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 princial 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 que 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.

Par 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 um arquitetura real enviada em milhões de produtos anualmente, mas é simplificada e fácil de aprender [1].

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, elabarou 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.

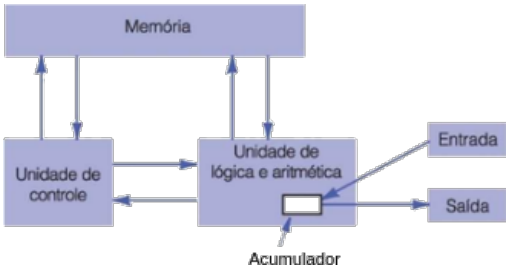

Figura 1. Máquina original de von Neumann

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) [4].

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 [1]. Para programar as instruções de uma CPU, utiliza-se a linguagem *assembly*, que é um 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, executada 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 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 um 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.

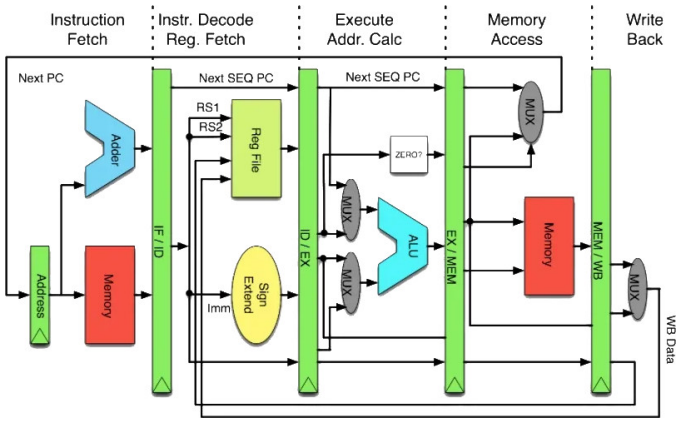

Figura 2. MIPS datapath

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 branchs, 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 [2].

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.

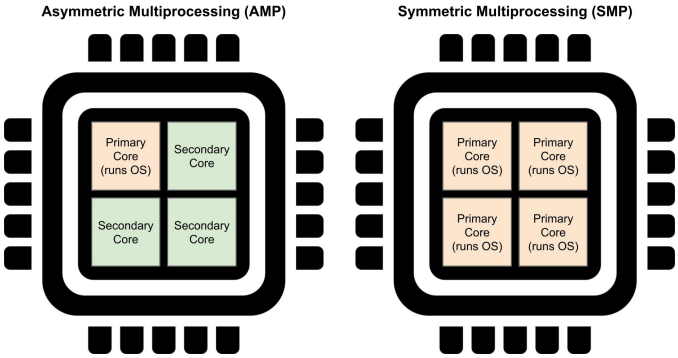

Figura 3. Arquitetura multicore.

Na máquina virtual proposta, utilizou-se o SMP, de modo de existe uma classe denominada *Core* que define o funcionamento de um processoador. 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 [3].

A parte do SO que faz esta escolha é chamada de escalonador, e o algoritmo que este utilizada 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 [3].

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 como o estado dos registradores, contador de programa e estado do *pipeline*. Estes dados são armazenadas em uma estrutura denominada bloco de controle de processo (PCB).

Assim, a 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.

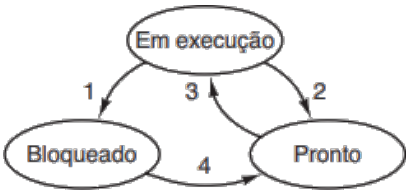

Figura 4. Estados de um processo.

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 implementados 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 uma 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.

III. Resultados

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 políticas descritas por Tanenbaum [3]. A primeira a ser testada foi a FCFS (figura 5), que obteve vazão de $92,91\ \mathrm{proc/s}$.

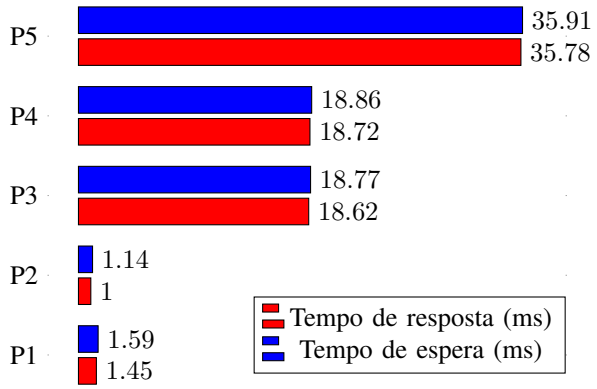

Figura 5. Métricas do escalonador FCFS.

Como é possível observar na figura 5, 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 5, 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 6), que apresentou $89.5481\ \mathrm{proc/s}$. 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 $1\ \mathrm{ms}$, 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.

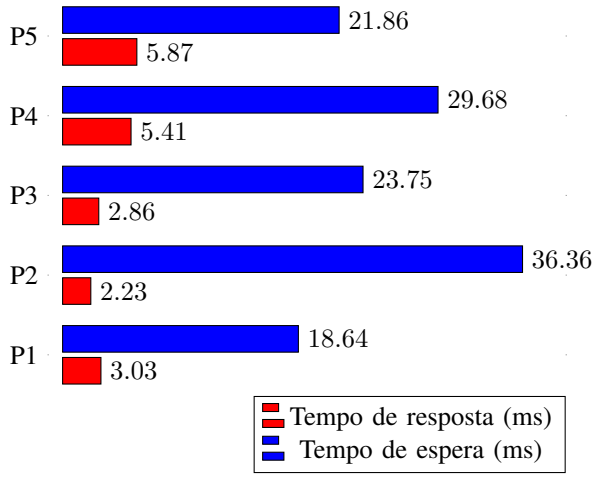

Figura 6. Métricas do escalonador Round-Robin.

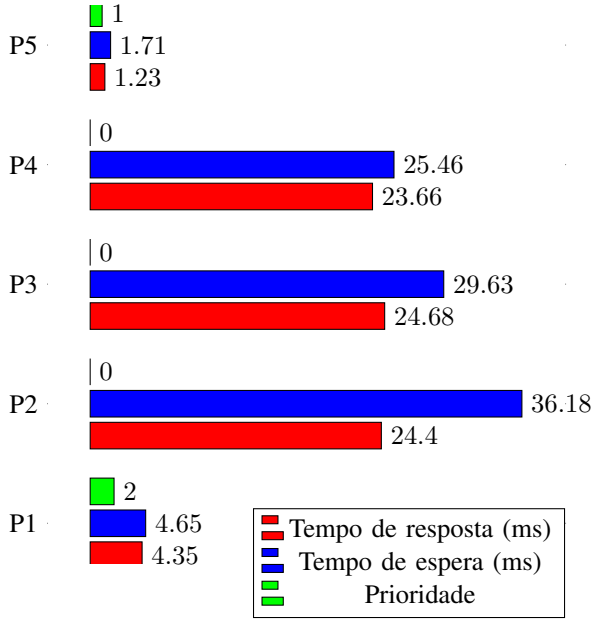

Figura 7. Métricas do escalonador por prioridades.

Os tempos de resposta foram pequenos (figura 6). Isso ocorreu devido ao caráter preemptivo desta política, que executou os dois primeiros processos, para depois chaveá-los para os próximos processos, e assim em diante. Entretanto, é possível observar que todos os processos tiveram um tempo de espera considerável.

Por fim, o escalonamento por prioridades (figura 7), com vazão de $91.0331\ \mathrm{proc/s}$, foi realizado. A prioridade dos processos foi selecionadas 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.

Observa-se (figura 7) que os processos de maior prioridade tiveram tempos de reposta e espera pequenos. Os dois *cores* da CPU executaram estes dois processos primeiro e, após terminá-los, deu 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.

IV. Conclusões

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 arquitetura MIPS com multiprocessadores funcionou corretamente, executando duas tarefas simultaneamento, cha-

veando os processos quando a política adotada apresentava preempção. Assim, foi observada a importância das diferentes políticas para diversos requerimentos.

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 baseados no tempo de execução na CPU.

REFERÊNCIAS

[1] HARRIS, Sarah; HARRIS, David. Digital design and computer architecture. Morgan Kaufmann, 2015.

[2] MA, Josué. Multicore. Instituto de Computação - Unicamp. Campinas, Brasil. 2016.

[3] TANENBAUM, Andrew S.; WOODHULL, Albert S. Sistemas Operacionais: Projetjos e Implementaçao. Bookman Editora, 2009.

[4] TANENBAUM, Andrew S.; AUSTIN, Todd. Organização estruturada de computadore. São Paulo: Pearson Prentice Hall, 2013.