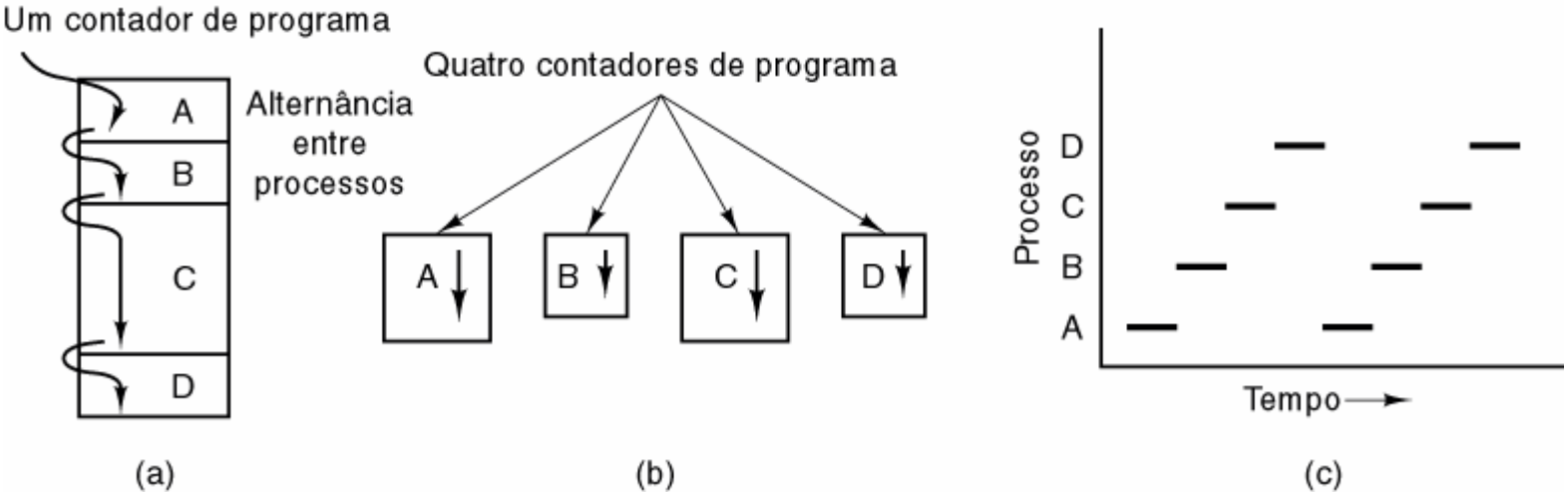
# Capítulo 2

# Processos e Threads

2.1 Processos
2.2 Threads
2.3 Comunicação interprocesso
2.4 Problemas clássicos de IPC
2.5 Escalonamento

# Processos

## O Modelo de Processo




- Multiprogramação de quatro programas
- Modelo conceitual de 4 processos sequenciais, independentes
- Somente um programa está ativo a cada momento

# Criação de Processos

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

# Término de Processos

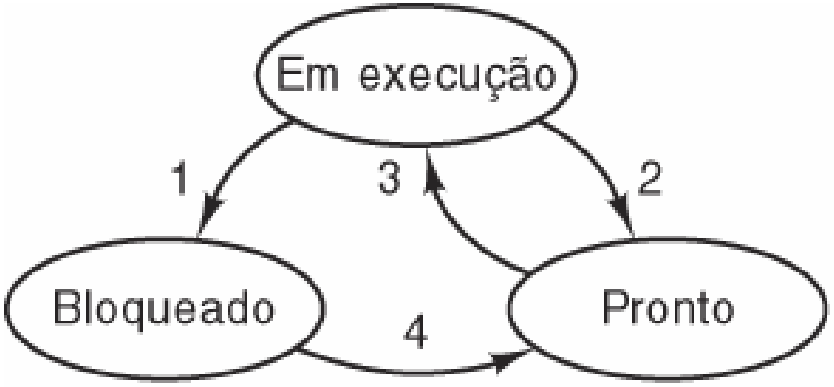
- Condições que levam ao término de processos
  - Saída normal (voluntária)
  - Saída por erro (voluntária)
  - Erro fatal (involuntário)
  - Cancelamento por um outro processo (involuntário)

# Hierarquias de Processos

- Pai cria um processo filho, processo filho pode criar seu próprio processo
- Formam uma hierarquia
  - UNIX chama isso de “grupo de processos”
- Windows não possui o conceito de hierarquia de processos
  - Todos os processos são criados iguais

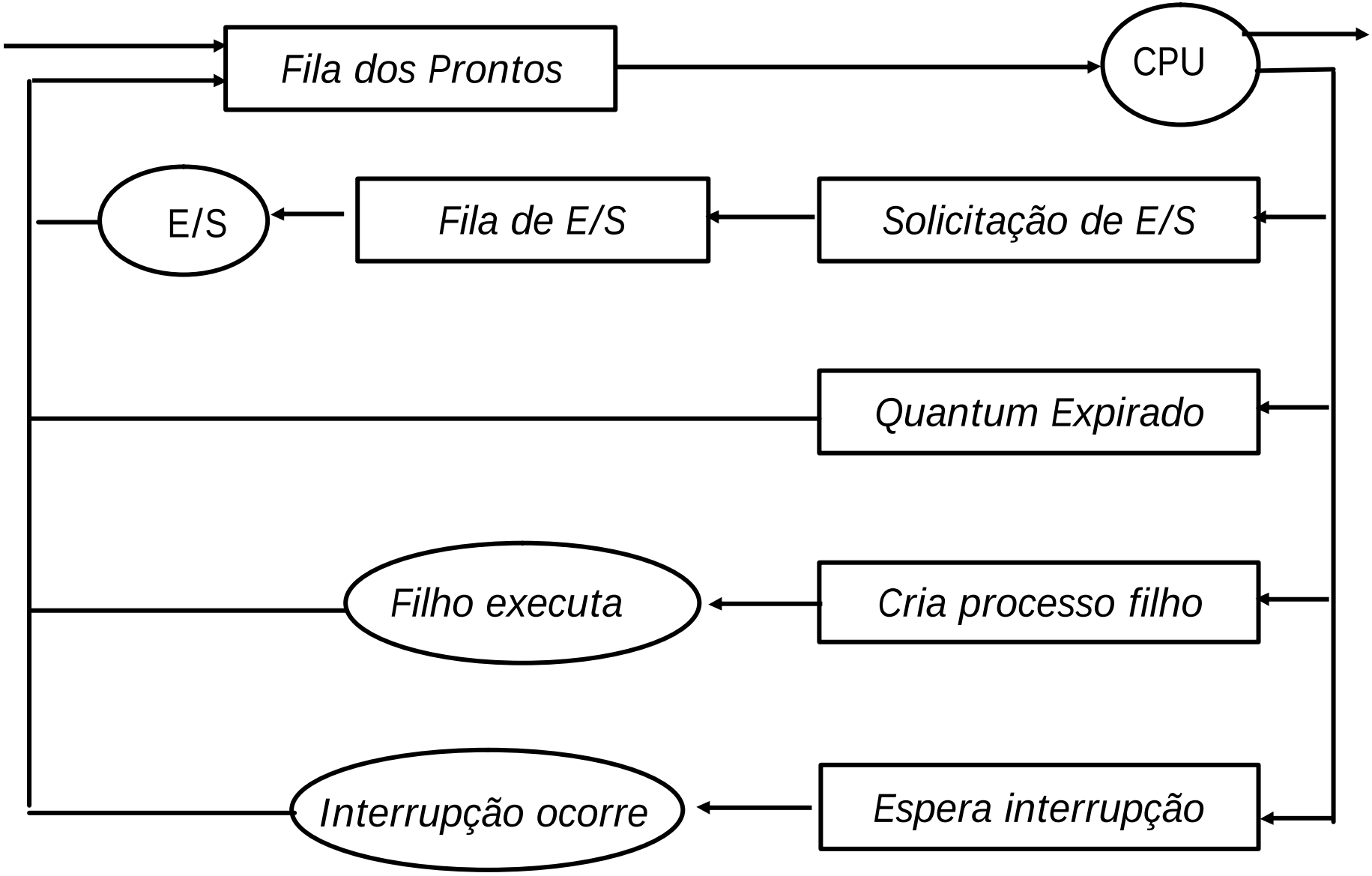
# Estados de Processos (1)

## Transições entre os estados de um processo



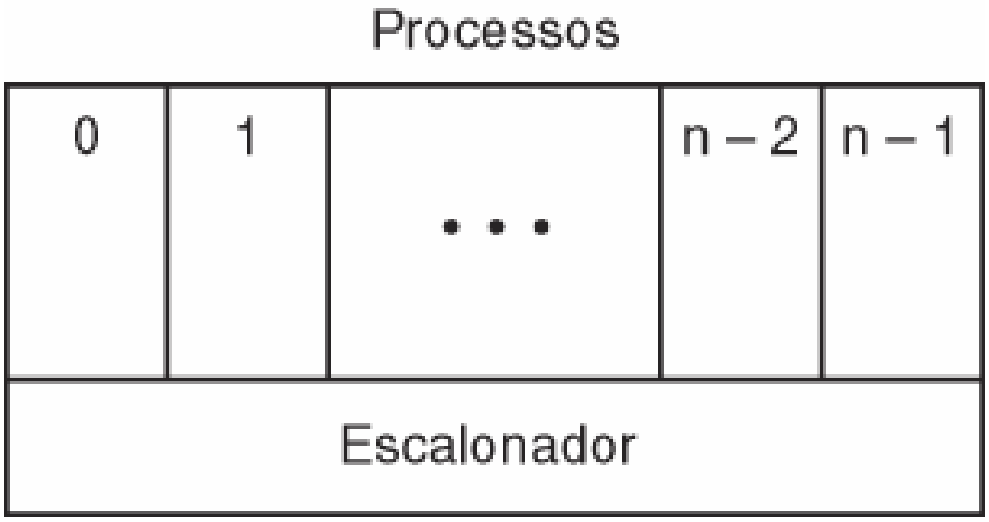

1. O processo bloqueia aguardando uma entrada
2. O escalonador seleciona outro processo
3. O escalonador seleciona esse processo
4. A entrada torna-se disponível

- Possíveis estados de processos
  - em execução
  - bloqueado
  - pronto para executar




# Filas de Escalonamento de Processos

# Estados de Processos (2)




- Camada mais inferior de um SO estruturado por processos
  - trata interrupções, escalonamento
- Acima daquela camada estão os processos sequenciais

# Implementação de Processos (1)

| 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 | | |

Campos da entrada de uma tabela de processos

# Implementação de Processos (2)

1. O hardware empilha o contador de programa etc.
2. O hardware carrega o novo contador de programa a partir do vetor de interrupção.
3. O procedimento em linguagem de montagem salva os registradores.
4. O procedimento em linguagem de montagem configura uma nova pilha.
5. O serviço de interrupção em C executa (em geral lê e armazena temporariamente a entrada).
6. O escalonador decide qual processo é o próximo a executar.
7. O procedimento em C retorna para o código em linguagem de montagem.
8. O procedimento em linguagem de montagem inicia o novo processo atual.

O que faz o nível mais baixo do SO quando ocorre uma interrupção

# Threads
# O Modelo de Thread (1)

- **Processo** ✍ um espaço de endereço e uma única linha de controle
- **Threads** ✍ um espaço de endereço e múltiplas linhas de controle
- O Modelo do Processo
  - Agrupamento de recursos (espaço de endereço com texto e dados do programa; arquivos abertos, processos filhos, tratadores de sinais, alarmes pendentes etc)
  - Execução
- O Modelo da Thread
  - Recursos particulares (PC, registradores, pilha)
  - Recursos compartilhados (espaço de endereço – variáveis globais, arquivos etc)
  - Múltiplas execuções no mesmo ambiente do processo – com certa independência entre as execuções

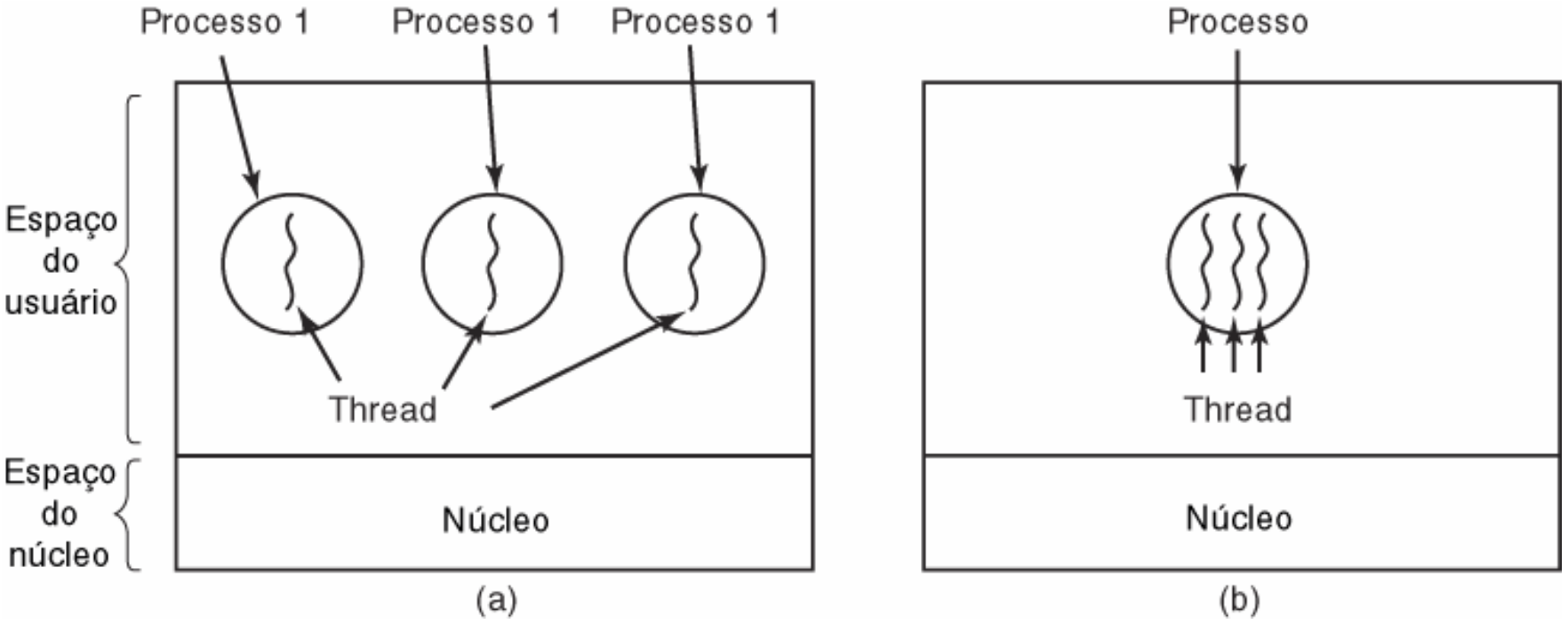
**Analogia**

Execução de múltiplos threads em paralelo em um processo *(multithreading)*

e

Execução de múltiplos processos em paralelo em um computador

# O Modelo de Thread (2)




(a) Três processos cada um com um thread

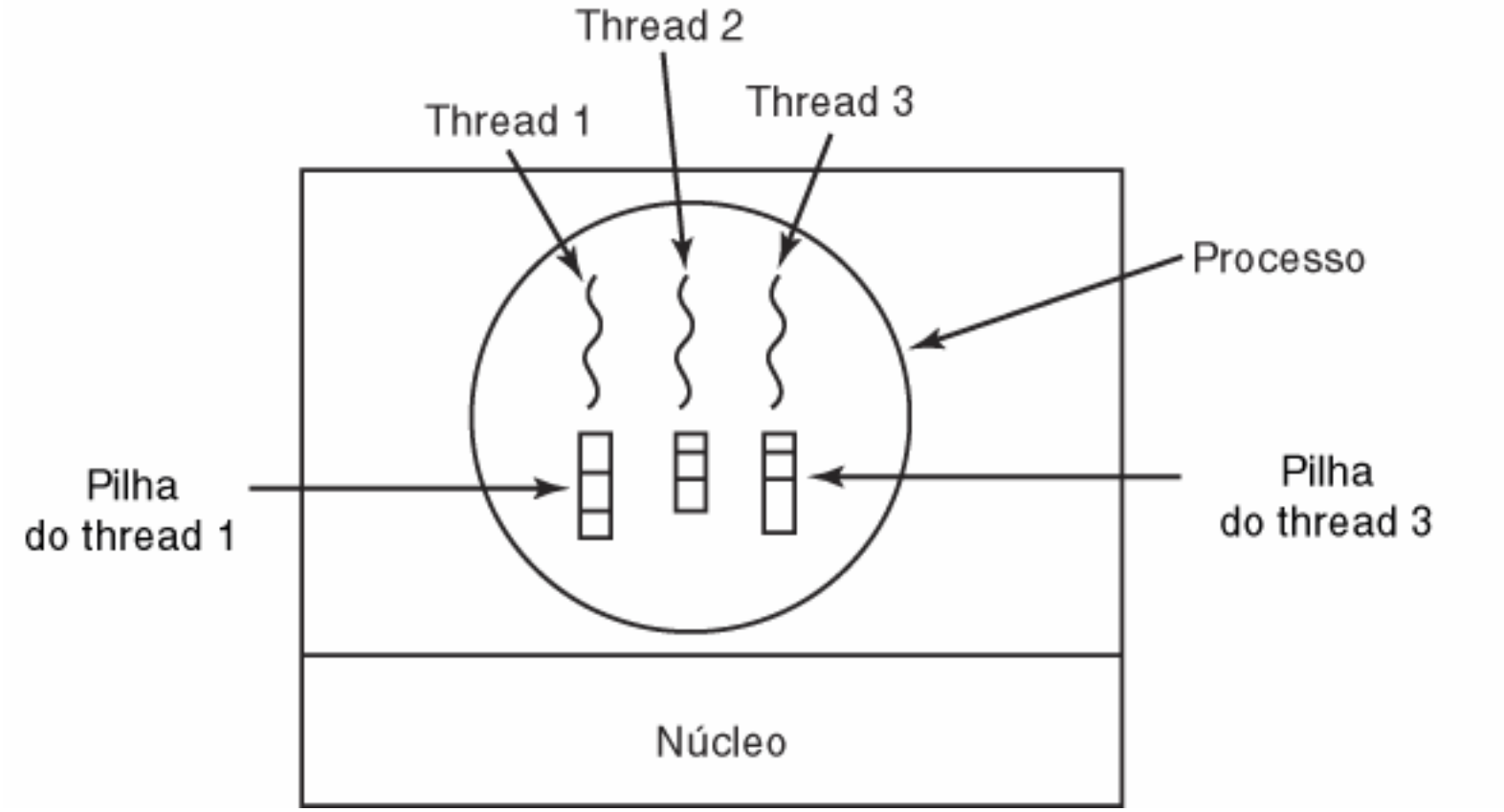
(b) Um processo com três threads

# O Modelo de Thread (3)

| Itens por processo | Itens por thread |
|---|---|
| Espaço de endereçamento | Contador de programa |
| Variáveis globais | Registradores |
| Arquivos abertos | Pilha |
| Processos filhos | Estado |
| Alarmes pendentes | |
| Sinais e tratadores de sinais | |
| Informação de contabilidade | |

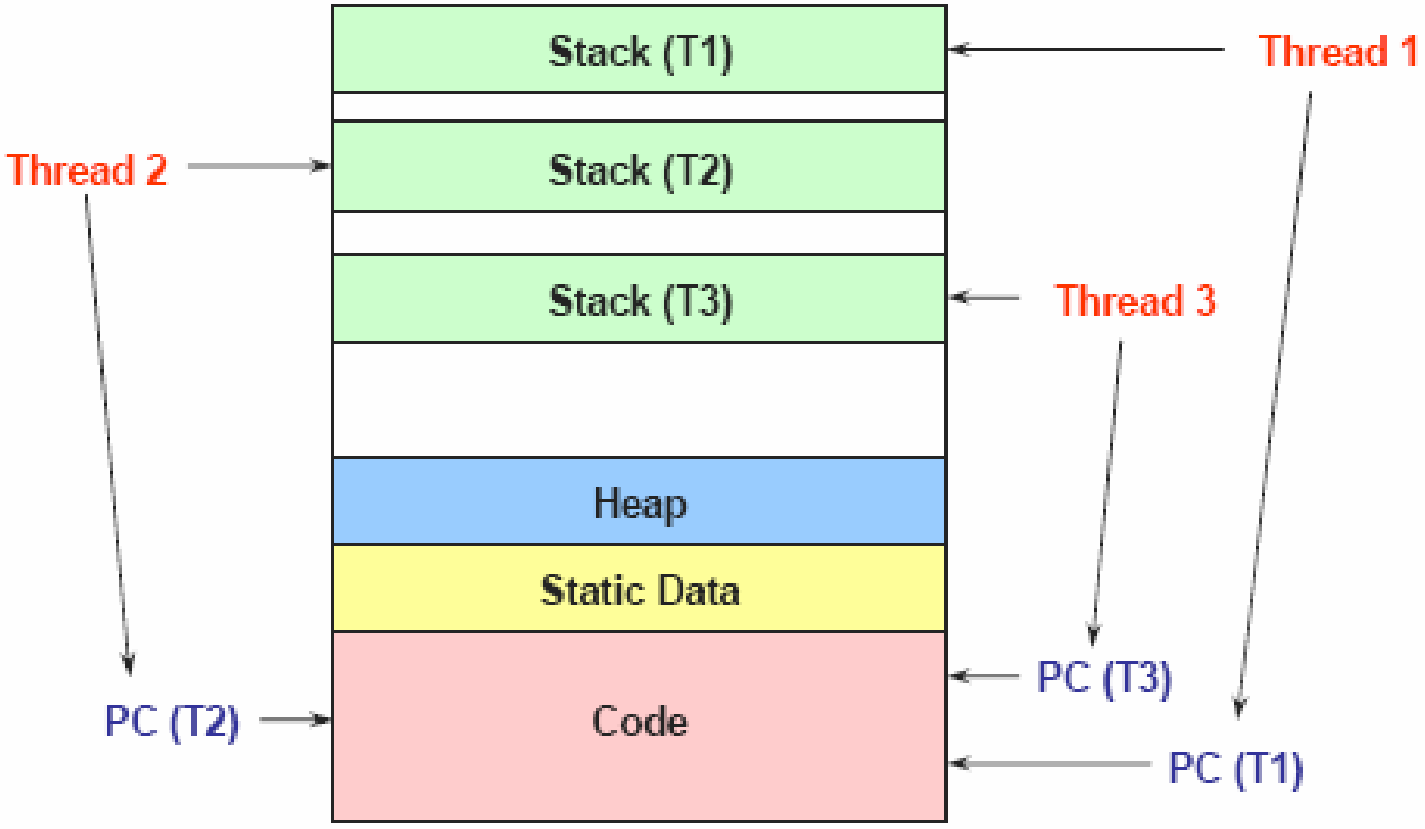
- Items compartilhados por todos os threads em um processo
- Itens privativos de cada thread

# O Modelo de Thread (4)




Cada thread tem sua própria pilha

# O Modelo de Thread (5)




Onde:

- PC ✍ Contador de Programa
- Ti ✍ Thread i

# O Modelo de Thread (6)

## Exemplos de chamadas de biblioteca

- create_thread
- thread_exit
- thread_destroy
  - destrói o thread – inclui o desligamento do segmento de pilha
- thread_wait (thread2)
  - thread 1 bloqueia até que thread 2 termine
- thread yield
  - Renuncia à CPU em prol de outro thread
- thread_status
  - Retorna o estado atual do thread

# O Modelo de Thread (7)

- Threads voluntariamente renunciam à CPU com *thread_yield*

Thread ping

```
while (1) {
    printf("ping\n");
    thread_yield();
}
```

Thread pong

```
while (1) {
    printf("pong\n");
    thread_yield();
}
```

- Qual o resultado da execução desses dois threads?

# O Modelo de Thread (8)

- Como funciona o *thread_yield( )?*
  - Semântica: *thread-yield* “abre mão” da CPU em favor de outro thread
  - Em outras palavras: chaveia o contexto para outro thread
- O que significa retornar de uma chamada *thread_yield?*
  - Significa que outro thread chamou *thread_yield*
- Resultado da execução de ping/pong

```
printf(“ping\n”);
thread_yield();
printf(“pong\n”);
thread_yield();
```

# Uso de Thread (1)

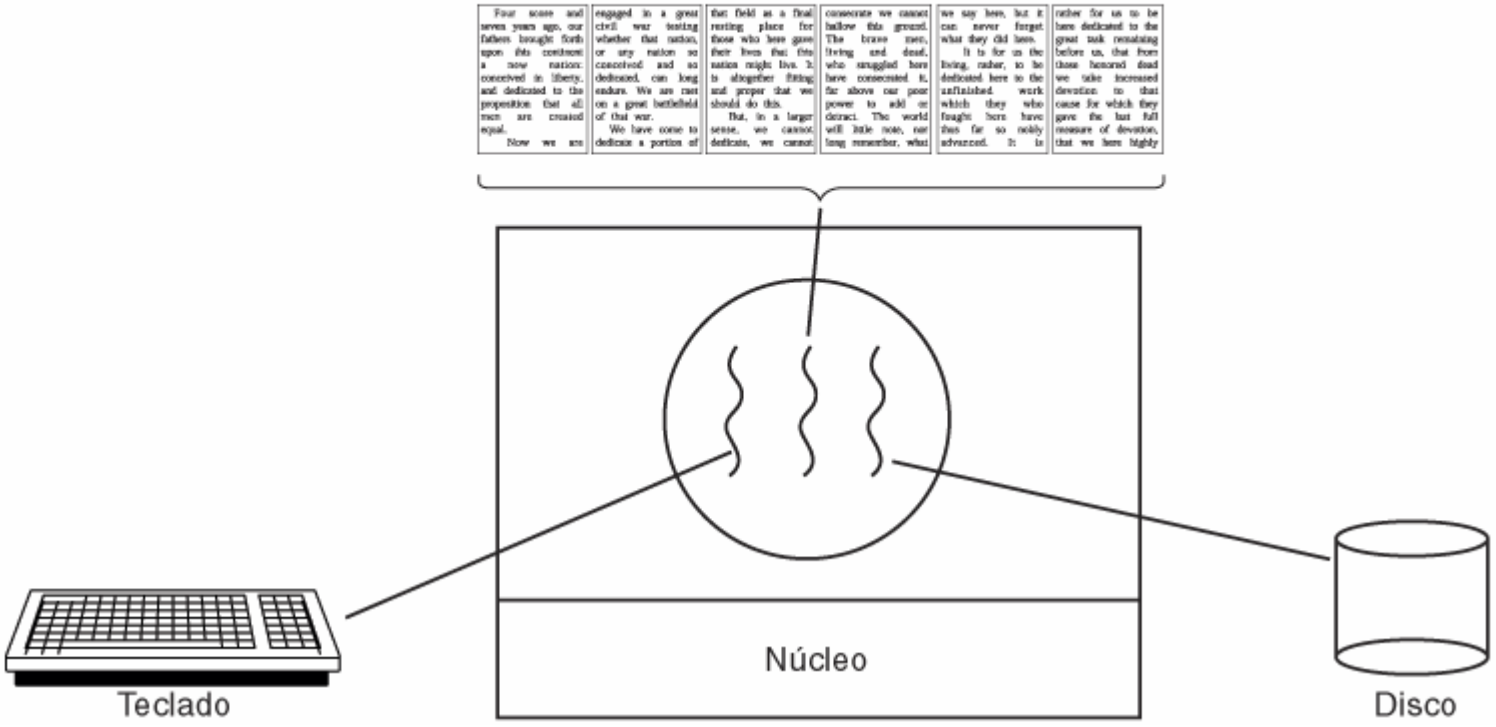
## Porquê threads?

- Simplificar o modelo de programação (aplicação com múltiplas atividades => decomposição da aplicação em múltiplas threads)
- Gerenciamento mais simples que o processo (não há recursos atachados – criação de thread 100 vezes mais rápida que processo)
- Melhoria do desempenho da aplicação (especialmente quando thread é orientada a E/S)
- Útil em sistemas com múltiplas CPUs

## Exemplo do uso de threads

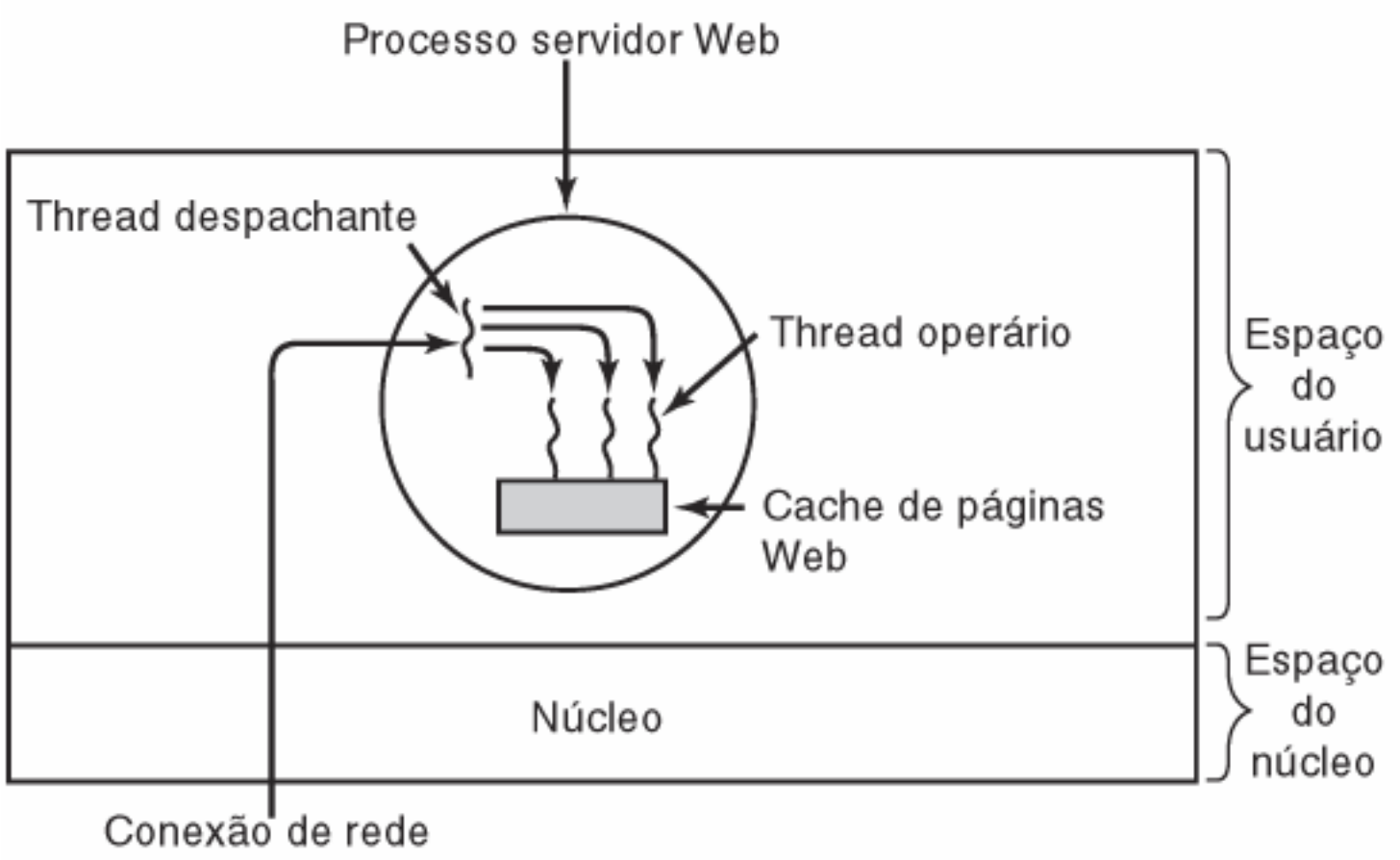
- Aplicação: Processador de texto
- Uso de 3 threads
  - Interação com o usuário
  - Formatação de texto
  - Cópia de documento
- Solução com 3 processos versus solução com 3 threads

# Uso de Thread (2)




Um processador de texto com três threads

# Uso de Thread (3)




Um servidor web com múltiplos threads

# Uso de Thread (4)

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if(page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

- Código simplificado do slide anterior
  - (a) Thread despachante
  - (b) Thread operário

# Uso de Thread (5)

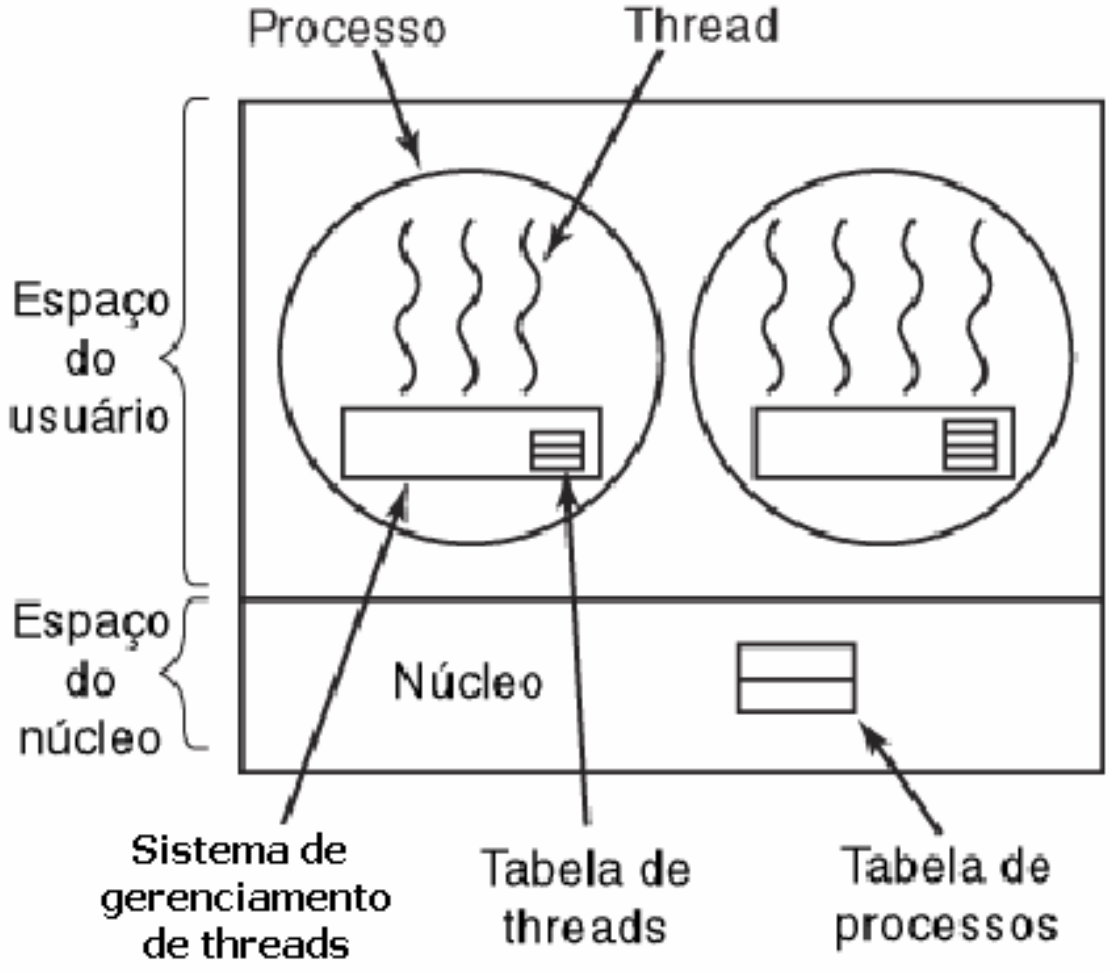
| Modelo | Características |
|---|---|
| Threads | Paralelismo, chamadas ao sistema com bloqueio |
| Processo monothread | Sem paralelismo, chamadas ao sistema com bloqueio |
| Máquina de estados finitos | Paralelismo, chamadas ao sistema sem bloqueio, interrupções |

Três maneiras de construir um servidor

# Uso de Thread (6)

- Threads não preemptivos têm que “abrir mão” da CPU voluntariamente
- Um thread com execução longa controlará a CPU (pode rodar para sempre)
- Chaveamento de contexto ocorre apenas quando são emitidas chamadas voluntárias para *thread_yield(), thread_stop(), ou thread_exit()*
- Escalonamento preemptivo causa chaveamento de contexto involuntário
- Necessário retomar controle do processador assíncronamente
  - Usar interrupção de clock
  - Tratador de interrupção de clock força thread atual a “chamar” *thread_yield*

# Implementação de Threads de Usuário(1)




Um pacote de threads de usuário

# Implementação de Threads de Usuário(2)

- Threads podem ser implementados em S.O. que não suporta thread
- Cada processo precisa ter sua tabela de threads
- São gerenciados pelo sistema de gerenciamento de threads *(run-time system)* – através de coleção de procedimentos de biblioteca
- Criação de thread novo, chaveamento entre threads e sincronização de threads => feitos via chamada de procedimento
- Ações de uma chamada de procedimento:
  - Verifica se thread muda para estado bloqueado
  - Salva PC, pilha, registradores
  - Busca na tabela de threads prontos para rodar
  - Carrega PC e ponteiro de pilha => novo thread começa a rodar
- Chaveamento de thread é uma ordem de magnitude mais rápido que mudar para o modo núcleo (grande vantagem sobre implementação no núcleo)
- Cada processo pode ter seu próprio algoritmo de escalonamento de threads personalizado

# Implementação de Threads de Usuário(3)

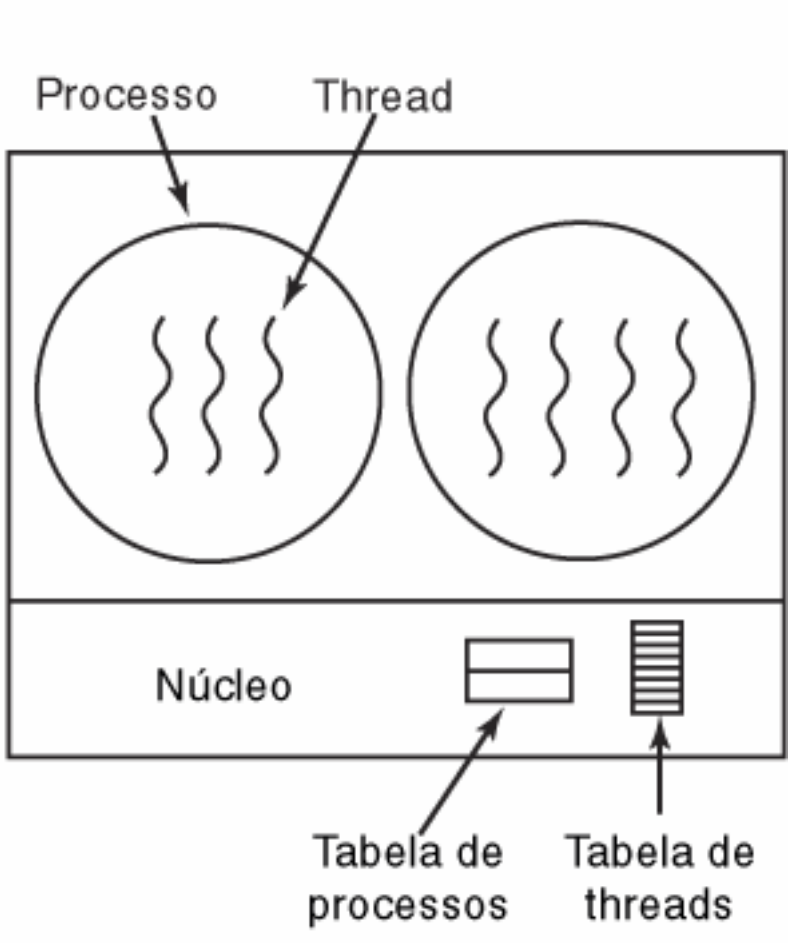
- Thread faz chamada de sistema que bloqueia
  - thread que bloqueia afeta os outros
  - processo que roda todos os threads é interrompido
- Soluções Possíveis:
  - Alterar todas as chamadas do sistema para não bloqueantes
    - Requer alteração do sistema operacional
    - Requer alterações em vários programas do usuário (alteração da semântica da chamada)
  - "Envelopar" as chamadas do sistema com procedimento que verifica se a chamada vai bloquear ou não
    - *Ex.: select* do Unix – *read* é substituído por outro *read* que primeiro faz a chamada *select* – se a chamada vai bloquear, é adiada => roda outro *thread*

# Implementação de Threads de Usuário(4)

## Desvantagens

- Não são bem integrados com o S.O. => S.O. pode tomar decisões ruins
  - Escalonar um processo com threads ociosos
  - Bloquear um processo cujo thread iniciou uma E/S mesmo que o processo tenha outros threads que podem executar
  - Retirar a CPU de um processo com um thread que retém uma variável de trancamento *(lock)*
- Solução: comunicação entre o núcleo e o sistema de gerenciamento de threads no espaço do usuário

# Implementação de Threads de Núcleo(1)




Um pacote de threads gerenciado pelo núcleo

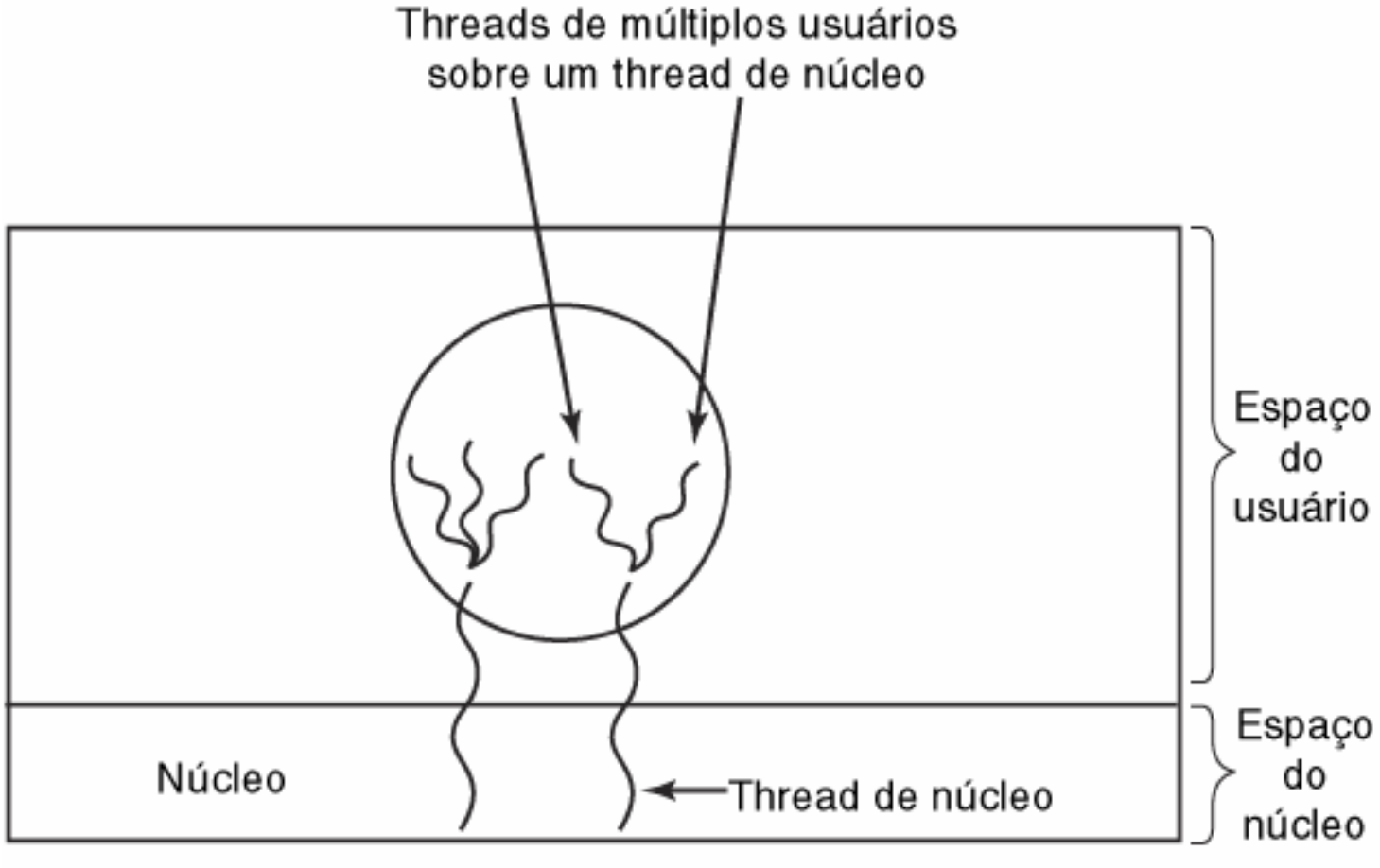
# Implementação de Threads de Núcleo(2)

- Não precisa de sistema de gerenciamento de threads *(run-time)*
  - o núcleo sabe sobre os threads
- Gerenciamento de threads através de chamadas para o núcleo (tabela de threads) – reciclagem de threads
- Chamadas que podem bloquear um thread
  - implementadas como chamadas ao sistema
- O que faz o núcleo quando um thread bloqueia?
- Não requer modificações nas chamadas ao sistema

**Desvantagem**

Custo alto => chamadas ao sistema

# Implementações Híbridas(1)




Multiplexação de threads de usuário sobre threads de núcleo

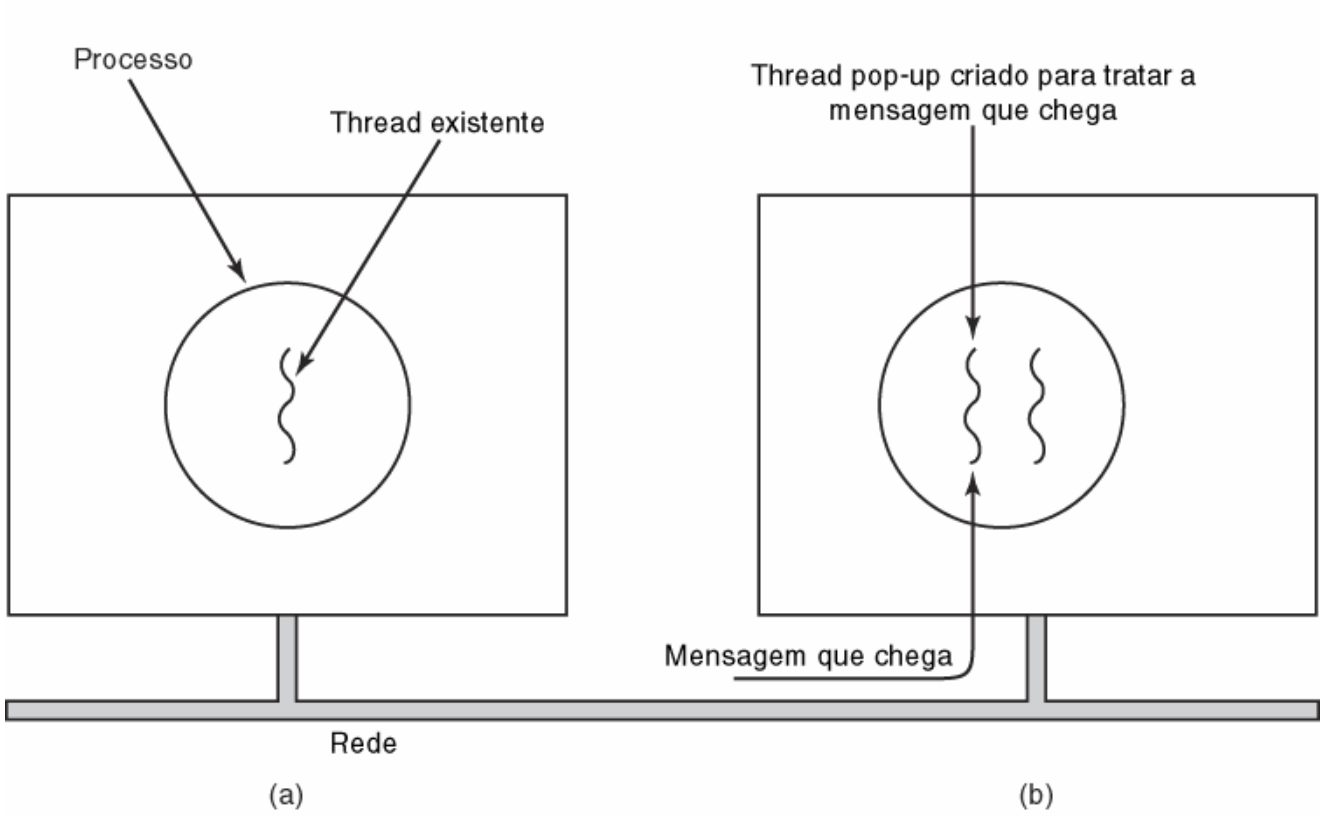
# Implementações Híbridas(2)

- Threads no usuário não usam truques como chamadas especiais não bloqueantes
  - Se tiver que bloquear => deve ser possível rodar outros threads do mesmo processo (se prontos)
- Thread espera por outro thread => não envolve o núcleo (sistema de execução – *runtime system* – trata isso)
- Uso de **upcall**
  - núcleo sabe que thread bloqueou
  - núcleo notifica sistema de execução (passa ID do thread e descrição do evento ocorrido)
  - núcleo ativa sistema de execução (num endereço conhecido)
  - Sistema de execução re-escala seus threads (pega da lista de prontos, seta registradores e inicializa)
- Volta do evento bloqueado
  - núcleo sabe que página foi trazida para memória ou dados foram lidos
  - núcleo notifica sistema de execução *(upcall)*
  - Sistema de execução decide se volta a rodar thread  ou coloca na fila de prontos
- Tratamento de interrupção de hardware
  - Thread em execução é interrompido e salvo (pode ou não voltar a rodar depois da interrupção ser tratada)
- **Desvantagem:** Viola o princípio do modelo em camadas

# Ativações do Escalonador

- Objetivo – imitar a funcionalidade dos threads de núcleo
  - ganha desempenho de threads de usuário
- Evita transições usuário/núcleo desnecessárias
- Núcleo atribui processadores virtuais para cada processo
  - deixa o sistema supervisor alocar threads para processadores
- Problema:
  Baseia-se fundamentalmente nos *upcalls* - o núcleo (camada inferior) chamando procedimentos no espaço do usuário (camada superior)

# Threads Pop-Up

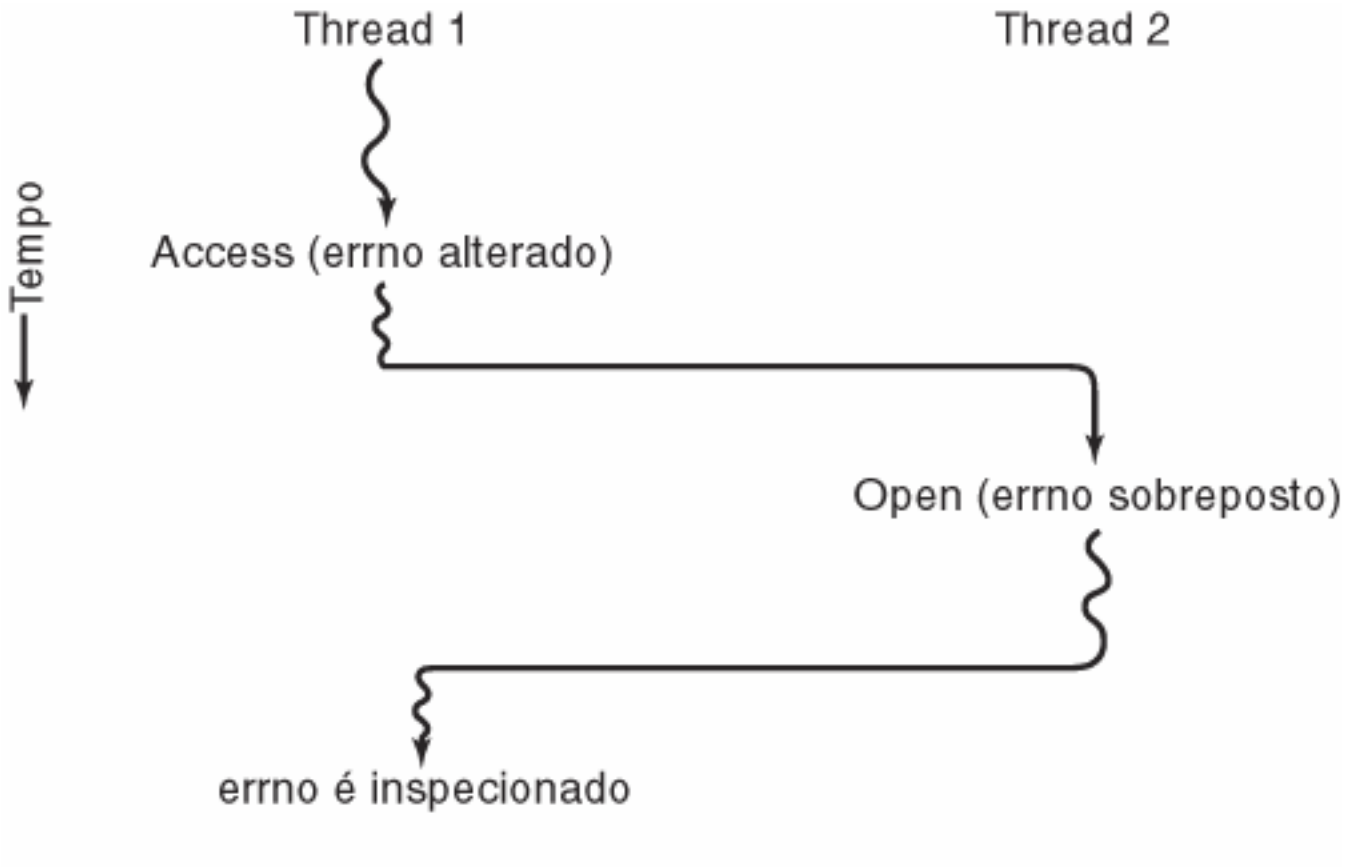



- Criação de um novo thread quando chega uma mensagem

  (a) antes da mensagem chegar

  (b) depois da mensagem chegar

# Convertendo Código Monothread em Código Multithread (1)

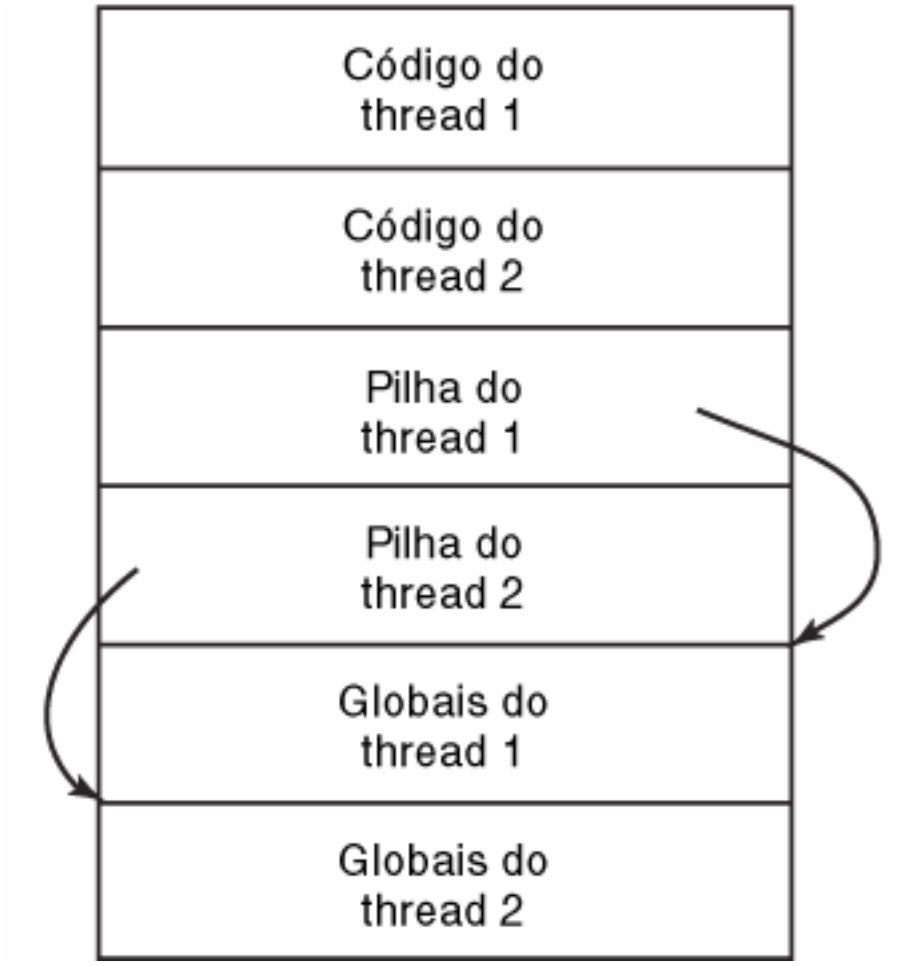



Conflitos entre threads sobre o uso de uma variável global

# Convertendo Código Monothread em Código Multithread (2)

- Um thread tem suas próprias variáveis locais e globais e parâmetros de procedimentos
- Variável global para um thread mas não global para multithreads => inconsistências (ex.: errno)
- Resolução de inconsistências em multithreads
  - Proibir variáveis globais
  - Atribui para cada thread suas próprias variáveis globais privadas (cópia do *errno* para cada thread)
  - Como acessar variáveis globais atribuídas para cada thread?
    - Alocar espaço de memória e passar para o thread como parâmetro extra
    - Novas bibliotecas *(create_global)*
- Código de procedimentos de biblioteca não são reentrantes (segunda chamada para procedimento não é feita enquanto primeira não foi finalizada)
  - Reescrever a bilbioteca
  - Flag que indica que biblioteca está em uso – elimina paralelismo
  - Reprojetar o sistema (no mínimo redefinir semântica das chamadas, reescrever bibliotecas) mantendo compatibilidade com programas e aplicações atuais.

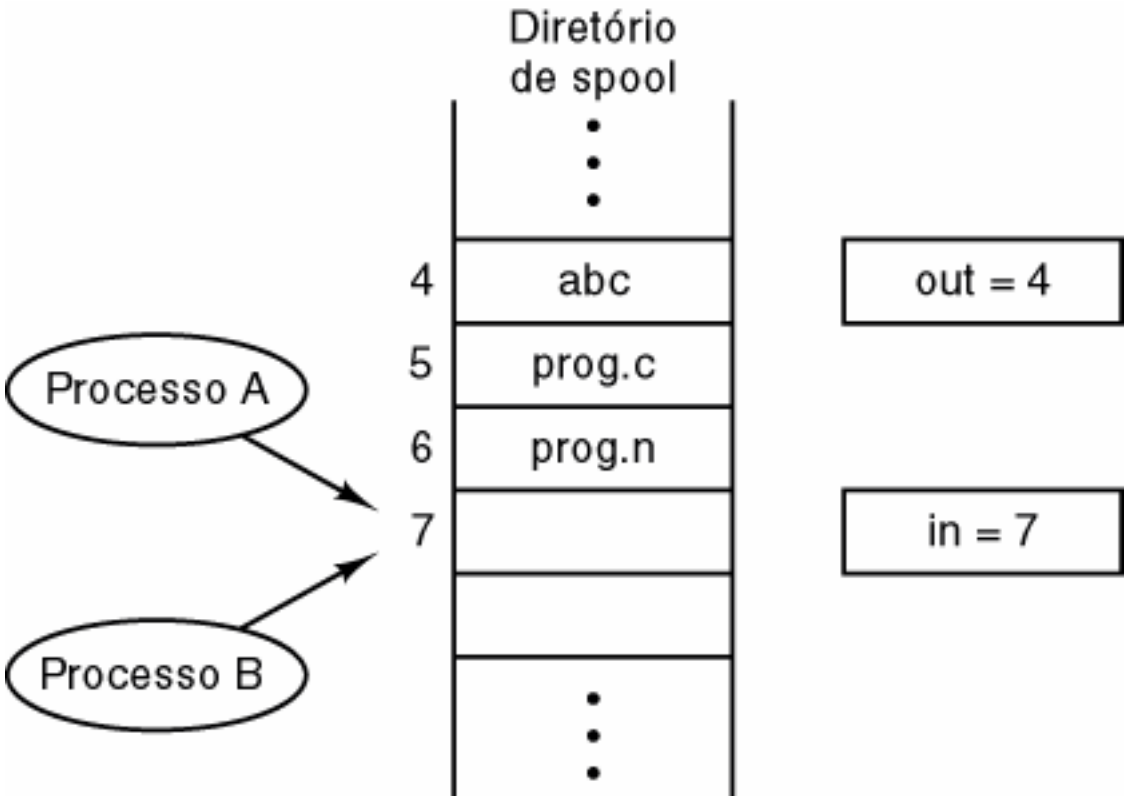
# Convertendo Código Monothread em Código Multithread (3)




Threads podem ter variáveis globais privadas

# Comunicação Interprocesso

## Condições de Disputa



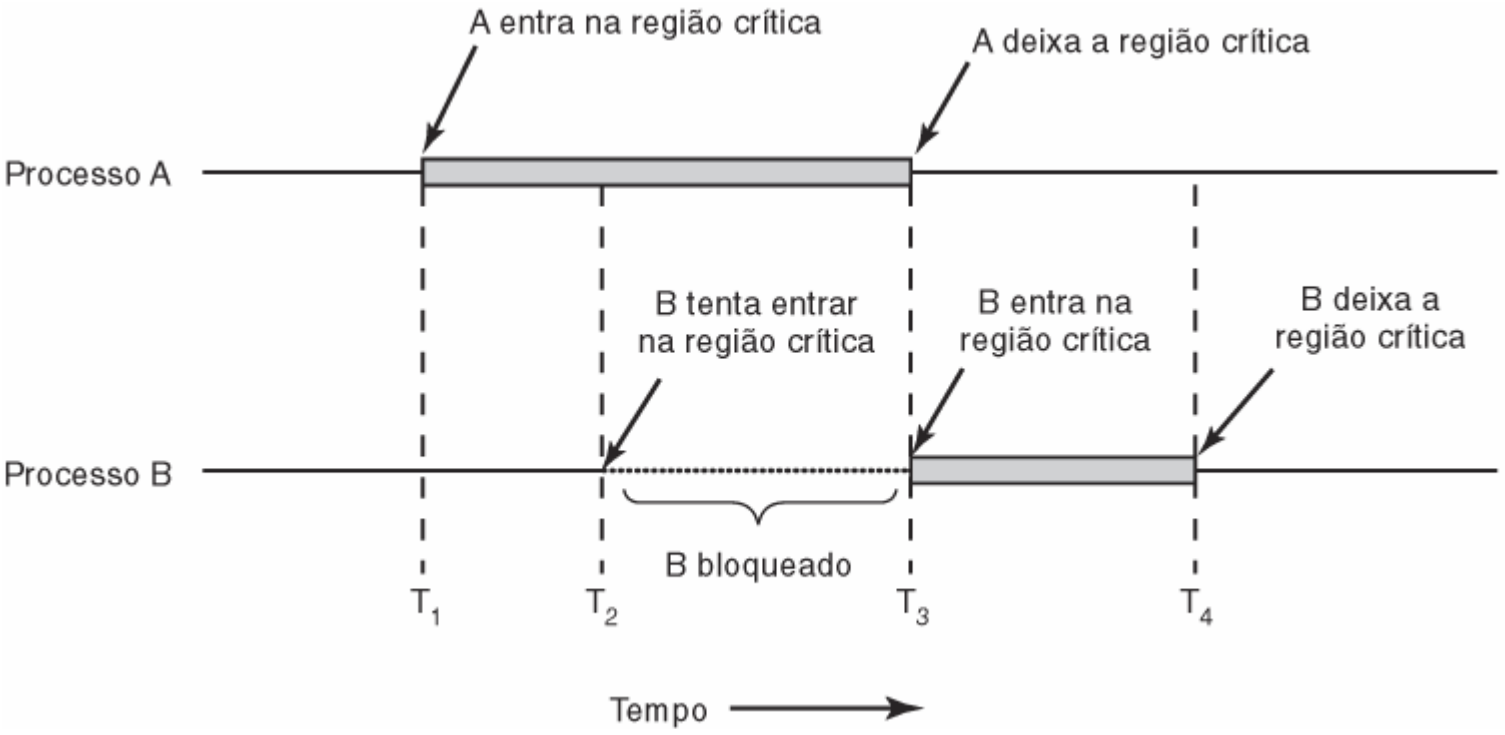

Dois processos querem ter acesso simultaneamente à memória compartilhada

# Regiões Críticas (1)

Quatro condições necessárias para prover exclusão mútua:

1. Nunca dois processos simultaneamente em uma região crítica
2. Nenhuma afirmação sobre velocidades ou números de CPUs
3. Nenhum processo executando fora de sua região crítica pode bloquear outros processos
4. 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 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 Peterson para implementar exclusão mútua

# Exclusão Mútua com Espera Ociosa (3)

```
enter_region:
        TSL REGISTER,LOCK          | copia lock para o registrador e põe lock em 1
        CMP REGISTER,#0            | lock valia zero?
        JNE enter_region           | se fosse diferente de zero, lock estaria ligado,
                                   | portanto continue no laço de repetição
        RET | retorna a quem chamou; entrou na região crítica

leave_region:
        MOVE LOCK,#0               | coloque 0 em lock
        RET | retorna a quem chamou
```

Entrando e saindo de uma região crítica usando a instrução TSL

# Dormir e Acordar

```
#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? */
      }
}


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

# Semáforos

```
#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;                            /* conta os lugares vazios no buffer */
semaphore full = 0 ;                            /* conta os lugares preenchidos no buffer */

void producer(void)
{
      int item;

      while (TRUE) {                            /* TRUE é a constante 1 */
            item = produce_item( );             /* gera algo para pôr no buffer */
            down(&empty);                       /* decresce o contador empty */
            down(&mutex);                       /* entra na região crítica */
            insert_item(item);                  /* põe novo item no buffer */
            up(&mutex);                         /* sai da região crítica */
            up(&full);                          /* incrementa o contador de lugares preenchidos */
      }
}


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

# Mutexes

```
mutex_lock:
        TSL REGISTER,MUTEX          | copia mutex para o registrador e o põe em 1
        CMP REGISTER,#0             | o mutex era zero?
        JZE ok                      | se era zero, o mutex estava desimpedido, portanto retorne
        CALL thread_yield           | o mutex está ocupado; escalone um outro thread
        JMP mutex_lock              | tente novamente mais tarde
ok:     RET | retorna a quem chamou; entrou na região crítica

mutex_unlock:
        MOVE MUTEX,#0               | põe 0 em mutex
        RET | retorna a quem chamou
```

Implementação de *mutex_lock* e *mutex_unlock*

# 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;
```

- Delineamento do problema do produtor-consumidor com monitores
  - somente um procedimento está ativo por vez no monitor
  - o buffer tem N lugares

# Monitores (3)

## Solução para o problema do produtor-consumidor em Java

```
public class ProducerConsumer {
    static final int N = 100;                   // constante com o tamanho do buffer
    static producer p = new producer( );        // instância de um novo thread produtor
    static consumer c = new consumer( );// instância de um novo thread consumidor
    static our_monitor mon = new our_monitor( );  // instância de um novo monitor

    public static void main(String args[ ]) {
        p.start( );                             // inicia o thread produtor
        c.start( );                             // inicia o thread consumidor
    }
    static class producer extends Thread {
        public void run( )  {                   // o método run contém o código do thread
            int item;
            while (true) {                      // laço do produtor
                item = produce_item( );
                mon.insert(item);
            }
        }
        private int produce_item( ) { ... }     // realmente produz
    }
    static class consumer extends Thread {
        public void run( )  {                   // método run contém o código do thread
            int item;
            while (true) {                      // laço do consumidor
                item = mon.remove( );
                consume_item (item);
            }
        }
        private void consume_item(int item) { ... }     // realmente consome
    }
    static class our_monitor {                  // este é o monitor
        private int buffer[ ] = new int[N];
        private int count = 0, lo = 0, hi = 0; // contadores e índices

        public synchronized void insert(int val) {
            if (count == N) go_to_sleep( );     // se o buffer estiver cheio, vá dormir
            buffer [hi] = val;                  // insere um item no buffer
            hi = (hi + 1) % N;                  // lugar para colocar o próximo item
            count = count + 1;                  // mais um item no buffer agora
            if (count == 1) notify( );          // se o consumidor estava dormindo, acorde-o
        }
        public synchronized int remove( ) {
            int val;
            if (count == 0) go_to_sleep( );     // se o buffer estiver vazio, vá dormir
            val = buffer [lo];                  // busca um item no buffer
            lo = (lo + 1) % N;                  // lugar de onde buscar o próximo item
            count = count – 1;                  // um item a menos no buffer
            if (count == N – 1) notify( );      // se o produtor estava dormindo, acorde-o
            return val;
        }
        private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
    }
}
```

# Monitores (4)

```
#define N 100                                   /* número de lugares no buffer */

void producer(void)
{
      int item;
      message m;                                /* buffer de mensagens */

      while (TRUE) {
            item = produce_item( );             /* gera alguma coisa para colocar no buffer */
            receive(consumer, &m);              /* espera que uma mensagem vazia chegue */
            build_message(&m, item);            /* monta uma mensagem para enviar */
            send(consumer, &m);                 /* envia item para consumidor */
      }
}

void consumer(void)
{
      int item, i;
      message m;

      for (i = 0; i < N; i++) send(producer, &m);  /* envia N mensagens vazias */
      while (TRUE) {
            receive(producer, &m);              /* pega mensagem contendo item */
            item = extract_item(&m);            /* extrai o item da mensagem */
            send(producer, &m);                 /* envia a mensagem vazia como resposta */
            consume_item(item);                 /* faz alguma coisa com o item */
      }
}
```

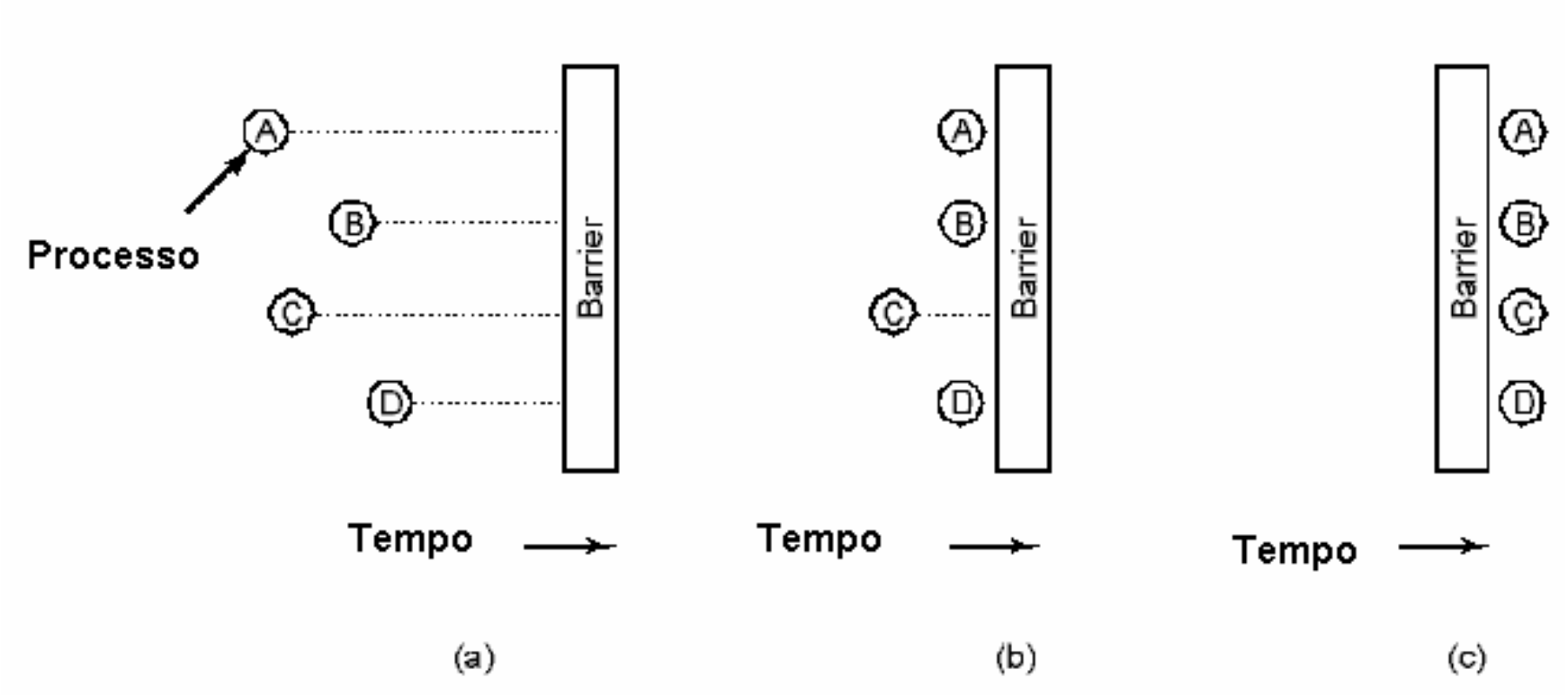
Solução para o problema do produtor-consumidor em Java (parte 2)

# Troca de Mensagens

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

O problema do produtor-consumidor com N mensagens

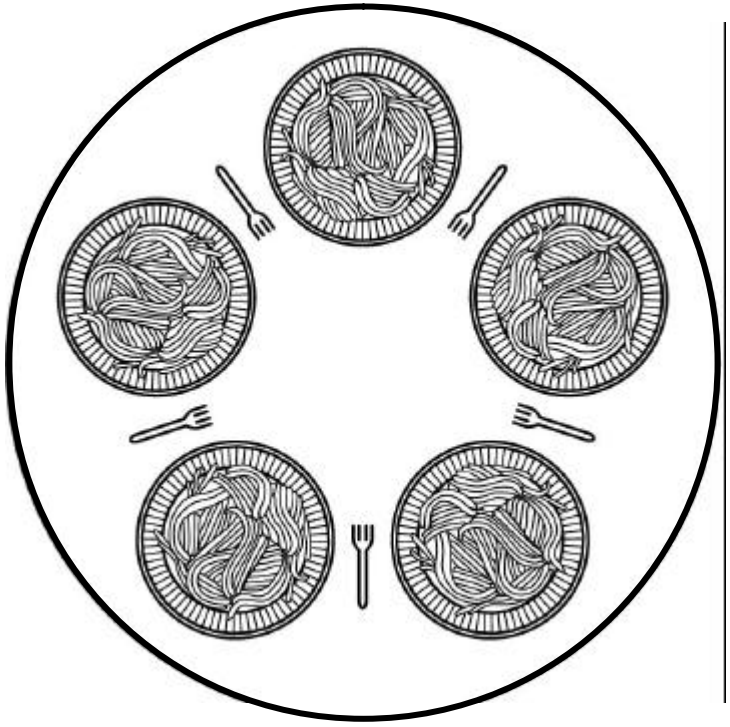
# Barreiras




- Uso de uma barreira
  a) processos se aproximando de uma barreira
  b) todos os processos, exceto um, bloqueados pela barreira
  c) último processo chega, todos passam

# Jantar dos Filósofos (1)

- Filósofos comem/pensam
- Cada um precisa de 2 garfos para comer
- Pega um garfo por vez
- Como prevenir deadlock



# Jantar dos Filósofos (2)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

Uma solução errada para o problema do jantar dos filósofos

# Jantar dos Filósofos (3)

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

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

Uma solução para o problema do jantar dos filósofos (parte 1)

# Jantar dos Filósofos (4)

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
        test(i);                        /* try to acquire 2 forks */
        up(&mutex);                     /* exit critical region */
        down(&s[i]);                    /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = THINKING;            /* philosopher has finished eating */
        test(LEFT);                     /* see if left neighbor can now eat */
        test(RIGHT);                    /* see if right neighbor can now eat */
        up(&mutex);                     /* exit critical region */
}

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

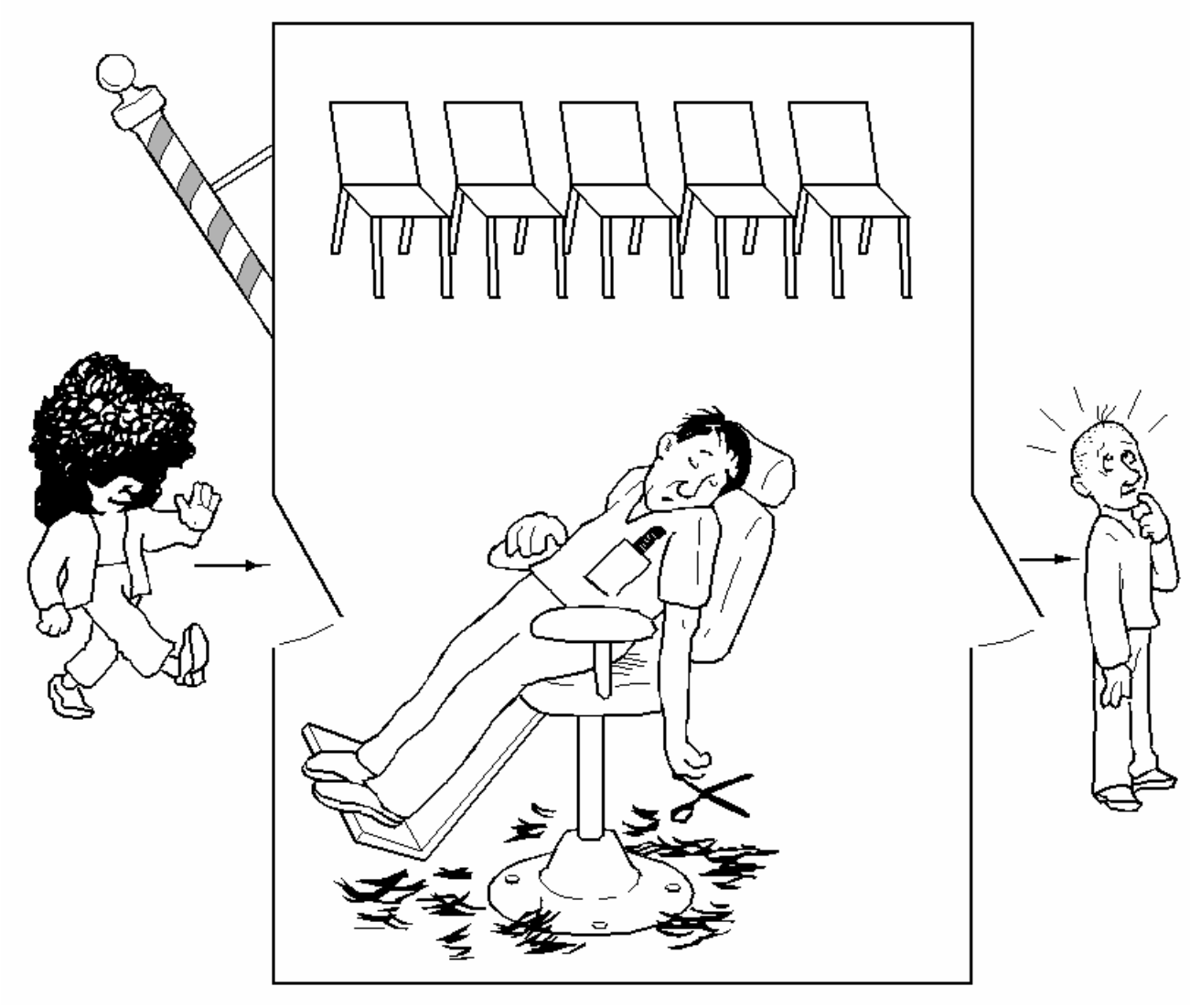
Uma solução para o problema do jantar dos filósofos (parte 2)

# O Problema dos Leitores e Escritores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        use_data_read( );               /* noncritical region */
    }
}


void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

Uma solução para o problema dos leitores e escritores

# O Problema do Barbeiro Sonolento (1)



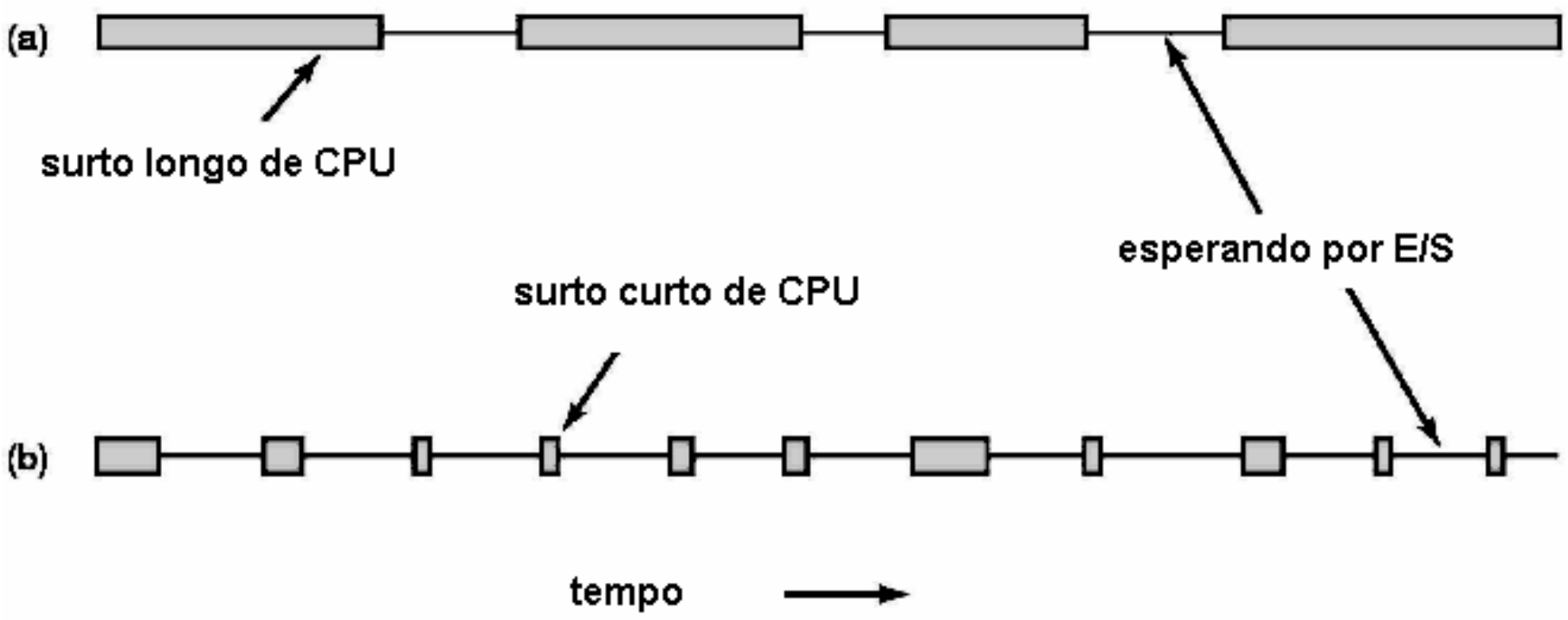
# O Problema do Barbeiro Sonolento (2)

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting – 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair( );                /* cut hair (outside critical region) */
    }
}


void customer(void)
{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut( );             /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

Solução para o problema do barbeiro sonolento

# Escalonamento

## Introdução ao Escalonamento (1)




- 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

# Introdução ao Escalonamento (2)

**All systems**

- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

**Batch systems**

- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

**Interactive systems**

- Response time - respond to requests quickly
- Proportionality - meet users' expectations

**Real-time systems**

- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

## Objetivos do algoritmo de escalonamento

# Escalonamento em Sistemas em Lote (1)


Tempo de retorno

| | (a) | (b) |
|---|---|---|
| | 8 para A | 4 para B |
| | 12 para B | 8 para C |
| | 16 para C | 12 para D |
| | 20 para D | 20 para A |

56 | 4
14

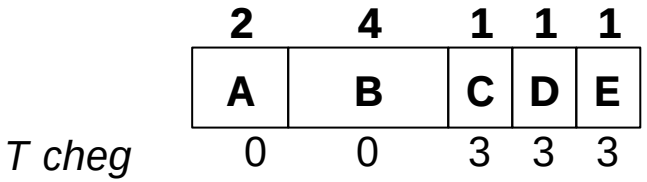
44 | 4
11

Um exemplo de escalonamento *job mais curto primeiro*

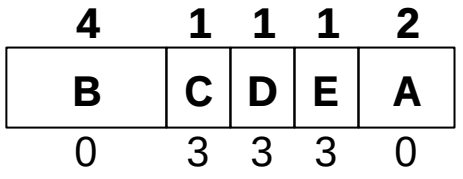
# Escalonamento em Sistemas em Lote (2)

## Job mais curto primeiro

(ótimo só quando todos os *jobs* estão disponíveis ao mesmo tempo)




(a)



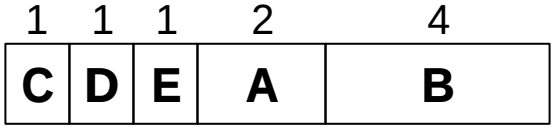

(b)

Tempo de retorno

| (a) | (b) |
|---|---|
| 2 para A | 4 para B |
| 6 para B | 2 para C |
| 4 para C | 3 para D |
| 5 para D | 4 para E |
| 6 para E | 9 para A |

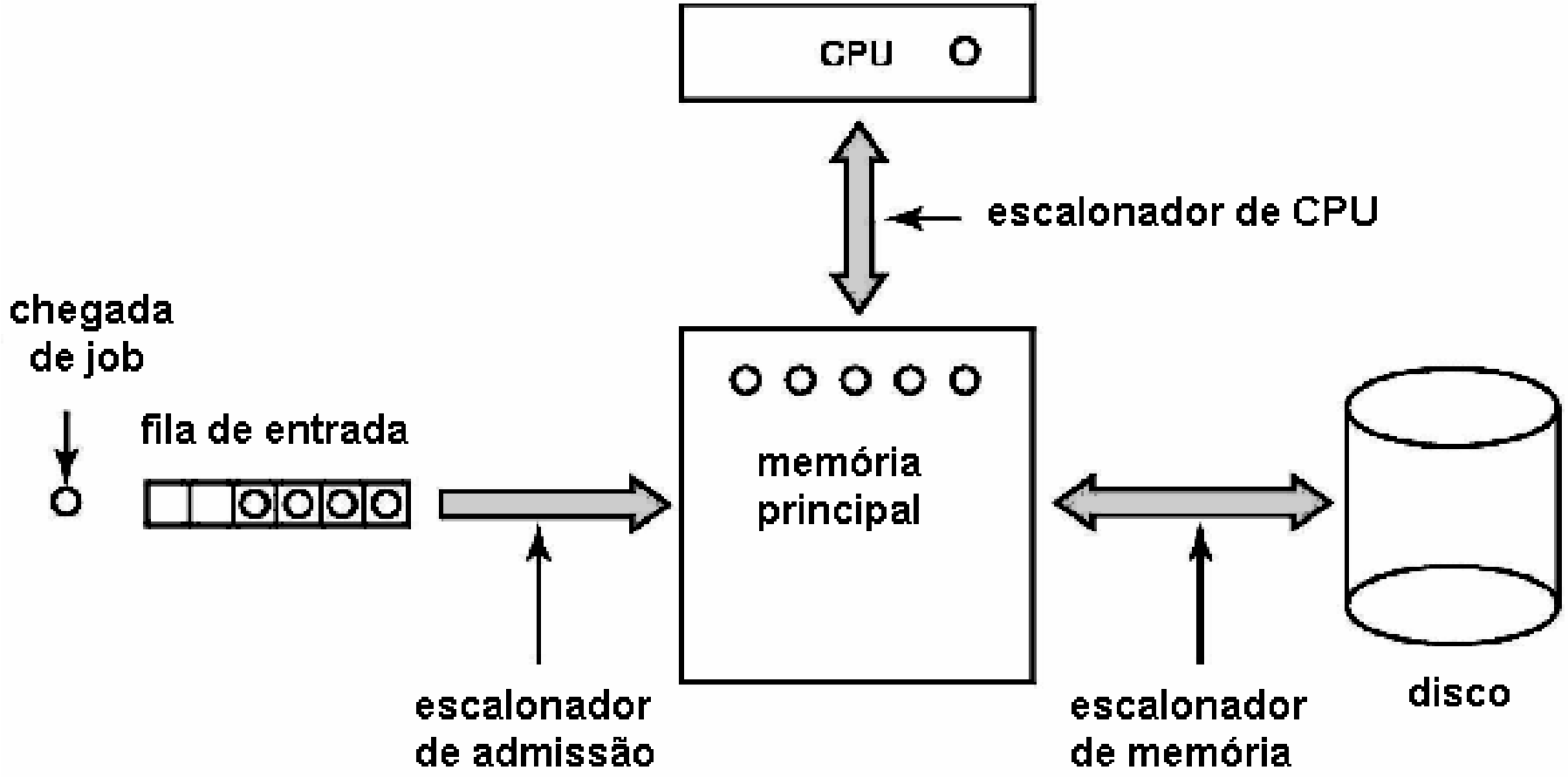
23 | 5 — 4.6

22 | 5 — 4.4



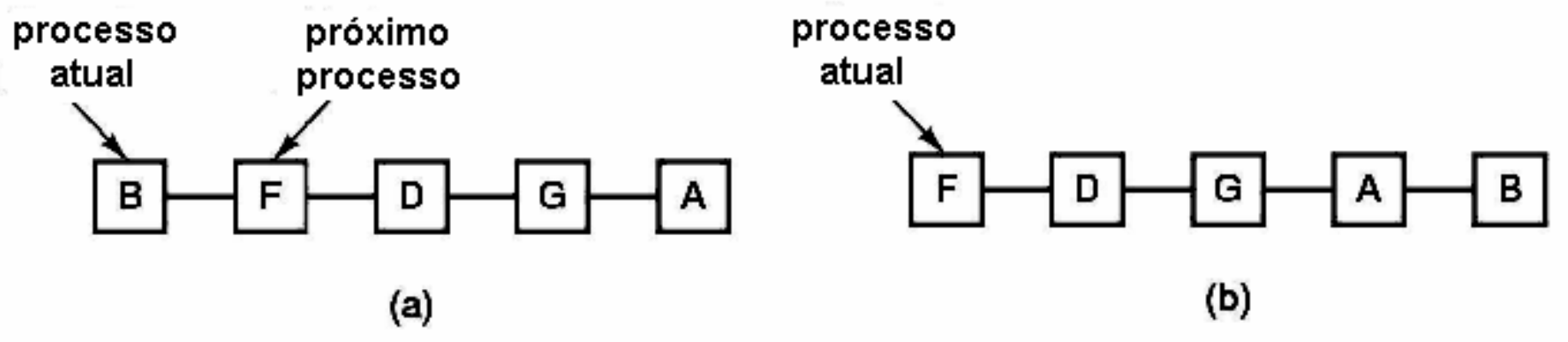

1 + 2 + 3 + 5 + 9 = 20 | 5 — 4

# Escalonamento em Sistemas em Lote (3)



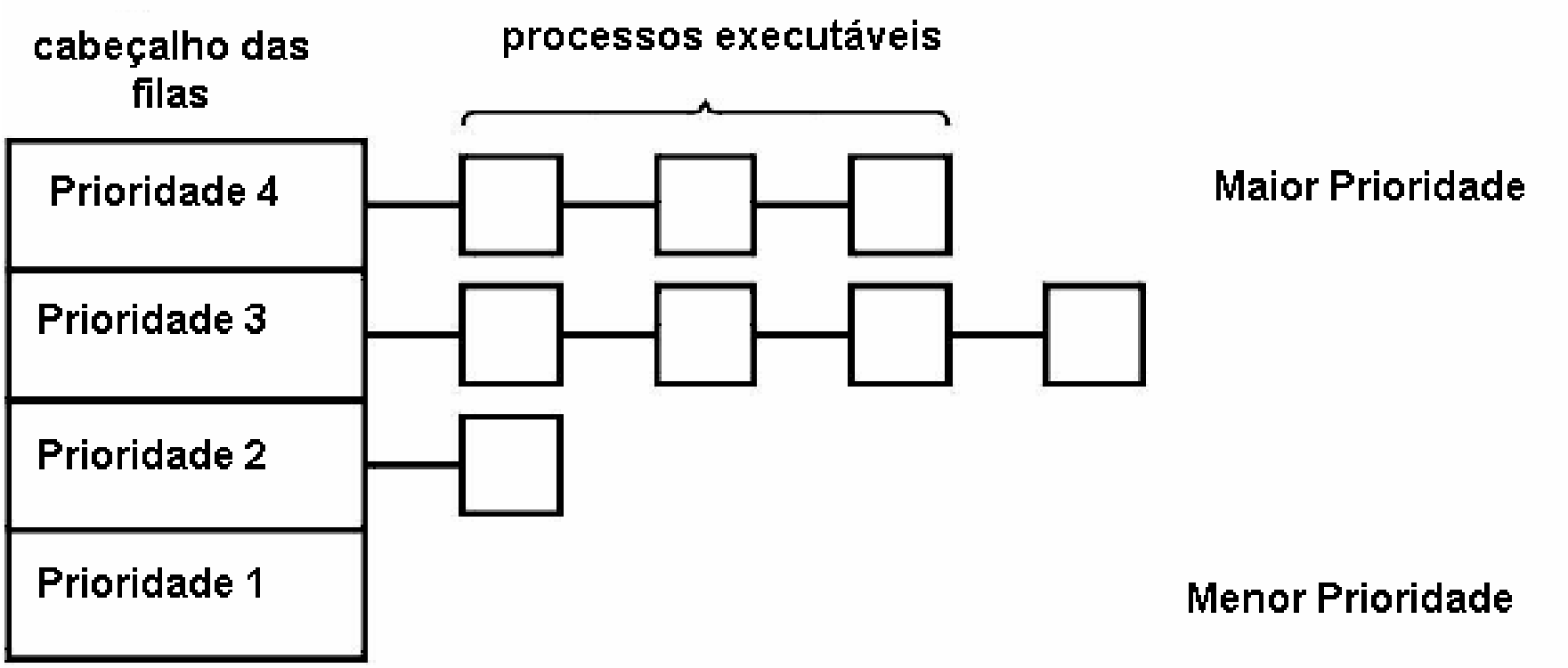

Escalonamento em três níveis

# Escalonamento em Sistemas Interativos (1)




- Escalonamento por 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 em Sistemas de Tempo-Real

## Sistema de tempo-real escalonável

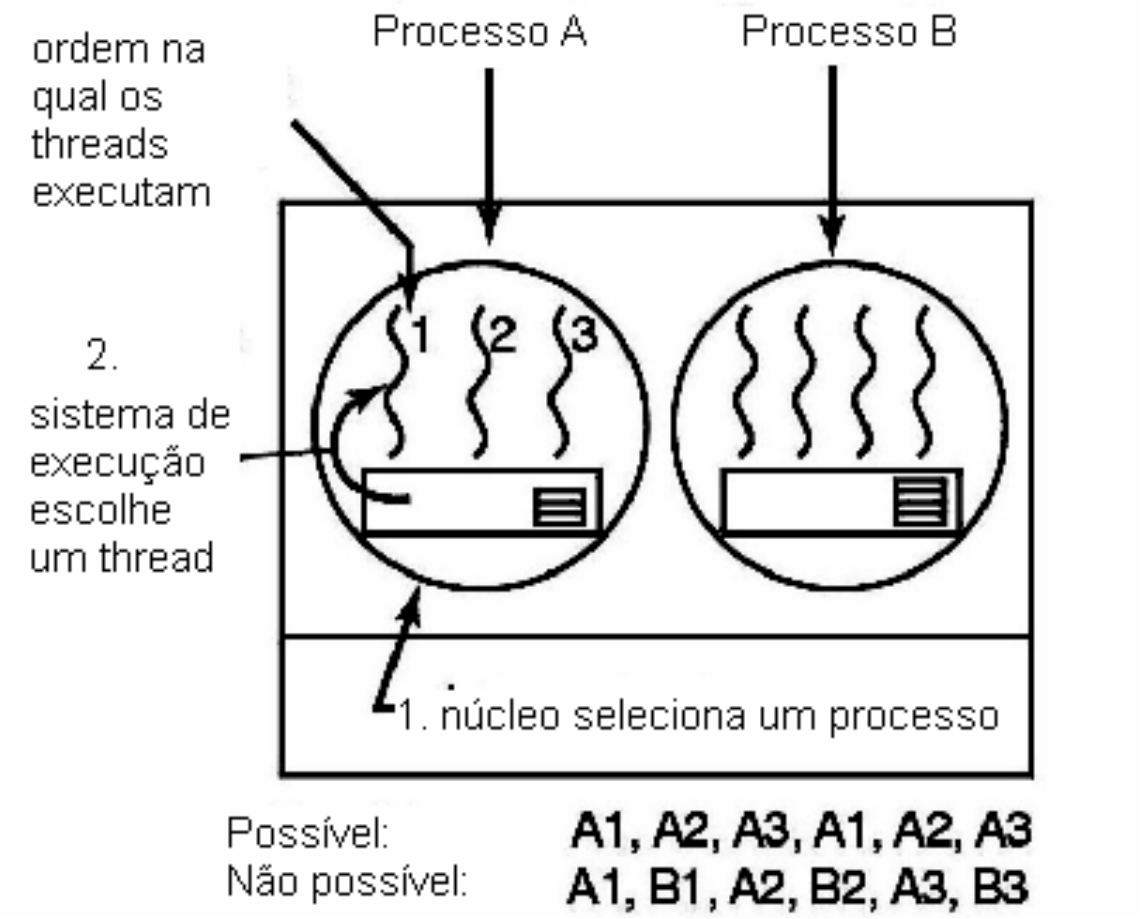
- Dados
  - *m* eventos periódicos
  - evento *i* ocorre dentro do período $P_i$ e requer $C_i$ segundos
- Então a carga poderá ser tratada somente se

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

# Política *versus* Mecanismo

- Separa o que é permitido ser feito do como é feito
  - um processo sabe quais de seus threads filhos são importantes e precisam de prioridade
- Algoritmo de escalonamento parametrizado
  - mecanismo no núcleo
- Parâmetros preenchidos pelos processos do usuário
  - política estabelecida pelo processo do usuário

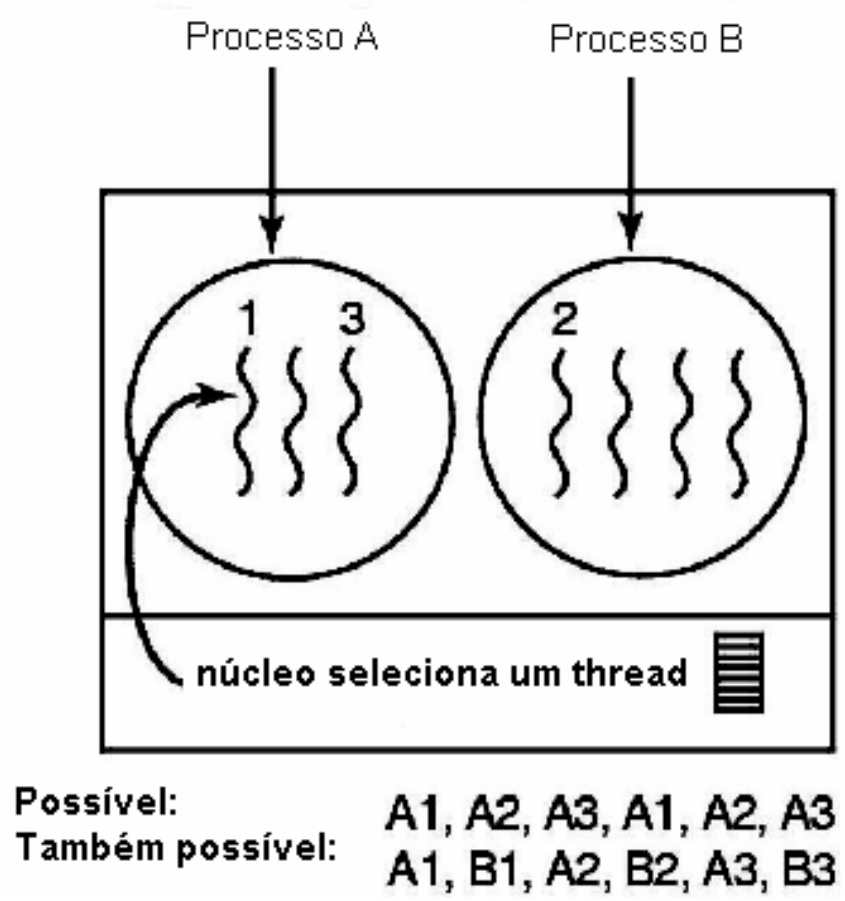
# Escalonamento de Threads (1)




Possível escalonamento de threads de usuário

- processo com quantum de 50-mseg
- threads executam 5 mseg por surto de CPU

# Escalonamento de Threads (2)




Possível escalonamento de threads de núcleo

- processo com quantum de 50mseg
- threads executam 5 mseg por surto de CPU