INFRA-ESTRUTURA DE SOFTWARE

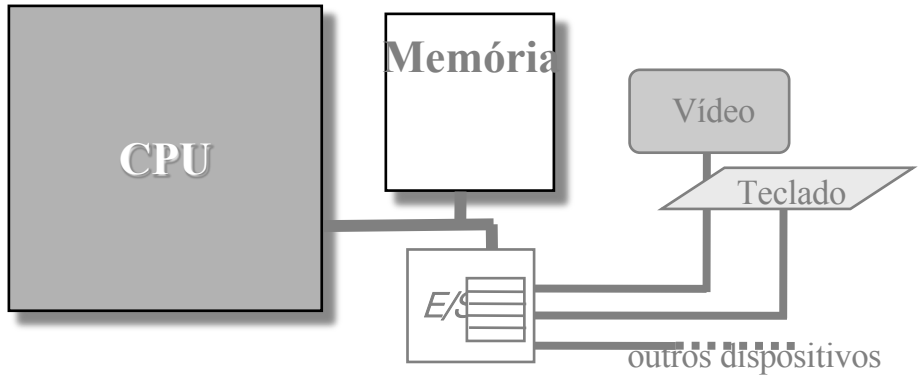
Gerência de Processos

Processo

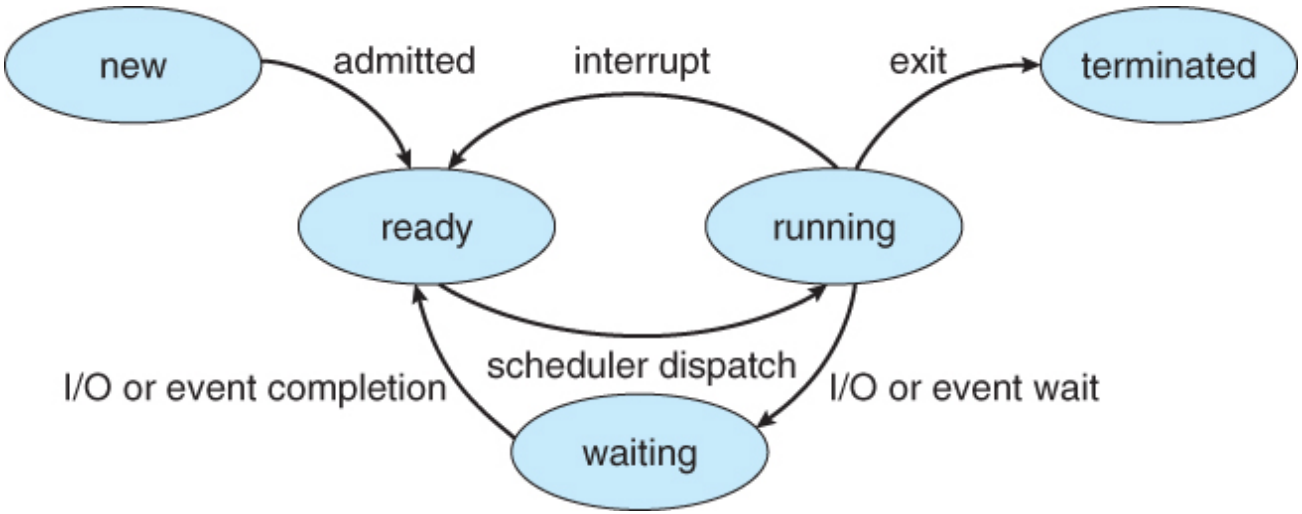
Um programa em execução

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



Contexto

ID do Processo
Estado
Prioridade
Program Counter
Ponteiros da Memória
Contexto (outros regs.)
I/O Status
Informações gerais • *tempo de CPU* • *limites, usuário, etc.*

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

ID do Processo
Estado
Prioridade
Program Counter
Ponteiros da Memória
Contexto (outros regs.)
I/O Status
Informações gerais • *tempo de CPU* • *limites, usuário, etc.*

Ciclo de vida de um processo...

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

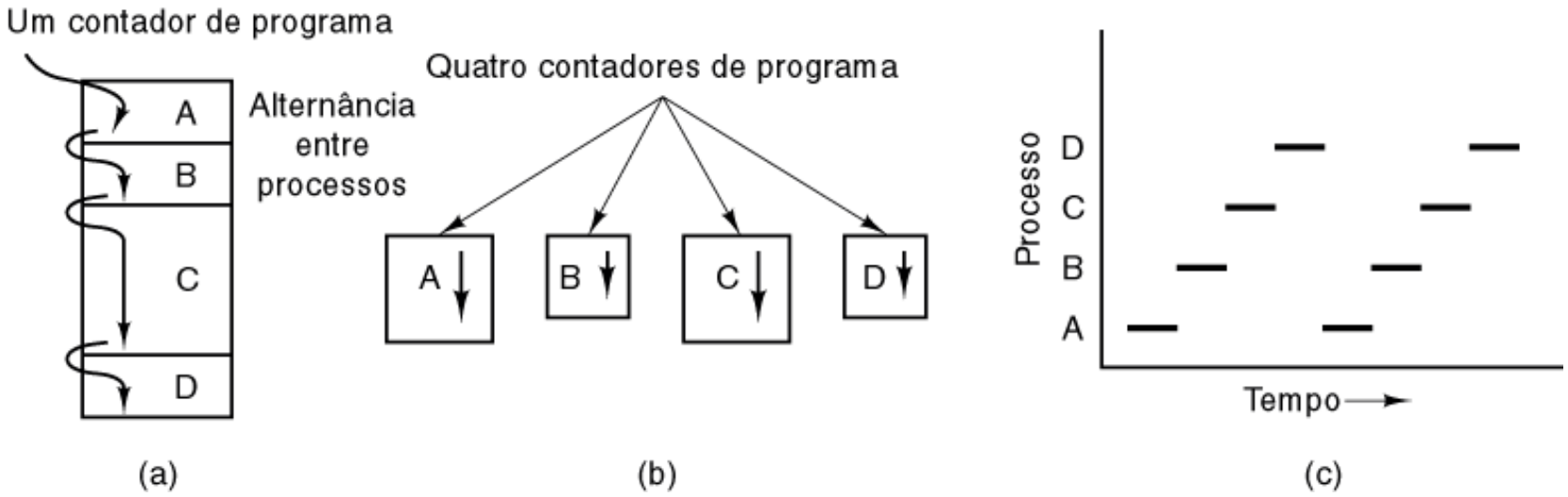
Criação de Processos

- 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 proceso
  - 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 ⇨ escalonamento

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 escalonador…
- 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 *threads*?

Algumas características e conceitos associados com processos

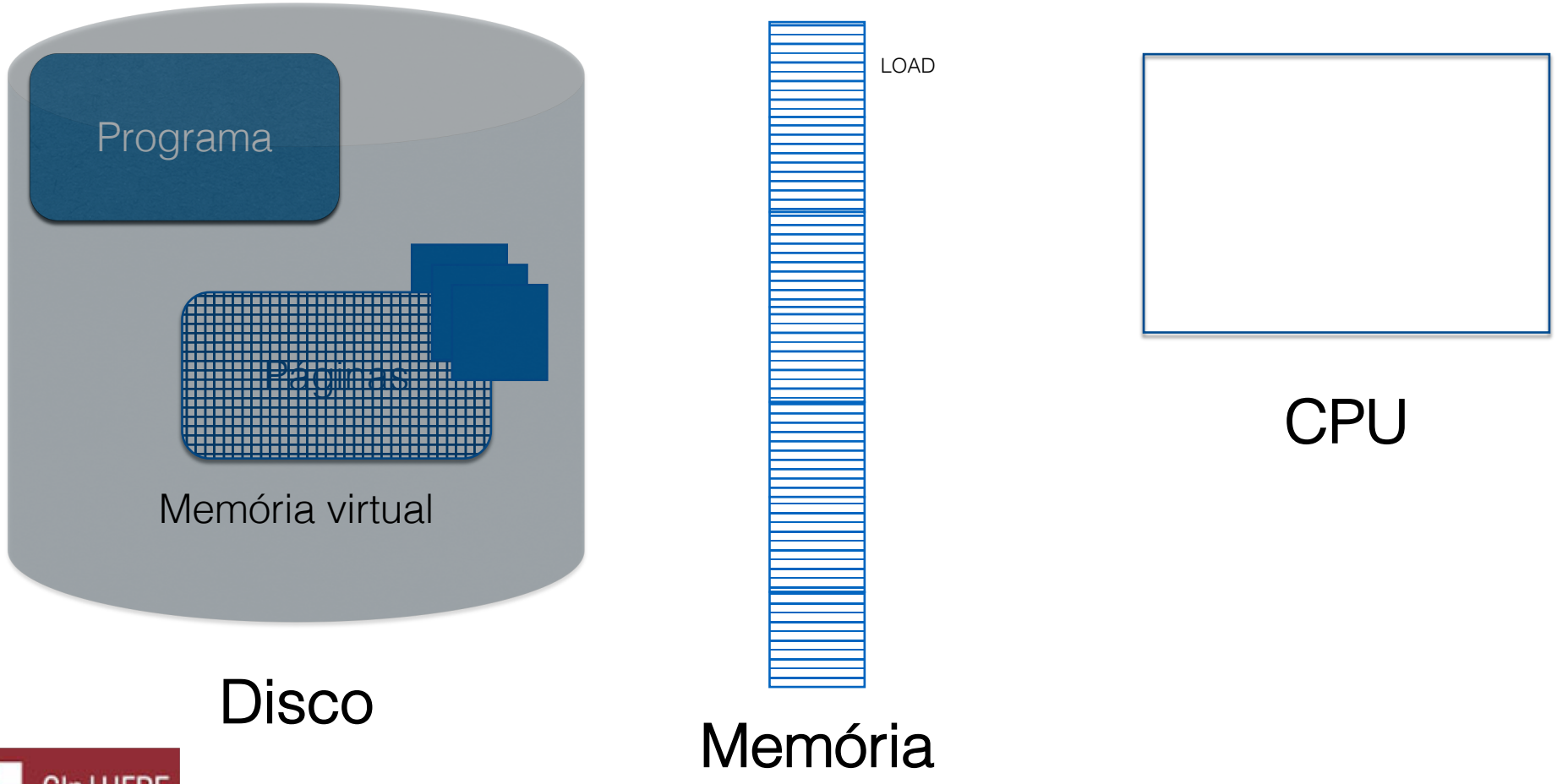
- **PID** - a identidade de um processo representado por um número inteiro e único
- **UID** - associação com um usuário que inicia um processo
- **Parent Process** - primeiro processo inicializado no kernel do sistema é o init. Este processo tem o PID 1 e é o pai de todos os outros processos no sistema
- **Parent Process ID** - o PID do processo pai, ou seja, o PID do processo que criou o processo em questão
- **Lifetime** - o tempo de vida de um processo em execução
- **Current Working Directory** - um diretório associado com cada processo
- **Enviroment** - cada processo tem suporte a uma lista de variáveis de ambiente

Algumas características e conceitos associados com processos

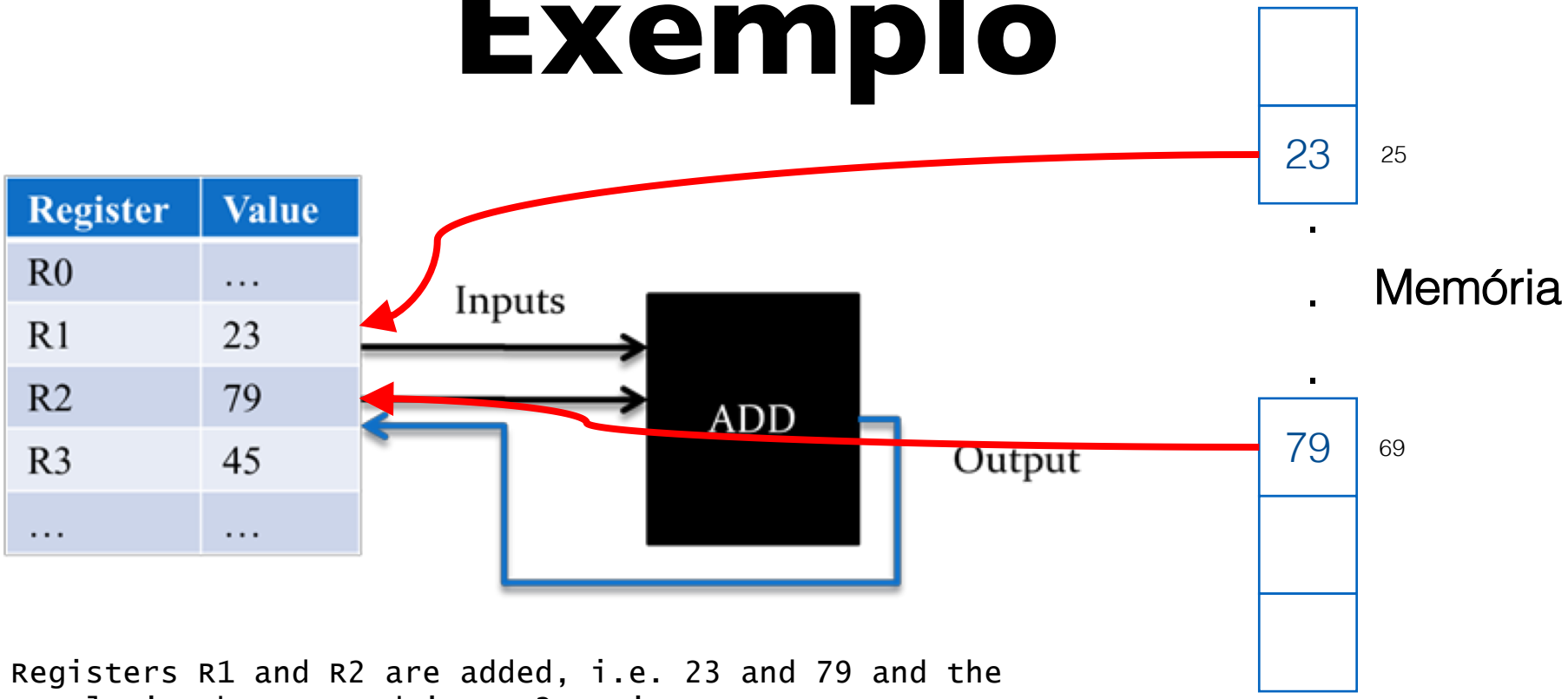
- **Enviroment -** cada processo tem suporte a uma lista de variáveis de ambiente

```
Carloss-MacBook-Pro:~ cagf$ ps e -ww -p 39566
  PID   TT  STAT      TIME COMMAND
39566 s000  S      0:00.06 -bash
TMPDIR=/var/folders/vz/5v_q_hb52ddbyfv6g5z5j3740000gn/T/
XPC_FLAGS=0x0
Apple_PubSub_Socket_Render=/private/tmp/com.apple.launchd.KWfqoouHTx/Render
TERM_PROGRAM_VERSION=400
LC_CTYPE=UTF-8
TERM_PROGRAM=Apple_Terminal
XPC_SERVICE_NAME=0
TERM_SESSION_ID=9A1B8CF7-2C53-416D-BFD8-69A60BABC3C8
SSH_AUTH_SOCK=/private/tmp/com.apple.launchd.9ROE0OakQX/Listeners
TERM=xterm-256color
SHELL=/bin/bash
HOME=/Users/cagf
LOGNAME=cagf
USER=cagf
PATH=/usr/bin:/bin
```

O elo Processador-Memória-Disco



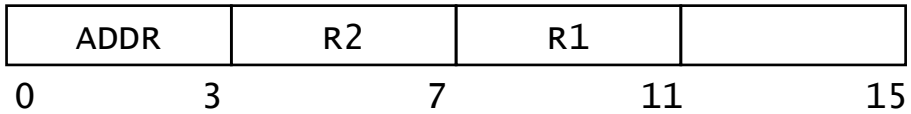
Exemplo



Registers R1 and R2 are added, i.e. 23 and 79 and the result is then stored into R2 register

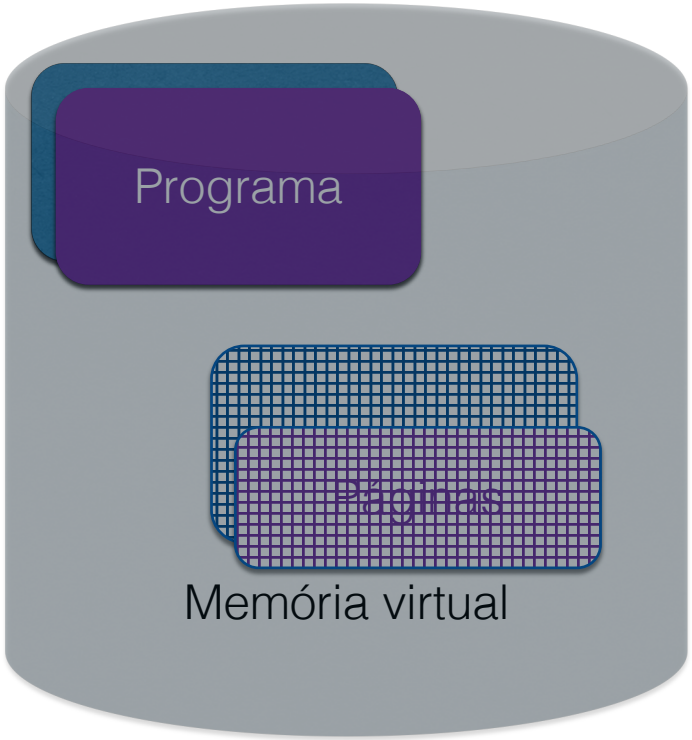
```
LOADM R1  M25 ; load from memory M25 and store the value in register R1
LOADM R2  M69 ; load from memory M69 and store the value in register R2
ADDR  R2  R1  ; add values of register R1 and R2
STORER R2 M2  ; store the value of register R2 to memory M2
```

```
Assembly Code: ADDR   R2 R1
Machine Code:  0101 0010 0001 0000
```



CIn | UFPE

Processos Concorrentes



Disco

Memória

CPU

Interrupção & Escalonamento

CIn | UFPE

Interrupção

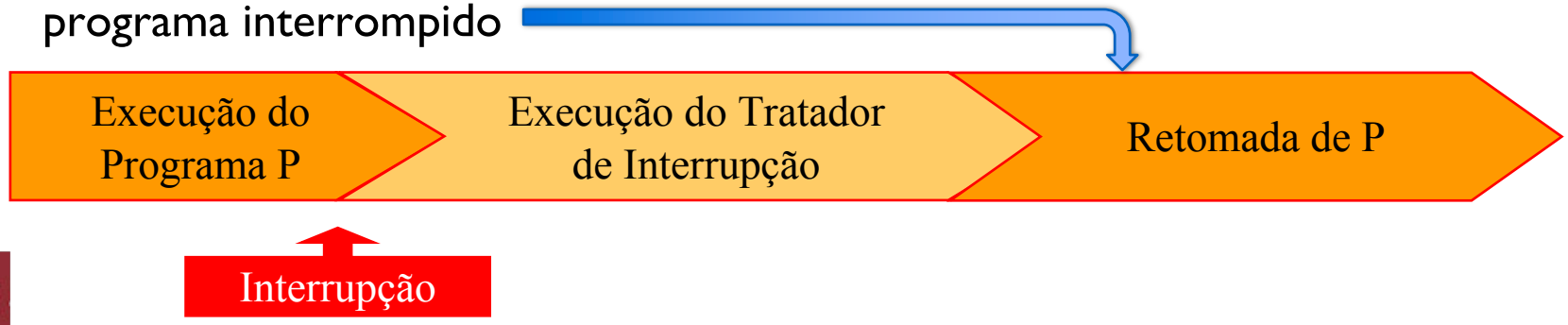
O elo Hardware-Software

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

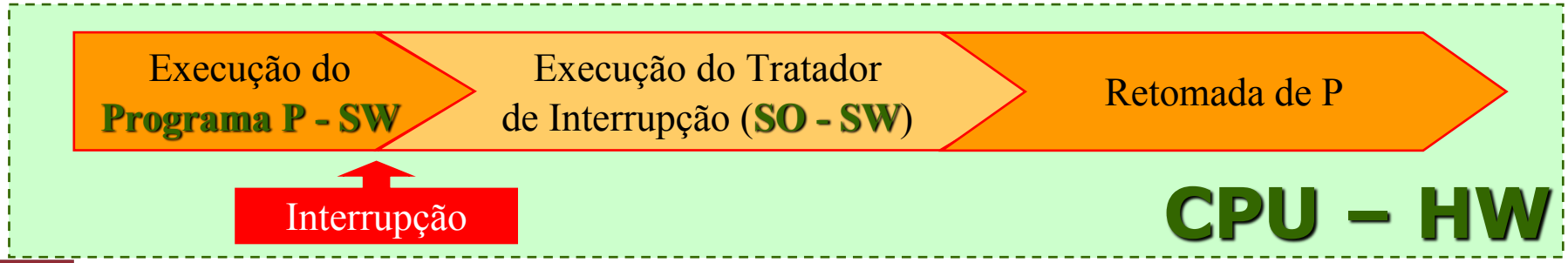
Interrupções de Hardware

- 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 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

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

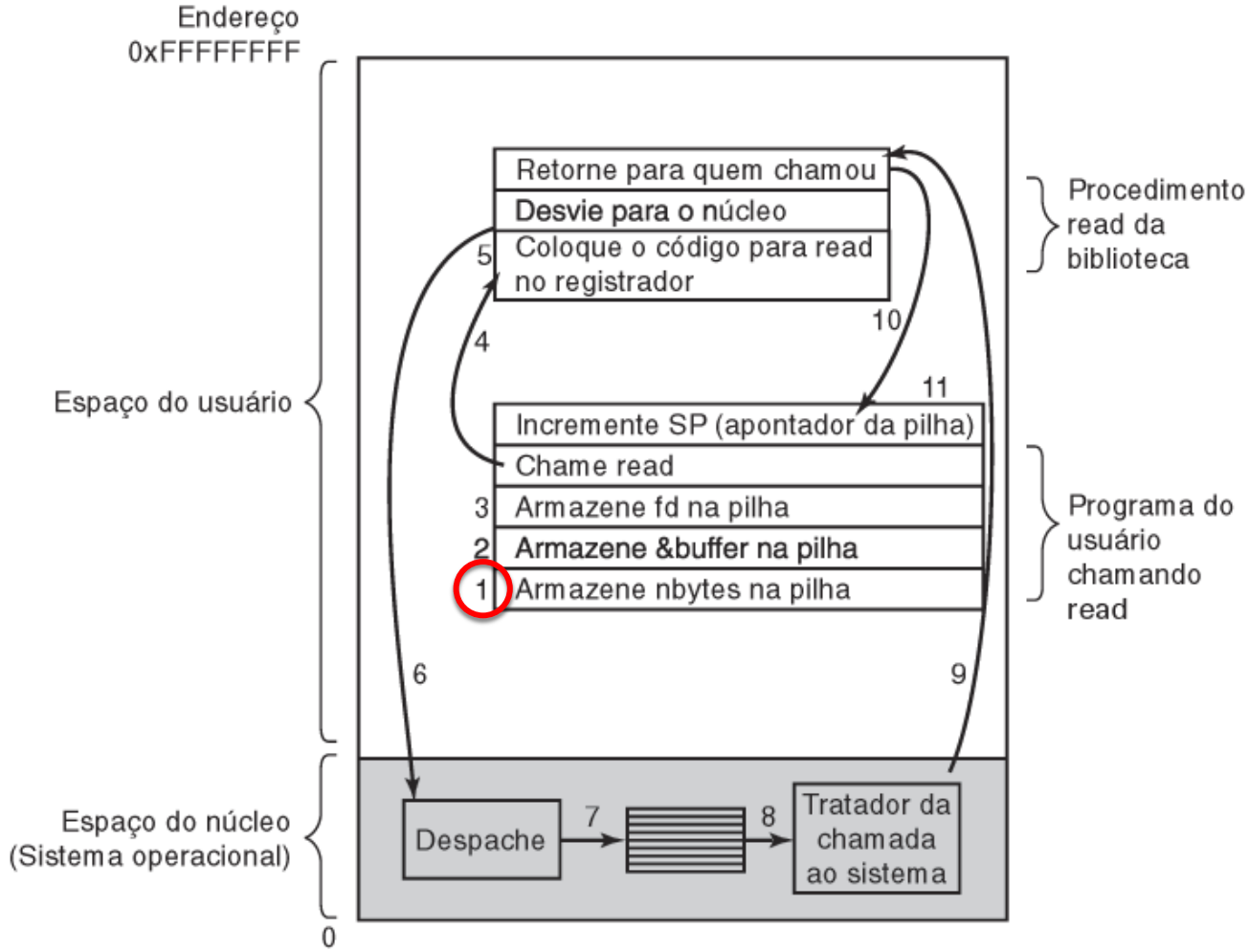
- Processo: um programa em execução
- Contexto de processo: informações para gerenciamento
- Página: parte de um programa capaz de caber na memória
- Memória virtual: espaço de armazenamento de páginas em disco
- Espaço de endereçamento e proteção
- Escalonamento: quando um ou mais processos estão prontos 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 intrução de programa (ex. READ)
    - situações em que o programa não teria como prosseguir (ex. *overflow* em operações aritméticas)
- Chamadas ao sistema formam a interface entre o SO e os programas de usuário

Chamadas ao Sistema

System Calls

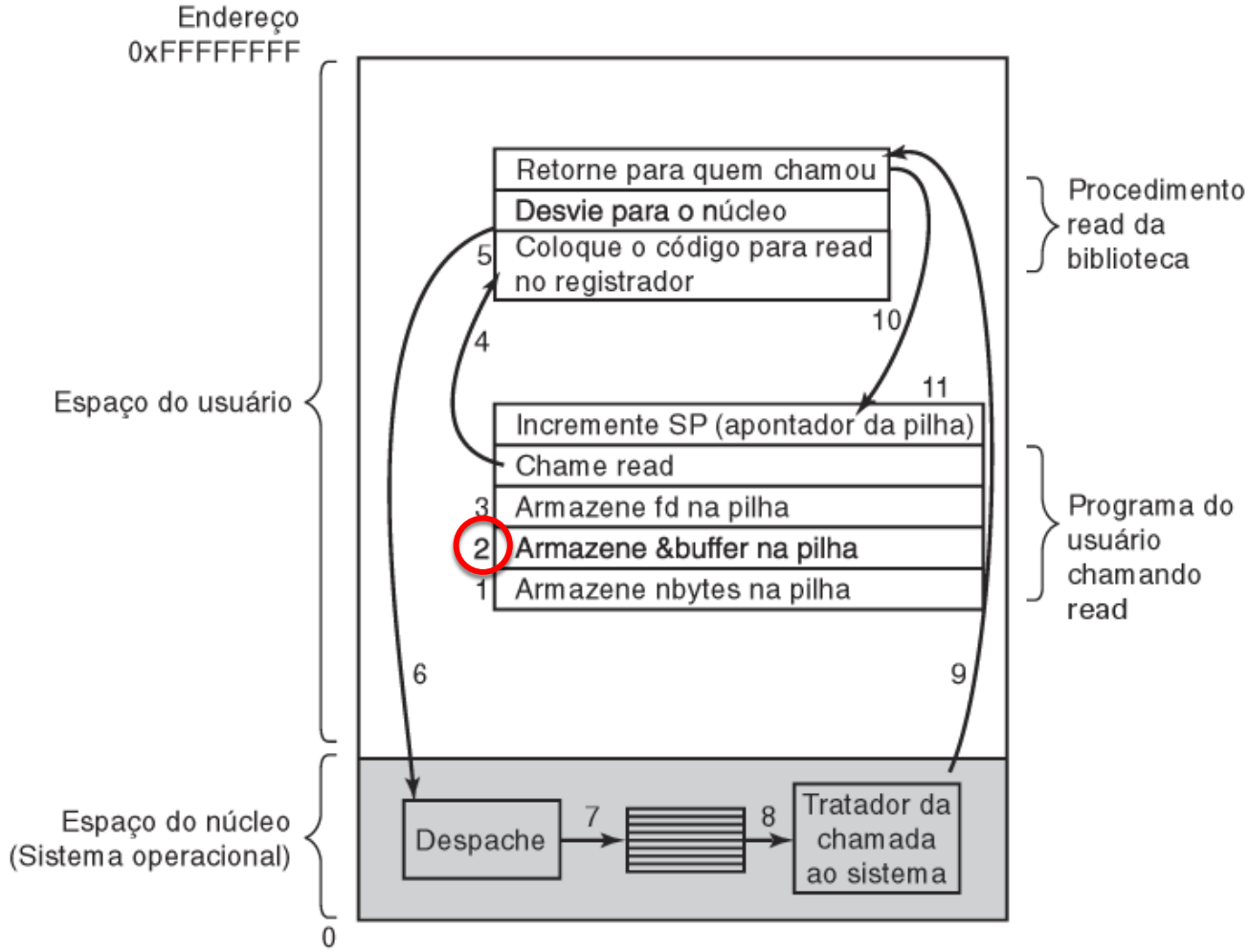
Os Passos de uma Chamada ao Sistema



Os 11 passos para fazer uma chamada ao Sistema

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

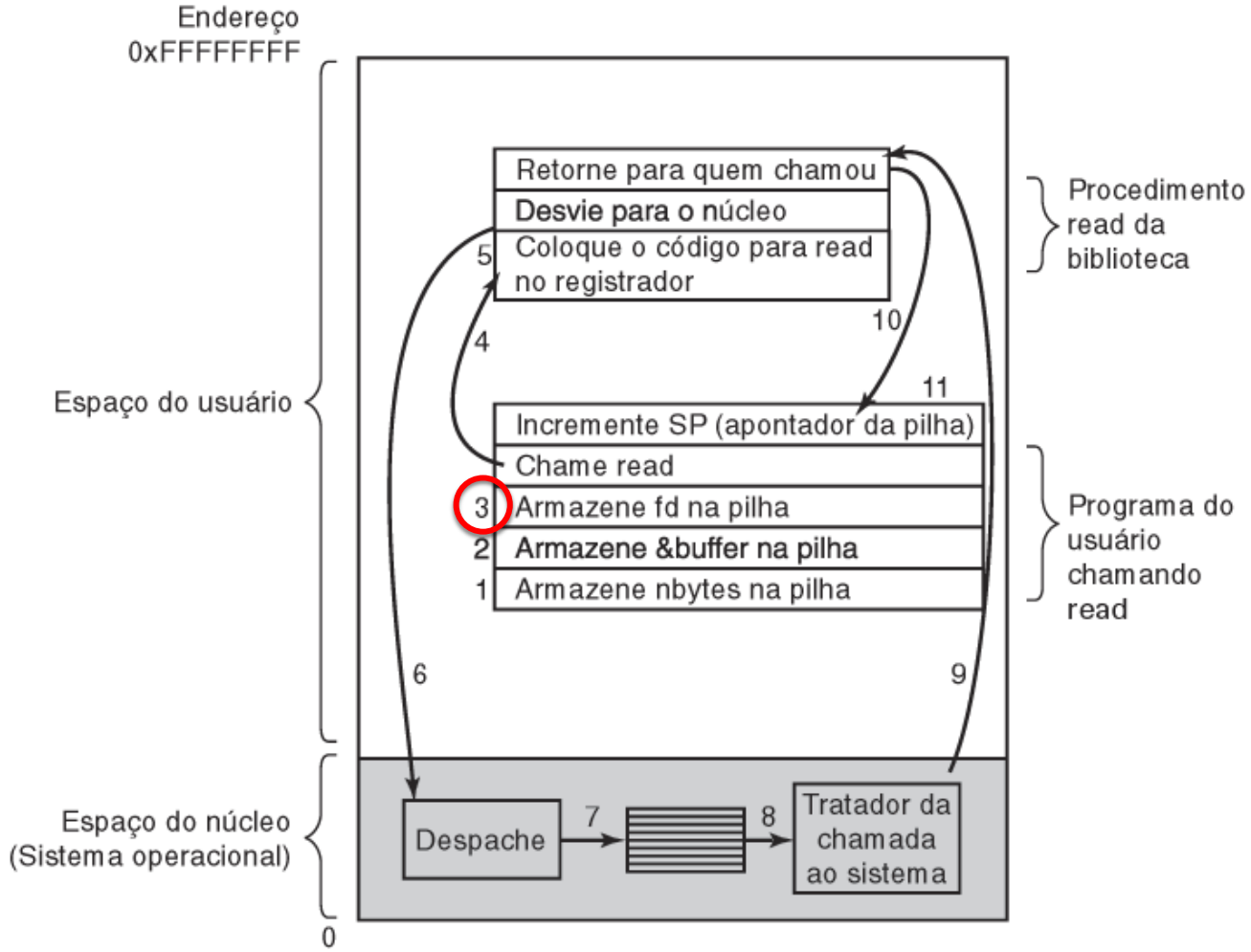
Os Passos de uma Chamada ao Sistema



Os 11 passos para fazer uma chamada ao Sistema

Ex. read (fd, buffer, nbytes)

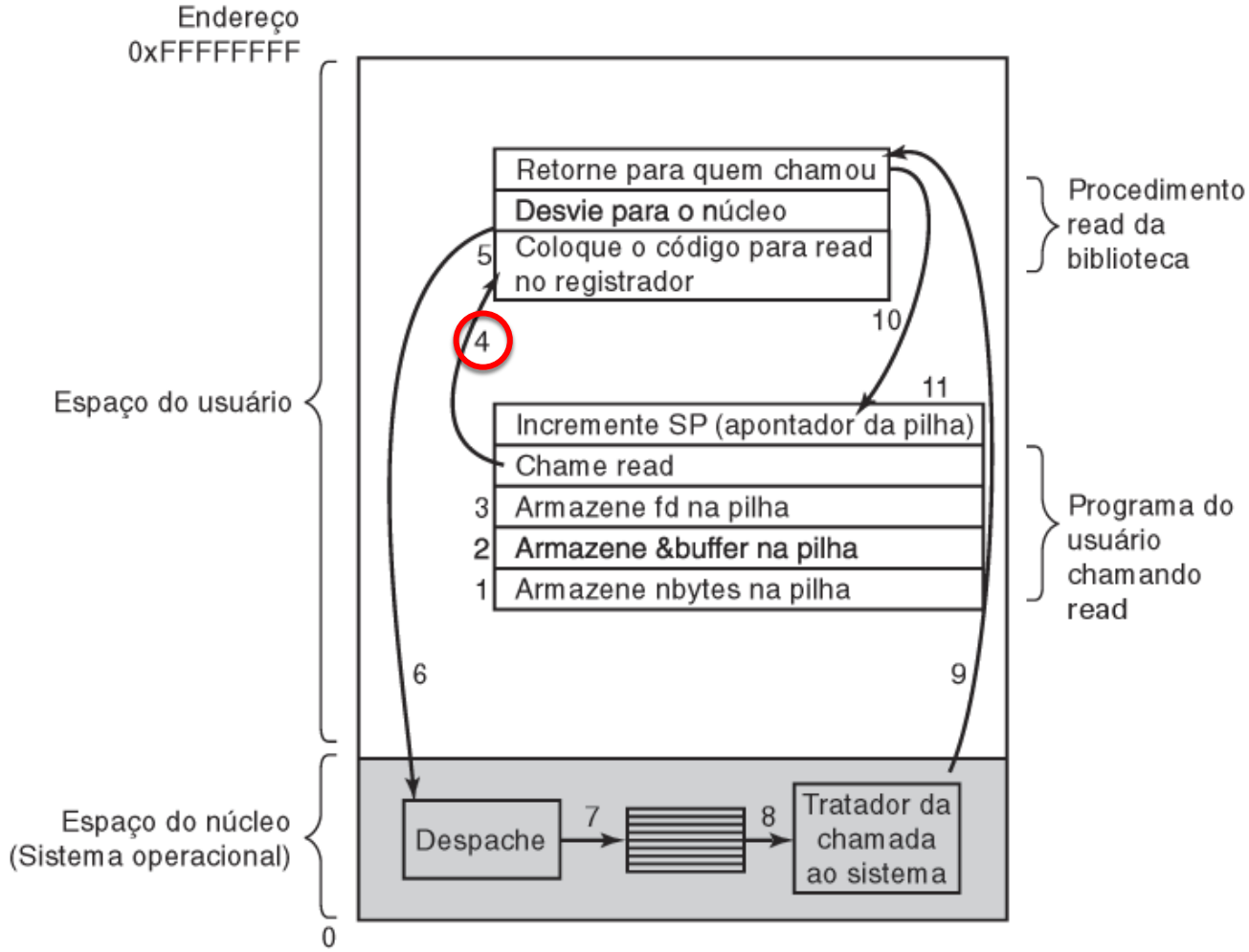
Os Passos de uma Chamada ao Sistema



Os 11 passos para fazer uma chamada ao Sistema

Ex. read (fd, buffer, nbytes)

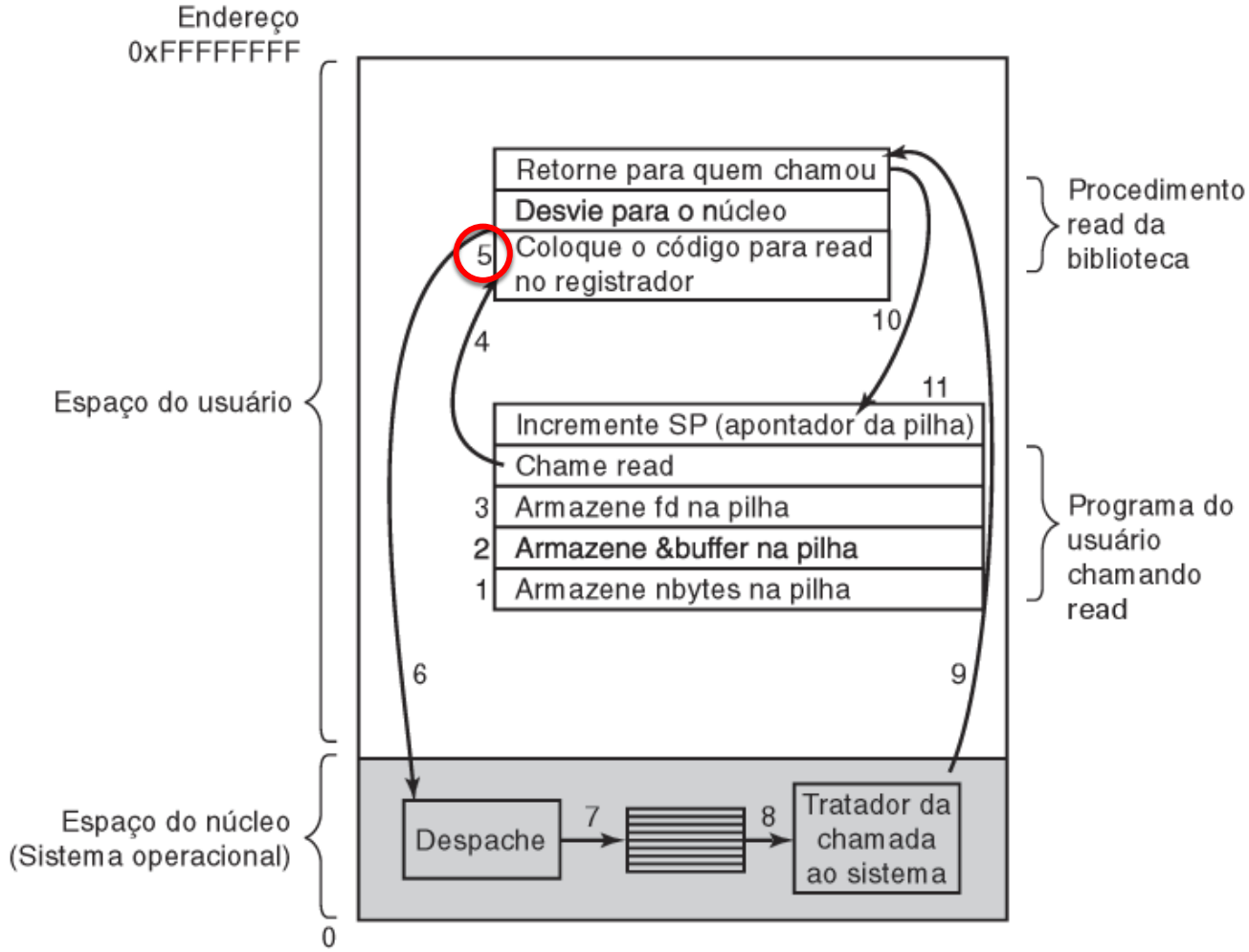
Os Passos de uma Chamada ao Sistema



Os 11 passos para fazer uma chamada ao Sistema

Ex. read (fd, buffer, nbytes)

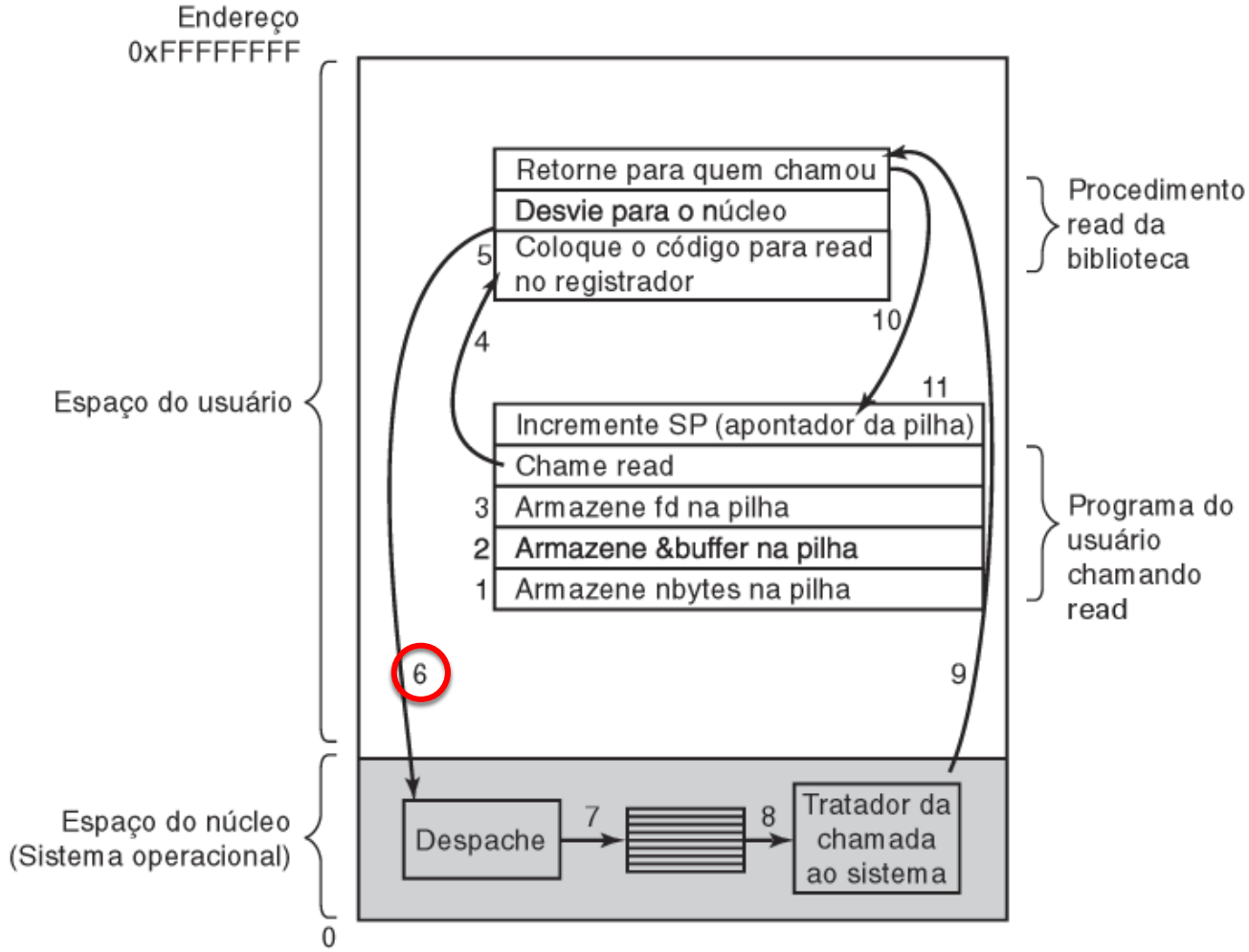
Os Passos de uma Chamada ao Sistema



Os 11 passos para fazer uma chamada ao Sistema

Ex. read (fd, buffer, nbytes)

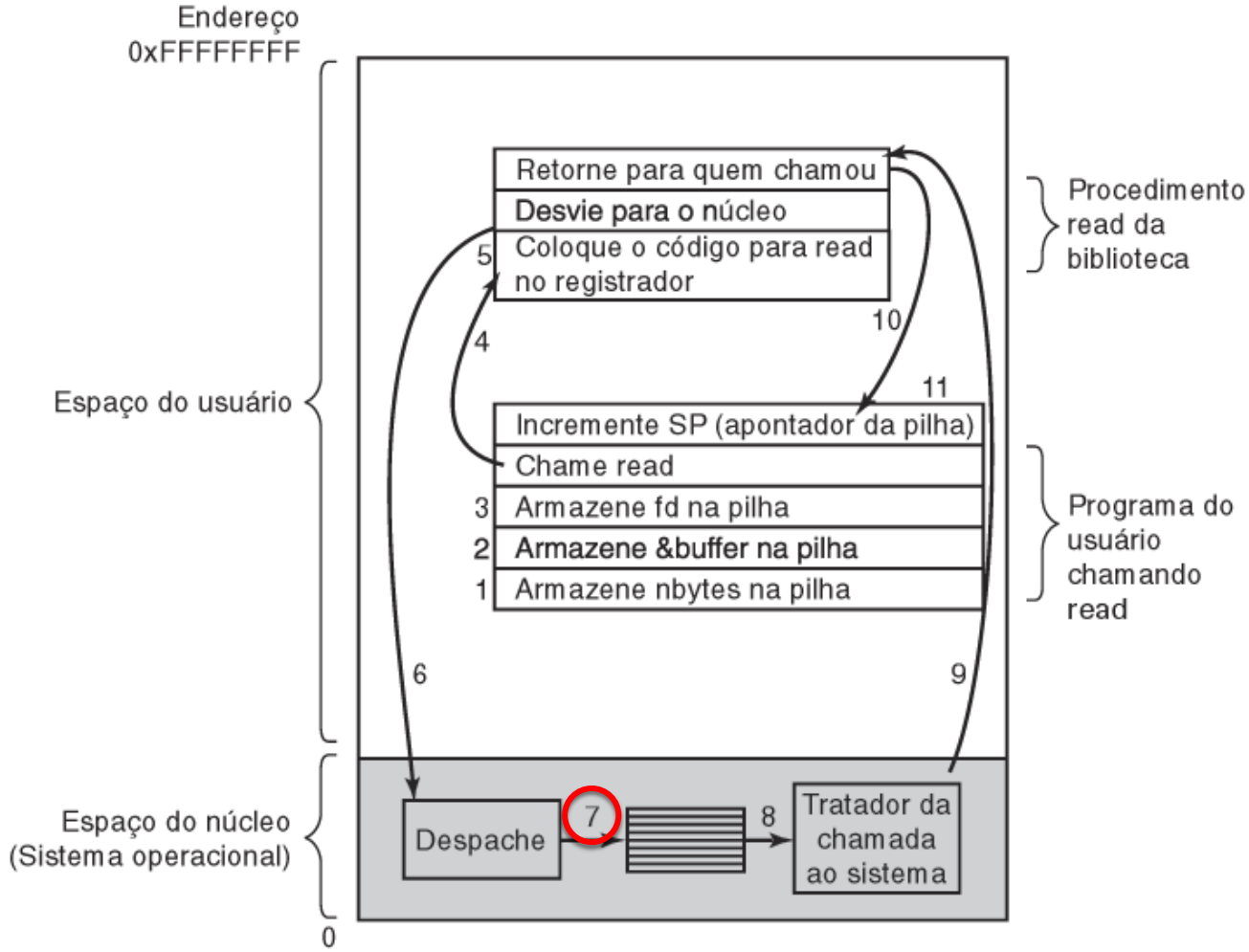
Os Passos de uma Chamada ao Sistema



Os 11 passos para fazer uma chamada ao Sistema

Ex. read (fd, buffer, nbytes)

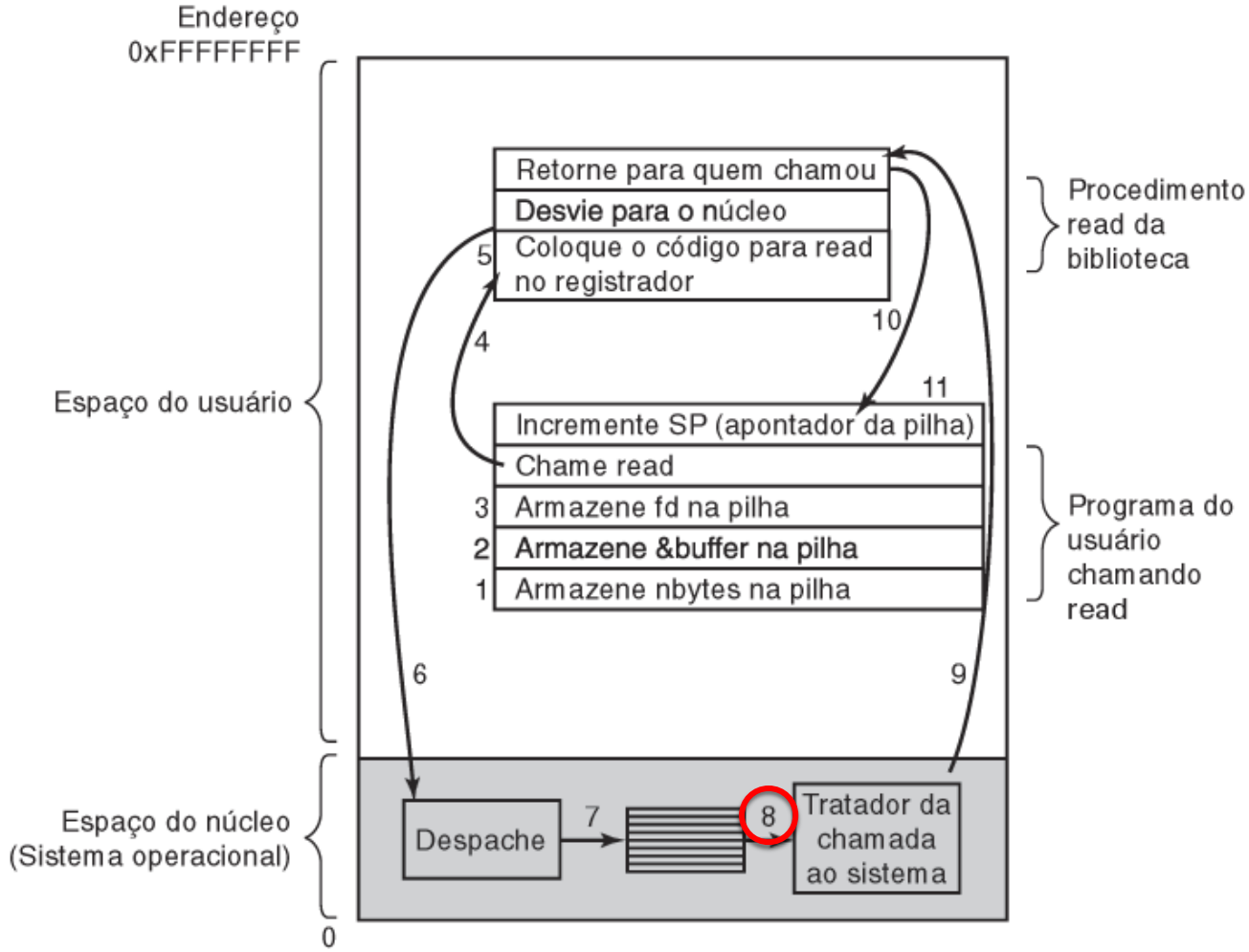
Os Passos de uma Chamada ao Sistema



Os 11 passos para fazer uma chamada ao Sistema

Ex. read (fd, buffer, nbytes)

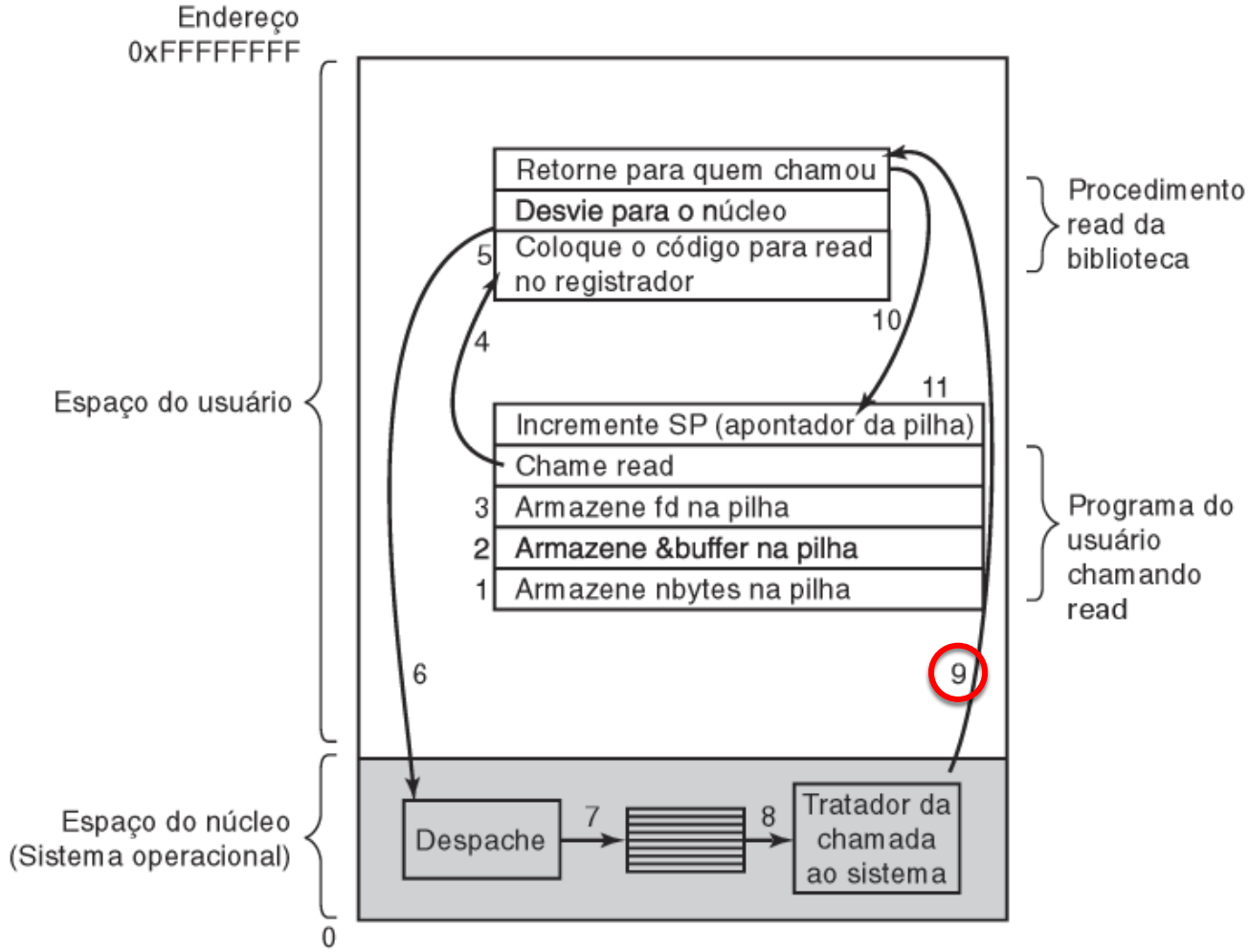
Os Passos de uma Chamada ao Sistema



Os 11 passos para fazer uma chamada ao Sistema

Ex. read (fd, buffer, nbytes)

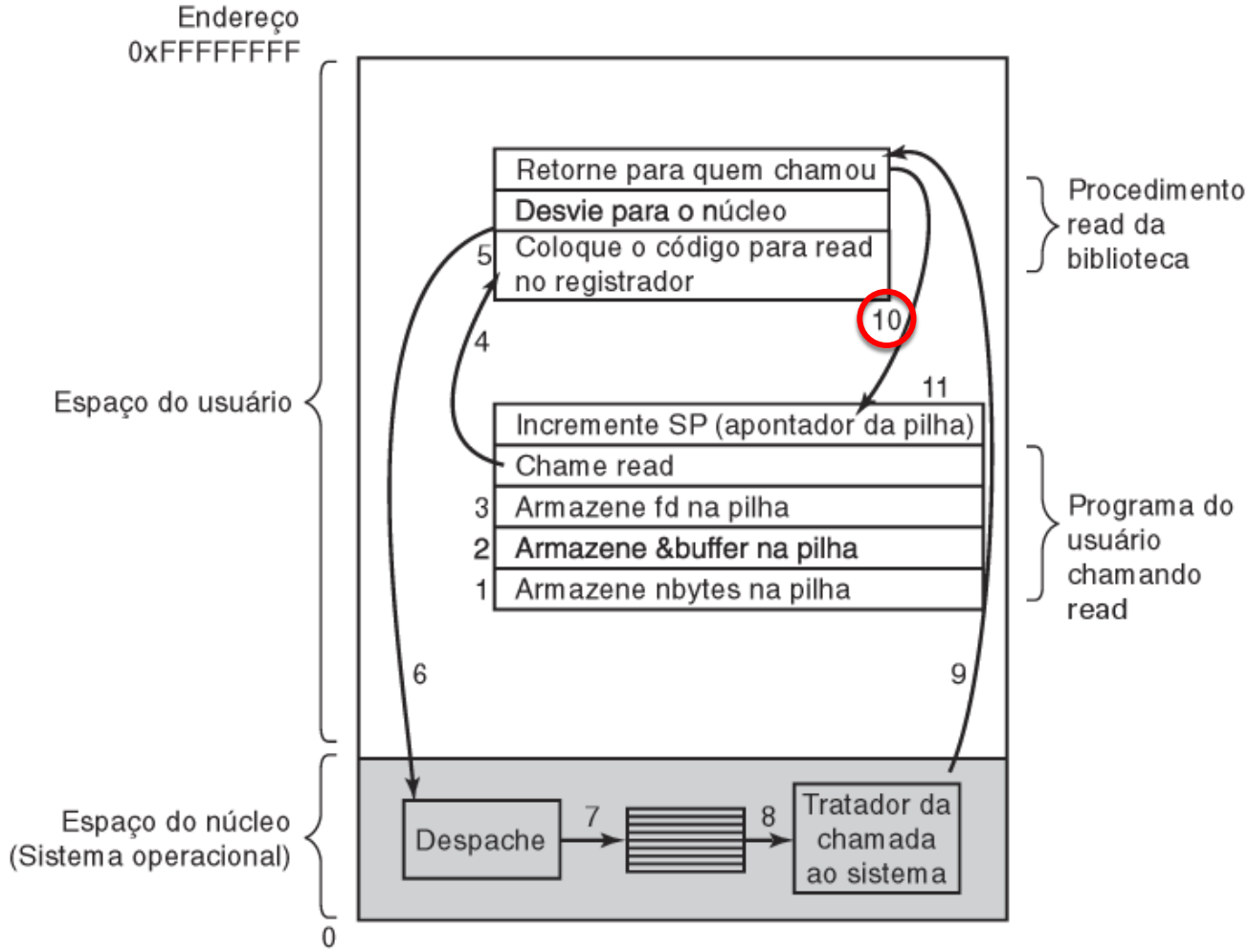
Os Passos de uma Chamada ao Sistema



Os 11 passos para fazer uma chamada ao Sistema

Ex. read (fd, buffer, nbytes)

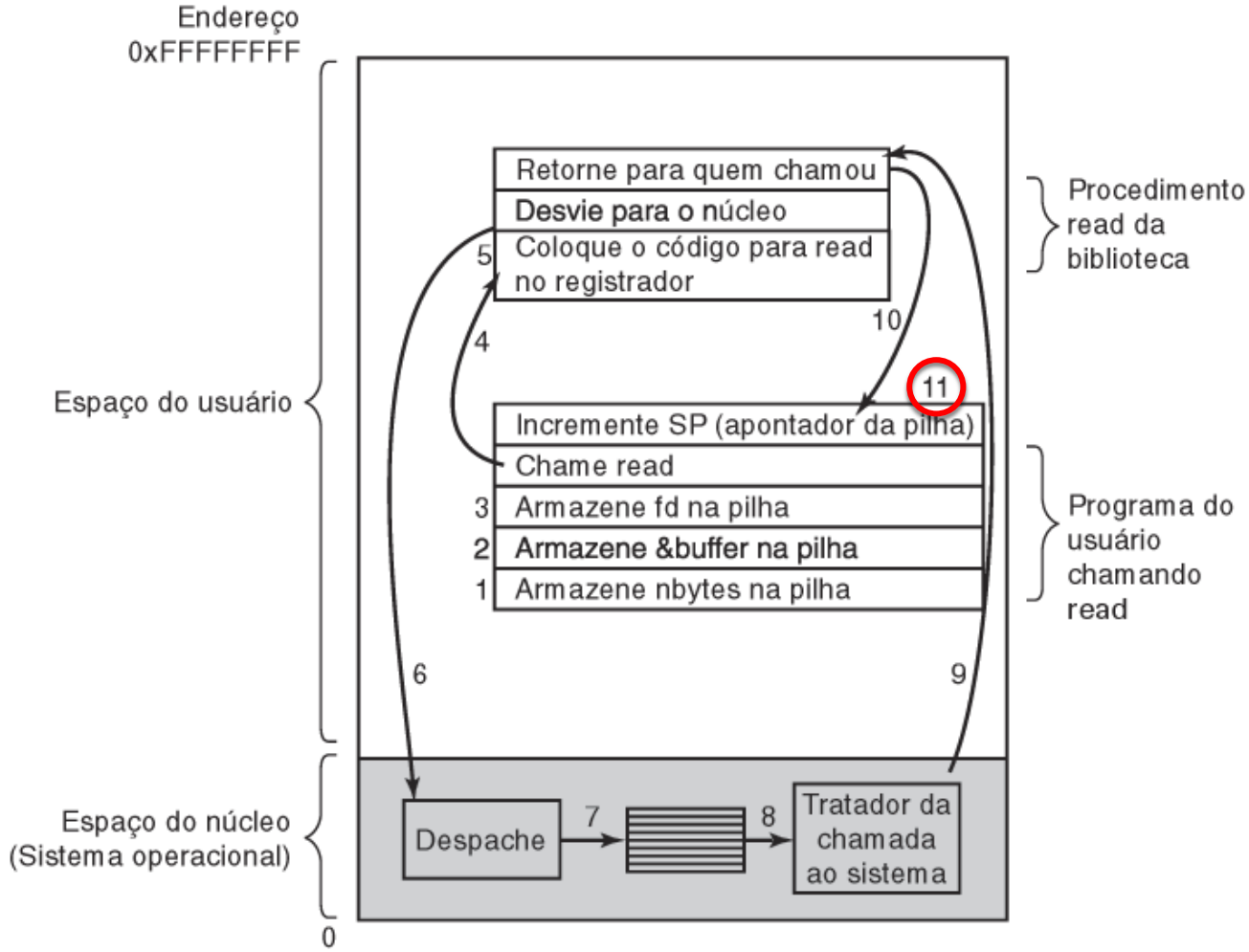
Os Passos de uma Chamada ao Sistema



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

Algumas Chamadas ao Sistema para Gerenciamento de Arquivos

Gerenciamento de arquivos

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

Algumas Chamadas ao Sistema para Gerenciamento de Diretório

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 Tarefas Diversas

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

```
#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

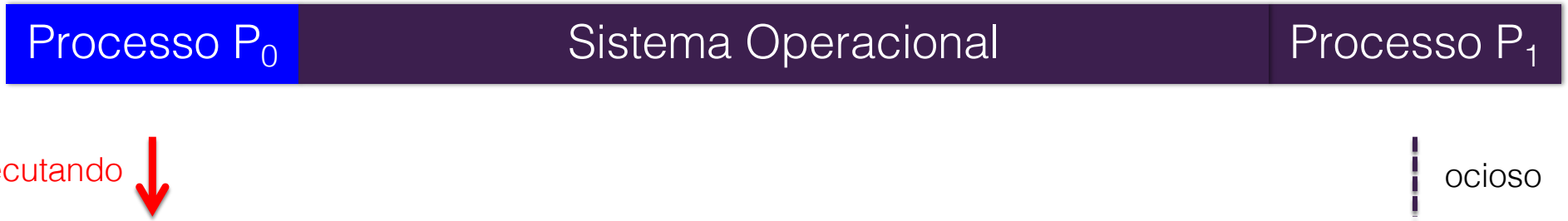
Processos e Threads

CIn | UFPE

SO: Multitarefa

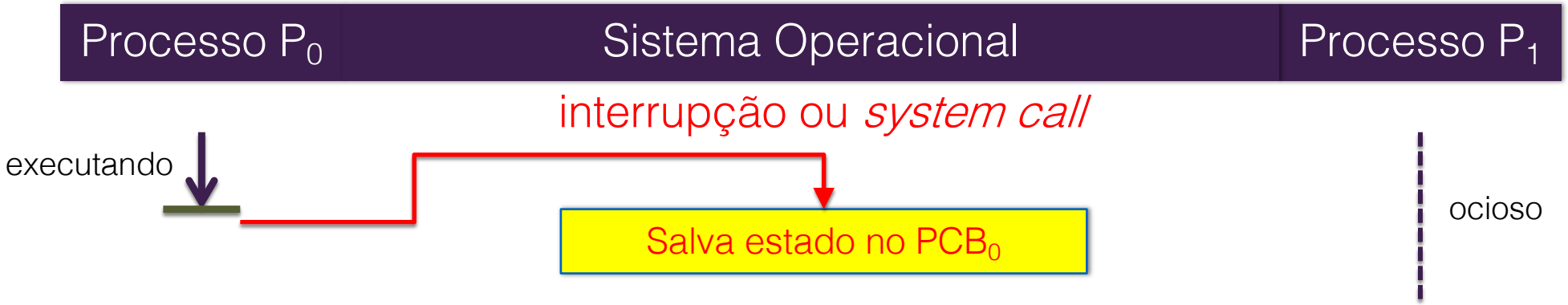
Processo P_0	Sistema Operacional	Processo P_1

SO: Multitarefa



Processo P_0 em execução

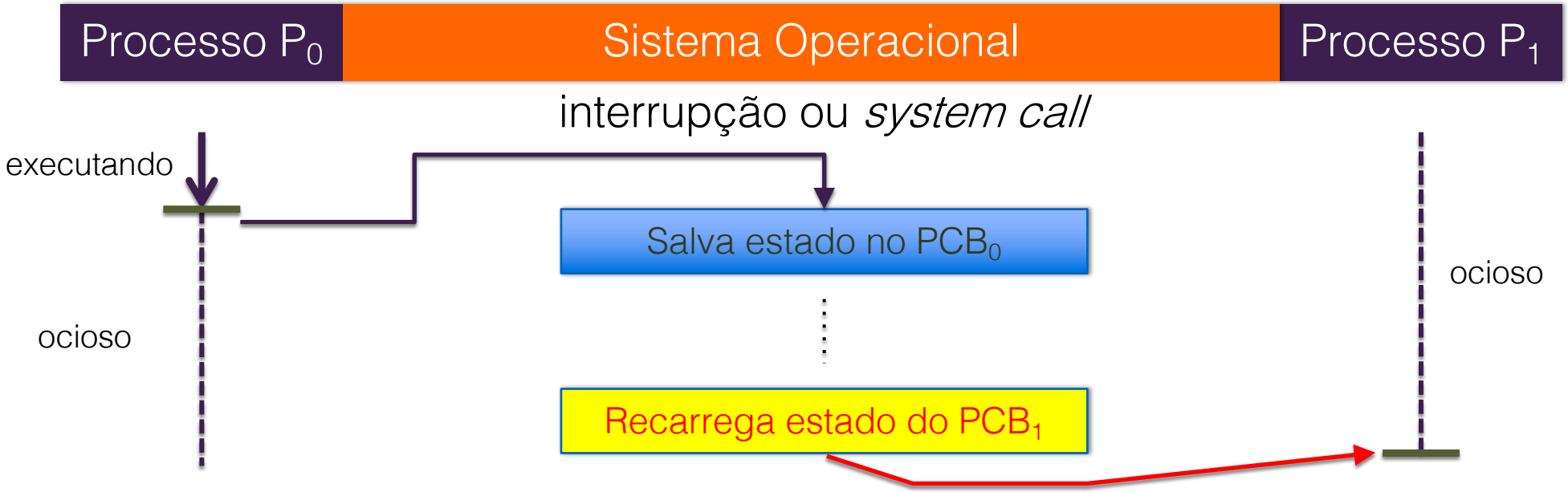
SO: Multitarefa



Process Control Block: informações de contexto do processo

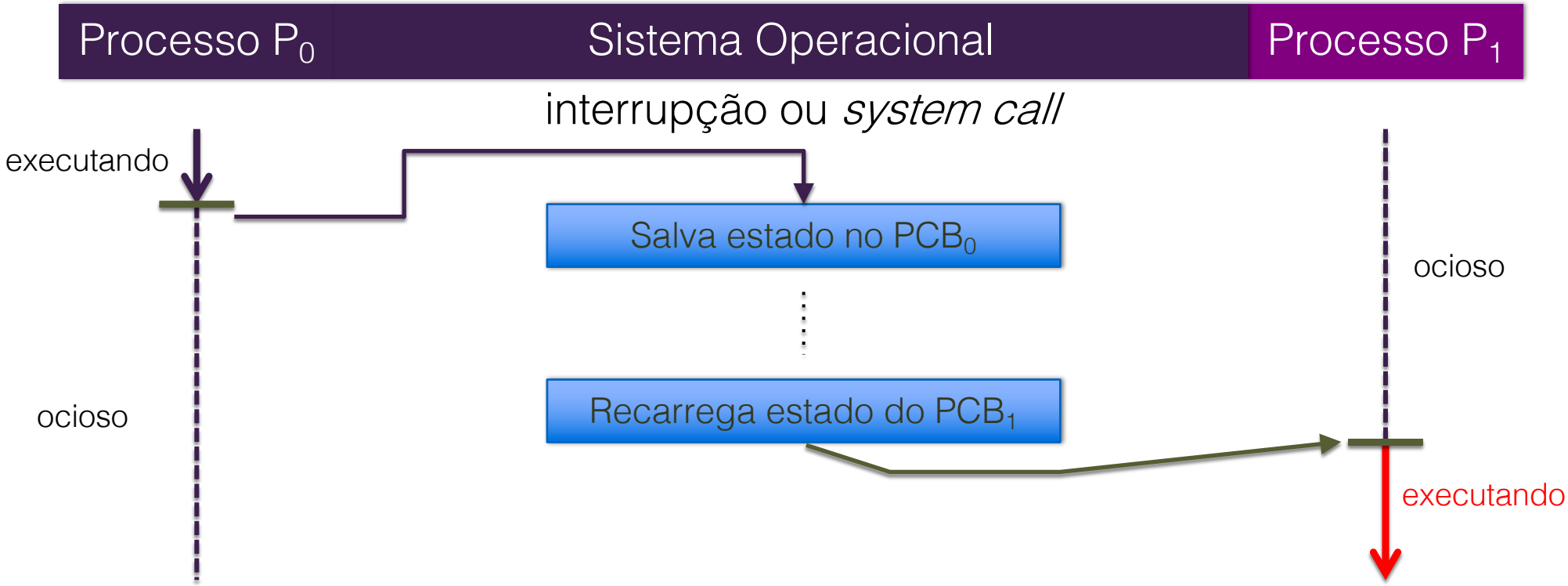
Salva estado do Processo P_0 no PCB_0

SO: Multitarefa



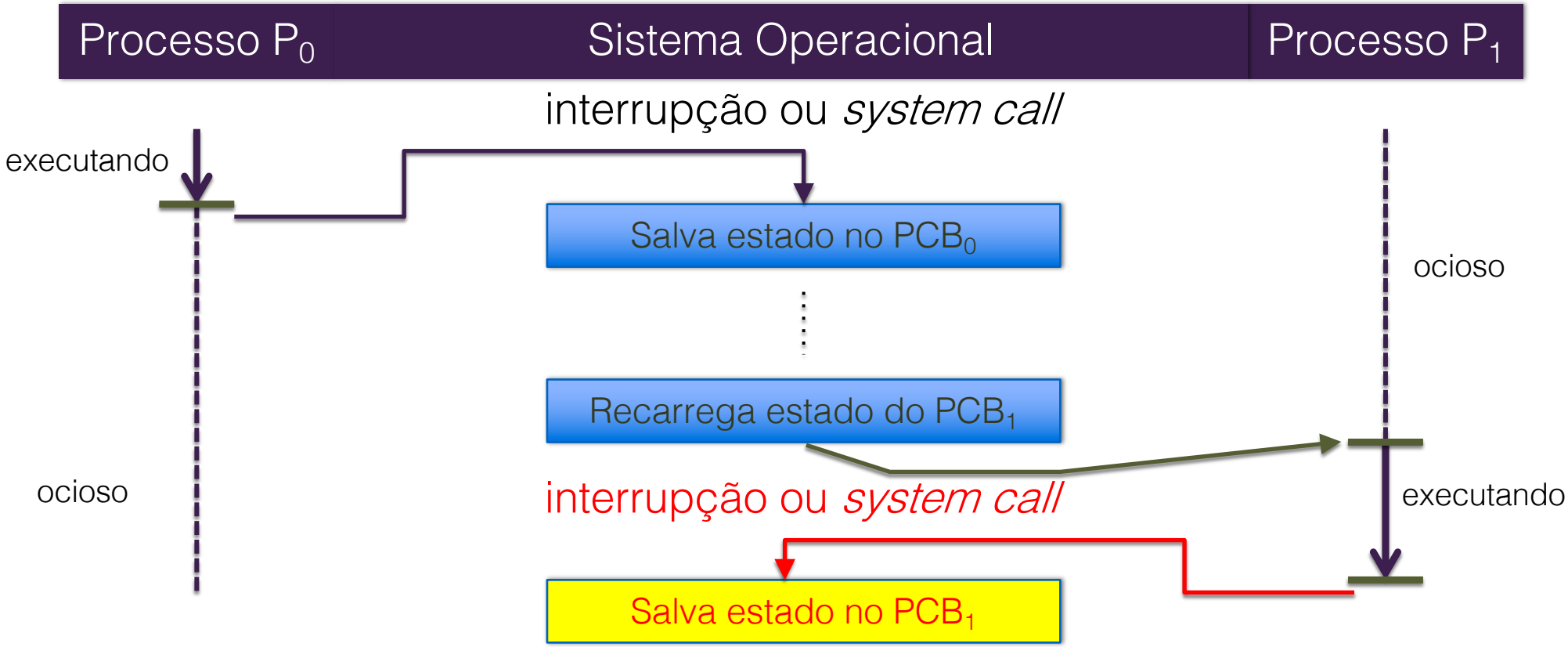
Estado do Processo P_1 é recarregado do PCB_1

SO: Multitarefa

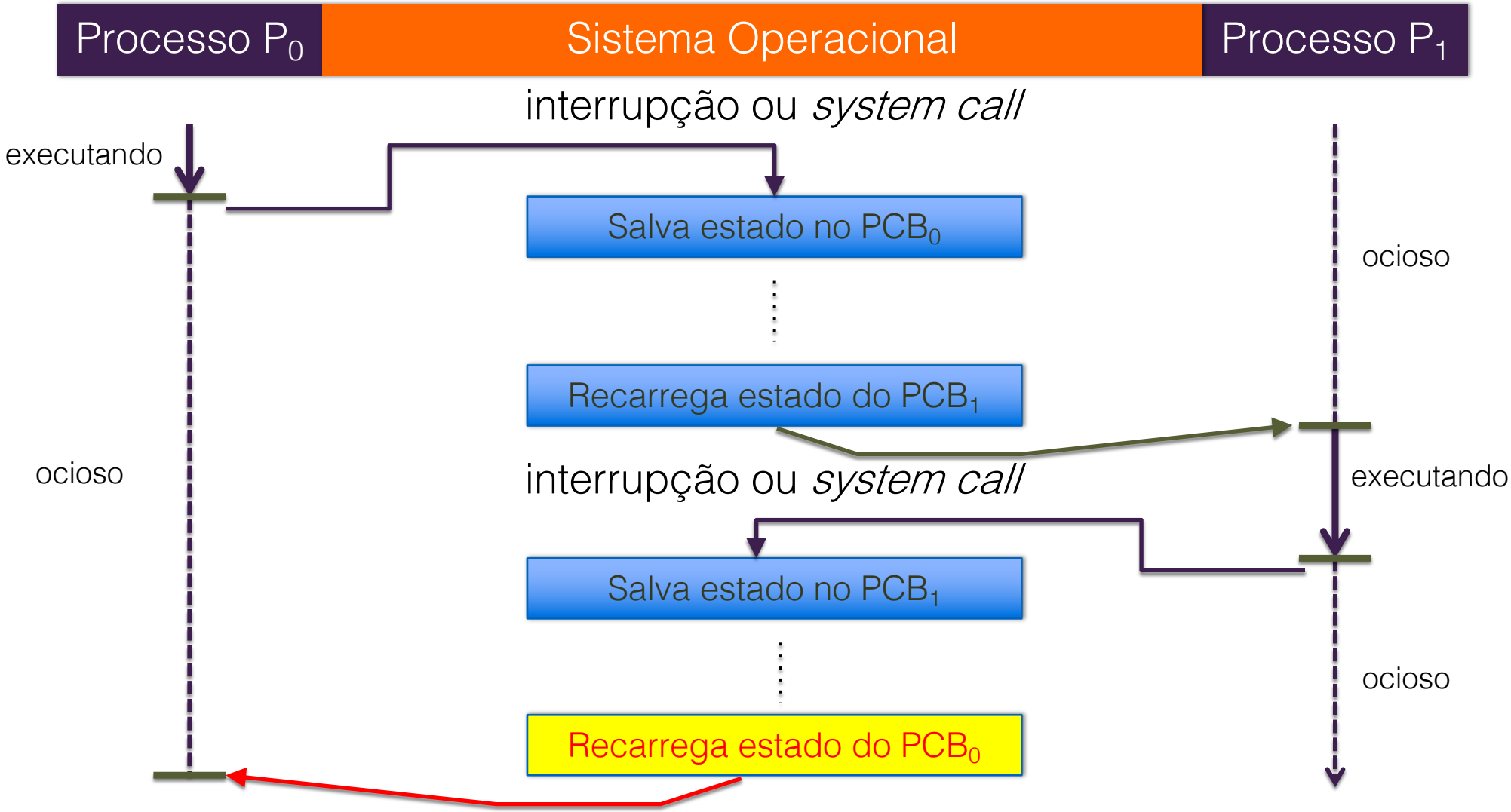


Processo P_1 em execução

SO: Multitarefa

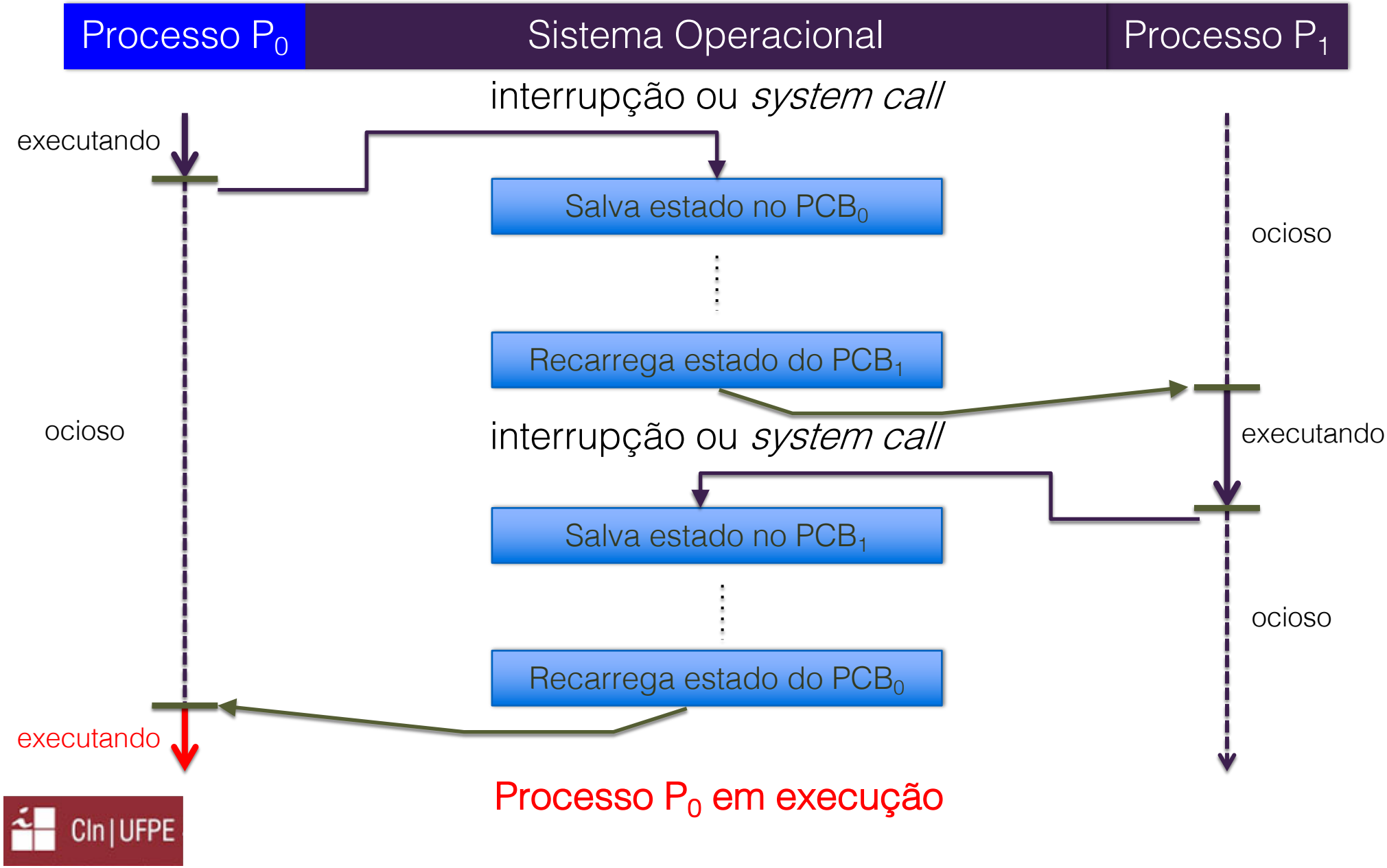


SO: Multitarefa



Estado do Processo P_0 é recarregado do PCB_0

SO: Multitarefa



Implementação de Processos

Gerenciamento de processos	Gerenciamento de memória	Gerenciamento de arquivos
Registradores	Ponteiro para o segmento de código	Diretório-raiz
Contador de programa	Ponteiro para o segmento de dados	Diretório de trabalho
Palavra de estado do programa	Ponteiro para o segmento de pilha	Descritores de arquivos
Ponteiro de pilha		Identificador (ID) do usuário
Estado do processo		Identificador (ID) do grupo
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		

Notificação da ocorrência de um evento

Exs.:

SIGILL - Illegal instruction

SIGINT - Interrupt

SIGKILL - Kill (terminate immediately)

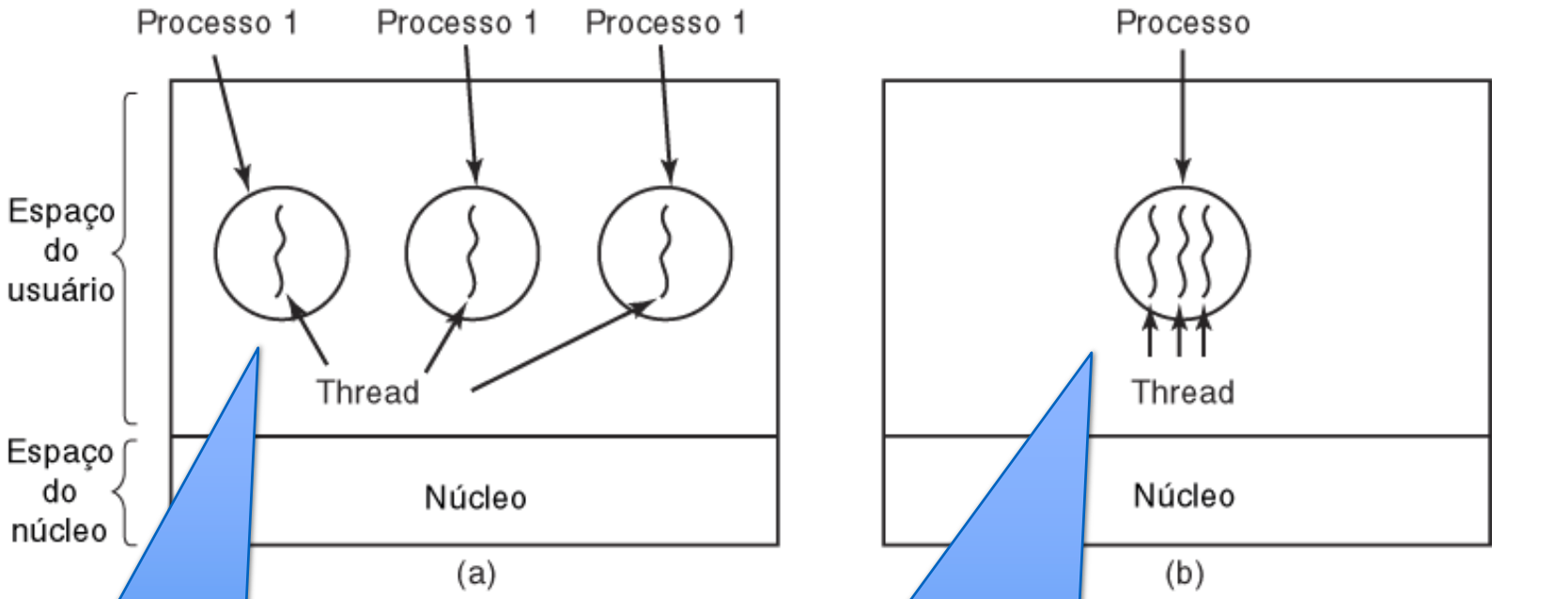
SIGSYS - Bad system call

SIGXCPU - CPU time limit exceeded

SIGXFSZ - File size limit exceeded

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

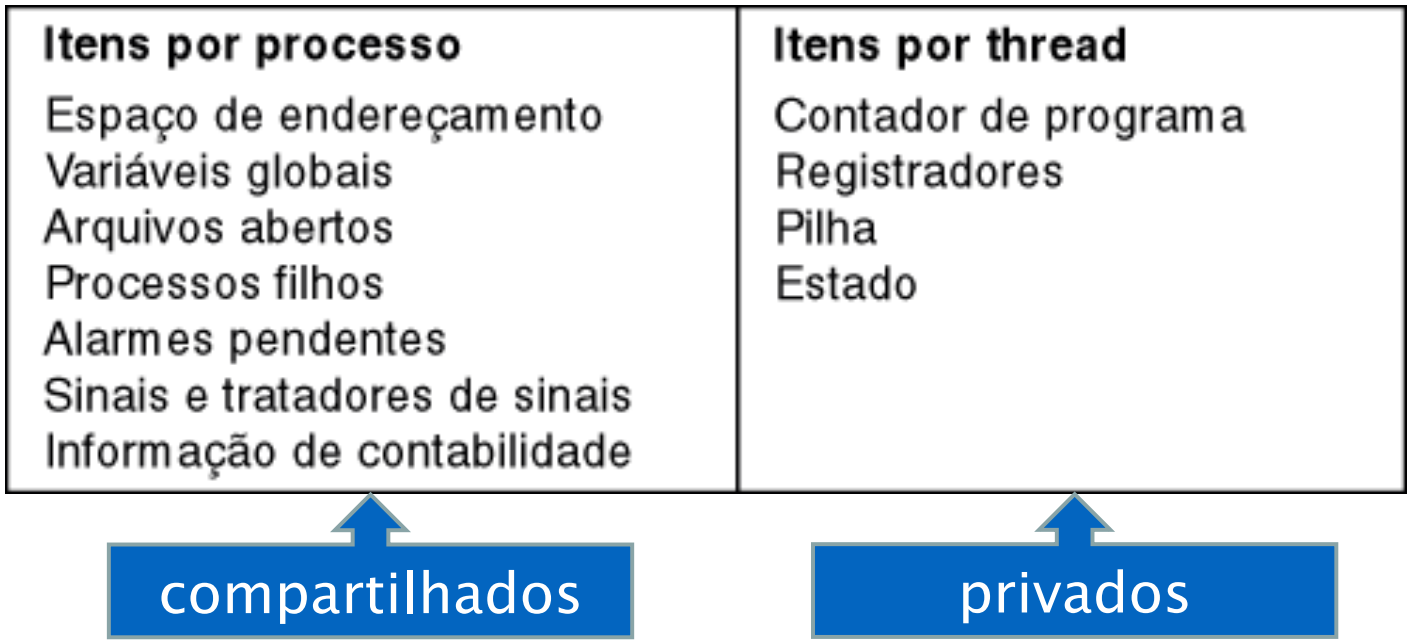
Threads: O Modelo (I)



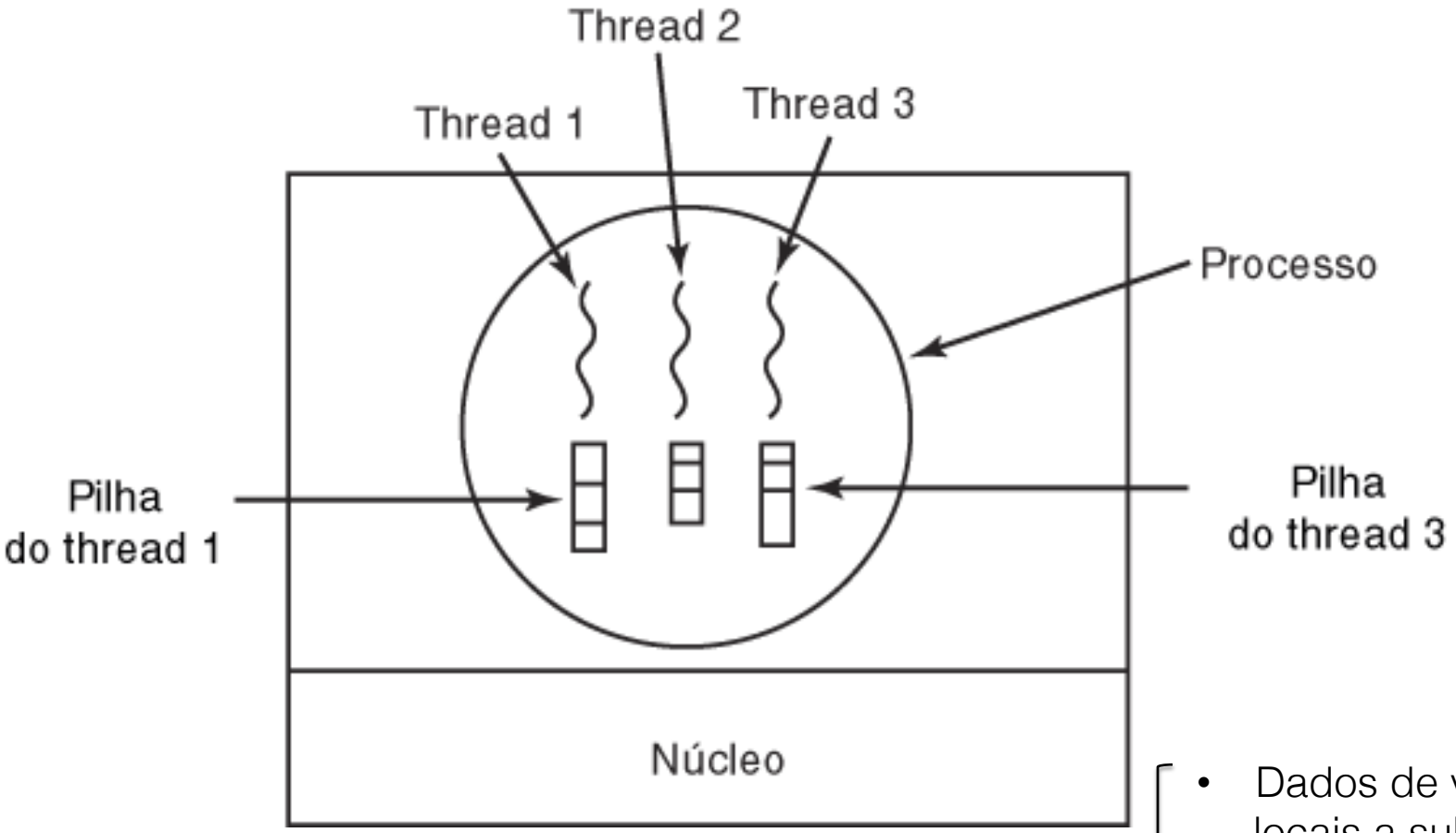
Cada processo tem um espaço de endereçamento e uma única linha (*thread*) de controle/execução

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)



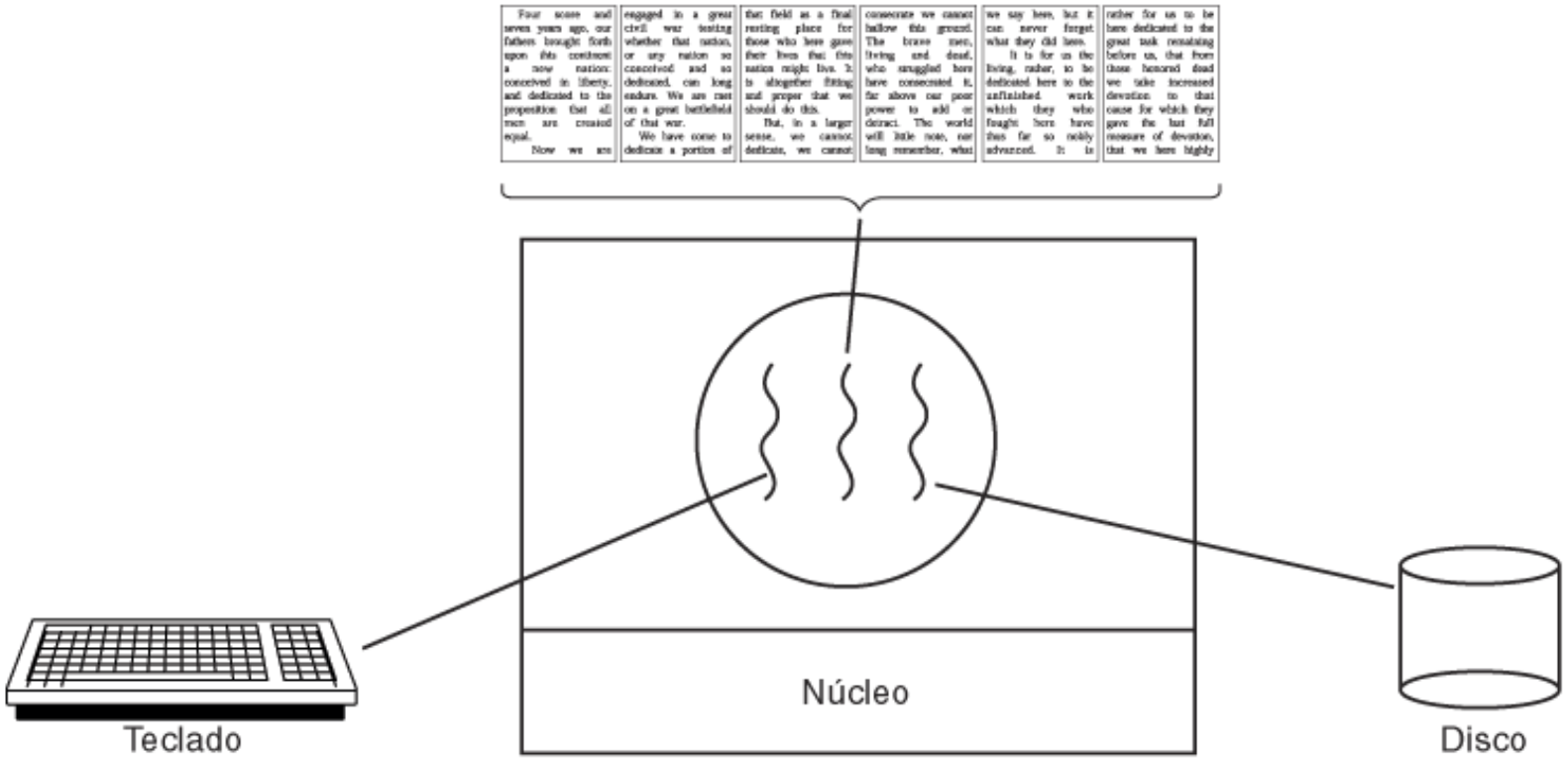
O Modelo de Thread (3)



Cada thread tem sua própria pilha

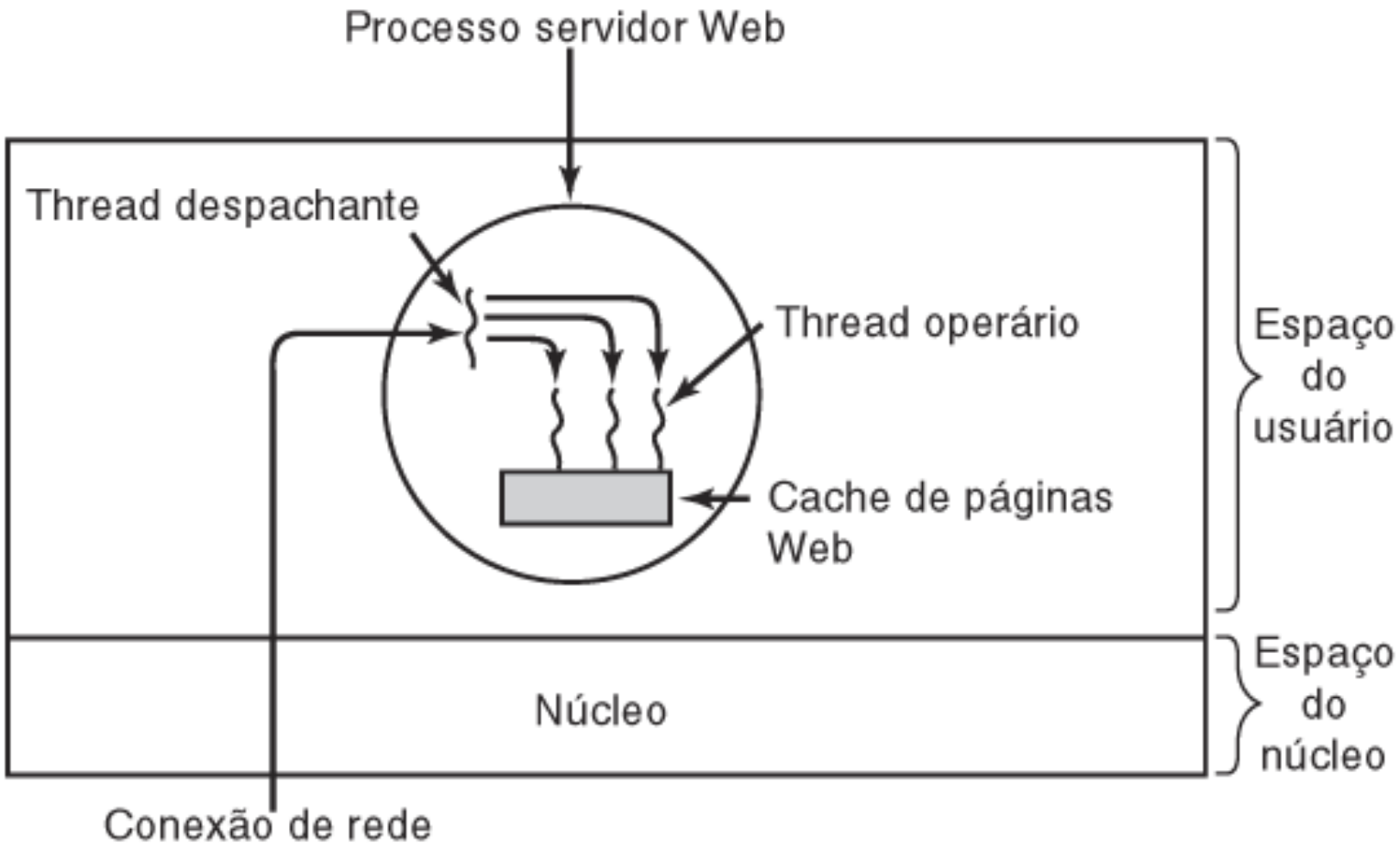
- Dados de variáveis locais a sub-rotinas
- Dados do endereço de retorno de uma sub-rotina

Uso de Thread (I)



Um processador de texto com três threads

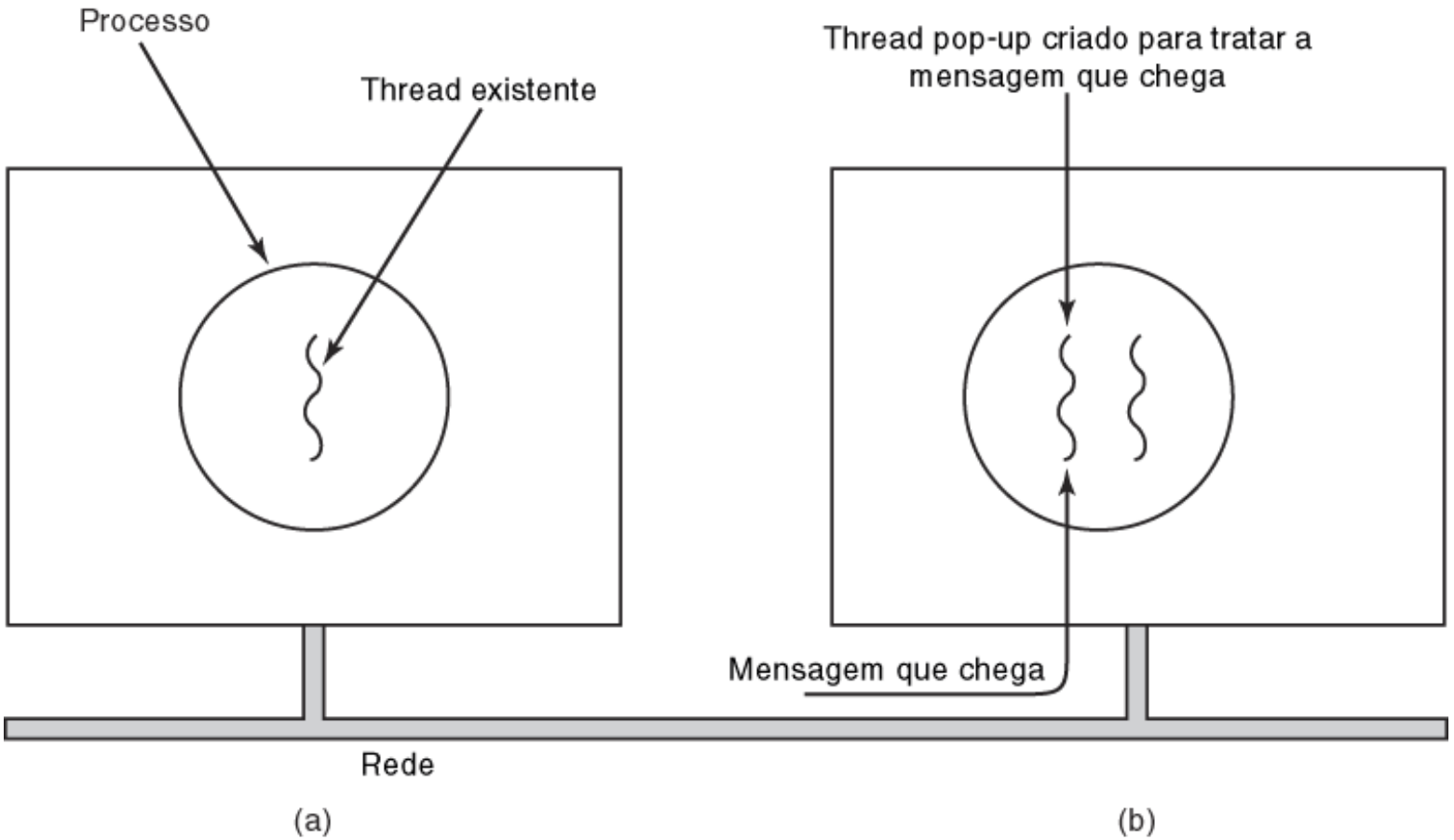
Uso de Thread (2)



Um servidor web com múltiplas threads

vs um serviço Web com múltiplos servidores (mais adiante: SD)

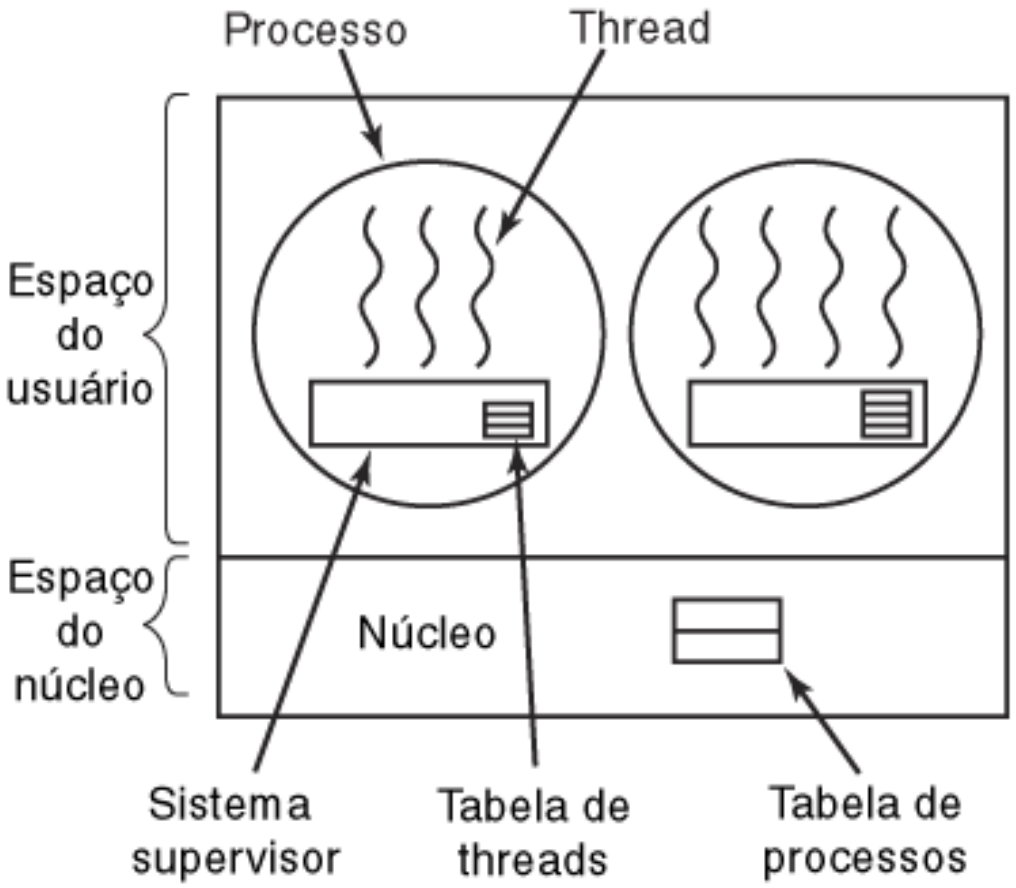
Criação de uma nova thread quando chega uma mensagem



Tipos de Threads

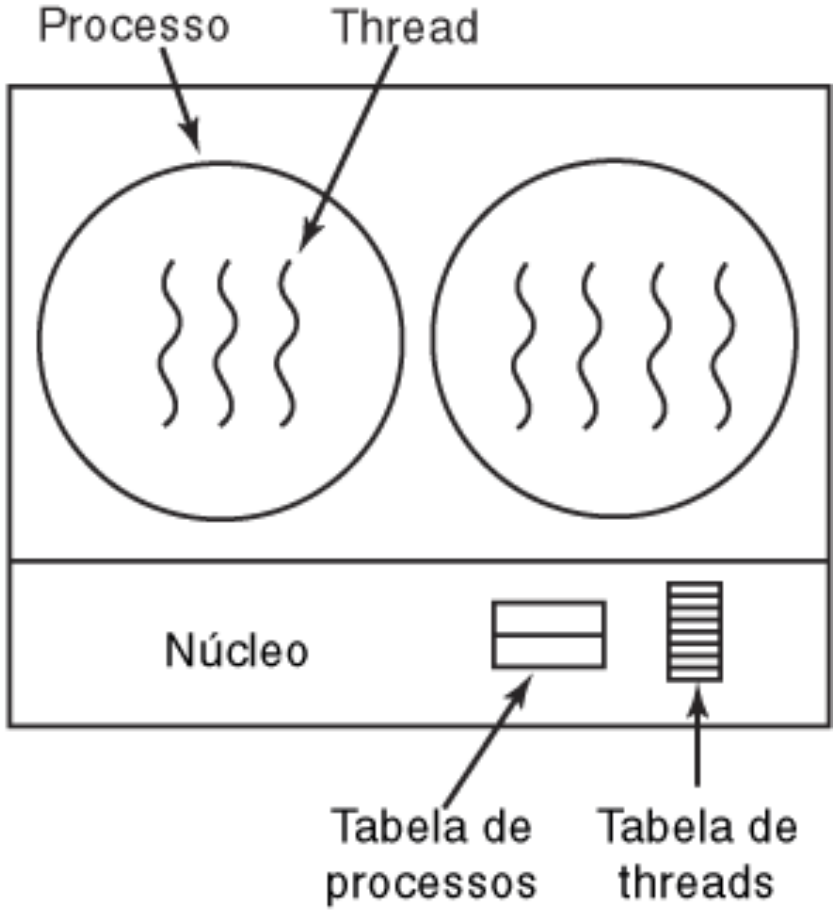
- Threads = processos leves
- Threads de usuário
  - SO gerencia tabela de processos
- Threads de núcleo
  - SO gerencia tabelas de processos e de threads (de núcleo)

Implementação de Threads de Usuário



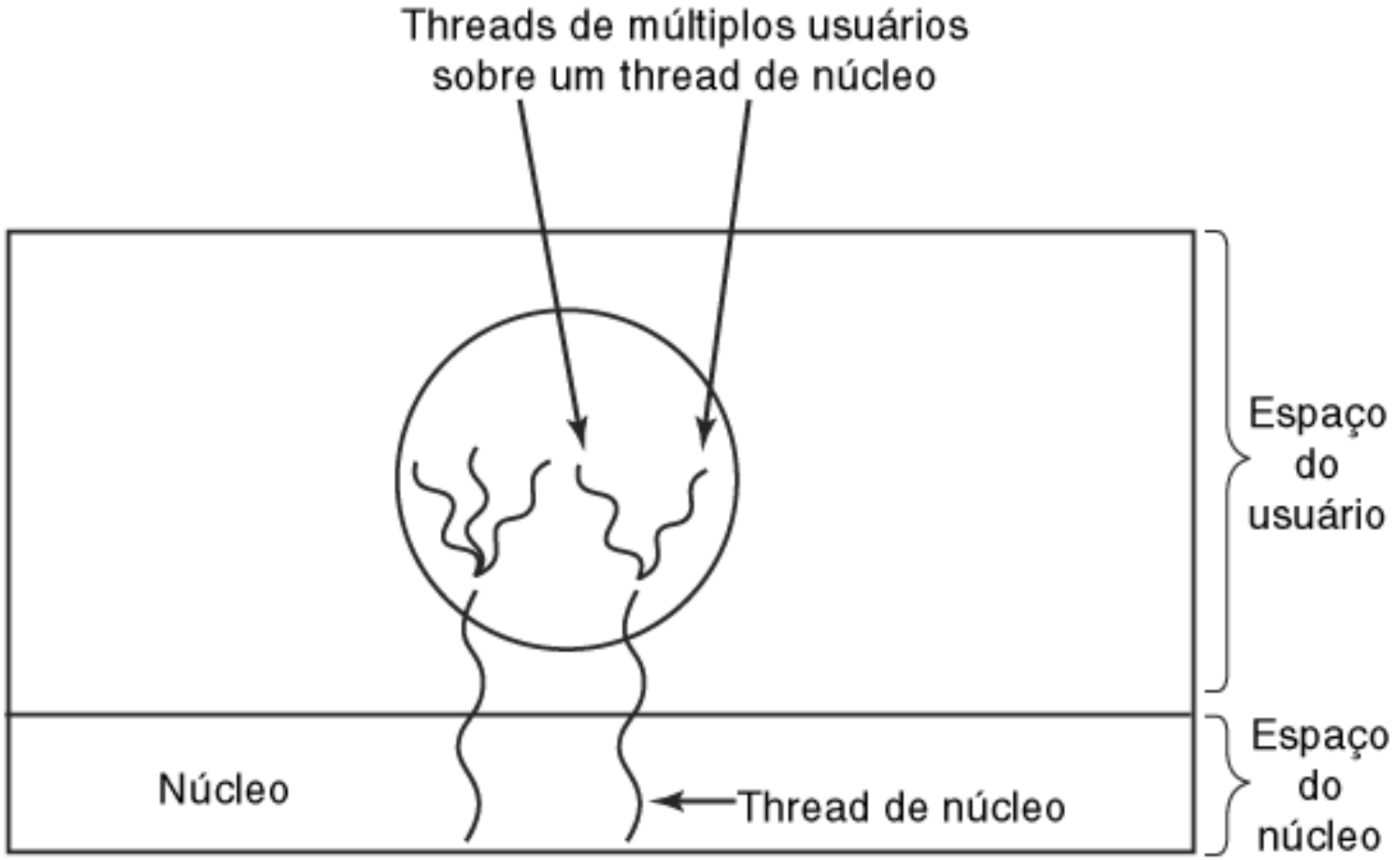
Um pacote de threads de usuário

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 fácies (rápidas) de criar e destruir do que processos
  - Criar uma *thread* pode ser **10-100 vezes mais rápido** que criar um processo
  - Quando a necessidade do número de *threads* muda dinâmica e rapidamente, esta propriedade é bastante útil

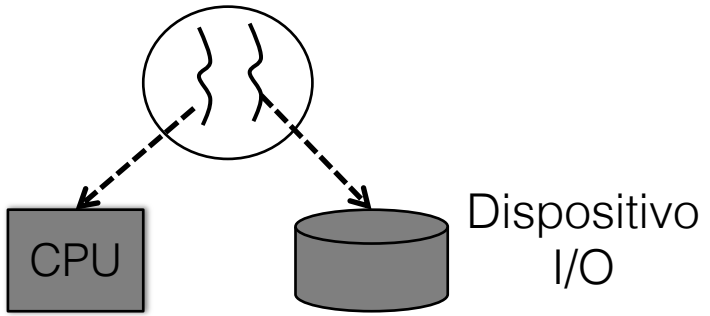
Comparação de desempenho: processo x thread

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...

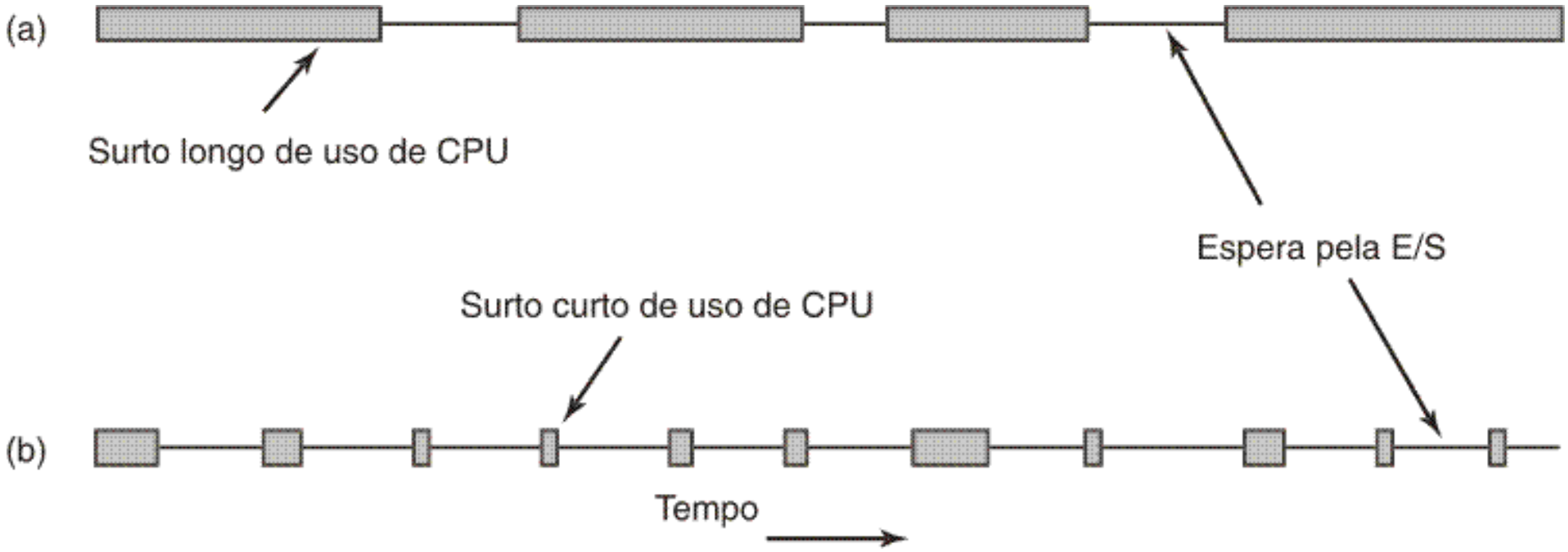
Tipos de Processo
(incl. *thread* – processo leve)

- 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)

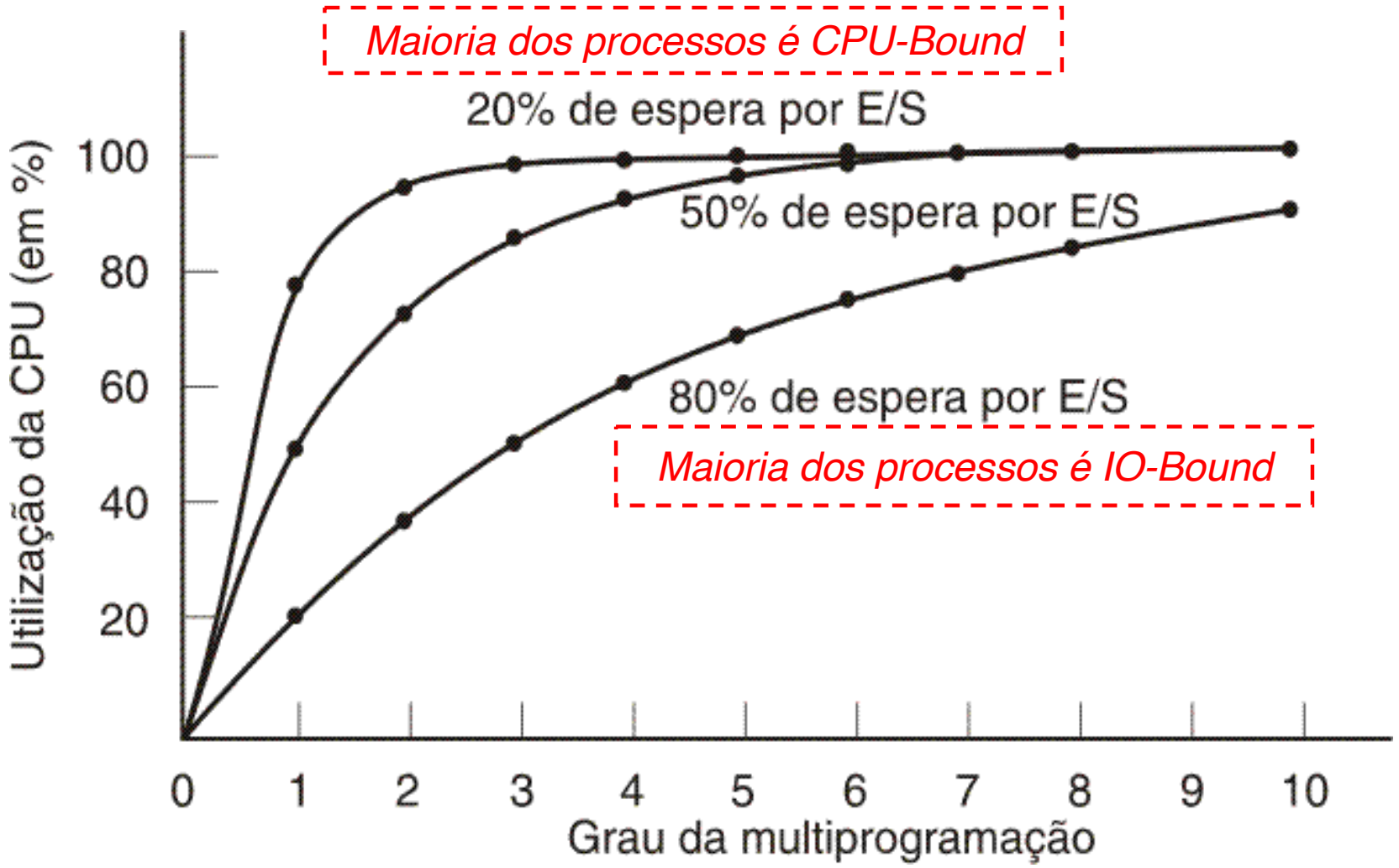
➢ processos I/O-bound devem ter prioridade sobre processos CPU-bound

Comportamentos de Processos

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

Centro de Informática U·F·P·E

Próximas Atividades

13/09/18	QUINTA	Prática de Linux e C/C++, **LabG5**
20/09/18	QUINTA	Prática de C/C++ e Concorrência, **LabG5**

25/09/18	TERÇA	Exercício de concorrência, **LabG3**
02/10/18	TERÇA	1o. EE, **D-002**

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 escalonador, e o algoritmo usado para tal é chamado de algoritmo de escalonamento

Comportamento escalonamento-processo

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

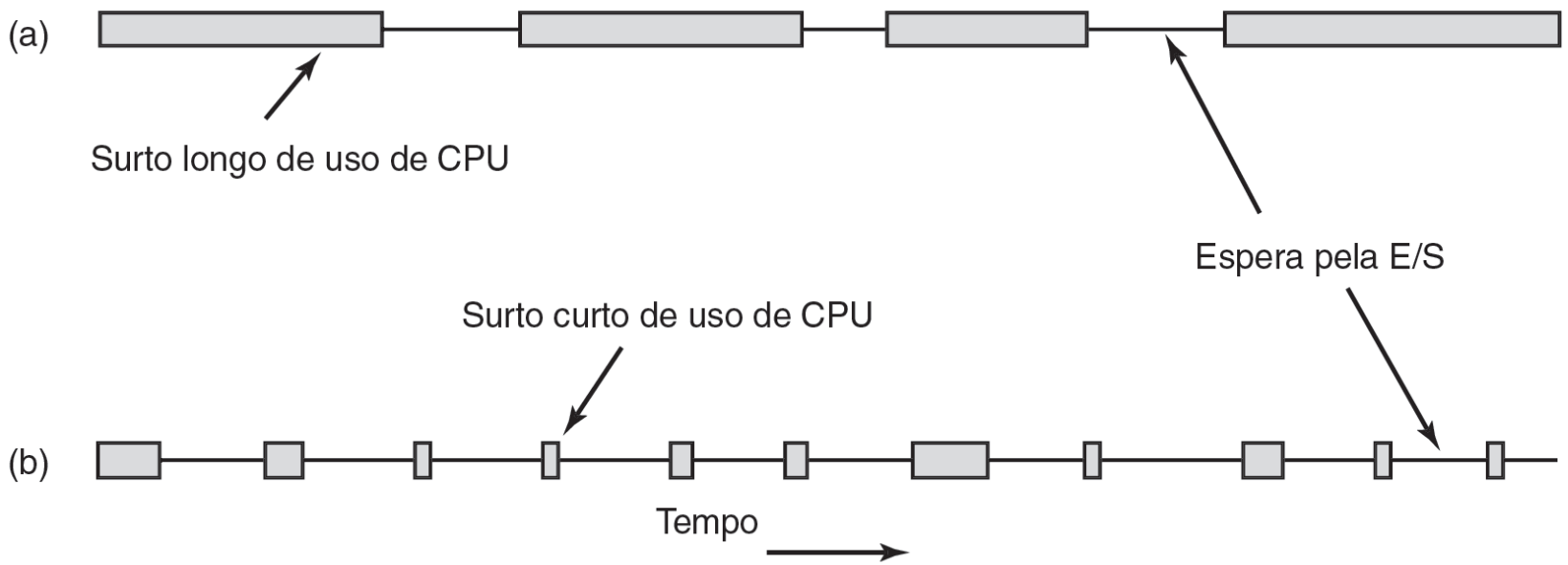
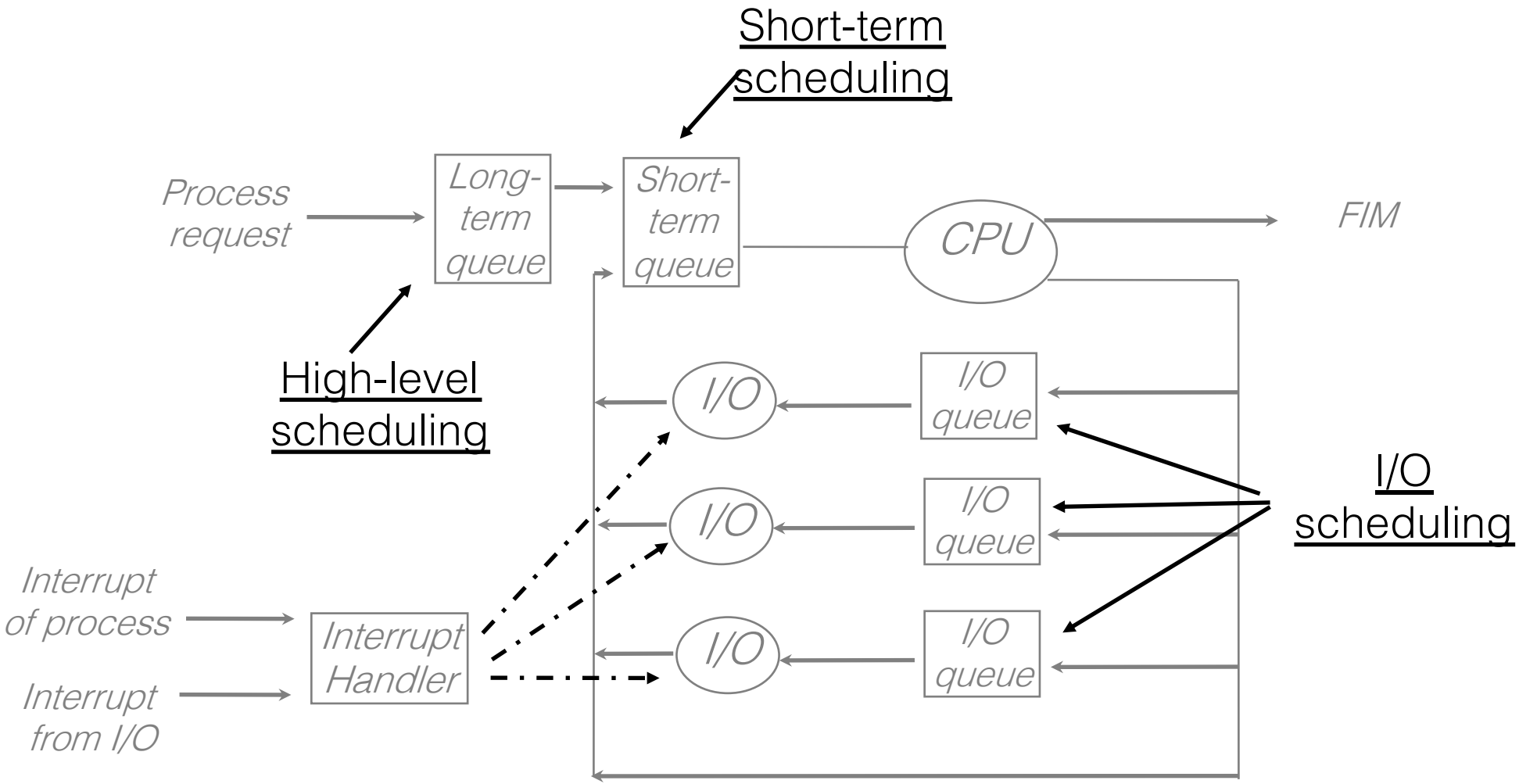


Figura 2.31 Usos de surtos de CPU se alternam com períodos de espera por E/S. (a) Um processo orientado à CPU. (b) Um processo orientado à E/S.

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)
- Interativo
- Tempo-real

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

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.

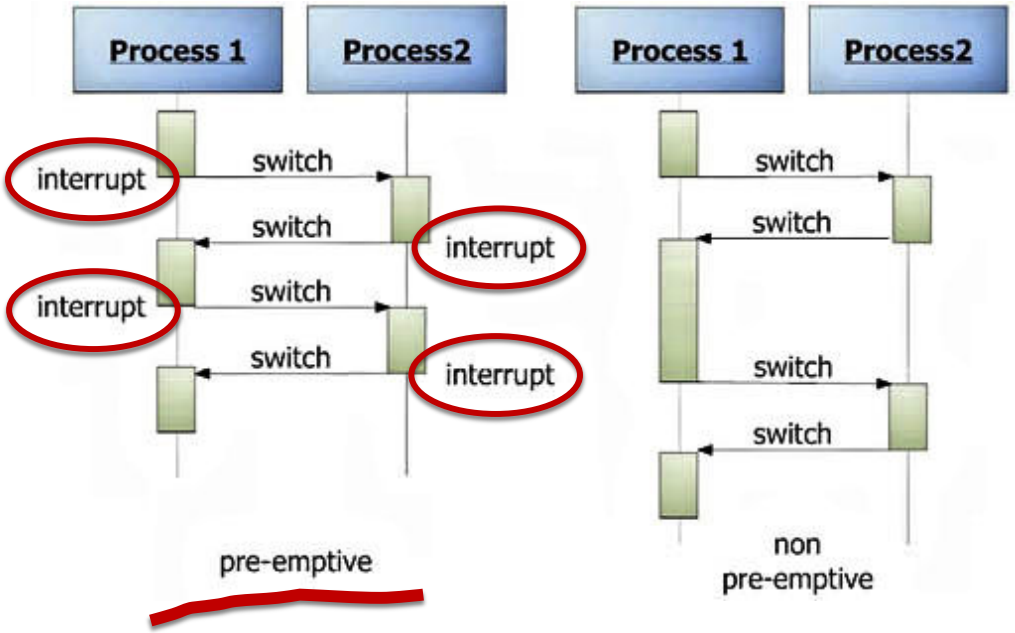
Tipos de Escalonamento

- 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

Escalonamento Preemptivo

- Permite a suspensão temporária de processos
- *Quantum* ou *time-slice*: período de tempo durante o qual um processo usa o processador a cada vez



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 ½ 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

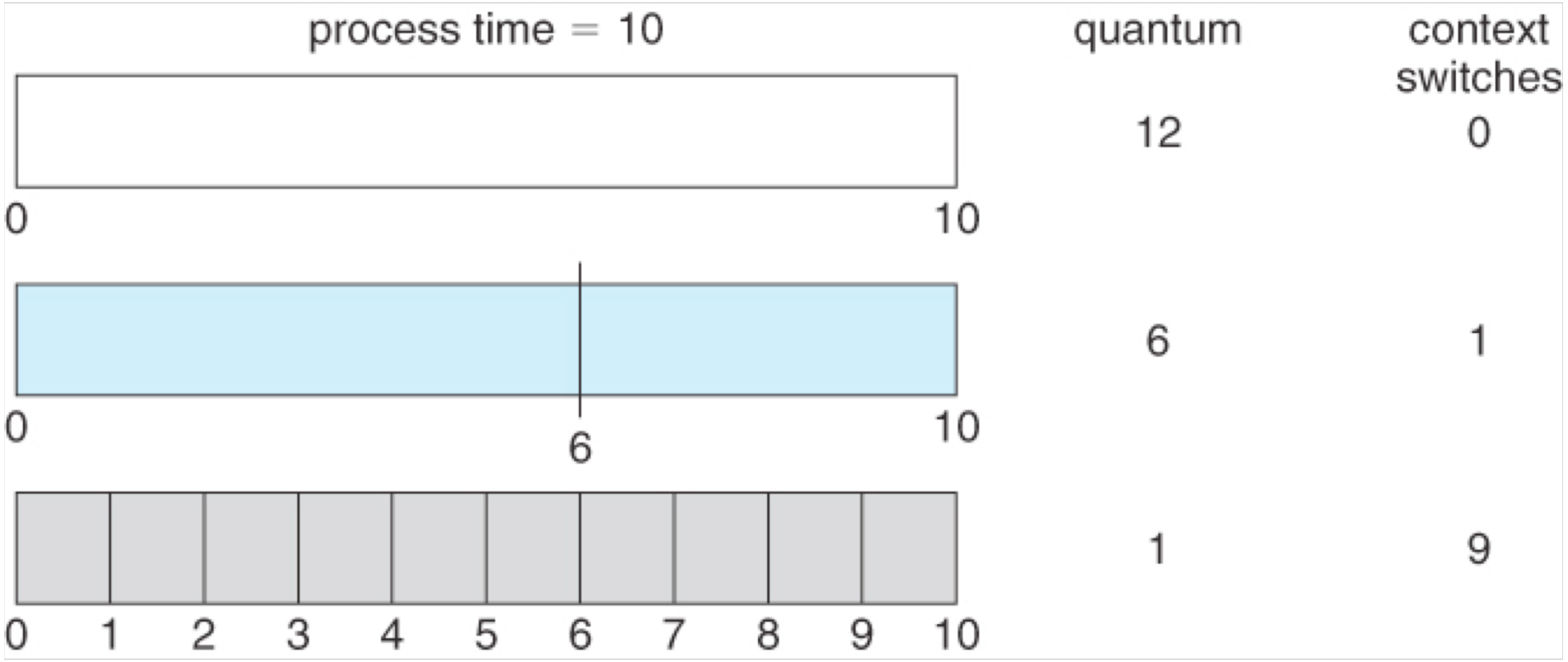
Quantum grande:
Diminui número de mudanças de contexto e *overhead* do S.O., mas...
Ruim para processos interativos

Escalonamento de Processos

Algoritmos (cont.)

CIn | UFPE

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

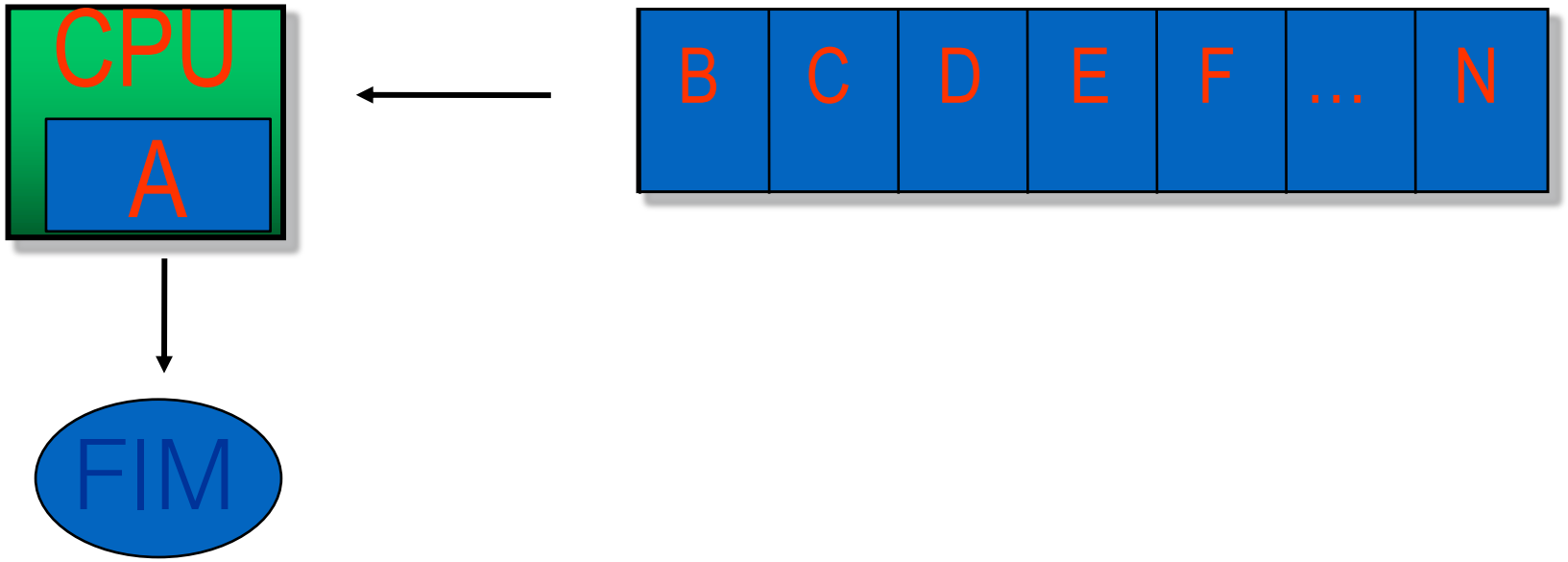
Priorização dinâmica

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

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 mútiplas
- Processo mais curto é o próximo
- Escalonamento garantido
- Escalonamento por loteria
- Escalonamento por fração justa

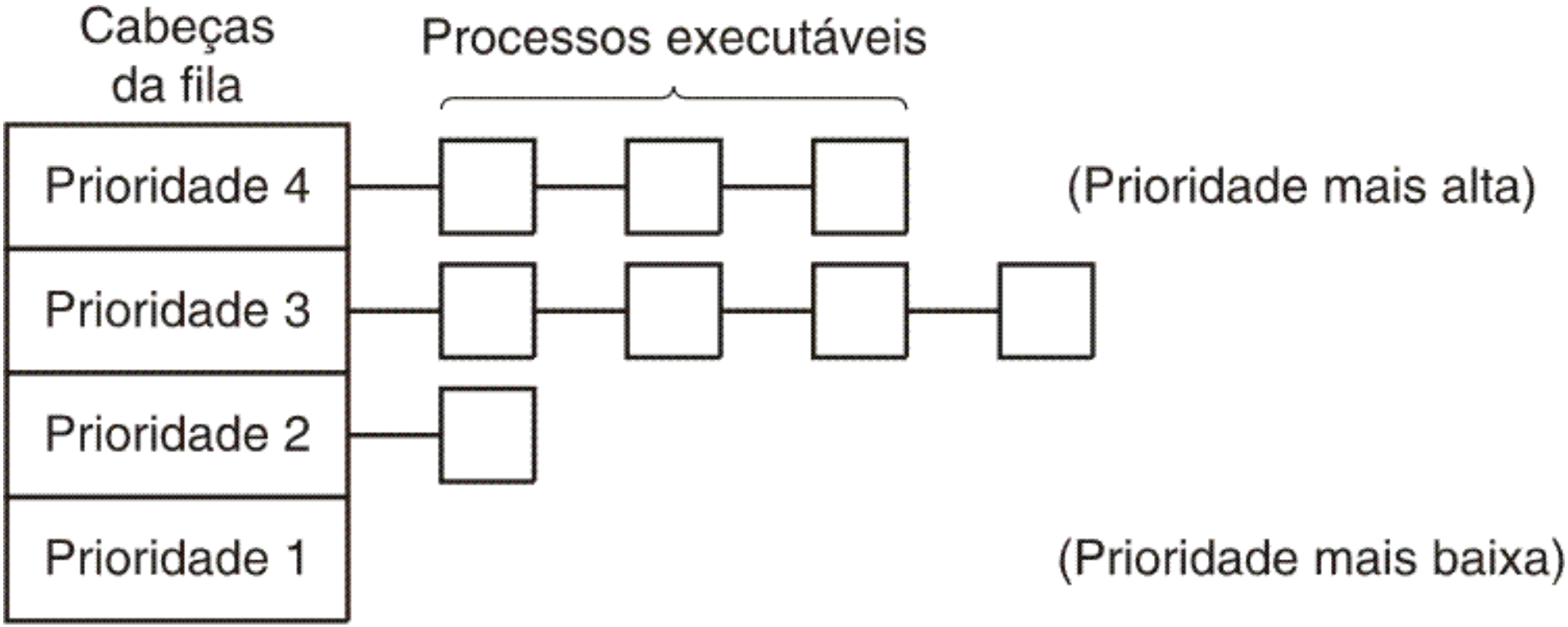
Escalonamento em Sistemas Interativos (1)

- 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

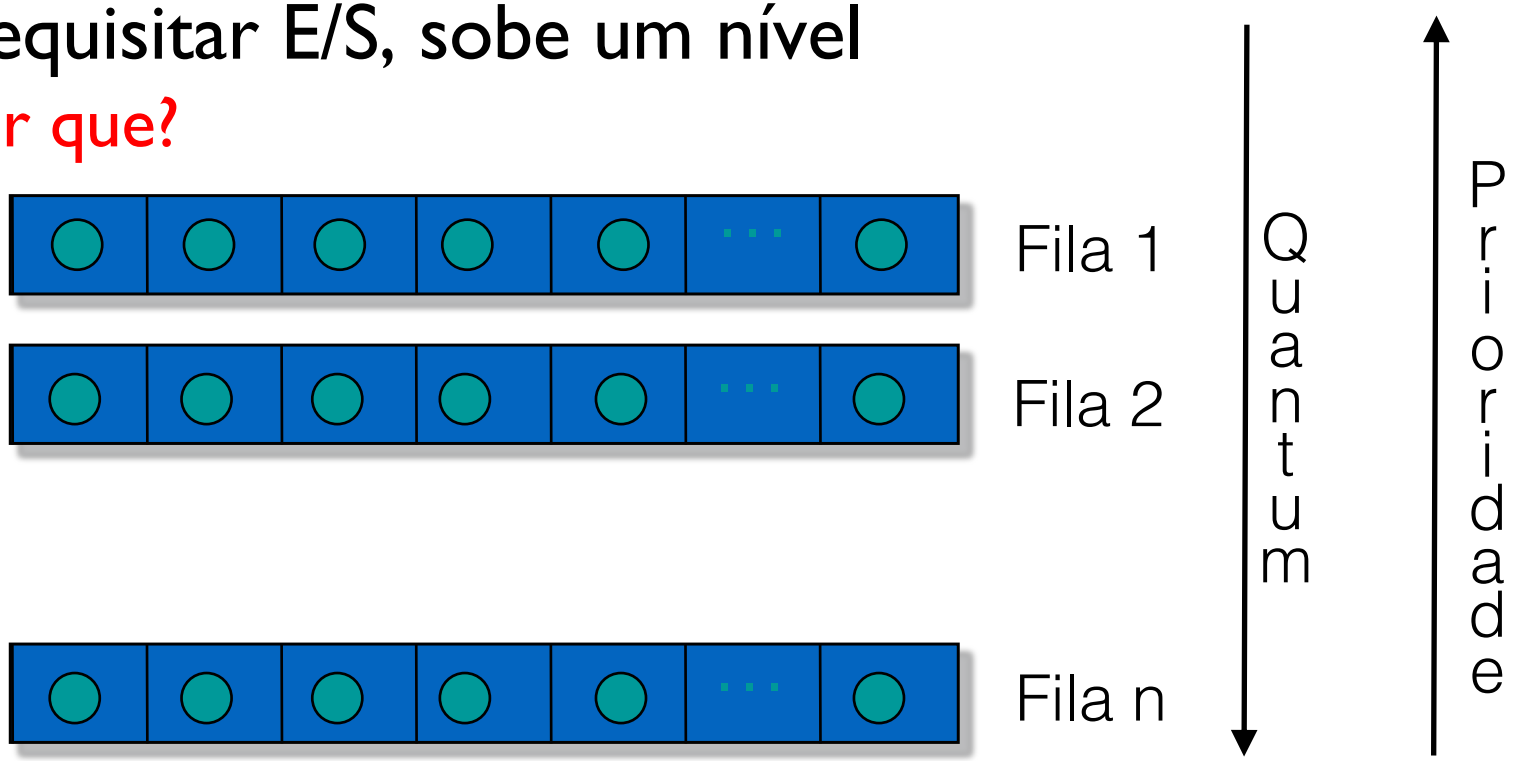
- Como saber a priori se o processo é CPU-bound ou I/O-bound?
- MFQ usa abordagem de prioridades dinâmicas
- Adaptação baseada no comportamento de cada processo
- Usado no VAX / VMS...

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?



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)

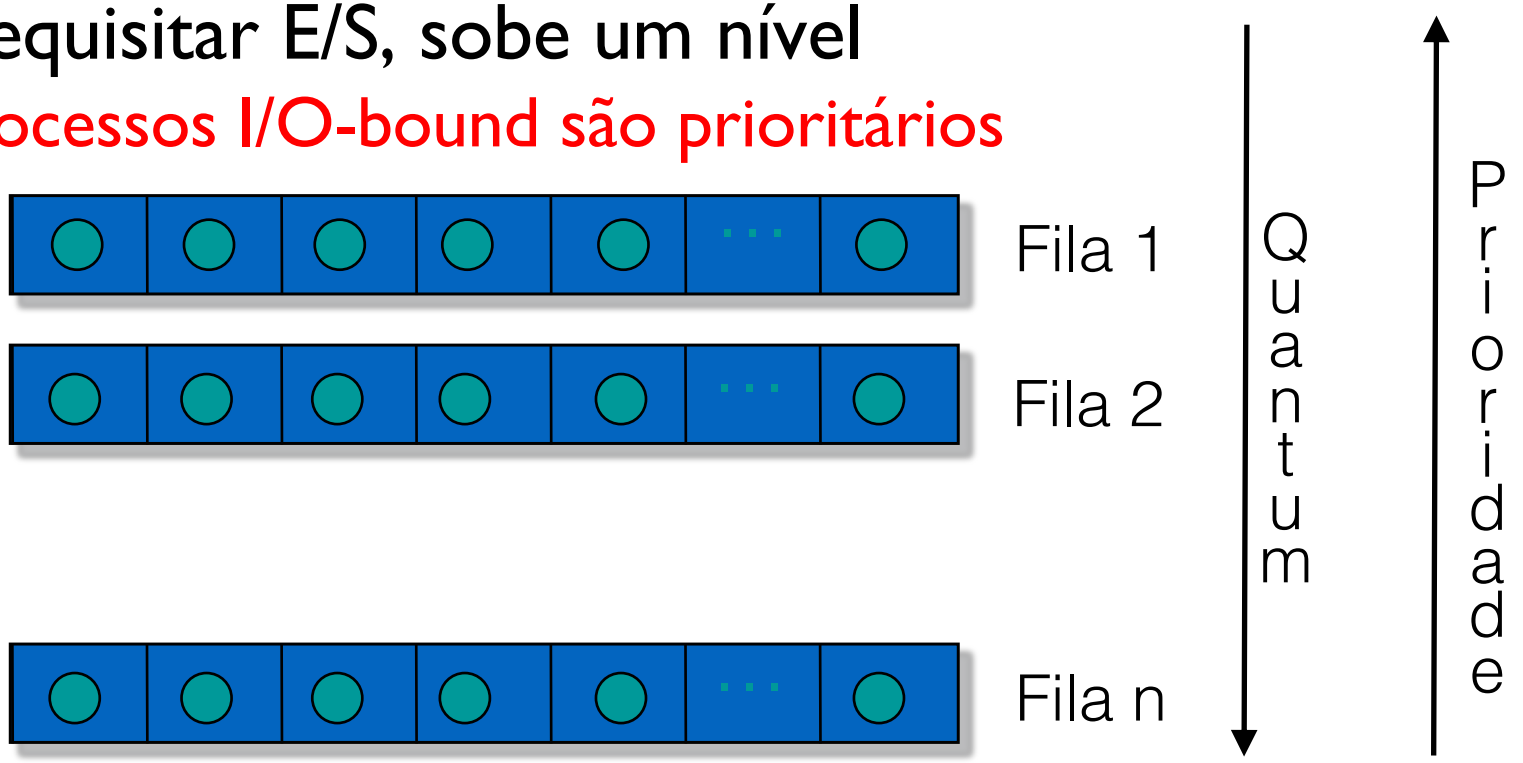
Em sistemas interativos

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

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



Fair-share Scheduling

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

Exemplo 1

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

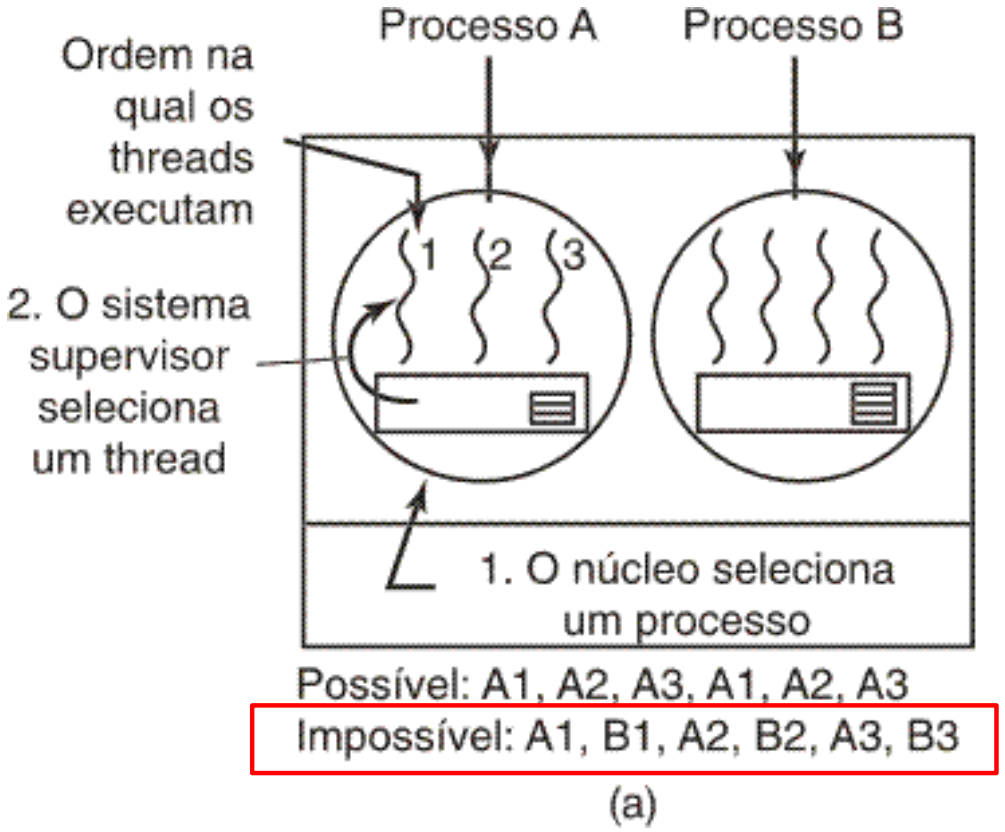
- Grupos 1, 2, 3:
  - 100% CPU / 3 grupos = 33.3% por grupo
  - Grupo 1: (33.3% / 3 usuários) = 11.1% por usuário
  - Grupo 2: (33.3% / 2 usuários) = 16.7% por usuário
  - Grupo 3: (33.3% / 4 usuários) = 8.3% por usuário

Algoritmos de Escalonamento

Round Robin	• Quantum constante • Sem prioridade
Com Prioridade	• 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	• Difícil estimar o tempo
Filas múltiplas	• Criação de classes de Prioridades alocadas em diferentes filas
Garantido	• Estimar (prometer) a um processo o tempo de sua execução e cumprir • Necessário conhecimento dos processos executando e a serem executados
Loteria	• Distribuição de bilhetes que dão acesso à CPU • Lembra o escalonamento com prioridade, mas bilhetes são trocados entre processos

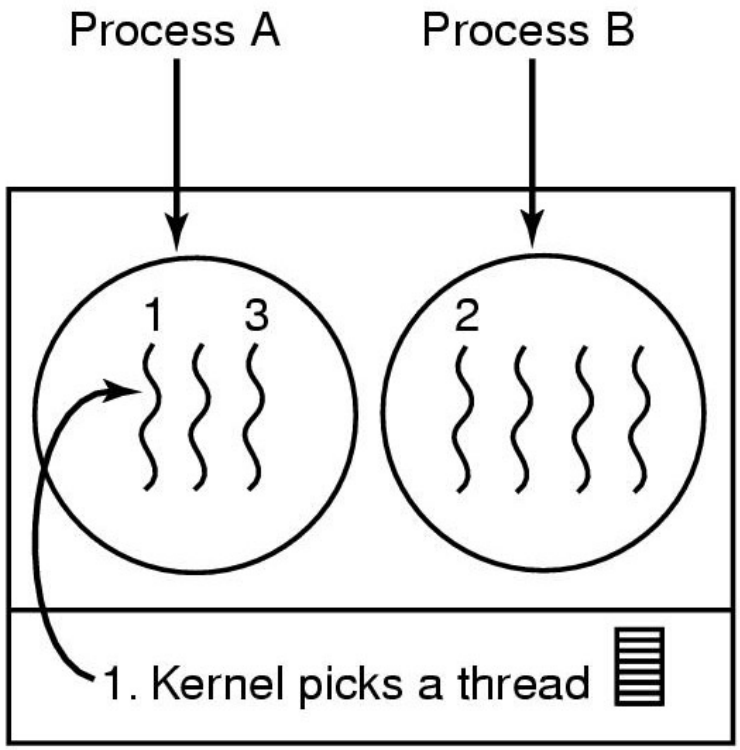
Fonte: Romildo Martins Bezerra / IFBA

Escalonamento de Threads (I)



- 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

POSIX Threads (Pthreads)

Modelo de execução de *threads* independente de linguagem, geralmente implementado como biblioteca
POSIX: Portable Operating System Interface

Obs.: boa parte já apresentada pela monitoria

CIn | UFPE

Próximas Atividades

Inversão de ordem

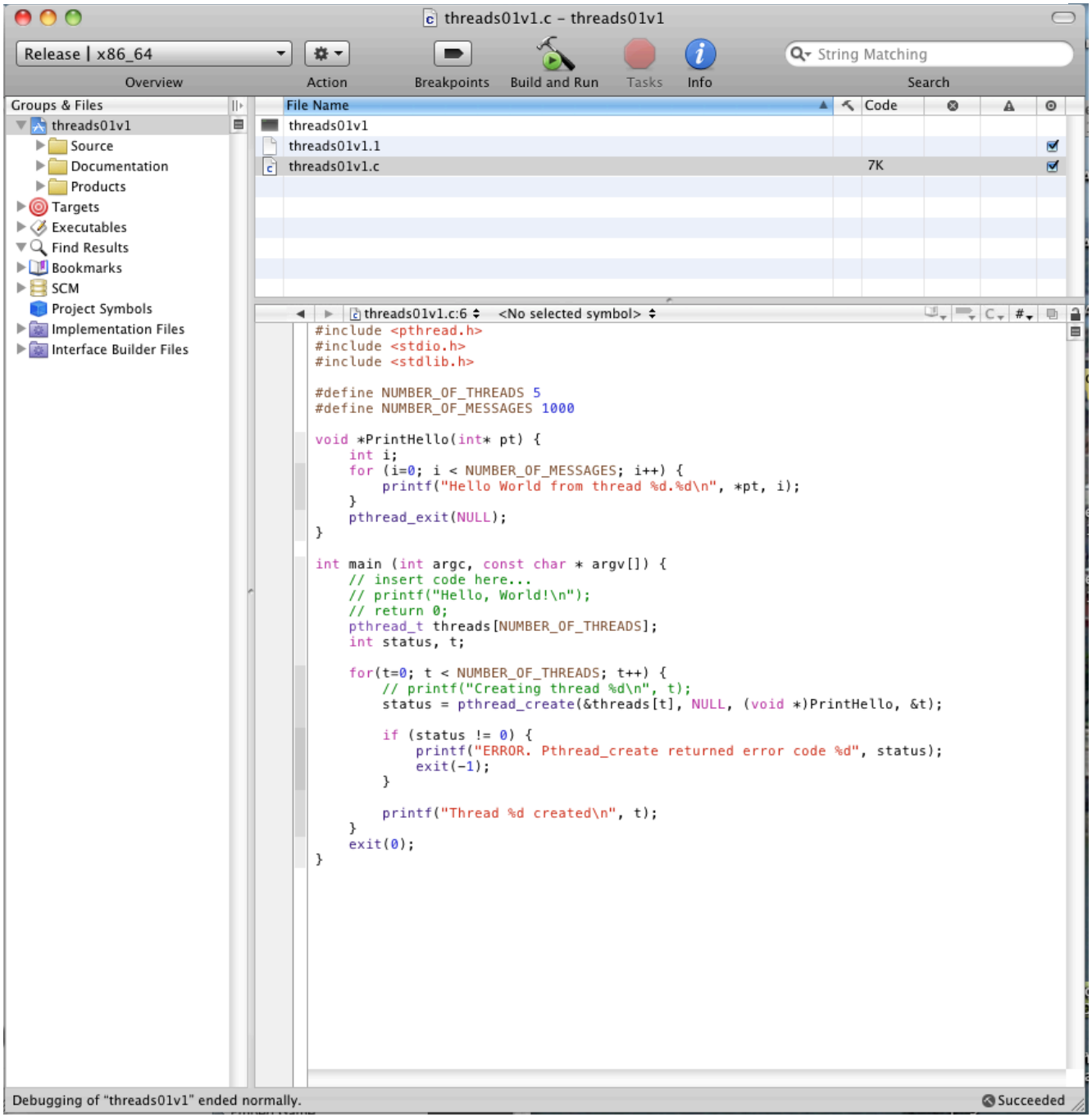
02/10/18	TERÇA	1o. EE, **D-002**
16/10/18	TERÇA	Exercício de concorrência, **LabG2**

POSIX Threads (1)

- Padrão IEEE POSIX 1003.1c (1995)
- Pthreads são um **conjunto de bibliotecas para a linguagem C**, por exemplo, que podem ser implementadas como uma biblioteca à parte ou parte da própria biblioteca C.
- Cerca de 60 subrotinas
- Algumas chamadas de funções Pthreads:

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

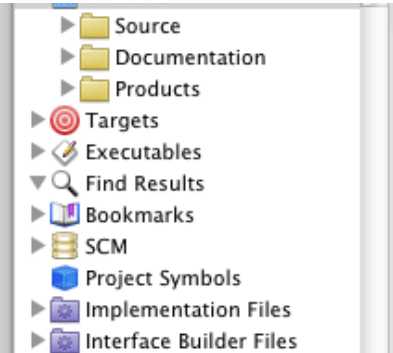
Hello World

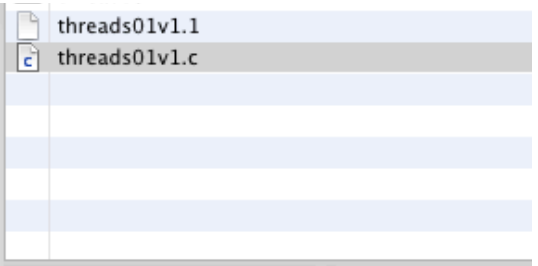


CIn | UFPE

```
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys000
Loading program into debugger…
Program loaded.
run
[Switching to process 5762]
Running…
Thread 0 created
Hello World from thread 0.0
Hello World from thread 1.1
Thread 1 created
Hello World from thread 1.2
Hello World from thread 1.0
Hello World from thread 2.3
Thread 2 created
Hello World from thread 2.0
Hello World from thread 2.1
Hello World from thread 2.4
Hello World from thread 3.1
Thread 3 created
Hello World from thread 3.0
Hello World from thread 3.2
Hello World from thread 3.5
Hello World from thread 3.2
Thread 4 created
Hello World from thread 4.0

Debugger stopped.
Program exited with status value:0.
[Session started at 2011-03-30 16:22:48 -0300.]
GNU gdb 6.3.50-20050815 (Apple version gdb-1515) (Sat Jan 15 08:33:48 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys001
Loading program into debugger…
Program loaded.
run
[Switching to process 5772]
Thread 0 created
Hello World from thread 0.0
Thread 1 created
Hello World from thread 1.1
Hello World from thread 1.0
Thread 2 created
Hello World from thread 2.2
Hello World from thread 2.0
Hello World from thread 2.1
Thread 3 created
Hello World from thread 3.3
Hello World from thread 3.0
Hello World from thread 3.1
Hello World from thread 3.2
Thread 4 created
Running…

Debugger stopped.
Program exited with status value:0.
```

Debugging of "threads01v1" ended normally. Succeeded





threads01v1.c:6 <No selected s

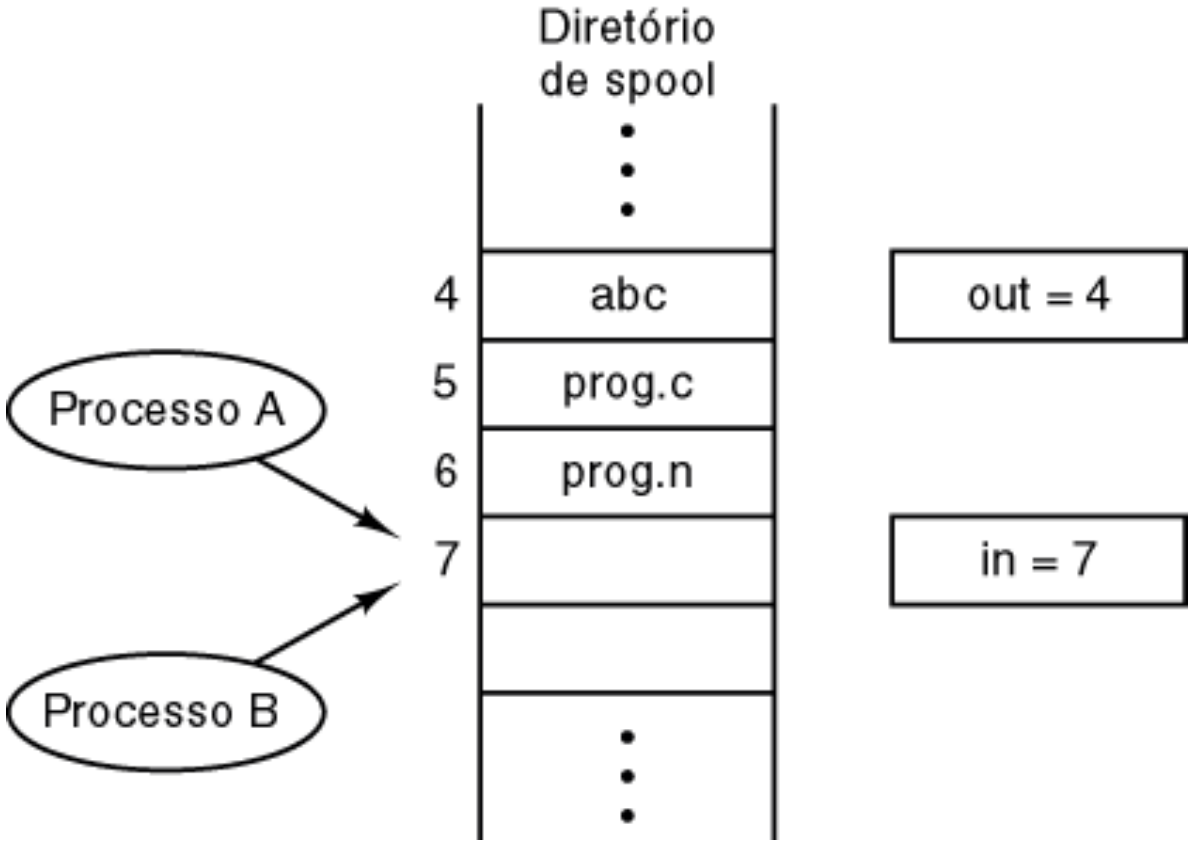
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS 5
#define NUMBER_OF_MESSAGES 1000

void *PrintHello(int* pt) {
    int i;
    for (i=0; i < NUMBER_OF_MESSA
        printf("Hello World from t
    }
    pthread_exit(NULL);
}

int main (int argc, const char * a
    // insert code here...
    // printf("Hello, World!\n");
    // return 0;
    pthread_t threads[NUMBER_OF_TH
    int status, t;

    for(t=0; t < NUMBER_OF_THREADS
        // printf("Creating threa
        status = pthread_create(&t

        if (status != 0) {
            printf("ERROR. Pthrea
            exit(-1);
        }

        printf("Thread %d created
    }
    exit(0);
}
```

Debugging of "threads01v1" ended normally.

Concorrência

Conceitos (Continuação)

Condições de Disputa/Corrida

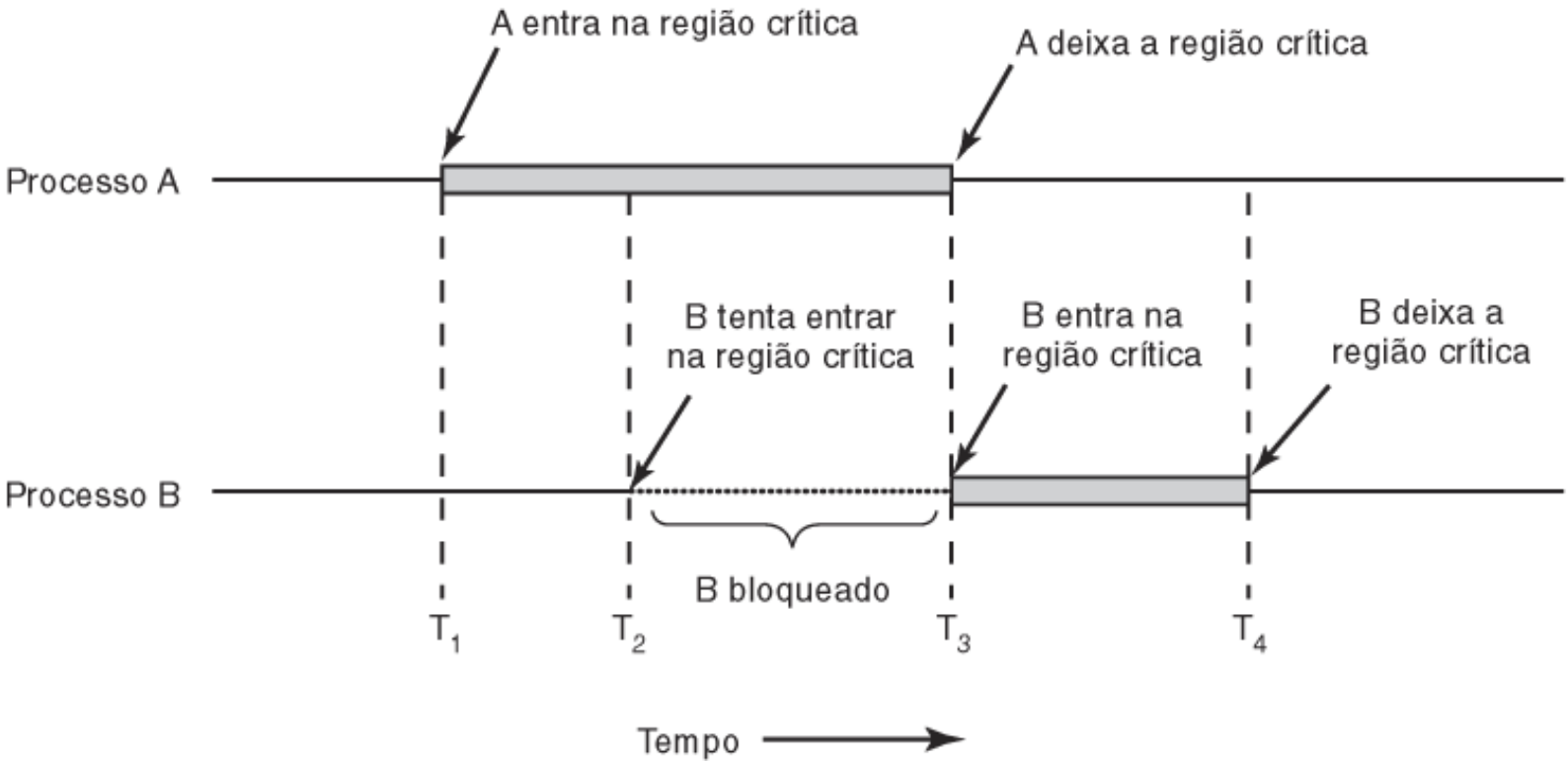


Dois processos querem ter acesso simultaneamente à memória compartilhada

Conceitos: Regiões Críticas (1)

- 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.

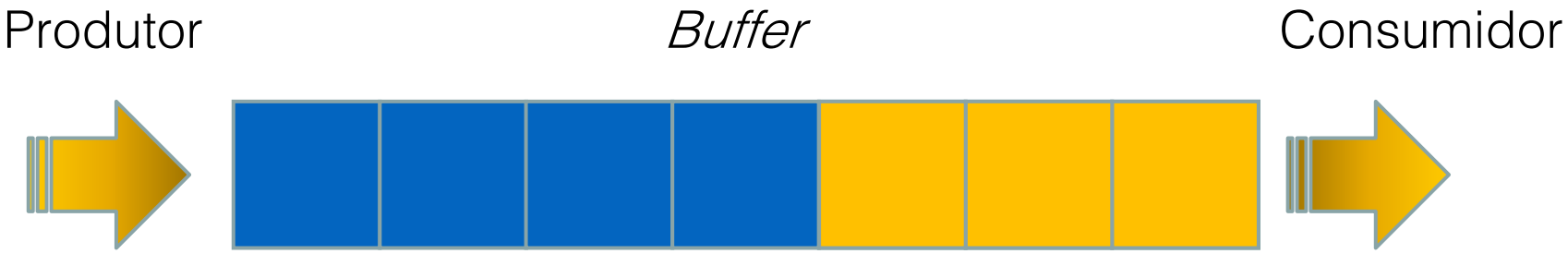
Centro de Informática U·F·P·E

Exclusão Mútua com Espera Ociosa (2)

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* número de processos */

int turn;                               /* 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 */
      turn = process;                   /* altera o valor de turn */
      while (turn == 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) {                              /* repita para sempre */
          if (count == 0) sleep( );              /* se o buffer estiver vazio, vá dormir */
          item = remove_item( );                 /* retire o item do buffer */
          count = count – 1;                     /* decresça de um o contador de itens no buffer */
          if (count == N – 1) wakeup(producer);  /* o buffer estava cheio? */
          consume_item(item);                    /* imprima o item */
     }
}
```

Problema do produtor-consumidor com uma condição de disputa fatal

Região Crítica e Exclusão Mútua

Conceitos Associados

CIn | UFPE

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                                   /* número de lugares no buffer */
typedef int semaphore;                          /* semáforos são um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso à região crítica */
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);
      }
}
```

```
#define N 100                                   /* número de
int count = 0;                                  /* número de

void producer(void)
{
      int item;

      while (TRUE) {
            item = produ
            if (count ==
            insert_item(
            count = cou
            if (count ==
      }
}
```

Sem mutex (semáforo): não evita condições de disputa

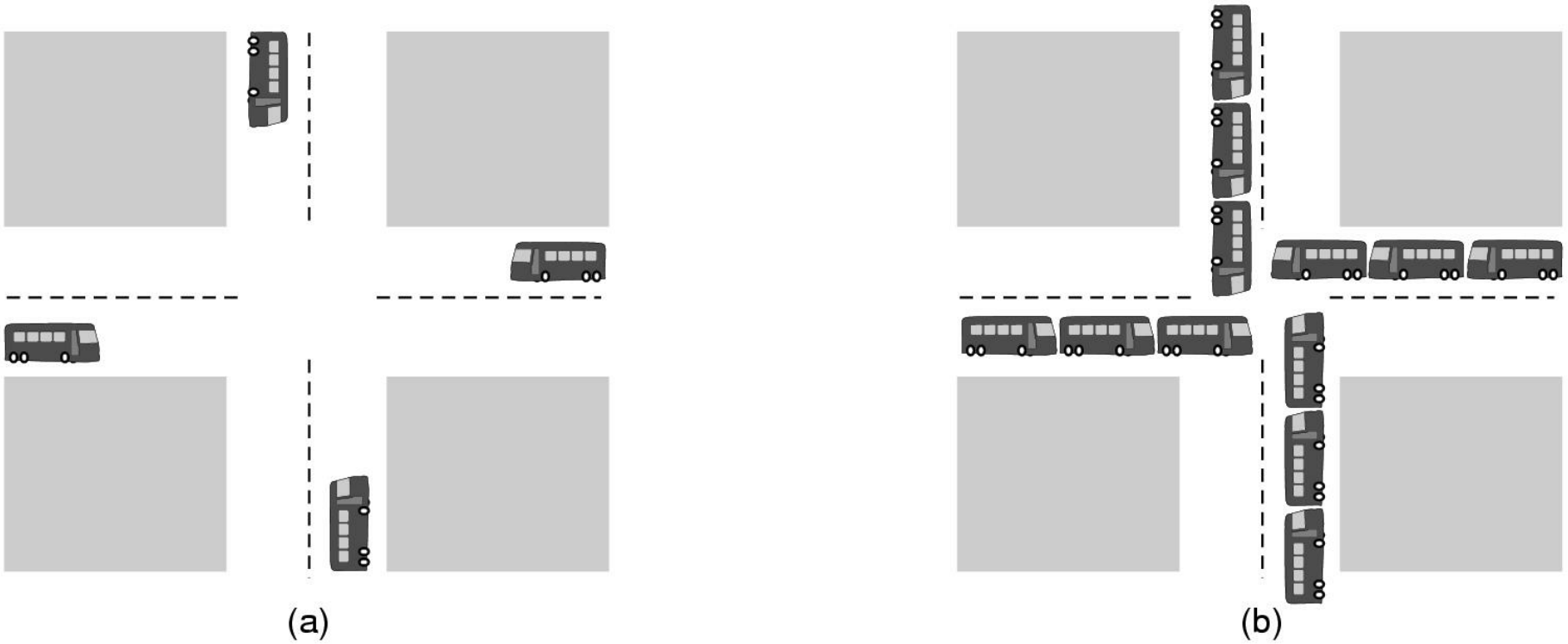
O problema do produtor-consumidor usando semáforos

Semáforos (2)

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

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

O problema do produtor-consumidor usando semáforos

Exemplo da necessidade de Semáforo



(a) Um *deadlock* potencial.

(b) Um *deadlock* real.

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;
```

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

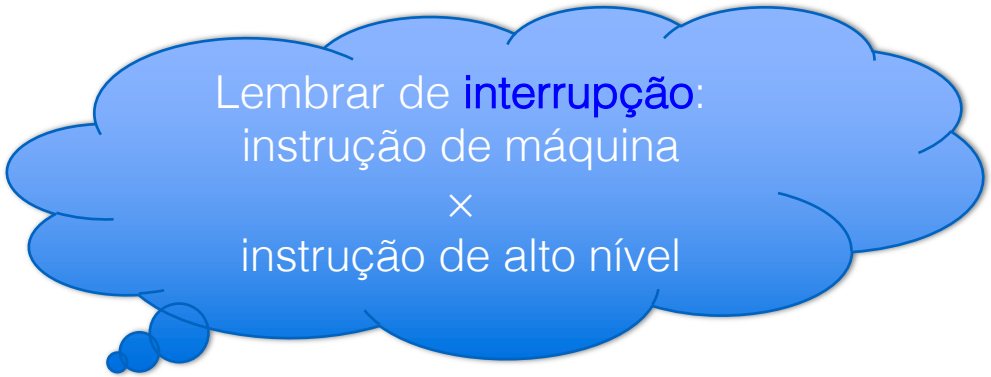
- somente um procedimento está ativo por vez no monitor
- o buffer tem N lugares

CIn | UFPE

Exclusão Mútua com Pthreads

FERNANDO CASTOR E
HTTPS://COMPUTING.LLNL.GOV/TUTORIALS/PTHREADS/

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);
}
```

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**)
  - 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

```c
#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);
}
```

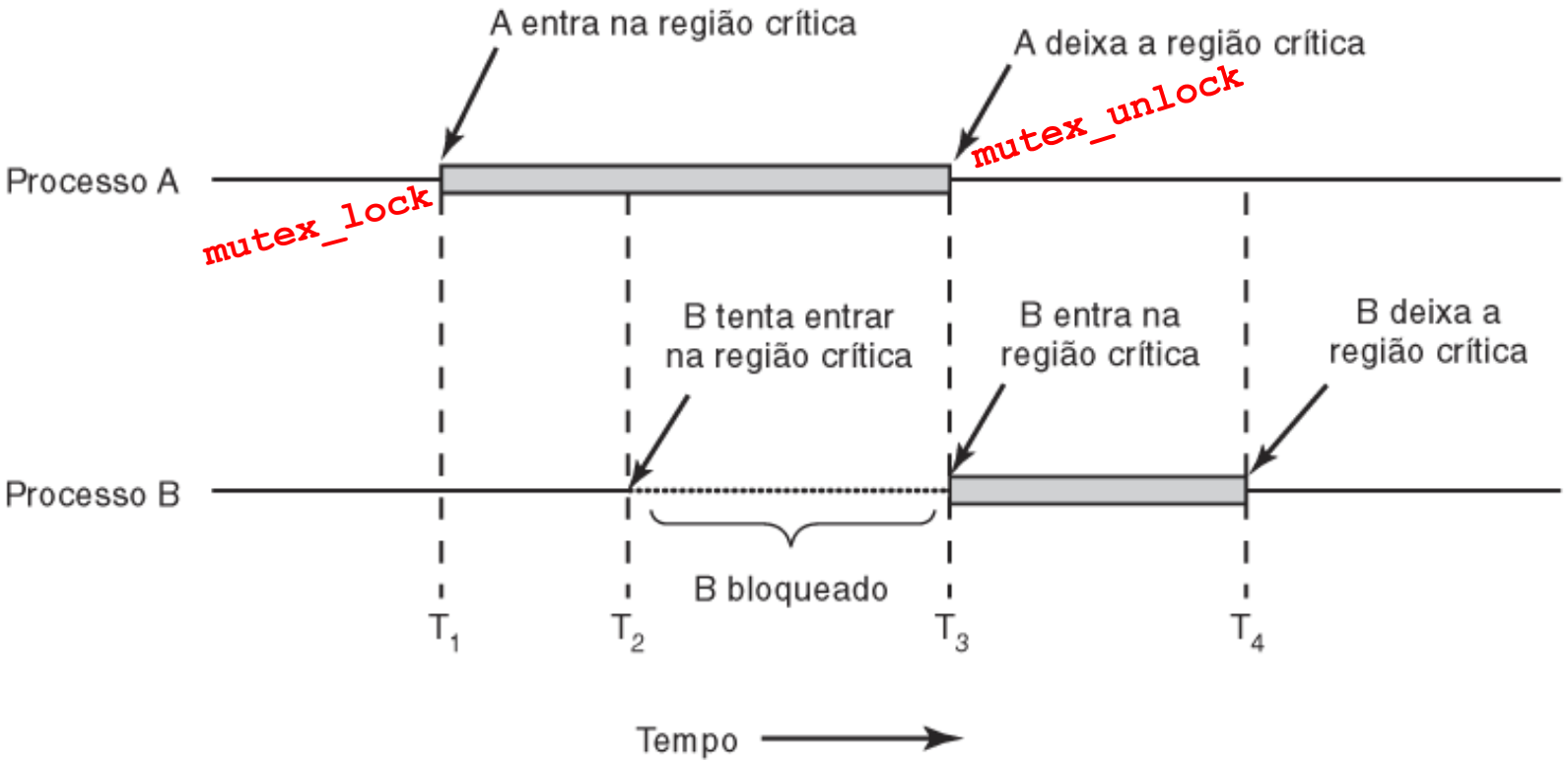
```c
#include <pthread.h>
#include <stdio.h>
long contador = 0;

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, NU
  pthread_create(&thread2, NULL, dec, NU
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  printf("Valor final do contador: %ld\n
  pthread_exit(NULL);
}
```

CIn | UFPE

Vistos em aula de monitoria

Concorrência

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



Vistos em aula de monitoria

Concorrência

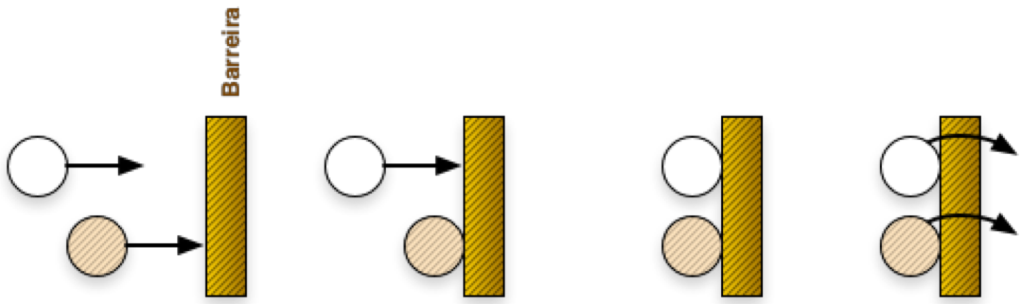
- **Mutexes**: garantem acesso exclusivo à uma região crítica
- **Variáveis Condicionais**: permitem que uma *thread* continue após determinada condição

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

Vistos em aula de monitoria

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

CIn | UFPE

Próximas Atividades

02/10/18	TERÇA	1o. EE, **D-002**
16/10/18	TERÇA	Exercício de concorrência, **LabG2**

Exclusão mútua pode não ser o bastante

- 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_thr
   ead,NULL,consumer,NULL);

   pthread_create(&producer_thr
   ead,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

CIn | UFPE

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; 7/ 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);
}
```

Exercício (opcional)

Escalonamento de Processos

Exercício: Escalonamento de Processos

(Individual ou em dupla, opcional :: vale ponto)

Cinco *jobs* de lote, *A* até *E*, chegam a um centro de computação quase ao mesmo tempo. Eles têm tempos de execução estimados de 10, 6, 2, 4 e 8 minutos. Suas prioridades (externamente determinadas) são 3, 5, 2, 1 e 4, respectivamente, com 5 sendo a maior prioridade.

Para cada um dos seguintes algoritmos de escalonamento, determine o tempo médio de retorno (*average turnaround time*) dos processos. Ignore a sobrecarga (*overhead*) da alternância de processos (troca de contexto).

a) Round robin
b) Escalonamento por prioridade
c) Primeiro a chegar, primeiro a ser servido (FCFS/FIFO)
d) *Job* mais curto primeiro (SJF).

Para (a), suponha que o sistema é multiprogramado e que cada *job* receba sua justa parte da CPU (escalonamento preemptivo) – para agilizar considere um *quantum* de 1 minuto. Para (b) a (d) suponha que só um *job* execute por vez até terminar. Todos os *jobs* são completamente orientados a CPU (*CPU-bound*).

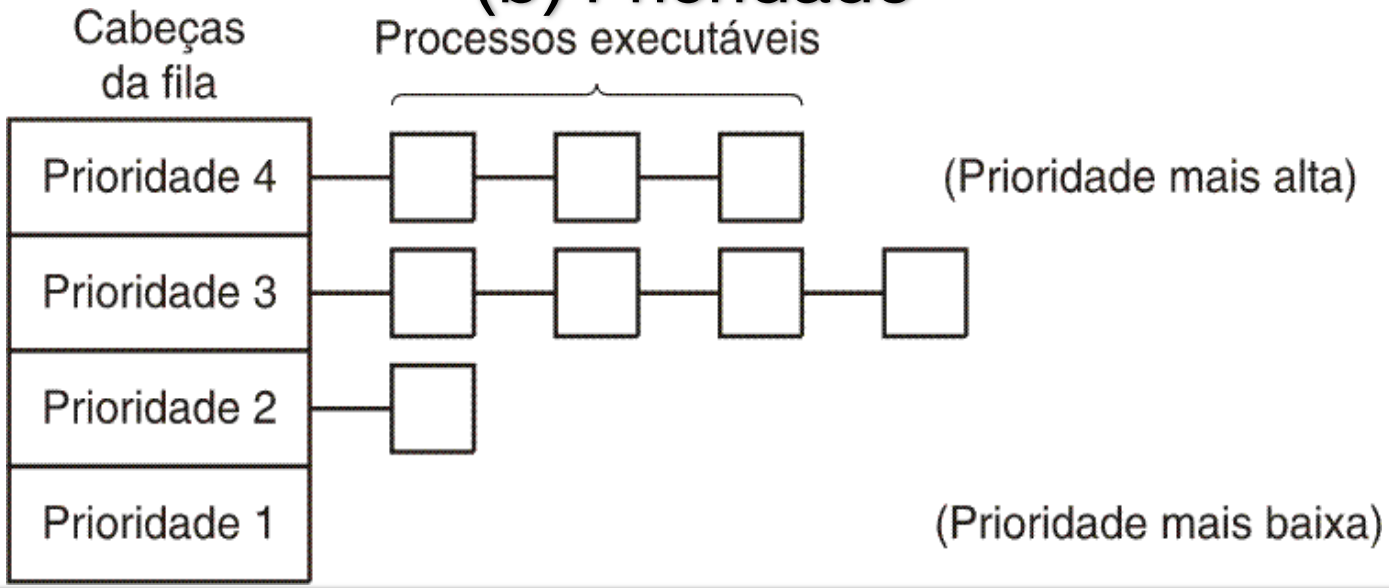
Cálculos

- Tempo de retorno(p) = t(p)$_{\text{término}}$ – t(p)$_{\text{submissão}}$
- Tempo médio de retorno = $\dfrac{\sum_{i=1}^{n} \text{Tempo de retorno}(p_i)}{n}$

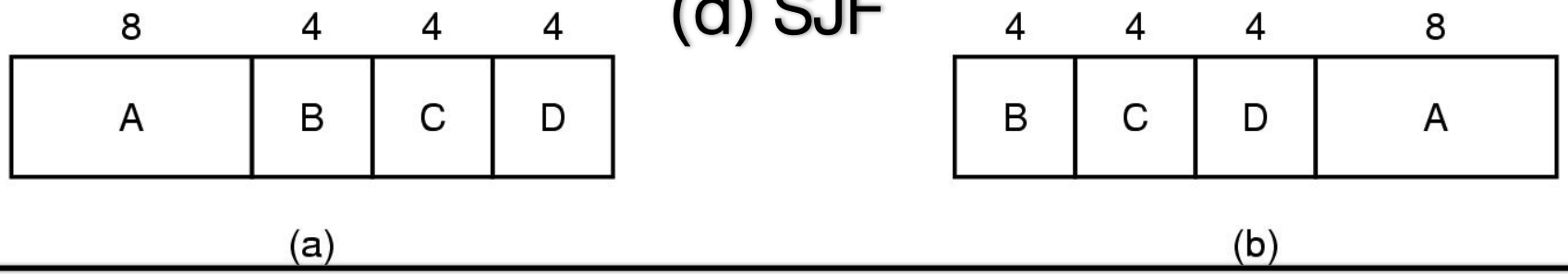
Onde:
p = processo
n = nº. de processos

(a) Round-robin

(b) Prioridade

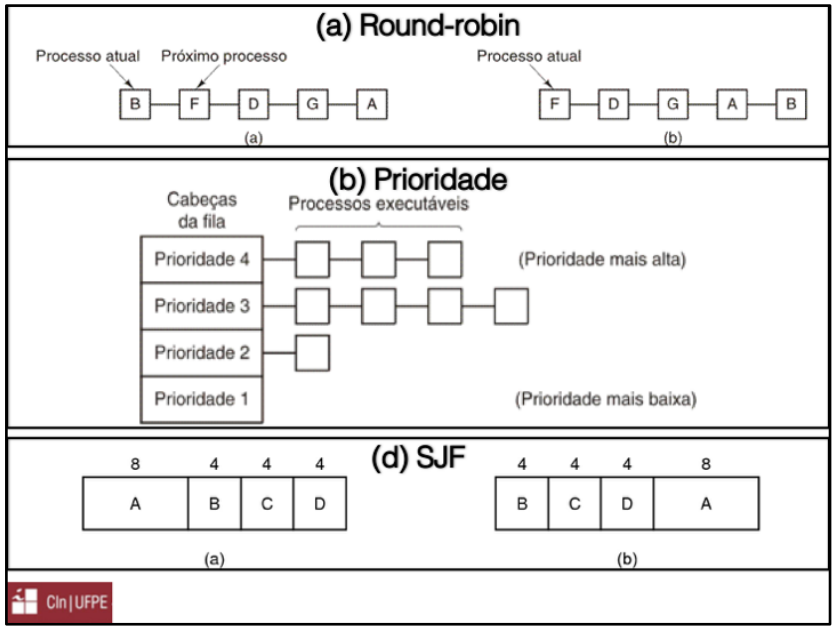


(d) SJF



Exercício: Escalonamento

(Individual ou em dupla, opcional :: vale ponto)



Cinco *jobs* de lote, *A* até *E*, chegam a um centro de computação quase ao mesmo tempo. Eles têm tempos de execução estimados de 10, 6, 2, 4 e 8 minutos. Suas prioridades (externamente determinadas) são 3, 5, 2, 1 e 4, respectivamente, com 5 sendo a maior prioridade.

Para cada um dos seguintes algoritmos de escalonamento, determine o **tempo médio de retorno** (*average turnaround time*) dos processos. Ignore a sobrecarga (*overhead*) da alternância de processos (troca de contexto).

a) Round robin (40%)

~~b) Escalonamento por prioridade~~

c) Primeiro a chegar, primeiro a ser servido (FCFS/FIFO) (30%)

d) *Job* mais curto primeiro (SJF) (30%).

Para (a), suponha que o sistema é multiprogramado e que cada *job* receba sua justa parte da CPU (escalonamento preemptivo) – **para agilizar considere um *quantum* de 1 minuto**. Para (b) a (d) suponha que só um *job* execute por vez até terminar. Todos os *jobs* são completamente orientados a CPU (*CPU-bound*).