Sistemas Operacionais

Prof. Jó Ueyama

Apresentação baseada nos slides da Profa. Dra. Kalinka Castelo Branco, do Prof. Dr. Antônio Carlos Sementille e da Profa. Dra. Luciana A. F. Martimiano e nas transparências fornecidas no site de compra do livro "Sistemas Operacionais Modernos"

Exemplo de Criação de Processo em C no Linux

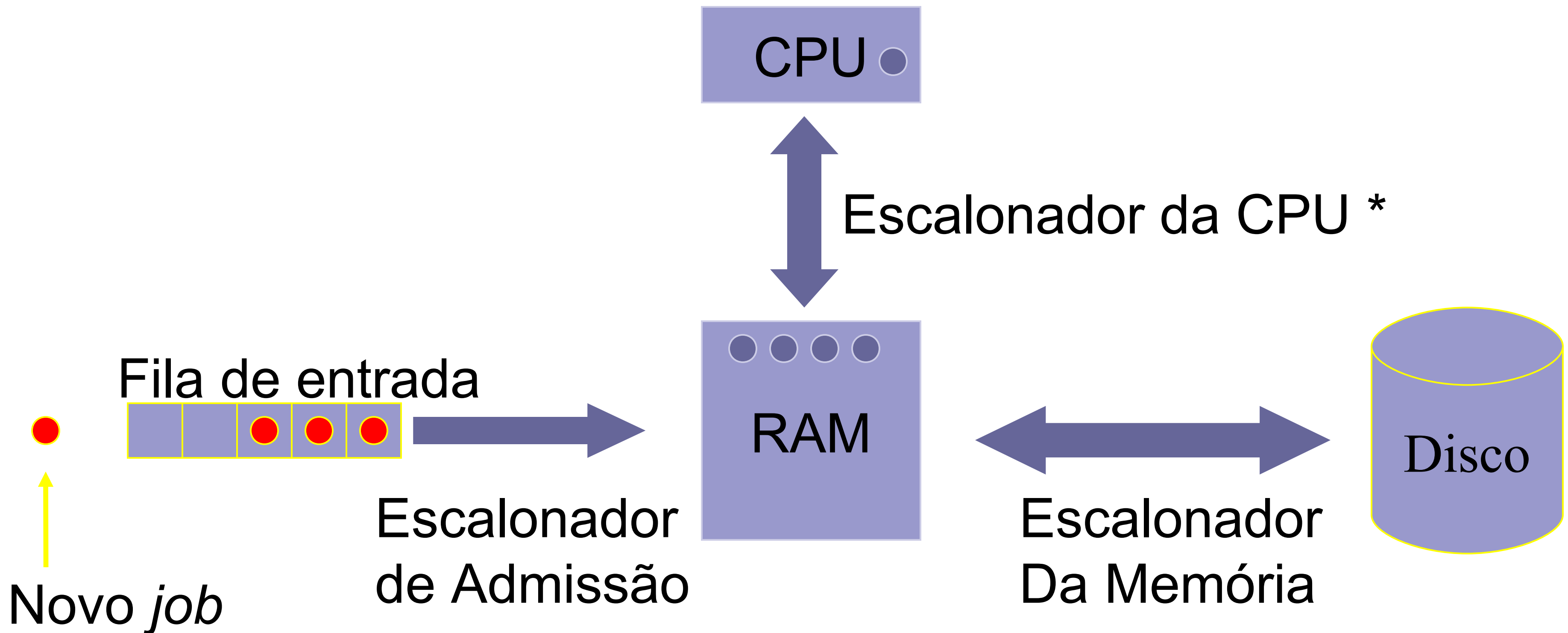
```
#include <stdio.h>
#include <unistd.h>

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 ("Sou o processo Pai.\n");
  }
}
```

Aula de Hoje (conteúdo detalhado)

1. **Escalonamento em Batch**
2. **Algoritmos de Escalonamento em Sistemas Batch**
3. **Algoritmos de Escalonamento em Sistemas Interativos**

Escalonamento de Processos Sistemas em *Batch*

- Escalonamento *Three-Level*



Escalonamento de Processos Sistemas em *Batch*

- Escalonamento *Three-Level*
  - **Escalonador de admissão**: processos menores primeiro; processos com menor tempo de acesso à CPU e maior tempo de interação com dispositivos de E/S;
  - **Escalonador da Memória**: decisões sobre quais processos vão para a MP:
    - A quanto tempo o processo está esperando?
    - Quanto tempo da CPU o processo já utilizou?
    - Qual o tamanho do processo?
    - Qual a importância do processo?
  - **Escalonador da CPU**: seleciona qual o próximo processo a ser executado;

Aula de Hoje (conteúdo detalhado)

1. Escalonamento em Batch
2. Algoritmos de Escalonamento em Sistemas Batch
3. Algoritmos de Escalonamento em Sistemas Interativos

Escalonamento de Processos Sistemas em *Batch*

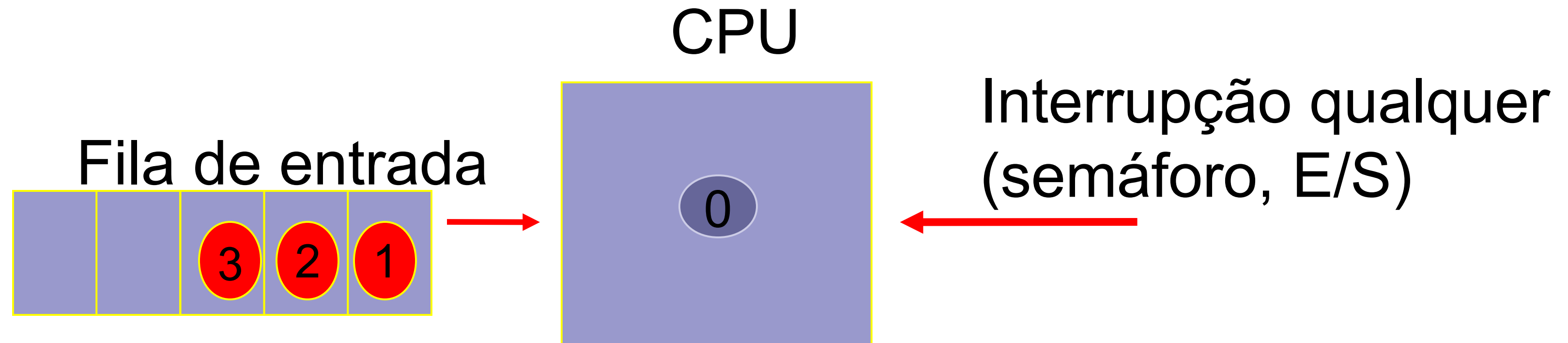
- Algoritmos para Sistemas em *Batch*:
  - *First-Come First-Served (ou FIFO)*;
  - *Shortest Job First (SJF)*;
  - *Shortest Remaining Time Next (SRTN)*;

Escalonamento de Processos
Sistemas em *Batch*

- Algoritmo *First-Come First-Served*
  - Não-preemptivo;
  - Processos são executados na CPU seguindo a ordem de requisição;
  - Fácil de entender e programar;
  - Desvantagem:
    - Ineficiente quando se tem processos que demoram na sua execução;

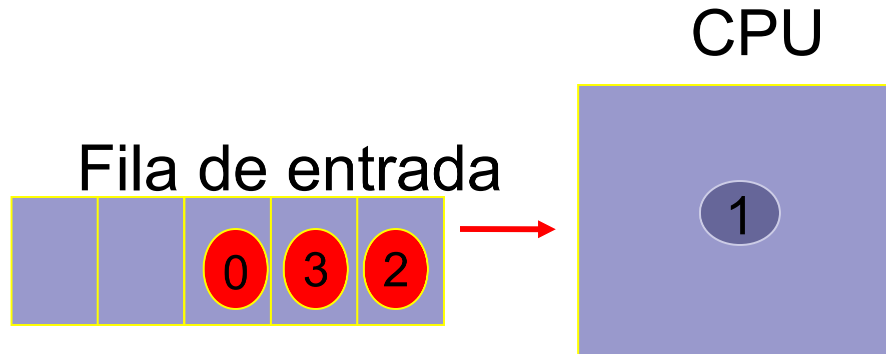
Escalonamento de Processos Sistemas em *Batch*

- Algoritmo *First-Come First-Served*



Escalonamento de Processos Sistemas em *Batch*

- Algoritmo *First-Come First-Served*



CPU não controla o tempo dos processos!
(não-preemptivo)

Escalonamento de Processos Sistemas em *Batch*

- Algoritmo *Shortest Job First*
  - Não-preemptivo;
  - Possível prever o tempo de execução do processo;
  - Menor processo é executado primeiro;
  - Menor *turnaround*;
  - Desvantagem:
    - Baixo aproveitamento quando se tem poucos processos prontos para serem executados;

Escalonamento de Processos

Sistemas em *Batch*

- Algoritmo *Shortest Job First*

A → a

B → b+a

C → c+b+a

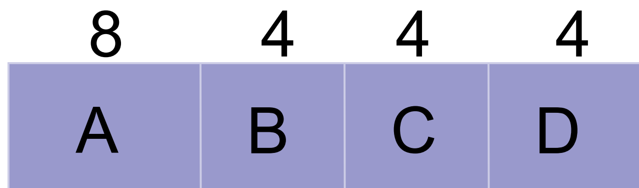
D → d+c+b+a

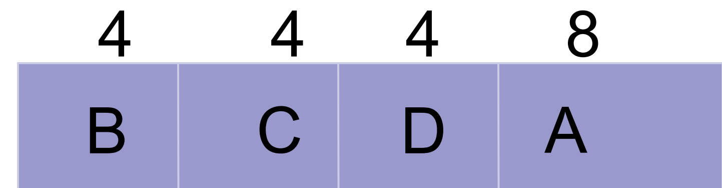
Tempo médio-*turnaround* (4a+3b+2c+d)/4

Contribuição → se $a<b<c<d$ tem-se o mínimo tempo médio;

Escalonamento de Processos Sistemas em *Batch*

- Algoritmo *Shortest Job First*





Em ordem:
Turnaround A = 8
Turnaround B = 12
Turnaround C = 16
Turnaround D = 20
Média → 56/4 = 14

Menor *job* primeiro:
Turnaround B = 4
Turnaround C = 8
Turnaround D = 12
Turnaround A = 20
Média → 44/4 = 11

(4a+3b+2c+d)/4 ← Número de Processos

Escalonamento de Processos Sistemas em *Batch*

- Algoritmo *Shortest Remaining Time Next*
  - Preemptivo;
  - Processos com menor tempo de execução são executados primeiro;
  - Se um processo novo chega e seu tempo de execução é menor do que do processo corrente na CPU, a CPU suspende o processo corrente e executa o processo que acabou de chegar;
  - <u>Desvantagem</u>: processos que consomem mais tempo podem demorar muito para serem finalizados se muitos processos pequenos chegarem!

Aula de Hoje (conteúdo detalhado)

1. **Escalonamento em Batch**
2. **Algoritmos de Escalonamento em Sistemas Batch**
3. **Algoritmos de Escalonamento em Sistemas Interativos**

Escalonamento de Processos Sistemas Interativos

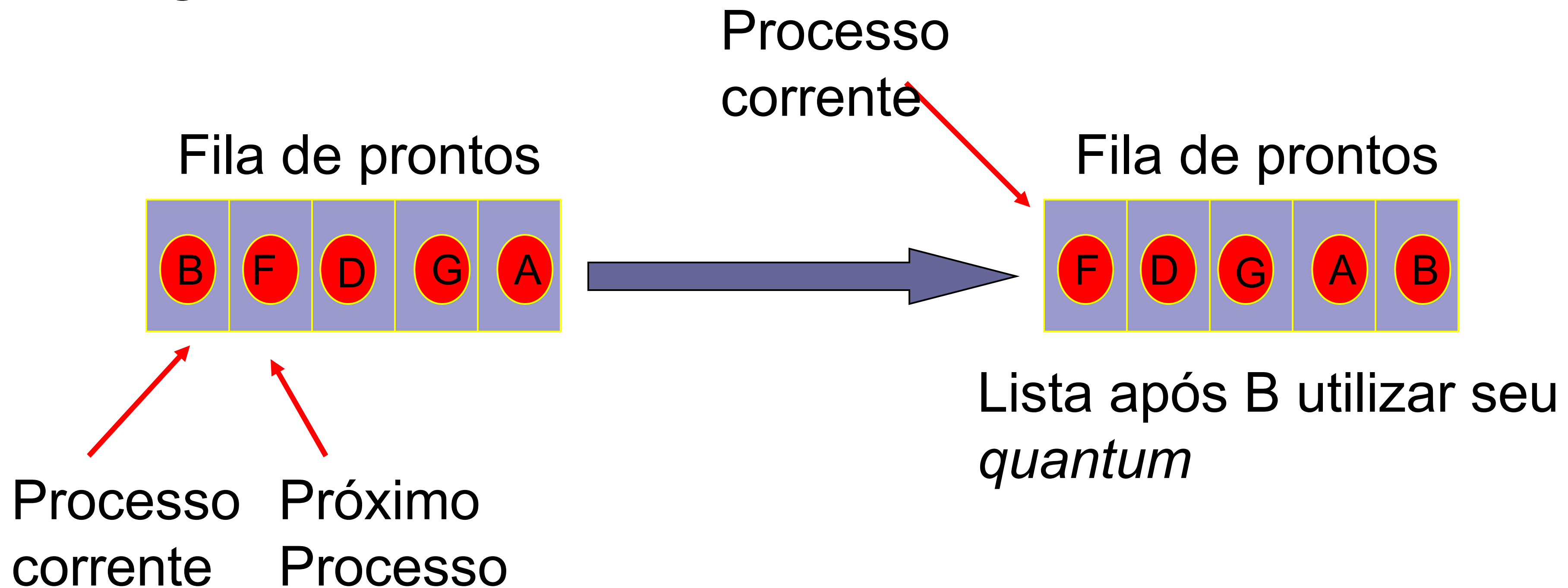
- Algoritmos para Sistemas Interativos:
  - *Round-Robin*;
  - Prioridade;
  - Múltiplas Filas;
  - *Shortest Process Next*;
  - Garantido;
  - *Lottery*;
  - *Fair-Share;*
- Utilizam escalonamento em dois níveis (escalonador da CPU e memória);

Escalonamento de Processos Sistemas Interativos

- Algoritmo *Round-Robin*
  - Antigo, mais simples e mais utilizado;
  - Preemptivo (quantum, I/O, system call pelo processo);
  - Cada processo recebe um tempo de execução chamado *quantum*; ao final desse tempo, o processo é suspenso e outro processo é colocado em execução;
  - Escalonador mantém uma lista de processos prontos;

Escalonamento de Processos Sistemas Interativos

- Algoritmo *Round-Robin*



Escalonamento de Processos Sistemas Interativos

- Algoritmo *Round-Robin*
  - Tempo de chaveamento de processos;
  - *quantum*: se for muito pequeno, ocorrem muitas trocas diminuindo, assim, a eficiência da CPU; se for muito longo o tempo de resposta é comprometido;

Escalonamento de Processos Sistemas Interativos

- Algoritmo *Round-Robin:*

Exemplos:

Δt = 4 mseg

x = 1mseg → 25% de tempo de CPU é perdido → menor eficiência

Δt = 100 mseg

x = 1mseg → 1% de tempo de CPU é perdido → Tempo de espera dos processos é maior

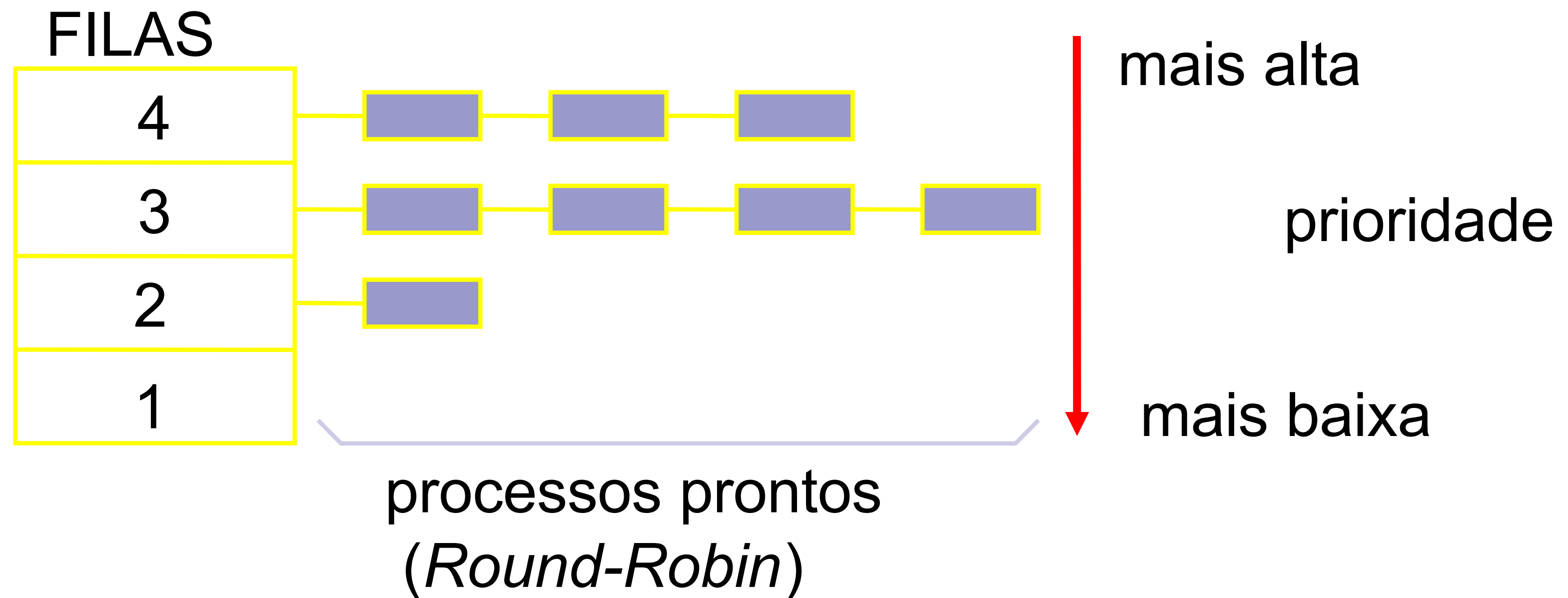
quantum razoável: 20-50 mseg

Escalonamento de Processos Sistemas Interativos

- Algoritmo com Prioridades
  - Cada processo possui uma prioridade → os processos prontos com maior prioridade são executados primeiro;
  - Prioridades são atribuídas dinâmica ou estaticamente;
  - Classes de processos com mesma prioridade;
  - Preemptivo;

Escalonamento de Processos Sistemas Interativos

- Algoritmo com Prioridades



Escalonamento de Processos Sistemas Interativos

- Algoritmo com Prioridades
  - Como evitar que os processos com maior prioridade sejam executado indefinidamente?
    - Diminuir a prioridade do processo corrente e troca-lo pelo próximo processo com maior prioridade (chaveamento);
    - Cada processo possui um *quantum*;

Escalonamento de Processos Sistemas Interativos

- Múltiplas Filas:
  - CTSS (*Compatible Time Sharing System*);
  - Classes de prioridades;
  - Cada classe de prioridades possui *quanta* diferentes;
  - Assim, a cada vez que um processo é executado e suspenso ele recebe mais tempo para execução;
  - Preemptivo;

Escalonamento de Processos Sistemas Interativos

- Múltiplas Filas:
  - Ex.: um processo precisa de 100 *quanta* para ser executado;
    - Inicialmente, ele recebe um *quantum* para execução;
    - Das próximas vezes ele recebe, respectivamente, 2, 4, 8, 16, 32 e 64 *quanta* (7 chaveamentos) para execução;
    - Quanto mais próximo de ser finalizado, menos frequente é o processo na CPU → eficiência

Escalonamento de Processos Sistemas Interativos

- Sistemas Linux e Windows
- Multilevel feedback queue
  - Dá preferência a processos curtos
  - Dá prioridades para processos I/O bound
  - Estuda o processo e escalona de acordo com o estudo
  - Cada fila usa um escalonamento round-robin
  - Os I/Os promovem o processos para as filas com maior prioridade

Escalonamento de Processos Sistemas Interativo

- Algoritmo *Shortest Process Next*
  - Mesma idéia do *Shortest Job First*;
  - Processos Interativos: não se conhece o tempo necessário para execução;
  - Como empregar esse algoritmo: ESTIMATIVA de TEMPO!
  - Verficar o comportamento passado do processo e estimar o tempo.

Escalonamento de Processos Sistemas Interativo

- Outros algoritmos:
  - Algoritmo Garantido:
    - Garantias são dadas aos processos dos usuários:
      - **n** usuários → **1/n** do tempo de CPU para cada usuário;
  - Algoritmo *Lottery*:
    - Cada processo recebe "*tickets*" que lhe dão direito de execução;

Escalonamento de Processos Sistemas Interativo

- Algoritmo *Fair-Share*:
  - O dono do processo é levado em conta;
  - Se um usuário A possui mais processos que um usuário B, o usuário A terá prioridade no uso da CPU;

Usuário 1 → A, B, C, D

  - Usuário 2 → E
  - Garantia de 50%

Circular → A, E, B, E, C, E, D, E

50% a mais para Usuário 1 → A, B, E, C, D, E

Escalonamento de Processos Sistemas em Tempo Real

- Tempo é um fator crítico; possui um deadline
- Sistemas críticos:
  - Aviões;
  - Hospitais;
  - Usinas Nucleares;
  - Bancos;
  - Multimídia;
- Ponto importante: obter respostas em atraso é tão ruim quanto não obter respostas;

Escalonamento de Processos Sistemas em Tempo Real

- Tipos de STR:
  - ***Hard Real Time***: atrasos não são tolerados;
    - Aviões, usinas nucleares, hospitais;
  - ***Soft Real Time***: atrasos são tolerados;
    - Bancos; Multimídia;
- Programas são divididos em vários processos;
- Eventos causam a execução de processos:
  - **Periódicos**: ocorrem em intervalos regulares de tempo;
  - **Aperiódicos**: ocorrem em intervalos irregulares de tempo;

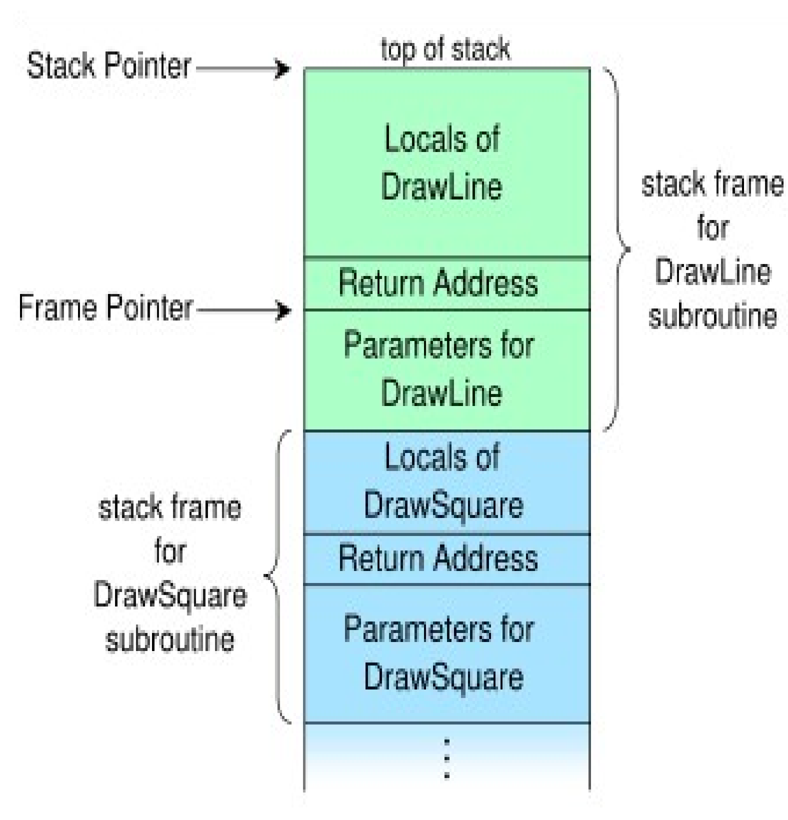
Escalonamento de Processos Sistemas em Tempo Real

- Algoritmos podem ser estáticos ou dinâmicos;
  - **Estáticos**: decisões de escalonamento antes do sistema começar;
    - Informação disponível previamente;
  - **Dinâmicos**: decisões de escalonamento em tempo de execução;

1. Threads (conceitos, motivações e tipos)

Antes disso...

O que é uma pilha de execução?

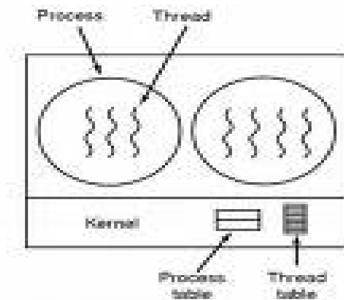


Conceito de Processo e Thread

O conceito de um Processo pode ser dividido em dois :

1: É um agrupamento de recursos necessários para a execução de um programa. Por exemplo :

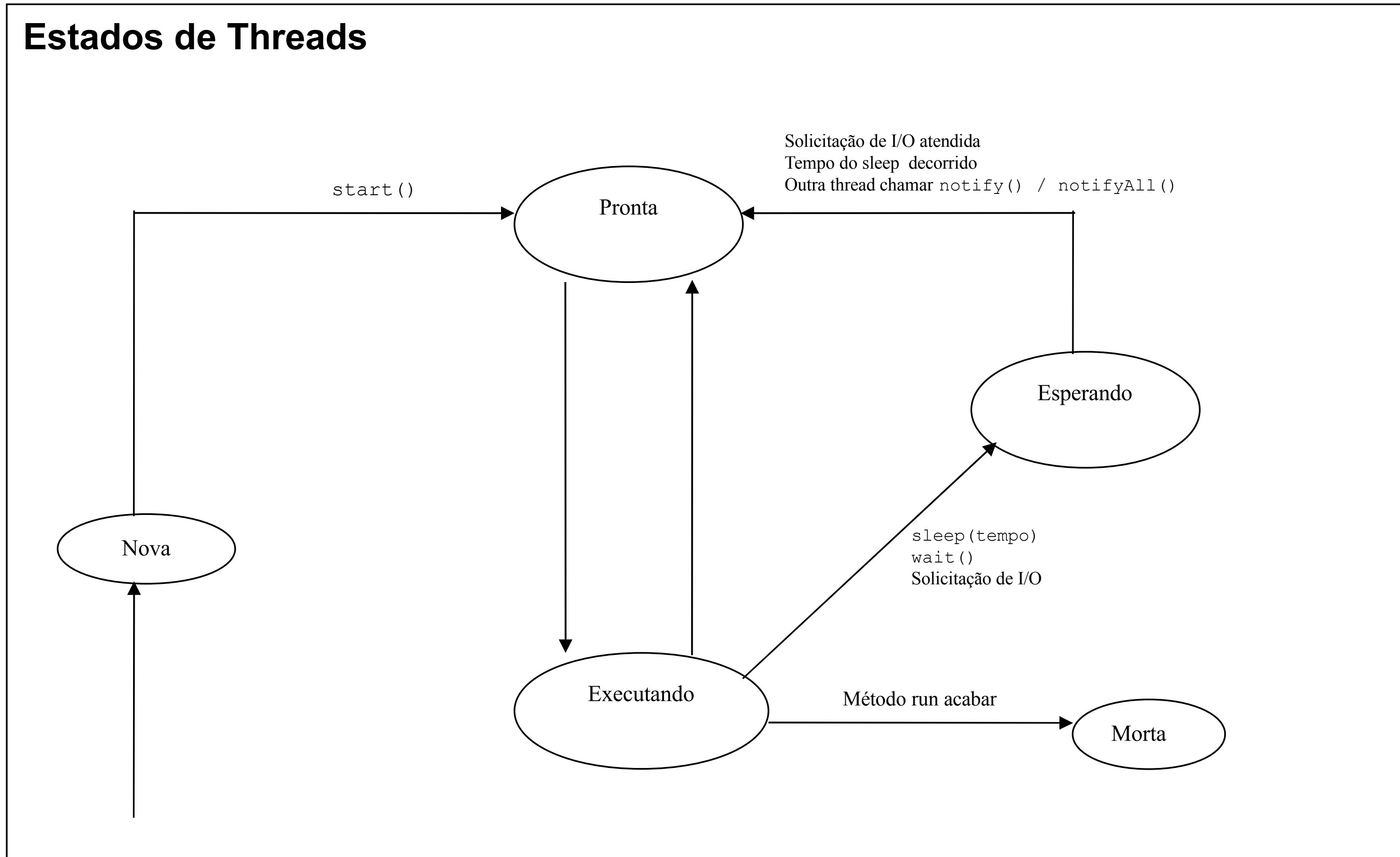
- um espaço de endereçamento (virtual address space) que contém o texto do programa e dos dados
- uma tabela de Descritores de arquivos abertos
- informação sobre processos filhos
- código para tratar de sinais (signal handlers)
- informação sobre Permissões, etc.



2 Consiste numa linha ou contexto de execução, normalmente chamada "Thread"

- Uma thread tem um programa counter (PC) que guarde o endereço sobre a próxima instrução a executar
- Registadores – valores das variáveis atuais
- Stack – contém o histórico de execução com um "frame" para cada procedimento chamado mas não terminado

Thread - Estados



Thread - Objetivos

- O conceito de thread foi criado com dois objetivos principais:
  - Facilidade de comunicação entre unidades de execução;
  - Redução do esforço para manutenção dessas unidades.

Thread

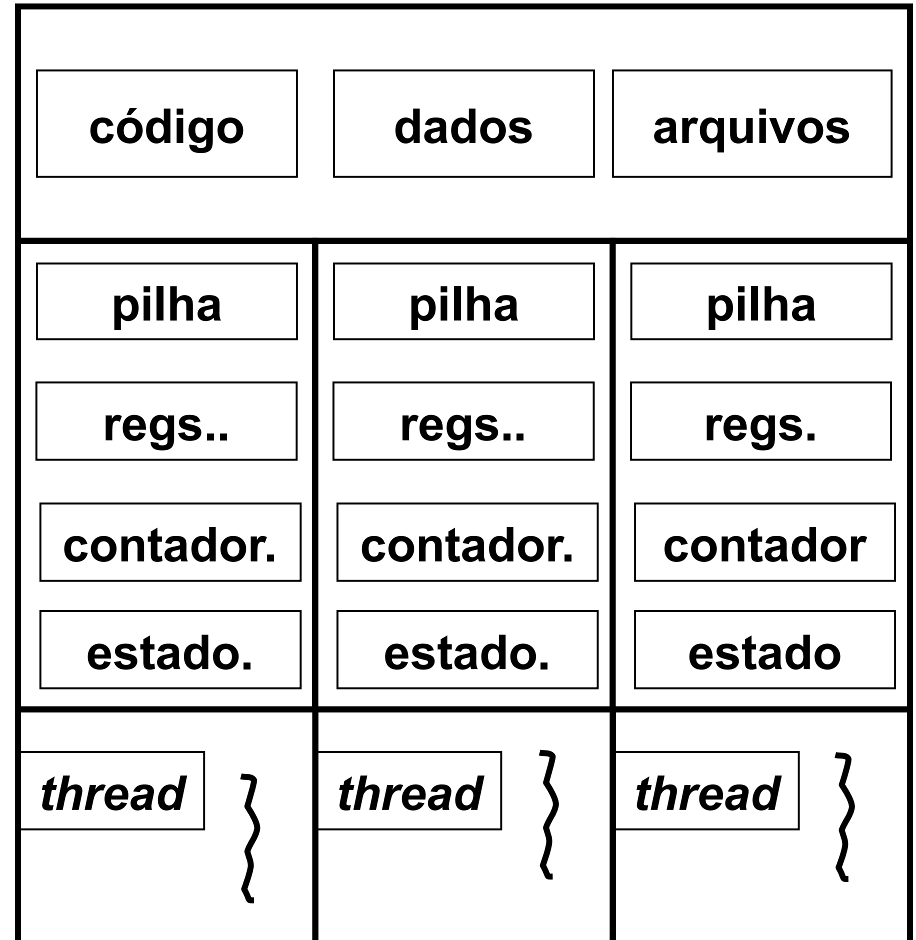
- **Processo ->** um espaço de endereço e uma única linha de controle
- **Threads ->** um espaço de endereço e múltiplas linhas de controle
  - O Modelo do Processo
    - Agrupamento de recursos (espaço de endereço com texto e dados do programa; arquivos abertos, processos filhos, tratadores de sinais, alarmes pendentes etc)
    - Execução
  - O Modelo da Thread
    - Recursos particulares (PC, registradores, pilha)
    - Recursos compartilhados (espaço de endereço – variáveis globais, arquivos etc)
    - Múltiplas execuções no mesmo ambiente do processo – com certa independência entre as execuções
- **Analogia**
  - Execução de múltiplos threads em paralelo em um processo *(multithreading)* e Execução de múltiplos processos em paralelo em um computador

Threads

Itens por Processo	Itens por *Thread*
■Espaço de endereçamento ■Variáveis globais ■Arquivos abertos ■Processos filhos ■Alarmes pendentes	■Contador de programa ■Registradores (contexto) ■Pilha ■Estado

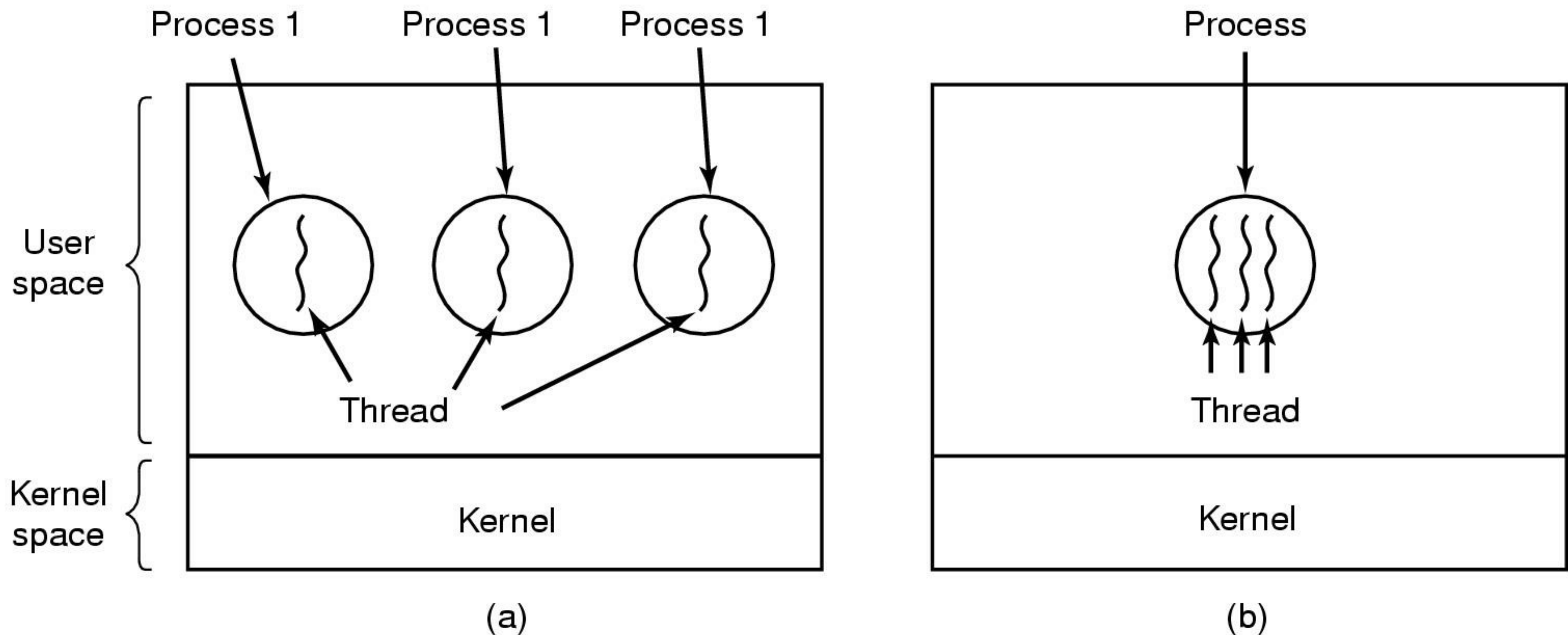
- Compartilhamento de recursos;
- Cooperação para realização de tarefas;

Threads



Processo com uma única *thread* **Processo com várias *threads***

Threads
O Modelo Thread (1)



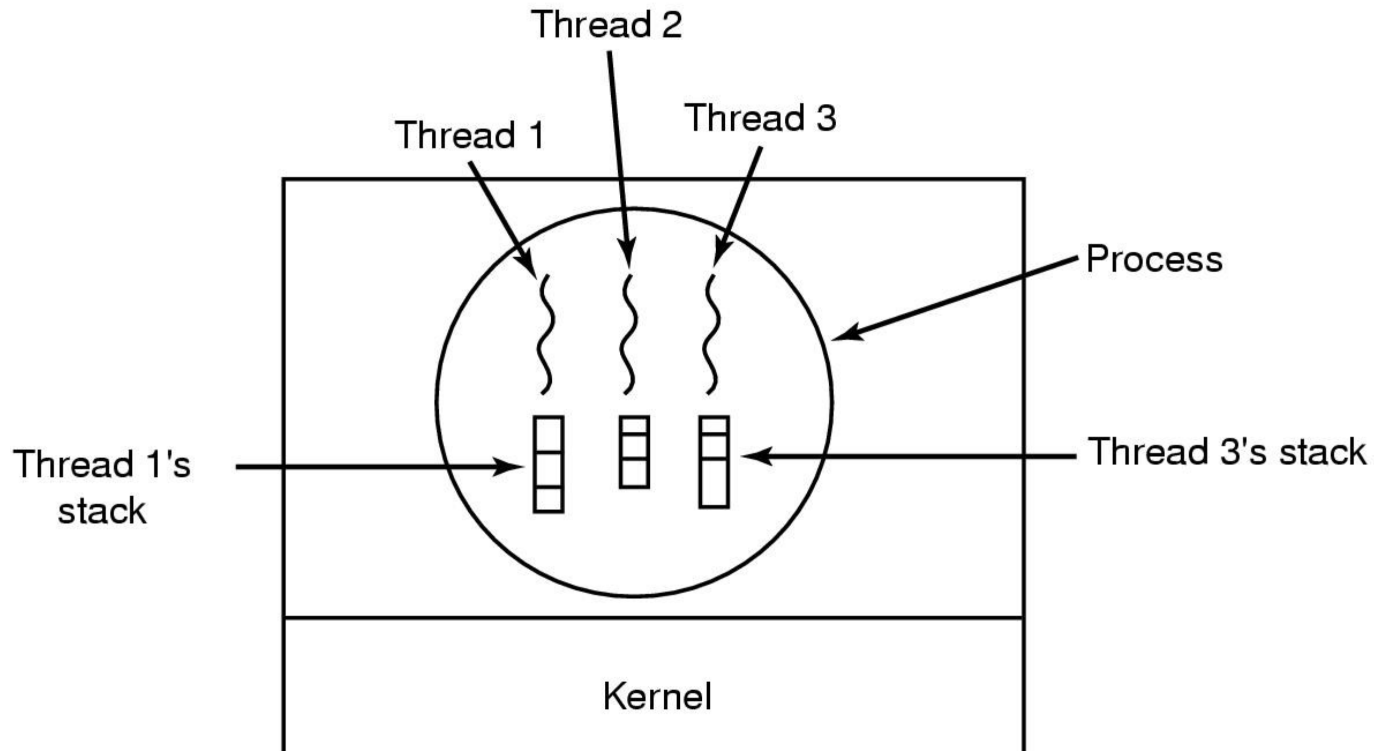
(a) Três processos, cada um com um *thread*

(b) Um processo com três *threads*

Thread - O Grande Benefício

- Não há nada de novo nesse conceito de processo com um único thread, pois o mesmo é idêntico ao conceito tradicional de processo.
- O grande benefício no uso de thread é quando temos vários threads num mesmo processo sendo executados simultaneamente e podendo realizar tarefas diferentes.

O Modelo Thread (3)



Cada thread tem sua própria pilha de execução

Threads

- Dessa forma pode-se perceber facilmente que aplicações multithreads podem realizar tarefas distintas ao “mesmo tempo”, dando idéia de paralelismo.
- Exemplo: navegador web HotJava
  - consegue carregar e executar applets;
  - executar uma animação;
  - tocar um som;
  - exibir diversas figuras;
  - permitir rolagem da tela;
  - carregar uma nova página; entre outros

- para o usuário todas essas atividades são simultâneas, mesmo possuindo um único processador (possível devido a execução de vários threads, provavelmente, uma para cada tarefa a ser realizada.)

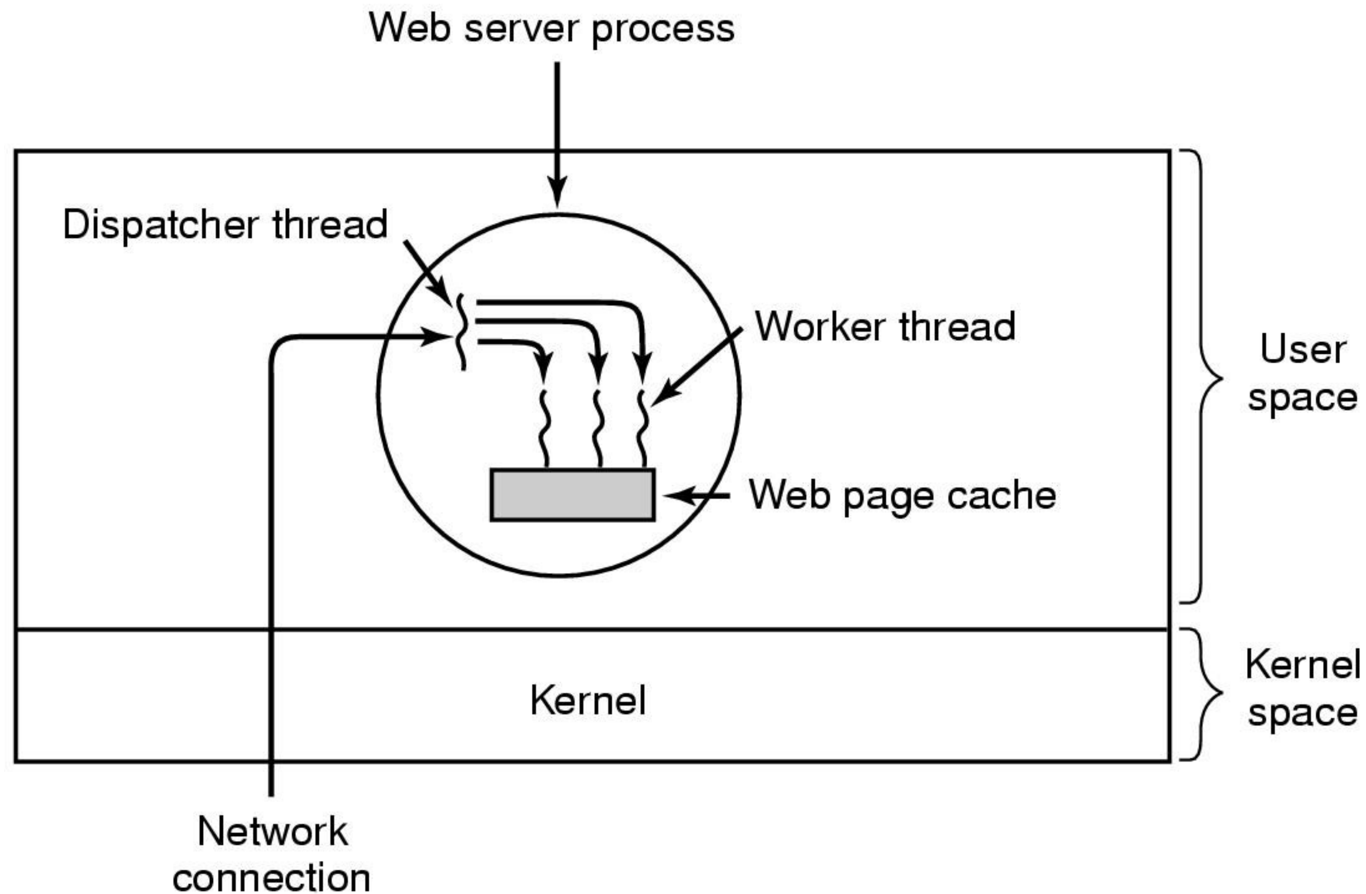
Threads

- Não há proteção entre threads, pois é desnecessário;
  - 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 de outra *thread*;
  - Porém, há a necessidade de sincronizá-las.

Threads

- Razões para existência de *threads*:
  - Em múltiplas aplicações ocorrem múltiplas atividades “ao mesmo tempo”, e algumas dessas atividades podem bloquear de tempos em tempos;
  - As *threads* são mais fáceis de gerenciar do que processos, pois elas não possuem recursos próprios → o processo é que tem!
  - Desempenho: quando há grande quantidade de E/S, as threads permitem que essas atividades se sobreponham, acelerando a aplicação;
  - Paralelismo Real em sistemas com múltiplas CPUs.

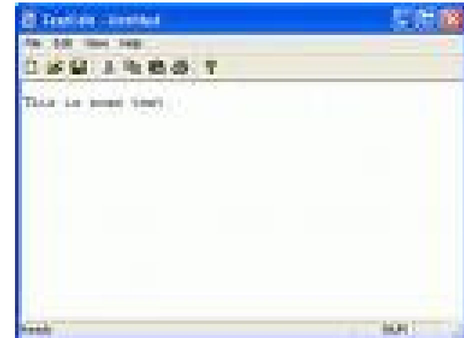
Threads – exemplo no web server



Threads

- Considere um navegador WEB:
  - Muitas páginas WEB contêm muitas figuras que devem ser mostradas assim que a página é carregada;
  - Para cada figura, o navegador deve estabelecer uma conexão separada com o servidor da página e requisitar a figura → tempo;
  - Com múltiplas *threads*, muitas imagens podem ser requisitadas ao mesmo tempo melhorando o desempenho;

Threads



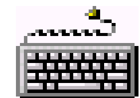
- Considere um Editor de Texto:
  - Editores mostram documentos formatados que estão sendo criados em telas (vídeo);
  - No caso de um livro, por exemplo, todos os capítulos podem estar em apenas um arquivo, ou cada capítulo pode estar em arquivos separados;
  - Diferentes tarefas podem ser realizadas durante a edição do livro;
  - Display, formatação de textos, contagem de caracteres, etc.

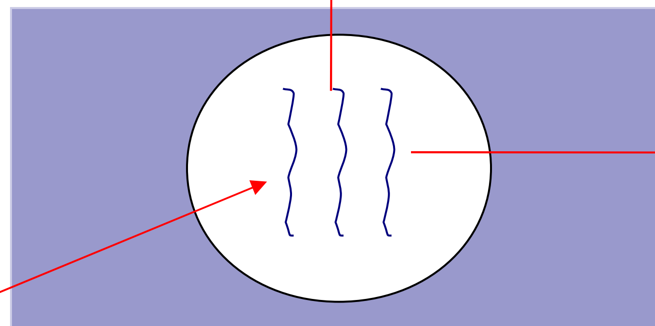
Threads

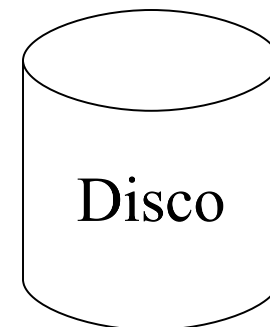
- *Threads* para diferentes tarefas;

Pipeline, antes uma exótica técnica exclusiva dos computadores topo de linha, tem se tornado um lugar comum no projeto de computadores.

A técnica de pipeline nos processadores é baseada no mesmo princípio das linhas de montagem das fábricas: não precisa-se esperar até que a unidade esteja completamente montada para começar a fabricar a próxima.







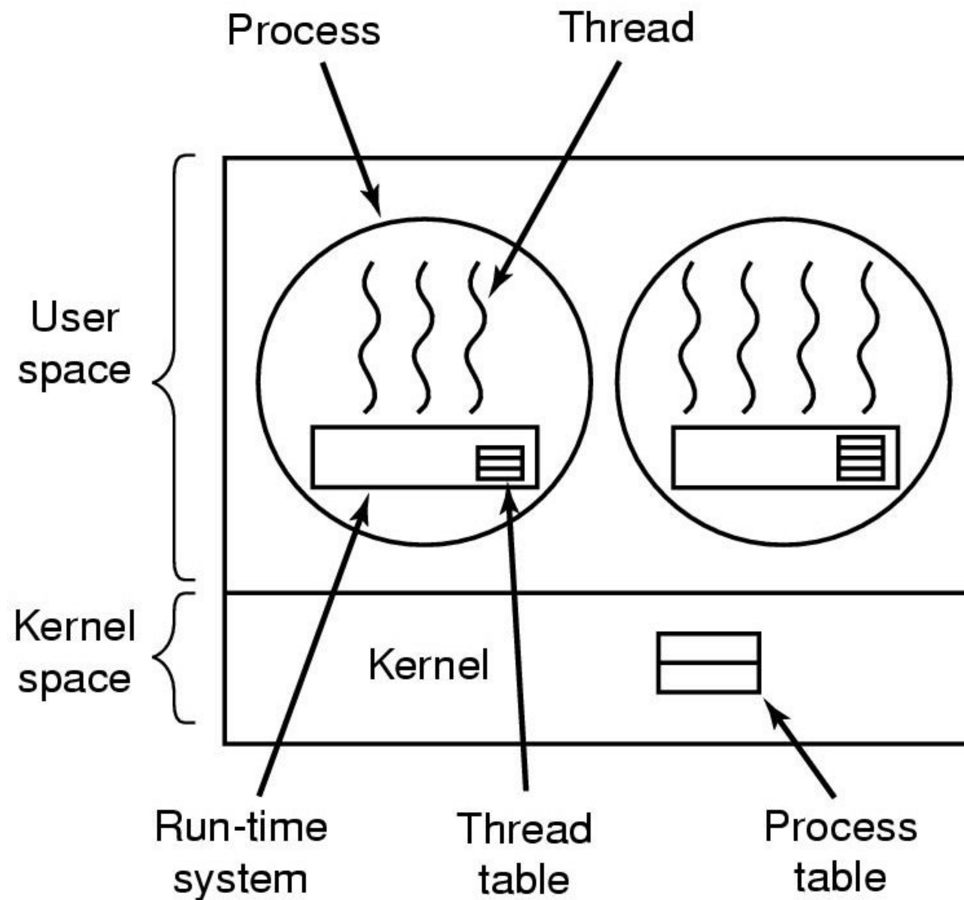
Threads

- Benefícios:
  - Capacidade de resposta: aplicações interativas; Ex.: servidor WEB;
  - Compartilhamento de recursos: mesmo endereçamento; memória, recursos;
  - Economia: criar e realizar chaveamento de *threads* é mais barato;
  - Utilização de arquiteturas multiprocessador: processamento paralelo;

Threads - Tipos

- Tipos de *threads*:
  - Em modo usuário (espaço do usuário): implementadas por bibliotecas no nível do usuário;
    - Criação e escalonamento são realizados sem o conhecimento do *kernel*;
      - Tabela de *threads* para cada processo;
    - Cada processo possui sua própria tabela de *threads*, que armazena todas as informações referentes à cada *thread* relacionada àquele processo;
    - Exemplo no Linux: GNU Portable Thread

Threads em modo usuário



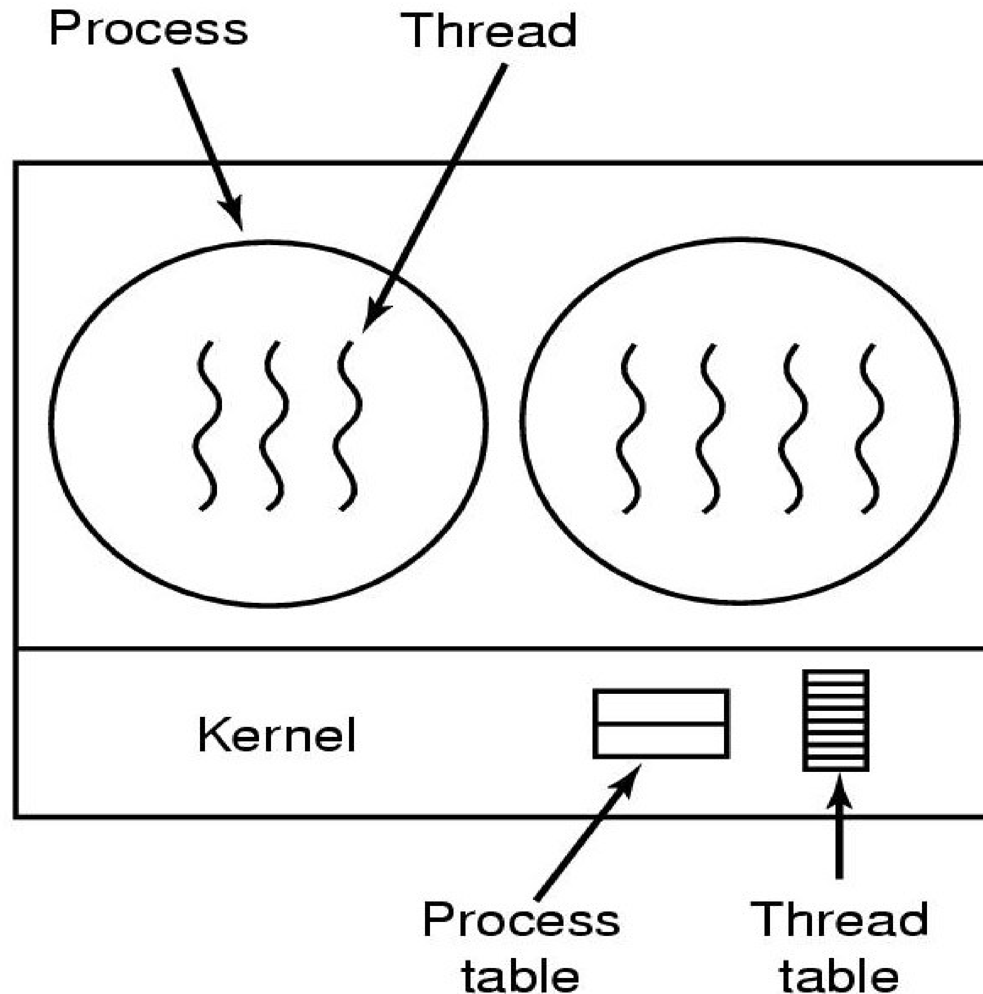
Threads em modo usuário

- Tipos de *threads*: Em modo usuário
- Vantagens:
  - Alternância de *threads* no nível do usuário é mais rápida do que alternância no *kernel*;
  - Menos chamadas ao *kernel* são realizadas;
  - Permite que cada processo possa ter seu próprio algoritmo de escalonamento;
- Principal desvantagem:
  - Processo inteiro é bloqueado se uma *thread* realizar uma chamada bloqueante ao sistema;

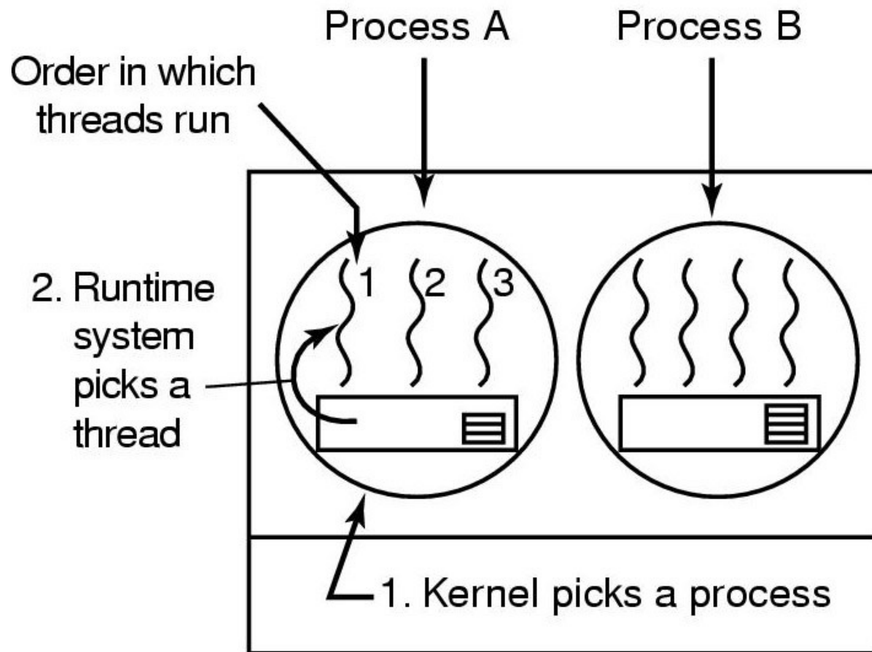
Tipos de *Threads*

- Tipos de *threads*:
- Em modo *kernel*: suportadas diretamente pelo SO;
- No linux e C, pode ser criada pelo comando kernel_thread()
- Criação, escalonamento e gerenciamento são feitos pelo *kernel*;
- Tabela de *threads* e tabela de processos separadas;
- as tabelas de *threads* possuem as mesmas informações que as tabelas de threads em modo usuário, só que agora estão implementadas no *kernel*;

Threads em modo *kernel*

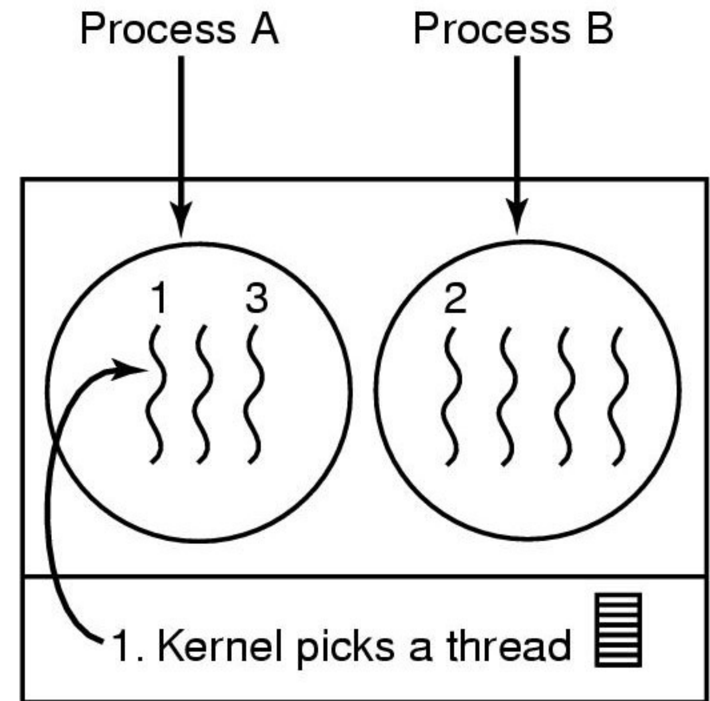


Threads em modo Usuário *x* *Threads* em modo *Kernel*



Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3

Threads em modo usuário



Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

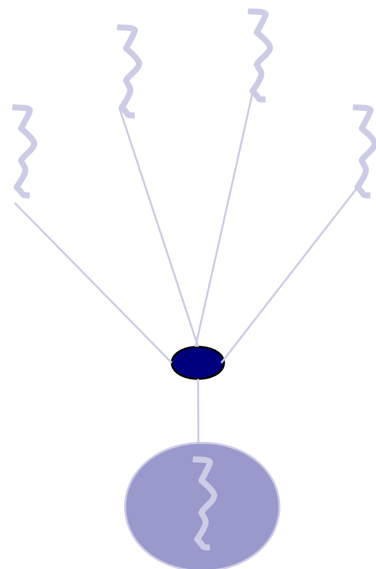
Threads em modo *kernel*

Threads em modo *kernel*

- Tipos de *threads*: em modo *kernel*
- Vantagem:
  - Processo inteiro não é bloqueado se uma *thread* realizar uma chamada bloqueante ao sistema;
- Desvantagem:
  - Gerenciar threads em modo *kernel* é mais caro devido às chamadas de sistema durante a alternância entre modo usuário e modo *kernel*;

Threads

- Modelos *Multithreading*
  - Muitos-para-um:
    - Mapeia muitas *threads* de usuário em apenas uma *thread* de *kernel*;
    - Não permite múltiplas *threads* em paralelo;

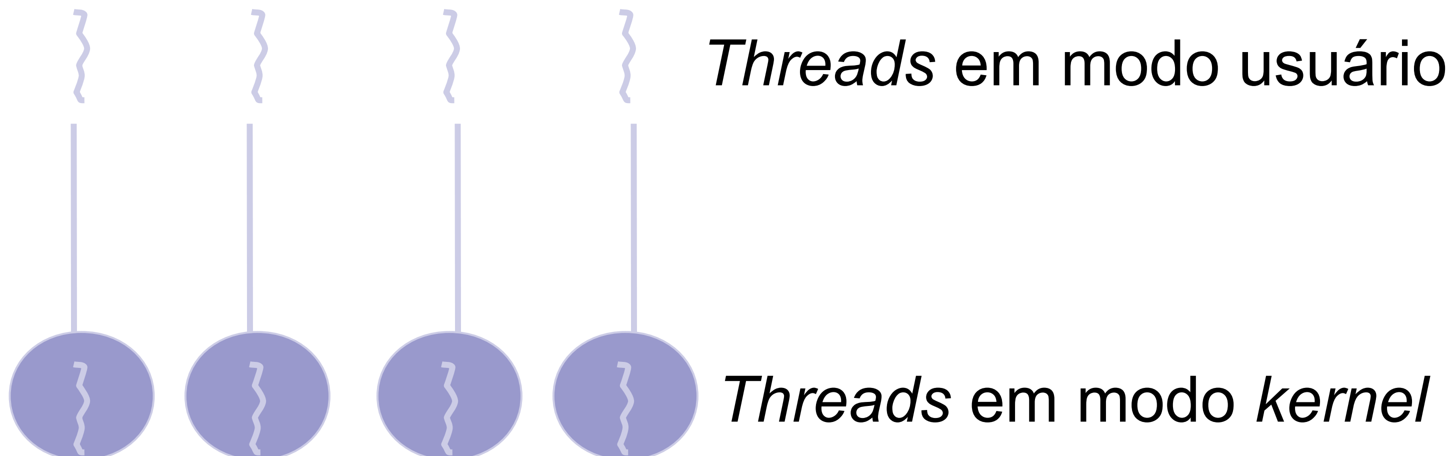


Threads em modo usuário

Thread em modo *kernel*

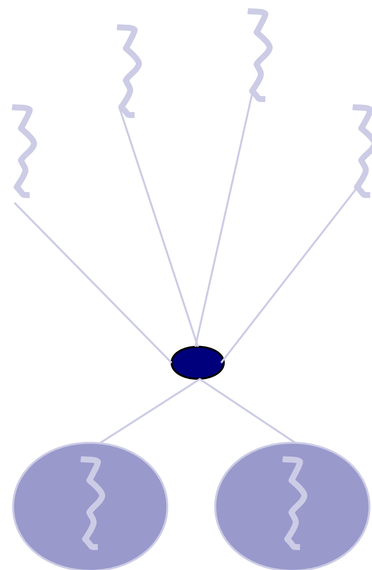
Threads

- Modelos *Multithreading*
  - Um-para-um: (Linux, Família Windows, OS/2, Solaris 9)
    - Mapeia para cada *thread* de usuário uma *thread* de *kernel*;
    - Permite múltiplas *threads* em paralelo;



Threads

- Modelos *Multithreading*
  - Muitos-para-muitos: (Solaris até versão 8, HP-UX, Tru64 Unix, IRIX)
    - Mapeia para múltiplos *threads* de usuário um número menor ou igual de *threads* de *kernel*;
    - Permite múltiplas *threads* em paralelo;
    - “pool” de threads



Threads em modo usuário

Thread em modo *kernel*

1. **Aspectos de Implementação de threads e processos**
2. **Comunicação interprocessos**

Threads

- Estados: executando, pronta, bloqueada;
- Comandos para manipular threads:
  - *Thread_create;*
  - *Thread_exit;*
  - *Thread_wait;*
  - *Thread_yield* (permite que uma *thread* desista voluntariamente da CPU);

Threads

- **Por que threads?**
  - Simplificar o modelo de programação (aplicação com múltiplas atividades => decomposição da aplicação em múltiplas threads)
  - Gerenciamento mais simples que o processo (não há recursos atachados – criação de thread 100 vezes mais rápida que processo)
  - Melhoria do desempenho da aplicação (especialmente quando thread é orientada a E/S)
  - Útil em sistemas com múltiplas CPUs

Lembrando...

1) Processos são usados para agrupar recursos.

2) Threads são as entidades escalonadas para execução na CPU.

Exemplo de Criação de Processo em C no Linux

```
#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 ("Processe Pai terminou.\n");
  }
}
```

Criando 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();
        // (new ThreadSimples()).start();
        System.out.println("Ola da thread original!");
    }
}
```

Criando Threads em Java implementando a interface Runnable

- Vantagem:

- 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();
        //(new Thread(new RunnableSimples())).start();
        System.out.println("Ola da thread original!");
    }
}
```

Joining a Thread

- Permite a uma thread esperar que outra termine

```
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!");
   }
 }
```

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();
  }
```