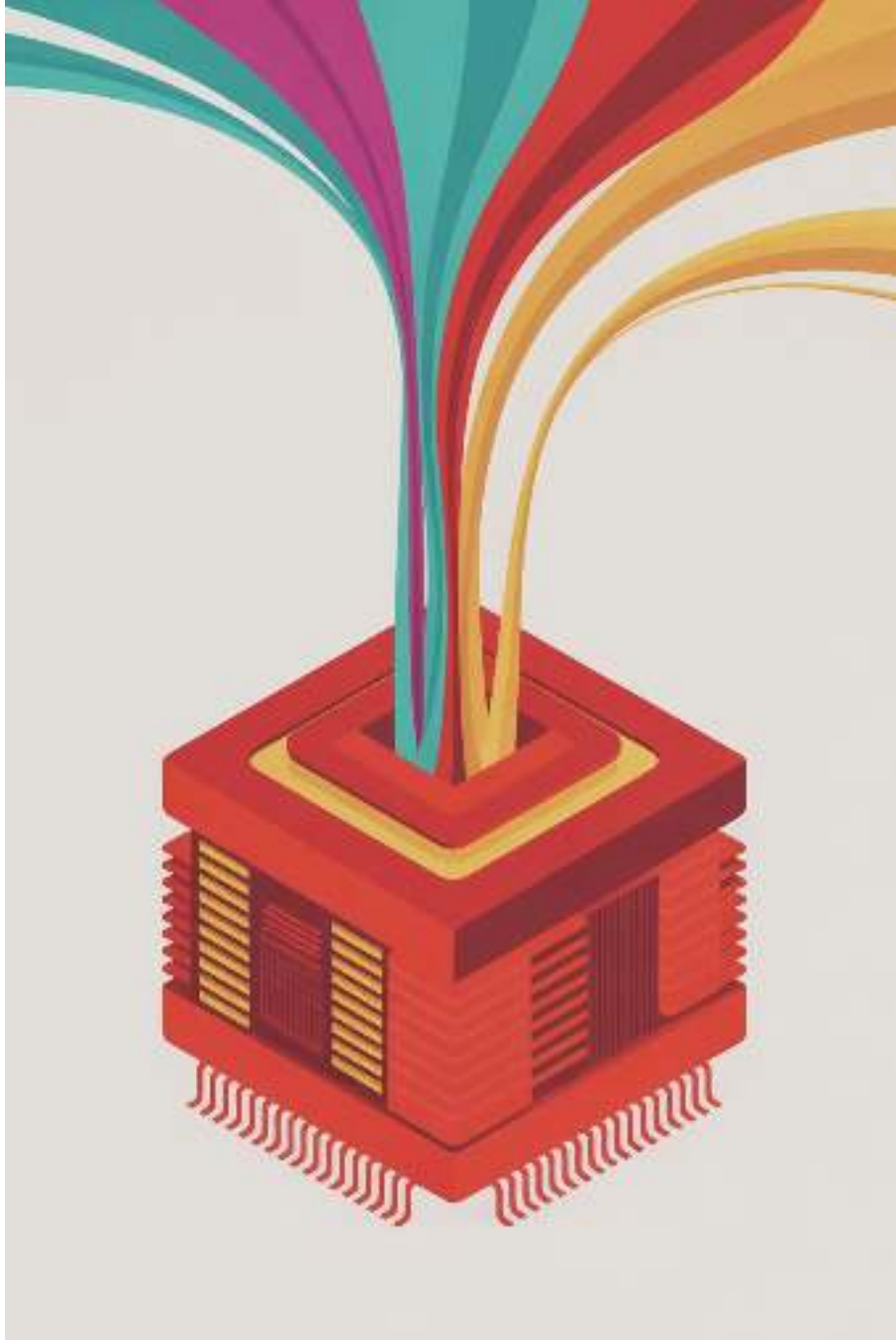
Implementação de Tarefas em Sistemas Operacionais

Este curso explora as técnicas e conceitos fundamentais para implementação de tarefas em sistemas operacionais modernos. Entenderemos como o SO gerencia processos e threads, permitindo a execução simultânea de múltiplas atividades.



Agenda

01

Contextos de Tarefas

O que compõe o estado interno de uma tarefa e como é representado

02

Trocas de Contexto

Mecanismos e políticas para alternar entre tarefas

03

Processos

Unidades de contexto com recursos isolados

04

Threads

Fluxos de execução independentes dentro de processos

05

Processos vs Threads

Comparação e critérios de escolha para diferentes aplicações

Tarefas em Sistemas Operacionais

Uma **tarefa** é a unidade básica de atividade dentro de um sistema operacional, representando um fluxo de execução independente.

Implementações comuns de tarefas:

- Processos
- Threads
- Transações
- Jobs

Neste capítulo, exploraremos as estruturas de dados e operações necessárias para implementar e gerenciar tarefas de forma transparente e eficiente.

Contextos de Tarefas

O **contexto** de uma tarefa representa seu estado interno em um determinado instante. Ele inclui:

Estado do Processador

Contador de programa (PC), apontador de pilha (SP), registradores e flags do processador

Áreas de Memória

Segmentos de código, dados, pilha e heap alocados para a tarefa

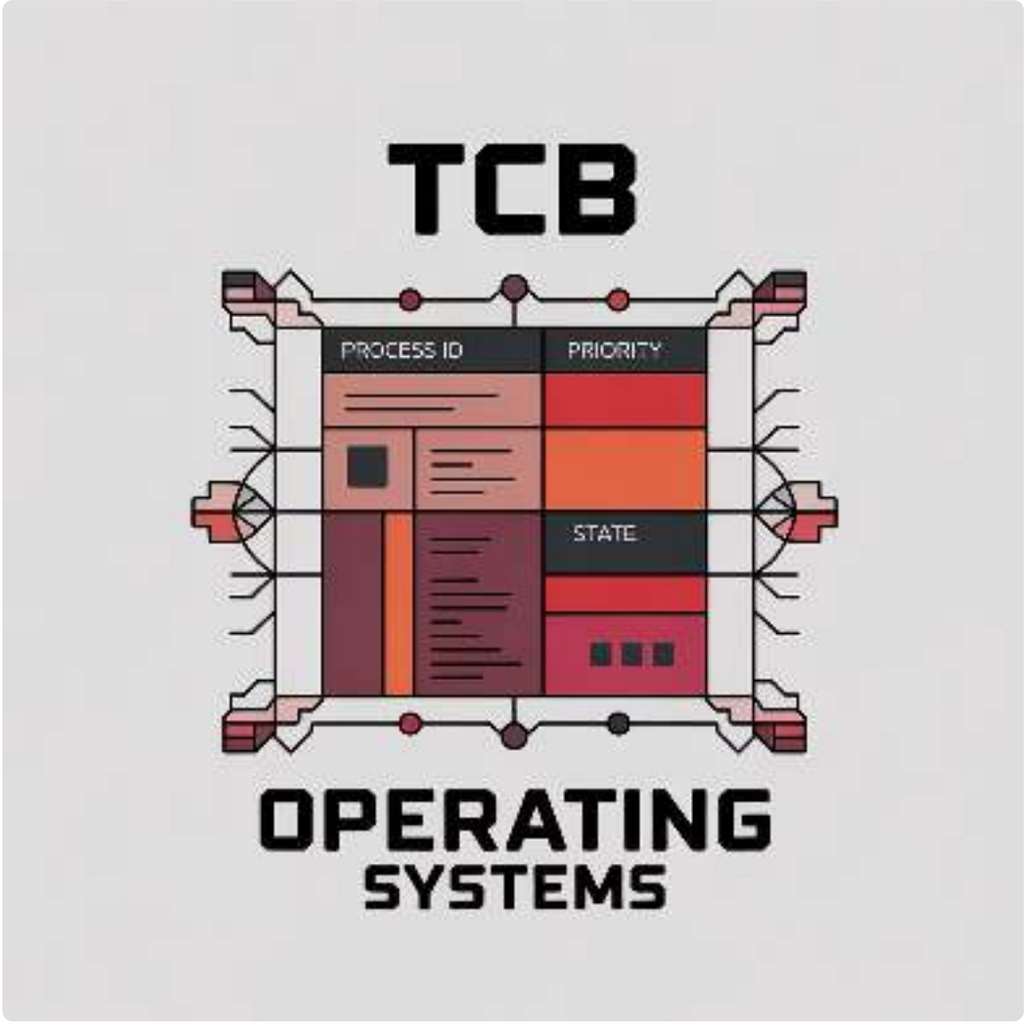
Recursos do Sistema

Arquivos abertos, conexões de rede, semáforos e outros recursos utilizados pela tarefa

O contexto muda constantemente à medida que a execução da tarefa evolui.

TCB - Task Control Block

Cada tarefa no sistema é representada por um **descritor de tarefa** ou **TCB (Task Control Block)**, uma estrutura de dados no núcleo que contém:



- Identificador único da tarefa
- Estado atual (nova, pronta, executando, suspensa, terminada)
- Informações de contexto do processador
- Lista de áreas de memória utilizadas
- Lista de recursos alocados (arquivos, conexões)
- Dados de gerência e contabilização (prioridade, usuário, tempos)

O núcleo organiza os TCBs em listas ou vetores para facilitar o gerenciamento das tarefas.

Trocas de Contexto

A **troca de contexto** é o processo de suspender a execução de uma tarefa e reativar outra. Requer operações para:

Salvar o contexto atual

Armazenar o estado da tarefa suspensa em seu TCB (registradores, PC, SP)

Escolher a próxima tarefa

Determinar qual tarefa deverá receber o processador a seguir

Restaurar o contexto

Carregar no processador o estado da tarefa que será ativada

É uma operação delicada, geralmente codificada em linguagem de máquina e específica para cada arquitetura de processador.

Mecanismos e Políticas na Troca de Contexto

Despachante (Dispatcher)

Implementa os **mecanismos** da gerência de tarefas:

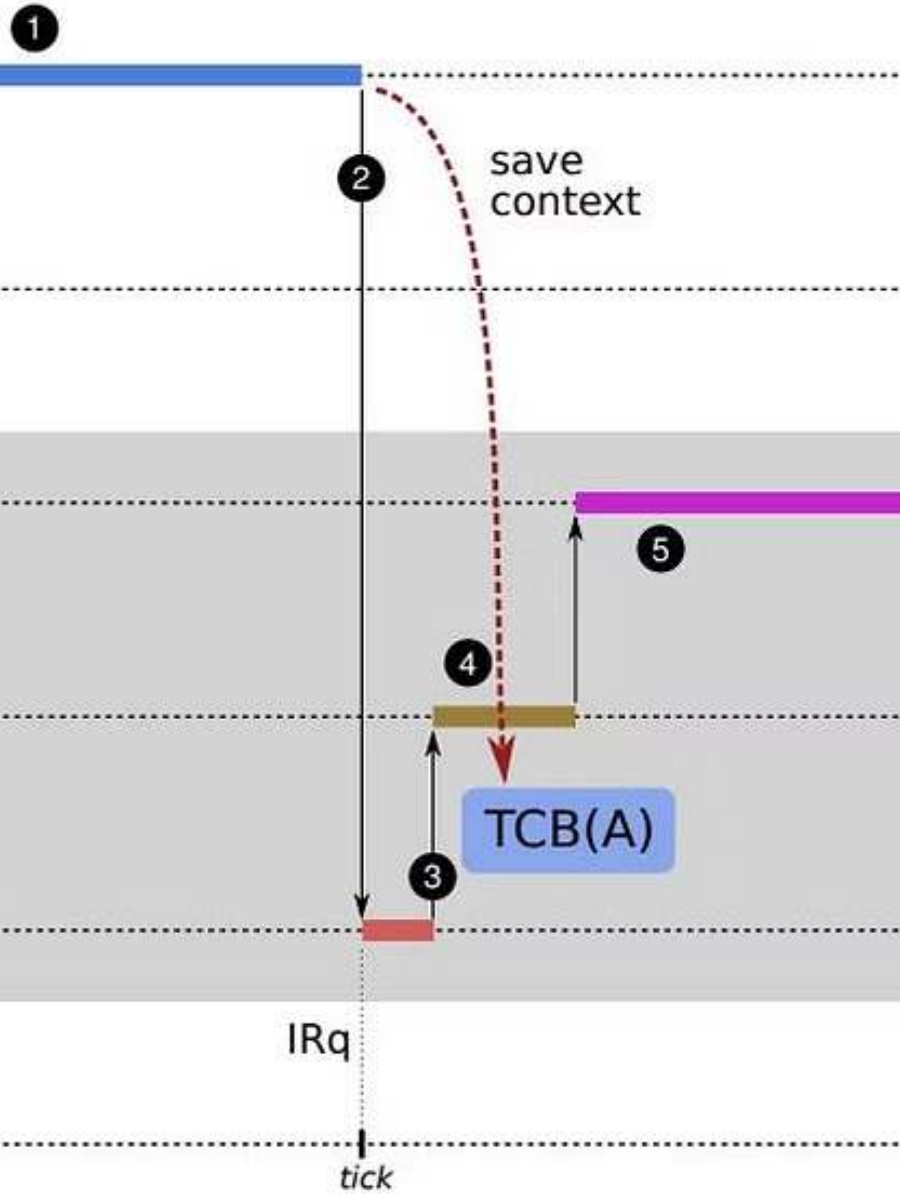
- Salvar e restaurar contextos
- Atualizar informações nos TCBs
- Transferir o controle entre tarefas

Escalonador (Scheduler)

Implementa as **políticas** da gerência de tarefas:

- Decidir qual tarefa executará a seguir
- Considerar prioridades, tempos e outros fatores
- Garantir distribuição justa de recursos

Esta separação entre mecanismos e políticas é um princípio fundamental no projeto de sistemas operacionais.



Passos de uma Troca de Contexto

Sequência de eventos durante uma troca de contexto:

1. A tarefa A está em execução
2. Ocorre uma interrupção (temporizador, dispositivo, chamada de sistema ou exceção)
3. A rotina de tratamento ativa o despachante
4. O despachante salva o estado da tarefa A em seu TCB
5. O escalonador é consultado para escolher a próxima tarefa (B)
6. O despachante restaura o estado da tarefa B e a reativa

Eficiência da Troca de Contexto

A frequência e duração das trocas de contexto afetam diretamente a eficiência do sistema.

$$\mathcal{E} = \frac{t_q}{t_q + t_{tc}}$$

Onde:

- □ = eficiência do uso do processador
- tq = duração média do quantum
- ttc = duração média da troca de contexto

Exemplo 1

Quantum = 10ms

Troca = 100µs

Eficiência = 99%

Exemplo 2

Quantum = 1ms

Troca = 100µs

Eficiência = 91%

A eficiência é influenciada pela carga do sistema (número de tarefas) e pelo perfil das aplicações (uso de CPU vs. E/S).

Processos: Conceito

Historicamente, um processo era definido como uma tarefa com seus recursos, em área de memória isolada. Atualmente, devemos entender o processo como:

Uma unidade de contexto

Um **contêiner de recursos** utilizados por uma ou mais tarefas para sua execução:

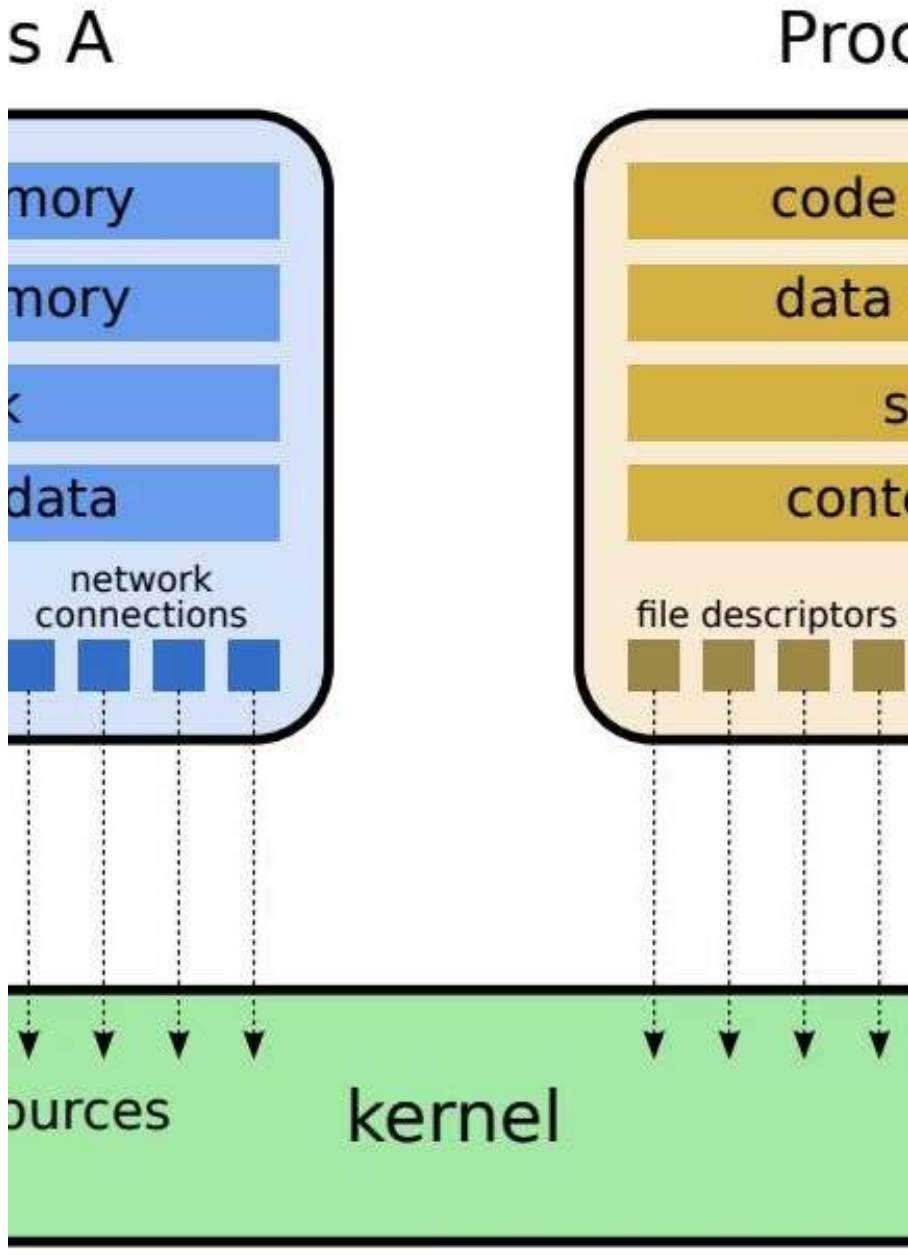
- Áreas de memória (código, dados, pilha)
- Informações de contexto
- Descritores de recursos do núcleo (arquivos abertos, conexões)

Processos são isolados entre si pelos mecanismos de proteção do hardware, impedindo acessos indevidos.

Processo como Contêiner de Recursos

Um processo pode conter múltiplas tarefas (threads) que compartilham os mesmos recursos.

Por padrão, sistemas operacionais atuais associam uma única tarefa a cada processo, correspondendo a um programa sequencial. Desenvolvedores podem criar tarefas adicionais (threads) conforme necessário.



PCB - Process Control Block

O núcleo mantém descritores de processos (PCBs) para armazenar informações sobre os processos ativos:

Identificação

- PID (Process IDentifier)
- Usuário proprietário
- Prioridade

Gerência

- Data de início
- Tempo de processamento
- Estado atual

Recursos

- Caminho do executável
- Áreas de memória
- Arquivos e conexões

Os PCBs são utilizados pelo núcleo para gerenciar a execução e o compartilhamento de recursos entre processos.

Exemplo: Processos no Linux

O comando `top` no Linux permite visualizar os processos ativos e suas informações:

- PID: identificador único do processo
- USUÁRIO: proprietário do processo
- PR/NI: prioridade e valor "nice"
- VIRT/RES/SHR: memória virtual, residente e compartilhada
- %CPU/%MEM: uso de processador e memória
- TIME+: tempo de CPU acumulado
- COMMAND: comando que iniciou o processo

Gestão de Processos

Os sistemas operacionais disponibilizam chamadas de sistema para gerenciar processos:

Ação	Windows	Linux
Criar um novo processo	CreateProcess()	fork(), execve()
Encerrar o processo corrente	ExitProcess()	exit()
Encerrar outro processo	TerminateProcess()	kill()
Obter o ID do processo corrente	GetCurrentProcessId()	getpid()

Cada sistema operacional implementa seu próprio conjunto de chamadas de sistema, com semânticas específicas.

Criação de Processos no UNIX/Linux

Em sistemas UNIX/Linux, a criação de processos ocorre em duas etapas:

Etapa 1: fork()

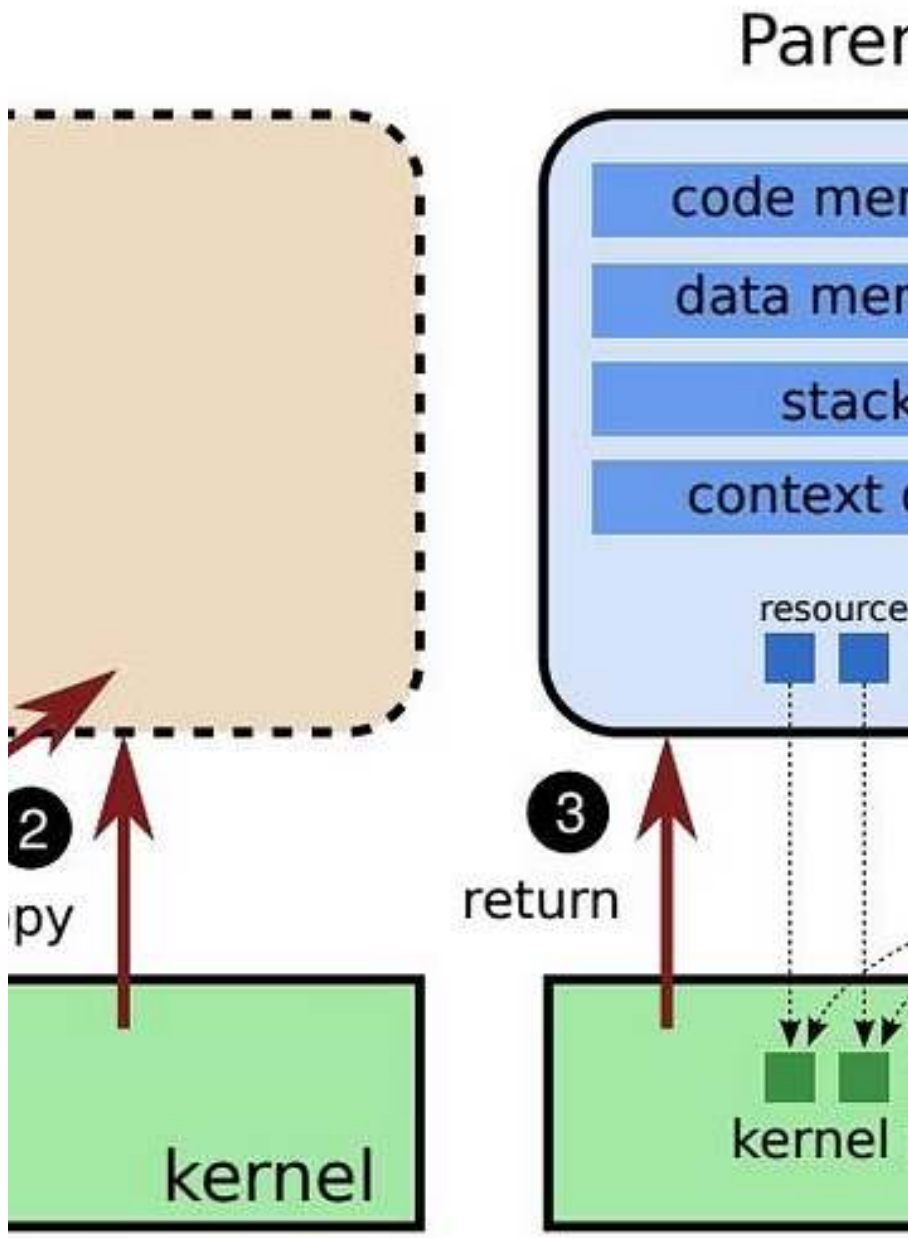
Cria uma réplica do processo atual:

- Copia o espaço de memória do processo pai
- Duplica os descritores de recursos
- Retorna em ambos os processos (pai e filho)
- Valor de retorno diferente para identificação

Etapa 2: execve()

Substitui o código do processo por um novo programa:

- Carrega o executável do arquivo especificado
- Inicializa o novo programa
- Mantém os descritores de recursos abertos
- Não retorna se for bem-sucedido



Exemplo de Criação de Processo no Linux

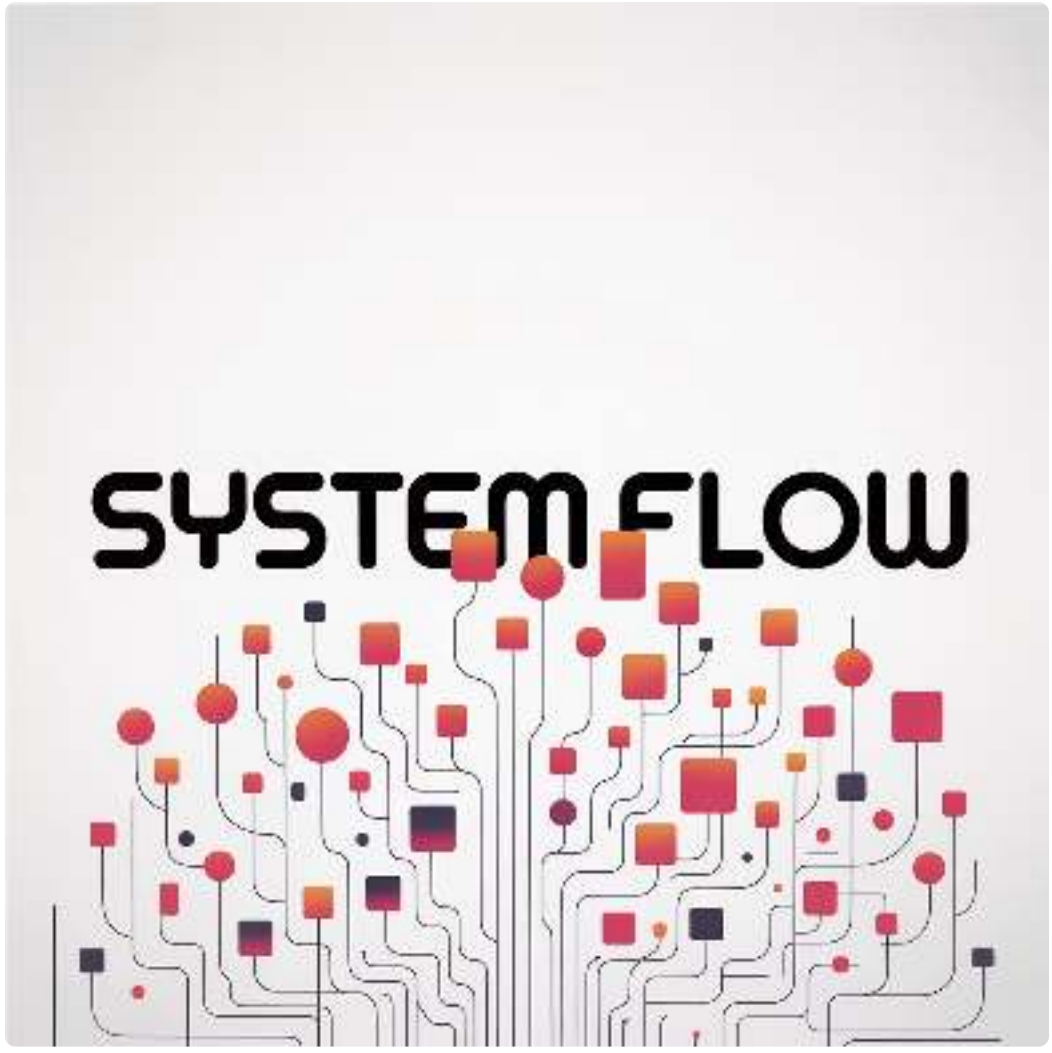
```
#include
#include
#include
#include
#include
#include

int main (int argc, char *argv[], char *envp[])
{
 int pid; // identificador de processo

 pid = fork(); // replicação do processo

 if (pid < 0) // fork funcionou?
 {
 perror("Erro: "); // não, encerra este processo
 exit(-1);
 }
 else // sim, fork funcionou
 if (pid > 0) // sou o processo pai?
 wait(0); // sim, vou esperar meu filho concluir
 else // não, sou o processo filho
 {
 // carrega outro código binário para executar
 execve("/bin/date", argv, envp);
 perror("Erro: "); // execve não funcionou
 }
 printf("Tchau!\n");
 exit(0); // encerra este processo
}
```

Hierarquia de Processos

Em sistemas UNIX/Linux, os processos formam uma **árvore hierárquica**:

- Cada processo é filho de outro processo
- O processo inicial (PID 1) é a raiz da árvore
- Quando um processo encerra, seus filhos são notificados
- Processos relacionados formam sub-árvores

Em sistemas Windows, todos os processos têm o mesmo nível hierárquico, sem distinção entre pais e filhos.



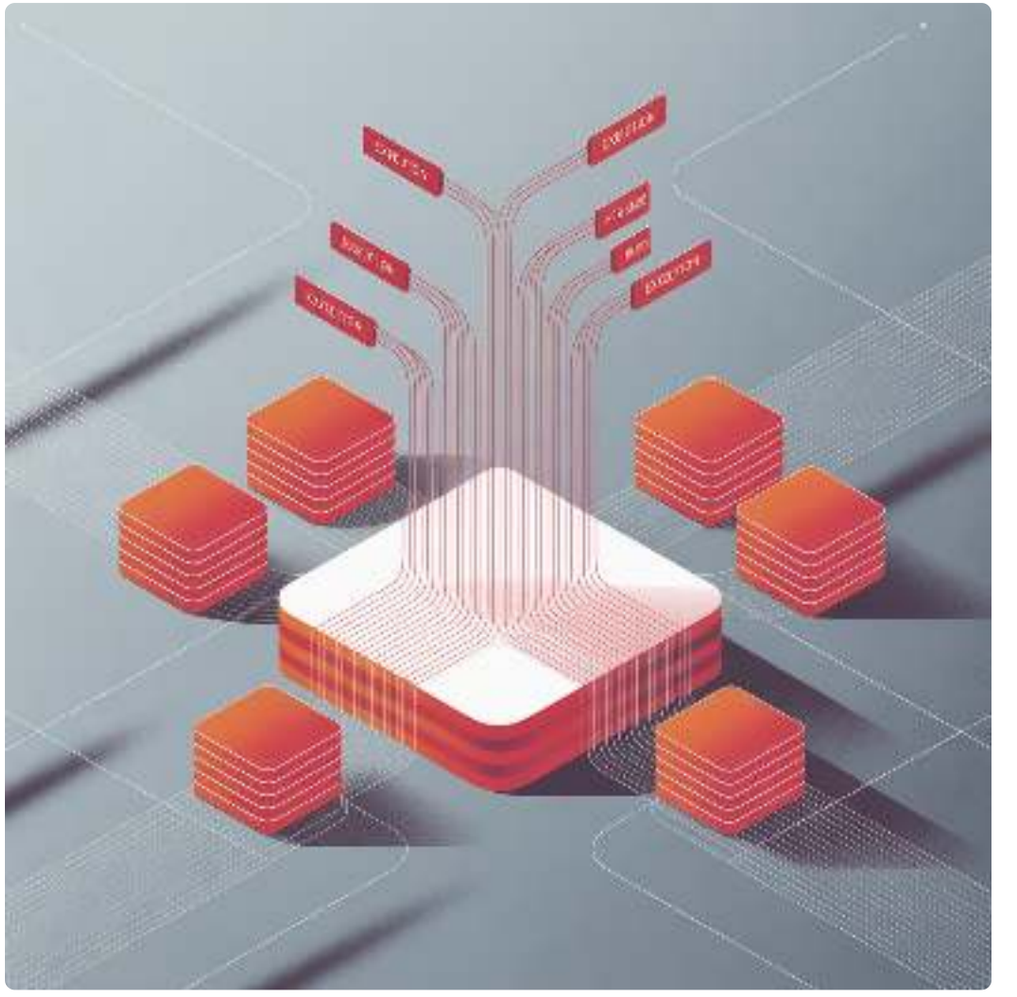
`pstree` no Linux permite visualizar a árvore de processos do sistema.

Threads: Definição

Um fluxo de execução independente

Uma thread é caracterizada por:

- Código em execução
- Contexto local (TLS - Thread Local Storage):
  - Registradores do processador
  - Área de pilha para variáveis locais
- Compartilhamento de recursos com outras threads do mesmo processo



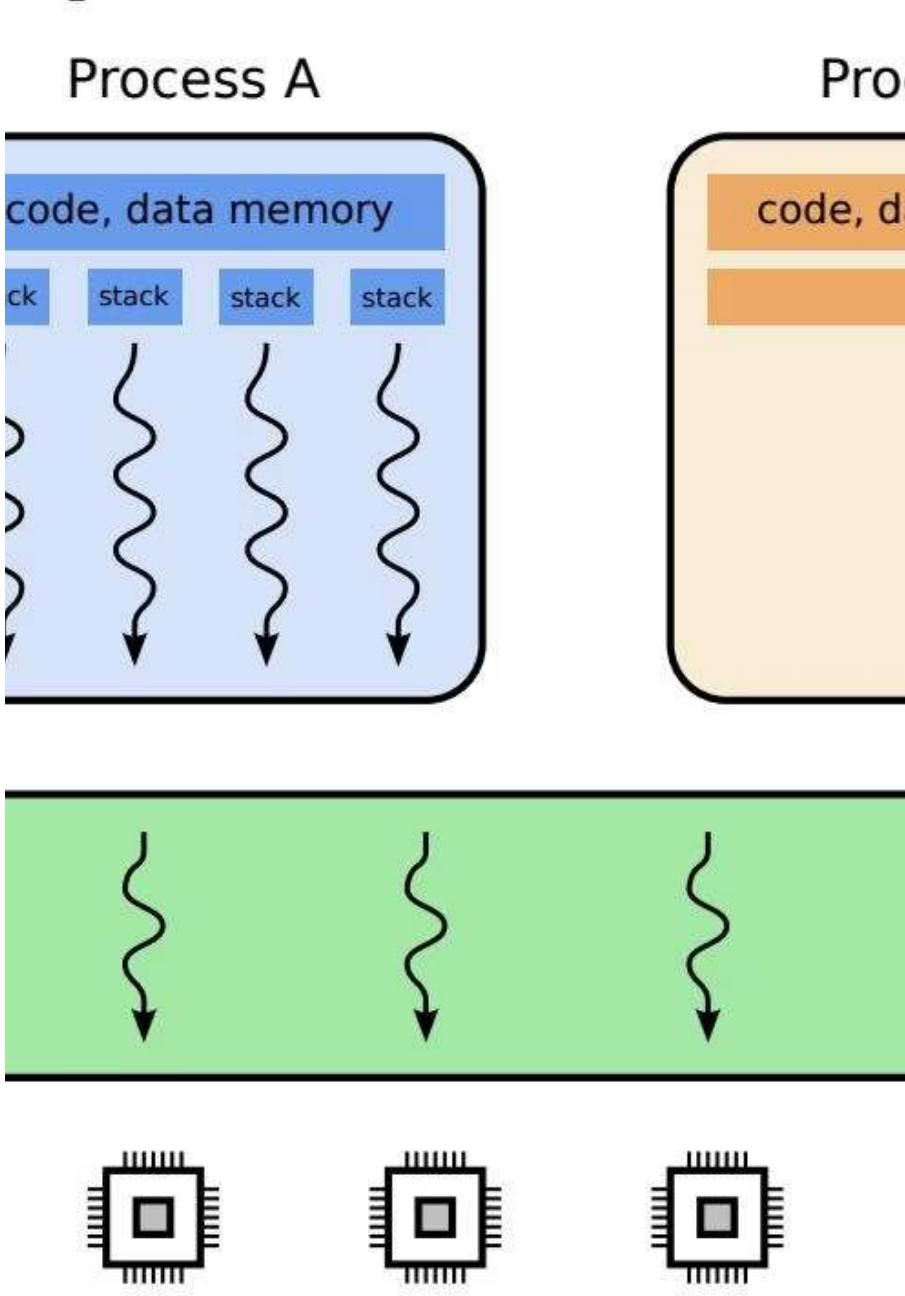
As threads surgiram da necessidade de suportar aplicações mais complexas, com múltiplas tarefas concorrentes operando sobre os mesmos dados.

Threads de Usuário e de Núcleo

Diferentes tipos de threads em um sistema operacional:

- **Threads de usuário**: Associadas a processos de aplicações, representam fluxos de execução dentro de programas de usuário
- **Threads de núcleo**: Representam threads de usuário dentro do núcleo e também incluem atividades internas do sistema operacional (drivers, tarefas de gerência)

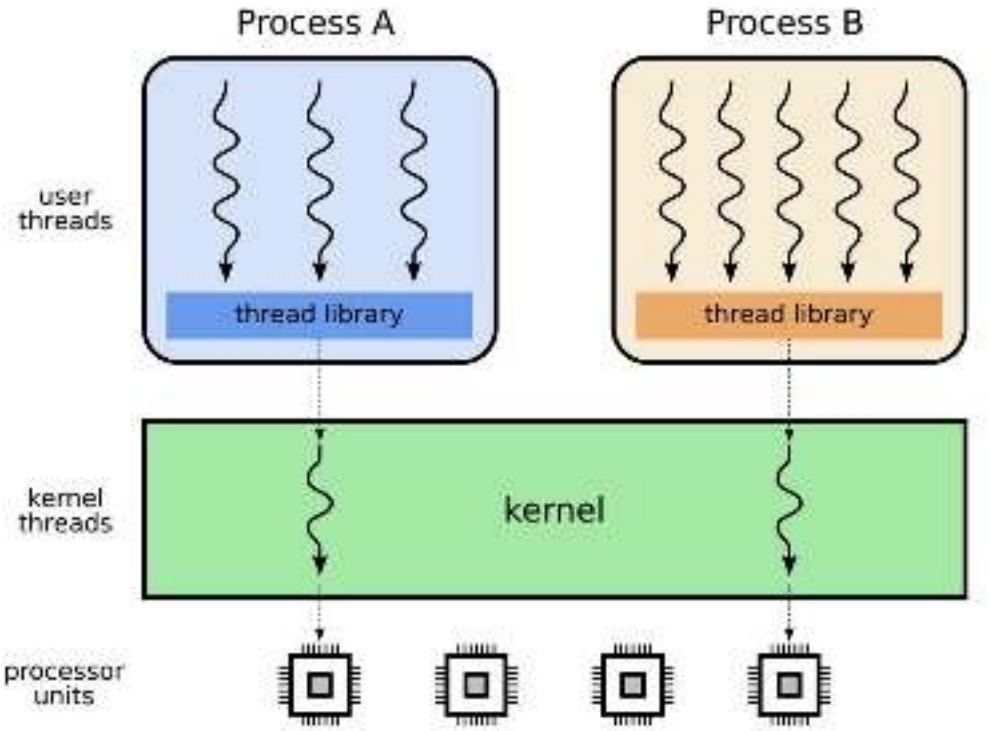
Na figura, o processo A tem várias threads, enquanto o processo B é sequencial (tem uma única thread).



Modelo de Threads N:1

No modelo **N:1** (ou *green threads*):

- N threads dentro do processo são mapeadas em uma única thread no núcleo
- Implementado por biblioteca no espaço do usuário
- O núcleo vê apenas um fluxo de execução por processo
- Trocas de contexto entre threads são rápidas (sem envolvimento do núcleo)



Exemplos: GNU Portable Threads, Microsoft UMS, Green Threads (Java)

Limitações do Modelo N:1

Bloqueio Total

Se uma thread solicitar uma operação de E/S bloqueante, todas as threads do processo ficarão bloqueadas até a conclusão da operação

Distribuição Injusta

Um processo com 100 threads recebe o mesmo tempo de processador que outro com apenas uma thread

Sem Paralelismo Real

Threads do mesmo processo não podem executar em paralelo, mesmo em sistemas com múltiplos processadores ou cores

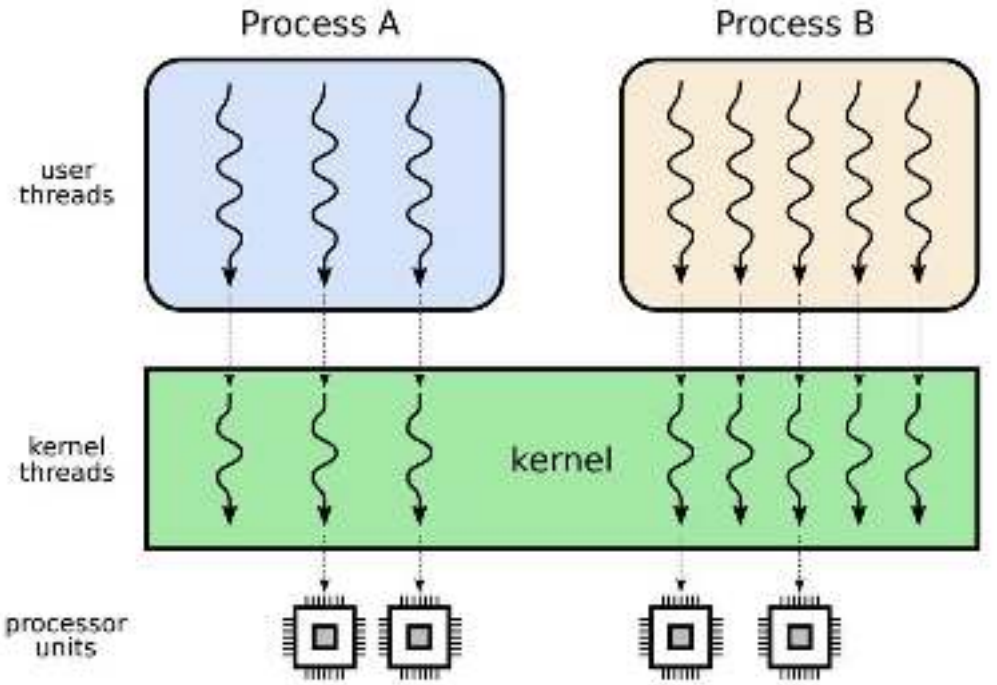
Apesar dessas limitações, o modelo N:1 é leve e escalável, sendo útil para aplicações com muitas threads, como jogos ou simulações.

Modelo de Threads 1:1

No modelo **1:1**:

- Cada thread de usuário corresponde a uma thread de núcleo
- Implementado diretamente pelo núcleo do sistema
- O escalonador do núcleo gerencia todas as threads
- Permite paralelismo real em sistemas multiprocessador

Exemplos: Windows NT e descendentes, maioria dos sistemas UNIX/Linux modernos



Modelo mais comum nos sistemas operacionais atuais, adequado para maioria das aplicações interativas e servidores de rede.

Vantagens e Limitações do Modelo 1:1

Vantagens

- Se uma thread bloqueia, as demais continuam executando
- Distribuição justa do processador entre todas as threads
- Threads do mesmo processo podem executar em paralelo
- Melhor desempenho para aplicações interativas

Limitações

- Maior sobrecarga do núcleo para gerenciar muitas threads
- Trocas de contexto mais lentas (envolvem o núcleo)
- Menor escalabilidade com grande número de threads
- Inviável para aplicações com milhares de threads

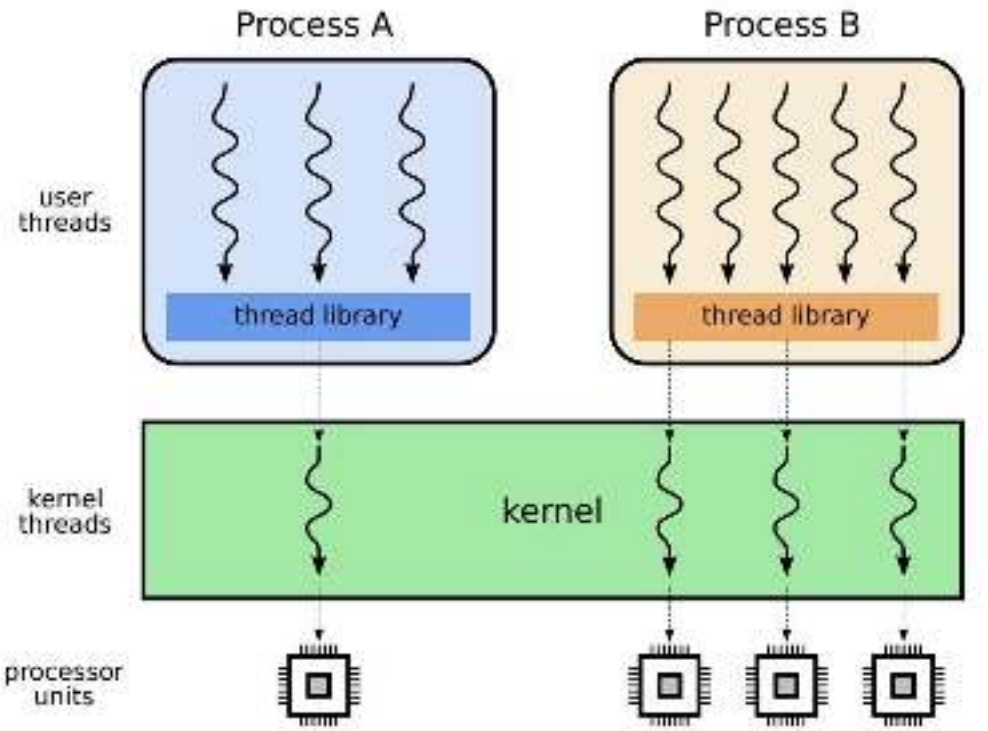
A escalabilidade limitada pode comprometer o desempenho de grandes servidores Web e simulações de grande porte.

Modelo de Threads N:M

No modelo híbrido **N:M**:

- N threads de usuário são mapeadas em M < N threads de núcleo
- Implementado tanto no espaço do usuário quanto no núcleo
- Thread pool gerenciado dinamicamente
- Combina vantagens dos modelos anteriores

Exemplos: Solaris, FreeBSD KSE



O thread pool geralmente contém uma thread para cada thread de usuário bloqueada, mais uma thread para cada processador disponível.

Comparação entre os Modelos de Threads

Característica	N:1	1:1	N:M
Implementação	Biblioteca (usuário)	Núcleo	Ambos
Complexidade	Baixa	Média	Alta
Custo de gerência	Baixo	Médio	Alto
Escalabilidade	Alta	Baixa	Alta
Paralelismo	Não	Sim	Sim
Troca de contexto	Rápida	Lenta	Rápida

A escolha do modelo depende das características da aplicação e dos requisitos de desempenho, paralelismo e escalabilidade.

Programando com Threads: POSIX Threads

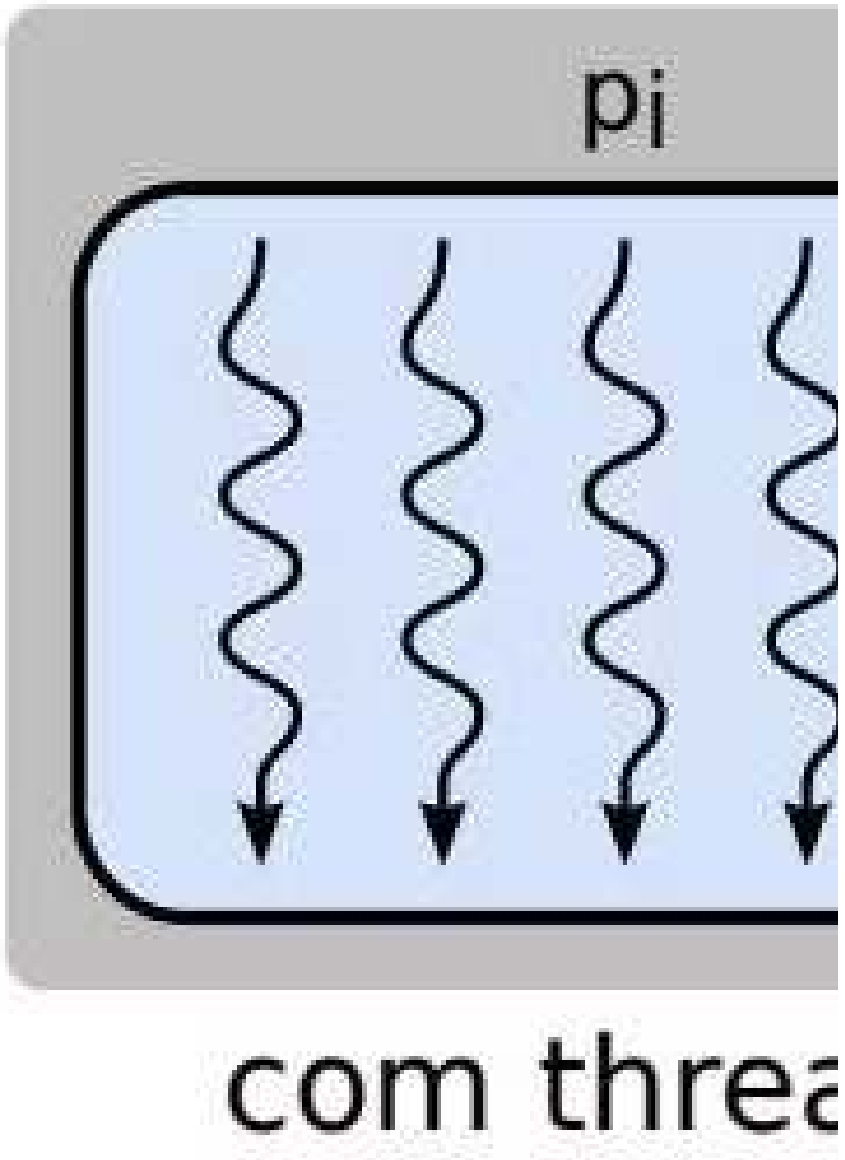
O padrão IEEE POSIX 1003.1c (PThreads) define uma interface padronizada para uso de threads em C:

```
// Exemplo de uso de threads Posix em C no Linux
// Compilar com gcc exemplo.c -o exemplo -lpthread

#include
#include
#include
#include
#include

#define NUM_THREADS 5

// cada thread vai executar esta função
void *print_hello(void *threadid)
{
 printf("%ld: Hello World!\n", (long)threadid);
 sleep(5);
 printf("%ld: Bye bye World!\n", (long)threadid);
 pthread_exit(NULL); // encerra esta thread
}

// thread "main" (vai criar as demais threads)
int main(int argc, char *argv[])
{
 pthread_t thread[NUM_THREADS];
 long status, i;

 // cria as demais threads
 for(i = 0; i < NUM_THREADS; i++)
 {
 printf("Creating thread %ld\n", i);
 status = pthread_create(&thread[i], NULL, print_hello, (void *)i);

 if (status) // ocorreu um erro
 {
 perror("pthread_create");
 exit(-1);
 }
 }

 // encerra a thread "main"
 pthread_exit(NULL);
}
```

Programando com Threads: Java

```
// save as MyThread.java; javac MyThread.java; java MyThread

public class MyThread extends Thread {
   int threadID;

   private static final int NUM_THREADS = 5;

   MyThread(int ID) {
      threadID = ID;
   }

   // corpo de cada thread
   public void run() {
      System.out.println(threadID + ": Hello World!");
      try {
         Thread.sleep(5000);
      }
      catch (InterruptedException e) {
         e.printStackTrace();
      }
      System.out.println(threadID + ": Bye bye World!");
   }

   public static void main(String args[]) {

      MyThread[] t = new MyThread[NUM_THREADS];

      // cria as threads
      for (int i = 0; i < NUM_THREADS; i++) {
         t[i] = new MyThread(i);
      }

      // inicia a execução das threads
      for (int i = 0; i < NUM_THREADS; i++) {
         t[i].start();
      }
   }
}
```

Java oferece suporte nativo a threads, com uma API mais simples e orientada a objetos.



Processos versus Threads

Três abordagens para implementar sistemas com múltiplas tarefas:

1. **Um processo para cada tarefa**: Maior robustez e isolamento, mas comunicação mais complexa
2. **Um processo com múltiplas threads**: Compartilhamento simples de dados, mais ágil, mas menor robustez
3. **Abordagem híbrida**: Múltiplos processos, cada um com múltiplas threads, combinando robustez e desempenho

Comparação: Processos vs Threads

Característica	Com processos	Com threads (1:1)	Híbrido
Custo de criação	Alto	Baixo	Médio
Troca de contexto	Lenta	Rápida	Variável
Uso de memória	Alto	Baixo	Médio
Compartilhamento de dados	Complexo (IPC)	Simples (memória compartilhada)	Ambos
Robustez	Alta	Baixa	Média
Segurança	Alta	Baixa	Alta

A escolha depende dos requisitos específicos da aplicação: desempenho, robustez, segurança e complexidade de desenvolvimento.

Exemplos de Aplicações e suas Estratégias

Apache HTTP Server

1.x: Modelo baseado em processos
2.x: Modelo híbrido com processos e threads

Navegadores Chrome/Firefox

Modelo híbrido: cada aba em um processo separado, com múltiplas threads para renderização e interação

Banco de Dados Oracle

Modelo híbrido: processos para isolamento e segurança, threads para paralelismo eficiente

Sistemas modernos tendem a adotar a abordagem híbrida para balancear robustez, segurança e desempenho.