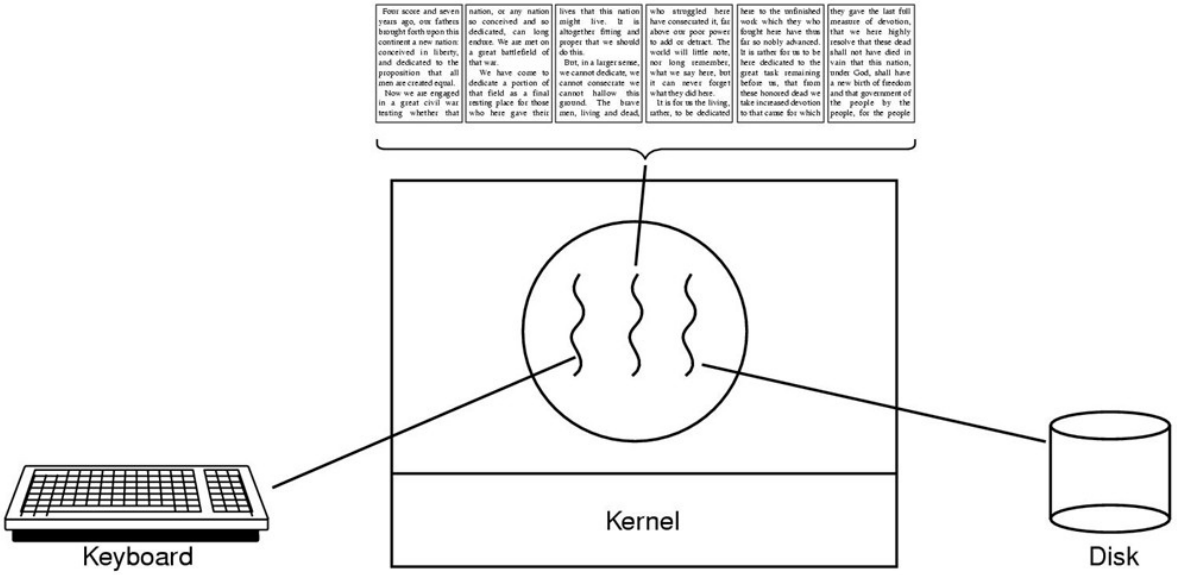
# Threads

# Fluxos de Execução

- Um programa seqüencial consiste de um único fluxo de execução, o qual realiza uma certa tarefa computacional.
  - A maioria dos programas simples tem essa característica: só possuem um único fluxo de execução. Por conseguinte, não executam dois trechos de código “simultaneamente”.
- Grande parte do software de maior complexidade  escrito hoje em dia faz uso de mais de uma linha de execução.

# Exemplos de Programas MT (1)

- Editor de Texto
  - Permite que o usuário edite o arquivo enquanto ele ainda está sendo carregado do disco.
  - Processamento assíncrono (salvamento periódico).



# Exemplos de Programas MT (2)

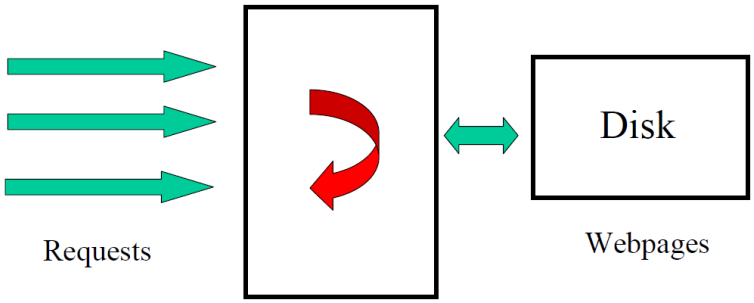
- Navegador (browser)
  - Consegue fazer o *download* de vários arquivos ao mesmo tempo, gerenciando as diferentes velocidades de cada servidor e, ainda assim, permitindo que o usuário continue interagindo, mudando de página enquanto os arquivos estão sendo carregados.
- Programas numéricos (ex: multiplicação de matrizes):
  - Cada elemento da matriz produto pode ser calculado independentemente dos outros; portanto, podem ser facilmente calculados por threads diferentes.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} e & f \\ g & h \end{pmatrix}=\begin{pmatrix} a.e+b.g & a.f+b.h \\ c.e+d.g & c.f+d.h \end{pmatrix}$$

# Exemplos de Programas MT (3)

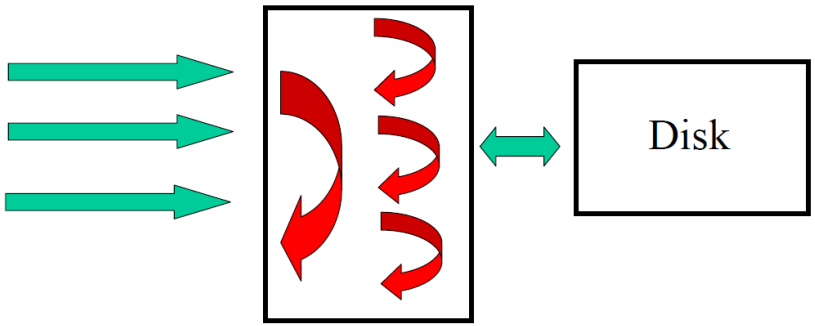
- Servidor Web

Single Threaded Web Server




Cannot overlap Disk I/O with listening for requests

Multi Threaded Web Server



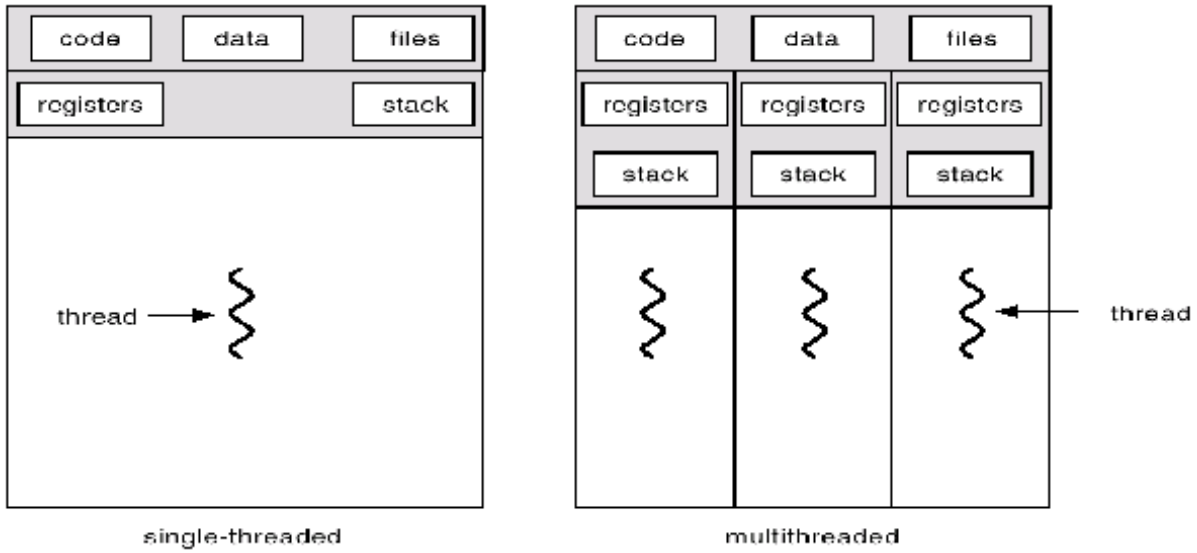
# Threads (1)

- Thread:
  - Thread = “fluxo”, “fio”.
  - Fluxo de execução dentro de um processo (seqüência de instruções a serem executadas dentro de um programa).
- Thread é uma abstração que permite que uma aplicação execute mais de um trecho de código simultaneamente. (ex: um método).
  - Processos permitem ao S.O. executar mais de uma aplicação ao mesmo tempo.
- Um programa *multithreading* pode continuar executando e respondendo ao usuário mesmo se parte dele está bloqueada ou executando uma tarefa demorada.

# Threads (2)

- Uma tabela de *threads*, denominada *Task Control Block,* é mantida para armazenar informações individuais de cada fluxo de execução.
- Cada thread tem a si associada:
  - Thread ID
  - Estado dos registradores, incluindo o PC
  - Endereços da pilha
  - Máscara de sinais
  - Prioridade
  - Variáveis locais e variáveis compartilhadas com as outras *threads*
  - Endereços das *threads* filhas
  - Estado de execução (pronta, bloqueada, executando)

# Threads (3)


code
data
files
registers
stack
thread
single-threaded
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded

# Threads (4)

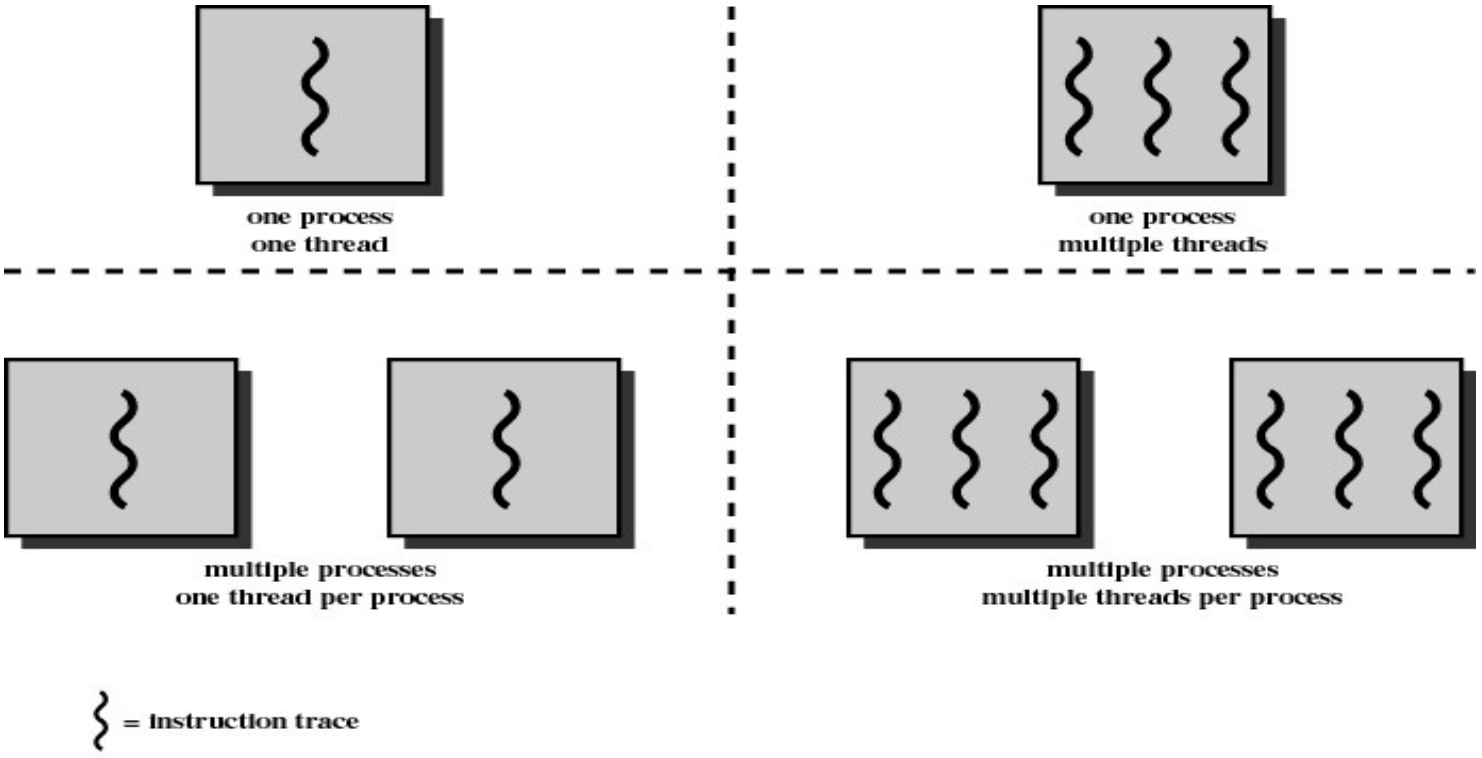



**Figure 4.1 Threads and Processes [ANDE97]**

# Modelo de Processo Multithreading

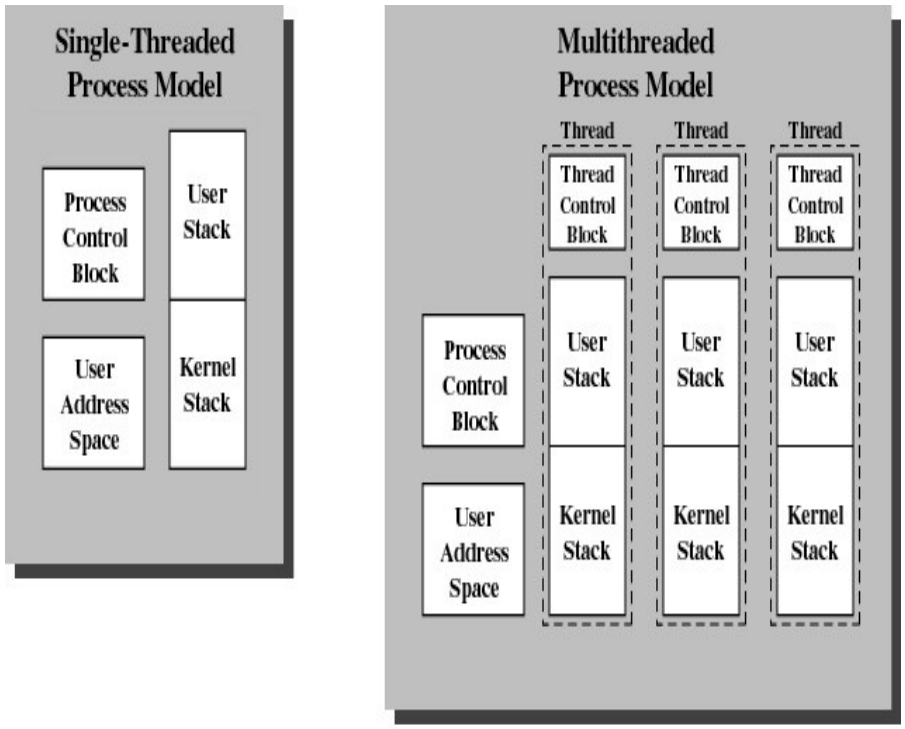
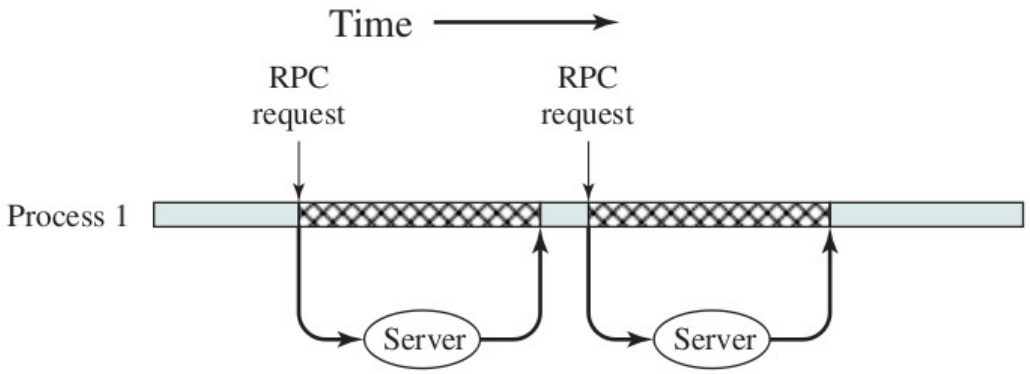
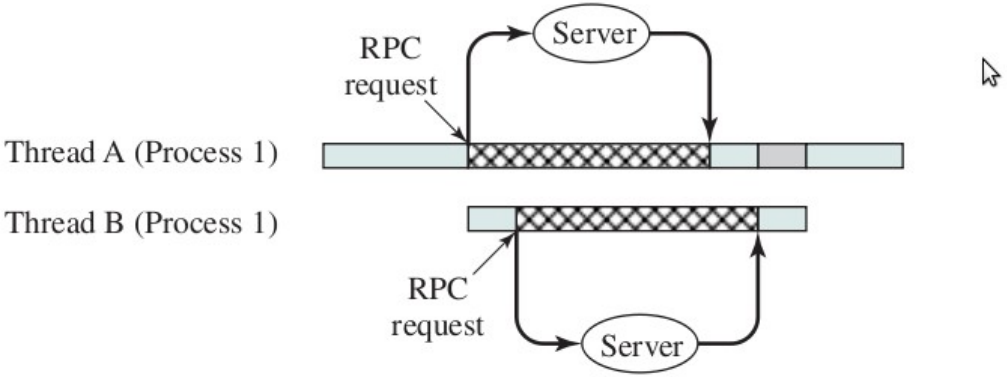



Figure 4.2 Single Threaded and Multithreaded Process Models

# Exemplo: RPC




(a) RPC using single thread




(b) RPC using one thread per server (on a uniprocessor)

# Threads e Processos (1)

- Existem duas características fundamentais que são usualmente tratadas de forma independente pelo S.O:
  - Propriedade de recursos ("*resource ownership*");
  - Escalonamento ("*scheduling / dispatching*").
- Propriedade de recursos:
  - Trata dos recursos alocados aos processos, e que são necessários para a sua execução.
  - Ex: memória, arquivos, dispositivos de E/S, etc.
- Escalonamento:
  - Relacionado à unidade de despacho do S.O.
  - Determina o fluxo de execução (trecho de código) que é executado pela CPU.

# Threads e Processos (2)

- Tradicionalmente o processo está associado a:
  - um programa em execução
  - um conjunto de recursos
- Em um S.O. que suporta múltiplas threads:
  - Processos estão associados somente à propriedade de recursos
  - *Threads* estão associadas às atividades de execução (ou seja, *threads* constituem as unidades de escalonamento em sistemas *multithreading*).

# S.O. Multithreading

- Multithreading refere-se à habilidade do *kernel* do S.O. em suportar múltiplas threads concorrentes em um mesmo processo.
- Exemplos:
  - MS-DOS: suporta uma única *thread*.
  - Unix “standard”: suporta múltiplos processos, mas apenas uma *thread* por processo.
  - Windows 2k, Linux, Solaris:  suportam múltiplas *threads* por processo.
- Em um ambiente *multithreaded*:
  - processo é a unidade de alocação e proteção de recursos;
  - processo tem um espaço de endereçamento virtual (imagem);
  - processo tem acesso controlado a outros processos, arquivos e outros recursos;
  - *thread*  é a  unidade de escalonamento;
  - *threads* compartilham o espaço de endereçamento do processo.

# Vantagens das Threads sobre Processos (1)

- A criação e terminação de uma *thread* é mais rápida do que a criação e terminação de um processo pois elas não têm quaisquer recursos alocados a elas.
  - (S.O. Solaris) Criação = 30:1, Troca de contexto = 5:1
- A comutação de contexto entre *threads* é mais rápida do que entre dois processos, pois elas compartilham os recursos do processo.
  - (S.O. Solaris) Troca de contexto = 5:1
- A comunicação entre *threads* é mais rápida do que a comunicação entre processos, já que elas compartilham o espaço de endereçamento do processo.
  - O uso de variáveis globais compartilhadas pode ser controlado através de primitivas de sincronização (monitores, semáforos, etc).

# Vantagens das Threads sobre Processos (2)

- É possível executar em paralelo cada uma das *threads* criadas para um mesmo processo usando diferentes CPUs.
- Primitivas de sinalização de fim de utilização de recurso compartilhado também existem. Estas primitivas permitem “acordar” uma ou mais *threads* que estavam bloqueadas.

# Bibliotecas de Threads (2)

- Uma biblioteca de threads contém código para:
  - criação e sincronização de *threads*
  - troca de mensagens e dados entre *threads*
  - escalonamento de *threads*
  - salvamento e restauração de contexto
- Na compilação:
  - Incluir o arquivo *pthreads.h*
  - “Linkar” a biblioteca *pthread*

```
$ gcc simple_threads.c -o simple -pthread
```

# Biblioteca Pthreads – Algumas Operações

| POSIX function | description |
|---|---|
| pthread_cancel() | terminate another thread |
| pthread_create() | create a thread |
| pthread_detach() | set thread to release resources |
| pthread_equal() | test two thread IDs for equality |
| pthread_exit() | exit a thread without exiting process |
| pthread_kill() | send a signal to a thread |
| pthread_join() | wait for a thread |
| pthread_self() | find out own thread ID |

# Thread APIs vs. System calls para Processos

| *Pthread API* | *system calls for process* |
|---|---|
| `Pthread_create()` | `fork()`, `exec*()` |
| `Pthread_exit()` | `exit()`, `_exit()` |
| `Pthread_self()` | `getpid()` |
| `sched_yield()` | `sleep()` |
| `pthread_kill()` | `kill()` |
| `Pthread_cancel()` | |
| `Pthread_sigmask()` | `sigmask()` |

# Bibliotecas de Threads (1)

- A interface para suporte à programação *multithreading* é feita via bibliotecas:
  - *libpthread* (padrão POSIX/IEEE 1003.1c)
  - *libthread* (Solaris).
- POSIX Threads ou *pthreads* provê uma interface padrão para manipulação de *threads*, que é independente de plataforma (Unix, Windows, etc.).

# Criação de Threads: pthread_create() (1)

- A função `pthread_create()` é usada para criar uma nova *thread* dentro do processo.

```
int pthread_create(
   pthread_t *restrict thread,
   const pthread_attr_t *restrict attr,
   void *(*start_routine)(void *),
   void *restrict arg);
```

- `pthread_t *thread` – ponteiro para um objeto que recebe a identificação da nova *thread*.
- `pthread_attr_t *attr` – ponteiro para um objeto que provê os atributos para a nova *thread*.
- `start_routine` – função com a qual a *thread* inicia a sua execução
- `void *arg` – arqumentos inicialmente passados para a função

# Finalizando uma Thread: pthread_exit()

- A invocação da função `phtread_exit()` causa o término da *thread* e libera todos os recursos que ela detém.

```
void pthread_exit(void *value_ptr);
```

- `value_ptr` – valor retornado para qualquer *thread* que tenha se bloqueado aguardando o término desta *thread.*

- Não há necessidade de se usar essa função na *thread* principal, já que ela retorna automaticamente.

# Esperando pelo Término da Thread: pthread_join() (1)

- A função `phtread_join()` suspende a execução da *thread* chamadora até que a *thread* especificada no argumento da função acabe.
- A *thread* especificada deve ser do processo corrente e não pode ser *detached*.

```
int pthread_join(thread_t tid, void **status)
```

- `tid` – identificação da *thread* que se quer esperar pelo término.
- `*status` – ponteiro para um objeto que recebe o valor retornado pela *thread* acordada.

# Esperando pelo Término da Thread: pthread_join() (2)

- Múltiplas *threads* não podem esperar pelo término da mesma *thread*. Se elas tentarem, uma retornará com sucesso e as outras falharão com erro ESRCH.
- Valores de retorno:
  - `ESRCH` – `tid` não é uma thread válida, undetached do processo corrente.
  - `EDEADLK` – `tid` especifica a *thread* chamadora.
  - `EINVAL` – o valor de `tid` é inválido.

# Retornando a Identidade da Thread: pthread_self()

- A função `pthread_self()` retorna um objeto que é a identidade da *thread* chamadora.

```
#include <pthread.h>

pthread_t pthread_self(void);
```