Técnicas de Sistemas Operacionais numa arquitetura Von Newmann e MIPS simulada

1st Getúlio Santos Mendes
DECOM-DV
CEFET-MG Campus V
Divinópolis, Brasil
getuliosantosmendes@gmail.com

2nd Frank Leite Lemos Costa
DECOM-DV
CEFET-MG Campus V
Divinópolis, Brasil
frankcefet090@gmail.com

***Abstract*—Este artigo apresenta o processo, escolhas e resultados do desenvolvimento de uma arquitetura simulada baseada na arquitetura MIPS; visando aplicar, analisar e discutir didaticamente conceitos fundamentais para sistemas operacionais modernos.**

***Index Terms*—Arquitetura de Von Neumann, Arquitetura MIPS, Sistemas Operacionais, CPU, Multi-core, Preempção, Pipeline**

I. Introdução

Esse trabalho foi desenvolvido como parte da disciplina de Sistemas Operacionais e subdivido em módulos correspondentes a conceitos fundamentais para sistemas operacionais modernos.

O principal objetivo é a implementação de um simulador de uma arquitetura simplificada com conceitos de pseudo-paralelismo (Pipeline) e paralelismo real e a construção de um sistema operacional simplificado para gerenciar esses recursos de Hardware simulados com finalidade de obter uma ferramenta de teste e aprendizado de conceitos e estratégias presentes em Sistemas Operacionais Modernos.

II. Metodologia

A linguagem de implementação escolhida foi C++, devido ao seu suporte a tanto recursos de mais alto nível de abstração, como a possibilidade de operações de baixo nível e controle direto sobre memória; além de seu bom desempenho.

A. Arquitetura MIPS customizada

A arquitetura MIPS é um exemplo de processador RISC (Reduced Instruction Set Computing), projetada para maximizar eficiência e desempenho com um conjunto reduzido e simplificado de instruções.

O pipeline é uma característica central da MIPS, permitindo a execução simultânea de diferentes etapas de várias instruções. Dividindo a execução em estágios independentes e aumentando o *throughput* geral do processador. Essa abordagem torna a arquitetura MIPS altamente eficiente e ideal para aplicações de alto desempenho, como sistemas embarcados e dispositivos de consumo.

O simulador foi diretamente inspirado pela arquitetura, buscando fidelidade dentro do possível, sendo dividido em Unidade Lógica e Aritmética (ULA), Unidade de Controle, Registradores e memória principal e memória secundária; usando os mesmos estágios de pipeline que a MIPS: busca, decodificação, execução, acesso à memória e escrita de volta (*Fetch, Decode, Execute, Memory Access, Write back*.

No estágio *Fetch*, a próxima instrução é buscada da memória, com base no contador de programa (PC), enquanto o PC é atualizado para a próxima instrução. Em seguida, no estágio *Decode*, a instrução é decodificada, identificando os registradores e valores necessários para a execução.

No estágio Execute, a operação especificada pela instrução é realizada, com suporte da Unidade Lógica e Aritmética (ULA) para cálculos ou ajustes no PC em caso de saltos e *loops*. O estágio *Memory Access* trata de leitura na memória para instruções como *load* e *print*. Por fim, no estágio *Write Back*, os resultados são armazenados dos registradores à memória, onde instruções como *store* são implementadas.

B. Arquitetura Multicore e Preempção básica

Neste modulo foi implementado uma arquitetura multicores com o uso da biblioteca *pthreads*, a arquitetura single core foi abstraída de modo que para a execução de cada core da CPU bastasse criar um *thread* dentro da linguagem de programação utilizada que execute essa abstração de core. É claro que isso inevitavelmente introduz problemas a respeito do acesso de recursos em paralelo, além de problemas a respeito da alocação de programas para executar em cada um dessas cores.

Para resolver esses problemas, foi implementado um mecanismo básico de preempção. O Sistema Operacional mantém uma fila de processos a serem executados, seus estados e recursos em seu *Program Control Block* (PCB). Cada *thread* representando um core busca um processo para a execução dentro dessa fila e executa esse processo por um determinado *quantum* que representa uma quantidade de instruções a serem executadas.

Além disso, um *thread* roda dedicadamente a função de Escalonador de Processos, que verifica se todos os processos foram totalmente executados para finalizar o simulador. Também de organizar a fila conforme o necessário para obter a estratégia de escalonamento desejada.

Um *thread* dedicado também faz o gerenciamento das requisições de *output*, ele utiliza um *delay* artificial para gerar o bloqueio dos processos, enquanto o processo está

bloqueado, o núcleo busca outro programa para executar; quando a requisição é atendida o processo é desbloqueado e pode voltar a executar.

C. Implementação do Escalonador de Processos

Foram implementados quatro políticas de escalonamento: FCFS (i.e., *First Come First Service*), baseado em prioridade, randomizado e SJF (i.e. *Shortest Job first*).

Em todas as estratégias, cada core busca o primeiro processo disponível da lista e o executa até atingir o quantum determinado. A diferença é como o *thread* do *scheduller* modifica a lista para atingir os resultados desejados. Isso permite a fácil implementação de outras estratégias de escalonamento com poucas modificações, pois basta pensar uma forma de organizar e priorizar os projetos.

1) FCFS: O *thread* do escalonador joga os processos que estão rodando no final da fila. Como os core pegam do início da fila, efetiva-se o *First come first served*. Essa estratégia busca equilibrar o tempo de execução entre os processos de uma forma simples de implementar.
2) Prioridade: Os processos são ordenados pelo escalonador com base na prioridade, definida com base na ordem dos argumentos passados ao programa. Além disso, processos com maior prioridade possuem maior quantum. Isso permite ao usuário escolher processos importantes para garantir que eles não sofram de inanição.
3) Randomizado: A lista de programas é embaralhada pelo escalonador, sendo o impossível de prever qual será o próximo processo a executar. Todos os programas possuem a mesma chance de serem o próximo.
4) SJF (*Shortest Job First*): É calculada uma estimativa de tempo de execução com base no número de instruções presentes no programa. Além disso, instruções de *jump* têm seu valor dez vezes maior na estimativa do que instruções normais. Isso é justificável, pois muitas vezes a maior parte do tempo de execução de programas acontece dentro de *loops*. Pressupor que cada *loop* será executado em média dez vezes é razoável considerando o escopo pequeno dos programas feitos para nossa arquitetura. O escalonador ordena a lista de processos com base nessas estimativas, dando preferência para programas pequenos.

D. Gerenciamento de Cache e Escalonamento por Similaridade

Neste módulo, implementou-se uma memória *cache* com o objetivo de otimizar o desempenho do simulador e reforçar a importância didática das hierarquias de memória na performance de sistemas modernos.

Para reproduzir condições reais de hardware, a cache foi configurada com um tamanho significativamente menor que o da memória principal. Essa escolha reflete os custos elevados de fabricação de memórias de alta velocidade e as restrições de espaço físico disponíveis para elas.

A fim de simular o acesso rápido característico das memórias cache, foi utilizada uma tabela *hash*, que possibilita acesso em tempo constante O(1). Essa estrutura é capaz de armazenar tanto dados provenientes da memória principal quanto os resultados de instruções, permitindo que o processador reaproveite informações geradas em *jobs* anteriores.

O tamanho dessa *hash* foi escolhido como dez valores de variáveis ou operações, um valor razoavelmente pequeno considerando o tamanho dos programas de teste que ficam entre 30 e 20 instruções cada.

Adicionalmente, implementou-se um tempo de espera para os acessos à memória principal, com o intuito de simular a diferença de desempenho entre o acesso direto a essa memória e o acesso via cache.

Sempre que um dado na memória principal ou operação na ULA é requisitado, verifica-se se a informação já se encontra na cache. Se sim, esse valor é utilizado ou atualizado, poupando tempo. Caso o valor não esteja presente, ele é adicionado à cache, as políticas de gerenciamento decidem qual valor deverá ser removido para acomodar o novo dado.

Foram implementadas duas políticas de substituição:

1) *LRU (Least Recently Used)*: Remove o elemento que não foi acessado por mais tempo.
2) *FIFO (First In First Out)*: Remove o elemento que está na cache há mais tempo.

Para possibilitar a aplicação dessas políticas, o instante do último acesso de cada valor é registrado juntamente com o próprio dado. No caso da política LRU, esse registro é atualizado a cada acesso, garantindo que a decisão de remoção se baseie no tempo de uso e não apenas no tempo de inserção.

Embora o escalonador pudesse se beneficiar do agrupamento de jobs semelhantes—com base em um escore de similaridade, a fim de maximizar o reaproveitamento de dados e operações—a análise de semelhança mostrou-se excessivamente complexa para este simulador. Dada a diversidade de operações envolvendo registradores e variáveis, identificar, com apenas uma varredura pelo código, quais operações serão realizadas não é trivial, especialmente se almejamos uma solução com complexidade O(n).

Como um dos objetivos do escalonador é minimizar os custos de processamento, a implementação de uma análise detalhada de similaridade em tempo de execução se torna inviável. Uma alternativa seria realizar essa análise durante a carga do programa na memória principal, mas seria necessário avaliar se o tempo adicional gasto compensa os ganhos obtidos ao agrupar jobs.

Diante dessas considerações, optou-se por não implementar um mecanismo de preempção baseado em similaridade. Em vez disso, buscou-se comparar a performance dos mecanismos de cache existentes em relação à memória principal e a operações de natureza similar.

E. Gerência de Memória e PCB

Programas para sistemas operacionais modernos não trabalham diretamente com a memória física do computar, mas em

cima da virtualização da memória feita pelo sistema operacional. O processo de virtualização é complicado e em parte implementado em nível de hardware devido a complexidade de gerir as tabelas, páginas e endereços virtuais.

A fim de simplificação, mas ainda com o uso de técnicas semelhantes às modernas, uma versão simplória do processo foi escolhida para o simulador. Apesar de simples pode ser expandida futuramente para análises mais consistente com sistemas reais.

O modelo adotado foi o de segmentação, devido a sua maior simplicidade comparado ao modelo de paginação. Isso limita a flexibilidade da virtualização (ex.: não há suporte a memória swap ou proteção granular).

Primeiramente, o carregamento dos programa passa a utilizar sempre endereços relativos ao seu próprio começo, sempre posição 0 de memória, de modo que todos os programas enxergam os recursos como se eles fossem os únicos dados da memória.

Uma simples MMU(*Memory management unit*) foi criada para gerenciar acesso dos programas às suas posições, fazem a tradução do endereço virtual (relativo ao início do programa em 0) e os endereços reais da RAM. Medidas de segurança também foram implementadas para impedir que programas acessem zonas de memória fora da sua. Todo esse processo se resume à somar o endereço base ao endereço de acesso e verificar se o resultado não ultrapassa o endereço de teto.

Em sistemas multicore, o gerenciamento de memória requer sincronização de estruturas (como as tabelas de páginas) para evitar inconsistências. No simulador, a MMU simplificada (base/teto) evita conflitos e simplifica o processo.

A ausência de estruturas compartilhadas (e.g., tabelas de páginas globais) elimina a necessidade de sincronização complexa (como *locks* ou operações atômicas) entre núcleos. Cada processo gerencia seu próprio espaço via base/teto, reduzindo contenção e riscos de inconsistência.

III. Resultados e Discussões

A. Arquitetura MIPS customizada

A implementação do simulador inspirado na arquitetura MIPS mostrou resultados alinhados aos princípios do design RISC, reproduzindo com fidelidade o *assembly* da arquitetura e até possivelmente executando programas feitos para MIPS com pequenas modificações.

Os cinco estágios do pipeline: *Fetch, Decode, Execute, Memory Access* e *Write Back*. Componentes como a Unidade Lógica e Aritmética (ULA), a Unidade de Controle, os registradores e as memórias foram integrados para criar um modelo funcional e realista, sem muitas abstrações computacionais que simplificam o trabalho feito em um processador real.

O pipeline demonstrou eficiência ao implementar o pseudoparalelismo e aumentar a efiência de execução. A busca e decodificação funcionaram de forma precisa, preparando os dados necessários para operações subsequentes. A ULA foi bem-sucedida na execução de cálculos e controle de fluxo, enquanto o acesso e a escrita na memória garantiram a consistência dos dados e a funcionalidade de operações como load, store e print.

No geral, o simulador capturou a essência da arquitetura MIPS, conseguindo executar um *subset* de seu *assembly* satisfatório, mostrando desempenho sólido em cenários variados e oferecendo um ambiente eficaz para explorar conceitos fundamentais de sistemas baseados em pipeline.

B. Arquitetura Multicore e Preempção básica

A implementação do processamento paralelo e preempção trouxe melhorias significativas em sua capacidade de simular comportamentos realistas de sistemas operacionais modernos. Uma vez que praticamente todo sistema operacional moderno, com exceção de alguns embarcados, suporta arquiteturas *multicore* e praticamente toda arquitetura moderna é *multicore*.

A utilização de *pthreads* para modelar a execução paralela permitiu a criação de um ambiente onde múltiplos processos podem ser gerenciados simultaneamente, respeitando conceitos de exclusão mútua. Esse avanço trouxe maior flexibilidade ao simulador, que agora consegue lidar com cargas de trabalho mais complexas.

A lógica de preempção foi integrada com sucesso, permitindo que processos em execução sejam interrompidos e substituídos conforme definido pelo término do quantum ou pela necessidade do sistema operacional. A troca de contexto entre processos foi realizada por meio de uma lista de PCBs (Process Control Blocks), garantindo a persistência das informações críticas de cada processo e a retomada correta de sua execução.

Com o gerenciamento eficiente de recursos de *I/O* através de requisições ao sistema e a troca de processos bloqueados enquanto esperam por recursos, o simulador ganha desempenho em cargas de trabalho com muita entrada e saída.

A adoção de estratégias como exclusão mútua provou-se eficaz para gerenciar regiões críticas, resultando em um desempenho robusto.

Esses resultados demonstram que o simulador agora oferece uma base sólida para explorar e experimentar com arquiteturas *multicore* e sistemas preemptivos, servindo como uma ferramenta poderosa para o estudo e análise de sistemas operacionais modernos.

C. Implementação do Escalonador de Processos

O *timestamp* (medido como número de *clocks* da *cpu*) dos processos serviu como principal métrica para os testes, medidas e comparações realizadas sobre as diferentes políticas implementadas. Considerando o significado do *timestamp*, um valor menor é preferido, por indicar que o processo gastou menos ciclos quebrando e inicializando a pipeline para executar o mesmo programa. Isso indica uma menor taxa de troca de contexto por parte daquele processo, logo, uma maior permanência em CPU; que indica uma preferência por parte do escalonador.

Observamos os escalonadores em diferentes condições, mas foi observado principalmente seu comportamento em relação à quantidade de *quantum*. A principal forma de teste foi a execução de todos os códigos de teste com duas cores.

1) Quantum baixo (Cinco instruções):: No FCFS, Randômico, os resultados foram equilibrados, sem nenhuma tendência discrepante. Em geral, como esperado, o FCFS foi mais constante nos testes do que o randômico, que às vezes apresentava resultados de fato inesperados, além de apresentar uma pequena perda de desempenho para quase todos os processos.

O escalonamento por prioridade operou como o esperado, deixando os processos com maior prioridade com o *timestamp* menor ou igual às demais formas de escalonamento.

No SJF, os processos menores foram favorecidos e os processos iguais ou parecidos foram os mais equilibrados entre todos os resultados.

2) Quantum alto (Vinte instruções):: O maior valor do quantum aumentou significativamente a desempenho do simulador, cortando quase pela metade o valor do *timestamp* em todos os casos.

Em geral, todas as estratégias operaram semelhantemente a como operavam em valores baixos de quantum.

D. Gerenciamento de Cache e Escalonamento por Similaridade

A implementação do sistema de cache demonstrou ganhos expressivos de desempenho. Foram realizados testes comparativos de políticas de substituição (FIFO e LRU) em diferentes algoritmos de escalonamento (FCFS, Prioridade, Randômico e SJF), medindo-se a média de *timestamp* entre processos idênticos executados em paralelo em ambiente *multicore* com cache compartilhada.

A Figura 1 ilustra os resultados para o escalonamento FCFS, onde a política FIFO apresentou desempenho superior. Essa vantagem pode ser atribuída à sua simplicidade operacional, mais adequada aos padrões de acesso dos programas teste - caracterizados por baixa reutilização de variáveis e operações não repetitivas. Ressalta-se, porém, que em cenários com programas de maior complexidade e padrões de acesso recorrentes, a LRU tende a oferecer melhores resultados devido à sua capacidade de adaptação. O ganho médio com uso de cache neste cenário foi de 35

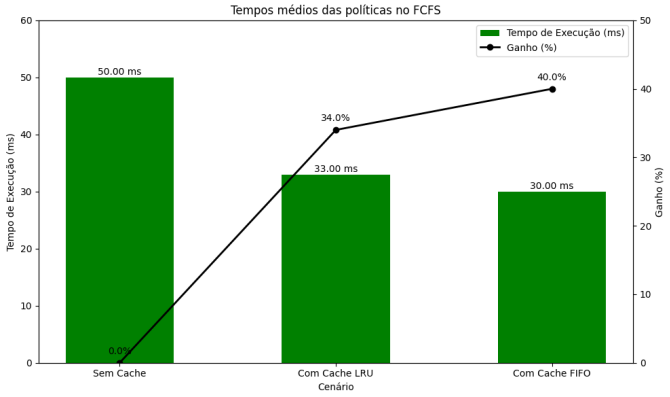

Fig. 1. Comparação de políticas de cache no escalonamento FCFS. Barras representam médias de *timestamp*.

No escalonamento por Prioridade, não foram observadas diferenças significativas entre as políticas de cache. Entretanto, hipotetiza-se que seu uso para agrupamento de processos por similaridade poderia amplificar os efeitos da cache.

Para o escalonamento Randômico (Figura 2), ambas as políticas apresentaram ganhos equivalentes (40%), sugerindo que a ausência de padrão na ordem de execução neutraliza as características distintivas dos algoritmos de substituição. Esse comportamento reforça a importância da relação entre previsibilidade do escalonador e eficácia das políticas de cache.

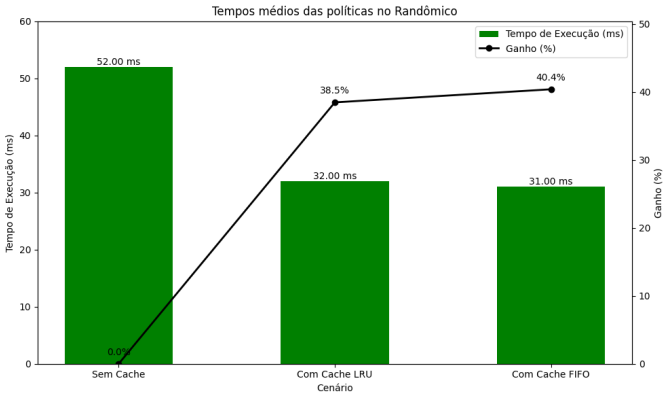

Fig. 2. Desempenho das políticas de cache em escalonamento randômico.

No SJF, os resultados assemelharam-se ao FCFS, porém com ganhos menos expressivos. Isso pode ser explicado pela natureza deste escalonamento: ao priorizar jobs curtos com menor necessidade de recursos, reduz-se o potencial de reaproveitamento do conteúdo armazenado em cache.

E. Gerência de Memória e PCB

A abordagem do simulador, por ser estática (base fixa), minimiza conflitos, mas não é totalmente coerente com sistemas *multicore* reais, onde a paginação dinâmica e escalonamento adaptativo são essenciais.

O isolamento dos processo toca numa questão de grande importância para a seguração dos sistemas e um dos maiores benefícios da virtualização da memória. Ele é um dos principais mecanismos de defesa da privacidade e segurança do usuário, pois, sem ele, qualquer processo comprometido teria acesso total à todas as informações do sistema e poderia modifica-lo o quanto quisesse.

Como o sistema operacional gerencia a tradução de endereços, programas não precisam ser recompilados ou reconfigurados para se adaptar a mudanças na alocação de RAM (ex.: adição de mais memória física ou execução em outro computador).

Desenvolvedores não precisam se preocupar com conflitos de endereços entre processos ou bibliotecas. Por exemplo, dois programas podem usar o mesmo endereço virtual 0x4000, mas a MMU mapeará cada um para regiões físicas distintas. Isso evita *bugs* de sobreposição de memória, comuns em sistemas sem virtualização.

Sem mecanismos como *swap* ou paginação, não há atrasos imprevisíveis causados por faltas de página ou operações de I/O em disco. O acesso à memória é determinístico, o que é vantajoso para simulações ou sistemas em tempo real.

A arquitetura modular da MMU permite adicionar funcionalidades futuras (e.g., proteção por segmento, compartilhamento de memória) sem refatorar completamente o sistema. Por exemplo, múltiplos registradores base/teto poderiam ser implementados para suportar mais segmentos por processo.

Mas um sistema simples de paginação e *swap* seriam necessários para destacar todos os problemas e discussões sobre a memória virtual. Tópicos como TLB(*Translation Lookaside Buffer*), *page faults* e sincronismo de paginas. Além de uma visão mais clara de como a divisão da memória em páginas afeta a alocação de memória, *cache*, escalonamento e performance de sistemas modernos.

IV. Conclusão

A implementação do simulador baseado na arquitetura MIPS e sua expansão para um ambiente *multicore* com suporte à preempção apresentou resultados satisfatórios e consistentes com os objetivos do projeto. A utilização de múltiplas *threads* permitiu uma execução eficiente e simultânea de diferentes componentes do sistema operacional, aproveitando o paralelismo disponível em arquiteturas modernas.

A transição para a arquitetura *multicore* trouxe avanços importantes, permitindo a simulação de sistemas com múltiplos núcleos, mais próximos da realidade de sistemas computacionais contemporâneos. O gerenciamento de processos, incluindo preempção e controle de concorrência, foi eficaz, embora tenha demandado uma atenção especial à sincronização de threads e à alocação de recursos de maneira eficiente. Além disso, a implementação de gerenciamento de recursos de entrada e saída, embora desafiadora, resultou em melhorias significativas de desempenho em cenários com alta demanda por I/O.

Os resultados obtidos ao testar as diferentes políticas de escalonamento mostraram como a escolha da estratégia pode impactar a eficiência do sistema, com destaque para a política de SJF em cenários de baixo quantum. A análise dos testes revelou que, ao otimizar o quantum, o desempenho do simulador foi amplamente melhorado, evidenciando a importância de ajustes finos para maximizar a eficiência do escalonamento.

A integração de um sistema de cache no simulador demonstrou ser um fator determinante para a melhoria do desempenho em ambientes *multicore* e *singlecore*, evidenciado a importância da hierarquias de memórias no sistema e como o uso adequado delas através de políticas pode aumentar significativamente o desempenho geral.

A virtualização de memória implementada é eficaz para fins didáticos, ilustrando conceitos básicos de virtualização. Porém, abstrai desafios críticos como fragmentação, paginação dinâmica e sincronização em *multicore*. Uma evolução natural seria integrar paginação (ex.: tabelas hierárquicas) e mecanismos de *swap*, permitindo explorar *trade-offs* entre desempenho, complexidade e funcionalidade, alinhando-se melhor a sistemas operacionais modernos.

Em síntese, o simulador não só valida conceitos fundamentais de arquiteturas baseadas em pipeline e *multicore*, mas também evidencia como a integração de sistemas de cache e a escolha de políticas de escalonamento podem impactar significativamente o desempenho. Embora os resultados obtidos sejam encorajadores, existem oportunidades de aprimoramento, como a introdução de políticas de escalonamento mais sofisticadas, estratégias modernas de gerenciamento de memória e recursos, e a otimização do uso de cache. Tais avanços podem ampliar ainda mais as capacidades do simulador, consolidando-o como uma ferramenta completa para a análise e o estudo dos desafios presentes em sistemas operacionais modernos.

References

[1] link do repositório: https://github.com/Getulio-Mendes/So-cache

[2] Tanenbaum, A. S., Bos, H. (2016). *Sistemas operacionais: projeto e implementação* (4ª ed.). Pearson.

[3] Silberschatz, A., Galvin, P. B., Gagne, G. (2013). *Fundamentos de sistemas operacionais* (9ª ed.). LTC.