

INFRA-ESTRUTURA DE SOFTWARE

Gerência de Processos



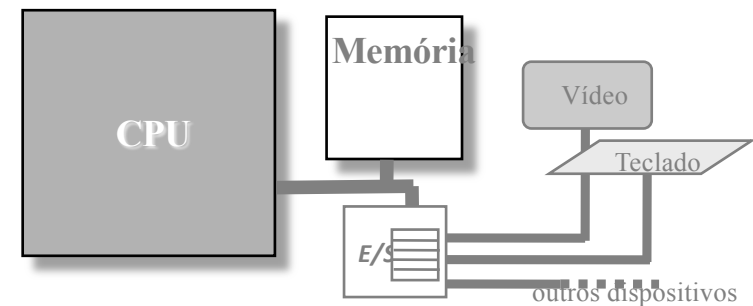
PROCESSO

Um programa em execução

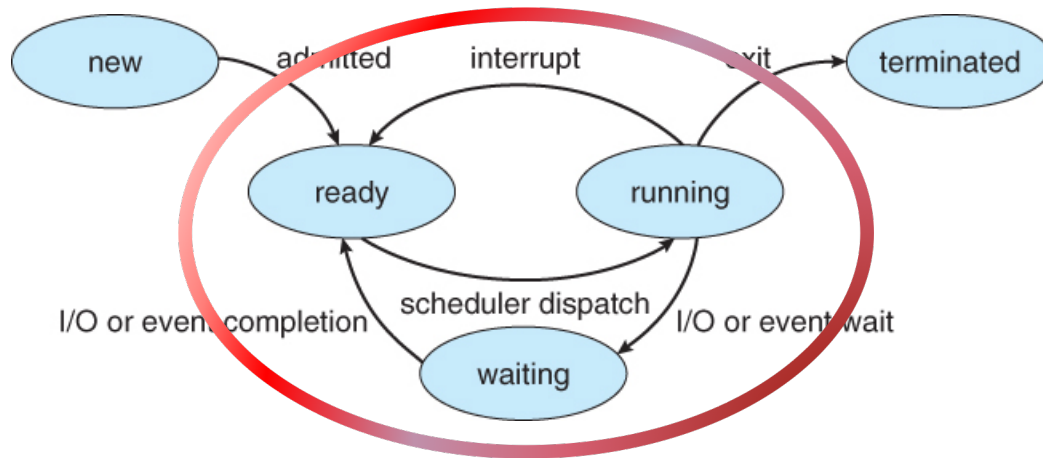


Contexto de Processo

- Conjunto de informações para gerenciamento de processo
 - CPU: Conteúdo dos registradores
 - Memória: Posições em uso
 - E/S: Estado das requisições
 - Estado do processo: Rodando, Bloqueado, Pronto
 - Outras



Estados de um Processo



Principais
(waiting = blocked)

Contexto

<i>ID do Processo</i>
<i>Estado</i>
<i>Prioridade</i>
<i>Program Counter</i>
<i>Ponteiros da Memória</i>
<i>Contexto (outros regs.)</i>
<i>I/O Status</i>
<i>Informações gerais</i> <ul style="list-style-type: none">• <i>tempo de CPU</i>• <i>limites, usuário, etc.</i>



Process Control Block

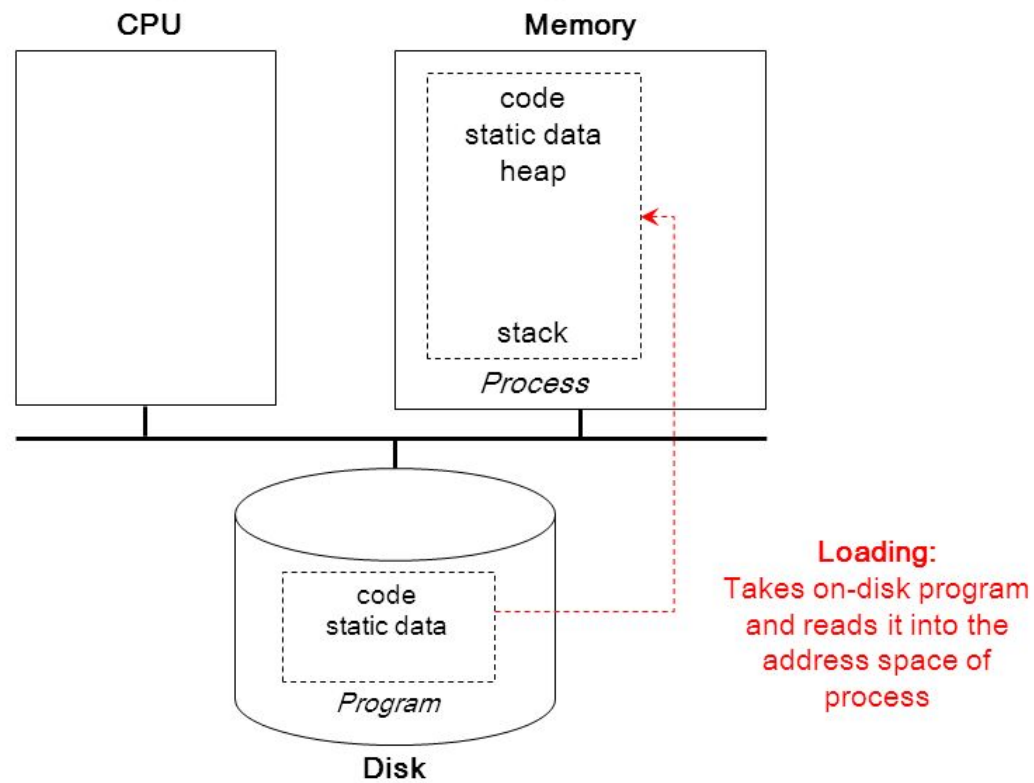
- **Estrutura de dados** que contém informações suficientes para que seja possível interromper um processo em execução e depois retomar sua execução de onde parou

PCB

<i>ID do Processo</i>
<i>Estado</i>
<i>Prioridade</i>
<i>Program Counter</i>
<i>Ponteiros da Memória</i>
<i>Contexto (outros regs.)</i>
<i>I/O Status</i>
<i>Informações gerais</i> <ul style="list-style-type: none">• <i>tempo de CPU</i>• <i>limites, usuário, etc.</i>



Loading: From Program To Process



Nos SOs modernos este processo é “preguiçoso” (*lazy*),



Ciclo de vida de um processo...

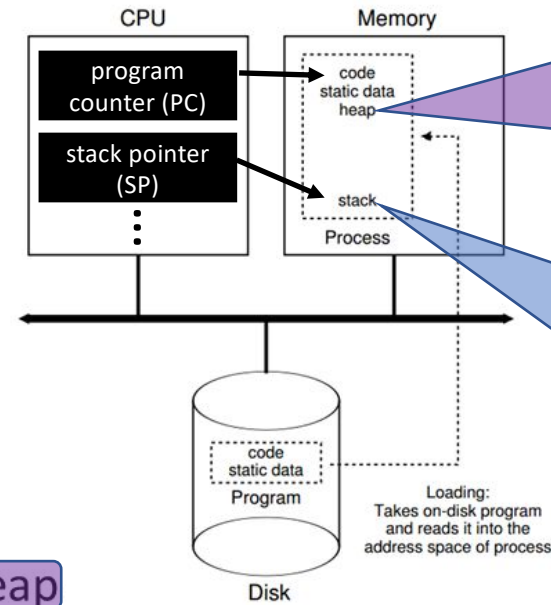
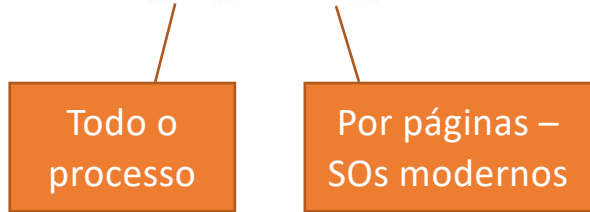
e o que acontece em termos de memória, E/S, sistema de arquivos etc.



READY

Process Creation

- Loading: code and static data
 - Eagerly vs lazily



Armazena estruturas de dados dinâmicas, como listas encadeadas, tabelas hash, árvores etc.

Armazena parâmetros de funções (do programa), variáveis locais e endereços de retorno

- Allocate memory for **stack**, **heap**
- Initialize file descriptors
- Jump to the main and give the CPU to the process

RUNNING





Criação de Processos

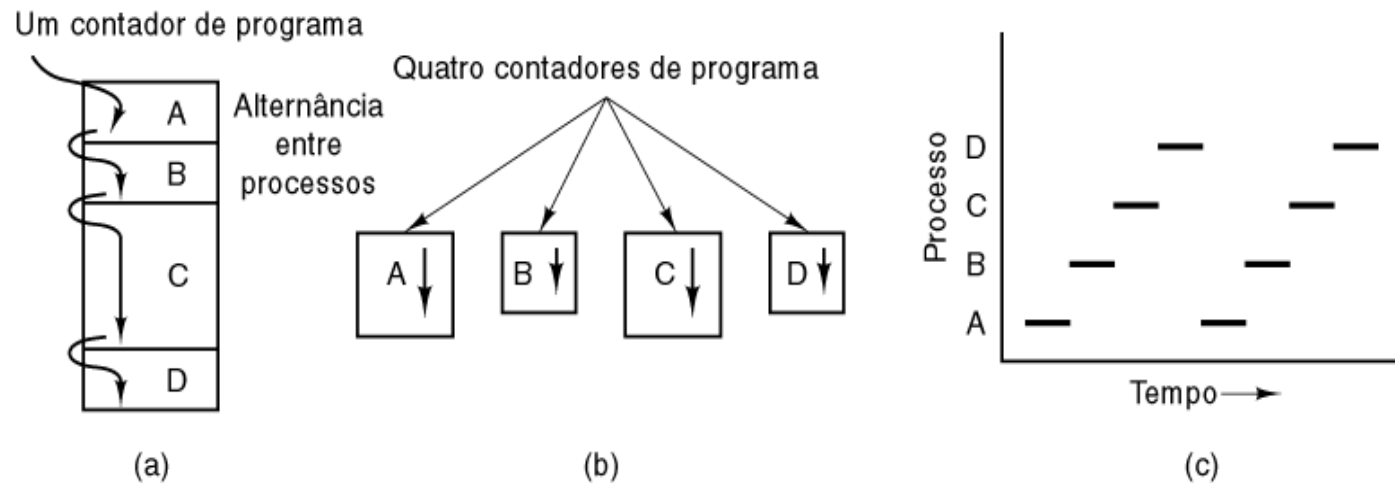
- Principais eventos que levam à criação de processos
 - Início do sistema
 - Execução de chamada ao sistema de criação de processos (ex. fork)
 - Solicitação do usuário para criar um novo processo
 - Início de um job em lote



Término de Processos

- Condições que levam ao término de processos
 - Saída normal (voluntária)
 - Saída por erro (voluntária) - programado
 - Erro fatal (involuntário) – SO/usuário termina o processo
 - Cancelamento por um outro processo (involuntário)

Conceito: Multiprogramação



- a) **Multiprogramação** de quatro programas
- b) Modelo conceitual de 4 processos sequenciais, independentes, mas
- c) Somente um processo está ativo a cada momento (compartilhamento de tempo/*time sharing*) \Rightarrow **escalonamento**





Rastreando estados de processos

Tempo	Processo ₀	Processo ₁	Obs.
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Processo ₀ inicia E/S
4	Blocked*	Running	Processo ₀ é bloqueado, então
5	Blocked	Running	Processo ₁ executa
6	Blocked	Running	
7	Ready	Running	E/S termina; Processo ₀ pronto para voltar
8	Ready	Running	Processo ₁ termina
9	Running	-	Processo ₀ volta a executar
10	Running	-	Processo ₀ termina

*Blocked = Waiting



Escalonamento de processos

- Quando dois ou mais processos estão prontos para serem executados, o sistema operacional deve decidir qual deles vai ser executado primeiro
- A parte do sistema operacional responsável por essa decisão é chamada **escalador...**
- e o algoritmo usado para tal é chamado de **algoritmo de escalonamento**
- Para que um processo não execute tempo demais, praticamente todos os computadores possuem um mecanismo de relógio (*clock*) que causa uma **interrupção**, periodicamente

E de *threads*?



Datas importantes

Exercícios Escolares

Data	Dia e Hora	Atividade	Assunto
10/04/19	QUA, 10-12h	1o. EE	Exercício de concorrência; LabG2
10/05/19	SEX, 08-10h	2o. EE	Gerência de Processos e Gerência de Memória
12/06/19	QUA, 10-12h	3o. EE	Sistema de Arquivos, Gerência E/S; Sistemas Distribuídos
26/06/19	QUA, 10-12h	PROVA FINAL	2a. Chamada também

E mais... valendo pontos extra

- 03/04: Exercício de Escalonamento de Processos
- 08/05: Exercício de Substituição de Página
- 07/06: Exercício de RPC/RMI



INTERRUPÇÃO

O elo Hardware-Software



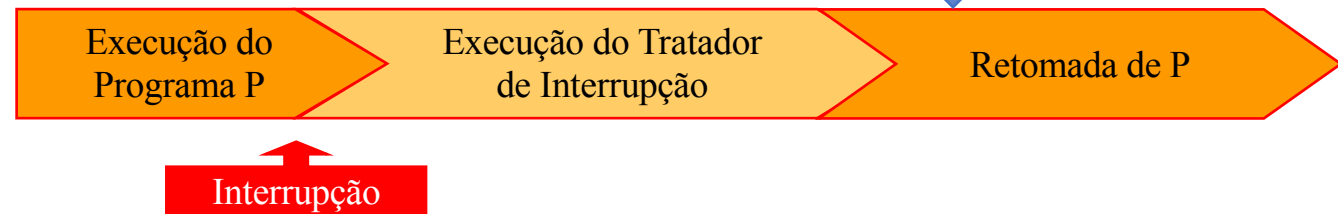
Motivação

- Para controlar entrada e saída de dados, **não é interessante** que a CPU tenha que ficar continuamente monitorando o status de dispositivos, como discos ou teclados
- O mecanismo de interrupções permite que o hardware "chame a atenção" da CPU quando há algo a ser feito



Interrupções de Hardware

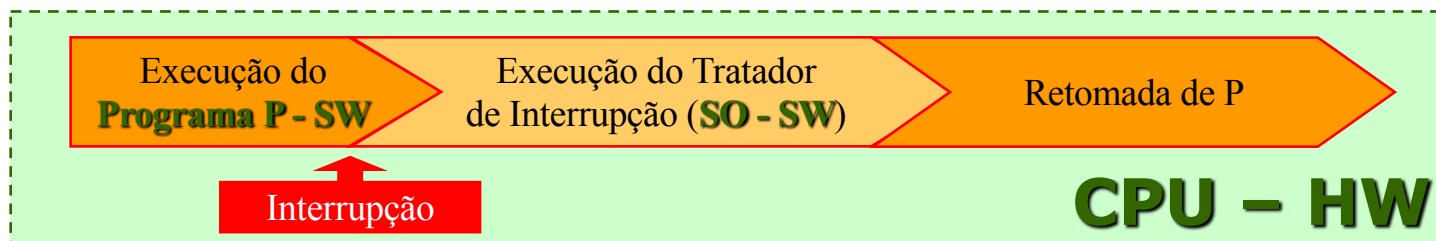
- Interrupções geradas por **algum dispositivo externo à CPU**, como teclado ou controlador de disco, são chamadas de interrupções de hardware ou **assíncronas** [ocorrem independentemente das instruções que a CPU está executando]
- Quando ocorre uma interrupção, a CPU interrompe o processamento do programa em execução e executa um pedaço de código (tipicamente parte do sistema operacional) chamado de **tratador de interrupção**
 - não há qualquer comunicação entre o programa interrompido e o tratador (parâmetros ou retorno)
 - em muitos casos, após a execução do tratador, a CPU volta a executar o programa interrompido





Interrupção: Suporte de HW

- Tipicamente, o hardware detecta que ocorreu uma interrupção,
 - aguarda o final da execução da instrução corrente e aciona o tratador,
 - antes salvando o contexto de execução do processo interrompido
- Para que a execução do processo possa ser reiniciada mais tarde, é necessário salvar o *program counter* (PC) e outros registradores de status
 - Os registradores com dados do programa devem ser salvos **pelo próprio tratador** (ou seja, **por software**), que em geral os utiliza
 - Para isso, existe uma pilha independente associada ao tratamento de interrupções





Interrupção de Relógio

(Um tipo especial de Interrupção de HW)

- O sistema operacional atribui *quotas de tempos de execução* (**quantum** ou **time slice** – fatias de tempo) para cada um dos processos em um sistema com *multiprogramação*
- A cada interrupção do relógio, o tratador verifica se a fatia de tempo do processo em execução já se esgotou e, se for esse o caso, suspende-o e aciona o *escalonador* para que esse escolha outro processo para colocar em execução



Interrupções Síncronas ou *Traps*

- *Traps* ocorrem em **consequência da instrução sendo executada** [no programa em execução]
- Algumas são geradas pelo hardware, para indicar, por exemplo, *overflow* em operações aritméticas ou acesso a regiões de memória não permitidas
 - **Essas são situações em que o programa não teria como prosseguir**
 - O hardware sinaliza uma interrupção para passar o controle para o tratador da interrupção (no SO), que tipicamente termina a execução do programa



Traps (cont.)

- Traps também podem ser geradas, explicitamente, por instruções do programa
 - Essa é uma forma do programa acionar o sistema operacional, por exemplo, para requisitar um serviço de entrada ou saída
 - Ex. **Read**
 - Um programa não pode chamar diretamente uma rotina do sistema operacional, já que o SO é um processo a parte, com seu próprio espaço de endereçamento...
 - Através do mecanismo de **interrupção de software**, um processo qualquer pode ativar um tratador que pode "encaminhar" uma chamada ao sistema operacional
- Como as interrupções síncronas ocorrem em função da instrução que está sendo executada (ex. READ – uma chamada ao sistema), nesse caso o programa passa algum parâmetro para o tratador



Interrupções: Resumo

Assíncronas (hardware)

- geradas por **algum dispositivo externo à CPU**
- **ocorrem independentemente das instruções que a CPU está executando**
- não há qualquer comunicação entre o programa interrompido e o tratador
- Exemplos:
 - interrupção de relógio, quando um processo esgotou a sua fatia de tempo (*time slice*) no uso compartilhado do processador
 - teclado, para uma operação de Entrada

Síncronas (*traps*)

- geradas pelo **programa em execução**, em consequência da instrução sendo executada
- algumas são geradas pelo hardware **em situações em que o programa não teria como prosseguir**
- Como as interrupções síncronas ocorrem em função da instrução que está sendo executada, nesse caso o programa **passa algum parâmetro para o tratador**
- Exs.: READ, *overflow* em operações aritméticas ou acesso a regiões de memória não permitidas



Conceitos

- **Processo**: um programa em execução
- **Contexto de processo**: informações para gerenciamento
- **Espaço de endereçamento e proteção**
- **Escalonamento**: quando dois ou mais processos estão prontos para serem executados, o sistema operacional deve decidir qual deles vai ser executado
- **Interrupção**
 - Por hardware
 - Algum dispositivo externo à CPU (ex. teclado)
 - Relógio (para suspender um processo)
 - Por software (trap)
 - Execução de instrução de programa (ex. READ)
 - situações em que o programa não teria como prosseguir (ex. overflow em operações aritméticas)
- **Chamadas ao Sistema (*System Calls*)** formam a interface entre o SO e os programas de usuário

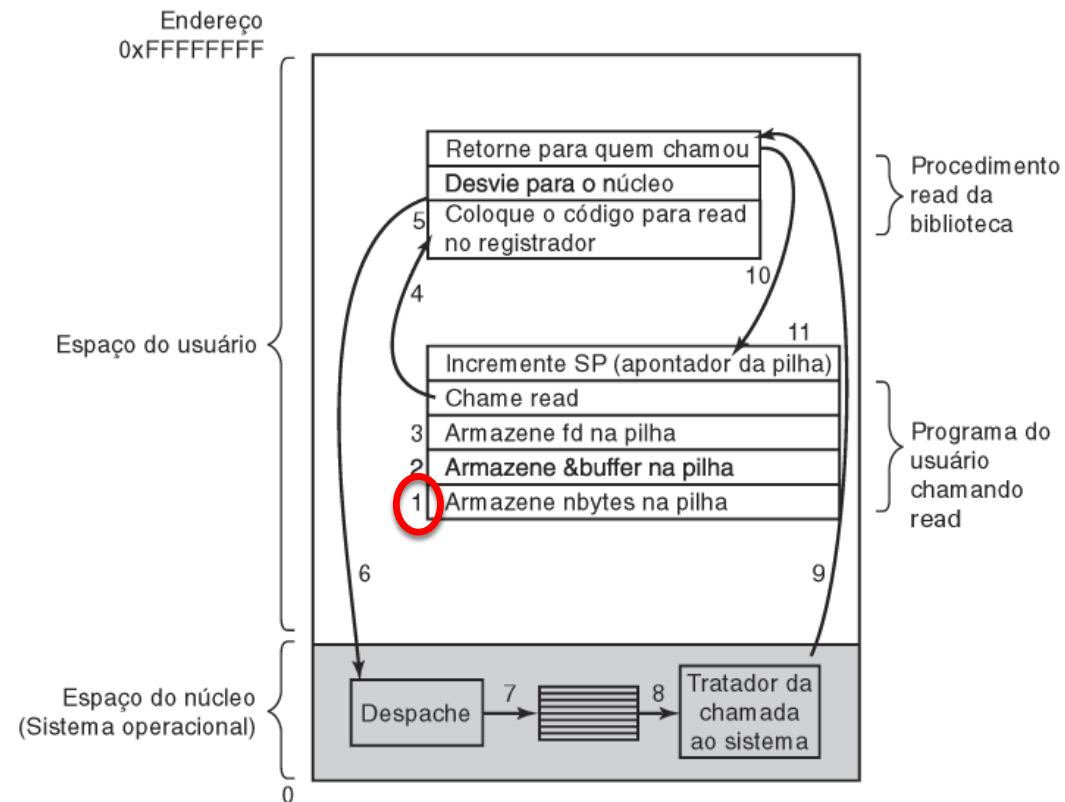


CHAMADAS AO SISTEMA

System Calls



Os Passos de uma Chamada ao Sistema



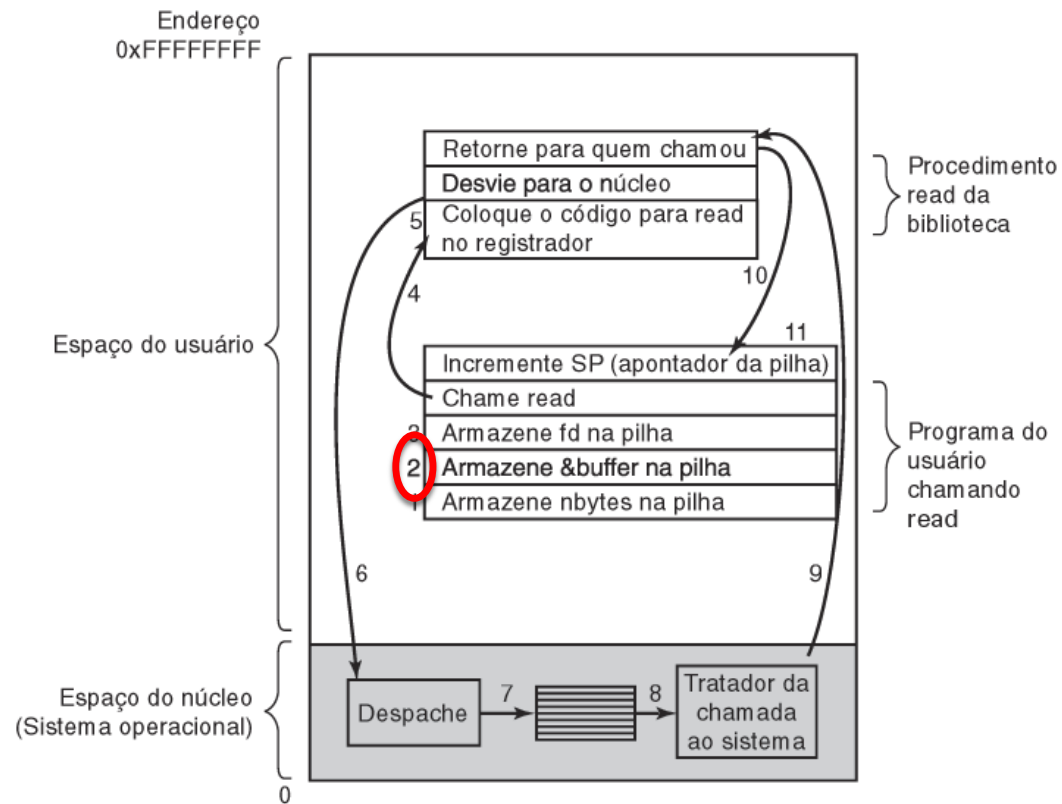
Os 11 passos para fazer uma chamada ao Sistema

Ex. **read (fd, buffer, nbytes)**



Os Passos de uma Chamada ao Sistema

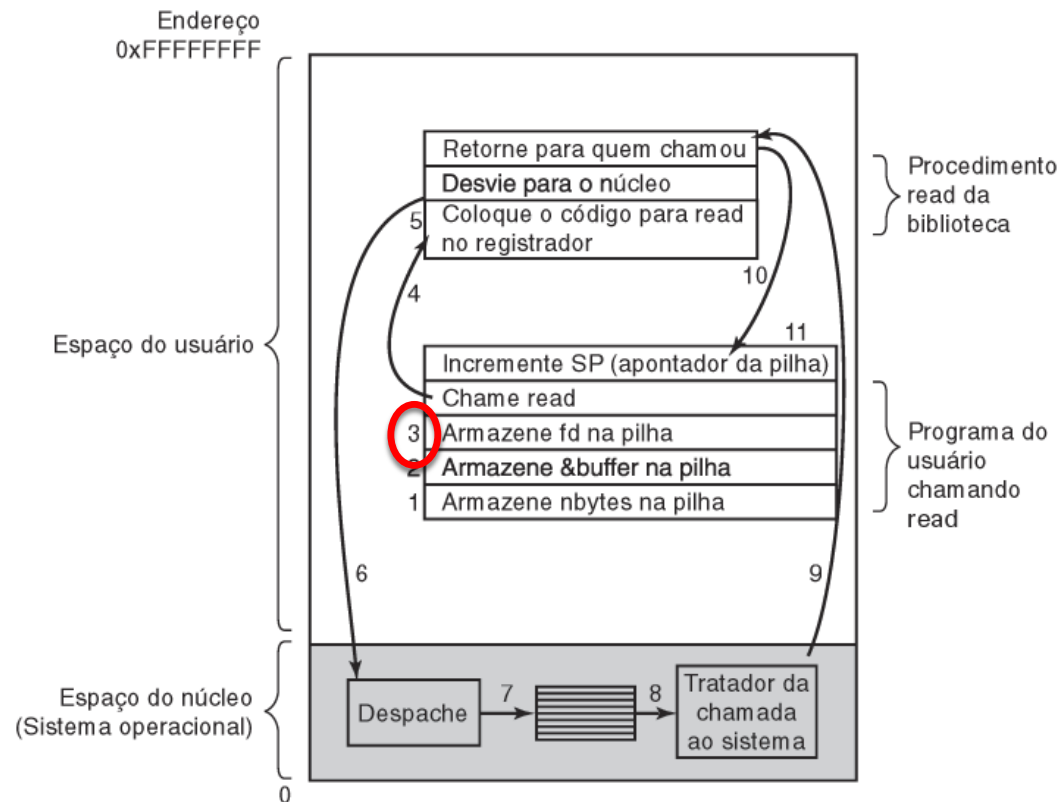
Os 11 passos para fazer
uma chamada ao sistema
Ex. read (fd, buffer, nbytes)





Os Passos de uma Chamada ao Sistema

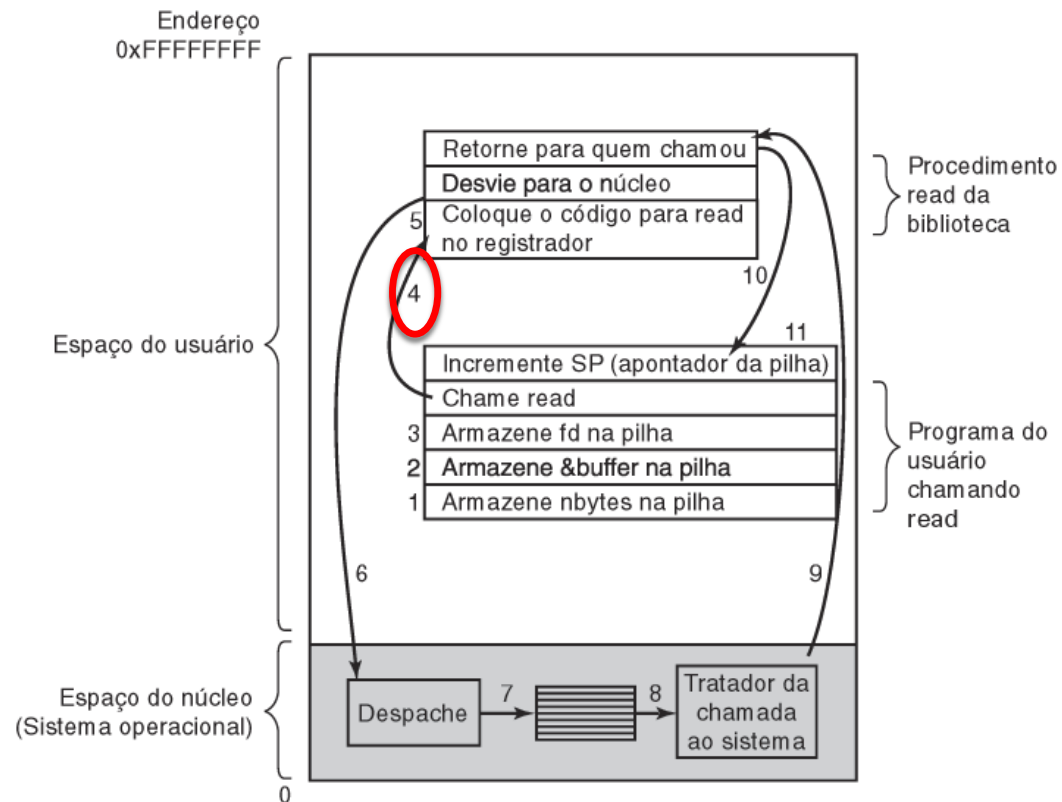
Os 11 passos para fazer
uma chamada ao sistema
Ex. read (fd, buffer, nbytes)





Os Passos de uma Chamada ao Sistema

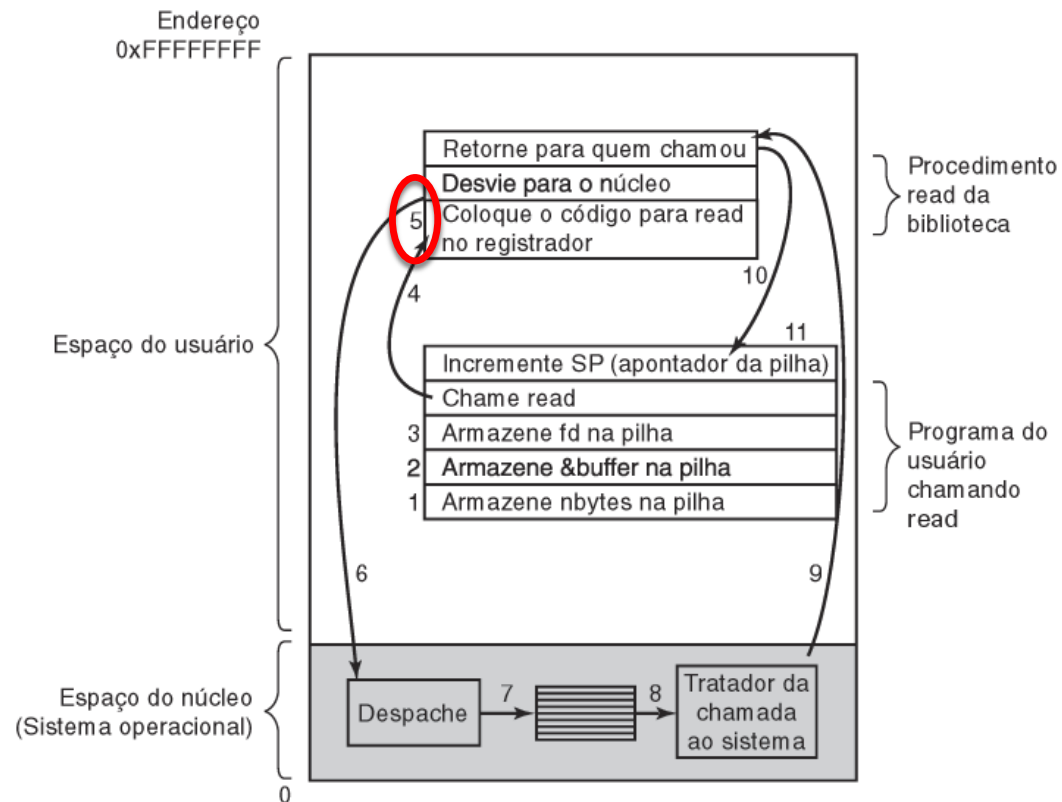
Os 11 passos para fazer
uma chamada ao sistema
Ex. read (fd, buffer, nbytes)





Os Passos de uma Chamada ao Sistema

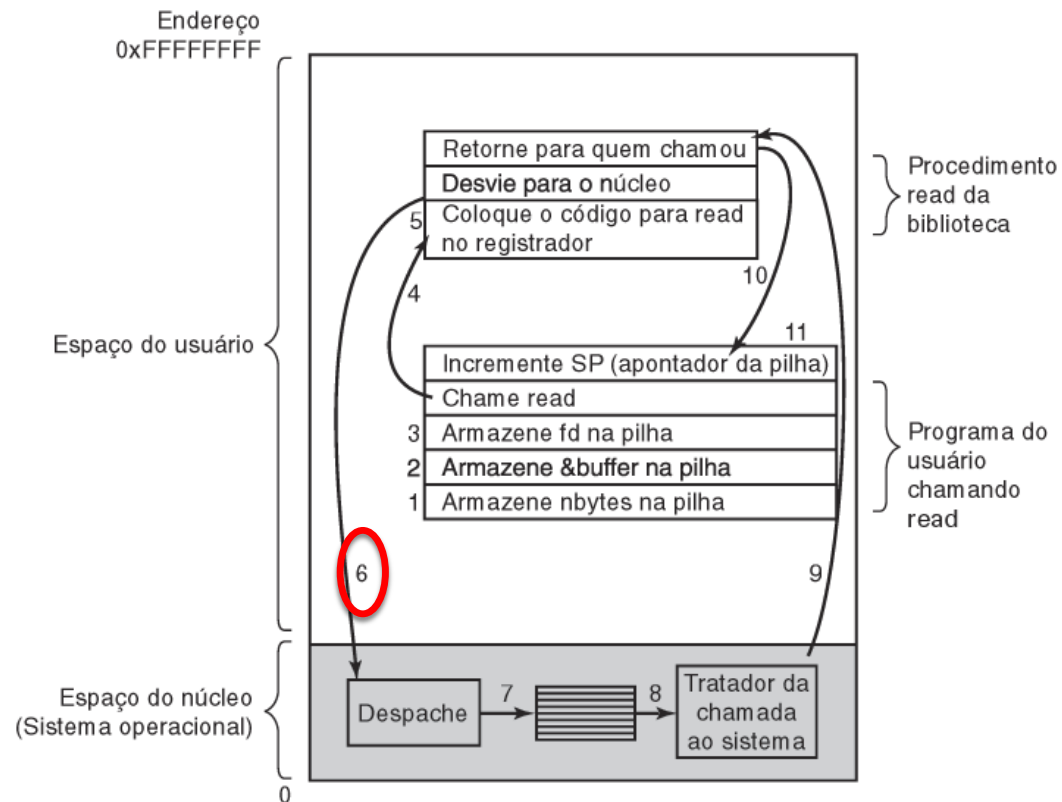
Os 11 passos para fazer
uma chamada ao sistema
Ex. read (fd, buffer, nbytes)





Os Passos de uma Chamada ao Sistema

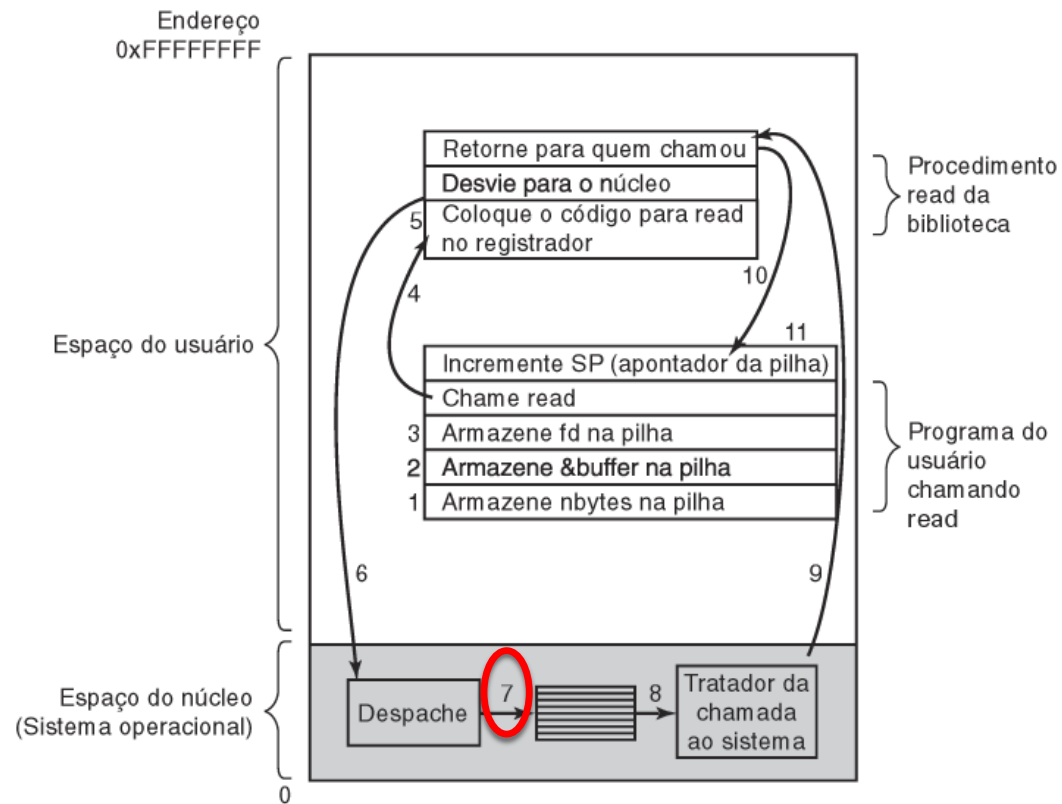
Os 11 passos para fazer
uma chamada ao sistema
Ex. read (fd, buffer, nbytes)





Os Passos de uma Chamada ao Sistema

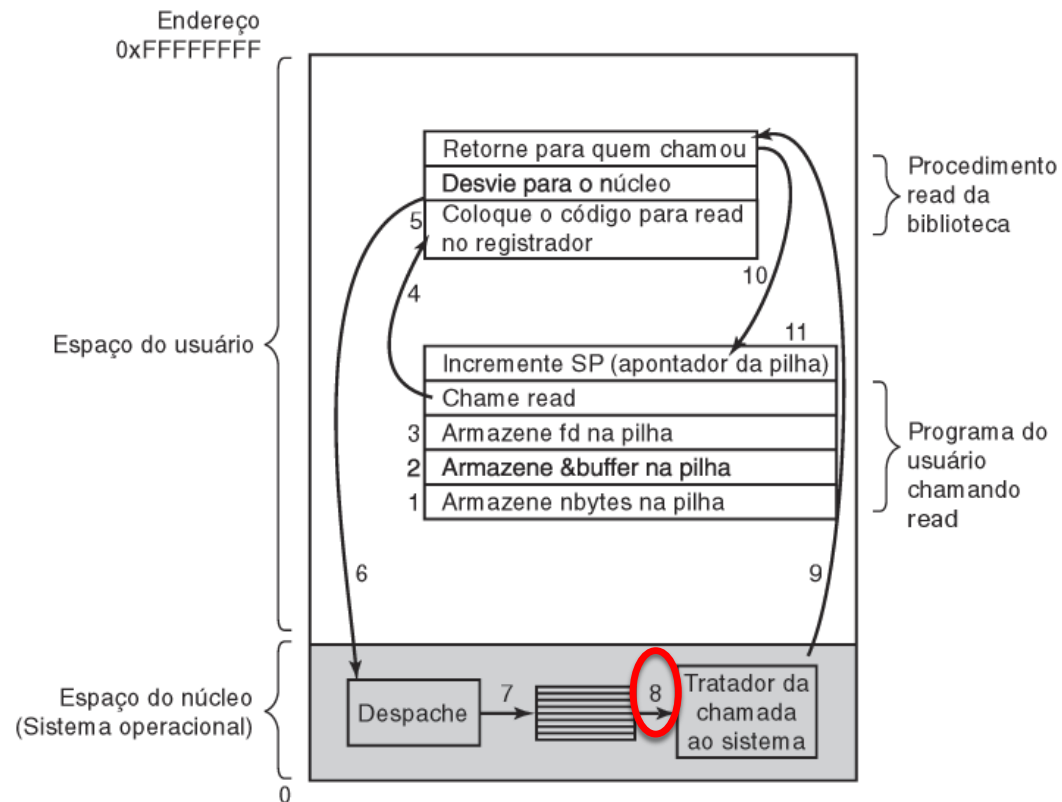
Os 11 passos para fazer
uma chamada ao sistema
Ex. read (fd, buffer, nbytes)





Os Passos de uma Chamada ao Sistema

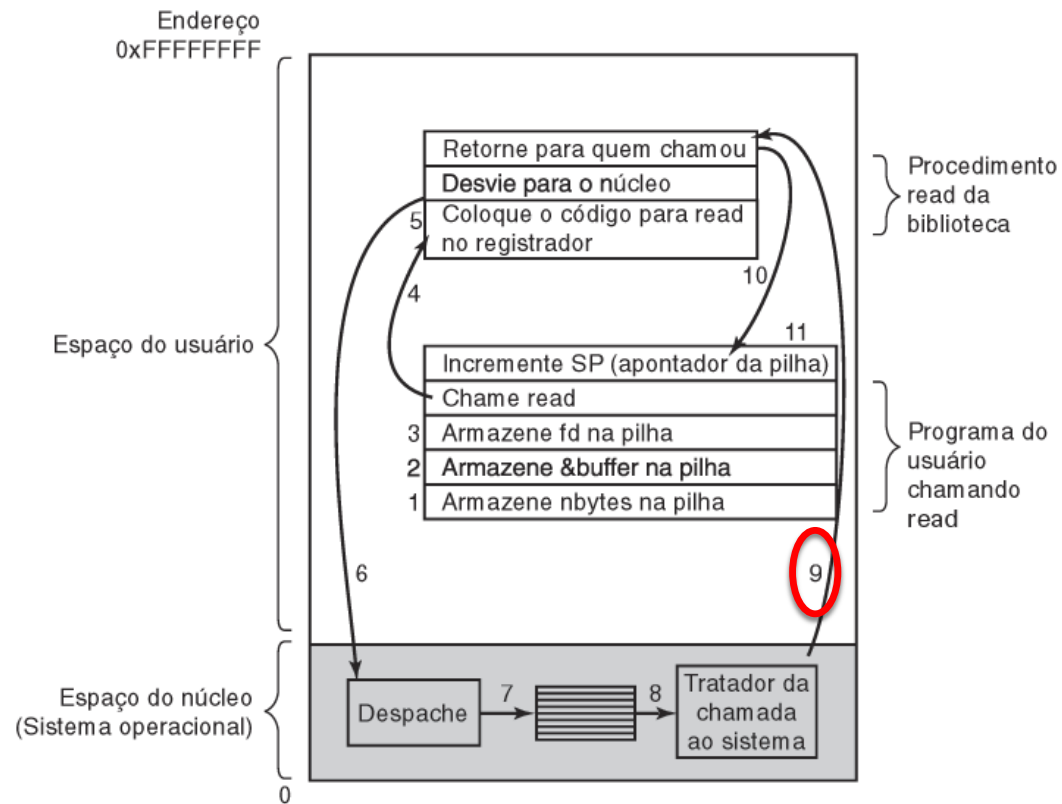
Os 11 passos para fazer
uma chamada ao sistema
Ex. read (fd, buffer, nbytes)





Os Passos de uma Chamada ao Sistema

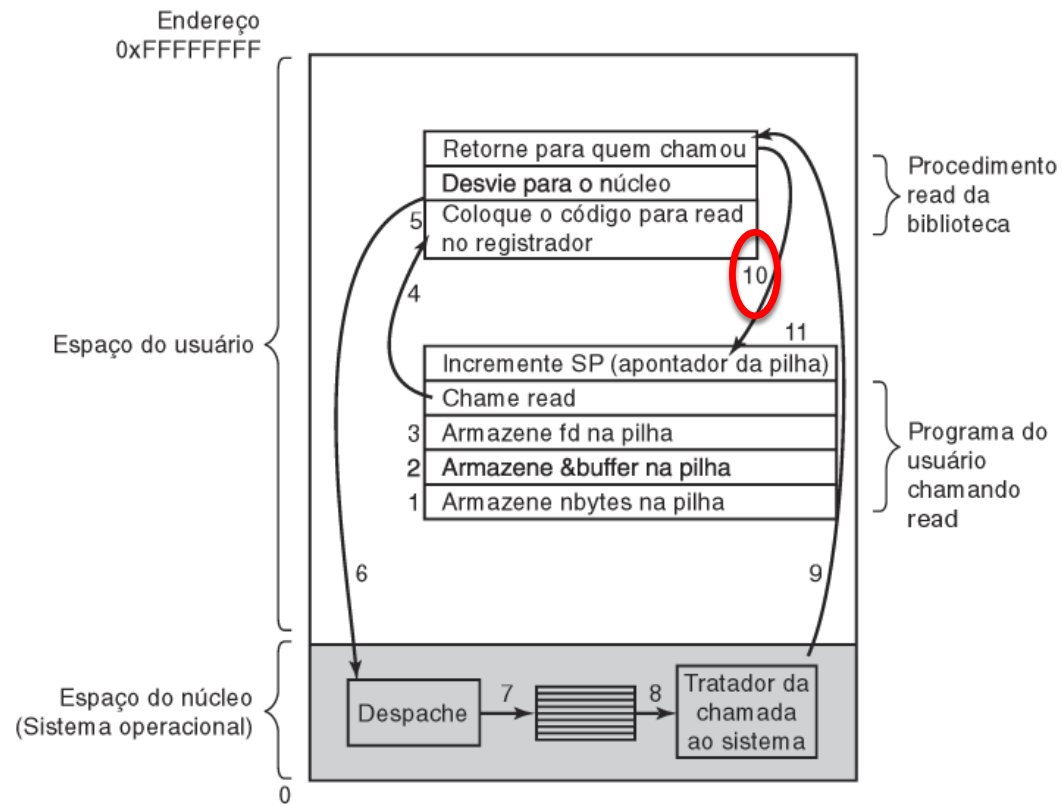
Os 11 passos para fazer
uma chamada ao sistema
Ex. read (fd, buffer, nbytes)





Os Passos de uma Chamada ao Sistema

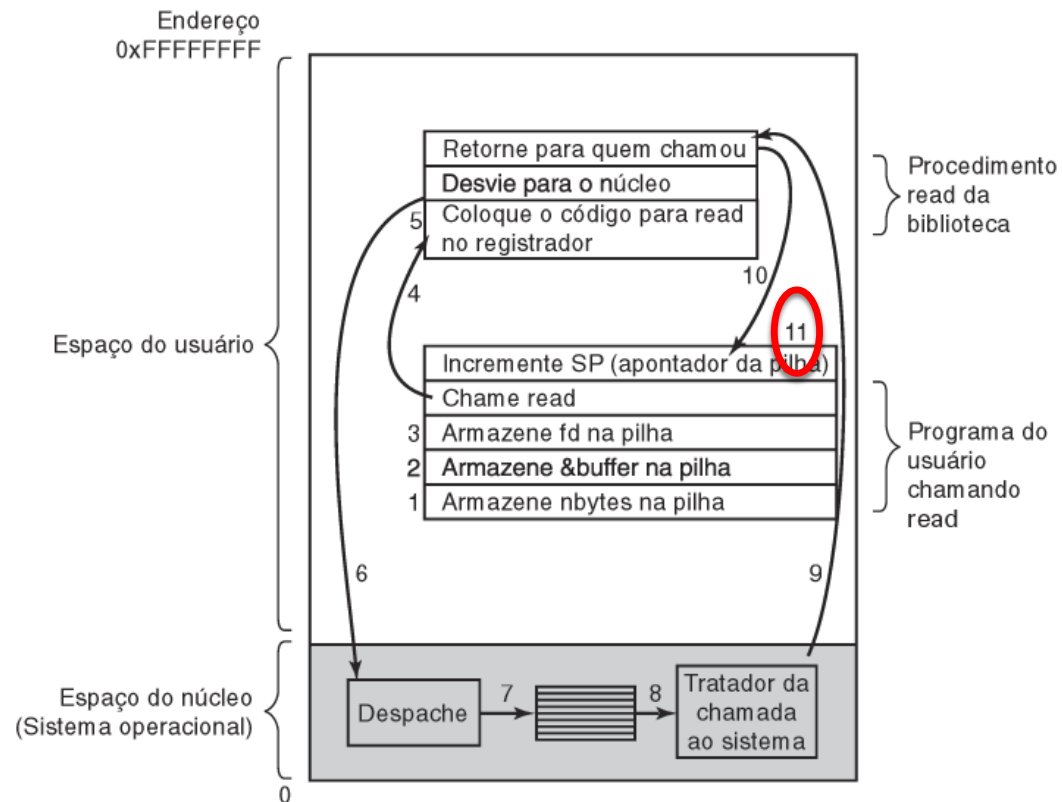
Os 11 passos para fazer
uma chamada ao sistema
Ex. read (fd, buffer, nbytes)





Os Passos de uma Chamada ao Sistema

Os 11 passos para fazer
uma chamada ao sistema
Ex. read (fd, buffer, nbytes)



Algumas Chamadas ao Sistema para Gerenciamento de Processos

Gerenciamento de processos

Chamada	Descrição
<code>pid = fork()</code>	Crie um processo filho idêntico ao processo pai
<code>pid = waitpid(pid, &statloc, options)</code>	Aguarde um processo filho terminar
<code>s = execve(name, argv, environp)</code>	Substitua o espaço de endereçamento do processo
<code>exit(status)</code>	Termine a execução do processo e retorne o estado



Algumas Chamadas ao Sistema para Gerenciamento de Processos

Gerenciamento de arquivos

Chamada	Descrição
<code>fd = open(file, how, ...)</code>	Abra um arquivo para leitura, escrita ou ambas
<code>s = close(fd)</code>	Feche um arquivo aberto
<code>n = read(fd, buffer, nbytes)</code>	Leia dados de um arquivo para um buffer
<code>n = write(fd, buffer, nbytes)</code>	Escreva dados de um buffer para um arquivo
<code>position = lseek(fd, offset, whence)</code>	Mova o ponteiro de posição do arquivo
<code>s = stat(name, &buf)</code>	Obtenha a informação de estado do arquivo



Algumas Chamadas ao Sistema para Gerenciamento de Processos

Gerenciamento do sistema de diretório e arquivo

Chamada	Descrição
s = mkdir(name, mode)	Crie um novo diretório
s = rmdir(name)	Remova um diretório vazio
s = link(name1, name2)	Crie uma nova entrada, name2, apontando para name1
s = unlink(name)	Remova uma entrada de diretório
s = mount(special, name, flag)	Monte um sistema de arquivo
s = umount(special)	Desmonte um sistema de arquivo



Algumas Chamadas ao Sistema para Gerenciamento de Processos

Diversas

Chamada	Descrição
s = chdir(dirname)	Altere o diretório de trabalho
s = chmod(name, mode)	Altere os bits de proteção do arquivo
s = kill(pid, signal)	Envie um sinal a um processo
seconds = time(&seconds)	Obtenha o tempo decorrido desde 1º de janeiro de 1970



Chamadas ao Sistema

- O interior de uma *shell*:

```
#define TRUE 1

while (TRUE) {                               /* repita para sempre */
    type_prompt( );                          /* mostra prompt na tela */
    read_command(command, parameters);      /* lê entrada do terminal */

    if (fork( ) !=0) {                       /* cria processo filho */
        /* Parent code. */
        waitpid(-1, *status, 0);           /* aguarda o processo filho acabar */
    } else {
        /* Child code. */
        execve(command, parameters, 0);    /* executa o comando */
    }
}
```



Unix	Win32	Descrição
fork	CreateProcess	Crie um novo processo
waitpid	WaitForSingleObject	Pode esperar um processo sair
execve	(none)	CrieProcesso = fork + execve
exit	ExitProcess	Termine a execução
open	CreateFile	Crie um arquivo ou abra um arquivo existente
close	CloseHandle	Feche um arquivo
read	ReadFile	Leia dados de um arquivo
write	WriteFile	Escreva dados para um arquivo
lseek	SetFilePointer	Mova o ponteiro de posição do arquivo
stat	GetFileAttributesEx	Obtenha os atributos do arquivo
mkdir	CreateDirectory	Crie um novo diretório
rmdir	RemoveDirectory	Remova um diretório vazio
link	(none)	Win32 não suporta ligações (link)
unlink	DeleteFile	Destrua um arquivo existente
mount	(none)	Win32 não suporta mount
umount	(none)	Win32 não suporta mount
chdir	SetCurrentDirectory	Altere o diretório de trabalho atual
chmod	(none)	Win32 não suporta segurança (embora NT suporte)
kill	(none)	Win32 não suporta sinais
time	GetLocalTime	Obtenha o horário atual



Algumas chamadas da interface API Win32



Linux SysCall table

- <http://www.di.uevora.pt/~lmr/syscalls.html>



INFRA-ESTRUTURA DE SOFTWARE

Processos e Threads

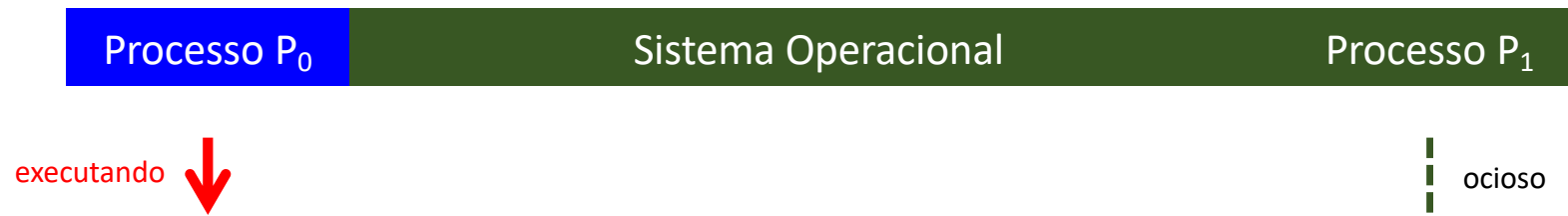
SO: Multitarefa

Processo P_0

Sistema Operacional

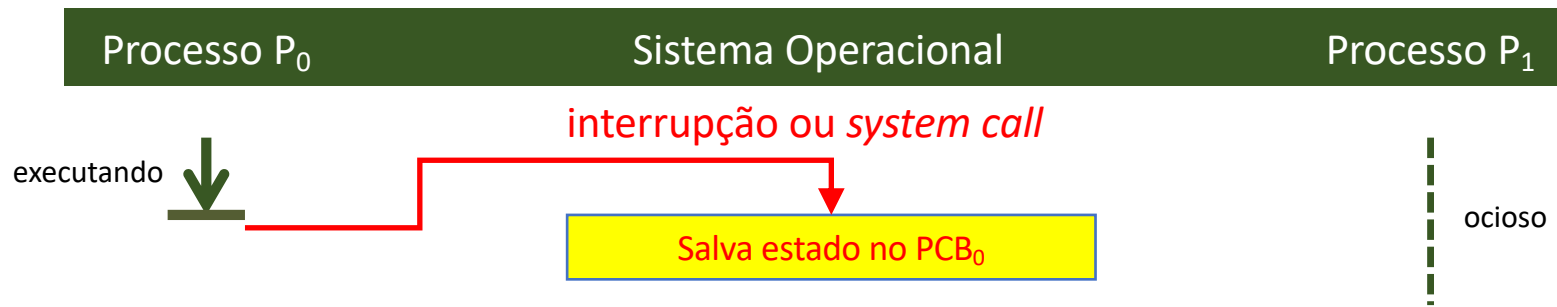
Processo P_1

SO: Multitarefa



Processo P₀ em execução

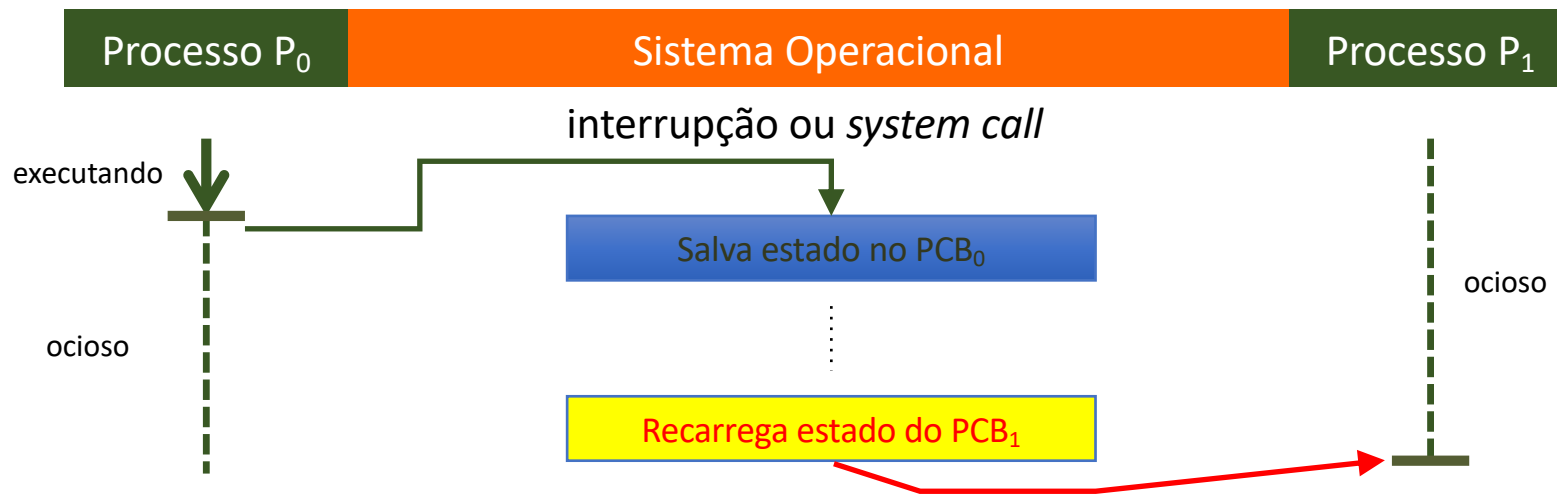
SO: Multitarefa



Salva estado do Processo P_0 no PCB_0

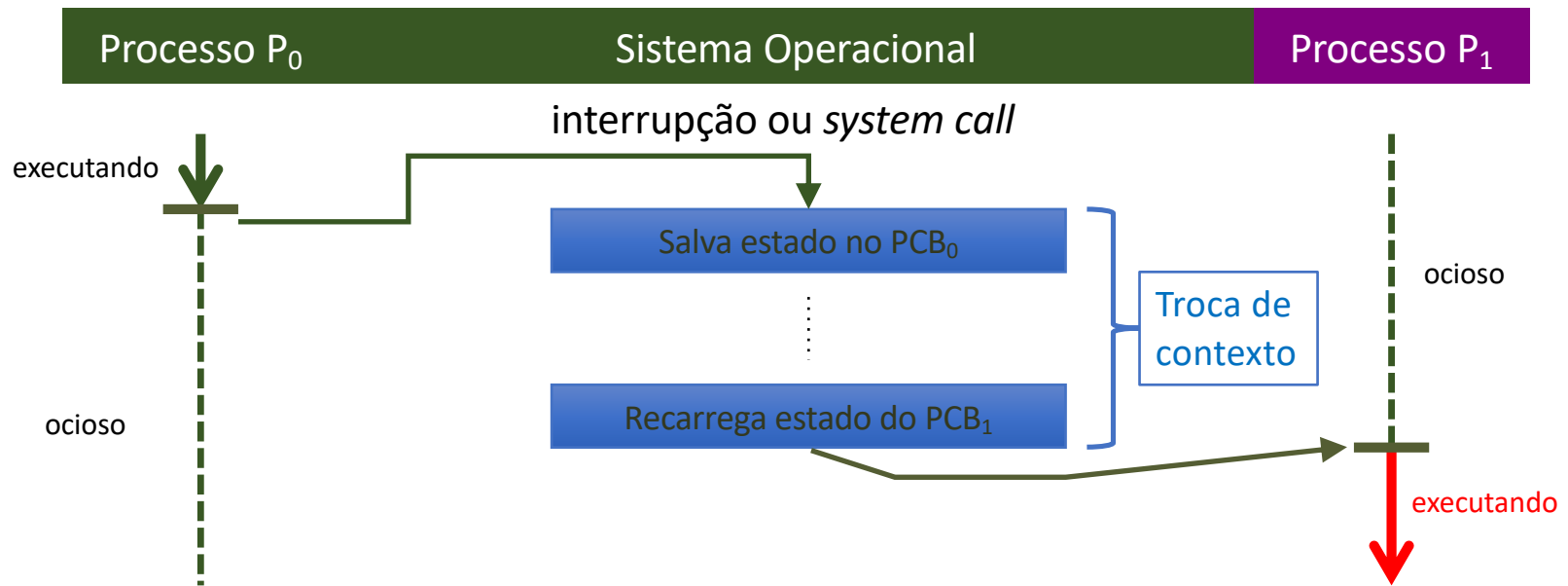
Process Control Block:
informações de
contexto do processo

SO: Multitarefa



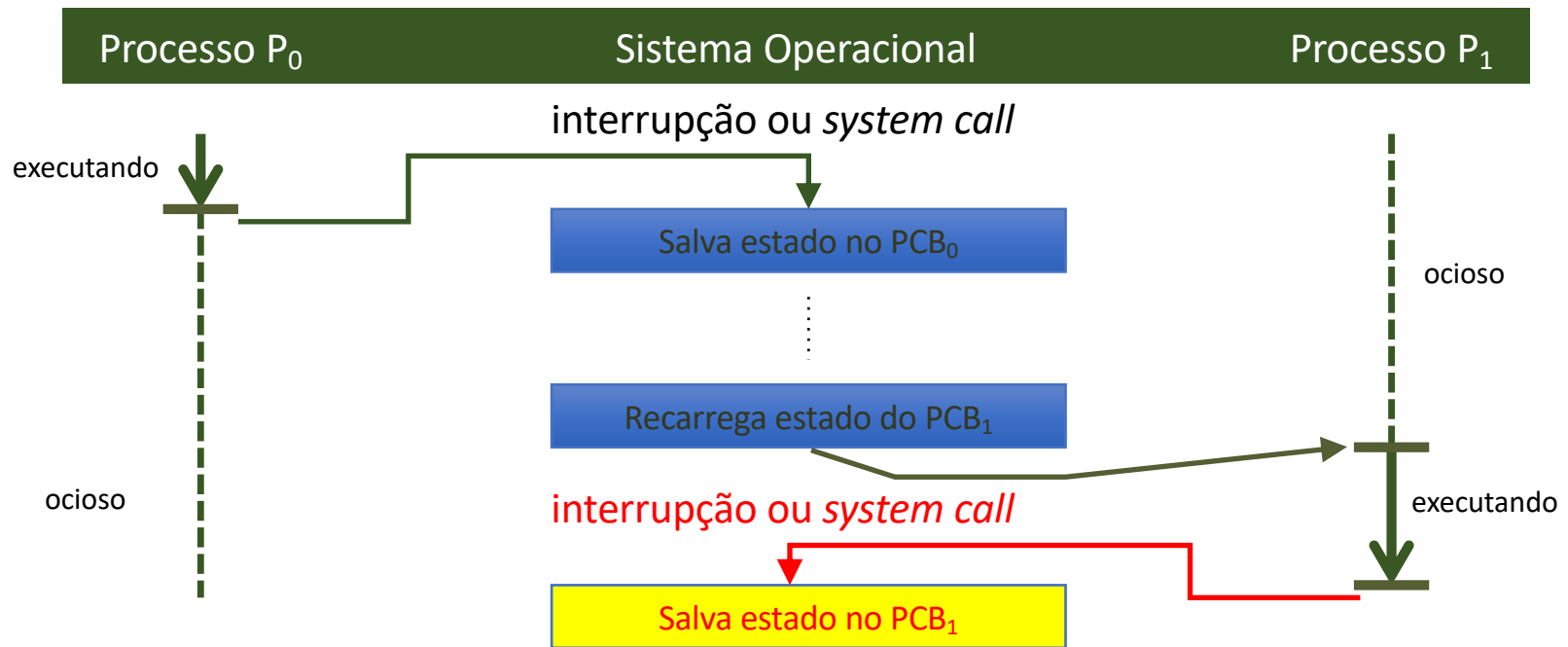
Estado do Processo P_1 é recarregado do PCB₁

SO: Multitarefa

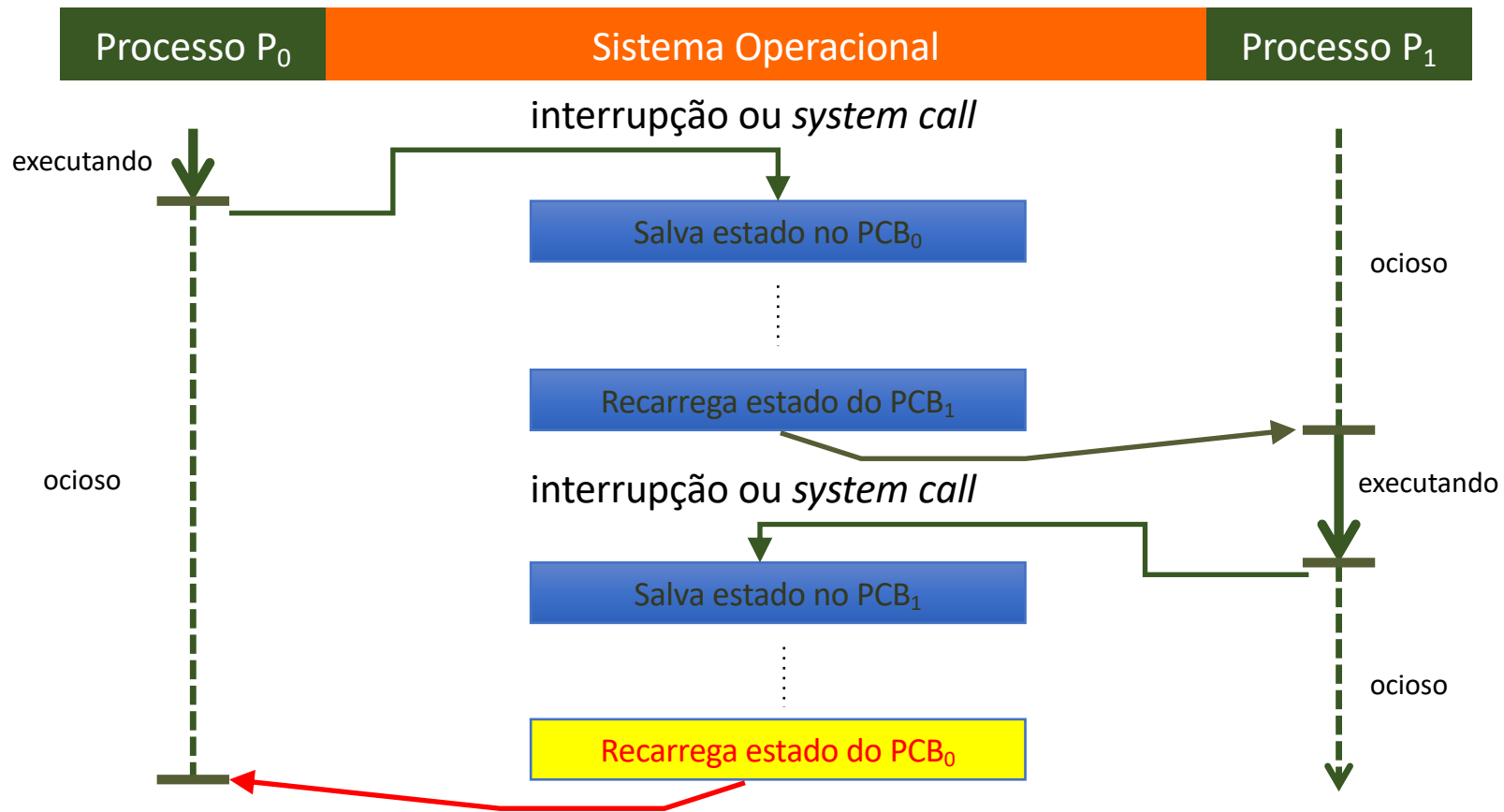


Processo P₁ em execução

SO: Multitarefa

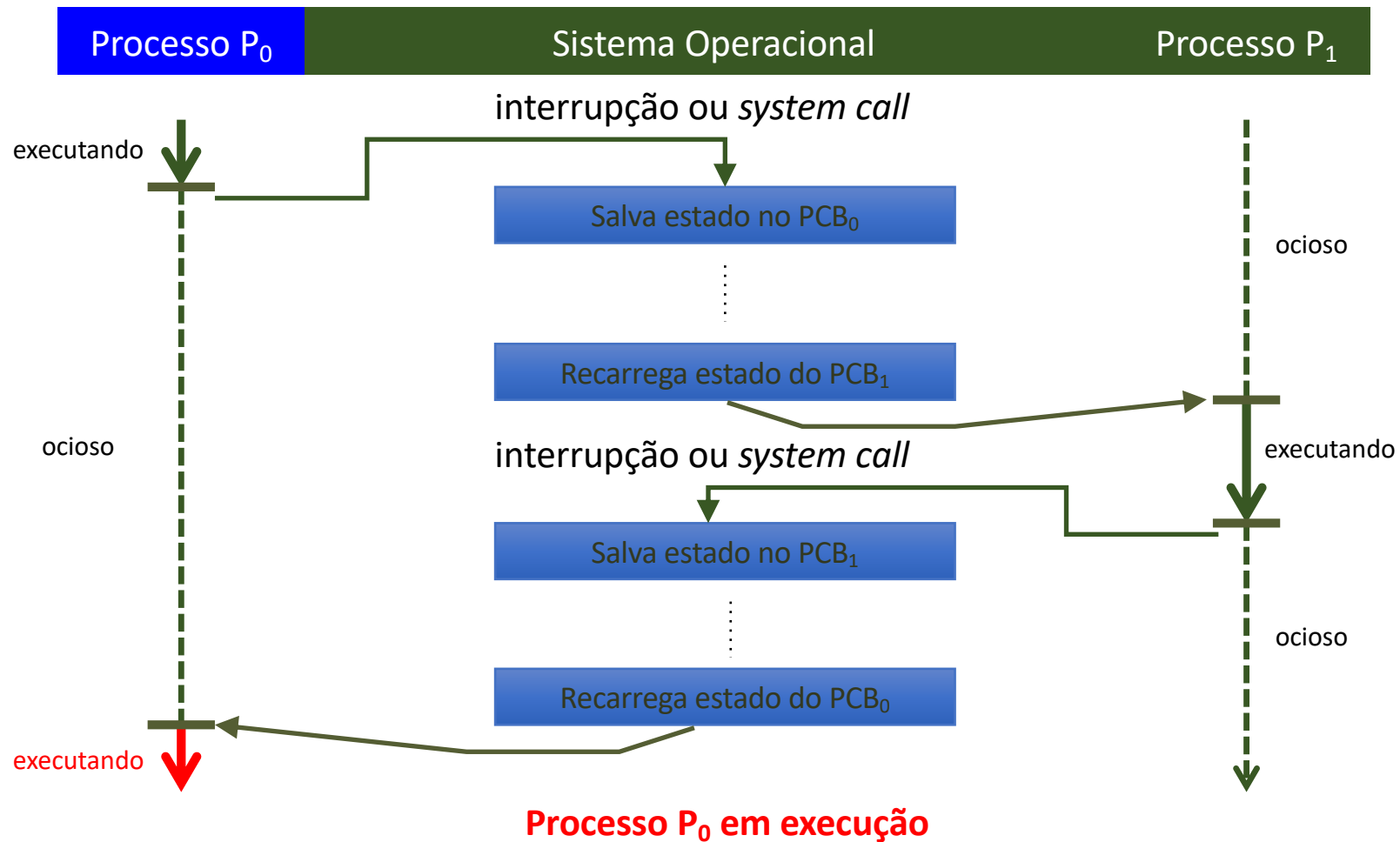


SO: Multitarefa



Estado do Processo P_0 é recarregado do PCB₀

SO: Multitarefa





Implementação de Processos

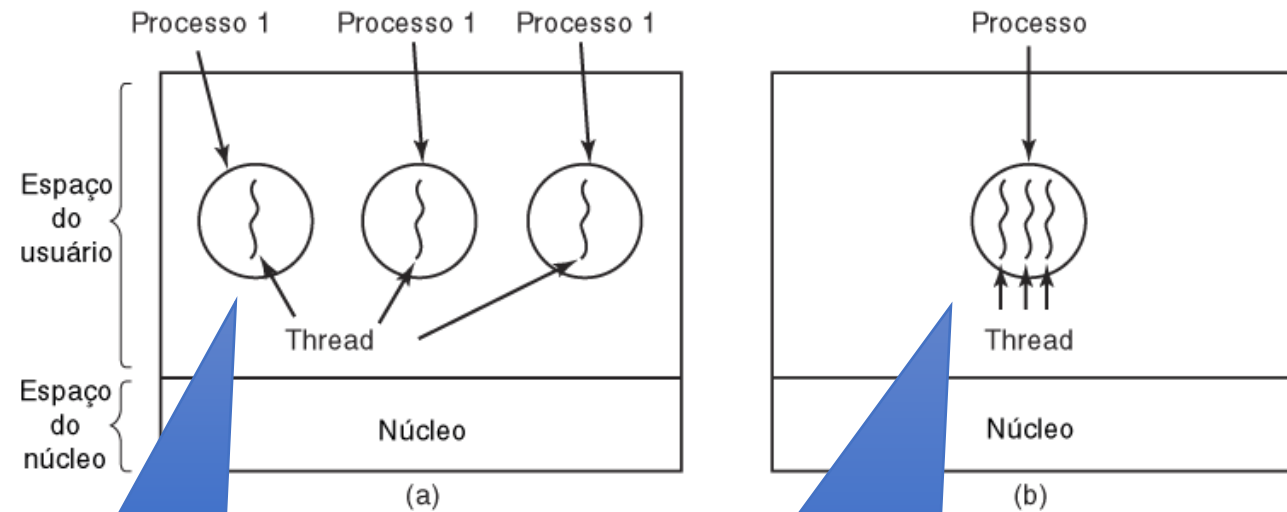
Gerenciamento de processos	Gerenciamento de memória	Gerenciamento de arquivos
Registradores Contador de programa Palavra de estado do programa Ponteiro de pilha Estado do processo Prioridade Parâmetros de escalonamento Identificador (ID) do processo Processo pai Grupo do processo Sinais Momento em que o processo iniciou Tempo usado da CPU Tempo de CPU do filho Momento do próximo alarme	Ponteiro para o segmento de código Ponteiro para o segmento de dados Ponteiro para o segmento de pilha	Diretório-raiz Diretório de trabalho Descritores de arquivos Identificador (ID) do usuário Identificador (ID) do grupo

Notificação da ocorrência de um evento
Exs.:
SIGILL - Illegal instruction
SIGINT - Interrupt
SIGKILL - Kill (terminate immediately)
SIGSYS - Bad system call
SIGXCPU - CPU time limit exceeded
SIGXFSZ - File size limit exceeded

Campos da entrada de uma tabela de processos (**contexto**)



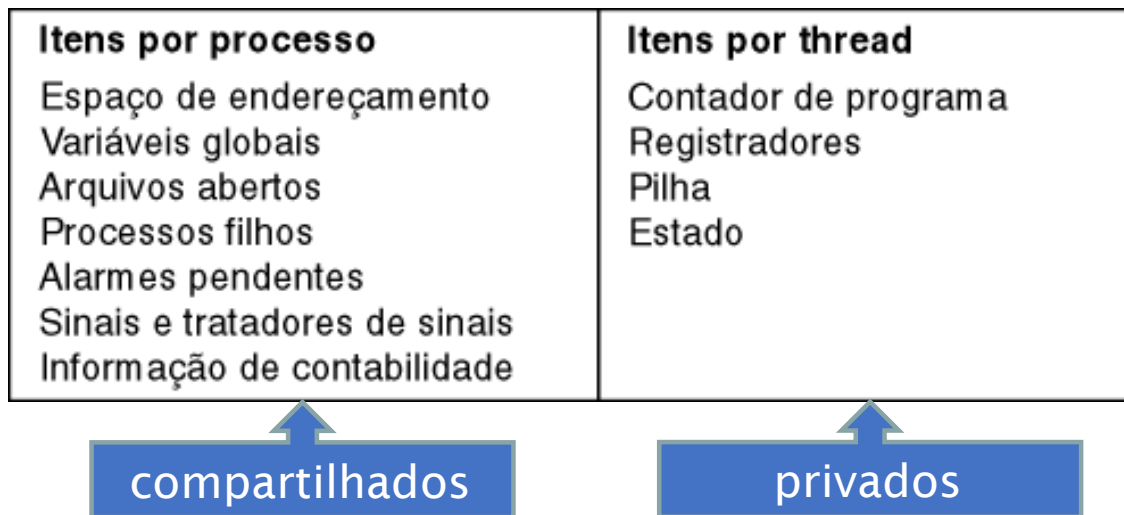
Threads: O Modelo (I)



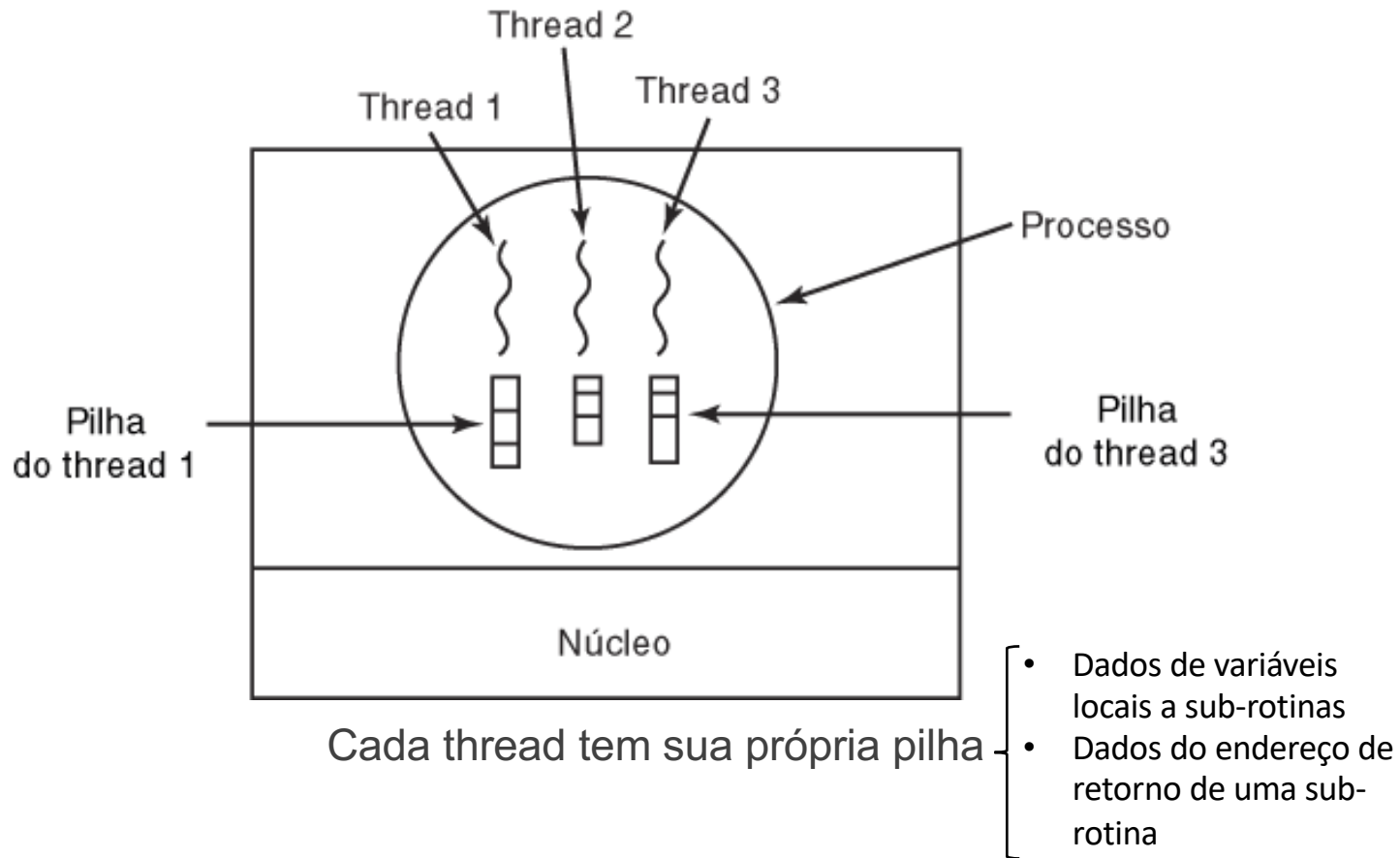
Cada processo tem um espaço de endereçamento e uma única linha (*thread*) de controle/execução

No entanto, há várias situações em que é desejável ter múltiplas linhas de controle no mesmo espaço de endereçamento rodando em "paralelo" como se fossem processos (*leves*) separados (exceto pelo *espaço de endereçamento compartilhado*)

O Modelo de Thread (2)

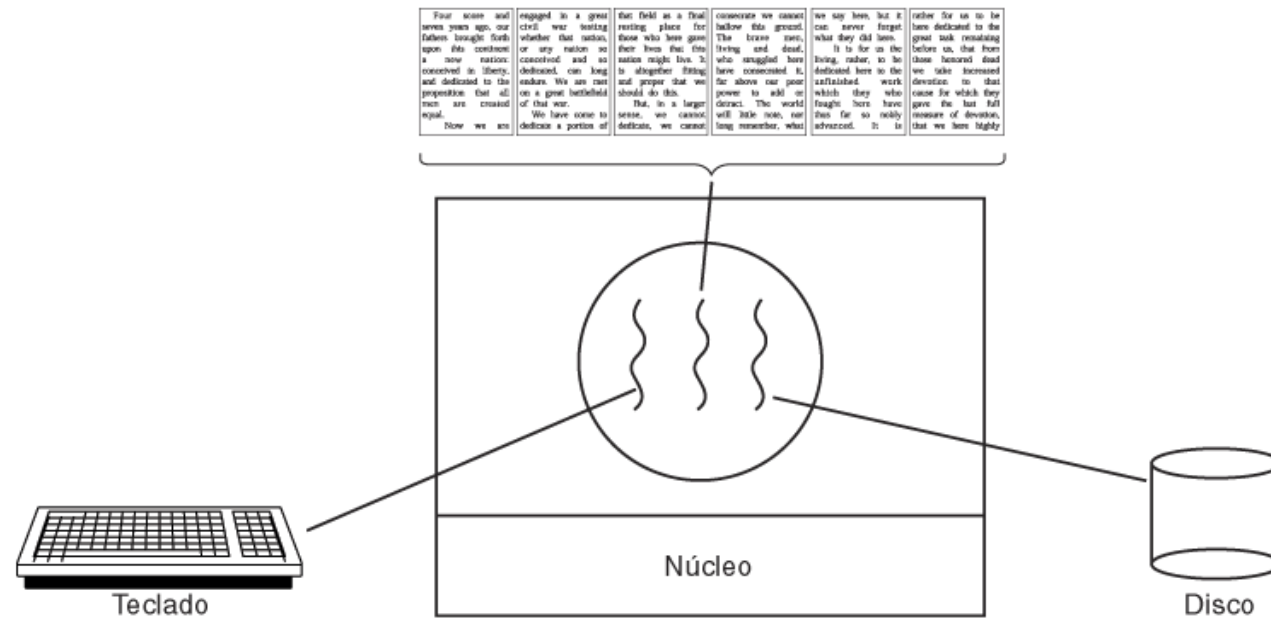


O Modelo de Thread (3)



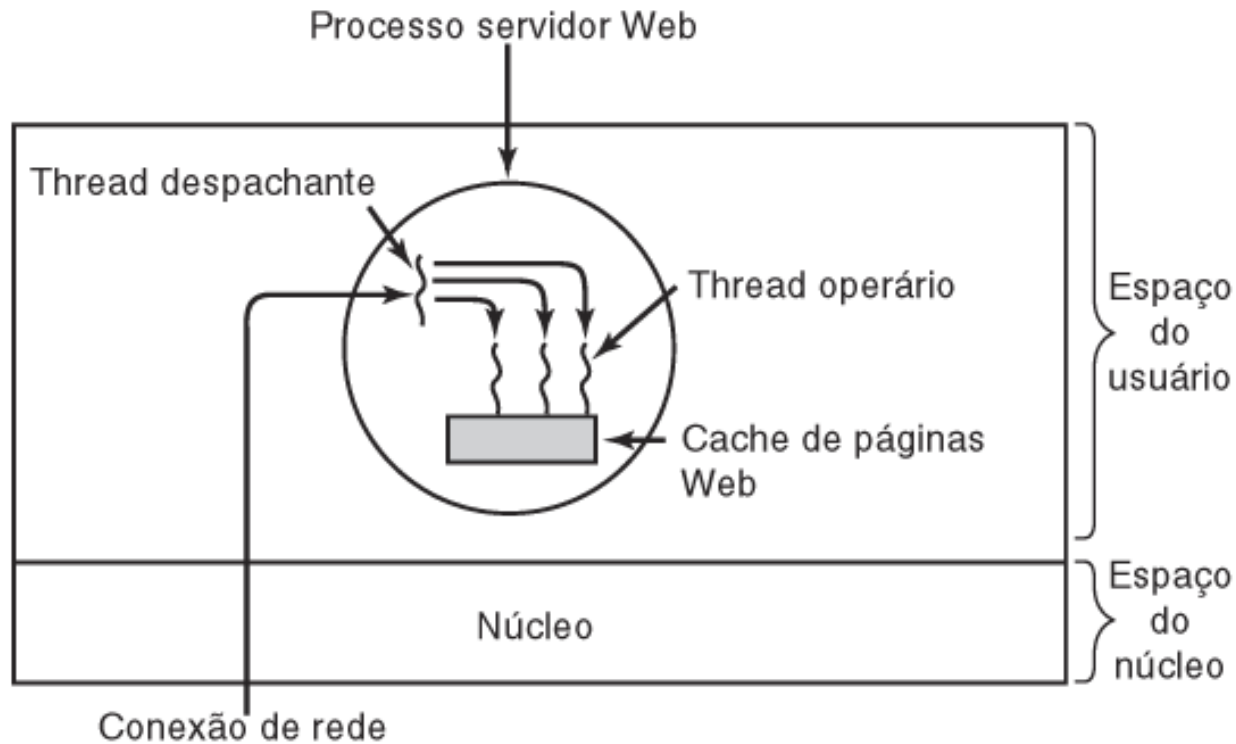


Uso de Thread (I)



Um processador de texto com três threads

Uso de Thread (2)

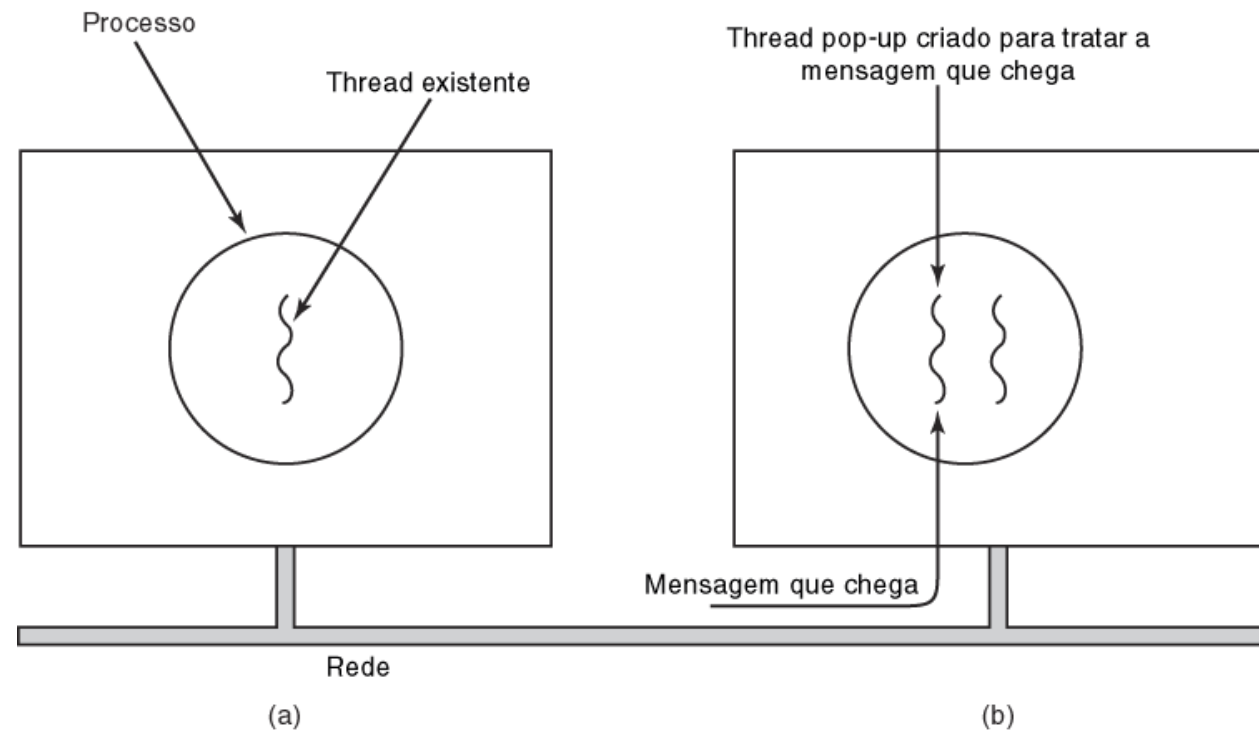


Um servidor web com múltiplas threads
vs um serviço Web com múltiplos servidores (mais adiante: SD)





Criação de uma nova thread quando chega uma mensagem



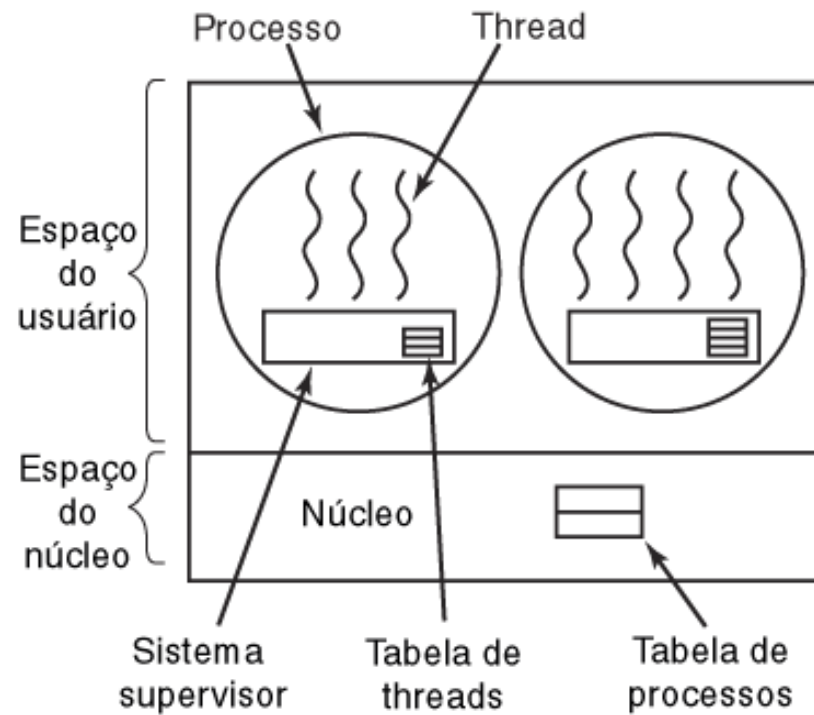


Tipos de Threads

- Threads = processos leves
- **Threads de usuário**
 - SO gerencia tabela de processos
- **Threads de núcleo**
 - SO gerencia tabelas de processos e **de threads (de núcleo)**



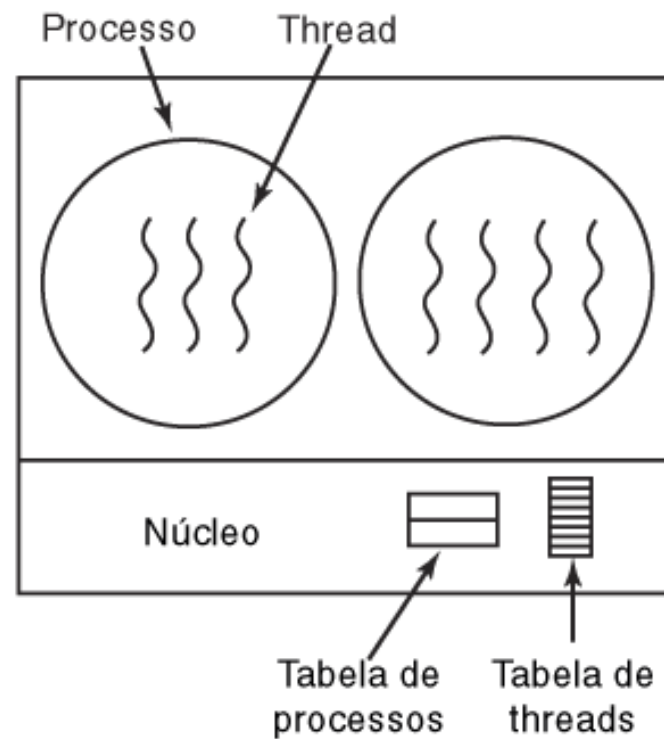
Implementação de Threads de Usuário



Um pacote de threads de usuário



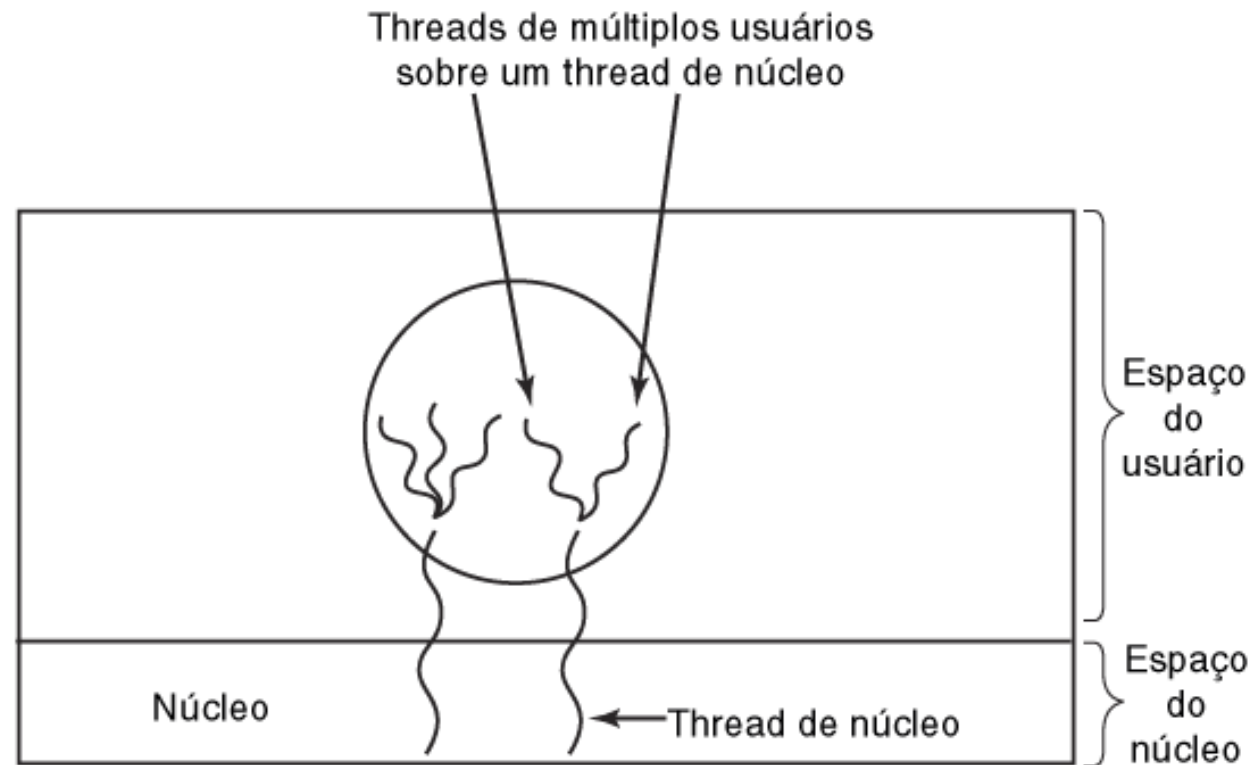
Implementação de Threads de Núcleo



Um pacote de threads gerenciado pelo núcleo



Implementações Híbridas



Multiplexação de threads de usuário sobre threads de núcleo



Motivação para Threads: Concorrência

- Problemas:
 - Programas que precisam de mais poder computacional
 - Dificuldade de implementação de CPUs mais rápidas
- Solução:
 - Construção de computadores capazes de executar várias tarefas “simultaneamente”



Concorrência vs Paralelismo

Concurrency is about **dealing with** lots of things at once.
Parallelism is about **doing** lots of things at once.

In programming, **concurrency** is the composition of independently executing processes, while **parallelism** is the simultaneous execution of (possibly related) computations.

Andrew Gerrand
<https://blog.golang.org/concurrency-is-not-parallelism>



Razões para ter processos leves (*threads*)

- Em muitas aplicações, várias atividades acontecem ao mesmo tempo – **mundo real**
 - O **modelo de programação** (*modelando o mundo real*) se torna mais simples (ou *realista*) decompondo uma aplicação em várias *threads* sequenciais que executam em “paralelo”
- Dado que *threads* são mais leves do que processos, elas são mais fáceis (rápidas) de criar e destruir
 - Criar uma *thread* pode ser **10-100 vezes mais rápido** que criar um processo
 - Quando a necessidade do número de *threads* muda dinamicamente e rapidamente, esta propriedade é bastante útil



Comparação de desempenho: processo x thread

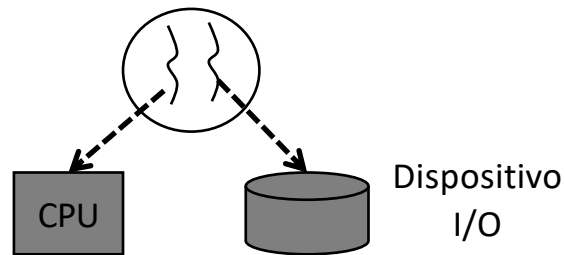
Plataforma	fork()	pthread_create()
AMD 2.4 GHz Opteron (8 cpus/node)	41.07	0.66
IBM 1.9 GHz POWER5 p5575 (8 cpus/node)	64.24	1.75
IBM 1.5 GHz POWER4 (8 cpus/node)	104.05	2.01
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.64
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	2.03

Tempos em ms



Razões para ter *threads* (cont)

- Ganhos de velocidade em processos onde atividades de I/O e de computação (CPU) podem ser sobrepostas



- ***Threads não levam a ganhos de desempenho quando todas são CPU-bound***

segue...



Tipos de Processo

(incl. *thread* – processo leve)

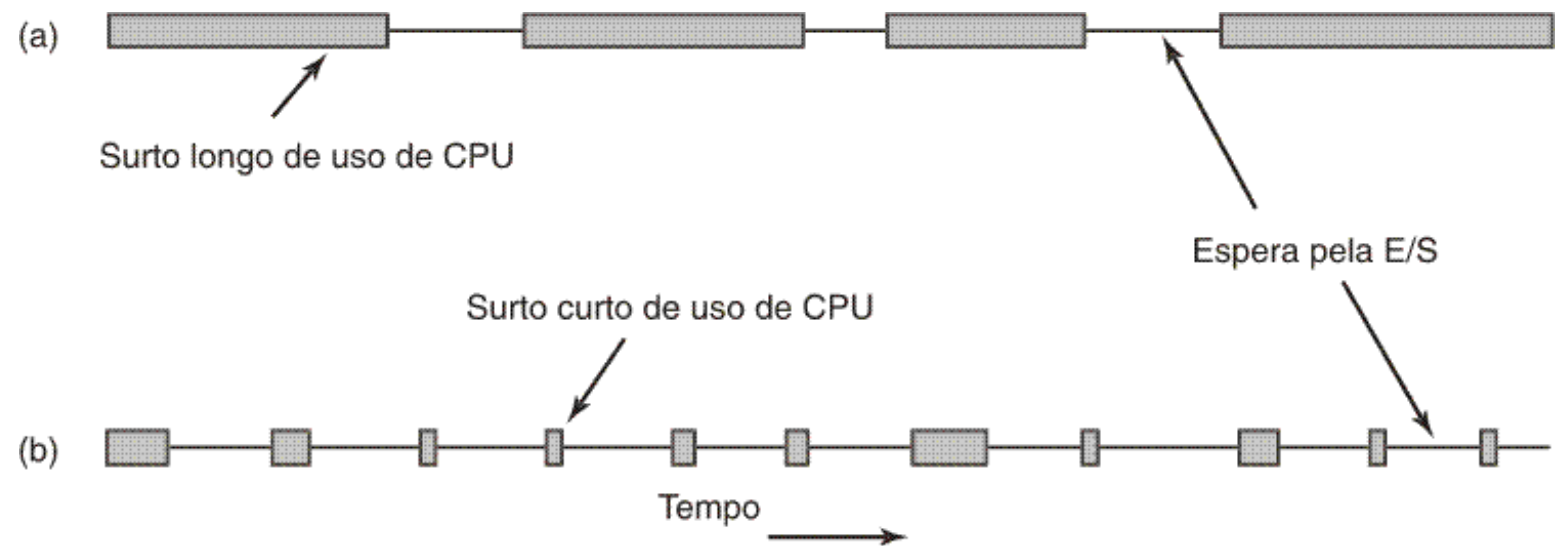
- CPU-bound:
 - Se o processo gasta a maior parte do seu tempo usando a CPU ele é dito orientado à computação (compute-bound ou CPU-bound)
 - processos com longos tempos de execução e baixo volume de comunicação entre processos
 - ex: aplicações científicas, engenharia e outras aplicações que demandam alto desempenho de computação
- I/O-bound:
 - Se um processo passa a maior parte do tempo esperando por dispositivos de E/S, diz-se que o processo é orientado à E/S (I/O-bound)

➤ processos I/O-bound devem ter prioridade sobre processos CPU-bound



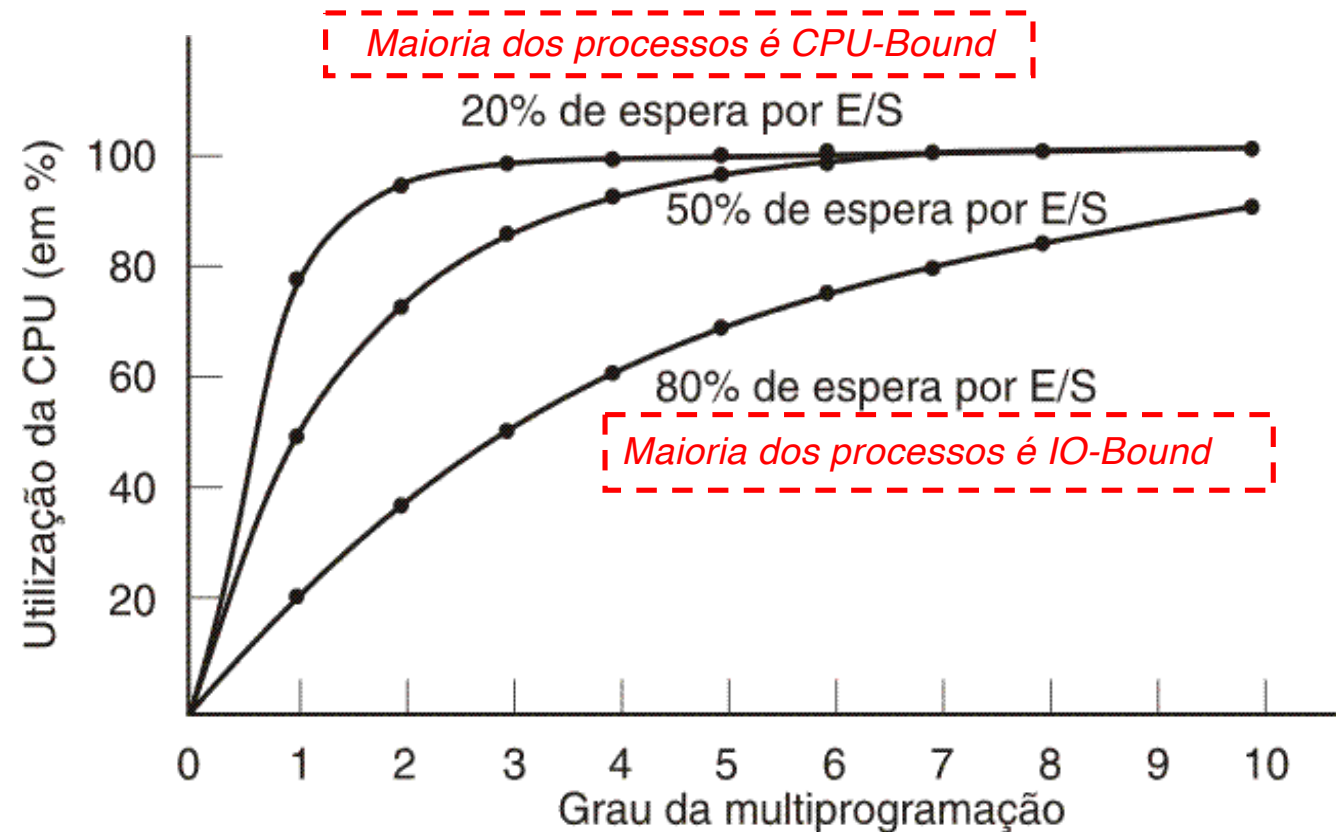
Comportamentos de Processos

- Surtos de uso da CPU alternam-se com períodos de espera por E/S
 - a. um processo orientado a CPU
 - b. um processo orientado a E/S





Modelagem de Multiprogramação comparação entre processos CPU-bound e IO-bound



Utilização da CPU como uma função do número de processos na memória



ESCALONAMENTO

Decidindo qual processo vai executar



Datas importantes

Data	Dia e Hora	Atividade	Local
27/03/19	QUA, 10-12h	Prática de Linux e C/C++	LabG2
29/03/19	SEX, 08-10h	Prática de C/C++ e Concorrência	LabG2
03/04/19	QUA, 10-12h	Exercício de Escalonamento de Processos	D-002
10/04/19	QUA, 10-12h	1o. EE: Exercício de concorrência	LabG2



Escalonamento de processos

- Quando dois ou mais processos estão prontos para serem executados, o sistema operacional deve decidir qual deles vai ser executado primeiro
- A parte do sistema operacional responsável por essa decisão é chamada **escalador**, e o algoritmo usado para tal é chamado de **algoritmo de escalonamento**



Comportamento escalonamento-processo

Processos I/O-bound têm “prioridade” no escalonamento

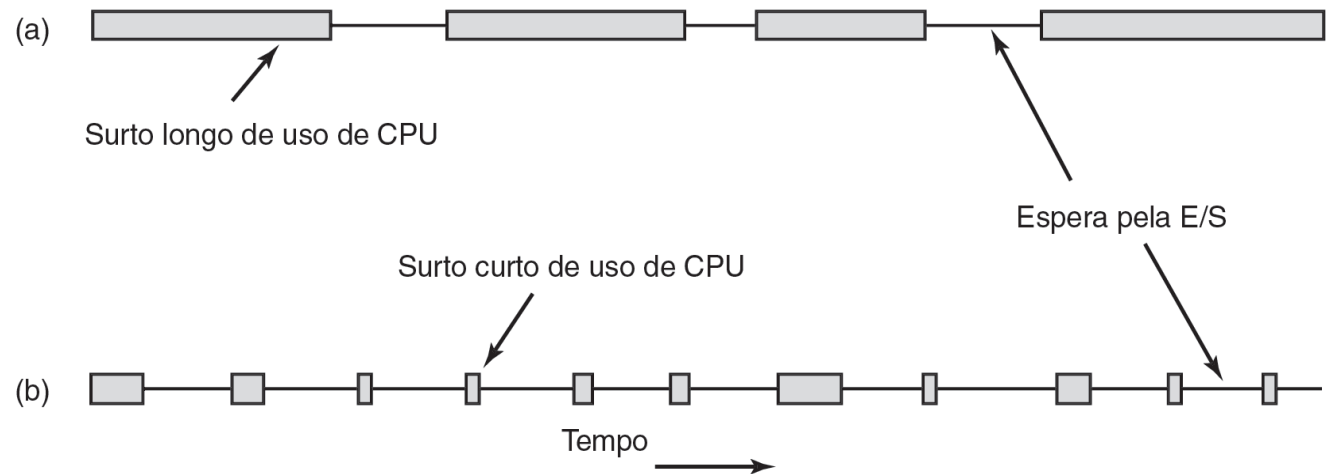


Figura 2.31 Usos de surtos de CPU se alternam com períodos de espera por E/S. (a) Um processo orientado à CPU. (b) Um processo orientado à E/S.

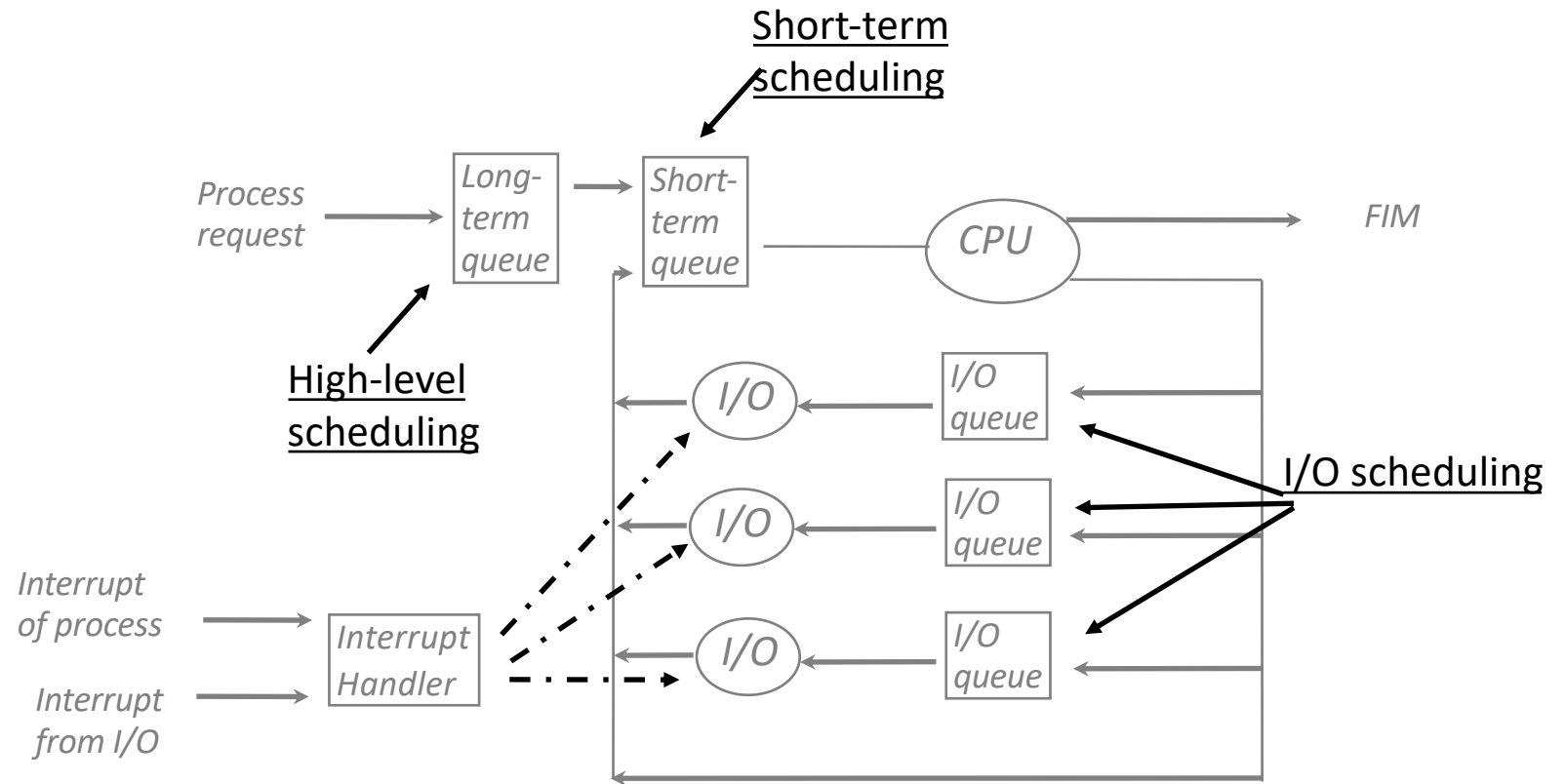


Filas de Escalonamento

- High-level
 - Decide quantos programas são admitidos no sistema
 - Aloca memória e cria um processo
 - Controla a long-term queue
- Short-term
 - Decide qual processo deve ser executado
 - Controla a short-term queue
- I/O
 - Decide qual processo (com I/O) pendente deve ser tratado pelo dispositivo de I/O
 - Controla a I/O queue



Filas de Escalonamento





Categorias de Algoritmos de Escalonamento

- Em lote (batch)

batch: uma quantidade de mercadorias produzidas de uma só vez. Ex. Uma 'fornada' de pão 😊

- Interativo

- Tempo-real



Objetivos dos algoritmos de escalonamento

Todos os sistemas

Justiça — dar a cada processo uma porção justa da CPU

Aplicação da política — verificar se a política estabelecida é cumprida

Equilíbrio — manter ocupadas todas as partes do sistema

Sistemas em lote

Vazão (*throughput*) — maximizar o número de tarefas por hora

Tempo de retorno — minimizar o tempo entre a submissão e o término

Utilização de CPU — manter a CPU ocupada o tempo todo

Sistemas interativos

Tempo de resposta — responder rapidamente às requisições

Proporcionalidade — satisfazer às expectativas dos usuários

Sistemas de tempo real

Cumprimento dos prazos — evitar a perda de dados

Previsibilidade — evitar a degradação da qualidade em sistemas multimídia

■ **Tabela 2.8** Alguns objetivos do algoritmo de escalonamento sob diferentes circunstâncias.



Tipos de Escalonamento

- Mecanismos de Escalonamento
 - Preemptivo x Não-preemptivo
- Políticas de Escalonamento
 - Round-Robin
 - FIFO (First-In First-Out)
 - Híbridos
 - Partições de Lote (Batch)
 - MFQ - Multiple Feedback Queue
 - SJF – Shortest Job First
 - SRJN – Shortest Remaining Job Next

Diz-se que um algoritmo/sistema operacional é **preemptivo** quando um processo entra na CPU e o mesmo pode ser retirado (da CPU) antes do término da sua execução



Problema das trocas de processos

- Mudar de um processo para outro requer um certo tempo para a administração — salvar e carregar registradores e mapas de memória, atualizar tabelas e listas do SO, etc
- Isto se chama **troca de contexto**
- Suponha que esta troca dure 5 ms
- Suponha também que o quantum está ajustado em 20 ms
- Com esses parâmetros, após fazer 20 ms de trabalho útil, a CPU terá que gastar 5 ms com troca de contexto. Assim, 20% do tempo de CPU (5 ms a cada 25 ms) é gasto com o *overhead* administrativo...



Solução?

- Para melhorar a eficiência da CPU, poderíamos ajustar o quantum para 500 ms
 - Agora o tempo gasto com troca de contexto é menos do que 1% - “desprezível”...
- Considere o que aconteceria se dez usuários apertassem a tecla <ENTER> exatamente ao mesmo tempo, disparando cada um processo:
 - Dez processos serão colocados na lista de processo aptos a executar
 - Se a CPU estiver ociosa, o primeiro começará imediatamente, o segundo não começará cerca de $\frac{1}{2}$ segundo depois, e assim por diante
 - O “azarado” do último processo somente começará a executar 5 segundos depois do usuário ter apertado <ENTER>, isto se todos os outros processos tiverem utilizado todo o seu quantum
 - Muitos usuários vão achar que o tempo de resposta de 5 segundos para um comando simples é “muita” coisa



“Moral da estória”

- Ajustar um *quantum* muito pequeno causa muitas trocas de contexto e diminui a eficiência da CPU, ...
- mas ajustá-lo para um valor muito alto causa um tempo de resposta inaceitável para pequenas tarefas interativas

Quantum grande:

Diminui número de mudanças de contexto e *overhead* do

S.O., *mas...*

Ruim para processos interativos



Lembrança: Datas importantes

Data	Dia e Hora	Atividade	Local
27/03/19	QUA, 10-12h	Prática de Linux e C/C++	LabG2
29/03/19	SEX, 08-10h	Prática de C/C++ e Concorrência	LabG2
03/04/19	QUA, 10-12h	Exercício de Escalonamento de Processos	D-002
10/04/19	QUA, 10-12h	1o. EE: Exercício de concorrência	LabG2

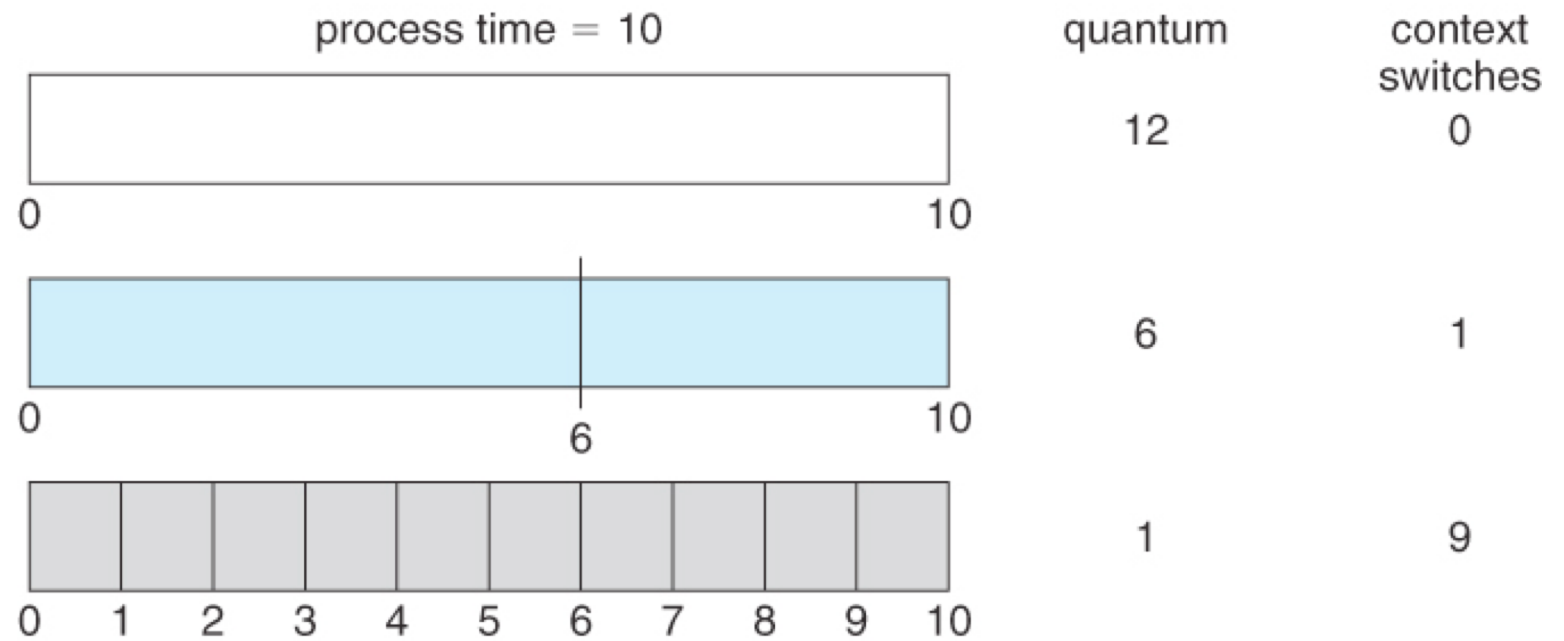


ESCALONAMIENTO DE PROCESSOS

Algoritmos (cont.)



Relembrando: *Quantum* e Troca de Contexto





Categorias de Algoritmos de Escalonamento (Agendamento)

- Em lote (batch)

- Interativo

- Tempo-real

- Híbrido

Earliest Deadline First (EDF)

- Preemptivo
- Considera o momento em que a resposta deve ser entregue
- Processo se torna mais prioritário quanto mais próximo do deadline
- Algoritmo complexo

Priorização dinâmica



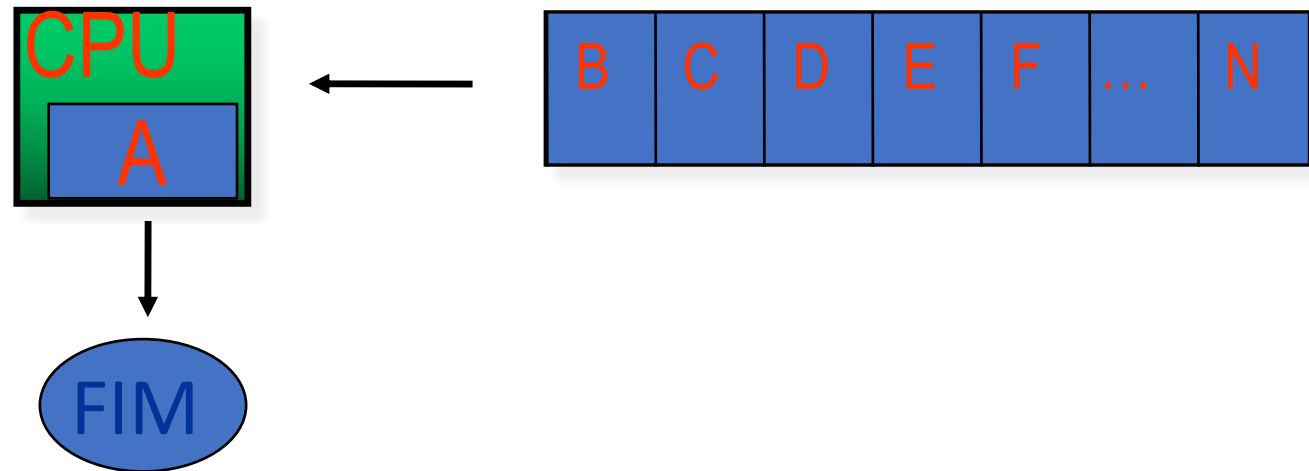
Escalonamento em Sistemas em Lote

- First-come first-served (ou FIFO)
- Shortest Job First (job mais curto primeiro) – SJF
- Shortest Remaining Time/Job First/Next – SRTF/SRJN



First-In First-Out (FIFO)

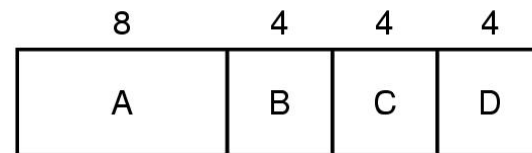
- Uso de uma lista de processos sem prioridade
- Escalonamento não-preemptivo
- Simples e justo



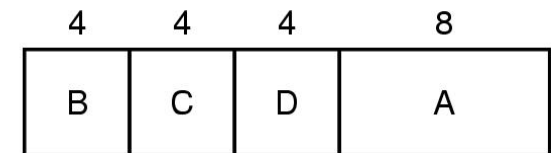


Escalonamentos baseados no tempo de execução

- Shortest Job First (não-preemptivo)
- Shortest Remaining Job Next (preemptivo)
- Melhora o tempo de resposta
- Não é justo: pode causar estagnação (*starvation*)
 - Pode ser resolvida alterando a prioridade dinamicamente



(a)

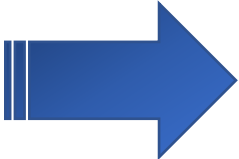


(b)

Exemplo de escalonamento *job mais curto primeiro* (*Shortest Job First – SJF*)

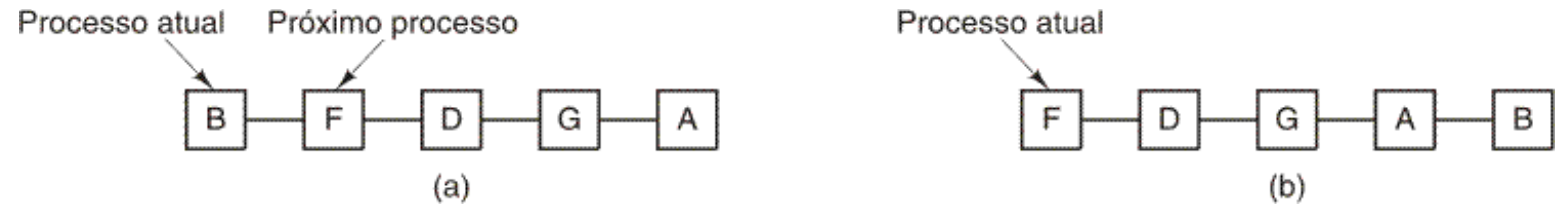


Escalonamento em Sistemas Interativos

- Round-robin
 - Priority
 - Multiple queues
 - Shortest process next
 - Guaranteed scheduling
 - Lottery scheduling
 - Fair-share scheduling
- 
- Chaveamento circular
 - Escalonamento por prioridades
 - Filas múltiplas
 - Processo mais curto é o próximo
 - Escalonamento garantido
 - Escalonamento por loteria
 - Escalonamento por fração justa

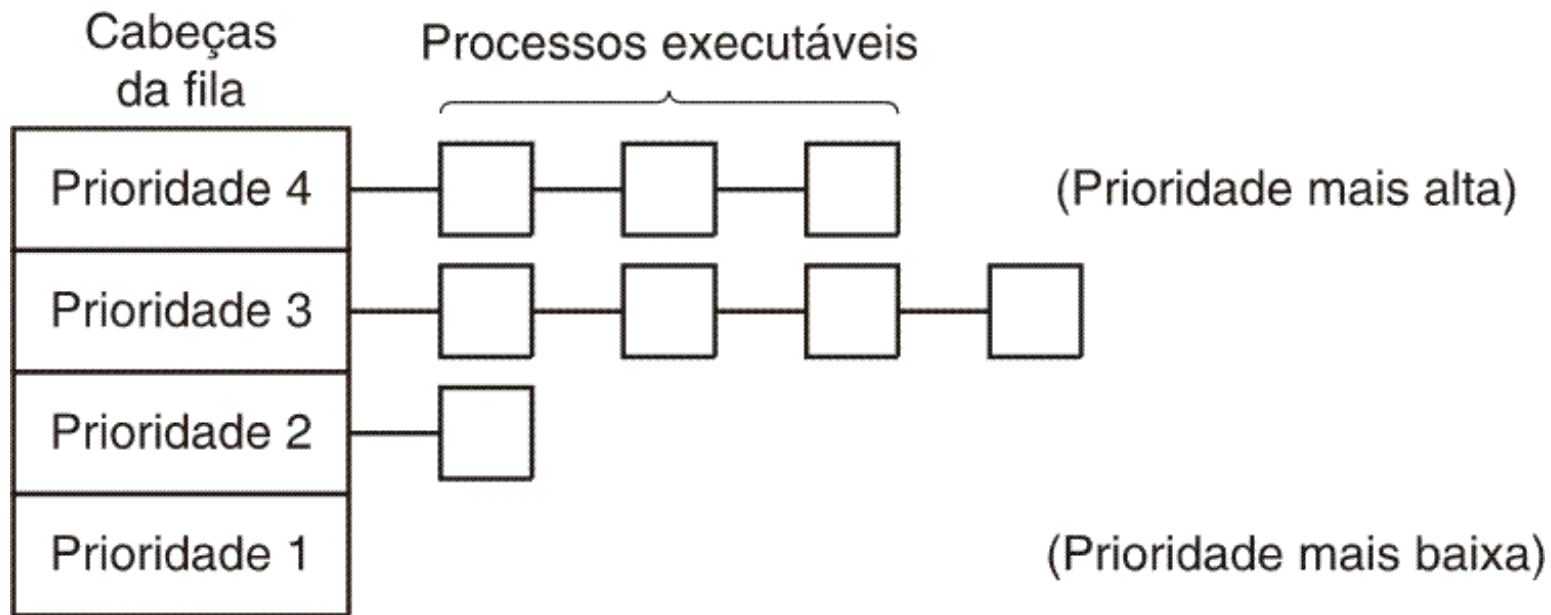


Escalonamento em Sistemas Interativos (1)



- Escalonamento por chaveamento/alternância circular (*round-robin*)
 - a) lista de processos executáveis
 - b) lista de processos executáveis depois que B usou todo o seu quantum

Escalonamento em Sistemas Interativos (2)



Um algoritmo de escalonamento com quatro classes de **prioridade**



Escalonamento Híbrido

Multiple Feedback Queue (MFQ)

- Como saber a priori se o processo é CPU-bound ou I/O-bound?
- MFQ usa abordagem de prioridades dinâmicas
- Adaptação baseada no comportamento de cada processo
- Usado no VAX / VMS...

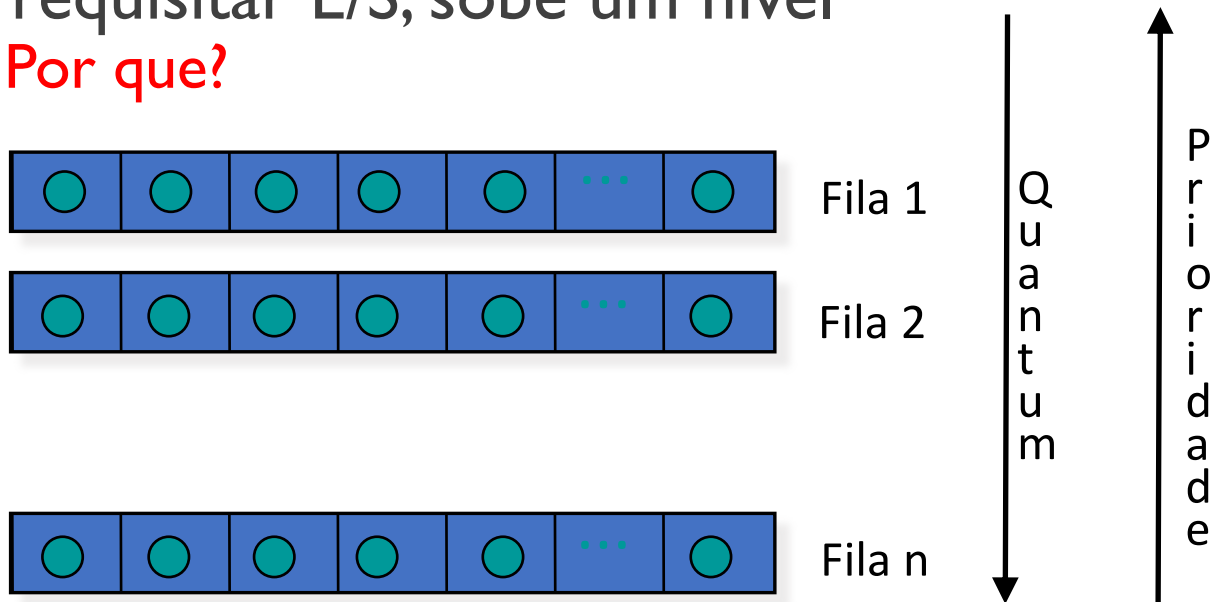
Segue...



Escalonamentos Híbridos

Multiple Feedback Queue

- Novos processos entram na primeira fila (prioridade mais alta)
- Se acabar o *quantum*, desce um nível
- Se requisitar E/S, sobe um nível
 - Por que?

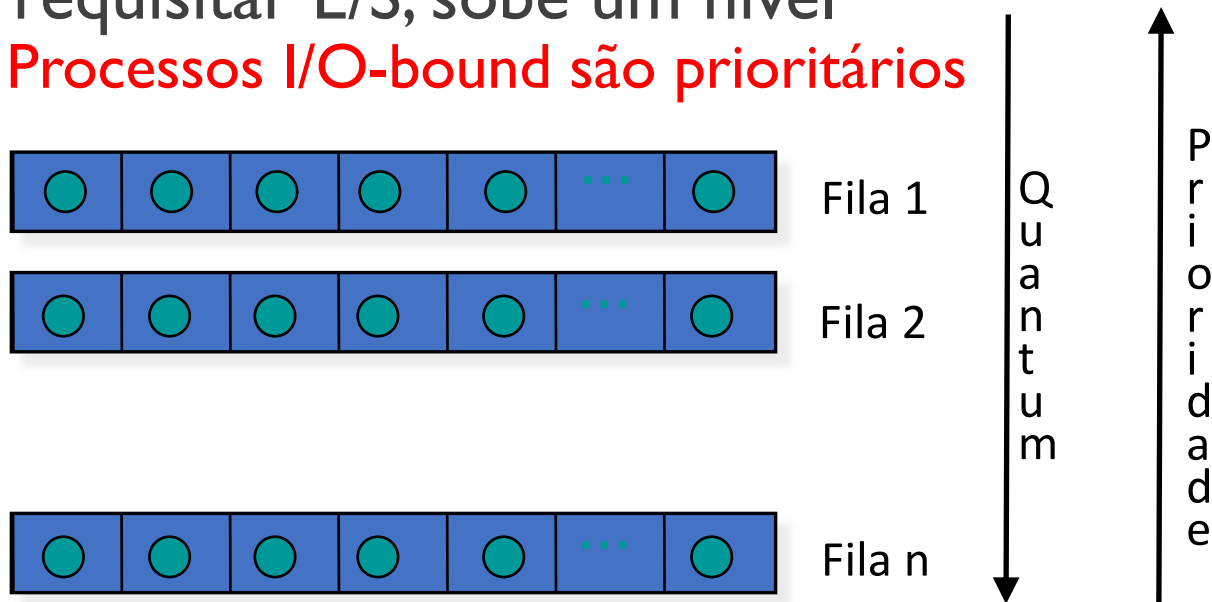




Escalonamentos Híbridos

Multiple Feedback Queue

- Novos processos entram na primeira fila (prioridade mais alta)
- Se acabar o *quantum*, desce um nível
- Se requisitar E/S, sobe um nível
 - Processos I/O-bound são prioritários





Algoritmos de Escalonamento

Em lote

- ✓ First-come first-served (ou FIFO)
- ✓ Shortest Job First (job mais curto primeiro) – SJF
- ✓ Shortest Remaining Job Next – SRJN

Em sistemas de tempo real

- ✓ Earliest Deadline First (EDF)

Em sistemas interativos

- ✓ Round-robin
- ✓ Priority
- ✓ Multiple queues (híbrido)
 - Shortest process next
 - Guaranteed scheduling
 - Lottery scheduling
 - Fair-share scheduling



Fair-share Scheduling

Uso da CPU distribuído igualmente entre usuários ou grupos de usuários (e não entre processos)

Exemplo I

- **Usuários** A, B, C, D, 1 processo cada:
 - $100\% \text{ CPU} / 4 = 25\%$ para cada usuário
 - Se B iniciar um segundo processo:
 - $25\% \text{ B} / 2 = 12,5\%$ para cada processo de B
 - Continua 25% para A, C, D
 - Se novo usuário E:
 - $100\% \text{ CPU} / 5 = 20\%$ para A, B, C, D, E



Fair-share Scheduling

Uso da CPU distribuído igualmente entre usuários ou grupos de usuários (e não entre processos)

Exemplo 2

- **Grupos** 1, 2, 3:
 - 100% CPU / 3 grupos = 33.3% por grupo
 - Grupo 1: $(33.3\% / 3 \text{ usuários}) = 11.1\%$ por usuário
 - Grupo 2: $(33.3\% / 2 \text{ usuários}) = 16.7\%$ por usuário
 - Grupo 3: $(33.3\% / 4 \text{ usuários}) = 8.3\%$ por usuário

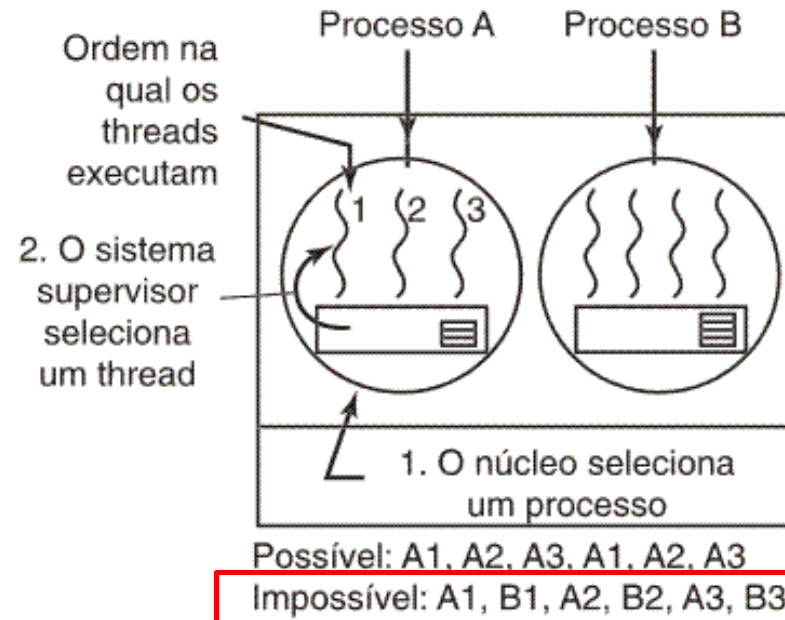


Algoritmos de Escalonamento

Round Robin	<ul style="list-style-type: none">• Quantum constante• Sem prioridade
Com Prioridade	<ul style="list-style-type: none">• Cada processo possui uma prioridade e o de maior prioridade executa primeiro• Para evitar que os processos de maior prioridade tomem conta do processador, a prioridade é decrementada
Menor Job Primeiro	<ul style="list-style-type: none">• Difícil estimar o tempo
Filas múltiplas	<ul style="list-style-type: none">• Criação de classes de Prioridades alocadas em diferentes filas
Garantido	<ul style="list-style-type: none">• Estimar (prometer) a um processo o tempo de sua execução e cumprir• Necessário conhecimento dos processos executando e a serem executados
Loteria	<ul style="list-style-type: none">• Distribuição de bilhetes que dão acesso à CPU• Lembra o escalonamento com prioridade, mas bilhetes são trocados entre processos



Escalonamento de Threads (I)

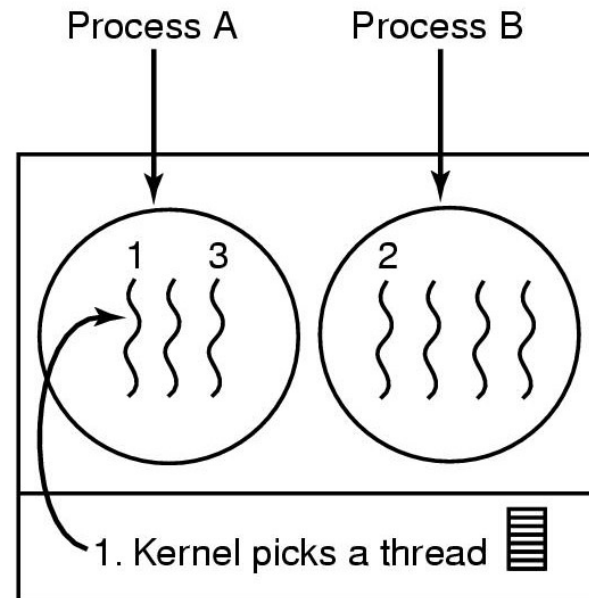


(a)

- Possível escalonamento de **threads de usuário**
- processo com quantum de 50-mseg
- threads executam 5 mseg por surto de CPU



Escalonamento de Threads (2)



Possible: A1, A2, A3, A1, A2, A3

Also possible: A1, B1, A2, B2, A3, B3

- Possível escalonamento de **threads de núcleo**
- processo com quantum de 50-mseg
- threads executam 5 mseg por surto de CPU



INFRAESTRUTURA DE SOFTWARE

Gerência de Processos



Recapitulando

- Processo
 - máquina de estados
 - PCB
- Interrupções
 - SW / Trap
 - HW (incl. Relógio)
- Chamadas ao Sistema (falando com o HW através do SO/Núcleo)
- Threads
- Concorrência
- Escalonamento



Algoritmos de Escalonamento

Em lote

- First-come first-served (ou FIFO)
- Shortest Job First (job mais curto primeiro) – SJF
- Shortest Remaining Job Next – SRJN

Em sistemas de tempo real

- Earliest Deadline First (EDF)

Em sistemas interativos

- Round-robin
- Priority
- Multiple queues (híbrido)
- Shortest process next
- Guaranteed scheduling
- Lottery scheduling
- Fair-share scheduling

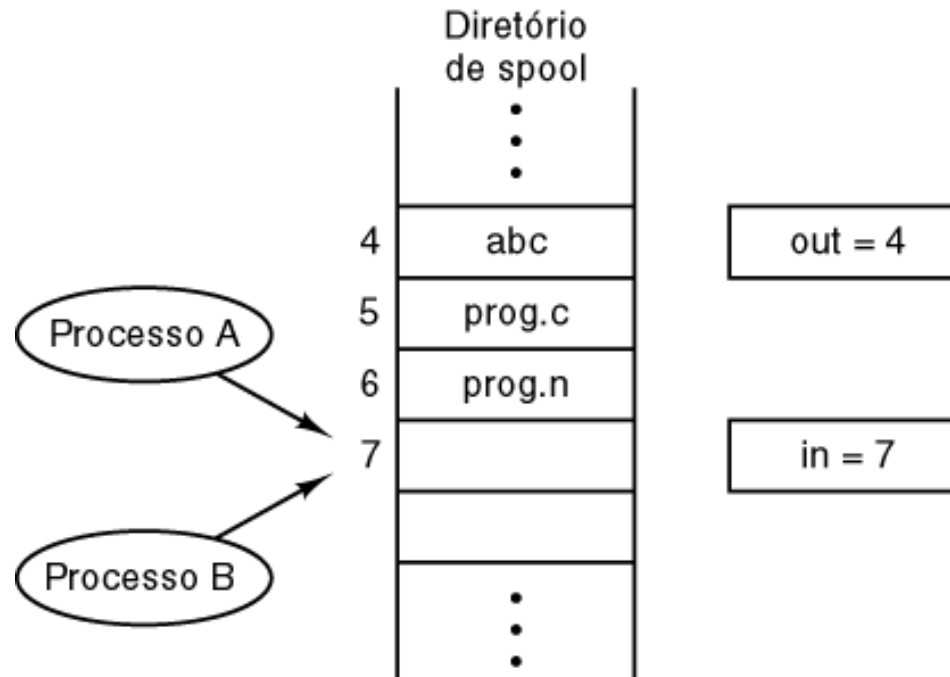


CONCORRÊNCIA

Conceitos (Continuação)



Condições de Disputa/Corrida



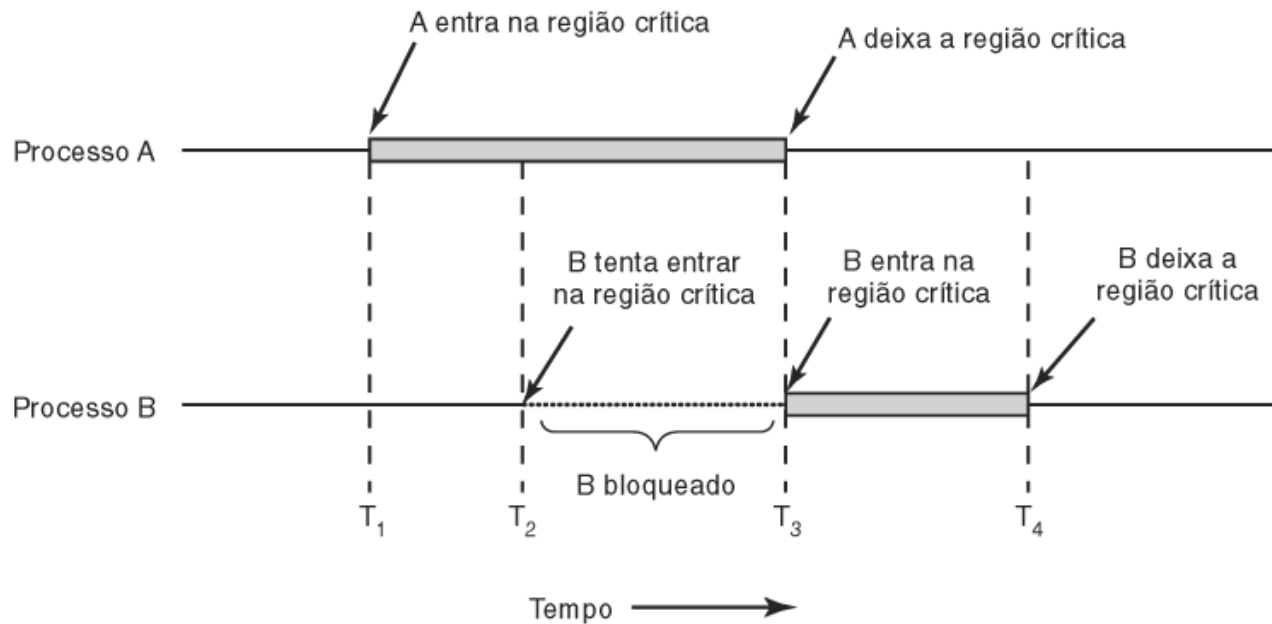
Dois processos querem ter acesso simultaneamente à memória compartilhada



Conceitos: Regiões Críticas (I)

- Quatro condições necessárias para prover **exclusão mútua**:
 - Nunca dois processos simultaneamente em uma região crítica
 - Não se pode considerar velocidades ou números de CPUs
 - Nenhum processo executando fora de sua região crítica pode bloquear outros processos
 - Nenhum processo deve esperar eternamente para entrar em sua região crítica

Regiões Críticas (2)



Exclusão mútua usando regiões críticas



Exclusão Mútua com Espera Ociosa (I)

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

(a)

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

(b)

Solução proposta para o problema da região crítica

(a) Processo 0.

(b) Processo 1.



Exclusão Mútua com Espera Ociosa (2)

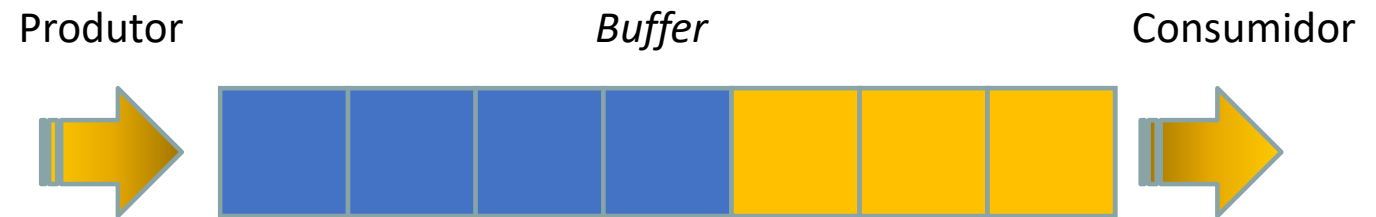
```
#define FALSE 0
#define TRUE 1
#define N      2          /* número de processos */
int tum;                /* de quem é a vez? */
int interested[N];      /* todos os valores inicialmente em 0 (FALSE) */
void enter_region(int process); /* processo é 0 ou 1 */
{
    int other;          /* número de outro processo */

    other = 1 - process; /* o oposto do processo */
    interested[process] = TRUE; /* mostra que você está interessado */
    tum = process;        /* altera o valor de tum */
    while (tum == process && interested[other] == TRUE) /* comando nulo */;
}
void leave_region(int process) /* processo: quem está saindo */
{
    interested[process] = FALSE; /* indica a saída da região crítica */
}
```

Solução de G. L. Peterson para exclusão mútua



Problema do Produtor-Consumidor



- se consumo $>$ produção
 - Buffer esvazia; Consumidor não tem o que consumir
- se consumo $<$ produção
 - Buffer enche; Produtor não consegue produzir mais

Dormir e Acordar (I)



```
#define N 100                                /* número de lugares no buffer */
int count = 0;                               /* número de itens no buffer */

void producer(void)
{
    int item;

    while (TRUE) {                           /* número de itens no buffer */
        item = produce_item( );              /* gera o próximo item */
        if (count == N) sleep( );            /* se o buffer estiver cheio, vá dormir */
        insert_item(item);                   /* ponha um item no buffer */
        count = count + 1;                   /* incremente o contador de itens no buffer */
        if (count == 1) wakeup(consumer);   /* o buffer estava vazio? */
    }
}
```

Problema do produtor-consumidor com uma condição de disputa fatal

Dormir e Acordar (2)

```
void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1) wakeup(producer);
        consume_item(item);
    }
}
```

Problema do produtor-consumidor com uma condição de disputa fatal





REGIÃO CRÍTICA E EXCLUSÃO MÚTUA

Conceitos Associados



Semáforo (I)

- **Semáforo** é uma variável que tem como função o controle de acesso a recursos compartilhados
- O valor de um semáforo indica **quantos** processos (ou *threads*) podem ter acesso a um recurso compartilhado
 - Para se ter **exclusão mútua**, só **um** processo executa por vez
 - Para isso utiliza-se um semáforo binário, com inicialização em 1
 - Esse semáforo binário atua como um **mutex**

Semáforo (2)

- As principais operações sobre semáforos são:
 - Inicialização: recebe um valor inteiro indicando a quantidade de processos que podem acessar um determinado recurso (exclusão mútua = 1, como dito antes)
 - Operação wait ou down ou P: decrementa o valor do semáforo. Se o semáforo está com valor zerado, o processo é posto para dormir.
 - Operação signal ou up ou V: se o semáforo estiver com o valor zero e existir algum processo adormecido, um processo será acordado. Caso contrário, o valor do semáforo é incrementado.
- As operações de **incrementar** e **decrementar** devem ser operações **atômicas**, ou **indivisíveis**, ou seja,
 - enquanto um processo estiver executando uma dessas **duas operações**, nenhum outro processo pode executar outra operação sob o mesmo semáforo, devendo esperar que o primeiro processo encerre sua operação.
 - essa obrigação evita condições de disputa entre vários processos





Semáforos (I)

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item( );
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
/* número de lugares no buffer */
/* semáforos são um tipo especial de int */
/* controla o acesso à região crítica */
```

```
#define N 100
int count = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item( );
        if (count == N) sleep( );
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}
```

```
/* número de lugares no buffer */
/* número de itens no buffer */
```

Sem mutex (semáforo): **não** evita condições de disputa

```
/* número de itens no buffer */
/* gera o próximo item */
/* se o buffer estiver cheio, vá dormir */
/* ponha um item no buffer */
/* incremente o contador de itens no buffer */
/* o buffer estava vazio? */
```

O problema do produtor-consumidor usando semáforos

Semáforos (2)



```
void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);           /* laço infinito */
        down(&mutex);         /* decresce o contador full */
        item = remove_item(); /* entra na região crítica */
        up(&mutex);           /* pega o item do buffer */
        up(&empty);           /* deixa a região crítica */
        consume_item(item);   /* incrementa o contador de lugares vazios */
    }
}
```

O problema do produtor-consumidor usando semáforos



Exemplo da necessidade de Semáforo



(a) Um **deadlock** potencial.

(b) Um **deadlock** real.

Assim como em **deadlock**, *threads* em **livelock** não podem progredir em sua execução. No entanto, as threads não ficam bloqueadas - elas ficam simplesmente ocupadas (vs bloqueadas – **deadlock**) respondendo umas às outras sem progresso

A principal diferença entre **livelock** e **deadlock** é que as threads não serão bloqueadas. Em vez disso, elas tentarão responder umas às outras continuamente



Monitores

```
monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;
  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;
  count := 0;
end monitor;
```

```
procedure producer;
begin
  while true do
  begin
    item = produce_item;
    ProducerConsumer.insert(item)
  end
end;
procedure consumer;
begin
  while true do
  begin
    item = ProducerConsumer.remove;
    consume_item(item)
  end
end;
```

○ problema do produtor-consumidor com **monitores**

- somente um procedimento está ativo por vez no monitor
- o buffer tem N lugares



EXCLUSÃO MÚTUA COM PTHREADS

Fernando Castor e <https://computing.lnl.gov/tutorials/pthreads/>

Obs.: boa parte já apresentada pela monitoria



Um contador errado com pthreads

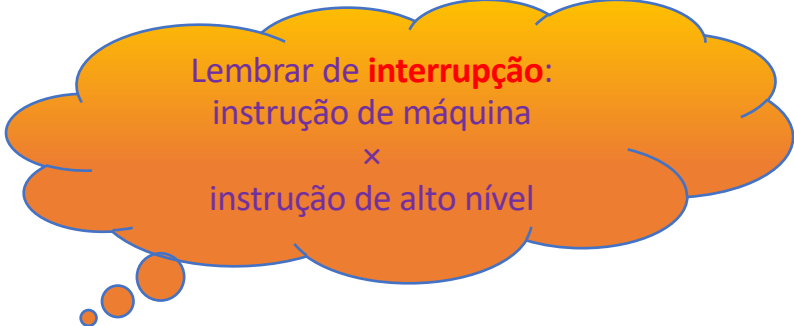
```
#include <pthread.h>
#include <stdio.h>

long contador = 0;

void *inc(void *threadid){
    int i = 0;
    for(; i < 9000000; i++) { contador++; } // condição de corrida!
}

void *dec(void *threadid){
    int i = 0;
    for(; i < 9000000; i++) { contador--; } // condição de corrida!
}

int main (int argc, char *argv[]){
    pthread_t thread1;
    pthread_t thread2;
    pthread_create(&thread1, NULL, inc, NULL);
    pthread_create(&thread2, NULL, dec, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Valor final do contador: %ld\n", contador);
    pthread_exit(NULL);
}
```



Lembrar de **interrupção:**
instrução de máquina
×
instrução de alto nível



Uma sequência típica no uso de um mutex

1. O mutex é criado e inicializado
2. Várias threads tentam se apoderar do mutex (travá-lo – *lock*)
3. Apenas uma única thread terá sucesso em travá-lo
4. A thread que está em poder do mutex executa um conjunto de operações sobre a região da memória protegida pelo respectivo mutex
5. A thread libera o mutex
6. Outra thread se apodera do mutex e repete o processo
7. Finalmente, o mutex é destruído



Exclusão mútua com pthreads

- Através do conceito de `mutex`
 - **Sincronizam** o acesso ao **estado compartilhado**
 - Independentemente do valor desse estado (diferentemente de **variáveis condicionais** – mais adiante)
 - Tipo especial de variável (**semáforo** binário)
 - Diversas funções para criar, destruir e usar *mutexes*
- Tipo de dados
 - `pthread_mutex_t`



Usando *mutexes*

```
int pthread_mutex_**lock(  
    pthread_mutex_t *mutex);
```

```
int pthread_mutex_**trylock(  
    pthread_mutex_t *mutex);
```

```
int pthread_mutex_**unlock(  
    pthread_mutex_t *mutex);
```

Um contador certo com pthreads



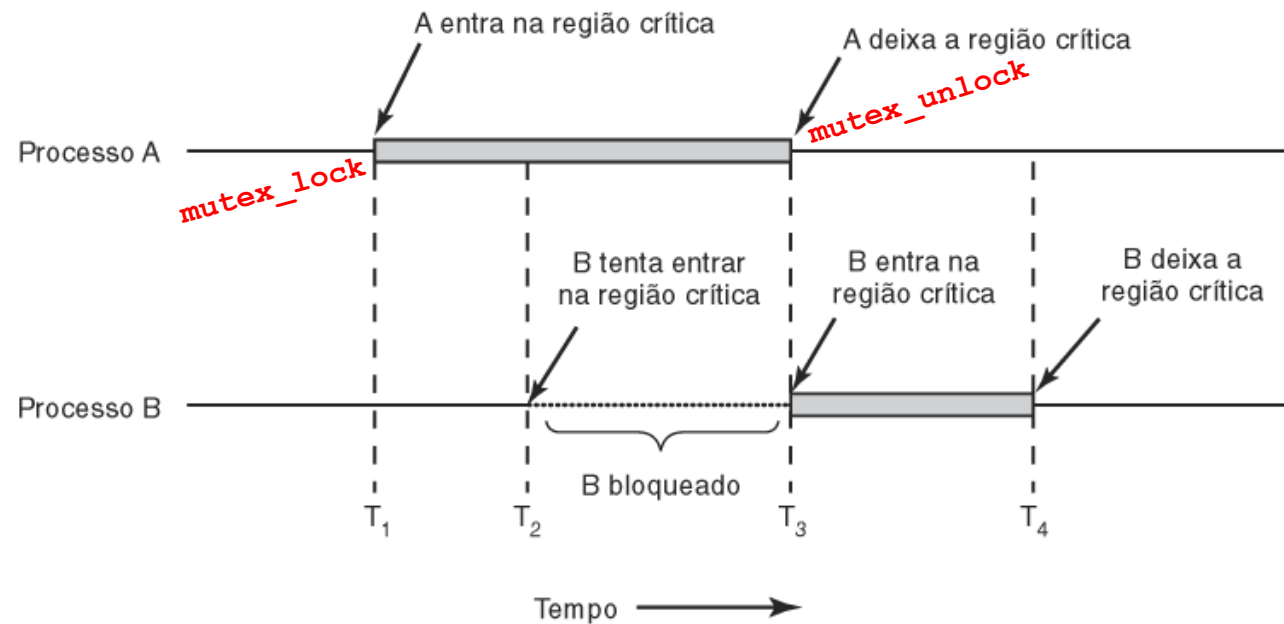
```
#include <pthread.h>
#include <stdio.h>
long contador = 0;
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
void *inc(void *threadid){
    int i = 0; for(; i < 9000000; i++) {
        pthread_mutex_lock(&mymutex);
        contador++;
        pthread_mutex_unlock(&mymutex); }
}
void *dec(void *threadid){
    int i = 0;
    for(; i < 9000000; i++) {
        pthread_mutex_lock(&mymutex);
        contador--;
        pthread_mutex_unlock(&mymutex); }
}
int main (int argc, char *argv[]){
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, inc, NULL);
    pthread_create(&thread2, NULL, dec, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Valor final do contador: %ld\n", contador);
    pthread_exit(NULL);
}
```

```
#include <pthread.h>
#include <stdio.h>
long contador = 0;
void *inc(void *threadid){
    int i = 0;
    for(; i < 9000000; i++) {
        contador++; // condição de corrida!
    }
}
void *dec(void *threadid){
    int i = 0;
    for(; i < 9000000; i++) {
        contador--; // condição de corrida!
    }
}
int main (int argc, char *argv[]){
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, inc, NULL);
    pthread_create(&thread2, NULL, dec, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Valor final do contador: %ld\n",
        contador);
    pthread_exit(NULL);
}
```



Concorrência

- **Mutexes:** garantem acesso exclusivo à uma região crítica





Vistos em aula de monitoria

Concorrência

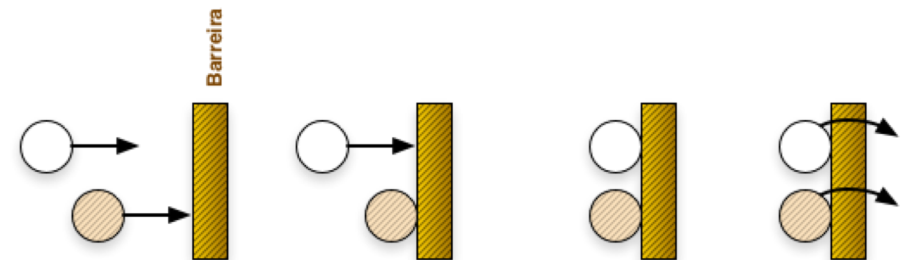
- **Mutexes:** garantem acesso exclusivo à uma região crítica
- **Variáveis Condicionais:** permitem que uma *thread* continue após determinada condição

- se consumo > produção
 - Buffer esvazia; **Consumidor não tem o que consumir**
- se consumo < produção
 - Buffer enche; **Produtor não consegue produzir mais**



Concorrência

- **Mutexes:** garantem acesso exclusivo à uma região crítica
- **Variáveis Condicionais:** permitem que uma *thread* continue após determinada condição
- **Barreira (Barrier):** Mecanismo de sincronização entre *threads*
 - threads executam até chegarem à barreira, e então, “adormecem”
 - quando todas as threads “alcançam” a barreira, elas são acordadas para continuar a execução





VARIÁVEIS CONDICIONAIS COM PTHREADS

Concorrência/Sincronização



Exclusão mútua pode não ser o suficiente

- Como visto antes, threads podem precisar cooperar
 - Uma thread só pode progredir caso outra tenha realizado uma certa ação...
 - Ou seja, se certa **condição for verdadeira**
 - A **condição** se refere aos valores dos elementos do estado compartilhado pelas threads
- Exemplo canônico: produtor-consumidor



Variáveis de condição

- **Complementam** *mutexes*
- Mecanismo de mais alto nível que semáforos
- **Evitam** a necessidade de **checar**
continuamente se a condição é verdadeira
 - Sem espera ocupada!!!



Produtor-Consumidor com espera ocupada

```
#include <stdio.h>
#include <pthread.h>

int b; /* buffer size = 1; */
int turn=0;
```

```
main() {
    pthread_t producer_thread;
    pthread_t consumer_thread;
    void *producer();
    void *consumer();
```

```
pthread_create(&consumer_thread, NULL, consumer, NULL);
```

```
pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_join(consumer_thread, NULL);
}
```

```
void put(int i){
    b = i;
}
```

```
int get(){
    return b ;
}
```

```
void *producer() {
    int i = 0;
    printf("Produtor\n");
    while (1) {
        while (turn == 1) ;
        put(i);
        turn = 1;
        i = i + 1;
    }
    pthread_exit(NULL);
}
```

```
void *consumer() {
    int i,v;
    printf("Consumidor\n");
    for (i=0;i<100;i++) {
        while (turn == 0) ;
        v = get();
        turn = 0;
        printf("Peguei %d \n",v);
    }
    pthread_exit(NULL);
}
```

Espera ocupada:
-Desperdiça recursos
-(neste caso) produção e consumo alternados



Usando variáveis de condição com pthreads

- Esperar que certa condição se torne verdadeira
 - `pthread_cond_wait(cond_var, mutex);`
- Avisar/sinalizar outra(s) thread(s) que a condição se tornou verdadeira
 - `pthread_cond_signal(cond_var);`
 - `pthread_cond_broadcast(cond_var)`



Produtor-Consumidor variáveis de condição (1/3)

```
#include <stdio.h>
#include <pthread.h>
#define BUFFER_SIZE 10
#define NUM_ITEMS 200

int buff[BUFFER_SIZE]; /* buffer size = 1; */
int items = 0; /* number of items in the buffer.
int first = 0;
int last = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t buffer_cond = PTHREAD_COND_INITIALIZER;

main() {
    pthread_t consumer_thread;
    pthread_t producer_thread;
    void *producer();
    void *consumer();
    pthread_create(&consumer_thread, NULL, consumer, NULL);
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);
}
```

Produtor-Consumidor variáveis de condição (2/3)

```
void put(int i){
    pthread_mutex_lock(&mutex);
    if (items == BUFFER_SIZE) {
        pthread_cond_wait(&buffer_cond, &mutex);
    }
    buff[last] = i;
    printf("pos %d: ", last);
    items++; last++;
    if(last==BUFFER_SIZE) { last = 0; }
    if(items == 1) {
        pthread_cond_broadcast(&buffer_cond);
    }
    pthread_mutex_unlock(&mutex);
}
```

```
void *producer() {
    int i = 0;
    printf("Produtor\n");
    for(i=0;i<NUM_ITEMS; i++) {
        put(i);
        printf("Produzi %d \n",i);
    }
    pthread_exit(NULL);
}
```



Produtor-Consumidor variáveis de condição (3/3)

```
int get(){
    int result;
    pthread_mutex_lock(&mutex);
    if (items == 0) {
        pthread_cond_wait(&buffer_cond, &mutex);
    }
    result = buff[first];
    printf("pos %d: ", first);
    items--; first++;
    if(first==BUFFER_SIZE) { first = 0; }
    if(items == BUFFER_SIZE - 1){
        pthread_cond_broadcast(&buffer_cond);
    }
    pthread_mutex_unlock(&mutex);
    return result;
}

void *consumer() {
    int i,v;
    printf("Consumidor\n");
    for (i=0;i<NUM_ITEMS;i++) {
        v = get();
        printf("Consumi %d \n",v);
    }
    pthread_exit(NULL);
}
```





DEADLOCK

Um problema de concorrência



Problemas com Concorrência

- Não-determinismo
- Dependência de Velocidade
- *Starvation*
- *Deadlock*



Recursos (I)

- Exemplos de recursos:
 - Impressoras
 - Tabelas
- Processos precisam de acesso aos recursos numa ordem racional
- Suponha que um processo detenha o recurso A e solicite o recurso B
 - ao mesmo tempo um outro processo detém B e solicita A
 - **ambos são bloqueados e assim permanecem**



Recursos (2)

- *Deadlocks* ocorrem quando ...
 - garante-se aos processos acesso exclusivo aos dispositivos
 - esses dispositivos são normalmente chamados de **recursos**
- **Recursos preemptíveis**
 - podem ser retirados de um processo sem quaisquer efeitos prejudiciais
- **Recursos não preemptíveis**
 - vão induzir o processo a falhar se forem retirados



Recursos (3)

- Sequência de eventos necessários ao uso de um recurso
 1. solicitar o recurso
 2. usar o recurso
 3. liberar o recurso
- Deve esperar se solicitação é negada
 - processo solicitante pode ser bloqueado
 - pode falhar resultando em um código de erro



Introdução aos Deadlocks

- Definição formal:
Um conjunto de processos está em situação de deadlock se todo processo pertencente ao conjunto estiver esperando por um evento que somente um outro processo desse mesmo conjunto poderá fazer acontecer
- Normalmente o evento é a liberação de um recurso atualmente retido
- Nenhum dos processos pode...
 - executar
 - liberar recursos
 - ser acordado



Semáforo: Deadlock (e Livelock?)



(a) Um **deadlock** potencial.

(b) Um **deadlock** real.

Assim como em **deadlock**, *threads* em **livelock** não podem progredir em sua execução. No entanto, as threads não ficam bloqueadas - elas ficam simplesmente ocupadas (vs bloqueadas – *deadlock*) respondendo umas às outras sem progresso

A principal diferença entre **livelock** e **deadlock** é que as threads não serão bloqueadas. Em vez disso, elas tentarão responder umas às outras continuamente



Quatro Condições para Deadlock

1. Condição de exclusão mútua
 - todo recurso está ou associado a um processo ou disponível
2. Condição de posse e espera
 - processos que retêm recursos podem solicitar novos recursos
3. Condição de não preempção
 - recursos concedidos previamente não podem ser forçosamente tomados
4. Condição de espera circular
 - deve ser uma cadeia circular de 2 ou mais processos
 - cada um está à espera de recurso retido pelo membro seguinte dessa cadeia



Modelagem de Deadlock (I)

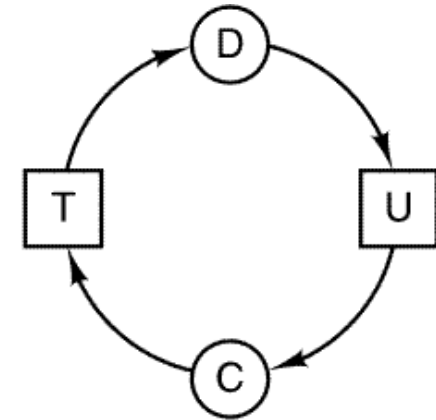
- Modelado com grafos dirigidos



(a)



(b)



(c)

- recurso R alocado ao processo A
- processo B está solicitando/esperando pelo recurso S
- processos C e D estão em deadlock sobre recursos T e U

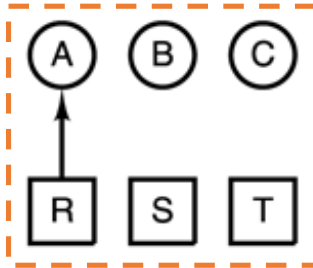


Modelagem de Deadlock (2)

- Estratégias para tratar Deadlocks
 - ignorar por completo o problema
 - detecção e recuperação
 - **evitação dinâmica**
 - alocação cuidadosa de recursos
 - prevenção
 - negação de uma das quatro condições necessárias

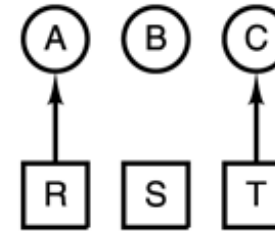
Modelagem de *Deadlock* (3)

1. A requisita R
 2. C requisita T
 3. A requisita S
 4. C requisita R
 5. A libera R
 6. A libera S
- nenhum deadlock

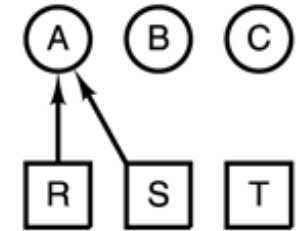


(k)

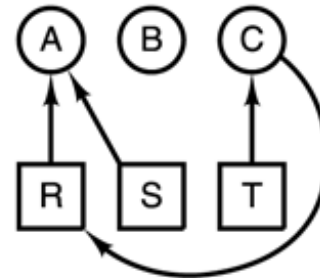
(l)



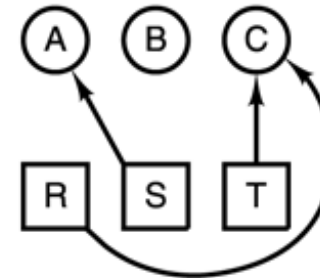
(m)



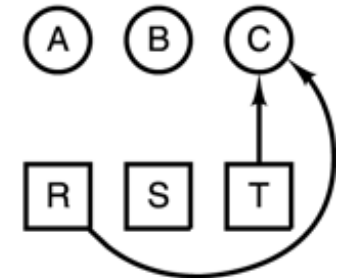
(n)



(o)



(p)



(q)

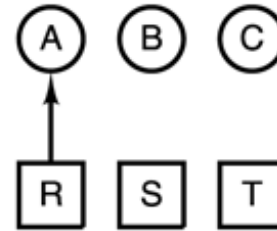
Como pode ser **evitado** um *deadlock*



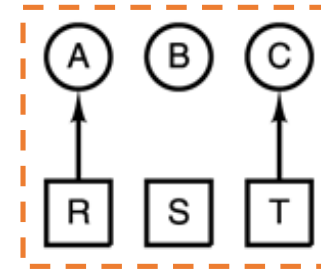
Modelagem de *Deadlock* (3)

1. A requisita R
 2. C requisita T
 3. A requisita S
 4. C requisita R
 5. A libera R
 6. A libera S
- nenhum deadlock

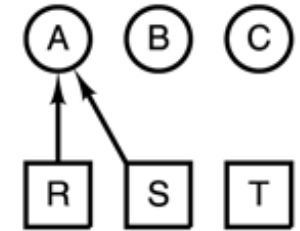
(k)



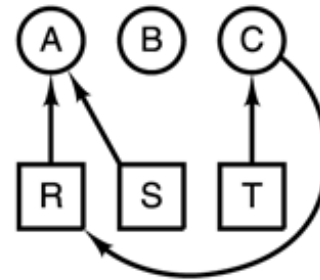
(l)



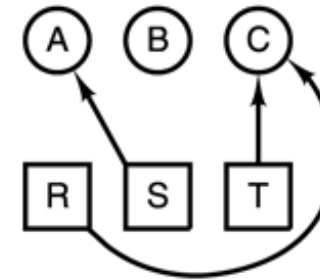
(m)



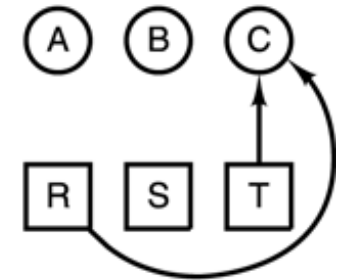
(n)



(o)



(p)



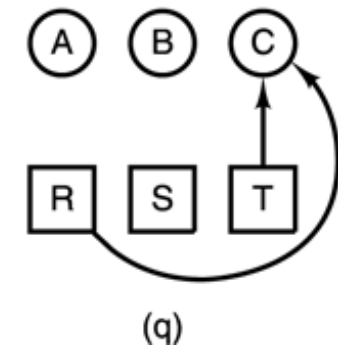
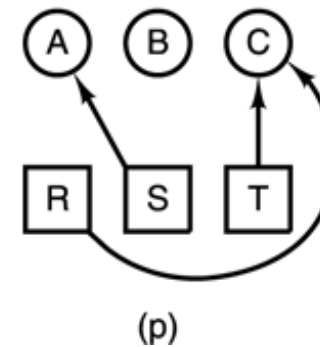
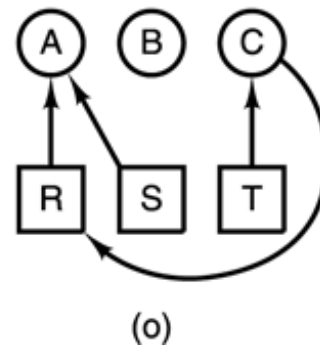
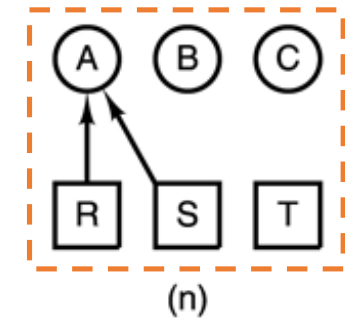
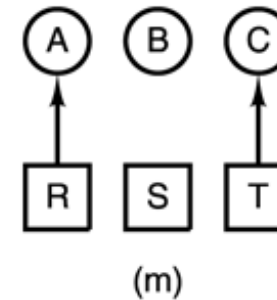
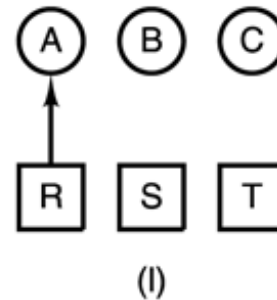
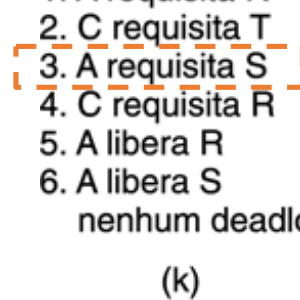
(q)

Como pode ser **evitado** um *deadlock*



Modelagem de *Deadlock* (3)

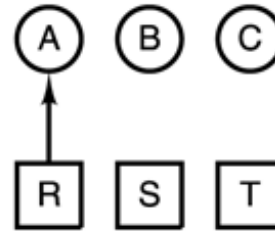
1. A requisita R
 2. C requisita T
 3. A requisita S
 4. C requisita R
 5. A libera R
 6. A libera S
- nenhum deadlock



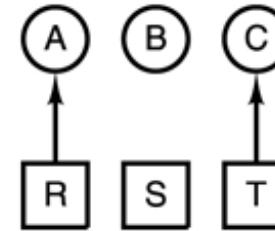
Como pode ser **evitado** um *deadlock*

Modelagem de *Deadlock* (3)

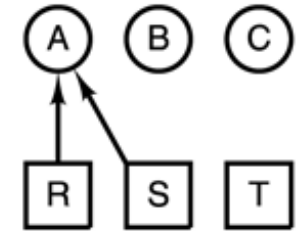
1. A requisita R
 2. C requisita T
 3. A requisita S
 4. C requisita R
 5. A libera R
 6. A libera S
- nenhum deadlock



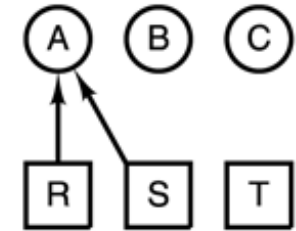
(k)



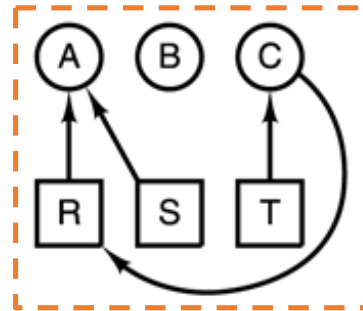
(l)



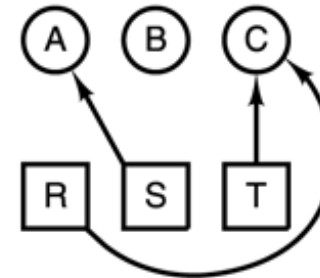
(m)



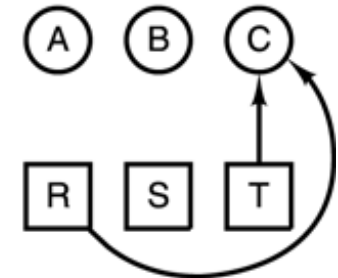
(n)



(o)



(p)

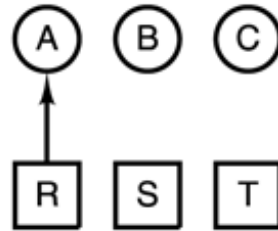


(q)

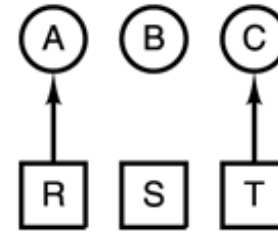
Como pode ser **evitado** um *deadlock*

Modelagem de *Deadlock* (3)

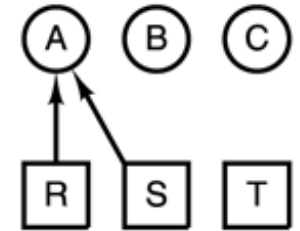
1. A requisita R
 2. C requisita T
 3. A requisita S
 4. C requisita R
 5. A libera R
 6. A libera S
- nenhum deadlock



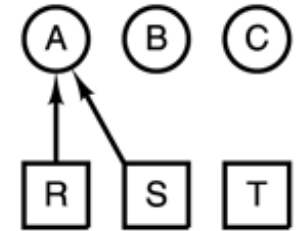
(k)



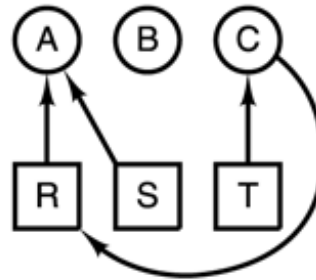
(l)



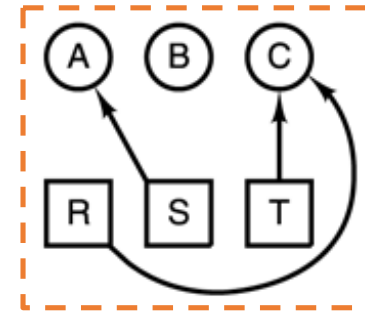
(m)



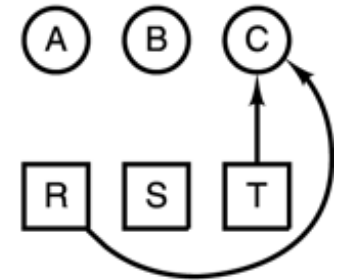
(n)



(o)



(p)



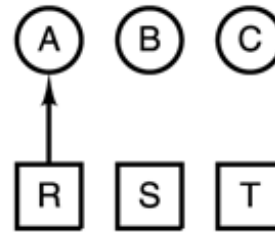
(q)

Como pode ser **evitado** um *deadlock*

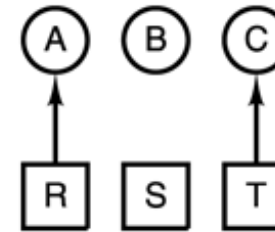


Modelagem de *Deadlock* (3)

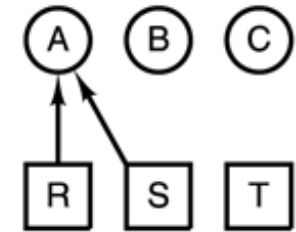
1. A requisita R
 2. C requisita T
 3. A requisita S
 4. C requisita R
 5. A libera R
 6. A libera S
- nenhum deadlock



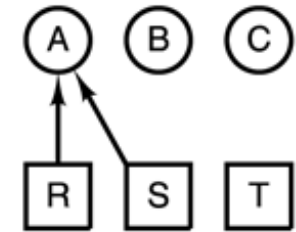
(k)



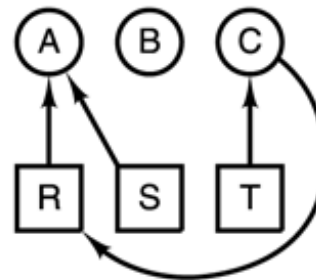
(l)



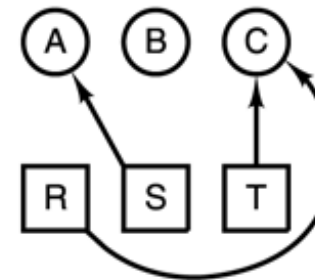
(m)



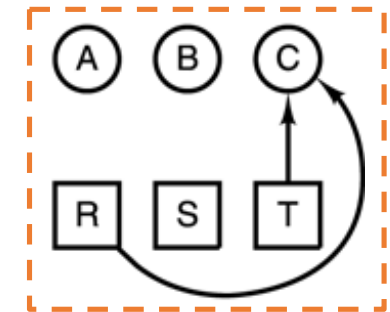
(n)



(o)



(p)



(q)

Como pode ser **evitado** um *deadlock*

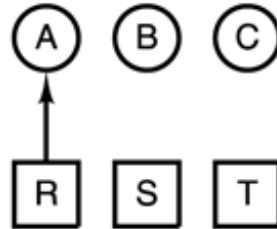


Modelagem de *Deadlock* (3)

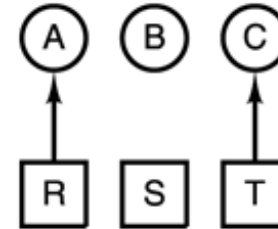
1. A requisita R
2. C requisita T
3. A requisita S
4. C requisita R
5. A libera R
6. A libera S

nenhum deadlock

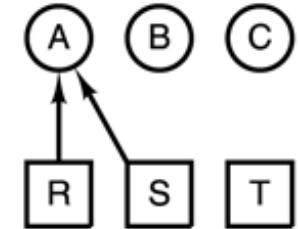
(k)



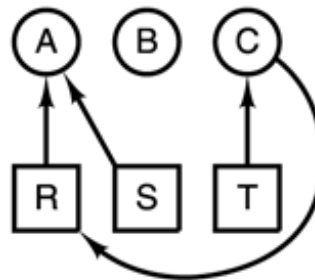
(l)



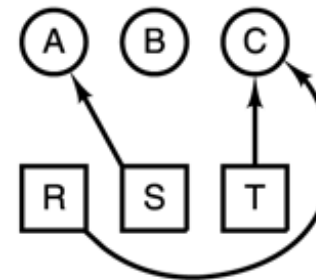
(m)



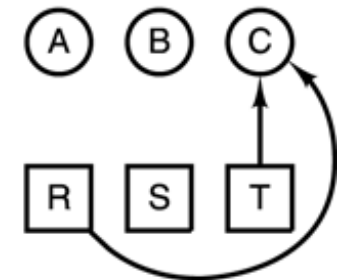
(n)



(o)



(p)



(q)

Como pode ser **evitado** um *deadlock*



Estratégias para tratar Deadlocks

- ignorar por completo o problema
- detecção e recuperação
- evitação dinâmica
 - alocação cuidadosa de recursos
- prevenção
 - negação de uma das quatro condições necessárias



“Algoritmo” do Avestruz

- Finge que o problema não existe
- Razoável se
 - *deadlocks* ocorrem muito raramente
 - custo da prevenção é alto
- UNIX e Windows seguem esta abordagem
- É uma ponderação entre
 - conveniência
 - correção

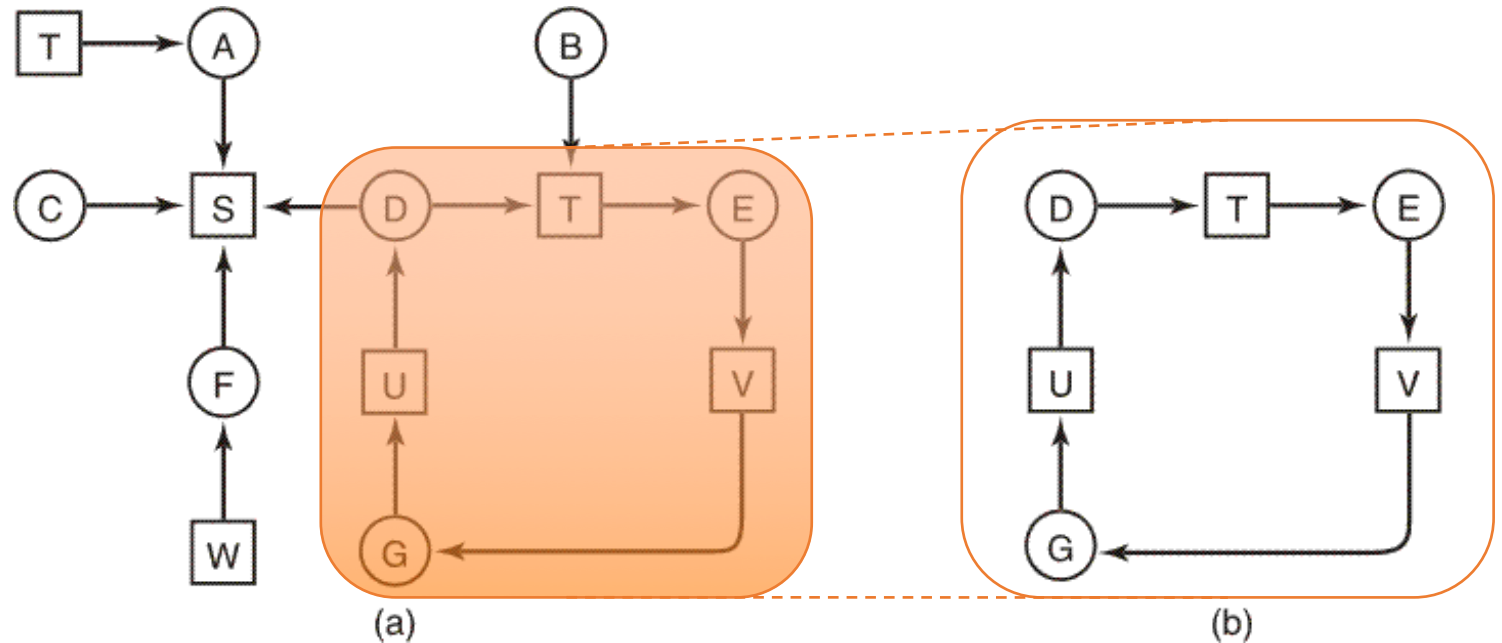


Estratégias para tratar Deadlocks

- ignorar por completo o problema
- **detecção e recuperação**
- evitação dinâmica
 - alocação cuidadosa de recursos
- prevenção
 - negação de uma das quatro condições necessárias



Detecção com um Recurso de Cada Tipo



- Observe a posse e solicitações de recursos
- Um **ciclo** pode ser encontrado dentro do grafo, denotando deadlock





Recuperação de Deadlock

- Recuperação através de **preempção**
 - retirar um recurso de algum outro processo
 - depende da natureza do recurso (preemptível?)
- Recuperação através de **reversão de estado**
 - verifica um processo periodicamente
 - reinicia o processo se este é encontrado em estado de deadlock (usa o último estado correto salvo)
- Recuperação através da **eliminação de processos**
 - forma mais grosseira, mas também mais simples de quebrar um deadlock
 - elimina um dos processos no ciclo de deadlock
 - os outros processos conseguem seus recursos
 - escolhe processo que pode ser reexecutado desde seu início



Estratégias para tratar Deadlocks

- ignorar por completo o problema
- detecção e recuperação
- evitação dinâmica
 - alocação cuidadosa de recursos
- **prevenção**
 - negação de uma das quatro condições necessárias

 **Quatro Condições para *Deadlock*** 

1. **Condição de exclusão mútua**
 - todo recurso está ou associado a um processo ou disponível
2. **Condição de posse e espera**
 - processos que retêm recursos podem solicitar novos recursos
3. **Condição de não preempção**
 - recursos concedidos previamente não podem ser forçosamente tomados
4. **Condição de espera circular**
 - deve ser uma cadeia circular de 2 ou mais processos
 - cada um está à espera de recurso retido pelo membro seguinte dessa cadeia

Pearson Education Sistemas Operacionais Modernos – 2ª Edição 6



Prevenção de Deadlock

Atacando a Condição de Exclusão Mútua

- **Princípio:**
 - evitar alocar um recurso quando ele não for absolutamente necessário
 - tentar assegurar que o menor número possível de processos possa de fato requisitar o recurso
- Alguns dispositivos (como uma impressora) podem fazer uso de *spool* [“fila”]
 - o *daemon* de impressão é o único que usa o recurso impressora
 - desta forma, *deadlock* envolvendo a impressora é eliminado



Prevenção de Deadlock Atacando a Condição de Posse e Espera

- Exigir que todos os processos requisitem os recursos antes de iniciarem
 - um processo nunca tem que esperar por aquilo que precisa
- Problemas
 - podem não saber quantos e quais recursos vão precisar no início da execução
 - e também retêm recursos que outros processos poderiam estar usando
- Variação:
 - processo deve desistir de todos os recursos
 - para então requisitar todos os que são **imediatamente** necessários



Prevenção de Deadlock

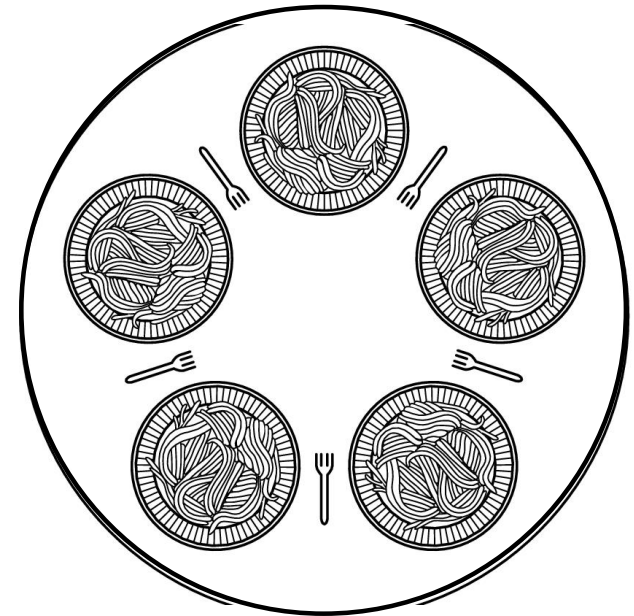
- Resumo das abordagens para prevenir deadlock

Condição	Abordagem contra deadlocks
Exclusão mútua	Usar spool em tudo
Posse-e-espera	Requisitar inicialmente todos os recursos necessários
Não preempção	Retomar os recursos alocados
Espera circular	Ordenar numericamente os recursos



O clássico problema dos ‘Filósofos Jantantes’

- Filósofos comem ou pensam
- Comem espaguete com muito molho e precisam de 2 garfos
- Pegam um garfo de cada vez
- Como prevenir *deadlock*?



```

#define N          5          /* number of philosophers */
#define LEFT      (i+N-1)%N  /* number of i's left neighbor */
#define RIGHT     (i+1)%N    /* number of i's right neighbor */
#define THINKING  0          /* philosopher is thinking */
#define HUNGRY    1          /* philosopher is trying to get forks */
#define EATING    2          /* philosopher is eating */
typedef int semaphore;      /* semaphores are a special kind of int */
int state[N];              /* array to keep track of everyone's state */
semaphore mutex = 1;       /* mutual exclusion for critical regions */
semaphore s[N];           /* one semaphore per philosopher */

void philosopher(int i)    /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {        /* repeat forever */
        think();          /* philosopher is thinking */
        take_forks(i);    /* acquire two forks or block */
        eat();            /* yum-yum, spaghetti */
        put_forks(i);     /* put both forks back on table */
    }
}

```

Solução (parte I)



```

void take_forks(int i)                                /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks(i)                                    /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(i)                                          /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

```

```

/* enter critical region */
/* record fact that philosopher i is hungry */
/* try to acquire 2 forks */
/* exit critical region */
/* block if forks were not acquired */

```

```

/* enter critical region */
/* philosopher has finished eating */
/* see if left neighbor can now eat */
/* see if right neighbor can now eat */
/* exit critical region */

```

Solução (parte 2)

