Simulador de Arquitetura de Von Neumann e Pipeline MIPS

1st Rafael Augusto Campos Moreira
DECOM-DV
CEFET-MG
Divinópolis, Brasil
rafaelaugustocampos@outlook.com

2nd Victor Ramos de Albuquerque Cabral
DECOM-DV
CEFET-MG
Divinópolis, Brasil
vramoscabral2020@gmail.com

***Abstract*—Este artigo propõe o desenvolvimento de um sistema operacional simplificado em linguagem C, inspirado na arquitetura de Von Neumann e no pipeline MIPS. O objetivo é explorar conceitos fundamentais de sistemas operacionais, como gerenciamento de processos, memória e dispositivos de entrada/saída. A implementação inclui a simulação de instruções, controle de memória com uso de cache e a execução sequencial de operações, permitindo a compreensão do funcionamento interno de um sistema operacional de forma prática e didática.**

***Index Terms*—Sistemas Operacionais, Arquitetura, Gerenciamento, Processos**

I. Introdução

Os sistemas operacionais (SOs) desempenham um papel essencial no funcionamento de dispositivos computacionais, servindo como uma interface entre o hardware e o software (Tanenbaum & Bos, 2014). Desde os primórdios da computação, a evolução dos sistemas operacionais tem sido um marco no avanço tecnológico. Nos anos 1950, sistemas batch eram utilizados para processar um trabalho de cada vez, mas foi apenas com o advento do time-sharing nos anos 1960 que os sistemas começaram a suportar múltiplos usuários e tarefas simultaneamente, abrindo caminho para os sistemas modernos (Ceruzzi, 2003).

Com base na arquitetura de Von Neumann, proposta em 1945, a computação estruturada em programas armazenados tornou-se um padrão universal (Von Neumann, 1945). Essa arquitetura, que define que dados e instruções compartilham o mesmo espaço de memória, é a base sobre a qual muitos sistemas operacionais e arquiteturas de computadores modernos são construídos. Neste projeto, a arquitetura de Von Neumann é explorada como fundamento para a implementação de um sistema operacional simplificado, destacando sua relevância histórica e conceitual.

A simulação de pipelines MIPS no projeto também remete ao desenvolvimento de arquiteturas RISC (Reduced Instruction Set Computer), uma evolução significativa na década de 1980 (Patterson & Hennessy, 2013). Os pipelines, responsáveis por dividir a execução de instruções em etapas sequenciais, aumentam a eficiência do processamento e são amplamente utilizados em arquiteturas modernas. O estudo do MIPS proporciona uma compreensão prática de como as instruções são processadas a nível de hardware, conectando conceitos de arquitetura de computadores e sistemas operacionais.

Outro conceito fundamental abordado no projeto é o gerenciamento de memória. A memória cache, por exemplo, surgiu como uma resposta ao crescente desafio de desempenho entre processadores e memória principal (Jacob, Ng, & Wang, 2010). No início, o armazenamento de dados era lento e custoso, mas o desenvolvimento de memórias de alta velocidade, como a cache, trouxe melhorias significativas. O projeto inclui uma implementação simulada de memória cache, que ilustra como a hierarquia de memória impacta diretamente o desempenho dos sistemas computacionais.

Além disso, o gerenciamento de processos é outro ponto central abordado no projeto, uma vez que é classificado como um dos pilares dos sistemas operacionais. Desde os primeiros sistemas multitarefa, como o Multics na década de 1960, até os sistemas modernos, a capacidade de gerenciar e escalonar processos é fundamental para o uso eficiente dos recursos computacionais (Tanenbaum & Bos, 2014). O projeto aborda a execução sequencial de instruções como uma forma simplificada de explorar esse conceito, evidenciando as complexidades envolvidas no gerenciamento de processos.

Os dispositivos de entrada/saída também são considerados, reforçando a importância de entender como sistemas operacionais interagem com o hardware. Desde os antigos sistemas que dependiam de fitas magnéticas até os dispositivos modernos de armazenamento e interfaces de rede, a evolução das tecnologias de I/O moldou a maneira como os SOs são projetados.

A proposta deste sistema operacional simplificado também reflete uma abordagem educacional, alinhando-se a métodos baseados em aprendizagem prática e construtiva. Inspirado por modelos acadêmicos e exercícios projetados para ensinar conceitos de arquitetura de computadores e sistemas operacionais, o projeto oferece uma experiência que vai além da teoria, permitindo que os alunos enfrentem desafios reais de implementação.

Outro aspecto relevante é a integração de conceitos abstratos, como ciclos de instruções e buffers de memória, em um ambiente concreto e programável. Essa abordagem ajuda a consolidar o aprendizado, conectando conhecimentos teóricos com aplicações práticas (Kernighan & Ritchie, 1988). Tal

conexão entre abstração e implementação se deu através da linguagem C, pelo motivo de esta ter sido projetada especificamente para o desenvolvimento de sistemas operacionais, como o Unix, na década de 1970. A linguagem C combina recursos de baixo nível, que permitem acesso direto ao hardware, com características de alto nível, que facilitam a legibilidade e a manutenção do código. Essa versatilidade faz com que a linguagem C seja amplamente utilizada até hoje para o desenvolvimento de sistemas embarcados, drivers e sistemas operacionais modernos, servindo como uma ponte entre o hardware e as camadas mais abstratas do software.

A computação é um campo em constante evolução, e o desenvolvimento de sistemas simplificados permite explorar os fundamentos que deram origem às tecnologias contemporâneas (Ceruzzi, 2003). Ao recriar os elementos básicos de um sistema operacional, os alunos podem apreciar as soluções engenhosas que impulsionaram avanços históricos, como os sistemas Unix, Windows e Linux.

Por fim, este projeto não se limita apenas a ensinar conceitos técnicos, mas também busca estimular o pensamento crítico e a resolução de problemas. Ao abordar desafios práticos, como o funcionamento interno de um sistema operacional, os alunos são incentivados a aplicar sua criatividade e habilidades de programação, com o intuito de construir soluções para problemas complexos presentes nessa pesquisa acadêmica.

II. Quadro Teórico

O quadro teórico envolve a revisão de literatura existente relacionada ao tema do estudo, incluindo teorias, modelos e conceitos que fundamentam a pesquisa e investigação (Flinck, 2018). Tendo isso em vista, essa seção se subdivide em três tópicos. São eles: Arquitetura de Von Neumann, Pipeline MIPS, Arquitetura Multicore, Linguagem C e Trabalhos Correlatos.

A. Arquitetura de Von Neumann

A Arquitetura de Von Neumann, proposta por John von Neumann em 1945, é um modelo teórico que define a organização básica de computadores modernos (Von Neumann, 1945). Este modelo revolucionou a computação ao apresentar o conceito de armazenar dados e instruções em uma única memória, acessada sequencialmente pela unidade de processamento. Antes disso, os computadores eram projetados com arquiteturas rígidas, onde os programas precisavam ser fisicamente reconfigurados para realizar diferentes tarefas. A ideia de uma memória unificada permitiu que computadores executassem uma ampla variedade de programas, tornando-os mais versáteis e eficientes.

Uma característica fundamental da Arquitetura de Von Neumann é o "ciclo de busca e execução", onde a unidade de controle busca instruções da memória, decodifica-as e as executa. Este processo é realizado de maneira sequencial, o que torna a arquitetura intuitiva, mas também suscetível ao "gargalo de Von Neumann". Esse gargalo refere-se à limitação na taxa de transferência de dados entre a CPU e a memória principal, que pode restringir o desempenho do sistema, especialmente em aplicações que demandam alta largura de banda (Tanenbaum & Bos, 2014).

A arquitetura de Von Neumann também introduziu os conceitos de unidade aritmética e lógica (ALU), unidade de controle, memória, e dispositivos de entrada/saída, todos conectados por um barramento comum. Esses componentes se tornaram pilares dos computadores digitais e, também, do trabalho desenvolvido em questão.

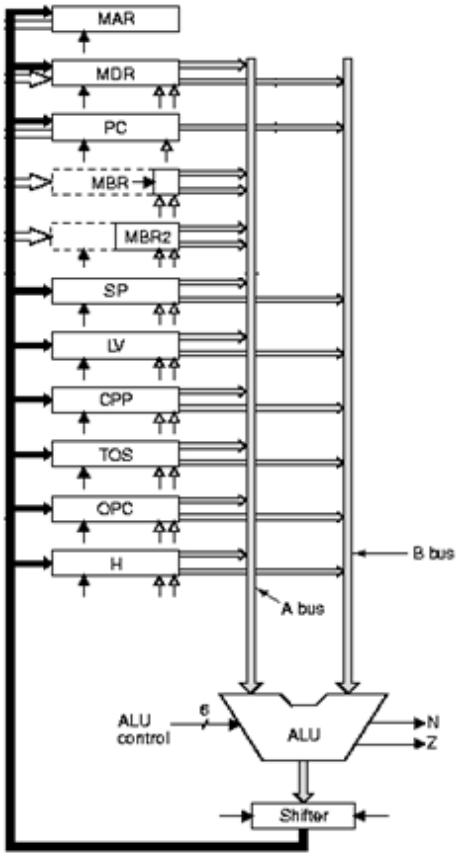

Fig. 1. Caminho de Dados Construído por Banco de Regitradores e ULA - Fonte: Tanembaum[1].

B. Pipeline MIPS

O pipeline MIPS é uma técnica de arquitetura de computadores que busca aumentar a eficiência do processamento ao dividir a execução de instruções em etapas. Tal técnica organiza as operações da CPU em 5 estágios sequenciais, sendo eles: busca de instrução, decodificação, execução, acesso de memórica, escrita de resultado (Patterson, D. A., & Hennessy, J. L., 2013).

- **Busca de instrução (IF - Instruction Fetch):** Nesta etapa, a CPU localiza e carrega a instrução da memória principal para o registrador de instrução. O contador de programa (PC) é usado para indicar o endereço da próxima instrução a ser buscada, e ele é incrementado automaticamente após cada ciclo para apontar para a instrução seguinte.

- **Decodificação (ID - Instruction Decode):** A instrução carregada no estágio anterior é decodificada para identificar qual operação será realizada. Nesta etapa, os operandos são lidos a partir dos registradores, e o tipo de operação (aritmética, lógica, acesso à memória, etc.) é determinado.
- **Execução (EX - Execute):** A operação especificada pela instrução é executada. Dependendo da instrução, isso pode envolver cálculos aritméticos ou lógicos realizados pela Unidade Lógica e Aritmética (ALU), ou o cálculo de endereços para operações de acesso à memória.
- **Acesso à memória (MEM - Memory Access):** Este estágio é responsável por acessar a memória, caso a instrução envolva leitura ou escrita de dados. Por exemplo, uma instrução de carregamento (load) irá buscar dados da memória, enquanto uma instrução de armazenamento (store) grava dados na memória. Caso a instrução não envolva memória, este estágio é ignorado.
- **Escrita de resultado (WB - Write Back):** Finalmente, o resultado da instrução é escrito de volta no registrador apropriado. Este estágio garante que os valores calculados ou recuperados da memória estejam disponíveis para as próximas instruções que precisem deles.

Cada estágio trabalha simultaneamente com uma instrução diferente, permitindo que várias instruções sejam processadas ao mesmo tempo. Essa abordagem resulta em maior aproveitamento dos recursos do processador, reduzindo o tempo de execução global das instruções.

Embora o pipeline MIPS traga ganhos significativos de desempenho, ele também apresenta desafios, como o gerenciamento de dependências de dados e de controle entre as instruções. A simplicidade e eficiência do pipeline MIPS tornaram-no um modelo amplamente estudado, tendo em vista sua alta aplicação. No trabalho em questão, seu uso foi fundamental para compreensão dos conceitos computacionais modernos.

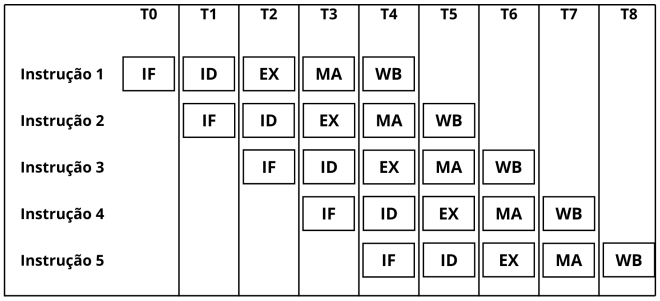

Fig. 2. Pipeline - Fonte: Própria.

C. Arquitetura Multicore

A arquitetura multicore é uma abordagem de design de processadores que integra dois ou mais núcleos de processamento dentro de um único chip. Cada núcleo é uma unidade de processamento independente que pode executar instruções e processos de forma autônoma. Essa configuração foi adotada como solução para as limitações físicas e térmicas enfrentadas pelo aumento da frequência dos processadores tradicionais. Em vez de aumentar a velocidade de um único núcleo, os projetistas passaram a incorporar múltiplos núcleos, permitindo que tarefas paralelas sejam realizadas de forma mais eficiente (Jacob, B., Ng, S., & Wang, D., 2010). Essa arquitetura é amplamente utilizada em aplicações modernas, como servidores, dispositivos móveis e sistemas embarcados, onde o desempenho e a eficiência energética são cruciais.

A principal vantagem da arquitetura multicore é a capacidade de executar várias tarefas simultaneamente, conhecida como paralelismo. Isso é particularmente benéfico para aplicações que requerem processamento intensivo, como simulações científicas, edição de vídeo e inteligência artificial. No entanto, o pleno aproveitamento dessa capacidade depende do software, que deve ser projetado para dividir tarefas entre os núcleos. Além disso, o gerenciamento eficiente de recursos, como memória compartilhada e comunicação entre núcleos, é um desafio importante nesse tipo de arquitetura.

D. Suporte à Preempção

O suporte à preempção é uma característica essencial em sistemas operacionais modernos que permite a interrupção temporária de um processo em execução para que outro processo, geralmente de maior prioridade, possa ser executado. Essa funcionalidade garante a implementação de um sistema multitarefa eficiente, onde os recursos do processador são compartilhados de maneira justa e responsiva entre os processos. A preempção é frequentemente usada em sistemas em tempo real, onde a execução de tarefas críticas deve ser priorizada (Silberschatz, Galvin & Gagne, 2018).

A implementação do suporte à preempção requer o uso de um escalonador preemptivo, responsável por tomar decisões sobre qual processo deve ser executado em cada momento. Quando uma interrupção ocorre, o estado do processo em execução é salvo, permitindo que ele seja retomado posteriormente exatamente de onde parou. Essa abordagem garante a continuidade e a consistência das operações, mesmo em ambientes com alta carga de trabalho.

No trabalho em questão, o suporte à preempção foi implementado para fins de simulação inicial. Por esse motivo, a preempção foi realizada com base no término do quantum alocado a cada processo.

E. Escalonador de Processos

O escalonador de processos é um componente fundamental dos sistemas operacionais modernos, responsável por gerenciar a execução de processos de forma eficiente, distribuindo os recursos computacionais de acordo com políticas predefinidas (Silberschatz, Galvin & Gagne, 2018). Ele é projetado para tomar decisões sobre qual processo deve ser executado a cada instante, considerando critérios como prioridade, tempo de execução e estado do processo.

O escalonamento pode ser implementado de forma preemptiva ou não preemptiva. No primeiro caso, o escalonador pode interromper um processo em execução para alocar o

processador a outro, com base em critérios de prioridade ou término de quantum. Já no modelo não preemptivo, o processo em execução continua até que termine ou entre em estado de espera.

As políticas de escalonamento variam de acordo com os objetivos do sistema. No simulador em questão, as políticas adotadas foram:

- **First-Come, First-Served (FCFS):** Os processos são executados na ordem em que chegam, sem preempção.
- **Round-Robin (RR):** Cada processo recebe um quantum fixo de tempo para executar. Caso não termine nesse período, ele retorna à fila.
- **Prioridade:** Os processos são escalonados com base em um nível de prioridade.

F. Memória Cache

A memória cache é um componente essencial no desempenho dos sistemas computacionais, pois reduz significativamente a latência no acesso à memória e melhora a eficiência geral do processamento (Patterson & Hennessy, 2013). Funcionando como uma camada intermediária de alta velocidade, a cache armazena temporariamente os dados e instruções mais frequentemente utilizados, diminuindo a necessidade de acesso à memória principal, que é mais lenta e consome mais recursos (Tanenbaum & Bos, 2014). Essa funcionalidade é especialmente crítica em sistemas modernos, onde a diferença de velocidade entre o processador e a memória principal pode impactar negativamente o desempenho.

Os sistemas de memória cache são organizados de forma hierárquica, comumente divididos nos níveis L1, L2 e L3. O nível L1, mais próximo do processador, possui a menor capacidade, mas oferece tempos de acesso extremamente rápidos, atendendo a operações críticas com eficiência (Hennessy & Patterson, 2017). À medida que se avança para os níveis L2 e L3, a capacidade aumenta, mas a velocidade diminui, criando um equilíbrio entre custo e desempenho. Além disso, cada nível utiliza políticas de substituição e mapeamento específicas, como associatividade total ou direta, para otimizar o armazenamento e acesso aos dados mais relevantes, maximizando a eficácia do sistema.

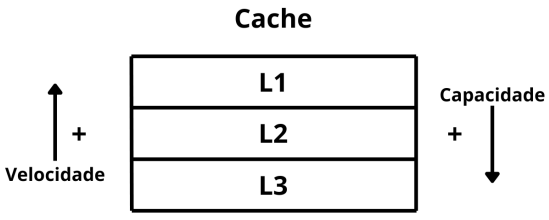

Fig. 3. Memória Cache - Fonte: Própria.

G. Trabalhos Correlatos

Nesta seção, são apresentados diversos estudos que exploram a aplicação e construção de um simulador da arquitetura de Von Neumann, cada um com seu contexto e objetivo específico.

Em se tratando da construção única e própria de um simulador, cita-se Erick Vagner Cabral de Lima Borges, Igor Lucena Peixoto Andrezza, Eduardo de Lucena Falcão, Glauco Sousa e Silva e Hamilton Soares da Silva como grandes contrubuidores. Esse grupo construiu o "SEAC", um simulador online para ensino de arquitetura de computadores, o qual utiliza de todos os conceitos propostos por Von Neumann.

Além disso, Artur Jordão Lima Correia, Mario Augusto Pazoti, Francisco Assis da Silva, Leandro Luiz de Almeida e Danillo Roberto Pereira implementaram um simulador de CPU com suporte à memória cache e pipeline, sendo esses tópicos primordiais trabalhos no projeto em questão.

Dessa forma, os trabalhos aqui citados serviram de modelo para a construção do trabalho e fomentaram a busca de conhecimento para a elaboração do simulador de maneira eficiente e robusta, buscando a utilização de habilidades aprendidas no curso de Arquitetura de Computadores e Sistemas Operacionais.

III. Metodologia

Para construir o simulador de arquitetura Von Neumann e pipeline MIPS, foi utilizada a linguagem de programação C, escolhida por sua proximidade com o hardware e pela sua eficiência no gerenciamento de recursos computacionais. A linguagem também foi selecionada por sua ampla utilização em sistemas operacionais, arquitetura de computadores e processamento de baixo nível, temas diretamente relacionados ao foco deste estudo (Kernighan & Ritchie, 1988).

A implementação do código foi realizada na IDE Visual Studio Code (VS Code), escolhida por sua versatilidade, suporte a múltiplas linguagens e extensões que facilitam o desenvolvimento e depuração de código em C (Microsoft, n.d.), e pela facilidade de acompanhamento nas alterações do código por outros desenvolvedores da equipe, com o recurso Git.

De modo geral, o trabalho foi dividido em três etapas principais:

- **Planejamento e Levantamento Teórico:** Pesquisa bibliográfica para embasar os conceitos fundamentais, como Arquitetura de Von Neumann, Pipeline MIPS e Arquitetura Multicore. A revisão foi essencial para compreender os fundamentos e identificar as ferramentas adequadas para as implementações. Também, as aulas ministradas pelo professor M.Sc. Michel Pires foram de suma importância para entendimento geral do projeto.
- **Desenvolvimento e Simulação:** Criação de programas em C utilizando o VS Code como IDE. O desenvolvimento foi orientado por conceitos abordados no quadro teórico, como a implementação de pipelines simplificados, simulação de processamento em multicore e análise de resultados. Utilizou-se também de metodologias ágeis e boas práticas de programação, como GitFlow.
- **Validação e Análise:** Testes foram realizados nos códigos implementados para verificar a aderência às teorias estudadas. Os resultados obtidos foram analisados e comparados com as expectativas teóricas, permitindo identificar

possíveis otimizações e limitações (erros) das abordagens utilizadas.

Na etapa intermediária da metodologia, correspondente ao Desenvolvimento e Simulação, a construção do software foi dividida em cinco etapas principais, a fim de garantir uma abordagem didática e estruturada. A primeira etapa consistiu na construção do simulador com pipeline MIPS, que serviu como base para a execução das instruções próximo ao comportamento de hardware real. Em seguida, foi implementada uma arquitetura multicore com suporte à preempção, possibilitando a execução simultânea de múltiplos processos e a interrupção de tarefas em andamento.

A terceira etapa envolveu a implementação de um escalonador de processos, com o objetivo de determinar a ordem de execução das tarefas de forma eficiente, respeitando prioridades definidas previamente no simulador. Na quarta etapa, foi desenvolvido o gerenciamento de cache e escalonamento baseado em similaridade, responsável por reduzir o tempo de acesso à memória e melhorar a localidade de referência. Por fim, na quinta etapa, foi implementado o gerenciamento de memória e a estruturação do PCB (Process Control Block), permitindo o controle detalhado dos estados e atributos dos processos em execução.

A. Construção do Simulador com Pipeline MIPS

Na primeira fase, iniciou-se com a Construção do Simulador com Pipeline MIPS, onde foram implementados os componentes fundamentais para a execução das instruções, como registradores, unidades de controle e pipeline MIPS. Esse processo foi baseado nos princípios da arquitetura de Von Neumann, que define um modelo onde a memória é compartilhada entre dados e instruções, permitindo que o processador acesse ambos sequencialmente por meio de um único barramento.

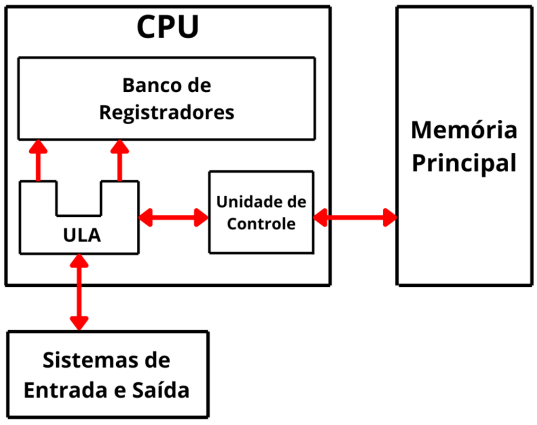

Fig. 4. Arquitetura de Von Neumann - Fonte: Própria.

Para viabilizar a construção da arquitetura, o código foi dividido em 23 arquivos, cada um desempenhando um papel específico no projeto. Tal abordagem promoveu uma organização clara das responsabilidades, facilitando a manutenção, leitura e evolução do código ao longo do desenvolvimento.

A estruturação seguiu uma separação lógica entre os arquivos de cabeçalho (.h) e os arquivos de código-fonte (.c). Os arquivos .h são responsáveis por declarar funções, estruturas, macros e variáveis globais, servindo como interface para o restante do código. Já os arquivos .c contêm a implementação das lógicas e funções declaradas nos arquivos de cabeçalho.

Para ilustrar essa organização, as tabelas a seguir detalham os arquivos utilizados no projeto. A primeira tabela lista os arquivos .h, enquanto a segunda lista os arquivos .c associados.

TABLE I
DESCRIÇÃO DOS ARQUIVOS .H E SEUS OBJETIVOS

Arquivo	Objetivo
`architecture.h`	Declara as funções e estruturas utilizadas na implementação da arquitetura.
`cache.h`	Declara as funções e estruturas relacionadas ao cache.
`control_unit.h`	Declara as funções e estruturas relacionadas à unidade de controle.
`cpu.h`	Declara as funções e estruturas da CPU.
`disc.h`	Declara as funções e estruturas relacionadas ao disco.
`interpreter.h`	Declara as funções do interpretador.
`libs.h`	Arquivo auxiliar contendo bibliotecas comuns utilizadas no projeto.
`peripherals.h`	Declara as funções e estruturas dos periféricos.
`pipeline.h`	Declara as funções e estruturas do pipeline.
`ram.h`	Declara as funções e estruturas da memória RAM.
`reader.h`	Declara as funções relacionadas à leitura de programas.
`uthash.h`	Biblioteca externa utilizada para implementar tabelas hash.

TABLE II
DESCRIÇÃO DOS ARQUIVOS .C E SEUS OBJETIVOS

Arquivo	Objetivo
`architecture.c`	Implementa a estrutura e funcionalidades relacionadas à arquitetura de Von Neumann.
`cache.c`	Implementa o gerenciamento de cache, simulando armazenamento intermediário de alta velocidade.
`control_unit.c`	Implementa a unidade responsável por coordenar a execução de instruções.
`cpu.c`	Implementa as funcionalidades relacionadas à simulação da unidade de processamento central.
`disc.c`	Simula operações de entrada/saída e armazenamento em disco.
`interpreter.c`	Implementa o interpretador das instruções do programa.
`main.c`	Arquivo principal, responsável por integrar os módulos e executar o programa simulador.
`peripherals.c`	Simula os dispositivos periféricos conectados à arquitetura.
`pipeline.c`	Implementa a simulação do pipeline MIPS.
`ram.c`	Implementa a memória RAM, incluindo leitura e escrita de dados.
`reader.c`	Implementa a leitura de programas e instruções a partir de arquivos.

Os programas executados pelo simulador seguem um conjunto de instruções específicas, que representam operações

essenciais para o seu funcionamento. Essas instruções são divididas em categorias como carregamento e armazenamento de dados, operações aritméticas, controle de fluxo e manipulação de memória, as quais são descritas abaixo:

- **LOAD:** Carrega um valor numérico ou o conteúdo de outro registrador em um registrador especificado.
  Formato: `LOAD <REG> <VAL>`
  Exemplo: `LOAD A0 10`
- **STORE:** Armazena o valor de um registrador em uma posição de memória especificada.
  Formato: `STORE <REG> <MEM>`
  Exemplo: `STORE A0 A250`
- **ADD:** Soma o conteúdo de dois registradores ou de um registrador com um valor numérico e armazena o resultado.
  Formato: `ADD <REG> <VAL/REG>`
  Exemplo: `ADD A0 B0`
- **SUB:** Subtrai o conteúdo de dois registradores ou de um registrador com um valor numérico e armazena o resultado.
  Formato: `SUB <REG> <VAL/REG>`
  Exemplo: `SUB A0 1`
- **MUL:** Multiplica o conteúdo de dois registradores ou de um registrador com um valor numérico e armazena o resultado.
  Formato: `MUL <REG> <VAL/REG>`
  Exemplo: `MUL A0 B1`
- **DIV:** Divide o conteúdo de um registrador pelo de outro ou por um valor numérico e armazena o resultado.
  Formato: `DIV <REG> <VAL/REG>`
  Exemplo: `DIV A0 2`
- **LOOP:** Define o início de um laço de repetição baseado no valor de um registrador.
  Formato: `LOOP <VAL/REG>`
  Exemplo: `LOOP A0`
- **L_END:** Marca o término do bloco de repetição iniciado por um `LOOP`.
  Formato: `L_END`
  Exemplo: `L_END`
- **IF:** Verifica uma condição lógica entre dois valores e executa o bloco associado se a condição for verdadeira.
  Formato: `IF <REG> <COND> <VAL/REG>`
  Exemplo: `IF D0 == 1`
- **ELSE:** Define o bloco de instruções a ser executado caso a condição do `IF` não seja satisfeita.
  Formato: `ELSE`
  Exemplo: `ELSE`
- **ELS_END:** Marca o término do bloco associado ao `ELSE`.
  Formato: `ELS_END`
  Exemplo: `ELS_END`
- **I_END:** Marca o término do bloco associado ao `IF`.
  Formato: `I_END`
  Exemplo: `I_END`

Para que essas instruções possam ter seus dados processados e armazenados de maneira correta, usam-se de duas ferramentas principais: o interpretador e os registradores.

O interpretador é responsável por validar as instruções fornecidas ao sistema, garantindo que elas estejam em conformidade com os formatos esperados antes de serem processadas. Essa validação inclui a análise de cada linha do programa para identificar o tipo de instrução e verificar sua estrutura sintática, como a presença correta de operandos e delimitadores. Dessa forma, o interpretador assegura que apenas instruções válidas sejam executadas, evitando erros durante o processamento.

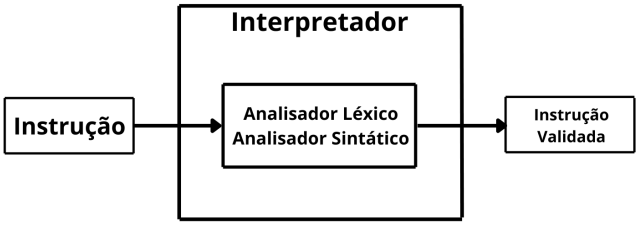

Fig. 5. Interpretador - Fonte: Própria.

Já os registradores são organizados em um vetor que armazenam os dados temporários utilizados nas operações. Os principais registradores incluem nomes como A0, B0, C0 e assim por diante, totalizando 32 registradores, organizados em duas faixas (de A0 a P0 e de A1 a P1). Cada registrador serve como um contêiner para armazenar valores intermediários de cálculo ou parâmetros necessários para controle de fluxo. Esses registradores são acessados rapidamente pelo processador, garantindo eficiência no processamento das instruções e no gerenciamento dos dados temporários.

Fig. 6. Interpretador - Fonte: Própria.

Dessa forma, para que cada operação siga seu fluxo adequado, a unidade de controle coordena a sua execução, interpretando e gerenciando o fluxo de dados entre os componentes da CPU. Ela identifica o tipo de instrução (como ADD, SUB, LOOP), já validada anteriormente pelo interpretador, e emite sinais de controle para garantir a execução correta das operações, incluindo estruturas de controle de fluxo, como

loops e condicionais. Após isso, o resultados dessas execuções serão armazenadas nos registradores.

Com relação à pipeline, esse componente foi implementada da forma descrita na seção **Quadro Teórico**, onde cada etapa possui seu próprio objetivo. Cada estágio da pipeline é responsável por uma parte específica do processamento da instrução, permitindo a execução simultânea de várias instruções, o que resulta em um aumento significativo na eficiência do processador.

Por fim, nessa primeira etapa, a adoção de conceitos das metodologias ágeis durante o desenvolvimento se mostrou crucial para o sucesso da implementação. Embora não tenha sido especificada uma metodologia ágil formal como Scrum ou Kanban em sua totalidade, os princípios ágeis permearam o processo, com ênfase em iterações curtas, feedback contínuo e adaptação às mudanças (Sommerville, 2016).

B. Arquitetura Multicore e Suporte à Preempção

Na segunda etapa do projeto, o foco foi expandir as funcionalidades do simulador, implementando uma arquitetura multicore e suporte à preempção, com o objetivo de simular, de forma mais realista, o comportamento de sistemas operacionais modernos.

A arquitetura multicore foi implementada para permitir a execução simultânea de processos em múltiplos núcleos, simulando o paralelismo encontrado em processadores modernos. Cada núcleo foi representado por uma thread, utilizando a biblioteca pthread para gerenciar a execução concorrente. Mecanismos como exclusão mútua e sincronização foram empregados para evitar condições de corrida e garantir a consistência dos dados compartilhados entre os núcleos.

Para permitir a execução simultânea de múltiplos processos, foi desenvolvida uma fila que organiza e armazena os programas a serem processados pelo simulador. Essa estrutura possibilita o bom gerenciamento da execução de diferentes processos, garantindo que cada um seja tratado de maneira ordenada e no momento correto.

O suporte à preempção foi incorporado ao sistema por meio de uma lógica de interrupção, que permite a troca de contexto entre os processos. A preempção ocorre quando o quantum de tempo alocado a um processo chega ao fim. Nesse momento, o sistema operacional pausa a execução do processo em curso e seleciona outro processo na fila de prontos para ser executado.

Além disso, a preempção também é acionada quando dois processos precisam acessar o mesmo recurso computacional, como a memória RAM. Nessa situação, o sistema interrompe um dos processos para garantir que ambos possam ser atendidos de forma eficiente e sem conflitos.

O ciclo de vida dos processos foi implementado com base no modelo descrito por Tanenbaum, que define três estados principais: PRONTO, BLOQUEADO e EXECUTANDO. Quando um processo está no estado BLOQUEADO, ele permanece na fila de processos, mas sua execução está suspensa até que a condição que causou o bloqueio seja resolvida. Assim que o bloqueio é retirado, o processo retorna ao estado PRONTO e pode ser escalonado novamente para execução.

Os processos que estão em execução podem ser preemptados ou finalizados. A preempção ocorre quando um processo está interrompido por outro que possui prioridade ou porque atingiu o fim de seu quantum de execução. Já os processos finalizados têm suas informações completamente processadas, e suas tarefas são concluídas de forma definitiva.

Para gerenciar adequadamente essa fila de processos e seus estados, foram utilizadas duas subestruturas principais: o processo em si e o PCB (Process Control Block) associado a cada processo. O PCB contém informações essenciais sobre o processo, como seu estado atual, seus recursos alocados e outros dados necessários para a gestão eficiente e segura da execução dos processos. Essas subestruturas permitem que o sistema operacional tenha um controle preciso sobre cada etapa do ciclo de vida dos processos, desde a sua criação até sua conclusão.

Para suportar essas funcionalidades, foram criados 3 novos arquivos .h e arquivos .c para gerenciar o PCB (Process Control Block), filas de processos e threads que controlam o comportamento do sistema operacional, sendo que cada thread (unidades de execução dentro de um processo) representa um núcleo, sendo responsável por executar os processos atribuídos a ele (Kerrisk, 2010).

TABLE III
DESCRIÇÃO DOS ARQUIVOS `.h` E SEUS OBJETIVOS

Arquivo	Objetivo
`pcb.h`	Declara as estruturas e funções relacionadas ao Process Control Block (PCB), incluindo estados, prioridades e memória.
`queues.h`	Define as funções para manipulação da fila de processos e gerenciamento de processos nela.
`threads.h`	Declara as funções e estruturas necessárias para a criação e controle das threads que simulam os núcleos do processador.

TABLE IV
DESCRIÇÃO DOS ARQUIVOS `.c` E SUAS IMPLEMENTAÇÕES

Arquivo	Descrição
`pcb.c`	Implementa as funções relacionadas à manipulação do PCB, como criação, atualização e leitura dos dados do processo.
`queues.c`	Implementa a lógica de gerenciamento da fila de processos, incluindo acesso sincronizado e controle de estados.
`threads.c`	Contém a implementação das threads e funções relacionadas à execução de processos e à preempção nos núcleos.

Por fim, nessa segunda etapa, a adoção de conceitos das metodologias ágeis continuou a desempenhar um papel essencial para o sucesso da implementação.

C. Escalonador de Processos

Na terceira etapa do simulador, focou-se na implementação do escalonador de processos, componente essencial para garantir a organização e eficiência na execução de múltiplas tarefas concorrentes. O escalonador simula a maneira como

o sistema operacional organiza a execução dos processos, priorizando e alocando os núcleos de CPU de acordo com diferentes políticas de escalonamento.

Conforme dito na seção **Quadro Teórico**, foram implementadas três políticas principais de escalonamento: First Come, First Served (FCFS), Prioridade Baseada no Menor Valor e Round-Robin. Cada uma dessas políticas oferece um comportamento distinto em relação à alocação de CPU para os processos.

Além disso, para garantir o controle do tempo de execução e a contabilização do ciclo de vida de cada processo, foi adicionado um timestamp que acompanha o processo durante toda sua execução. Esse timestamp registra o tempo total de vida de cada processo, desde sua chegada até sua conclusão, permitindo uma análise do desempenho do sistema e o comportamento de cada processo no ambiente simulado.

A preempção foi incorporada de forma que um processo pode ser interrompido caso seu quantum expire, permitindo que outro processo seja executado, seja por prioridade ou pelo ciclo de execução do Round-Robin. Quando o quantum de um processo é esgotado, ele é automaticamente bloqueado e o escalonador seleciona outro processo da fila, de acordo com as políticas desenvolvidas em questão.

Para gerenciar o comportamento dos processos dentro dessas políticas de escalonamento, foram introduzidos novos atributos no PCB, como o tempo total de execução do processo e o momento exato em que o processo se encontra. Esses atributos são continuamente atualizados à medida que o processo avança ou é interrompido. O escalonador, então, utiliza essas informações para tomar decisões informadas sobre qual processo deve ser alocado para execução em cada ciclo, garantindo um gerenciamento eficiente dos recursos disponíveis.

Para suportar essas novas funcionalidades, foram criados dois novos arquivos, com o intuito de gerenciar o escalonador de processos e as respectivas filas de execução, controlando, assim, o comportamento do sistema operacional durante sua execução.

TABLE V
DESCRIÇÃO DO ARQUIVO `.h` E SEU OBJETIVO

Arquivo	Objetivo
`scheduler.h`	Declara as estruturas e funções relacionadas ao escalonador de processos.

TABLE VI
DESCRIÇÃO DO ARQUIVO `.c` E SUA IMPLEMENTAÇÃO

Arquivo	Descrição
`scheduler.c`	Implementa a lógica de escalonamento de processos.

Por fim, nessa terceira etapa, a adoção de conceitos das metodologias ágeis também continuou a desempenhar um papel essencial para o sucesso da implementação, como nas demais anteriores.

D. Gerenciamento de Cache e Escalonamento Baseado em Similaridade

Na quarta etapa do projeto, o foco esteve na implementação do gerenciamento de cache e na adoção de uma nova abordagem de escalonamento baseada em similaridade. Essa etapa visou otimizar o desempenho do simulador de arquitetura Von Neumann, reduzindo a latência de acesso à memória e melhorando a eficiência do escalonamento dos processos.

O gerenciamento de cache foi desenvolvido para simular o funcionamento real das hierarquias de memória encontradas nos processadores modernos, contudo de uma maneira mais simples. Ao invés de implementar uma estrutura multi-nível com caches L1, L2 e L3, cada uma com suas próprias características de capacidade e tempo de acesso, foi implementado apenas um nível, com capacidade e tempo de acesso igual para todos os processos. Dessa forma, mesmo se tratando de uma abordagem mais simplista, conseguiu-se reproduzir o impacto do cache no desempenho do sistema retratado na literatura.

A cache foi construída utilizando a biblioteca uthash, permitindo uma estrutura eficiente de armazenamento e recuperação de processos com base em seus identificadores. A biblioteca uthash, desenvolvida por Troy D. Hanson e Arthur O'Dwyer, é uma implementação eficiente de tabelas hash em C, permitindo a criação de estruturas dinâmicas de dados com acesso rápido. Ela fornece funções para inserção, busca e remoção de elementos. Sendo assim, a estrutura da cache é composta por uma tabela hash que armazena instâncias do PCB, garantindo que cada processo seja rapidamente localizado e acessado.

A implementação seguiu os seguintes passos:

- **Inicialização da Cache:** A cache é inicializada como uma tabela vazia, pronta para armazenar processos conforme necessário.
- **Adição de Processos:** Quando um processo precisa ser armazenado, ele é adicionado à cache com seu identificador único e um ponteiro para sua estrutura PCB.
- **Busca de Processos:** A cache permite a busca eficiente de processos armazenados utilizando seu identificador, reduzindo o tempo de acesso em comparação com a memória principal.
- **Remoção e Esvaziamento:** A cache pode remover processos específicos ou ser completamente esvaziada para liberar memória.

A integração do gerenciamento de cache com o escalonador de processos também foi uma etapa fundamental desta fase do projeto. O escalonamento baseado em similaridade foi introduzido para agrupar processos que possuem padrões de acesso à memória semelhantes. Esse método se mostrou eficiente ao reduzir falhas de cache, pois processos que compartilham dados ou instruções semelhantes são escalonados para execução consecutiva, maximizando a reutilização da cache.

Dessa forma, o escalonador pode agrupar e priorizar processos que compartilham padrões de acesso similares, melhorando a eficiência do sistema.

Por fim, a adoção de conceitos das metodologias ágeis continuou a desempenhar um papel crucial no desenvolvimento dessa etapa, permitindo ajustes contínuos e melhorias na eficiência do simulador.

IV. Resultados e Discussões

Esta seção apresenta os resultados obtidos com a simulação do sistema operacional simplificado e discute suas implicações. Devido à natureza do simulador, os resultados se concentram na validação do comportamento do pipeline MIPS, na execução correta das instruções implementadas, na simulação do gerenciamento de memória com cache, bem como na implementação da arquitetura multicore e do suporte à preempção.

A. Validação do Pipeline MIPS

O pipeline MIPS foi implementado com sucesso, garantindo uma execução eficiente e concorrente de diferentes estágios de instruções. Para validar o funcionamento do pipeline, foram executados programas de teste com diferentes sequências de instruções, observando o comportamento dos estágios IF, ID, EX, MEM e WB.

B. Execução das Instruções

Todas as instruções implementadas (LOAD, STORE, ADD, SUB, MUL, DIV, LOOP, L_END, IF, ELSE, ELS_END, I_END) foram testadas individualmente e em conjunto, dentro de programas mais complexos. Os resultados demonstraram a execução correta das operações aritméticas, lógicas e de controle de fluxo.

Em particular, a implementação das instruções de controle de fluxo (LOOP, IF, ELSE) permitiu a simulação de programas com estruturas de repetição e desvio condicional, demonstrando a capacidade do simulador de executar programas mais complexos.

C. Arquitetura Multicore e Suporte à Preempção

A arquitetura multicore foi validada com a execução simultânea de processos em múltiplos núcleos, representados por threads. Para testar essa funcionalidade, foram executados programas com múltiplos processos, observando o comportamento das threads e sua interação concorrente.

Os resultados indicaram que a arquitetura multicore foi eficaz em simular o paralelismo, com múltiplos processos sendo executados simultaneamente em diferentes núcleos. A utilização de sincronização e exclusão mútua garantiu a consistência dos dados compartilhados entre as threads, evitando condições de corrida. A preempção, por sua vez, demonstrou sua utilidade na troca de contexto entre os processos, permitindo que o sistema operacional simulasse de maneira realista a interrupção e a execução de novos processos a cada quantum alocado.

D. Escalonador de Processos

A implementação do escalonador de processos foi testada com a simulação das diferentes políticas de escalonamento, incluindo First-Come, First-Served (FCFS), Prioridade e Round-Robin. O objetivo principal foi garantir a correta alocação dos processos nos núcleos e a adequada troca de contexto entre eles, respeitando os limites de tempo e as condições de preempção.

Durante os testes, o escalonador foi responsável por selecionar o próximo processo a ser executado com base nas políticas definidas, considerando o tempo de execução acumulado, a prioridade dos processos e o quantum de tempo disponível para cada execução. A implementação da preempção foi crucial para a correta troca de contexto, permitindo que o sistema operacional simulasse interrupções e alternasse entre os processos de maneira correta.

E. Gerenciamento de Memória com Cache

A implementação da memória cache permitiu a simulação do comportamento de um sistema de memória hierárquico, proporcionando um modelo eficiente de armazenamento temporário de processos. Para validar a eficácia do mecanismo, foram realizados testes variando os tamanhos da cache e analisando diferentes padrões de acesso à memória.

Os testes evidenciaram que a inclusão da cache reduz significativamente o tempo médio de acesso à memória, especialmente para processos que exibem alta localidade espacial e temporal. Além disso, a estratégia de escalonamento baseada em similaridade contribuiu para minimizar falhas de cache, agrupando processos com padrões de acesso semelhantes e otimizando a reutilização dos dados em memória. Essa abordagem reforça a importância do gerenciamento eficiente da cache na melhoria do desempenho geral do sistema, validando as estratégias implementadas no simulador.

F. Discussão

O desenvolvimento do simulador permitiu a compreensão prática dos conceitos de arquitetura de Von Neumann, pipeline MIPS, arquitetura multicore e escalonador de processos. A implementação em linguagem C proporcionou um controle preciso sobre os detalhes da simulação, permitindo a observação do comportamento dos diferentes componentes do sistema.

A análise do desempenho do simulador foi complementada por gráficos que comparam o número de núcleos de processamento utilizados com o tempo de execução dos processos simulados, tanto antes quanto depois da implementação da cache. No teste de simulação, usou-se 4 processos, com 1, 2, 3 e 4 cores trabalhando em sua execução. O primeiro gráfico revela uma tendência clara de redução no tempo de execução à medida que o número de cores aumenta, destacando a eficiência do paralelismo. Este comportamento é esperado, pois a arquitetura multicore permite a execução simultânea de múltiplos processos, distribuindo a carga de trabalho entre os núcleos.

Fig. 7. Gráfico de execução sem cache - Fonte: Própria.

O gráfico da Figura 7 apresenta uma clara relação inversa entre o número de núcleos disponíveis e o tempo total de execução dos processos. Observou-se que, com o aumento no número de núcleos, há uma redução significativa no tempo necessário para a conclusão das tarefas. Esse comportamento é consistente com o que se espera de sistemas multicore, nos quais a divisão da carga de trabalho entre os núcleos resulta em um melhor aproveitamento dos recursos computacionais disponíveis.

Contudo, vale ressaltar que os ganhos de desempenho não crescem de forma linear com o número de núcleos. Embora o aumento no número de núcleos tenha mostrado uma redução considerável no tempo de execução, é importante destacar que a complexidade do simulador também influenciou nesses resultados. Fatores como a eficiência do escalonador e a necessidade de gerenciar a comunicação entre os núcleos afetaram a escalabilidade do sistema. Em sistemas mais complexos, o benefício de adicionar mais núcleos pode ser reduzido pela sobrecarga de gerenciamento e pela necessidade de coordenação entre os diferentes núcleos.

Com a introdução do gerenciamento de cache, foi possível avaliar seu impacto na redução do tempo de execução dos processos. O gráfico da Figura 8 mostra os tempos de execução com a cache ativada, evidenciando ganhos expressivos na performance devido à diminuição dos acessos à memória principal.

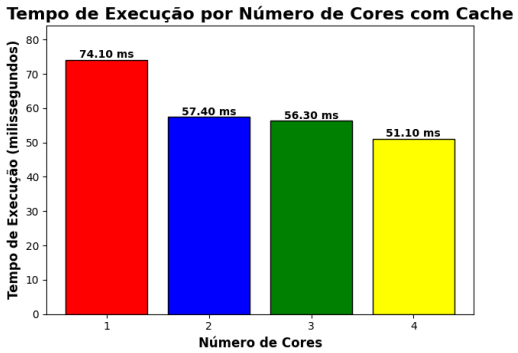

Fig. 8. Gráfico de execução com cache - Fonte: Própria.

Comparando os dois gráficos (Figuras 7 e 8), observa-se que o uso da cache resultou em uma redução substancial no tempo de execução dos processos. A introdução da cache resultou em uma redução de aproximadamente 14,89% no tempo de execução, evidenciando a eficiência de seu uso na otimização do acesso à memória. Isso ocorre devido à maior reutilização de dados previamente armazenados, minimizando as penalidades de acesso à memória. Além disso, o escalonamento baseado em similaridade complementou essa otimização, agrupando processos com padrões de acesso semelhantes, o que contribuiu para reduzir as falhas de cache e melhorar a eficiência geral do sistema.

Por fim, a tabela a seguir apresenta uma comparação mais clara entre os tempos de execução de um processo utilizando diferentes números de núcleos, tanto com cache quanto sem cache.

TABLE VII
COMPARAÇÃO DO TEMPO DE EXECUÇÃO COM E SEM CACHE

Núcleo(s)	Sem Cache (s)	Com Cache (s)
1 core	0,0854 s	0,0741 s
2 cores	0,0671 s	0,0574 s
3 cores	0,0656 s	0,0563 s
4 cores	0,0626 s	0,0511 s

Além da tabela, um gráfico comparativo foi elaborado para visualizar a diferença entre os tempos médios de execução com e sem cache, reforçando os benefícios da abordagem implementada. Vale destacar que os resultados apresentados foram influenciados pela estruturação do projeto, o que pode impactar na interpretação dos dados obtidos.

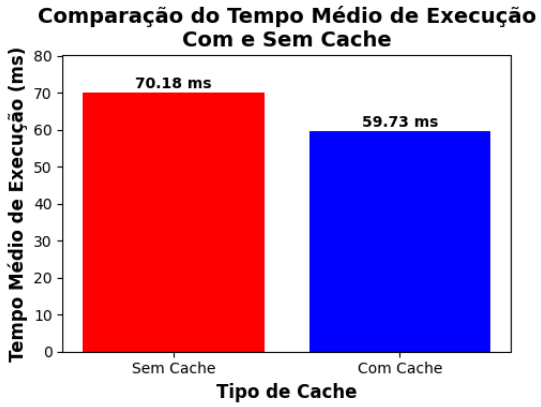

Fig. 9. Comparação entre execução com e sem cache - Fonte: Própria.

Como trabalhos futuros, pretende-se expandir o simulador com a implementação de outros recursos, como interrupções, gerenciamento de processos mais robusto e suporte a instruções mais complexas. Além disso, pretende-se realizar uma avaliação mais quantitativa do desempenho do simulador, comparando os resultados com modelos teóricos e benchmarks, a fim de validar ainda mais a eficácia das otimizações introduzidas.

V. Agradecimentos

Fica em evidência a gratidão ao Professor M.Sc. Michel Pires pelas aulas ministradas na disciplina de Sistemas Operacionais. Sua clareza na exposição dos conteúdos foram fundamentais para a compreensão dos conceitos que fundamentaram este trabalho. As discussões em sala de aula e o suporte oferecido foram essenciais para o desenvolvimento deste projeto.

Também, ressalta-se o agradecimento à instituição Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG) pela oportunidade de desenvolver este trabalho acadêmico. O ambiente de aprendizado proporcionado pela instituição, bem como a infraestrutura disponibilizada, foram cruciais para a concretização deste projeto.

References

[1] Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems* (4th ed.). Pearson.

[2] Von Neumann, J. (1945). *First Draft of a Report on the EDVAC*. University of Pennsylvania.

[3] Patterson, D. A., & Hennessy, J. L. (2013). *Computer Organization and Design: The Hardware/Software Interface* (5th ed.). Morgan Kaufmann.

[4] Hennessy, J. L., & Patterson, D. A. (2017). *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann.

[5] Jacob, B., Ng, S., & Wang, D. (2010). *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann.

[6] Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language* (2nd ed.). Prentice Hall.

[7] Ceruzzi, P. E. (2003). *A History of Modern Computing* (2nd ed.). MIT Press.

[8] U. Flick, (2018). *An Introduction to Qualitative Research* (6th ed.). Sage Publications.

[9] E. V. C. L. Borges, I. L. P. Andrezza, E. L. Falcão, G. S. Silva, H. S. da Silva, (2012) *SEAC: Um Simulador Online para Ensino de Arquitetura de Computadores*. Universidade Federal da Paraíba.

[10] A. J. L. Correia, M. A. Pazoti, F. A. da Silva, L. L. de Almeida, D. R. Pereira, (2014) *Simulador de UCP com Suporte à Memória Cache e Pipeline*. Universidade do Oeste Paulista.

[11] Microsoft. (n.d.). *Visual Studio Code*. Retrieved from https://code.visualstudio.com/

[12] Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education Limited.

[13] Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.

[14] Kerrisk, M. (2010). *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press.

[15] Hanson, T. D., & O'Dwyer, A. (n.d.). *uthash: a hash table for C structures*. Retrieved from https://troydhanson.github.io/uthash/.