Laboratório de Pesquisa em Redes e Multimídia

Sistemas Operacionais

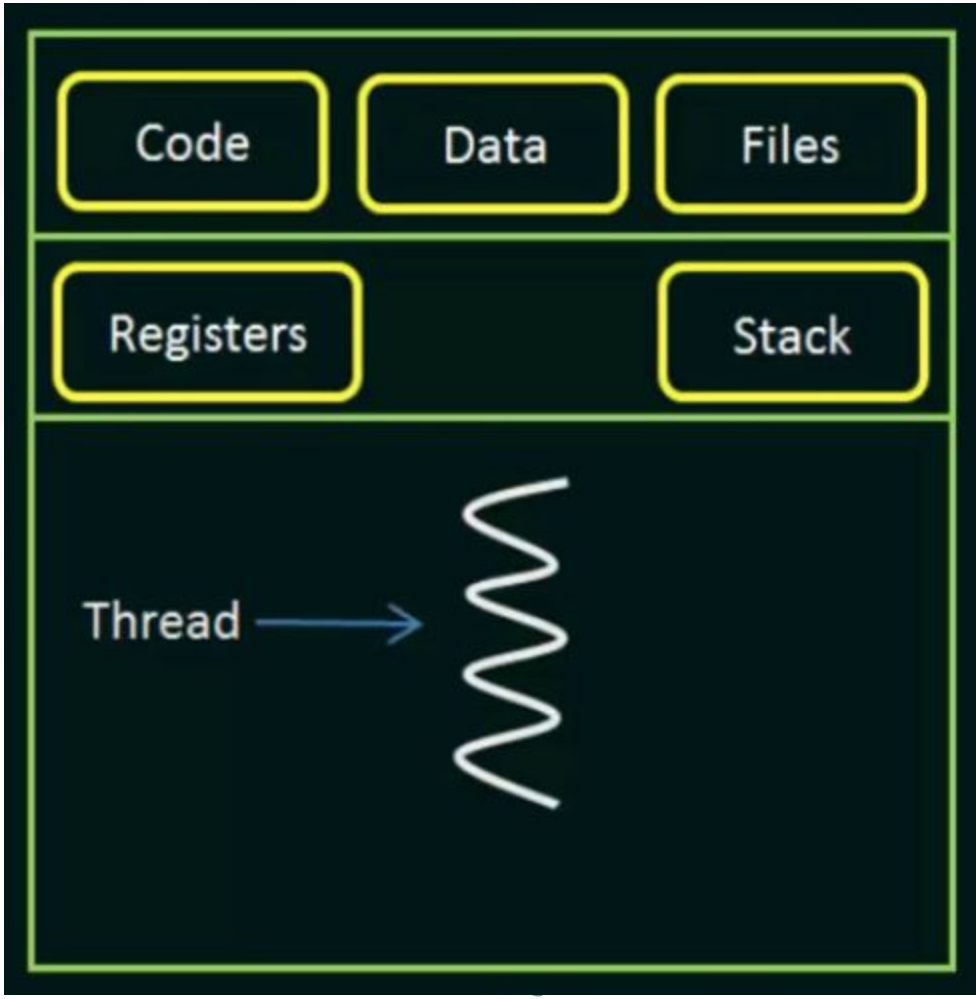
Threads

Universidade Federal do Espírito Santo
Departamento de Informática

Fluxos de Execução (1)

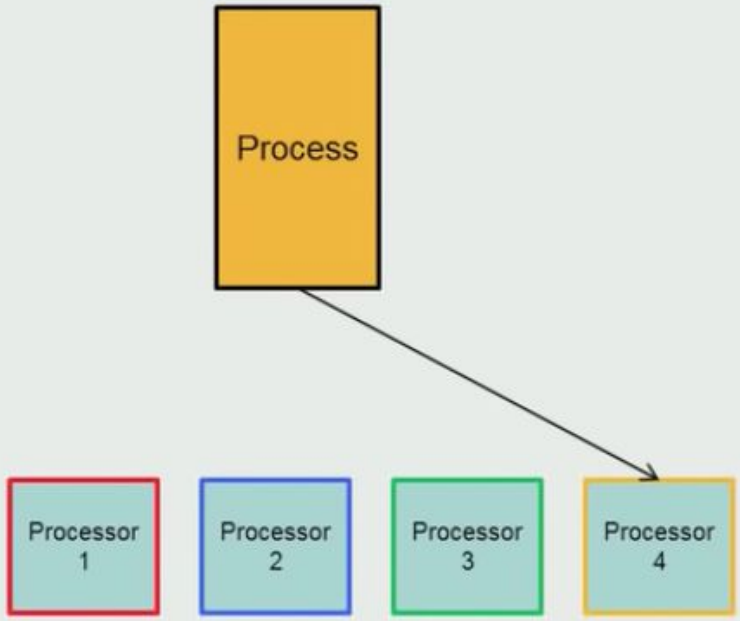
- Um programa sequencial consiste de um único fluxo ou *thread* de execução , que é responsável por realizar uma certa tarefa computacional.
- A maioria dos programas simples tem essa característica: só possuem uma única *thread* de execução ("*single thread*"). Por conseguinte, não conseguem executar duas ou mais tarefas em paralelo, mesmo em um ambiente de multiprocessamento.
- Entretanto, hoje grande parte do software de maior complexidade faz uso de mais de uma thread de execução.

"Single Threading"

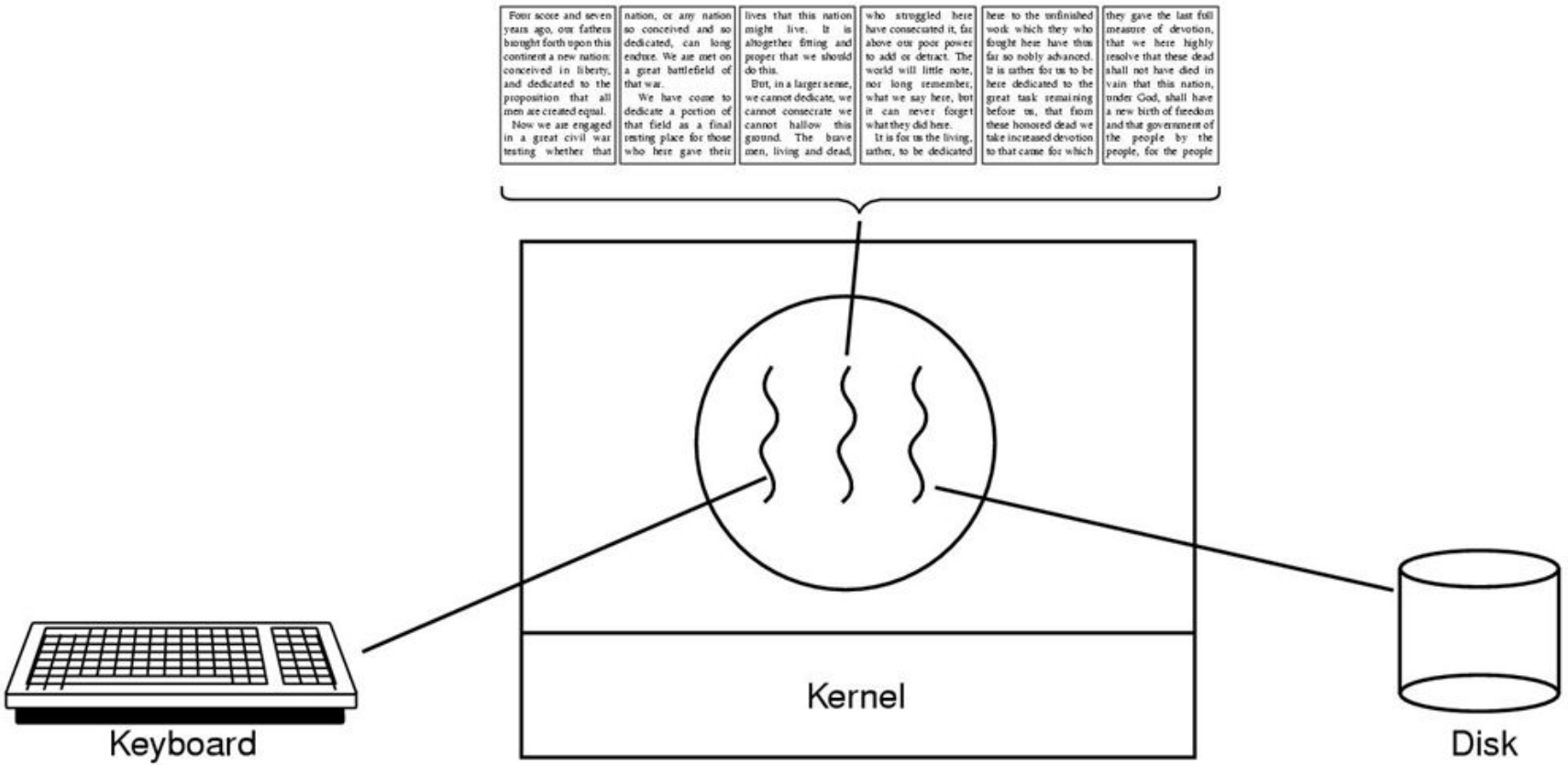


"Single Threading" (2)

```c
#include <stdio.h>


unsigned long addall(){
  int i=0;
  unsigned long sum=0;

  while (i< 10000000){
     sum += i;
     i++;
  }
  return sum;
}


int main()
{
   unsigned long sum;
   srandom(time(NULL));
   sum = addall();
   printf("%lu\n", sum);
}
```

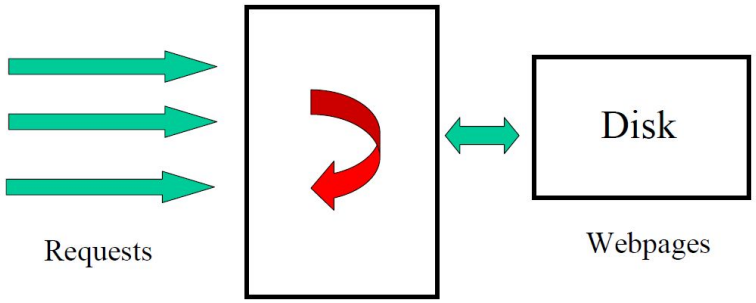


Exemplo de Multithreading: Editor de Texto (1)



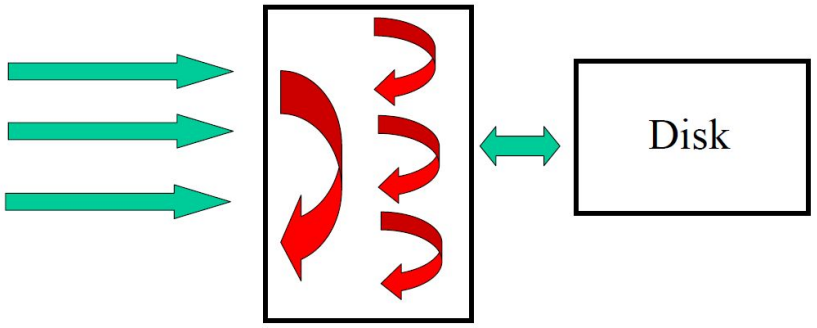
Exemplo de Multithreading: Servidor Web (2)

Single Threaded Web Server

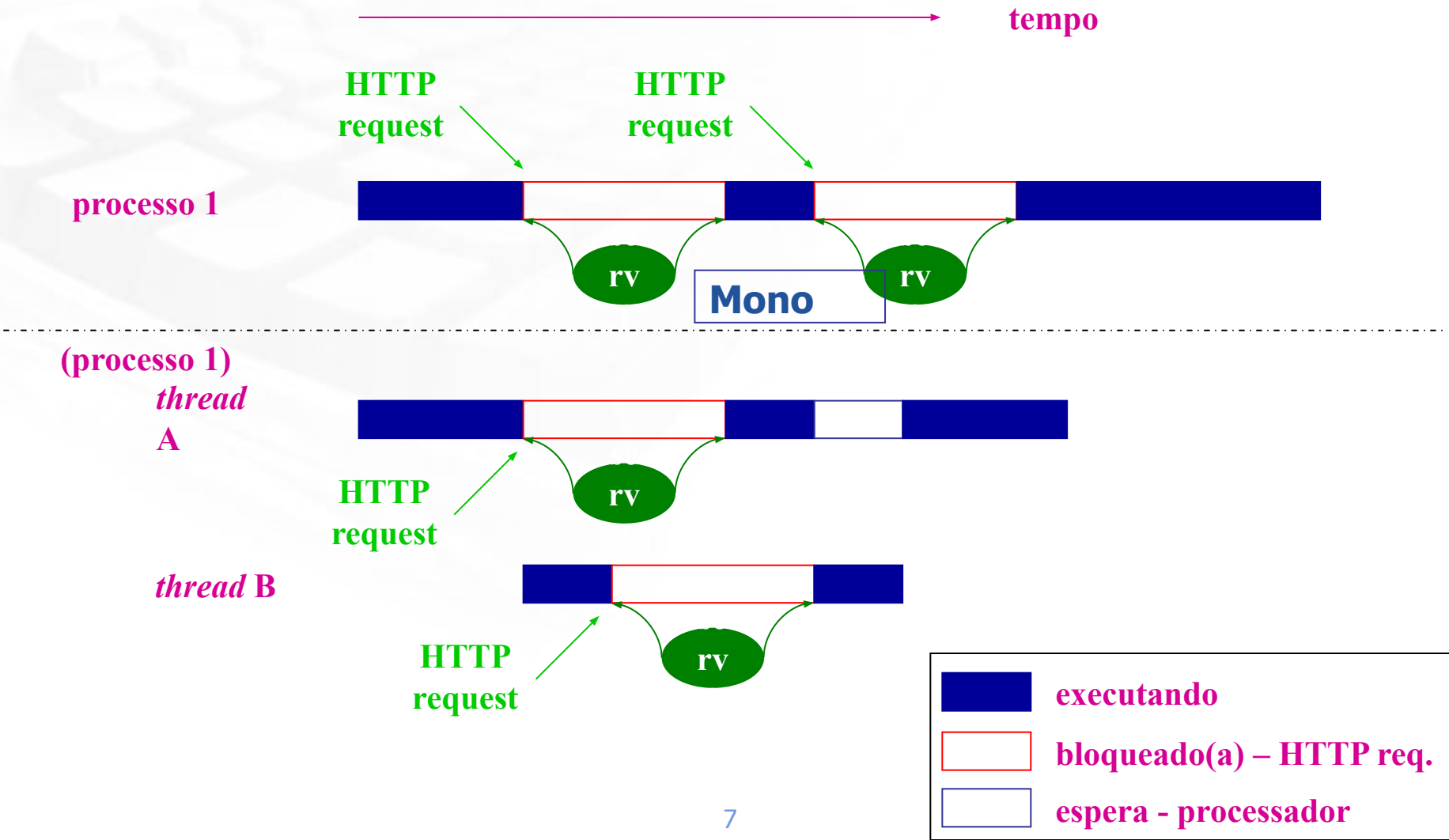


Cannot overlap Disk I/O with listening for requests

Multi Threaded Web Server



Exemplo de Multithreading: Servidor Web (3)



Exemplo de Multithreading: Programas Numéricos (4)

- Multiplicação de matrizes:
  - Cada elemento da matriz produto pode ser calculado independentemente dos outros; portanto, podem ser facilmente calculados por *threads* diferentes.

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

- De modo análogo, existe uma série de programas, de naturezas distintas, cujas tarefas podem ser paralelizadas, aumentando a velocidade e desempenho de execução.

Recursos *vs* Escalonamento (1)

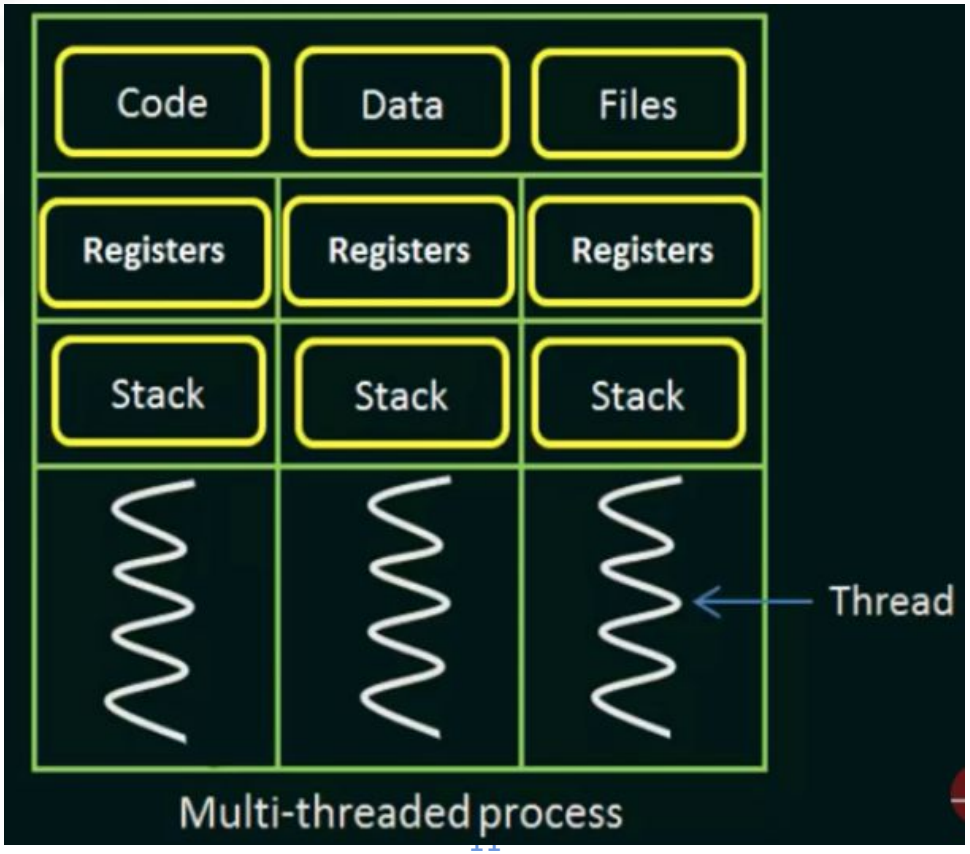
- Existem duas características fundamentais que são usualmente tratadas de forma independente pelo S.O:
  - Propriedade de recursos ("*resource ownership*");
  - Escalonamento ("*scheduling / dispatching*").
- **Propriedade de recursos:**
  - Trata dos recursos alocados aos processos, e que são necessários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

Recursos *vs* Escalonamento (2)

- Tradicionalmente um processo está associado a:
  - um programa em execução
  - um conjunto de recursos
- Em um sistema multithread:
  - **processos** estão associados somente à **propriedade de recursos**
  - ***threads*** estão associadas às **unidades de execução** (ou seja, *threads* constituem as unidades de escalonamento em sistemas *multithreading*).

Definindo Threads (1)

- Thread é uma abstração que permite que uma aplicação execute mais de um dos seus trechos de código simultaneamente.



Definindo Threads (2)

A thread is a basic unit of CPU utilization.

It comprises

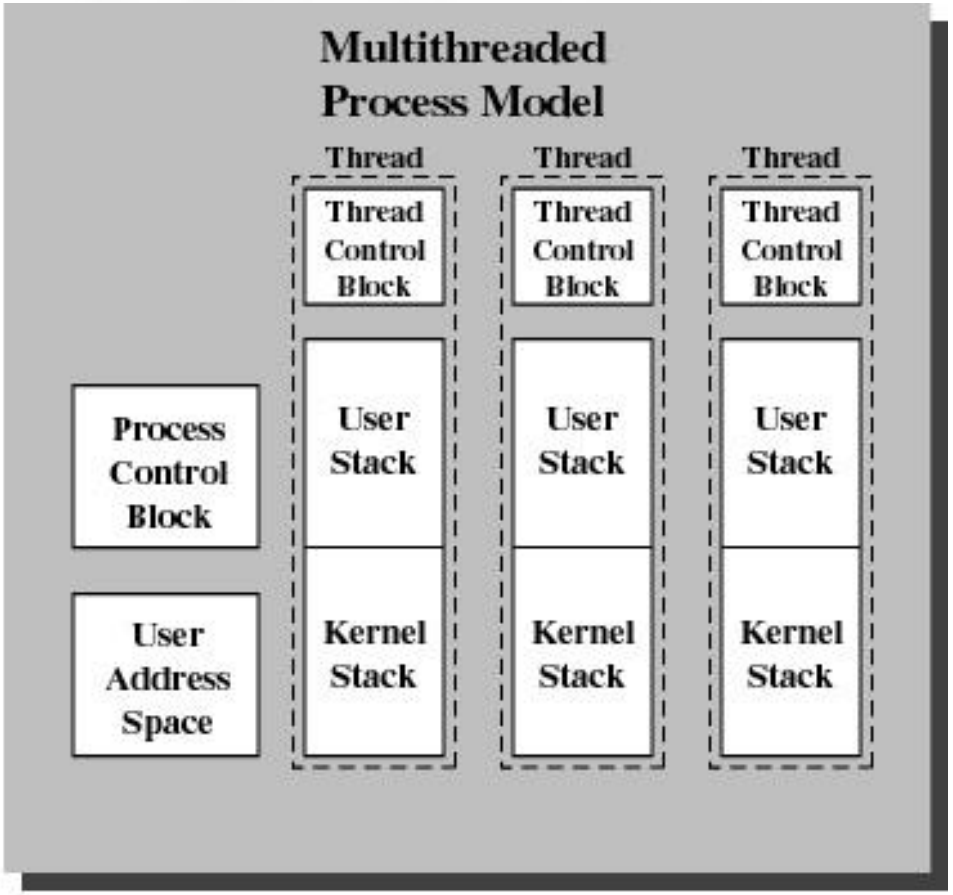
- A thread ID
- A program counter
- A register set and
- A stack

It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

A traditional / heavyweight process has a single thread of control.

If a process has multiple threads of control, it can perform more than one task at a time.

Modelo de Processo Multithreading

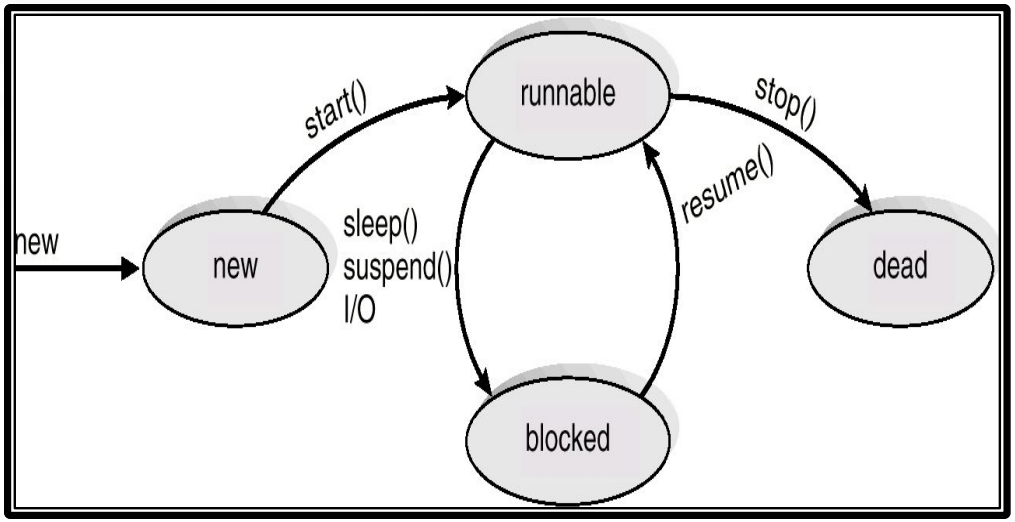


Task Control BLock

- 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
  - Endereços da pilha
  - Máscara de sinais
  - Prioridade
  - Variáveis locais e variáveis compartilhadas com as outras *threads*
  - Estado de execução (pronta, bloqueada, executando)

Estados de uma Thread

- 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).



Vantagens das Threads sobre Processos (1)

- Economia:
  - Alocar memória e recursos para a criação de um processo é custoso.
  - Uma vez que threads compartilham recursos do processo ao qual elas pertencem, é mais econômico criar e fazer o escalonamento (troca de contexto) de threads do mesmo processo.
  - A criação e terminação de uma *thread* é mais rápida do que a criação e terminação de um processo pois elas pouquíssimos recursos são alocados a elas (essencialmente algumas poucas estruturas de controle).
  - A troca de contexto entre *threads* (de um mesmo processo) é mais rápida do que entre dois processos, pois elas compartilham os recursos do processo.
    - Por exemplo, cache e alguns registradores de endereço são mantidos

Vantagens das Threads sobre Processos (2)

"For example, the following table compares timing results for the fork() subroutine and the pthread_create() subroutine. Timings reflect 50,000 process/thread creations, were performed with the time utility, and units are in seconds, no optimization flags"
https://hpc-tutorials.llnl.gov/posix/why_pthreads/

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Vantagens das Threads sobre Processos (3)

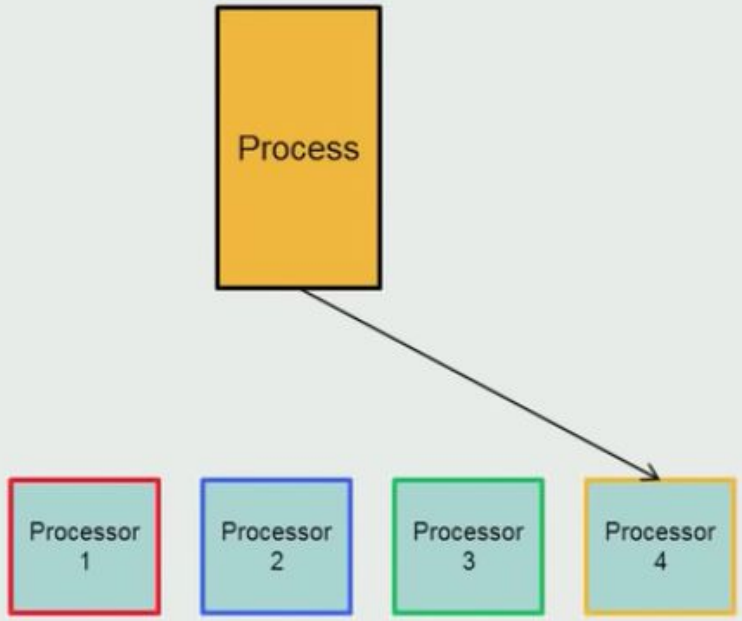
- Responsividade:
  - Transformando uma aplicação interativa em multithread pode permitir que o programa continue executando mesmo se uma parte dele se bloqueie, ou executando uma sequencia grande de operações (em background), aumentando com isso a responsividade ao usuário.
- Compartilhamento de recursos:
  - Por default, threads compartilham a memória e os recursos alocados pelo processo ao qual pertencem.
  - Com o compartilhamento de código e dados, uma mesma aplicação pode possuir várias threads realizando atividades diferentes em cima do mesmo espaço de endereçamento.
  - A comunicação entre *threads* é mais rápida do que a comunicação entre processos, já que elas compartilham o espaço de endereçamento do processo.

Vantagens das Threads sobre Processos (4)

- Utilização de arquiteturas multiprocessadores:
  - Os benefícios de multhreading podem ser aumentados consideravelmente em uma arquitetura com multiprocessamento, em que threads de um mesmo processo podem estar executando em paralelo em diferentes processadores.
  - Um processo single-threaded pode rodar em apenas um processador, não importando quantos processadores disponíveis a máquina apresente.
  - Multithreading em uma máquina multi-CPU aumenta os níveis de concorrência do sistema.

Threads vs Processos

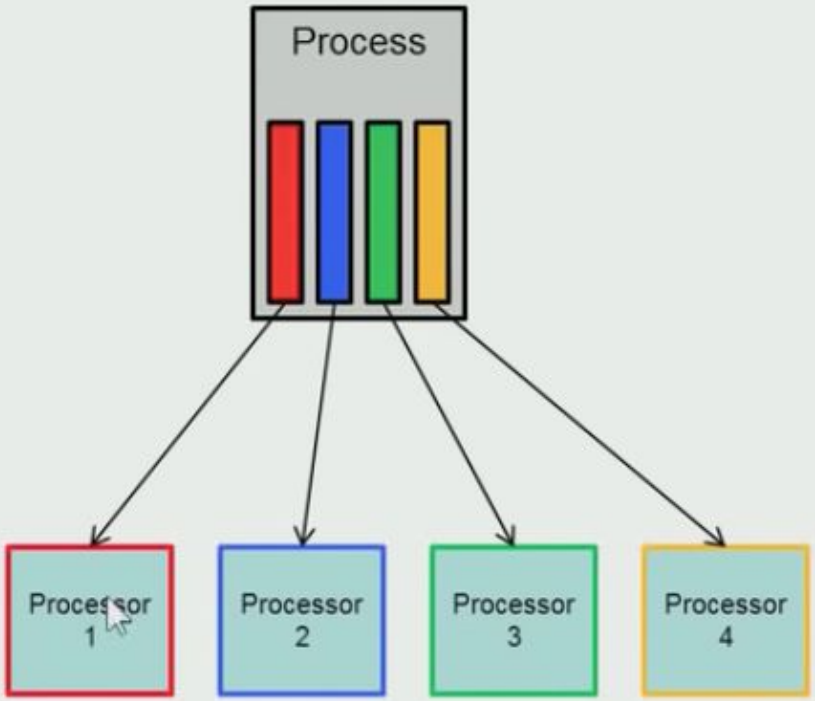
```c
#include <stdio.h>


unsigned long addall(){
  int i=0;
  unsigned long sum=0;

  while (i< 10000000){
     sum += i;
     i++;
  }
  return sum;
}


int main()
{
   unsigned long sum;
   srandom(time(NULL));
   sum = addall();
   printf("%lu\n", sum);
}
```



Threads vs Processos

10000000 / 4 = 2500000

Create 1 process with 4 threads; each loop does 1/4th of the work

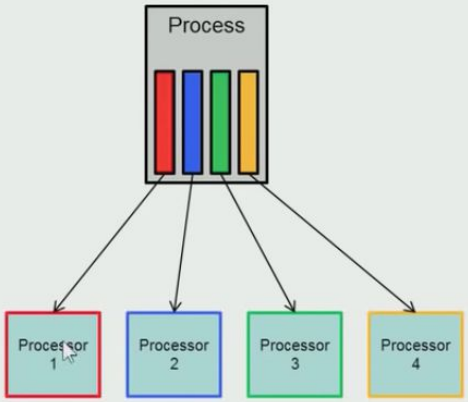


Threads vs Processos

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

unsigned long sum[4];

void *thread_fn(void *arg){
  long id = (long) arg;
  int start = id * 2500000;
  int i=0;

  while(i < 2500000){
     sum[id] += (i + start);
     i++;
  }
  return NULL;
}

int main(){
  pthread_t t1, t2, t3, t4;

  pthread_create(&t1, NULL, thread_fn, (void *)0);
  pthread_create(&t2, NULL, thread_fn, (void *)1);
  pthread_create(&t3, NULL, thread_fn, (void *)2);
  pthread_create(&t4, NULL, thread_fn, (void *)3);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);
  pthread_join(t4, NULL);
  printf("%lu\n", sum[0] + sum[1] + sum[2] + sum[3]);
  return 0;
}
```

10000000 / 4 = 2500000

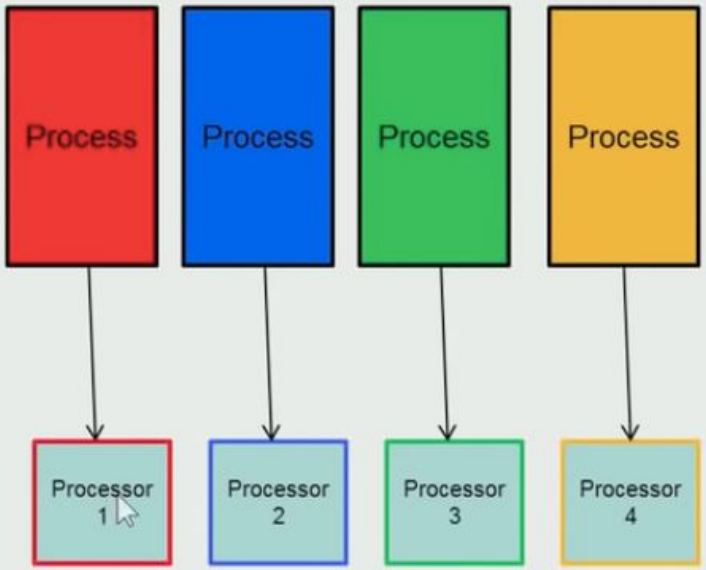
Create 1 process with 4 threads; each loop does 1/4th of the work



Threads vs Processos

10000000 / 4 = 2500000

Create 4 processes, each loop does 1/4th of the work



Properties:

4 fork system calls needed; one for creating each process

Each process is isolated from each other

IPC mechanisms to communicate – more system calls

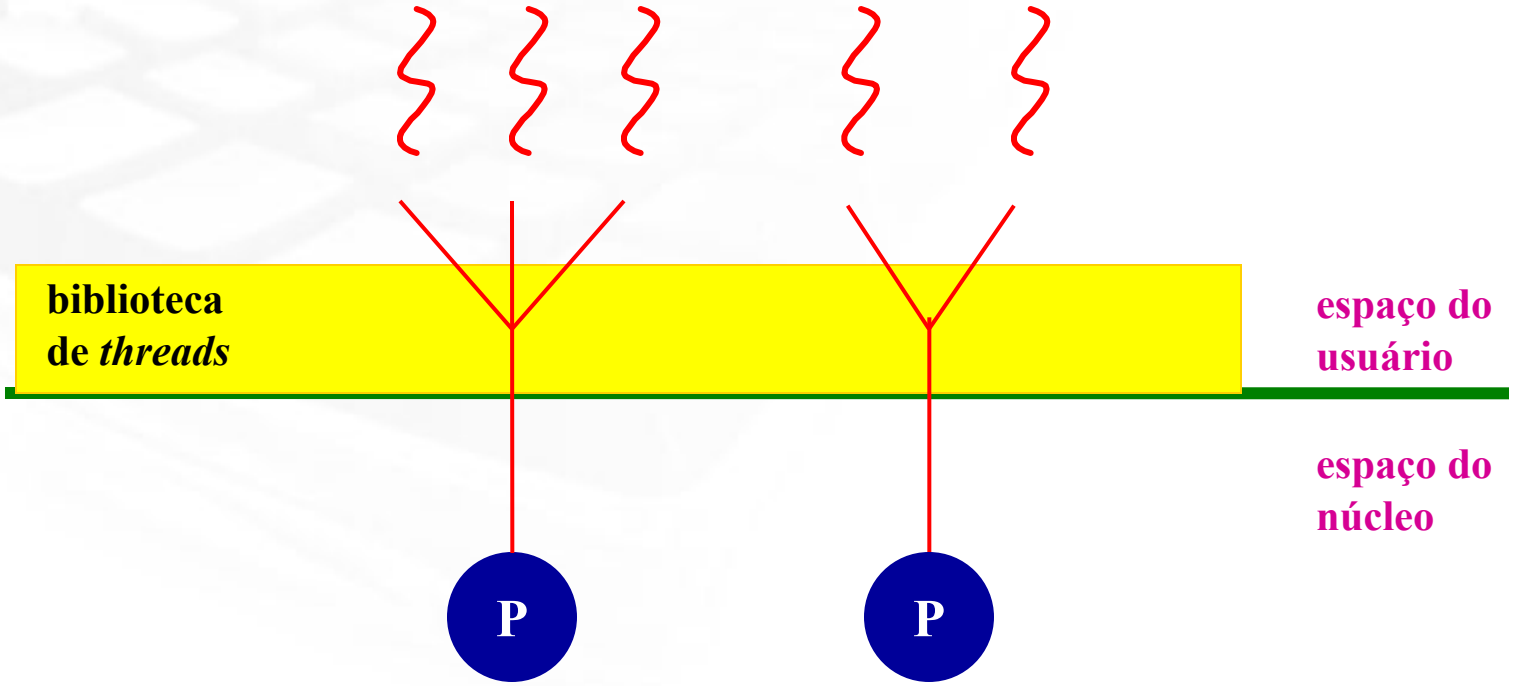
Process management with system calls

Each process has its own memory map – its own instructions, data, stack, heap, etc.

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*

User-level Threads - ULT (1)



User-level Threads - ULT (2)

- O gerenciamento das *threads*, incluindo o seu escalonamento, 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.

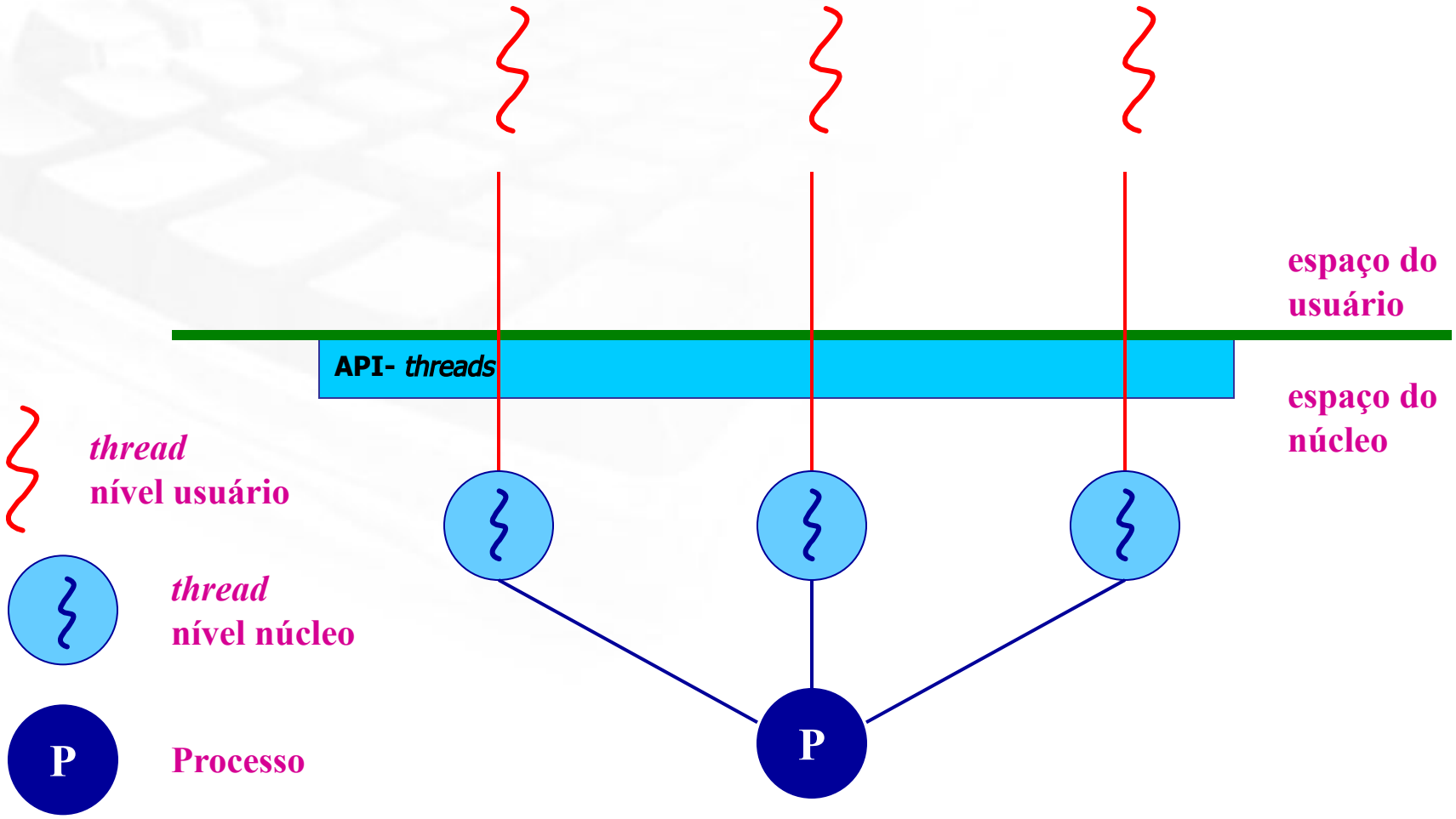
Benefícios das ULT

- 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.
- 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 do mesmo processo não podem executar simultaneamente numa arquitetura com múltiplas CPUs.
- Processos com várias threads tem a mesma fatia de tempo de CPU daqueles com poucas ou apenas uma única thread.

Kernel-level Threads - KLT (1)



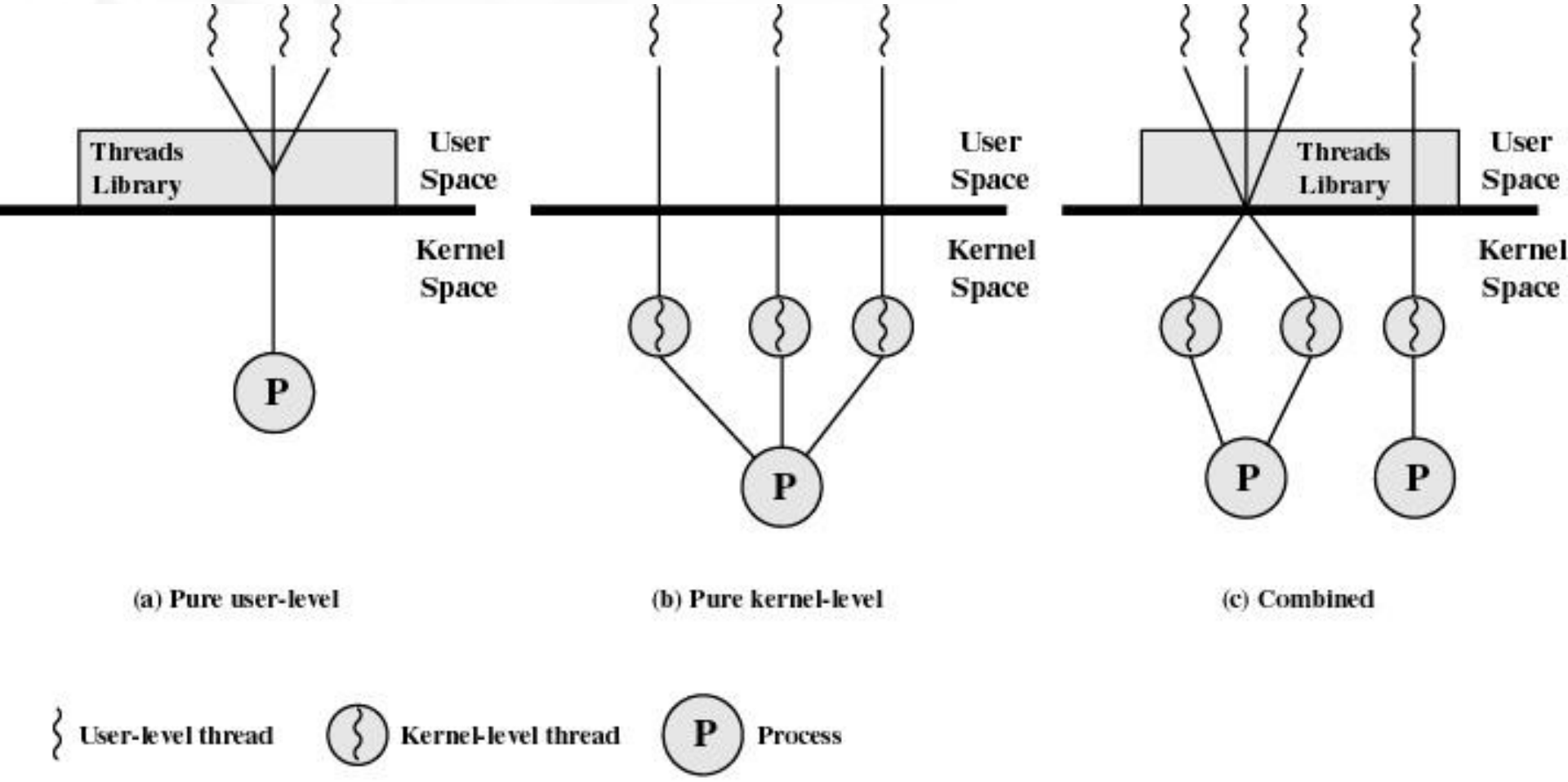
Kernel-level Threads – KLT (2)

- 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. Assim, KLT é interessante para aplicações que bloqueiam frequentemente.
- O *kernel* mantém a informação de contexto para processo e *threads*.

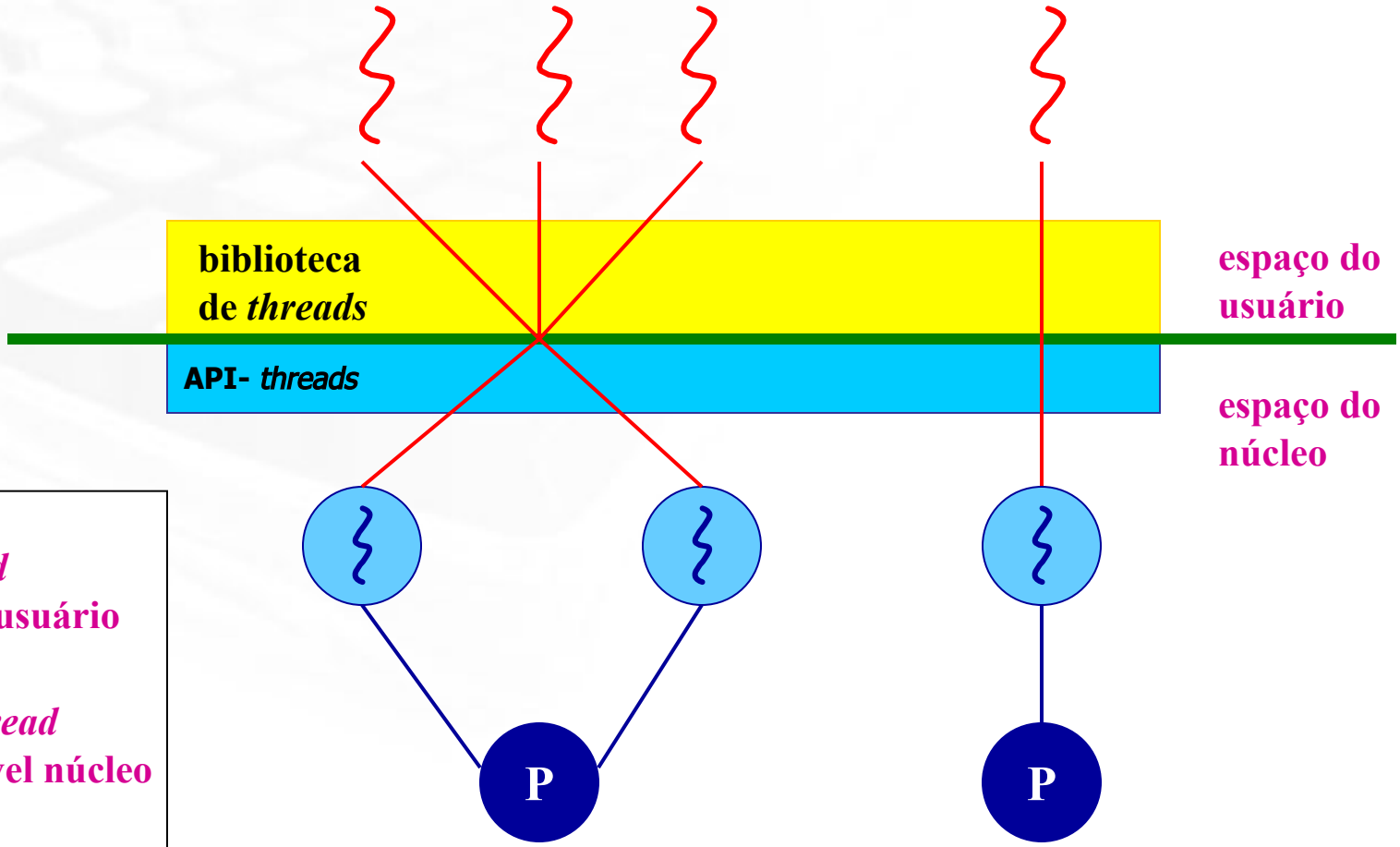
Kernel-level Threads – KLT (3)

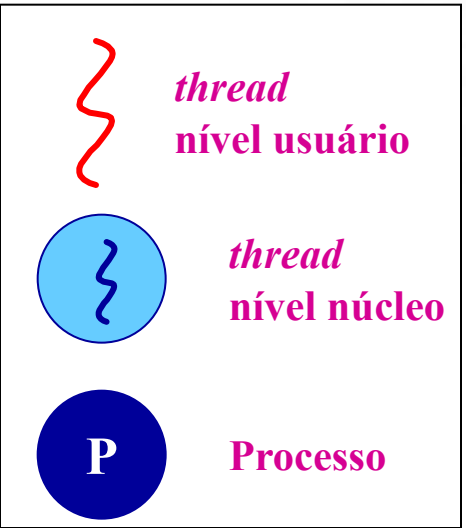
- O nível de usuário enxerga uma API para *threads* do núcleo.
- A transferência de controle entre *threads* de um mesmo processo requer chaveamento para modo *kernel*.
- Virtualmente todos os S.O. modernos - Windows, Linux, Mac OS, Solaris, suportam kernel threads.

Modelos de Implementação



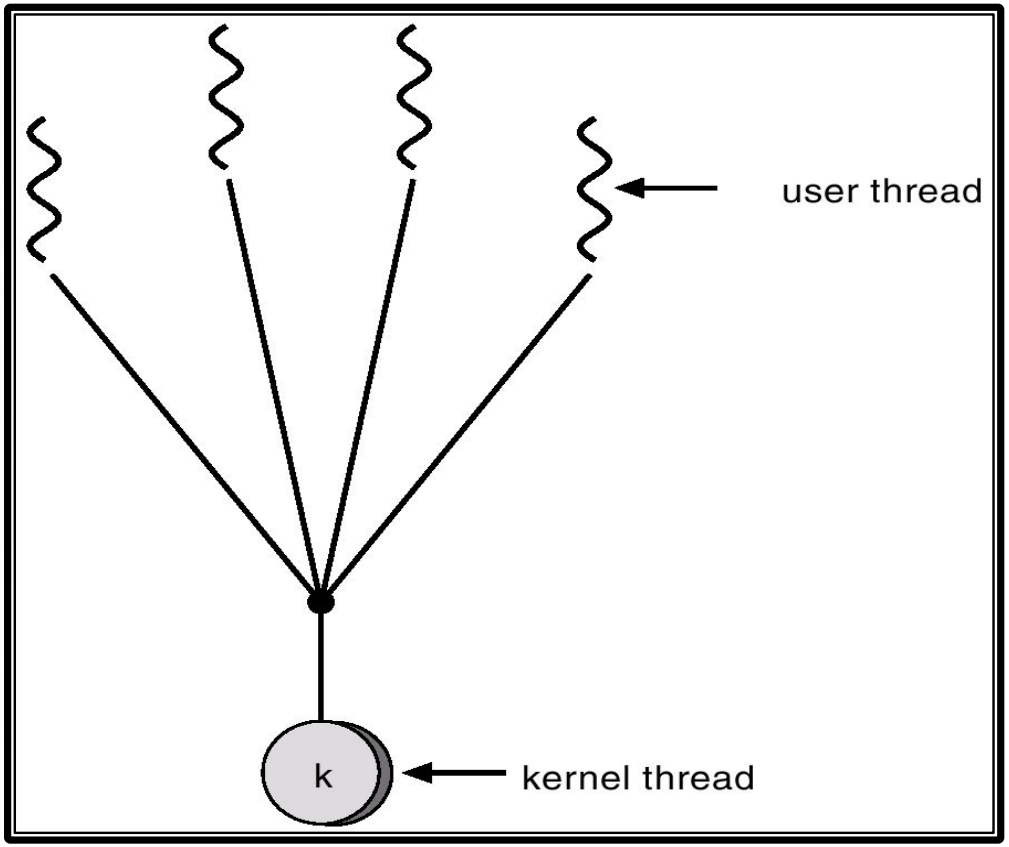
Modelos de Implementação





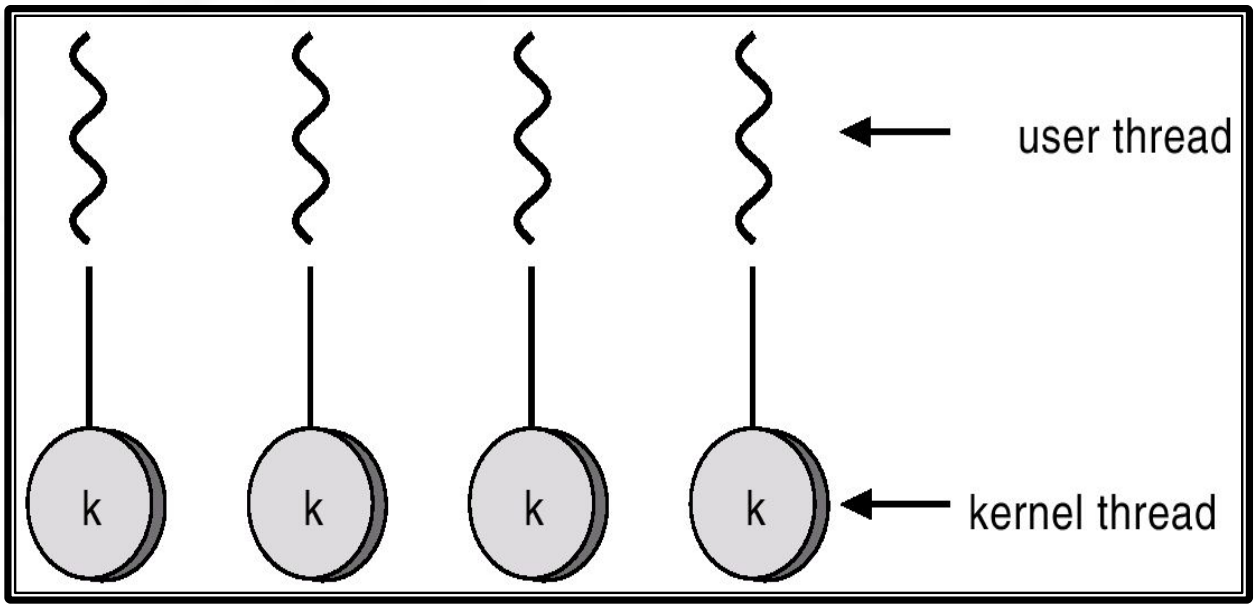
Modelo M:1

- Muitas *user-level threads* mapeadas em uma única *kernel thread*.
- Modelo usado em sistemas que não suportam *kernel threads*.



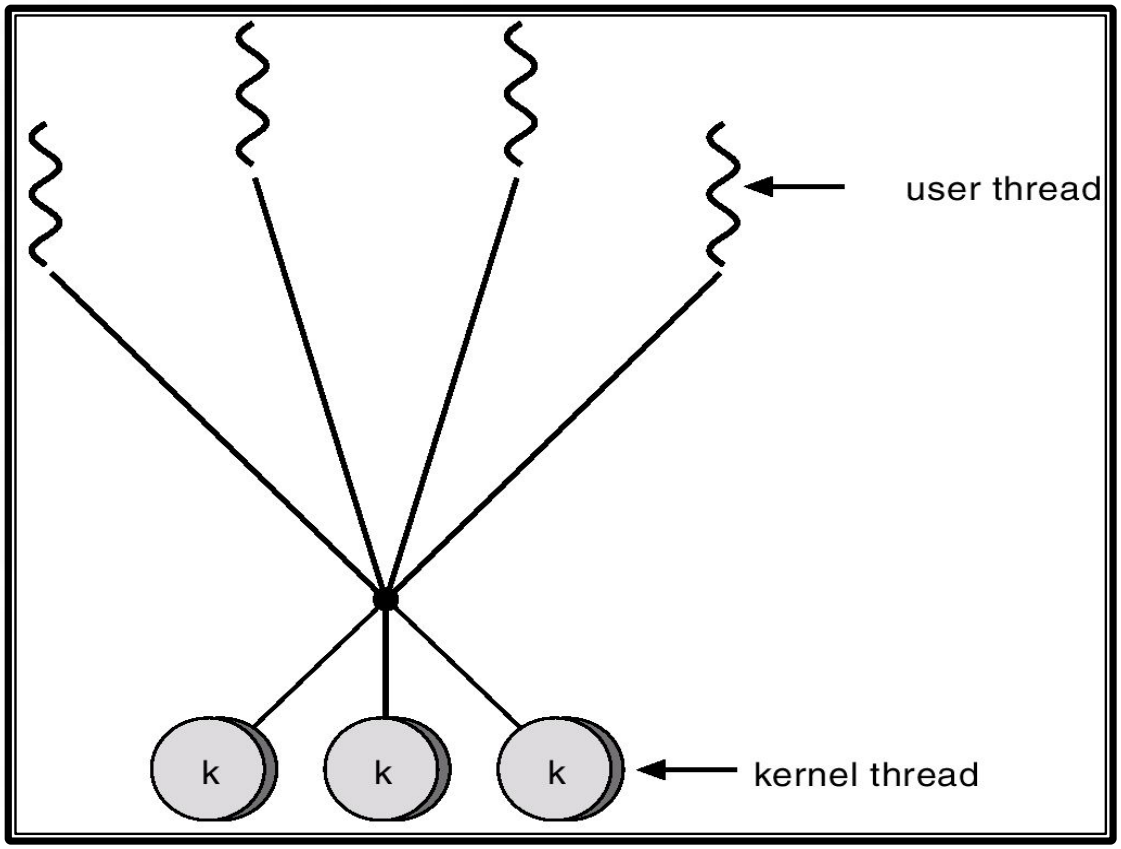
Modelo 1:1

- Cada *user-level thread* é mapeada em uma única *kernel thread*.



Modelo M:n

- Permite que diferentes *user-level threads* de um processo possam ser mapeadas em *kernel threads* distintas.
- Permite ao S.O. criar um número suficiente de *kernel threads*.



Threads no Linux

- O modelo de implementação de threads adotado no Linux é modelo 1:1.
- As versões atuais do Linux disponibilizam o **NPTL - Native POSIX Thread Library**, uma biblioteca de threads na qual threads criadas pelo usuário tem correspondência 1:1 com as entidades escalonáveis do kernel (tasks, no caso do Linux). Esta é a implementação mais simples possível de threads de núcleo.
- A biblioteca GNU C (glibc) implementa a interface **Pthreads** via biblioteca NPTL, onde cada ULT é mapeada em uma entidade de escalonamento do kernel.

Bibliotecas de Threads

- 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
- POSIX Threads ou **pthreads** provê uma interface padrão para manipulação de threads, que é independente de plataforma de S.O.

```
#include <pthreads.h>
$ gcc -o simple -pthread simple_threads.c
```

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()`

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()`.

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, 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>          OBS: %gcc –o pthread-create.c -lpthread
#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("Novo thread: %d\n", global);
    return NULL;
}
```

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;
     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);
               exit(-1);
          }
     }
     pthread_exit(NULL);
}
```

Exemplo 4

```
#include <pthread.h>
pthread_attr_t tattr;
pthread_t tid;
extern void *start_routine(void *arg);
void *arg;
int ret;

/* default behavior*/
ret = pthread_create(&tid, NULL, start_routine, arg);

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* default behavior specified*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

Exemplo 5

```
int main (int argc, char *argv[])
{
  pthread_t thread[100];
  int err_code, i=0;
  char *filename;

  printf ("Enter thread name at any time to create thread\n");
  while (1) {
   filename = (char *) malloc (80*sizeof(char));
   scanf ("%s", filename);
   printf("In main: creating thread %d\n", i);
   err_code = pthread_create(&thread[i],NULL,PrintHello,(void *)filename);
   if (err_code){
      printf("ERROR code is %d\n", err_code);
      exit(-1);
    }
    else i++;
  }
  pthread_exit(NULL);
}
```

Exemplo 6

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void* function(void* arg) {
  printf( "This is thread %d\n", pthread_self() );
  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;
}
```

Outros Exemplos:

Exemplo 1: Duas threads são criadas e para cada uma é passada uma variável do tipo inteiro. A primeira thread soma o valor 30 à variável, a segunda thread decrementa a variável de 10.

Exemplo 2: Uso de mutex para controlar o acesso a uma variavél compartilhada.

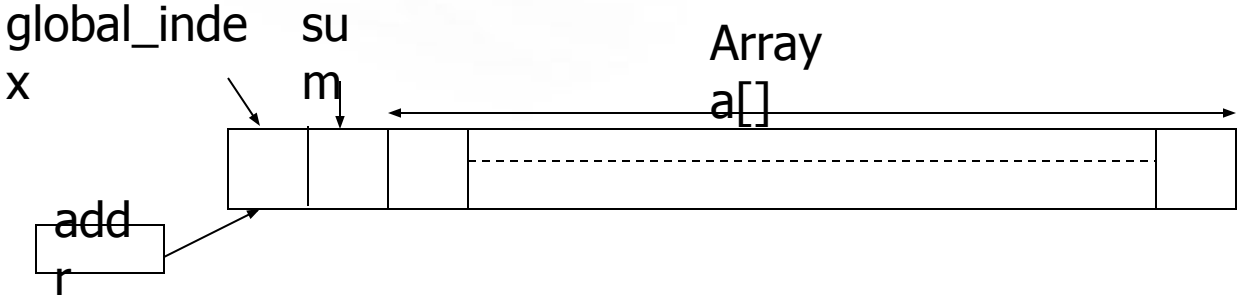
Exercício: Soma

- 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], NUL) != 0)
      {
         perror("Pthread_join falhou");
         exit(1);
      }

  printf("A soma é %d \n", sum)
}
```

Acesso a Dados Compartilhados: Mutexes

- A biblioteca *pthreads* 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);
}
```

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

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

- A. S. Tanenbaum, "Sistemas Operacionais Modernos", 2a. Edição, Editora Prentice-Hall, 2003.
  - Seção 2.2
- Silberschatz A. G.; Galvin P. B.; Gagne G.; "Fundamentos de Sistemas Operacionais", 6a. Edição, Editora LTC, 2004.
  - Capítulo 5
- Deitel H. M.; Deitel P. J.; Choffnes D. R.; “Sistemas Operacionais”, 3ª. Edição, Editora Prentice-Hall, 2005
  - Capítulo 4