# Uso de Thread

**Definição: "tipo de miniprocesso dentro de um processo"**

Em sistemas operacionais tradicionais:

- **Cada processo: → um espaço de endereçamento**
  **→ um único thread de controle**

Em algumas situações são desejáveis:

- Múltiplos threads de controle (mesmo espaço de endereçamento)
- Execução "quase-paralelo"
  - **Ex: Processador de texto usado na confecção de um livro**

1. **um thread cuida do que está sendo digitado**
2. **outro thread cuida da formatação**
3. **outro thread cuida de salvar no disco com uma certa periodicidade**

# Uso de Thread

**Razões para a existência de threads:**

**Argumento principal:** **em muitas aplicações ocorrem múltiplas atividades ao mesmo tempo. Algumas dessas atividades podem bloquear de tempos em tempos**

**⇒ maior simplicidade se decompormos uma aplicação em múltiplos threads sequenciais que executam em *quase-paralelo***

***Com processos:*** **interrupções, temporizadores e chaveamentos de contextos ⇒** processos paralelos.

***Com threads:*** ***capacidade de entidades paralelas compartilharem um mesmo espaço de endereçamento e todos os seus dados entre as mesmas***

# Uso de Thread

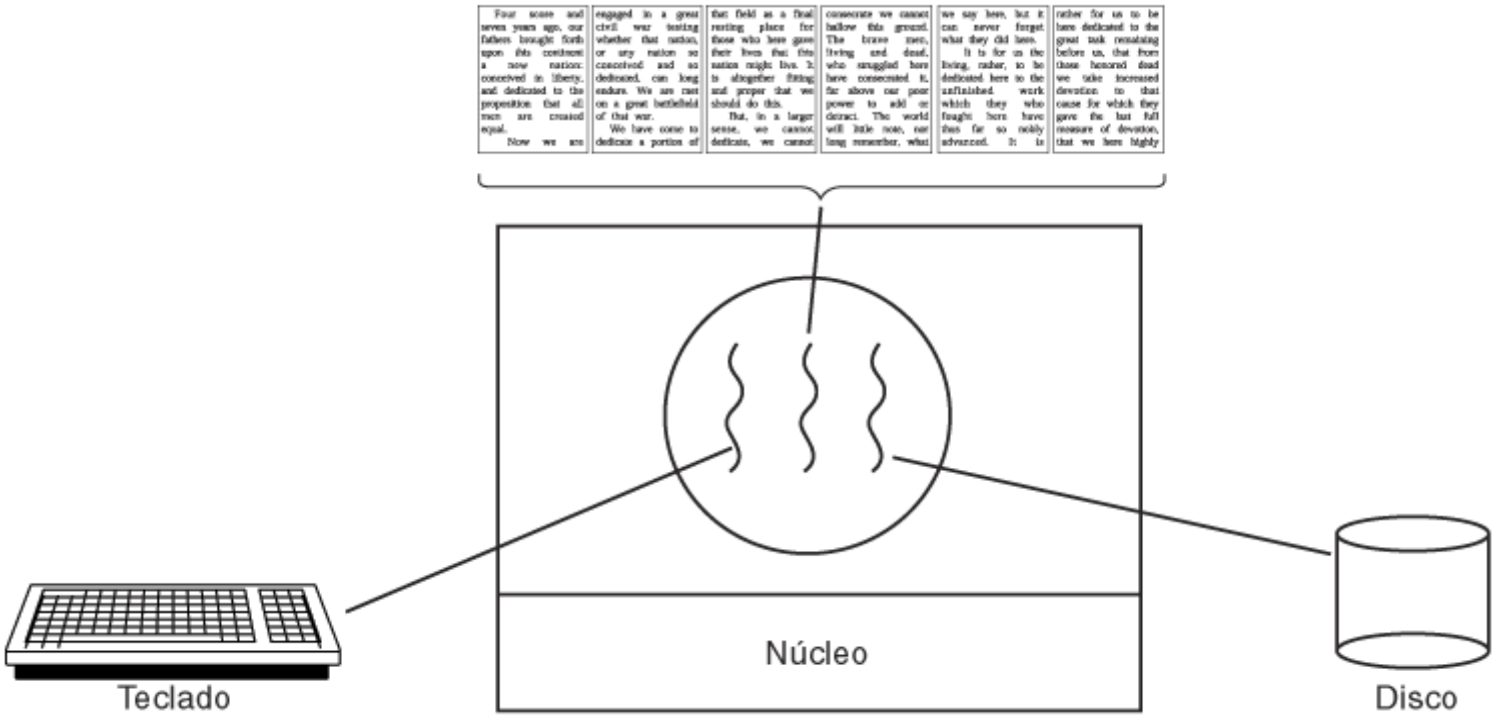
- **2º Argumento:** **são mais fáceis de criar e destruir** do que os processos, pois não têm quaisquer recursos associados a eles. Em alguns sistemas criar um thread pode ser até 100 vezes mais rápido do que criar um processo

  ⇒ **número de threads pode se alterar rapidamente**

- **3º Argumento:** **desempenho**: quando há grande quantidade de computação e de E/S, os threads permitem que essas atividades se sobreponham e acelerem a aplicação

- **4º Argumento:** **Fundamentalmente úteis em sistemas com múltiplas CPUs** ⇒ ***paralelismo real é possível.***

# Uso de Thread - exemplo




**Um processador de texto com três threads:**

**1. interativo (teclado e mouse);** **2. reformata o texto;**

**3. grava periodicamente no disco** **E se fosse só uma thread?**

# Uso de Thread - exemplo

**Observação importante:**

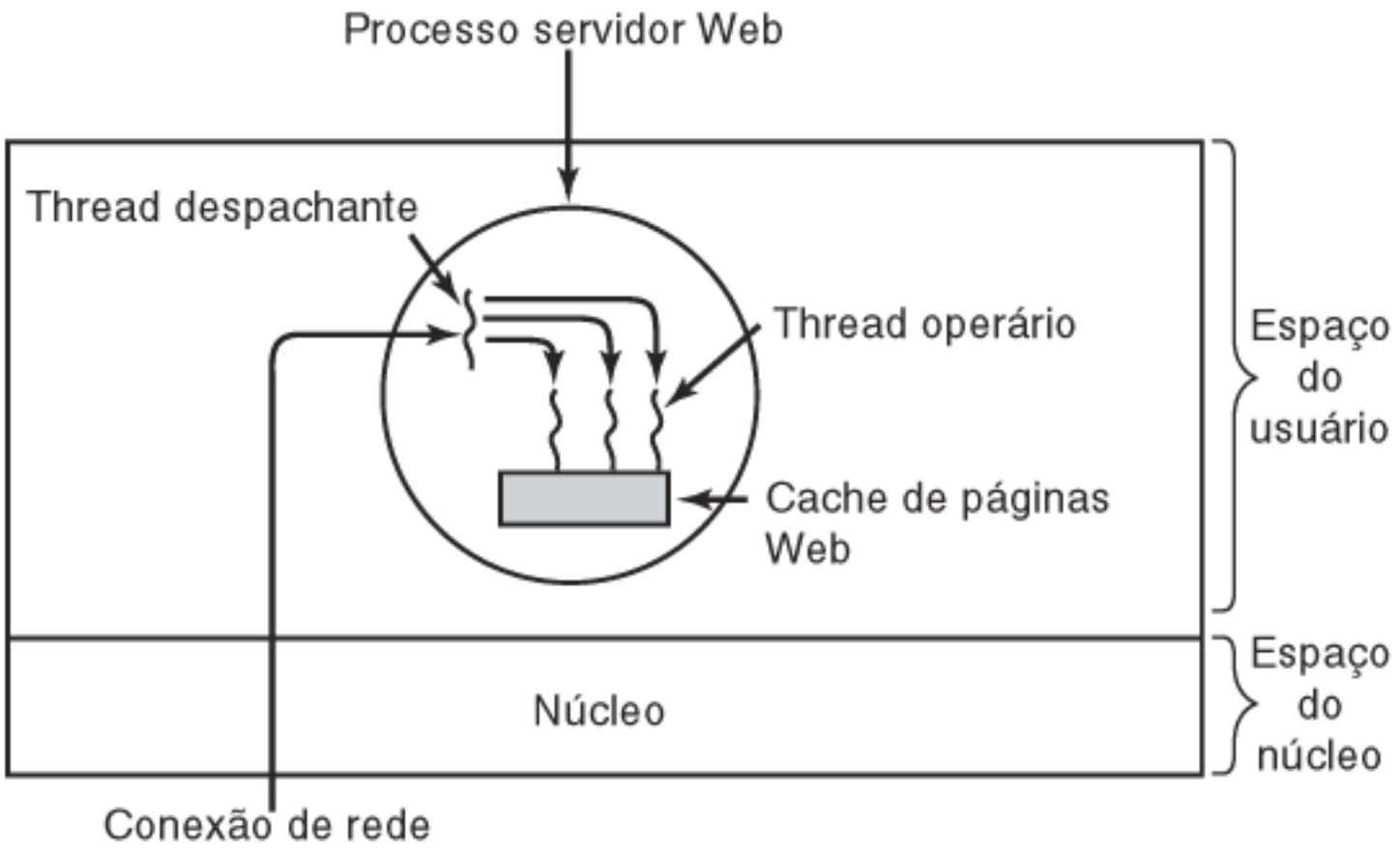
No exemplo anterior, o uso de **3 processos separados não funcionaria**, pois repare que os 3 threads precisam operar sobre o **mesmo documento.**

⇒ 3 threads compartilham uma **memória comum** e, desse modo, têm todo o acesso ao documento que está sendo editado

**Outro exemplo:**

*Planilha eletrônica* ⇒ **edição, cálculo e backup periódico**

# Uso de Thread – outro exemplo




**Ex. 3: Um servidor Web com múltiplos threads**

# Uso de Thread

**Código simplificado para o servidor Web:**

a) Thread despachante

b) Um ou mais ***threads operários***

```
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)

***E se fosse um único thread?***

# Uso de Thread

*Terceira opção:* **read (leitura em disco) não bloqueante**

- Um thread $\Rightarrow$ examina solicitação que chegou
- Se não for satisfeita pelo cache $\Rightarrow$ operação de disco
- Registra o estado da solicitação numa tabela e trata próximo evento, que pode ser:
  - **Uma nova solicitação**
  - **Resposta do disco a uma solicitação anterior terá a forma de um sinal de *interrupção a ser tratada pelo SO***

**Observação:** estado deve ser salvo na tabela e atualizado cada vez que o servidor chaveia entre solicitações

$\Rightarrow$ ***Máquina de estados finitos***

***(simula threads: é mais difícil de programar)***

# Uso de Thread

**Em resumo, existem três maneiras de construir um servidor:**

1. ***Vários threads***: facilidade de implementação e melhora no desempenho;
2. ***Único thread***: mantém simplicidade mas piora desempenho;
3. ***Read não bloqueante***: programação mais difícil, mas melhora o desempenho

| 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 |

# Uso de Thread

**Outro exemplo:**

***Aplicações que lêem, processam e escrevem grande qtde de dados***

⇒ *lê bloco de dados do disco, processa-o e escreve novamente no disco*

***Com chamadas de sistema com bloqueio ⇒ CPU ociosa durante a leitura/escrita***

***Com 3 threads ⇒ processamento "paralelo"***

- ✓ thread **lê** do disco e coloca no buffer de entrada
- ✓ thread **processa os dados** e os coloca no buffer de saída
- ✓ thread **escreve** os dados de saída no disco

# O Modelo de Thread Clássico

**Modelo de processos - baseado em dois conceitos independentes:**

- ***agrupamento de recursos*** ***(código, dados, arquivos abertos, processos filhos, etc)***
- ***execução***

**Threads - relacionados à *execução***

***Um thread contém:*** ***PC, registradores com variáveis atuais de trabalho, pilha com história da execução***

**Conceitos diferentes:**

- ***Processos*** **são usados para agrupar recursos;**
- ***Threads*** **são entidades escalonadas para execução sobre a CPU**

# O Modelo de Thread Clássico

**Característica principal dos threads:**

Múltiplas execuções ***independentes*** ocorrem no ***mesmo ambiente do processo (mesmo espaço de endereçamento)*****,** com alto grau de independência entre elas

***multithread*** **-** existência de múltiplos threads no mesmo processo.

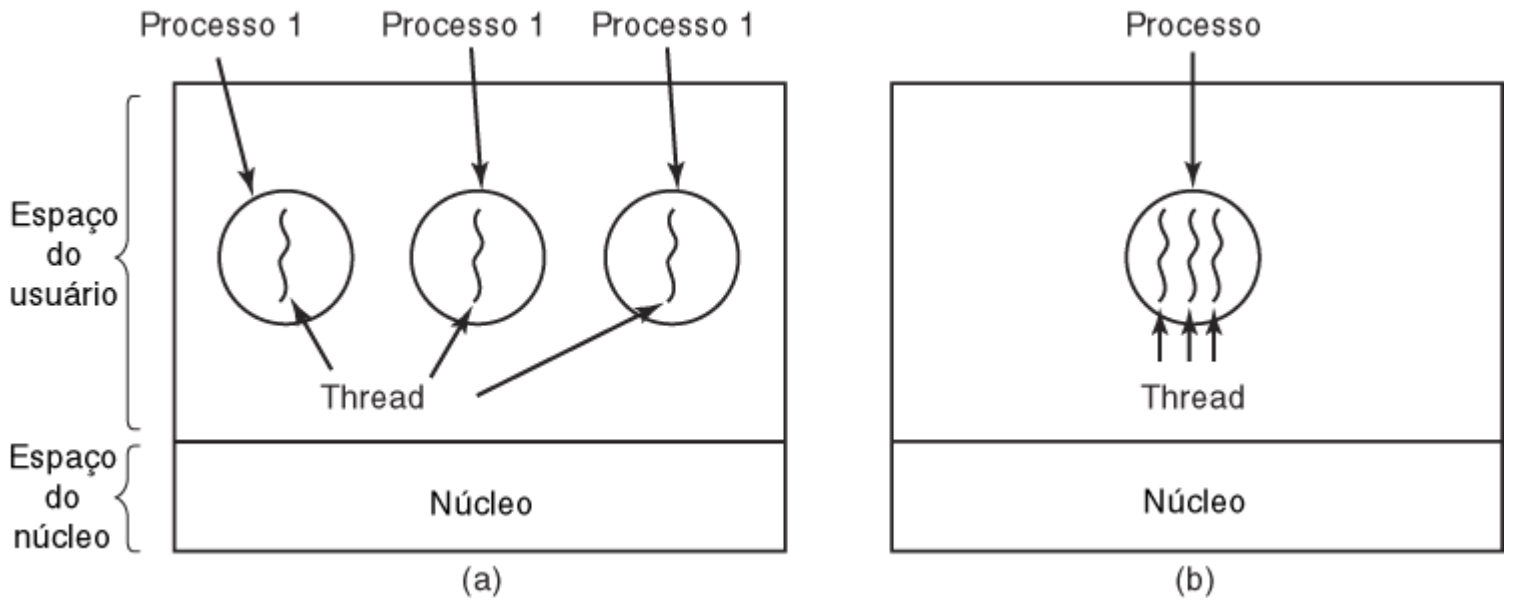
***Obs importante***:

***múltiplos threads em paralelo em um processo é análogo a múltiplos processos em paralelo em um computador:***

**1º caso:** threads compartilham o mesmo espaço de endereçamento e alguns recursos

**2º caso:** processos compartilham memória, discos, etc

# O Modelo de Thread Clássico




a) Três processos: cada um com um thread

b) Um processo com três threads

# O Modelo de Thread Clássico

**Observações importantes:**

1. Um sistema multiprogramado, ao alternar entre vários processos, dá a impressão de processos seqüenciais distintos executando em paralelo

2. O multithread funciona do mesmo modo. A CPU alterna rapidamente entre os threads dando a impressão de que os threads estão executando em paralelo

# O Modelo de Thread Clássico

3. Threads distintos em um processo não são tão independentes quanto processos distintos: todos os threads tem o mesmo espaço de endereçamento, o que significa que eles compartilham as mesmas variáveis globais

4. Um thread pode ler, escrever ou apagar completamente a pilha de outro thread. Não há proteção porque é impossível e porque não é necessário. Por que?

**Resposta**: Processos diversos podem ser de usuários diversos. Threads que pertencem ao mesmo processo **foram criados para cooperarem entre si** e não para competirem.

# O Modelo de Thread Clássico

| 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 | |

Coluna 1: Items compartilhados por todos os threads em um processo (propriedades deste)

Coluna 2: Itens privativos de cada thread

# O Modelo de Thread Clássico

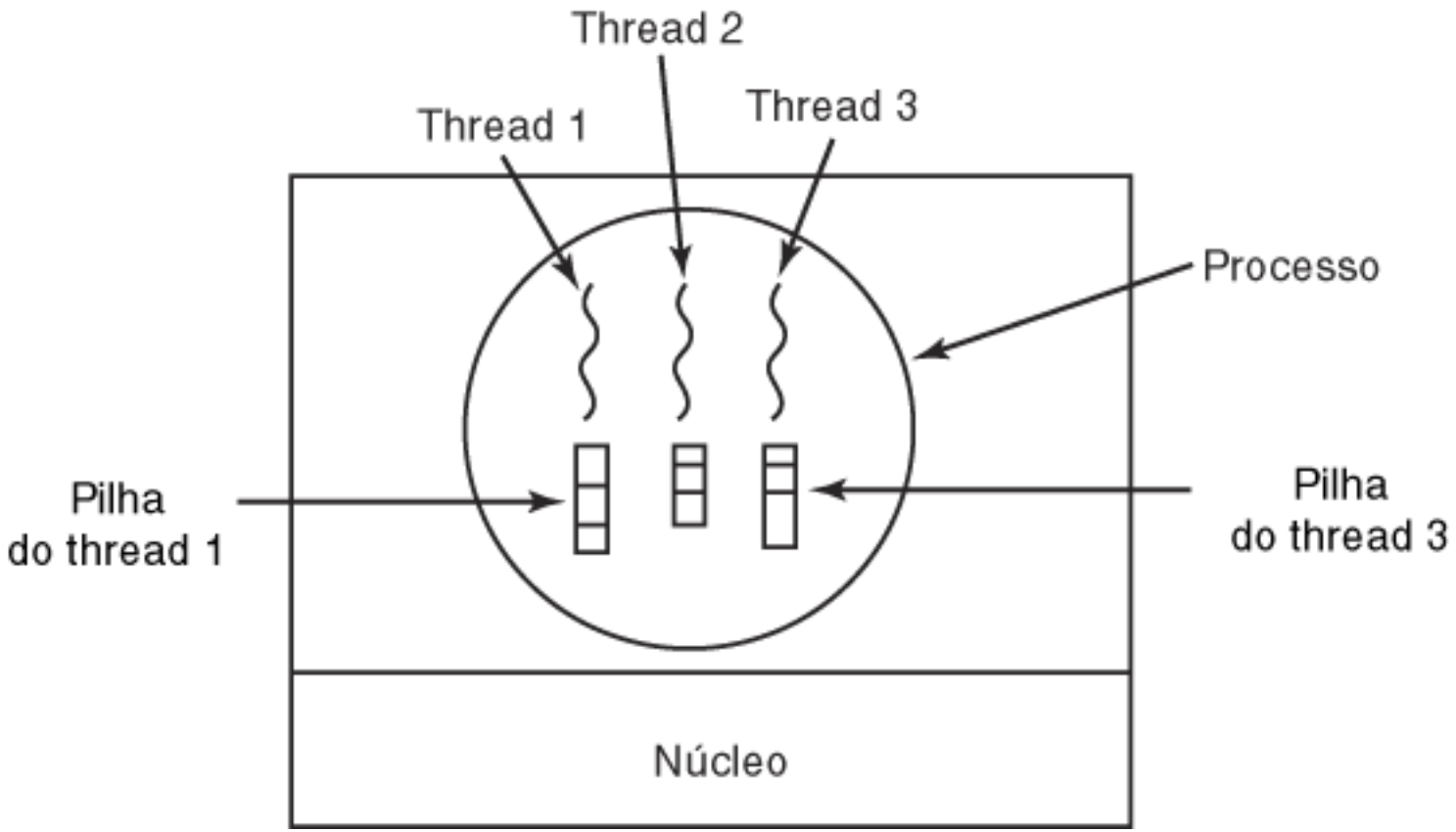
**Objetivo principal de múltiplos threads:**

***Capacidade de compartilhar um conjunto de recursos, podendo cooperar na realização de uma tarefa***

- Um thread também pode estar em um dos vários estados: ***execução, bloqueado, pronto* ou *finalizado (mesmas transições)***

- Cada thread tem sua própria pilha, onde cada pilha contém uma estrutura para cada rotina chamada.
  - ✓ **variáveis locais**
  - ✓ **endereço de retorno**

# O Modelo de Thread Clássico




**Cada thread tem sua própria pilha (cada thread pode chamar rotinas diferentes e ter um histórico de execução diferente)**

# O Modelo de Thread Clássico

Processos começam com um único thread

→ Chamadas de Sistema para criar e terminar threads:

***thread_create*** **e** ***thread_exit***

- ***thread_join*** → faz um thread esperar pela saída do thread que foi chamado
- ***thread_yield*** → faz um thread "abrir mão" da CPU para ceder a outro.

Obs: Não existe interrupção de relógio para forçar a troca *de threads*

⇒ um ***thread*** deve, deliberadamente, ceder a CPU para outro ***thread*** poder executar

# O Modelo de Thread Clássico

Possíveis problemas com threads:

Problema 1: um thread fecha um arquivo enquanto outro thread está lendo o mesmo arquivo.

Problema 2: alocação simultânea de memória por dois threads que identificam pouca memória (mesmas posições).

***O projetista do SO deve tomar cuidado com esses tipos de problemas quando estiver programando!***

# Implementação de Threads

Existem 3 formas de se implementar um pacote de threads

1. no espaço do usuário;
2. no núcleo do SO;
3. híbrida

Primeiro caso:

1. **Pacote de threads totalmente no espaço do usuário:**

   **⇒ núcleo não é informado sobre eles;**

# Implementação de Threads no espaço do usuário

1ª vantagem:

Pacote pode ser implementado em um SO que não suporte threads

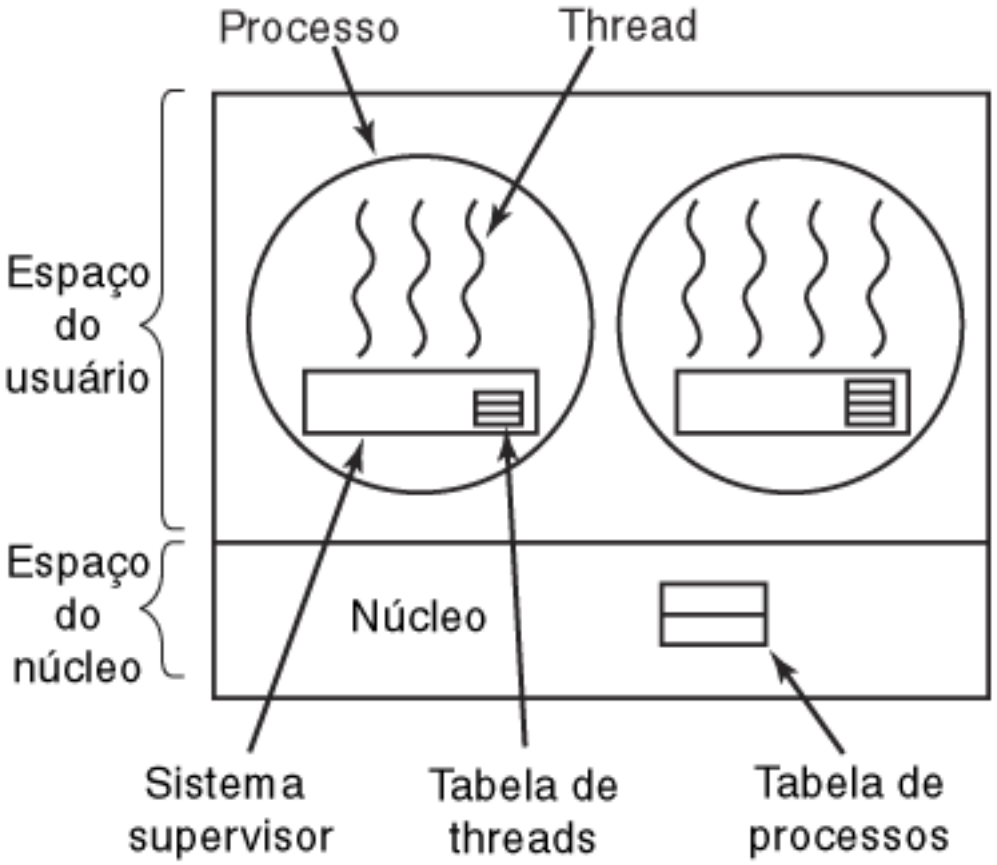
Características:

- Threads são implementadas através de ***bibliotecas*** sobre um ***sistema de tempo de execução (coleção de rotinas que gerenciam os threads)***
- Cada processo precisa de sua própria ***tabela de threads*** para manter o **controle dos *threads* naquele processo** (análoga à tabela de processos no núcleo de um SO): diferente nos tipos de informações

A tabela de threads é gerenciada por um ***sistema de tempo de execução (ou supervisor):***

**gerencia as informações necessárias para reiniciar as threads quando estes vão para o estado *pronto ou bloqueado***

# Implementação de Threads no espaço do usuário




**Um pacote de threads de usuário**

# Implementação de Threads no espaço do usuário

**2ª vantagem:**

alternância entre os threads (procedimentos locais) é feita através de poucas instruções, não sendo necessário desviar o controle para o núcleo

⇒ **agiliza o escalonamento** das threads em, pelo menos, uma ordem de grandeza

**3ª vantagem:**

rotinas que salvam o estado do thread e que chamam o escalonador são locais (salvam o estado na própria tabela de threads)

⇒ não necessitam de chamadas ao núcleo

⇒ sem necessidade de chaveamento de contexto e esvaziamento de cache

***Torna o escalonamento do thread muito mais rápido***

# Implementação de Threads no espaço do usuário

**4ª vantagem:**

Cada processo/usuário também pode ter o seu próprio algoritmo de escalonamento personalizado

**5ª vantagem:**

Escala melhor pois, se implementados no núcleo, seria necessário nova tabela e pilha no núcleo

**⇒ problema se o nº de threads for grande**

# Implementação de Threads no espaço do usuário

**Desvantagens:**

- ✓ **o bloqueio de um thread**, por ex., que esteja lendo o teclado (chamada de sistema ***read***), pode parar todos os outros threads. Opção: ***read*** não bloqueante
- ✓ **page fault:** se um thread causa uma falta de página, o núcleo do SO bloqueia o processo inteiro (lembrar que o SO não tem conhecimento das threads) até que a E/S do disco termine, mesmo que outros threads possam ser executados
- ✓ **Não existe controle do relógio** (núcleo do SO)

  **⇒ um thread pode executar indefinidamente devido a uma programação mal feita (a cessão da vez é voluntária)**
- ✓ Programadores normalmente querem threads em aplicações que fazem **constantes chamadas de sistema (ao núcleo)**

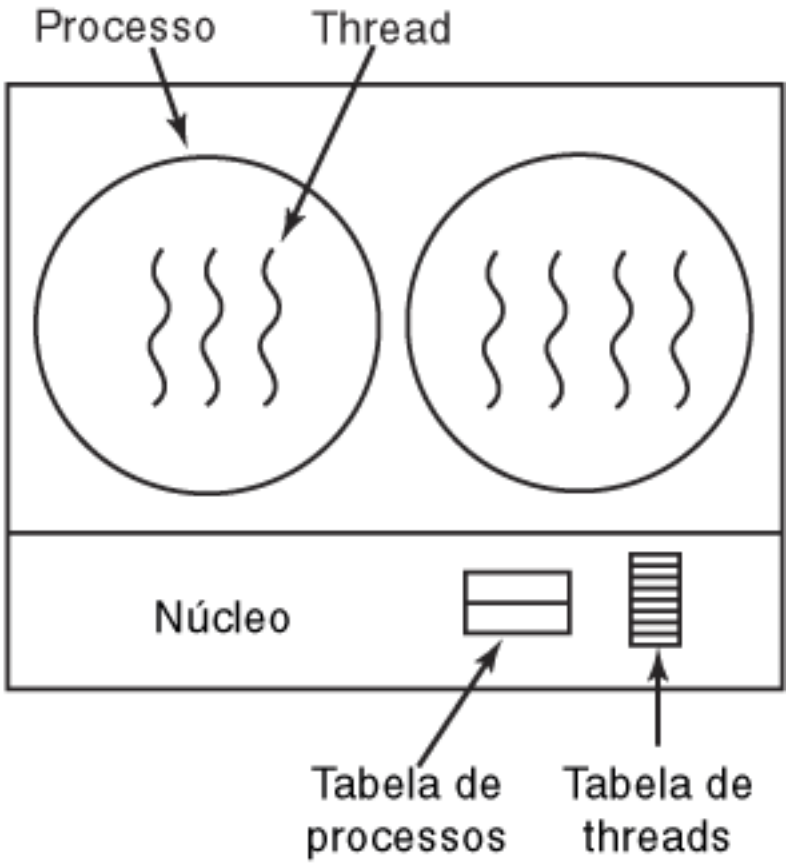
  **⇒ núcleo poderia fazer o chaveamento entre threads**

# Implementação de Threads no Núcleo

**Características:**

1. Não é necessário um supervisor
2. Não há tabelas de threads em cada processo: o núcleo do SO tem uma tabela de threads que acompanha todos os threads no sistema
3. O núcleo também mantém a tradicional tabela de processos
4. A criação ou destruição de um thread é feita através de uma chamada ao núcleo do SO, que a realiza e atualiza a tabela de threads (custo maior)
5. Quando um thread bloqueia, núcleo pode escalonar um thread do mesmo processo ou de um processo diferente

# Implementação de Threads no Núcleo




**Um pacote de threads gerenciado pelo núcleo**

# Implementação de Threads no Núcleo

**Vantagens:**

- ***Falta de página*** $\Rightarrow$ núcleo verifica facilmente se o processo tem threads prontos para execução e os escalona enquanto aguarda a página requisitada ser trazida do disco

- ***Reciclagem de threads destruídas*** $\Rightarrow$ aproveitamento das estruturas de dados de um thread destruído.

  $\Rightarrow$ ***evita sobrecarga adicional na criação de um novo thread***

- ***Threads de núcleo não exigem chamadas de sistemas novas, não bloqueantes***

# Implementação de Threads no Núcleo

**Algumas desvantagens:**

- Chamadas que bloqueiam um thread são **chamadas ao sistema**

  ⇒ **custo bem maior** que uma chamada para um procedimento do sistema supervisor

- **Ocorrência freqüente** de operações de **criação ou término de threads** causará uma **sobrecarga maior** (chamadas de sistema)

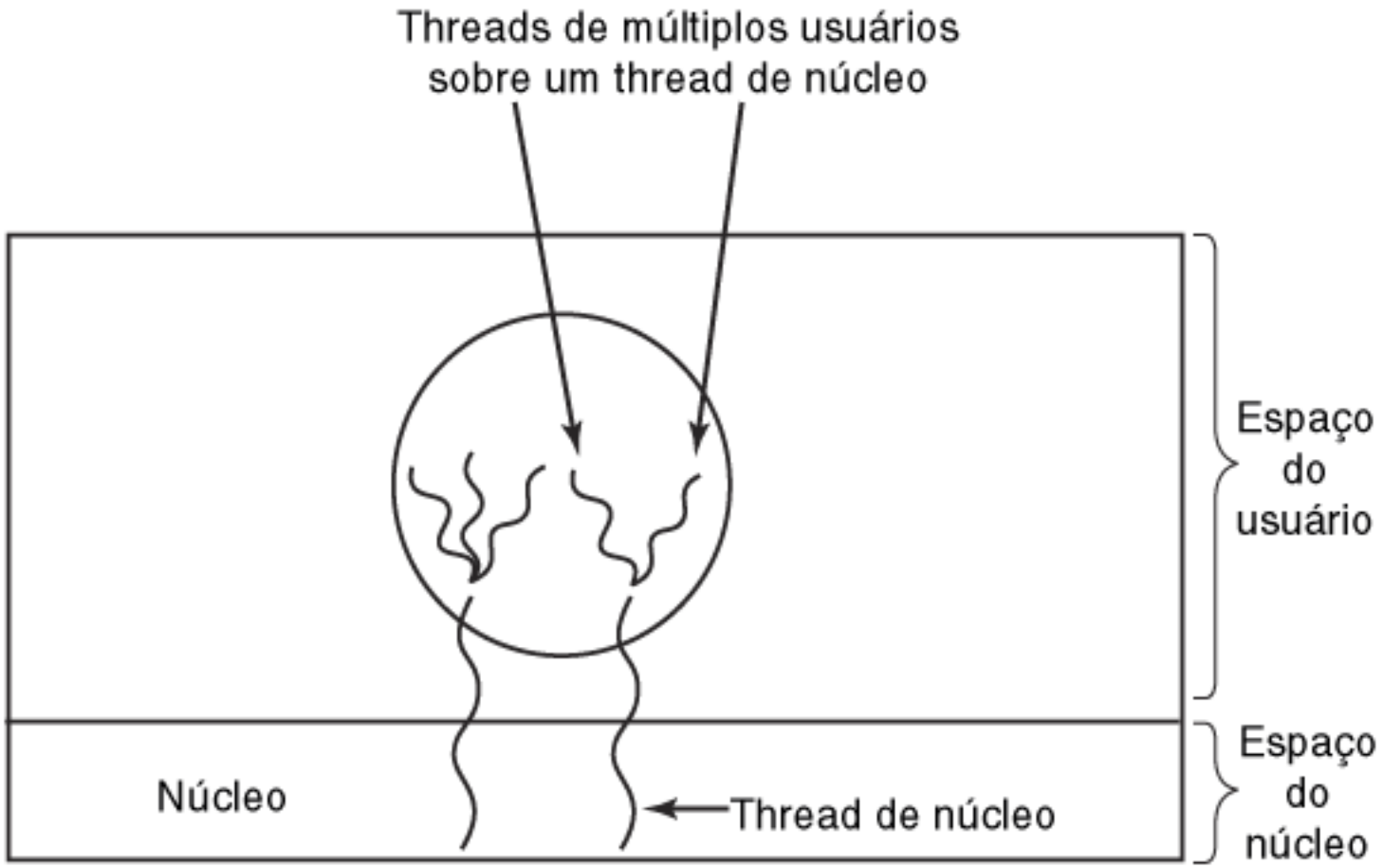
# Implementações Híbridas

**Ideia:** *juntar as vantagens dos dois tipos de threads*

**Características:**

- O núcleo sabe apenas sobre os threads de núcleo e os escalona
- Alguns desses threads podem ser o resultado da multiplexação de diversos threads de usuário que desejam o mesmo serviço do núcleo
- Os threads do usuário são criados, destruídos e escalonados no espaço do usuário
- Cada thread do núcleo atende a algum conjunto de threads de usuário que aguarda sua vez para usá-lo

# Implementações Híbridas




**Multiplexação de threads de usuário sobre threads de núcleo**

# Ativações do Escalonador

**Objetivo:**

**Imitar a funcionalidade dos threads de núcleo,**
**porém no espaço do usuário (melhor desempenho)**

⇒ **ganha o desempenho e flexibilidade de threads de usuário**

- Soluciona o problema de bloqueio de um thread em uma chamada ao sistema ou falta de página (pode executar outro thread)
- Evita transições usuário/núcleo desnecessárias ⇒ maior eficiência
- Núcleo atribui processadores virtuais para cada processo e deixa o sistema supervisor alocar os threads a esses "processadores" (nº varia).

**Obs: Esse mecanismo pode ser usado em um sistema multiprocessador, nos quais os processadores virtuais são CPUs reais**

# Ativações do Escalonador

Idéia básica**:**

Quando o núcleo sabe que um thread bloqueou (ex. falta de página), o supervisor é notificado pelo núcleo. Este passa ao supervisor (como parâmetros, na pilha) **o nº do thread que bloqueou e uma descrição do evento** ⇒ *UPCALL*

⇒ *supervisor escalona outra thread da lista de prontos*

⇒ *qdo bloqueio é suspenso, núcleo faz* ***nova upcall***

Problema:

Baseia-se fundamentalmente nos *upcalls* - o núcleo (camada inferior) chamando procedimentos no espaço do usuário (camada superior) ⇒ viola a estrutura de qualquer sistema em camadas

# Threads Pop-Up

Threads são úteis em sistemas distribuídos

***Ex: tratamento de mensagens que chegam da rede***

**Abordagem tradicional:**

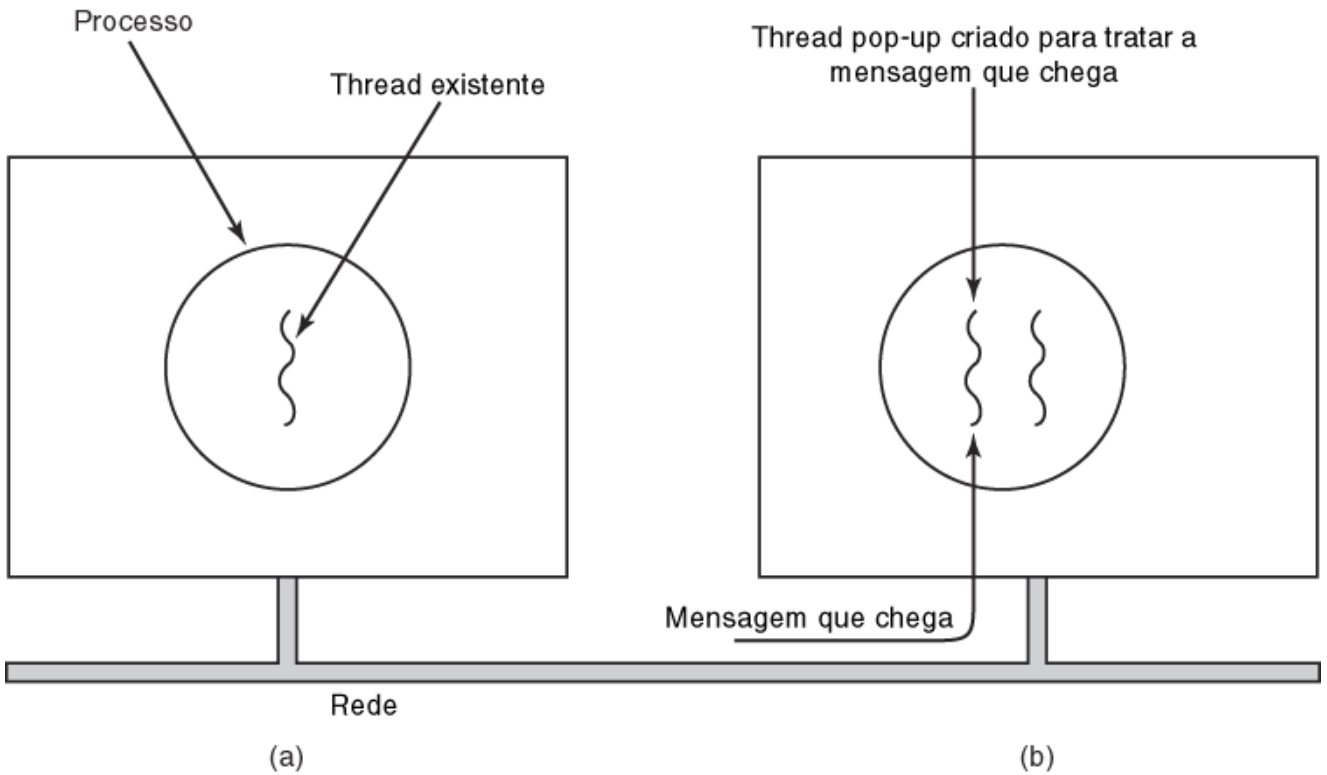
Bloqueia-se um thread ou processo em uma chamada ao sistema (*receive*) e aguarda-se a chegada da mensagem. O mesma thread recebe, abre e processa a mensagem

**Abordagem dos threads pop-ups**:

Chegada de uma mensagem faz com que um novo thread (*thread pop-up*) **seja criado** para tratar a mensagem

**Não possuem história (registradores, pilha, etc, pois acabaram de ser criados) ⇒ criação rápida**

# Threads Pop-Up




**Criação de um novo thread quando chega uma mensagem**

**(a) antes da mensagem chegar**

**(b) depois da mensagem chegar**

# Threads Pop-Up

**Vantagem:**

**latência baixa** entre a chegada da mensagem e o início do processamento

**Dúvidas:**

1. Em qual processo o thread vai executar?
2. Vai executar no núcleo ou no espaço do usuário?
   - executá-lo no núcleo é mais fácil e mais rápido
   - acesso fácil às tabelas e aos dispositivos de E/S necessários ao processamento de interrupções.

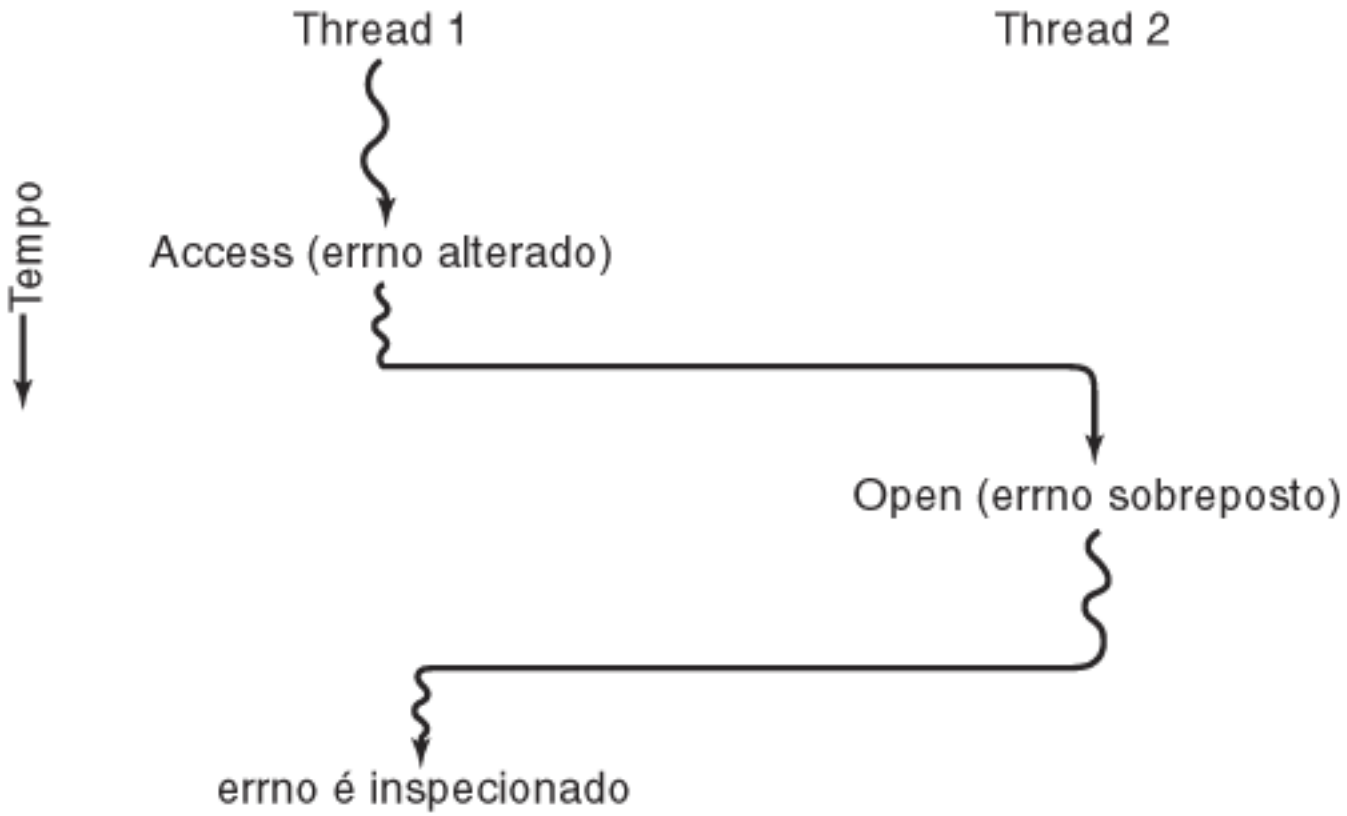
**Problema:** Erros no thread causam mais danos ao sistema.
Ex: execução lenta pode levar a perder dados que chegam

# Problema da Conversão de Código Monothread em Código Multithread

**Muitos dos programas existentes normalmente são escritos para processos monothread ⇒ conversão para multithread é complicada.**

- O código de um thread (assim como um processo) é composto por múltiplas rotinas (cada uma delas com variáveis locais, globais e parâmetros)
- **Problema:** variáveis que são globais a um thread (muitas rotinas dentro do thread as utilizam), mas que outros threads deviam deixá-las isoladas, podem ser alteradas por outros threads (**mesmo espaço de endereçamento**).
- **Exemplo**: variável *errno* do Unix. Falha em alguma chamada de sistema → código de erro colocado na variável *errno* (*thread* 1 na figura)
- Antes que o *thread* 1 possa tratar o erro (ler o valor da variável *errno*) o escalonador passa a CPU para o *thread* 2, que também executa uma chamada ao sistema que falha, sobrepondo o valor de *errno* → *thread* 1 lerá o valor errado de *errno*

# Problema da Conversão de Código Monothread em Código Multithread




**Conflitos entre threads sobre o uso de uma variável global**

# Problema da Conversão de Código Monothread em Código Multithread

**Soluções possíveis:**

Sol. 1: atribuir a cada *thread* suas variáveis globais privadas (ver slide seguinte)

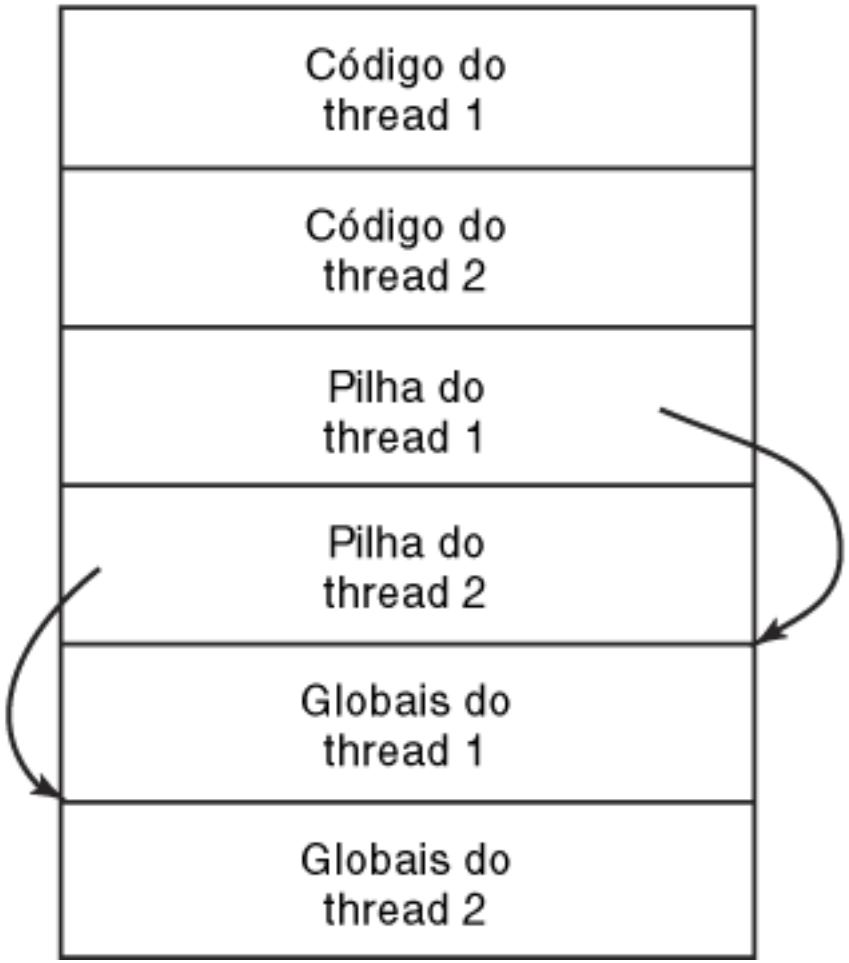
⇒ cada thread tem sua própria cópia de *errno*,

além de outras variáveis globais

Problema:

Maioria das linguagens de programação não suporta este *nível intermediário* de variável

# Problema da Conversão de Código Monothread em Código Multithread




**Threads poderiam ter variáveis globais privadas**

# Problema da Conversão de Código Monothread em Código Multithread

**Soluções possíveis:**

Sol. 2: criar novos procedimentos de biblioteca, com abrangência restrita ao thread, para criar, alterar e ler essas variáveis globais.

*ex:* ***create_global("bufptr");*** ***para alocar memória para o ponteiro chamado bufptr***

**A área alocada é de acesso restrito ao thread que a alocou**

***set_global("bufptr", &buf);*** *armazena o valor do ponteiro na posição de memória criada anteriormente*

***buf ptr = read_global("bufptr");*** *retorna o endereço armazenado na variável global*