

Simulador para a Arquitetura de Von Neumann e Pipeline MIPS*

1st Anderson Rodrigues Dos Santos
Engenharia de Computação
CEFET-MG - Campus V
Divinópolis, Brasil
andersonlagoagrandesantos@gmail.com

2nd Bruno Prado Dos Santos
Engenharia de Computação
CEFET-MG - Campus V
Divinópolis, Brasil
bruno.pradosantos1910@gmail.com

3rd Eduardo Henrique Queiroz Almeida
Engenharia de Computação
CEFET-MG - Campus V
Divinópolis, Brasil
eduardohenriquecruzeiro123@gmail.com

4th Sergio Henrique Quedas Ramos
Engenharia de Computação
CEFET-MG - Campus V
Divinópolis, Brasil
sergohenriquequedasramos@gmail.com

Abstract—Este trabalho apresenta o desenvolvimento e a análise de desempenho de um simulador de arquitetura multicore com memória compartilhada, projetado para consolidar conceitos fundamentais de sistemas operacionais e arquitetura de computadores. O sistema proposto evolui uma arquitetura base single-core MIPS (Von Neumann) para um ambiente de n núcleos de processamento, integrando mecanismos robustos de gerenciamento de memória segmentada e múltiplas estratégias de escalonamento de processos. A implementação contempla a execução de cargas de trabalho sob políticas de escalonamento preemptivas e não preemptivas, incluindo First Come, First Served (FCFS), Shortest Job Next (SJN), Round Robin (RR) e Prioridade. O gerenciamento de memória adota um modelo de mapeamento de endereços e políticas de substituição (FIFO/LRU), inspirado na literatura de Tanenbaum. Os resultados foram obtidos através de métricas quantitativas, como throughput, tempo médio de espera e utilização da CPU, permitindo uma análise comparativa crítica entre a eficiência da arquitetura paralela proposta e o baseline sequencial.

Index Terms—Arquitetura Multicore, Escalonamento de Processos, Gerenciamento de Memória, Simulação, MIPS.

I. INTRODUÇÃO

A computação de alto desempenho contemporânea depende intrinsecamente do paralelismo. A transição de processadores uniprocessados para arquiteturas multicore introduziu novos paradigmas no design de sistemas operacionais, exigindo estratégias sofisticadas para o compartilhamento de recursos críticos, como tempo de CPU e memória principal. A compreensão prática desses mecanismos é fundamental para a engenharia de sistemas modernos.

Este artigo descreve o desenvolvimento e a avaliação de um simulador de arquitetura multicore, projetado como uma evolução direta de uma arquitetura baseline de Von Neumann com pipeline MIPS. O objetivo central do trabalho foi estender o modelo single-core original para suportar n núcleos de processamento concorrentes, implementando camadas completas de gerenciamento de memória e escalonamento de processos que inexistem na versão base.

Diferentemente de simulações puramente teóricas, esta implementação aborda problemas práticos de concorrência e gestão de recursos. No subsistema de memória, desenvolveu-se um gerenciador baseado no modelo de mapeamento por palavras de bits e segmentação, suportando políticas de substituição de páginas FIFO (First-In, First-Out) e LRU (Least Recently Used). Essa camada permite a carga e execução de múltiplos programas em um espaço de endereçamento compartilhado, simulando a disputa real por recursos de armazenamento.

No que tange ao gerenciamento de processos, o simulador incorpora quatro estratégias distintas de escalonamento: First Come, First Served (FCFS), Shortest Job Next (SJN), Round Robin (RR) com quantum definido e escalonamento por Prioridade (preemptiva e não preemptiva). A implementação suporta a troca de contexto e a preservação de estado dos processos interrompidos, permitindo uma análise comparativa direta entre a eficiência de algoritmos preemptivos e não preemptivos em um ambiente paralelo.

A validação do sistema foi realizada sob uma carga de trabalho determinística (lote estático de programas), coletando-se métricas de throughput, tempo médio de espera e utilização da CPU. Os resultados apresentados nas seções subsequentes demonstram o impacto da arquitetura multicore desenvolvida sobre o desempenho global, contrastando-a com o baseline sequencial e evidenciando os trade-offs entre as diferentes políticas de escalonamento implementadas.

II. REFERENCIAL TEÓRICO

Esta seção fundamenta as decisões de projeto do simulador, baseando-se na literatura clássica de sistemas operacionais e arquitetura de computadores. A discussão é dividida em arquiteturas de multiprocessamento, hierarquia de memória e algoritmos de escalonamento.

A. Arquiteturas Multicore e Pipeline

A evolução histórica dos processadores atingiu barreiras físicas relacionadas à dissipação térmica e ao consumo de energia, fenômeno descrito como *Power Wall*. Para superar tais limitações e manter a Lei de Moore, a indústria migrou para arquiteturas *multicore*, explorando o Paralelismo em Nível de Thread (TLP) [3].

O simulador proposto modela um sistema de Multiprocessamento Simétrico (SMP). Segundo Stallings [2], em um ambiente SMP, dois ou mais processadores idênticos compartilham a mesma memória principal e são controlados por uma única instância do sistema operacional. O principal desafio neste modelo é a gestão da concorrência no acesso ao barramento e à memória, exigindo mecanismos de arbitragem para evitar gargalos de desempenho.

Internamente, cada núcleo segue o modelo RISC (*Reduced Instruction Set Computer*) com a implementação de um *pipeline* de instruções. Conforme detalhado por Hennessy e Patterson [3], o *pipelining* aumenta o *throughput* instrucional ao sobrepor a execução de múltiplas instruções. O modelo clássico MIPS, adotado neste trabalho, divide a execução em cinco estágios: Busca de Instrução (IF), Decodificação (ID), Execução (EX), Acesso à Memória (MEM) e Escrita nos Registradores (WB).

B. Gerência de Memória e Hierarquia

A memória é um recurso finito e de alta latência em comparação à velocidade da CPU, criando o problema conhecido como *Memory Wall*. Para mitigar essa disparidade, utiliza-se o princípio da localidade (temporal e espacial), descrito por Bryant e O'Hallaron [4], que justifica o uso de hierarquias de memória (caches).

O gerenciamento de memória em sistemas multitarefa deve garantir isolamento e proteção. Tanenbaum e Bos [1] explicam que técnicas como a segmentação e a paginação permitem abstrair a memória física em endereços lógicos. Quando a memória física se esgota, o sistema operacional deve decidir quais dados remover, utilizando algoritmos de substituição de páginas:

- **FIFO (First-In, First-Out):** Seleciona a página mais antiga para remoção. Apesar da simplicidade de implementação, Tanenbaum [1] alerta para a "Anomalia de Belady", onde o aumento da memória disponível pode, contra-intuitivamente, aumentar a taxa de falhas de página.
- **LRU (Least Recently Used):** Baseia-se na premissa de que páginas usadas recentemente provavelmente serão usadas novamente em breve (localidade temporal). Embora tenha desempenho próximo ao ótimo, sua implementação exata é custosa em hardware, exigindo frequentemente aproximações em software [2].

C. Políticas de Escalonamento de Processos

O escalonamento de CPU é central para sistemas multiprogramados, onde o objetivo é maximizar a utilização do pro-

cessador. Tanenbaum [1] categoriza os algoritmos conforme o ambiente (lote, interativo ou tempo real) e a preempção.

1) *Escalonamento Não Preemptivo*: Nestes algoritmos, o processo retém a CPU até que termine ou realize uma operação de I/O.

- **FCFS (First-Come, First-Served):** O processo que solicita a CPU primeiro é o primeiro a ser servido. É justo, mas ineficiente para processos I/O-bound, podendo gerar o "efeito comboio", onde processos rápidos aguardam indevidamente atrás de processos longos [2].
- **SJN (Shortest Job Next):** Prioriza processos com menor tempo de execução estimado. Embora provável de minimizar o tempo médio de espera, sofre com a dificuldade prática de prever a duração exata de um processo [1].

2) *Escalonamento Preemptivo*: Introduz a capacidade de interromper um processo em execução, exigindo mecanismos de troca de contexto para salvamento de estado (registradores, PC, flags) [4].

- **Round Robin (RR):** Atribui fatias de tempo (*quantum*) iguais a cada processo. A escolha do tamanho do *quantum* é crítica: se muito curto, o *overhead* de troca de contexto domina o desempenho; se muito longo, degenera-se em FCFS [1].
- **Prioridade:** Processos importantes são executados antes. Stallings [2] destaca o problema de *starvation* (inanição) para processos de baixa prioridade, que pode ser mitigado por técnicas de "envelhecimento" (*aging*), onde a prioridade de um processo aumenta conforme o tempo de espera.

III. METODOLOGIA E IMPLEMENTAÇÃO

A arquitetura do simulador foi projetada para emular um ambiente de Multiprocessamento Simétrico (SMP) sobre uma infraestrutura de software orientada a objetos. O sistema foi desenvolvido em C++ (padrão C++17), priorizando o gerenciamento explícito de memória e o uso de estruturas de dados eficientes para minimizar o *overhead* da própria simulação. O código-fonte e as instruções de compilação via CMake estão disponíveis no repositório do projeto [5].

A. Arquitetura do Simulador e Ambiente

O núcleo do sistema é a classe *MultiCore*, que atua como o *backplane* do hardware simulado. Diferente de abordagens puramente funcionais, este trabalho adota uma simulação orientada a ciclos (*cycle-accurate simulation*). Um *clock* global sincroniza os componentes, garantindo que as operações de busca, decodificação e execução ocorram em etapas discretas, permitindo a aferição precisa do tempo de CPU utilizado por cada processo.

A entrada de dados simula o comportamento de um *Loader* de Sistema Operacional. Utilizando a biblioteca *nlohmann/json* [6], o sistema realiza o *parsing* de arquivos de definição de processos, que contêm o segmento de código (instruções mnemônicas MIPS) e o segmento de dados. Esta abordagem desacopla a definição da carga de trabalho da lógica do simulador, garantindo reprodutibilidade nos testes.

B. Modelagem do Processador e Pipeline

Cada unidade de processamento é instanciada pela classe `Core`, que modela uma arquitetura RISC baseada no MIPS. A implementação do hardware compreende:

- **Banco de Registradores:** Um vetor de estruturas representando os 32 registradores de propósito geral, além do PC (*Program Counter*) e IR (*Instruction Register*).
- **Unidade de Controle e ULA:** A classe `CONTROL_UNIT` decodifica os *opcodes* e direciona as operações para a Unidade Lógica e Aritmética (ULA), simulando o fluxo de dados no *datapath*.

O *pipeline* é simulado logicamente. Embora a execução no *host* seja sequencial, o simulador contabiliza as latências de cada estágio (Busca, Decodificação, Execução, Memória, Escrita), permitindo analisar o impacto de riscos de dados e controle [3].

C. Gerenciamento de Memória e Virtualização

O subsistema de memória, gerido pela classe `MemoryManager`, implementa uma abstração de memória virtual segmentada. A memória física (`MAIN_MEMORY`) é tratada como um recurso compartilhado protegido por *mutexes* para garantir a coerência em acessos concorrentes pelos múltiplos núcleos.

O gerenciador é responsável pela tradução de endereços lógicos (utilizados pelos programas simulados) para endereços físicos reais do vetor de armazenamento. As políticas de substituição de páginas (FIFO e LRU) foram implementadas através de listas encadeadas e mapas de acesso, monitorando o bit de referência de cada página para decidir a vítima em caso de *page fault*, conforme descrito por Tanenbaum [1].

D. Escalonamento e Troca de Contexto

O componente `Scheduler` centraliza a decisão de alocação de CPU. A estrutura fundamental para o gerenciamento de processos é o PCB (*Process Control Block*), que armazena:

- Estado dos registradores e PC (Contexto de Hardware);
- Identificação (PID) e prioridade;
- Estatísticas de execução (tempo de chegada, tempo de *burst*).

A troca de contexto (*context switch*) não é apenas uma mudança de ponteiros; o simulador realiza a cópia efetiva dos valores dos registradores do núcleo para o PCB do processo que sofre preempção, e vice-versa. Para políticas preempativas, como o *Round Robin*, um temporizador de *quantum* decrementa a cada ciclo de instrução. Ao atingir zero, uma interrupção de software é disparada, forçando a chamada da rotina de escalonamento [2].

E. Protocolo Experimental

Para validar o sistema, foram definidos cenários de teste cobrindo cargas de trabalho CPU-bound e I/O-bound. As métricas de desempenho — *Throughput* (processos/unidade de tempo), Tempo Médio de Espera e Tempo de Retorno

(*Turnaround*) — são coletadas automaticamente pela classe `Metrics` ao final da execução de cada lote, permitindo a comparação quantitativa entre as políticas FCFS, SJN, Prioridade e Round Robin.

IV. RESULTADOS E DISCUSSÃO

A avaliação do simulador foi conduzida em um ambiente controlado (Linux, G++ 11.4.0), utilizando uma carga de trabalho determinística composta por 20 processos sintéticos. O objetivo foi validar a corretude funcional dos módulos e quantificar o ganho de desempenho obtido com a introdução do paralelismo.

A. Validação Funcional

Antes da coleta de métricas de desempenho, o sistema foi submetido a um conjunto de testes de integração e estresse. Conforme os relatórios de validação, o simulador obteve aprovação em 100% dos 28 subtestes, cobrindo casos de borda, violação de segmento de memória e coerência de pipeline. Destaca-se a validação do subsistema de memória, que manteve a integridade dos dados mesmo sob alta concorrência de acesso.

B. Análise de Escalonamento

O primeiro cenário avaliou o impacto das políticas de escalonamento em uma configuração *single-core*. A Tabela I apresenta os tempos médios de espera e retorno (*turnaround*) obtidos.

TABLE I
DESEMPENHO COM 1 NÚCLEO (20 PROCESSOS)

Política	Espera (ciclos)	Retorno (ciclos)	Throughput
FCFS	69.80	78.95	0.110
Round Robin	69.80	78.95	0.110
Prioridade	69.20	78.35	0.110
SJN	69.80	78.95	0.110

Observa-se que a política de **Prioridade** apresentou o melhor desempenho, reduzindo o tempo médio de retorno em 0.60 ciclos comparado ao *baseline* (FCFS). A similaridade entre FCFS, RR e SJN deve-se à natureza da carga de trabalho: processos curtos e homogêneos tendem a nivelar o desempenho de algoritmos baseados em tempo, mitigando o “efeito comboio” típico do FCFS.

C. Impacto do Multiprocessamento

O segundo cenário analisou a escalabilidade da arquitetura ao aumentar o número de núcleos para $n = 2$ e $n = 4$, mantendo a mesma carga de 20 processos. A Tabela II compara o tempo de retorno da melhor política (Prioridade) em diferentes configurações.

A introdução de múltiplos núcleos reduziu drasticamente os tempos de resposta. Com 4 núcleos, o tempo médio de retorno caiu para 20.35 ciclos. Nota-se, contudo, um leve aumento no tempo de espera relativo em configurações de 4 núcleos (menionado nos logs de teste), característico do aumento da disputa (*contention*) pelo barramento de memória compartilhada.

TABLE II
TEMPO DE RETORNO (PRIORITY) VS. N° DE NÚCLEOS

Configuração	Tempo de Retorno	Redução (%)
1 Core	78.35	-
2 Cores	31.55	59.7%
4 Cores	20.35	74.0%

D. Speedup e Eficiência

Para quantificar o ganho do paralelismo, calculou-se o *Speedup* (S) conforme a Equação (1), onde T_1 é o tempo de execução sequencial e T_n o tempo com n núcleos.

$$S = \frac{T_1}{T_n} \quad (1)$$

Os resultados obtidos demonstram uma escalabilidade quase linear:

- **2 Núcleos:** Speedup de **1.91x** (Eficiência de 95.5%).
- **4 Núcleos:** Speedup de **3.73x** (Eficiência de 93.2%).

Esses valores indicam que o *overhead* de gerenciamento de processos e sincronização de memória foi mantido baixo. A ligeira perda de eficiência (de 95% para 93%) ao dobrar de 2 para 4 núcleos é esperada e condizente com a Lei de Amdahl, refletindo a saturação progressiva do acesso à memória principal.

E. Gerenciamento de Memória

Durante os testes de estresse, o mecanismo de cache simulado atingiu uma taxa de acerto (*hit rate*) média de 60%. Isso valida a eficácia da implementação do princípio da localidade temporal, reduzindo a latência efetiva média de acesso aos dados (AMAT) mesmo em cenários de alta demanda por parte de múltiplos núcleos.

V. CONCLUSÃO E TRABALHOS FUTUROS

O desenvolvimento deste simulador de arquitetura multicore atingiu o objetivo de integrar, em um ambiente prático, os conceitos fundamentais de sistemas operacionais e organização de computadores. A transição da arquitetura base Von Neumann para um sistema de Multiprocessamento Simétrico (SMP) permitiu observar empiricamente os desafios de sincronização e coerência descritos na literatura.

Os resultados experimentais demonstraram que a introdução do paralelismo oferece ganhos significativos de desempenho, atingindo um *speedup* de 3.73x com quatro núcleos, uma eficiência de 93.2%. Essa observação valida a Lei de Amdahl, evidenciando que, embora o paralelismo acelere a execução, o acesso concorrente a recursos compartilhados (memória) impõe limites físicos ao ganho linear. Entre as políticas de escalonamento avaliadas, a estratégia baseada em Prioridade mostrou-se superior para a carga de trabalho mista, oferecendo o melhor equilíbrio entre *throughput* e tempo de retorno.

No que tange ao gerenciamento de memória, a simulação da hierarquia com cache e a implementação de políticas de substituição (LRU/FIFO) comprovaram a importância do princípio da localidade para mitigar o *Memory Wall*.

Para trabalhos futuros, sugere-se a evolução do modelo de memória para suportar paginação multinível com TLB (*Translation Lookaside Buffer*), aproximando a simulação de sistemas modernos como Linux e Windows. Além disso, a implementação de suporte a I/O assíncrono e interrupções de hardware reais permitiria testar algoritmos de escalonamento mais complexos, como Filas de Múltiplos Níveis com Realimentação.

REFERENCES

- [1] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Pearson, 2015.
- [2] W. Stallings, *Computer Organization and Architecture*, 10th ed. Pearson, 2016.
- [3] J. L. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed., Morgan Kaufmann, 2019.
- [4] R. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed., Pearson, 2016.
- [5] Equipe de Desenvolvimento, “Simulador de Arquitetura Multicore com Gerenciamento de Memória”, Repositório GitHub. Disponível em: <https://github.com/anderrsantos/so-simuladorvonnewmann>. Acesso em: 06 dez. 2025.
- [6] N. Lohmann, “JSON for Modern C++”, 2023. [Online]. Disponível em: <https://github.com/nlohmann/json>.