FACULDADE DE TECNOLOGIA DE PRAIA GRANDE
CURSO SUPERIOR TECNOLÓGICO DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

ANDRESSA MACHADO, DOUGLAS MASUZZO, GABRIELA HORMINA, GABRIEL KRIEGER E HENRY BITTENBINDER

THREADS

Diferenças, Vantagens e Fundamentos

PRAIA GRANDE - SP
Outubro/2025

ANDRESSA MACHADO, DOUGLAS MASUZZO, GABRIELA HORMINA, GABRIEL KRIEGER E HENRY BITTENBINDER

THREADS

Diferenças, Vantagens e Fundamentos

PRAIA GRANDE - SP

Outubro/2025

SUMÁRIO

1. INTRODUÇÃO

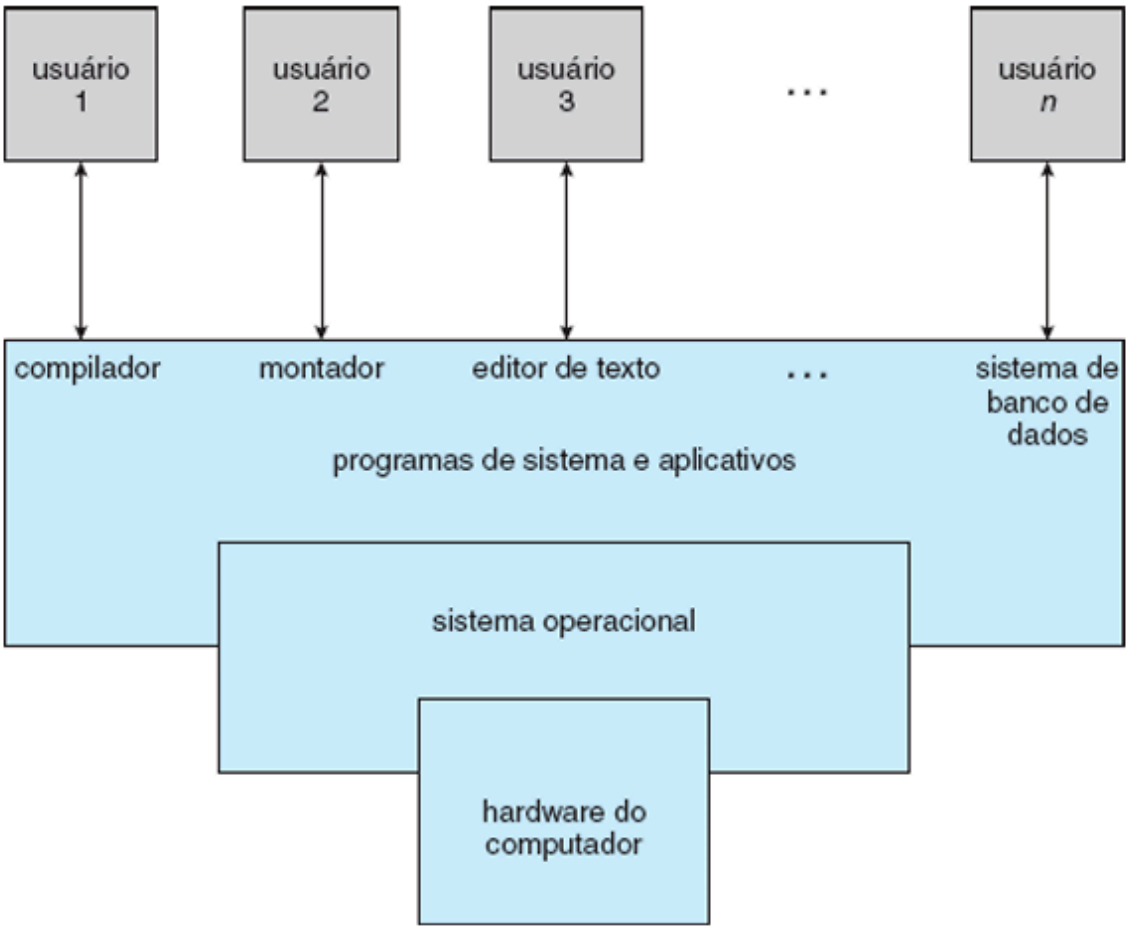
Esta seção tem como objetivo apresentar uma visão geral sobre os sistemas operacionais, abordando seus princípios, funções e importância para o funcionamento dos dispositivos computacionais modernos. O sistema operacional (S.O) é o núcleo que permite a interação entre o usuário e a máquina, garantindo a execução de programas e o gerenciamento dos recursos disponibilizados pelo sistema. Visando compreender seu posto na computação, serão explorados os conceitos gerais, a importância e a relevância atual.

1.1. O QUE É SISTEMA OPERACIONAL?

O sistema operacional (SO) é um software que atua como intermediário entre o usuário e o hardware do dispositivo. Inicialmente, sua função é designada entre gerenciar os recursos do sistema, como o processador, a memória, os dispositivos de entrada e saída e as unidades de armazenamento, garantindo uma utilização eficiente e um bom funcionamento entre diferentes componentes. Sem a existência de um comunicador, o usuário precisaria se relacionar diretamente com o hardware por meio de instruções de baixo nível, o que tornaria o processo complexo e inviável aos usuários finais. (NASCIMENTO, 2025).

Sob uma perspectiva alternativa, o sistema operacional pode ser relacionado à existência de um governo, do qual não realiza suas tarefas diretamente, mas projeta um ambiente que inclui a execução de outros programas funcionais. A analogia reforça a ideia de uma caracterização abstrata entre o usuário e o hardware, transformando a interação humano-computador acessível e produtiva. (SILBERSCHATZ; GALVIN; GAGNE).

Figura 1 - Visão abstrata dos componentes de um sistema de computação



Fonte : Silberschatz; Galvin; Gagne (2018).

1.2. IMPORTÂNCIA E CONTEXTO

O sistema operacional é um elemento importante para o funcionamento de dispositivos computacionais, do qual se responsabiliza por integrar e gerenciar os recursos incluídos pelo sistema. Segundo NASCIMENTO(2025, p. 7), “ele atua como intermediário entre o usuário e o hardware, permitindo que programas sejam executados de forma eficiente e segura”. Sua função é gerenciar as execuções de programas de usuários para evitar possíveis erros e o uso impróprio do dispositivo se tornem comuns, notando-se uma postura cautelosa em relação à operação e ao controle de dispositivos (I/O).

Considerando outra análise, o tema abordado apresenta diferentes descrições conforme sua funcionalidade, mas buscando especificar e simplificar a complexidade deste assunto, é possível defini-lo como o único programa que permanece em execução durante todo o tempo útil do dispositivo, nomeado como “Kernel” ou “núcleo”. Deste modo, o sistema operacional garante o funcionamento adequado do hardware, criando estabilidade, segurança e otimização do uso de recursos disponíveis conforme o surgimento de novas tecnologias.

2. ORIGEM

Esta seção apresenta os fundamentos teóricos sobre a origem e a evolução das threads nos sistemas operacionais, destacando sua relação com os processos e sua importância para a execução paralela de tarefas. O estudo das threads é essencial para compreender como os sistemas modernos otimizam o desempenho, compartilham recursos e realizam múltiplas operações de forma simultânea.

2.1. PROCESSO

Originalmente, o sistema operacional organiza a execução de programas seguindo um processo sequencial, que se restringe ao ser executado apenas uma vez. Ao ser interceptado com uma instrução determinada pelo usuário, o sistema carregava o programa diretamente na memória principal, ocupando todo o espaço disponível voltado àquela execução instruída, sem permitir interrupções ou tarefas paralelas.

Este modelo, conhecido como sistema monoprogramável, foi adequado até determinado momento, mas tornou-se limitado à medida que a demanda por desempenho e eficiência cresceu. Assim, surgiram novas abordagens de gerenciamento, como a multiprogramação e o conceito de processo, que permitiram ao sistema operacional controlar diversas execuções simultaneamente, otimizando o uso dos recursos do hardware.

2.1.1. DEFINIÇÃO

Um processo é a unidade de trabalho do sistema, composto por um conjunto de instruções em execução concorrente, que podem ser categorizadas como processos do sistema operacional (aqueles que executam códigos de sistema) e processos do usuário (que executam códigos do usuário). Inicialmente, não podemos considerar que qualquer programa seja definido como um processo sem considerar a presença de um arquivo executável, que possui instruções armazenadas no disco que determinam o que será realizado, junto ao conjunto de recursos associados, que interpreta, lê e executa tais orientações informadas à memória.

Embora exista a limitação de operar programas e suas operações simultaneamente, os processos podem estar associados ao mesmo programa e ainda assim, possuir sequências executáveis individualmente. Segundo Tanenbaum e Bos (2016, p. 60), “ um processo é apenas uma instância de um programa em execução, incluindo os valores atuais do contador do programa, registradores e variáveis “. Esta ideia afirma que o processo não é definido como apenas o código-fonte executável de um programa, mas todo o processo dinâmico enquanto está sendo executado.

2.1.2. ESTRUTURA

A parte estrutural de um Processo é crucial para o seu gerenciamento pelo Sistema Operacional e pode ser dividido em camadas: a Estrutura de Memória (os componentes ativos do programa) e a Estrutura de Controle (o metadado que o SO utiliza para gerenciar a execução). Durante sua execução, cada processo ocupa diferentes áreas da memória, organizadas de modo a permitir o funcionamento correto do programa e o controle eficiente de recursos.

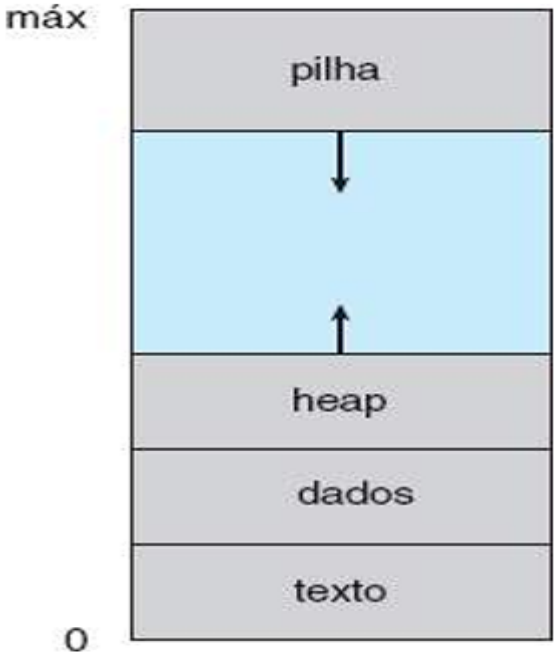
O campo relacionado à Seção de Código (text segment) é responsável por armazenar instruções que constituem um programa, contendo códigos executáveis traduzidos por um compilador, que é interpretado pela CPU e executado seguindo a ordem de instruções. Considerando o alto nível de complexidade, o conteúdo possui métricas de segurança para evitar possíveis modificações que afetam diretamente as instruções transmitidas e as execuções dessas sequência de ações. O mesmo se

aplica à Seção de Dados, que é capaz de armazenar diversas variáveis estáticas e globais, podendo sofrer alterações durante a sua execução, conforme os valores contidos que impactam diretamente o comportamento do funcionamento de um programa. O sistema também se responsabiliza por organizar a área relacionada que cada processo possui, junto ao conjunto de variáveis, visando evitar possíveis interferências entre os processos distintos.

A área destinada à alocação dinâmica de memória é denominada como "Heap”. Esta região descreve quando o programa solicita reservar um espaço adicional em tempo de execução, geralmente aplicada conforme a variação apresentada pelas estruturas de dados, como listas de vetores dinâmicos ou objetos. A expansão e a liberação desta área são controladas por funções específicas em cada linguagem de programação, como “ new “ em linguagens orientadas a objetos. Assim, o gerenciamento da alocação de memória enfatiza o risco de obter um vazamento de memória no sistema, principalmente quando um programa reserva uma área para ser utilizada, mas não libera espaço após o uso, apresentando um excesso de memória que limita o uso de recursos disponíveis do sistema, comprometendo o desempenho e apresentando falhas de execução de processos.

O sistema possui uma estrutura de memória de acesso utilizada para armazenar informações temporárias, como variáveis locais, parâmetros de funções e endereços de retorno. O funcionamento deste processo inicia toda vez que uma função é chamada, cria um novo quadro de ativação que contém todos os dados necessários para sua execução, chamado de “Stack Frame”. Ao finalizar a operação da função, o quadro é removido e o espaço é desocupado, assim, a pilha permite controlar o crescimento e a redução eficiente dos fluxos de execuções de um programa.

Figura 2.1.2 - Processo na memória



Fonte : Silberschatz; Galvin; Gagne (2018).

2.1.3. CONTROLE E ISOLAMENTO DE RECURSOS

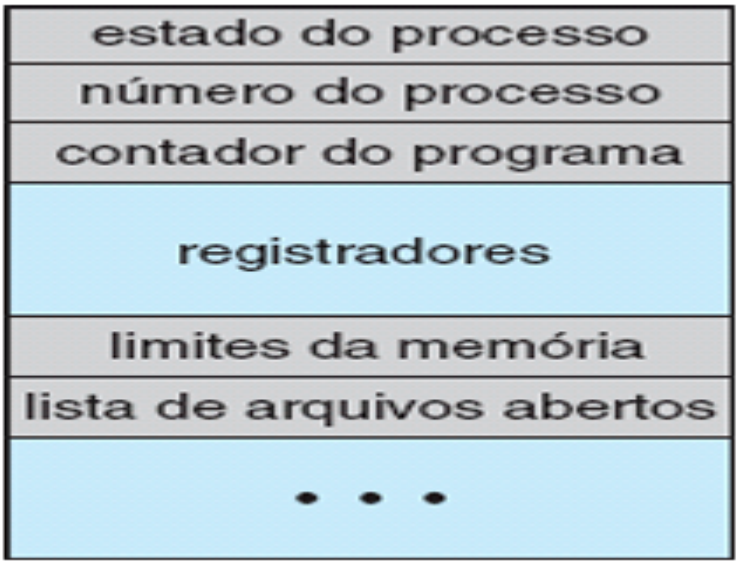
O Bloco de Controle de Processo (PCB) é uma estrutura de dados que permite o monitoramento e o gerenciamento da execução de cada processo ativo. Ele contém todas as informações necessárias para que o sistema possa interromper e retomar a execução de um processo de maneira segura e consistente.

Entre os elementos destacados neste conceito, destacam-se o Estado de Processo, que define o nível de status do funcionamento de um processo, e o Contador de Programa, que é responsável por identificar o endereço da próxima instrução a ser executada. Além desses elementos, o processo de Registro da CPU menciona conteúdos que incluem acumuladores, registradores de uso geral, ponteiros de pilha e outros recursos, que são essenciais para planejar meios de evitar possíveis interrupções em diferentes processos.

Algumas informações também são protegidas conforme as funcionalidades das Informações de Scheduling, que define a prioridade de um processo seguindo os ponteiros de scheduling e seus demais parâmetros de execução. Além disso, as

Informações de Gerenciamento de Memória podem envolver diferentes valores de registradores base e limite, incluindo tabelas de páginas ou de segmentos, conforme o modelo de memória utilizado. Contudo, a organização de dados possibilita ao sistema operacional gerenciar controles precisos e o isolamento eficiente entre os processos, garantindo o funcionamento estável e seguro do ambiente computacional.

Figura 2.1.3 - Bloco de Controle de Processo (PCB)



Fonte : Silberschatz; Galvin; Gagne (2018).

2.1.4. ESTADOS DE EXECUÇÃO

Quando um processo é executado, o seu estado de funcionamento varia conforme a execução deste processo. O estado de um processo é definido por sua atividade corrente, onde cada processo pode representar os seguintes estados:

- Novo : Onde o processo está sendo criado .
- Pronto : Onde o processo está na memória principal e está aguardando para ser escalonado à CPU. Os processos neste estado residem em uma fila de espera, e o escalonador atua para decidir qual será a próxima instrução executada.
- Em Execução : As instruções de um processo são executadas nesta etapa. Em um sistema com único processador (CPU), apenas um processo pode estar em

execução a cada instante. Em sistemas com múltiplos processadores, consequentemente, pode haver múltiplos processos em execução.

- Em Espera : Ocorre quando o processo está aguardando o encerramento de algum evento programado. Se o processo faz uma solicitação e esta solicitação entra em espera, isto demonstra que o sistema está ocupado operando uma ação, sendo automaticamente encaminhado à uma fila de espera.
- Encerrado : Ocorre quando o processo finaliza sua execução. Neste momento, o sistema desloca todos os recursos que o processo anterior estava utilizando.

Figura 2.1.4 - Diagrama de estado do processo

Fonte : Silberschatz; Galvin; Gagne (2018).

2.1.5. TRANSIÇÕES DE ESTADOS E TROCA DE CONTEXTO

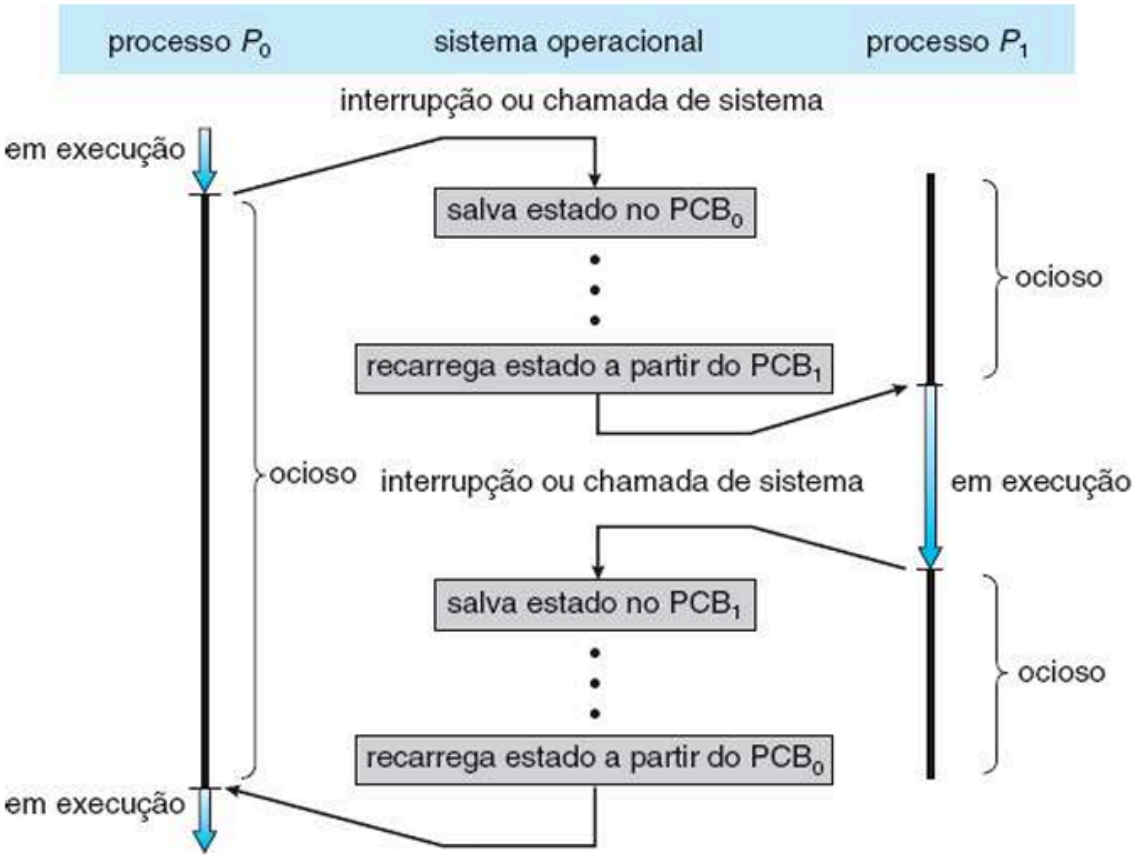
O ciclo de vida de um processo é composto por transições entre diferentes estados de execução, controladas pelo sistema operacional por meio de mecanismos de escalonamento e troca de contexto. Inicialmente, o processo é criado no estado Novo e, após ser admitido na memória principal, é inserido na “Fila de Prontos“, onde aguarda o funcionamento do “Escalonamento de Curto Prazo“ para utilizar a CPU.

Quando o processo é selecionado para execução, as instruções informadas são processadas, ocasionando determinados eventos que resultam na mudança de um estado: a solicitação de uma operação de entrada / saída (input / output), que define o estado como “Em Espera“ até a conclusão da operação; a Preempção por Interrupção

ocorre quando o tempo de execução da CPU expira e o processo retorna à fila de prontos; e o Término da Execução, quando o processo finaliza e atinge o status de estado Encerrado.

Essas transições são viabilizadas pela Troca de Contexto (Context Switch), procedimento em que o Kernel (núcleo) é responsável por salvar o estado atual do processo pelo Bloco de Controle de Processo (PCB) e carregar o contexto do próximo processo a ser executado. Esse processo garante a continuidade da execução dos programas de forma ordenada e segura, ainda que o tempo gasto na troca de contexto represente uma sobrecarga dos recursos adicionais que o S.O precisa gastar para realizar uma tarefa (overhead).

Figura 2.1.5 - Diagrama de alternância da CPU para um outro processo



Fonte : Silberschatz; Galvin; Gagne (2018).

2.1.6. COMUNICAÇÃO ENTRE PROCESSOS

A comunicação entre processos pode ser implementada a partir de diferentes modelos de comunicação, normalmente, o Modelo de Passagem de Mensagens é o mais aplicável, pois os processos compartilham informações por meio de operações explícitas de envio e recebimento, e o Modelo de Memória Compartilhada, que garante a múltiplos processos o acesso à mesma área de memória para troca destas informações. Esses modelos são sustentados por diversos mecanismos de Comunicação entre Processos (IPC), como por exemplo:

- Filas de Mensagens : É o meio de comunicação assíncrono entre diferentes serviços ou aplicações, onde mensagens são armazenadas temporariamente até serem processadas pelo destinatário.
- Pipes (Tubos) : Fornecem um mecanismo unidirecional que permite a transmissão de informações entre os processos, atuando como um ”arquivo temporário“ na memória do dispositivo. Geralmente são anônimos, permitindo comunicação apenas entre processos relacionados , que somente processos relacionados podem se comunicar (“ pai - filho “); e por Pipes Nomeados (First In - First Out), que permitem a comunicação entre processos não relacionados e continuam sendo existentes na memória mesmo após o encerramento dos processos.
- Sockets : São interfaces de comunicação que permitem que o IPC (Comunicação entre Interprocessos) possua o acesso local ou remoto de um serviço. Um socket é definido pela comunicação presente de um par de processos através de uma rede empregada, do qual exige a identificação do IP concatenado com um número de porta daquele serviço. Geralmente, por se tratar de uma arquitetura cliente-servidor, o servidor aguarda solicitações de clientes que desejam possuir uma conexão proveniente aos serviços específicos (como HTTP, com sua “ porta “ 80).
- Condição de Corrida : Ocorre quando múltiplos processos acessam e manipulam os mesmos dados concorrentemente, interferindo diretamente no resultado devido a ordem em que o acesso ocorre. Para evitar esta situação, é necessário assegurar que o recurso seja utilizado apenas uma única vez, exigindo uma comunicação síncrona (com bloqueio ou receptor até que a mensagem seja recebida).

- Semáforos : É um mecanismo que reforça a sincronização de processos durante a sua execução, porém, podem resultar em “ Erros de Tempo “ (Timing Errors) que ocorrem quando são aplicados em uma sequência de execução específica e essa sequência não resulta em uma execução de um programa.
- Spinlocks : Os SpinLocks são ações de sincronização de baixo nível usadas para proteger recursos compartilhados do sistema, onde uma thread que não consegue adquirir o bloqueio entra em um loop de verificação repetidamente até se certificar que o bloqueio foi liberado, ao invés de ser encaminhado ao estado de “Em Espera“ do processo. Esta verificação pode ser mais eficiente em relação aos demais bloqueios em cenários, em que a espera é curta, mas consome ciclos de CPU em uma espera contínua.
- Deadlocks : A definição de “Deadlock” é mencionada em uma situação em que dois ou mais processos ficam permanentemente bloqueados, pois cada processo espera finalizar um determinado evento que utiliza de tal recurso mantido por outro conjunto de processo, travando todo o funcionamento e execução de processos no sistema.
- Comunicação Síncrona e Assíncrona : A comunicação entre processos têm lugar por meio de chamadas primitivas que são definidas por “ Send() “ e “ Receive()”. A comunicação pode ser com Bloqueio (síncrona), na qual o emissor é bloqueado até que a mensagem seja enviada, ou sem Bloqueio (assíncrona), onde o processo retoma a operação imediatamente.

Contudo, os modelos e mecanismos de Comunicação de Interprocessos assumem um papel central no funcionamento de sistemas operacionais modernos, permitindo a cooperação entre processos locais e remotos de maneira segura, sincronizada e confiável.

Tabela 2.1.6 - Características entre os tipos de Comunicação

Característica	Comunicação por Memória Compartilhada	Comunicação por Troca de Mensagem
Desempenho	Alto desempenho (acesso direto à memória), pois a comunicação ocorre na velocidade de transferência da memória.	Baixo desempenho por ser mais lento devido à cópia de dados e à intervenção do Kernel (chamadas do sistema).
Segurança / Isolamento	Exige que os processos concordem em remover a restrição de acesso à memória de outro processo.	Não permite acessar diretamente à memória de outro processo, do qual se responsabiliza por compartilhar esta informação.
Sincronização	Exige controle rigoroso de concorrência (semáforos) para evitar condições de corrida.	Simples de implementar, evitando conflitos para pequenas quantidades de dados.
Ambiente	Adequado para o IPC Local (dentro do mesmo dispositivo).	Recomendado de implantar em sistemas distribuídos (entre computadores).

Fonte : Autoria própria.

2.2. THREADS

Esta seção tem como objetivo apresentar uma visão geral sobre as threads, abordando seus conceitos fundamentais, estrutura, e papel no contexto dos sistemas operacionais modernos. As threads representam unidades básicas de execução dentro de um processo, permitindo que múltiplas tarefas sejam realizadas de maneira simultânea e eficiente. Elas foram desenvolvidas para aprimorar o desempenho dos sistemas multitarefa, otimizando o uso do processador e reduzindo o tempo de resposta das aplicações executadas.

Compreender o funcionamento e a organização das threads é essencial para analisar como o sistema operacional gerencia a execução concorrente de processos, bem como distribui recursos entres os diferentes fluxos de instruções. Dessa forma, serão explorados seus conceitos gerais, sua importância no processamento paralelo e as diferenças entre os modelos de execução no espaço do kernel e no espaço do usuário.

2.2.1. DESCRIÇÃO

Uma thread pode ser definida como a menor unidade de processamento que um sistema operacional para execução de um programa. Enquanto um processo é como um “programa em execução“ com seu próprio espaço de memória, uma thread é como um “fluxo de instruções“ dentro desse processo. (NASCIMENTO, 2025) .

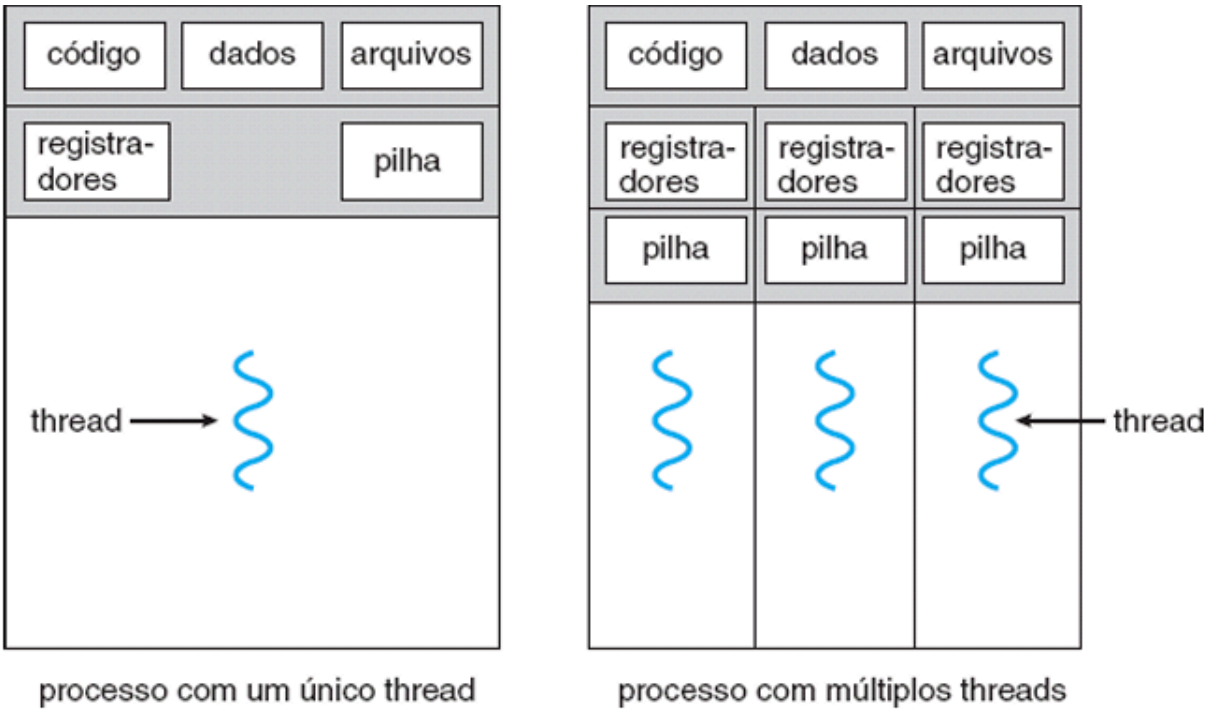
Um processo tradicional é nomeado de processo de uma “ Única Thread “ ou “ Pesado “. No entanto, os sistemas operacionais modernos estendem o conceito de processo para permitir que ele tenha múltiplos threads de execução, permitindo que desempenhe mais de uma tarefa simultaneamente.

Tabela 2.2.1 - Diferença entre Processo e Thread

Característica	Processo	Thread
Espaço de memória	Isolado	Compartilhado com outras threads do mesmo processo
Criação	Mais “pesada” e lenta	Mais leve e rápida
Comunicação	Necessita mecanismos como *pipes* ou *sockets*	Pode compartilhar variáveis na memória
Independência	Se um falha, os outros continuam	Falha de uma pode afetar todo o processo

Fonte : Nascimento (2025).

Figura 2.2.1 - Processos com um único thread e com múltiplos threads



Fonte : Silberschatz; Galvin; Gagne (2018).

2.2.2. MODELOS DE PROCESSOS MULTITHREADS

O modelo multithread (ou “ Modelo de Processos Multithreads “) permite que um único processo possua várias tarefas executando em paralelo ou concorrência, oferecendo benefícios significativos, especialmente em sistemas multicore. Os benefícios do funcionamento de um sistema multi-threading podem ser categorizados por 4 etapas : Escalabilidade, Capacidade de Resposta, Economia de Recursos e o Compartilhamento de Recursos.

- Escalabilidade : Os benefícios da criação de múltiplos threads podem ser essencial para sistemas de arquitetura multicore ou com multiprocessadores, onde as threads podem ser executadas paralelamente em diferentes núcleos de processamento. Além disso, um processo com apenas uma thread só pode ser executado em um único processador, desconsiderando a disponibilidade entre os demais processadores.
- Capacidade de Resposta : Tornar uma aplicação multithread interativa permite que um programa continue sua execução, independentemente de haver outra operação

em execução ou aguardando o fim da interrupção, o que acrescenta a capacidade final de resposta ao usuário . O uso deste cenário permite considerar que cada ação selecionada pelo usuário gera a execução de uma nova thread ou processo correspondente.

- Economia : A alocação de memória e recursos para a criação de um processo é custosa, já que as threads compartilham os recursos do processo ao qual pertencem, é mais econômico criar threads e permutar seus contextos. Inicialmente, a criação de um processo é mais lento naturalmente, entretanto, o gerenciamento e a criação de threads minimiza o risco de obter um overhead (sobrecarga de processamento).
- Compartilhamento de Recursos : Os processos só podem compartilhar outros recursos por meio de técnicas como a “ Memória Compartilhada “ e a “ Transmissão de Mensagens “. Estas técnicas devem ser organizadas explicitamente pelo programador, porém, pela configuração de fábrica (default), os threads compartilham a memória e os recursos do processo pertencentes. Esse funcionamento permite que o código e os dados sejam compartilhados em múltiplos threads, em diferentes atividades dentro do mesmo espaço de endereçamento.
- Paralelismo : Em geral, há dois tipos de Paralelismo : o Paralelismo de Dados e o Paralelismo de Tarefas. O paralelismo de dados foca na distribuição de subconjuntos dos mesmos dados, por múltiplos núcleos de computação e na execução da mesma operação em cada núcleo. O paralelismo de tarefas envolve a distribuição de threads em vários núcleos de computação separados, em que cada thread executa uma única operação, podendo estar operando sobre os mesmos dados ou sobre diferentes conjuntos.

2.3. MODELOS DE THREADS

Os modelos de threads representam as diferentes formas de relacionamento entre os diferentes tipos de threads existentes, determinando como ocorre a comunicação e o gerenciamento entre os níveis do sistema operacional e da aplicação. Esses modelos foram desenvolvidos com o objetivo de equilibrar desempenho, eficiência e flexibilidade , na execução de tarefas concorrentes. Cada abordagem define a maneira como as threads são mapeadas para o kernel e como o sistema lida com operações de criação, sincronização e escalonamento.

Tabela 2.2.1 - Modelos de Threads

Modelo de Mapeamento	Descrição	Implicações Principais
Many-to-One (Muitos para Um)	Mapeia muitas threads de nível de usuário para um único thread de kernel	Não resulta em concorrência real, pois apenas um thread pode acessar o núcleo, e não aproveita múltiplos núcleos de processamento.
One-to-One (Um para Um)	Mapeia cada thread de usuário para um thread de kernel correspondente.	Oferece maior concorrência e permite a execução paralela em multiprocessadores. No entanto, impõe um custo maior (overhead) na criação de threads, pois cada thread de usuário requer a criação de um thread kernel.
Many-to-Many (Muitos para Muitos)	Multiplexa muitas threads de nível de usuário para um menor número ou igual de threads de núcleo.	É flexível e eficiente, permitindo que os desenvolvedores criem muitos threads de usuário sem sobrecarregar o sistema com muitos threads kernel, permitindo o funcionamento do paralelismo.

Fonte : Autoria própria.

2.4. TIPOS DE THREADS

Os sistemas operacionais modernos implementam diferentes tipos de threads, que se distinguem principalmente pelo nível em que são gerenciadas. As threads de usuário são controladas por bibliotecas no nível de aplicação e operam acima do núcleo, sem o envolvimento direto do sistema operacional, o que proporciona maior agilidade na troca de contexto, entretanto, limita o paralelismo em sistemas multiprocessados.

Já as threads de kernel são totalmente administradas pelo próprio sistema operacional, que é responsável por seu agendamento, sincronização e execução. Embora apresentem maior custo de gerenciamento, oferecem suporte completo e apresentam melhor aproveitamento dos recursos do processador.

2.4.1. THREADS DO USUÁRIO

As threads de usuário destacam-se pela eficiência na execução de operações de troca, criação e sincronização, uma vez que essas ações ocorrem inteiramente no espaço do usuário, sem a necessidade de intervenção do kernel. Isso reduz significativamente o custo de processamento e o tempo de troca de contexto, tornando-as ideais para aplicações que exigem alta responsividade e um grande número de threads leves.

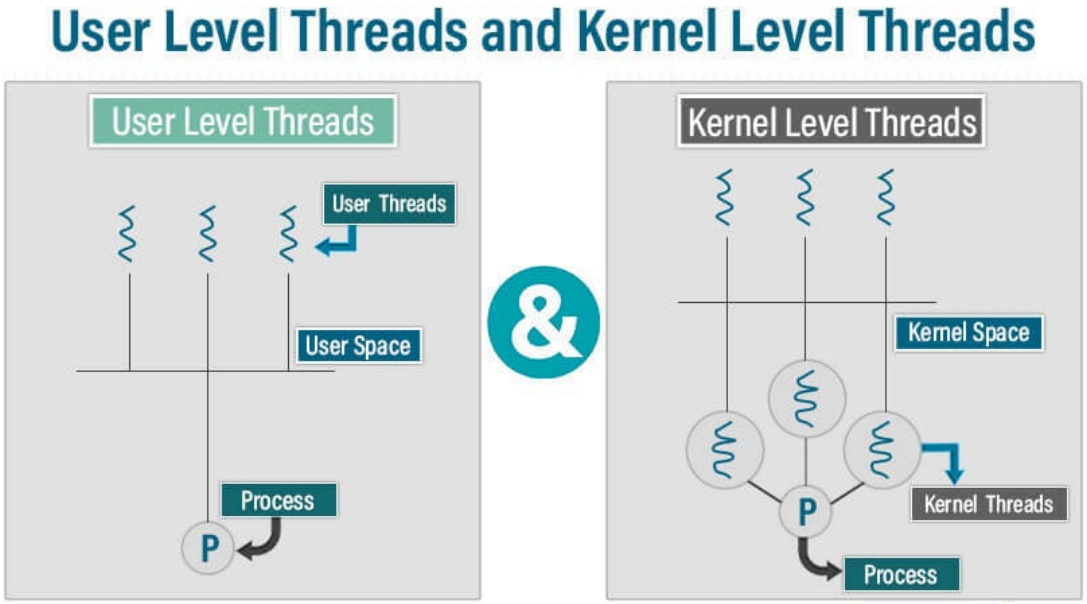
No entanto, esta arquitetura tem como desvantagem que, se uma thread realizar uma chamada bloqueante ao sistema, todo o processo permanecerá inativo até que a operação seja concluída, comprometendo o paralelismo real em sistemas multiprocessadores. Para viabilizar sua implementação, utiliza-se com frequência a biblioteca POSIX Threads (PTHREADS) , empregada em sistemas Unix e Linux. Esta biblioteca fornece um conjunto de funções padronizadas que permitem criar, controlar e gerenciar threads de maneira portável e eficiente, estabelecendo uma interface robusta para o desenvolvimento de aplicações concorrentes em nível de usuário.

2.4.2. THREADS KERNEL (NÚCLEO)

As threads de Kernel são gerenciadas diretamente pelo Sistema Operacional, o que significa que o kernel tem conhecimento de sua existência, sendo responsável por realizar operações de criação, escalonamento, sincronização e troca de contexto. Esse gerenciamento em nível de núcleo permite que múltiplas threads de um mesmo processo sejam executadas simultaneamente em diferentes processadores, aproveitando de maneira mais eficiente os sistemas multiprocessados e possibilitando o funcionamento do paralelismo.

Diferentemente das threads de usuário, uma thread de kernel bloqueada não impede a execução dos demais, já que uma vez que o sistema operacional pode alternar o processamento entre outras threads ativas. No entanto, essa abordagem implica um maior custo de desempenho, pois cada operação de criação ou troca de contexto requer uma chamada ao sistema, o que resulta em maior consumo de tempo e recursos de processamento.

Figura 2.4.2 - Figura dos níveis de threads de Usuário e threads de Kernel



Fonte : Bista (2023).

3. CARACTERÍSTICAS

Esta área tem como objetivo apresentar as principais características das threads, destacando seus atributos estruturais e funcionais dentro dos sistemas operacionais. As threads se diferenciam por representar a menor unidade de processamento executável, capaz de compartilhar os recursos do processo principal, como a memória e os arquivos abertos, ao mesmo tempo mantendo elementos próprios, como o contador de programa, os registradores e a pilha de execução.

Essa arquitetura compartilhada permite que múltiplas threads de um mesmo processo executem tarefas de forma concorrente e cooperativa, reduzindo a sobrecarga na criação de novos processos e otimizando o desempenho do sistema. A

compreensão dessas características é fundamental para analisar como os sistemas modernos implementam o paralelismo, o balanceamento de carga, e a eficiência na utilização dos recursos computacionais.

3.1. CARACTERÍSTICAS DAS THREADS

As threads podem ser definidas como fluxos independentes de execução que operam dentro de um mesmo processo, compartilhando seus recursos e espaço de endereçamento. Diferentemente dos processos, que possuem memória isolada, cada thread mantém apenas os elementos essenciais para sua execução como identificador, o contador de programa, os registradores e uma pilha própria.

Essa estrutura compacta torna a criação e a alternância de threads mais rápidas e eficientes em comparação com os processos tradicionais. Além disso, as threads compartilham com outras threads do mesmo processo a seção de código, a seção de dados e recursos do sistema operacional como arquivos abertos e sinais, o que facilita a comunicação direta entre elas, dispensando a necessidade de mecanismos complexos de troca de mensagens. Essas características de compartilhamento e leveza tornam as threads elementos fundamentais para o processamento paralelo e o desempenho otimizado dos sistemas multitarefa.

3.2. VANTAGENS DO MULTITHREADING

O modelo *multithread* traz diversos benefícios para o desempenho e a escalabilidade dos sistemas operacionais. Uma das principais vantagens é a capacidade de resposta, pois permite que partes diferentes de uma aplicação sejam executadas simultaneamente, garantindo que o sistema permaneça ativo mesmo durante operações intensivas, especialmente em interfaces gráficas e aplicações interativas. Seguindo a mesma ideia, o *multithreading* proporciona melhor aproveitamento de arquiteturas multicore, distribuindo threads em múltiplos núcleos de processamento e assim aumentando a eficiência do sistema. O compartilhamento de

recursos entre threads reduz o custo de memória e simplifica a troca de dados, já que diversas threads operam no mesmo espaço de endereçamento.

Outro benefício presente neste contexto é o processo de economia de recursos, pois permite criar e alternar o uso de threads sem consumo excessivo de tempo e memória durante a inicialização de processos, tornando o sistema leve e responsivo. Logo, o uso de múltiplas threads representa um avanço na otimização de atividades ordenadas paralelamente e no desempenho global das aplicações modernas.

3.3. DESVANTAGENS E CONSEQUÊNCIAS

Apesar dos benefícios, o modelo multithread também apresenta riscos e problemas que exigem atenção durante o desenvolvimento de gestão de sistemas concorrentes. Um dos principais problemas é o risco de falha generalizada, pois se uma thread apresentar erro crítico, todo o processo ao qual ela pertence pode ser comprometido. Igualmente, o acesso simultâneo a dados compartilhados pode causar condições de corrida, resultando em inconsistências e comportamentos imprevisíveis.

Uma medida para solucionar esses problemas é utilizar mecanismos de sincronização, semáforos e monitores, que garantem a integridade dos dados, porém, interferem na complexidade do sistema; outro problema recorrente é o bloqueio permanente entre dois processos, que se impedem mutuamente de utilizar os recursos. Por fim, as aplicações multithreaded apresentam dificuldades ao serem depuradas e verificadas, devido à natureza paralela de execução, e à multiplicidade de caminhos possíveis. Assim, embora o conceito amplie a eficiência e o desempenho, ainda requer planejamento rigoroso e controle de sincronização, bem como medidas que assegurem a confiabilidade e a estabilidade do sistema.

4. APLICABILIDADE

Esta seção tem como objetivo apresentar a aplicabilidade das threads nos sistemas operacionais e nas diversas áreas da computação moderna. As threads desempenham papel fundamental na execução de tarefas paralelas e no aumento da eficiência de aplicações que exigem múltiplas operações simultâneas. Geralmente, são utilizadas

em sistemas multitarefas, servidores web, ambientes de banco de dados, interfaces gráficas interativas e aplicações científicas de alto desempenho, onde a divisão de tarefas em múltiplos fluxos reduz o tempo de resposta e melhora o aprimoramento dos recursos do processador.

Além disso, em sistemas distribuídos e em arquiteturas multi-core, o uso de threads permite a execução concorrente e cooperativa de processos, favorecendo o balanceamento de carga e escalabilidade das aplicações. A aplicabilidade das threads e seus mecanismos de suporte podem ser observados em diferentes componentes internos do sistema operacional, conforme demonstrado na Tabela 4, a qual exemplifica como esses recursos são utilizados no gerenciamento de processos, memória e arquivos.

Tabela 4 - Aplicabilidade de gerenciamento de componentes internos

Mecanismo Interno	Aplicação no Sistema
Escalonamento de Processos	Define qual processo será executado e quanto será sua duração. Em sistemas modernos, é uma aplicação típica que se ajusta ao comportamento dos processos.
Memória Virtual	Permite que programas sejam maiores que a memória física, aplicando um aumento no grau de multiprogramação e a utilização da CPU. Assim, também viabiliza a criação eficiente de processos (“ Cópia após Gravação “) e o compartilhamento de bibliotecas.
Sistemas de Arquivos	Responsável por funções como mapeamento de dados para o disco, gerenciamento de espaço e controle de permissões de acesso. Tipos como NTFS e EXT4 são aplicados em plataformas específicas para suportar criptografia, compressão e journaling.
Multithreading	Aplicação prática em softwares de edição de vídeo, encarregado pela renderização e interface paralela. Além disso, permite que as aplicações tirem proveito de sistemas multicore para obter paralelismo.
Sincronização	Uso de locks mutex, semáforos,

	spinlocks e variáveis condicionais para garantir a consistência de dados compartilhados em sistemas com processos cooperativos.

Fonte : Autoria própria.

4.1.THREADS NO LINUX

Em aplicações práticas, o uso do multithreading é implementado em diferentes contextos da computação. Os navegadores Web utilizam múltiplas threads para processar cada aba ou realizar a recuperação de dados de forma independente, garantindo maior desempenho e responsividade. Em jogos eletrônicos, as threads são utilizadas para dividir o processamento entre diferentes subsistemas, como gráficos, física e som, o que melhora a fluidez e o tempo de ação. Da mesma forma, os servidores web utilizam a criação de threads para atender simultaneamente múltiplas solicitações de clientes, otimizando o tempo de resposta e uso do processador.

No contexto dos sistemas operacionais, o Linux é um dos exemplos mais expressivos de implementação do modelo multithreaded, permitindo que múltiplos threads sejam executados diretamente no núcleo para realização de tarefas específicas. O processo de criação de threads no Linux é realizado por meio de chamada de sistema “ **Clone()** “**,** diferenciando-se da chamada natural do sistema utilizando o comando “ **Fork()** “. Enquanto o comando “ Fork() “ cria um novo processo independente, sem compartilhamento de recursos com o processo pai, a chamada “ Clone() “ permite definir o nível de compartilhamento entre as tarefas semelhantes. Esta configuração é realizada por meio de flags, que são passadas como argumentos que determinam quais contextos serão duplicados, e consequentemente, quais devem ser compartilhados.

O funcionamento interno desta operação é singular, pois o núcleo não distingue conceitualmente os processos e os threads, sendo mutuamente tratados como tarefas “ tasks “ em execução. Deste modo, o sistema interpreta cada fluxo de controle como uma tarefa, permitindo que o gerenciamento de processos e threads seja unificado e flexível. Este modelo de implementação simplifica o controle de concorrência e torna

um sistema eficiente para ambientes que exigem o paralelismo, desempenho e escalabilidade.

Tabela 4.1. - Flags passados quando o Clone() é invocado

Flag	Descrição
CLONE_FS	As informações do sistema de arquivos são compartilhadas.
CLONE_VM	O mesmo espaço de memória é compartilhado.
CLONE_SIGHAND	Os manipuladores de sinais são compartilhados.
CLONE_FILES	O conjunto de arquivos abertos é compartilhado.

Fonte : Silberschatz; Galvin; Gagne (2018).

4.1.1. REPRESENTAÇÃO DO NÚCLEO

A unificação entre processos e threads é possível devido a forma modular que o núcleo gerencia o contexto de execução. Diferente de outros sistemas que armazenam todas as informações de uma tarefa em uma única estrutura de dados, o sistema distribui esse contexto em diversas estruturas interligadas. O núcleo utiliza a estrutura “ **Task_Struct** ” (equivalente ao Bloco de Controle de Processos), para armazenar informações essenciais de cada tarefa.

Contudo, essas estrutura não contém diretamente todos os dados de execução, em vez disso, mantém ponteiros para outras estruturas menores, responsáveis por armazenar informações específicas, como a tabela de arquivos abertos, o sistema de arquivos associado, os manipuladores de sinais e a memória virtual. Quando uma nova tarefa é criada por meio de chamada de sistema “ **Clone()** “, ela herda esses ponteiros, possibilitando o compartilhamento eficiente de recursos com a task original, reduzindo o custo de criação de novas threads e agilizando ambientes em multitarefa.

4.1.2. MODELAGEM DE MAPEAMENTO E ESCALONAMENTO

O sistema, como os outros demais sistemas, adota o modelo “ One - to - One “ (Um-para-Um) de mapeamento de threads, no qual cada thread de nível de usuário é diretamente associado a uma thread de nível de kernel correspondente. Neste modelo, cada thread é reconhecida e gerenciada individualmente pelo sistema operacional, o que possibilita uma distribuição eficiente da carga de trabalho entre os múltiplos núcleos de processamento disponíveis. Esse mapeamento garante que as threads possam ser executadas em paralelo real, otimizando o desempenho e o tempo de resposta das aplicações.

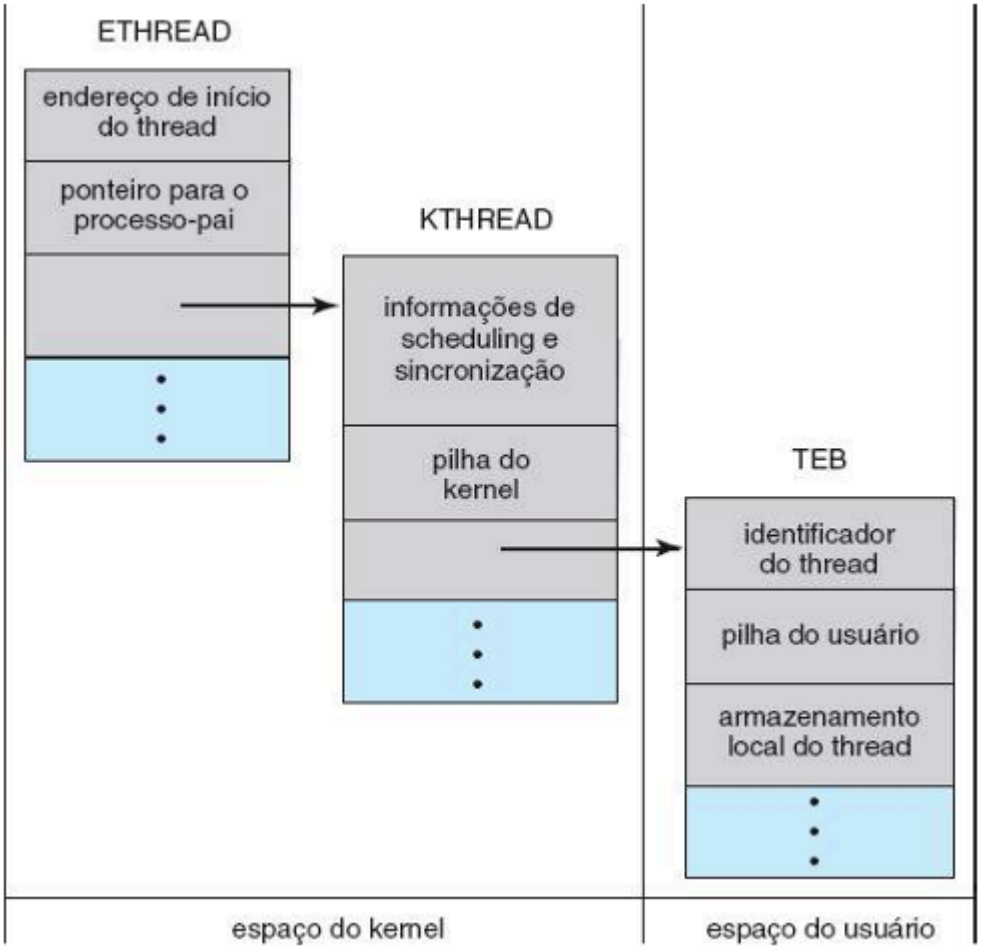
Para determinar qual thread deve ser atribuída a cada CPU, o núcleo do sistema utiliza o Escopo de Disputa do Sistema (System Contention Scope - SCS), onde todos os threads do sistema competem igualmente pelos recursos do processador. O escalonamento (Scheduling) é controlado pelo “ Completely Fair Scheduler " (CFS), o algoritmo padrão do Linux, que busca distribuir o tempo de CPU de maneira equilibrada, garantindo que cada thread receba uma fração justa do processamento.

4.2.THREADS NO WINDOWS

Inicialmente, o Windows aplicou as threads através de API desenvolvidos para o sistema operacional da Microsoft, adicionando em outras versões do sistema em diferentes pacotes de atualização. E a aplicação é executada como a operação de um processo separadamente, onde cada processo contém uma única thread ou demais threads. Geralmente, os componentes gerais de uma thread incluem : um ID que identifica o thread de maneira exclusiva, um conjunto de registradores representando o status do processador; uma pilha de usuário, empregada quando o thread está sendo executado em modalidade de usuário, e uma pilha de kernel, empregada quando o thread está sendo executado em modalidade de kernel; e uma área de armazenamento privado usado por bibliotecas de tempo de execução e bibliotecas de link dinâmico (DLL’s).

A técnica de criação de threads com o uso da biblioteca de threads Windows é semelhante ao conceito de PThreads (Posix Threads), como mencionado anteriormente, os dados são compartilhados por threads separadamente, sendo definidos de acordo com menções de variáveis globais e as funções que realizam operações lógicas e executáveis.

Figura 4.2 - Estruturas de dados de um thread do Windows



Fonte : Silberschatz; Galvin; Gagne (2018).

Os threads são criados na API Windows com o uso da função **CreateThread()**, criando comunicações através de um conjunto de atributos de threads que são ordenadas por meio de funções. Esses atributos incluem informações de segurança, o tamanho da pilha e uma flag que determina o estado de suspensão de uma thread.

Uma vez que a thread de soma seja criada, a thread pai deve esperar que ele seja concluído antes de exibir o valor à variável global, já que o valor é definido pela thread de soma. Em situações que requerem a espera de uma conclusão de múltiplos threads, a função **WaitForMultipleObjects()** é utilizada, exigindo quatro parâmetros para sua execução:

- O número de objetos que serão determinados
- Um ponteiro para o array de objetos
- Um flag que indique que os objetos são sinalizados
- Uma definição de tempo limite

Tabela 4.2 - Código Pthread para o agrupamento de dez threads

```
#define NUM_THREADS 10
 /* um array de threads para o agrupamento */
 pthread_t workers[NUM THREADS];
 for (int i = 0; i < NUM_THREADS; i++)
 pthread_join(workers[i], NULL);
```

Fonte : Silberschatz; Galvin; Gagne (2018).

4.3. THREADS NO MacOS

A compreensão de threads no sistema operacional MacOs requer uma breve explicação sobre a arquitetura interna, que se caracteriza como um sistema operacional híbrido, combinando elementos de Microkernel Mach e dos Kernel BSD UNIX. Esta estrutura é conhecida como núcleo NXU, que constitui a base de Darwin, o núcleo é disponibilizado como open source.

Segundo SILBERSCHATZ, GARVIN e GAGNE (2018), o Mach é o mecanismo responsável pelo gerenciamento de processos e pelo escalonamento de threads, enquanto o BSD provê a interface POSIX (Portable Operating System Interface), garantindo a compatibilidade com padrões UNIX e suporte à API Pthreads. Esta arquitetura modular permite que o sistema operacional MacOs forneça a eficiência de um microkernel com a flexibilidade de um sistema UNIX tradicional, resultando em um ambiente estável e escalável.

Explorando o contexto estrutural, o sistema implementa dois mecanismos de threading : o PThreads (POSIX Thread) e o Grand Central Dispatch (GCP). O Pthreads segue o padrão desenvolvido no sistema POSIX e oferece uma interface tradicional para a criação, sincronização e controle de threads, sendo compatível com os demais sistemas UNIX (como Linux e Solaris). Já o GCP representa uma evolução no gerenciamento de threads, introduzindo o conceito de “threading implícito”, no qual o desenvolvedor não precisa gerenciar manualmente a criação ou o encerramento das threads.

O GCD é composto por extensões da linguagem C, uma API e uma biblioteca de tempo de execução que automatizam a execução paralela de tarefas, otimizando o desempenho do sistema e o uso dos núcleos de processamento. Esta abordagem simplifica o desenvolvimento de aplicações paralelas e melhora a eficiência geral do sistema operacional.

Além das bases conceituais, o MacOs se destaca pela ágil integração de tecnologias de threading a um ecossistema de desenvolvimento robusto e privado, presentes também em modelos que repartem do mesmo sistema da empresa. Outros estudos aprofundam a análise dos mecanismos internos do Mach/BSD, detalhando como o núcleo XNU (sistema híbrido que abrange dois elementos : um sistema microkernel e um sistema monolítico) manipula específicas tarefas e suas filas de execução de maneira hierárquica e distribuída.

A documentação técnica e o acesso ao código open-source do sistema operacional Darwin reforçam a transparência e a fluidez do sistema, exemplificando que o modelo multithreading do S.O apresenta o equilíbrio entre a compatibilidade POSIX, arquitetura modular e eficiência operacional, conforme o gerenciamento de threads em sistemas modernos.

4.4.CODIFICAÇÃO

A linguagem Java fornece suporte nativo à criação e gerenciamento de Threads, permitindo que o desenvolvedor implemente o paralelismo e a execução concorrente diretamente no código. Essa capacidade reflete sobre o funcionamento dos sistemas

operacionais, dos quais as threads compartilham seus recursos, mas executam tarefas de forma independente.

A utilização de threads em Java está diretamente relacionada aos conceitos estudados nos módulos de processos e gerenciamento de threads, sendo uma forma prática de compreender como ocorrem suas operações em nível de aplicação. As execuções de threads são delegadas através da Java Virtual Machine (JVM), que integra o modelo de mapeamento “One-to-One” como os demais modelos de sistemas operacionais, onde cada “Thread Java” corresponde a uma thread original do núcleo, gerenciada pelo escalonador do sistema. Assim, o desempenho e o paralelismo dos blocos de códigos refletem diretamente a eficiência subjacente do S.O.

4.4.1. CRIAÇÃO DE THREADS

Cada instância da classe “MeuThread” representa um fluxo independente de execução. O método “start()” invoca o método “run()” internamente, criando uma nova thread de comportamento equivalente ao “Fork()” de um processo no S.O, porém, sob nível de aplicação.

Tabela 4.4.1 - Criação de Threads em Java

```
class MeuThread extends Thread {
    public void run() {
        System.out.println("Thread em execução: " + getName());
    }

    public static void main(String[] args) {
        MeuThread t1 = new MeuThread();
        MeuThread t2 = new MeuThread();
        t1.start();
        t2.start();
    }
}
```

Fonte : Macedo (2025).

4.4.2. IMPLEMENTAÇÃO COM “RUNNABLE”

O código abaixo representa a separação do raciocínio lógico da tarefa (“class Tarefa”) do controle da thread (“class Thread”), refletindo o modelo de separação entre thread de usuário e thread de kernel.

Tabela 4.4.2 - Implementação com Runnable

```
class Tarefa implements Runnable {
    private String nome;

    public Tarefa(String nome) {
        this.nome = nome;
    }

    public void run() {
        System.out.println("Executando tarefa: " + nome);
    }

    public static void main(String[] args) {
        Thread t1 = new Thread(new Tarefa("Thread A"));
        Thread t2 = new Thread(new Tarefa("Thread B"));
        t1.start();
        t2.start();
    }
}
```

Fonte : Macedo (2025).

4.4.3. SINCRONIZAÇÃO E CONTROLE

O código apresentado abaixo ilustra o problema de condição de corrida e sua resolução por meio da exclusão mútua (synchronized), conceito diretamente ligado aos mecanismos de sincronização estudados (como semáforos e monitores).

Tabela 4.4.3 - Resolução de Condição de Corrida em Java

```
class Contador {
    private int valor = 0;

    public synchronized void incrementar() {
        valor++;
    }
```

```
    public int getValor() {
        return valor;
    }
}

public class ExemploSincronizado {
    public static void main(String[] args) throws InterruptedException {
        Contador contador = new Contador();

        Thread t1 = new Thread(() -> { for(int i=0; i<1000; i++) contador.incrementar();
});
        Thread t2 = new Thread(() -> { for(int i=0; i<1000; i++) contador.incrementar();
});

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Valor final: " + contador.getValor());
    }
}
```

Fonte : Macedo (2025).

5. CONCLUSÃO

O estudo abordado sobre *“Sistemas Operacionais : Conceitos e Algoritmos“* revela a importância central dos sistemas possuem como elo entre o hardware e o software, sendo responsáveis pelo gerenciamento de processos, recursos e pela execução eficiente das aplicações. De acordo com SILBERSCHATZ, GALVIN e GAGNE (2018), o sistema operacional é “ um programa que atua como intermediário entre o usuário e o hardware do computador, gerenciando recursos e promovendo um ambiente conveniente e eficiente para a execução de programas”. Esta definição reflete o papel essencial do sistema operacional como o núcleo da computação moderna.

Com o avanço dos sistemas multiprocessadores e multicore, os conceitos de concorrência e paralelismo tornaram-se fundamentais. As threads, unidades mínimas de execução, possibilitam a divisão de tarefas e o uso mais eficiente dos recursos, aumentando a responsividade e o desempenho das aplicações (NASCIMENTO, 2025).

Sistemas como Linux, Windows e MacOS adotam modelos de threads em um nível de kernel, permitindo a execução paralela em múltiplos núcleos. Especificamente no sistema MacOs, sua estrutura híbrida é composta por microkernel Mach e o núcleo BSD, integra suporte às PThreads e ao Grand Central Dispatch (GCD), que automatizam o gerenciamento de threads e simplificam o paralelismo.

Entretanto, a programação concorrente exige controle rigoroso de sincronização e escalonamento, evitando condições de corrida e aos deadlocks. O uso de semáforos, mutexes e monitores garante a consistência dos dados e a integridade do sistema. Além disso, conceitos como memória virtual e sistemas de arquivos exemplificam a capacidade dos sistemas operacionais de abstrair e gerenciar recursos físicos de forma eficiente (TANENBAUM; BOS, 2015);

Em suma, os sistemas operacionais representam a união entre a teoria e a prática, conciliando o desempenho, segurança e flexibilidade. A separação entre o mecanismo e política, conforme SILBERSCHATZ; GALVIN e GAGNE (2018), continua sendo o princípio que sustenta o design eficiente dos sistemas contemporâneos, essenciais para a era da computação distribuída e paralela.

6.REFERÊNCIAS BIBLIOGRÁFICAS

BISTA, N. **Introduction to User Level threads and Kernel-Level threads**. EDUCBA, 5 dez. 2023. Disponível em: https://www.educba.com/user-level-threads-and-kernel-level-threads/. Acesso em: 30 out. 2025.

NASCIMENTO, F. **Sistemas Operacionais I**. Praia Grande, SP. Faculdade de Tecnologia (FATEC) de Praia Grande, 2025.

MACEDO, V. dos Santos. **Aula 03 - S.O1 - Threads em Java : Exemplos práticos**. Praia Grande, SP. Faculdade de Tecnologia (FATEC) de Praia Grande, 2025.

SILBERSCHATZ, Abraham.; GALVIN, P. B.; GAGNE, Greg. **Fundamentos de Sistemas Operacionais**. 9. ed. Grupo Editorial Nacional, 2018. Tradução de Aldir José Coelho Correa da Silva. Revisão técnica de Elisabete do Rego LIns.

TANENBAUM, Andrew. S.; BOS, Herbert. **Sistemas Operacionais Modernos**. 4. ed. São Paulo: Pearson Education do Brasil, 2016. Tradução de Daniel Vieira e Jorge Ritter. Revisão técnica de Raphael Y. de Camargo.