



Laboratório de Pesquisa em Redes e Multimídia

# Threads



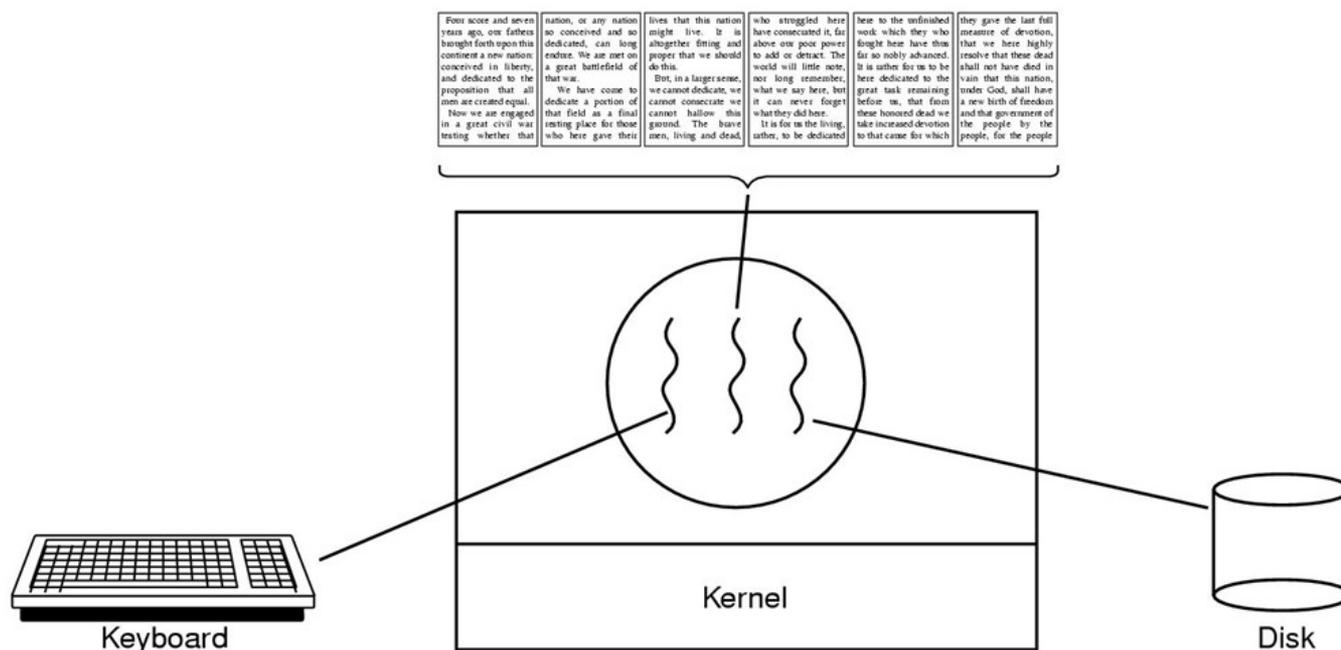
Universidade Federal do Espírito Santo  
Departamento de Informática

## 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.

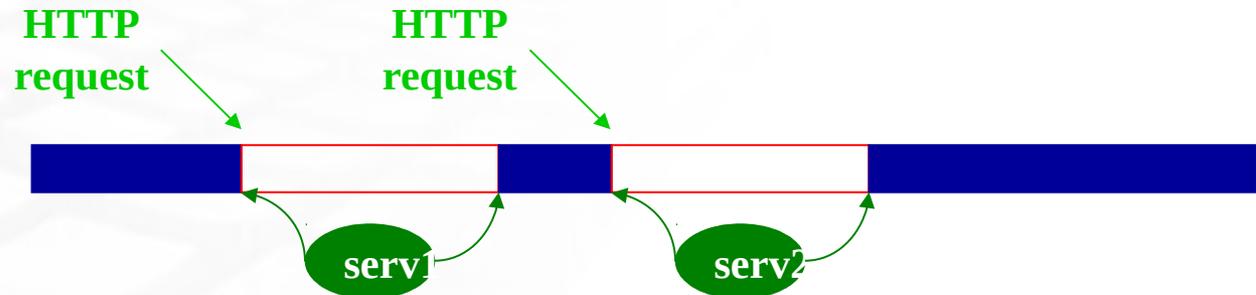
$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} a.e + b.g & a.f + b.h \\ c.e + d.g & c.f + d.h \end{pmatrix}$$

# Exemplos de Programas MT (3)

tempo →

**Mono**

processo 1



(processo 1)

thread A



HTTP request

thread B



HTTP request

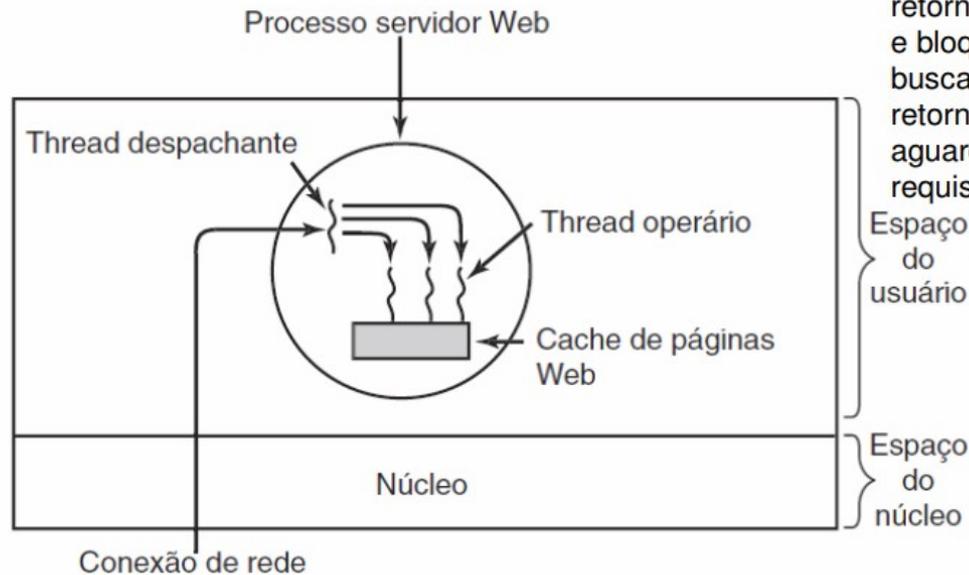
**Multi**

	executando
	bloqueado(a) – HTTP req.
	espera - processador

# Exemplos de Programas MT (4)

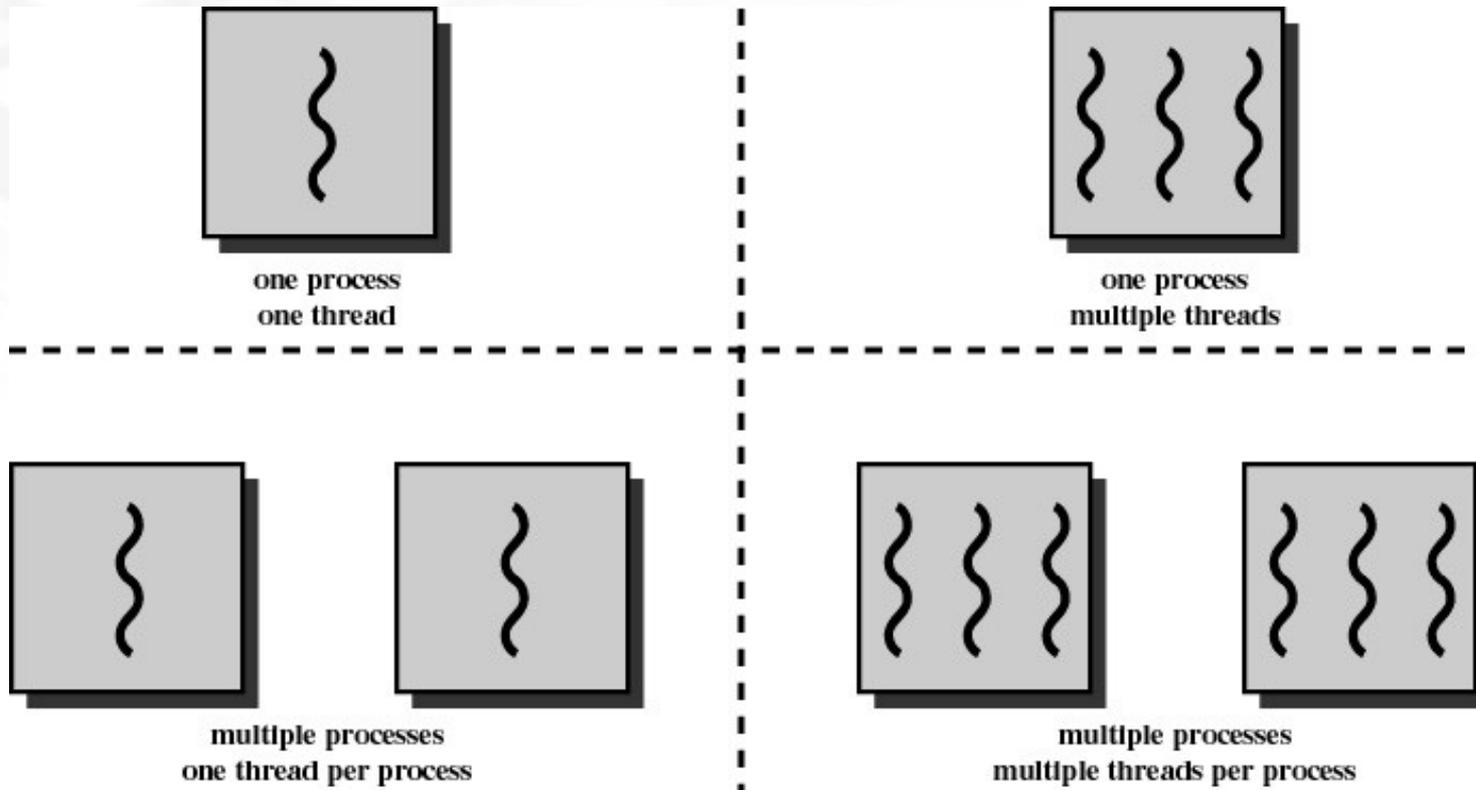
## ▪ Servidor Web

Executa em um laço infinito, recebendo requisições de trabalho e entregando a um operário



O operário verifica se a página está disponível em cache, se sim retorna ao cliente e bloqueia, se não, busca no disco, retorna e bloqueia aguardando nova requisição

## ■ Multiprogramação x multithreading



# Threads (1)

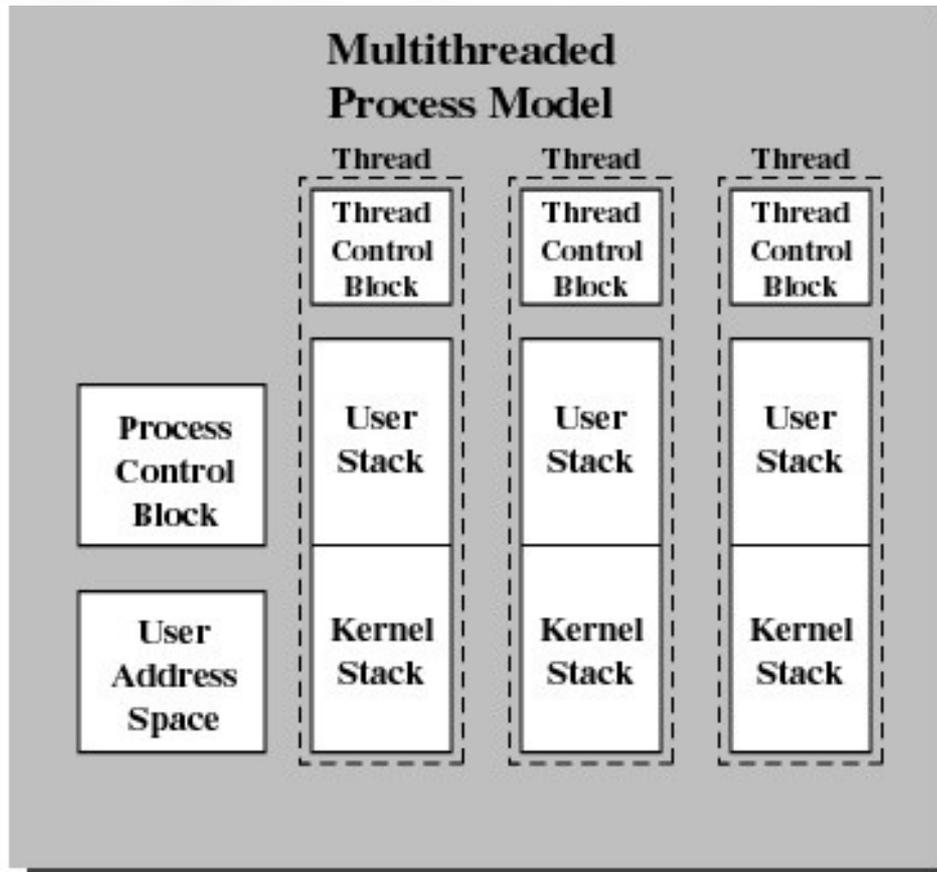
- Thread:
  - Thread = “fluxo”, “fio”.
  - Fluxo de execução dentro de um processo (seqüência de instruções a serem executadas dentro de um programa).
- Thread é uma abstração que permite que uma aplicação execute mais de um trecho de código simultaneamente. (ex: um método).
  - Processos permitem ao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

- Estrutura de um processo com multithreading



## 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)

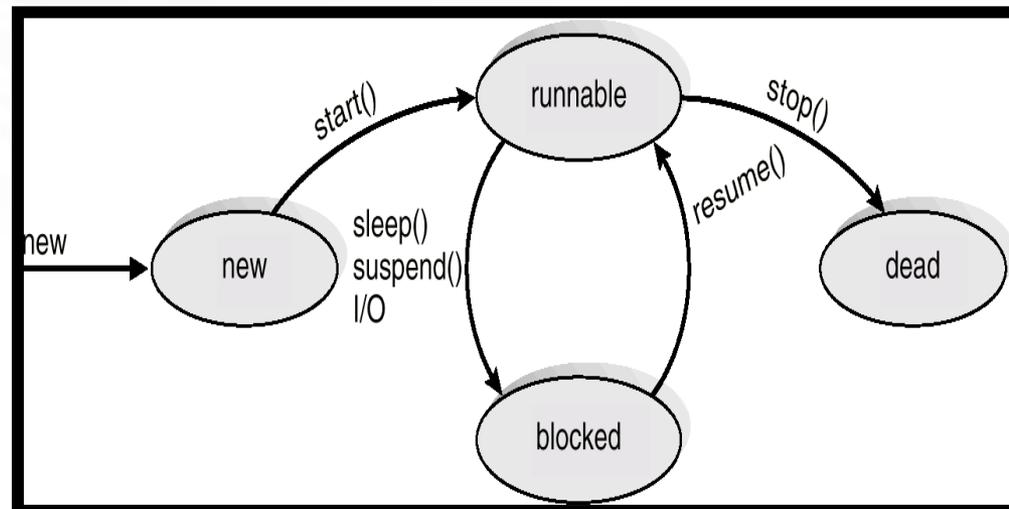
- Tradicionalmente o processo está associado a:
  - um programa em execução
  - um conjunto de recursos
- Em um S.O. que suporta múltiplas threads:
  - **Processos** estão associados somente à **propriedade de recursos**
  - **Threads** estão associadas às atividades de execução (ou seja, *threads* constituem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

# 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).

## Estados de uma Thread (1)

- Estados fundamentais: executando, pronta e bloqueada.
- Não faz sentido associar o estado “suspenso” com *threads* porque tais estados são conceitos relacionados a processos (swap in/swap out).



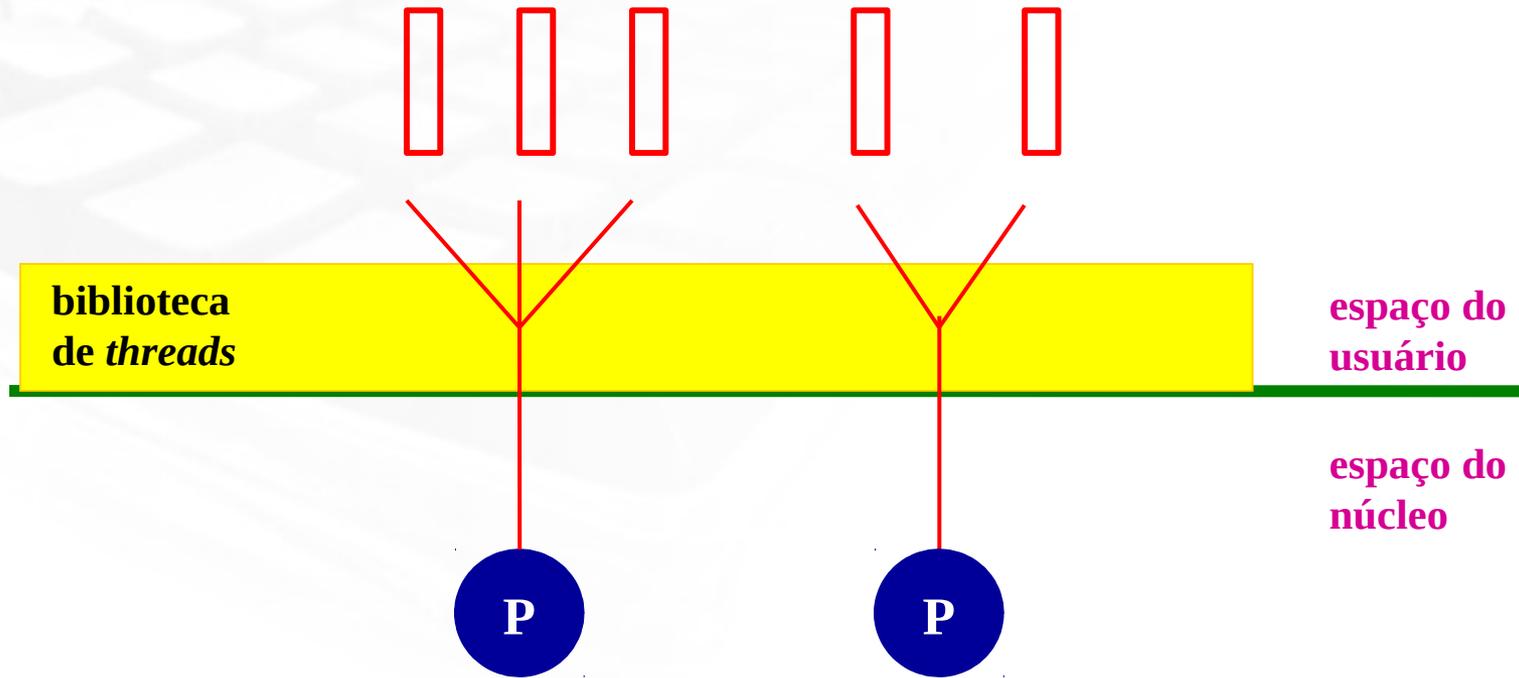
## Estados de uma Thread (2)

- O que acontece com as *threads* de um processo quando uma delas bloqueia?
- Suspende um processo implica em suspender todas as *threads* deste processo?
- O término de um processo implica no término de todas as *threads* do processo?

## Tipos de Threads

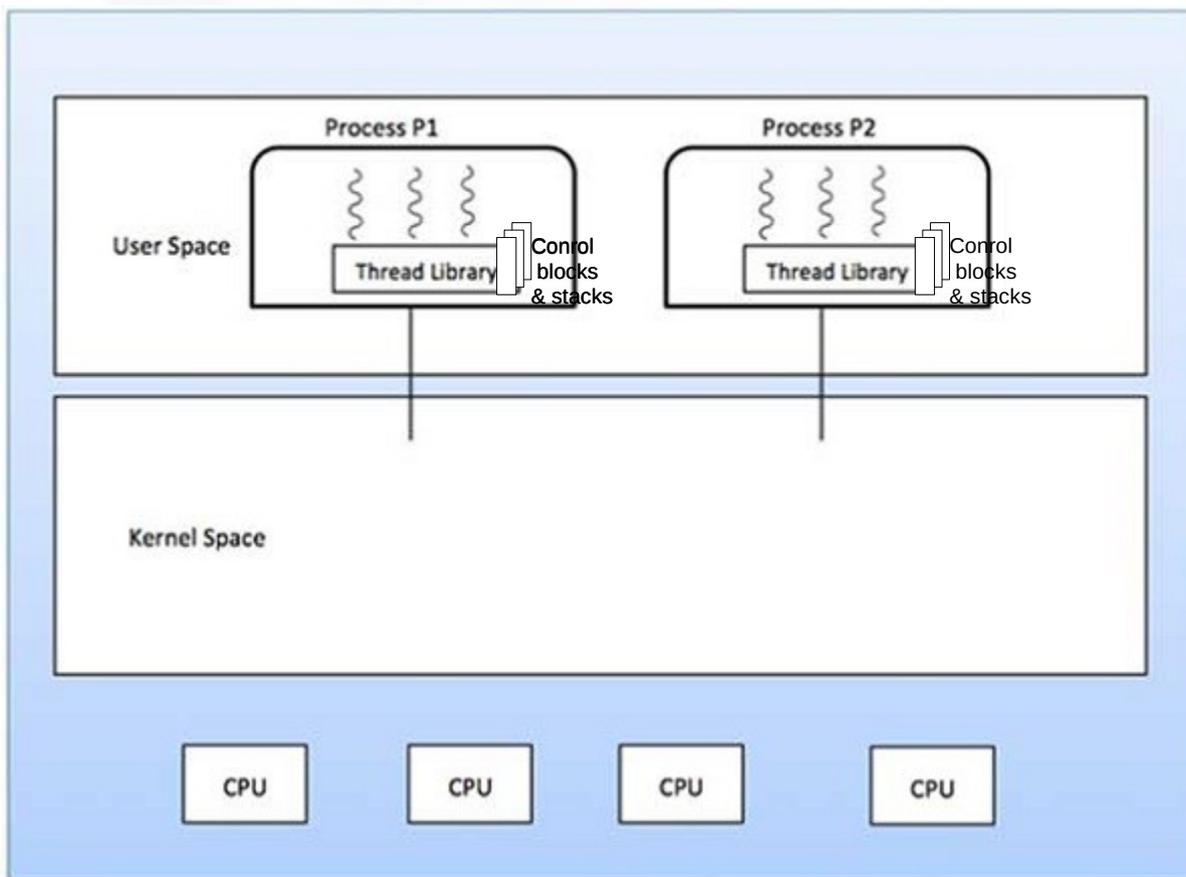
- A implementação de *threads* pode ser feita de diferentes maneiras, sendo as duas principais:
  - **User-level threads (ULT)** – nível de usuário
  - **Kernel-level threads (KLT)** – nível de *kernel*
- A abstração *Lightweight process (LWP)*, implementada no S.O. Solaris, será discutida adiante.

# User-level Threads - ULT (1)



	<i>thread</i> nível usuário		Processo
---	--------------------------------	---	----------

## User-level Threads - ULT (2)



## User-level Threads - ULT (3)

- 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 (4)

- 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*.

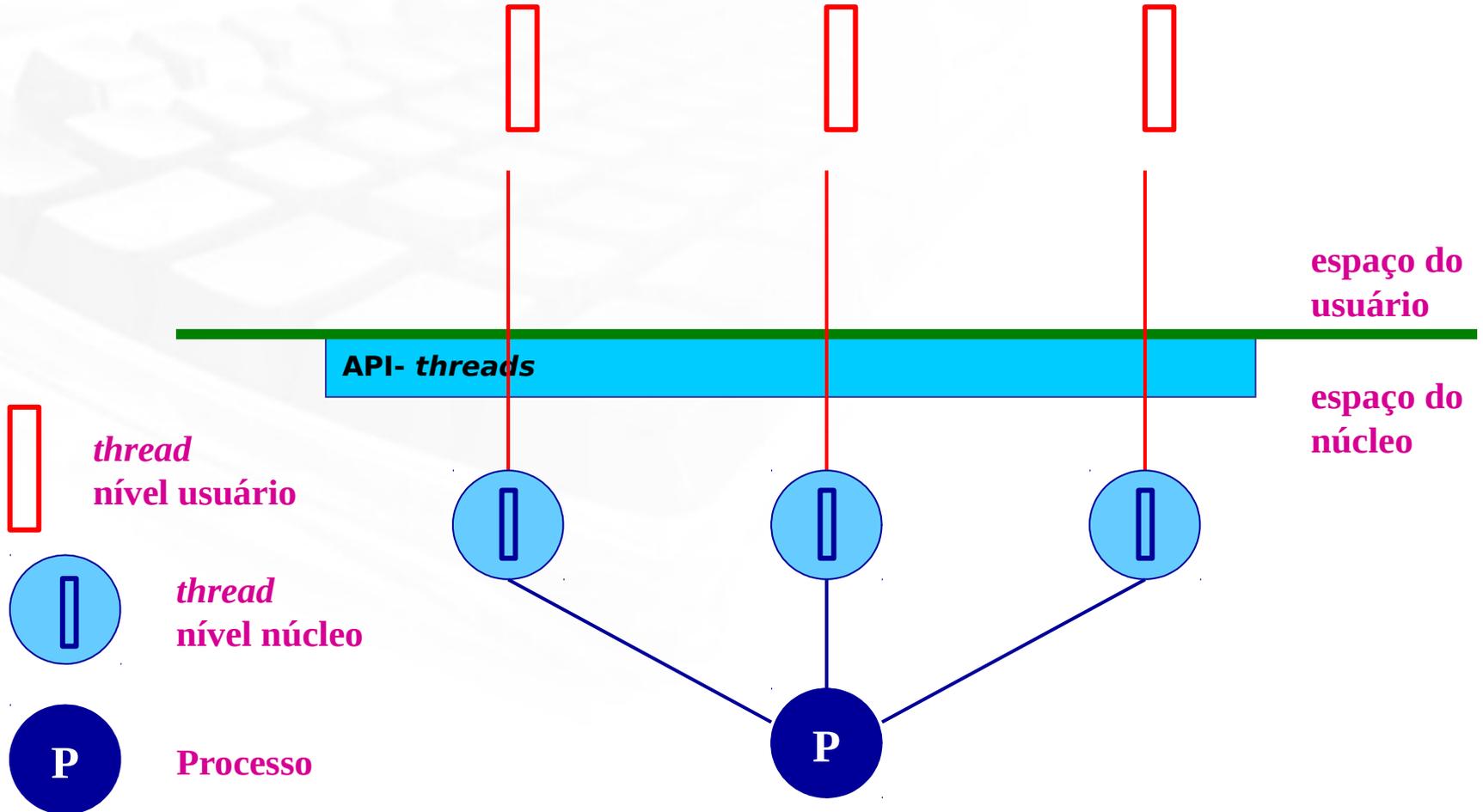
## Resumo: Benefícios das ULT

- 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-to-user.
- 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.

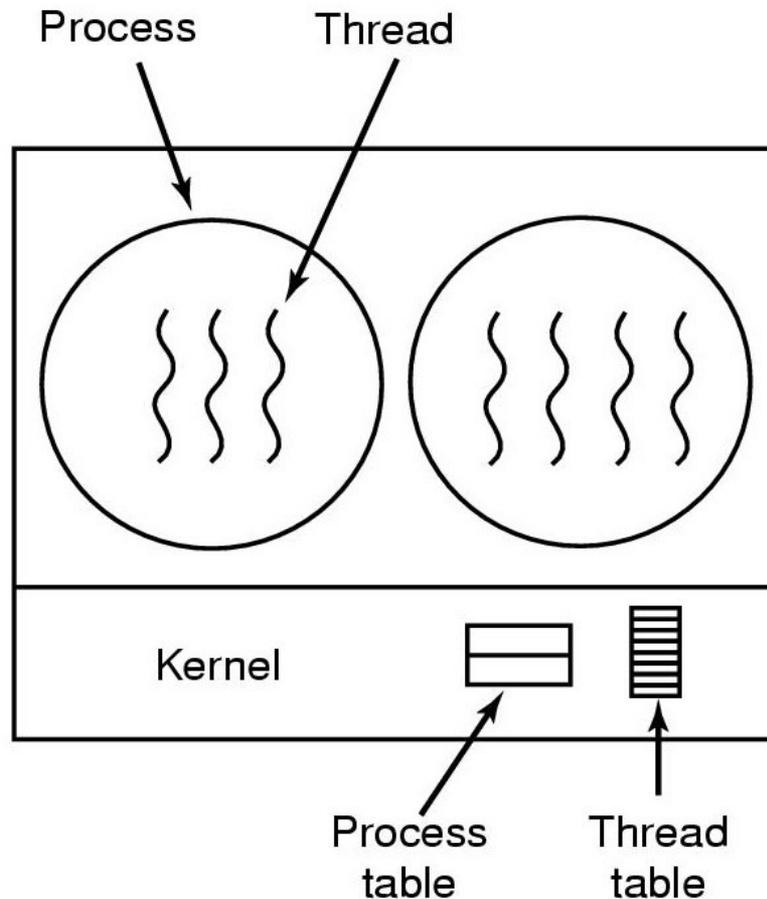
## E as desvantagens das ULT...

- 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 arquitectura com múltiplos processadores.

# Kernel-level Threads - KLT (1)



## Kernel-level Threads - KLT (2)



## Kernel-level Threads – KLT (3)

- 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*.

## S.O. Multithreading (KLT)

- 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 , Android, 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.

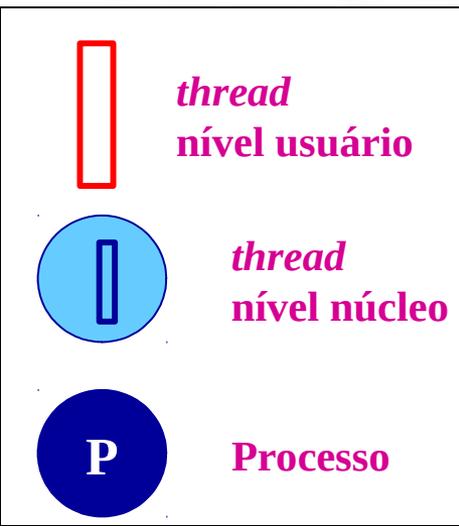
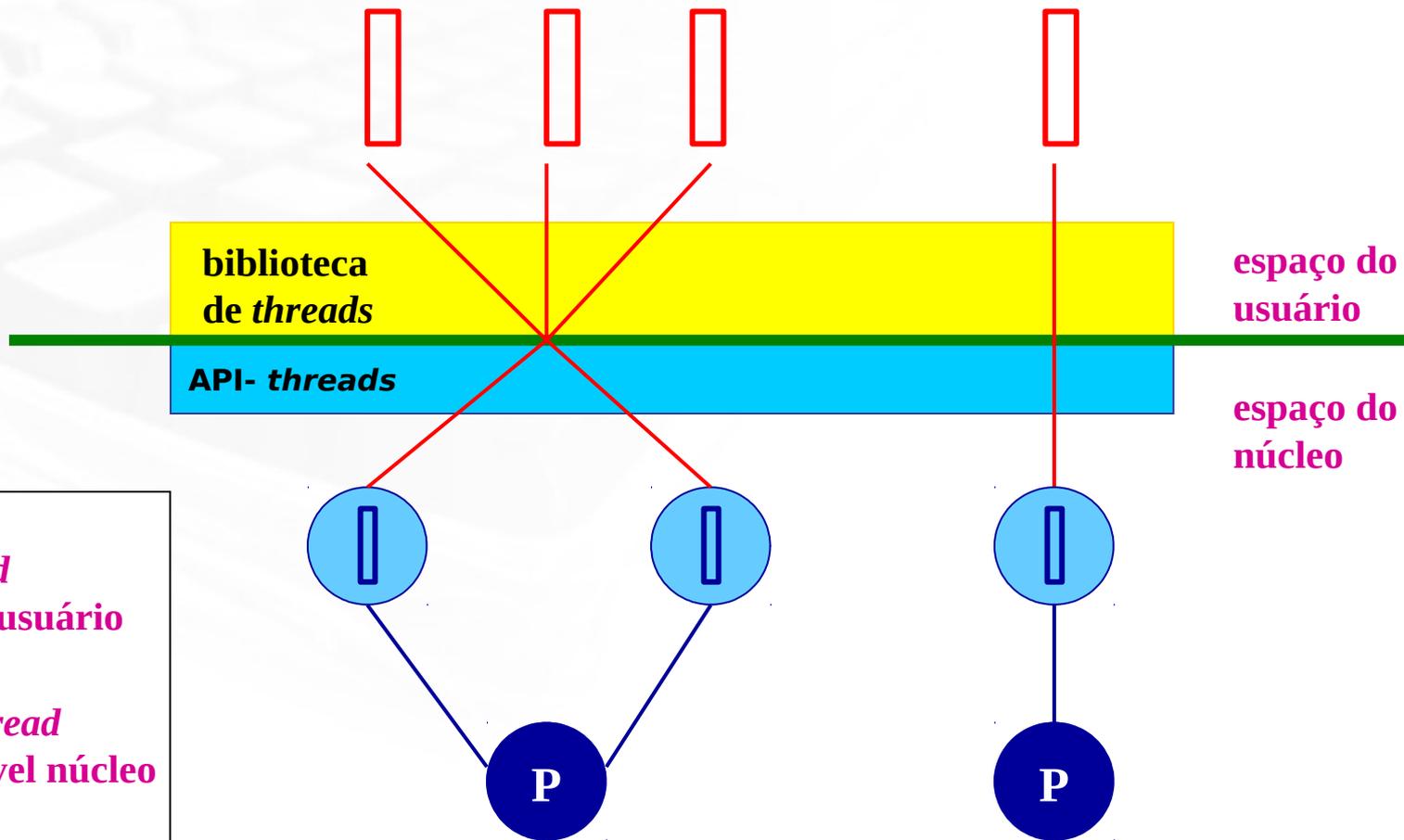
## Vantagens das KLTs sobre Processos

- 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 geralmente são disponibilizadas
  - Estas primitivas permitem “acordar” um ou mais *threads* que estavam bloqueadas.

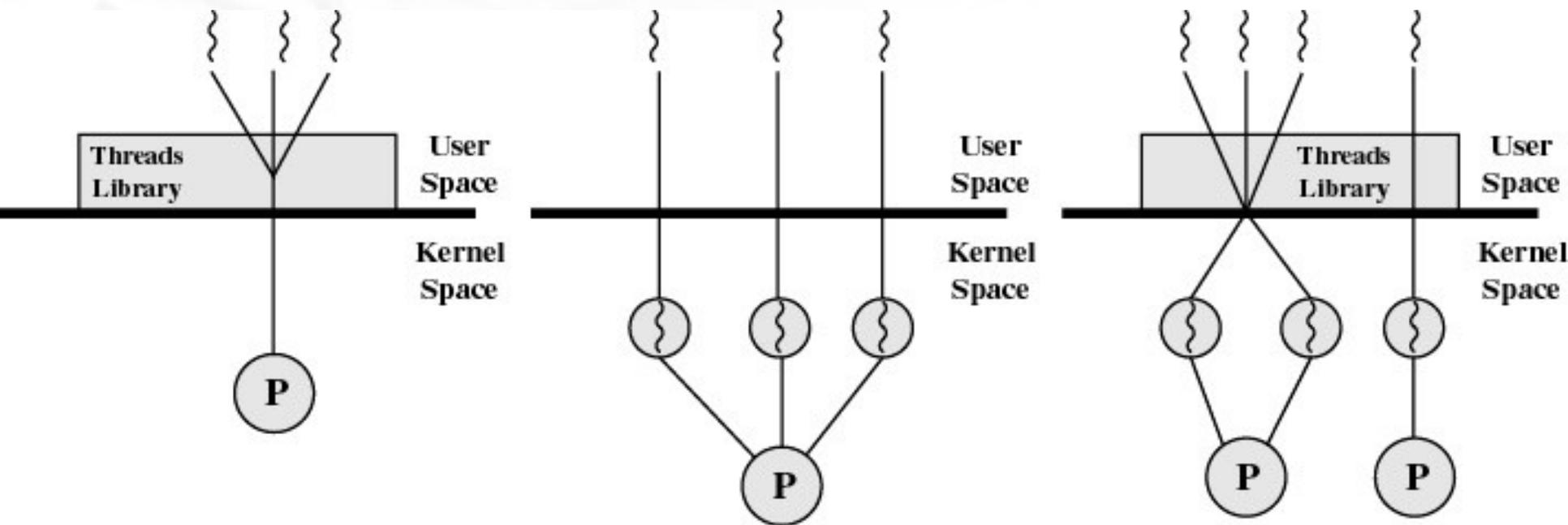
## Kernel-level Threads – KLT (4)

- 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.

# Combinando Modos



# Resumindo ...



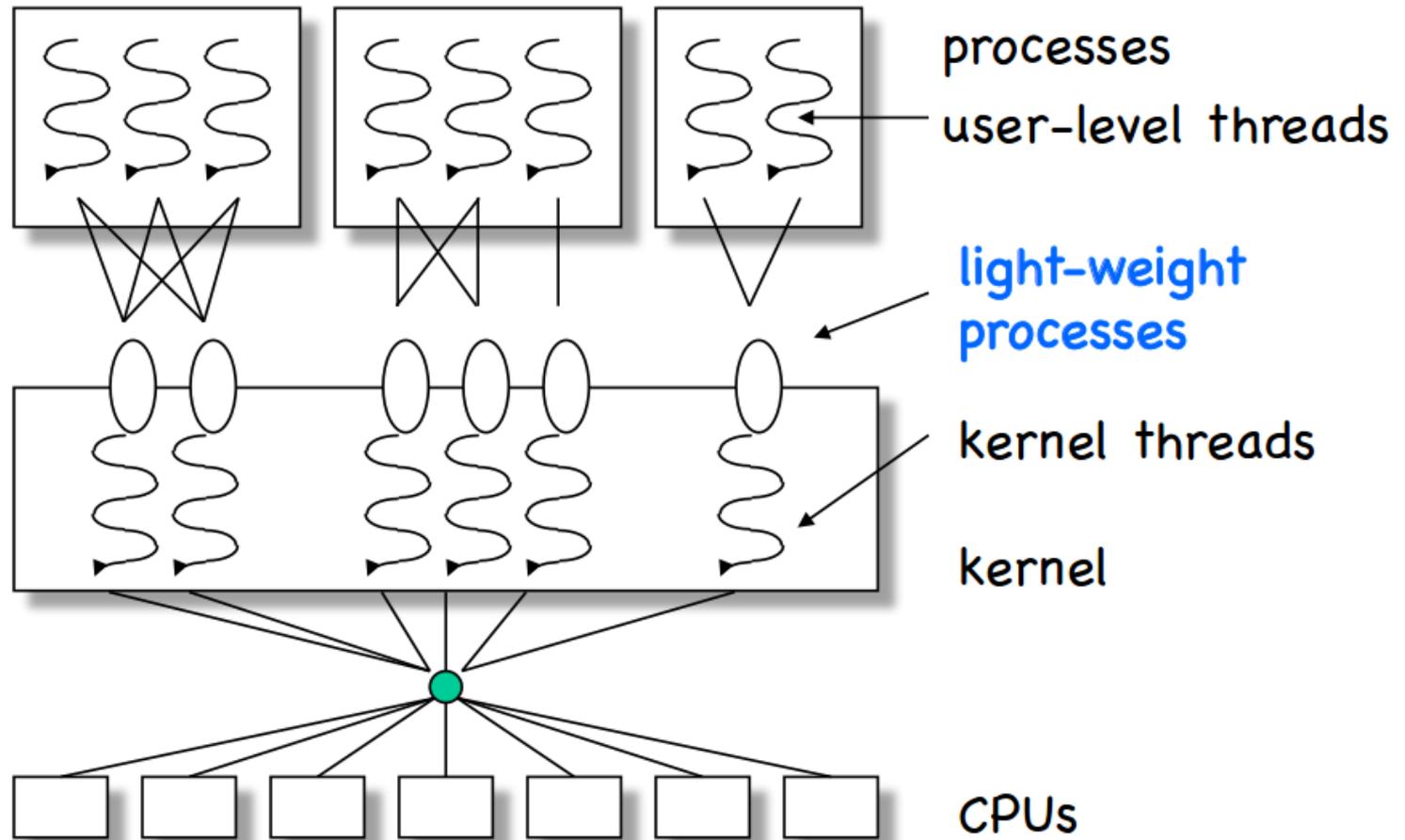
(a) Pure user-level

(b) Pure kernel-level

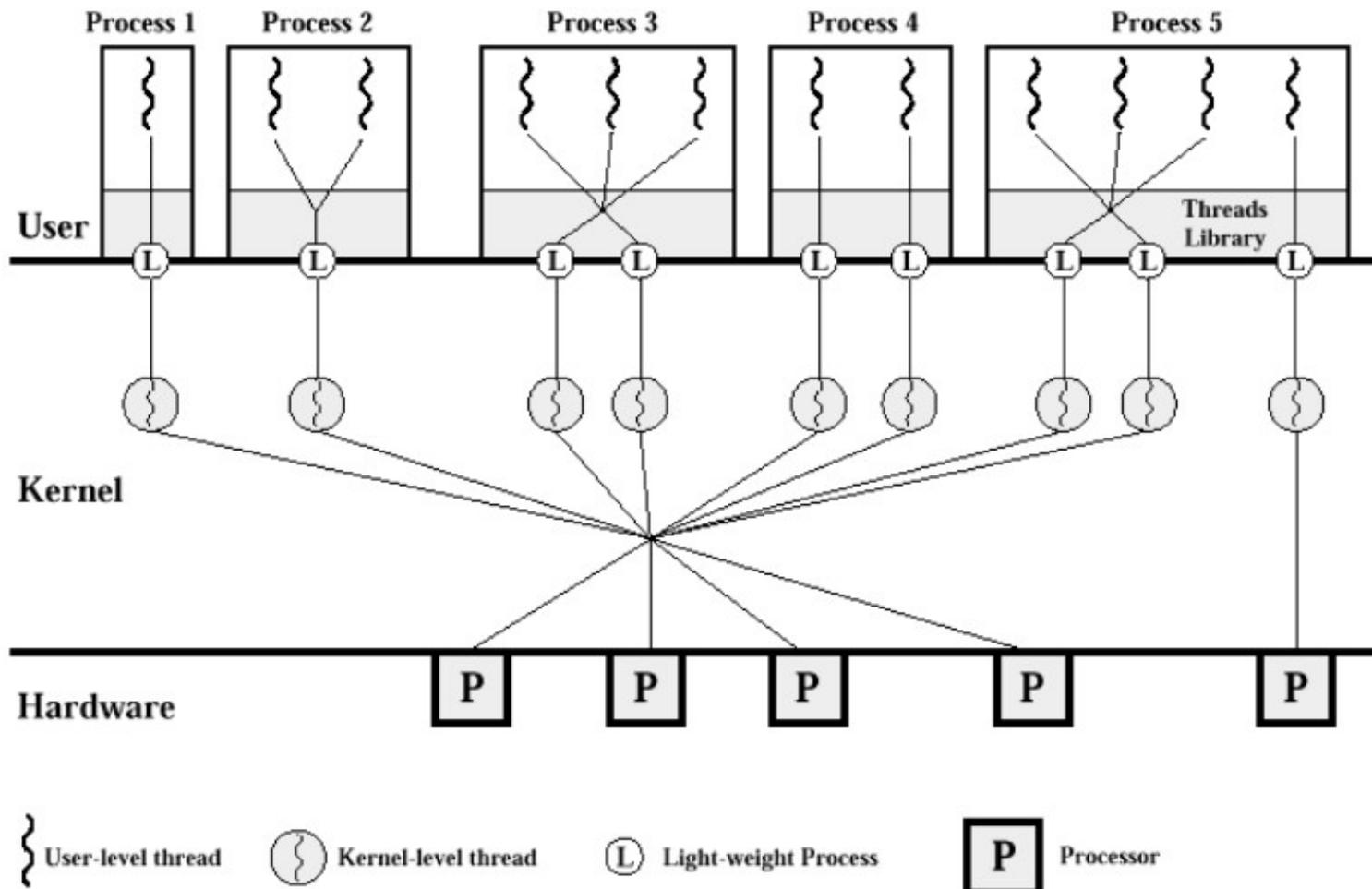
(c) Combined



# Light-weight process



# Modelo de Multithreading do S.O. Solaris (1)



# Modelo de Multithreading do S.O. Solaris

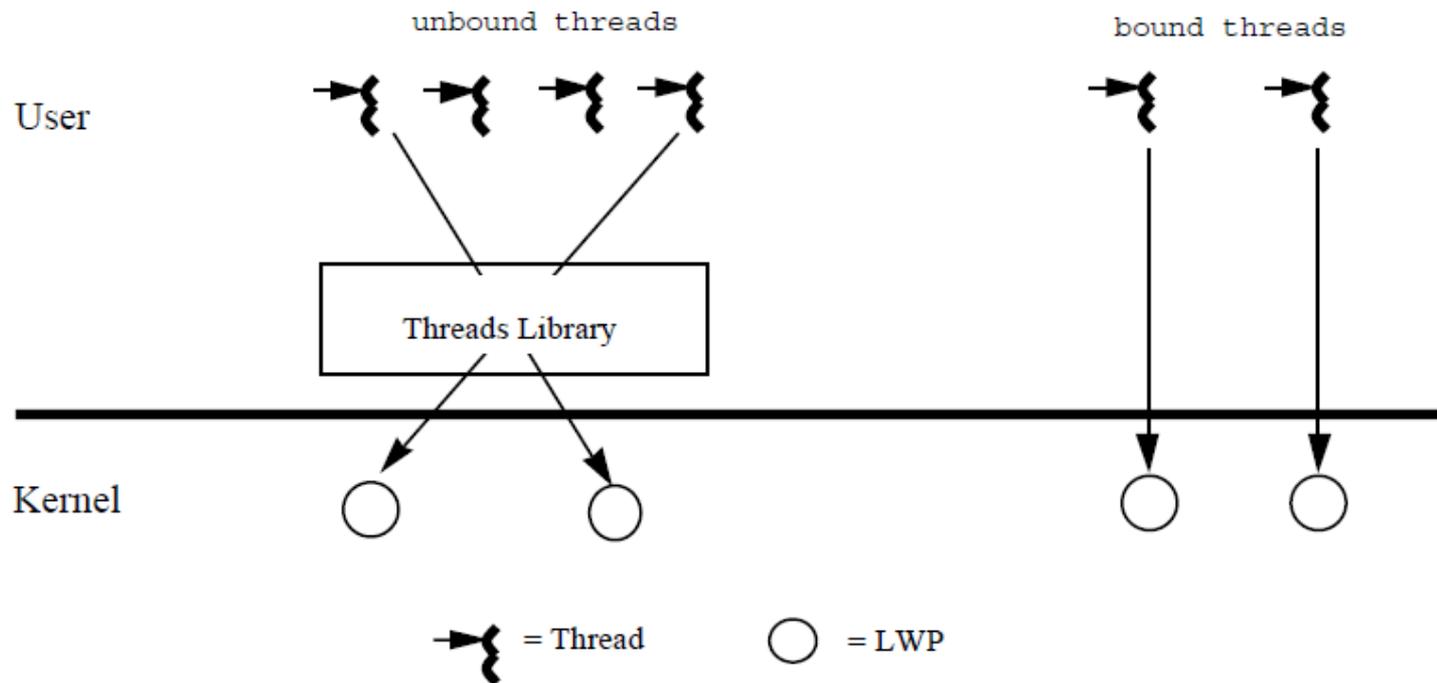
(2)

- LWP (*Lightweight Processes*) são ULT com suporte do *kernel*, isto é, requerem suporte de KLT para serem implementadas.
- LWP constituem uma abstração de alto nível baseadas em KLT.
  - Assim como rotinas da biblioteca *stdio* (ex: `fopen()` e `fread()`), usam as funções `open()` e `read()` do *kernel*, ULT podem usar uma abstração de alto nível (as LWP) como interface de acesso às *threads* de *kernel*.
- LWP são recursos do *kernel*, executam código de *kernel* e SVCs. Eles formam uma **ponte entre os níveis de usuário e o de *kernel***.

# Modelo de Multithreading do S.O. Solaris

(3)

- *Bound threads* são threads que estão permanentemente conectadas a um LWP.
- *Unbound threads* são threads cujas trocas de contexto são feitas de maneira muito rápida, sem o suporte de kernel. É a *thread* default no Solaris.



## Bibliotecas de Threads <sup>(1)</sup>

- A interface para suporte à programação *multithreading* é feita via bibliotecas:
  - *libpthread* (padrão POSIX/IEEE 1003.1c)
  - *libthread* (Solaris).
- POSIX Threads ou *pthread* 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 *pthread.h*
  - “Linkar” a biblioteca *pthread*

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

# Biblioteca Pthreads – Algumas Operações

<b>POSIX function</b>	<b>Description</b>
<code>pthread_cancel</code>	terminate another thread
<code>pthread_create</code>	create a thread
<code>pthread_detach</code>	set thread to release resources
<code>pthread_equal</code>	test two thread IDs for equality
<code>pthread_exit</code>	exit a thread without exiting process
<code>pthread_kill</code>	send a signal to a thread
<code>pthread_join</code>	wait for a thread
<code>pthread_self</code>	find out own thread ID

# Thread APIs vs. System calls para Processos

<i>Pthread API</i>	<i>system calls for process</i>
<code>Pthread_create()</code>	<code>fork()</code> , <code>exec*()</code>
<code>Pthread_exit()</code>	<code>exit()</code> , <code>_exit()</code>
<code>Pthread_self()</code>	<code>getpid()</code>
<code>sched_yield()</code>	<code>sleep()</code>
<code>pthread_kill()</code>	<code>kill()</code>
<code>Pthread_cancel()</code>	
<code>Pthread_sigmask()</code>	<code>sigmask()</code>

## 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);
```

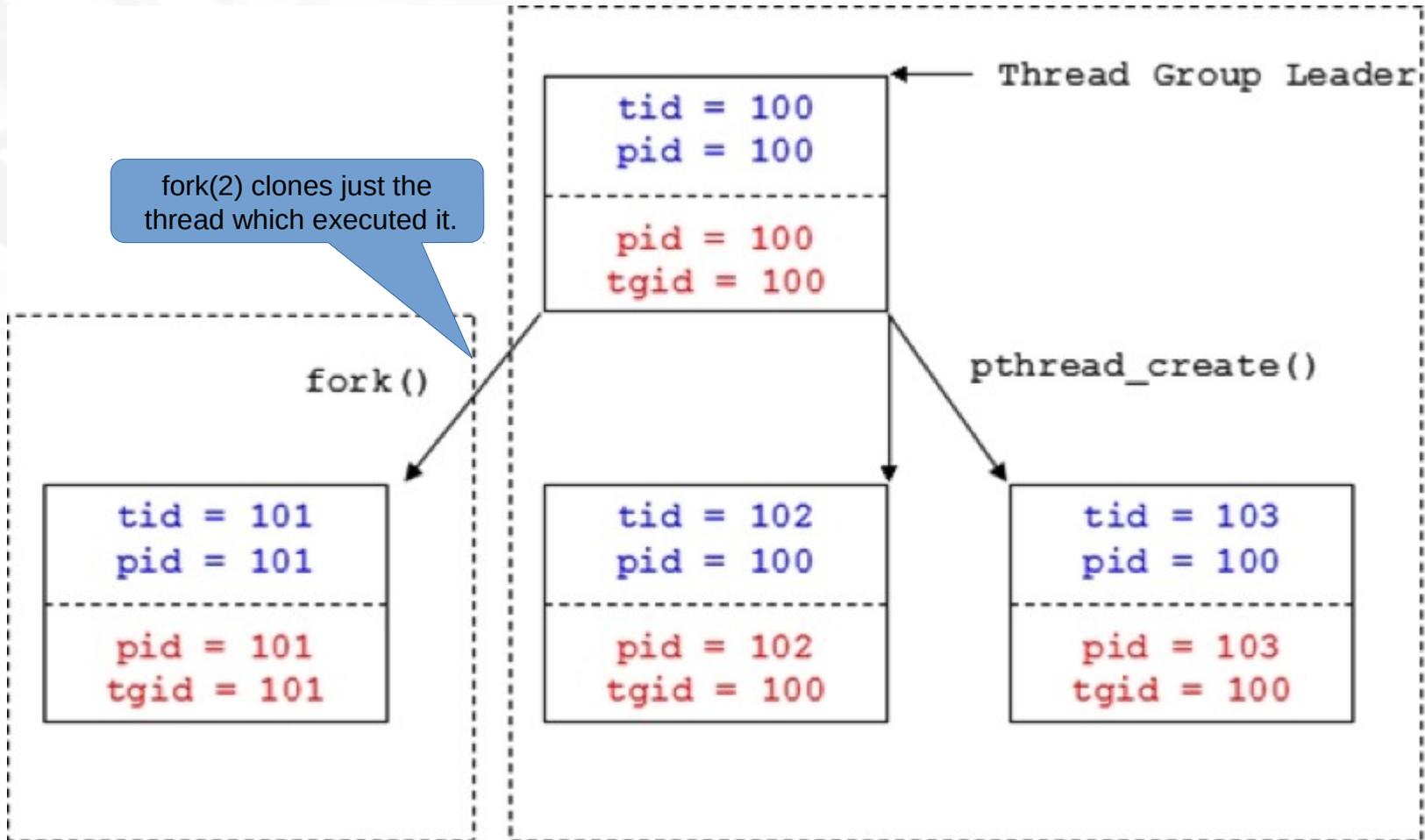
- `pthread_t *thread` - ponteiro para um objeto que recebe a identificação da nova *thread*.
- `pthread_attr_t *attr` - ponteiro para um objeto que provê os atributos para a nova *thread*.
- `start_routine` - função com a qual a *thread* inicia a sua execução
- `void *arg` - arqumentos inicialmente passados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

função

## Criação de Threads: `pthread_create()` (2)

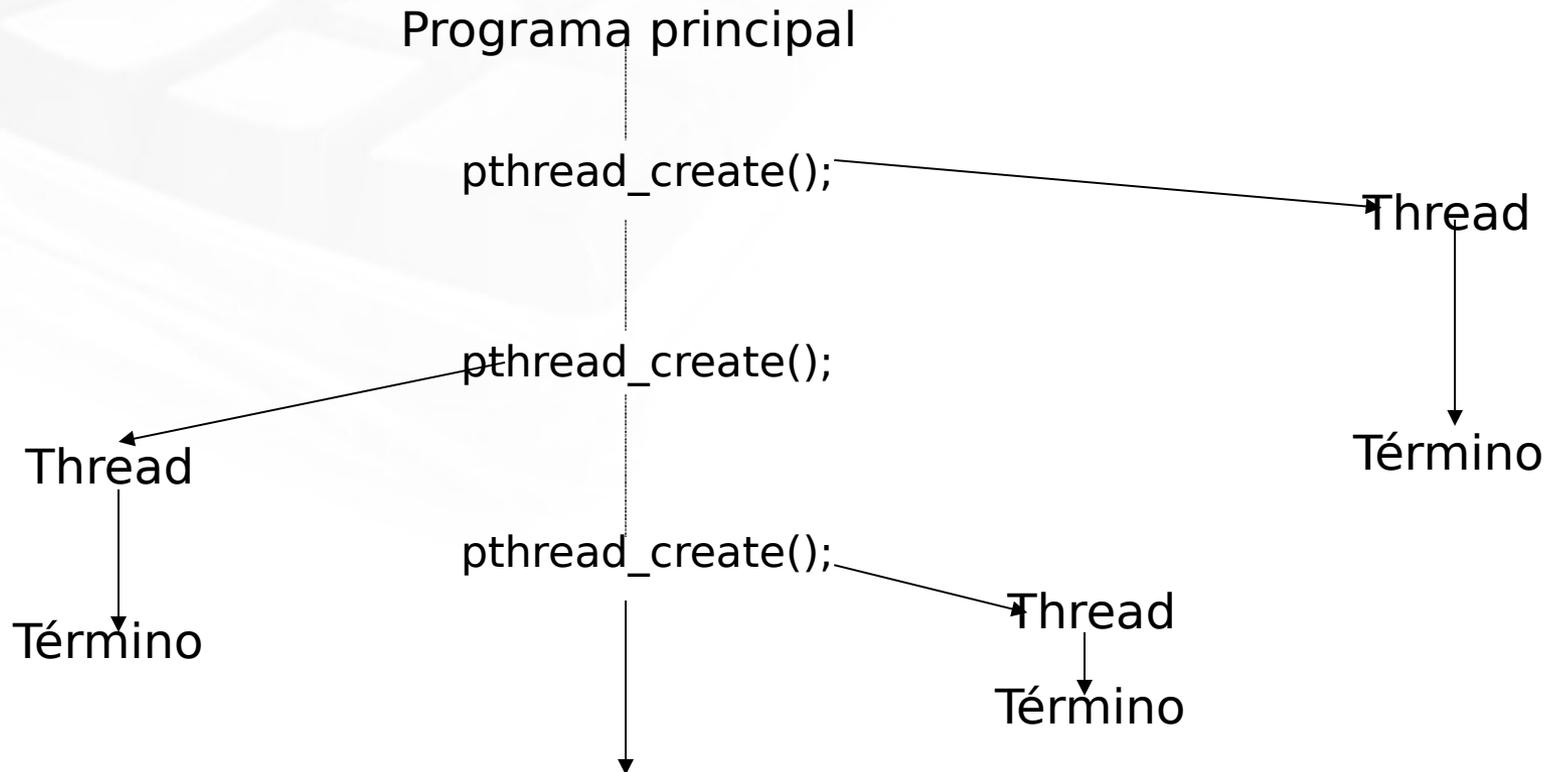
- Quando se cria uma nova *thread* é possível especificar uma série de atributos e propriedades através de uma variável do tipo `pthread_attr_t`.
- Os atributos que afetam o comportamento da *thread* são definidos pelo parâmetro `attr`. Caso o valor de `attr` seja `NULL`, o comportamento padrão é assumido para a *thread* :
  - (i) ***unbound***; (ii) ***nondetached***; (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()`.

# Criação de Threads: pthread\_create()



# Threads Desunidas (“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 `pthread_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()` <sup>(1)</sup>

- A função `pthread_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);
```

## Exemplo 1

```
#include <stdio.h>
#include <pthread.h>
int global;
void *thr_func(void *arg);
int main(void)
{
```

```
    pthread_t tid;
    global = 20;
    printf("Thread principal: %d\n", global);
    pthread_create(&tid, NULL, thr_func, NULL);
    pthread_join(tid, NULL);
    printf("Thread principal: %d\n", global);
    return 0;
```

```
}
void *thr_func(void *arg)
{
    global = 40;
    printf("Nova thread: %d\n", global);
    sleep(10);
    printf("END Nova thread\n");
    return NULL;
}
```

**OBS: %gcc -o exemplo1.c -lpthread  
%ps 1 -L**

## Exemplo 2

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

/* Prints x's to stderr. The parameter is unused. Does not return. */
void* print_xs (void* unused)
{
    while (1)
        fputc ('x', stderr);
    return NULL;
}

/* The main program. */

int main ()
{
    pthread_t thread_id;
    /* Create a new thread. The new thread will run the print_xs function. */
    pthread_create (&thread_id, NULL, &print_xs, NULL);
    /* Print o's continuously to stderr. */
    while (1)
        fputc ('o', stderr);
    return 0;
}
```

## Exemplo 3

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid) {
    int tid;
    tid = (int)threadid; sleep(10);
    printf("Hello World! It's me, thread #%d!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc) {
            printf("ERROR code is %d\n", rc);
            printf("Check PTHREAD_THREADS_MAX in /proc/sys/kernel/threads-
max\n");
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

## Exemplo 4

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void* function(void* arg) {
    printf( "This is thread %d, PID %d\n", (int)
           pthread_self(),getpid() );
    sleep(5);
    return (void *)99;
}
int main(void) {
    pthread_t t2;
    void *result;
    pthread_attr_init( &attr );
    pthread_create( &t2, &attr, function, NULL );
    pthread_join(t2,&result);
    printf("Thread t2 returned %d\n", result);
    return 0;
}
```

## Exemplo 5

## Exercício: Soma

- Somar os elementos de um array

`a[1000]`

```
int sum, a[1000]
```

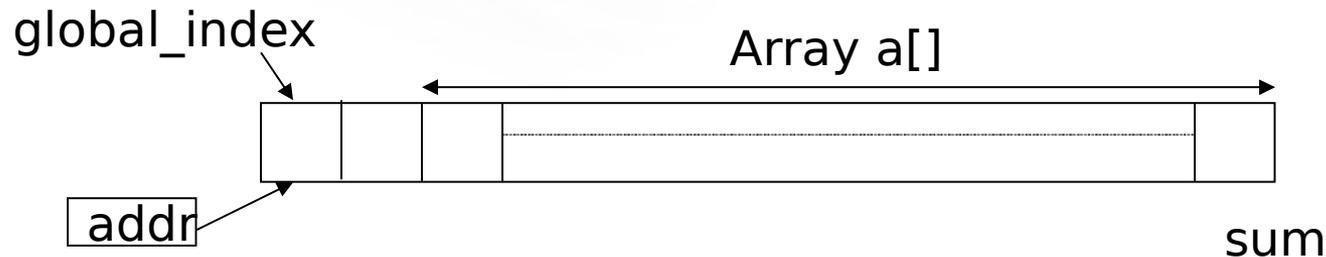
```
sum = 0;
```

```
for (i = 0; i < 1000; i++)
```

```
    sum = sum + a[i];
```

## Exemplo: Soma

- São criadas  $n$  *threads*. Cada uma obtém os números de uma lista, os soma e coloca o resultado numa variável compartilhada **sum**
- A variável compartilhada **global\_index** é utilizada por cada *thread* para selecionar o próximo elemento de **a[]**
  - Após a leitura do índice, ele é incrementado para preparar para a leitura do próximo elemento
- Estrutura de dados utilizada:



```
#define array_size 1000
#define no_threads 10

int a[array_size];
int global_index = 0;
int sum = 0;
pthread_mutex_t mutex1;

void * slave ( void *nenhum )
{
    int local_index, partial_sum =0;
    do {
        pthread_mutex_lock(&mutex1);
        local_index = global_index;
        global_index++;
        pthread_mutex_unlock(&mutex1);
        if (local_index < array_size)
            partial_sum += *(a+local_index);
    } while (local_index < array_size);

    pthread_mutex_lock(&mutex1);
    sum+= partial_sum;
    pthread_mutex_unlock(&mutex1);
    return(NULL);
}
```

```
main()
{
    int i;
    pthread_t  thread [no_threads] ;

    pthread_mutex_init(&mutex1, NULL);
    for (i = 0; i < array_size; i++)
        a[i] = i+1;

    for (i = 0; i < no_threads; i++)
        if (pthread_create(&thread[i], NULL, slave, NULL)
            != 0)
        {
            perror("Pthread_create falhou");
            exit(1);
        }

    for (i = 0; i < no_threads; i++)
        if (pthread_join(thread[i], NULL) != 0)
        {
            perror("Pthread_join falhou");
            exit(1);
        }

    printf("A soma é %d \n", sum)
}
```

# Acesso a Dados Compartilhados: Mutexes

- A biblioteca *pthread* fornece funções para acesso exclusivo a dados compartilhados através de *mutexes*.
- O mutex garante três coisas:
  - Atomicidade: o travamento de um *mutex* é sempre uma operação atômica, o que significa dizer que o S.O. ou a biblioteca de *threads* garante que se uma *thread* alocou (travou) o *mutex*, nenhuma outra *thread* terá sucesso se tentar travá-lo ao mesmo tempo.
  - Singularidade: se uma *thread* alocou um *mutex*, nenhuma outra será capaz de alocá-lo antes que a *thread* original libere o travamento.
  - Sem espera ocupada: se uma *thread* tenta travar um *mutex* que já está travado por uma primeira *thread*, a segunda *thread* ficará suspensa até que o travamento seja liberado. Nesse momento, ela será acordada e continuará a sua execução, tendo o *mutex* travado para si.

# Criando e Inicializando um Mutex

```
pthread_mutex_lock ( &mutex1);  
<seção crítica>  
pthread_mutex_unlock( &mutex1 );
```

# Threads - O uso de mutex (1)

- Initialization

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                        const pthread_mutexattr_t *attr);
```

- Destroy

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Lock request

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Lock release

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

## Threads - O uso de mutex (2)

```
pthread_mutex_t meu_mutex = PTHREAD_MUTEX_INITIALIZER;
int somatotal=0;

void *realiza_soma(void *p){
    int resultado=0, i;
    int meu_id = ((ARGS *)p)->id;

    /* soma N numeros aleatorios entre 0 e MAX */
    for(i=0 ; i<N ; i++)
        resultado += rand()%MAX;

    /* armazena a soma parcial */
    pthread_mutex_lock(&meu_mutex);
    somatotal += resultado;
    pthread_mutex_unlock(&meu_mutex);
    printf("\nThread %d: parcial %d",meu_id,resultado);

    pthread_exit((void *)0);
}
```

# Threads - O uso de variáveis de condição (1)

- Initialization

```
pthread_cond_t cond= PTHREAD_COND_INITIALIZER;  
int pthread_cond_init(pthread_cond_t *cond,  
                      const pthread_condattr_t *attr);
```

- Waiting on condition variable

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);  
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,  
                           const struct timespec *abstime);
```

- Waking condition variable waiters

```
int pthread_cond_signal(pthread_cond_t *cond);  
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Destory

```
– int pthread_cond_destroy(pthread_cond_t *cond);
```

# Threads - O uso de variáveis de condição (1)

Variáveis condicionais	Pthread_cond_init Pthread_cond_destroy Pthread_cond_wait Pthread_cond_timedwait Pthread_cond_signal Pthread_cond_broadcast
------------------------	---

- A API de pthreads implementa as funções *pthread\_cond\_wait* e *pthread\_cond\_signal* sobre uma variável declarada como *pthread\_cond\_t*.
- Esta variável tem, necessariamente, que trabalhar associada a um mutex

# Threads - O uso de variáveis de condição (2)

- Procedimento básico para implementar variáveis de condição

Thread Principal	Thread A	Thread B
Declara mutex		
Declara cond		
Inicializa mutex		
Inicializa cond		
Cria A e B		
	Realiza Trabalho	Realiza Trabalho
	Trava o mutex	
	Verifica condição	
	Chama wait destravando mutex	
		Satisfaz a condição
		Sinaliza
		Destrava Mutex
	Acorda travando mutex	Continua
	Faz trabalho na S.Crítica	
	Destrava mutex	
	Continua	

Trava o mu

# Threads - O uso de variáveis de condição (3)

```
int recurso=42;
pthread_mutex_t meu_mutex;
pthread_cond_t minha_cond;

void *produtor(){
    /* espera um pouco para permitir
     * o consumidor iniciar primeiro */
    sleep(3);

    /* executa seção crítica */
    pthread_mutex_lock(&meu_mutex);
    recurso = rand();
    pthread_cond_signal(&minha_cond);
    pthread_mutex_unlock(&meu_mutex);

    pthread_exit(NULL);
}

void *consumidor(){
    pthread_mutex_lock(&meu_mutex);
    pthread_cond_wait(&minha_cond, &meu_mutex);
    printf("Valor do recurso: %d\n",recurso);
    pthread_mutex_unlock(&meu_mutex);
}
```

Esta função realiza 3 operações atomicamente:

1. destrava o mutex
2. espera, propriamente, ser sinalizado
3. trava o mutex

# Producer - Consumer

```
void producer()
{
    pthread_mutex_lock(mutex);
    while (isFull == 1)
        pthread_cond_wait(condFull, mutex);
    buffer = rand();
    isFull = 1;
    pthread_cond_signal(condEmpty, mutex);
    pthread_mutex_unlock(mutex);
}
...
char consumer()
{
    char item;
    pthread_mutex_lock(mutex);
    while(isFull == 0)
        pthread_cond_wait(condEmpty, mutex);
    item = buffer; //consumes!!
    isFull = 0;
    pthread_cond_signal(condFull);
    pthread_mutex_unlock(mutex);
    return(item);
}
```

Esta função realiza 3 operações atômicas:

1. destrava o mutex
2. espera, propriamente, ser sinalizado
3. trava o mutex

# 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 por meio 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 (task), mas não é feita uma cópia, como no *fork()*;
  - O novo processo aponta p/ as estruturas de dados do pai

## Java Threads

- Threads em Java podem ser criadas das seguintes maneiras:
  - Estendendo a classe Thread
  - Implementando a interface Runnable.
- As threads Java são gerenciadas pela JVM.
- A JVM só suporta um processo
  - Criar um novo processo em java implica em criar uma nova JVM p/ rodar o novo processo

## Referências

- Silberschatz A. G.; Galvin P. B.; Gagne G.; "Fundamentos de Sistemas Operacionais", 6a. Edição, Editora LTC, 2004.
  - Capítulo 5
- A. S. Tanenbaum, "Sistemas Operacionais Modernos", 3a. Edição, Editora Prentice-Hall, 2010.
  - Seção 2.2
- Deitel H. M.; Deitel P. J.; Choffnes D. R.; "Sistemas Operacionais", 3ª. Edição, Editora Prentice-Hall, 2005
  - Capítulo 4