Desenvolvimento e Implementação de uma Arquitetura Multicore com Escalonamento e Gerência de Memória

1st Luan Gonçalves Santos

Dept. de Engenharia de Computação
Centro Federal de Educação Tecnologica
Divinópolis, Brasil
luanlulu2010@hotmail.com

***Resumo*—Este trabalho propõe a construção de um simulador básico em C/C++ que implementa a arquitetura de Von Neumann, utilizando componentes como CPU, memória principal, cache, memória secundária e periféricos. O simulador incorpora a execução de instruções no modelo de pipeline MIPS, permitindo o processamento paralelo de múltiplas instruções. O objetivo é oferecer uma visão prática de como esses componentes interagem durante a execução de um programa e como otimizações como escalonamento de processos e gestão de memória impactam o desempenho do sistema. O simulador também permite a análise de políticas de substituição de cache e seus efeitos em acertos e falhas, além de explorar conceitos de multiprocessamento e multicomputação. Através dessa experiência, busca-se entender o impacto da hierarquia de memórias no desempenho computacional e otimizar a utilização da CPU. A implementação permitirá, ainda, o estudo das práticas de gerenciamento de recursos computacionais.**

***Palavras-chave*—Arquitetura de Von Neumann, simulador, C/C++, CPU, memória cache, pipeline MIPS, escalonamento de processos, gestão de memória, multiprocessamento, políticas de substituição de cache, desempenho computacional, acertos e falhas, otimização**

I. Introdução

A arquitetura de Von Neumann, proposta por John von Neumann na década de 1940, é a base dos computadores modernos. Esta arquitetura caracteriza-se pelo uso de uma única memória compartilhada para armazenar tanto dados quanto instruções, o que leva ao fenômeno conhecido como o bottleneck de Von Neumann, onde a CPU fica limitada pela velocidade de transferência entre a memória e o processador.

Para mitigar esse problema e otimizar o desempenho dos sistemas computacionais, é comum utilizar uma hierarquia de memórias. Nesse contexto, a memória cache — uma memória de alta velocidade, porém com capacidade limitada — atua como intermediário entre a CPU e a memória principal, armazenando temporariamente dados frequentemente acessados para reduzir o tempo de acesso.

O presente trabalho tem como objetivo a construção de um simulador básico, utilizando a linguagem C/C++, que implementa os principais componentes da arquitetura de Von Neumann. Esses componentes incluem a Unidade Central de Processamento (CPU), a memória principal, a cache, a memória secundária e os periféricos. Além disso, o simulador também irá incorporar a execução de instruções modelada com base no pipeline MIPS (Microprocessor without Interlocked Pipeline Stages), que permite a execução de múltiplas instruções de forma paralela, em diferentes ciclos de clock. Nesse caso, considera-se um ciclo de clock a movimentação de uma instrução de um estágio para outro do referido pipeline.

Por meio deste simulador, será possível observar o impacto da hierarquia de memórias, em especial o uso da cache, no desempenho geral do sistema, além de compreender como as instruções são processadas em uma arquitetura de pipeline. Ainda sob esse conceito, será possível trabalhar conceitos como escalonamento de processos, deadlocks, gestão de memória, multiprocessamento e multicomputação.

II. Objetivos

O objetivo principal deste simulador é proporcionar uma visão detalhada e prática de como os diferentes componentes de um sistema computacional, como a CPU, a memória e os periféricos, interagem durante a execução de um programa. O simulador também permitirá a implementação de otimizações como o escalonamento de processos e a gestão de memória. Dessa forma, será possível avaliar o impacto dessas otimizações no tempo total de processamento e na eficiência global do sistema.

Além disso, o simulador permitirá que o aluno explore diferentes políticas de substituição de cache, analisando como essas políticas influenciam o desempenho do sistema em termos de acertos (cache hits) e falhas (cache misses). A integração do pipeline MIPS permitirá o estudo detalhado de como as instruções são processadas de maneira paralela em diferentes estágios, otimizando a utilização da CPU.

III. Componentes do Simulador

O simulador é composto por classes que representam a **1) CPU**, a qual possui uma **2) Unidade de Controle**, a **3) Memória RAM**, um **4) Banco de Registradores**, uma **5) Tabela de Recursos** do sistema e uma **6) Memória**

Chache. Além disso, o código principal executa as instruções a partir de arquivos TXTs que são abstraídos como sendo a **7) Memória ROM da arquitetura**. Com o intuito de organizar e proporcionar uma visão simplificada da comunicação entre essas estruturas do simulador, foi-se criado um desenho das classes e structs utilizadas no sistema, observe mais na Figura 1.

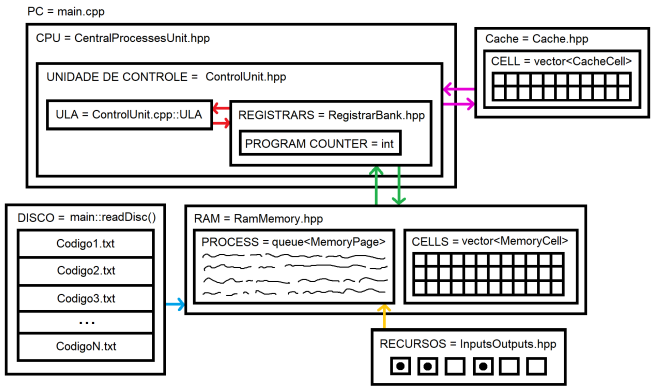

Figura 1: Desenho sobre o Simulador da Arquitetura de Von Neumann e Pipeline MIPS

1) **CPU**: A classe `CentralProcessesUnit` é responsável pelo controle do fluxo de instruções. Isso acontece a partir das seguintes funções:

   - `CentralProcessesUnit::execute()`: Implementa a lógica de um escalonador com time-sharing, utilizando time quantum para dividir o tempo da CPU entre diferentes processos. A função carrega uma página de memória do processo na RAM (swapProgram), executa instruções enquanto o contador de programa (PC) estiver dentro do tamanho do processo e não exceder o tempo máximo (TIME_QUANTUM). Durante a execução, a unidade lógica aritmética (ULA) processa as instruções, os ciclos de clock são incrementados, e, se o limite de time quantum for atingido, o processo é suspenso e outro pode ser escalonado. Quando o processo termina, ou é preemptado, a página de memória é descarregada da RAM.

   - `CentralProcessesUnit::swapProgram()`: Esta função gerencia a troca de contextos entre processos em execução e a memória principal (RAM), essencial para o funcionamento de um simulador baseado na arquitetura de Von Neumann e no pipeline MIPS. Dependendo do tipo de troca especificado pela variável typeOfSwap, a CPU executa diferentes ações para lidar com o estado dos registradores e da memória.

     a) **Tipo 0 - Processo finalizado**: Neste caso, a função é responsável por limpar todos os recursos associados a um processo que terminou sua execução. Os registradores utilizados pelo processo são marcados como limpos, indicando que estão disponíveis para novos processos. Além disso, todas as células de memória na RAM associadas ao processo (identificadas pelo idProcess) são apagadas. Esse procedimento garante que nenhum dado residual permaneça na CPU ou na memória, evitando conflitos futuros. Por fim, a CPU é liberada, ficando pronta para receber outro processo.

     b) **Tipo 1 - Time Quantum expirado**: Quando o tempo de time quantum de um processo termina, esta lógica salva o estado atual do processo para que ele possa ser retomado posteriormente. O valor do contador de programa (PC) é armazenado em uma struct do processo (MemoryPage) para registrar o ponto onde foi interrompido. Os valores dos registradores usados pelo processo são copiados para a memória RAM, preservando o estado atual do processo, por meio da struct MemoryCell, que representa uma célula de memória. Após o salvamento, os registradores são marcados como limpos e a CPU é liberada, permitindo que outro processo seja escalonado. Essa abordagem implementa o compartilhamento de recursos relacionando a memória principal (RAM), fundamental em sistemas multitarefa.

     c) **Tipo 2 - Novo processo entrando**: Quando um novo processo é escalonado para execução, esta lógica inicializa o contexto necessário na CPU. O status da CPU é atualizado para ocupado, indicando que está em uso. O contador de programa (PC) é configurado para retomar a execução do processo no ponto onde ele foi interrompido anteriormente. Em seguida, os valores da memória RAM associados ao processo (registradores e valores) são carregados nos registradores da CPU, restaurando o estado do processo para continuar sua execução. Isso assegura a continuidade sem perda de dados ou inconsistências no processamento.

   Essa lógica de troca de contexto é essencial para gerenciar múltiplos processos em ambientes multitarefa, garantindo que os dados de cada processo sejam preservados e restaurados corretamente. Em caso de tipo de troca inválido, uma exceção é lançada para evitar comportamento inesperado.

2) **Unidade de Controle**: A classe `ControlUnit` representa a Unidade de Controle de um simulador de arquitetura de computadores, sendo responsável por gerenciar a execução de instruções de um processo em nível de registradores e memória. Atributos privados, como `bankOfRegistrars`, armazenam um banco de registradores que é essencial para o controle e execução das instruções, enquanto o método

`ControlUnit::splitLineOfCodeBySpace()` é usado internamente para dividir uma linha de código em tokens, facilitando o processamento de instruções.

No lado público, o construtor da classe inicializa a unidade de controle com um banco de registradores do tamanho especificado. Seu método principal é `ControlUnit::ULA()` (Unidade Lógica e Aritmética), o qual executa as instruções do bloco de memória do processo (MemoryPage), interagindo diretamente com a memória RAM e os registradores para processar operações lógicas e aritméticas. Em conjunto, esses elementos fazem da Unidade de Controle um componente central que orquestra a execução das operações no simulador, replicando o comportamento de uma CPU real.

3) **RAM**: A classe `RamMemory` simula a memória do sistema, com um array de 32 inteiros, onde cada posição pode ser lida ou escrita por meio de funções de acesso. O estado da RAM pode ser exibido na saída padrão. A classe também implementa mecanismos para gerenciar os processos vindos da Memória ROM, nesse sentido ele são armazenados na fila de processos, onde ficam aguardando seu momento de execução. Esse gerenciamento acontece a partir de algumas funções indispensáveis:

   - `RamMemory::RamMemory()`: O construtor da classe inicializa uma instância de memória RAM com um tamanho definido pelo parâmetro size. Sempre que o sistema inicializa, é verificado se o tamanho fornecido é maior que zero; caso contrário, lança uma exceção (invalid_argument) para garantir que a memória tenha uma capacidade válida. Em seguida, o construtor dimensiona o vetor memoryCells para comportar o número especificado de células de memória. Cada célula é inicializada com valores padrões zerados Isso assegura que a RAM comece com todas as células devidamente alocadas e prontas para serem utilizadas, refletindo o estado inicial de uma memória limpa.

   - `RamMemory::write()`: A função de escrita da classe é responsável por alocar uma célula de memória (MemoryCell) na RAM, seguindo diferentes tipos de operações definidos pelo parâmetro *kindOfWrite*. No caso 0 (operação STORE), escreve no endereço especificado se ele estiver vazio ou ocupado pelo mesmo processo. No caso 1, limpa o endereço, marcando-o como disponível após a finalização de um processo. No caso 2, realiza um swap de registradores sujos para o primeiro espaço livre na memória; se não houver espaço, lança uma exceção indicando que a memória está cheia. A função também valida o endereço fornecido, garantindo que ele esteja dentro do intervalo permitido, e lança exceções em caso de erro, como endereço fora do intervalo ou tipo de escrita inválido. Essa função é essencial para gerenciar a alocação e liberação de células de memória no simulador, evitando conflitos e garantindo eficiência no uso da RAM.

4) **RegisterBank**: A classe `RegistrarsBank` simula o banco de registradores simula o banco de registradores de uma Unidade de Controle, funcionando como um conjunto de 32 registradores utilizados para os valores durante a execução de instruções de um processo.

   Nesse sentido, para simular os componentes, a classe possui um vetor privado, onde é armazenados os registradores. Estes, por sua vez, é representado por um valor númerico inteiro e um estado lógico que indica se ele está ocupado ou sendo usado. Existe também atributo *Pc* (Program Counter) cujo é responsável por armazenar o endereço da próxima instrução a ser executada, simulando o fluxo de controle.

5) **Recursos do Sistema**: A classe `InputsOutputs` abstrai os periféricos de uma arquitetura computacional, permitindo o gerenciamento do uso concorrente desses dispositivos por diferentes processos. Ela funciona como uma tabela que armazena os periféricos – por padrão, o simulador possui apenas 5 periféricos disponíveis: Mouse, Teclado, Monitor, Impressora e Fone de Ouvido – e associa a cada um deles um par contendo o identificador do processo que o está utilizando e um estado booleano que indica se o periférico está ocupado. Em resumo, InputsOutputs facilita o controle de concorrência sobre os periféricos, garantindo que apenas um processo possa usar um dispositivo por vez, evitando conflitos.

6) **Memória Cache**: A estrutura da cache desenvolvida é composta por células de memória, denominadas `CacheCell`, cada uma contendo três atributos principais: uma instrução, o resultado associado a essa instrução e um contador de uso . O contador é utilizado para implementar uma política de substituição baseada na quantidade de acessos, similar ao algoritmo **Least Frequently Used (LFU)**. Essa escolha tem como objetivo manter na cache os dados que possuem maior probabilidade de reutilização, otimizando o desempenho durante a execução de jobs semelhantes.

   A implementação do método de substituição está centralizada na função `Cache::findReplaceIndex()`, que percorre todas as células da cache e identifica aquela que foi menos reutilizada. Quando a capacidade máxima da cache é atingida, a célula com menor frequência de uso é substituída por uma nova entrada. O método `Cache::save()` realiza essa lógica de forma eficiente, permitindo a gravação de instruções e seus respectivos resultados na cache.

   O objetivo dessa estrutura é identificar instruções simila-

res e evitar redundâncias no processamento. Para isso, o método `Cache::isSimilar()` realiza uma busca na cache para verificar se o resultado de uma instrução já está armazenado. Caso positivo, o valor correspondente é imediatamente retornado, e o contador de uso da célula é incrementado. Essa abordagem elimina a necessidade de recomputação de operações previamente realizadas, resultando em uma economia significativa de ciclos na pipeline. Como consequência, o sistema otimiza o tempo de processamento e melhora o desempenho geral.

7) **Memória ROM**: No simulador, a memória ROM não é implementada como uma classe, mas sim através de arquivos de texto (`.txt`) que armazenam os comandos a serem executados. Esses arquivos seguem uma convenção de nomenclatura padronizada: `codigo1.txt`, `codigo2.txt`, `codigo3.txt`, e assim por diante, até `codigoN.txt`. Essa estrutura permite que vários códigos/processos sejam executados de forma sequencial ou conforme a necessidade, bastando ajustar a variável que define a quantidade de códigos no arquivo `main.cpp` do sistema.

   Os códigos contidos na ROM devem seguir um padrão rígido para serem processados corretamente. O **primeiro requisito** é que a **primeira linha** de cada arquivo obrigatoriamente contenha o nome de um periférico do sistema (um dispositivo de entrada ou saída), já que todos os processos precisam interagir com algum recurso físico. Isso garante que cada processo seja associado a um dispositivo específico, como ”Mouse”, ”Teclado”ou ”Impressora”.

   Além disso, a **primeira linha** deve conter um **número de prioridade**, que será utilizado para determinar a ordem de execução dos processos. O número de prioridade deve estar presente logo após o nome do periférico, separado por um ponto e vírgula — por exemplo, uma linha de entrada válida poderia ser `Mouse;3`. O número de prioridade pode variar de acordo com a necessidade do processo, e o simulador utilizará este valor para ordenar os processos conforme sua importância ou urgência. Quanto maior o número de prioridade, maior a urgência para execução do processo associado.

   As próximas linhas são reservadas para as instruçoes MIPS. Cada arquivo deve conter uma única instrução MIPS por linha, respeitando uma estrutura clara e legível. Essa abordagem garante que o simulador consiga interpretar cada comando de forma sequencial e sem ambiguidades. Não há limite para o número de instruções em cada arquivo, permitindo que processos de diferentes complexidades sejam representados.

   É importante ressaltar que a memória ROM do simulador funciona como um repositório de códigos em formato de texto, com um padrão de nomeação e estrutura que garante a correta associação dos processos a periféricos e a execução ordenada das instruções do sistema.

IV. Instruções MIPS

As instruções MIPS implementadas no simulador seguem um conjunto de formatos e operações padronizadas, projetadas para executar tarefas básicas de manipulação de dados, controle de fluxo e armazenamento. A seguir, cada instrução é apresentada com sua descrição, formato e exemplo de uso:

A. LOAD

Formato: `LOAD <valor> <dest>`
Descrição: Carrega um valor imediato no registrador `<dest>`.
Exemplo: `LOAD 10 16` ⇒ Carrega o valor `10` no registrador `16`.

B. ADD

Formato: `ADD <src1> <src2> <dest>`
Descrição: Soma os valores dos registradores `<src1>` e `<src2>` e armazena o resultado no registrador `<dest>`.
Exemplo: `ADD 18 16 17` ⇒ Soma o valor dos registradores `18` e `16` para armazenar o resultado no registrador `17`.

C. SUB

Formato: `SUB <src1> <src2> <dest>`
Descrição: Subtrai o valor no registrador `<src2>` do valor no registrador `<src1>` e armazena o resultado no registrador `<dest>`.
Exemplo: `SUB 16 17 18` ⇒ Subtrai o valor no registrador `17` do valor no registrador `16` e armazena o resultado no registrador `18`.

D. STORE

Formato: `STORE <src> <endereco>`
Descrição: Armazena o valor do registrador `<src>` na posição de memória `<endereco>`. Caso o armazenamento não seja possível, o processo entrará em estado de inanição.
Exemplo: `STORE 18 16` ⇒ Armazena o valor do registrador `18` na posição de memória `16`.

E. IF

Formato: `IF <operation> <src1> <src2> <dest>`
Descrição: Executa uma comparação entre os valores dos registradores `<src1>` e `<src2>` com base na operação `<operation>` (< ou >). O resultado da comparação (`6666` para verdadeiro, `-6666` para falso) é armazenado no registrador `<dest>`.
Exemplo: `IF > 10 20 30` ⇒ Compara se o valor no registrador `10` é maior que o valor no registrador `20`. O resultado (`6666` ou `-6666`) será armazenado no registrador `30`.

F. FOR

Formato: `FOR <operation> <times> <value> <dest>`
Descrição: Realiza uma operação repetida (`SUB` ou `ADD`) um número de vezes especificado em `<times>`, usando `<value>` como operando, e armazena o resultado acumulado no registrador `<dest>`.
Exemplo: `FOR ADD 5000 10 15` ⇒ Realiza a operação de adição, somando o valor `10` ao registrador `15` repetidamente por `5000` vezes.

V. Arquitetura Multicore com Suporte a Concorrência

A solução foi projetada para simular uma arquitetura multicore com suporte à execução concorrente de processos. Para isso, utilizou-se a biblioteca padrão de threads em C++ (`std::thread`), permitindo que cada processo fosse executado de forma independente em um núcleo virtual. Os processos são gerenciados através de múltiplas threads, armazenadas em um vetor que permite o controle das suas execuções. Esse pseudoparalelismo foi integrado ao simulador, garantindo que múltiplos processos compartilhem os recursos do sistema de maneira segura, por meio de mecanismos de exclusão mútua (`std::mutex`) e sincronização. Dessa forma, a execução simultânea de processos, prevene condições de corrida e inconsistências nos dados compartilhados.

Para garantir o uso eficiente do processador, implementou-se um sistema de preempção baseado em tempo quantum. Cada processo foi limitado a 35 ciclos de clock, com cada instrução consumida equivalendo a um clock. Essa estratégia assegura que nenhum processo monopolize a CPU, promovendo um compartilhamento justo dos recursos. Caso o tempo quantum de um processo seja esgotado antes de sua conclusão, ele é interrompido e devolvido à fila de processos por meio de um mecanismo de troca de contexto. Essa funcionalidade simula de forma realista o funcionamento de sistemas operacionais modernos, respeitando as regras de escalonamento baseadas no tempo.

Nesse sentido, o ciclo de vida dos processos foi modelado conforme os estados descritos por Tanenbaum: **pronto**, textbfbloqueado e textbfexecutando. Processos que solicitam recursos ocupados, como dispositivos de entrada e saída, entram no estado bloqueado e aguardam até que o recurso seja liberado. Isso envolve o sistema de gerenciamento de recursos implementado na classe `InputsOutputs`, descrita anteriormente. Assim, se o recurso está ocupado, os processos bloqueados retornam para o final da fila de processos e ficam aguardando um novo ciclo de execução.

Cada processo foi representado por uma struct que abstraí um bloco de controle de processos (PCB), denominado`MemoryPage`. Este *PCB* armazena informações essenciais, como o ID do processo, seu estado atual, as instruções associadas e os recursos necessários (como dispositivos de entrada e saída). O *PCB* também gerencia dados relacionados à memória e ao quantum restante de cada processo. Essa abstração facilitou a troca eficiente de contexto durante a preempção e a recuperação do estado de execução do processo.

Para evitar problemas de concorrência durante a execução dos processos, foi utilizado um mecanismo de bloqueio por meio da classe `std::mutex`. Isso acontece na função `executeProcessInThread()`, a qual protege a região crítica onde o processo é executado, utilizando o recurso `lock_guard`. Isso garantiu que múltiplas threads não acessassem simultaneamente os mesmos recursos compartilhados, como memória ou dispositivos de entrada e saída. Dessa forma, foi possível assegurar a consistência dos dados e a integridade do sistema, mesmo em um ambiente com múltiplas threads concorrentes.

Adicionalmente, a função `monitorProcesses()` foi implementada como um monitor central, responsável por gerenciar o ciclo de vida dos processos e distribuir os recursos do sistema. Esse monitor verifica periodicamente a fila de processos e despacha-os para execução em threads separadas. Ele também detecta quando um recurso está ocupado, devolvendo o processo à fila de processos e aguardando a próxima oportunidade de execução. Esse design modular permitiu que o sistema operacional simulado tomasse decisões dinâmicas e inteligentes, respeitando as regras de preempção e escalonamento.

A solução também foi otimizada para garantir um uso eficiente dos recursos computacionais. O tempo total de execução do sistema é calculado em milissegundos para avaliar o desempenho do simulador. Além disso, a inclusão de um pequeno atraso (1 milissegundo) entre as verificações na função que monitora os processos ajudou a reduzir a sobrecarga do sistema, garantindo uma execução fluida sem comprometer o pseudoparalelismo.

Conclui-se então que a implementação expande as capacidades do simulador para suportar arquiteturas multicore e preempção, proporcionando uma simulação robusta e realista de sistemas operacionais modernos. A utilização de threads para simular o paralelismo, combinada com mecanismos de exclusão mútua e gerenciamento eficiente de recursos, permitiu modelar um sistema que reflete fielmente os desafios e soluções de ambientes reais. Com o suporte ao ciclo de vida dos processos, ao gerenciamento de recursos e à troca de contexto, esta solução oferece uma base sólida para estudos e simulações avançadas de sistemas operacionais.

VI. Implementação dos Escalonadores de Processos

As políticas de escalonamento desempenham um papel essencial na gerência de processos em sistemas operacionais, sendo responsáveis por definir a ordem de execução das tarefas com base em diferentes critérios. Neste trabalho, é implementado três políticas clássicas: First Come First Served (FCFS), Shortest Job First (SJF) e Highest Priority First (HPF). A seguir, é apresentado os detalhes sobre a implementação de cada uma dessas abordagens.

A. Escalonador First Come First Served (FCFS)

First Come First Served (FCFS) é uma política de escalonamento não preemptiva que processa as tarefas na ordem de chegada. Essa abordagem é simples e justa, mas pode resultar em altos tempos de espera quando processos longos bloqueiam a execução de processos curtos [5].

O escalonador First Come First Service (FCFS) baseia-se em uma abordagem simples e direta: os processos são atendidos na ordem em que chegam à fila. Este método utiliza uma fila FIFO (First In, First Out), onde o primeiro processo a entrar é o primeiro a ser executado. A lógica do método é implementada removendo o processo localizado na frente da fila pode ser observada no Pseudocódigo 1.

Algorithm 1 Reponsável por gerir o escalonamento baseado em uma fila com ordem de chegada

```
0: procedure GETPROCESSBYFIFO
0:     if processQueue is empty then
0:         throw error "No processes in the queue"
0:     end if
0:     nextProcess ← processQueue.front()
0:     Remove nextProcess from processQueue
0:     return nextProcess
0: end procedure=0
```

"Se a fila de processos estiver vazia, uma exceção será lançada indicando que não há processos para executar. Caso contrário, o primeiro processo da fila é selecionado e removido. Este comportamento garante a ordem de chegada como critério de execução."

B. Escalonador Shortest Job First (SJF)

Shortest Job First (SJF) é uma política de escalonamento não preemptiva que seleciona o processo com o menor tempo estimado de execução. Essa abordagem reduz o tempo médio de resposta, mas pode ser ineficiente para cenários com tarefas de longa duração [5].

Por sua vez, o escalonador Shortest Job First (SJF) prioriza processos que requerem o menor tempo estimado de execução. Nesse sentido, durante a leitura do memória ROM, o simulador estima o tempo de execução de cada processo através do número de operações MIPS a serem executadas pelo mesmo. Observe a representaçao do escalonador no Pseudocódigo 2.

"Ao verificar cada processo na fila, é realizada uma comparação entre o número estimado de ciclos. Quando um processo com menor tempo é encontrado, ele é armazenado como candidato para execução. A fila é então reconstruída sem o processo selecionado. Este mecanismo assegura que o processo mais rápido seja sempre priorizado."

C. Escalonador Highest Priority Job First (HPJF)

Highest Priority First (HPF) é uma política de escalonamento que prioriza os processos com maior prioridade. Embora eficiente para cenários críticos, essa política pode levar à fome de processos de baixa prioridade [5].

Algorithm 2 Responsável por gerir o escalonamento baseado no processo com menor tempo estimado de execução

```
0: procedure GETPROCESSBYSJF()
0:     if processQueue is empty then
0:         throw error "No processes in the queue"
0:     end if
0:     minProcess ← processQueue.front()
0:     for all process in processQueue do
0:         if process.estimatedClocks less than
   minProcess.estimatedClocks then
0:             minProcess ← process
0:         end if
0:     end for
0:     Remove minProcess from processQueue
0:     return minProcess
0: end procedure=0
```

O terceiro método implementado, Highest Priority Job First (HPJF), utiliza a prioridade como critério de seleção. Os processos são avaliados em relação ao valor de prioridade associado, e aquele com a maior prioridade é escolhido para execução. A lógica empregada também utiliza uma fila temporária para avaliar e reorganizar os processos após a seleção, observe no Pseudocódigo 3.

Algorithm 3 Responsável por gerir o escalonamento baseado na prioridade dos processos

```
0: procedure GETPROCESSBYPRIORITY
0:     if processQueue is empty then
0:         throw error "No processes in the queue"
0:     end if
0:     maxProcessPriority ← processQueue.front()
0:     for all process in processQueue do
0:         if process.priority bigger than maxProcessPrio-
   rity.priority then
0:             maxProcessPriority ← process
0:         end if
0:     end for
0:     Remove maxProcessPriority from processQueue
0:     return maxProcessPriority
0: end procedure=0
```

"Inicialmente, a prioridade máxima é atribuída ao primeiro processo da fila. Em seguida, os processos subsequentes são comparados, e aquele com maior prioridade é selecionado. Assim como no método SJF, a fila é reconstruída sem o processo escolhido, permitindo que a execução prossiga com base nas prioridades estabelecidas."

D. Escalonador Greatest Similarity First (GSF)

O escalonador de processos baseado em similaridade (GFS) é uma política eficiente para a seleção do próximo processo a ser executado. Este escalonador prioriza sempre o processo mais similar ao último processo executado. Para isso, a similaridade entre processos é previamente calculada e armazenada

em uma matriz simétrica, que é continuamente atualizada ao longo da execução.

Uma vez que o Greatest Similarity First (GSF) prioriza os processos mais parecidos entre si, o cálculo da similaridade é realizado antes do processamento, durante a leitura da memória ROM. Assim sendo, a lista de processos é percorrida, calculando a similaridade percentual pela razão entre as instruções MIPS coincidentes entre os processos entre cada par de processos sobre a quantidade de linha destes.

Tendo em mãos os dados de similaridade entre os processos, esses valores são armazenados em uma matriz simétrica para depois serem utilizados pelo política. Além disso, para garantir o funcionamento do método, quando um processo é finalizado, a similaridade deste é desconsiderada. é possivel verificar a logica do escalonador por similaridade no Pseudocódigo 4.

Algorithm 4 Responsável por obter o processo mais semelhante ao último executado

```
0: procedure GETPROCESSBYSIMILARITY
0:   if processQueue is empty then
0:     throw error "No processes in the queue"
0:   end if
0:   if lastProcessRuned is the first or the last process then
0:     selectedProcess ← processQueue.front()
0:     Remove selectedProcess from processQueue
0:     return selectedProcess
0:   end if
0:   tempQueue ← processQueue
0:   processList ← Convert tempQueue to list
0:   maxSimilarity ← -1.0
0:   mostSimilarProcess ← empty
0:   mostSimilarIndex ← -1
0:   for i = 0 to similarityMatrix.size() - 1 do
0:     if i ≠ lastProcessRuned then
0:       if isProcessesInQueue then
0:         similarity ← value of similarityMatrix
0:         if similarity bigger than maxSimilarity then
0:           maxSimilarity ← similarity
0:           mostSimilarIndex ← i
0:         end if
0:       end if
0:     end if
0:   end for
0:   mostSimilarProcess ← get from processesQueue
0:   updatedQueue ← remove mostSimilarProcess
0:   return mostSimilarProcess
0: end procedure=0
```

"Inicialemte é verificado se o último processo é válido. Caso esse cenários seja falso, o primeiro processo da fila é retornado. Em seguida, a fila é convertida em uma lista temporária, facilitando a manipulação e a busca. O próximo passo é realizar a busca por similaridade, percorrendo a matriz de similaridade para identificar o processo mais semelhante ao último executado, ignorando o próprio processo. Após encontrar o processo mais semelhante, ele é selecionado com base no índice encontrado na matriz. Em seguida, o processo selecionado é removido da fila, que é atualizada, e finalmente, o processo selecionado é retornado."

VII. Gerenciamento de Cache e Escalonamento Baseado em Similaridade

A implementação de uma estrutura de memória cache no simulador de uma Arquitetura Multicore trouxe modificações significativas no comportamento do sistema. Esse aprimoramento foi motivado pela necessidade de otimizar a execução de instruções repetitivas, reduzindo a carga computacional e, consequentemente, o tempo de processamento. A seguir, detalham-se os principais aspectos dessa solução e seus impactos no desempenho do sistema.

A. Implementação da Política de Cache

Em sua completude, a estrutura da cache foi projetada para armazenar os resultados das operações lógicas, como ADD e SUB, assim que são resolvidas pela primeira vez. A escolha por armazenar apenas essas instruções deve-se ao fato de que operações de comparação (IF) e manipulação de dados (LOAD e STORE) não apresentam um custo computacional elevado a ponto de justificar seu armazenamento em cache.

A partir dessa configuração, sempre que uma nova instrução é recebida, o sistema consulta a cache para verificar se o resultado correspondente já está armazenado. Caso a instrução seja encontrada, a execução na Unidade Lógica e Aritmética (ULA) é omitida, e o resultado previamente armazenado é utilizado diretamente. Essa abordagem reduz a quantidade de ciclos napipeline necessários para processar instruções, liberando recursos do processador de forma mais rapida para outras operações.

B. Assemelhando o sistema com e sem Cache

O sistemas sem cache, todas as instruções são obrigatoriamente executadas na ULA, independentemente de já terem sido processadas anteriormente. Esse cenário leva a um consumo elevado de recursos computacionais, maior número de acessos à memória principal e um aumento significativo do tempo de processamento. Além disso, o congestionamento na pipeline é mais frequente, o que compromete o desempenho geral do sistema.

Com a introdução da cache, observa-se uma redução expressiva na latência das operações, pois o número de acessos à memória principal diminui consideravelmente. Essa redução é particularmente importante em arquiteturas multicore, onde diversos núcleos compartilham o acesso a uma hierarquia de memória. O reaproveitamento de resultados já computados minimiza conflitos de acesso e melhora a eficiência da execução paralela.

VIII. Gerência de Memória a partir da MMU

A **Memory Management Unit (MMU)** desempenha um papel fundamental no gerenciamento da memória dos computadores hoje em dia, especialmente no que diz respeito à tradução de endereços lógicos para físicos. Esta tarefa não só

facilita a execução de processos no sistema operacional, mas também garante a segurança, proteção e eficiência no uso da memória. Nesse sentido, visando mostrar tais características, foi implementado um sistema de memórias virtuais baseado no MMU no simulador presente neste trabalho. Para compreender completamente o funcionamento da MMU, é necessário entender o contexto em que ela opera, incluindo o conceito de alguns tópicos explicados a seguir.

A. Endereços Lógicos vs. Endereços Físicos

O sistema de endereçamento lógico é uma abstração usada pelos programas de usuário, em que cada processo possui sua própria visão da memória, que é chamada de *endereço lógico*. Esses endereços são gerenciados pelo sistema operacional e mapeados para endereços físicos, que são as localizações reais na memória principal (RAM).

- **Endereço Lógico:** É o endereço utilizado pelo programa para acessar a memória. Ele é gerado pela CPU durante a execução do código de um processo [5].
- **Endereço Físico:** Refere-se à localização física na RAM, onde os dados reais são armazenados [5].

O MMU é responsável por realizar a conversão entre esses dois tipos de endereços. Cada vez que um processo faz uma requisição de acesso à memória, a MMU converte o *endereço lógico* em *endereço físico*, permitindo que o processo interaja com a memória de forma eficiente e segura. No contexto do simulador, se trata apenas de uma simples conversão de um número binário em decimal.

B. Tradução de Endereços pela MMU

A tradução de endereços lógicos para endereços físicos é uma tarefa crucial para garantir que os processos possam acessar a memória sem conflitos. A MMU usa uma *tabela de páginas* ou *tabela de segmentos* (dependendo do modelo de gerenciamento de memória) para mapear endereços lógicos para endereços físicos.

- **Tabela de Páginas:** Divide a memória em páginas fixas, e cada entrada da tabela mapeia uma página virtual para uma página física [5].
- **Tabela de Segmentos:** Divide a memória em segmentos, que podem ter tamanhos variáveis, com o mapeamento entre segmentos virtuais e físicos [5].

Esse processo ocorre em tempo real, ou seja, a MMU faz a conversão durante a execução de uma instrução, garantindo que o processo tenha acesso à memória principal de maneira transparente.

C. Proteção de Memória

A **proteção de memória** é um dos principais objetivos da MMU. Visto que, ao isolar o espaço de memória de cada processo, a MMU garante que um processo não consiga acessar ou corromper a memória de outro processo. Caso contrario, em tese, poderia haver falhas de segurança ou até mesmo travamentos do sistema.

Isso é feito utilizando os *endereços lógicos*. Como os processos do usuário só têm acesso aos seus próprios endereços lógicos, eles não podem acessar diretamente os endereços físicos da memória de outros processos. O simulador agora configura a MMU para permitir ou restringir acessos à memória de acordo com as permissões definidas.

D. Gerenciamento de Memória e Alocação de Processos

A alocação eficiente de memória é essencial para o bom desempenho do sistema. O sistema operacional deve alocar os processos na memória principal de forma a minimizar a *fragmentação* e garantir que os processos não ocupem espaços de memória desnecessários.

- **Fragmentação Interna:** Ocorre quando a memória alocada para um processo é maior do que o necessário [5].
- **Fragmentação Externa:** Ocorre quando há espaços livres na memória, mas eles são muito pequenos para alocar novos processos [5].

A MMU ajuda a gerenciar esses problemas ao permitir que o sistema operacional realoque os processos na memória sem que o próprio processo precise se preocupar com isso.

E. Relocação de Processos

O *registrador de relocação* é um componente essencial para a realocação de processos na memória. Ele armazena o *endereço base* de um processo, permitindo que a MMU ajuste os endereços lógicos de forma eficiente sempre que um processo é movido na memória.

Isso é crucial para a alocação dinâmica de memória, onde os processos podem ser movidos para diferentes locais na memória principal enquanto continuam a ser executados, sem que o código do processo precise ser modificado.

No que se trata do simulador implementado, para gerenciar a realocação de processos, cada processo é é dono de um espaço, representado por uma variável que armazena as informações de onde suas páginas começam e terminam na memória RAM.

Espera-se, portanto que a **Memory Management Unit (MMU)** seja um componente que gerencie a memória no simulador de forma mais eficiente, proporcionando um ambiente mais estável e seguro para a execução de múltiplos processos simultaneamente.

IX. Resultados

O presente trabalho descreve a implementação de um simulador que visa representar, de forma realista, a dinâmica de sistemas operacionais modernos em uma arquitetura multicore com suporte à preempção. O simulador desenvolvido está disponível em um repositório no GitHub [7]. É importante ressaltar que, para utilizar o algoritmo apresentado no repositório, pode ser oportuno o uso de algumas tecnologias, tais como C++ Orientado a Objeto [2], Visual Studio Code [3] e WSL Ubuntu [4]. O programa ainda possui um arquivo Makefile que realiza todo o procedimento de compilação e execução. Para tanto, o simulador possui as diretrizes de execução apresentadas na Tabela I.

Tabela I: Comandos úteis para compilar e executar o programa de computador.

Comando	Função
make clean	Apaga a última compilação realizada contida na pasta build
make	Executa a compilação do programa utilizando o gcc, e o resultado vai para a pasta build
make run	Executa o programa da pasta build após a realização da compilação

Nesta seção, apresentam-se os resultados obtidos a partir de simulações realizadas com o sistema, enfatizando o funcionamento dos núcleos, a gestão de recursos e a troca de contexto. Além disso, é discutido tamém uma breve comparação entre os métodos de escalonamentos existentes no sistema com e sem a utilizaçao da Mémoria Cache.

A. Saída do Sistema

A implementação da arquitetura multicore no simulador proporcionou a execução de múltiplos processos concorrentemente de forma preemptiva ou não preemptiva. Nesse sentido, a simulação inicia com a escolha do método de escalonamento, seguido pela decisão sobre a execução preemptiva. Esse funcionamento é ilustrado na Saíde de Terminal 1.

Saída de Terminal 1: Menu de configuração do Simulador da Arquitetura de Von Neumann e Pipeline MIPS

```
Escolha algum metodo de escalonamento:

1 - First Job First Service (FIFO)
2 - Shortest Job First (SJF)
3 - Highest Priority First (HPF)
4 - Sair

Digite sua opcao: 1

Gostaria de executar de forma preemptiva (sim / nao)
```

Logo após as configurações serem realizadas, os resultados são evidenciados com a interação dos processos na arquitetura multicore e seus periféricos, bem como os estados de execução, bloqueio ou retornos para à fila de processos em espera. Durante essa execução, cada núcleo do sistema é associado a um processo, representando uma unidade independente que busca acesso à área crítica da CPU para realizar suas tarefas. No entanto, a execução de um núcleo pode ser impedida em duas condições principais: (i) a CPU já estar ocupada por outro núcleo ou (ii) o periférico requisitado pelo processo já estar em uso. Essa lógica de controle é essencial para garantir a consistência e o funcionamento correto do simulador. Veja um exemplo de um possível resultado na Saída de Terminal 2.

Saída de Terminal 2: Exemplo de saída do Simulador da Arquitetura de Von Neumann e Pipeline MIPS

```
6 processos restantes

        Iniciando execucao do processo 1
        Processo 1 usa o Fone de Ouvido.
        Processo 3 nao pode executar agora
        pois o Fone de Ouvido esta ocupado.
        Processo 1 terminou a execuo.
        Liberando o Fone de Ouvido.
        Iniciando execucao do processo 2
        Processo 2 usa o Monitor.
        Acabou o tempo quantum do Processo 2
        Processo 2 retornando para a fila.
        Liberando o Monitor.
```

No resultado observado, um núcleo do sistema inicia o Processo 1 utilizando o Fone de Ouvido, enquanto o Processo 3, que também necessita do mesmo periférico, tenta ser executado por um segundo núcleo, mas é bloqueado até que o recurso seja liberado. Após a conclusão do Processo 1, o Fone de Ouvido é liberado, permitindo que o próximo processo avance. Em seguida, outro núcleo começa a execução do Processo 2, utilizando o Monitor, mas tem sua execução interrompida quando seu tempo de quantum expira, retornando à fila de processos emespera. Esse ciclo demonstra a lógica do simulador, onde processos alternam entre os estados de "bloqueado" e "pronto" dinamicamente, de acordo com a disponibilidade de recursos e as regras do escalonamento escolhido pelo usuário.

B. Comparação entre Políticas

Para avaliar o impacto e o desempenho dos processos, foram realizados testes de tempo de execução com as diferentes políticas de escalonamento, tanto em cenários preemptivos quanto não preemptivos. Embora as políticas empregadas, por padrão, sejam não preemptivas, ambos os modos foram considerados para avaliar o impacto da preempção no tempo total de execução. O objetivo foi analisar como cada política se comporta ao executar o mesmo conjunto de processos, permitindo uma avaliação comparativa entre os dois cenários.

Foram gerados gráficos para cada política de escalonamento (FCFS, SJF, HPF e GSF), mostrando os tempos de execução em 10 testes para os modos preemptivo e não preemptivo. Esses gráficos permitem visualizar a consistência e a variabilidade dos tempos de execução em cada cenário.

O algoritmo *First Come First Service* (FCFS) apresentou tempos de execução significativamente diferentes entre os modos preemptivo e não preemptivo. Observe o gráfico do FCFS na Figura 2, o qual mostra que o modo não preemptivo é consistentemente mais rápido, com tempos de execução variando entre 9 ms e 17 ms, enquanto o modo preemptivo variou entre 72 ms e 79 ms.

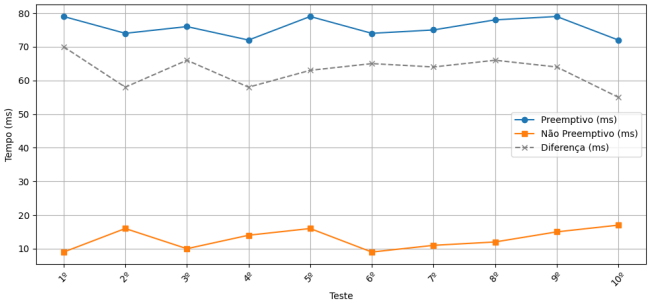

Figura 2: Comparação de Tempos FCFS Preemptivo vs Não Preemptivo

Esses resultados sugerem que o FCFS não preemptivo é mais eficiente, pois a ausência de interrupções e troca de contexto entre processos permite uma execução mais rápida. Em contrapartida, no modo preemptivo, o escalonador precisa alternar constantemente entre os processos, o que introduz overhead adicional, aumentando o tempo de execução.

Já no algoritmo *Shortest Job First* (SJF), o comportamento também variou consideravelmente entre os modos preemptivo e não preemptivo. O gráfico do SJF, apresentado na Figura 3, revela que o modo não preemptivo também é mais eficiente, com tempos de execução entre 11 ms e 22 ms, enquanto o modo preemptivo variou entre 71 ms e 89 ms.

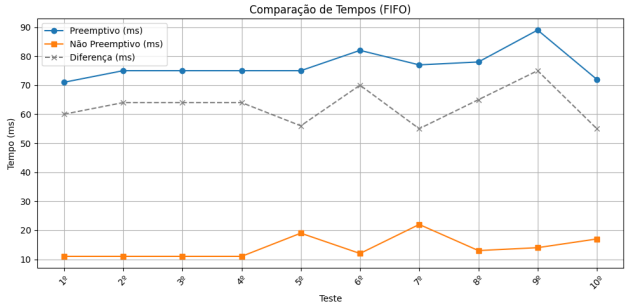

Figura 3: Comparação de Tempos SJF Preemptivo vs Não Preemptivo

O SJF preemptivo, assim como o FCFS preemptivo, sofre com o overhead de interrupções frequentes, o que aumenta o tempo de execução. No entanto, a política de priorizar os processos menores e mais rápidos no modo não preemptivo resulta em um desempenho semelhante em comparação com o FIFO.

Se trantando do algoritmo *Highest Priority First* (HPF) a diferença significativa entre os modos preemptivo e não preemptivo persiste. É possível visualizar isso no gráfico do HPF, Figura 4, o qual demonstra que o modo não preemptivo é superior, com tempos de execução entre 10 ms e 18 ms, enquanto o modo preemptivo variou entre 79 ms e 115 ms.

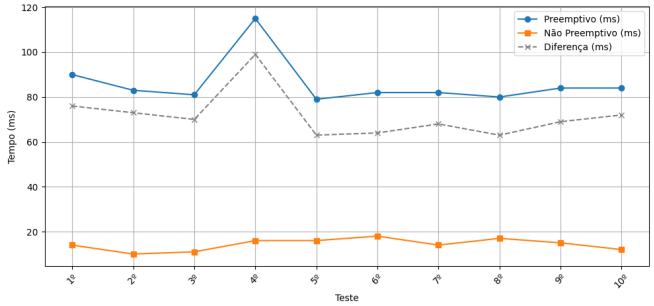

Figura 4: Comparação de Tempos HPF Preemptivo vs Não Preemptivo

O HPF preemptivo, similarmente ao SJF e FIFO preemptivo, apresenta maior tempo de execução devido à necessidade de alternar entre processos com diferentes prioridades, introduzindo overhead adicional. Isso confirma mais uma vez que, devido à preempção de processos, o número de ciclos na pipeline aumenta proporcionalmente, o que, por sua vez, justifica a diferença significativa no tempo de execução dos processos.

Por fim, no algoritmo *Greatest Similarity First (GSF)*, os resultados de execução também mostraram uma diferença significativa entre os modos preemptivo e não preemptivo. O gráfico do GSF, apresentado na Figura 5, demonstra que o modo não preemptivo apresenta tempos de execução consideravelmente mais baixos, com variação entre 10 ms e 20 ms. Em contraste, o modo preemptivo apresenta tempos que variam entre 73 ms e 87 ms.

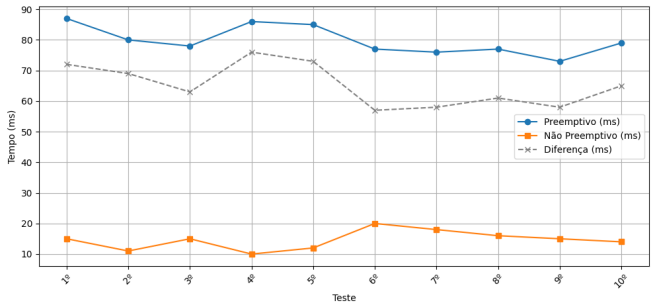

Figura 5: Comparação de Tempos GSF Preemptivo vs Não Preemptivo

Esses resultados indicam, mais uma vez, que a ausência de preempção proporciona um desempenho superior, como observado nos outros algoritmos. No entanto, ao contrário do que foi registrado para os algoritmos FCFS, SJF e HPF, a política Greatest Similarity First (GSF) apresenta um custo adicional significativo em termos de consumo de memória, uma vez que depende de uma matriz de similaridade para ser executada. Embora seus tempos de execução sejam comparáveis aos dos outros algoritmos, o uso de recursos extras

torna essa política menos eficiente no contexto geral. Assim, mesmo com um desempenho semelhante em termos de tempo, o custo adicional em recursos faz com que o GSF seja uma escolha subótima para o simulador, considerando o impacto global no sistema.

Esses dados reforçam a conclusão de que, em cenários que envolvem escalonamento de processos com alta frequência de alternância, o overhead introduzido pela preempção pode impactar negativamente o desempenho geral, tornando o modo não preemptivo uma escolha mais eficiente para a maioria dos cenários testados.

A comparação entre as políticas de escalonamento é resumida na Figura 6, que apresenta uma visão geral, comparando as médias dos tempos de execução em cada política.

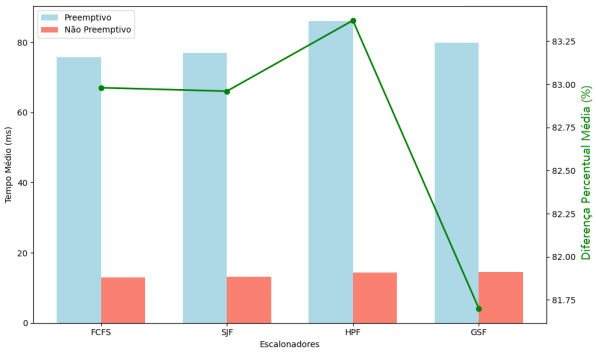

Figura 6: Comparação de Tempos Médios Entre os Escalonadores

A análise do gráfico e dos dados revela que a ordem de execução dos processos, bem como as diferenças nos modelos de programação adotados em cada política de escalonamento, apresentaram um **impacto mínino** no desempenho geral do simulador. Essa constatação é evidente ao observar que, nas comparações entre os escalonadores FCFS, SJF, HPF e GSF não há variações expressivas no tempo total necessário para a execução do mesmo conjunto de processos.

A ausência de diferenças significativas nos tempos de execução entre as políticas de escalonamento pode ser atribuída a uma série de fatores, como a ordem de execução dos processos e o overhead introduzido pela preempção. Além disso, a pequena variação nos tempos entre as políticas pode estar relacionada ao modelo de programação adotado por cada uma delas, já que cada política é projetada de forma a atender suas próprias particularidades e especificidades. Nesse contexto, os resultados práticos obtidos com o simulador ressaltam a importância de levar em consideração não apenas os aspectos teóricos, mas também as características específicas do sistema e dos processos práticos para escolher a política de escalonamento mais adequada.

Embora as políticas de escalonamento tenham suas vantagens e desvantagens teóricas, sua eficácia prática depende de fatores como a distribuição dos tempos de ciclo dos processos, tempo desembolsado na organização para manter a fila de processos em espera e o overhead associado à preempção. Sendo assim, ao escolher uma política de escalonamento, é essencial realizar testes práticos que simulem as condições reais do sistema em questão. A Tabela II resume as principais características de cada política.

Tabela II: Resumo da comparação de Políticas de Escalonamento

Política	Vantagens	Desvantagens	Cenário Ideal
FCFS	Simplicidade e previsibilidade	Pode gerar altos tempos de espera	Sistemas simples
SJF	Minimiza o tempo médio de resposta	Possibilidade de starvation	Processos com tempos bem definidos
HPF	Prioriza processos críticos	Starvation de processos de baixa prioridade	Sistemas com tarefas críticas

Os resultados do simulador indicam que a escolha final deve ser fundamentada em uma análise detalhada dos requisitos específicos do sistema, bem como nos resultados obtidos a partir de testes práticos. Portanto, ao selecionar uma política de escalonamento, é essencial considerar não apenas os resultados provenientes de simulações teóricas, mas também as características particulares do sistema em que ela será aplicada. Fatores como a preempção, a ordem de execução dos processos e o modelo de programação adotado precisam ser cuidadosamente avaliados, a fim de garantir que a política escolhida atenda de maneira eficaz aos requisitos de desempenho e eficiência do sistema.

C. O Impacto da Cache sobre as Políticas de Escalonamento

Após a ativação da cache no simulador, observou-se uma variação significativa nos tempos médios de execução dos processos ao comparar os cenários com e sem cache para os algoritmos de escalonamento Preemptivo. O Gráfico na Figura 7 apresenta os resultados obtidos, destacando os principais aspectos dessa análise. E logo após, as principais percepções observadas são discutidas.

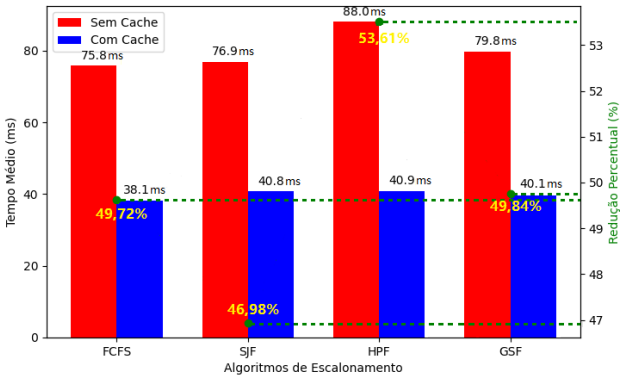

Figura 7: Comparação do Tempo Médio de Execução entre Diferentes Políticas de Escalonamento e sua Redução Percentual com e sem o uso da Cache

No cenário sem o uso de cache, os algoritmos apresentaram diferenças expressivas nos tempos médios de execução entre os modos Preemptivo e Não Preemptivo. Os algoritmos FCFS,

SJF, HPF e GSF apresentaram uma diferença percentual média de 82% entre os dois cenários, evidenciando a sobrecarga causada pelas constantes preempções.

Como solução, foi introduzida uma memória cache no simulador com o objetivo de reduzir o tempo médio de execução dos algoritmos Preemptivos. Após a ativação do cache, o sistema apresentou uma redução média no tempo de processamento, variando entre 37 ms e 43 ms a menos em comparação com a execução sem cache. Esse resultado demonstra o impacto positivo do uso de cache na otimização do processamento de dados e instruções, melhorando a eficiência do sistema.

Na teoria, a utilização de cache tem como objetivo economizar ciclos na pipeline, permitindo que dados frequentemente acessados ou instruções já decodificadas estejam prontamente disponíveis, evitando a necessidade de novos acessos à memória principal, que possuem maior latência. Essa teoria foi claramente confirmada na prática, já que os resultados previamente armazenados na cache permitiram uma execução mais ágil dos processos.

O algoritmo HPF apresentou a maior redução, com uma economia de 47.1 ms no tempo médio de execução, o que representa uma economia percentual de aproximadamente 53% no tempo de processamento. Esse comportamento pode ser explicado pelo fato de que, em algoritmos de prioridade, algumas instruções tendem a se repetir ao lidar com processos de alta prioridade, o que maximiza o aproveitamento da cache.

Nos algoritmos FIFO, SJF e GSF, as reduções também foram expressivas, com melhorias de 37.7 ms, 36.2 ms e 39.7 ms, respectivamente. Todas essas políticas apresentaram uma economia superior a 47% no tempo de processamento, evidenciando o impacto positivo da cache na execução dos processos.

No caso do FIFO, a natureza sequencial do escalonamento facilita o reaproveitamento das instruções armazenadas. Já no SJF, que prioriza processos de menor duração, a cache foi eficaz ao armazenar instruções repetidas em processos curtos e frequentemente acessados.

No caso específico do algoritmo GSF, que prioriza a similaridade entre os processos e, teoricamente, deveria ter sido a política de escalonamento mais beneficiada pelo uso da cache devido a priorização de processos semelhantes, essa expectativa não se confirmou na prática.

Esse comportamento pode ser explicado pelo modelo de programação implementado para o Greatest Similarity First (GSF), que apresenta uma complexidade significativamente maior em comparação aos outros métodos de escalonamento. A lógica intricada para identificar e ordenar processos com base em sua similaridade introduziu sobrecarga computacional, o que resultou em um tempo de processamento superior as espectativas teóricas, mesmo com o uso da cache.

Essa limitação evidência a importância de uma implementação eficiente para maximizar os benefícios da cache, especialmente em políticas de escalonamento com modelos complexos. Uma abordagem mais otimizada poderia potencializar ainda mais o aproveitamento da cache, alinhando o comportamento prático ao esperado na teoria.

Entretanto, os testes iniciais foram conduzidos com processos de similaridade completamente aleatórias, resultando nos tempos médios de execução analisados anteriormente. Em uma tentativa de otimizar a política de escalonamento Greatest Similarity First (GSF), a similaridade entre os processos foi gradualmente aumentada e estabilizada em percentuais fixos, como 60%, 70% e 80%. Durante esses experimentos, observou-se uma leve redução no tempo de processamento, o que pode ser explicado pelo fato de que processos mais semelhantes compartilham conjuntos de instruções e dados recorrentes e esse compartilhamento favorece um maior reaproveitamento das informações armazenadas na cache, o que, por sua vez, reduziu ainda mais o tempo de processamento das instruções.

Todavia, ao testar cenários onde a similaridade entre processos era muito grande, o tempo médio de execução estabilizou e não apresentou mais reduções significativas. Teoricamente, esse fenômeno pode ser compreendido a partir do princípio do *cache hit rate* [7], que representa a taxa de acertos na recuperação de dados da cache. Em níveis muito altos de similaridade, o *cache hit rate* já está próximo do seu máximo, pois a maioria das instruções necessárias já estão armazenadas e acessíveis rapidamente. Assim, mesmo aumentando ainda mais a similaridade entre os processos, não há ganho adicional, pois o tempo de acesso à cache já está otimizado ao máximo dentro das limitações impostas pela modelo de construção do simulador.

Esse comportamento evidencia um limite prático na otimização proporcionada pelo cache em sistemas operacionais. Embora o uso de processos mais similares favoreça um melhor desempenho para o **GSF**, há um ponto a partir do qual o ganho torna-se marginal. Esse achado reforça a importância do equilíbrio na organização das filas de escalonamento, onde garantir um certo grau de similaridade entre os processos pode ser benéfico, mas depender exclusivamente disso para melhorar o desempenho não traz vantagens além de um determinado ponto.

Em todos os casos, a cache desempenhou um papel fundamental ao reduzir o número de acessos à memória principal, eliminando gargalos na pipeline e garantindo uma execução mais eficiente dos processos. Esse comportamento reforça a importância de técnicas de cache em sistemas operacionais modernos, especialmente em cenários com escalonamento Preemptivo, onde interrupções frequentes aumentam a necessidade de acessos rápidos a instruções previamente processadas.

D. Uma implementação funcional da MMU

Sabemos que, conforme a teoria, a *Memory Management Unit* (MMU) é um componente que deveria possibilitar uma gestão mais eficiente e segura da memória RAM no simulador [4]. Tendo isso em mente, ao analisar os resultados obtidos na comparação dos tempos de execução do algoritmo preemptivo *First Come First Service* (FIFO), tanto com MMU quanto sem

MMU, observou-se na prática que a MMU conseguiu proporcionar as melhorias esperadas. Observe melhor no Gráfico 8, o qual ilustra os tempos de execução de cada teste nos dois cenários, além de evidenciar a diferença entre os tempos registrados.

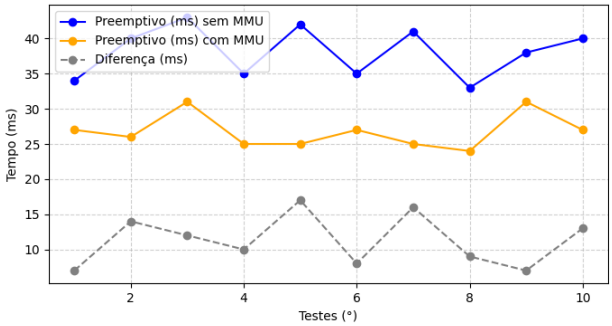

Figura 8: Comparação de Tempo de Execução: Preemptivo com e sem MMU.

A partir dos resultados, é possível visualizar uma redução significativa no tempo de execução dos processos, o que não pode ser ignorado. Assim, fica claro que a introdução da MMU no simulador resultou em uma diminuição considerável no tempo de execução, embora se trate de uma abstração rudimentar e de caráter educacional, projetada para estudar o funcionamento básico da Unidade de Gerenciamento de Memória.

Por exemplo, no primeiro teste, o tempo de execução sem MMU foi de 34 ms, enquanto com MMU foi de 27 ms, uma diferença de 7 ms. Esta tendência de redução se manteve ao longo dos demais testes, com a maior diferença observada no quarto teste, onde o tempo de execução sem MMU foi de 43 ms e com MMU foi de 31 ms. Logo a MMU contribuiu para um ganho de desempenho, o que indica que a abstração por meio dos endereços virtuais, mesmo de forma simples, teve um impacto positivo no tempo de execução dos processos.

Conclui-se entao que, no contexto do simulador apresentado nesse trabalho, embora a MMU não seja tão avançada quanto as implementações reais em sistemas operacionais modernos, o impacto observado sugere que a gestão eficiente da memória é um fator crucial na redução do tempo de execução de processos.

X. Conclusão

O desenvolvimento do simulador de sistemas operacionais multicore com suporte à preempção apresentou resultados significativos para a análise do impacto de diferentes políticas de escalonamento. Os experimentos realizados evidenciaram que as políticas FIFO, SJF, HPF e GSF em modo não preemptivas demonstram maior eficiência se comparadas ao modo preemptivo de execução. Isso ocorre por causa da ausência de overhead de alternâncias frequentes entre processos na forma não preemptiva. Em contrapartida, as políticas preemptivas podem oferecer maior flexibilidade e são mais adequadas para cenários que demandam rápida resposta e prioridade diferenciada entre processos.

Os resultados reforçam que a escolha da política de escalonamento deve ser cuidadosamente alinhada aos requisitos específicos do sistema, considerando o dilema entre desempenho e flexibilidade. Nessas circuntâncias, é possivel utilizar o simulador para validar as características teóricas das políticas de escalonamento, destacando a importância de considerar aspectos práticos, como a gestão de recursos e a troca de contexto, para otimizar o desempenho de sistemas operacionais modernos.

O aprimoramento do simulador a partir da ativação da cache demonstrou impactos positivos muito bons nos tempos médios de execução das políticas de escalonamento. Os resultados mostraram uma redução média de 50% no tempo de processamento dos algoritmos como HPF, FIFO, SJF e GSF. Apesar de todas essas melhorias também serem observadas no GSF, a complexidade de sua implementação limitou os benefícios proporcionados pela cache. Logo, é impossivel não perceber a importância do uso de cache em sistemas com escalonamento, pois essa estrutura adicional contribuiu significativamente para a otimização do tempo de processamento no simulador.

Além disso tudo, a introdução de um sistema de gerenciamento para a memória RAM, como o Memory Management Unit, demonstrou ser importante no contexto do simulador apresentado. A MMU contribuiu para tornar a paginação dentro da memória principal mais eficiente, resultando em uma melhoria no tempo de execução dos processos.

Assim, o trabalho contribui para o entendimento do comportamento e da eficiência de políticas de escalonamento em arquiteturas multicore com cache, oferecendo uma ferramenta valiosa para aprendizado, análise e tomada de decisão em projetos de sistemas computacionais.

Referências

[1] BERTINI, Luciano. Capítulo 5 Pipeline. 2019. Disponível em: https://www.professores.uff.br/Cap-5-Pipeline.pdf.. Acesso em: 14 jan. 2025.

[2] CPLUSPLUS. C++ Language v3.3.4s. 2000-2023. Disponível em: https://cplusplus.com/doc/tutorial/. Acesso em: 14 jan. 2024.

[3] Microsoft. Download for free of Visual Studio Code (Version 1.84). [S.l.: s.n., s.d.]. Disponível em: https://code.visualstudio.com/. Acesso em: 14 jan. 2024.

[4] GRACIELLY, J. WSL 2 - A solução para rodar Linux dentro do Windows 10 - Root #08 [Vídeo]. YouTube, 2021. Disponível em:https://www.youtube.com/watch?v=hd6lxt5iVsg. Acesso em: 14 jan. 2024.

[5] Andrew S. Tanenbaum. Sistemas Operacionais Modernos Pearson Prentice Hall. 2003 Segunda edição. Título original: Modern Operating Systems

[6] SANTOS, Luan. SO: Simulador da Arquitetura de Von Neumann e Pipeline MIPS. 2025. Disponível em: github.com/LuanLuL/Sistemas_Operacionais. Acesso em: 16 jan. 2025.

[7] UPLOADCARE. Cache hit rate: what it is and how to optimize it. Disponível em: https://uploadcare.com/learning/cdn/cache-hit-rate/. Acesso em: 18 fev. 2025.