Simulador de Arquitetura de Von Neumann

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 e paralela 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) são fundamentais para o funcionamento de dispositivos computacionais, atuando como a interface entre o hardware e o software, além de gerenciar os recursos do sistema de maneira eficiente (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 por vez, mas foi com o advento do time-sharing na década de 1960 que os sistemas começaram a suportar múltiplos usuários e tarefas simultaneamente, pavimentando o caminho para os sistemas modernos (Ceruzzi, 2003). Esse desenvolvimento foi um divisor de águas na forma como os recursos computacionais passaram a ser compartilhados entre múltiplas tarefas, abrindo caminho para a inovação em áreas como escalonamento de processos e gerenciamento de memória.

A arquitetura de Von Neumann, proposta em 1945, estabeleceu a base para a computação moderna ao definir que dados e instruções compartilham o mesmo espaço de memória, o que facilitou o desenvolvimento de sistemas de processamento mais eficientes. Essa arquitetura, que ainda fundamenta a maioria dos sistemas e arquiteturas computacionais, é a base sobre a qual este projeto de sistema operacional simplificado é construído, refletindo sua importância histórica e conceitual (Von Neumann, 1945). Ela também é central para a compreensão de como os sistemas de processamento modernos operam, intercalando os aspectos de memória, processamento e execução de instruções.

Dentro desse contexto, a implementação de pipelines MIPS no projeto remonta ao desenvolvimento das arquiteturas RISC (Reduced Instruction Set Computer), uma evolução crucial nos anos 1980 (Patterson & Hennessy, 2013). O uso de pipelines, que divide a execução de instruções em etapas sequenciais, maximiza a eficiência de processamento e é uma característica essencial nas arquiteturas modernas. A simulação de pipelines MIPS, por sua vez, oferece uma compreensão prática de como as instruções são processadas a nível de hardware e como a arquitetura influencia a execução de processos dentro de um sistema operacional.

No âmbito do gerenciamento de memória, a memória cache se destaca como uma das inovações mais significativas para reduzir a latência entre o processador e a memória principal. Ao criar uma camada de memória de acesso rápido, a cache melhora o desempenho dos sistemas, permitindo que o processador acesse frequentemente dados e instruções com maior rapidez (Jacob, Ng, & Wang, 2010). Este projeto inclui uma implementação simulada de memória cache, permitindo observar como a hierarquia de memória impacta diretamente o desempenho de sistemas computacionais.

Outro tema central no projeto é o gerenciamento de processos, que é um dos pilares fundamentais dos sistemas operacionais. Desde os primeiros sistemas multitarefa, como o Multics na década de 1960, até as arquiteturas modernas, a evolução do escalonamento de processos e do suporte à preempção permitiu a construção de sistemas mais responsivos e eficientes (Tanenbaum & Bos, 2014). O escalonador de processos é responsável por alocar o tempo de CPU de forma justa e eficiente entre os processos, otimizando o uso dos recursos e garantindo a execução das tarefas com base em diferentes políticas, como FIFO, Round-Robin, Prioridade e Similaridade. No projeto, essas políticas são implementadas de forma a ilustrar como o escalonamento influencia diretamente o desempenho do sistema e a alocação de recursos.

A preempção, um conceito fundamental, é a capacidade de interromper um processo em execução para dar prioridade a outro, geralmente mais urgente ou com mais recursos disponíveis. Este mecanismo permite que sistemas multitarefa sejam mais responsivos, especialmente em sistemas de tempo real, onde é crucial garantir a execução de processos críticos dentro de prazos rigorosos (Silberschatz, Galvin & Gagne, 2018). O suporte à preempção no projeto simula o comportamento de sistemas reais, oferecendo uma visão detalhada de como os escalonadores de processos funcionam para garantir a execução eficiente e equilibrada das tarefas.

Além disso, o gerenciamento de memória virtual é um conceito que possibilita que programas acessem mais memória do

que a fisicamente disponível, usando técnicas como paginação e segmentação. A tradução de endereços lógicos para físicos, utilizando uma tabela de páginas, é um dos principais desafios abordados neste projeto. Essa abordagem permite simular o comportamento de sistemas modernos de gerenciamento de memória, onde a memória é virtualizada, garantindo que os processos possam utilizar o espaço de memória de maneira mais flexível e eficiente (Hennessy & Patterson, 2017). A implementação desses conceitos no simulador torna o aprendizado mais prático, permitindo um claro entendimento de como as operações de tradução de endereços e gerenciamento de memória impactam o desempenho de sistemas complexos.

A linguagem C, escolhida para o desenvolvimento do projeto, é a base de muitos sistemas operacionais modernos, como o Unix e o Linux, devido à sua combinação de controle de baixo nível e flexibilidade de alto nível. Criada para ser eficiente e portável, a C é a linguagem ideal para este tipo de aplicação, pois permite a manipulação direta de hardware e a construção de sistemas operacionais com alto desempenho (Kernighan & Ritchie, 1988).

Em um cenário de constante evolução na computação, o desenvolvimento de sistemas operacionais simplificados oferece uma oportunidade única de compreensãom dos fundamentos que deram origem às tecnologias contemporâneas. Ao recriar os elementos básicos de um sistema operacional, é perceptível as soluções inovadoras que impulsionaram os avanços nos sistemas operacionais, como o Windows, o Linux e o macOS.

Por fim, esse tipo de projeto proporciona uma base sólida para a aplicação de conceitos avançados em sistemas operacionais e computação, criando uma ponte entre a teoria e a prática no contexto de tecnologias emergentes.

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, Escalonador de Processos, Suporte à Preempção, Memória Cache, Gerenciamento de Memória (Endereçamento Virtual), 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.

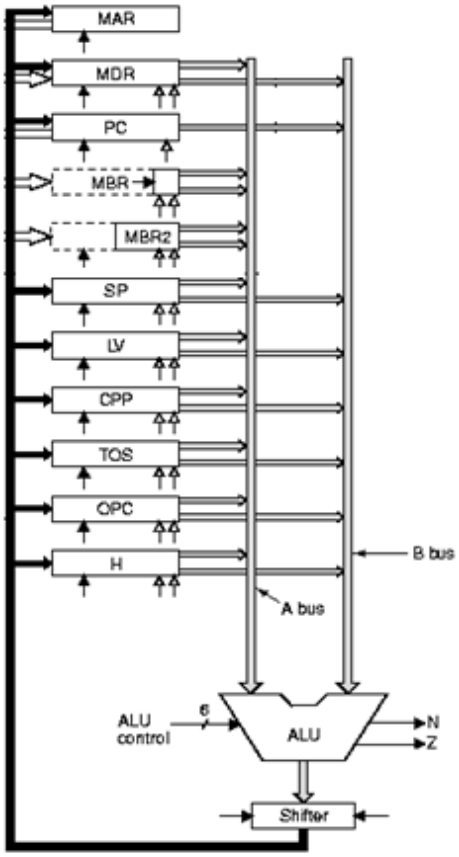


Fig. 1. Caminho de Dados Construído por Banco de Regitradores e ULA - Fonte: Tanembaum[1].

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.

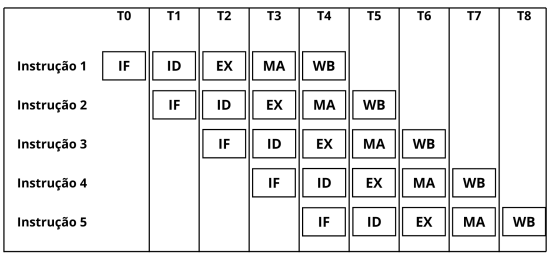


Fig. 2. Pipeline - Fonte: Própria.

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. Escalonador de Processos

O escalonador de processos se define como a parte responsável por gerenciar a execução de processos de forma eficiente no sistema operacional, 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. É importante salientar que existem diversas políticas de escalonamento, contudo, no simulador em questão, para fins pedagógicos, as políticas adotadas foram:

- **First-Come, First-Served (FCFS):** Os processos são executados na ordem em que chegam.
- **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.
- **Similaridade:** Os processos são escalonados com base em sua similaridade, garantindo que aqueles com características semelhantes sejam executados em sequência.

Cada escalonador aplica sua política com base nas informações armazenadas no Process Control Block (PCB). O PCB é uma estrutura de dados fundamental que mantém detalhes essenciais sobre cada processo, sendo alguns deles:

- **ID do Processo**
- **Estado do Processo**
- **Prioridade do Processo**
- **Quantum Total**

E. Suporte à Preempção

Como já introduzido na seção anterior, suporte à preempção é a característica dos sistemas operacionais modernos que permite a interrupção temporária de um processo em execução para que outro processo, geralmente de maior prioridade ou de quantum inalterado, possa ser executado. O quantum é o tempo máximo que um processo pode ocupar a CPU antes de ser interrompido e realocado na fila de processos (estrutura de dados utilizada pelo sistema operacional para organizar e gerenciar os processos em diferentes estados de execução). 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.

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.

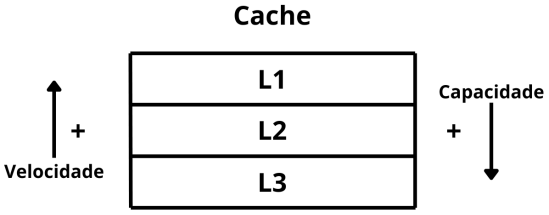


Fig. 3. Memória Cache - Fonte: Própria.

Quando a memória cache atinge sua capacidade máxima, é necessário remover alguns dados para liberar espaço para novas informações. Para isso, diferentes políticas de substituição são utilizadas, determinando quais blocos serão descartados. Entre as estratégias mais comuns estão:

- **Least Recently Used (LRU):** Remove os dados menos acessados recentemente, priorizando os blocos que não foram utilizados por mais tempo.
- **First-In, First-Out (FIFO):** Descarta o bloco mais antigo armazenado na cache, removendo o primeiro bloco que entrou no sistema.
- **Random Replacement:** Escolhe aleatoriamente um bloco para ser substituído, sem considerar o tempo de uso ou a ordem de chegada.

A escolha da política adequada impacta diretamente no desempenho do sistema, influenciando a taxa de acertos da cache e reduzindo a latência no acesso à memória.

G. Gerência de Memória (Endereçamento Virtual)

O gerenciamento de memória assegura a alocação eficiente de recursos e o isolamento seguro entre processos. Um dos principais avanços nesse campo é o uso do endereçamento virtual, que proporciona a cada processo a ilusão de um espaço de memória contínuo e exclusivo, mesmo que a memória física seja fragmentada.Esse processo é possibilitado através de uma tabela de tradução de endereços, que mapeia os endereços virtuais para os endereços físicos correspondentes (Tanenbaum & Bos, 2014). A tabela de páginas, por exemplo, é uma estrutura fundamental nesse processo, permitindo que o sistema operacional gerencie eficientemente o acesso à memória e promova o isolamento entre os processos, evitando conflitos de acesso.

No contexto do simulador, a estrutura do PCB foi ampliada para incluir informações essenciais ao gerenciamento de memória, como endereço base, limite e mapeamento de páginas. Com essa expansão, tornou-se possível implementar a tradução de endereços lógicos, que são mais simples e eficientes, para endereços físicos, que exigem maior complexidade e recursos computacionais (Hennessy & Patterson, 2017). No entanto, essa implementação foi feita de forma abstrata, visando simplificar o modelo do sistema operacional.

H. Linguagem C

Para o desenvolvimento desse simulador, foi escolhida a linguagem C, devido ao seu controle preciso sobre o hardware e à sua eficiência. Ela foi criada na década de 1970 por Dennis Ritchie para o sistema operacional Unix, combinando características de linguagens de baixo nível, manipulação direta de memória, e facilidade de uso de linguagens de alto nível (Kernighan & Ritchie, 1988).

A escolha do C se justifica por sua capacidade de acessar diretamente os recursos do sistema, o que é essencial para a implementação de funções como o escalonamento de processos e o gerenciamento de memória. A linguagem também facilita o desenvolvimento de sistemas modulares e eficientes, sendo fundamental para construir simuladores de arquitetura como o proposto (Patterson & Hennessy, 2013).

I. 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

Conforme dito na seção **Quadro Teórico**, para construir o simulador de arquitetura Von Neumann, foi utilizada a linguagem de programação C, escolhida por sua proximidade com o hardware e pela sua eficiência no gerenciamento de recursos computacionais. 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 aqueles descritos na seção **Quadro Teórico**. 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. Tais resultados podem ser encontrados na seção **Resultados e Discissões**.

No segundo estágio da metodologia, correspondente ao **Desenvolvimento e Simulação**, a construção do software foi dividida em cinco etapas principais para garantir uma abordagem didática e estruturada. A primeira parte consistiu na criação do simulador com pipeline MIPS, que forneceu a base para a execução das instruções, aproximando-se do comportamento real do hardware. Na segunda fase, foi implementada uma arquitetura multicore com suporte à preempção, permitindo a execução simultânea de múltiplos processos e a interrupção de tarefas em andamento. A terceira etapa envolveu a implementação do escalonador de processos, enquanto a quarta focou no gerenciamento de cache e na implementação de um mecanismo de escalonamento baseado na similaridade entre os processos. Por fim, a quinta etapa foi dedicada ao gerenciamento de memória, visando otimizar a alocação de recursos e o desempenho geral do sistema.

Para viabilizar a construção do simulador, o projeto foi organizado em quatro diretórios principais, cada um com responsabilidades bem definidas. Essa estrutura modular garantiu uma melhor organização, facilitando a manutenção e a evolução do código ao longo do desenvolvimento.

Os diretórios são os seguintes:

- **build:** Diretório responsável por armazenar os arquivos gerados durante o processo de compilação.
- **dataset:** Diretório responsável por conter os os processos que serão executados pelo simulador.
- **src:** Diretório central, onde o código-fonte foi implementado. Ele contém os arquivos necessários para a execução do simulador.
- **output:** Destinado a armazenar os resultados e logs gerados pelo simulador.

1) dataset: Nesse diretório, estão armazenados os programas executados pelo simulador, os quais seguem um conjunto de instruções específicas, que representam operações essenciais para o seu funcionamento. Essas instruções são

divididas em três categorias, sendo elas: operações aritméticas, instruções de controle de fluxo e manipulação de memória.

- **Operações Aritméticas:** Estas instruções são responsáveis pela realização de cálculos numéricos, como soma, subtração, multiplicação e divisão.
  ADD - Formato - `ADD <REG> <VAL/REG>`
  SUB - Formato - `SUB <REG> <VAL/REG>`
  MUL - Formato - `MUL <REG> <VAL/REG>`
  DIV - Formato - `DIV <REG> <VAL/REG>`

- **Instruções de Controle de Fluxo:** Essas instruções controlam o fluxo de execução do programa, permitindo saltos entre diferentes partes do código com base em condições específicas ou em contagem de ciclos.
  LOOP - Formato - `LOOP <VAL/REG>`
  L_END - Formato - `L_END`
  IF - Formato - `IF <REG> <COND> <VAL/REG>`
  I_END - Formato - `I_END`
  ELSE - Formato - `ELSE`
  ELS_END - Formato - `ELS_END`

- **Manipulação de Memória:** As instruções de manipulação de memória lidam com o armazenamento e recuperação de dados na memória do sistema.
  LOAD - Formato - `LOAD <REG> <VAL>`
  STORE - Formato - `STORE <REG> <MEM>`

A seguir, um exemplo de possíveis processos que podem ser executados no simulador:

	LOAD A0 10
LOAD A0 10	**LOAD C0 1**
LOAD B0 20	**LOOP A0**
ADD A0 B0	**ADD C0 5**
ADD B0 10	**L_END**
STORE A0 A1000	**STORE C0 A1050**
LOAD A0 4	**LOAD A0 8**
LOOP A0	**IF A0 > 2**
ADD A0 2	**ADD A0 5**
L_END	**I_END**
STORE A0 A2010	**STORE A0 A2000**
	STORE B0 A2005

Fig. 4. Exemplo de Processo - Fonte: Própria.

2) src: Para garantir o processamento e armazenamento corretos dos dados, o diretório **src** foi estruturado com 4 subdiretórios, cada um responsável por uma parte essencial do simulador. Essa organização modular facilita a manutenção, leitura e evolução do código ao longo do desenvolvimento, ao mesmo tempo em que separa as funcionalidades de forma clara e eficiente.

- **hardware:** Responsável pela implementação dos componentes físicos simulados, como a CPU, memória e periféricos.
- **software:** Contém a lógica de gerenciamento de processos, escalonamento e execução das instruções. Inclui também o controle dos processos e a manipulação das instruções aritméticas, de controle de fluxo e de manipulação de memória, conforme descrito no diretório **dataset**.
- **system:** Gerencia a interação entre o simulador e o ambiente de execução, conectando todos os recursos necessários para a operação do sistema.
- **utils:** Diretório de suporte que contém funções e imports auxiliares utilizados por várias partes do simulador.

3) output: O diretório **output** foi criado para armazenar os resultados gerados pelo simulador, garantindo o registro adequado da execução dos processos e facilitando a análise do desempenho e comportamento do sistema. Ele contém arquivos de logs detalhados, permitindo depuração e avaliação do funcionamento do escalonador, do gerenciamento de memória e das operações realizadas pela CPU. Essa estrutura possibilita uma melhor rastreabilidade dos dados e auxilia na validação do correto funcionamento do simulador ao longo do desenvolvimento e testes.

Com a estrutura dos diretórios principais definida, a próxima seção detalha a construção do simulador com pipeline MIPS. Cada estágio será descrito junto às decisões técnicas e soluções implementadas, proporcionando uma visão clara e detalhada do desenvolvimento.

A. Construção do Simulador com Pipeline MIPS

Nessa etapa, 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.

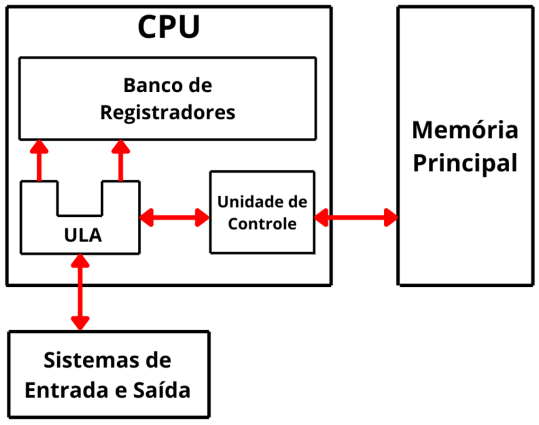


Fig. 5. Arquitetura de Von Neumann - Fonte: Própria.

A execução do simulador tem início no diretório **system**, onde todas as estruturas são inicializadas e conectadas para o

correto funcionamento. Após isso, todos os programas que serão executados pelo simulador são carregados na memoria RAM. Contudo, é necessário verificar se tais instruções estão corretamente escritas, para o bom funcionamento da simulação. Tal tarefa é responsabilidade do Interpretador. Localizado no diretório **software**, o interpretador é responsável por validar as instruções fornecidas ao sistema, garantindo que estejam no formato correto antes de serem processadas. Ele verifica a sintaxe de cada linha do programa, assegurando que os operandos e delimitadores estejam corretos e que o tipo de instrução seja identificado corretamente. Com isso, é garantido que apenas instruções válidas sejam executadas, prevenindo erros que possam ocorrer durante o processamento.

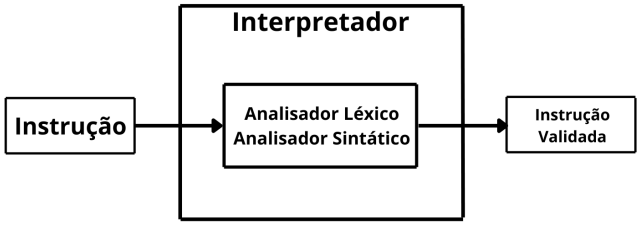


Fig. 6. Interpretador - Fonte: Própria.

Após a validação das instruções pelo interpretador, elas são enviadas para o pipeline de processamento, onde passam por estágios como busca, decodificação e execução. 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. Ao final desse processo, os valores necessários para a execução das operações ou resultantes das operações são carregados e armazenados nos registradores da CPU.

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. 7. Registradores - Fonte: Própria.

Dessa forma, para que cada operação siga seu fluxo adequado, desde seu processamento até seu armazenamento nos registradores, 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.

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 (unidade de execução), 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.

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. Dessa forma, o processo pode ser finalizado, tendo suas informações completamente processadas e suas tarefas 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 novos diretórios, contendo 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).

- **software/process:** Diretório responsável pelo gerenciamento de processos, contendo a implementação do bloco de controle de processos (**PCB**) e das estruturas de fila de processos. Inclui os arquivos **pcb.h**, **pcb.c**, **queue_process.h** e **queue_process.c**.
- **software/threads:** Responsável pelo gerenciamento das threads do simulador, permitindo a execução concorrente dos processos nos núcleos simulados. Contém os arquivos **threads.h** e **threads.c**.

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), Round-Robin e Prioridade Baseada no Menor Valor. Cada uma dessas políticas oferece um comportamento distinto em relação à alocação de CPU para os processos.

No escalonador First Come, First Served (FCFS), os processos são executados na ordem de chegada, sem preempção. Assim que um processo entra na fila de execução, ele aguarda até que todos os processos à sua frente sejam concluídos antes de ocupar a CPU. O tempo de execução (quantum) é atribuído de forma aleatória, o que pode influenciar a duração total do processamento.

No gerenciador de processos Round-Robin, a ordem de execução também segue a sequência de chegada dos processos, mas, diferentemente do FCFS, cada processo recebe um quantum fixo, determinando um limite de tempo para a execução antes de ser temporariamente interrompido e realocado para o final da fila. Esse mecanismo garante uma distribuição mais equitativa do tempo de CPU entre os processos, evitando que um único processo monopolize a execução.

Já no escalonador baseado em Prioridade, cada processo recebe uma prioridade atribuída aleatoriamente no momento da entrada na fila. Antes da execução, os processos são reorganizados do menor para o maior valor de prioridade, garantindo que aqueles com valor mais baixo (prioridade mais baixa) sejam executados primeiro. O quantum de cada processo é definido aleatoriamente, o que pode impactar o tempo total de processamento e a alternância entre os processos na CPU.

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.

Com o intuito de suportar essas novas funcionalidades, foi criado um novo diretório, 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.

- **software/scheduler:** Diretório responsável pelo gerenciamento do escalonamento de processos, contendo a implementação das políticas de escalonamento e a organização da fila de execução. Inclui os arquivos **scheduler.h** e **scheduler.c**.

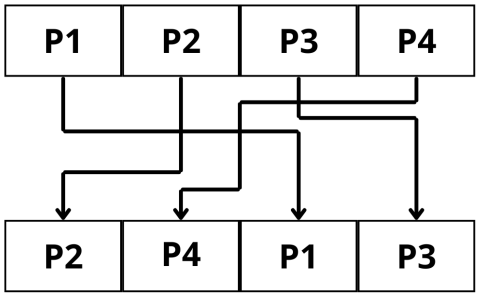


Fig. 8. Escalonador - Fonte: Própria.

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 foi a implementação do gerenciamento de cache e a adoção de uma nova abordagem de escalonamento baseada em similaridade. Essa fase visou

otimizar o desempenho do simulador de arquitetura Von Neumann, com ênfase na redução da latência de acesso à memória e na melhoria da eficiência do escalonamento dos processos.

O gerenciamento de cache foi desenvolvido para simular o funcionamento das hierarquias de memória encontradas nos processadores modernos, com um design simplificado. Ao invés de implementar uma estrutura multi-nível com caches L1, L2 e L3, cada uma com características de capacidade e tempo de acesso distintos, foi adotado um único nível de cache, com capacidade e tempo de acesso uniformes para todos os processos. Mesmo com essa abordagem simplificada, foi possível reproduzir o impacto do cache no desempenho do sistema, conforme descrito na literatura.

A estrutura da cache foi implementada utilizando duas tabelas hash, visando otimizar o armazenamento e a recuperação de dados. A primeira tabela hash, a hash_process, é utilizada para armazenar informações de processos bloqueados, associando o identificador do processo (*process_id*) ao do bloco de controle de processo (*PCB*). A segunda tabela hash, a hash_instruction, armazena instruções recorrentes, associando uma instrução à sua frequência de uso e ao resultado da execução. Essas duas tabelas permitem a otimização do acesso à memória, tanto para processos quanto para instruções frequentemente acessadas.

O gerenciamento da cache segue as seguintes etapas:

- **Inicialização da Cache:** A cache é iniciada com ambas as tabelas hash vazias, prontas para armazenar processos e instruções conforme necessário.
- **Adição de Processos:** Quando um processo é bloqueado, ele é adicionado à tabela hash de processos com seu identificador único e um ponteiro para sua estrutura PCB.
- **Armazenamento de Instruções:** Instruções recorrentes, juntamente com seus resultados e frequência de uso, são armazenadas na tabela hash de instruções, permitindo acessos rápidos e otimização do processamento.
- **Busca e Recuperação:** A cache permite uma busca eficiente por processos e instruções, utilizando as chaves únicas (ID de processo e instrução) para localizar rapidamente as informações necessárias.
- **Remoção e Esvaziamento:** Processos ou instruções podem ser removidos da cache quando não forem mais necessários, ou a cache pode ser esvaziada completamente para liberar memória.

A integração do gerenciamento de cache com o escalonador de processos foi crucial para a melhoria do desempenho do simulador. O escalonamento baseado em similaridade foi introduzido para agrupar processos com padrões de acesso à memória semelhantes, garantindo que processos com comportamentos parecidos fossem executados em sequência, otimizando a reutilização da cache e diminuindo a latência. Como parte dessa mudança, o escalonador foi alterado para incorporar a análise de similaridade antes da execução dos escalonamentos FIFO e Round-Robin. Agora, os processos são classificados com base em um algoritmo de similaridade, que utiliza três métricas: aritmética, controle de fluxo e memória. Os processos são organizados na fila de execução com base nessas pontuações, priorizando aqueles que compartilham características semelhantes, o que melhora a eficiência geral da execução.

Além disso, o gerenciamento da cache foi ajustado para lidar com seu tamanho fixo. Quando a cache atinge sua capacidade máxima, três políticas de remoção são aplicadas para garantir que os dados mais relevantes sejam mantidos. A política FIFO remove o item mais antigo armazenado, enquanto a política Least Recently Used (LRU) descarta o dado que foi menos acessado recentemente. Já a política de Random Replacement escolhe aleatoriamente um item para remoção. Essas políticas são fundamentais para otimizar o uso da cache e minimizar falhas de acesso durante a execução dos processos.

Por fim, a aplicação de conceitos das metodologias ágeis continuou a ser um elemento essencial no desenvolvimento dessa etapa, permitindo melhorias contínuas e ajustes no gerenciamento de cache e no escalonamento dos processos.

E. Gerenciamento de Memória (Endereçamento Virtual)

Por fim, a quinta etapa iniciou com a modificação da estrutura do Bloco de Controle de Processos (PCB), incluindo campos necessários ao gerenciamento de memória. Cada processo passou a ter um endereço base e limite, além de uma tabela de páginas, permitindo o mapeamento eficiente entre os endereços lógicos e físicos.

Para a tradução de endereços lógicos para físicos, optou-se por uma abordagem simplificada de mapeamento de páginas, na qual cada processo possui um endereço virtual associado. Embora, por motivos de simplificações, não tenha sido implementada uma tabela de páginas completa, a ideia foi representar a relação entre endereços virtuais e físicos de maneira direta, mantendo a clareza no entendimento do processo. A tradução dos endereços lógicos é feita por meio da verificação de informações no PCB, utilizando o endereço virtual associado, sem a complexidade adicional de uma tabela de páginas tradicional.

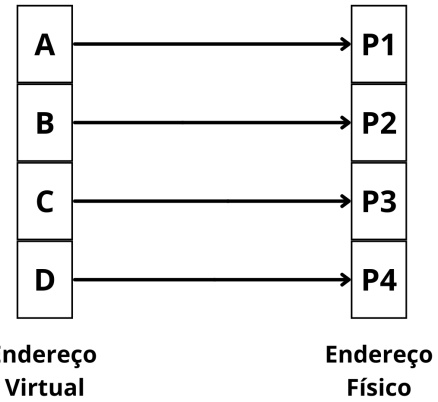


Fig. 9. Endereçamento Virtual - Fonte: Própria.

IV. Resultados e Discussões

Nesta seção, são apresentados os resultados da simulação do sistema operacional simplificado, destacando a validação das etapas e componentes principais.

A. Simulador com Pipeline MIPS

O pipeline MIPS foi implementado com sucesso, garantindo a execução eficiente dos estágios de instruções. Durante os testes, diferentes sequências de instruções foram executadas, e os estágios IF, ID, EX, MEM e WB funcionaram conforme o esperado. A execução ocorreu de forma fluida, com os dados sendo processados de maneira sincronizada em todos os estágios do pipeline.

B. Arquitetura Multicore e Suporte à Preempção

A arquitetura multicore foi validada com a execução de múltiplos processos em paralelo, utilizando threads para representar os núcleos. Nos testes, diferentes processos foram executados simultaneamente em múltiplos núcleos, e a sincronização entre as threads foi garantida, evitando condições de corrida. A preempção mostrou-se eficaz, permitindo a troca de contexto entre os processos e simulando de forma realista a execução concorrente com interrupções controladas.

C. Escalonador de Processos

O escalonador de processos foi testado com as políticas FCFS, Prioridade e Round-Robin. O objetivo foi garantir a alocação correta de processos nos núcleos, respeitando as condições de preempção e os limites de tempo. O sistema foi capaz de selecionar os processos adequados para execução, alternando entre eles com base no tempo de execução acumulado, na prioridade e no quantum de tempo disponível. A preempção foi essencial para a correta troca de contexto, simulando de forma eficiente interrupções e mudanças no fluxo de execução.

D. Gerenciamento de Cache e Escalonamento Baseado em Similaridade

A implementação do gerenciamento de cache demonstrou um sistema eficiente de armazenamento temporário de processos, simulando uma memória hierárquica. Testes com diferentes tamanhos de cache e padrões de acesso à memória mostraram que a cache reduziu o tempo de acesso à memória, especialmente para processos com alta localidade espacial e temporal. O escalonamento baseado em similaridade, que agrupa processos com padrões de acesso semelhantes, otimizou o uso da cache e reduziu falhas de memória. Essa estratégia validou a importância de um gerenciamento eficiente da cache para melhorar o desempenho geral do sistema.

E. Gerenciamento de Memória

A implementação do gerenciamento de memória, com o uso de endereçamento virtual e uma abordagem simplificada de mapeamento de páginas, garantiu a tradução eficiente dos endereços lógicos para físicos. Embora a tabela de páginas não tenha sido implementada de forma completa, a abstração permitiu validar a tradução de endereços e o gerenciamento eficiente da memória. A estrutura do PCB foi ampliada para incluir o endereço base, limite e mapeamento de páginas, possibilitando uma gestão mais eficiente dos recursos de memória, sem comprometer a simplicidade do simulador.

F. Discussão

A análise de desempenho do simulador foi realizada por meio de dez simulações, comparando o número de núcleos utilizados com o tempo de execução dos processos em diferentes configurações. Os testes envolveram oito processos, explorando diversas combinações de instruções aritméticas, de controle e de memória. O número de núcleos variou entre 1, 2, 3 e 4, permitindo diferentes níveis de paralelismo. Para garantir a precisão dos resultados, cada cenário foi executado cinco vezes, e a média desses valores foi utilizada.

Os resultados foram representados em um gráfico com dez colunas, cada uma correspondendo a um cenário de simulação. O primeiro cenário, usado como referência, não emprega cache nem escalonamento avançado. Os nove seguintes combinam três políticas de escalonamento (FCFS, Prioridade e Round-Robin) com três estratégias de cache (FIFO, LRU e substituição aleatória). Isso permitiu avaliar o impacto dessas configurações no desempenho do simulador.

Teoricamente, o aumento no número de núcleos deve reduzir o tempo de execução devido à maior distribuição da carga de trabalho. No entanto, o ganho pode ser limitado por contenção de recursos, sincronização entre threads e sobrecarga na troca de contexto. Além disso, o gerenciamento de cache e as políticas de escalonamento influenciam diretamente o desempenho do simulador. Estratégias de cache como FIFO, LRU e substituição aleatória afetam a taxa de acertos e o tempo de acesso à memória, sendo esperado que LRU reduza falhas e melhore a eficiência. No escalonamento, FCFS pode ser ineficiente para cargas heterogêneas, enquanto Round-Robin melhora a padronização. O uso de prioridades favorece processos críticos, mas pode comprometer a execução de tarefas de menor prioridade.

Por fim, espera-se que análise do gráfico construído permita validar essas previsões teóricas, evidenciando como diferentes combinações de escalonamento e cache influenciam o desempenho do simulador.

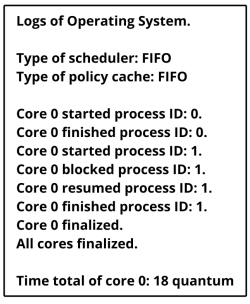
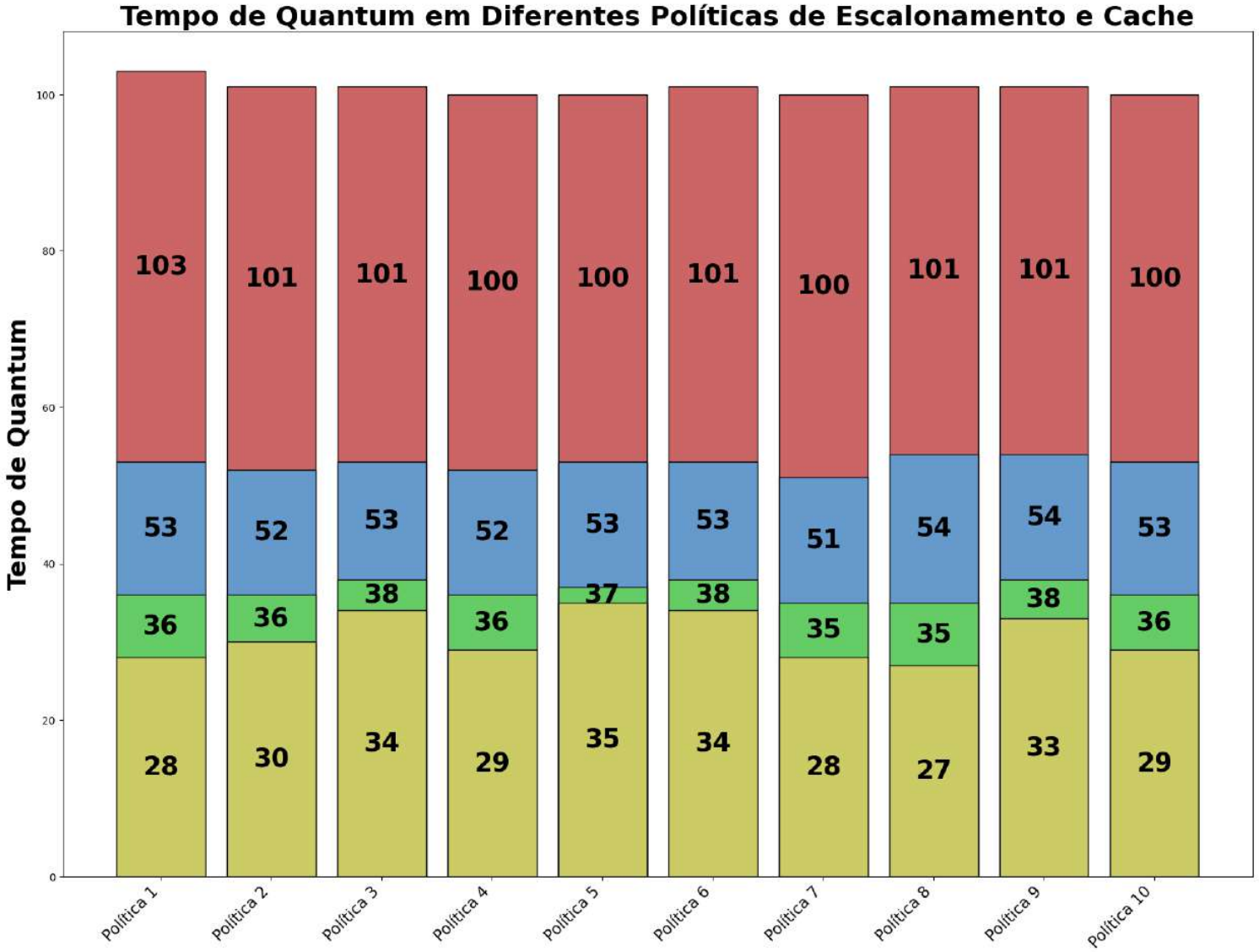


Fig. 10. Exemplo de Simulação - Fonte: Própria.



Legenda

- 1 Core
- 2 Cores
- 3 Cores
- 4 Cores

Política 1: Escalonamento: FIFO sem Similaridade, Sem Cache
Política 2: Escalonamento: FIFO com Similaridade, Cache: FIFO
Política 3: Escalonamento: ROUND_ROBIN com Similaridade, Cache: FIFO
Política 4: Escalonamento: PRIORITY com Similaridade, Cache: FIFO
Política 5: Escalonamento: FIFO com Similaridade, Cache: LEAST_RECENTLY_USED
Política 6: Escalonamento: ROUND_ROBIN com Similaridade, Cache: LEAST_RECENTLY_USED
Política 7: Escalonamento: PRIORITY com Similaridade, Cache: LEAST_RECENTLY_USED
Política 8: Escalonamento: FIFO com Similaridade, Cache: FIFO
Política 9: Escalonamento: ROUND_ROBIN com Similaridade, Cache: FIFO
Política 10: Escalonamento: PRIORITY com Similaridade, Cache: FIFO

G. Análise dos Resultados

No gráfico, observa-se o impacto positivo da combinação entre escalonamento e política de cache na otimização do desempenho. Os cenários que apresentaram os melhores resultados, como as políticas 7 e 8, evidenciam a sinergia entre a organização eficiente de dados e instruções. A política de cache Least Recently Used (LRU) demonstra seu potencial ao reduzir a recarga frequente de dados, explorando a localidade temporal e minimizando a latência de acesso à memória. Da mesma forma, o escalonamento baseado em similaridade se destaca ao agrupar tarefas com padrões de execução semelhantes, reduzindo a frequência de trocas de contexto e melhorando o aproveitamento dos recursos computacionais.

Embora o estudo tenha sido conduzido em um simulador, sujeito a erros inerentes à modelagem e implementação, a grande maioria dos resultados obtidos está alinhada com a teoria de sistemas operacionais e gerenciamento de memória. O comportamento observado para as diferentes políticas corrobora princípios amplamente aceitos, como a superioridade de caches mais inteligentes na redução da latência e a importância do escalonamento eficiente para a distribuição equilibrada dos recursos. Pequenas variações nos resultados podem ser atribuídas a limitações do modelo ou a fatores específicos do ambiente simulado, mas não comprometem a validade das conclusões.

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/.