



# SISTEMAS OPERACIONAIS

## Módulo 3 – Threads

Prof. Daniel Sundfeld  
[daniel.sundfeld@unb.br](mailto:daniel.sundfeld@unb.br)



# AULA PASSADA

- Definição de Processos
- Diferenciar Processo e Programa
- Tabela de Processos
- Escalonamento de Processos
- Troca de contexto
- Criação/Término/Hierarquia de processos



# AULA PASSADA

- Estados: Rodando/Bloqueado/Pronto
- Preempção/Não-Preempção
- Algoritmos de Escalonamento
  - FCFS
  - Round-Robin
  - Prioridades
  - Shortest Job First



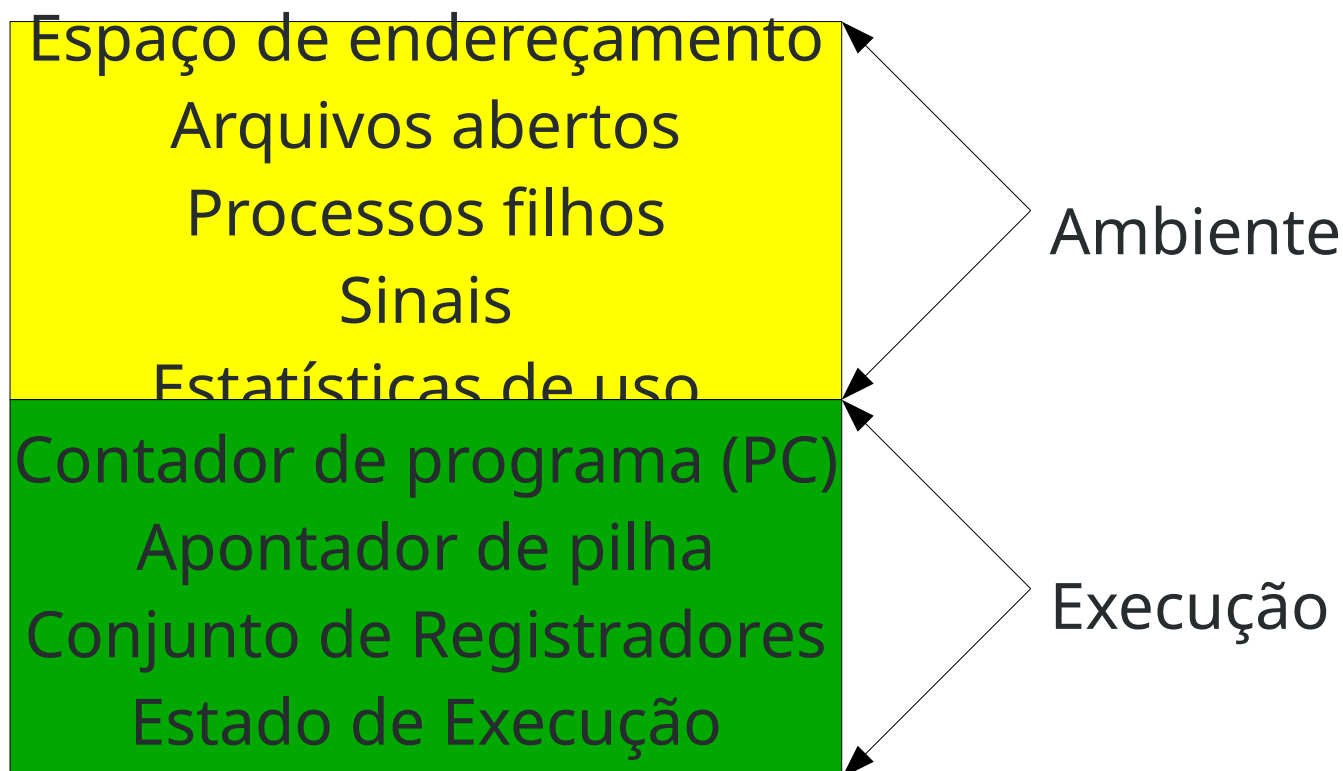
# MODELO DE PROCESSO

- Classificação dos modelos de processos quanto ao custo de troca de contexto e de manutenção
  - Heavyweight (processo tradicional)
  - Lightweight (threads)



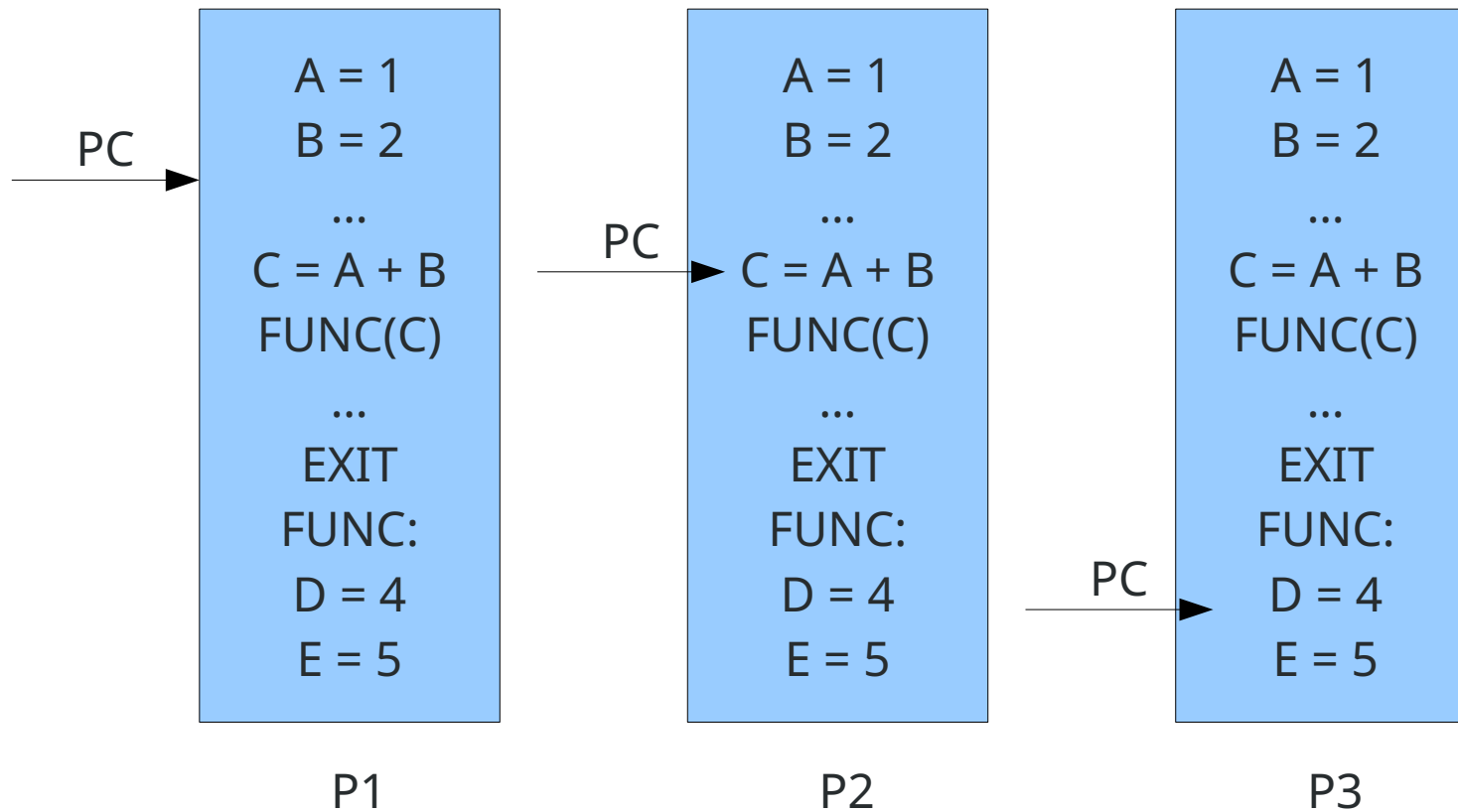
# MODELO DE PROCESSO

- Informações relevantes a respeito de um processo:





# MODELO DE PROCESSO TRADICIONAL





# MODELO DE PROCESSO TRADICIONAL

- A troca de contexto entre processos tradicionais é pesada para o sistema.
  - Contexto = ambiente + execução
- Processos tradicionais não compartilham memória
- Possuem uma única thread (fluxo) de controle



# Threads

- As threads separam os conceitos de agrupamento de recursos e execução
- Processos agrupam recursos
- Threads são escalonadas para execução
- Permitem que múltiplas execuções ocorram no mesmo ambiente do processo com um grau de independência entre elas
- Multithread: termo para descrever ambiente com mais de uma thread





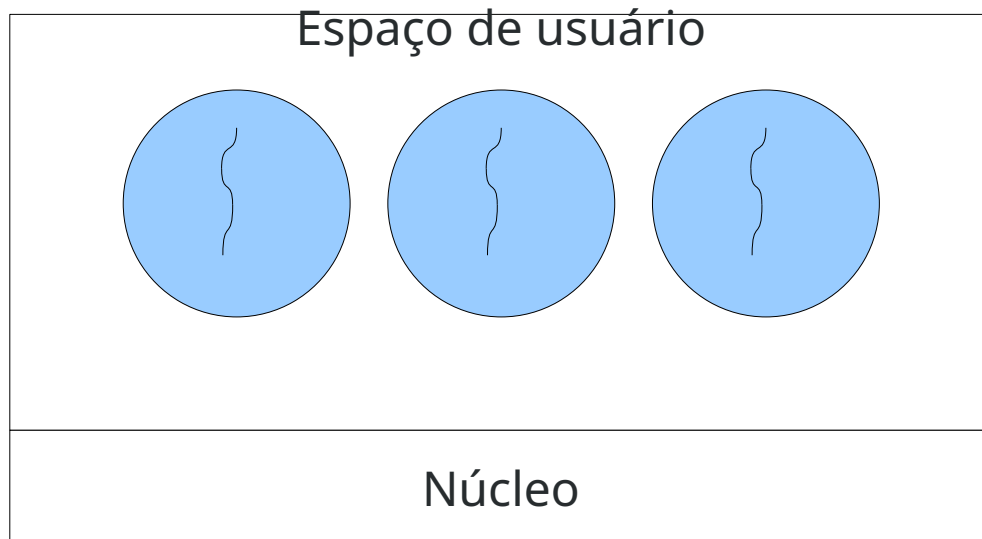
# THREADS

- Devido ao maior simplicidade de escalonamento, também são chamadas de “Processos leves”
- No modelo multithread, a entidade processo é dividida em processo e thread.
- O processo corresponde ao ambiente
- Thread corresponde ao estado de execução
- Um processo é composto por várias threads que compartilham o ambiente: memória, descritor de arquivos, entre outros

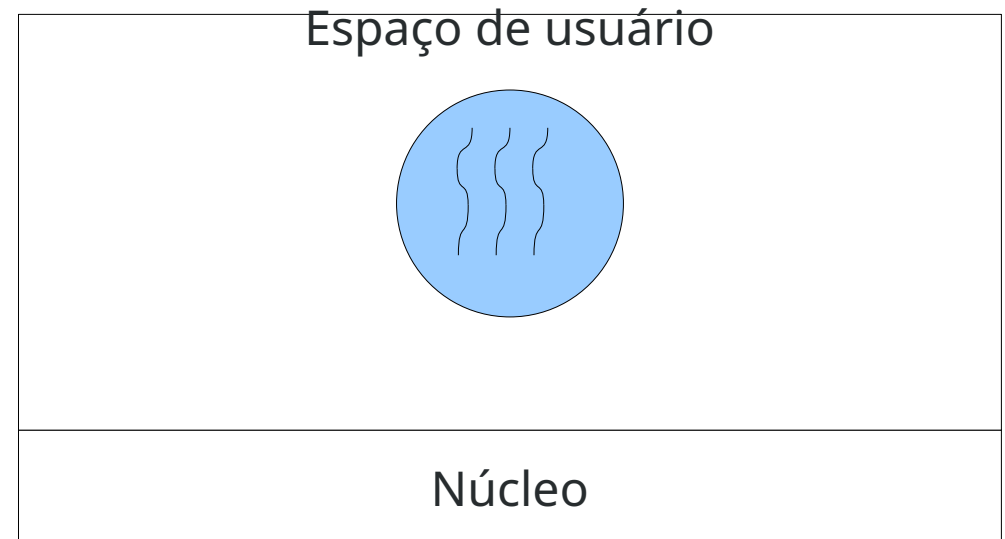


# THREADS

- Três processos com uma thread vs. um processo com três threads:



Modelo tradicional, cada thread possui seu espaço de endereçamento e sua thread de controle



Modelo multithread, um processo possui vários fluxos de execução



# THREADS

- No modelo multithread, existem duas entidades na tabela de processo: processo que armazena as informações de ambiente e thread que armazena as informações de execução
- Desta forma, um processo é composto por diversas threads, cada uma possui seu contexto de hardware (execução) e compartilham o contexto de software (ambiente + espaço endereçamento)



# THREADS

- Uma thread pode se bloquear à espera de um recurso. Neste momento, uma outra thread pode se executar (ou uma thread de outro processo)
- A troca de contexto é mais leve
- Threads distintos em um processo não são tão independentes quanto processos distintos

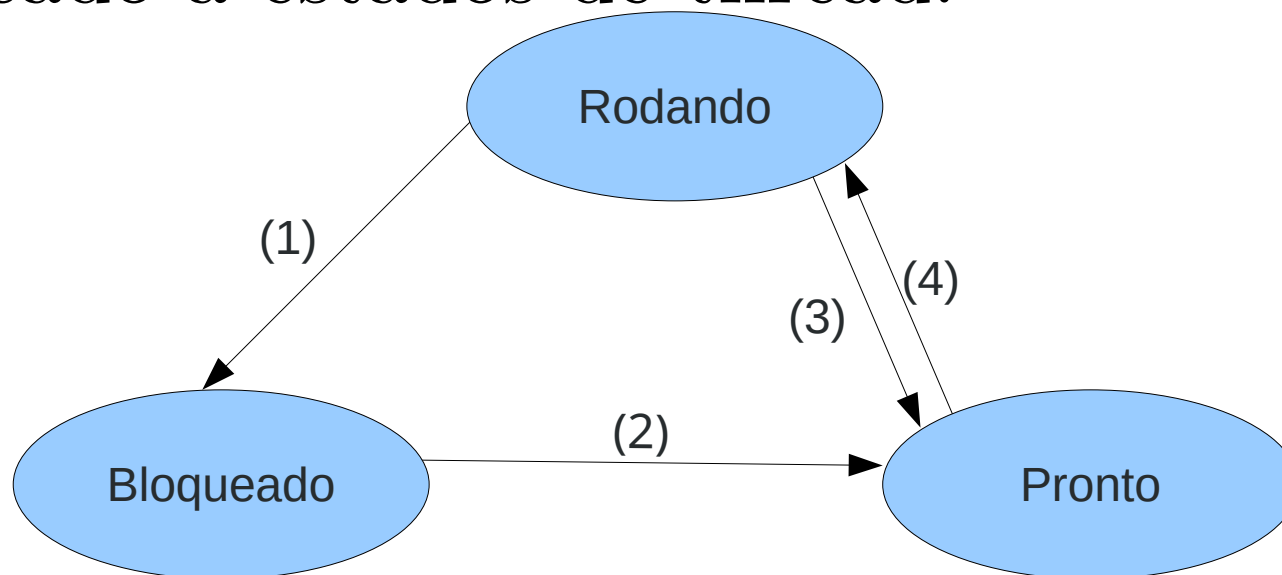


# THREADS

- Threads compartilham as mesmas variáveis globais
- Uma thread pode apagar completamente o que a outra está fazendo: esta proteção não é garantida pelo SO pois é impossível e desnecessário
- A proteção dos recursos entre threads é responsabilidade do programador
- São necessários mecanismos de sincronização

# ESTADOS DE THREADS

- O modelo de estados de processos é facilmente aplicado a estados de thread:



- (1) A thread bloqueia-se aguardando uma entrada
- (2) O evento aguardado pela thread ocorreu, pode-se iniciar a executar.
- (3) O tempo de posse do processador esgotou-se
- (4) A thread é escolhido pelo escalonador para executar



# THREADS

- A pilha é uma estrutura em memória, porém cada thread possui a sua própria pilha e não compartilha esses dados com as outras threads
- As threads chamam procedimentos diferentes, em tempos diferentes, resultando em uma história de execução diferente e por isso precisam de pilhas próprias



# THREADS

- Chamadas de controle de threads:
  - `thread_create`: cria uma thread nova, passando uma função como argumento para iniciar a execução
  - `thread_exit`: termina a thread em execução
  - `thread_yield`: permite que uma thread desista voluntariamente da CPU





# THREADS

- Exemplo de código sequencial:
- A primeira mensagem que imprime é “Oi!” ou “Ola!”?

```
int main()  
{  
    func1();  
    func2();  
    return 0;  
}
```

```
void func1()  
{  
    printf("Oi!\n");  
    return 0;  
}
```

```
void func2()  
{  
    printf("Ola!\n");  
    return 0;  
}
```



# THREADS

- Exemplo de código multi-thread:
- A primeira mensagem que imprime é “Oi!” ou “Ola!”?

```
int main()
{
    thread_create(func1);
    thread_create(func2);
    thread_exit(0);
}

void func1()
{
    printf("Oi!\n");
    thread_exit(0);
}

void func2()
{
    printf("Ola!\n");
    thread_exit(0);
}
```



# THREADS

- Exemplo, soma de vetores (1 thread)

```
int main()
{
    int i;
    int vetor1[100], vetor2[100], vetor3[100];
    inicializa vetores ...

    for (int i = 0; i < 100; i++)
    {
        vetor3[i] = vetor1[i] + vetor2[i];
    }
    return 0;
}
```



# THREADS

- Exemplo de soma de vetores: (2 threads):

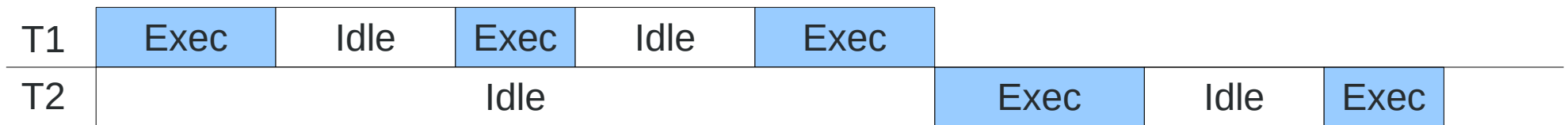
```
int main()
{
    int vetor1[100], vetor2[100], vetor3[100];
    inicializa vetores ...
    create_thread(thread, 0, 50, vetor1, vetor2, vetor3);
    create_thread(thread, 50, 100, vetor1, vetor2, vetor3);
    return 0;
}

int thread(int inicio, int fim, int* vet1, int* vet2, int* vet3)
{
    int i;
    for (i = inicio; i < fim; i++)
    {
        vet3[i] = vet1[i] + vet2[i];
    }
    return 0;
}
```



# POR QUE UTILIZAR MÚTIPLAS THREADS?

- Sem concorrência:



- Com concorrência



- Pelo mesmo motivo que processos concorrentes são melhores que sem concorrência
- Threads possuem troca de contexto mais leve que processos



# POR QUE UTILIZAR MÚLTIPLAS THREADS?

- Podemos dividir o programa caso exista muitas chamadas blocantes. Se as chamadas blocantes forem de diversas fontes, melhora-se o tempo
- Para explorar melhor os recursos da máquina: Os sistemas computacionais com múltiplos processadores são uma realidade hoje
- A Intel fez uma campanha avisando que a tendência é aumentar o número de núcleos, não a velocidade deles



# POR QUE UTILIZAR MÚLTIPLAS THREADS?

- Threads são mais fáceis de criar e destruir do que processos, afinal apenas área de execução precisa ser alocada
- Em alguns sistemas, criar uma thread é 100 vezes mais rápido que criar um processo
- Threads podem ser mais interessantes caso seja necessário criar várias dinamicamente



# POR QUE UTILIZAR MÚLTIPLAS THREADS?

- Um modelo de programação mais simples
- Quando um programa deve tratar dados de diversas fontes
  - Ex: ler e processar um arquivo, ler dados de rede, e receber informações do sistema operacional
- É possível criar um paradigma mais simples ao decompor múltiplas tarefas em diversas threads mais simples





# A ARTE DE MULTIPROGRAMAR

- Apesar de ser um conceito interessante e relativamente de fácil aplicação, o modelo multithread levanta diversas questões
- Fork(): quando o pai cria um processo filho, ele deve conter o mesmo tanto de threads que o pai ou apenas um? E se os filhos forem necessários?
- Escalonamento: quando uma thread estiver bloqueada esperando dados do teclado, ele deveria ser bloqueado?



# A ARTE DE MULTIPROGRAMAR

- O compartilhamento de dados pode causar muitos problemas
- O que acontece quando uma thread fecha um arquivo que outra thread está lendo?
- Alguns desafios são solucionados com boas práticas de programação



# A ARTE DE MULTIPROGRAMAR

- Projetar programas multithread que cooperem para resolver o mesmo problema é muito difícil
- Em sistemas modernos, normalmente as threads estão sujeitas a um escalonador preemptivo e que não sabemos como vai se comportar



# A ARTE DE MULTIPROGRAMAR

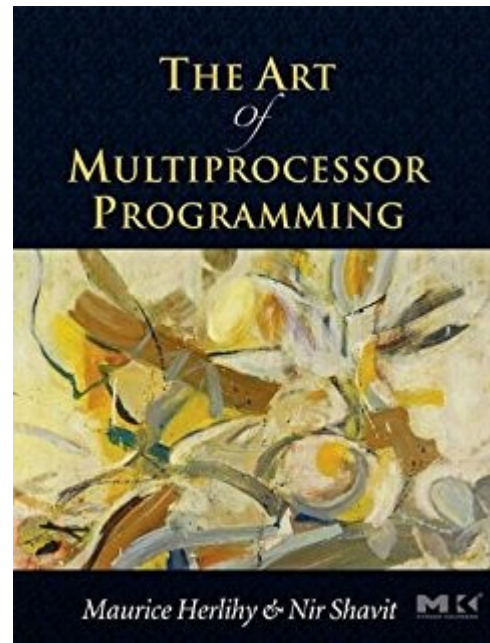
- “Multithreading Programming: Expectation vs. Reality”





# A ARTE DE MULTIPROGRAMAR

- Alguns autores brincam chamando a multiprogramação de uma arte e não uma ciência





# IMPLEMENTAÇÃO DE THREADS

- Para a implementação das threads, existem diferentes formas para sua implementação
- Implementar o modelo de processos e threads a nível de sistema operacional, criando abstrações de processos e de threads
  - O SO deve se tornar responsável por isso
- Implementar o modelo de processos heavyweight e simular múltiplas threads através de bibliotecas
  - Mais viável em SO com kernel não-monolítico



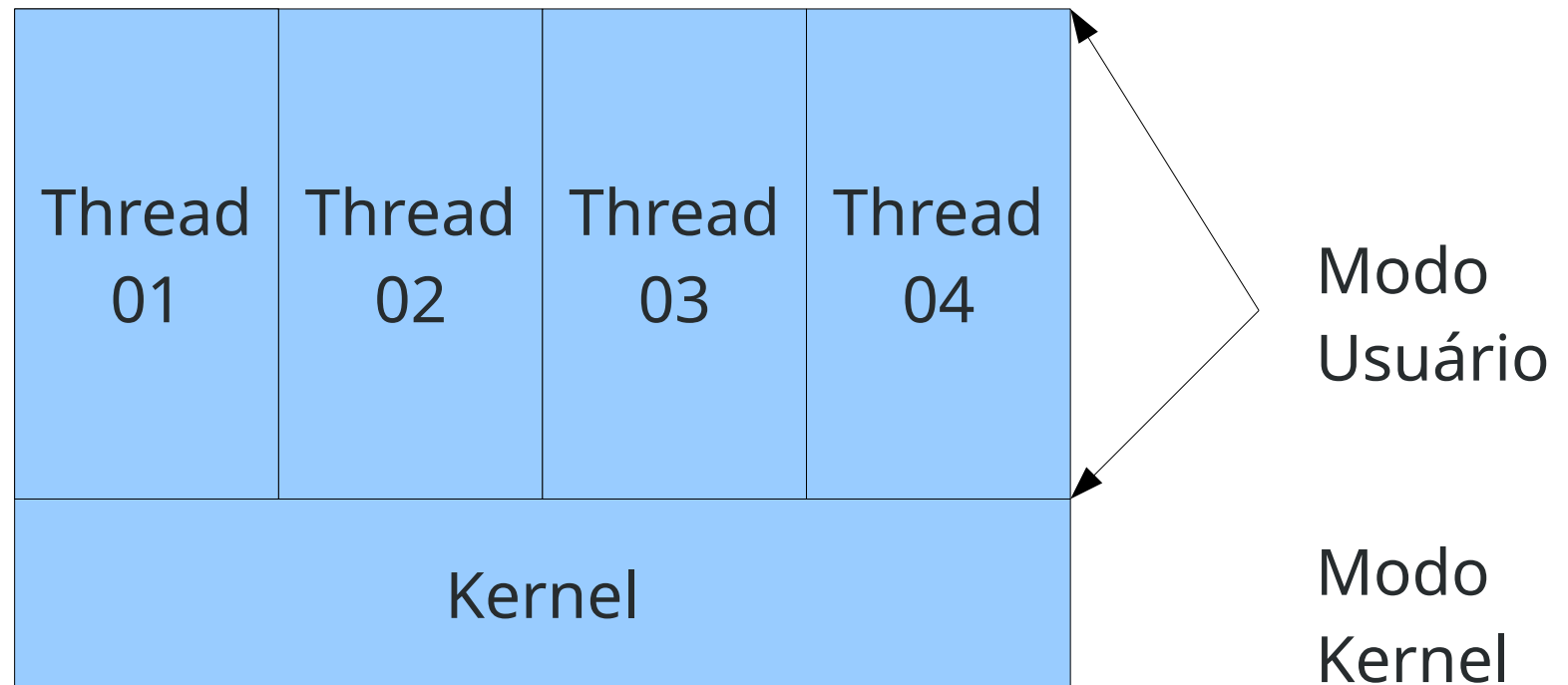
# IMPLEMENTAÇÃO DE THREADS

- Implementar threads no sistema operacional
- O kernel do sistema operacional deve criar as threads, organiza seu escalonamento e término
- Existência de uma tabela de threads no kernel, que contém os dados de cada thread
- Quando uma thread é bloqueada, o kernel é responsável por escalonar outra thread para rodar, mesmo que seja de outro processo



# IMPLEMENTAÇÃO DE THREADS

- Implementação de Threads em modo kernel:







# IMPLEMENTAÇÃO DE THREADS

- Implementação de threads de usuário
- As threads são simuladas no processo de usuário
- Cada processo precisa de sua própria tabela de threads
- Threads manipuladas por funções
- Geralmente, o escalonador do SO é não-preemptivo
- Quando uma thread for perder o controle, ela chama um procedimento do ambiente de execução para selecionar outra thread para executar



# IMPLEMENTAÇÃO DE THREADS

- Troca de contexto muito rápida entre as threads
- Cada processo pode ter seu próprio algoritmo de escalonamento. Muitas vezes, o tipo de algoritmo de escalonamento é melhor para certos problemas.
- Desvantagem: muito cuidado ao usar chamadas bloqueantes do sistema. Elas irão bloquear todas as threads



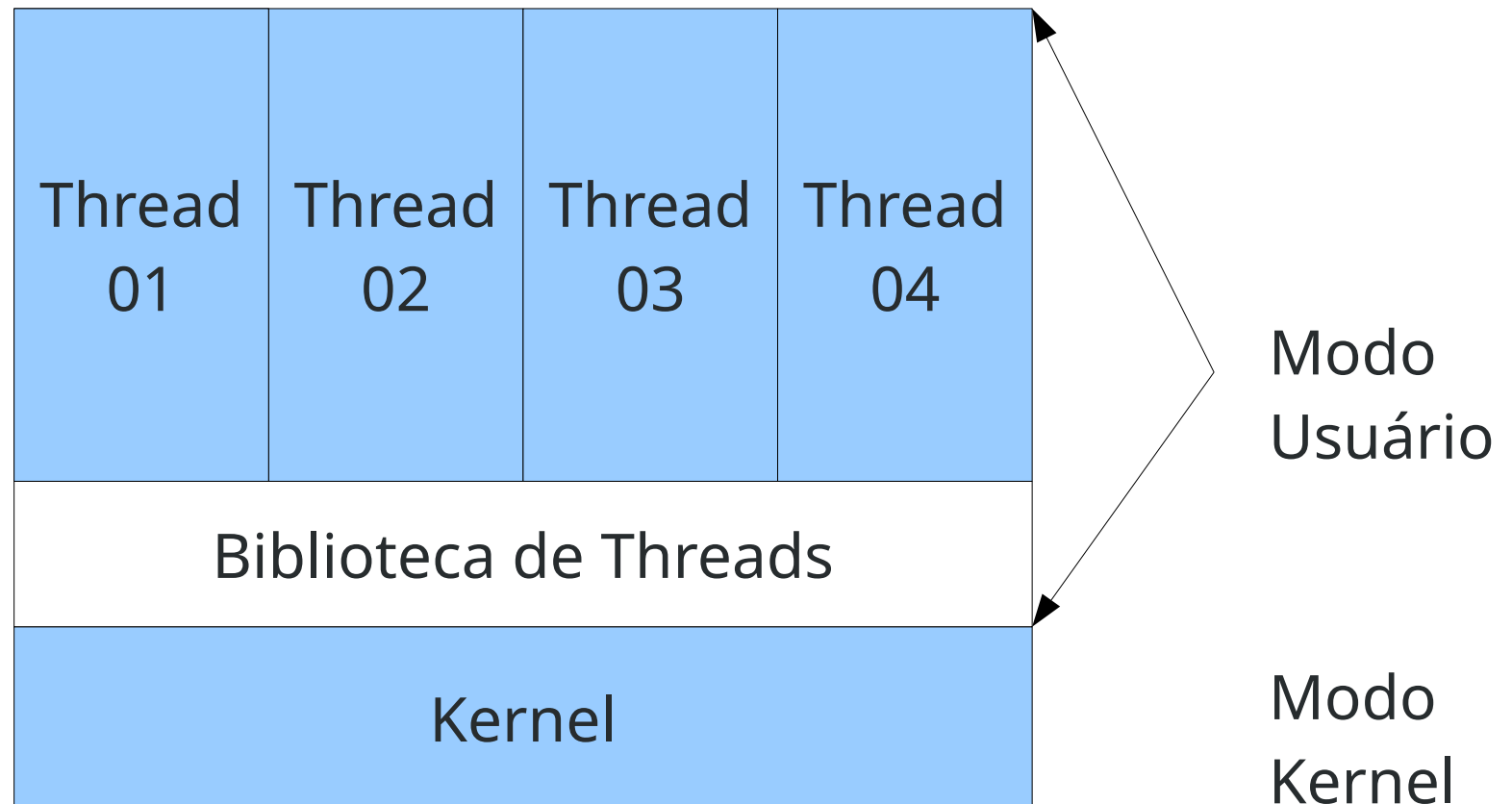
# IMPLEMENTAÇÃO DE THREADS

- Solução, deve se colocar uma “capa” antes de todas as chamadas blocantes do sistema
- Sendo assim se uma chamada blocante for realizada, ela é mascarada pela biblioteca de threads, que faz o teste de bloqueio
- Se a chamada realmente for bloquear, ela só é realizado caso não exista thread para executar
- Caso contrário, outra thread é executada
- Causa um overhead na execução de funções



# IMPLEMENTAÇÃO DE THREADS

- Implementação de Threads em modo usuário:



Fonte: adaptado de Machado & Maia 2013.



# IMPLEMENTAÇÃO DE THREADS

- Entre as diversas implementações tem algumas vantagens e desvantagens
- Implementar as threads em SO continua inserindo um custo caro de troca de contexto
- Enquanto utilizar threads no espaço de usuário reduz bastante o custo de troca, porém as operações de I/O exigem mais e podem ocasionar o bloqueio de todas as threads



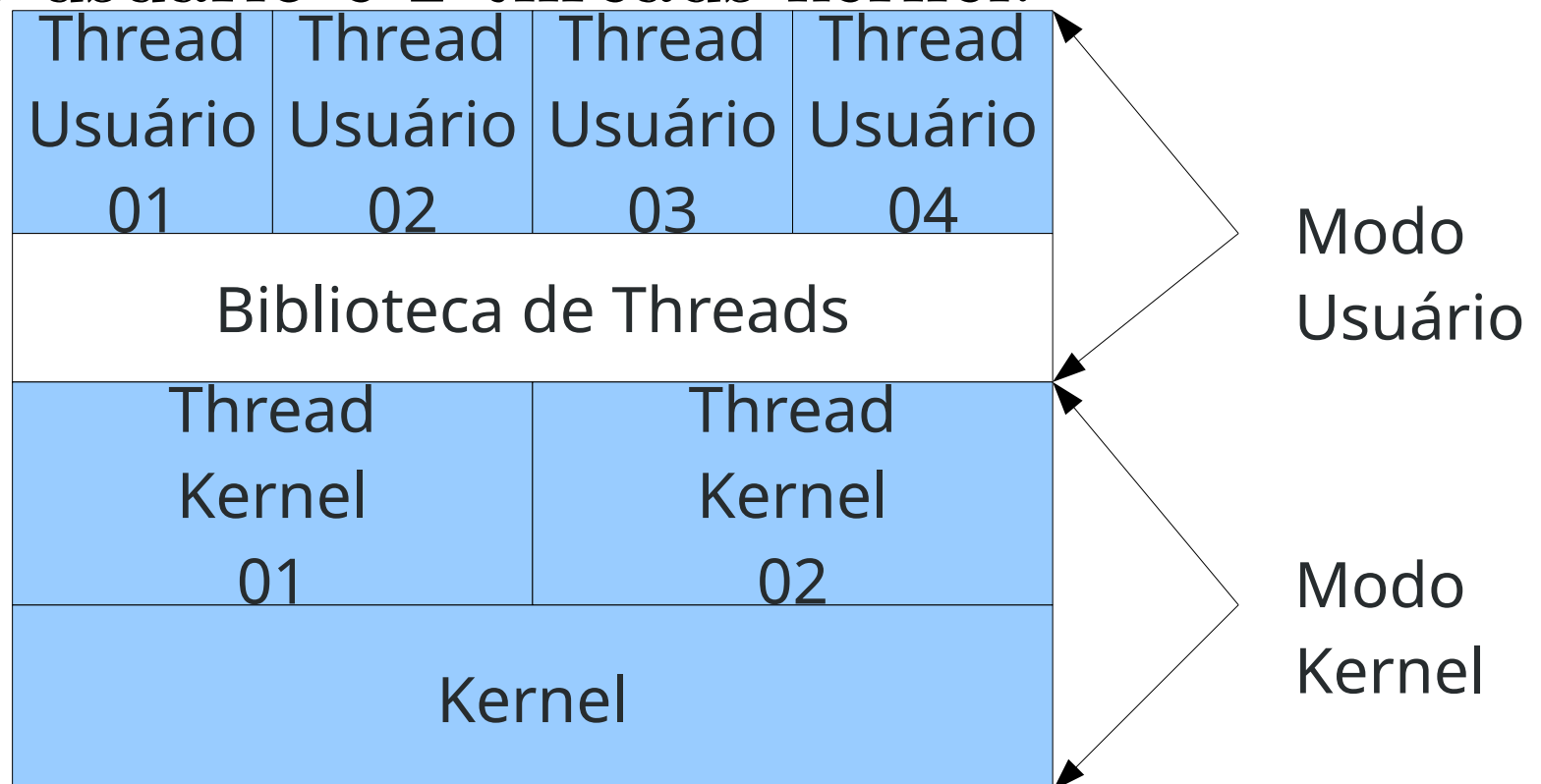
# THREADS EM MODO HÍBRIDO

- As arquiteturas de threads no modo híbrido busca combinar as vantagens das threads em modo usuário e threads em modo kernel
- Um processo pode ter várias threads de kernel
- Por sua vez, cada thread kernel pode conter diversas threads em modo usuário



# IMPLEMENTAÇÃO DE THREADS

- Implementação de Threads em modo híbrido 4 threads usuário e 2 threads kernel:





# IMPLEMENTAÇÃO DE THREADS

- Comparativo de threads em diversos SOs:

SO	Arquitetura
Distributed Computing Enviroment	Usuário
Compaq OpenVMS 6	Usuário
MS Windows 2000	Kernel
Compaq Unix	Kernel
Compaq OpenVMS 7	Kernel
Sun Solaris 2	Híbrido





# IMPLEMENTAÇÃO DE THREADS

- Pacote POSIX threads (pthreads)
- Exemplo de biblioteca amplamente utilizada para suportar as threads
- Inclui mecanismo de controle e sincronização
- Sua implementação varia de acordo com o SO utilizado, mas ela padroniza as chamadas de criação/manipulação entre diferentes sistemas operacionais



# MODELO DE EXECUÇÃO DE THREADS

- Existem alguns padrões que podem ser seguidos para a solução comum de criação e término de threads
- Thread dinâmicas, onde uma thread é criada para tratar cada requisição
- Thread estática: o número de threads é fixo



# MODELO

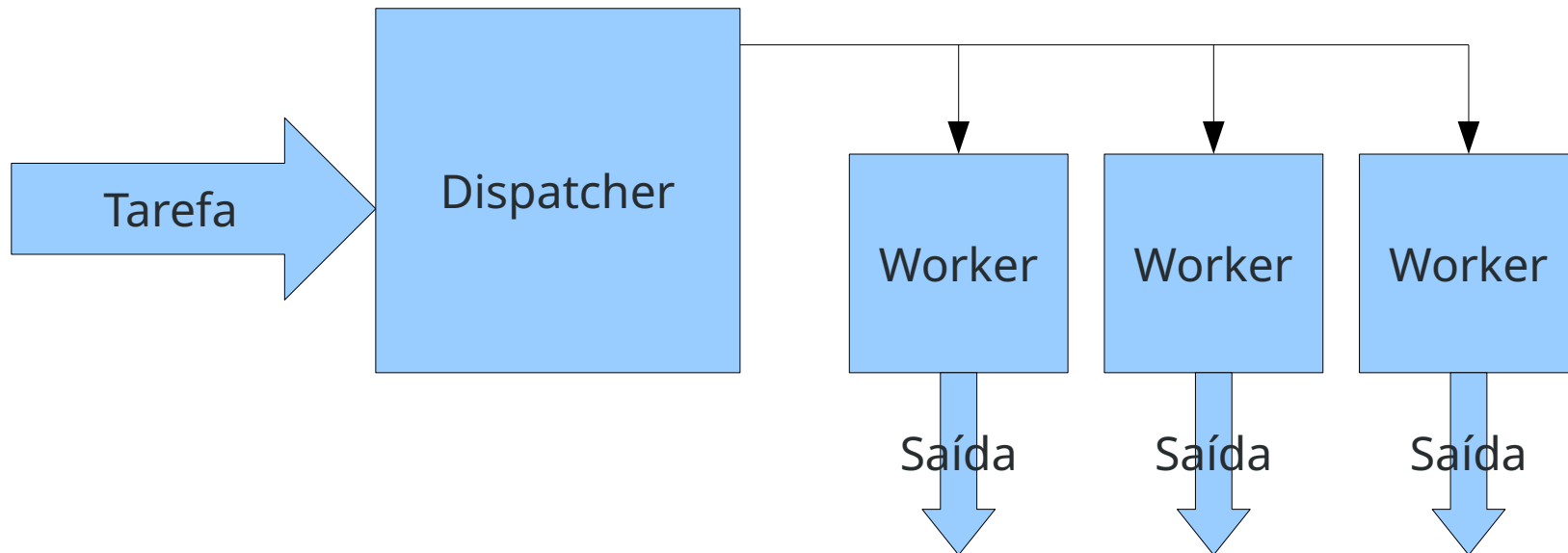
## DESPACHANTE/TRABALHADOR

- Uma thread despachante (Dispatcher) é responsável por receber o trabalho, porém não o processa
- O despachante seleciona uma thread trabalhadora para entregar o trabalho
- A thread trabalhadora executa a solicitação e sinaliza o dispatcher



# MODELO DESPACHANTE/TRABALHADOR

- $E_x$





# MODELO

## DESPACHANTE/TRABALHADOR

- Exemplo: servidor web
- Um servidor web recebe várias requisições de diversos clientes
- As requisições podem envolver leitura de disco
- Se a mesma thread é responsável por receber uma nova requisição e ler o disco, pode-se ter um problema. Especialmente se o equipamento de rede for mais rápido que o disco...



# MODELO

## DESPACHANTE/TRABALHADOR

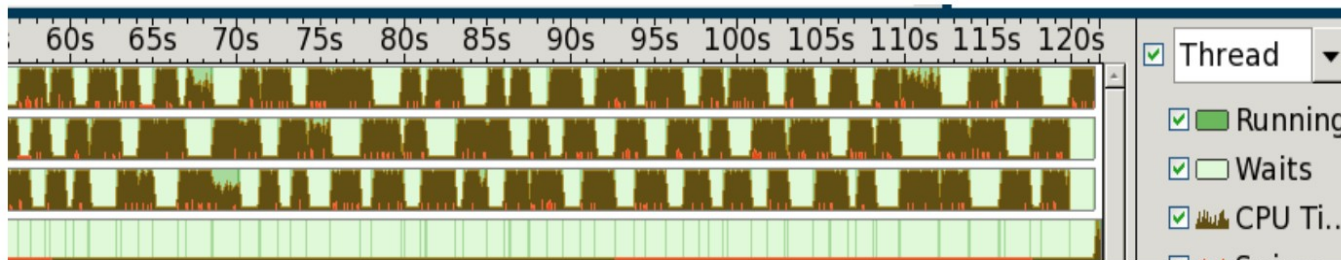
- Vantagens:
- Consumo rápido de mensagens
- Boa distribuição das requisições
- Flexibilidade: podemos facilmente mudar o número de threads



# MODELO

## DESPACHANTE/TRABALHADOR

- Desvantagem pouco uso de CPU pela thread despachante
- Ex: 4 threads trabalhadora e 1 thread despachante. Em alguns benchmarks pode considerar apenas 80% do uso total de CPU...

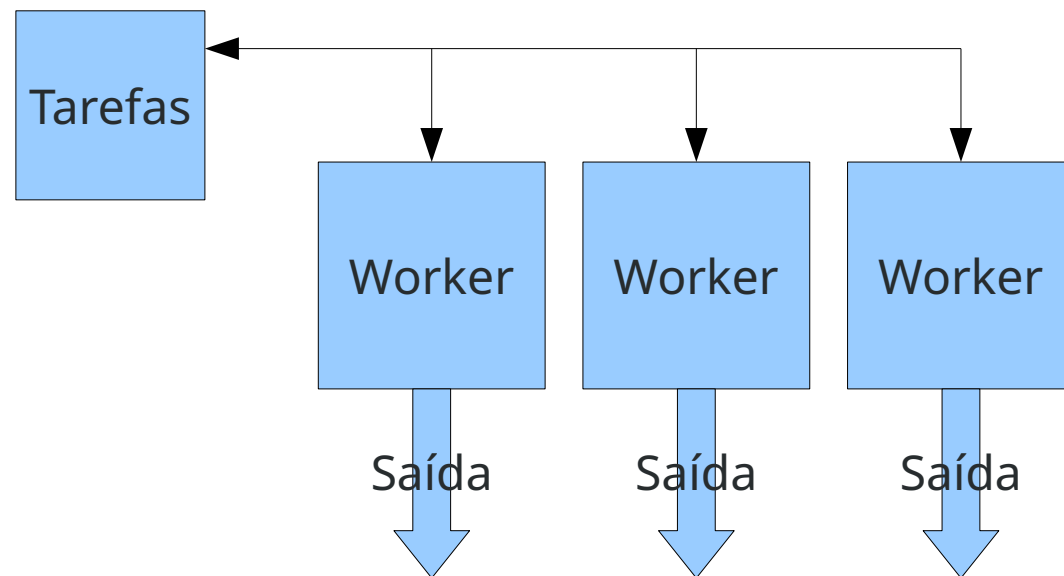


Fonte: Sundfeld D, Havgaard JH, Melo ACMA, Gorodkin J. Supplementary material for Foldalign 2.5: multithreaded implementation for pairwise structural RNA alignment. Bioinformatics. 2016;32(8):1238-1240



# MODELO TIME

- Nesse modelo as threads são autônomas e gulosas por serviço. Elas acessam um “poll” de tarefa, obtém e as executam







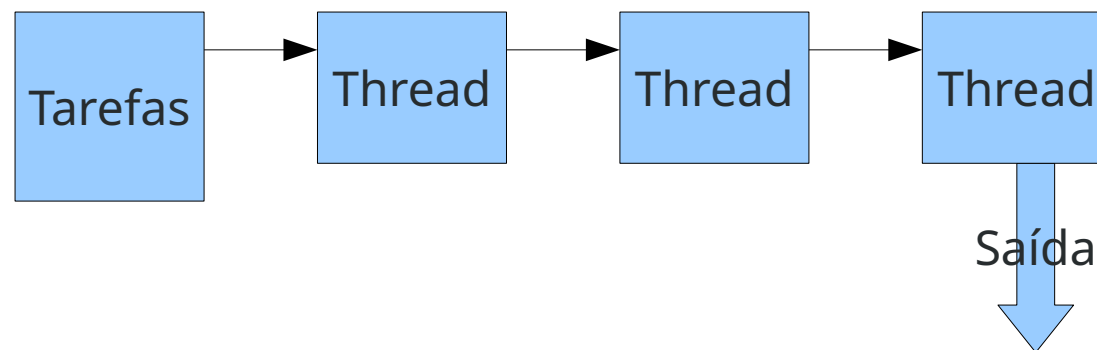
# MODELO TIME

- Vantagens:
- Bom consumo de mensagens
- Boa distribuição de requisições
- Flexibilidade em mudar o número de tarefas
- Desvantagens:
- Cuidado na implementação. Em computadores modernos, uma thread inicia o programa e deve ser responsável por criar todas as outras. Após isso, ela deve se tornar uma thread normal do time



# MODELO PIPELINE

- Cada thread realiza uma tarefa específica produzindo dados de entrada de outra thread. Os dados de saída final são produzidos pela última thread





# MODELO PIPELINE

- Desvantagens do pipeline: se uma thread for muito mais lenta que as outras, todo o processamento é desperdiçado
- Muitas vezes não é fácil dividir a tarefa em um pipeline



# A ARTE DE MULTIPROGRAMAR 2

- Convertendo código monothread em código multithread é uma tarefa árdua, especialmente quando lidamos com variáveis globais
- Ex: não-thread-safe / thread-safe

```
int vetor[256];
```

```
int thread(int val, int pos)
{
    vetor[pos] = val;
}
```

```
int thread(int val, int pos)
{
    int vetor[256];
    vetor[pos] = val;
}
```



# A ARTE DE MULTIPROGRAMAR 2

- Algumas variáveis globais foram definidas em sistemas Unix e são amplamente utilizadas! Por exemplo: errno
- Recebe dados via rede, verifica se vai bloquear

```
while (1) {  
    char buf[128];  
    int len = recv(socket_client, buf, sizeof(buf), 0);  
    if (len == -1) {  
        if (errno != EAGAIN && errno != EWOULDBLOCK) {  
            ...  
            return -1;  
        }  
    }  
}
```



# A ARTE DE MULTIPROGRAMAR 2

- Esse problema pode acontecer com chamadas de sistema! Por exemplo: aloca uma região de 10 inteiros na memória para o programa

```
int *p;  
p = malloc(sizeof(int)*10);
```

- Problema: e se houver troca de contexto no meio da chamada? O malloc salva informações em uma tabela global de memórias



# A ARTE DE MULTIPROGRAMAR 2

- Nem todos os problemas são causados apenas por variáveis globais
- Utilizar variáveis compartilhadas entre as threads requer cuidado
- Em computadores modernos, a ordem de execução é definida pelo SO
- Alguns desses problemas são conhecidos como Condições de Corrida



# CONDIÇÃO DE CORRIDA

- Também chamadas de condições de disputa
- Como o sistema operacional determinar através do seu escalonador como os processos irão executar, não sabemos a ordem que os processos podem executar
- Trocas de contexto podem acontecer a qualquer momento!!!





# CONDIÇÃO DE CORRIDA 1

- Considere os seguinte Processos/Thread incrementando uma variável em memória compartilhada

Processo / Thread A	Processo / Thread B
$x = x + 1$	$x = x + 1$

- Considere,  $x = 0$  inicialmente. Quais valores possíveis que  $x$  pode obter ao final?



# CONDIÇÃO DE CORRIDA 1

- Escalonamento:  $A \rightarrow B$
- Assumindo que X está na posição de memória

0x2000

Processo / Thread A

LOAD R1, 0x2000 (x=0)  
INC R1  
STORE R1, 0x2000 (x=1)

Processo / Thread B

LOAD R1, 0x2000 (x=1)  
INC R1  
STORE R1, 0x2000 (x=2)

Tempo

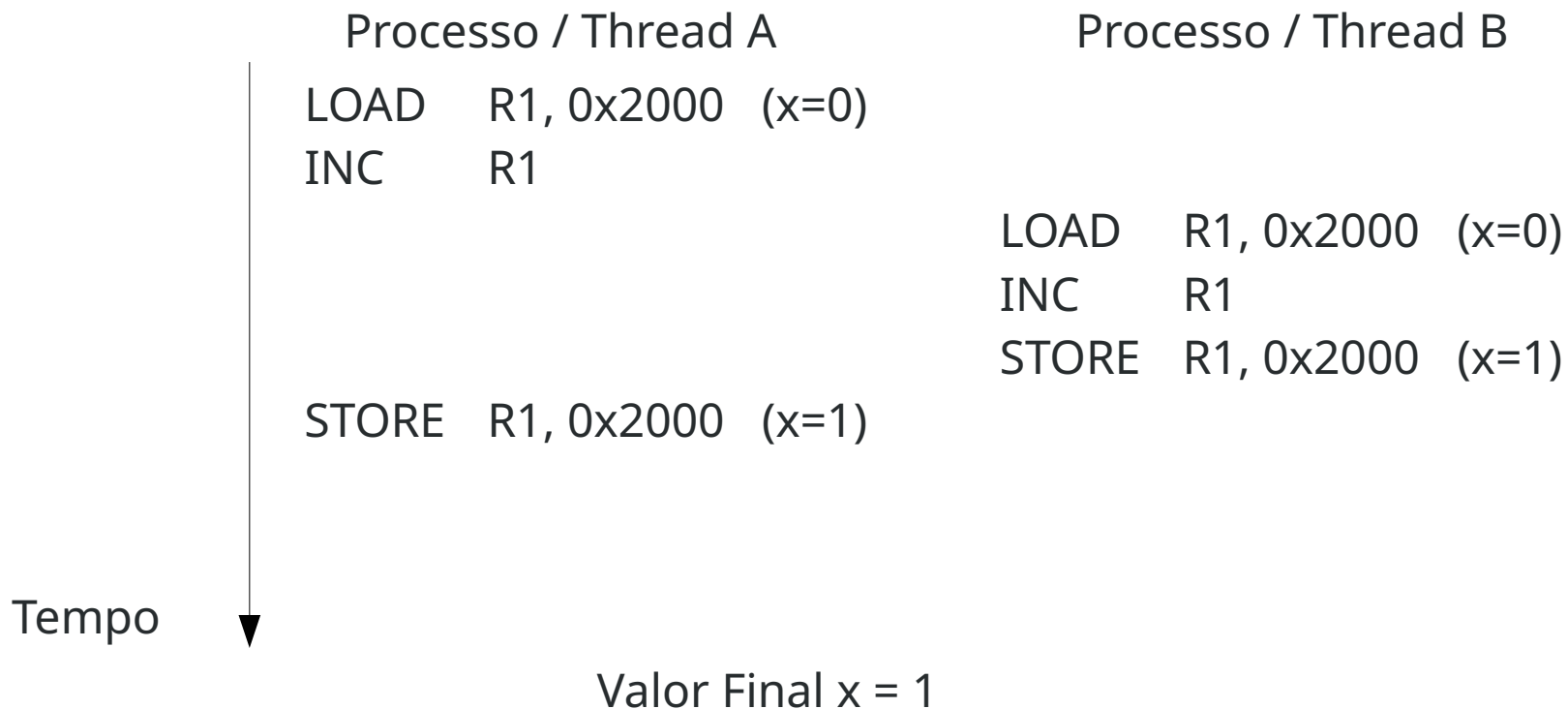


Valor Final x = 2



# CONDIÇÃO DE CORRIDA 1

- Escalonamento:  $A \rightarrow B \rightarrow A$





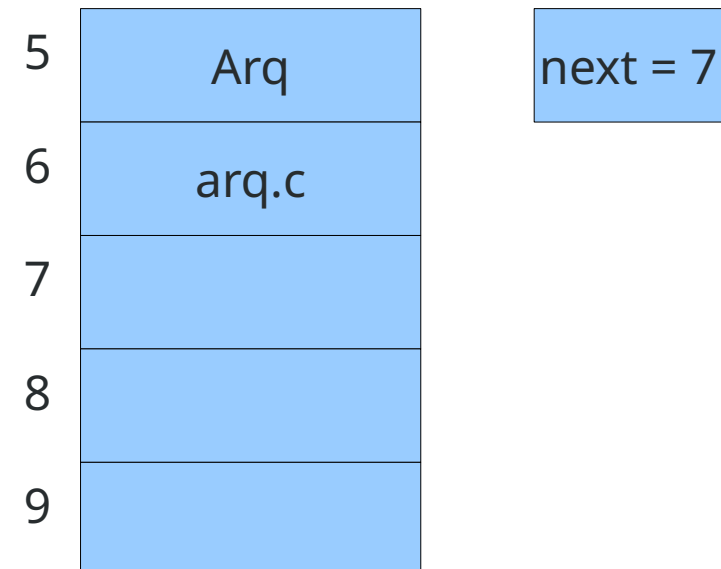
# CONDIÇÃO DE CORRIDA

- Esse comportamento tende a ser bem indesejável... Afinal na cabeça do programador “A variável X foi incrementada duas vezes”
- Porém, alguma hora ela apareceu com apenas um incremento
- Debug/Depuração dessas operações podem ser extremamente complexos



## CONDIÇÃO DE CORRIDA 2

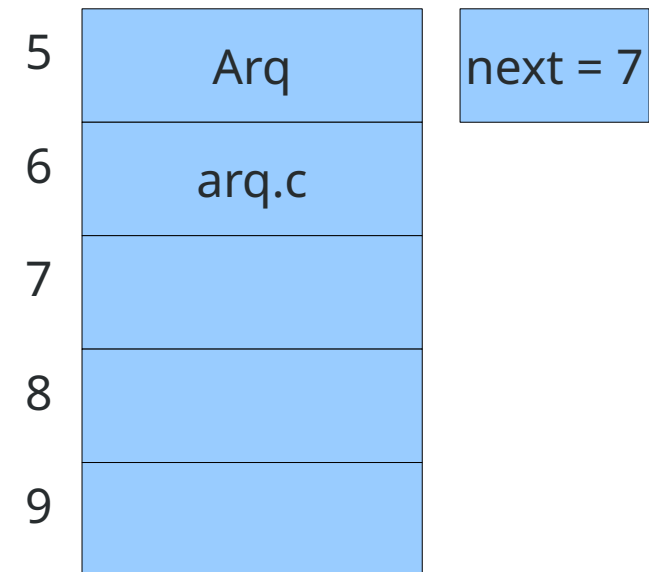
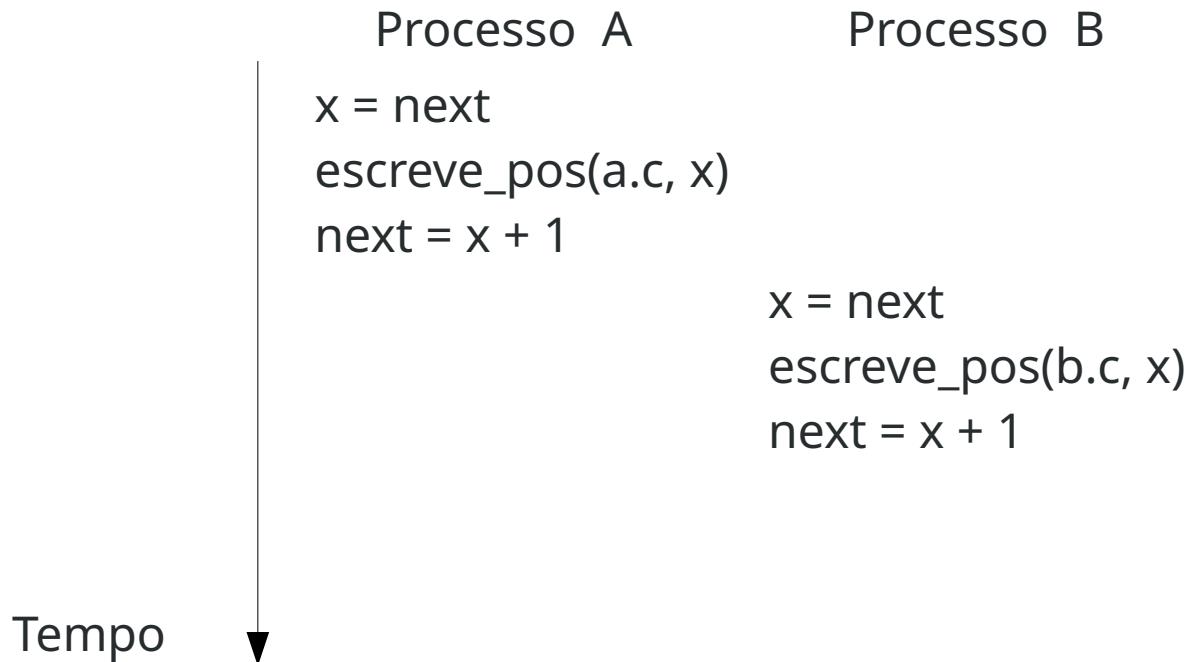
- Imagina que um servidor de impressão enumere as vagas dos arquivos impressos 0, 1, 2, ...
- Uma variável next aponta para a próxima posição livre a ser impressa
- Imagina agora, que o processo A e B desejaram imprimir um arquivo e o servidor está com a seguinte configuração





# CONDIÇÃO DE CORRIDA 2

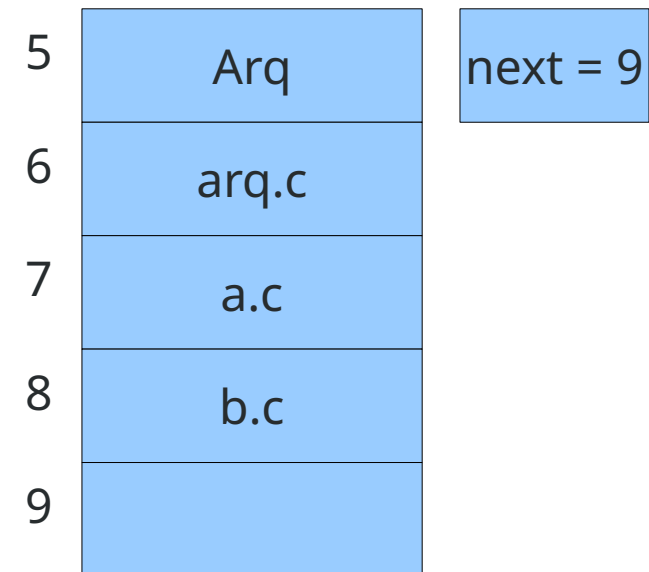
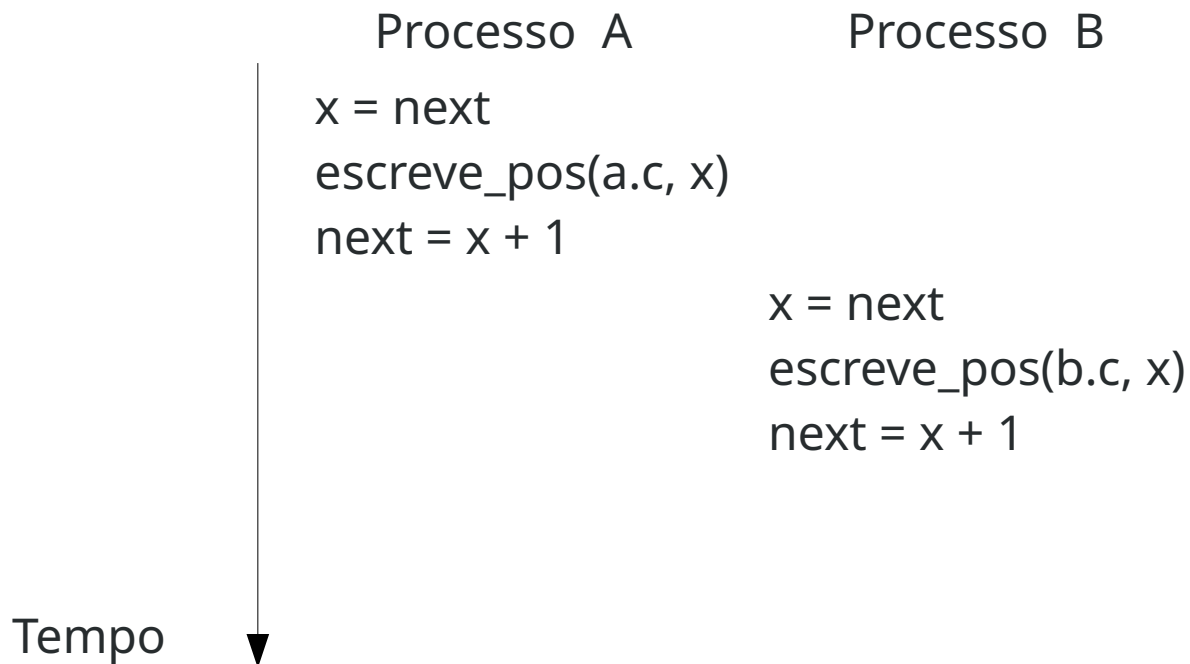
- Escalonamento:  $A \rightarrow B$
- $x$  é uma variável local





# CONDIÇÃO DE CORRIDA 2

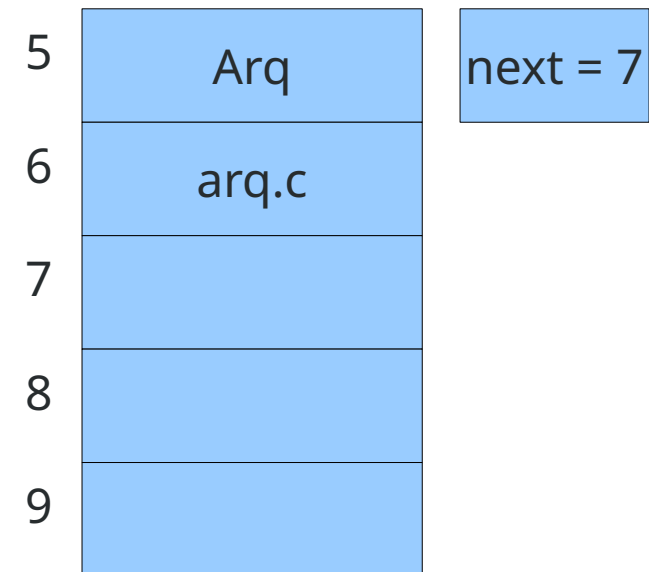
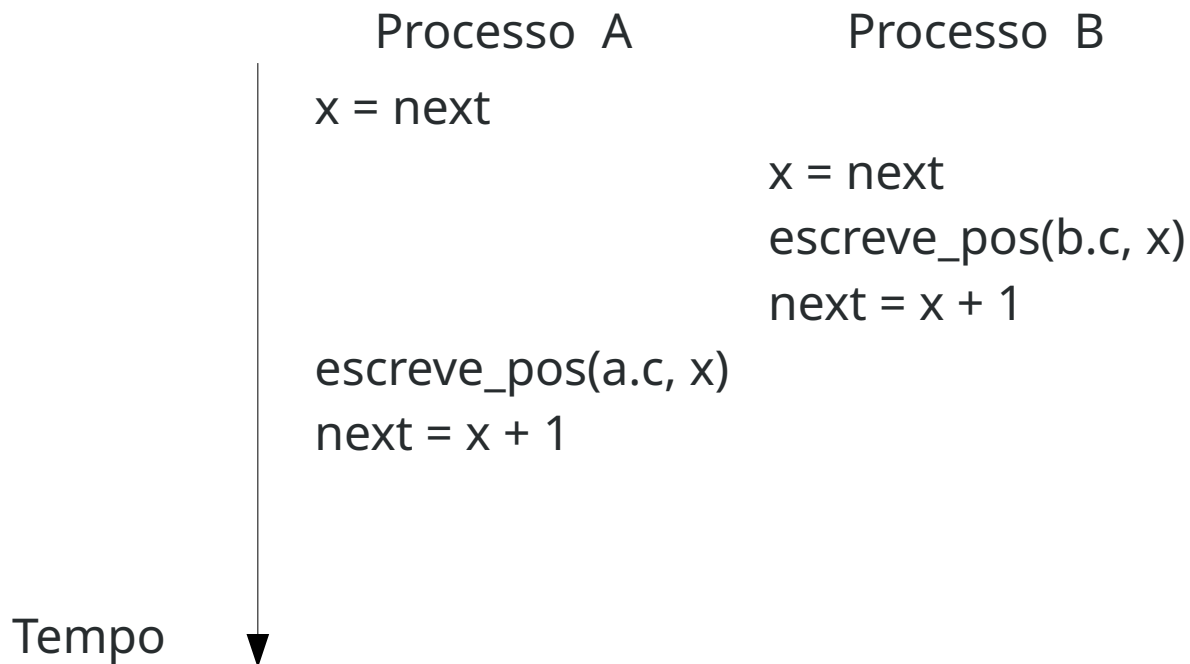
- Escalonamento:  $A \rightarrow B$
- $x$  é uma variável local





# CONDIÇÃO DE CORRIDA 2

- Escalonamento:  $A \rightarrow B \rightarrow A$
- $x$  é uma variável local

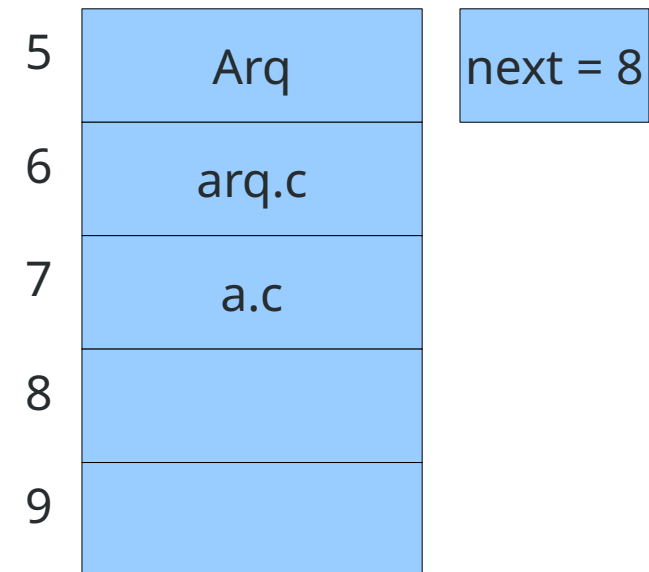
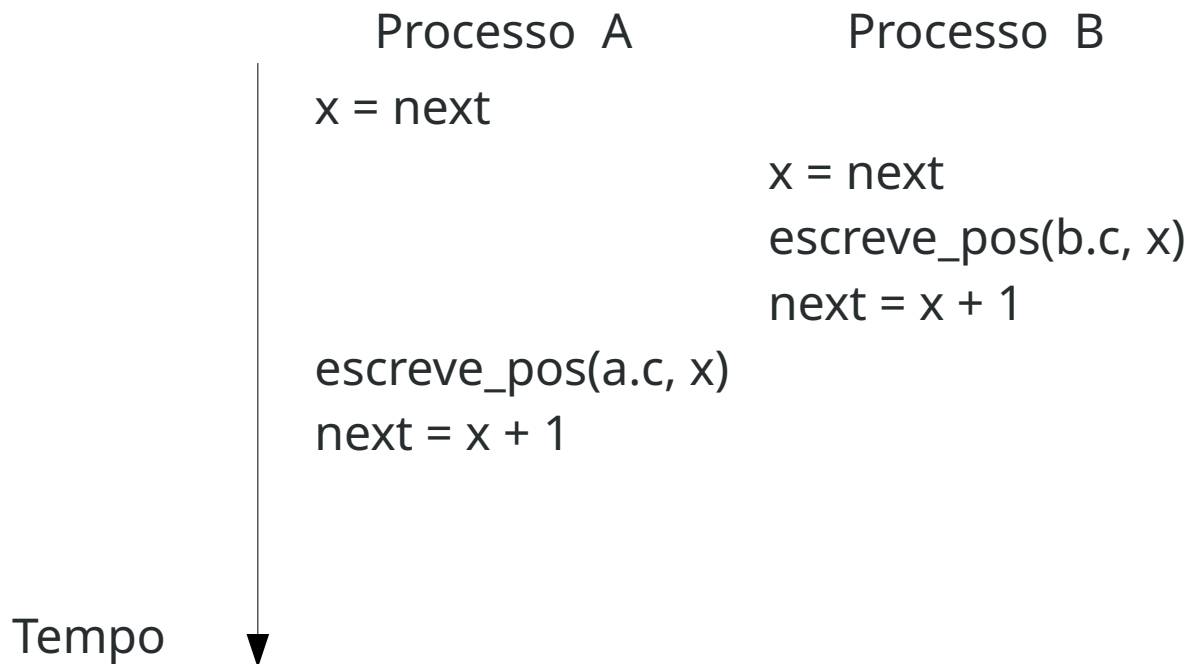






# CONDIÇÃO DE CORRIDA 2

- Escalonamento:  $A \rightarrow B \rightarrow A$
- $x$  é uma variável local





# CONDIÇÕES DE CORRIDA

- As condições de corrida levam a resultados inesperados e devem ser evitadas
- Precisa-se assegurar que os processos que estejam trabalhando na mesma região de memória não sejam interrompidos, ou aguardem o término do outro processo antes de iniciar suas atividades
- Esse processo é conhecido como **exclusão mútua** (mutual exclusion)



# REGIÕES CRÍTICAS

- Um processo precisa ter acesso à dados compartilhados para poder cooperar entre si
- O trecho de código que há acesso de leitura ou escrita à dados compartilhados é chamado de **seção crítica** (ou região crítica)
- No primeiro exemplo a seção crítica é a operação de incremento. No segundo exemplo é toda a operação de escrever. A seção crítica normalmente é MAIS de uma instrução



# REGIÕES CRÍTICAS

- Para evitar as condições de corrida, são colocadas funções antes de entrar e depois de sair da seção crítica
- Essas funções utilizam diversas técnicas para impedir que dois processos estejam simultaneamente na seção crítica e garantir a exclusão mútua



# VARIÁVEIS DE IMPEDIMENTO

- Variável de impedimento busca marcar se existe alguém na seção crítica. Se for 1, não procede
- Analise o trecho de código:

```
int thread()  
{  
    while (true)  
    {  
        while (lock == 1) {}  
        lock = 1;  
        regioao_critica();  
        lock = 0;  
    }  
}
```


```
int thread()  
{  
    while (true)  
    {  
        while (lock == 1) {}  
        lock = 1;  
        regioao_critica();  
        lock = 0;  
    }  
}
```



# VARIÁVEIS DE IMPEDIMENTO

- Se ocorrer uma troca de contexto depois de sair do loop e antes do processo/thread trocar o valor para 1, há uma condição de corrida

```
int thread()
{
    while (true)
    {
        while (lock == 1) {}
        lock = 1;
        regioao_critica();
        lock = 0;
    }
}
```



```
int thread()
{
    while (true)
    {
        while (lock == 1) {}
        lock = 1;
        regioao_critica();
        lock = 0;
    }
}
```



# VARIÁVEIS DE IMPEDIMENTO

- Se ocorrer uma troca de contexto antes do processo/thread trocar o valor para 1, há uma condição de corrida
- A solução para exclusão mútua não é trivial!
- Entendeu?





# TÉCNICAS DE IMPLEMENTAÇÃO DE EXCLUSÃO MÚTUA

- Inibir Interrupções
- Com espera ocupada:
  - Estrita Alternância
  - Algoritmo de Peterson
  - Utilizar hardware adicional
- Com bloqueio de processos:
  - Semáforos
  - Mutexes
  - Locks
  - Monitores
  - Variáveis de condição





# Referências

- Capítulo 2 – TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 4ª ed. Prentice Hall, 2016.
- Capítulo 6 – MACHADO, F. B; MAIA, L. P. *Arquitetura de Sistemas Operacionais*. 5ª ed. Rio de Janeiro: LTC, 2013.