

Infra-Estrutura de Software

Processos e Threads

SO: Multitarefa

Processo P_0

Sistema Operacional

Processo P_1

SO: Multitarefa

Processo P_0

Sistema Operacional

Processo P_1

executando ↓

⋮ ocioso

Processo P_0 em execução

SO: Multitarefa

Processo P_0

Sistema Operacional

Processo P_1

interrupção ou *system call*

executando



Salva estado no PCB_0

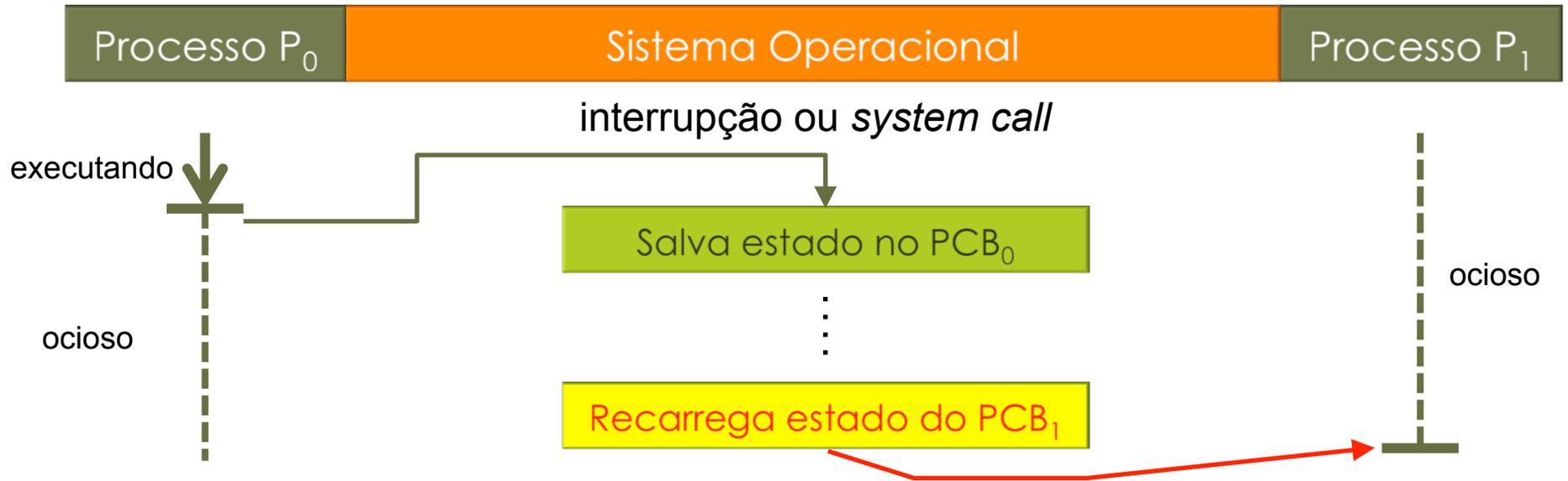


ocioso

Process Control Block: informações de contexto do processo

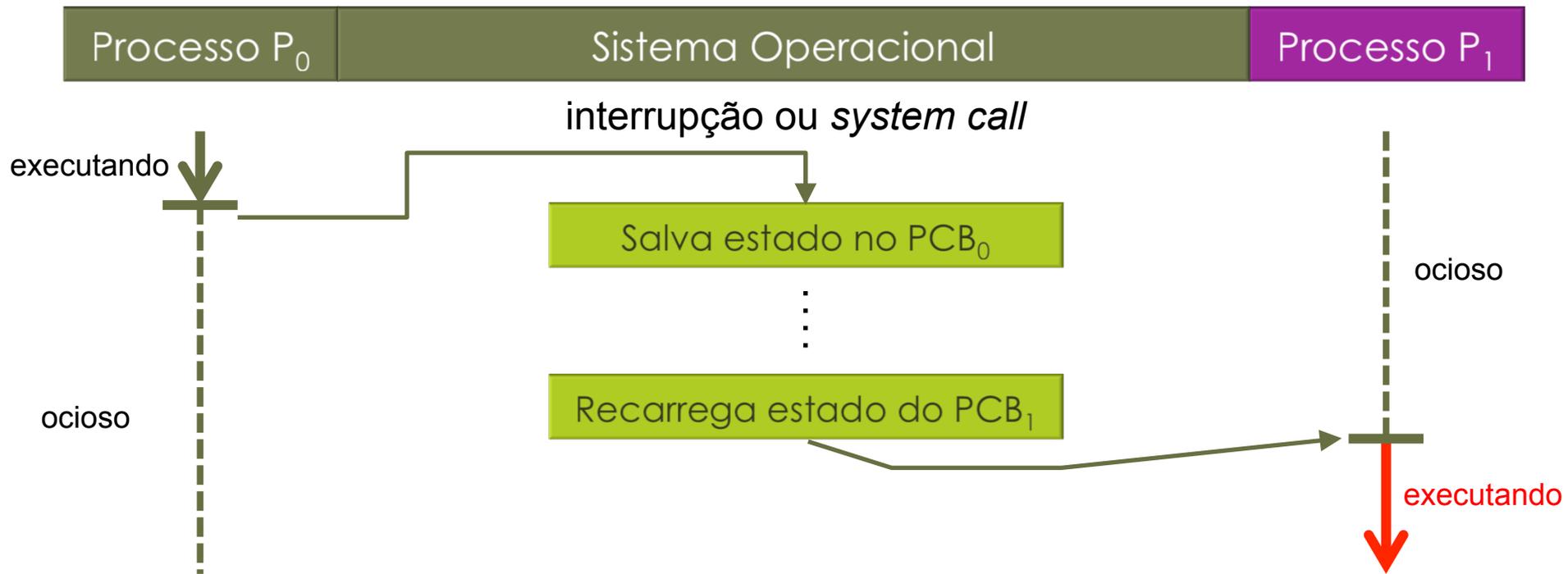
Salva estado do Processo P_0 no PCB_0

SO: Multitarefa



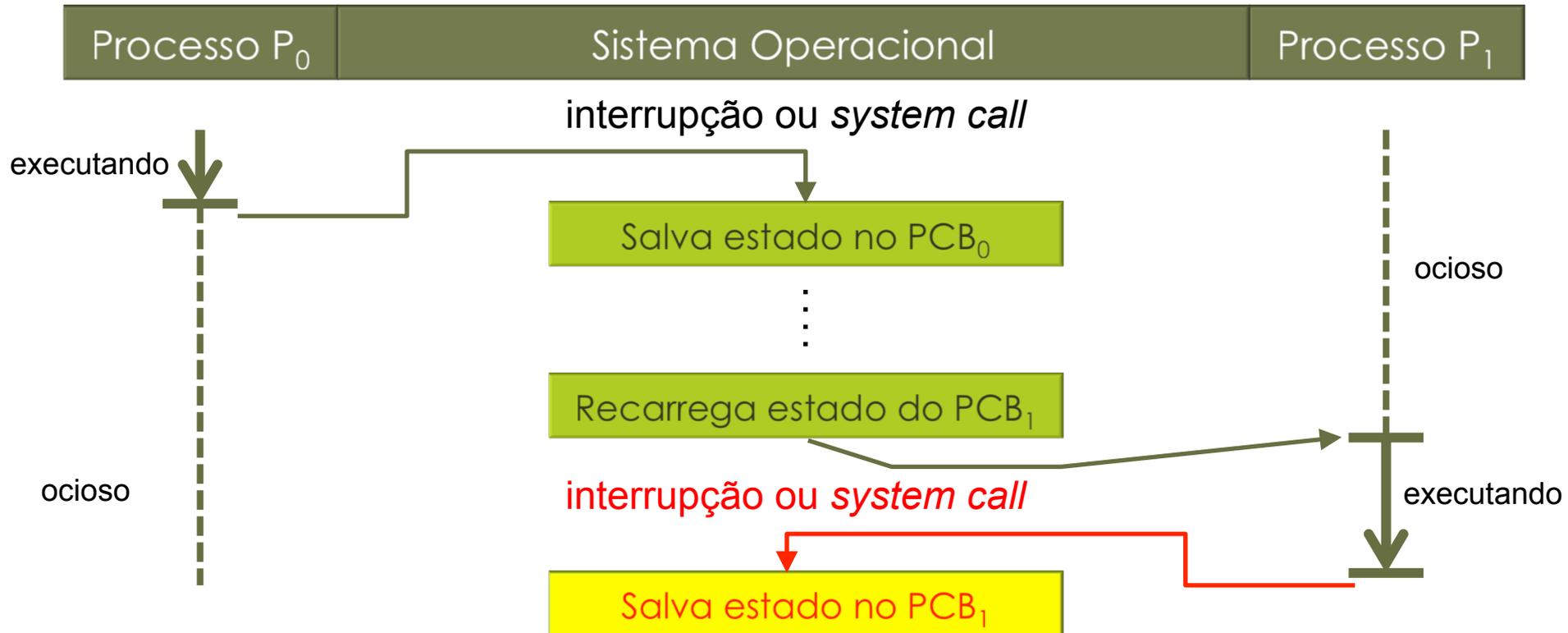
Estado do Processo P₁ é recarregado do PCB₁

SO: Multitarefa

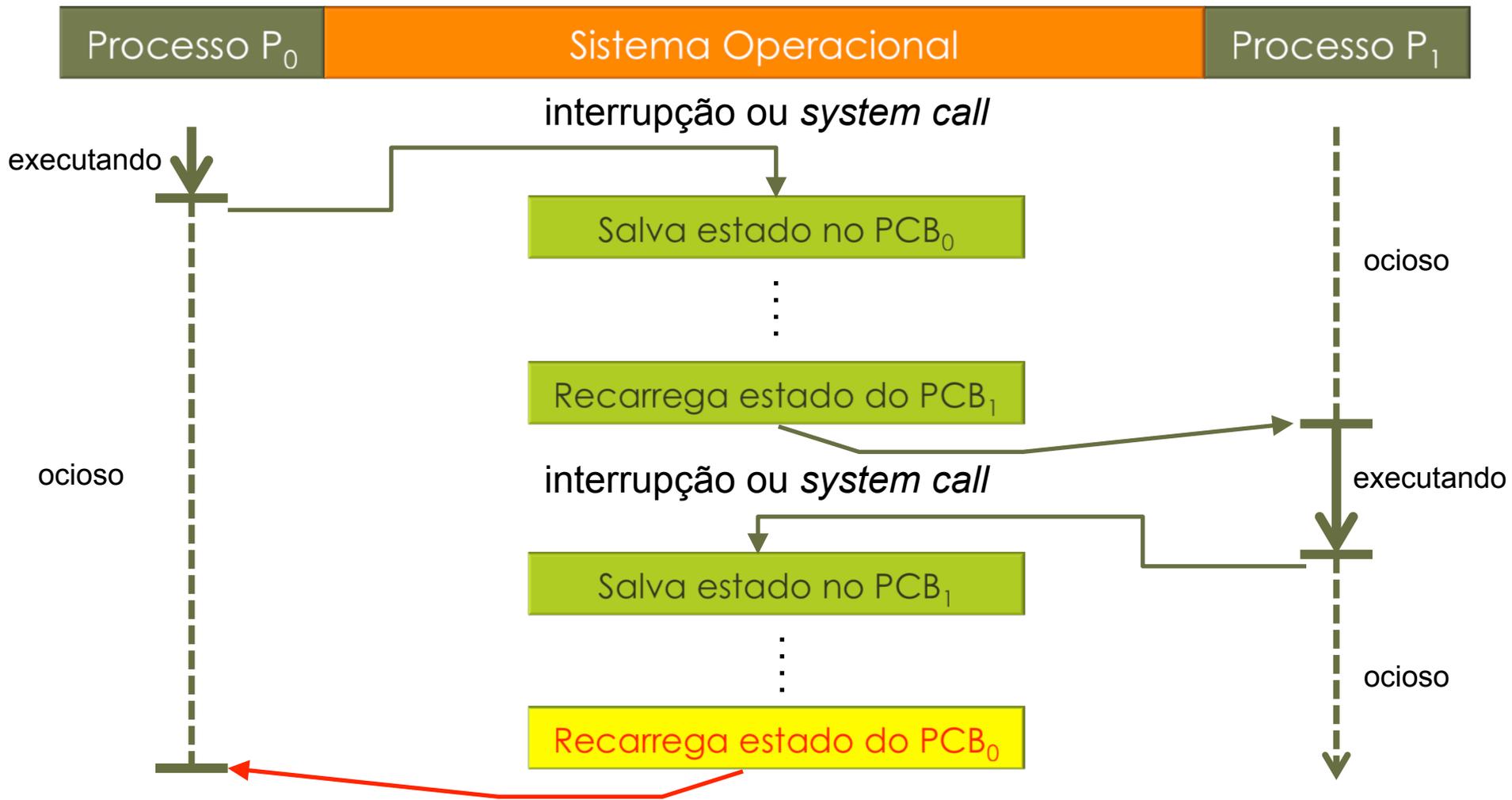


Processo P₁ em execução

SO: Multitarefa

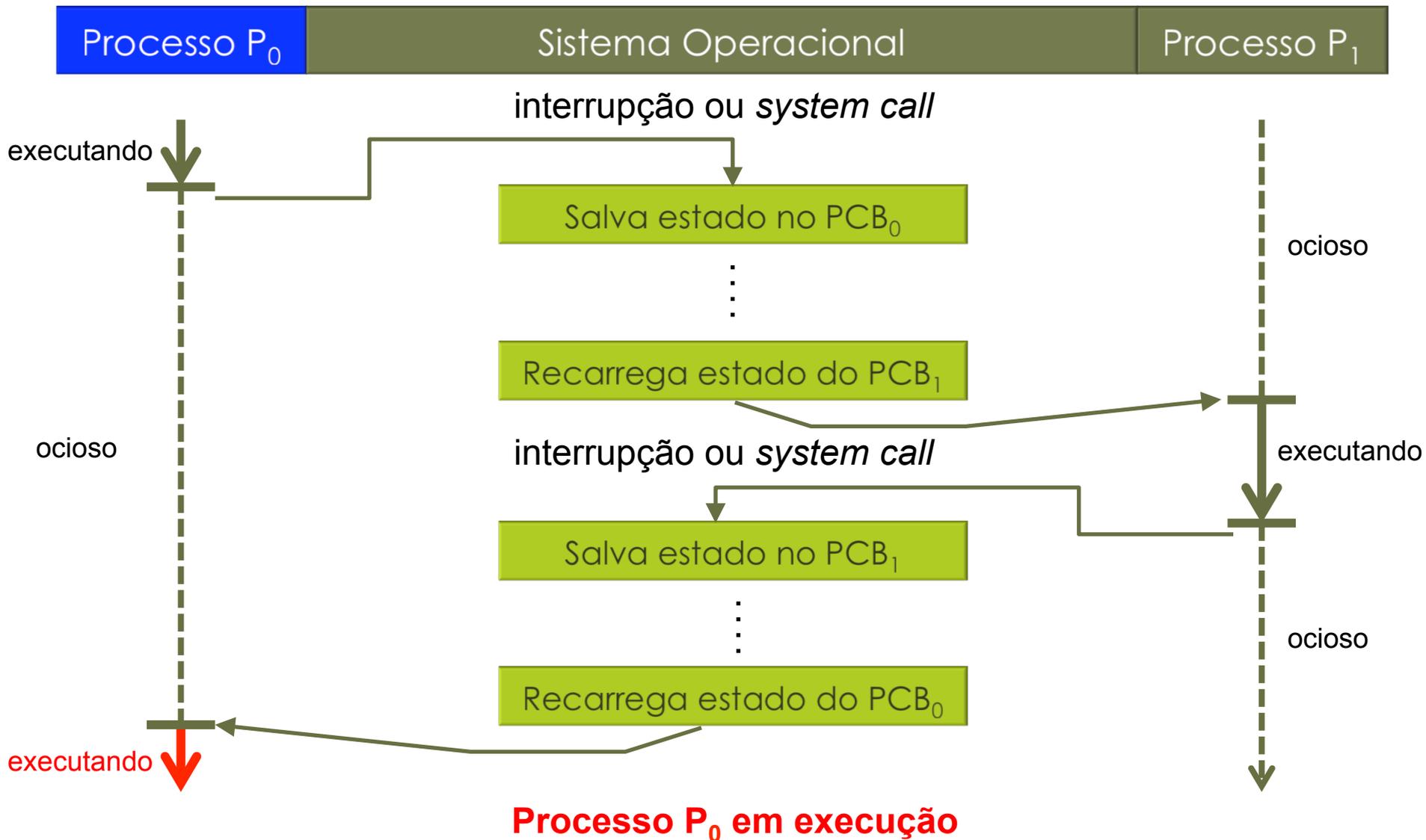


SO: Multitarefa



Estado do Processo P_0 é recarregado do PCB₀

SO: Multitarefa



Implementação de Processos

Gerenciamento de processos

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

Gerenciamento de memória

Ponteiro para o segmento de código
Ponteiro para o segmento de dados
Ponteiro para o segmento de pilha

Gerenciamento de arquivos

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)

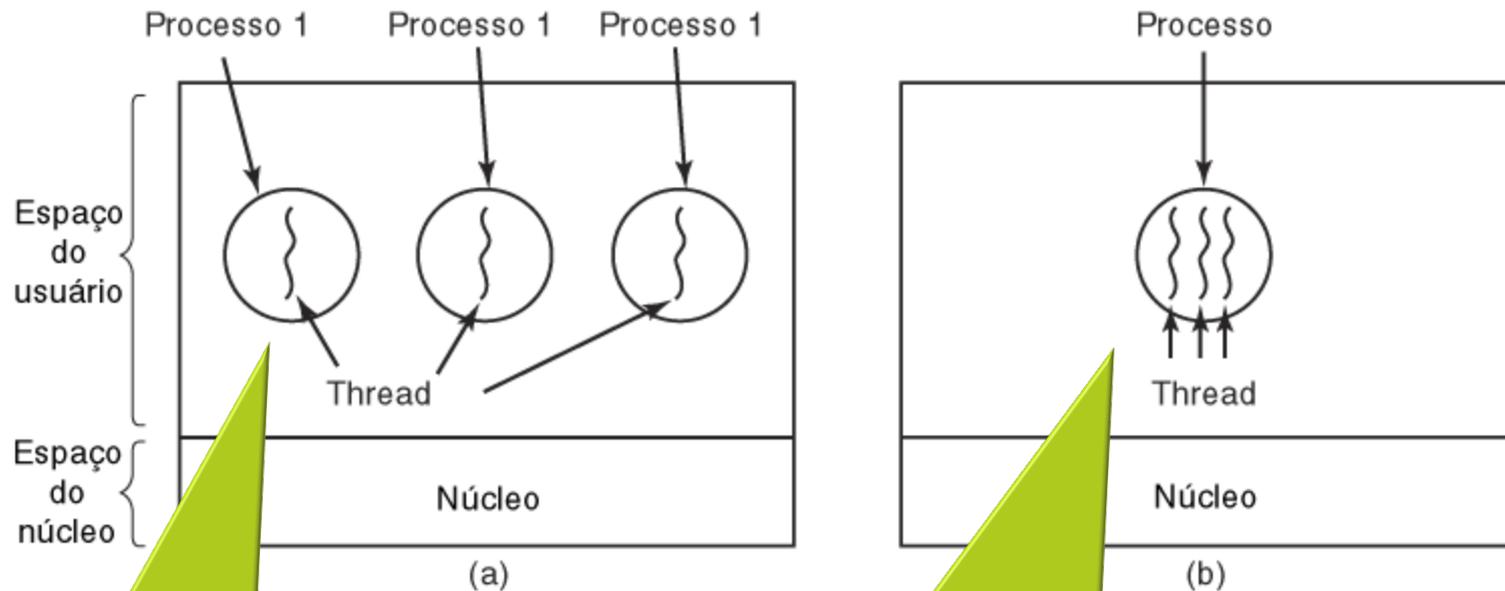
SIGSYS - Bad system call

SIGXCPU - CPU time limit exceeded

SIGXFSZ - File size limit exceeded

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

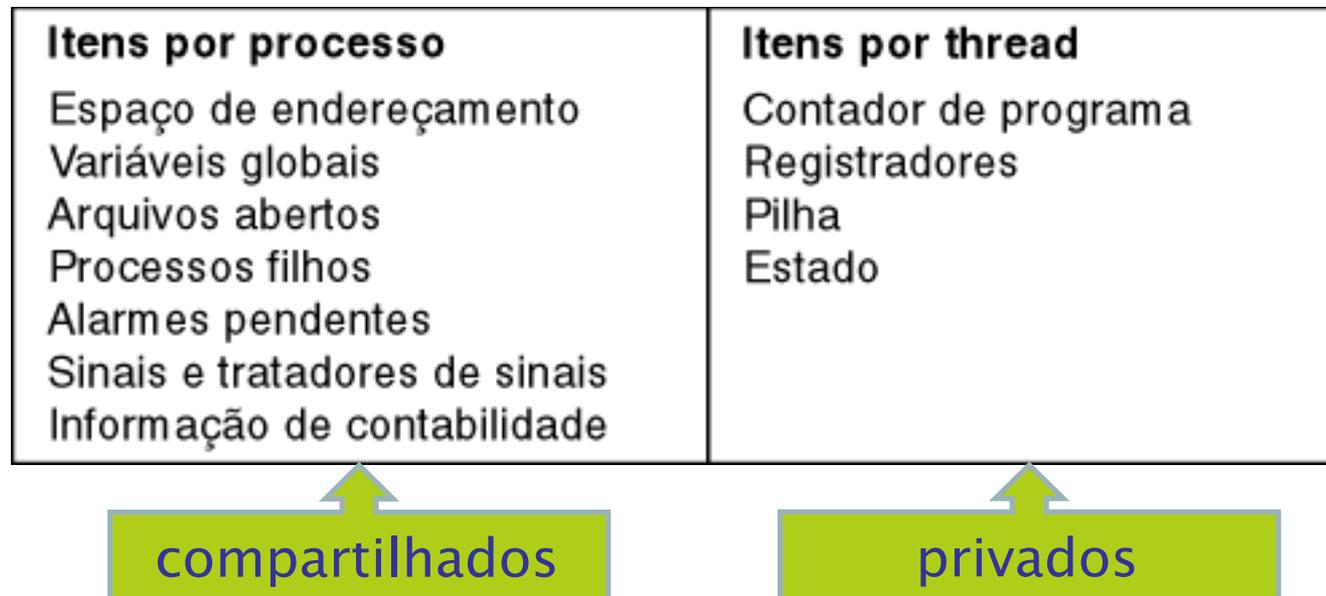
Threads: O Modelo (1)



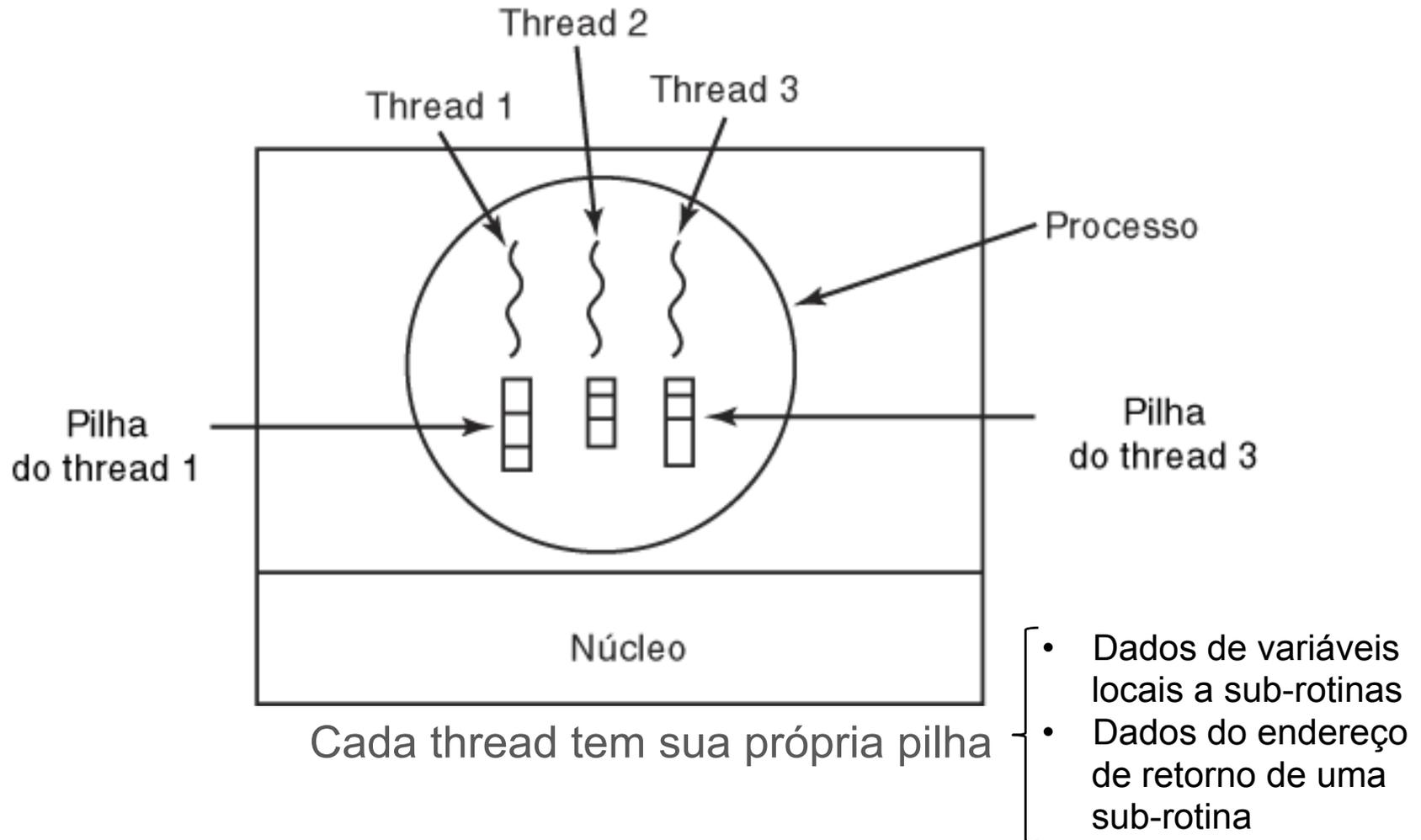
Cada processo tem um espaço de endereçamento e uma única linha (*thread*) de controle/execução

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*)

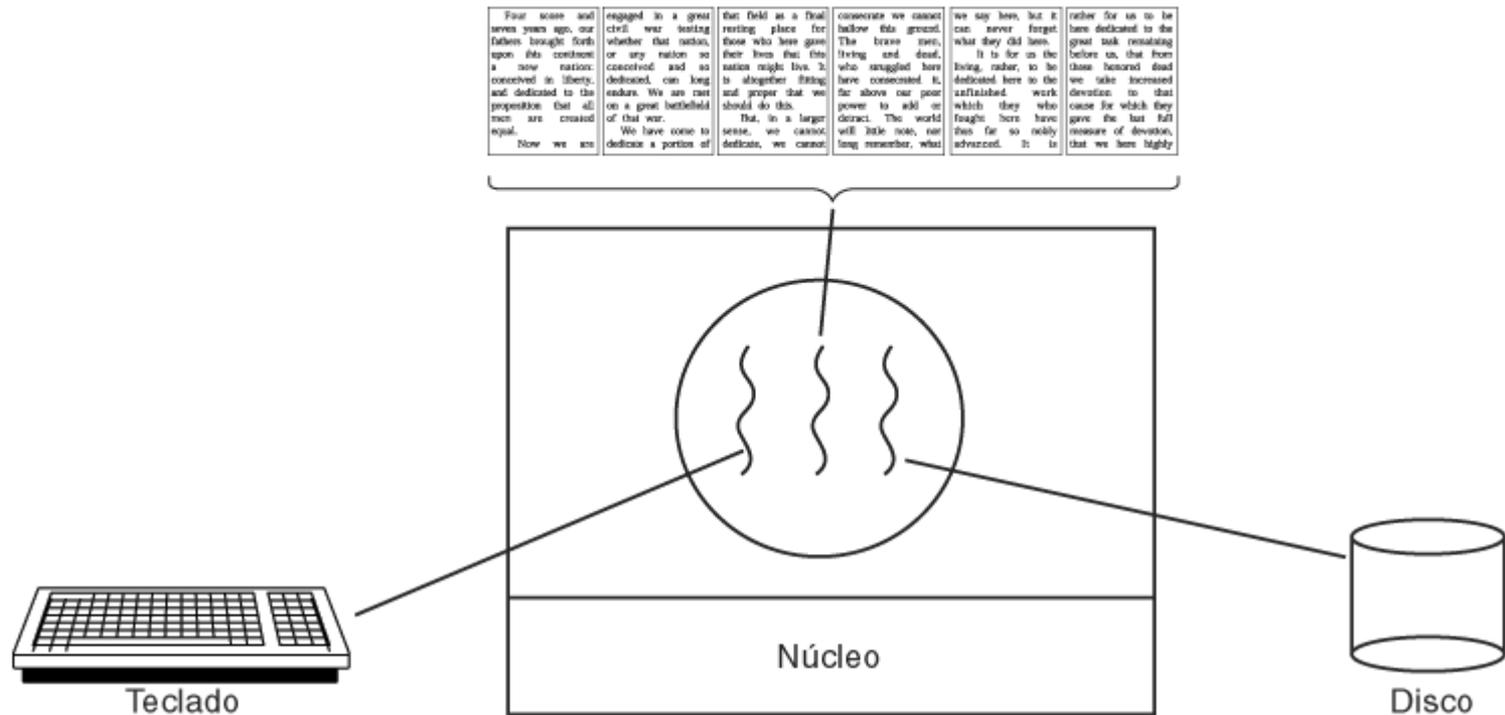
O Modelo de Thread (2)



O Modelo de Thread (3)

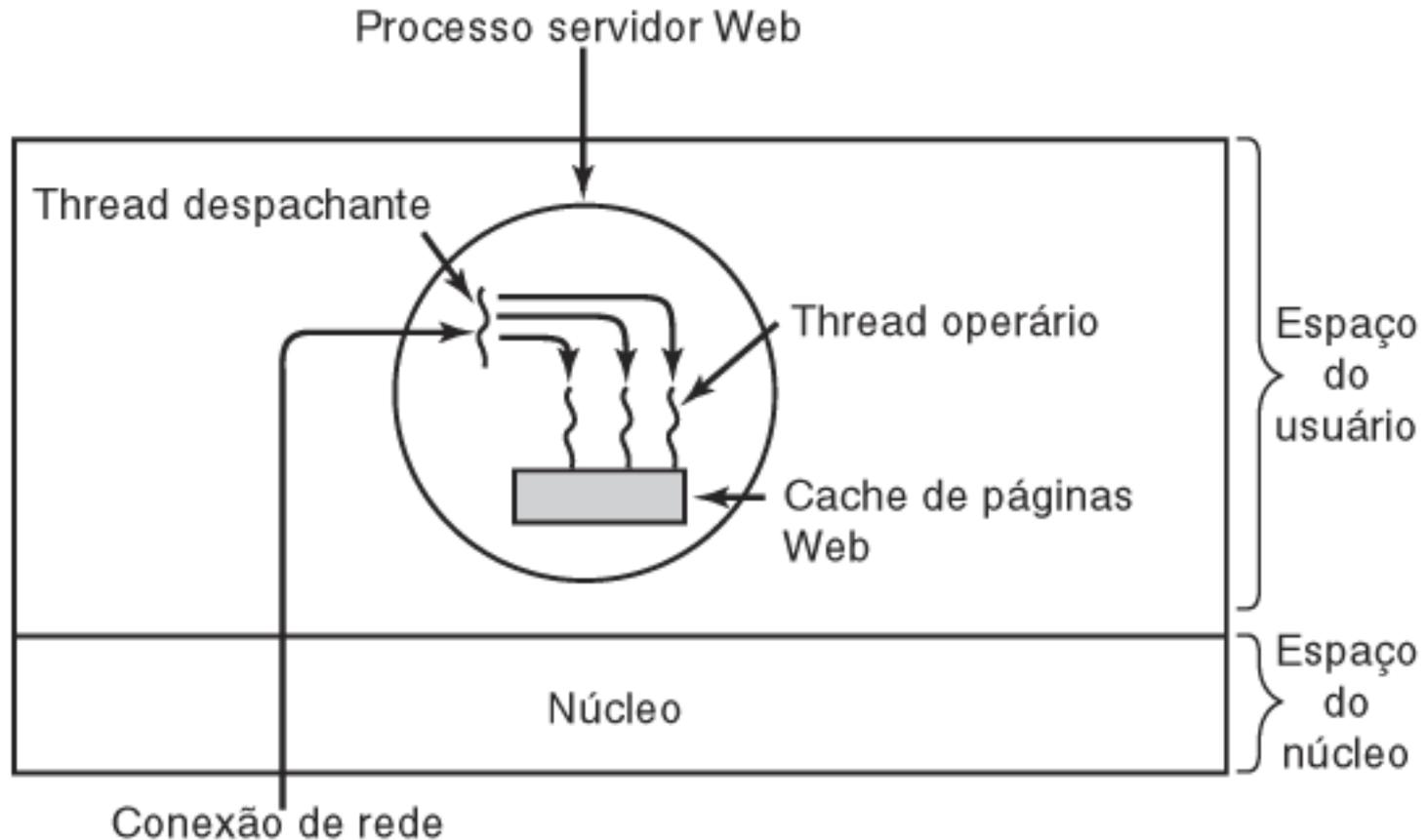


Uso de Thread (1)



Um processador de texto com três threads

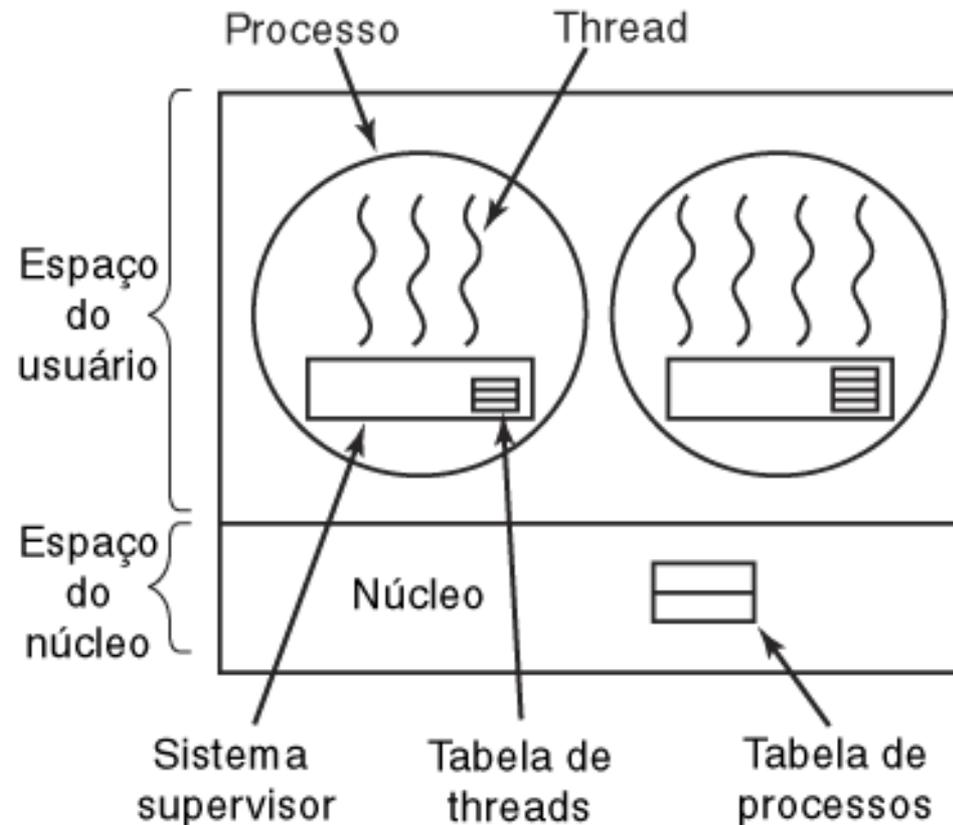
Uso de Thread (2)



Um servidor web com múltiplas threads

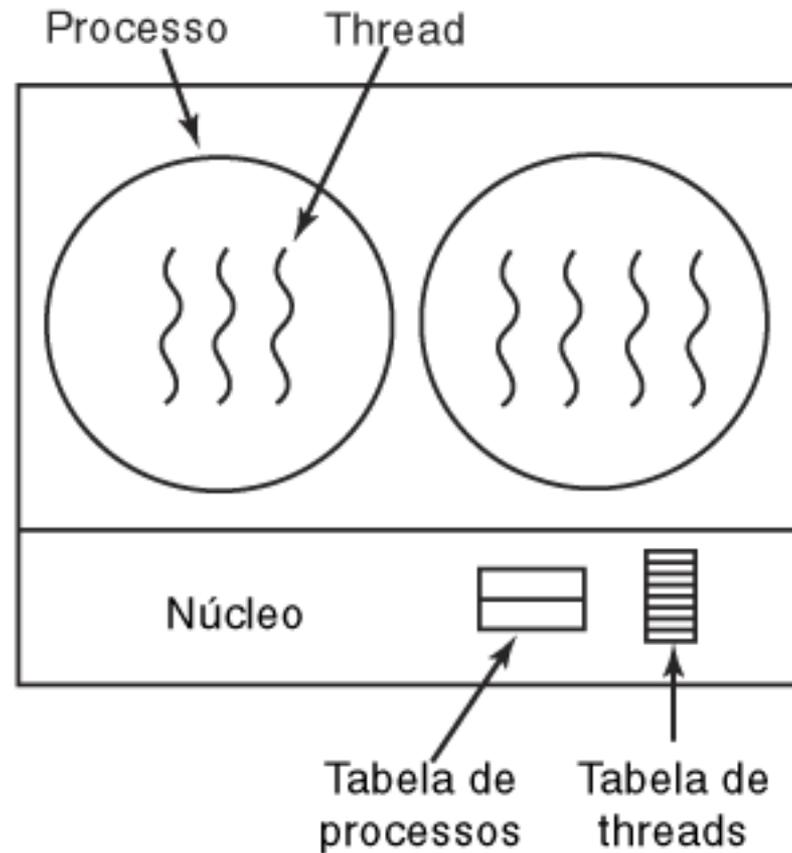
vs um serviço Web com múltiplos servidores (mais adiante)

Implementação de Threads de Usuário



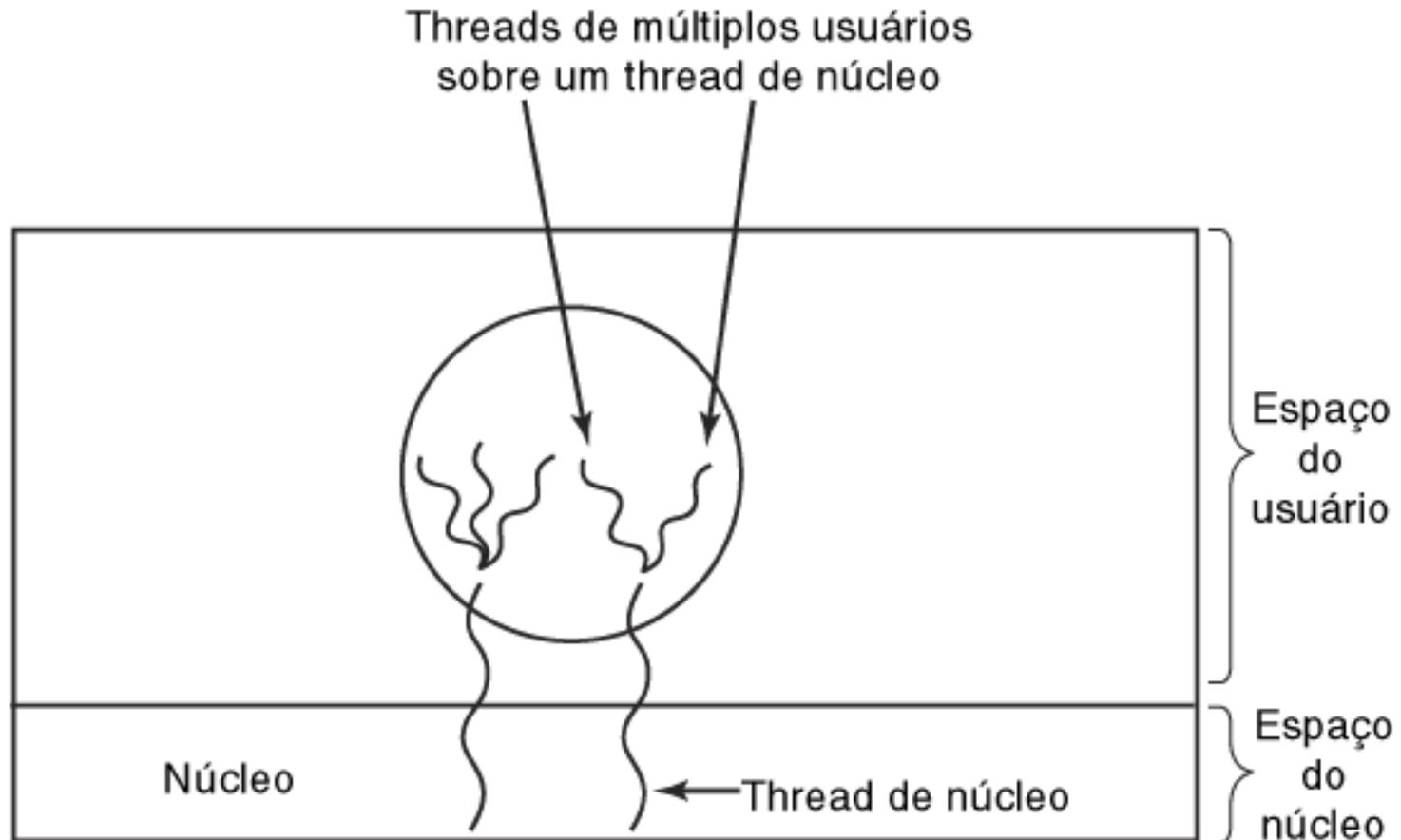
Um pacote de threads de usuário

Implementação de Threads de Núcleo



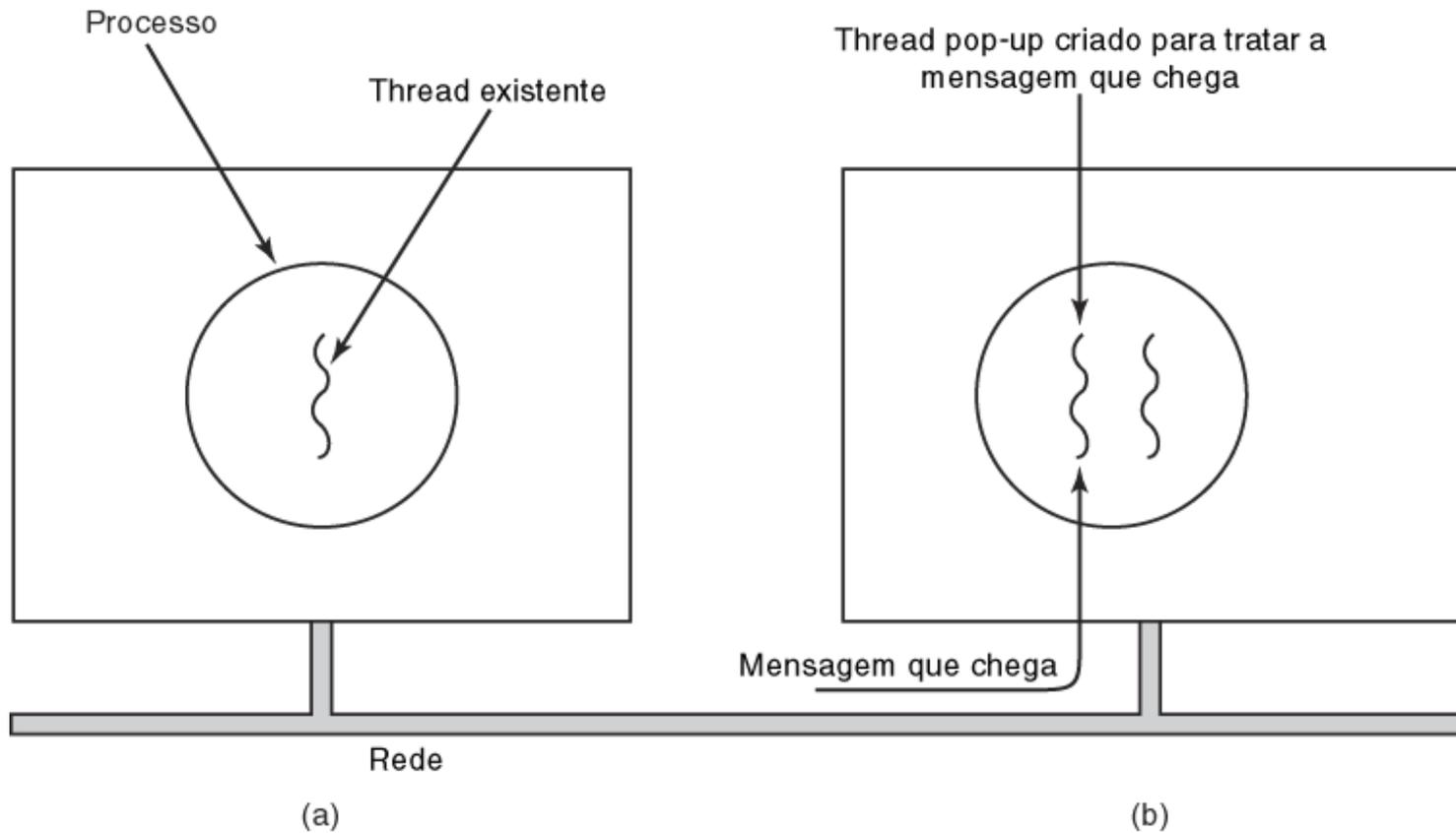
Um pacote de threads gerenciado pelo núcleo

Implementações Híbridas



Multiplexação de threads de usuário sobre threads de núcleo

Criação de um novo thread quando chega uma mensagem



Processos e 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)

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

Problemas com Concorrência

- Não-determinismo
 - $x = 1$ || $x = 2$
 - Qual o valor de “x” após a sua execução?
- Dependência de Velocidade
 - $[[f(); x = 1]]$ || $[[g(); x = 2]]$
 - O valor final de x depende de qual das funções, f() e g(), terminar primeiro
- *Starvation*
 - Processo de baixa prioridade precisa de um recurso que nunca é fornecido a ele...

Hermano P. Moura

Problemas com Concorrência (cont.)

- *Deadlock*
 - Um sistema de bibliotecas só fornece o “nada consta” para alunos matriculados e o sistema de matrícula só matricula os alunos perante a apresentação do “nada consta”
- **Definição:** dois processos bloqueiam a sua execução pois um precisa de um recurso bloqueado pelo outro processo

Veremos mais detalhes

Conceitos: starvation e deadlock

Hermano P. Moura

Apesar dos problemas...

Mais sobre concorrência

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 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

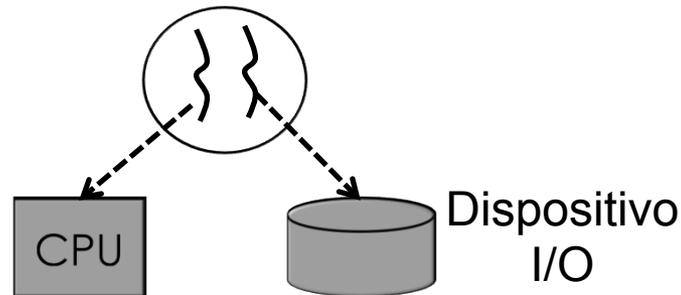
Comparação de desempenho: processo x thread

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...

- *Threads* são úteis, claro, em sistemas com múltiplas CPUs ou cores → paralelismo real

Tipos de Processo

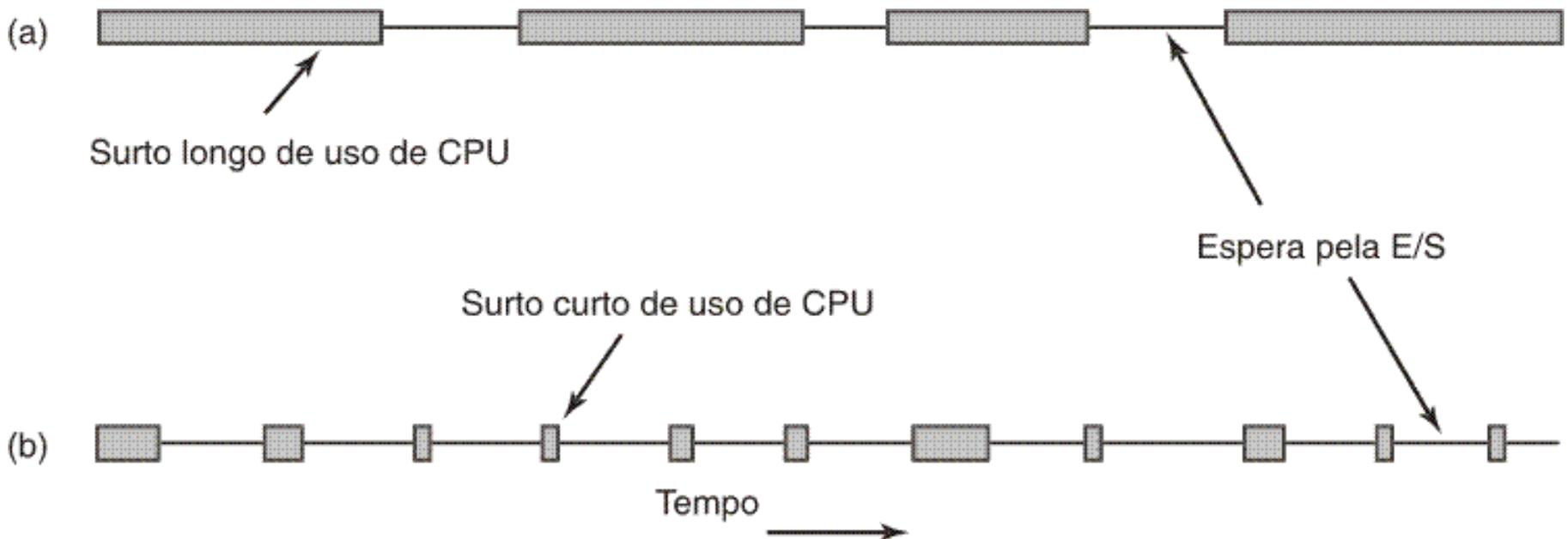
(incl. *thread* – processo leve)

- 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

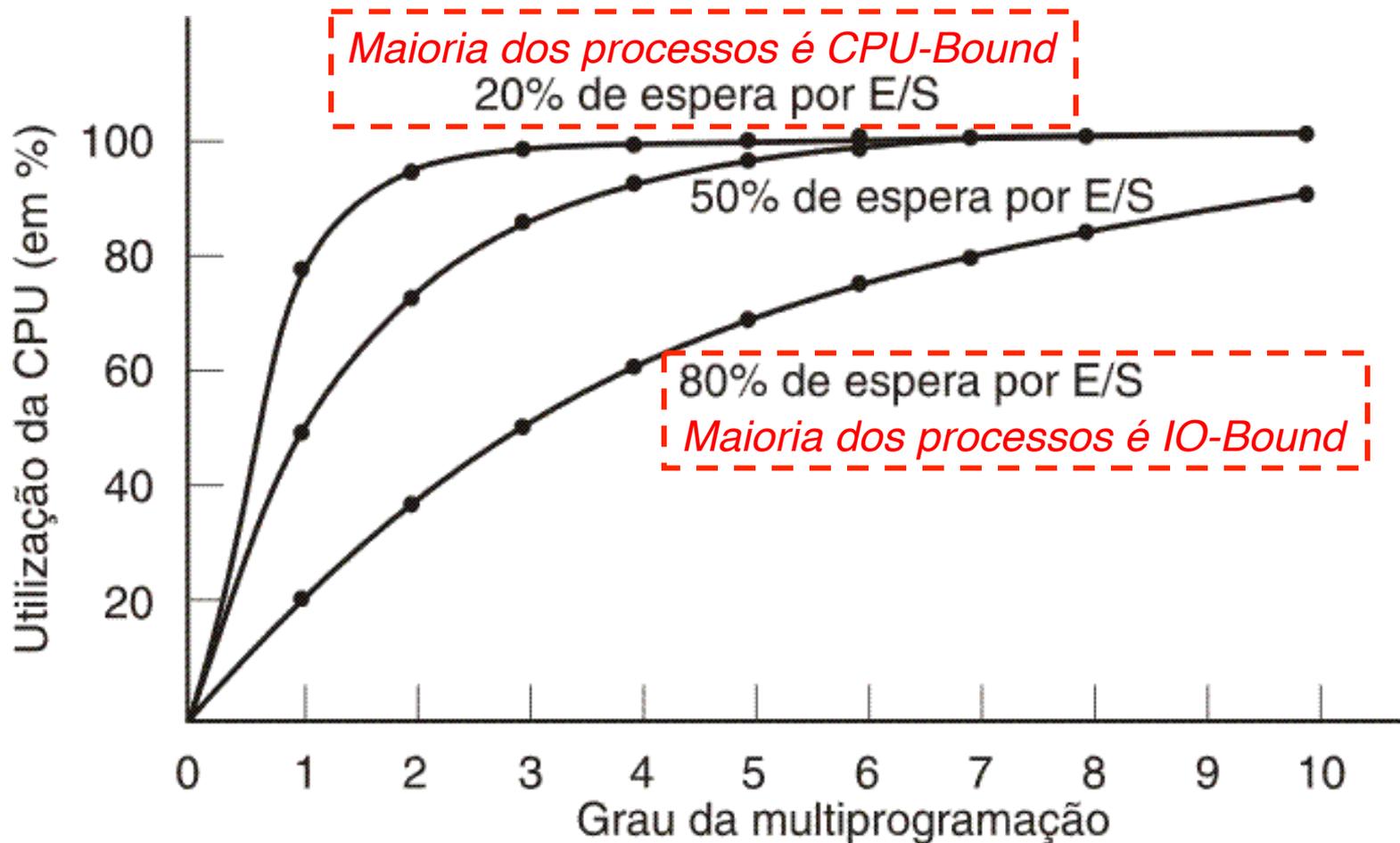
Comportamentos de Processos

- ▣ 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

POSIX Threads (Pthreads)

Modelo de execução de *threads* independente de linguagem, geralmente implementado como biblioteca

POSIX: Portable Operating System Interface

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 a parte ou parte da própria biblioteca C.
- ❑ Existem versões da biblioteca
- ❑ Cerca de 60 subrotinas
- ❑ Algumas chamadas de funções Pthreads:

- Thread Creation, Control, Cleanup
- Thread Scheduling
- Thread Synchronization
- Signal Handling

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

POSIX Threads (2)

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

#define NUMBER_OF_THREADS    10

void *print_hello_world(void *tid)
{
    /* This function prints the thread's identifier and then exits. */
    printf("Hello World. Greetings from thread %d\n", tid);
    pthread_exit(NULL);
}

...
```

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

```
#define NUMBER_OF_THREADS 10
```

```
void *print_hello_world(void *tid)
```

```
{
    /* This function prints the thread's identifier and then exits. */
    printf("Hello World. Greetings from thread %d0, tid);
    pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[])
```

```
{
    /* The main program creates 10 threads and then exits. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d0, i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d0, status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

POSIX Threads

(3)

Exercício

- Executando n vezes e verificando a ordem de execução das *threads* criadas ...

threads01v1 - Debugger Console

Release | x86_64

Overview Breakpoints Build and Run Tasks Restart Pause Clear Log

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/tty000
 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/tty001
 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.

threads01v1.c - threads01v1

Release | x86_64

Overview Action Breakpoints Build and Run Tasks Info

String Matching Search

Groups & Files

- threads01v1
 - Source
 - Documentation
 - Products
 - Targets
 - Executables
 - Find Results
 - Bookmarks
 - SCM
 - Project Symbols
 - Implementation Files
 - Interface Builder Files

File Name	Code		
threads01v1			
threads01v1.1			
threads01v1.c	7K		

```

threads01v1.c:6 <<No selected symbol>>
#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_MESSAGES; i++) {
        printf("Hello World from thread %d.%d\n", *pt, i);
    }
    pthread_exit(NULL);
}

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

    for(t=0; t < NUMBER_OF_THREADS; t++) {
        // printf("Creating thread %d\n", t);
        status = pthread_create(&threads[t], NULL, (void *)PrintHello, &t);

        if (status != 0) {
            printf("ERROR. Pthread_create returned error code %d", status);
            exit(-1);
        }

        printf("Thread %d created\n", t);
    }
    exit(0);
}
  
```

Debugging of "threads01v1" ended normally. Succeeded

```

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.

```

- Source
- Documentation
- Products
- Targets
- Executables
- Find Results
- Bookmarks
- SCM
- Project Symbols
- Implementation Files
- Interface Builder Files

```

threads01v1.1
c threads01v1.c

```

```

threads01v1.c:6 <No selected symbol>
#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_MESSAGES; i++)
        printf("Hello World from thread %d\n", *pt);
    pthread_exit(NULL);
}

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

    for(t=0; t < NUMBER_OF_THREADS; t++)
        // printf("Creating thread %d\n", t);
        status = pthread_create(&threads[t], NULL, PrintHello, (void*)t);

    if (status != 0) {
        printf("ERROR: Pthread_create\n");
        exit(-1);
    }

    printf("Thread %d created\n", t);
}
exit(0);

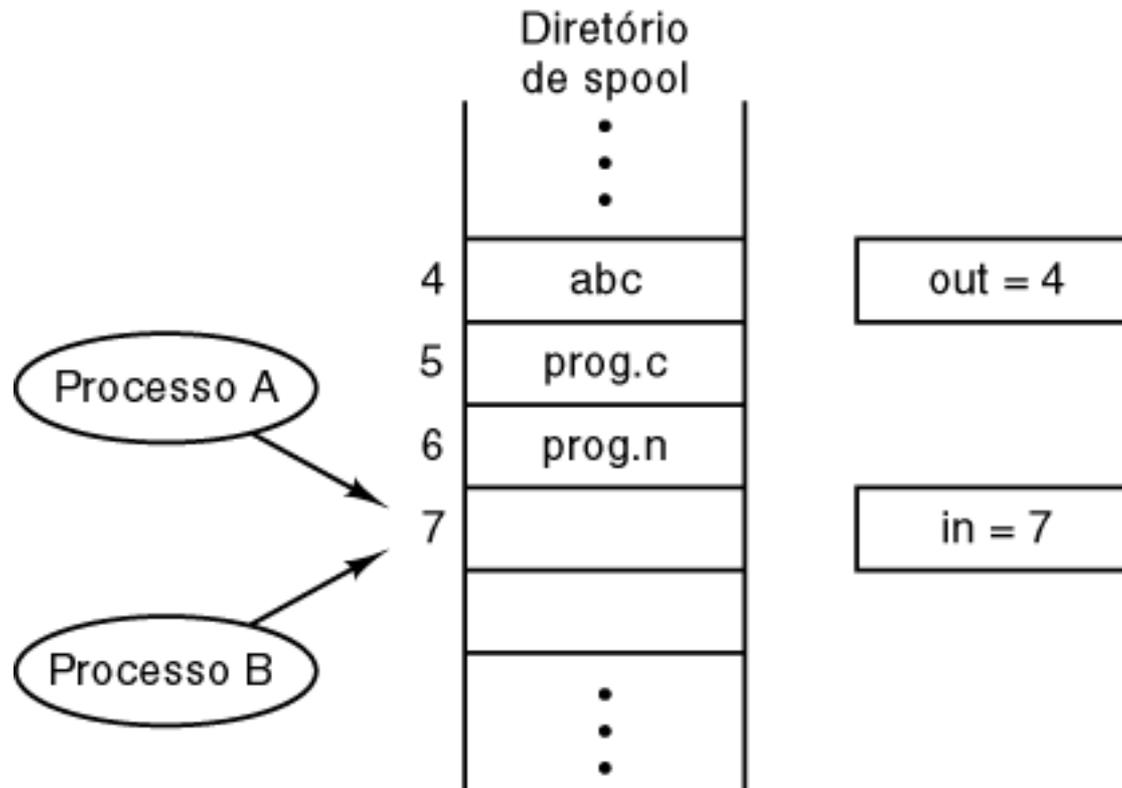
```

Succeeded Debugging of "threads01v1" ended normally.

Concorrência

Continuação

Condições de Disputa/Corrida

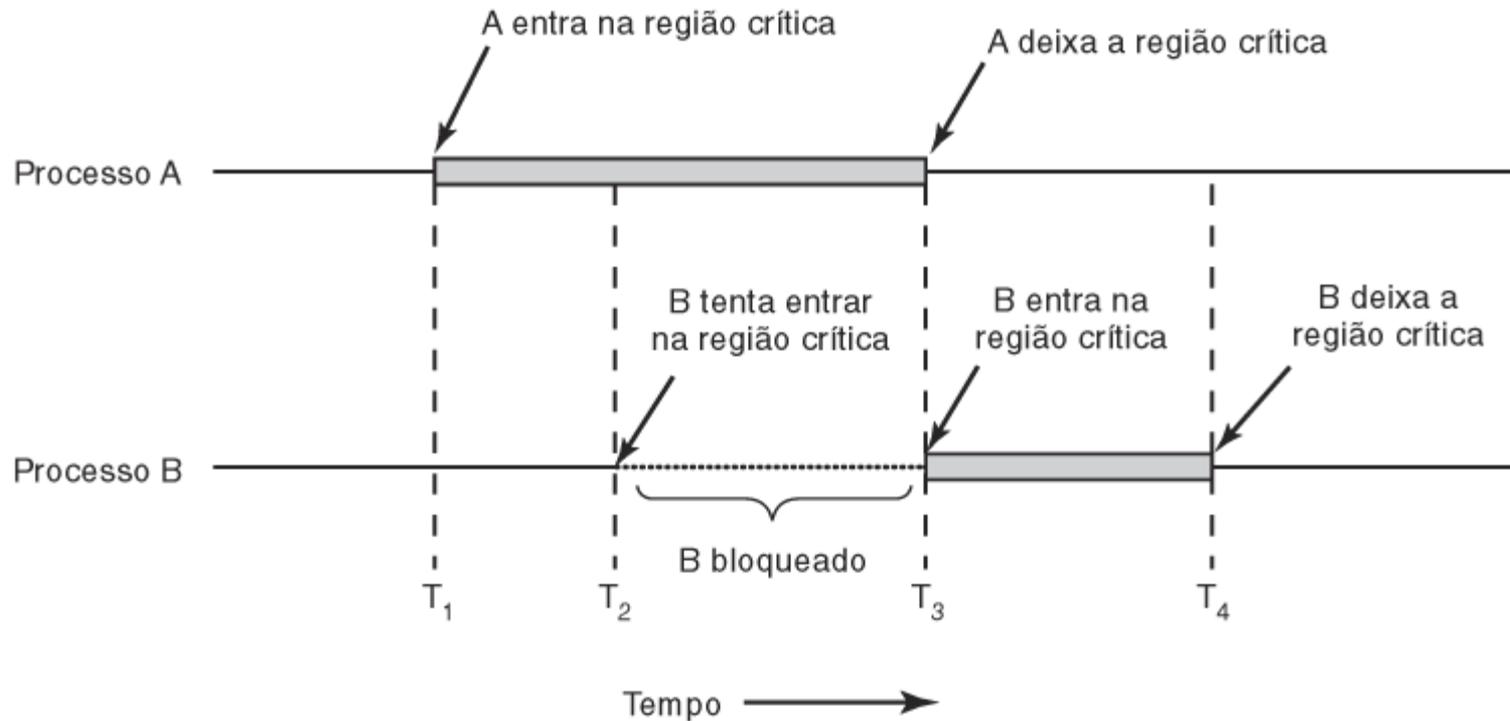


Dois processos querem ter acesso simultaneamente à memória compartilhada

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 (1)

```
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.

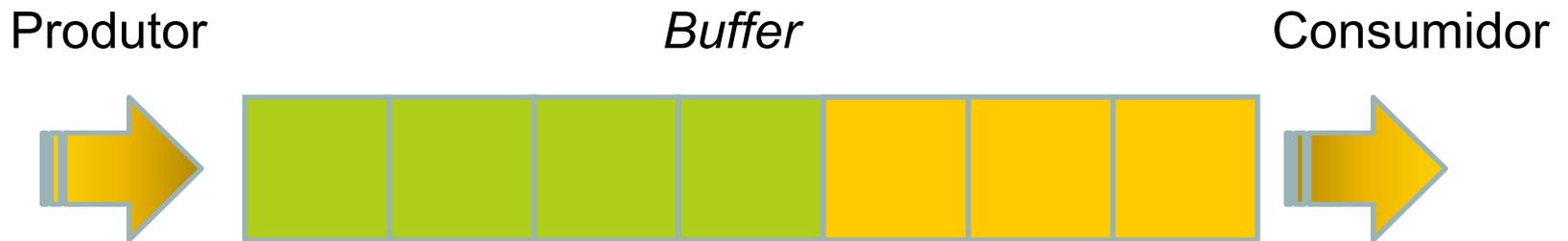
Exclusão Mútua com Espera Ociosa (2)

```
#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 (1)

```
#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);
    }
}
```

/ repita para sempre */*
/ se o buffer estiver vazio, vá dormir */*
/ retire o item do buffer */*
/ decresça de um o contador de itens no buffer */*
/ o buffer estava cheio? */*
/ imprima o item */*

Problema do produtor-consumidor com uma condição de disputa fatal

Região Crítica e Exclusão Mútua

Conceitos Associados

Leia: [doc submetido ao CTD 2015](#)

Conceitos fundamentais

- ✓ Condição de corrida/disputa
- ✓ Região Crítica
- ✓ Exclusão Mútua

Semáforo (1)

- **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 (1)

```
#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);
    }
}
```

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

O problema do produtor-consumidor usando semáforos

Semáforos (2)

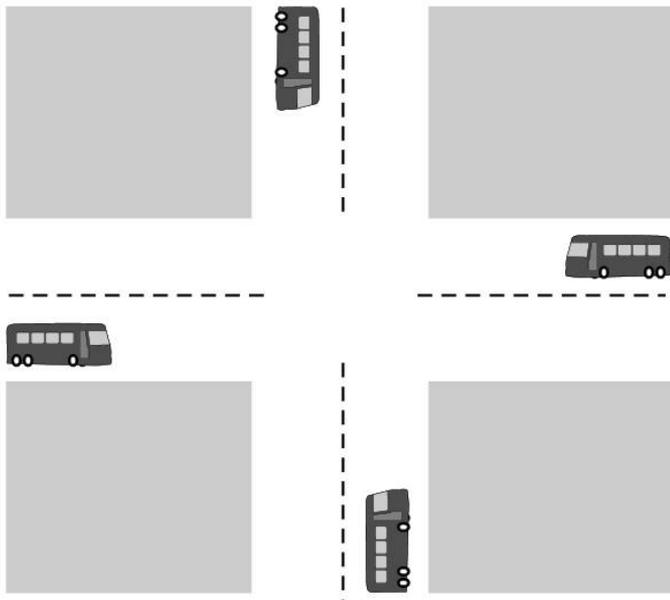
```
void consumer(void)
{
    int item;

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

/ laço infinito */*
/ decresce o contador full */*
/ entra na região crítica */*
/ pega o item do buffer */*
/ deixa a região crítica */*
/ incrementa o contador de lugares vazios */*
/ faz algo com o item */*

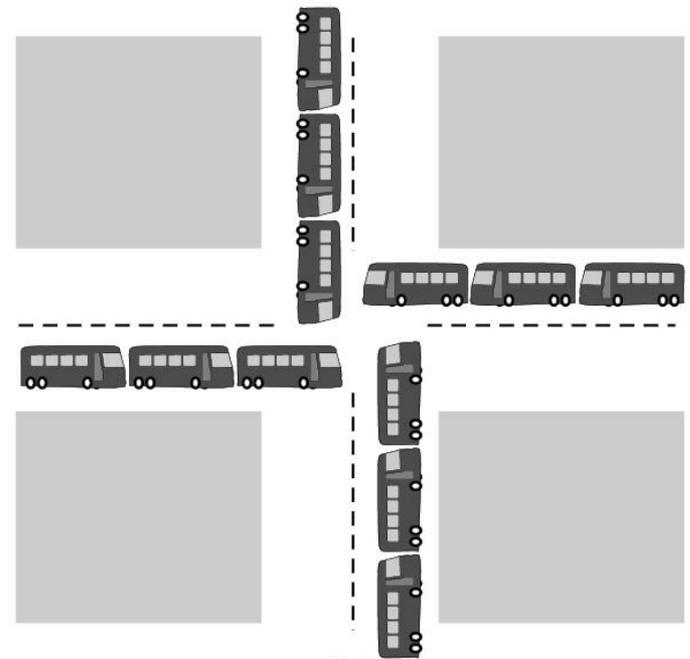
O problema do produtor-consumidor usando semáforos

Exemplo da necessidade de Semáforo



(a)

(a) Um *deadlock* potencial.



(b)

(b) Um *deadlock* real.

Monitores (1)

```
monitor example  
  integer i;  
  condition c;  
  
  procedure producer( );  
  .  
  .  
  .  
  end;  
  
  procedure consumer( );  
  .  
  .  
  .  
  end;  
end monitor;
```

Exemplo de um monitor

Monitores (2)

```
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;
end;
```

- ❑ O 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.llnl.gov/tutorials/pthreads/>

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

```
#define NUMBER_OF_THREADS 10
```

```
void *print_hello_world(void *tid)
```

```
{
    /* This function prints the thread's identifier and then exits. */
    printf("Hello World. Greetings from thread %d0, tid);
    pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[])
```

```
{
    /* The main program creates 10 threads and then exits. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d0, i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d0, status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

POSIX Threads

threads01v1 - Debugger Console

Release | x86_64

Overview Breakpoints Build and Run Tasks Restart Pause Clear Log

```

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/tty000
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/tty001
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 - threads01v1

Release | x86_64

Overview Action Breakpoints Build and Run Tasks Info

String Matching Search

Groups & Files

- threads01v1
 - Source
 - Documentation
 - Products
 - Targets
 - Executables
 - Find Results
 - Bookmarks
 - SCM
 - Project Symbols
 - Implementation Files
 - Interface Builder Files

File Name	Code		
threads01v1			
threads01v1.1			
threads01v1.c	7K		

```

threads01v1.c:6 <<No selected symbol>>
#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_MESSAGES; i++) {
        printf("Hello World from thread %d.%d\n", *pt, i);
    }
    pthread_exit(NULL);
}

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

    for(t=0; t < NUMBER_OF_THREADS; t++) {
        // printf("Creating thread %d\n", t);
        status = pthread_create(&threads[t], NULL, (void *)PrintHello, &t);

        if (status != 0) {
            printf("ERROR. Pthread_create returned error code %d", status);
            exit(-1);
        }

        printf("Thread %d created\n", t);
    }
    exit(0);
}

```

Debugging of "threads01v1" ended normally.

Succeeded

Um contador sequencial

```
#include <stdio.h>

long contador = 0;

void *inc(){
    int i = 0;
    for(; i < 9000000; i++) { contador++; }
}

void *dec(){
    int i = 0;
    for(; i < 9000000; i++) { contador--; }
}

int main (int argc, char *argv[]){
    inc();
    dec();
    printf("Valor final do contador: %ld\n", contador);
}
```

Um contador com pthreads

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

long contador = 0;

void *inc(void *threadid){
    int i = 0;
    for(; i < 9000000; i++) { contador++; }
}

void *dec(void *threadid){
    int i = 0;
    for(; i < 9000000; i++) { contador--; }
}

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);
}
```

```
#include <stdio.h>

long contador = 0;

void *inc(){
    int i = 0;
    for(; i < 9000000; i++) { contador++; }
}

void *dec(){
    int i = 0;
    for(; i < 9000000; i++) { contador--; }
}

int main (int argc, char *argv[]){
    inc();
    dec();
    printf("Valor final do contador: %ld\n", contador);
}
```

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

Relembrando: exclusão mútua

- ❑ Dois processos nunca podem estar simultaneamente na mesma região crítica
- ❑ Não se pode considerar velocidades ou número 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

- ❑ A typical sequence in the use of a mutex is as follows:
 - ❑ Create and initialize a mutex variable
 - ❑ Several threads attempt to lock the mutex
 - ❑ Only one succeeds and that thread owns the mutex
 - ❑ The owner thread performs some set of actions
 - ❑ The owner unlocks the mutex
 - ❑ Another thread acquires the mutex and repeats the process
 - ❑ Finally the mutex is destroyed

- ❑ When several threads compete for a mutex, the losers block at that call - an unblocking call is available with "trylock" instead of the "lock" call.
- ❑ When protecting shared data, it is the programmer's responsibility to make sure every thread that needs to use a mutex does so. For example, if 4 threads are updating the same data, but only one uses a mutex, the data can still be corrupted.

Exclusão mútua com pthreads

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

Criação de *mutexes*

▣ Estática:

```
pthread_mutex_t mymutex =  
    PTHREAD_MUTEX_INITIALIZER;
```

▣ Dinâmica:

```
pthread_mutex_t mymutex;  
  
...  
  
pthread_mutex_init(&mymutex, NULL);
```

Gerenciamento de *mutexes*

```
int pthread_mutex_destroy(  
    pthread_mutex_t *mutex);
```

```
int pthread_mutex_init(  
    pthread_mutex_t *restrict mutex,  
    const pthread_mutexattr_t  
        *restrict attr);
```

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);
```

- ❑ The `pthread_mutex_lock()` routine is used by a thread to acquire a lock on the specified *mutex* variable. If the mutex is already locked by another thread, *this call will block* the calling thread until the mutex is unlocked.
- ❑ `pthread_mutex_trylock()` will attempt to lock a mutex. However, if the mutex is already locked, *the routine will return immediately with a "busy" error code*. This routine may be useful in *preventing deadlock conditions*, as in a priority-inversion situation.

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

Aula de Linux

Terça, 19/04, 12h

D-002

There is nothing "magical" about mutexes...in fact they are akin to a "gentlemen's agreement" between participating threads. It is up to the code writer to insure that the necessary threads all make the mutex lock and unlock calls correctly. The following scenario demonstrates **a logical error**:

Thread 1	Thread 2	Thread 3
Lock	Lock	
$A = 2$	$A = A + 1$	$A = A * B$
Unlock	Unlock	

Question

- ❑ Question: When more than one thread is waiting for a locked mutex, which thread will be granted the lock first after it is released?
- ❑ ANSWER: Unless thread priority scheduling (not covered) is used, the assignment will be left to the native system scheduler and may appear to be more or less random.

Variáveis Condicionais com Pthreads

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

Concorrência/Sincronização

Exclusão mútua com pthreads: um contador certo

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

Exclusão mútua pode não ser o bastante

- ❑ Como visto antes, *threads* podem precisar cooperar
 - ❑ Uma *thread* só pode progredir se outra tiver realizado certa ação...
 - ❑ Ou seja, se certa **condição** for verdadeira
 - ❑ A condição refere-se aos valores dos elementos do estado compartilhado pelas *threads*

❑ Exemplo canônico: **produtor-consumidor**

Variáveis de condição

- ❑ Um dos mecanismos implementados pela biblioteca de pthreads
- ❑ Complementam *mutexes*
 - ❑ Sincronização no acesso a dados vs. Sincronização dependente de condições [**based upon the actual value of data**]
- ❑ Mecanismo de mais alto nível que semáforos
- ❑ Evitam a necessidade de **checar continuamente** se a condição é verdadeira
 - ❑ Sem **espera ocupada!!!**

This can be **very resource consuming** since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.

Produtor-Consumidor espera ocupada (1/2)

```
#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 ;
}
```

Produtor-Consumidor espera ocupada (2/2)

```
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
- E se houvesse vários produtores ou consumidores?

Usando variáveis de condição com pthreads

- ❑ Sempre junto com um *mutex*
- ❑ Duas operações básicas
 - ❑ Esperar que certa condição torne-se verdadeira
 - ❑ `pthread_cond_wait(cond_var, mutex)`
 - ❑ Avisar outra(s) *thread(s)* que a condição tornou-se verdadeira
 - ❑ `pthread_cond_signal(cond_var)` **OU**
 - ❑ `pthread_cond_broadcast(cond_var)`

Criação e destruição de variáveis de condição

Estática:

```
pthread_cond_t mycv =  
    PTHREAD_COND_INITIALIZER;
```

Dinâmica:

```
pthread_cond_t mycv;  
...  
pthread_cond_init(&mycv, attr);
```

NULL, em geral



Destruição:

```
pthread_cond_destroy(&mycv)
```

Usage

- ❑ It is a **logical error** to call `pthread_cond_signal()` before calling `pthread_cond_wait()`.
- ❑ **Proper locking and unlocking** of the associated mutex variable is essential when using these routines. For example:
 - ❑ Failing to lock the mutex before calling `pthread_cond_wait()` may cause it NOT to block...
 - ❑ Failing to unlock the mutex after calling `pthread_cond_signal()` may not allow a matching `pthread_cond_wait()` routine to complete (it will remain blocked....

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);
}
```

Vem aí
Deadlock →