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. A diferença é como o *thread* do *scheduller* modifica a lista para atingir os resultados desejados e permitindo a fácil implementação de outras estratégias de escalonamento com poucas modificações.

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*.
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.
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.

III. Resultados

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.

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.

Este simulador oferece uma plataforma robusta para o estudo e a compreensão dos princípios de sistemas computacionais, com ênfase em arquiteturas baseadas em pipeline e *multicore*. No entanto, ainda existem oportunidades de aprimoramento, como a introdução de políticas de escalonamento mais sofisticadas, uso de estratégias modernas de gerenciamento de memória e recursos e o uso de cache. Esses avanços podem expandir ainda mais as capacidades do simulador, tornando-o uma ferramenta mais completa para a análise de sistemas operacionais modernos e o estudo de suas características e desafios.

References

[1] link do repositório: https://github.com/frankleitelemoscosta/Multcore-SO-work.git (Escalonador em sua branch separada)

[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.