BC1518 - Sistemas Operacionais

Aula 5: Threads

Tópicos desta aula

- Visão geral
- Benefícios com o uso de *Threads*
- *Threads* de usuário e de k*ernel*
- Modelos de *Multithreading*
- *Threads* em Java
- *Threads* no Windows XP
- *Threads* no Linux

Visão geral

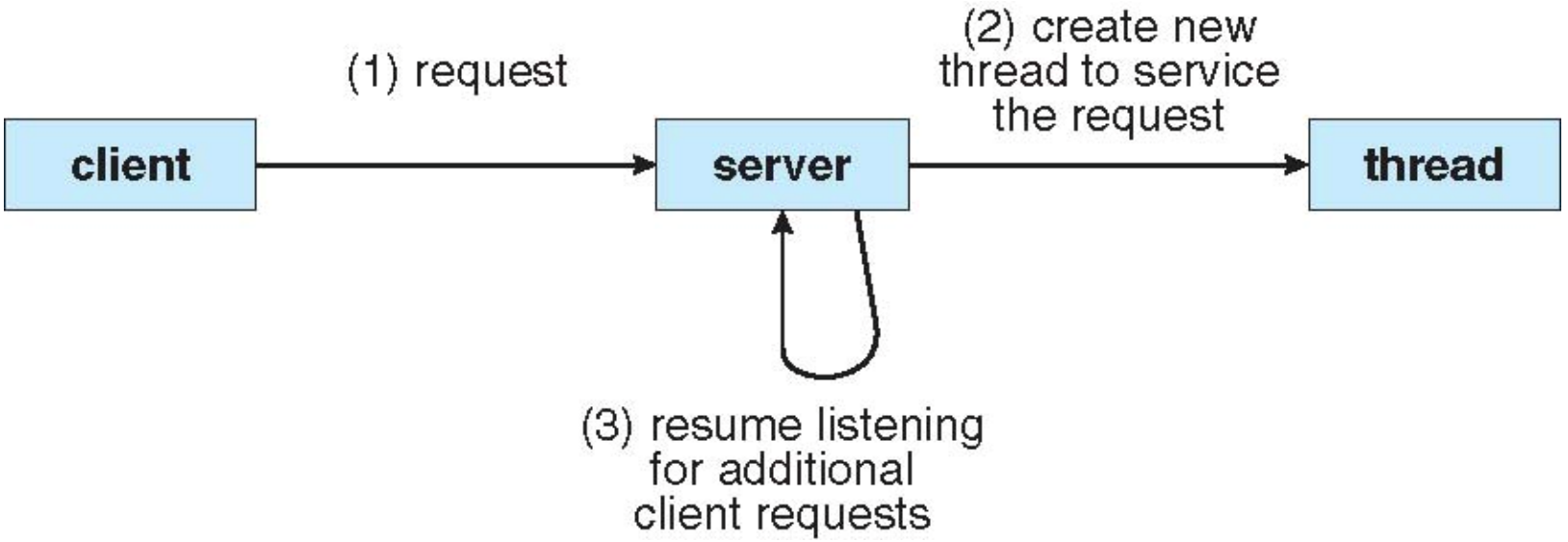
- Uma *thread* é um **fluxo de controle** dentro de um processo
  - *Thread* (fio, linha, filamento): também conhecida como linha de execução
- Um processo tradicional (ou pesado, modelo visto até aqui) possui apenas uma *thread* de controle
  - *Monothread* ou *single-threaded* (dotado de uma única *thread*)
  - Permite que o processo execute apenas uma tarefa de cada vez

- Se um processo possui várias *threads* de controle, este poderá executar várias tarefas ao mesmo tempo
  - *Multithread* ou *Multithreaded* (dotado de múltiplas *threads*)
  - Exemplos:
    - Um navegador Web pode estar fazendo *download* de um vídeo e ao mesmo tempo pode exibir as partes do vídeo já baixadas
    - Um processador de texto permite que o usuário digite os caracteres e ao mesmo tempo faz correções ortográficas

Visão geral

- Exemplos (cont.):
  - Um servidor Web precisa tratar milhares de requisições concorrentes de clientes (se for executado como um processo tradicional, com uma única *thread*, teria que tratar cada requisição sequencialmente)
  - O servidor pode ser implementado como um processo que aceita as requisições e para cada requisição, cria um novo processo separado para atender a requisição
    - Problemas: a criação de processos é demorada e exige muitos recursos; cada novo processo realiza as mesmas tarefas dos outros existentes
  - Uma solução mais eficaz: servidor com múltiplas *threads*, para cada requisição de cliente, em vez de um novo processo, cria uma nova *thread* para atender à requisição

Arquitetura de servidor com várias threads



Arquitetura de servidor com várias threads [Silberschatz2]

Visão Geral

➢Uma ***Thread*** (ou **Processo Leve**) é uma unidade básica da utilização da CPU, compreendendo:

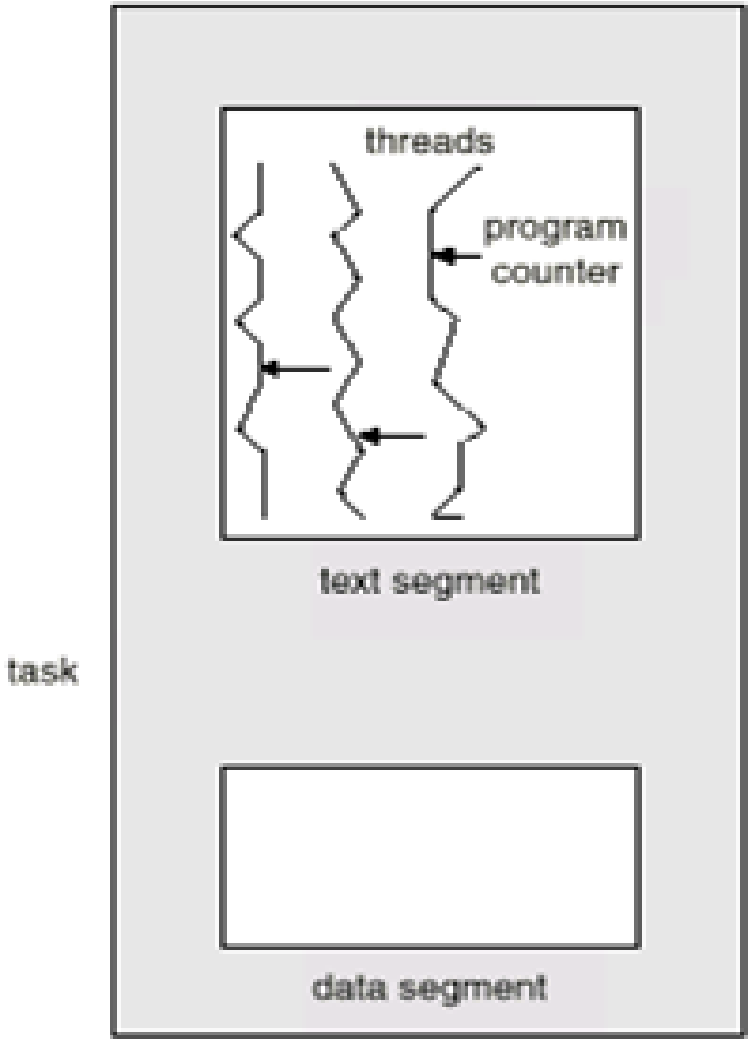
- um identificador da *thread* (ID)
- um contador de programa
- um conjunto de registradores
- uma pilha

➢Uma ***Thread*** compartilha com suas ***Threads*** afins:

- sua seção de código
- sua seção de dados
- seus recursos de sistema operacional como arquivos abertos

➢Um **processo tradicional**, ou **pesado**, é igual a uma tarefa com uma **única *Thread***

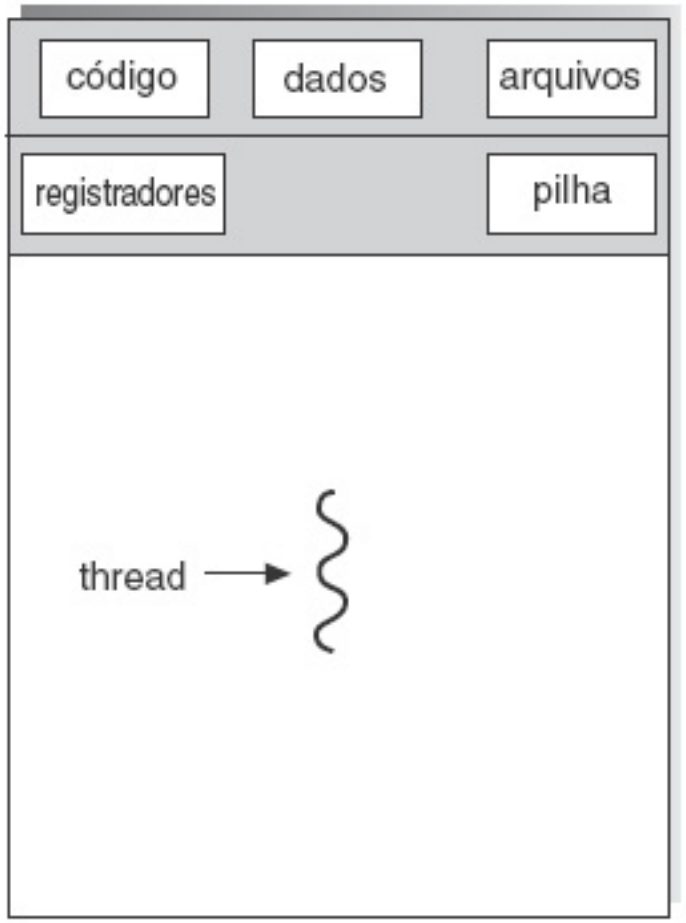
Múltiplas threads em uma tarefa (processo)



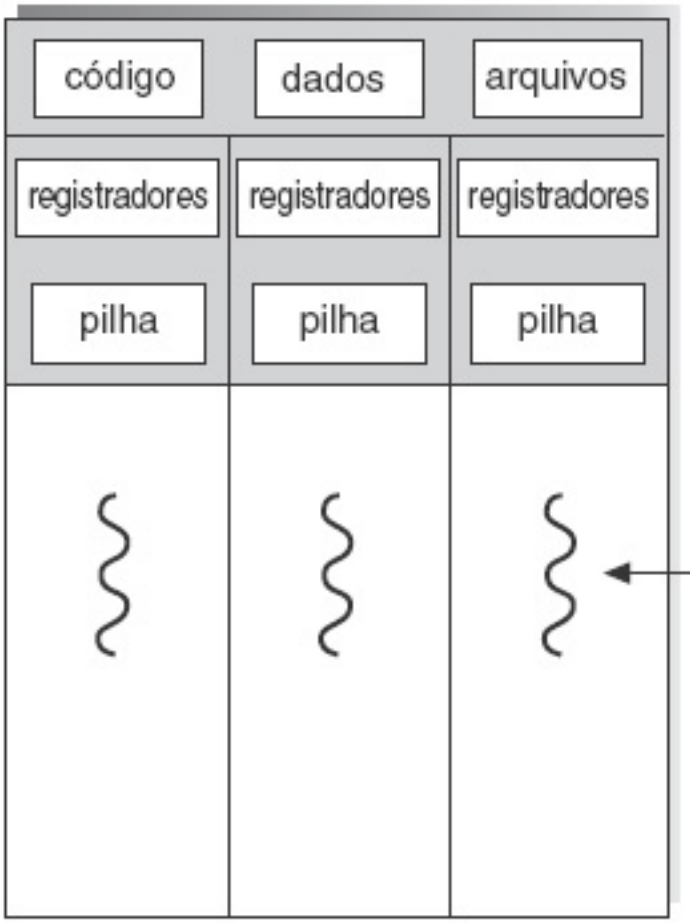
Uma ***Thread*** é um **fluxo de controle** em um processo

Um processo ***multithread*** contém **vários fluxos de controle distintos** no mesmo espaço de endereçamento

Processos com uma ou com múltiplas threads



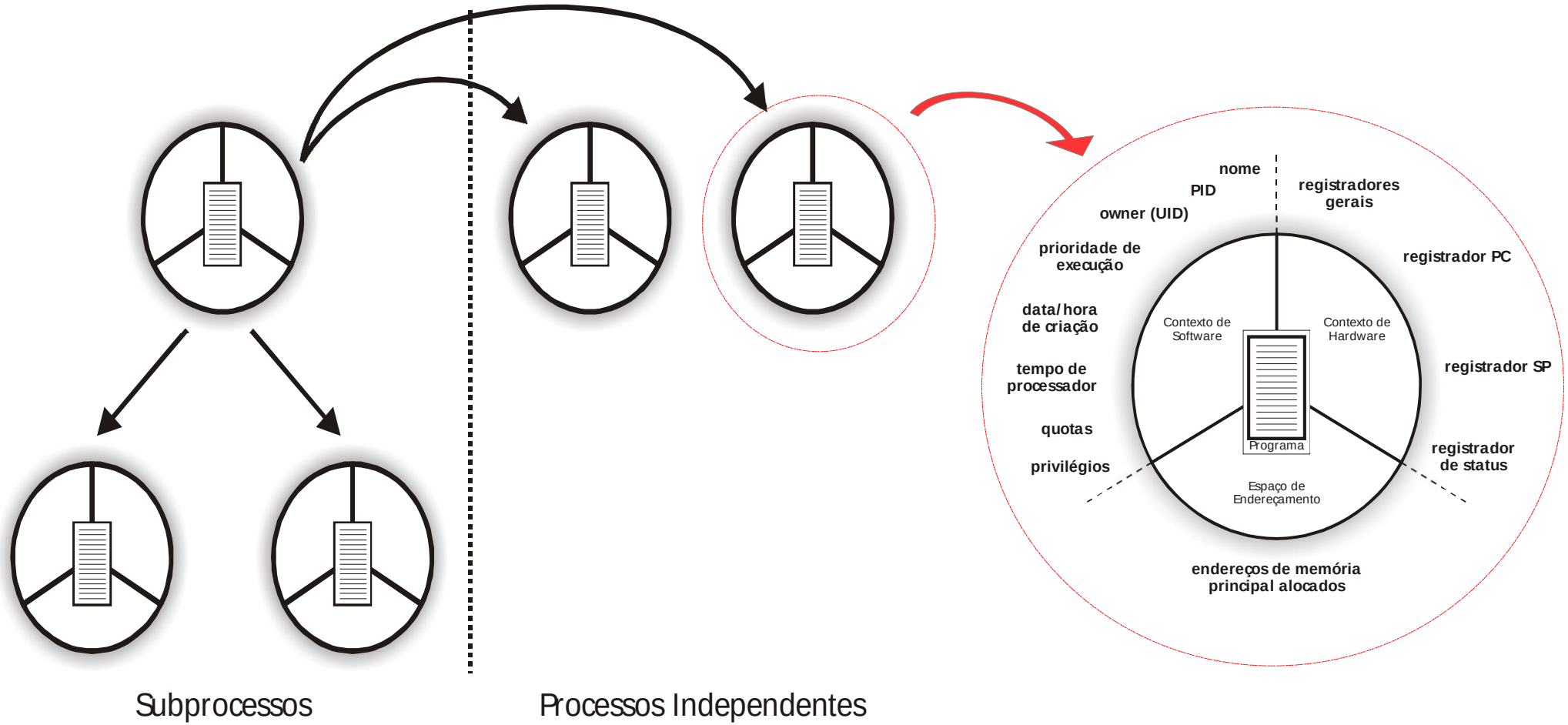
processo dotado de única thread



processo dotado de múltiplas threads

[Silberschatz]

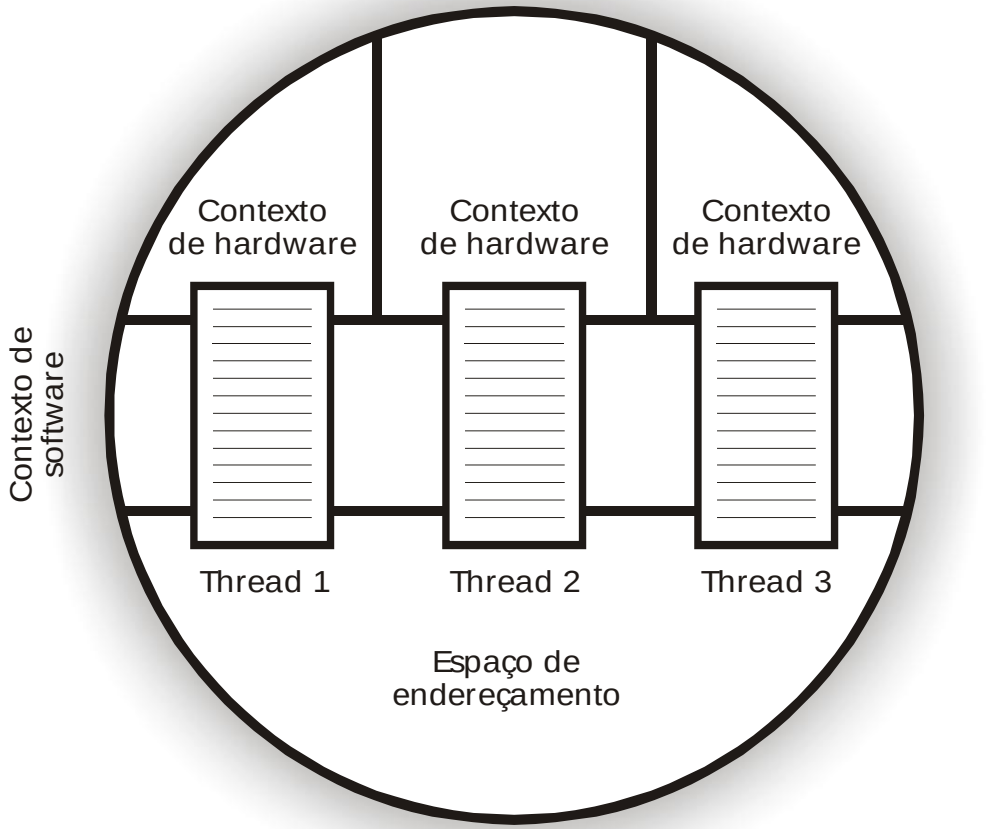
Ambiente Monothread



Concorrência com subprocessos e processos independentes [Machado]

Num ambiente *monothread*, a concorrência pode ser alcançada com: subprocessos ou processos independentes. Em ambos os casos, cada processo (ou subprocesso) possui seu próprio espaço de endereçamento, a comunicação requer algum mecanismo (IPC) e não há compartilhamento de recursos comuns aos processos (concorrentes) como memória e arquivos abertos

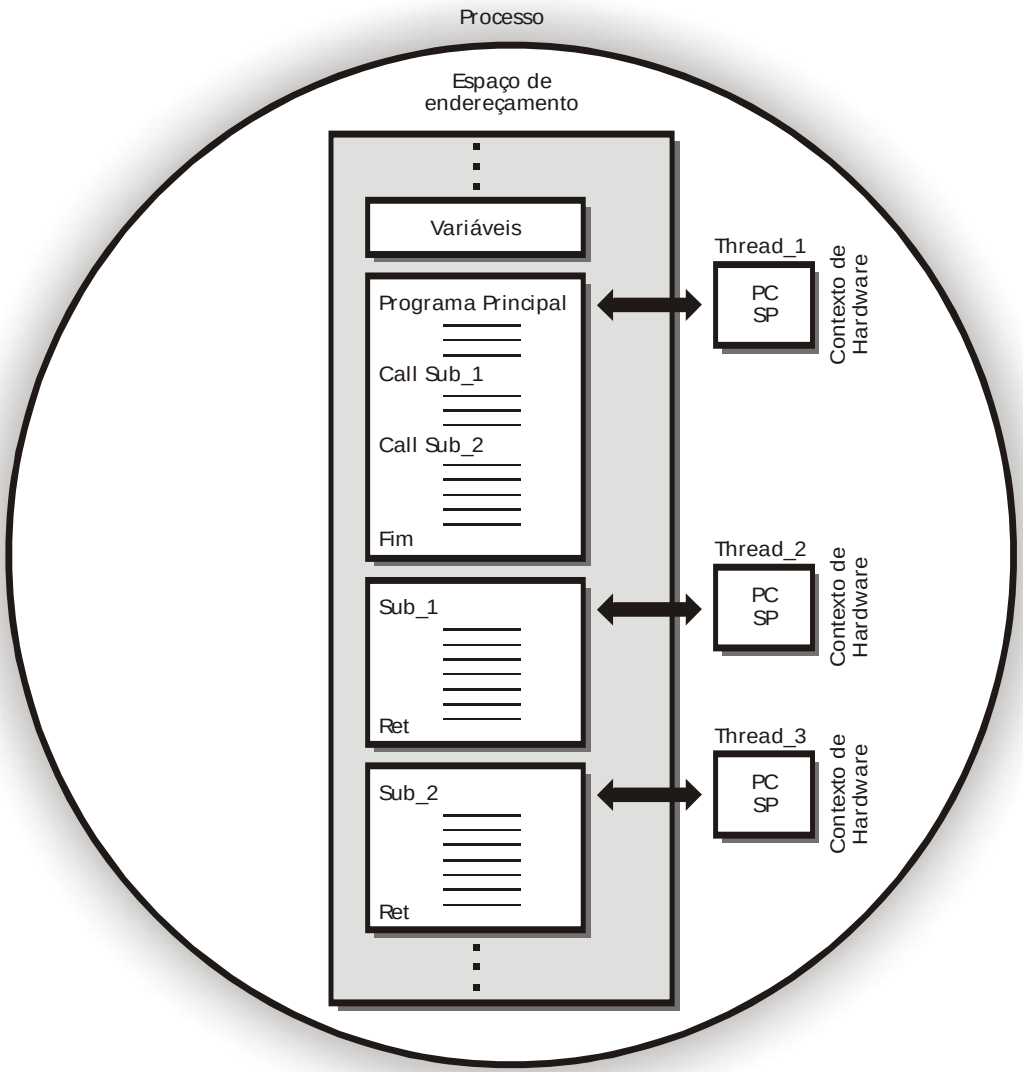
Ambiente Multithread



Ambiente Multithread [Machado]

No ambiente *multithread* os processos possuem várias *threads* que compartilham o mesmo espaço de endereçamento: permite o compartilhamento de dados, facilita a comunicação. Porém, requer mecanismos de sincronização para garantir acesso seguro aos dados.

Ambiente Multithread



Aplicação Multithread [Machado]

Inicialmente, o processo é criado apenas com a Thread_1.

Quando o programa principal chama as s programa principal.

Neste processo, **as três *threads* são executados concorrentemente**.

Por que usar threads?

➢Em um **processo com múltiplas *threads***, enquanto uma *thread* está **bloqueada** e esperando, uma segunda *thread* do mesmo processo pode ser **executada**

- A **cooperação** de múltiplas *threads* na mesma tarefa confere **maior *throughput*** e **melhor desempenho**
- As aplicações que precisam **compartilhar um *buffer*** comum (ou seja, como no exemplo do Produtor-Consumidor) **tiram proveito** da utilização de *threads*

➢A maioria dos *kernels* dos sistemas operacionais atualmente são *multithread*

- Várias *threads* operam no *kernel* e
- Cada *thread* executa uma tarefa específica (como, gerenciamento de dispositivos, gerenciamento de memória livre, manipulação de interrupções, etc.)

Benefícios com o uso de threads

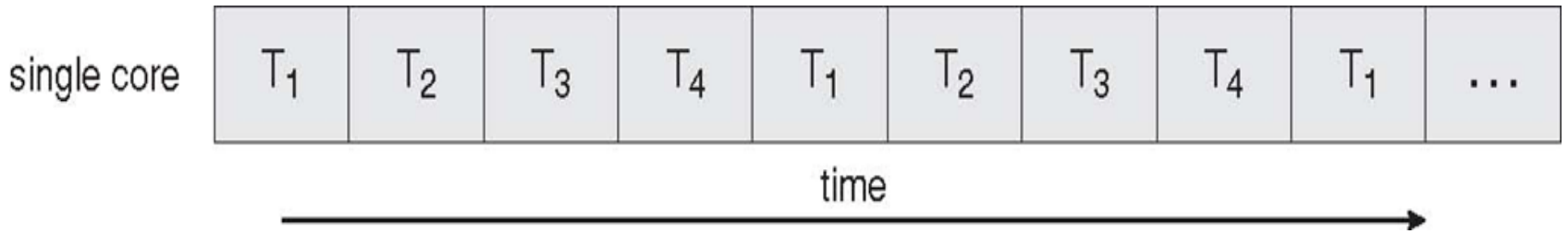
➢**Capacidade de Resposta**: o *multithreading* pode permitir que um programa continue executando mesmo se parte dele estiver bloqueada ou executando uma operação demorada

➢**Compartilhamento de Recursos**: *threads* compartilham a memória e os recursos do processo aos quais pertencem

➢**Economia**: devido ao compartilhamento de recursos, é mais econômico criar e realizar trocas de contexto de *threads*

➢**Utilização de Arquiteturas Multiprocessador**: cada *thread* pode estar executando em paralelo em um processador diferente

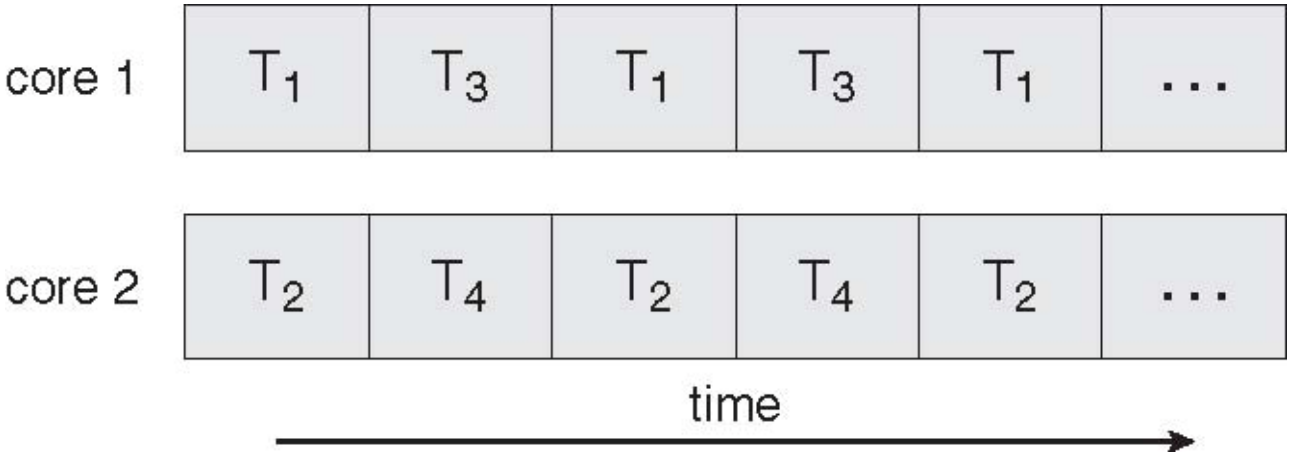
Execução concorrente em um sistema com um único núcleo (core)



Execução concorrente em um sistema com um único núcleo (core) [Silberschatz2]

- Supondo uma aplicação com 4 *threads*, em um sistema com um único processador (*core*)
  - Concorrência significa simplesmente que a execução das *threads* será intercalada com o passar do tempo, já que o núcleo de processamento pode executar apenas uma *thread* de cada vez

Execução paralela em um sistema multicore



Execução paralela em um sistema multicore [Silberschatz2]

- Em um sistema com com múltiplos núcleos (*multicore*)
  - As *threads* podem ser executadas em paralelo, já que o sistema pode atribuir uma *thread* separada a cada núcleo

Desafios na programação de sistemas multicore

- Projetistas de sistemas operacionais:
  - Devem escrever algoritmos de escalonamento que utilizem vários núcleos de processamento para possibilitar a programação paralela
- Programadores de aplicações:
  - Modificar programas existentes e o projeto de novos programas *multithread* para se beneficiar dos sistemas *multicore*
  - Principais desafios:
    - Divisão de atividades: analisar aplicações em busca de atividades que possam ser executadas concorrentemente
    - Equilíbrio: tarefas devem requerer o mesmo esforço (computacional) para serem executadas
    - Divisão de dados: devem ser também separados para execução
    - Dependência de dados: a execução de tarefas devem ser sincronizadas
    - Teste e depuração: mais difíceis de depurar e testar

Implementação de threads

➢O suporte a *threads* pode ser feito:

- No nível usuário **(*thread* de usuário):** implementado acima do *kernel*, através de um conjunto de chamadas de biblioteca no nível do usuário

- Pelo sistema operacional (***thread* de *kernel***): as *threads* são gerenciadas pelo *kernel*

- **Método Híbrido** implementa tanto a ***thread* de usuário** quanto a ***thread* de *kernel***

Implementação de threads no nível do usuário

- Implementadas por bibliotecas de *threads* no nível do usuário, sem necessidade da intervenção do *kernel*
  - Permite a implementação de *threads* em um SO que não dê suporte a *threads*
  - A biblioteca oferece rotinas para a criação/gerenciamento de *threads*, que são utilizadas pelos programadores da aplicação
  - É responsabilidade da aplicação gerenciar e sincronizar as várias *threads*
  - A troca de contexto entre *threads* (chaveamento) é feito por rotina local, que salva o estado da *thread* (é eficiente, pois não requer chamada ao *kernel*)
  - Uma chamada bloqueante (operação de E/S) de uma *thread* bloqueia todas as *threads* do processo
  - Não se beneficia do multiprocessamento, um processo *multithread* é visto como um processo com uma única *thread* pelo *kernel*
- Alguns exemplos:
  - *Pthreads* do POSIX

Implementação de threads no kernel

- Implementadas e gerenciados no espaço do *kernel* do próprio sistema operacional
  - O núcleo mantém uma tabela de *threads* para controlar todas as *threads* do sistema (além da tabela de processos)
  - A tabela contém informações como os registradores e o estado de de cada uma das *threads*
  - Possibilita que **múltiplas *threads*** executem em paralelo em um sistema com múltiplos processadores
  - Quando uma *thread* faz uma chamada de sistema bloqueante, outras *threads* não ficam bloqueadas
  - Há um custo maior para o gerenciamento de *threads*, uma vez que as operações sobre *threads* envolvem o núcleo do sistema, o que aumenta a sobrecarga
- Exemplos
  - Windows XP/2000
  - Solaris
  - Linux, Tru64 UNIX

Implementação de threads nos níveis usuário / kernel

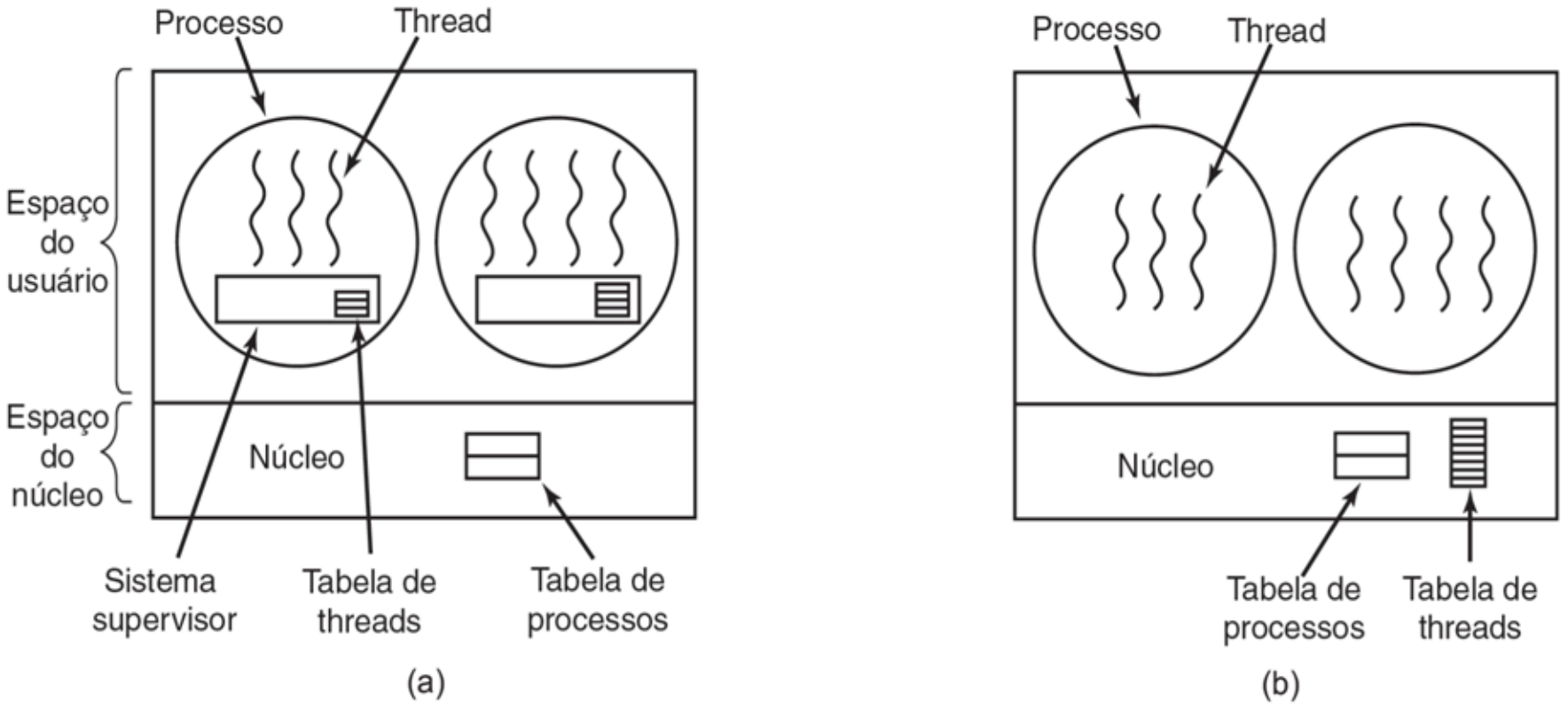


Figura 2.11 (a) Um pacote de threads de usuário. (b) Um pacote de threads administrado pelo núcleo. [Tanenbaum]

(a) O gerenciamento de *threads* é feito no espaço do usuário, cada processo mantém uma tabela de *threads* (contendo propriedades de cada *thread* – contador de programa, ponteiro de pilha, registradores, etc.). O sistema gerenciador contém rotinas para a criação e outras operações sobre as *threads*. (b) O gerenciamento de *threads* é feita pelo *kernel*, que mantém a tabela de *threads*, e possibilita o escalonamento de *threads* de um mesmo processo a diferentes processadores ao mesmo tempo.

Modelos de *Multithreading*

➢Há algumas estratégias comuns para o relacionamento entre *threads* de usuário e *threads* de kernel

- Modelo muitos-para-um
- Modelo um-para-um
- Modelo muitos-para-muitos

Modelo Muitos-para-Um

➢Mapeia **muitas *threads* de usuário** em **uma *thread* de *kernel***

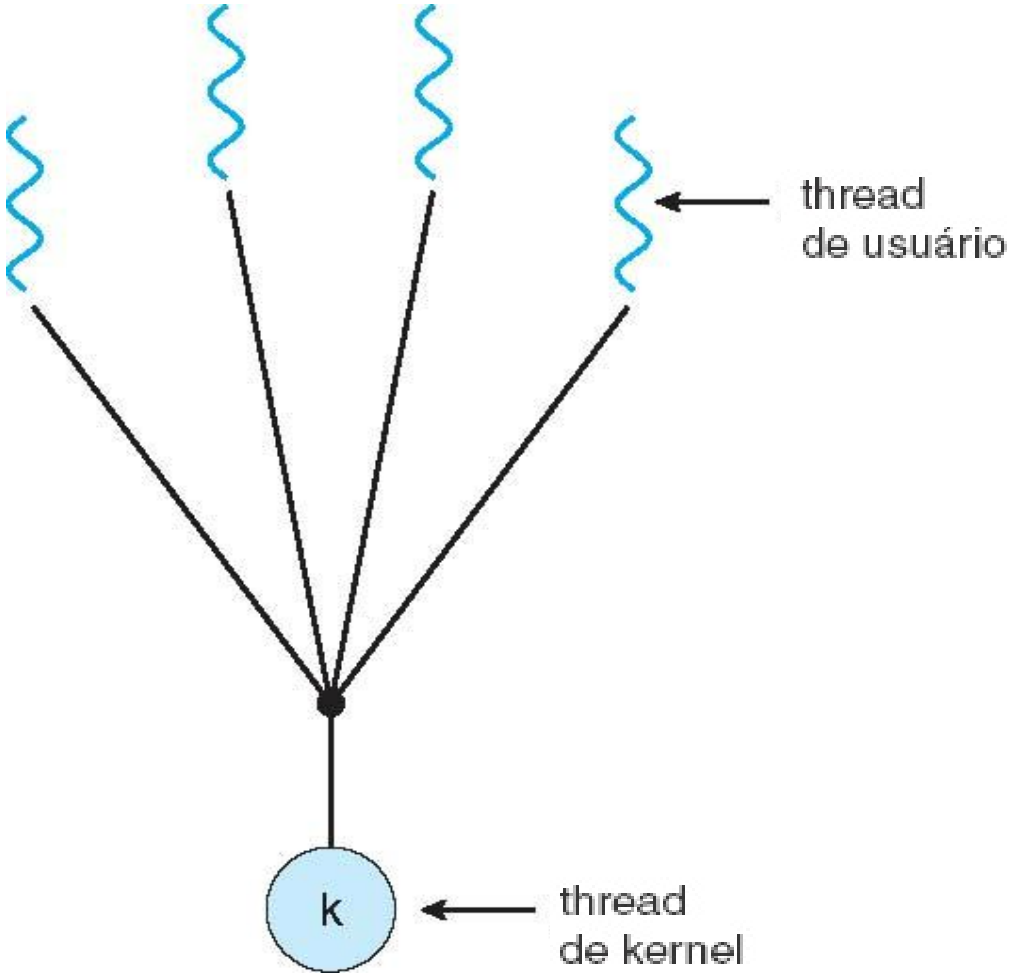
➢A gerência de *threads* é feita no espaço do usuário pela biblioteca de *threads*

- É eficiente pois não requer o envolvimento do *kernel*
- Porém, o processo inteiro será **bloqueado** se uma *thread* efetuar uma chamada bloqueante ao sistema (operação de E/S) → *thread* de *kernel* suspensa até o fim da operação
- Apenas uma *thread* pode acessar o *kernel* de cada vez, portanto, não permite execução paralela de *threads* em vários processadores

➢Exemplos

- Solaris Green Threads
- GNU Portable Threads

Representação do modelo Muitos-para-Um



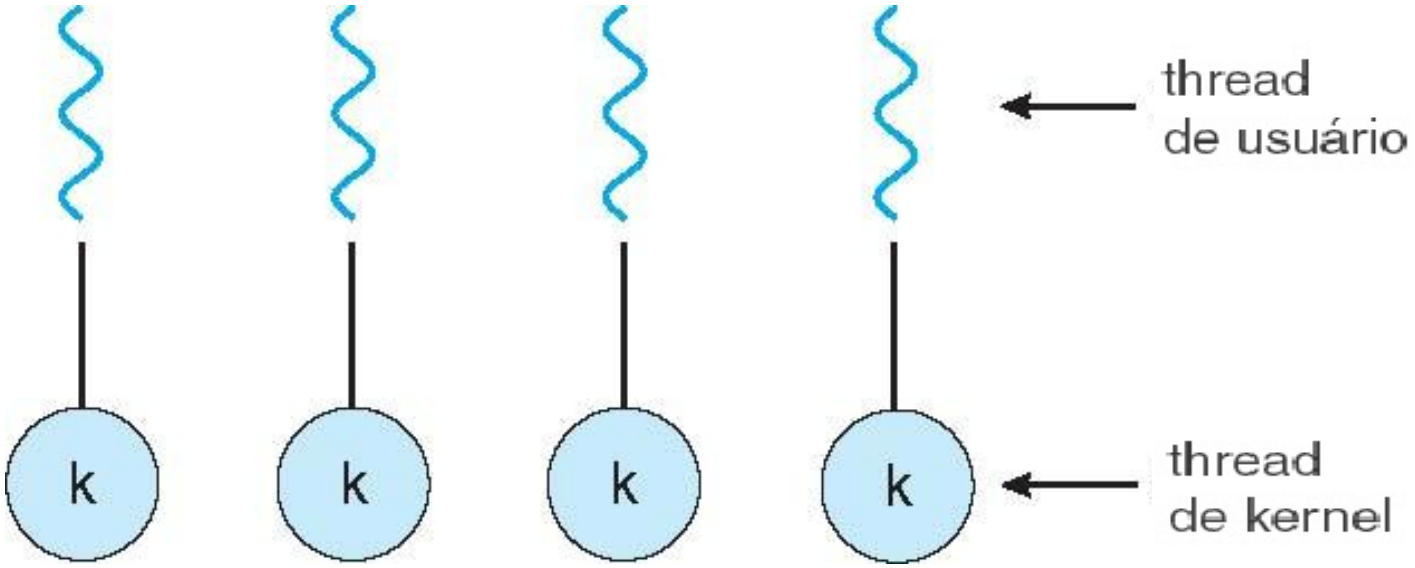
- A gerência de *threads* é feita no espaço do usuário pela biblioteca de *threads*
- É eficiente pois não requer o envolvimento do *kernel*
- Apenas uma *thread* pode acessar o *kernel* de cada vez

Modelo Muitos-para-Um [Silberschatz]

Modelo Um-para-Um

- Mapeia **cada *thread* de usuário** em **uma *thread* de *kernel***
- O gerenciamento das *threads* é feito pelo *kernel*
  - Fornece mais concorrência do que o modelo muitos-para-um, permitindo que outra ***thread*** execute quando uma ***thread*** efetuar uma chamada bloqueante ao sistema
  - Permite que **múltiplas *threads*** executem em paralelo em **multiprocessador**
  - Há um custo adicional para o gerenciamento de *threads*, pode prejudicar o desempenho quando há um grande número de *threads*
  - A maioria das implementações desse modelo colocam um limite no número de *threads* que podem ser criadas
- Exemplos
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 e acima

Representação do modelo Um-para-Um



Modelo Um-para-Um [Silberschatz]

- Suporte a threads do usuário no núcleo do sistema
  - Concorrência: uma operação bloqueante não bloqueia todo o processo, outras *threads* podem continuar executando
  - Permite que **múltiplas *threads*** executem em paralelo em **multiprocessador**

Modelo Muitos-para-Muitos

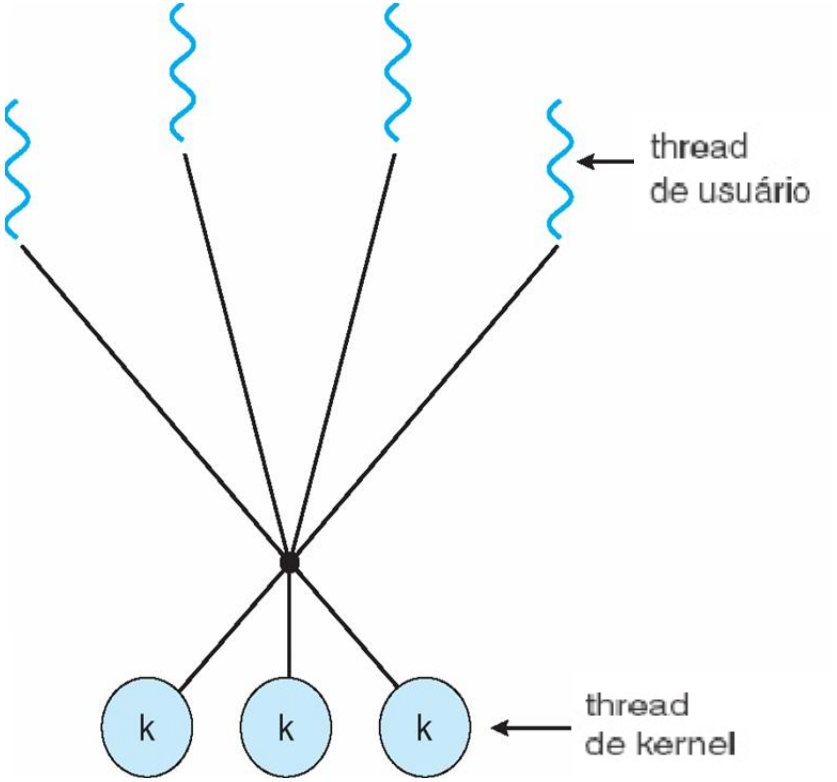
➢Permite que **muitas *threads* de usuário** sejam associadas a **muitas *threads* de *kernel***

- Permite que o sistema operacional crie um **número suficiente** de ***threads* de *kernel*** (pode ser ajustado de acordo com o número de CPUs existentes ou necessidades da aplicação)
- Assim como no modelo Um-para-Um, quando há uma operação bloqueante, o *kernel* pode selecionar outra *thread* para execução
- Exemplos
  - Solaris antes da versão 9
  - Windows NT/2000 com o pacote *ThreadFiber*

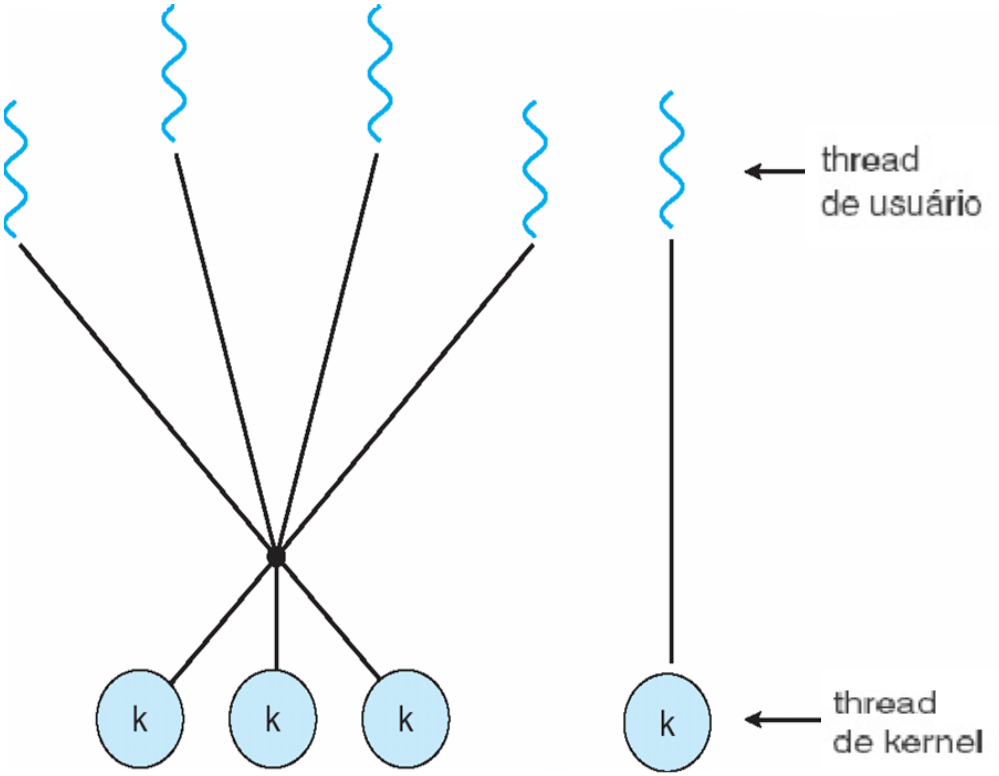
➢Uma variação comum desse modelo é possibilitar também que uma *thread* de usuário seja limitada a uma *thread* de kernel, conhecido como *Modelo de 2 níveis*

- Exemplos:
  - IRIX
  - HP-UX

Representação do modelo Muitos-para-Muitos



Modelo Muitos-para-Muitos [Silberschatz]



Modelo de 2 níveis [Silberschatz]

Bibliotecas de threads

- Uma biblioteca de *threads* oferece uma API para a criação e gerenciamento de *threads* ao programador de aplicações
- As bibliotecas de *threads* podem ser implementadas:
  - No espaço do usuário: código e estruturas de dados existem no espaço do usuário
  - No nível do *kernel*: com suporte do sistema operacional. O código e estruturas de dados existem no espaço do *kernel*

- Três principais bibliotecas de threads em uso:
  - Pthreads (padrão POSIX, pode ser fornecido como uma biblioteca no nível usuário ou kernel)
  - Threads do Win32 (biblioteca no nível kernel, disponível no Windows)
  - Threads do Java (como o Java roda em uma JVM, a implementação de threads depende do sistema operacional e do hardware onde roda a JVM, isto é, Pthreads ou Win32, dependendo do sistema)

Pthreads

- Padrão POSIX (IEEE 1003.1c) que define uma API para criação e sincronismo de *thread*
  - Trata-se de uma especificação, não é a implementação
  - Vários sistemas operacionais implementam a especificação como Solaris, Linux, Mac OSX, Tru64, e também versões *shareware* em domínio público para o Windows

- A seguir um exemplo de programa em C utilizando Pthreads que calcula o somatório de um inteiro (não negativo) em uma *thread* separada

Pthreads

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }

   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}
```

```
/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Exemplo usando Pthr

[Silberschatz]

Win32

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */

DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;        //DWORD: tipo inteiro de 32 bits sem sinal
   HANDLE ThreadHandle;
   int Param;
   /* perform some basic error checking */
   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }

   // create the thread
   ThreadHandle = CreateThread(
      NULL, // default security attributes
      0, // default stack size
      Summation, // thread function
      &Param, // parameter to thread function
      0, // default creation flags
      &ThreadId); // returns the thread identifier

   if (ThreadHandle != NULL) {
      // now wait for the thread to finish
      WaitForSingleObject(ThreadHandle,INFINITE);

      // close the thread handle
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
   }
}
```

Exemplo usando AP
(é similar ao Pthreds

[Silberschatz]

Threads em Java

- O suporte para *threads* é fornecido ou pelo Sistema Operacional (**nível de *kernel***) ou por uma biblioteca de *threads* (**nível de usuário**)
- Java fornece suporte no **nível de linguagem** para criação e gerência de *threads*
- Como as ***threads* Java** não são gerenciados por bibliotecas e sim pela **Máquina Virtual Java (JVM)**, é difícil classificar as *threads* Java como de nível de usuário ou de *kernel*
- Todos os programas Java compreendem pelo menos um *thread* de controle
- Podem ser criadas através de:
  - Extensão da classe *Thread* e redefinição do método *run()*
  - Implementação da interface *Runnable*

Estendendo a classe *Thread*

```
class Worker1 extends Thread {

   public void run() {

      System.out.println("Eu sou uma Thread Operária");

   } // (o método run() é chamado pelo método start()

}
```

```
public class First {

   public static void main(String args[]) {

      Worker1 w1 = new Worker1();

      w1.start(); // (aqui a nova thread é realmente criada!)

      System.out.println("Eu sou a Thread Principal");

   }

}
```

Após a chamada de **start()**, a *thread* **w1** inicia a execução e a *thread* do programa principal continua a execução. Usando **w1.join()** fará com que a thread principal espere o término de **w1**.

Quando este programa executa, duas *threads* são criadas pela JVM: a *thread* que começa a execução do método *main()* e a *thread w1.*

Ao chamar *start()*, é inicializada uma nova *thread* na JVM e o método *run()* é invocado para iniciar a sua execução.

A Interface *Runnable*

➢Outra opção para criar uma thread separada é definir uma classe que implementa a interface *Runnable*, definida da seguinte forma:

```
➢public interface Runnable {
➢    public abstract void run();
➢}
```

➢Implementar a interface *Runnable* é similar a estender a classe *Thread*: em ambos os casos é preciso implementar o método **run()**

➢Porém criar uma nova *thread* a partir da classe que implementa *Runnable* é ligeiramente diferente de criar uma thread de uma classe que estende *Thread*

Implementando a interface Runnable

```
class Worker2 implements Runnable {
    public void run() {
        System.out.println("Eu sou um Thread Operário");
    }
}
```

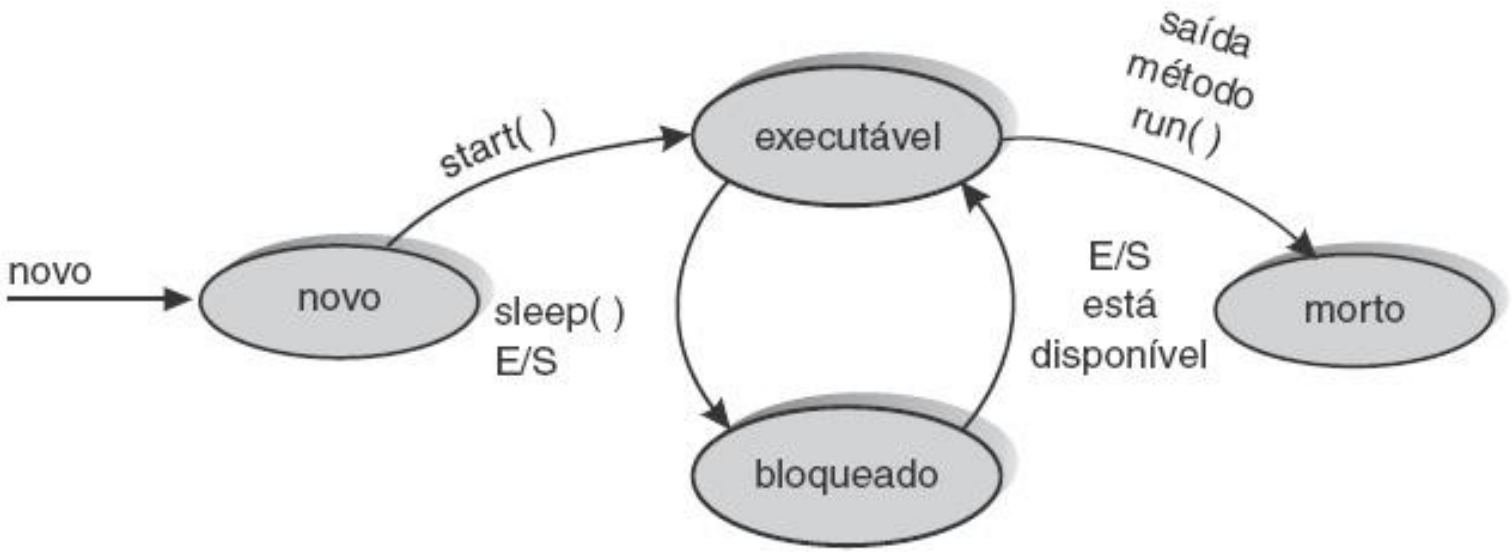
```
public class Second {
    public static void main(String args[]) {
        Worker2 w2 = new Worker2();
        Thread thrd = new Thread(w2);
        w2.start();
        System.out.println("Eu sou o Thread Principal");
    }
}
```

Como **Java não suporta herança múltipla**, se uma classe já for derivada de outra, não pod

Manipulando threads em Java

- Alguns métodos da classe Thread:
  - **t.start()**
    - Inicia a execução da *thread* t, chama o método run() da *thread*
  - **t.join()**
    - Faz com a *thread* que esteja executando suspenda e aguarde que a *thread* t execute (até terminar)
  - **t.sleep(tempo)**
    - Faz com que a *thread* t suspenda a execução por um tempo em milisegundos
  - **t.interrupted()**
    - Interrompe a execução da *thread*
  - **t.isAlive()**
    - Retorna verdadeiro se a *thread* está viva (se iniciou mas não terminou)

Estados de uma *thread*



[Silberschatz]

Estados e transições possíveis de uma *thread*. Esses estados se referem à máquina virtual Java e não necessariamente estão associados ao estado da *thread* executando no SO

Não é possível definir o estado exato de uma *thread*, embora o método ***isAlive()*** retorne um **valor booleano** que um programa pode usar para determinar se uma *thread* está ou não no **estado morto (terminado)**

Problema do Produtor-Consumidor

```
public class Factory  {
    // construtor
public Factory()   {
      // primeiro cria o buffer de mensagens (caixa de correio
      Channel mailBox = new MessageQueue();   // compartilhada)

      // agora cria os threads do produtor e do consumidor
      Thread producerThread = new Thread(new Producer(mailBox));
      Thread consumerThread = new Thread(new Consumer(mailBox));

      producerThread.start();
      consumerThread.start();
   }


   public static void main(String args[]) {
      Factory server = new Factory();
   }
}
```

Passa a referência para a caixa de correio (mailbox) na criação do Produtor e Consumidor.

Obs.: Código disponível no Tidia em Repositório → Exemplos

Classe Factory [Silberschatz]

Caixa de correio e interface

```
public class MessageQueue implements Channel
{
    private Vector queue;

    public MessageQueue() {
       queue = new Vector();
    }

    // This implements a nonblocking send
    public void send(Object item) {
       queue.addElement(item);
    }

    // This implements a nonblocking receive
    public Object receive() {
       if (queue.size() == 0)
          return null;
       else
          return queue.remove(0);
    }
}
```

```
public interface Channel
{
   // Send a message to the channel
   public abstract void send(Object item);

   // Receive a message from the channel
   public abstract Object receive();
}
```

Caixa de correio e interface para a troca de mensagem [Silberchatz]

Thread do Produtor

```
class Producer implements Runnable {
   private  Channel mbox;

   public Producer(Channel mbox) {
      this.mbox = mbox;
   }

   public void run() {
      Date message;

      while (true) {
         SleepUtilities.nap(); // dorme por um tempo
         message = new Date(); // produz um item
         System.out.println("Produtor produziu " +  message);

         // Insere o item produzido no buffer
         mbox.send(message);
      }
   }
}
```

O construtor recebe a referência para a caixa de mensagem (mailbox).

Thread do consumidor

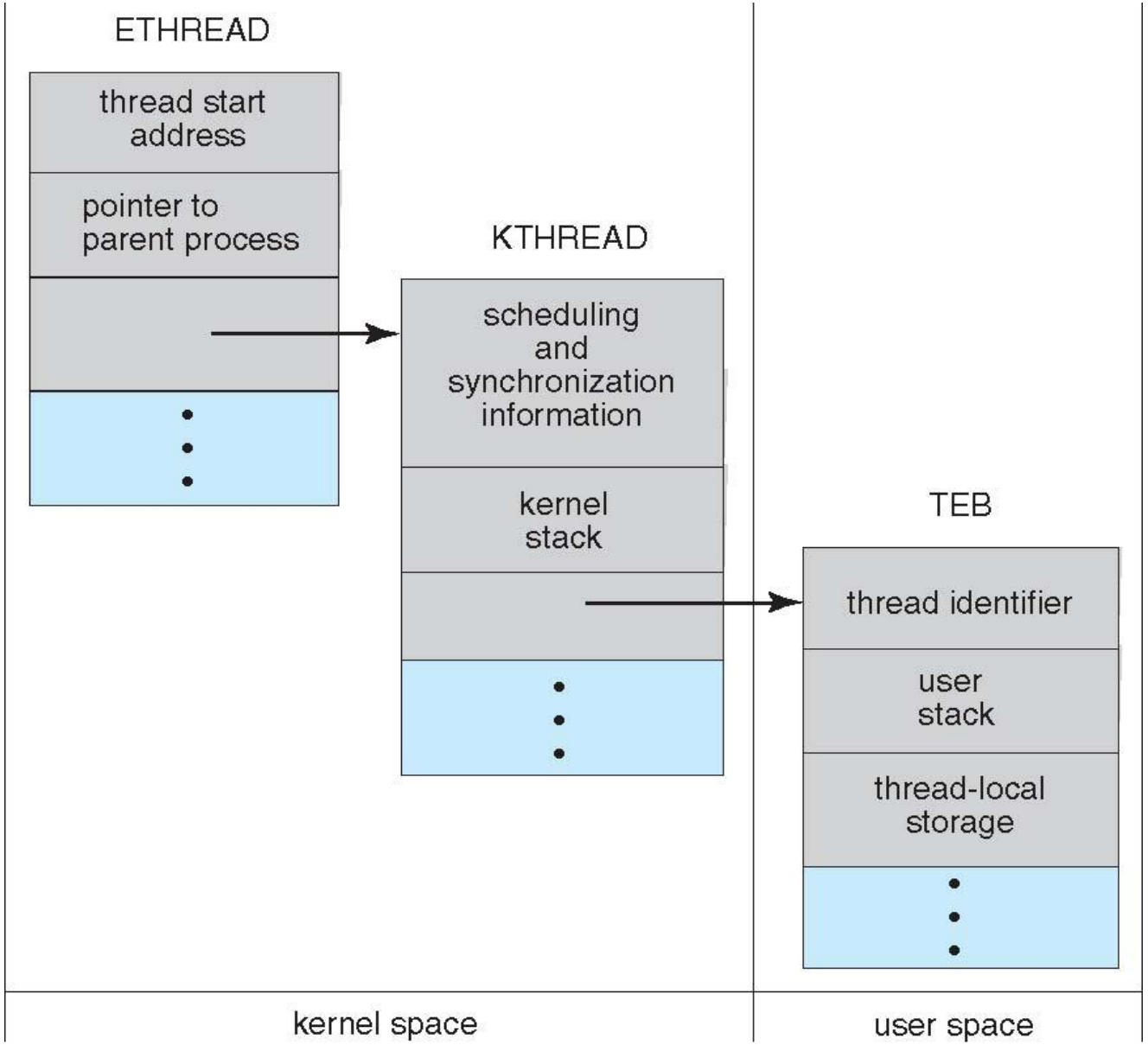
```
class Consumer implements Runnable {
   private  Channel mbox;

   public Consumer(Channel mbox) {
      this.mbox = mbox;
   }

   public void run() {
      Date message;

      while (true) {
         SleepUtilities.nap(); // dorme por um tempo
         // consome um item do buffer
         message = (Date)mbox.receive();
         if (message != null)
            System.out.println("Consumidor consumiu " +  message);
      }
   }
}
```

O construtor recebe a referência para a caixa de mensagem (mailbox).

Threads no Windows XP

- Implementa o **mapeamento um-para-um**
- Cada *thread* contém
  - Uma ID de *thread* identificando unicamente a *thread*
  - Um conjunto de registradores representando o estado do processador
  - Pilhas do usuário e do *kernel* separadas, usadas quando a *thread* está executando em **modo usuário** ou **monitor**
  - Área de armazenamento privada
- O **conjunto de registradores**, as **pilhas** e a **área de armazenamento privada** são conhecidos como o **contexto das *threads***
- As estruturas de dados primárias de uma *thread* incluem:
  - ETHREAD (bloco de *thread* do executivo) – inclui um ponteiro para o processo ao qual a *thread* pertence
  - KTHREAD (bloco de *thread* do *kernel*) – informação sobre o escalonamento
  - TEB (bloco do ambiente da *thread*) – memória local da *thread*

Threads no Windows XP



Estruturas de dados de uma thread no Windows XP [Silberchatz]

Threads no Linux

- O **Linux** refere-se tanto aos **processos** como as ***threads*** como ***tarefas***, mas oferece uma **chamada ao sistema** para criar **processos leves** que se comportam como ***threads***
- A criação de um *thread* é feita através da chamada de sistema **clone()**
- **clone()** permite que uma tarefa filha compartilhe o espaço de endereços da tarefa pai (o processo)
- Quando a chamada ao sistema ***fork*** é invocada, um **novo processo** é criado com uma ***cópia*** de todas as estruturas de dados associadas do **processo pai**
- Quando a chamada ao sistema ***clone*** é invocada, um **novo processo** é criado, porém o novo processo ***aponta*** para as estruturas de dados do **processo pai**
  - É possível determinar quanto compartilhamento deverá ocorrer entre entre as tarefas pai e filha

Threads no Linux

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Exemplos de flags [Silberschatz]

➢É possível determinar quanto compartilhamento deverá ocorrer entre entre as tarefas pai e filha através de flags, conforme a tabela

- Usando esses flags, as tarefas pai e filha compartilharão a mesma informação do sistema de arquivos (diretório de trabalho atual), o mesmo espaço de memória, arquivos abertos
- Se nenhum flag for definido na chamada de clone(), não ocorre nenhum compartilhamento (semelhante à chamada fork())

Referências

- [Silberschatz] SILBERCHATZ, A., GALVIN, P. B. e GAGNE, G. **Sistemas Operacionais com Java**. 7ª ed., Rio de Janeiro: Elsevier, 2008.
- [Tanenbaum] TANENBAUM, A. **Sistemas Operacionais Modernos**. 3ª ed. São Paulo: Prentice Hall, 2009.
- [MACHADO] MACHADO, F. B. e MAIA, L. P. **Arquitetura de Sistemas Operacionais**. 4ª ed., Rio de Janeiro: LTC, 2007.
- [Silberschatz2] SILBERCHATZ, A., GALVIN, P. B. e GAGNE, G. **Fundamentos de Sistemas Operacionais**. 8ª ed., Rio de Janeiro: LTC, 2012.