

SO Minix

1º Felipe Parreiras Dias

Departamento de Informática Gestão e Design

CEFET-MG

Divinópolis, MG

felipeparreiras86@gmail.com

2º Rafael Guimarães Gontijo

Departamento de Informática Gestão e Design

CEFET-MG

Divinópolis, MG

rafaelg000@gmail.com

Resumo—Os computadores atuais consistem em certos componentes, como processadores, memória principal e periféricos, tornando-os sistemas complexos. Para gerenciar e abstrair estes componentes, o sistema operacional fornece programas de usuários, deixando o modelo de computador mais simples. Por isso, este trabalho tem por objetivo implementar uma máquina virtual baseada na arquitetura MIPS com *pipeline* de cinco estágios, com o objetivo de explorar aspectos fundamentais de sistemas operacionais, como escalonamento de processos, concorrência de dados e gerenciamento de memória. A máquina virtual Minix demonstrou eficiência ao executar programas exemplo, validando a funcionalidade do *pipeline* e a interação dos processos no sistema.

Palavras-chave—máquina virtual, sistemas operacionais, MIPS, escalonamento

I. INTRODUÇÃO

Um computador é formado por diversas partes de *hardware* e, por isso, é um sistema muito complexo para realizar o desenvolvimento e gerenciamento destas partes diretamente. Por esta razão, os computadores possuem um *software* chamado sistema operacional (SO), que tem como função gerenciar os recursos do computador e abstrair conceitos complicados relacionados ao *hardware*, com o objetivo de oferecer um ambiente melhor para os desenvolvedores.

Dentro do SO, são realizados trabalhos relacionados ao escalonamento de processos, condições de corrida, gerência de memória, sistemas de arquivos, dentre outros. Para entender melhor o funcionamento de um SO, este trabalho tem por objetivo a construção de uma máquina virtual, denominada Minix, que apresenta estes e outros aspectos relacionados a um SO. Com isso, espera-se adquirir uma melhor compreensão do funcionamento de um SO.

Para implementar a máquina virtual, utilizou-se a arquitetura MIPS com um *pipeline* de cinco estágios: *Instruction Fetch* - IF; *Instruction Decode* - ID; *Execute* - EX; *Memory Access* - MEM; *Write Back* - WB. O MIPS é uma arquitetura ideal porque é uma arquitetura real enviada em milhões de produtos anualmente, mas é simplificada e fácil de aprender [2].

O desenvolvimento desta máquina virtual foi dividido em diversas etapas, onde, a cada passo, se implementa uma função importante de um SO. Assim, é possível focar individualmente nas tarefas executadas por um sistema operacional, garantindo um melhor entendimento de um SO como um todo. Deste modo, o artigo expõe as etapas do desenvolvimento e discute

os principais conceitos envolvidos, buscando conectar a teoria à prática.

II. METODOLOGIA

A. Arquitetura de Von Neumann e Pipeline MIPS

John von Neumann percebeu que um programa podia ser representado em forma digital na memória do computador, junto com os dados. Com isso, ele concebeu o projeto básico da máquina de von Neumann (figura 1), que foi utilizada no EDSAC, o primeiro computador de programa armazenado. Mais de meio século depois, esta máquina ainda é a base de quase todos os computadores digitais.

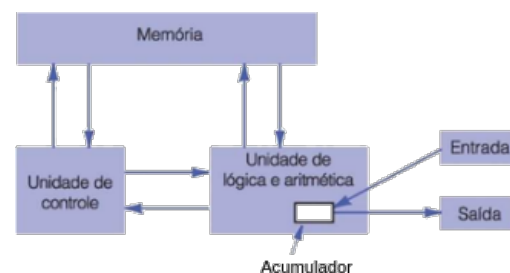


Figura 1. Máquina original de von Neumann

A máquina de von Neumann tinha cinco partes básicas: a memória, a unidade de lógica e aritmética, a unidade de controle e o equipamento de entrada e saída. Juntas, a unidade de lógica e aritmética e a unidade de controle formavam o “cérebro” do computador. Em computadores modernos, elas são combinadas em um único chip, denominado CPU (Central Processing Unit) [6].

Cada CPU pertence a uma determinada arquitetura, que define um conjunto de instruções que o computador pode executar. Cada instrução especifica a operação a ser realizada, assim como os operandos necessários para realizar a instrução [2]. Para programar as instruções de uma CPU, utiliza-se a linguagem *assembly*, que é uma representação de linguagem nativa do computador legível para humanos. Foi utilizada, na máquina virtual proposta, a arquitetura MIPS, que é simples o suficiente para entender e construir.

O conjunto de instrução MIPS é, de modo geral, executado rapidamente. O número de instruções é pequeno, para que o *hardware* necessário para decodificar as instruções seja simples e rápido. Algumas operações mais complexas são

realizadas utilizando múltiplas operações simples. Assim, o MIPS é uma arquitetura *reduced instruction set computer* (RISC).

As instruções precisam acessar os operandos rapidamente para que possam ser executadas no menor tempo possível. Porém, operandos armazenados na memória primária gastam muito tempo para serem acessados. Portanto, a maioria das arquiteturas especifica um pequeno número de registradores que armazenam dados operandos comumente usados. A arquitetura MIPS usa 32 registros, chamados de conjunto de registros ou *register file*.

Para aumentar o espaço de armazenamento de operandos, é possível também armazenar dados em memória. Quando comparado com o arquivo de registro, a memória tem muitos locais de dados, mas acessá-los leva mais tempo. Dado que o *register file* é pequeno e rápido, ao passo que a memória é grande e lenta, as variáveis mais utilizadas durante a execução do programa são mantidas em registros. Usando a combinação de registradores e memória, um programa pode acessar uma grande quantidade de dados e com relativa rapidez.

O MIPS possui um *pipeline* de cinco estágios (figura 2) que permite executar várias instruções simultaneamente, melhorando significativamente o rendimento do processador. O *pipelining* adiciona lógica para lidar com as instruções em execução simultaneamente em seus estágios. A complexidade da lógica e os registros adicionados valem a pena, visto que todos os processadores comerciais de alto desempenho usam *pipeline* hoje.

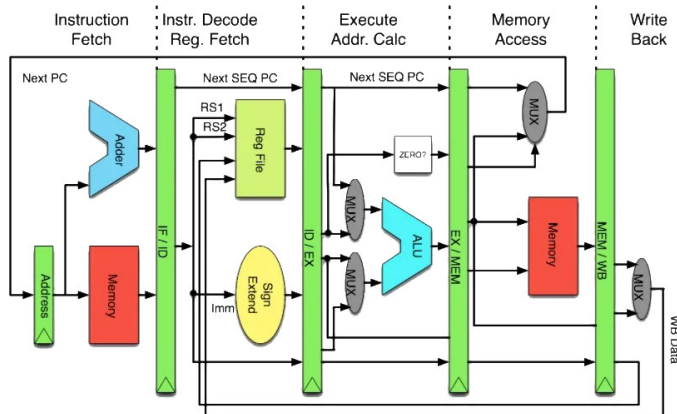


Figura 2. MIPS datapath

O *pipelining* é uma maneira poderosa de melhorar o rendimento de um sistema digital. Com esta técnica, cinco instruções podem ser executadas simultaneamente, uma em cada estágio, pois cada estágio tem apenas um quinto de toda a lógica, e a frequência do clock é quase cinco vezes mais rápida. Portanto, a latência de cada instrução é idealmente inalterada, mas o rendimento é idealmente cinco vezes melhor. Abaixo são apresentados os cinco estágios do *pipeline* da máquina virtual, adaptada da arquitetura MIPS.

- Instruction Fetch (IF): busca a instrução na memória a partir do Program Counter (PC);

- Instruction Decode (ID): decodifica as instruções para branches, jumps, operações na ULA, etc.;
- Execute (EX): executa as instruções decodificadas;
- Memory Access (MEM): busca ou escreve dados na memória principal;
- Write Back (WB): verifica se o programa terminou.

Vale destacar que, a fim de simplificar a parte de arquitetura, que não é o foco do trabalho, mas sim o SO, foi implementada uma versão mais simples da arquitetura MIPS. O conjunto de instruções é bem pequeno, de modo a realizar apenas as instruções mais básicas para verificar o funcionamento dos processos dentro do SO.

B. Multiprocessamento

Até um certo momento, os fabricantes de processadores ampliavam o poder de processamento apenas aumentando o número de transistores e a frequência dos processadores. Porém, tal solução tornava a tarefa de gerenciar a energia e o resfriamento dos processadores cada vez mais difícil [4].

Com isso, surgiu a ideia de diminuir a frequência dos processadores e aumentar o número de cores em cada processador, gerando ganhos de processamento sem aumentar significativamente a geração de calor e o consumo de energia.

Existem duas maneiras de realizar multiprocessamento (figura 3): o simétrico (SMP) e o assimétrico (ASMP). No SMP todos os processadores do sistema são iguais e têm acesso ao mesmo espaço de memória compartilhada. Já no ASMP existe um processador principal que controla o sistema e distribui tarefas para outros processadores secundários.

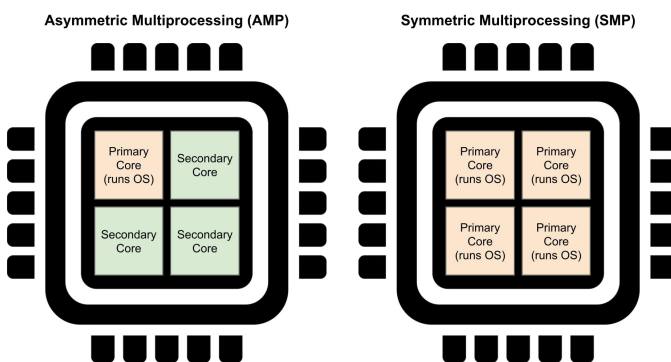


Figura 3. Arquitetura multicore.

Na máquina virtual proposta, utilizou-se o SMP, de modo que existe uma classe denominada *Core* que define o funcionamento de um processador. Assim, o número de instâncias dessa classe define o número de processadores na máquina virtual.

Para utilizar os processadores de maneira simultânea, utilizou-se a biblioteca *threads* do C++. Com isso, é possível modificar o número de núcleos no código do programa, possibilitando uma melhor observação de como o SO se comporta com um ou múltiplos núcleos.

C. Escalonamento de processos

Na multiprogramação, vários processos podem ser executados simultaneamente em um mesmo computador. Se houver mais processos a serem executados que CPUs disponíveis, o SO deverá realizar uma escolha para determinar qual processo deve ser executado em seguida [5].

A parte do SO que faz esta escolha é chamada de escalonador, e o algoritmo que este utiliza é denominado algoritmo de escalonamento. Estes podem ser divididos em duas categorias em relação a como lidar com interrupções de relógio: não preemptivo e preemptivo.

Nos algoritmos de escalonamento não preemptivos o escalonador escolhe um processo a ser executado e o deixa executando até que ele seja bloqueado, esperando pelo resultado de um outro processo ou E/S, ou acabe [5].

Em um algoritmo preemptivo o processo escolhe um processo e o deixa ser executado por no máximo um certo tempo previamente fixado (*quantum*). Se este processo ainda estiver executando quando o *quantum* acabar, ele é suspenso e o escalonador determina um novo processo para ser executado.

Quando um processo é interrompido pelo escalonador ocorre a troca de contexto, escrevendo em memória as informações do processo, como o estado dos registradores, contador de programa e estado do *pipeline*. Estes dados são armazenados em uma estrutura denominada bloco de controle de processo (PCB).

Assim, à medida que os processos entram e saem da CPU, eles vão mudando de estado. Um processo pode estar em três estados (figura 4): em execução, bloqueado ou pronto. O processo está em estado de “execução” quando está utilizando a CPU, no estado “bloqueado” quando espera entrada/saída, ou no estado “pronto” para ser executado, esperando o escalonador selecioná-lo para execução.

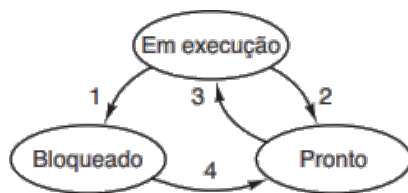


Figura 4. Estados de um processo.

Conforme ilustrado na figura 4, quatro transições são possíveis entre esses três estados:

- 1) O processo é bloqueado aguardando uma entrada
- 2) O escalonador seleciona outro processo
- 3) O escalonador seleciona esse processo
- 4) A entrada torna-se disponível

Assim como os estados de um processo refletem sua posição no ciclo de execução da máquina virtual, as métricas de tempo estão diretamente associadas às transições entre esses estados. Algumas das métricas comumente utilizadas para mensurar o desempenho do escalonador de processos são:

- **Vazão (Throughput):** número de tarefas por unidade de tempo que o sistema completa;

- **Tempo de resposta:** tempo entre a chegada do processo e o momento em que ele começa a ser executado pela CPU pela primeira vez;
- **Tempo de espera:** tempo total que um processo fica na fila de prontos esperando para ser executado.

A vazão é um dos objetivos dos sistemas em lote, onde deseja-se maximizar o número de tarefas por hora. Já em sistemas interativos, é mais importante minimizar o tempo de resposta, isto é, o tempo entre emitir um comando e receber o resultado, garantindo uma máquina mais responsiva às solicitações do usuário.

Com estas métricas, é possível analisar e comparar diferentes políticas de escalonamento na máquina virtual proposta. Para tanto, foram implementadas as seguintes políticas de escalonamento: *First-Come First-Served (FCFS)*, *Round Robin* e por prioridades.

O *First-Come First-Served (FCFS)* é um escalonador não-preemptivo, onde os processos são organizados em uma fila simples com base no seu tempo de chegada. Assim, processo que chegou primeiro é escolhido para execução e permanece em execução até sua conclusão. Não há uso de *quantum* nesse algoritmo, resultando em tempos de resposta potencialmente altos para processos que chegam após processos longos.

O *Round Robin* é um escalonador preemptivo que utiliza o *quantum* para determinar quanto tempo cada processo pode executar antes de ser suspenso. Cada processo é atendido de acordo com seu tempo de chegada, mas é interrompido quando o *quantum* expira, garantindo uma alocação mais justa de CPU. O escalonador então seleciona o próximo processo na fila e reinicia o *quantum*. Esse método é eficaz para sistemas que precisam de equidade no tempo de CPU, mas em sistemas de lote, o *quantum* pode ser configurado para ser mais longo, minimizando o *overhead* de troca de contexto.

O escalonamento circular pressupõe que todos os processos são de igual importância. Porém, algumas vezes existe a necessidade de terminar uma tarefa antes de outra, dando mais uso de CPU para este processo mais prioritário. Para tanto, utiliza-se escalonamento por prioridades. A ideia básica é direta: a cada processo é designada uma prioridade, e o processo executável com a prioridade mais alta é autorizado a executar. Porém, nesta política, pode ocorrer de processos de maior prioridade não darem espaço para os de menor prioridade ganharem tempo de CPU, gerando assim *starvation* de processos.

D. Gerenciamento de Cache e Escalonamento Baseado em Similaridade

A memória principal, preferivelmente, não deve ser lenta a ponto de atrasar o trabalho de uma CPU. Porém, com a tecnologia disponível hoje, esta exigência não é satisfeita, gerando assim a necessidade da implementação de uma hierarquia de camadas (figura 5). As camadas superiores apresentam maior velocidade, menor capacidade e custo mais elevado [5].

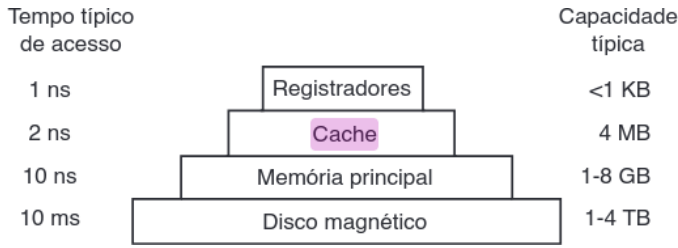


Figura 5. hierarquia de memória típica.

A memória cache, em segunda posição na hierarquia, é utilizada dentro ou próximas à CPU. Assim, quando um programa requisita a leitura de um dado na memória, a cache verifica se este dado está disponível nela, reduzindo o tempo de leitura. Entretanto, como dito anteriormente, ela possui um custo mais elevado que a memória principal e, por isso, possui tamanho reduzido.

Foi implementada a política de cache *Least Recently Used* na máquina virtual proposta. O objetivo aqui é observar quais instruções passíveis de serem guardadas em cache foram realizadas por último. Com isso, retiram-se do cache os dados que não foram utilizados recentemente, dando lugar a novos dados, que possivelmente serão utilizados novamente.

Dado o ganho de desempenho ao se guardar as últimas instruções utilizadas em cache para reaproveitamento, torna-se possível o uso do escalonamento por similaridade. Nesta política de escalonamento, os *jobs* que apresentam instruções semelhantes são organizados dentro do escalonador de modo que os cores da CPU possam aproveitar os mesmos resultados destas operações guardadas na cache.

A fim de classificar os *jobs* por similaridade, foi utilizado o *Locality Sensitive Hashing* (LSH). Este algoritmo tem por objetivo encontrar pares semelhantes em um grande conjunto de dados. Para um conjunto de dados de tamanho N , o método de força bruta de comparação de cada par possível levaria $N!/(2!(N-2)!) \approx N^2/2 = O(N^2)$ de custo de tempo. O método LSH visa reduzir isso para o tempo $O(N)$.

Para vetorizar os *jobs*, é necessário encontrar os *K-shingles*. Em seguida, cria-se um grande conjunto consistindo de *K-shingles* e atribui-se a cada *shingle* um índice exclusivo. Assim, tem-se uma matriz (N, D) , onde N representa o número de documentos e D um número igual ao número total de *K-shingles* exclusivos [3].

A métrica de similaridade utilizada foi o índice de Jaccard. Esta métrica, também conhecida como interseção sobre união, compara a similaridade entre dois conjuntos:

$$Jaccard(A, B) = \frac{A \cap B}{A \cup B} \quad (1)$$

Assim, a similaridade entre um conjunto $A = \{1, 2, 3\}$ e $B = \{2, 3, 4\}$ seria $2/4 = 0,5$. O índice de Jaccard vai de 0 a 1, de modo que 1 indica que os *jobs* são exatamente iguais.

Para verificar a eficácia do trabalho da memória cache em conjunto com escalonamento por similaridade, foi realizada a comparação desta política com a *Round-Robin*, também utilizando cache.

E. Gerenciamento de Memória

Segundo a lei de Parkinson, “programas tendem a expandir-se a fim de preencher a memória disponível para contê-los”. Com isso, criou-se a necessidade de executar programas que são grandes demais para se encaixar na memória. Além disso, deseja-se executar múltiplos programas simultaneamente, de modo que cada um deles se encaixe na memória.

Para tanto, pode-se utilizar a troca de processos (*swapping*). Esta técnica consiste em trazer cada processo em sua totalidade, executá-lo por um tempo e então colocá-lo de volta no disco.

Porém, não é interessante realizar transferências de memória principal para secundária e vice-versa, pois esse processo possui uma taxa de transferência muito baixa.

Para resolver este problema, Forthringham propôs, em 1961, a memória virtual [1]. Esta tem por objetivo garantir que cada programa tenha seu próprio espaço de endereçamento, dividido em blocos chamados de páginas. As páginas contêm uma série contígua de endereços, que são mapeados na memória física.

Quando um computador a memória virtual é utilizada, os endereços virtuais vão para o *Memory Management Unit* (MMU) [5], que mapeia os endereços virtuais em endereços de memória física, conforme ilustrado na figura 6.

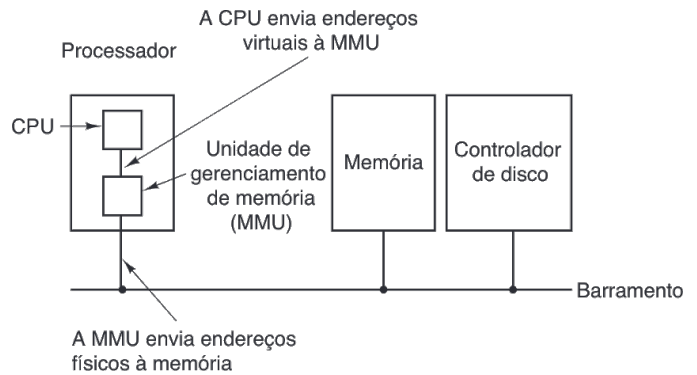


Figura 6. Memory Management Unit.

O espaço de endereçamento virtual possui unidades de tamanho fixo chamadas de páginas, enquanto no endereço de memória física as unidades correspondentes a estas páginas são chamadas de quadros de página. As páginas e os quadros de página geralmente possuem o mesmo tamanho.

Foi definido um tamanho de página de 4 KB, que guarda cada PCB. Para simplificar a operação da MMU, considerou-se endereços virtuais como números binários, onde a MMU os mapeia para números decimais. Estes números decimais são utilizados para acessar os PCBs, ou seja, eles representam os endereços físicos.

Os PCBs guardam com endereços base e limite do processo. O endereço base define onde o endereço físico do programa começa na memória, enquanto o endereço limite estabelece o comprimento do programa.

Considerou-se que os endereços base possuem o mesmo tamanho do quadro de página. Com isso, é possível guardar os

endereços virtuais no escalonador, mapeá-los para endereços físicos, para então encontrar os PCBs referenciados pelo escalonador.

Assim, é possível executar programas maiores do que a memória principal consegue armazenar. Quando houver mais páginas virtuais do que físicas, um bit presente/ausente controla quais páginas estão fisicamente presentes na memória.

É importante destacar que, numa implementação mais robusta, quando a MMU observa que a página não está mapeada, ela faz a CPU desviar para o sistema operacional. Essa interrupção é chamada de falta de página (*page fault*).

O sistema operacional escolhe um quadro de página pouco utilizado e grava seus dados na memória secundária. Ele então carrega a página recém-referenciada no quadro de página recém-liberado, muda o mapa e reinicia a instrução que causou a interrupção.

III. RESULTADOS

A. Escalonamento de processos

Foram testadas as três políticas de escalonamento descritas acima em um processador de dois *cores*. Em cada escalonador, foram iniciados três processos com um *loop* de tamanho 100. Com as métricas de tempo destes três processos, foi possível observar algumas das características de cada política descritas por Tanenbaum [5]. A primeira a ser testada foi a FCFS (figura 7), que obteve vazão de 92,91 proc/s.

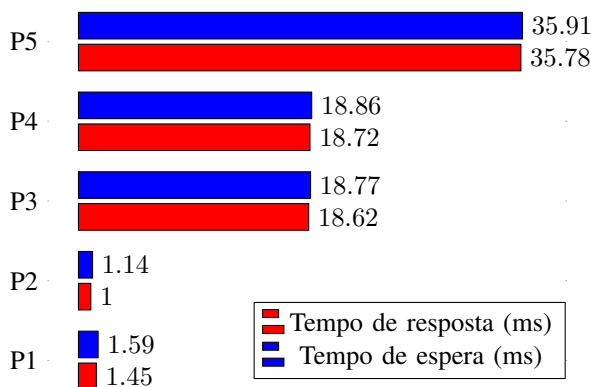


Figura 7. Métricas do escalonador FCFS.

Como é possível observar na figura 7, os processos têm tempos de resposta e espera similares. Isso ocorre porque o tempo de resposta (intervalo de tempo em que o processo chega ao escalonador e é executado) é igual ao tempo de espera (tempo que o processo fica no escalonador esperando para ser executado). Ou seja, o processo espera apenas uma vez para entrar em uma CPU e ser executado.

Como visto na figura 7, os dois primeiros processos são executados primeiro na CPU de dois *cores*, pois chegaram primeiro. Logo, houve o chaveamento e os dois processos seguintes foram executados. Por último, o processo de PID 5 foi executado, tendo que esperar todos os processos anteriores terminarem.

A próxima política de escalonamento é o *Round-Robin* (figura 8), que apresentou 89.5481 proc/s. Esta vazão é um pouco menor que a do FCFS (uma perda de 3,61% de desempenho), devido ao chaveamento de processos. Isso está de acordo com o que foi descrito por Tanenbaum, que diz que o chaveamento leva 1 ms, incluindo o chaveamento dos mapas de memória, carregar e descarregar o cache etc. Esta política é utilizada em sistemas interativos, ou seja, espera-se tempos de resposta pequenos.

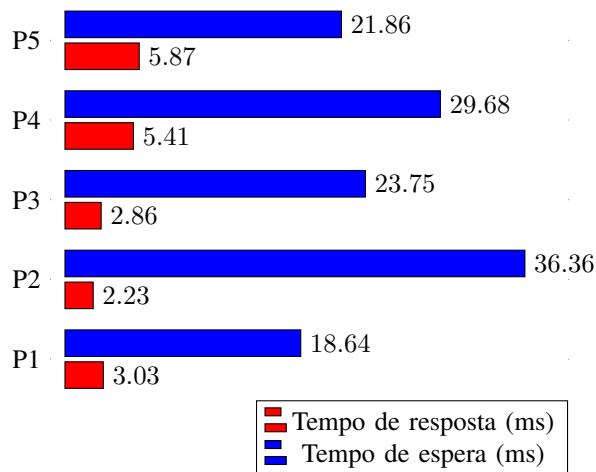


Figura 8. Métricas do escalonador Round-Robin.

Os tempos de resposta foram pequenos (figura 8). Isso ocorreu devido ao caráter preemptivo desta política, que executou os dois primeiros processos, para depois chaveá-los para os próximos processos, e assim por diante. Entretanto, é possível observar que todos os processos tiveram um tempo de espera considerável.

Por fim, o escalonamento por prioridades (figura 10), com vazão de 91.0331 proc/s, foi realizado. A prioridade dos processos foi selecionada aleatoriamente de 0 a 2 e, com isso, os processos P1 e P5 possuem maior prioridade em relação aos demais, com prioridades de valor 2 e 1, respectivamente.

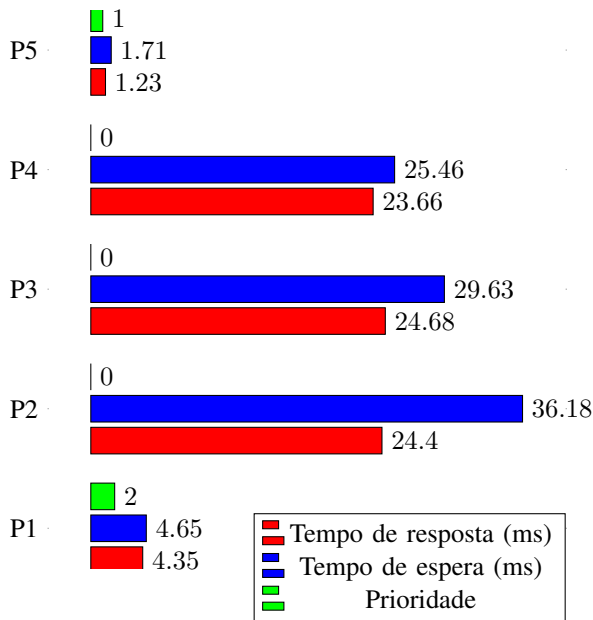


Figura 9. Métricas do escalonador por prioridades.

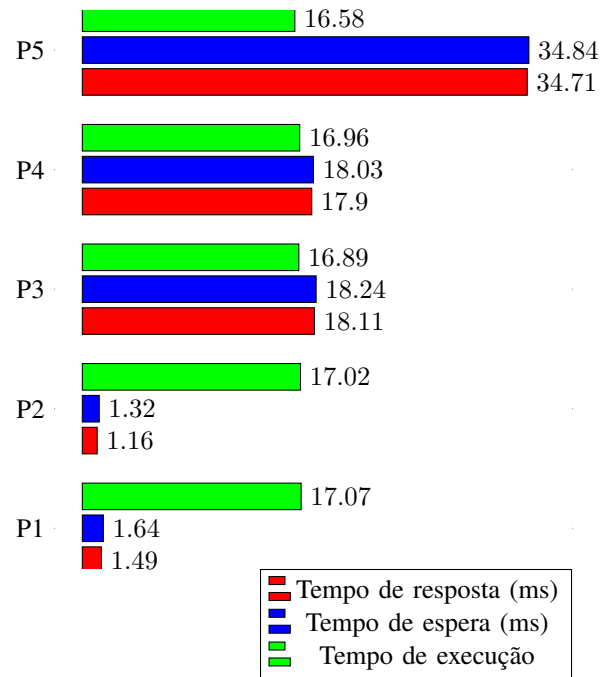


Figura 10. Métricas do escalonador por similaridade.

Observa-se (figura 10) que os processos de maior prioridade tiveram tempos de resposta e espera pequenos. Os dois *cores* da CPU executaram estes dois processos primeiro e, após terminá-los, deram uma fração de tempo justa para os três processos restantes, como é possível observar pelos seus tempos de espera.

Caso os processos de maior prioridade exigissem maior tempo de execução, ocorreria o *starvation* mencionado anteriormente. Para resolver este problema, poderia ser implementado um chaveamento que diminuísse a prioridade dos processos que não deixam os outros processos de menor prioridade serem executados.

B. Gerenciamento de Cache e Escalonamento baseado em prioridade

Foram utilizados, para realizar a comparação, cinco *jobs*, onde o primeiro, o terceiro e o quinto *jobs* são idênticos. O segundo *job* também é idêntico ao quarto. Desse modo, o escalonamento por similaridade tentará colocar os *jobs* similares juntos, e assim será possível observar a diferença de desempenho comparando esta política com a Round-Robin.

No escalonamento por similaridade, a ordem gerada pelo LSH foi: $1 \rightarrow 3 \rightarrow 5 \rightarrow 2 \rightarrow 4$. A vazão foi de 81.74 proc/s. Com isso, pode-se afirmar que o algoritmo LSH conseguiu organizar os *jobs* mais similares próximos um ao outro na fila do escalonador. Os resultados obtidos são ilustrados na figura 10.

É importante frisar que os *jobs* são colocados em uma fila (FIFO) pelo LSH, de modo que a grande diferença com a política de escalonamento FCFS para a por similaridade é a organização da fila feita antes da execução dos *jobs*.

A vazão do escalonamento por Round-Robin foi de 86,99 proc/s. Esta vazão é um pouco maior que a do escalonamento por similaridade. Isto parece dar a vantagem ao Round-Robin, porém, o tempo de vazão conta o processo de organização do escalonamento por similaridade. Os resultados são apresentados na figura 11.

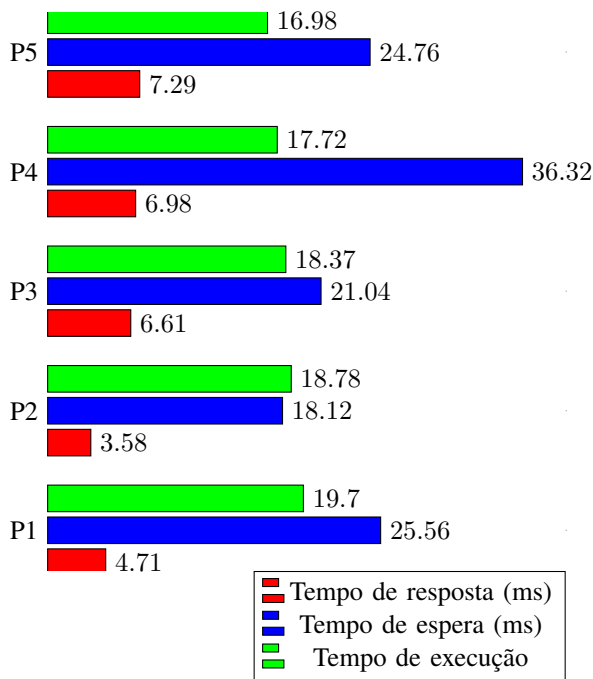


Figura 11. Métricas do escalonador Round-Robin 2.

A soma dos tempos de execução dos *jobs* no escalonamento por similaridade foi de 84,52 ms, ao passo que para o Round-Robin foi 91,55 ms. A porcentagem de ganho de desempenho foi de 7,68% que, embora pequena, é notável para grandes lotes de *jobs*. Como não foi utilizado nenhum tipo de *sleep* no acesso à memória principal, é possível que o acesso à memória cache não seja significativamente mais rápido em relação à principal.

É importante destacar que a vazão menor do escalonamento por similaridade é maior devido à organização dos *jobs* na fila. Esse tempo tende a se tornar irrelevante à medida que uma grande quantidade de *jobs* é colocada no *batch*, já que o tempo de organização será compensado pelo uso da memória cache nos *jobs* similares, visto o menor tempo de execução dos *jobs* no escalonamento por similaridade.

IV. CONCLUSÕES

A arquitetura MIPS com multiprocessadores funcionou corretamente, executando duas tarefas simultaneamente, chaveando os processos quando a política adotada apresentava preempção. Assim, foi observada a importância das diferentes políticas para diversos requerimentos.

Os resultados obtidos na comparação entre as diferentes políticas de escalonamento ficaram dentro do esperado, com o FCFS obtendo maior vazão e o *Round-Robin* apresentando menor tempo de resposta. O escalonamento por prioridades privilegiou os processos de maior prioridade, executando-os antes dos demais, conforme previsto.

A implementação da memória cache em conjunto com a política de escalonamento por similaridade apresentou uma melhoria notável para o desempenho da execução dos *jobs*. Porém, este efeito pode ser melhor observado em *batches* com

muitos *jobs*, visto que o *overhead* causado pela organização dos *jobs* feita pelo LSH pode não ser compensado pela diminuição do tempo de execução dos *jobs*.

Com isso, é possível concluir que a máquina virtual Minix cumpriu os seus requisitos, de forma a implementar um mini conjunto de instruções baseado na arquitetura MIPS. Assim, foi possível realizar o escalonamento na máquina multicore, de modo a fazer o chaveamento de processos baseado no tempo de execução na CPU.

REFERÊNCIAS

- [1] FOTHERINGHAM, John. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. Communications of the ACM, v. 4, n. 10, p. 435-436, 1961.
- [2] HARRIS, Sarah; HARRIS, David. Digital design and computer architecture. Morgan Kaufmann, 2015.
- [3] KERNES, Jonathan. Locality Sensitive Hashing: How to Find Similar Items in a Large Set, with Precision. Medium, 2021. Disponível em: <https://medium.com/towards-data-science/locality-sensitive-hashing-how-to-find-similar-items-in-a-large-set-with-precision-d907c52b05fc>. Acesso em: 4 fev. 2025.
- [4] MA, Josué. Multicore. Instituto de Computação - Unicamp. Campinas, Brasil. 2016.
- [5] TANENBAUM, Andrew S.; WOODHULL, Albert S. Sistemas Operacionais: Projetos e Implementação. Bookman Editora, 2009.
- [6] TANENBAUM, Andrew S.; AUSTIN, Todd. Organização estruturada de computadores. São Paulo: Pearson Prentice Hall, 2013.