

Sistemas Operacionais

Cap.3 -Processos



Prof. MSc. Renzo P. Mesquita
renzo@inatel.br

Capítulo 3

Processos

- 3.1. Conceito de Processo;*
- 3.2. Estados de um Processo;*
- 3.3. Representação de um Processo;*
- 3.4. Scheduling de Processos;*
- 3.5. Operações sobre Processos;*
- 3.6. Comunicação Interprocessos;*
- 3.7. Comunicação em Sistemas Cliente-servidor;*



Objetivos

- Introduzir a noção de Processo - um programa em execução que forma a base de toda comunicação;
- Descrever as diversas características dos processos (Scheduling, criação, encerramento etc.);
- Descrever a comunicação em sistemas cliente-servidor;

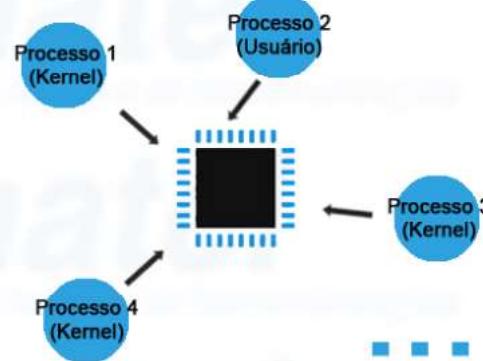


3.1. Conceito de Processo

Os Sistemas de Computação atuais permitem que **vários programas** **sejam carregados na memória** e executados concorrentemente. Essa evolução demandou um controle mais firme dos diversos programas em execução. Essas necessidades resultaram na noção de **PROCESSO**.

Um Processo nada mais é que um programa em execução.

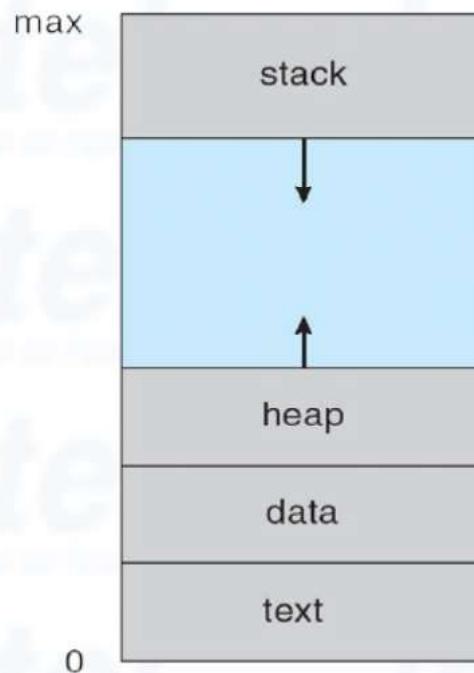
- Um Processo é uma **unidade ativa**, enquanto um Programa é uma **unidade passiva**;
- O Processo é a **unidade básica** de trabalho do Sistema Operacional;
- Um Sistema Operacional gerencia um conjunto de processos que **executam concorrentemente** em uma ou mais CPU's.
- Estes processos podem ser de dois tipos: **Processos de Kernel** e **Processos de Usuário**;



3.1. Conceito de Processo

Um Processo é mais do que o código do programa em execução.
Um Processo na memória é formado pelas seguintes partes:

- Código do Programa (**text section**);
- **pilha (stack)**, que armazena dados temporários como parâmetros de funções, endereços de retorno e variáveis locais;
- Seção de dados (**data Section**) que guarda as variáveis globais;
- **heap**, que é a memória dinamicamente alocada durante o tempo de execução do processo.



3.1. Conceito de Processo

Duas técnicas comuns para execução de programas são:

- Clicar duas vezes em um ícone representando o arquivo executável;
- Digitar o nome do arquivo executável na linha de comando (Ex: programa.exe ou prog.out);

Embora dois processos possam estar associados ao mesmo programa, mesmo assim eles são considerados duas sequências de execução separadas;

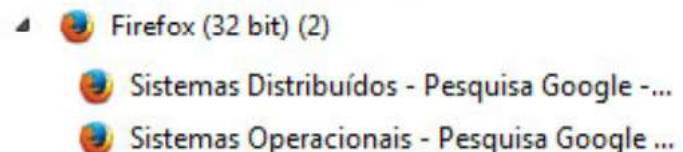
Ex:



Processo Firefox 1



Processo Firefox 2



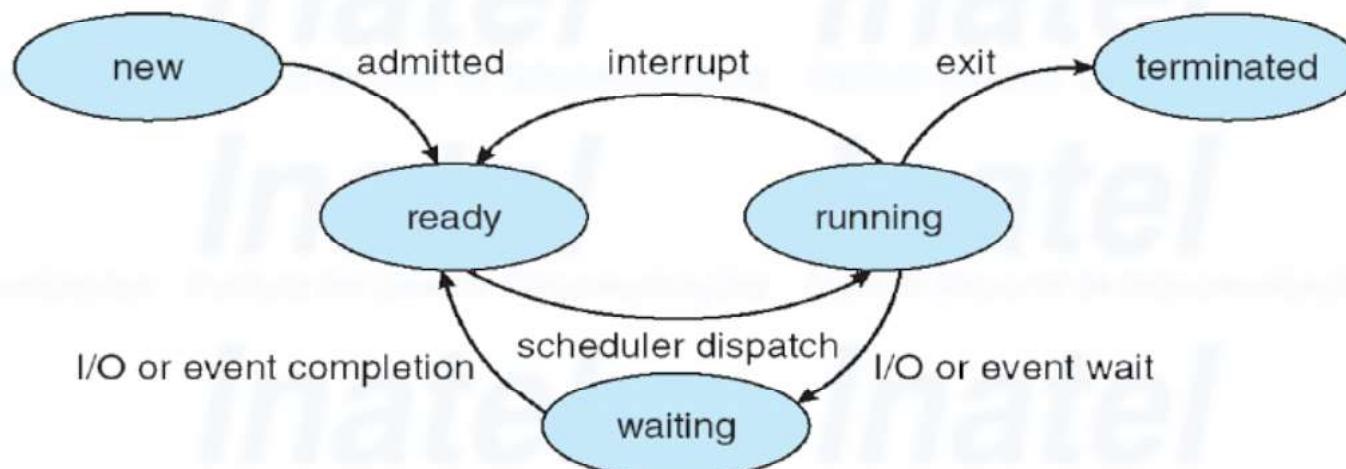
Gerenciador de Tarefas (Windows 8)

Cada uma das sequências de execução é um Processo separado e, embora as seções de texto sejam equivalentes, os dados, o heap e a pilha variam.

3.2. Estados de um Processo

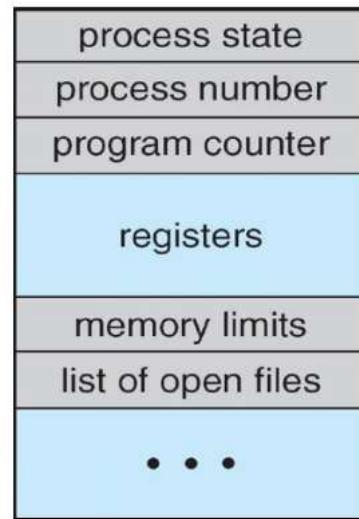
Quando um Processo é executado, ele muda de estado. O estado de um Processo é definido em parte pela atividade corrente deste processo. Cada processo pode estar em um dos estados a seguir:

- Novo (**New**): quando o Processo está sendo criado, alocando recursos;
- Em Execução (**Running**): Instruções estão sendo executadas;
- Em Espera (**Waiting**): O Processo está esperando que algum evento ocorra (Ex: esperando a finalização de uma operação de IO);
- Pronto (**Ready**): O Processo está esperando ser atribuído a um processador;
- Concluído (**Terminated**): O Processo terminou sua execução;



3.3. Representação de um Processo

Cada processo é representado no Sistema Operacional por um **Bloco de Controle de Processo (PCB, Process Control Block)**. Um PCB de um processo possui as seguintes informações:



PCB (Process Control Block)

1) Estado do Processo (*process state*)

Armazena em que estado se encontra o processo (novo, pronto, em execução...);

2) Número do Processo - PID (*process number*)

Número único dado ao processo pelo SO quando o mesmo começa sua execução;

3) Contador do Programa (*program counter*)

Indica o endereço da próxima instrução a ser executada para este processo;

3.3. Representação de um Processo

4) Registradores da CPU (registers)

Os registradores variam em número e tipo, dependendo da arquitetura do computador. Junto com o PC, as informações dos registradores devem ser salvas quando ocorrer uma interrupção, para permitir que o processo seja retomado corretamente mais tarde;

5) Informações de gerenciamento de Memória (memory limits)

Inclui os valores dos registradores base e limite, tabela de páginas ou as tabelas de segmentos, dependendo do sistema de memória usado;

6) Lista de Arquivos Abertos (list of open files)

Guarda o caminho de todos os arquivos externos que estão sendo usados por um processo específico;

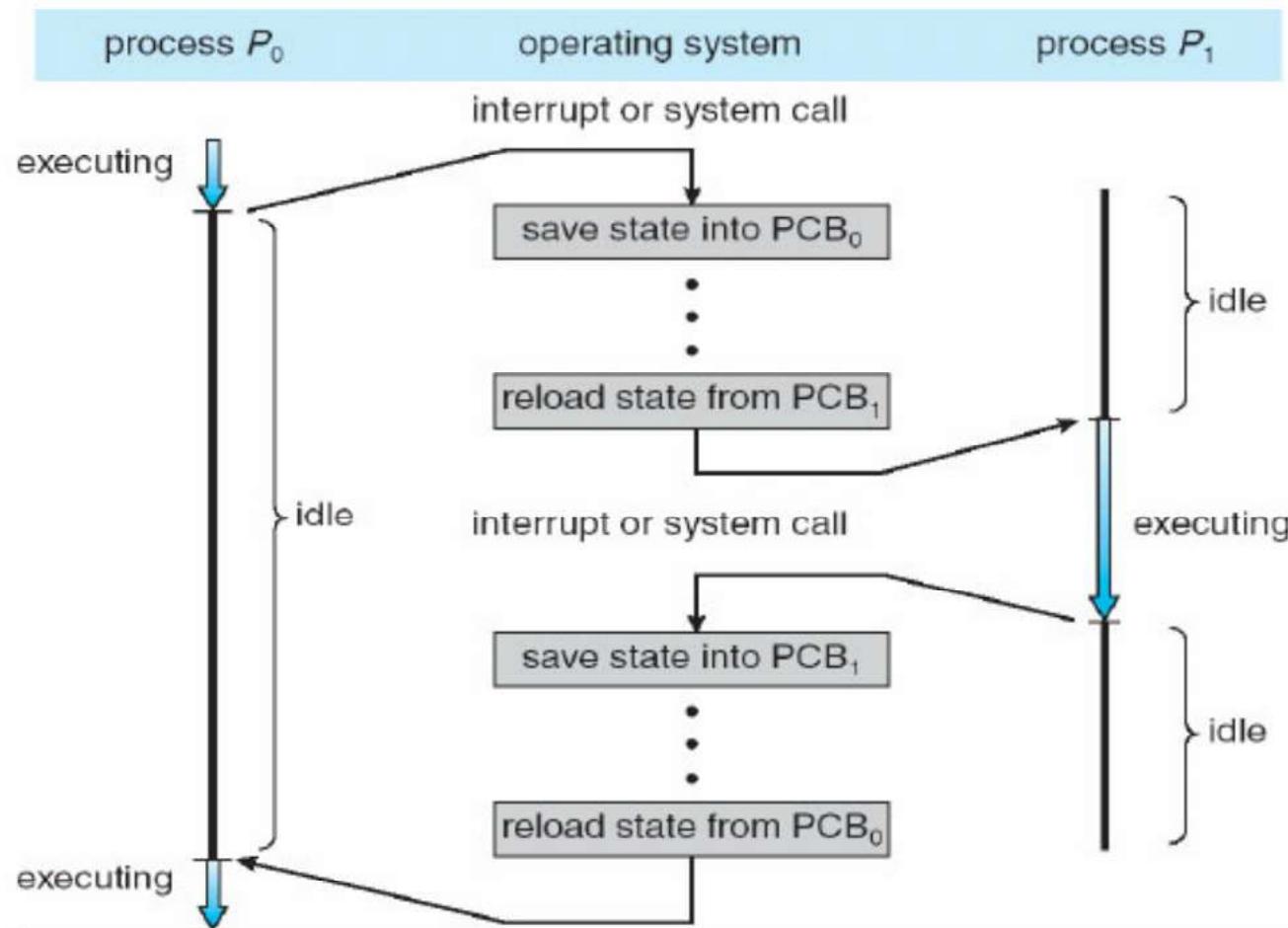
E muitas outras informações como:

Informações de Scheduling da CPU, Informações de Contabilização, Informações de Status de IO entre outras informações;

Resumindo, o PCB de um processo é como se fosse um repositório que guarda/gerencia todas as informações deste processo para o SO.

3.3. Representação de um Processo

Diagrama mostrando a alternância da CPU de um processo para outro.



3.4. Scheduling de Processos

O Objetivo da multiprogramação é **sempre termos algum processo em execução** para otimização do uso da CPU.

O Objetivo do compartilhamento de tempo é a alternância da CPU entre os processos com tanta frequência que os usuários possam interagir com cada programa, com a impressão que todos estivessem sendo executados em paralelo.

Para alcançar estes objetivos, um componente chamado **SCHEDULER DE PROCESSOS** seleciona um processo disponível entre muitos para a execução na CPU;



3.4. Scheduling de Processos

3.4.1. Filas de Scheduling

Geralmente, todos os Sistemas Operacionais possuem 3 tipos de filas a fim de organizar o escalonamento de processos. São elas:

- *Fila de Jobs (job queue)*

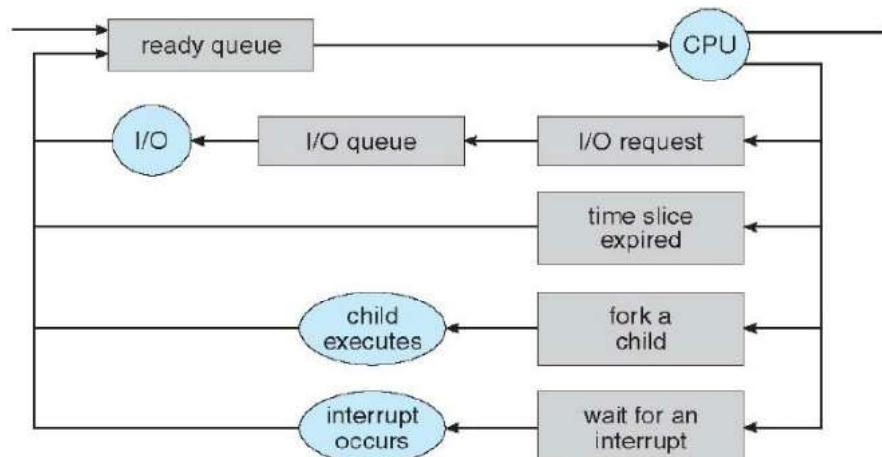
Composta por todos os programas do sistema que aguardam para serem alocados na RAM;

- *Fila de Prontos (ready queue)*

Processos que estão residindo na memória principal e estão prontos esperando execução;

- *Fila de Dispositivo (device queue ou i/o queue)*

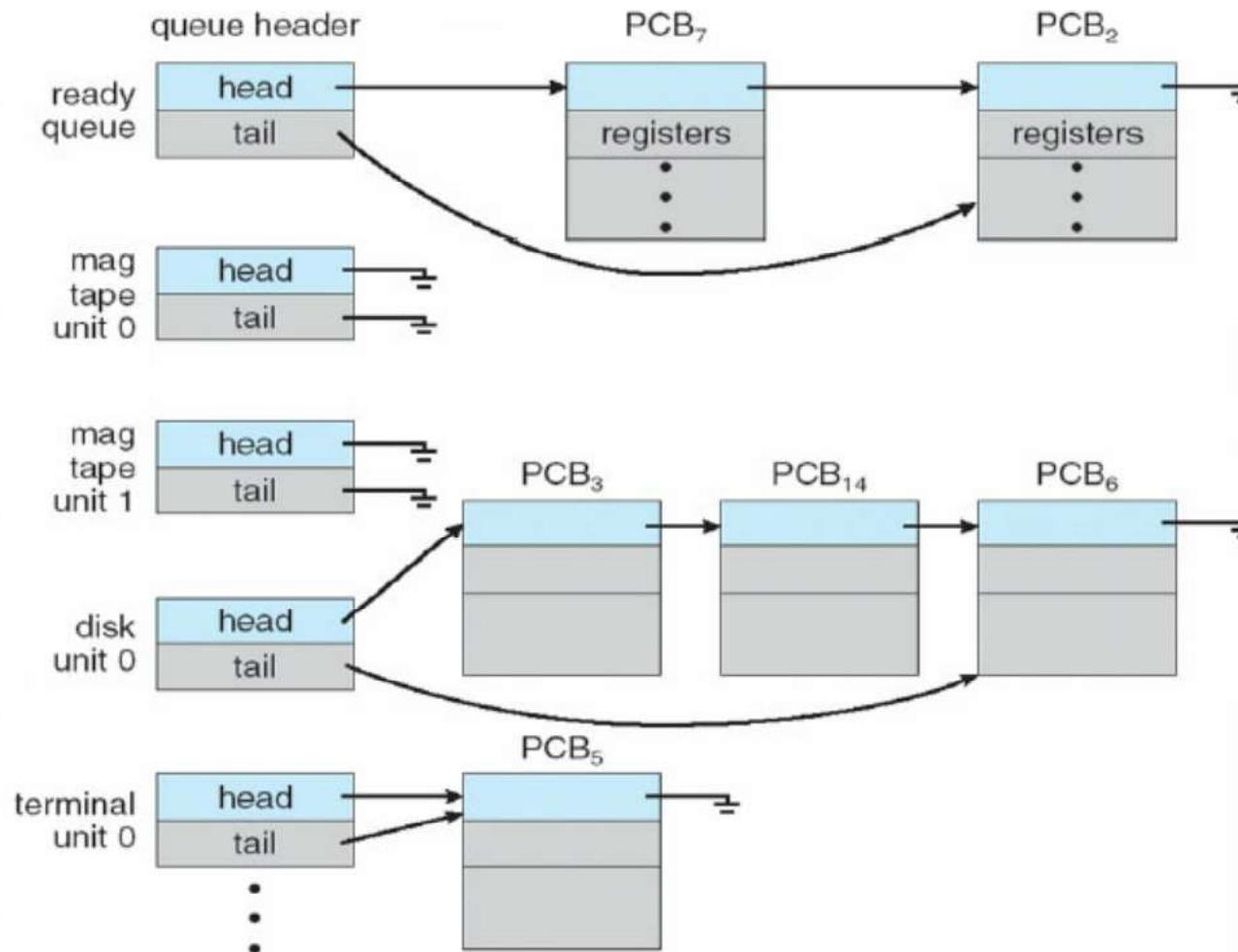
Lista de processos em espera por um dispositivo de I/O. Cada dispositivo tem sua própria fila de dispositivo;



3.4. Scheduling de Processos

3.4.1. Filas de Scheduling

3.4.1.1. Exemplo das Filas de Scheduling



3.4. Scheduling de Processos

3.4.2. Schedulers

Um processo passa por várias Filas de Scheduling durante seu tempo de vida.

Diferentes tipos de Schedulers:

1) Scheduler de Longo Prazo (Long-term Scheduler)

Determina quais processos serão admitidos para poderem executar no sistema. Escalona processos da fila de new para a fila de ready.

2) Scheduler de Curto Prazo (Short-term Scheduler)

Seleciona entre os processos que estão prontos para execução e aloca CPU para um deles.

3) Scheduler de Médio Prazo (Medium-term Scheduler)

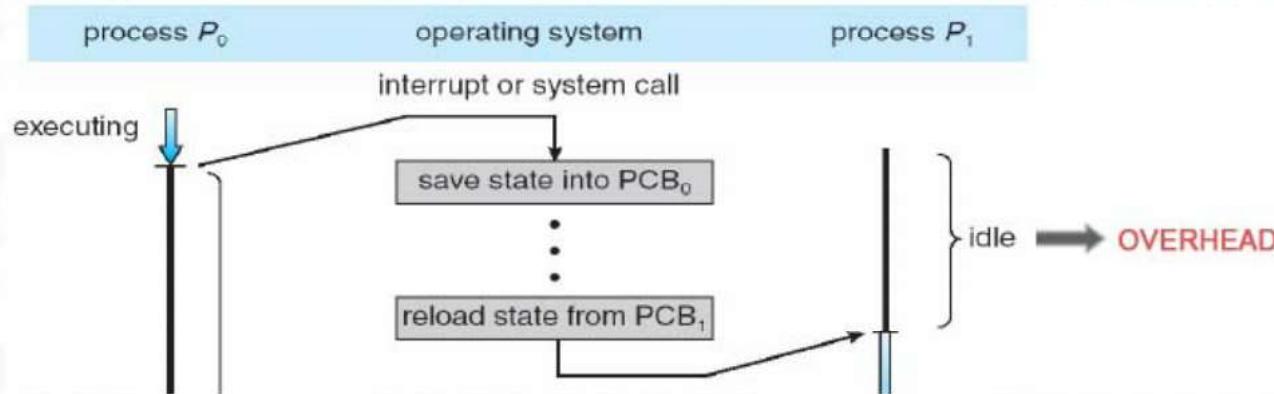
Seleciona qual processo irá da memória para o Swapfile, ou retornará do Swapfile para a memória.

3.4. Scheduling de Processos

3.4.3. Mudança de Contexto (Context Switching)

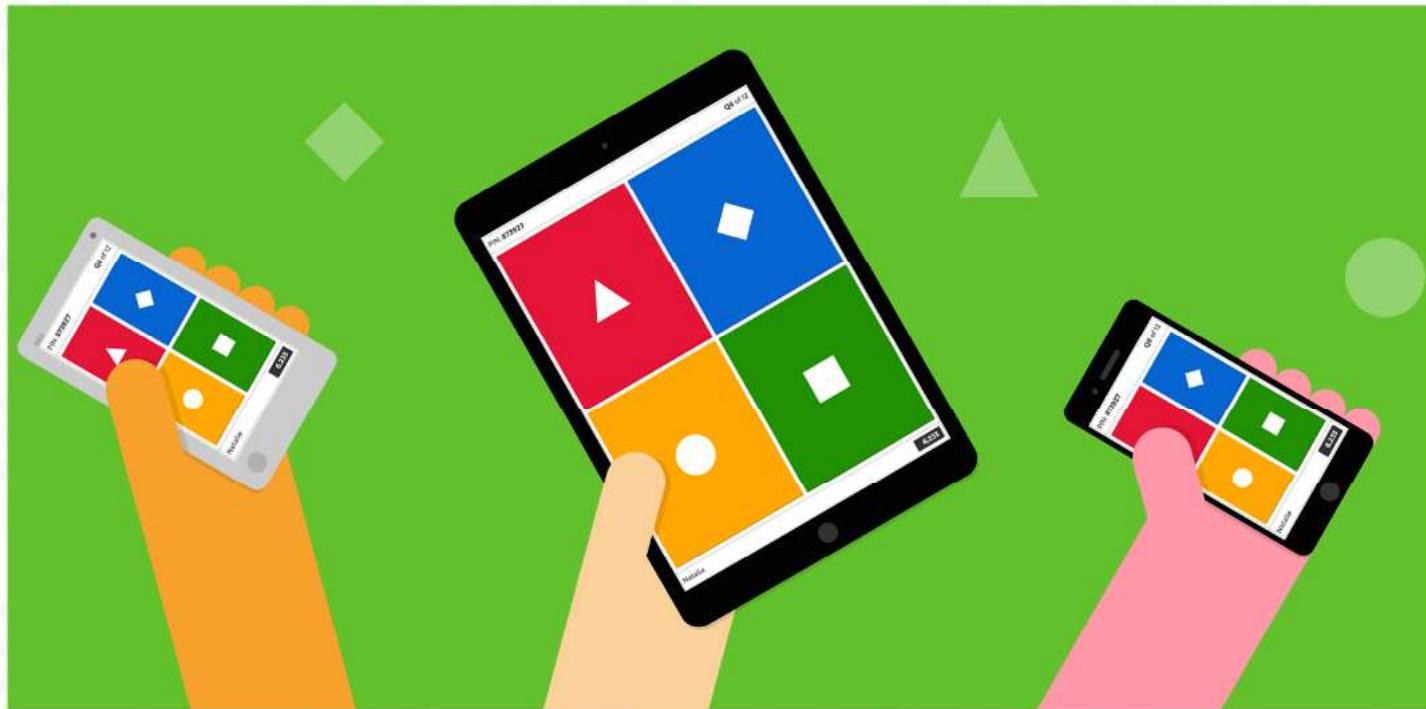
A alocação da CPU a outro Processo requer a execução do salvamento do estado do processo corrente e a restauração do estado de um Processo diferente. Essa tarefa é conhecida como **Mudança de Contexto**. Normalmente uma Mudança de Contexto demora alguns milissegundos;

O intervalo de mudança de contexto é puro **Overhead**, ou seja, o sistema não executa trabalho útil algum durante a mudança.



HORA DO

Kahoot!



ACESSE:

<https://kahoot.it>

3.5. Operações sobre Processos

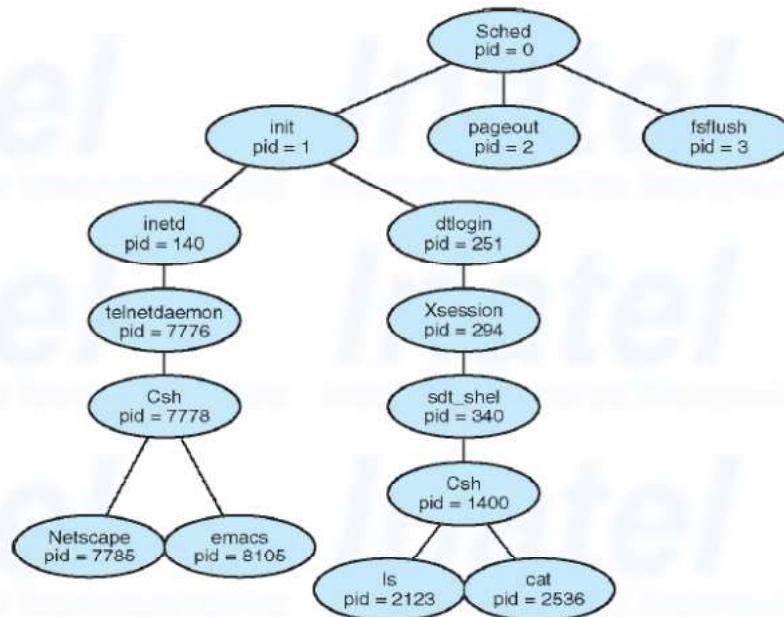
3.5.1 Criação de Processos

Um processo pode criar vários novos processos, através de uma chamada de sistema de criação de processos.

- *Processo Pai (Parent Process)*: Processo criador de novos processos;
- *Processo Filho (Child Process)*: Processos criados;

Cada um dos novos processos criados pode criar outros processos, formando uma **Árvore de Processos**;

Ex:



Árvore de Processos do Sistema Operacional Solaris.

3.5. Operações sobre Processos

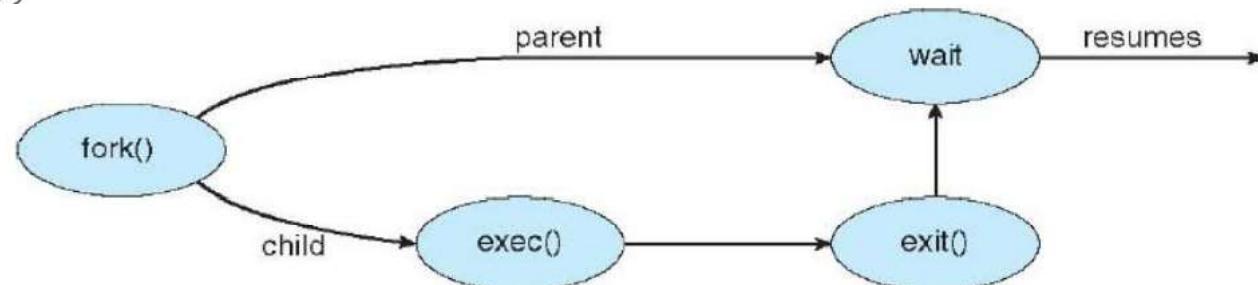
3.5.1 Criação de Processos

A maioria dos Sistemas Operacionais identifica os Processos de acordo com um **Identificador de Processo** (Process ID ou **pid**).

Quando um Processo cria um Subprocesso, esse Subprocesso pode obter seus recursos de duas maneiras:

- Obtendo recursos diretamente do SO;
- Obtendo recursos do Processo Pai;

Um novo Processo é criado pela Chamada de Sistema **fork()** (UNIX);



Criação de processo com o uso da Chamada de Sistema `fork()`.

3.5. Operações sobre Processos

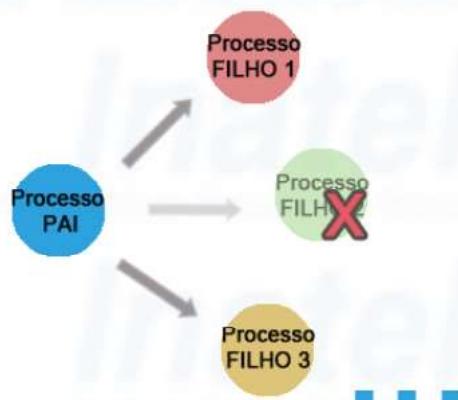
3.5.2. Encerramento de Processos

Um Processo é encerrado usando a Chamada de Sistema `exit()`.

- Todos os recursos do processo, memória física e virtual, arquivos abertos e buffers de I/O são desalocados pelo Sistemas Operacional.

Um Processo Pai pode causar o encerramento de seus processos filhos. Um pai pode encerrar a execução de um de seus filhos por várias razões, são elas:

- O filho excedeu o uso de alguns dos recursos que recebeu;
- A tarefa atribuída ao filho não é mais necessária;
- O pai está sendo encerrado e o SO não permite que um filho continue executando (Encerramento em Cascata);



3.6. Comunicação Interprocessos

3.6.1. Cooperação entre Processos

Processos concorrentes podem ser:

- Independentes: não afetam e/ou são afetados por outros processos;
- Cooperantes: afetam e/ou são afetados por outros processos;

Os Processos Cooperantes compartilham dados entre si, porém, cooperação requer algum mecanismos para comunicação (fornecida pelo SO) e sincronização entre processos. Este mecanismo é denominado **IPC - Interprocess Communication**;

Existem dois modelos básicos de IPC, são eles:

- 1) *Transmissão de Mensagens;*
- 2) *Memória Compartilhada;*

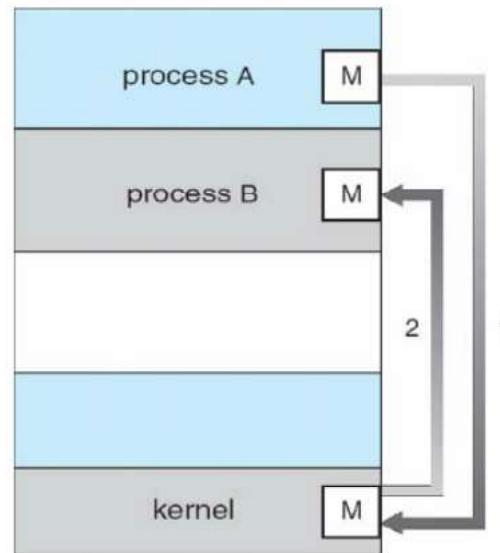
Os dois modelos são comuns nos Sistemas Operacionais, e muitos sistemas implementam ambos.

Em que situações cada uma delas se encaixam?

3.6. Comunicação Interprocessos

3.6.1. Transmissão de Mensagens

- Técnica útil para troca de **pequenas quantidades de dados**, pois não é necessário evitar conflitos;
- Mais fácil de implementar e ideal para troca de dados entre processos remotos;
- Um recurso que utiliza da técnica de Transmissão de Mensagens fornece pelo menos **duas operações: send() e receive()**;
- Se dois Processos, A e B, quiserem se comunicar, um **link de comunicação** deve existir entre eles;

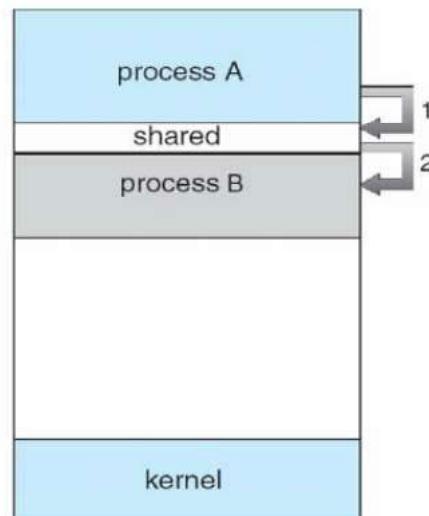


Transmissão de Mensagens
(Message Passing)

3.6. Comunicação Interprocessos

3.6.2. Memória Compartilhada

- É o melhor modelo para troca de **grandes volumes de dados** entre processos locais;
- **Mais rápida** que a Transmissão de Mensagens (Exige um menor número de System Calls);
- Geralmente o SO tenta impedir que um processo acesse a memória de outro processo. A memória compartilhada requer que **dois ou mais processos concordem** em eliminar essa restrição;
- Normalmente a região de memória compartilhada reside no **espaço de endereço do processo que cria** o segmento de memória compartilhada;



Memória Compartilhada
(Shared Memory)

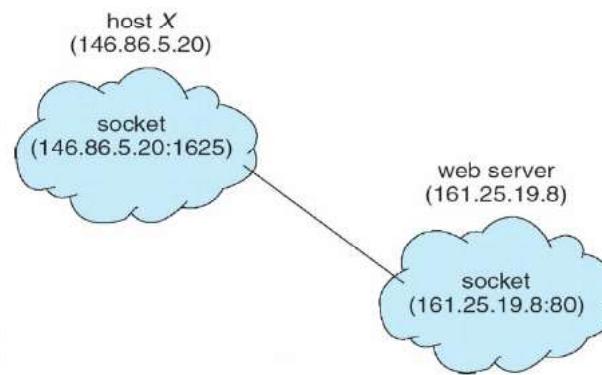
3.7. Comunicação em Sistemas Cliente-servidor

3.7.1. Sockets

Um socket é definido com uma extremidade de comunicação. Dois processos se comunicam em uma rede via um par de Sockets - Um Socket no processo local e o outro no processo remoto.

- Um Socket é definido como um par (IP,Porta);
- Um Processo escreve ou lê em uma porta para se comunicar com o processo remoto;
- Qualquer número de Porta válida pode ser usada. As Portas com numeração **abaixo de 1024** são conhecidas como '**well-known ports**';
- Normalmente os Sockets podem ser implementados de 2 (duas) maneiras: Sockets UDP e Sockets TCP;

Ex:



3.7. Comunicação em Sistemas Cliente-servidor

3.7.2. Pipes

Um Pipe atua como um canal que permite que dois Processos se comuniquem. Normalmente fornecem uma das maneiras mais simples para Processos se comunicarem uns com os outros, porém com algumas limitações.

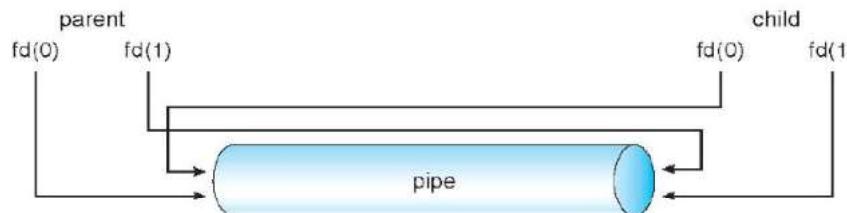
Os Pipes foram um dos primeiros mecanismos IPC dos sistemas UNIX iniciais. Podem ser de dois tipos:

1) Pipes Comuns (Ordinary Pipes)

- Permite a comunicação em apenas uma direção;
- Permite a comunicação apenas entre Processos Pai e Filho;

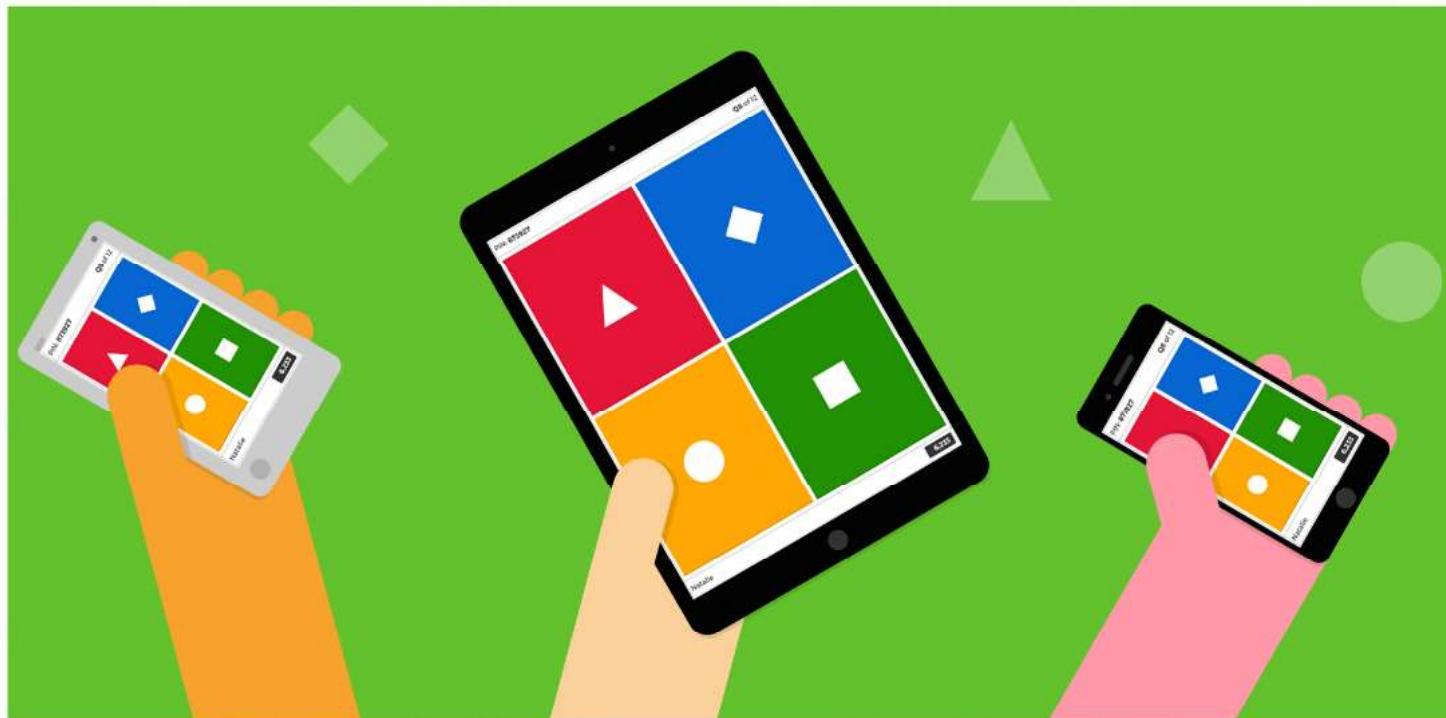
2) Pipes Nomeados (Named Pipes)

- Permite a comunicação bidirecional;
- Permite a comunicação entre quaisquer Processos;



(Pipe Comum)

HORA DO **Kahoot!**



ACESSE: <https://kahoot.it>

Fim do Capítulo 3



Próxima Aula
Threads

Sistemas Operacionais

Cap.4 - Threads



Prof. Me. Renzo P. Mesquita

Capítulo 4

Threads

- 4.1. Visão Geral;*
- 4.2. Modelos de Geração de Multithreads;*
- 4.3. Bibliotecas de Threads;*



Objetivos

- Introduzir o conceito de Thread - uma unidade básica de utilização da CPU que forma a base dos Sistemas de Computação Multithreaded;
- Examinar questões da programação multithreaded;
- Discutir o que são bibliotecas de Threads e quais as mais populares;

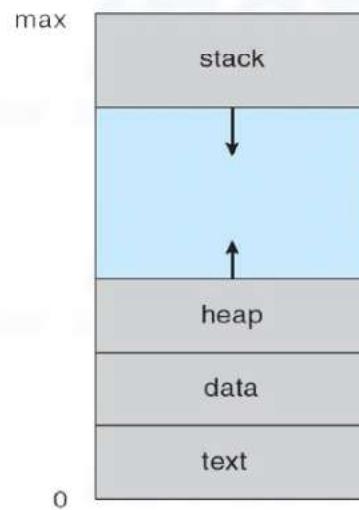


4.1 Visão Geral

4.1.1. Processos

Como já vimos, um Processo possui 4 partes em memória:

- 1) Code Section (ou Text segment): contém a parte de código (instruções) de um programa;
- 2) Data Section (ou data segment): armazena as variáveis globais de um programa;
- 3) Stack (pilha): armazena as variáveis locais de um programa;
- 4) Heap (monte): armazena as variáveis alocadas dinamicamente em um programa;

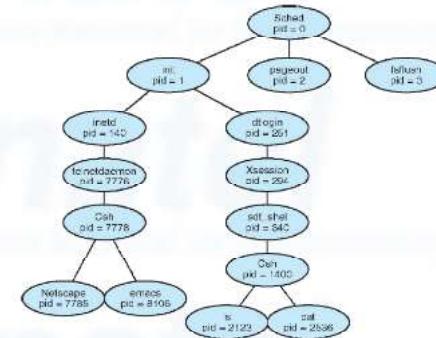


4.1 Visão Geral

4.1.2. Sub-Processos

Um Sub-processo, ou processo-filho é, na verdade, um NOVO PROCESSO;

- Um Sub-processo, é criado por algum outro Processo, denominado Processo-Pai (não existem subprocessos “bastardos”)
- Um Sub-Processo possui, como próprio:
 - o seu process ID;
 - o seu próprio PC;
 - o seu próprio register set;
 - o sua própria stack;
 - o sua própria heap;
 - o sua própria code section (pode ou não ser a mesma do processo-pai);
 - o sua data section;
 - o seus recursos do SO, tais como, open files e signals;



4.1 Visão Geral

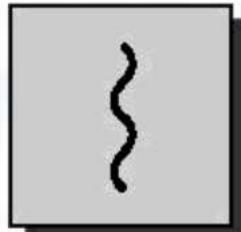
4.1.3. Threads

Uma Thread, ou Kernel Thread ou Lightweight Process (LWP) é uma unidade básica de utilização de CPU.

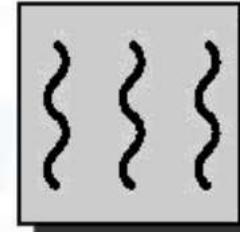
Um Processo pode possuir várias Threads. E caso ele possua, ele pode realizar mais de uma tarefa concorrentemente;

Um Processo tradicional, com apenas uma Thread, é chamado de **heavyweight process**;

Um Processo
Uma Thread



Um Processo
Várias Threads



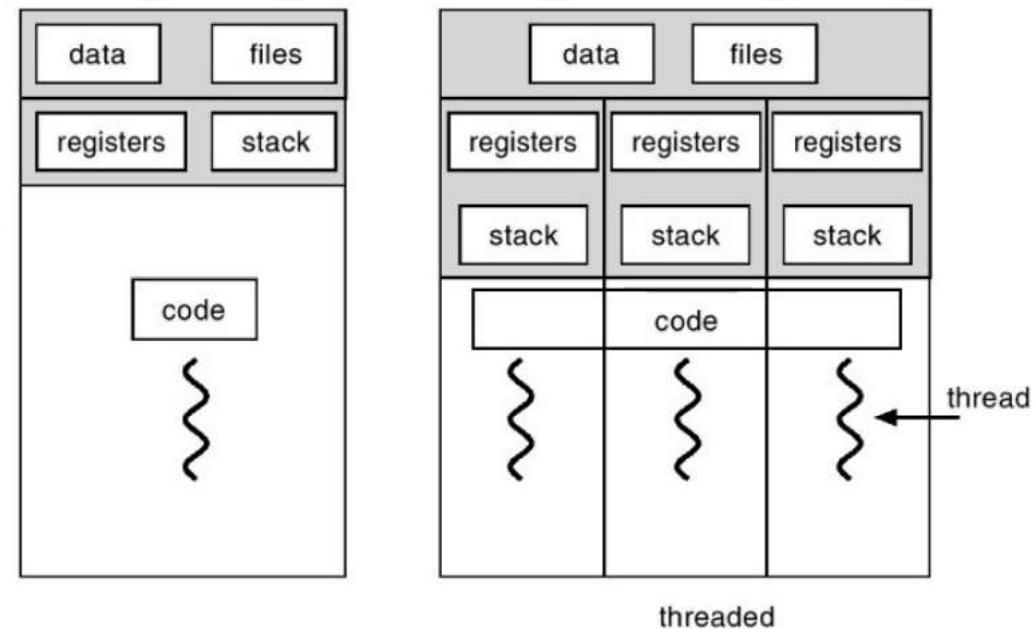
4.1 Visão Geral

Cada Thread possui como próprio:

- Um Thread ID;
- Seu próprio PC;
- Seus próprios conjuntos de registradores;
- Sua própria stack (pilha);
- Sua própria heap;

Uma Thread compartilha com outras Threads de um mesmo Processo:

- Seu Text Section (Code);
- Data Section;
- Arquivos abertos;



4.1 Visão Geral

4.1.3. Threads

Exemplo do uso de Threads para realização de Tarefas Simultâneas:

1) Um Editor de Texto

- Ler um caracter digitado;
- Imprimí-lo na tela;
- Correção ortográfica;
- Salvamento automático...



2) Um Servidor Web

- Receber várias requisições simultâneas;
- Responder assíncronamente;
- Salvar dados das requisições em logs...



Entre infinitas outras aplicações...

4.1 Visão Geral

4.1.3. Threads

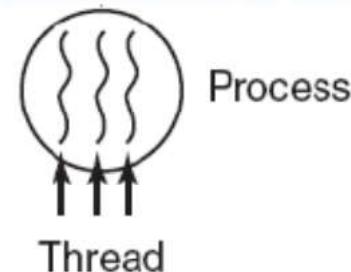
4.1.3.1 Benefícios do uso de Threads

1) Capacidade de Resposta

O uso de várias Threads em uma aplicação pode permitir que um programa **continue a ser executado mesmo se parte dele estiver bloqueada ou executando uma operação demorada;**

2) Compartilhamento de Recursos

Os processos só podem compartilhar recursos através de técnicas como a memória compartilhada ou transmissão de mensagens. No entanto, **por default, os Threads compartilham a memória e os recursos do processo ao qual pertencem.**



4.1 Visão Geral

4.1.3. Threads

4.1.3.1 Benefícios do uso de Threads

3) Economia

Geralmente demora muito mais criar e gerenciar processos do que criar e gerenciar Threads, pois as Threads **compartilham os recursos do processo** ao qual pertencem, logo, é **mais fácil alterar seus contextos**;

4) Escalabilidade

Um Processo com um único Thread só pode ser executado em um Processador. O uso de várias Threads em uma máquina com muitas CPU's **aumenta o paralelismo**;

Mas todas as Threads são iguais?

4.2 Modelos de Geração de Multithreads

Até agora tratamos Threads de um modo geral. No entanto, o suporte a Threads pode ser fornecidos em dois níveis:

- 1) Threads de Usuário;*
- 2) Threads de Kernel;*

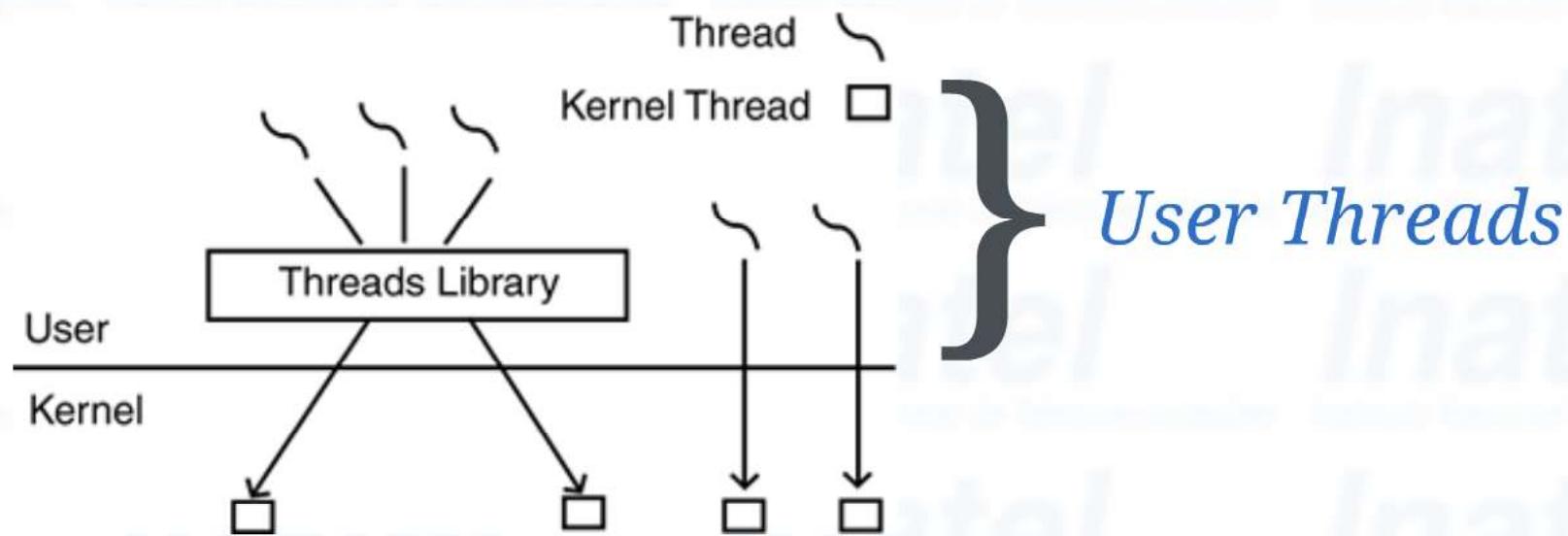
A **Threads de Usuário** são suportadas **acima do Kernel** e gerenciadas **sem suporte dele**. Enquanto as **Threads de Kernel** são suportadas e gerenciadas pelo **Sistema Operacional**.

Vamos entendê-las um pouco mais em detalhes?

4.2 Modelos de Geração de Multithreads

4.2.1 Threads de Usuário

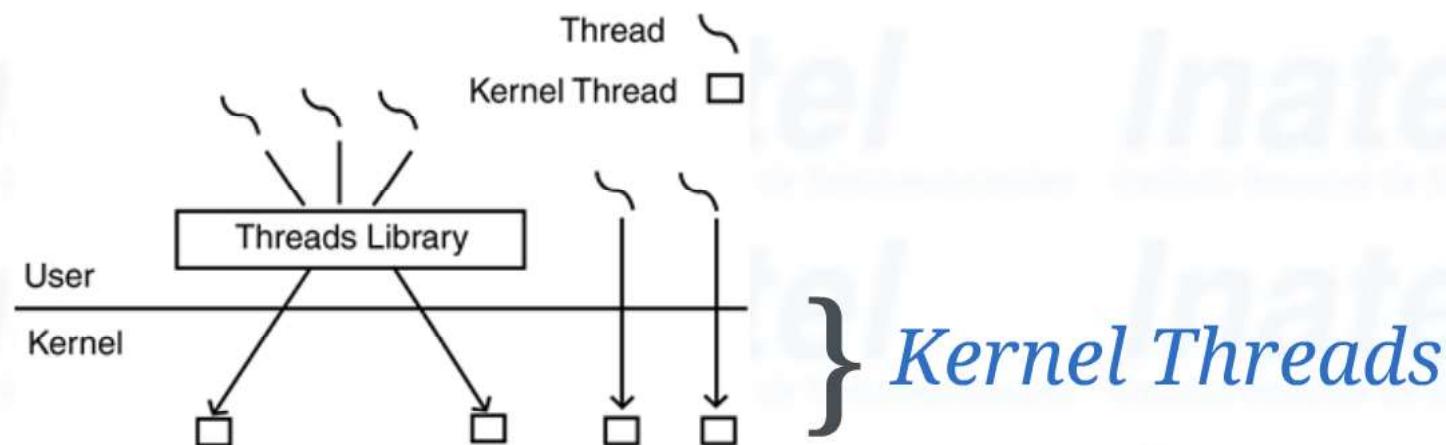
- São implementadas por uma thread library em nível de usuário;
- A thread library possui rotinas para a criação, gerenciamento, remoção, etc. de user Threads;
- Não são “vistas” pelo Kernel;
- Não entram diretamente no escalonamento do SO;



4.2 Modelos de Geração de Multithreads

4.2.2 Threads de Kernel

- São suportados diretamente pelo SO;
- O SO provê recursos para a criação, escalonamento, gerenciamento, remoção, etc. de Kernel threads;
- São “vistas” pelo Kernel;
- Entram no escalonamento (process scheduling) do SO;
- São criados por chamadas a System Calls (clone());



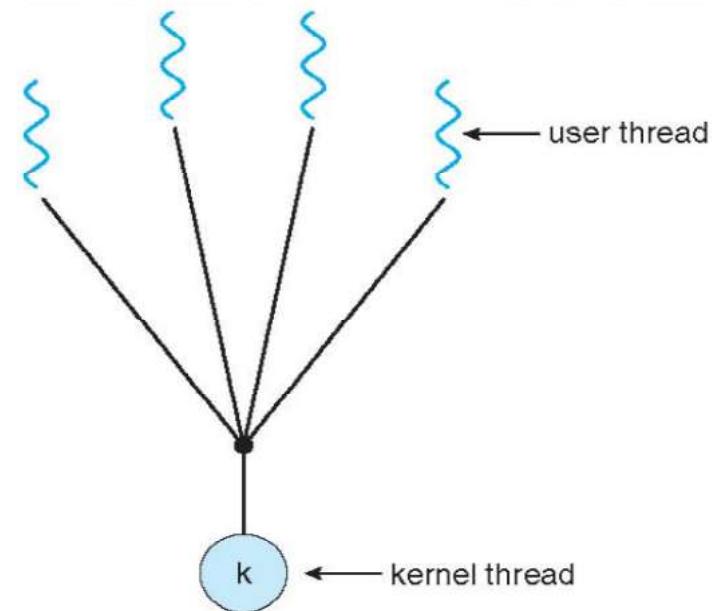
4.2 Modelos de Geração de Multithreads

4.2.3. Relacionamento entre Threads de Usuário e Kernel

No final das contas, deve existir um relacionamento entre os Threads de usuários e os Threads de Kernel. Examinaremos 3 (três) maneiras comuns de estabelecer esse relacionamento.

4.2.3.1. Modelo MUITOS para UM

- Mapeia muitas Threads de nível de usuário para um Thread de Kernel;
- O gerenciamento das User Threads é feita pela biblioteca de Threads no espaço de usuário;
- Desvantagem: O Processo inteiro será bloqueado se uma User Thread fizer uma System Call bloqueadora;

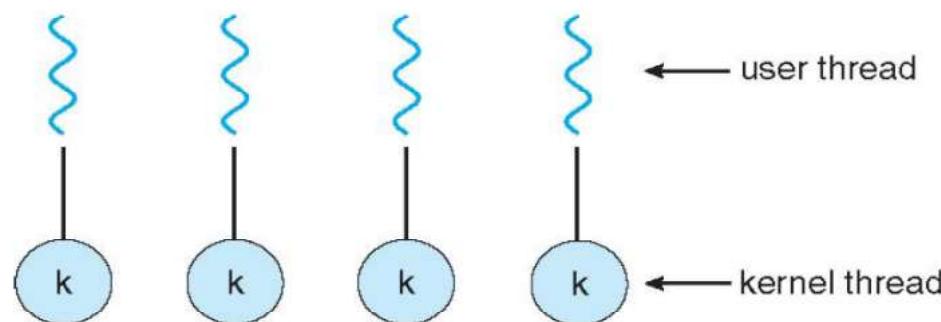


4.2 Modelos de Geração de Multithreads

4.2.3. Relacionamento Threads Usuário e Kernel

4.2.3.2. Modelo UM para UM

- Mapeia cada Thread de usuário para uma Thread de Kernel;
- Mesmo que uma User Thread bloqueie, os demais continuam executando;
- Podem executar em paralelo (em Sistemas Multicore);
- O [Linux](#), junto com a família de Sistemas Operacionais [Windows](#), implementam o modelo um-para-um;
- Desvantagem: [alto overhead para criação de Kernel Threads](#);

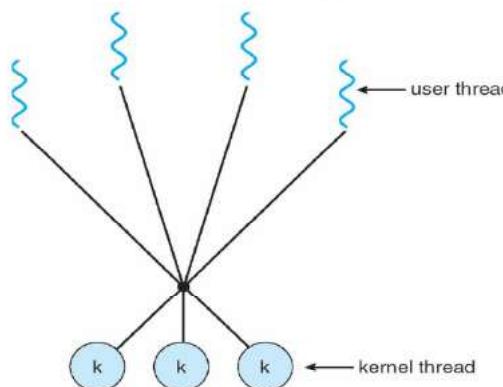


4.2 Modelos de Geração de Multithreads

4.2.3. Relacionamento Threads Usuário e Kernel

4.2.3.3. Modelo MUITOS para MUITOS

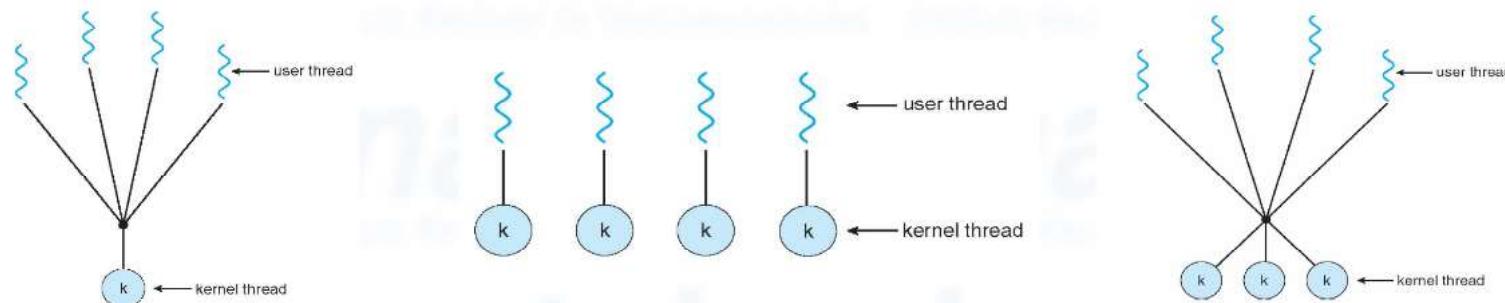
- Muitas User Threads são multiplexadas por um número menor ou igual de Kernel Threads;
- Mesmo que um User Thread bloqueie, os demais continuam executando;
- Aqui os User Threads podem ser executados em paralelo (em sistemas multicore);
- O usuário cria quantas User Threads quiser, e elas vão sendo distribuídas para as Kernel Threads;
- Os Sistemas Operacionais IRIX e HP-UX utilizam deste modelo.



4.2 Modelos de Geração de Multithreads

4.2.4. Comparativo entre os Modelos

- O Modelo Muitos-para-um permite que o desenvolvedor crie quantos Threads de Usuário quiser, mas não há um ganho de concorrência porque o Kernel só pode agendar um Thread de cada vez;
- O Modelo Um-para-um permite maior concorrência, mas o desenvolvedor precisa tomar cuidado para não criar Threads demais dentro de uma aplicação;
- O Modelo Muitos-para-muitos não sofre de nenhuma dessas deficiências;



4.3 Bibliotecas de Threads

Uma biblioteca de Threads fornece ao programador uma API para a criação e gerenciamento de Threads.

Há duas maneiras principais de implementar uma biblioteca de Threads:

- 1) Fornecer uma **biblioteca inteiramente no espaço do usuário** sem suporte do Kernel;
- 2) Fornecer uma **biblioteca no nível de Kernel**, com suporte direto do Sistema Operacional;

Três Bibliotecas de Threads são mais usadas recentemente:

- Pthreads (Permite trabalhar com nível de usuário e Kernel. Ex: Linux);
- Win32 (Apenas nível de Kernel);
- Java (Dependente do SO);



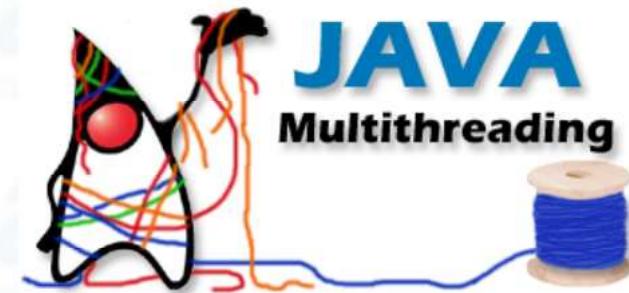
4.3 Bibliotecas de Threads

TRABALHO PRÁTICO 1

Agora que relembramos de alguns conceitos importantes do uso de Threads em Java, crie uma aplicação que faça uso deste recurso para resolver algum problema, seja ele real ou lúdico.

IMPORTANTE:

- Use de seus conhecimentos e criatividade :)
- O Trabalho pode ser individual ou em dupla;
- Fique atento à data de apresentação proposta pelo professor;
- Este conhecimento também será essencial para realização do próximo trabalho prático;



HORA DO

Kahoot!



ACESSE:

<https://kahoot.it>

**Fim
do
Capítulo 4**



**Exercícios
Cap.3 e 4**

