Aula 07 – Escalonamento e Threads

Norton Trevisan Roman
Clodoaldo Aparecido de Moraes Lima

26 de setembro de 2014

Escalonamento de Tempo Real

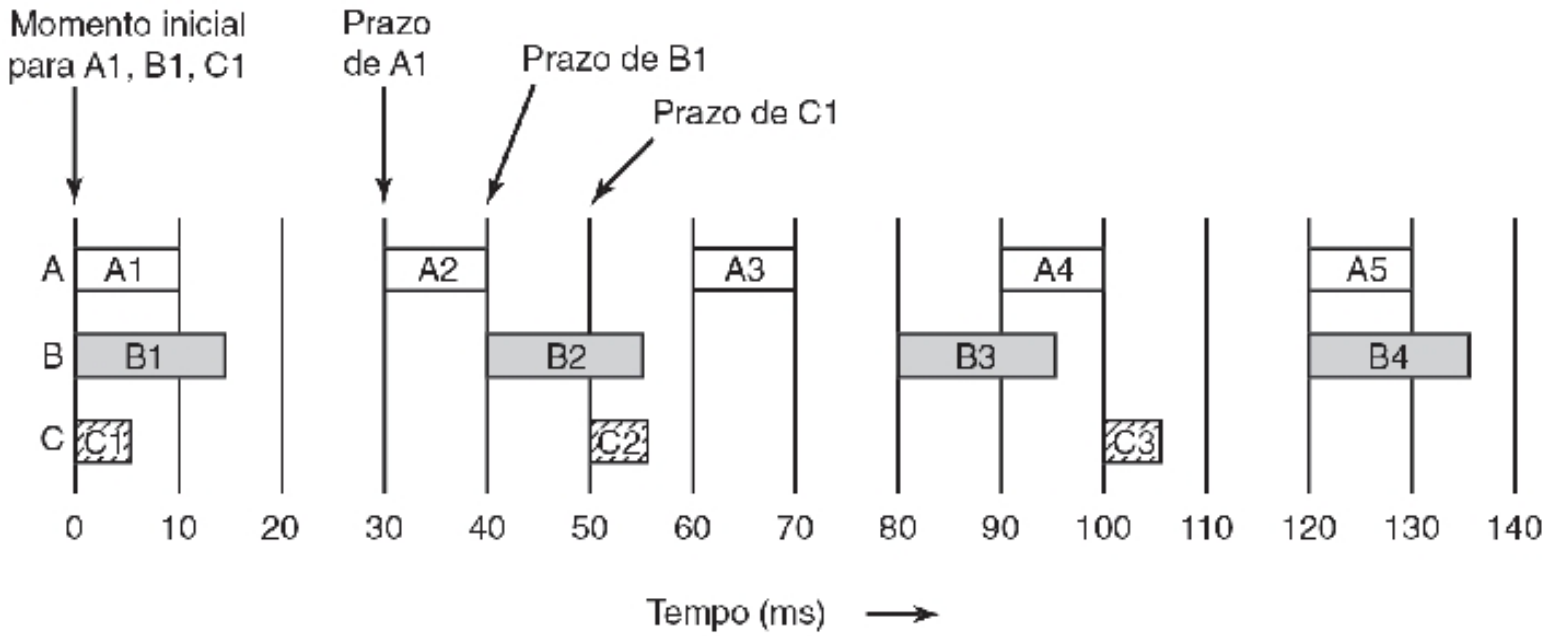
- Como escalonar processos periódicos de modo que seus prazos sejam cumpridos?
- Antes de mais nada, são escalonáveis?
  - Sim, se

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

onde m é o número de eventos periódicos, e o evento i ocorre com período P_i e requer C_i segundos de CPU para ser tratado ($C_i/P_i \rightarrow$ fração da CPU usada por i)

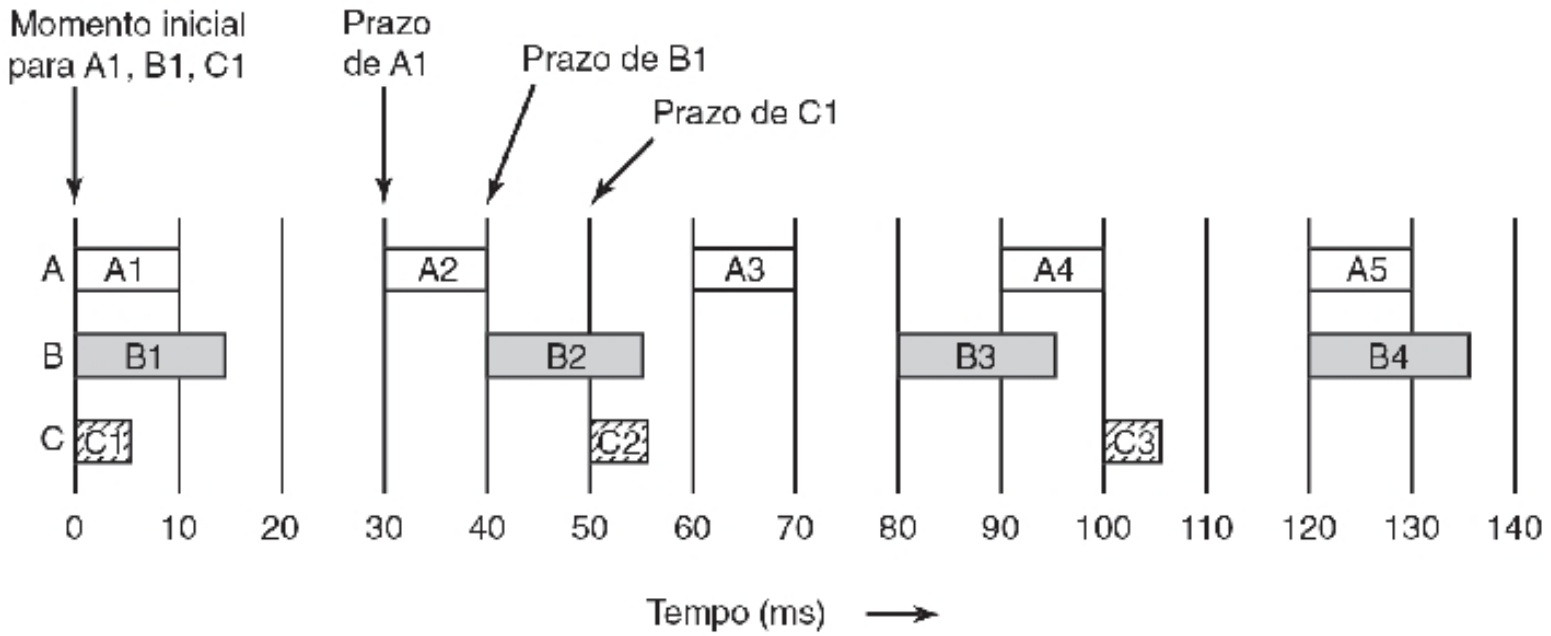
Escalonamento de Tempo Real

- Ex: O sistema abaixo é escalonável?



Escalonamento de Tempo Real

- Ex: O sistema abaixo é escalonável?



$$\frac{10}{30} + \frac{15}{40} + \frac{5}{50} = \frac{97}{120} < 1$$

Escalonamento de Tempo Real

- Algoritmos para STR podem ser:
  - Estáticos: decisões de escalonamento são tomadas antes do sistema começar a rodar
    - Necessita de informação disponível previamente sobre tarefas e prazos
    - Atribuem antecipadamente uma prioridade fixa a cada processo, e então escalonam
  - Dinâmicos: decisões de escalonamento tomadas em tempo de execução
    - Não apresentam prioridades fixas

Escalonamento de Tempo Real

- Qualquer que seja o algoritmo, pressupõe que se sabe:
  - Quanto trabalho deve ser feito
  - Qual seu prazo
- E, dependendo do algoritmo...
  - Que se sabe a frequência na qual cada processo deve executar

Escalonamento de Tempo Real

Rate Monotonic Scheduling

- Algoritmo de escalonamento estático
  - Útil para processos preemptivos e periódicos
- Condições:
  - Cada processo deve terminar dentro de seu período
  - Nenhum processo é dependente de outro
  - A cada surto de processamento, um mesmo processo precisa da mesma quantidade de tempo de CPU
  - Processos não periódicos não podem ter prazos
  - A preempção ocorre instantaneamente e sem sobrecargas (aproximadamente)

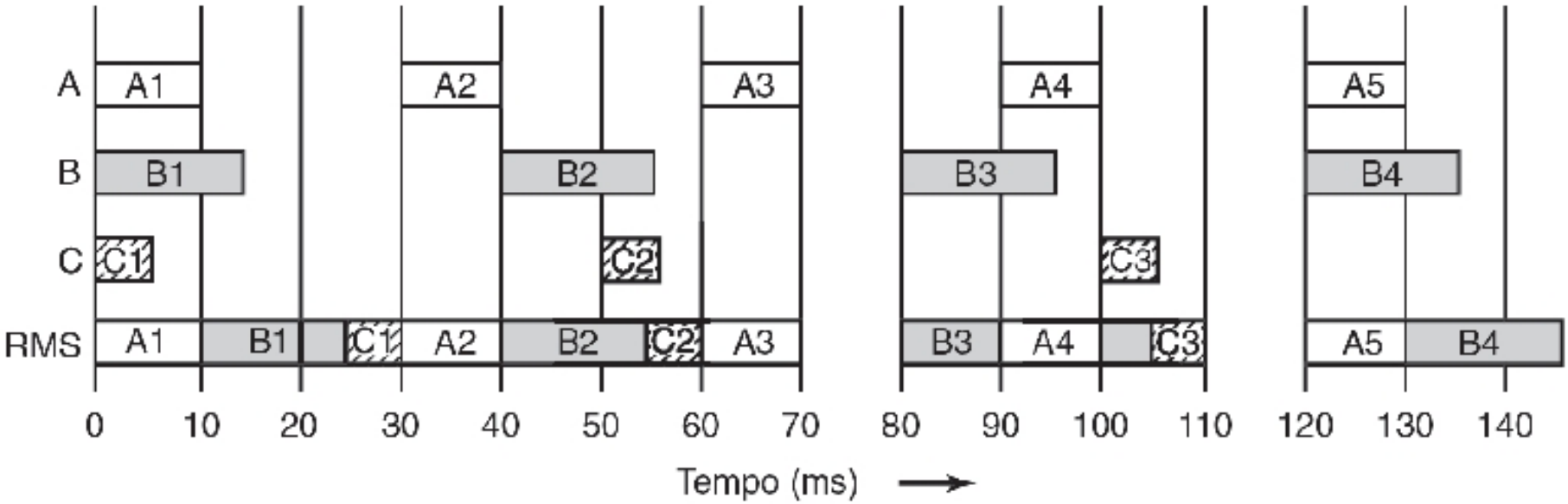
Escalonamento de Tempo Real

Rate Monotonic Scheduling

- Atribua a cada processo uma prioridade fixa igual à frequência de ocorrência de seu evento de disparo
  - Ex: Processos que executam a cada 30ms (ou seja, 33 vezes/s) recebem prioridade 33
- Em tempo de execução, execute o processo que estiver pronto e com maior prioridade
  - Fazendo a preempção do processo em execução se necessário (se esse for de menor prioridade)
  - O de maior prioridade dentre todos acaba nunca interrompido em seu surto

Escalonamento de Tempo Real

Rate Monotonic Scheduling



Assim que ficar pronto, A pode interromper B ou C
B pode interromper C, mas B não pode interromper A

C só roda se a CPU ficar ociosa

Escalonamento de Tempo Real

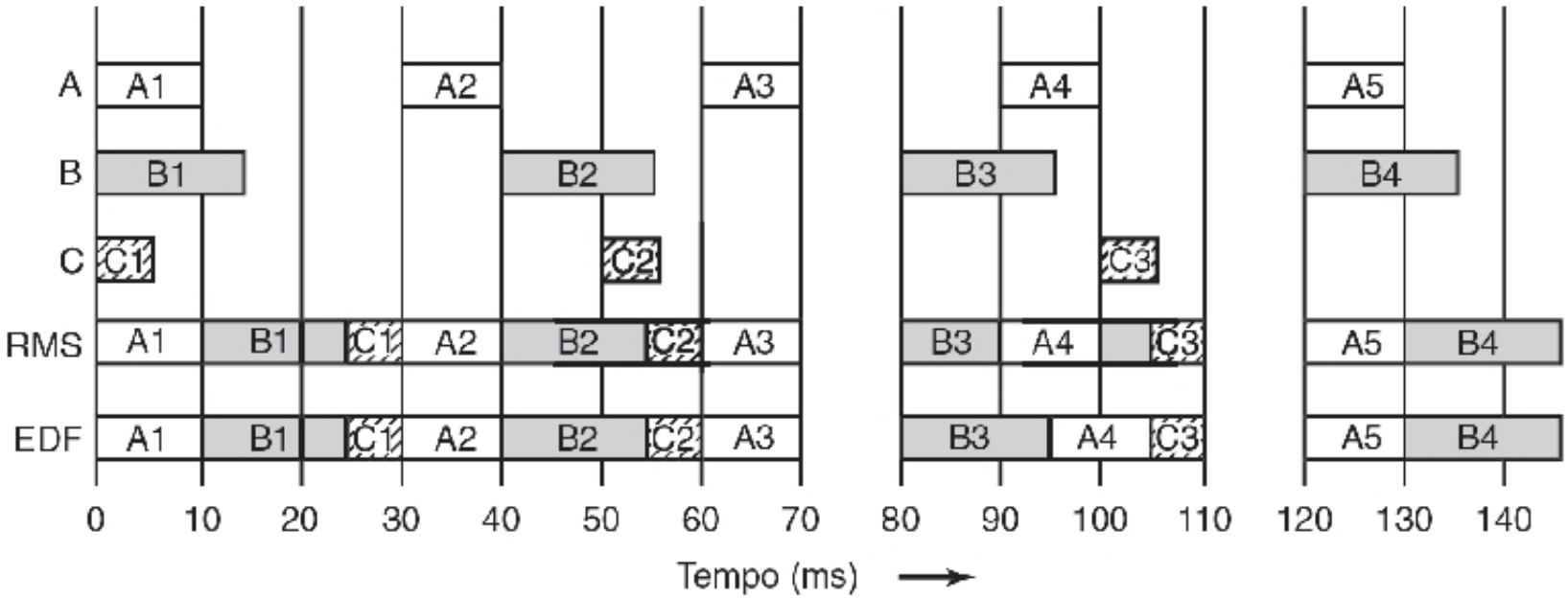
Earliest Deadline First

- Algoritmo para escalonamento dinâmico
  - Não requer que os processos sejam periódicos
  - Nem que tenham o mesmo tempo de execução por surto de CPU
- Se precisar de tempo de CPU, o processo avisa sua presença e seu prazo para obter a CPU
  - O escalonador tem uma lista de processos prontos, em ordem de vencimento de prazo
  - Executa sempre o primeiro da lista (menor prazo a vencer)

Escalonamento de Tempo Real

Earliest Deadline First

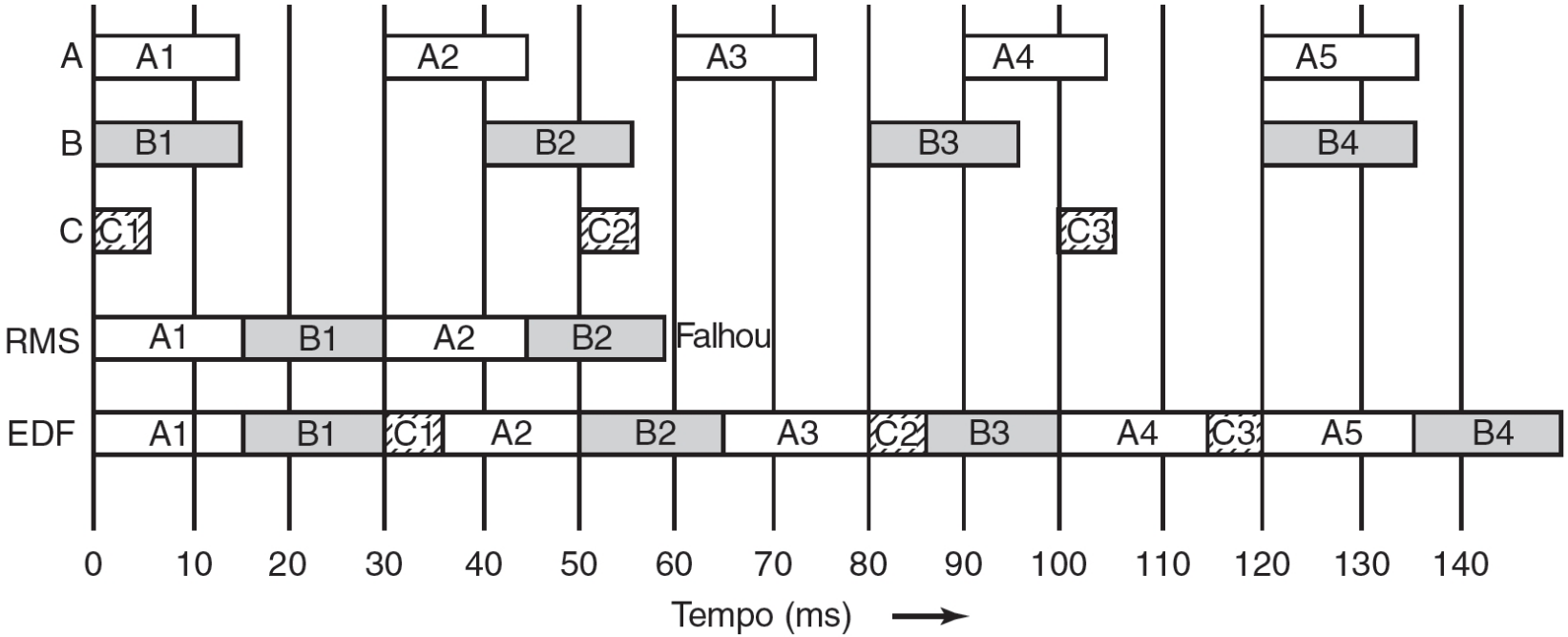
- Se um novo processo fica pronto, o escalonador vê se seu prazo vence antes do prazo do processo em execução
  - Se sim, faz a preempção do que estiver executando



Em 0, A tem prioridade, pois reinicia antes (prazo menor)
Em 90, A fica pronto. Tanto A quanto B reiniciam em 120. Como B já está rodando, fica.

Escalonamento de Tempo Real

Comparação



Em 30, há disputa entre A2 e C1. Como o prazo de C1 vence em 50 e de A2 em 60, C1 vence. No caso do RMS, a preempção de A mata C, que tem seu prazo estourado

Em 90, A fica pronto novamente. Como seu prazo é igual a B, e B está rodando, ele fica.

Processo × Thread

Threads

Processo × Thread

- Processo
  - Um único espaço de endereço e uma única linha de controle (thread), representada pelo PC, pilha de execução e demais registradores
  - O Modelo do Processo
    - Usados para agrupar recursos
    - Ex: espaço de endereço com texto, dados e pilha de execução do programa; arquivos abertos, processos filhos, tratadores de sinais, alarmes pendentes etc
    - Processos diferentes correspondem a tarefas diferentes (essencialmente não correlacionadas)

Processo × Thread

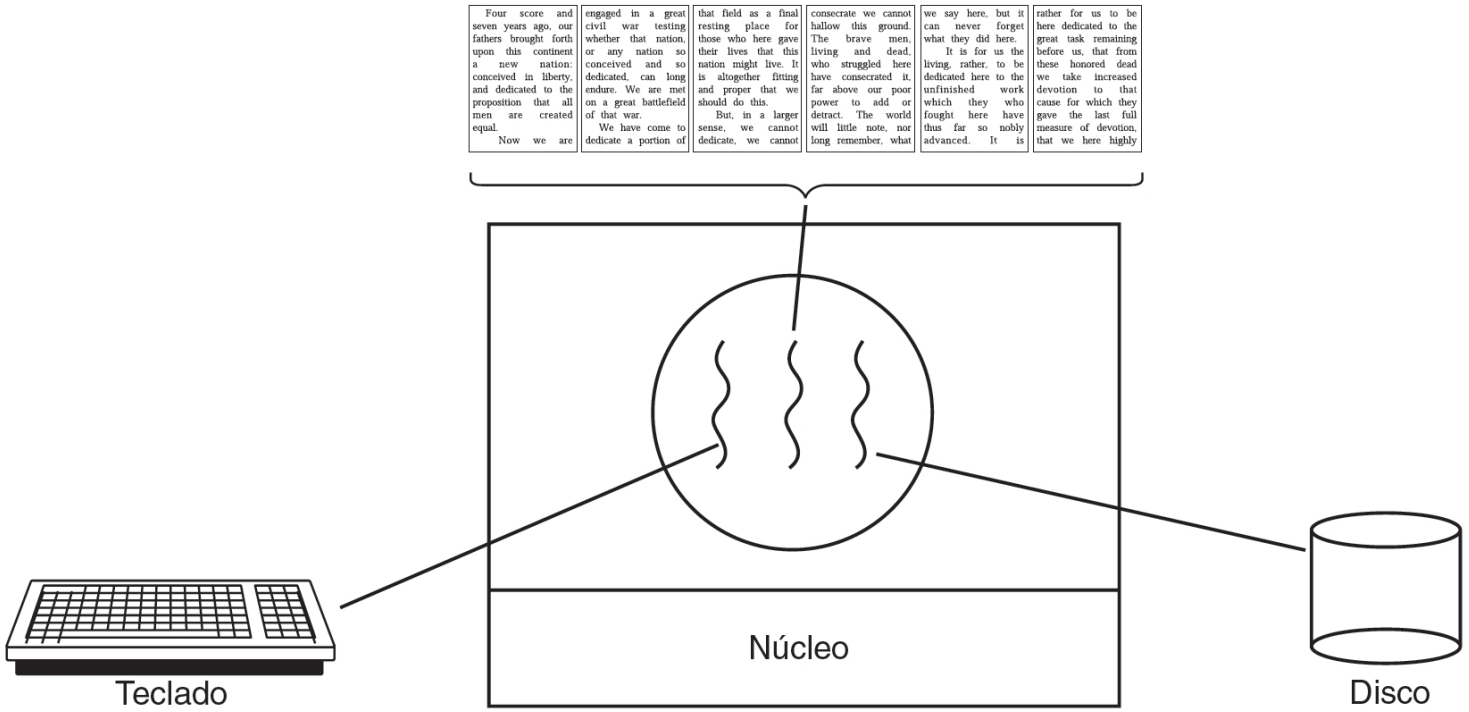
- Threads
  - São as entidades escalonadas para execução na CPU
    - Um espaço de endereço e múltiplas linhas de controle
    - Consistem numa linha ou contexto de execução
  - O Modelo da Thread
    - Subtarefas de uma mesma tarefa, cooperando umas com as outras
    - Permitem múltiplas execuções no mesmo ambiente do processo – com grande independência entre elas
    - Threads compartilham um mesmo espaço de endereço (sendo menos independentes que processos)
    - Possuem recursos particulares (PC, registradores, pilha)

Threads – Vantagens

- Em muitas aplicações há múltiplas atividades ao mesmo tempo
  - Podemos decompô-las em atividades paralelas
  - Algumas tarefas precisam do compartilhamento do espaço de endereçamento
  - CPU-bound e I/O-bound podem se sobrepor, acelerando a aplicação
- São mais rápidas de criar e destruir que processos
  - Algumas vezes até 100 vezes mais rápidas
- Úteis em sistemas com múltiplas CPUs $\rightarrow$ paralelismo real

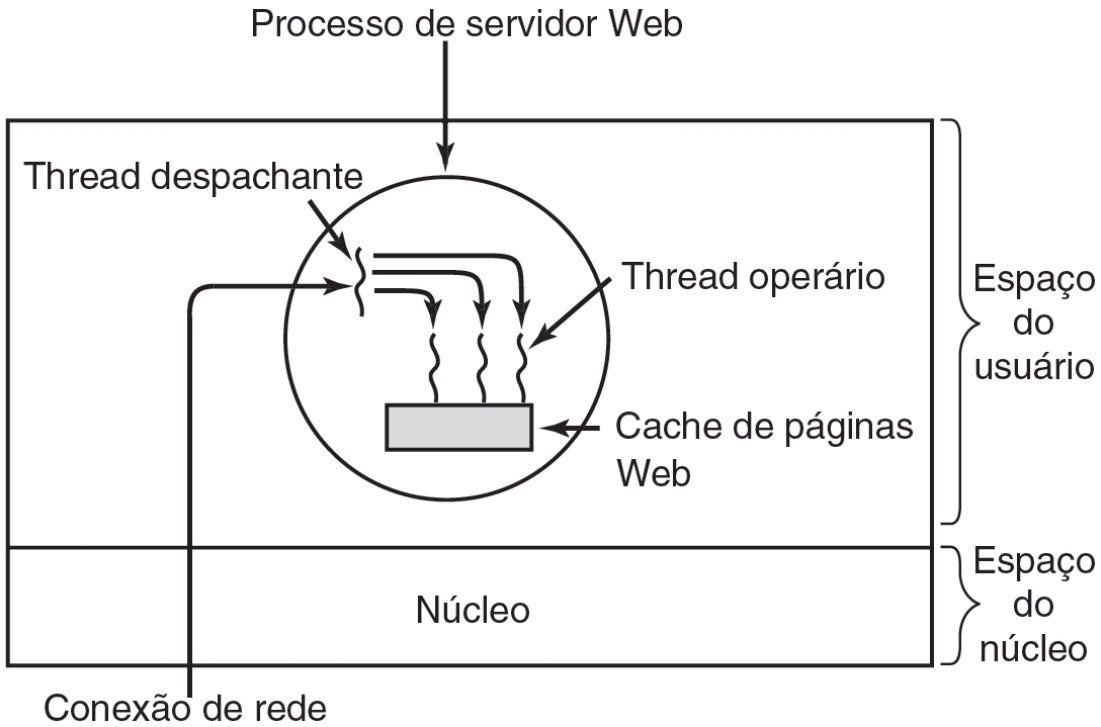
Threads – Exemplos

- Processador de texto:
  - Processos separados não funcionam – o documento tem que estar compartilhado



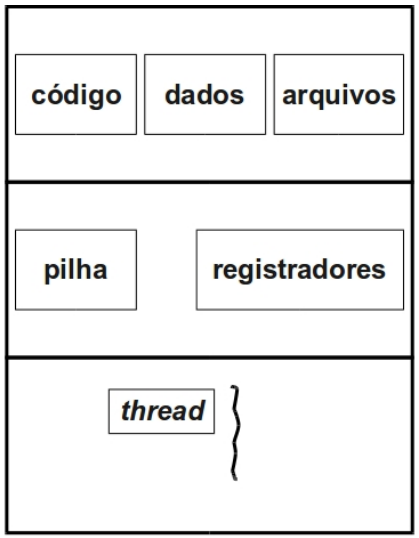
Threads – Exemplos

- Servidor web:

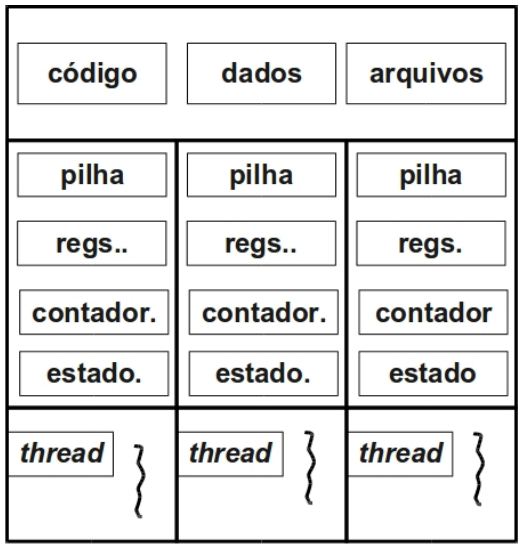


O despachante lê as requisições de trabalho que chegam da rede, escolhe uma thread operario ociosa e entrega a requisição. A thread operario lê a cache, caso não encontre a informação buscada, inicializa uma leitura de disco

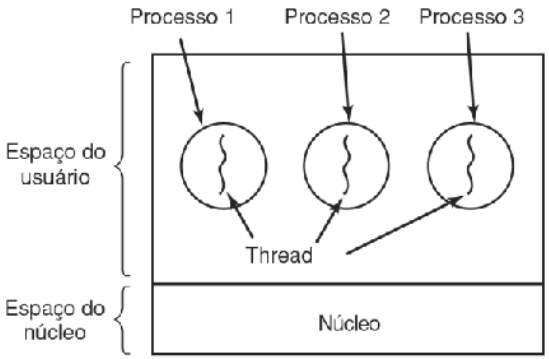
Processo × Thread



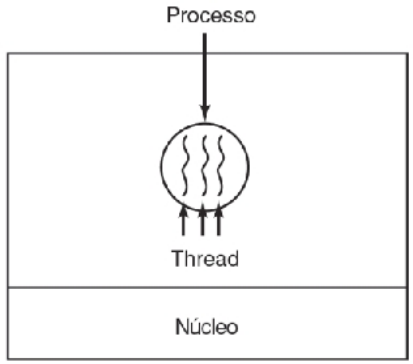
Processo com uma única *thread*



Processo com várias *threads*



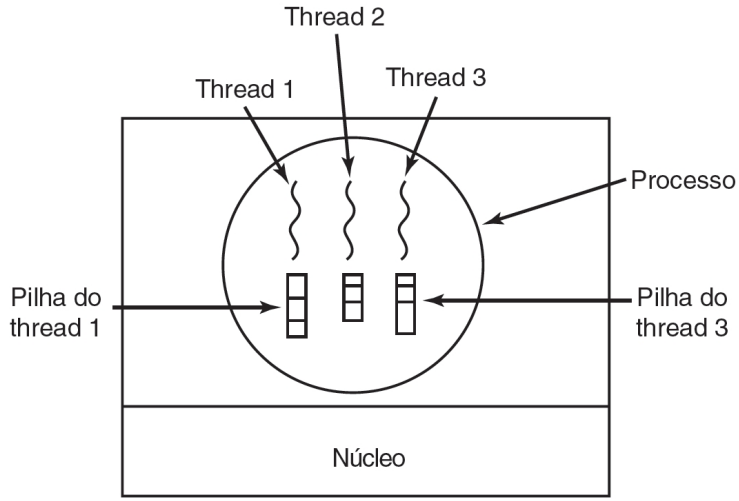
Três processos tradicionais (cada um com uma thread)



Um processo com três threads (multithread)

Processo × Thread

Itens por processo	Itens por thread
Espaço de endereçamento	Contador de programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos filhos	Estado
Alarmes pendentes	
Sinais e manipuladores de sinais	
Informação de contabilidade	



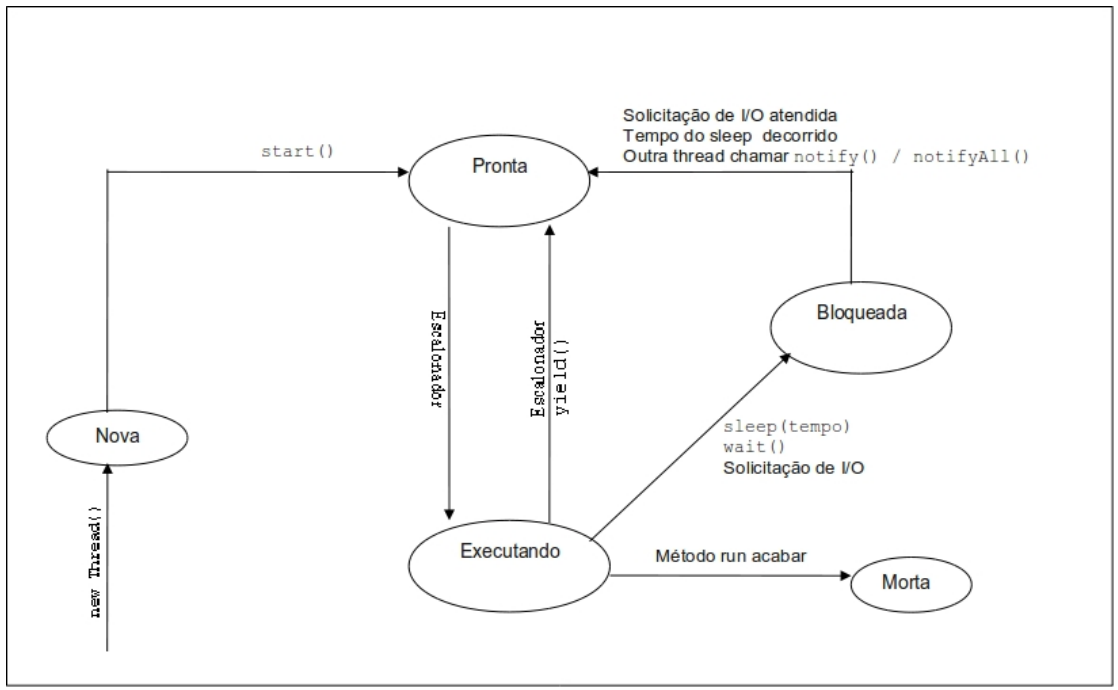
Cada thread tem sua própria pilha de execução (pois chamam rotinas diferentes), embora compartilhe o espaço de endereçamento e todos seus dados

Threads – Cuidados

- Não há proteção entre threads
  - Como cada thread pode ter acesso a qualquer endereço de memória dentro do espaço de endereçamento do processo, uma thread pode ler, escrever ou apagar a pilha ou as variáveis globais de outra thread
    - Raramente um problema, já que fazem parte do mesmo processo, com o mesmo dono
  - Há também a necessidade de sincronizá-las

Threads – Estados

- Assim como processos, threads também podem estar no estado *executando*, *bloqueado* ou *pronto*
- Mudanças de estado:



Threads – Manipulação

- IEEE 1003.1c – pacote Pthreads do POSIX (1995)
- Ex:

Chamada de thread	Descrição
pthread_create	Cria um novo thread
pthread_exit	Conclui a chamada de thread
pthread_join	Espera que um thread específico seja abandonado
pthread_yield	Libera a CPU para que outro thread seja executado
pthread_attr_init	Cria e inicializa uma estrutura de atributos do thread
pthread_attr_destroy	Remove uma estrutura de atributos do thread

Processos × Threads – Manipulação

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

int main(int argc, char *argv[]) {
  int pid;
  pid = fork();

  /* Ocorreu um erro */
  if (pid < 0) {
    fprintf( stderr, "Erro ao criar processo" );
  }

  /* Processo filho */
  else if (pid == 0) {
    execlp ("/bin/ls", "ls", NULL);
  }

  /* Processo pai */
  else if (pid > 0) {
    wait (NULL);
    printf ("Processo Pai terminou.\n");
  }
}
```

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

#define NUMERO_DE_THREADS 10

void *ola_mundo(void *tid) {
  printf("Ola mundo. Saudacoes da thread %d\n",tid);
  pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
  pthread_t threads[NUMERO_DE_THREADS];
  int status, i;

  for (i=0; i<NUMERO_DE_THREADS; i++) {
    printf("Thread principal. Criando thread %d\n",i);
    status = pthread_create(&threads[i], NULL,
                                    ola_mundo, (void *)i);
    if (status != 0) {
      printf("pthread_create retornou o codigo de erro
                                          %d\n",status);
      exit(-1);
    }
  }
  exit(NULL);
}
```

Threads em Java

- Estendendo a class Thread
  - Criar thread: Escrever classe que deriva da classe 'Thread'
  - 'Thread' possui todo o código para criar e executar threads

```
class ThreadSimples extends Thread {

    public void run() {
        System.out.println("Ola de uma
                            nova thread!");
    }

    public static void main(String
                                  args[]) {
     Thread thread =
                 new ThreadSimples();
     thread.start();

        System.out.println("Ola da
                     thread original!");
    }
}
```

Threads em Java

- Implementando Runnable
  - Vantagem: A classe que implementa Runnable pode estender outra classe

```
class RunnableSimples
                    implements Runnable {

    public void run() {
        System.out.println("Ola de um
                        novo Runnable!");
    }

    public static void main(
                         String args[]) {
     RunnableSimples runnable =
               new RunnableSimples();
     Thread thread =
                new Thread(runnable);
     thread.start();

        System.out.println("Ola da
                    thread original!");
    }
}
```

Threads em Java

- Joining a Thread
  - Permite a uma thread esperar que outra termine
  - A thread principal esperará *thread2* morrer

```
class ThreadSimples extends Thread {

   public void run() {
      System.out.println("Ola de uma nova thread! "
                         + super.toString() + ".");
   }

   public static void main(String args[]) {
      ThreadSimples thread = new ThreadSimples();
      ThreadSimples thread2 = new ThreadSimples();

      thread.start();
      thread2.start();

      try {
         thread2.join();
      }
      catch (InterruptedException e) {
         e.printStackTrace();
      }

      System.out.println("Ola da thread original!");
   }
}
```

Threads em Java

- Sleeping a Thread
  - A thread atual fica bloqueada por um número de milisegundos
  - Precisa capturar InterruptedException

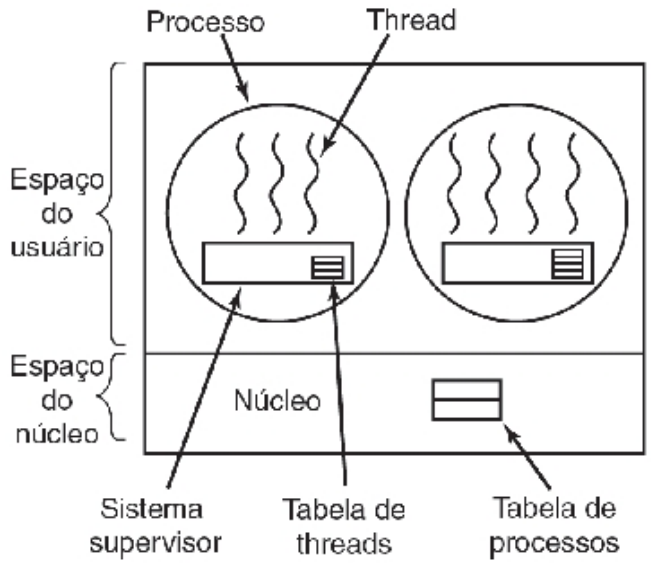
```
try {
   Thread.sleep(1);
}
catch (InterruptedException e) {
   e.printStackTrace();
}
```

Threads

- Modos de Implementação
  - No espaço do usuário
  - No espaço do núcleo
  - Nos dois (híbridas)
- Threads no Espaço do Usuário
  - Implementadas totalmente no espaço do usuário
    - Por meio de uma biblioteca (criação, exclusão, execução etc, não necessariamente gerenciamento)
  - Criação e escalonamento são realizados sem o conhecimento do kernel

Threads no Espaço do Usuário

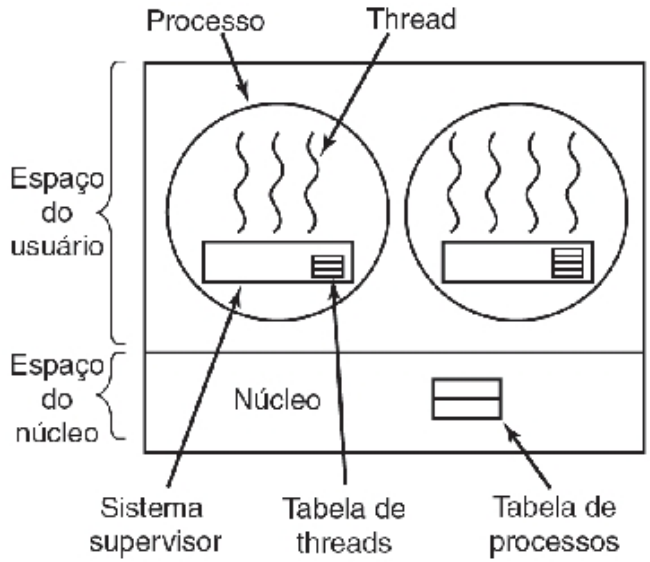
- Para o kernel, é como se rodasse um programa monothread
  - Gerenciadas como processos são gerenciados no S.O. (por uma tabela)



- Necessitam de um sistema supervisor
  - Runtime environment

Threads no Espaço do Usuário

- Cada processo possui sua própria tabela de threads
  - Como uma tabela de processos, gerenciada pelo *runtime*
  - Controla apenas as propriedades da thread (PC, ponteiro da pilha, registradores, estado etc)
    - Quando uma thread vai para o estado de pronto ou bloqueado, a informação necessária para trazê-la de volta é armazenada na tabela de threads

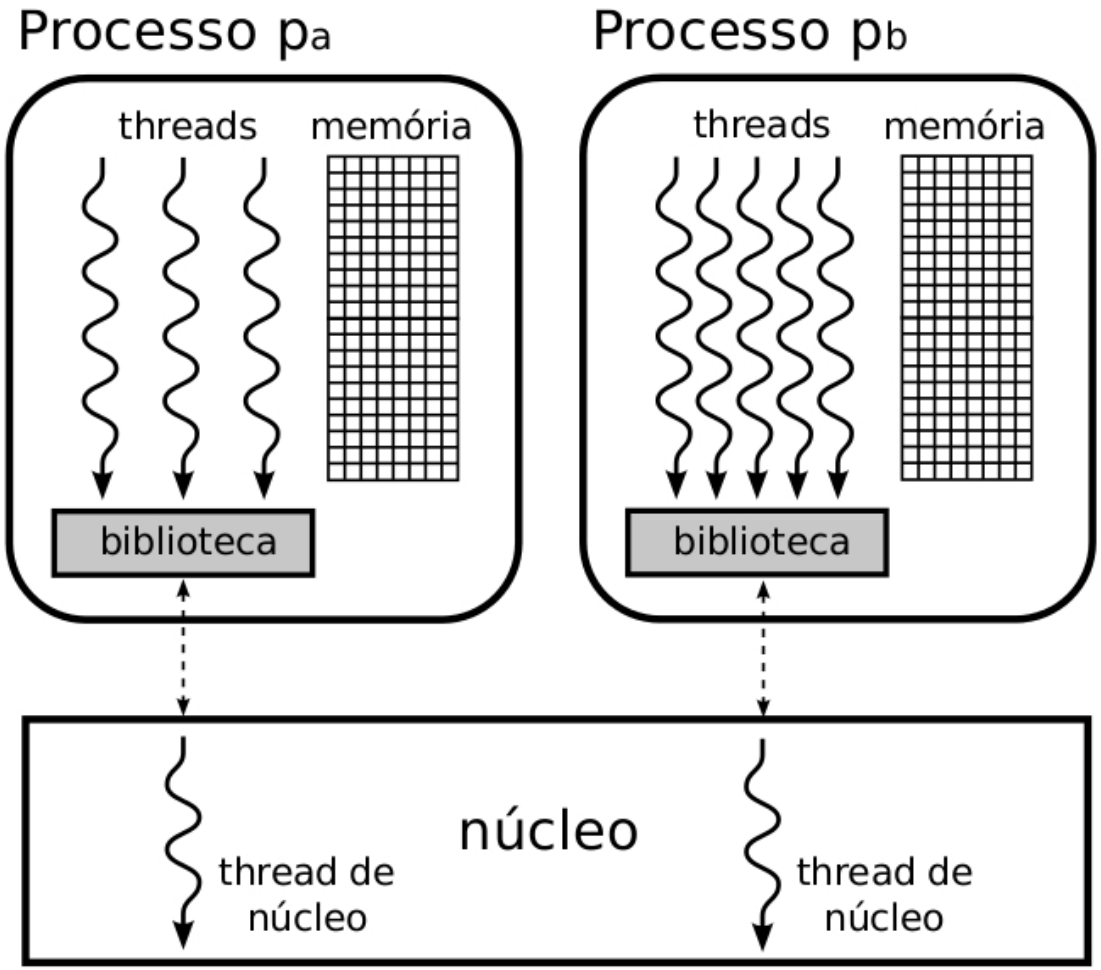


Threads no Espaço do Usuário

- Seguem o modelo N para 1:
  - N threads no processo, mapeadas em uma única thread de núcleo
  - Não necessariamente permite múltiplas threads (no modo núcleo) em paralelo
  - O núcleo do sistema divide o tempo do processador entre as threads de núcleo
    - Uma aplicação com 100 threads de usuário pode vir a receber o mesmo tempo de processador que outra aplicação com apenas uma thread $\rightarrow$ divisão injusta
  - Usado, dentre outras coisas quando não há suporte a multithread no núcleo

Threads no Espaço do Usuário

- N para 1:



Threads no Espaço do Usuário

- O bloqueamento local (espera de outra thread, não E/S) é feito por um procedimento do runtime environment
  - Chamado pela própria thread
  - Cuida de verificar se a thread pode ser parada, de armazenar registradores na tabela de threads, e ver que outra thread pode rodar
  - Não há desvio de controle para o núcleo
    - Mais rápido que a alternância de processos

Threads no Espaço do Usuário

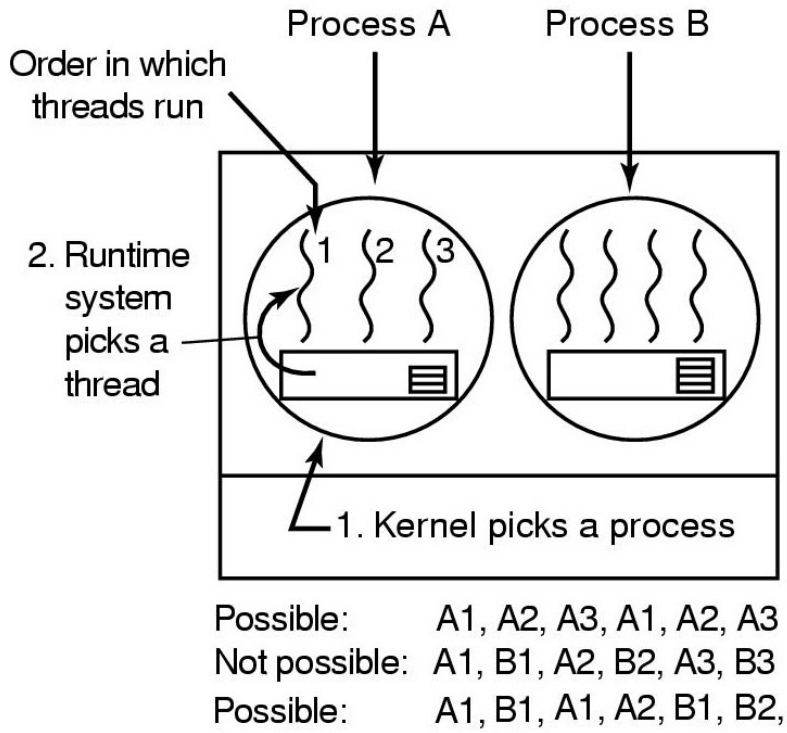
- Outras vantagens:
  - Podem ser implementadas mesmo quando o S.O. não suporta threads
  - Permite que cada processo possa ter seu próprio algoritmo de escalonamento (que seja ideal para o problema abordado)
  - Modelo ainda usado em jogos e simulações
- Problemas:
  - Processo inteiro é bloqueado se uma thread realizar uma chamada bloqueante ao sistema
    - Vai contra o uso principal delas, que é fazer alguma outra coisa enquanto espera por E/S

Threads no Espaço do Usuário

- Problemas:
  - Em um único processo, não há interrupções de relógio (que causariam uma troca de processo)
    - É impossível interromper uma thread para escalonar
  - Assim, quando o escalonador retorna o processo, a thread que estava rodando continua a rodar
    - Não há possibilidade de escolha
  - Se uma thread executa, nenhuma outra naquele processo executará, a menos que a primeira abra mão da CPU

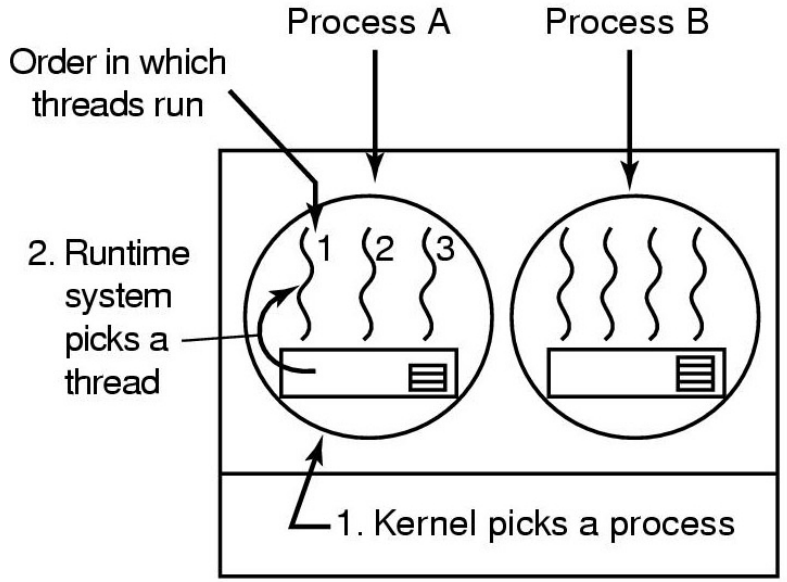
Threads no Espaço do Usuário

- Escalonamento:
  - O núcleo escolhe um processo e passa o controle a ele
  - O escalonador do processo (uma thread) decide qual thread executar (se ele estiver rodando)



Threads no Espaço do Usuário

- Escalonamento:
  - O núcleo escolhe um processo e passa o controle a ele
  - O escalonador do processo (uma thread) decide qual thread executar (se ele estiver rodando)



A sequência do meio é impossível porque, após interromper A (e consequentemente A1), o escalonador, ao voltar a A, continua rodando A1 (ele desconhece as threads e A1 não "largou o osso" ainda).

Threads no Espaço do Usuário

- Escalonamento:
  - Como não há interrupções do clock para interromper threads, a thread continua enquanto quiser
  - Ao fim do quantum, o núcleo seleciona outro processo