SISTEMAS OPERACIONAIS

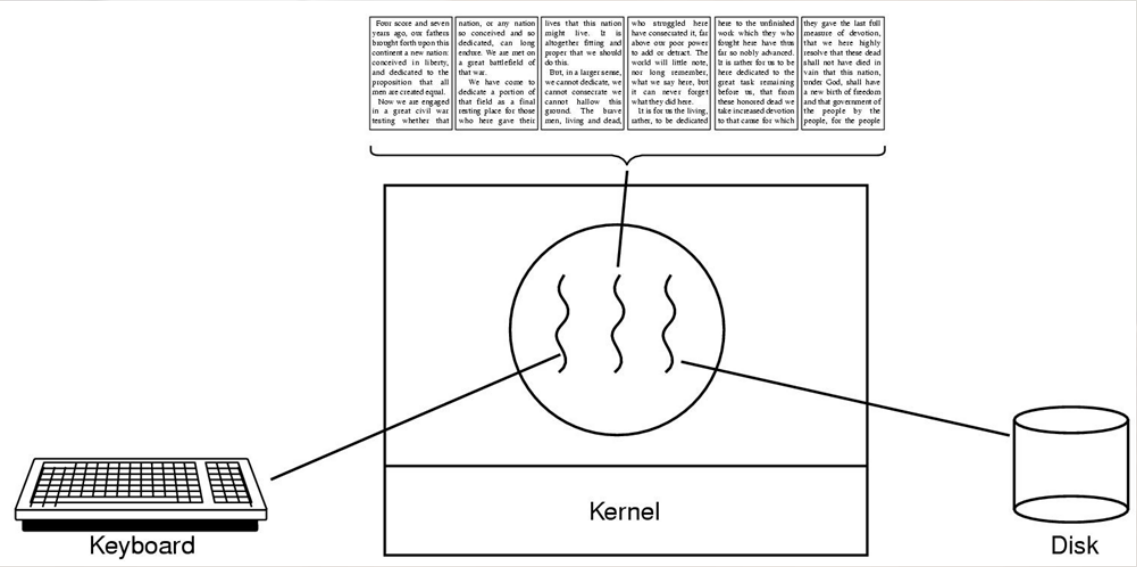
Threads

Fluxos de Execução

- Um programa sequencial 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)

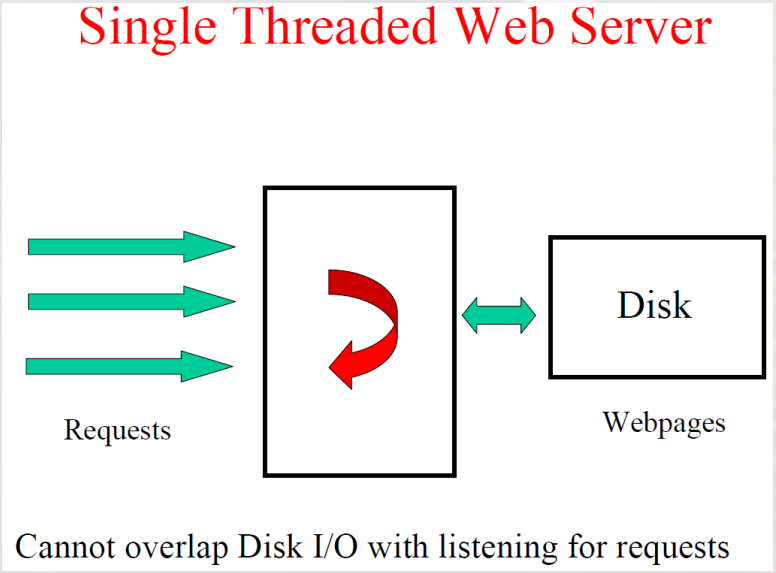
- Navegador (browser)
  - Consegue fazer o download de vários arquivos ao mesmo tempo, gerenciando as diferentes velocidades de cada servidor e, ainda assim, permitindo que o usuário continue interagindo, mudando de página enquanto os arquivos estão sendo carregados

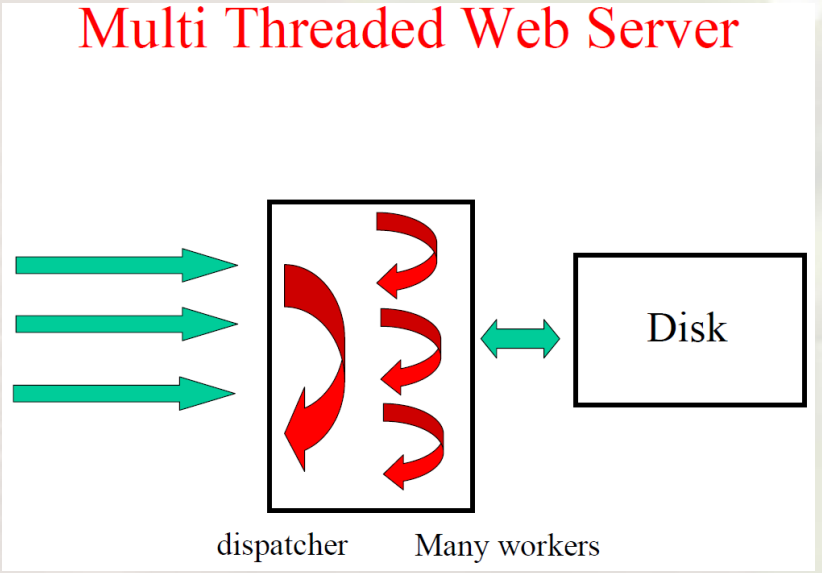
- Programas numéricos (ex: multiplicação de matrizes):
  - Cada elemento da matriz produto pode ser calculado independentemente dos outros; portanto, podem ser facilmente calculados por threads diferentes.

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

Exemplos de Programas MT (3)

- Servidor Web





Threads (1)

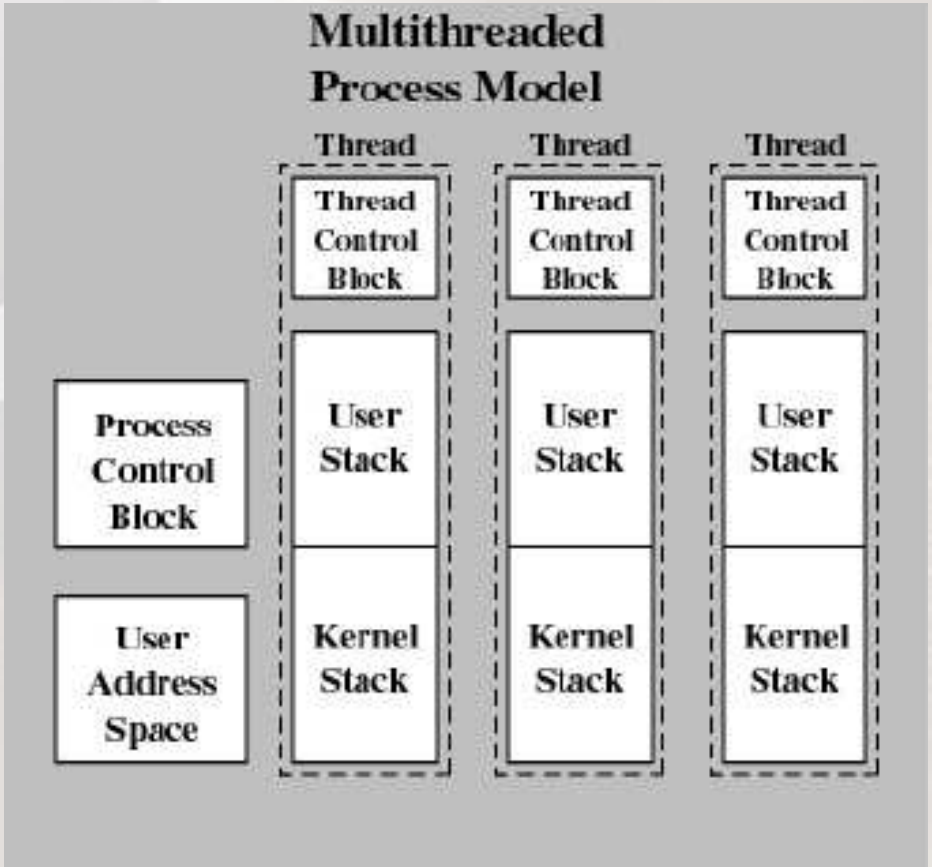
- Thread:
  - Thread = “fluxo”, “fio”.
  - Fluxo de execução dentro de um processo (sequência de instruções a serem executadas dentro de um programa).
- Thread é uma abstração que permite que uma aplicação execute mais de um trecho de código simultaneamente. (ex: um método).
  - Processos permitem ao 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)

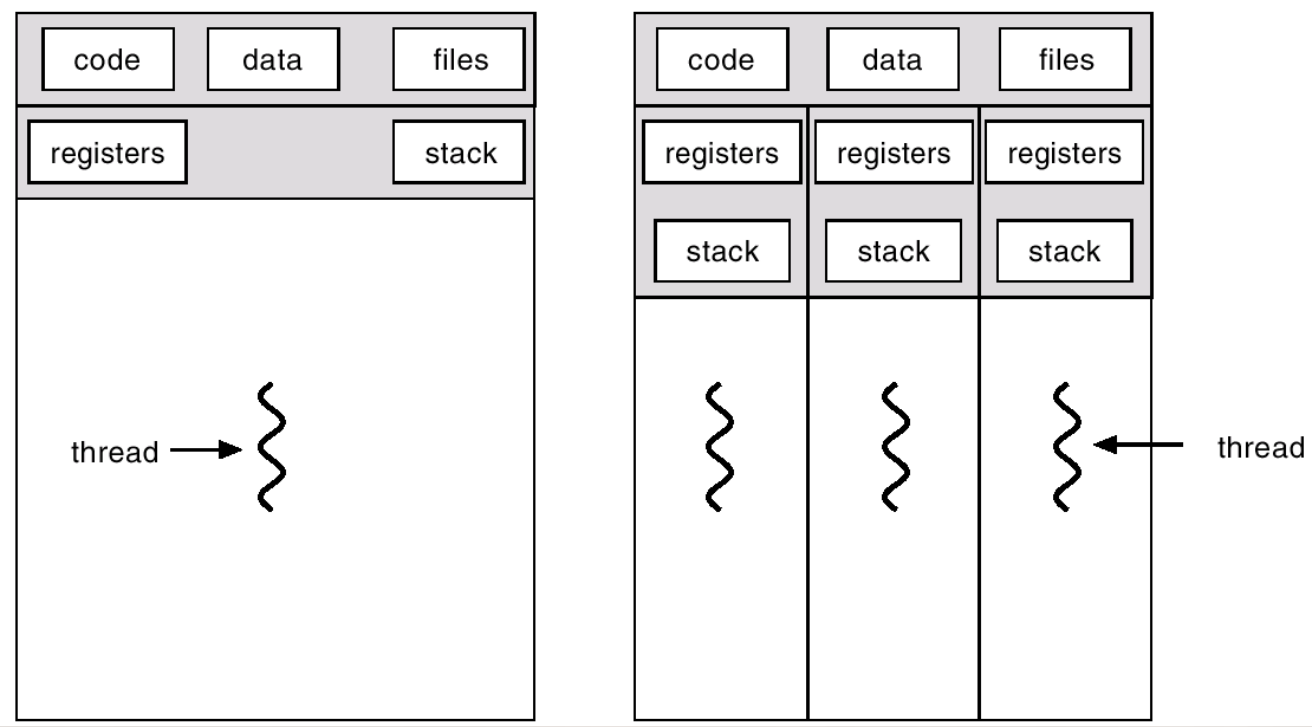
- Uma tabela de threads, denominada Task Control Block, é mantida para armazenar informações individuais de cada fluxo de execução
- Cada thread tem a si associada:
  - Thread ID
  - Estado dos registradores, incluindo o PC
  - Endereços da pilha
  - Máscara de sinais
  - Prioridade
  - Variáveis locais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

Threads (3)

- Estrutura de um processo com *multithreading*

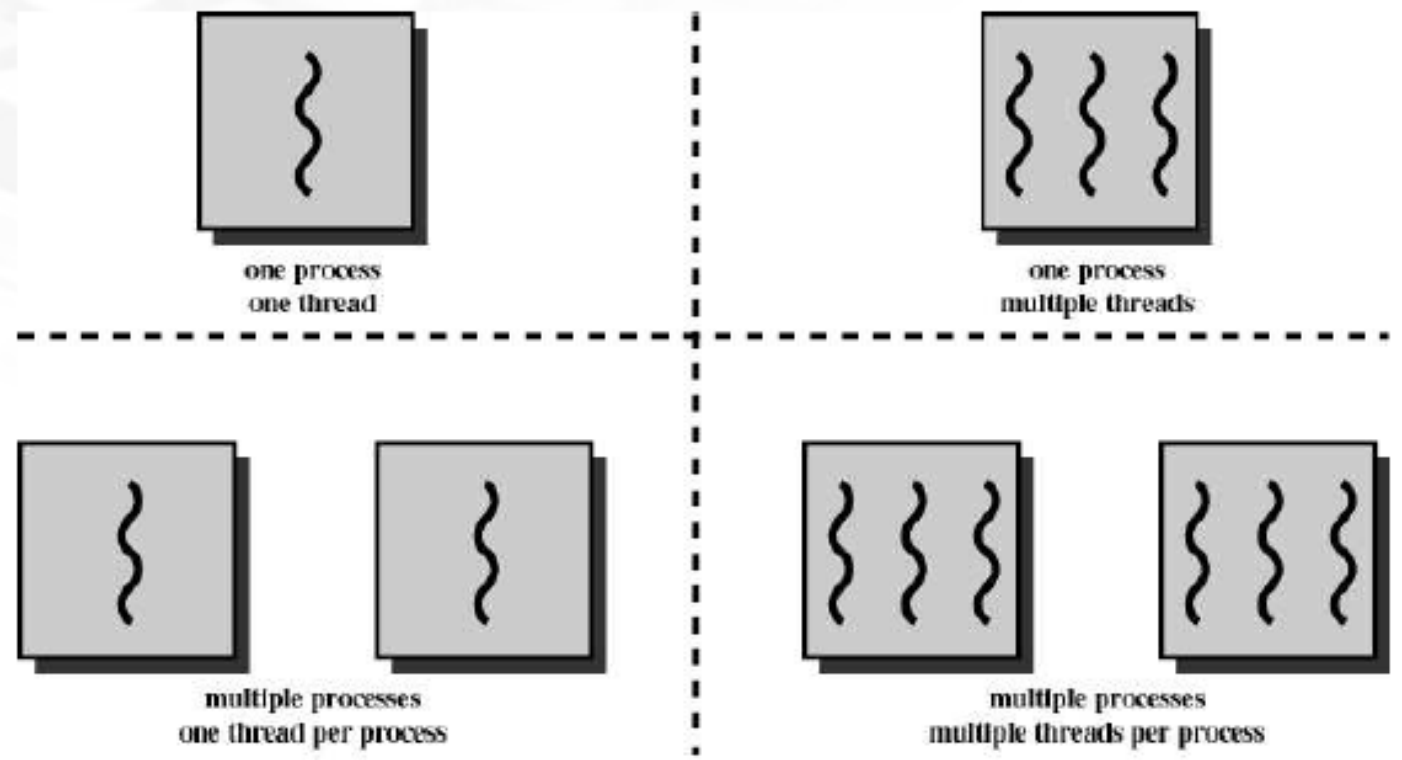


Threads (4)


code
data
files
registers
stack
thread
code
data
files
registers
registers
registers
stack
stack
stack
thread

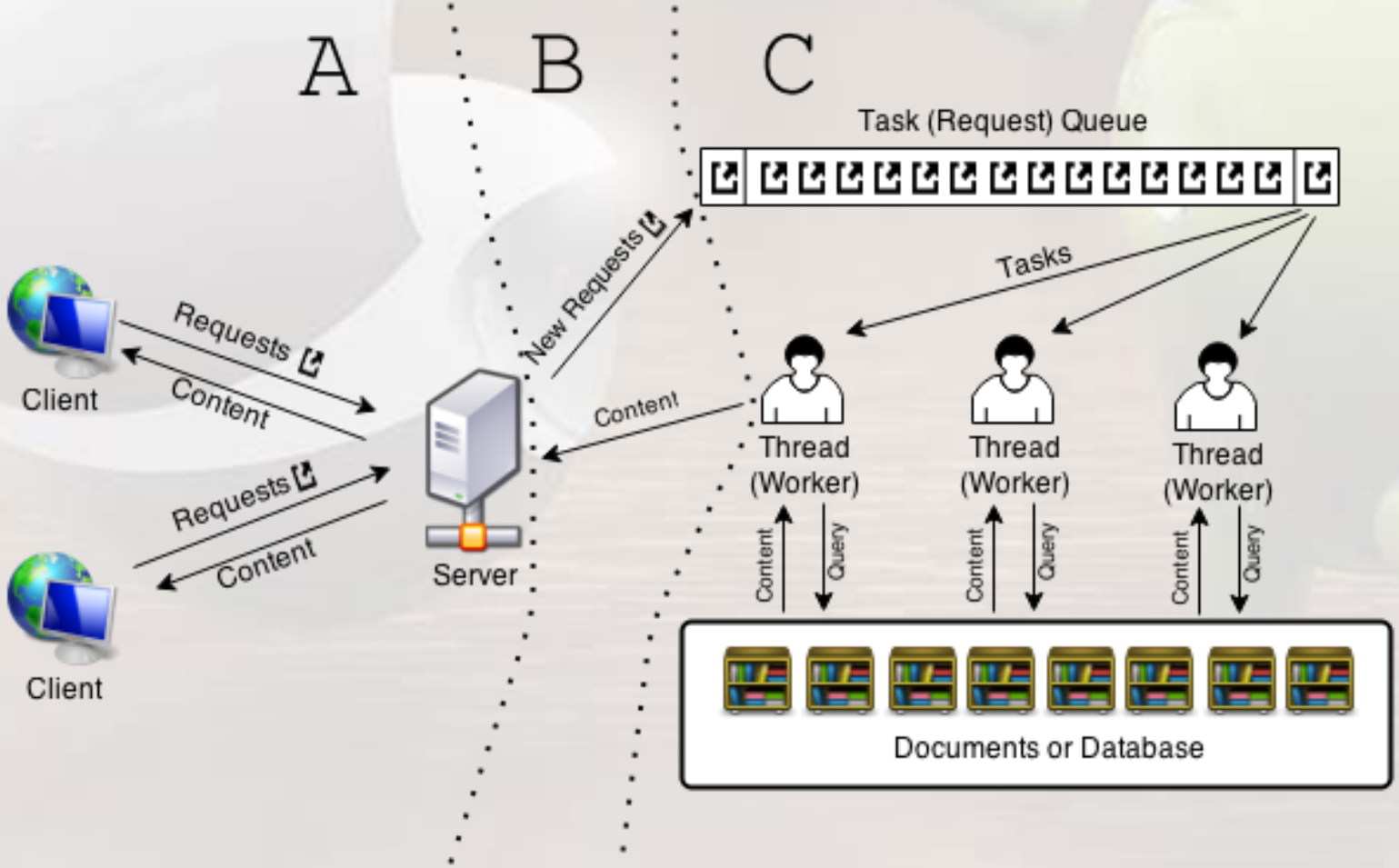
Threads (5)

- Multiprogramação x *multithreading*



Threads (6)

- Exemplo



Threads e Processos (1)

- Existem duas características fundamentais que são usualmente tratadas de forma independente pelo S.O:
  - **Propriedade de recursos** ("*resource ownership*")
    - Trata dos recursos alocados aos processos, e que são necessários para a sua execução.
      - Ex: memória, arquivos, dispositivos de E/S, etc.
  - **Escalonamento** ("*scheduling / dispatching*")
    - 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
- A **troca 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" um ou mais *threads* que estavam bloqueadas.

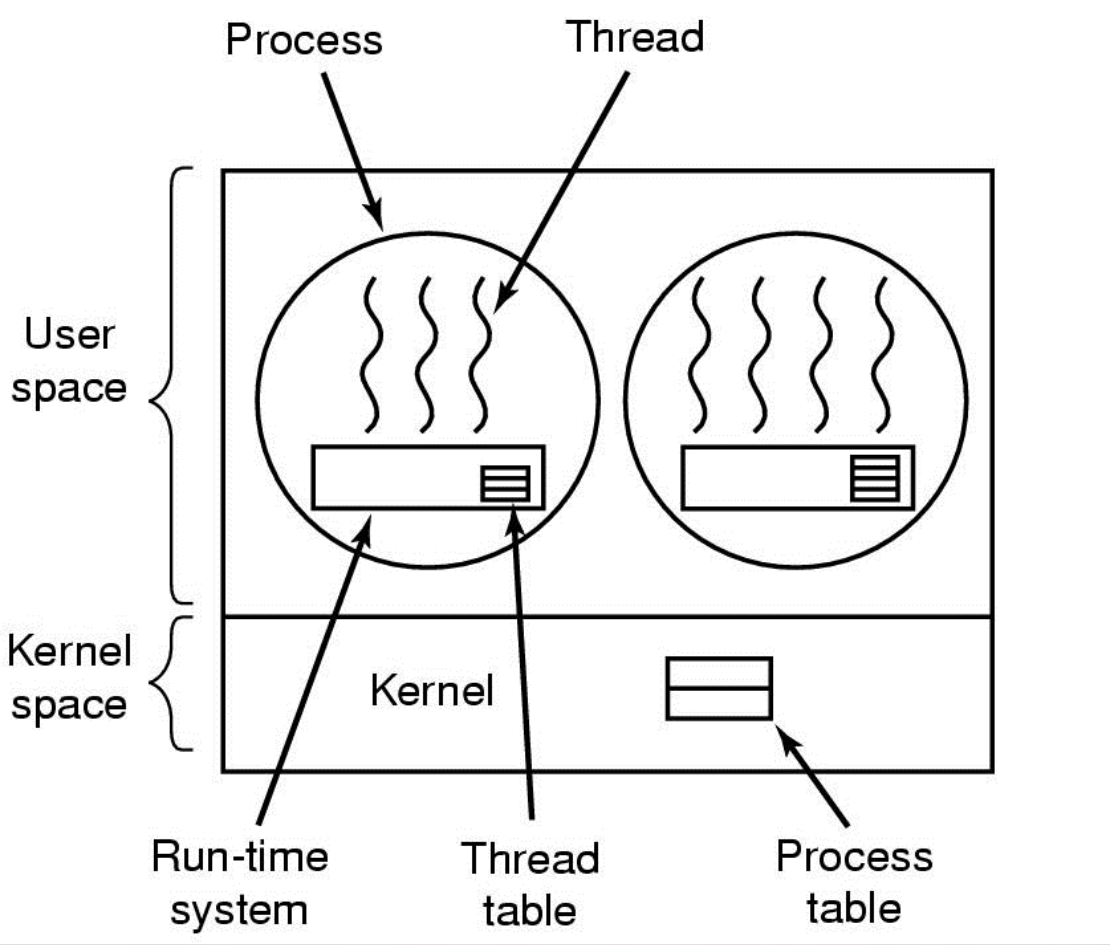
Estados de uma Thread

- Estados fundamentais: executando, pronta e bloqueada.
  - O que acontece com as threads de um processo quando uma delas bloqueia?
  - Suspender um processo implica em suspender todas as *threads* deste processo?
    - Não faz sentido associar o estado "suspenso" com *threads* porque tais estados são conceitos relacionados a processos (swap in/swap out).
  - O término de um processo implica no término de todas as *threads* do processo?

Tipos de Threads

- A implementação de threads pode ser feita de diferentes maneiras, sendo as duas principais:
  - User-level threads (ULT) – nível de usuário
  - Kernel-level threads (KLT) – nível de kernel
- A abstração Lightweight process (LWP), implementada no S.O. Solaris, será discutida adiante.

User-level Threads - ULT (1)



User-level Threads - ULT (2)

- O gerenciamento das threads é feito no espaço de endereçamento de usuário, por meio de uma biblioteca de threads.
  - A biblioteca de threads é um conjunto de funções no nível de aplicação que pode ser compartilhada por todas as aplicações.
- Como o kernel desconhece a existência de threads, o S.O. não precisa oferecer apoio para threads. É, portanto, é mais simples.

User-level Threads - ULT (3)

- A biblioteca de threads pode oferecer vários métodos de escalonamento. Assim, a aplicação pode escolher o melhor algoritmo para ela.
- Exemplos:
  - POSIX Pthreads, Mach C-threads e Solaris threads

Benefícios das ULT

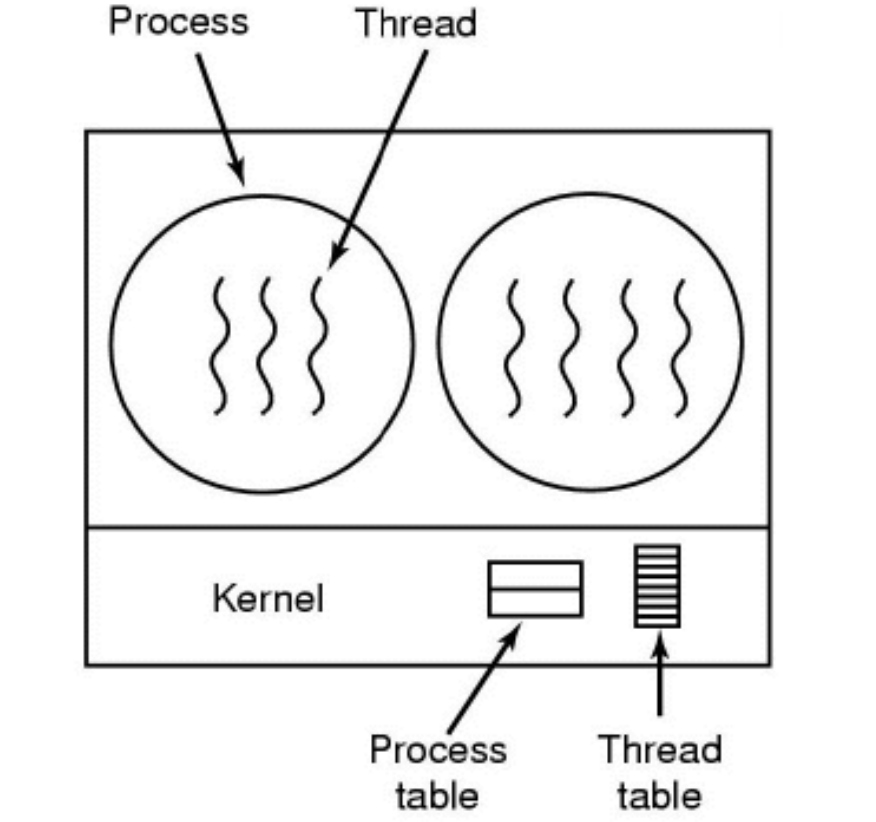
- O chaveamento das threads não requer privilégios de kernel porque todo o gerenciamento das estruturas de dados das threads é feito dentro do espaço de endereçamento de um único processo de usuário.
  - Economia de duas trocas de contexto: user-to- kernel e kernel-touser.
- O escalonamento pode ser específico da aplicação.
  - Uma aplicação pode se beneficiar mais de um escalonador Round Robin, enquanto outra de um escalonador baseado em prioridades.
- ULTs podem executar em qualquer S.O. As bibliotecas de código são portáveis.

Desvantagens das ULT

- Muitas das chamadas ao sistema são bloqueantes e o kernel bloqueia processos – neste caso todos as threads do processo podem ser bloqueados quando uma ULT executa uma SVC .

- Num esquema ULT puro, uma aplicação multithreading não pode tirar vantagem do multiprocessamento.
  - O kernel vai atribuir o processo a apenas um CPU; portanto, duas threads dentro do mesmo processo não podem executar simultaneamente numa arquitetura com múltiplos processadores.

Kernel-level Threads - KLT (1)



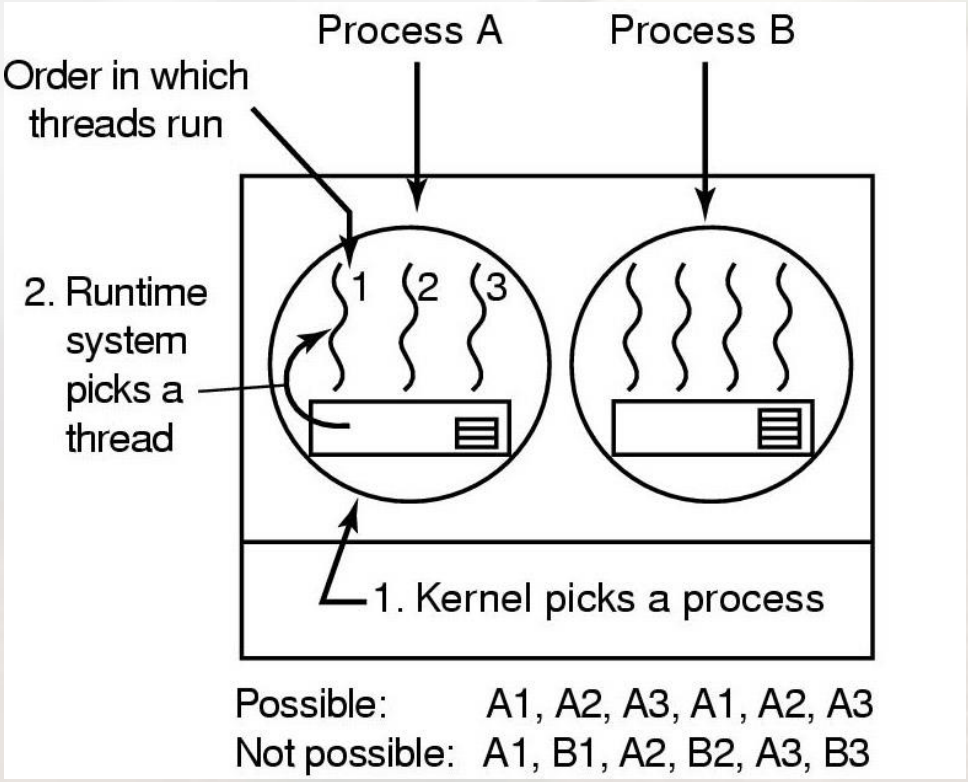
Kernel-level Threads – KLT (2)

- O gerenciamento das threads é feito pelo kernel.
  - O kernel pode melhor aproveitar a capacidade de multiprocessamento da máquina, escalonando as várias threads do processo em diferentes processadores.
- O chaveamento das threads é feito pelo núcleo e o escalonamento é "thread-basis".
  - O bloqueio de uma thread não implica no bloqueio das outras threads do processo.
- O kernel mantém a informação de contexto para processo e threads.

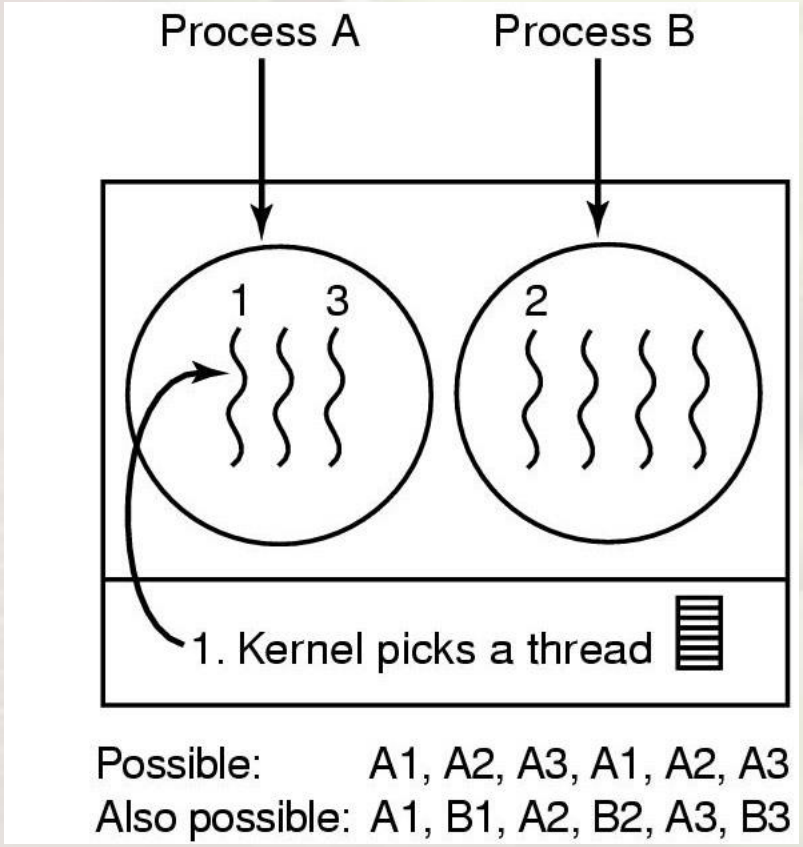
Kernel-level Threads – KLT (3)

- O usuário enxerga uma API para threads do núcleo; porém, a transferência de controle entre threads de um mesmo processo requer chaveamento para modo kernel.
  - Ações do kernel geralmente tem um custo que pode ser significativo.
- Windows 2K, Linux, e OS/2 são exemplos desta abordagem.

ULT x KLT

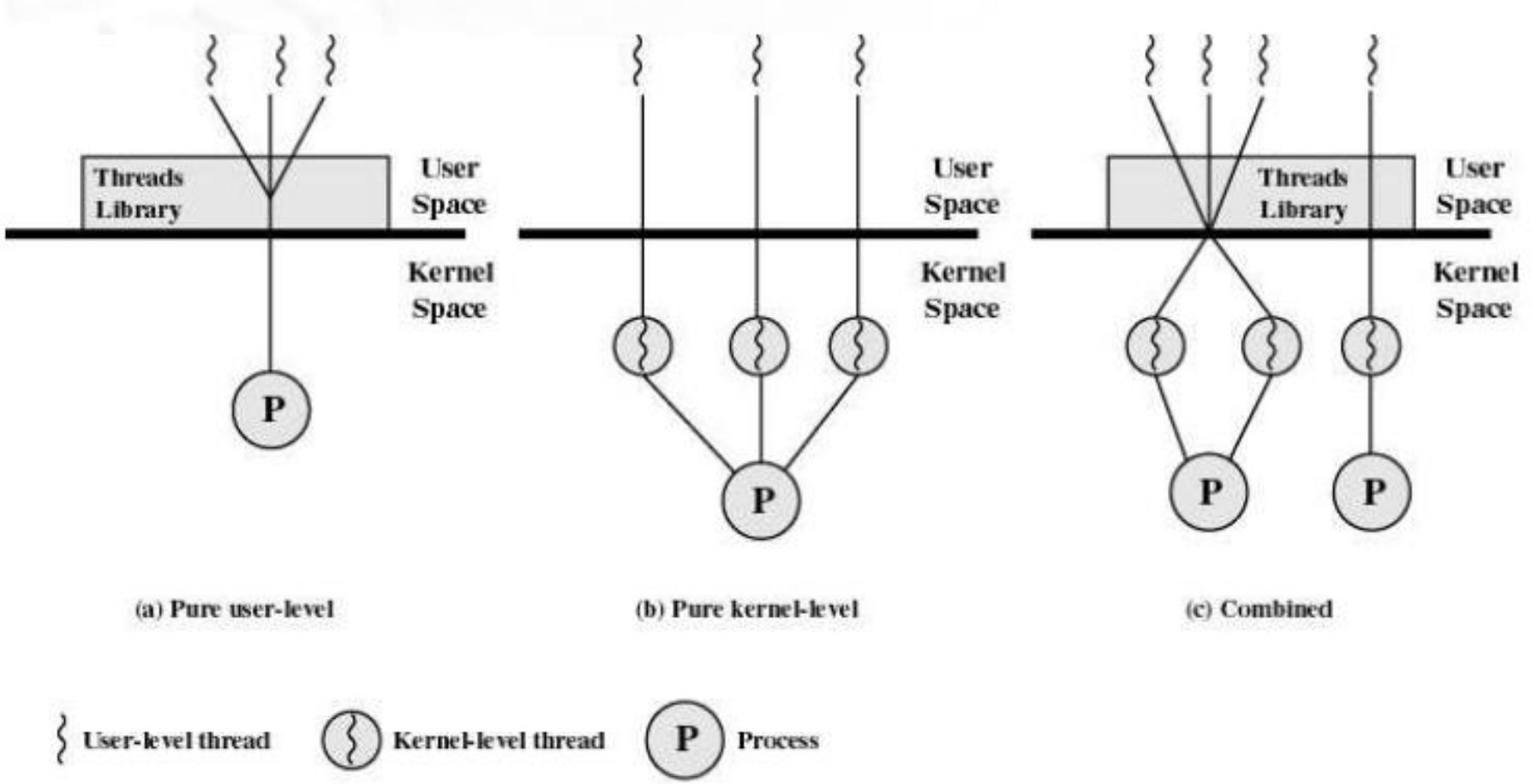


Threads em modo usuário



Threads em modo *kernel*

Combinando Modos



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.).

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 lpthread

$ gcc -o simple -lpthread simple_threads.c

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

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()`

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

Criação de Threads: pthread_create() (2)

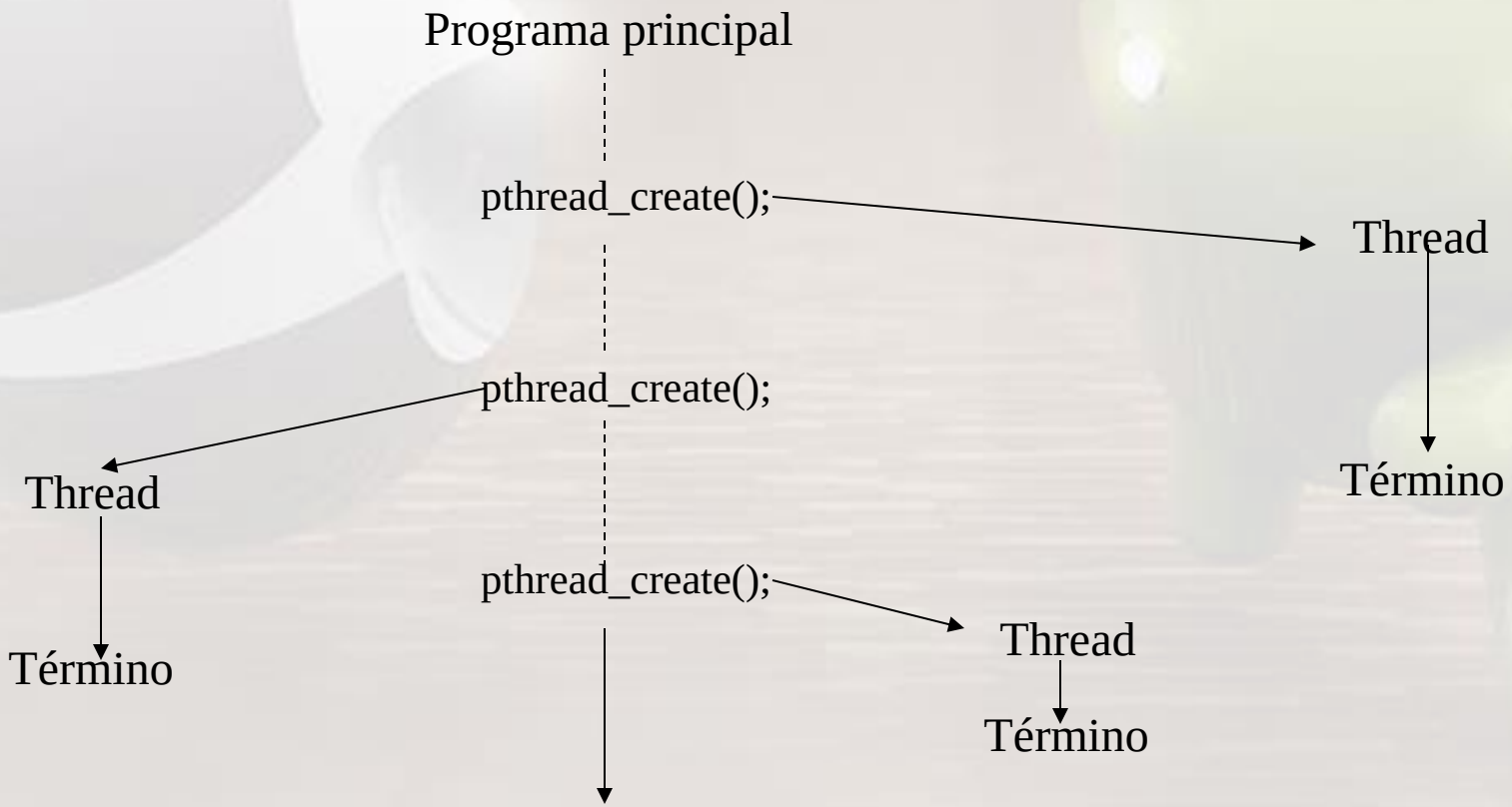
- Quando se cria uma nova thread é possível especificar uma série de atributos e propriedades através de uma variável do tipo `pthread_attr_t`.

- Os atributos que afetam o comportamento da thread são definidos pelo parâmetro `attr`. Caso o valor de `attr` seja NULL, o comportamento padrão é assumido para a thread :
  - (i) unbound; (ii) nondettached; (iii) pilha e tamanho de pilha padrão; (iv) prioridade da thread criadora.

- Os atributos podem ser modificados antes de serem usados para se criar uma nova thread. Em especial, a política de escalonamento, o escopo de contenção, o tamanho da pilha e o endereço da pilha podem ser modificados usando as funções `attr_setxxxx()`.

Detached Threads

- Pode ser que uma thread não precise saber do término de uma outra por ela criada. Neste caso, diz-se que a thread criada é *detached* (desunida) da thread mãe.



Atributos de Threads: pthread_attr_init() (1)

- Para se alterar os atributos de uma *thread*, a variável de atributo terá de ser previamente inicializada com o serviço `pthread_attr_init()` e depois modificada através da chamada de serviços específicos para cada atributo usando as funções `attr_setxxxx()`.
- Por exemplo, para criar um thread já no estado de detached:

```
…
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, …, …);
…
pthread_attr_destroy(&attr);
...
```

Atributos de Threads: pthread_attr_init() (2)

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_setstacksize(pthread_attr_t *attr, int size);
int pthread_attr_getstacksize(pthread_attr_t *attr, int *size);
int pthread_attr_setstackaddr(pthread_attr_t *attr, int addr);
int pthread_attr_getstackaddr(pthread_attr_t *attr, int *addr);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int state);
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *state);
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(pthread_attr_t *attr, int *scope);
int pthread_attr_setinheritsched(pthread_attr_t *attr, int sched);
int pthread_attr_getinheritsched(pthread_attr_t *attr, int *sched);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy);
int pthread_attr_setschedparam(pthread_attr_t *attr,
struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr,struct sched_param
*param);
```

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, *detached* 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.

```
pthread_t pthread_self(void);
```

Exemplo 1

- Criando uma thread. Main espera thread terminar para continuar
  - thread_1.c
  - gcc thread_1.c –o thread_1 -lpthread

Exemplo 2

- Printando na thread e na main. Main não espera terminar
  - thread_2.c

Exemplo 3

- Criando múltiplas threads
  - thread_3.c

Exemplo 4

- Criando múltiplas threads alterando atributos e juntando (JOINABLE)
  - thread_4.c

Linux Threads

- No Linux as *threads* são referenciadas como *tasks* (tarefas).
- Implementa o modelo de mapeamento um-para-um.
- A criação de *threads* é feita através da SVC (chamada ao sistema) `clone()`.
- `clone()` permite à tarefa filha compartilhar o mesmo espaço de endereçamento que a tarefa pai (processo).
  - Na verdade, é criado um novo processo, mas não é feita uma cópia, como no `fork()`;
  - O novo processo aponta p/ as estruturas de dados do pai

Referências

- Slides adaptados de Roberta Lima Gomes (UFES)

- Bibliografia
  - Vahalia, U. *Unix Internals: the new frontiers*. Prentice-Hall, 1996.
    - Capítulo 4 (até seção 4.7)
  - A. S. Tanenbaum, *Sistemas Operacionais Modernos*, 3a. Edição, Editora Prentice-Hall, 2010.
    - Seção 2.2
  - Silberschatz A. G.; Galvin P. B.; Gagne G.; *Fundamentos de Sistemas Operacionais*, LTC, 2004.
    - Capítulo 5