi
Mandlate, Enoque
Xavier, Omar Davide

Agenda

Programas multithreads;

- Threads
- Modelo de processo modelo de thread
- Ciclo de Vida de uma thread
- Tipos de threads

Comunicação e Sincronização de Processos

Primitivas de Sincronização

Problemas Clássicos

Construções concorrentes de alto nível

Definição.

Uma Thread(linha de execução), é uma unidade basica de processamento que pode ser executada simultaneamente com outras threads dentro de um processo.

As Threads permitem que um programa realize multiplas tarefas simultaneamente, melhorando o desempenho e a eficiencia geral do sistema. Elas são amplamente utilizadas em aplicativos que precisam lidar com tarefas assincronas, como aplicativos de rede, processamento de dados, interface grafica de usuario e jogos

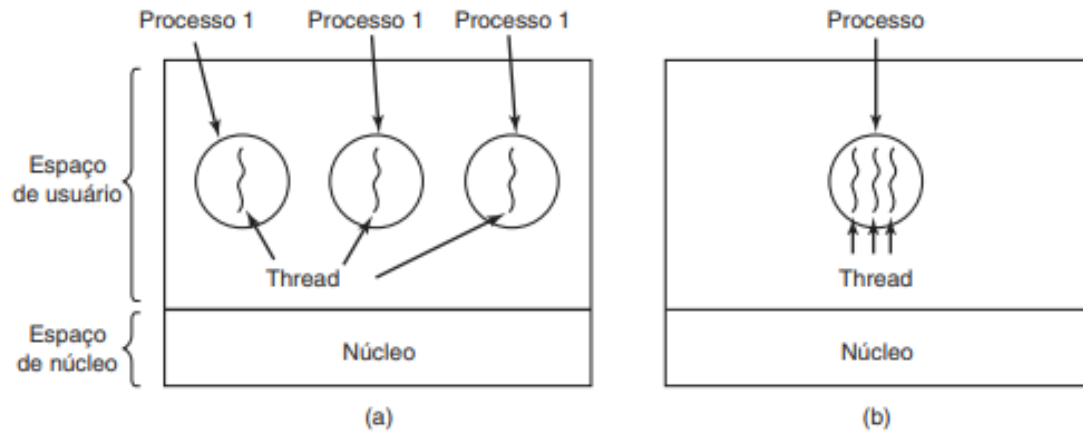


Figura 2-6 (a) Três processos, cada um com uma *thread*. (b) Um processo com três *threads*.

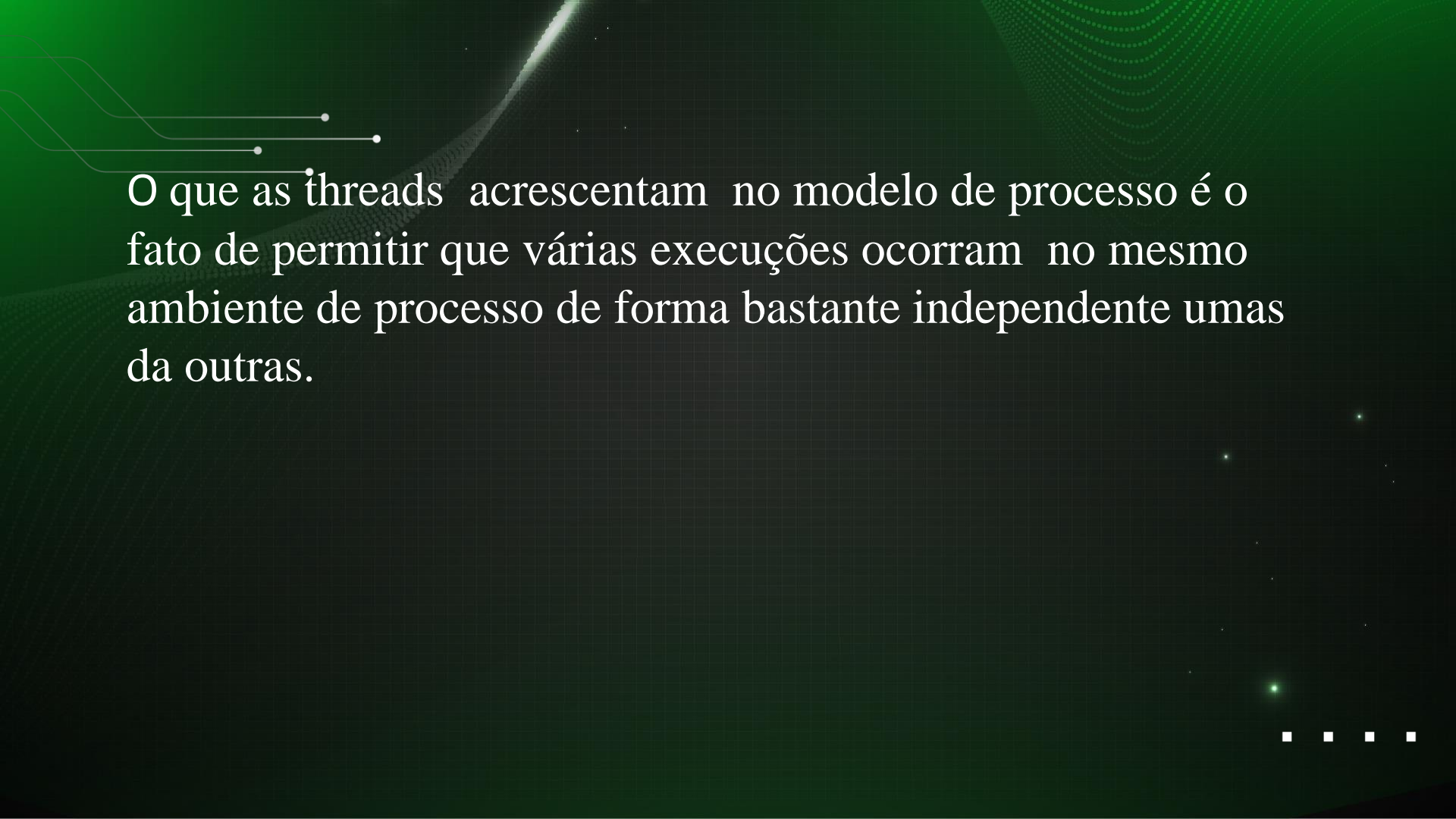
A idéia de multithreading é associar vários fluxos de execução (várias threads) a um único processo.

Modelo de Processo & de Threads

Embora uma thread deva ser executada em algum processo, a thread e seu processo são conceitos diferentes e podem ser tratados separadamente. Os processos são usados para agrupar recursos;

Itens por processo	Itens por threads
<ul style="list-style-type: none">• Espaço de endereçamento• Variáveis globais• Arquivos abertos• Processos filhos• Alarmes pendentes• Sinais e rotinas de tratamento de sinais• Informações de contabilização	<ul style="list-style-type: none">• Contador de programa• Registradores• Pilha de execução• Estado

Figura:2-7 A primeira coluna lista alguns itens compartilhados por todas as threads em um processo. A segunda lista alguns itens privativos de cada thread.



O que as threads acrescentam no modelo de processo é o fato de permitir que várias execuções ocorram no mesmo ambiente de processo de forma bastante independente umas das outras.

■ ■ ■ ■

Estados de threads, ciclo de vida de uma thread

Uma thread pode estar em um dos seguintes estados de execução:

Novo(Nascido): Quando uma thread é criada, ela está no estado novo. Nesse estado, o sistema operacional está configurando os recursos necessários para que a thread seja executada.

Pronta(Ready): Esperando para ser executada.

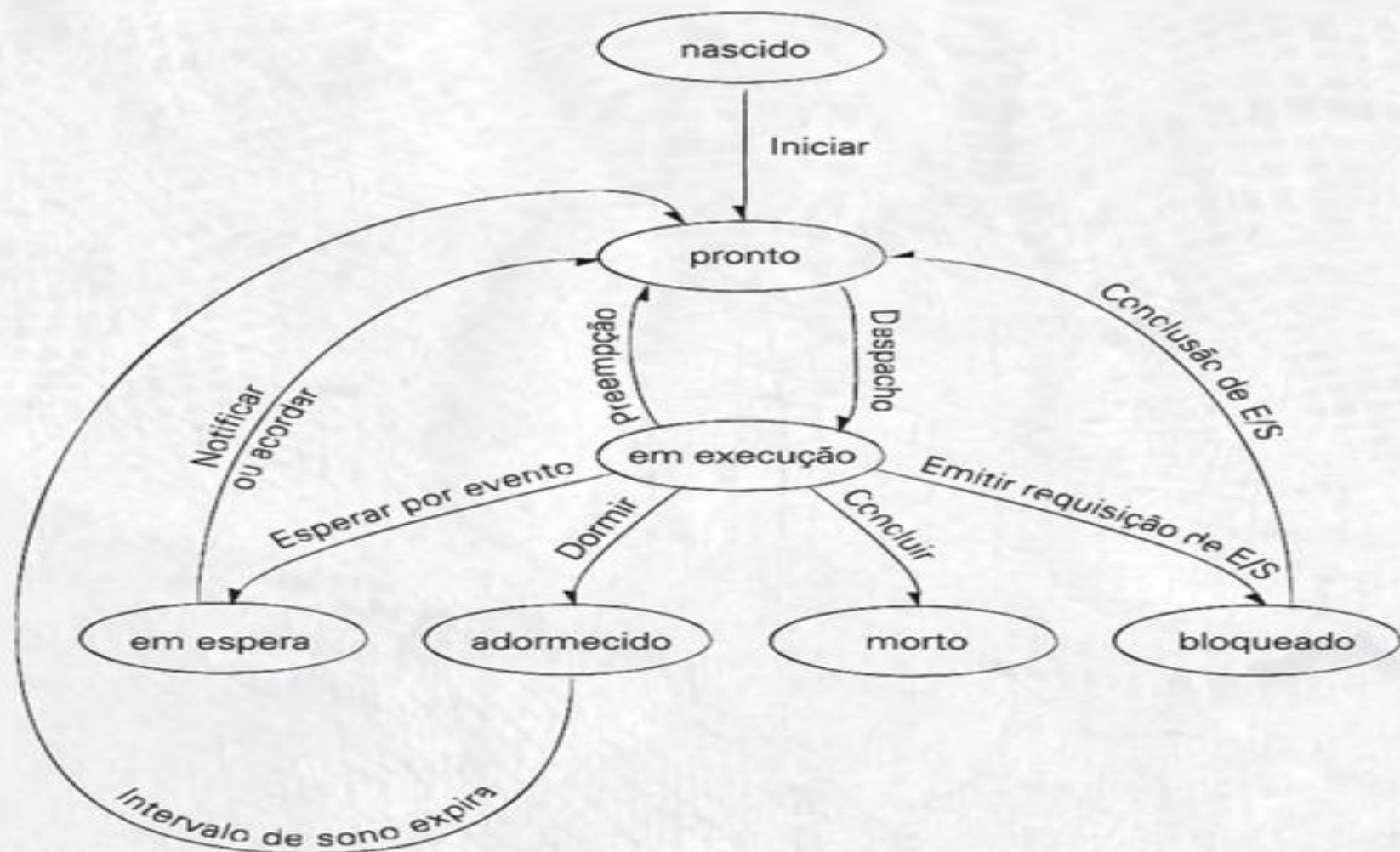
Executando(Running): sendo executada pela CPU.

Bloqueado: aguardando por algum recurso, como entrada/saída.

Espera: espera por um evento.

Suspensa/adormecido(Suspended): temporariamente interrompida e pode ser retomada posteriormente.

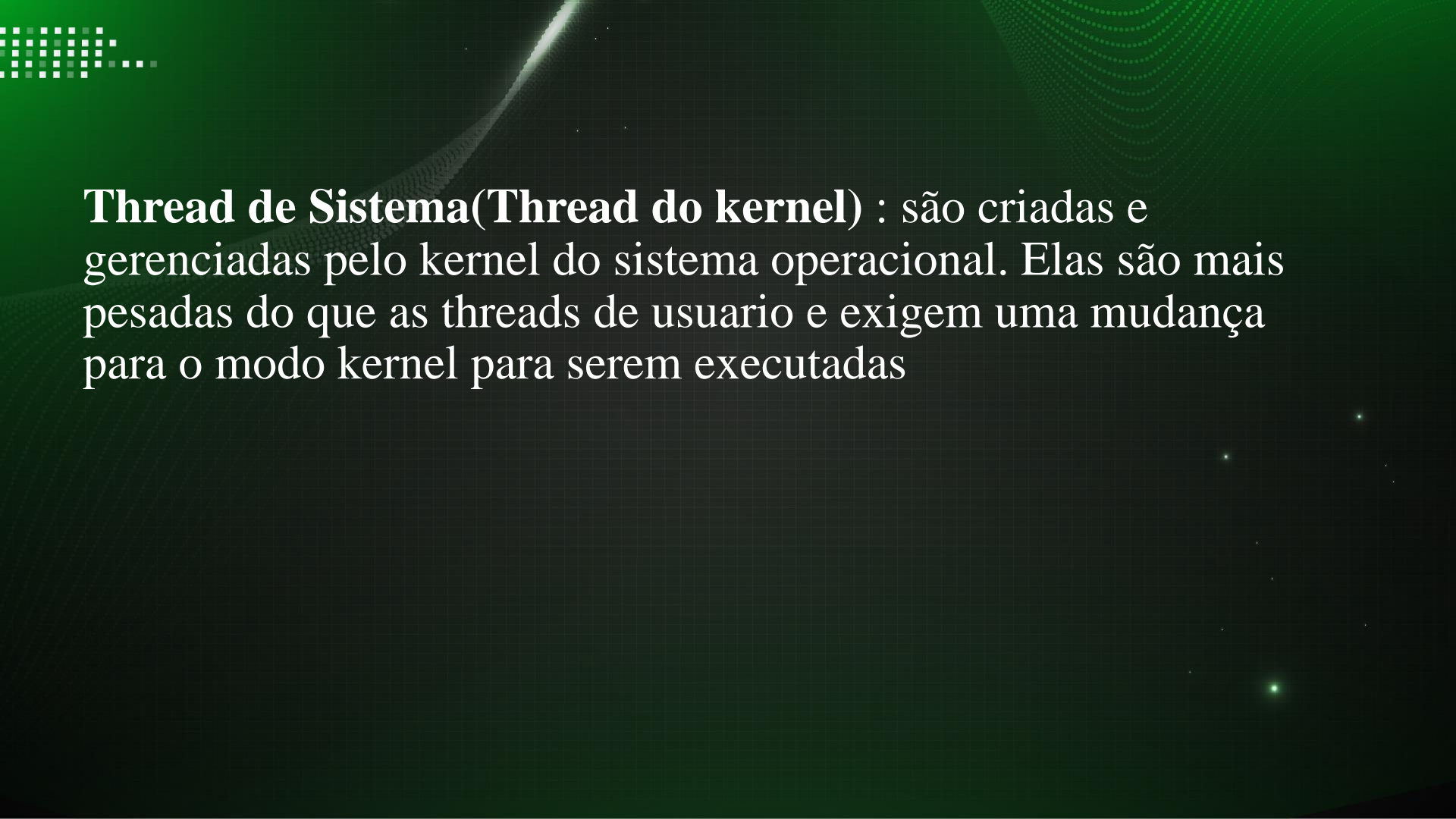
Morto (Dead): conclui sua execução e não pode mais ser executada.



Tipos de Threads

Existem dois tipos de threads:

Thread de Usuario: são criadas e gerenciadas pelo proprio processo do usuario, sem a intervencao do kernel. Essas threads são leves e são executadas em espaco de usuario, sem a necessidade de mudar para o proprio kernel. Elas são mais flexiveis do que as threads do sistema e permitem que o aplicativo tenha mais controle sobre o gerenciamento de threads, incluindo o escalonamento e o controle de concorrencia. No entanto, o sistema operacional não esta ciente das threads de usuario e, portanto, não pode escalona-las de forma eficiente entre os nucleos de processamento.



Thread de Sistema(Thread do kernel) : são criadas e gerenciadas pelo kernel do sistema operacional. Elas são mais pesadas do que as threads de usuário e exigem uma mudança para o modo kernel para serem executadas

Programas multithread

São programas que usam multiplas threads para executar varias tarefas simultaneamente em um mesmo processo. Os sistemas operacionais modernos permitem que os programas sejam executados de forma concorrente, usando a tecnologia de threads.

Os programas multithreads oferecem diversas vantagens, como melhor aproveitamento dos recursos de processamento e a possibilidade de manter a interface grafica do usuario responsivo, enquanto o programa executa tarefas em segundo plano.

Porem, a programação multithread requer cuidados especiais com relação a sincronização do acesso aos recursos compartilhados e prevencao de condições de corrida, que podem levar a erros de execução e falhas no programa. Alem disso, programas multithread podem ser mais dificeis de depurar e testar devido a complexidade adicional introduzida pela concorrência

Comunicação e Sincronização de Processos

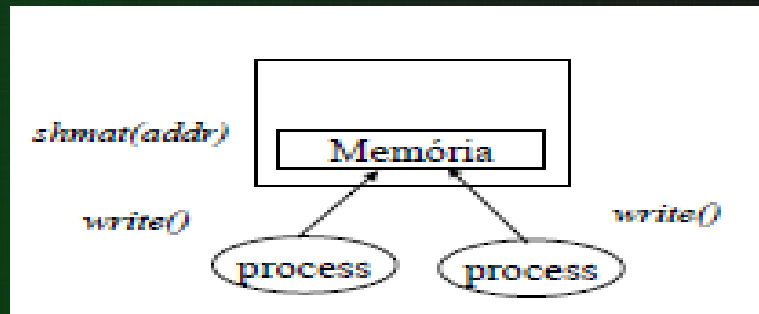
Em uma aplicação concorrente, ou na própria execução do sistema, muitas vezes é necessário que processos ou *threads* troquem dados entre si.

Tal comunicação pode ser implementada de diversas formas, por exemplo, usando um arquivo compartilhado, memória compartilhada, ou por troca de mensagens. Nesses casos, é necessário que os processos tenham suas execuções sincronizadas.

Comunicação e sincronização entre processos: duas abordagens.

1. Baseada em memória compartilhada

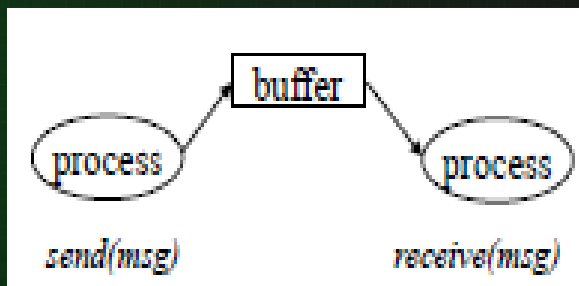
- Comunicação é implícita (dados compartilhados, sem canal de comunicação);
- Sincronização para acesso precisa ser feita explicitamente



Comunicação e sincronização entre processos: duas abordagens.

2. Baseada em troca de mensagens:

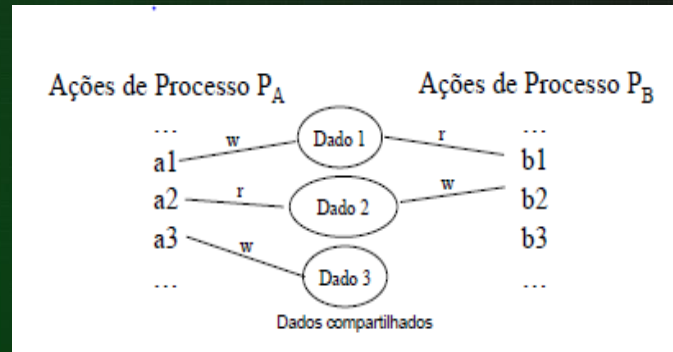
- Canal de comunicação é explícito;
- Sincronização é implícita (processos bloqueiam nas primitivas)



Condição de Corrida

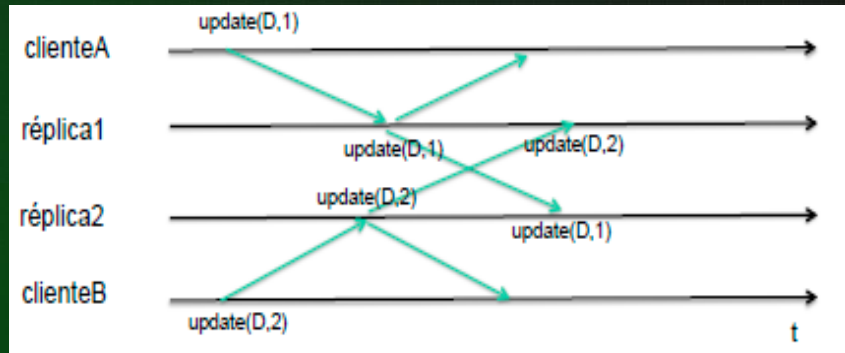
Ocorre sempre que:

- Cada processo/*thread* precisa executar uma sequência de ações (a_1, \dots, a_N) que envolvem mais de um dado/recurso compartilhado;
- Os dados/recursos precisam manter um estado consistente entre si;
- Antes que complete a execução de toda a sequência de ações, um dos processos é interrompido pelo outro



Para troca de mensagens (entre processos clientes e processos servidores)

1. Suponha um serviço tolerante a falhas que seja implementado com replicação (distribuída) de dados.
2. Assim que recebe uma requisição, um servidor responde ao cliente e envia a atualização seu dados para o outro servidor, que atualiza seu dado correspondentemente.



Região Crítica

Região crítica (ou Sessão crítica) é uma parte do código (programa) que contém as operações sobre dados compartilhados, a serem executadas em exclusão mútua.

Requisitos básicos que precisam ser satisfeitos para implementar a regiões críticas

Exclusão Mútua: Se um processo P está executando sua região crítica nenhum outro poderá executar a sua região crítica (segurança = safety).

Progresso: Um processo executando dentro da região crítica não pode ser bloqueado/terminado por outro processo (progresso = liveness).

Espera Limitada: Um processo não deve esperar indefinidamente para entrar em sua região crítica (justiça);

Requisitos básicos que precisam ser satisfeitos para implementar a regiões críticas

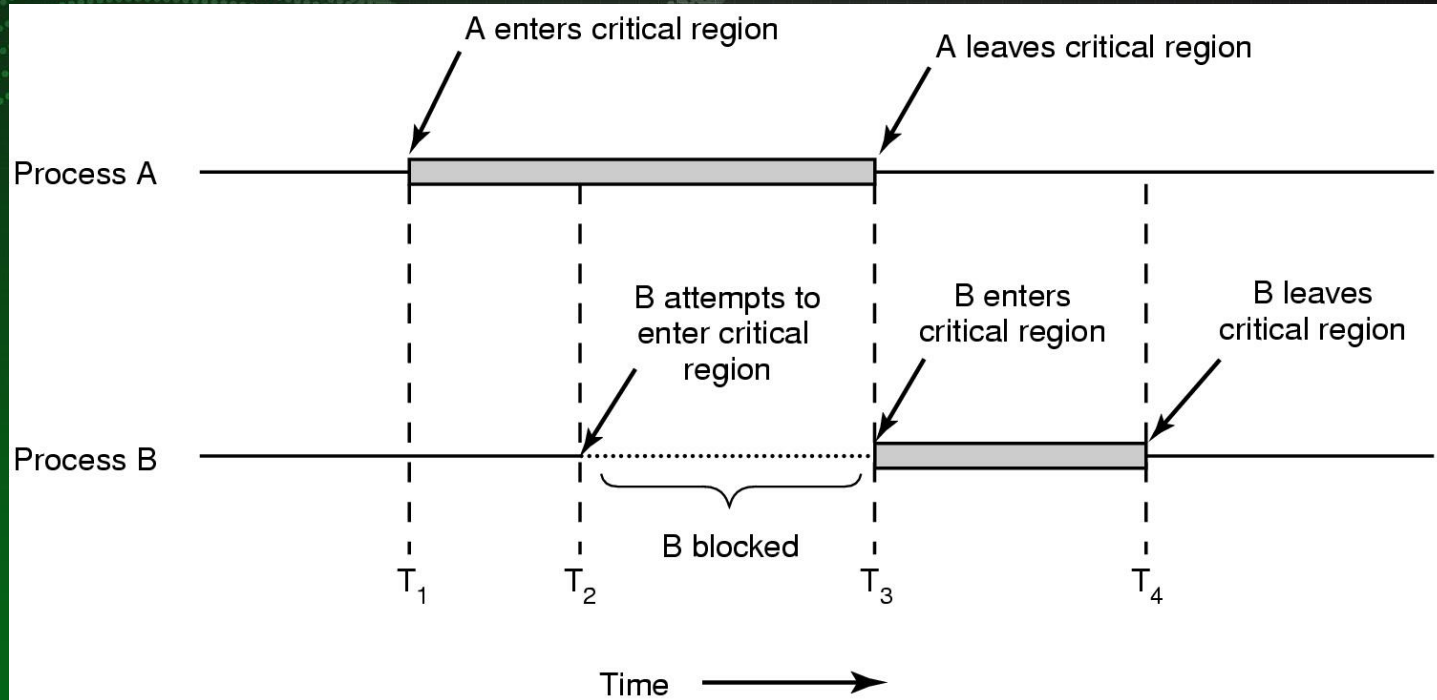
Para implementar uma região crítica deve haver um mecanismo/protocolo para garantir a entrada e saída segura (sincronizada, coordenada) desta parte do código.

Código em um processo:

```
...  
Enter-region;           // bloqueia se outro processo estiver dentro  
instr 1;  
instr 2;  
instr 3;  
...  
Exit-region;           // sai da região, e libera outros processos esperando  
...
```

} Região crítica

Região Crítica



Exclusão mútua usando Regiões Críticas

Possíveis formas de Implementar Regiões Críticas

Alternativas:

Desabilitar interrupções:

- Pode ser feito pelo núcleo, mais não por um processo em modo usuário.

Processos compartilham uma flag “lock”:

- Se lock=0, trocar valor para 1 e processo entra RC, senão processo espera;
- Se leitura e atribuição do lock não for atômica, então exclusão mutua não é garantida.

Espera Ocupada:

- Alternância regular de acesso por dois processos (PID= 0; PID= 1);
- Só funciona para 2 processos e apenas se a frequência de acesso ao recurso for mais ou menos a mesma

Cont. Espera Ocupada:

```
while (TRUE) {  
    while (turn != 0)    /* loop */ ;  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

```
while (TRUE) {  
    while (turn != 1)    /* loop */ ;  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

Região Crítica com Espera Ocupada

```
#define FALSE 0
#define TRUE 1
#define N      2                /* number of processes */

int turn;                       /* whose turn is it? */
int interested[N];              /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;                  /* number of the other process */

    other = 1 - process;        /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;             /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```


Solução de Peterson:

Variável `turn` e o vector `interested[]` são variáveis compartilhadas;

Se dois processos $PID = \{0,1\}$ executam simultaneamente `enter_region`, o primeiro valor de `turn` será sobrescrito (e o processo correspondente vai entrar), mais `interested[first]` vai manter o registro do interesse do segundo processo;

Exclusão Mútua com Espera Ocupada

Algumas arquiteturas possuem uma instrução de máquina especial:

Test- and- Set- Lock (TSL):

Algumas arquiteturas possuem uma instrução de máquina TSL para leitura de um lock para um registrador e atribuição de um valor diferente de zero ($\neq 0$) de forma atômica!

Exclusão Mútua com Espera Ocupada

Processos que desejam entrar RC executam TSL:

- Se registrador (e lock= 0), então entram na RC, senão esperam em loop;

```
enter_region:
    TSL REGISTER,LOCK           | copy lock to register and set lock to 1
    CMP REGISTER,#0             | was lock zero?
    JNE enter_region            | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0                | store a 0 in lock
    RET | return to caller
```

Exclusão Mútua com Espera Ocupada

Espera Ocupada vs. Bloqueio:

- Solução de Peterson e Test- and – Set- Lock apresentam o problema da espera ocupada que consome ciclos de processamento;
- Outro problema da espera Ocupada: Inversão de prioridades;
- Se um processo com baixa prioridade estiver na RC, demora mais a ser escalonado (e sair da RC), pois processos de maior prioridade estarão executando em espera ocupada (para a RC).

Exclusão Mútua com Espera Ocupada

A alternativa:

Chamadas ao núcleo que bloqueiam o processo e o fazem esperar por um sinal de outro processo.

Exemplo:

Lock/ unlock (Mutex);

Wait/Signal;

Semáforos.

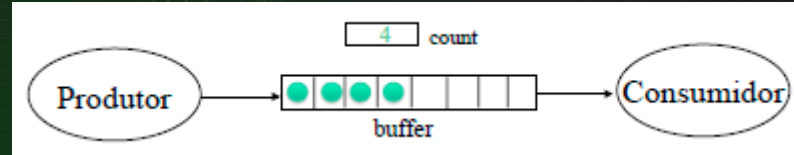
Problema do Produtor e Consumidor

Sincronização de dois processos que compartilham um buffer (um produz itens, o outro consome itens do buffer), e que usam uma variável compartilhada count para controlar o fluxo de controle.

- Se $\text{count} = N$, produtor deve esperar;
- Se $\text{count} = 0$ consumidor deve esperar.

Cada processo deve “acordar” o outro processo quando o conteúdo do buffer mudar a ponto de permitir o prosseguimento do processamento.

Problema do Produtor e Consumidor



Esse tipo de sincronização está relacionada ao estado do recurso -> Sincronização de condição.

Primitivas de Sincronização

A sincronização é a maneira de garantir que as tarefas sejam executadas de maneira organizada e previsível, evitando problemas como condições de corrida, em que duas ou mais tarefas tentam acessar o mesmo recurso ao mesmo tempo.

Na programação concorrente, a sincronização pode ser alcançada por meio de várias técnicas, como semáforos, monitores, mutexes e barreiras de sincronização. Essas técnicas ajudam a garantir que as tarefas sejam executadas de maneira ordenada, permitindo que elas compartilhem recursos de maneira segura e eficiente.

Por exemplo, se duas threads precisam acessar um arquivo simultaneamente, a sincronização garante que apenas uma thread tenha acesso ao arquivo de cada vez. Isso evita que os threads escrevam no arquivo ao mesmo tempo e acabem sobrescrevendo informações importantes.

Semáforos

Os semáforos são uma das primitivas de sincronização mais utilizadas em programação concorrente. Eles consistem em uma variável inteira que é usada para controlar o acesso a um recurso compartilhado entre as threads ou processos. Os semáforos possuem duas operações básicas: `wait()` e `signal()`.

`wait()` é usada para solicitar o acesso ao recurso compartilhado, e a operação `signal()` é usada para liberar o acesso ao recurso compartilhado.

Mutexes

Um mutex é uma primitiva de sincronização usada em programação concorrente para controlar o acesso a um recurso compartilhado, garantindo que apenas uma thread por vez possa acessá-lo. A palavra "mutex" é uma abreviação de "mutual exclusion", que significa exclusão mútua.

Um mutex possui dois estados possíveis: bloqueado e desbloqueado. Quando uma thread deseja acessar um recurso protegido por um mutex, ela primeiro tenta bloqueá-lo. Se o mutex estiver desbloqueado, a thread o bloqueia e pode acessar o recurso. Se o mutex já estiver bloqueado por outra thread, a thread atual é colocada em espera até que o mutex seja desbloqueado. A operação de desbloqueio do mutex é chamada de "unlock", que é realizada quando a thread que bloqueou o mutex termina de usar o recurso compartilhado e libera o mutex para que outra thread possa acessá-lo.

Monitores

Monitores são estruturas de sincronização que garantem o acesso exclusivo a um recurso compartilhado por um conjunto de threads. Os monitores possuem dois componentes principais: um bloqueio (ou lock) e uma condição (ou condition variable). O bloqueio é usado para garantir que apenas uma thread possa acessar o recurso compartilhado em um determinado momento. A condição é usada para que as threads esperem até que o recurso esteja disponível.

Quando uma thread quer acessar o recurso compartilhado, ela primeiro adquire o bloqueio do monitor. Se o recurso estiver disponível, a thread pode acessá-lo e depois liberar o bloqueio. Se o recurso estiver em uso por outra thread, a thread atual espera na condição até que o recurso esteja disponível novamente. Quando a thread que está usando o recurso o libera, ela notifica as threads que estão esperando na condição, permitindo que elas tentem acessar o recurso novamente.

Barreiras

Barreiras de sincronização, também conhecidas como barreiras de memória, são mecanismos utilizados na programação concorrente para garantir que as operações realizadas por uma thread em um determinado ponto do programa estejam visíveis para outras threads antes que elas possam prosseguir com suas próprias operações.

Essencialmente, uma barreira de sincronização cria um ponto de sincronização no programa, no qual todas as threads que atingiram esse ponto precisam esperar até que todas as outras threads também tenham chegado nesse ponto antes que possam prosseguir. Isso é importante para garantir que todas as operações realizadas antes da barreira tenham sido concluídas antes que outras operações dependentes possam começar.

As barreiras de sincronização são especialmente úteis em cenários em que várias threads estão trabalhando em conjunto para realizar uma tarefa complexa, como processamento de imagens ou renderização gráfica.

Uma desvantagem das barreiras de sincronização é que elas podem levar a um atraso no desempenho, especialmente em cenários em que as threads precisam esperar por outras threads para concluir suas operações antes de prosseguir.

Problemas clássicos de programação concorrente

Os problemas clássicos de coordenação retratam situações de coordenação entre tarefas ocorrem com muita frequência na programação de sistemas complexos, permitindo assim compreender como podem ser implementadas suas soluções. Alguns problemas clássicos são:

Produtores/consumidores

Descrição do problema

Este problema também é conhecido como o problema do buffer limitado, e consiste em coordenar o acesso de tarefas (processos ou threads) a um buffer compartilhado com capacidade de armazenamento limitada a N itens (que podem ser inteiros, registros, mensagens, etc.). São considerados dois tipos de processos com comportamentos cíclicos e simétricos:

Problemas clássicos de programação concorrente

Produtor: produz e deposita um item no buffer, caso o mesmo tenha uma vaga livre. Caso contrário, deve esperar até que surja uma vaga. Ao depositar um item, o produtor “consome” uma vaga livre.

Consumidor: retira um item do buffer e o consome; caso o buffer esteja vazio, aguarda que novos itens sejam depositados pelos produtores. Ao consumir um item, o consumidor “produz” uma vaga livre no buffer.

Deve-se observar que o acesso ao buffer é bloqueante, ou seja, cada processo fica bloqueado até conseguir fazer seu acesso, seja para produzir ou para consumir um item.

Problemas clássicos de programação concorrente

A solução do problema dos produtores/consumidores envolve três aspectos de coordenação distintos e complementares:

- A exclusão mútua no acesso ao buffer, para evitar condições de disputa entre produtores e/ou consumidores que poderiam corromper o conteúdo do buffer.
- A suspensão dos produtores no caso do buffer estar cheio: os produtores devem esperar até que surjam vagas livres no buffer.
- A suspensão dos consumidores no caso do buffer estar totalmente vazio: os consumidores devem esperar até que as vagas no buffer sejam preenchidas.

Problemas clássicos de programação concorrente

1. Jantar dos Filósofos

Definição do problema

Há cinco filósofos em torno de uma mesa um garfo é colocado entre cada filósofo;

Cada filósofo deve, alternadamente, refletir e comer, para que um filósofo coma, ele deve possuir dois garfos e os dois garfos devem ser aqueles logo a sua esquerda e a sua direita;

Para pegar um garfo somente pode ser pego por um filósofo e somente pode ser pego não estiver em uso por nenhum outro filósofo após comer, o filósofo deve liberar o garfo que utilizou, um filósofo pode segurar o garfo da sua direita ou o da sua esquerda assim que estiverem disponíveis mas, só pode começar a comer quando ambos estiverem sob sua posse.

Problemas clássicos de programação concorrente

1. Jantar dos Filósofos

Proposta de Solução do problema

A listagem a seguir é o pseudocódigo de uma implementação do comportamento básico dos filósofos, na qual cada filósofo é uma tarefa e cada palito (substituindo o garfo por hashis para os chineses) é um semáforo:

Problemas clássicos de programação concorrente

1. Jantar dos Filósofos

```
1 #define NUMFILO 5
2 semaphore hashi [NUMFILO] ; // um semáforo para cada palito (iniciam em 1)
3
4 task filosofo (int i)          // filósofo i (entre 0 e 4)
5 {
6     int dir = i ;
7     int esq = (i+1) % NUMFILO ;
8
9     while (1)
10     {
11         meditar () ;
12         down (hashi [dir]) ;      // pega palito direito
13         down (hashi [esq]) ;     // pega palito esquerdo
14         comer () ;
15         up (hashi [dir]) ;       // devolve palito direito
16         up (hashi [esq]) ;      // devolve palito esquerdo
17     }
18 }
```

Problemas clássicos de programação concorrente

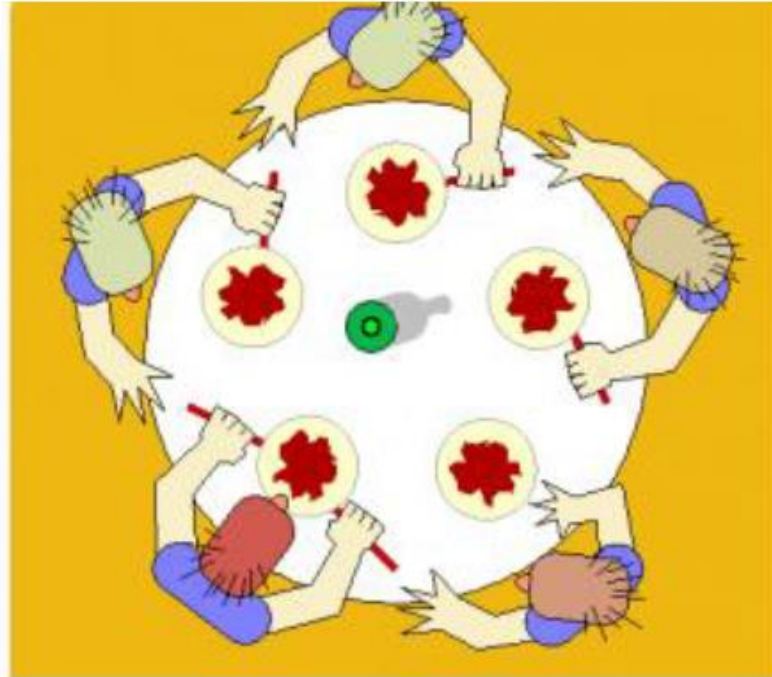
Como podemos ver, soluções simples para esse problema podem provocar impasses, ou seja, situações as quais todos os filósofos ficam bloqueados. Outras soluções podem provocar inanição (starvation), ou seja, alguns dos filósofos nunca conseguem comer.

Proposta de solução

Usando semáforos!

Após a estágio de pensamento tem-se ou a captura dos recursos necessário para comer ou o aguarde de uma notificação de que está apto a comer.

O jantar dos filósofos



Construções Concorrentes de Alto Nível

Construções concorrentes de alto nível são abstrações de programação que ajudam a simplificar a criação e gerenciamento de processos e threads em um programa concorrente. Algumas das construções concorrentes de alto nível mais comuns incluem:

- **Threads:** Uma thread é uma unidade básica de processamento que pode ser executada em paralelo com outras threads dentro de um mesmo processo. As threads compartilham o mesmo espaço de endereço e recursos do processo pai, o que as torna mais leves do que os processos.
- **Processos:** Um processo é uma instância de um programa que pode ser executada de forma independente de outros processos. Os processos têm seu próprio espaço de endereço e recursos, o que os torna mais pesados que as threads, mas também mais isolados e seguros.

Construções Concorrentes de Alto Nível

- Canais: Um canal é uma abstração de comunicação entre processos e threads que permite que os dados sejam trocados de forma assíncrona e segura. Os canais podem ser usados para enviar e receber mensagens ou dados entre as tarefas concorrentes.
- Semáforos: Um semáforo é uma estrutura de dados que pode ser usada para controlar o acesso concorrente a recursos compartilhados. Os semáforos são usados para garantir que apenas uma thread ou processo possa acessar um recurso compartilhado por vez.

Construções Concorrentes de Alto Nível

Mutexes: Um mutex é uma abstração de programação que pode ser usada para sincronizar o acesso concorrente a recursos compartilhados. Os mutexes permitem que apenas uma thread ou processo possa acessar um recurso compartilhado por vez.

Programação orientada a eventos: utiliza eventos como gatilhos para a execução de código concorrente, eliminando a necessidade de gerenciar threads ou processos.

Programação reativa: permite que os sistemas reajam a eventos externos de forma assíncrona, por meio da observação de fluxos de eventos.

Construções Concorrentes de Alto Nível

Essas são apenas algumas das construções concorrentes de alto nível disponíveis na programação concorrente. A escolha das construções adequadas depende do problema a ser resolvido e das características da linguagem de programação e do sistema operacional em uso. Essas construções são comumente usadas em linguagens de programação como Java, Python, C++ e outras que suportam programação concorrente.



Obrigado pela atenção
dispensada