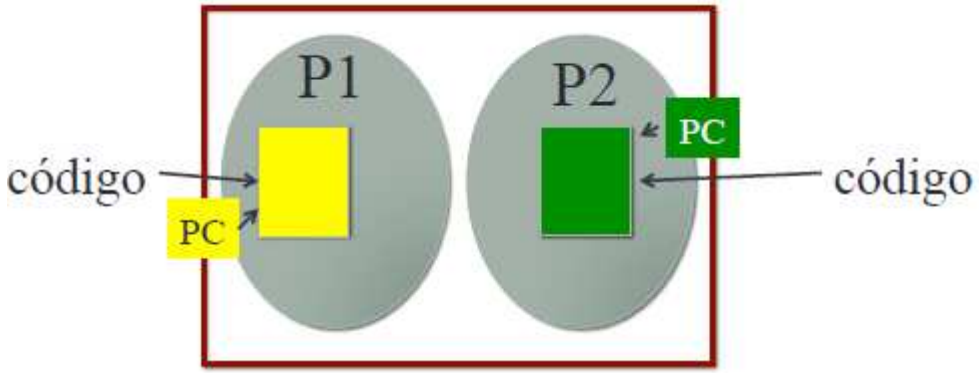
Sistemas Operacionais

Unidade 2: Gerência de Processos
- *Threads*

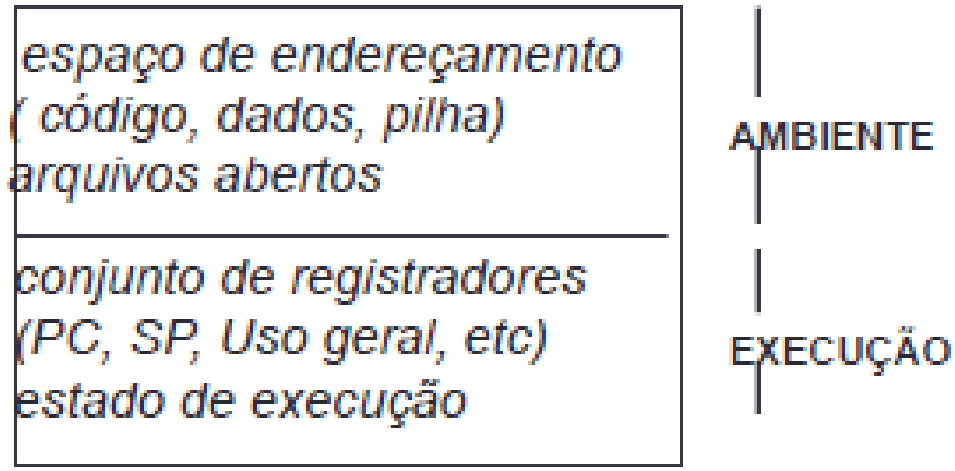
Modelos de Processos

- Classificação dos modelos de processos quanto ao custo de troca de contexto:
  - *Heavyweight* (Processo pesado)
  - *Lightweight* (Processo leve)
- Modelo de processo tradicional (*heavyweight*)
  - Neste caso, o processo é composto tanto pelo ambiente quanto pela execução.
- O modelo padrão de processos supõe o uso de apenas uma *thread* (um fluxo de execução) por processo.



Processos

- Itens da tabela de processos:



- A troca de contexto entre processos tradicionais é pesada para o sistema. Neste caso, o contexto é o ambiente e o estado de execução do processo.

O Processo é...

- Um programa em execução
  - Uma unidade de escalonamento
  - Um fluxo de execução
- Um conjunto de recursos (contexto) gerenciados pelo Sistema Operacional
  - Registradores (PC, SP, ...)
  - Memória
  - Descritores de arquivos
  - Etc...
- Logo, para manter o modelo de processos, o SO mantém a Tabela de Processos, com uma entrada para cada processo.

Processo

- Alguns autores chamam essas entradas para cada processo de Blocos de Controle de Acesso – PCB (*Process Control Blocks*);
- Essa entrada contém informações sobre o estado do processo, seu contador de programa, o ponteiro da pilha, a alocação de memória, os estados de seus arquivos abertos, sua informação sobre contabilidade e escalonamento e tudo mais sobre o processo que deva ser salvo para uma troca de contexto.

Threads

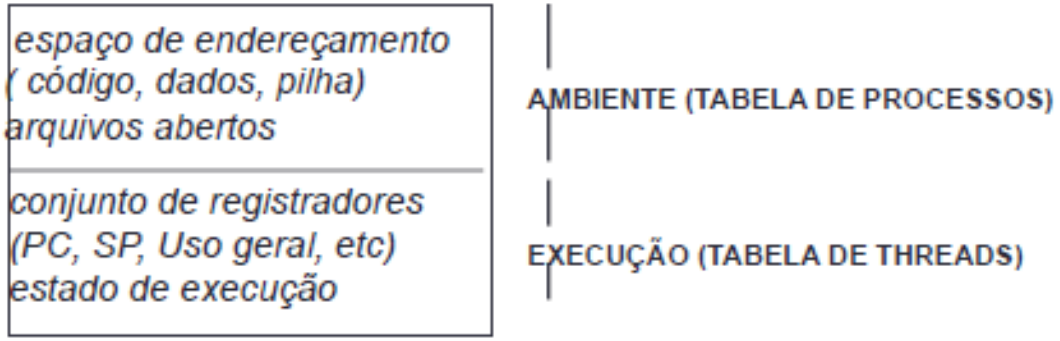
- As linhas de controle múltiplas foram criadas para permitir maior concorrência na execução dos processos.
- Assim, as *threads* são fluxos de execução, e compreendem:
  - Id – identificador da *thread*
  - Endereço da próxima instrução a ser executada
  - Conjunto de registradores em uso
  - Uma pilha de execução
- Compartilham com outras *threads* (do mesmo processo) recursos, tais como:
  - Trecho de código
  - Dados
  - Arquivos abertos
  - Etc...

Thread

- ***Thread***, às vezes denominadas processo leve (Lightweight Process - LWP) é a maneira de um programa dividir a si mesmo em duas ou mais tarefas simultâneas.
- O interessante está na execução de várias threads ao mesmo tempo, executando diferentes tarefas para o mesmo programa.
- Entretanto, as threads não são planejadas para existirem sozinhas – elas pertencem a processos tradicionais (Heavyweight Process – HWP).
- Neste modelo, a entidade processo é dividida em duas entidades: processo e *thread*. O processo corresponde ao ambiente e a *thread* corresponde ao estado de execução.

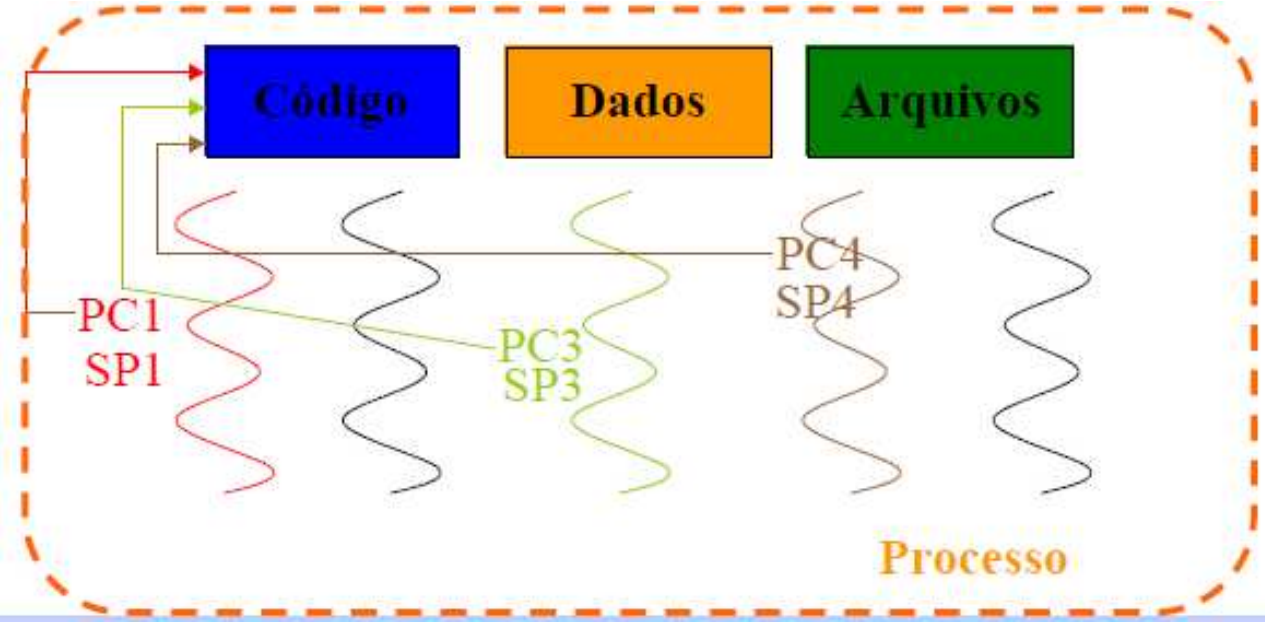
Thread

- No modelo leve, existem duas entidades:
  - Processo: armazena informações de ambiente
  - *Thread*: armazena informações de execução
- Assim, todo processo possui, pelo menos, uma *thread*.
- Logo, cada processo possui uma tabela de *threads* associada.



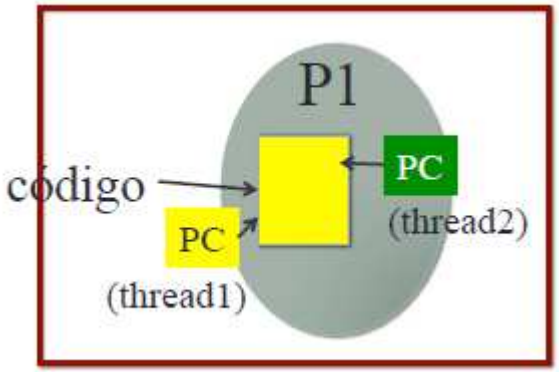
Thread

- Assim, as *threads:*
  - Compartilham recursos do PCB;
  - O PCB deve incluir a lista de *threads* de cada processos.



Thread

- No exemplo, tem-se 1 processo com 2 *threads* .

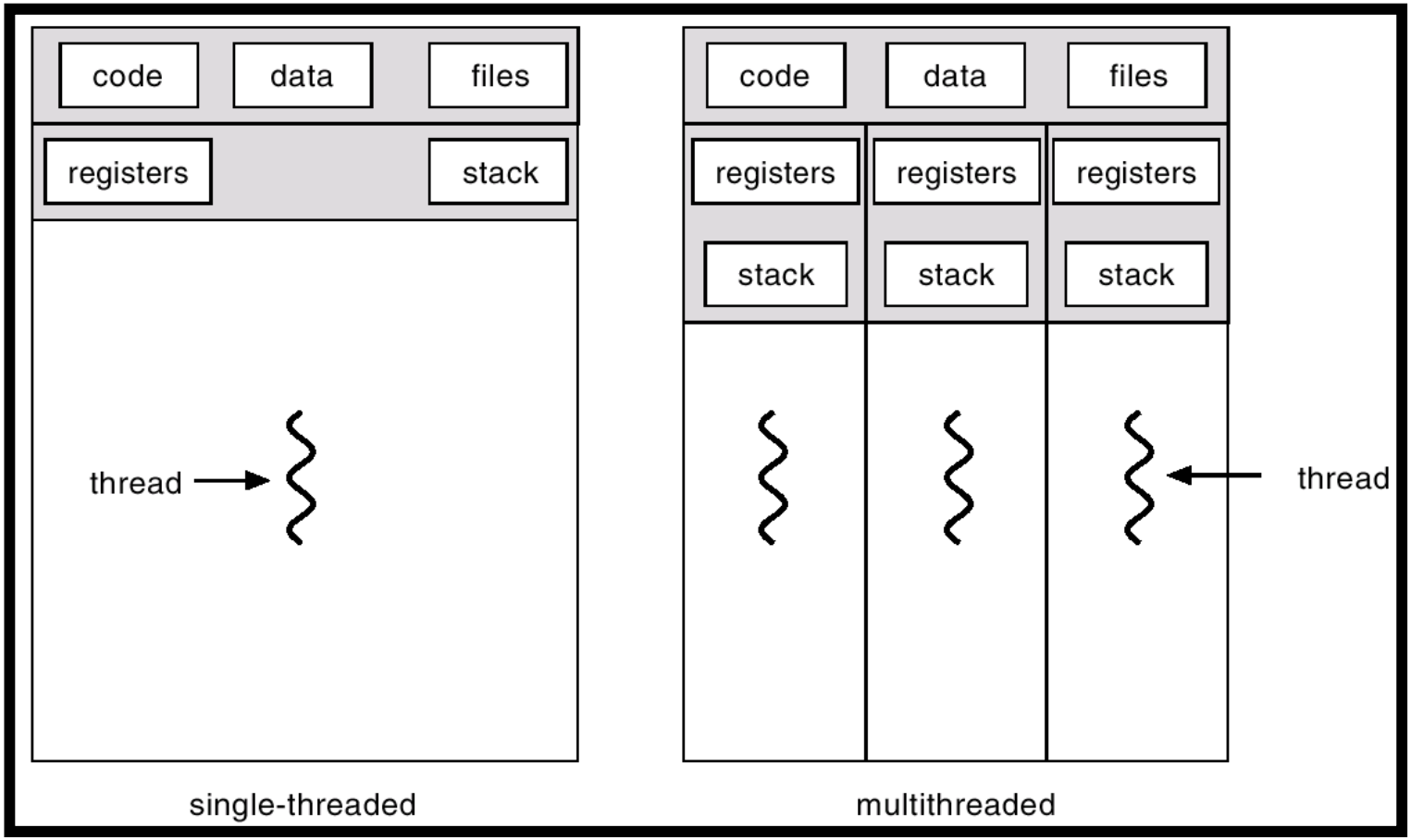


- Uma *thread* pode se bloquear à espera de um recurso. Neste momento, uma outra *thread* (do mesmo processo ou não) pode ser executada.
- A troca de contexto é mais leve. Se a *thread* t1 do processo p1 estiver rodando e entrar em execução, logo após, a *thread* t2 também do processo p1, a troca de contexto só diz respeito à execução. O ambiente continua o mesmo.

Thread

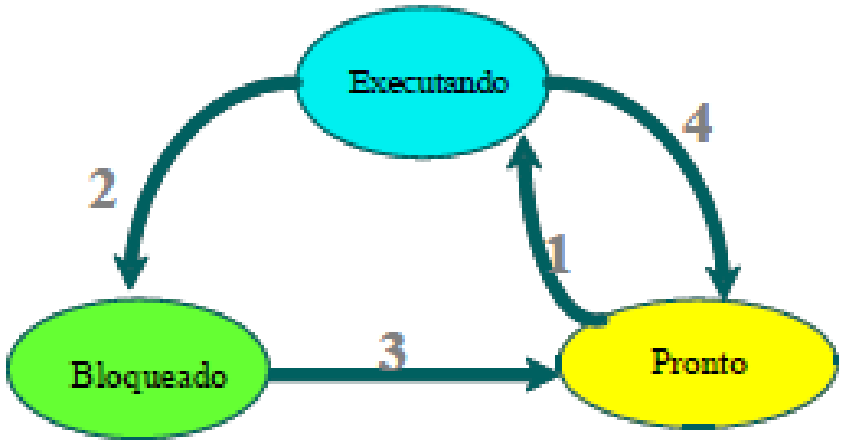
- O Bloco Descritor das *threads* inclui:
  - Registradores privados;
    - PC, SP, registradores de uso comum.
  - Uma pilha
    - Histórico da execução, com a várias chamadas a subrotinas que não completaram e suas variáveis locais.
    - Endereço de retorno após completar a chamada.
  - *Thread* ID
  - Ponteiros para outras *threads*
  - Ponteiro para o PCB em que se encontra.
    - Inclusive o espaço de endereçamento do processo pai!
  - Informação de escalonamento
    - Prioridade, estado, tipo de escalonamento...;

Processos Simples e *Multithreads*



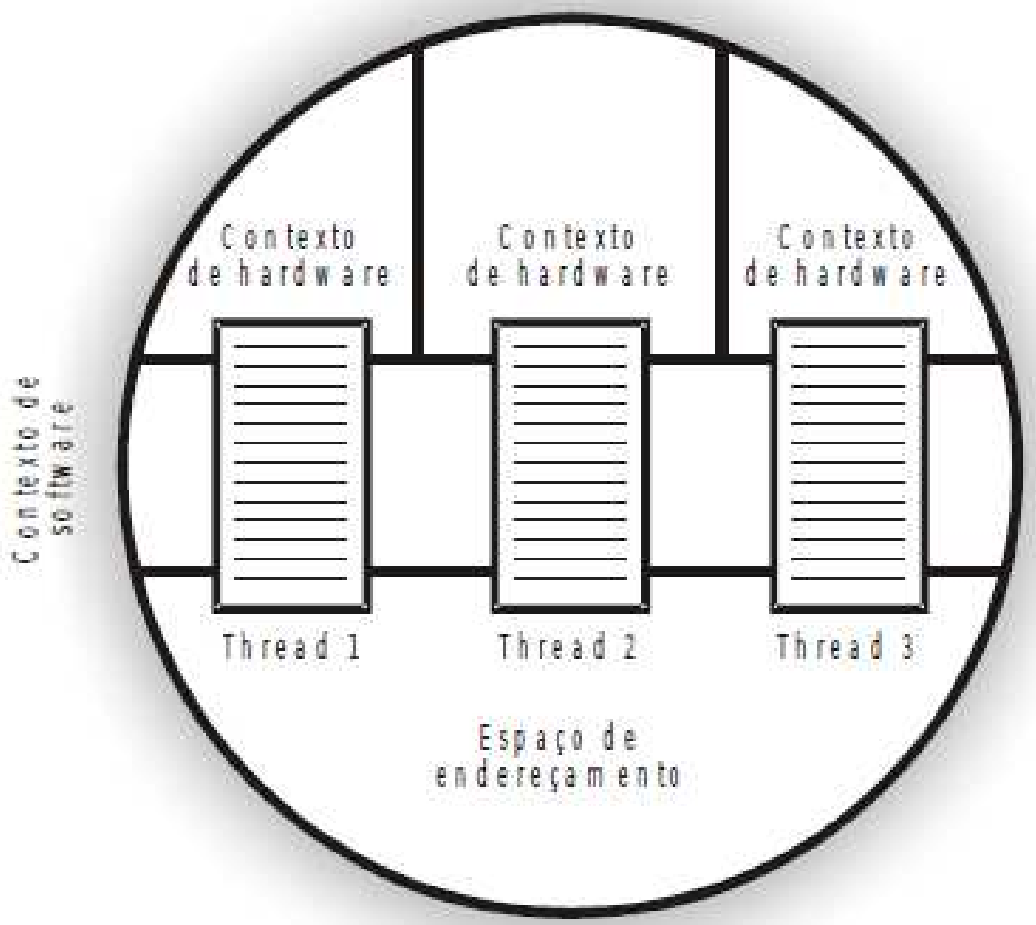
Estados da *Thread*

- A entidade que realmente executa é a *thread*. O processo é só o ambiente.
- Estados de execução das threads:



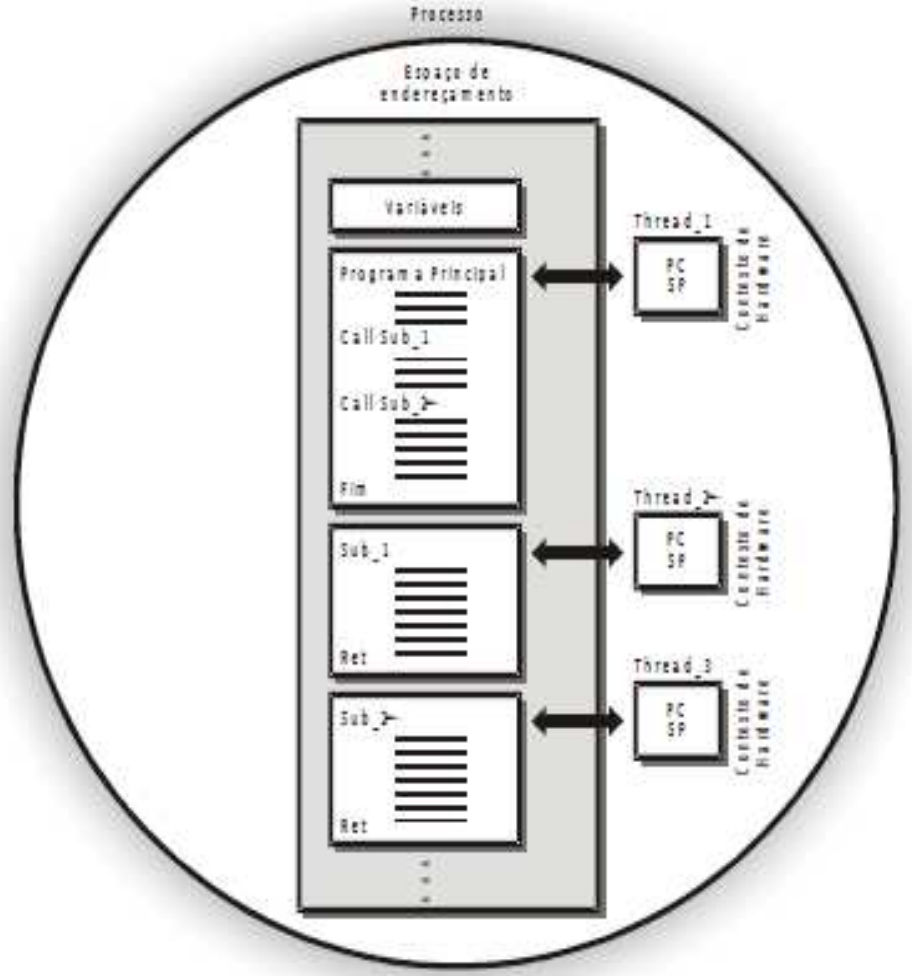
- As threads compartilham as variáveis globais do programa, os descritores abertos etc. Assim, há necessidade de mecanismos de sincronização.

Ambiente Multithread



Ambiente Multithread

- **Cada processo possui pelo menos uma *thread***



Vantagens das *Threads*

- Tempo de resposta
  - Uma aplicação interativa pode continuar sendo executada se parte dela está bloqueada, ou executando uma operação lenta.
- Compartilhamento de recursos
  - Por padrão as *threads* compartilham
    - Memória
    - Qualquer recurso alocado pelo processo ao qual são subordinadas
  - Não é necessária a alocação de mais recursos no sistema
- Economia
  - É mais econômico criar *thread* e trocar o contexto entre as mesmas.
- Utilização de arquiteturas multiprocessadas
  - Cada *thread* pode ser executada de forma paralela, em processadores distintos.

Vantagens das *Threads*

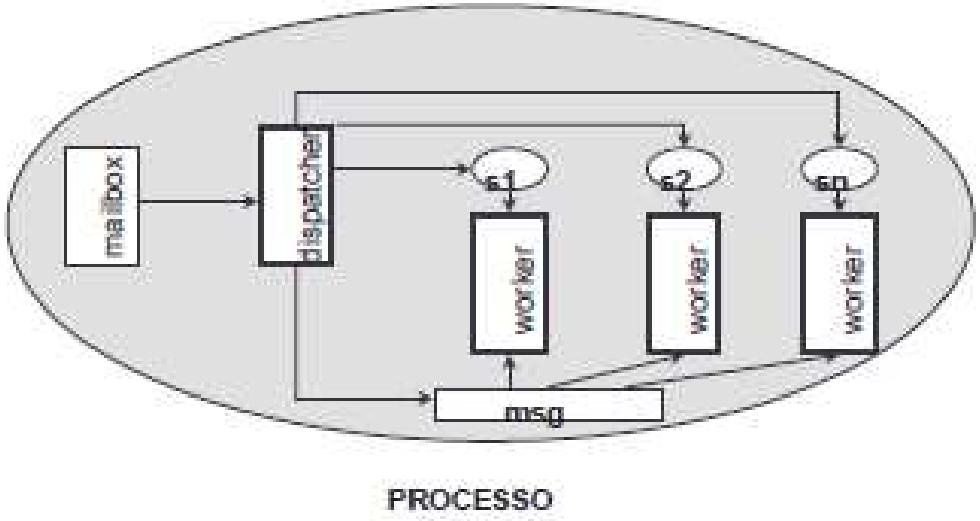
- São processos **leves (!= processos pesados).**
  - Troca de contexto mais rápida
    - Fator 5 no caso do S.O. SOLARIS, por exemplo.
  - Tempo de criação menor
    - Fator 30 no caso do S.O. SOLARIS, por exemplo.
- Logo, diminui o tempo de resposta do sistema.
- Maior facilidade para mesclar *threads* I/O-bound com *threads* CPU-bound.
- O compartilhamento de recursos facilita a comunicação entre as *threads:*
  - Através da área de memória compartilhada (global, *heap*).

Modelos de Execução de *Threads*

- O modelo de execução determina como as *threads* irão se organizar para resolver um problema. É claro que esta organização é altamente dependente do problema a ser resolvido.
- Em servidores, por exemplo, o modelo pode ser:
  - Threads dinâmicas: uma thread é criada para tratar cada requisição;
  - Threads estáticas: o número de *threads* é fixo:
    - Dispatcher/worker;
    - team ;
    - Pipeline.

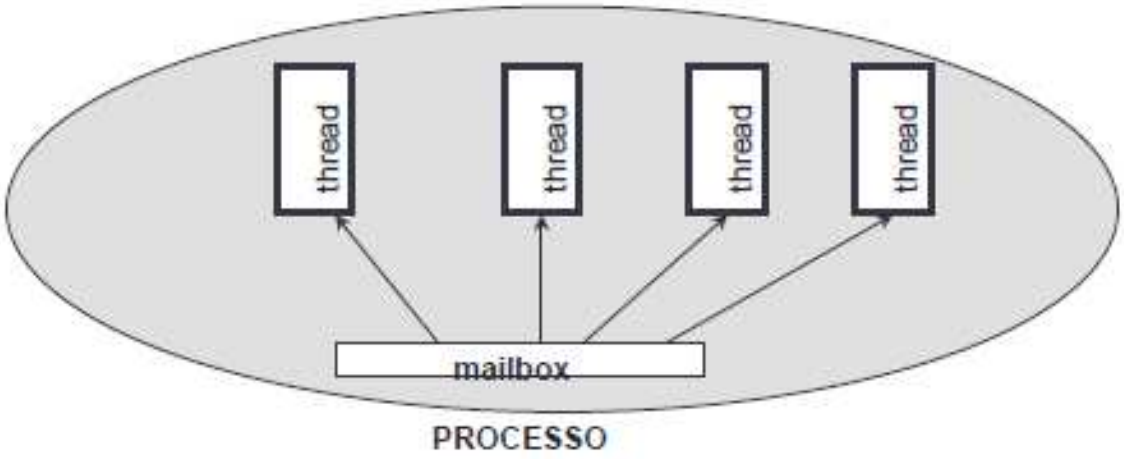
Modelo Dispatcher/worker

- Nesta organização, a thread dispatcher recebe todas as requisições. Ela escolhe, então, uma thread trabalhadora e a acorda para tratar a requisição.
- A thread trabalhadora executa a solicitação e, quando acaba, sinaliza o dispatcher.



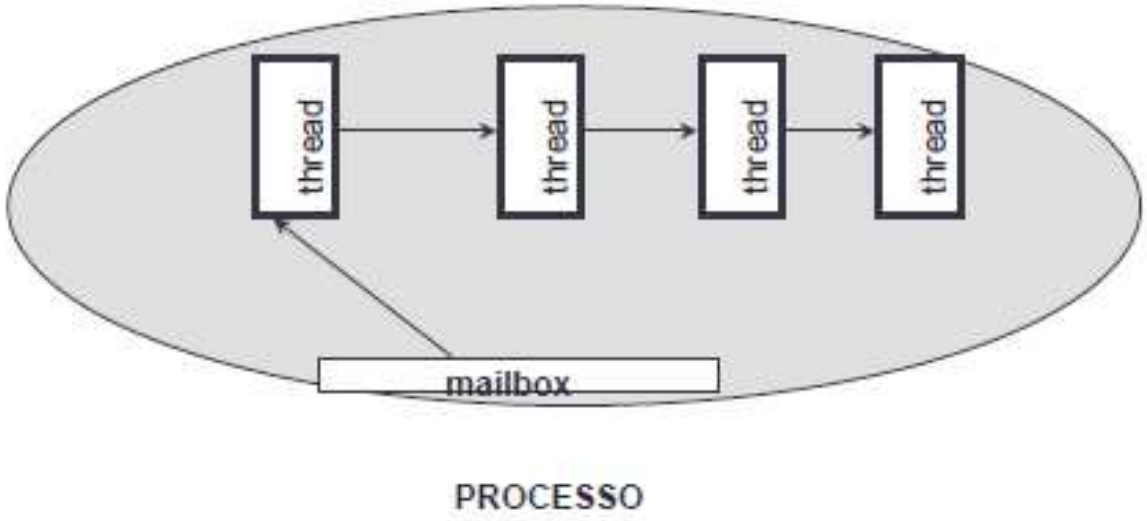
Modelo Team

- Nesta organização, cada thread é inteiramente autônoma. Todas as threads acessam a caixa postal, obtém requisições e as executam.



Modelo Pipeline

- Nesta organização, cada thread tem uma tarefa específica e os dados de entrada de uma thread são produzidos pela thread anterior.



Criação e Sincronização de *Threads*

- Estáticas: o número de linhas de controle que comporão um processo é determinado em tempo de compilação. A execução começa com *n* threads.
- Dinâmicas: as threads são criadas e destruídas dinamicamente, ao longo da execução. A execução começa com uma thread apontando para o início da rotina main.
  - Técnica mais utilizada
- Como as threads compartilham memória, elas necessitam de mecanismos de sincronização. A maioria dos pacotes de threads oferece dois mecanismos: variáveis mutex e variáveis de condição.

Threads

- Todavia, embora muitos SOs suportem *threads*, as implementações variam muito.
- Dependendo da implementação de *threads*, elas podem ser gerenciadas pelo SO ou pela aplicação de usuário que os cria.
- *Threads* Win32, C-threads e threads POSIX são exemplos de bibliotecas de suporte a *threads* com APIs diferentes.

Gerenciamento de *Threads*

- As *threads* podem ser criadas em dois níveis:
  - ***Threads* de Usuário - *User-Level Thread* (ULT)**
    - São admitidas em nível do usuário e gerenciadas sem o suporte do *kernel*.
    - Exemplo: bibliotecas Pthreads (POSIX).
  - ***Threads de Kernel (ou Sistema) - Kernel-Level Thread (KLT)***
    - São admitidas e gerenciadas diretamente pelo sistema operacional.
    - Quase todos os sistemas operacionais admitem *threads* de *kernel*: Windows, Linux, Mac OS X, etc.

Gerenciamento de *Threads*

- *Threads* de Usuário:
  - A responsabilidade pela gerência e sincronização das *threads* é da aplicação;
  - O núcleo não interfere nas *threads;*
  - Possuem uma grande limitação, pois o sistema operacional gerencia cada processo como se existisse apenas uma única *thread*.
  - No momento em que a *thread* chama uma rotina do sistema que o coloca em estado de espera (rotina bloqueante), todo o processo é colocado no estado de espera, mesmo havendo outras *threads* prontas para execução.

Gerenciamento de *Threads*

- *Threads* de Usuário:
  - As *threads* de usuário são escalonadas pelo programador, tendo a grande vantagem de que cada processo pode usar um algoritmo de escalonamento que melhor se adapte a situação.
  - Neste modo, o programador é responsável por criar, executar, escalonar e destruir cada *thread*;
  - *Threads* em modo usuário são implementas por chamadas a uma biblioteca de rotinas,que são ligadas e carregadas em tempo de execução (*run-time*) no mesmo espaço de endereçamento do processo e executadas em modo usuário.

Gerenciamento de *Threads*

- *Threads* de Usuário:
  - *Threads* em modo usuário são rápidas e eficientes, por **dispensar acesso ao *kernel*** do sistema para a criação, a eliminação, a sincronização e a troca de contexto das *threads*.
  - A biblioteca oferece todo o suporte necessário em modo usuário, sem a necessidade de chamadas ao sistema (*system calls*);
  - Como é o caso da **thread.h** (biblioteca padrão da linguagem C);
  - Outras linguagens que dão suporte: Ada, Modula-3, Python, SmallTalk, Objective-C, Java, etc.

Gerenciamento de *Threads*

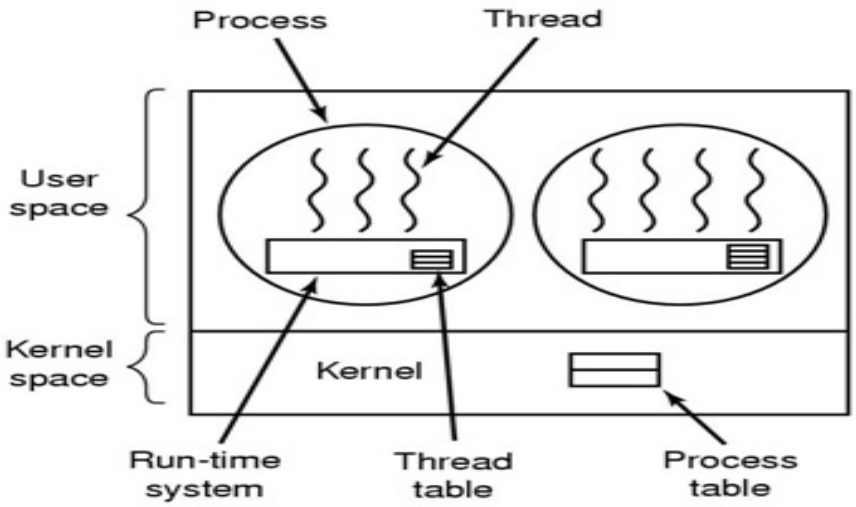
- Vantagens das *threads* de usuário:
  - Permitem a implementação de *threads* mesmo em sistemas operacionais que não suportam *threads*;
  - Eficiência no aproveitamento dos recursos;
  - Modularização das atividades;
  - Muito rápidas de gerenciar e eficientes, pois não necessitam do núcleo (chamada de sistema...).

Gerenciamento de *Threads*

- Desvantagens das *threads* de usuário:
  - O sistema operacional gerencia o processo como se houvesse uma única *thread;*
  - Tratamento individual de sinais, ou seja, os sinais enviados para um processo devem ser reconhecidos e encaminhados a cada *thread* para tratamento;
  - Redução do grau de paralelismo. Não é possível que múltiplas *threads* possam ser executadas em diferentes CPUs simultaneamente.

Gerenciamento de *Threads*

- *Threads* de Usuário:



Gerenciamento de *Threads*

- *Threads* de Núcleo:
  - Sistema operacional provê rotinas para gerenciamento das *threads* (necessita de mudanças nos modos de acesso);
  - Elas são implementadas diretamente pelo núcleo do sistema operacional, através de chamadas a rotinas do sistema que oferecem todas as funções de gerenciamento e sincronização;
  - Comumente são mais lentas que as *threads* ULT pois a cada chamada elas necessitam consultar o sistema, exigindo assim a mudança total de contexto do processador, memória e outros níveis necessários para alternar um processo.

Gerenciamento de *Threads*

- *Threads* de Núcleo:
  - O sistema operacional sabe da existência de cada *thread* e pode escaloná-las individualmente;
  - O Sistema operacional não entrega a CPU ao processo e sim a uma *thread* deste processo.
  - Exemplos: Windows, Solaris, Linux, etc.

Gerenciamento de *Threads*

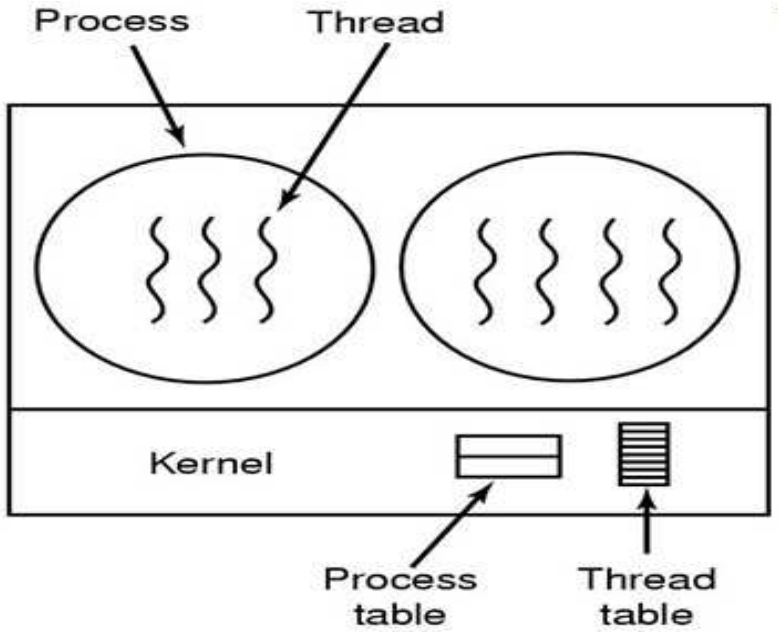
- Vantagens das *threads* de núcleo:
  - O sistema operacional sabe da existência de cada *thread* e pode escaloná-las individualmente;
  - Caso haja múltiplos processadores, *threads* de um mesmo processo podem ser executadas simultaneamente;
  - Possibilidade de bloquear uma parte do processo, e a outra continuar sua execução.

Gerenciamento de *Threads*

- Desvantagens das *threads* de núcleo:
  - Baixo desempenho. Enquanto nos pacotes em modo usuário todo o tratamento é feito sem a ajuda do sistema operacional, no modo *kernel* são utilizadas chamadas a rotinas do sistema e, conseqüentemente, há várias mudanças no modo de acesso.

Gerenciamento de *Threads*

- *Threads* de Núcleo:

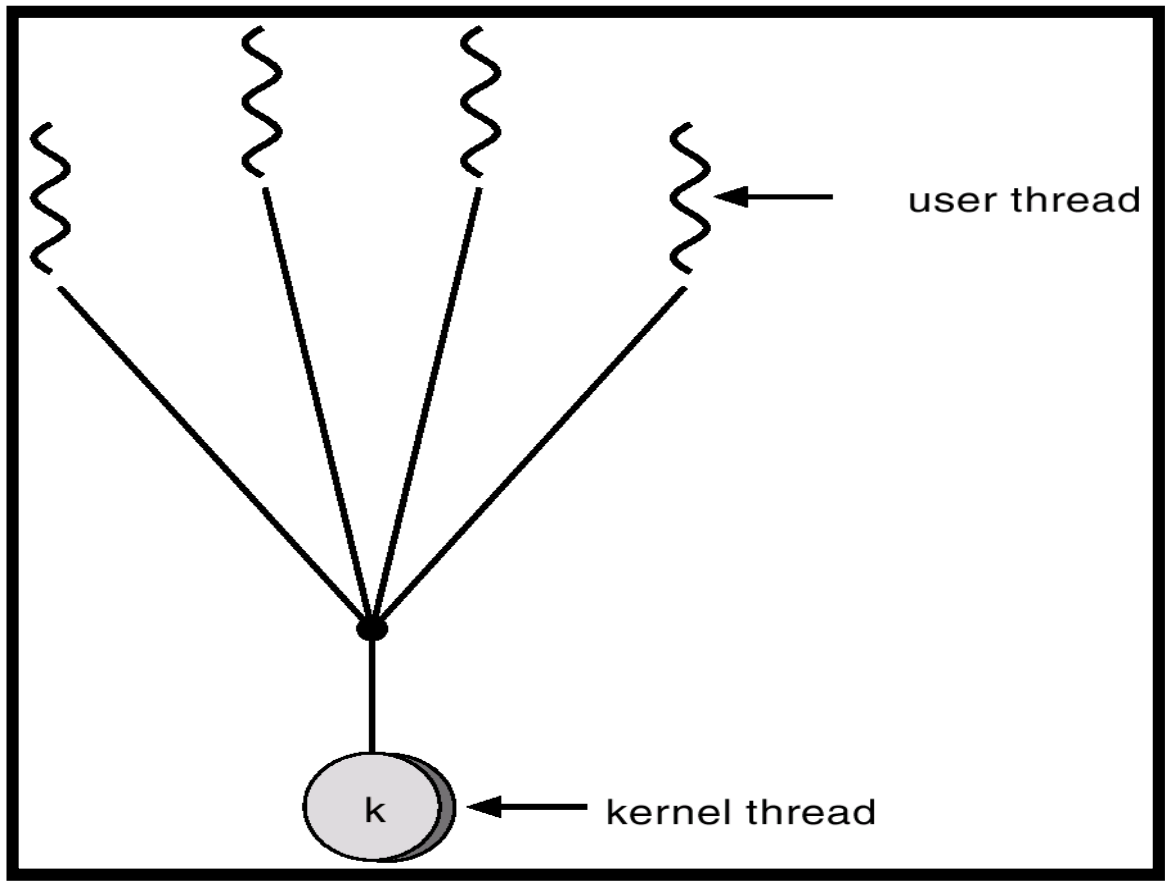


Conciliando os dois Níveis - Modelos de *Multithreading*

- **Modelo N para 1**
  - N *threads* de usuário para 1 *thread* de sistema.
  - O gerenciamento da *thread* é feito pela biblioteca de *threads* no nível do usuário.
  - Se uma *thread* fizer uma chamada ao sistema que bloqueia o processamento, todo o processo será bloqueado.
  - Utilizado em sistemas que não suportam *threads* em nível de sistema.
  - Ex.: Biblioteca Green threads, disponível para o Solaris; NACHOS; MACH C-threads; POSIX Pthreads.

Modelos de *Multithreading*

- Modelo N para 1

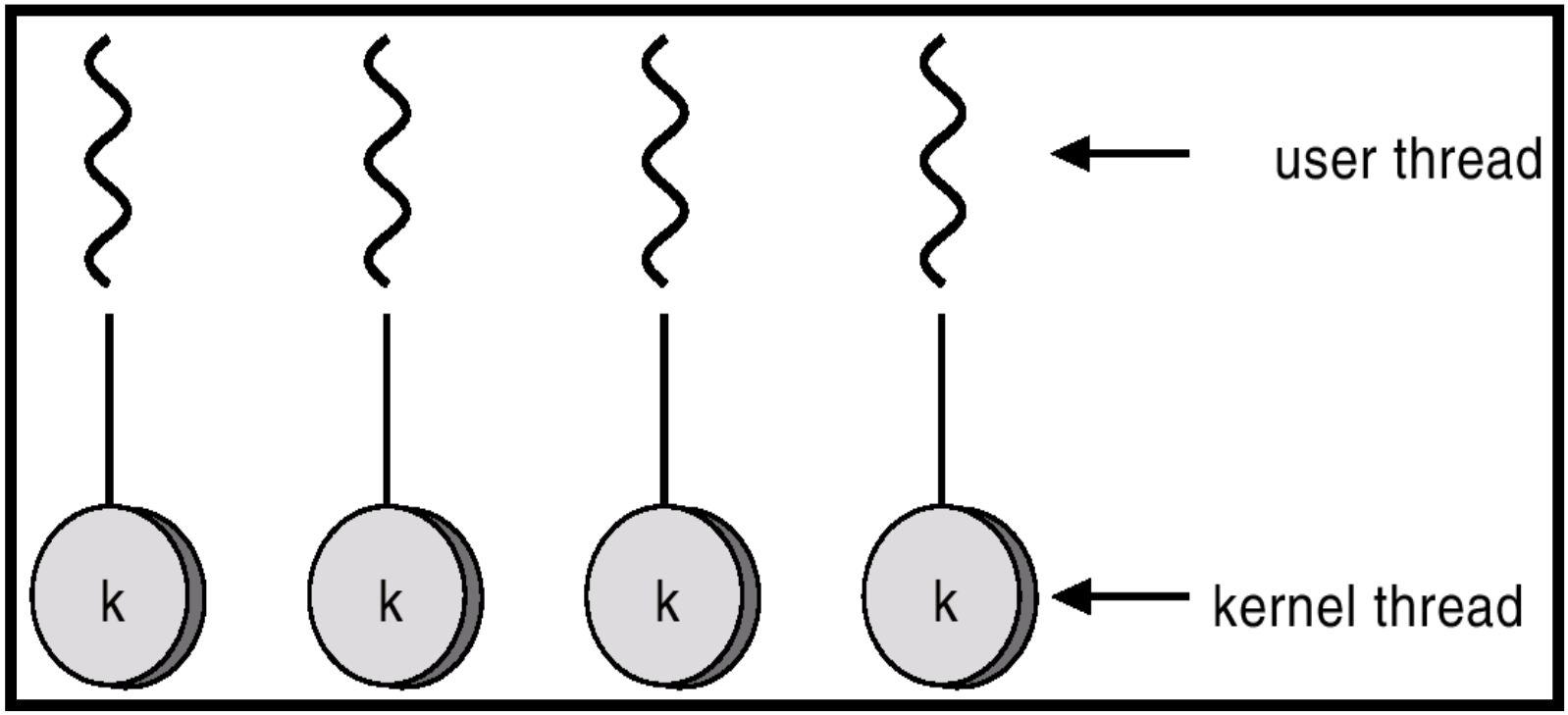


Modelos de *Multithreading*

- Modelo 1 para 1
  - Associa cada *thread* de usuário para 1 *thread* de sistema.
  - Provê maior concorrência do que o modelo anterior, permitindo que outra *thread* seja executada quando uma *thread* faz uma chamada de sistema bloqueante.
  - Permite que várias *threads* sejam executadas em paralelo em multiprocessadores.
  - Desvantagem: o custo adicional da criação de *threads* do *kernel* pode prejudicar o desempenho de uma aplicação.
  - Exemplos: Windows 95, 97 e 98, OS/2 e Linux versões atuais.

Modelos de *Multithreading*

- Modelo 1 para 1

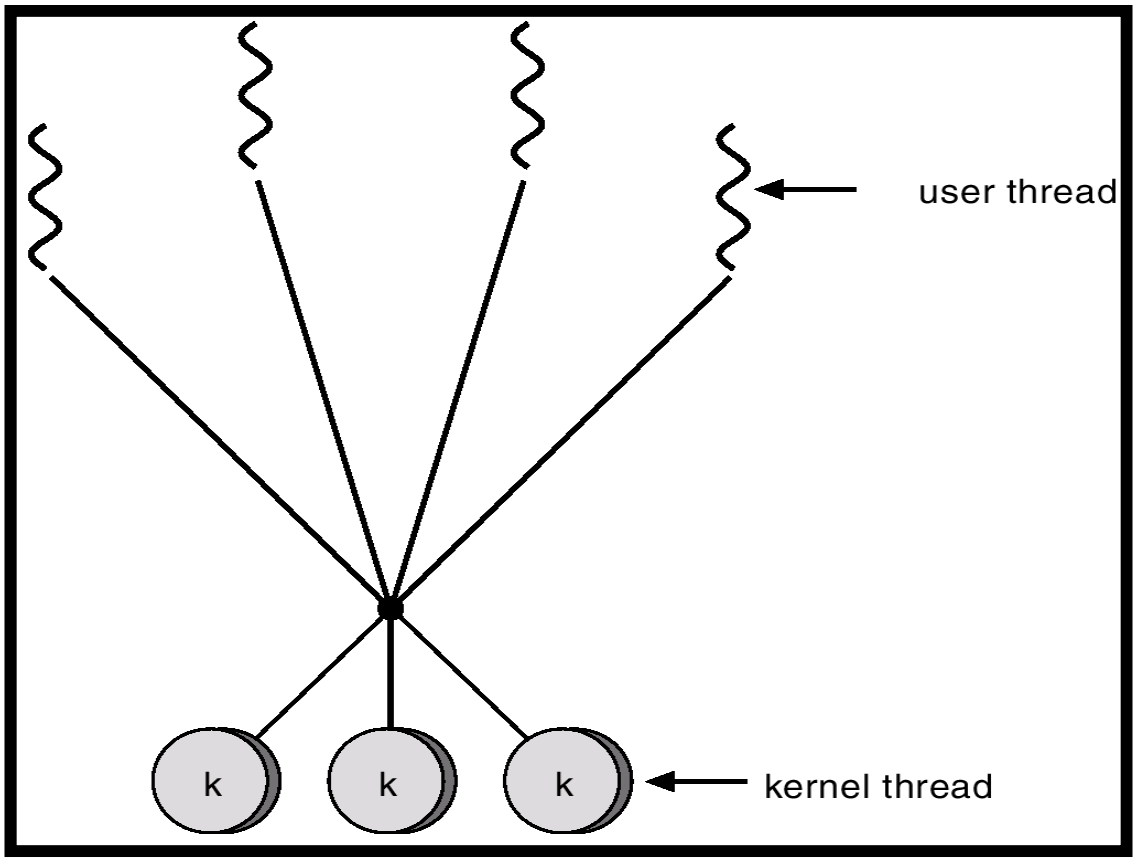


Modelos de *Multithreading*

- Modelo N para M
  - N *threads* de usuário para M *threads* de sistema.
  - O número de *threads* de *kernel* pode ser específico a determinada aplicação ou a determinada máquina.
  - Quando uma *thread* realiza uma chamada de sistema bloqueante, o *kernel* pode escalonar outra *thread*.
  - Exemplos:
    - Solaris 2
    - Windows NT/2000 com o Pacote ThreadFiber

Modelos de *Multithreading*

- Modelo N para M



Bibliotecas *Threads*

- Uma biblioteca *thread* fornece ao programador uma API para a criação e o gerenciamento de *threads*.
- Existem duas maneiras de implementar bibliotecas:
  - Fornecer uma biblioteca no espaço do usuário, sem suporte do *kernel*.
  - Fornecer uma biblioteca no nível do *kernel*, com suporte direto do S.O.
- As três bibliotecas de *threads* mais comuns são:
  - Win32 (nível do *kernel*);
  - Java (no nível do usuário, mas usa sempre a biblioteca do S.O. hospedeiro);
  - POSIX Pthreads (fornecida no nível do usuário ou do *kernel*).

Escalonamento de *Threads*

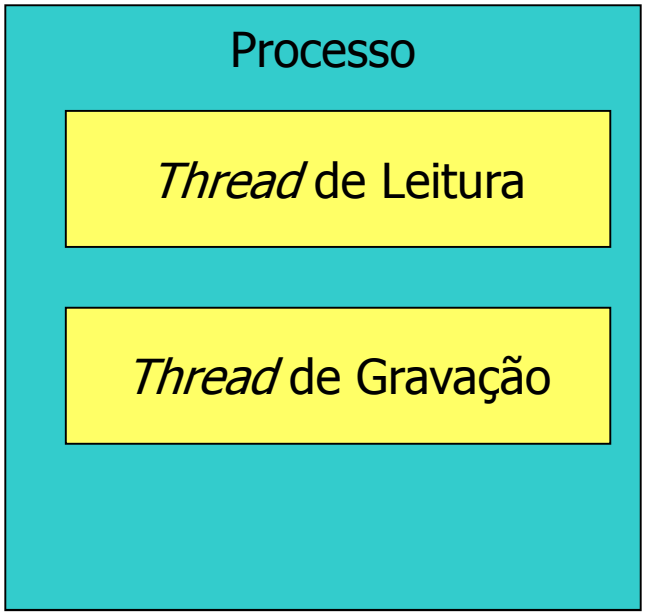
- Nos sistemas operacionais que admitem a criação de *threads* no nível de *kernel*, são as *threads* em nível de *kernel* que são escalonadas, e não os processos.
- As *threads* em nível de usuário não são conhecidas pelo S.O.
- Em sistemas que implementam os modelos N para 1, ou N para M, a biblioteca *thread* escalona as *threads* em nível de usuário para executarem:
  - Esse esquema é conhecido como Escopo de Disputa do Processo (*Process Contention Scope* - PCS).
  - Pois, nesse caso a disputa pela CPU ocorre entre as *threads* pertencentes ao mesmo processo.

Escalonamento de *Threads*

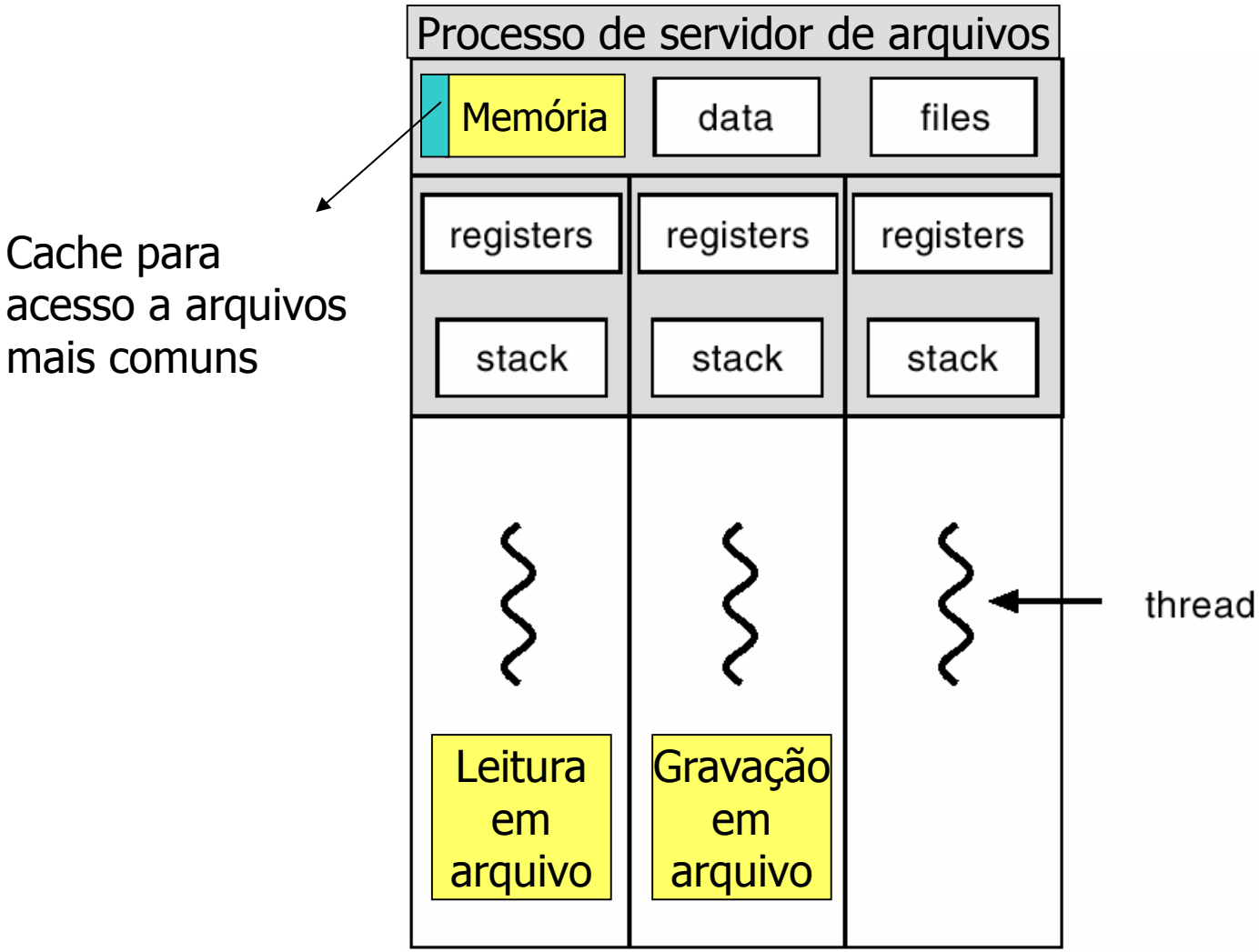
- As *threads* no nível do *kernel* são escalonadas para o uso da CPU.
  - Esse esquema é conhecido como Escopo de Disputa do Sistema (*System Contention Scope* - SCS).
  - A disputa pela CPU com escalonamento SCS ocorre entre todas as *threads* no sistema.
- Os sistemas operacionais que utilizam o modelo 1 para 1 (como Windows XP, Solaris 9 e Linux) escalonam as *threads* usando apenas o SCS.
- As prioridades das *threads* do usuário são definidas pelo programador.

Exemplo Prático de *Thread*

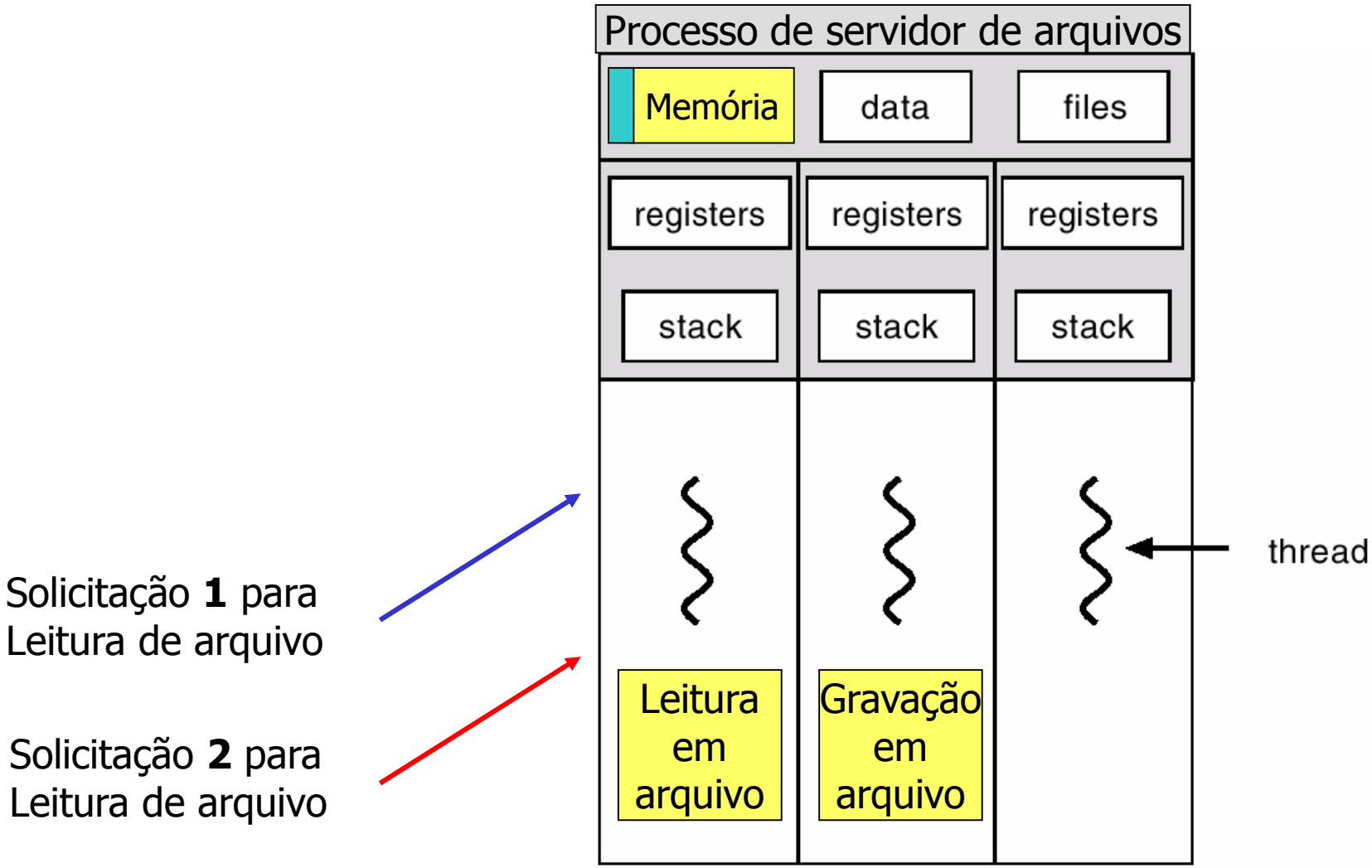
- Servidor de Arquivos
  - Há um processo no servidor de arquivos que
    - Recebe requisições para:
      - Ler
      - Gravar
    - Envia resposta:
      - Os dados lidos
      - Atualização dos dados



Exemplo Prático de *Thread*

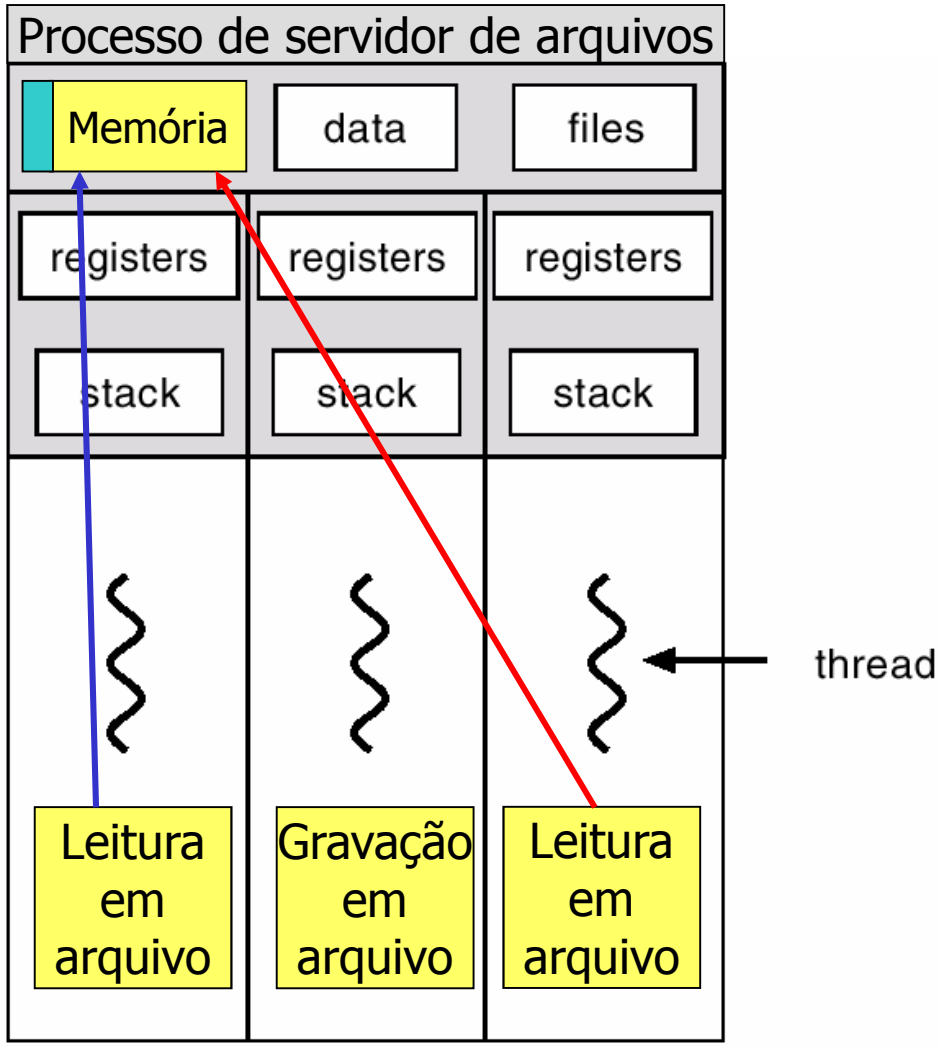


Exemplo Prático de *Thread*



Exemplo Prático de *Thread*

- É criada uma *thread* para cada solicitação
- As *threads* verificam se os arquivos se encontram em cache para acesso mais rápido

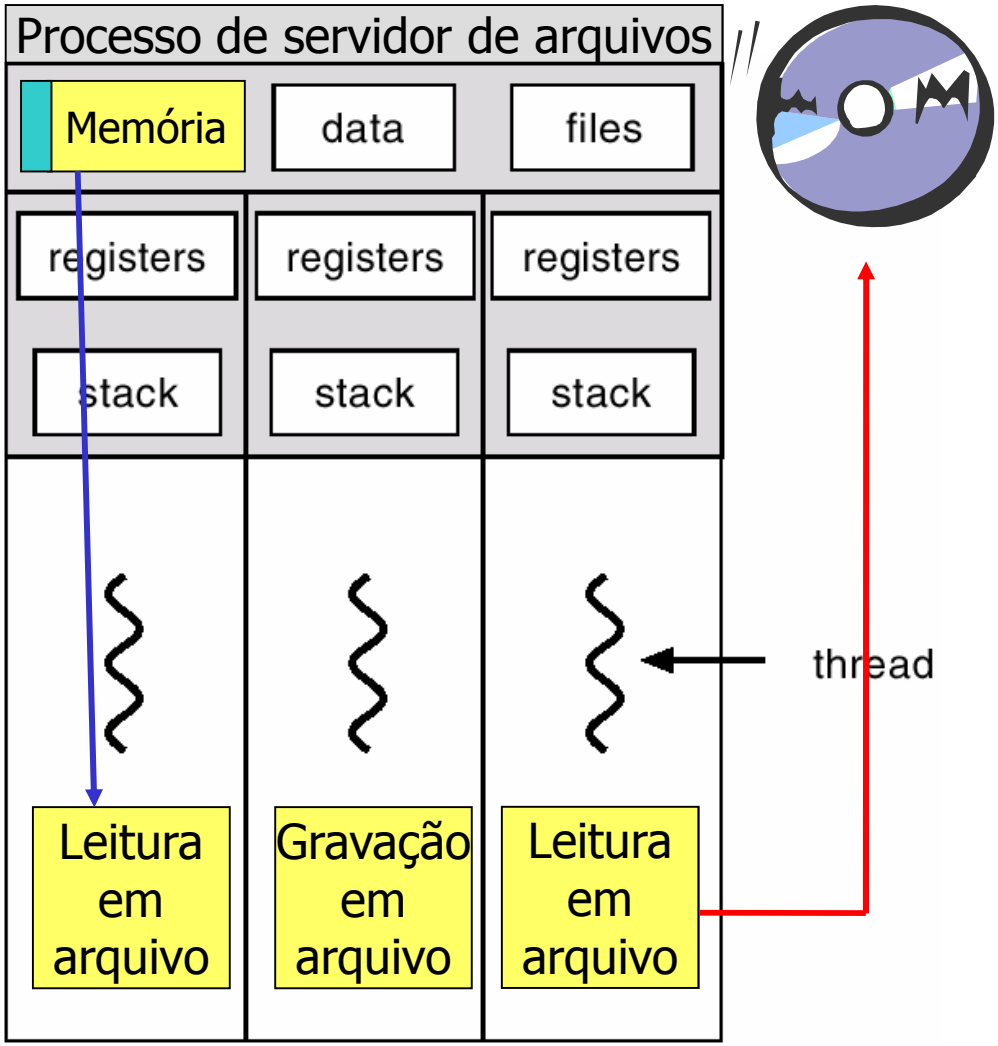


Solicitação **1** para Leitura de arquivo

Solicitação **2** para Leitura de arquivo

Exemplo Prático de *Thread*

- Arquivo da solicitação 1 está em cache
- Arquivo da solicitação 2 não está em cache

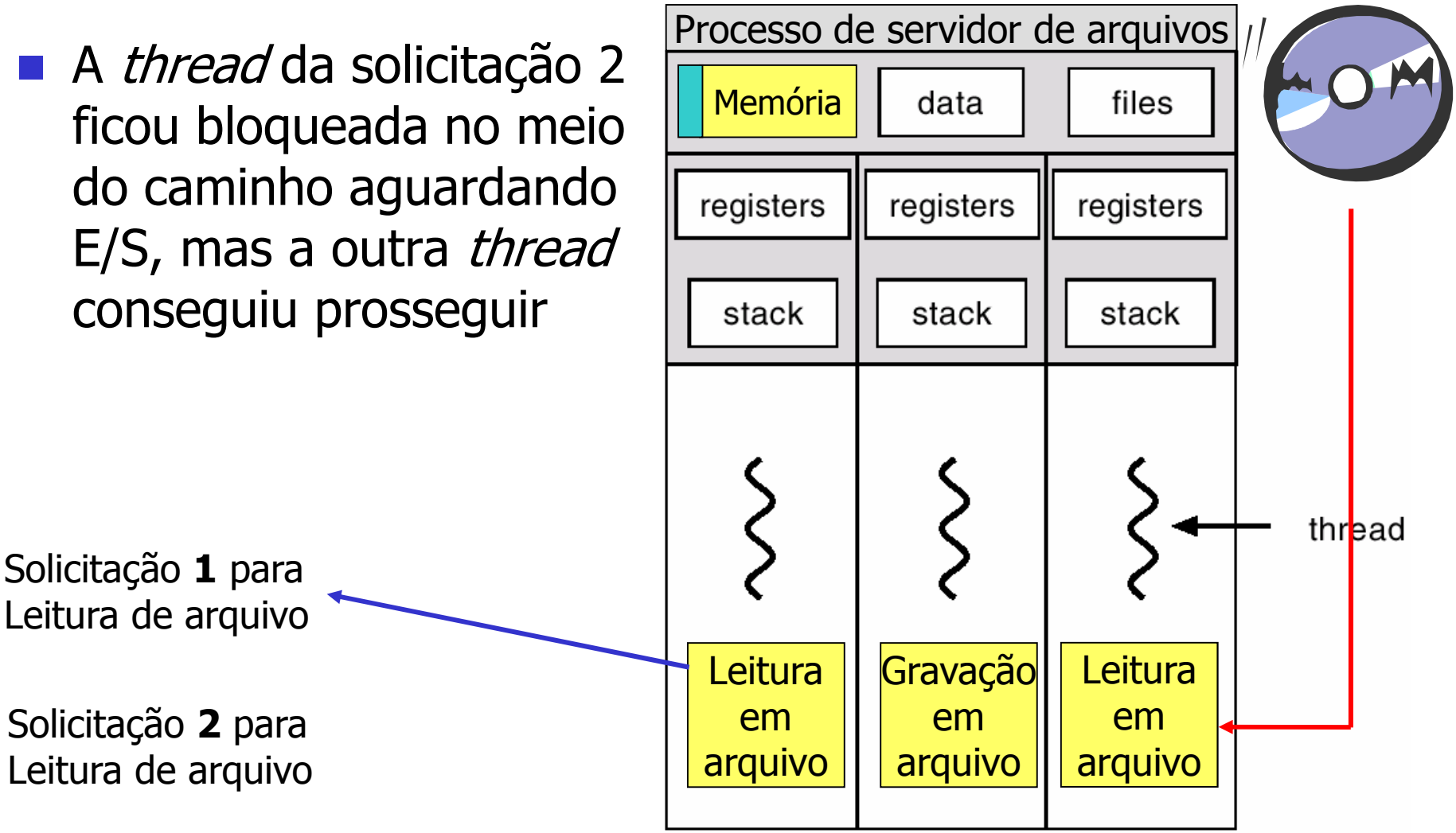


Solicitação **1** para Leitura de arquivo

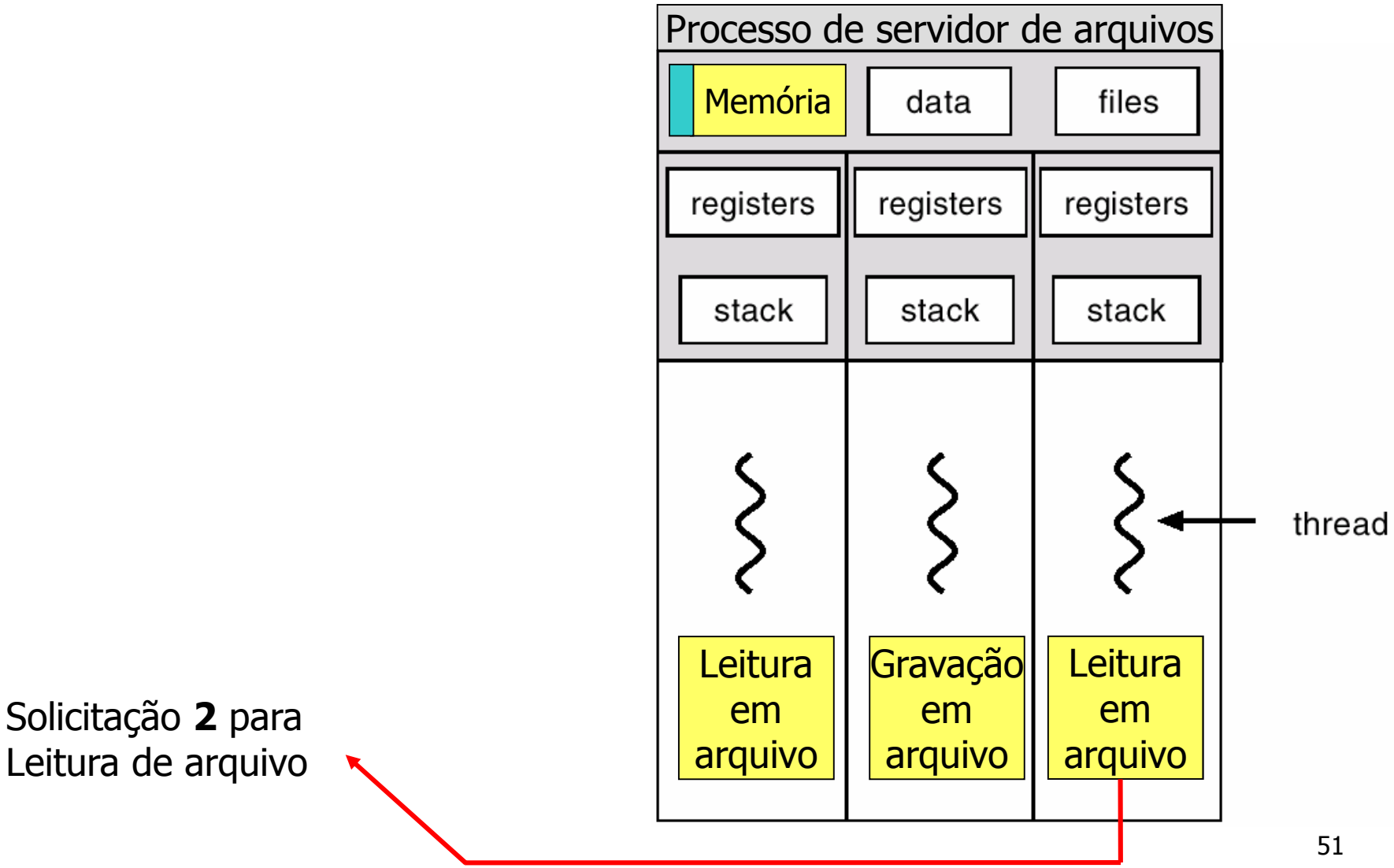
Solicitação **2** para Leitura de arquivo

Exemplo Prático de *Thread*

- A *thread* da solicitação 2 ficou bloqueada no meio do caminho aguardando E/S, mas a outra *thread* conseguiu prosseguir



Exemplo Prático de *Thread*



Referências Bibliográficas

[1] Tanenbaum, A. S., Woodhull, A. S. **Sistemas Operacionais: projeto e implementação**. 2ª ed. Porto Alegre: Bookman, 2000.

[2] Stallings, William. **Operating Systems: internals and Design Principles**. 4ª ed. New Jersey: Prentice Hall. 2001.

[3] Silberschatz, A., Galvin, P. B., Gagne, G. **Operating System Concepts**. 6ª ed. Editora John Wiley & Sons, Inc. 2002.

[4] SHAY, William A., Sistemas Operacionais. Makron Books. São Paulo. 1996.

[5] Deitel, Deitel, Choffnes – Sistemas Operacionais . Prentice Hall – São Paulo 3a Edição, 2005.