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

*Módulo 1: Arquitetura Multicore e Suporte a Preempção

1st Celso Vinícius Sudário Fernandes
Bacharelado em Engenharia de Computação
CEFET-MG, Campus V
Divinópolis, Brasil
celso.23@aluno.cefetmg.br

2nd Pedro Henrique Pires Dias
Bacharelado em Engenharia de Computação
CEFET-MG, Campus V
Divinópolis, Brasil
pedro.dias@aluno.cefetmg.br

***Resumo*—Este artigo apresenta uma arquitetura computacional multi-core desenvolvida para simular a execução e escalonamento de processos em um sistema operacional simplificado. A implementação utiliza componentes fundamentais, como pipeline de execução de instruções, gerenciamento de recursos e controle de processos, com base em conceitos de sistemas operacionais modernos. A metodologia emprega uma estrutura modular e multi-threaded, buscando demonstrar como núcleos paralelos podem impactar o desempenho geral do sistema. Os resultados obtidos a partir de logs detalhados permitem avaliar a eficiência do simulador e identificar áreas que podem ser otimizadas, como o balanceamento de carga e o gerenciamento de recursos compartilhados. Este trabalho visa contribuir para o entendimento do funcionamento interno de sistemas operacionais e propor a aprendizagem através de ambientes de simulação.**

***Palavras Chave*—Sistemas Operacionais, Escalonamento, Processos, Arquitetura Multi-core, Pipeline, Execução, Gerenciamento de Memória, Simulação Computacional, Núcleos, Paralelismo;**

I. Introducão

O estudo de sistemas operacionais aborda questões essenciais, como o escalonamento de processos, gerenciamento de recursos e execução de instruções. Com o advento de arquiteturas multi-core, problemas como o impacto do escalonamento e a coordenação entre núcleos paralelos tornaram-se ainda mais relevantes. Este trabalho propõe a criação de um simulador multi-core que explora conceitos fundamentais desses sistemas, como pipelines de execução e gerenciamento de recursos, buscando analisar o comportamento e o desempenho em diferentes cenários de execução paralela. Após esta introdução, a seção II apresenta a metodologia detalhada, e os resultados são discutidos na seção III. Por fim, as conclusões e perspectivas futuras são apresentadas na seção IV.

II. Metodologia

O presente estudo utilizou um simulador multi-core implementado para explorar o comportamento de sistemas operacionais simplificados. A arquitetura do simulador foi construída com componentes essenciais, como pipeline de execução de instruções, controle de processos e escalonamento dinâmico. Os dados de entrada consistem em um conjunto de processos gerados artificialmente, cada um configurado com atributos como tempo de execução e instruções específicas para análise de cenários variados.

A. Ferramentas Utilizadas

O simulador foi implementado utilizando a linguagem de programação C++ pela sua eficiência em desempenho e suporte a programação paralela. Foram utilizadas bibliotecas padrão da linguagem, como `<thread>` para manipulação de threads e `<mutex>` para sincronização. Os logs de execução foram gerados em arquivos `.txt` para análise posterior, utilizando `<fstream>`. A organização dos módulos seguiu os padrões de organização e padronização, com headers e implementações específicas para cada componente do sistema.

B. Etapas de Implementação

O desenvolvimento do simulador seguiu etapas sequenciais, cada uma responsável por uma funcionalidade essencial. Essas etapas foram estruturadas para viabilizar a interação entre os componentes e garantir um funcionamento coeso. A seguir, são detalhadas as principais etapas, relacionadas ao pseudocódigo apresentado:

Algoritmo 1: Simulador da Arquitetura de Von Neumann com Pipeline MIPS

Entrada: Configurações do sistema (`configBootloader.txt`), arquivos de instruções (*instrFiles*)
Saída: Logs de execução e estados finais dos processos

Inicialização:
Carregar configurações do `Bootloader`
Inicializar `RAM` e `Disco`
Criar `PCBs` a partir de *instrFiles*
Associar recursos periféricos aos processos
Adicionar os `PCBs` à fila de prontos no `Escalonador`
Inicializar núcleos (`Cores`) como threads independentes
Execução Multi-core:
for cada núcleo em `Cores` **do**
    Iniciar *thread* para execução do núcleo
end for
Execução por Núcleo:
for cada núcleo em execução **do**
    while existem processos na fila do `Escalonador` **do**
        Obter próximo processo (`PCB`) do `Escalonador`
        Restaurar estado do processo (*registradores*, *pipeline*, `PC`)
        while processo não está finalizado ou bloqueado **do**
            Executar *Instruction Fetch* na `RAM`
            Decodificar e executar a instrução (*Instruction Decode e Execute*)
            Atualizar `PC` e decrementar `quantum`
            if recurso periférico indisponível ou acesso inválido **then**
                Atualizar estado para `BLOQUEADO` e retornar ao `Escalonador`
            end if
        end while
        Salvar estado do processo (*registradores*, *pipeline*, `PC`)
        Atualizar estado no `Escalonador` (`FINALIZADO` ou `PRONTO`)
    end while
end for
Finalização:
Liberar memória e recursos alocados
Salvar logs de execução para análise
return Logs de execução

1) Multicore: A implementação inicial concentrou-se no suporte a múltiplos núcleos (`Cores`). Cada núcleo foi configurado como uma *thread* independente, permitindo a execução paralela de processos. O `Bootloader` inicializou os núcleos, enquanto o `Escalonador` foi ajustado para distribuir processos entre eles. O controle de sincronização para acesso à `RAM` foi fundamental para tentar garantir uma consistência na execução.

2) PCB (Process Control Block): Com os núcleos em funcionamento, foi desenvolvida a estrutura dos `PCBs`, responsáveis por armazenar o estado completo de cada processo. As informações incluíram registradores, contador de programa (`PC`), quantum restante, memória alocada e recursos associados. O `PCB` também foi integrado ao `Core` e ao `Escalonador`, permitindo salvar e restaurar estados durante a troca de contexto.

3) Estados do Ciclo de Vida dos Processos: Nesta etapa, foram implementados os estados `PRONTO`, `EXECUÇÃO`, `BLOQUEADO` e `FINALIZADO`, que definem o ciclo de vida de um processo. O `Escalonador` gerencia a transição entre os estados. Por exemplo, processos que atingem o limite do quantum são bloqueados e retornam à fila de prontos.

4) Preempção: A última etapa consistiu na implementação da preempção. O núcleo interrompe a execução de um processo ao término do quantum, salvando seu estado no `PCB` e devolvendo-o ao `Escalonador`. Essa abordagem buscou garantir que os processos fossem alternados de acordo com a política de escalonamento `FCFS`, buscando uma distribuição mais simples da CPU.

Essas etapas foram implementadas de forma linear, com base no pseudocódigo apresentado. Esse processo buscou criar um simulador modular e funcional, capaz de lidar com múltiplos processos e recursos compartilhados, priorizando a integração entre os demais componentes do simulador e a coerência no fluxo de execução.

C. Estrutura do Simulador

O simulador utiliza uma estrutura modular, onde cada componente desempenha funções específicas, desde a inicialização até a execução dos processos. A seguir, são apresentados os principais elementos e suas interações no fluxo de execução.

1) `Bootloader`: é responsável pela inicialização do simulador. Ele lê as configurações do sistema, cria os PCBs com base nos arquivos de instruções e associa recursos periféricos a processos específicos. Também aloca memória para os processos e configura os logs de execução. O `Bootloader` busca organizar o ambiente para que todos os componentes estejam preparados antes da execução. Após a inicialização, ele delega os PCBs ao `Escalonador` para que sejam geridos durante o runtime.

2) `PCB (Process Control Block)`: armazena o estado de cada processo, incluindo o PC, quantum, registradores, memória alocada e recursos periféricos associados. Ele permite troca de contexto ao salvar o estado intermediário do pipeline e os dados do processo, buscando possibilitar que a execução possa ser retomada corretamente. Ele interage diretamente com o `Core`, que utiliza seu estado para executar instruções, e com o `Escalonador`, que organiza os processos de acordo com o estado registrado.

3) `Core`: executa as instruções dos processos atribuídos pelo `Escalonador`. Cada núcleo opera de forma independente em threads, utilizando o pipeline para processar instruções. O `Core` interage com o `PCB` para restaurar e salvar o estado dos processos e verifica a disponibilidade de recursos periféricos antes da execução. Em caso de indisponibilidade, ele atualiza o estado do processo para `BLOQUEADO`, devolvendo-o ao `Escalonador`. Ao final da execução, o `Core` busca liberar os recursos associados ao processo finalizado.

4) `Escalonador`: atua na organização dos processos em diferentes estados (`PRONTO`, `BLOQUEADO`, `EXECUÇÃO` e `FINALIZADO`). Ele distribui os processos para os núcleos disponíveis e administra a fila de bloqueados, buscando movimentá-los de volta para a fila de prontos quando os recursos necessários são liberados. A política utilizada pelo escalonador é baseada em FCFS (*First Come, First Served*) para processos prontos, onde os processos são atendidos na ordem de chegada. Sua interação contínua com o `Core` e os `PCBs` busca viabilizar que os núcleos sempre tenham processos disponíveis para execução.

5) `ProcessManager`: é responsável pela criação inicial dos `PCBs`, utilizando os arquivos de instruções encontrados no `Disco`. Ele define os valores de quantum, configura os registradores e associa recursos periféricos aos processos.

Também gerencia a alocação de memória na RAM, determinando faixas de endereços para cada processo. Após configurar os PCBs, o ProcessManager os repassa ao Bootloader para inicialização.

6) Pipeline: realiza o processamento das instruções dividindo-o em cinco etapas: Fetch, Decode, Execute, Memory Access e Write Back. Ele interage com o Core para processar instruções e atualiza o estado intermediário do PCB, buscando possibilitar uma troca de contexto eficiente e execução contínua. Além disso, as informações sobre o estado do pipeline são armazenadas para permitir rastreamento e depuração.

7) RAM e Disco: trabalham juntos para fornecer os dados e instruções necessários aos processos. O Disco armazena os arquivos de instruções que são carregados na RAM pelo Bootloader. Durante a execução, a RAM é utilizada diretamente pelo Core para leitura e escrita, sincronizando o acesso entre múltiplos núcleos.

8) Periféricos: são dispositivos compartilhados, como impressoras ou teclados, gerenciados pelos PCBs. O uso dos periféricos ocorre por meio de verificações realizadas pelo Core, que consulta o PCB para determinar se o recurso está disponível.

De modo geral, os componentes do simulador buscam interagir de maneira integrada para realizar o fluxo completo de execução. O Bootloader configura os PCBs e delega ao Escalonador, que distribui os processos para os Cores. Durante a execução, o Pipeline processa instruções enquanto acessa a RAM para leitura e escrita. O Escalonador coordena o estado dos processos, utilizando informações fornecidas pelos PCBs, e interage com os núcleos para reatribuir processos prontos.

D. Modelagem dos Experimentos

A modelagem dos experimentos foi projetada para validar o funcionamento do simulador em diferentes cenários, avaliando aspectos como o comportamento do escalonador, a execução multi-core e o gerenciamento de recursos. A seguir, são descritas as etapas principais para a aplicação da metodologia e a obtenção de resultados.

1) Definição dos Cenários: Através do arquivo configBootloader.txt foram definidos cenários de simulação variando o número de núcleos (NUM_NUCLEOS) e os valores mínimo e máximo do quantum dos processos (QUANTUM_PROCESS_MAX e QUANTUM_PROCESS_MIN). Já o arquivo setRegisters.txt é responsável por definir a quantidade de registradores e seus valores. A partir do diretório /data/instr foram definidos as instruções que serão executadas pelos processos, onde cada arquivo .txt desse diretório será atribuído à um processo.

Cada cenário foi projetado para testar aspectos específicos:

- Cenários com diferentes números de processos para avaliar a capacidade do escalonador em gerenciar filas grandes.
- Cenários com múltiplos núcleos para validar a execução paralela e a preempção.

2) Execução dos Experimentos: Para cada cenário, o simulador foi configurado com os parâmetros definidos citados acima e os seguintes passos foram realizados:

1) Configuração inicial por meio do Bootloader, carregando a quantidade de instruções disponíveis e definindo o número de processos.
2) Registro detalhado dos logs de execução para análise do ciclo de vida dos processos.

3) Métricas Analisadas: Os resultados dos experimentos foram avaliados com base em métricas definidas, utilizando os logs de execução para validar e analisar o comportamento do simulador.

- **Tempo total de execução**: Mede o tempo necessário para completar todos os processos.
- **Taxa de utilização dos núcleos**: Avalia o tempo em que os núcleos estavam ocupados processando versus ociosos, refletindo a eficiência do paralelismo na arquitetura do sistema.
- **Comportamento dos estados**: Verifica a transição correta entre os estados PRONTO, BLOQUEADO e FINALIZADO.
- **Gerenciamento de recursos**: Compara os valores dos registradores antes e depois da execução para validar a integridade das operações solicitadas.

Os experimentos buscaram identificar possíveis gargalos e inconsistências no simulador, fornecendo subsídios para futuras melhorias.

III. Resultados e Discussão

A análise dos resultados obtidos com o simulador multicore visa avaliar o comportamento do sistema na execução e escalonamento de processos em cenários paralelos. Para isso, foram gerados processos com características variadas, distribuídos entre os núcleos disponíveis, considerando a alocação de recursos e a eficiência do pipeline de execução. O objetivo desta seção é explorar como as decisões de escalonamento, alocação de memória e execução impactam o desempenho geral do sistema, destacando interações entre os núcleos e os processos, além de identificar pontos críticos para futuras otimizações.

Com base na seção anterior, esta seção apresenta os resultados obtidos e discute o impacto das configurações no desempenho do simulador. São analisados o tempo total de execução, a eficiência dos núcleos e o comportamento dos processos, permitindo identificar interações críticas e oportunidades de otimização.

A. Reprodução

A fim de atestar a transparência e a reprodutibilidade dos resultados apresentados neste estudo, todo o código-fonte e a documentação detalhada deste estudo estão disponíveis em um repositório GitHub público. O diretório ”src/”contém os scripts C++ usados para a análise, permitindo que outros

pesquisadores validem, reproduzam ou expandam a pesquisa. A URL do repositório Git é fornecida na seção de Referências [2] deste artigo.

B. Experimentos Práticos

Os experimentos foram conduzidos para validar a execução multi-core do simulador, avaliar o comportamento do escalonador em diferentes cenários e analisar o gerenciamento de recursos.

Os cenários foram configurados variando a quantidade de núcleos (1, 2, 4, 8) e a quantidade de processos (1, 2, 4, 10, 30). O quantum foi definido com valores entre 20 e 50, conforme especificado no arquivo `configBootloader.txt`. Os registradores foram configurados utilizando o arquivo `setRegisters.txt`, enquanto as instruções dos processos foram lidas a partir do diretório `data/instr`.

No cenário 1, avaliou-se o funcionamento do simulador em uma configuração básica, com 1 núcleo e 2 processos. No cenário 2, testou-se a execução de um único processo em um ambiente com 2 núcleos, analisando a utilização mínima dos recursos disponíveis. No cenário 3, examinou-se a distribuição e o gerenciamento de recursos em uma configuração com 2 núcleos e 4 processos. No cenário 4, o foco foi a escalabilidade, utilizando 4 núcleos e 10 processos para validar a eficiência em um ambiente intermediário. Por fim, no cenário 5, analisou-se o desempenho sob carga elevada, com 8 núcleos e 30 processos, avaliando a capacidade do simulador em gerenciar grandes volumes de dados e alta concorrência.

Para cada cenário, o simulador foi iniciado utilizando o `Bootloader`, que carregou as configurações definidas nos arquivos. Durante a execução, logs detalhados foram gerados, registrando o tempo total de execução, a utilização dos núcleos e as transições entre estados dos processos. Cada experimento foi repetido três vezes para garantir consistência e confiabilidade nos resultados.

Os cenários modelados criaram a base para a análise crítica e os resultados discutidos nas seções subsequentes, evidenciando as capacidades e limitações do simulador em diferentes contextos.

A seguir, apresenta-se a configuração do cenário 3, com 2 núcleos e 4 processos, destacando como os arquivos do simulador foram organizados para representar as instruções, registradores e parâmetros do sistema.

A Figura 1 apresenta a estrutura do diretório de configuração, com os arquivos disponíveis em data/instr, representando cada processo.

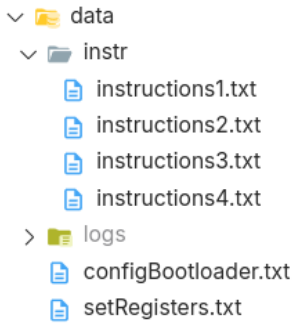

Figura 1. Estrutura do diretório de configuração.

Cada arquivo do diretório data/instr contém as instruções que serão executadas pelos processos. A Figura 2 exemplifica um conjunto de instruções configuradas para um processo

data > instr > instructions1.txt
You, há 1 segundo | 1 author (You)

```
1  ADD, 1, 2, 3
2  SUB, 4, 1, 3
3  STORE, 3, 2, 0
4  LOAD, 0, 2, 0
5  ENQ, 7, 3, 4
6  IF_igual, 8, 4, 7
```

Figura 2. Exemplo de instruções para um processo.

Abaixo a configuração do `configBootloader.txt`, onde é decidido além do diretório onde irão ficar os logs, também a quantidade de núcleos disponíveis e os valores máximo e mínimo para os quantums dos processos.

data > configBootloader.txt
You, há 1 segundo | 1 author (You)

```
1  OUTPUT_LOGS_DIR=data/logs
2  NUM_NUCLEOS=2
3  QUANTUM_PROCESS_MIN=20
4  QUANTUM_PROCESS_MAX=50
5
```

Figura 3. Configuração do Bootloader.

Por último a distribuição em `setRegisters.txt` que representa a configuração dos registradores e seus respectivos valores.

data > setRegisters.txt
You, há 1 segundo | 1 author (You)

```
1  0,1
2  1,2
3  2,10
4  3,5
5  4,7
```

Figura 4. Configuração dos registradores e seus respectivos valores no arquivo setRegisters.txt

C. Resultados Obtidos

Os resultados experimentais foram registrados em logs e analisados com base em métricas estabelecidas. A Tabela I apresenta os dados para os cinco cenários avaliados.

Tabela I
RESULTADOS CONSOLIDADOS DOS CENÁRIOS EXPERIMENTAIS.

Cenário	Núcleos	Processos	Tempo Total (ms)	Taxa de Utilização (%)
1	1	2	2.017	34.495
2	2	1	1.557	16.333 (Núcleo 1) 0.000 (Núcleo 2)
3	2	4	2.047	35.892 (Núcleo 1) 34.613 (Núcleo 2)
4	4	10	2.332	42.035 (Núcleo 1) 38.504 (Núcleo 2) 34.335 (Núcleo 3) 35.637 (Núcleo 4)
5	8	30	2.916	49.484 (Núcleo 1) 50.530 (Núcleo 2) 50.116 (Núcleo 3) 49.476 (Núcleo 4) 44.055 (Núcleo 5) 43.798 (Núcleo 6) 49.153 (Núcleo 7) 47.922 (Núcleo 8)

D. Análise por Cenário

Cenário 1: Com 1 núcleo e 2 processos, o tempo total de execução foi de 2.017 ms, e a taxa de utilização do núcleo foi de 34.495%. O núcleo permaneceu ocioso por aproximadamente 65.505% do tempo, evidenciando que a carga de trabalho era leve.

Cenário 2: No cenário com 2 núcleos e 1 processo, o tempo total de execução foi de 1.557 ms. Apenas o Núcleo 1 foi utilizado (16.333%), enquanto o Núcleo 2 permaneceu completamente ocioso (0.000%). Este resultado sugere que para cargas leves, múltiplos núcleos podem ser utilizados de forma ineficiente, ressaltando a importância de tentar equilibrar a configuração com base na carga de trabalho.

Cenário 3: Com 2 núcleos e 4 processos, o tempo total foi de 2.047 ms. Ambos os núcleos apresentaram taxas de utilização próximas, com 35.892% e 34.613%, respectivamente. A carga foi distribuída de forma relativamente equilibrada, embora períodos de ociosidade ainda foram registrados, destacando a necessidade de ajustes para cenários com baixa complexidade.

Cenário 4: No cenário com 4 núcleos e 10 processos, o tempo total de execução foi de 2.332 ms. As taxas de utilização variaram entre 34.335% e 42.035%, evidenciando uma boa distribuição da carga de trabalho. O aumento do número de processos mostrou-se proporcional ao aumento no tempo total de execução.

Cenário 5: Com 8 núcleos e 30 processos, o tempo total foi de 2.916 ms. As taxas de utilização dos núcleos variaram entre 43.798% e 50.530%. Apesar da alta concorrência, a distribuição foi eficiente, embora houvesse variações na ociosidade entre os núcleos. Este cenário evidenciou a capacidade do simulador em lidar com cargas complexas, mas apontou para possíveis melhorias no escalonamento.

E. Discussão

Os resultados indicam como o simulador respondeu às configurações testadas:

- **Desempenho Geral:** O tempo total aumentou com a complexidade dos cenários, mas o uso de múltiplos núcleos permitiu reduzir o tempo de execução por processo individual.
- **Eficiência dos Núcleos:** A ociosidade foi maior em cenários leves (ex.: Cenário 2), enquanto os cenários complexos apresentaram melhor aproveitamento dos recursos.
- **Comportamento dos Estados:** A ocorrência de processos bloqueados foi mais evidente nos cenários de alta carga, destacando a importância do gerenciamento de recursos.
- **Interações Críticas:** Ajustes no algoritmo de escalonamento podem melhorar a distribuição de processos, reduzindo o tempo de espera e aumentando a eficiência geral.

F. Conclusão Parcial

Os experimentos demonstraram que o simulador lida relativamente bem com diferentes cargas, mas apresenta oportunidades de melhoria em cenários de alta concorrência. Ajustes na política de escalonamento e no gerenciamento de recursos podem contribuir para otimizar ainda mais o desempenho do sistema.

IV. CONCLUSÃO

Este estudo explorou a implementação de um simulador multi-core para a execução e escalonamento de processos, buscando analisar como conceitos fundamentais de sistemas operacionais, como controle de processos e threads, e como se comportam em um ambiente paralelo. A abordagem adotada permitiu observar dinâmicas importantes, como o impacto do escalonamento e a execução concorrente, além de destacar os desafios envolvidos no gerenciamento eficiente de threads e dos núcleos.

Embora a metodologia tenha sido suficiente para explorar as principais questões propostas, algumas limitações foram identificadas. Entre elas, destaca-se que, embora o simulador possua suporte básico para operações de entrada e saída, essa funcionalidade ainda pode ser aprimorada para refletir cenários mais complexos e realistas. Além disso, o simulador não contempla ainda mecanismos avançados de gerenciamento de memória, como paginação ou segmentação, que poderiam ampliar sua aplicabilidade e complexidade.

Estudos futuros podem focar na inclusão de elementos mais complexos, como a criação de políticas de escalonamento com base em prioridades, o gerenciamento de cache, além de explorar o impacto do balanceamento de carga com gerenciamento de memória. Acredita-se que o aprimoramento do simulador em tais direções contribuirá para uma análise mais detalhada e aplicável ao estudo de sistemas operacionais modernos, permitindo também uma maior compreensão das dinâmicas que influenciam o desempenho em ambientes paralelos.

REFERÊNCIAS

[1] Sistemas operacionais modernos / Andrew S. Tanenbaum, Herbert Bos; tradução Jorge Ritter; revisão técnica Raphael Y. de Camargo. – 4. ed. – São Paulo: Pearson Education do Brasil, 2016.

[2] Simulador Multicore / Celso V. Sudário, Pedro H. Dias; Disponível em: https://github.com/celzin/Simulador_SO.