Universidade Federal do Rio Grande do Norte
(UFRN)
Lucas Marques dos Santos
(lucas.marques.111@ufrn.edu.br)
Kayo Gonçalves e Silva
(kayo@imd.ufrn.br)
Intituto Metrópole Digital

Processos e Threads

Natal/RN, 30 de setembro de 2025

Universidade Federal do Rio Grande do Norte
(UFRN)
Lucas Marques dos Santos
(lucas.marques.111@ufrn.edu.br)
Kayo Gonçalves e Silva
(kayo@imd.ufrn.br)
Intituto Metrópole Digital

Sumário

Universidade Federal do Rio Grande do Norte
(UFRN)
Lucas Marques dos Santos
(lucas.marques.111@ufrn.edu.br)
Kayo Gonçalves e Silva
(kayo@imd.ufrn.br)
Intituto Metrópole Digital

Resumo

O presente relatório tem como objetivo avaliar o desempenho computacional da multiplicação de matrizes quadradas, utilizando três abordagens distintas: código sequencial, paralelização por processos e por threads. A proposta visa aplicar na prática os conhecimentos adquiridos ao longo das aulas. Foram realizados diversos testes, variando o tamanho das matrizes e a quantidade de processos e threads utilizadas. Os resultados demonstram que a implementação com threads apresentou melhor desempenho, o que vai ao encontro do que foi discutido em teoria. Além disso, o experimento evidenciou a eficácia do paralelismo e da divisão de trabalho, que se destacam significativamente em relação à abordagem sequencial, especialmente à medida que o tamanho das matrizes aumenta.

Universidade Federal do Rio Grande do Norte
(UFRN)
Lucas Marques dos Santos
(lucas.marques.111@ufrn.edu.br)
Kayo Gonçalves e Silva
(kayo@imd.ufrn.br)
Intituto Metrópole Digital

Introdução

As pessoas geralmente não gostam de ditadores, mas existe um que é essencial — especialmente se você usa computadores ou celulares: o Sistema Operacional (SO). Ele exerce um papel de controle absoluto, sendo responsável por gerenciar todos os processos que ocorrem no seu dispositivo. É o SO quem decide, por exemplo, qual programa será executado, quanto de memória cada processo pode usar e em que ordem as tarefas serão realizadas.

Mas a final o que é um processo? Para isso, primeiramente precisamos diferenciar processos de programa. Um programa é nada mais que um conjunto de dados e instruções. A instância desse programa, chamamos de processo. Cada processo possui uma estrutura chamada “Process Block Control” (PCB), que armazena todas as informações necessárias para seu controle, como o estado atual do processo (em execução, pronto, bloqueado etc.), o contador de programa (que indica a próxima instrução a ser executada), os registradores da CPU, dados de escalonamento, informações de gerenciamento de memória, entre outros. É por meio do PCB que o sistema operacional consegue pausar, retomar e gerenciar processos de forma eficiente.[1]

Quando um processo é criado, ele entra na fila de “job”, que representa o conjunto de todos os processos no sistema. Caso seja admitido — ou seja, tenha todos os recursos necessários para sua execução — ele passa para o estado de pronto, aguardando que o escalonador de processos aloque uma CPU (Unidade Central de Processamento). Assim que isso ocorre, o processo entra no estado de execução. No entanto, se durante sua execução ele precisar, por exemplo, de uma entrada do teclado, ele é movido para o estado de espera. Após o término da operação de entrada/saída, o processo retorna ao estado de pronto, aguardando novamente a alocação da CPU. Se, por algum motivo, o processo estiver ocupando a CPU por tempo demais, o sistema operacional pode interrompê-lo à força, liberando a CPU para outros processos, evitando assim o acúmulo e garantindo maior fluidez na execução geral do sistema. Quando o processo conclui todas as suas tarefas, ele entra no estado de encerrado, sendo finalizado. Vale destacar que um processo também pode criar outros processos, dando origem a uma hierarquia entre eles.[1]

Por fim, vamos falar sobre “Threads”. Uma thread é um “fluxo de execução dentro de um processo”, caracterizada por ter seu próprio contador de programa, registradores de estado e pilha. Todo processo possui pelo menos uma thread, mas pode ter várias, compartilhando entre si os recursos do processo, como arquivos abertos e espaço de memória. Diferente dos processos, threads não são independentes entre si, o que torna sua comunicação mais rápida, porém também mais suscetível a problemas como condições de corrida. Existem dois tipos principais de threads: threads de núcleo e threads de usuário. As threads de núcleo são conhecidas pelo sistema operacional, que realiza seu gerenciamento através do escalonador, assim como acontece com os processos. Já as threads de usuário são gerenciadas por uma biblioteca de threads interna ao processo e não são diretamente visíveis ao sistema operacional. Existem três modelos clássicos de mapeamento entre threads de usuário e de núcleo: o modelo N:1, onde N threads de usuário são mapeadas para uma única thread de núcleo; o modelo 1:1, em que cada thread de usuário corresponde a uma thread de núcleo; e o

Universidade Federal do Rio Grande do Norte
(UFRN)
Lucas Marques dos Santos
(lucas.marques.111@ufrn.edu.br)
Kayo Gonçalves e Silva
(kayo@imd.ufrn.br)
Intituto Metrópole Digital

modelo N:M, onde N threads de usuário são mapeadas em M threads de núcleo. Vale lembrar que, do ponto de vista do sistema operacional, apenas threads de núcleo são visíveis.[2]

Universidade Federal do Rio Grande do Norte
(UFRN)
Lucas Marques dos Santos
(lucas.marques.111@ufrn.edu.br)
Kayo Gonçalves e Silva
(kayo@imd.ufrn.br)
Intituto Metrópole Digital

Metodologia

Para os experimentos, foram desenvolvidos quatro programas em Linguagem C, executados no ambiente WSL(Windows Subsystem for Linux). O "CriarMatriz.c*"* gera duas matrizes M1 e M2 com números aleatórios de 0 a 100, recebendo como entrada suas dimensões e produzindo dois arquivos .txt contendo os elementos e o tamanho de cada matriz. O "Sequencial.c*"* realiza a multiplicação dessas matrizes de forma sequencial, utilizando como entrada os arquivos gerados. Já os programas "Processos.c" e "Threads.c" implementam a multiplicação paralela, recebendo os arquivos .txt e um valor inteiro P, e realizando a divisão do trabalho por meio de processos e threads, respectivamente. A saída dos programas consiste em arquivos .txt contendo os resultados das multiplicações e os tempos de execução correspondentes. No caso do programa sequencial, será gerado um único arquivo com o resultado completo e, ao final, o tempo total gasto no cálculo. Já nos programas que utilizam paralelismo, Processos.c e Threads.c, cada processo ou thread gera um arquivo .txt individual registrando o resultado e o tempo gasto na execução da sua parte do trabalho. O código-fonte completo e informações adicionais estão disponíveis no repositório, cujo link é apresentado ao final deste relatório.[3]

Vamos ter dois experimentos E1 e E2:

- **E1:** Vamos executar cada um dos programas Sequencial, Processos e Threads, com diversos tamanho de matrizes. Iniciando com 100x100, de modo a dobrar os tamanhos, até chegar em uma matriz em que no sequencial fique no mínimo dois minutos em execução. No caso nos programas paralelos, vamos ter um valor de P igual (colunas de M1 * Linhas de M2) / 8.
- **E2:** Para as matrizes achadas em no E1, vamos executar novamente os programas Processos e Threads, realizando diversos teste com diferentes valores de P, com o objetivo de encontrar um novo valor que faça sentido para discussão proposta neste relatório.

Para ambos os experimentos vamos executar cada programa 10 vezes, com o objetivo de pegar o tempo médio de execução e em seguida, plotar um gráfico com o tempo médio decorrido por cada programa em função do tamanho da matriz para E1 e em função do valor de P para o E2. O Tempo de execução do sequencial está no final do arquivo .txt. Já nos programas paralelos, vamos pegar o maior tempo de execução, também contido no final de cada arquivo .txt produzido por cada processo e thread.

Com isso, esperamos responder 3 perguntas principais:

- Qual o motivo dos resultados obtidos nos experimentos E1? O que pode ter causado o comportamento observado?
- Qual o motivo dos resultados obtidos nos experimentos E2? O que pode ter causado o comportamento observado?

Universidade Federal do Rio Grande do Norte
(UFRN)
Lucas Marques dos Santos
(lucas.marques.111@ufrn.edu.br)
Kayo Gonçalves e Silva
(kayo@imd.ufrn.br)
Intituto Metrópole Digital

- Qual o valor de P ideal para a multiplicação das Matrizes M1 e M2? Sendo justificado através dos experimentos realizados.

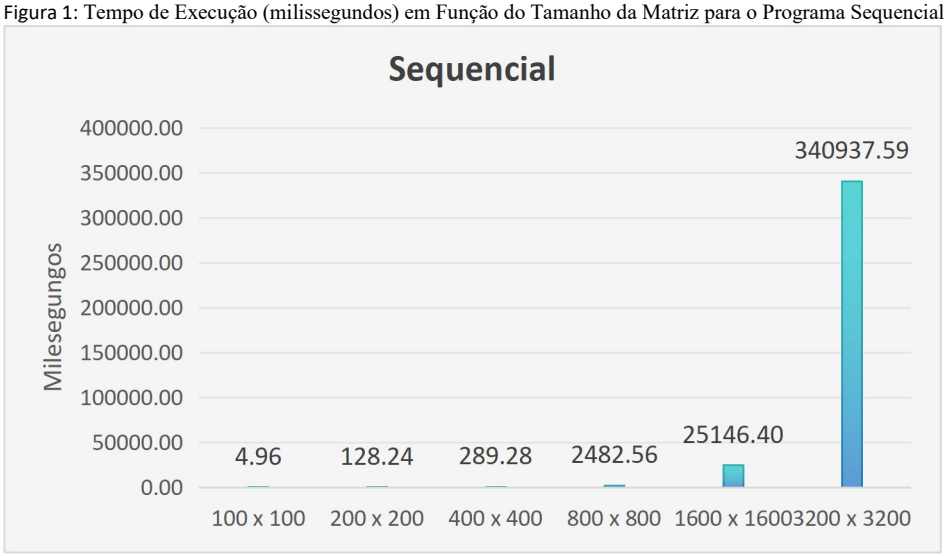
Os testes foram realizados em um notebook DELL, modelo Vostro 3525, equipado com um processador AMD 5500U com seis núcleos físicos(cores), cada um suportando duas threads. Para garantir a precisão das medições, todos os aplicativos foram fechados e o Wi-Fi desativado, evitando interferências externas durante a execução dos experimentos.

Universidade Federal do Rio Grande do Norte
(UFRN)
Lucas Marques dos Santos
(lucas.marques.111@ufrn.edu.br)
Kayo Gonçalves e Silva
(kayo@imd.ufrn.br)
Intituto Metrópole Digital

Resultados e discussões

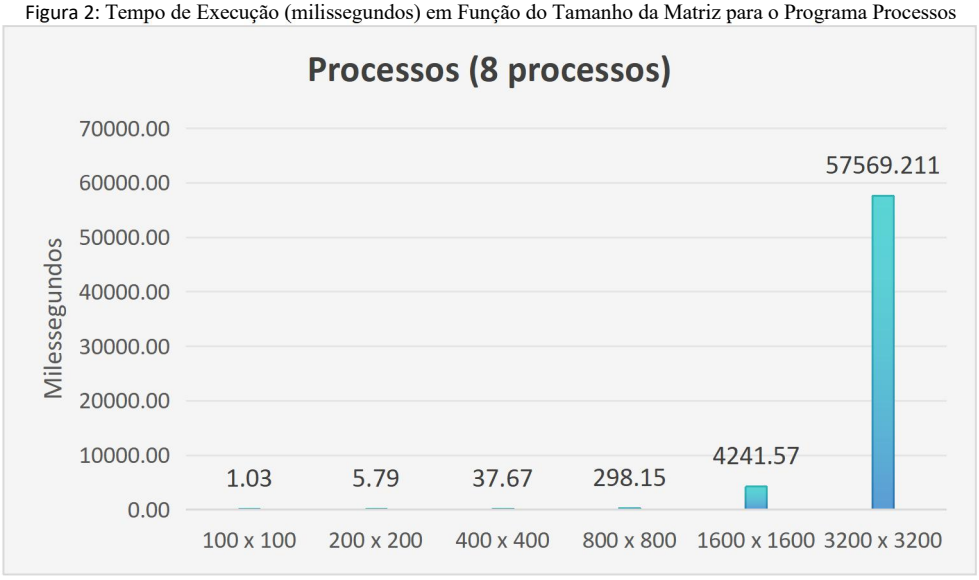
Começando então com o primeiro experimento(E1), os resultados obtidos estão apresentados a seguir para: Sequencial, Processos e Threads, respectivamente.

Figura 1: Tempo de Execução (milissegundos) em Função do Tamanho da Matriz para o Programa Sequencial



Fonte: Autoria própria.

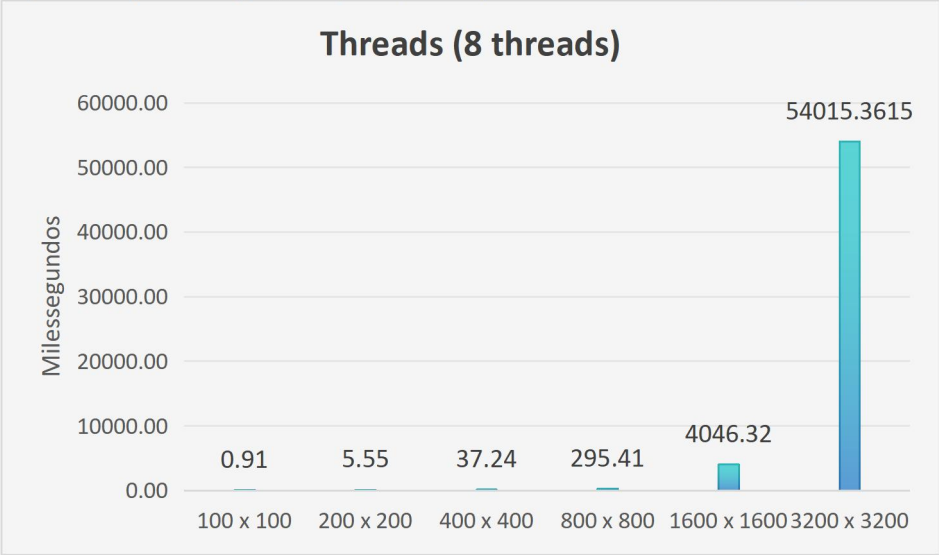
Figura 2: Tempo de Execução (milissegundos) em Função do Tamanho da Matriz para o Programa Processos



Fonte: Autoria própria.

Universidade Federal do Rio Grande do Norte
(UFRN)
Lucas Marques dos Santos
(lucas.marques.111@ufrn.edu.br)
Kayo Gonçalves e Silva
(kayo@imd.ufrn.br)
Intituto Metrópole Digital

Figura 3: Tempo de Execução (milissegundos) em Função do Tamanho da Matriz para o Programa Threads



Fonte: Autoria própria.

Com base nos dados obtidos, podemos ver que a resolução da matriz por meio da divisão em processos e threads apresentou desempenho superior à implementação sequencial. Além disso, as threads foram mais rápidas do que os processos, o que era esperado. Isso se deve ao fato de que, ao criar um novo processo, o sistema precisa duplicar todo o seu espaço de memória, incluindo: código, arquivos abertos, registradores e pilha. Já as threads compartilham o mesmo espaço de memória do processo principal, o que reduz significativamente o custo de criação e comunicação entre elas.[2] Portanto, a utilização de threads tende a ser mais eficiente em termos de desempenho e consumo de recursos.

A divisão de trabalho por meio da paralelização se mostra eficiente, pois, enquanto na implementação sequencial há apenas um processo e, consequentemente, apenas uma thread, responsável por todo o cálculo, na paralelização as tarefas são distribuídas entre múltiplos processos e/ou threads. Isso permite o aproveitamento de todos os núcleos (cores) do processador, resultando em maior eficiência computacional.

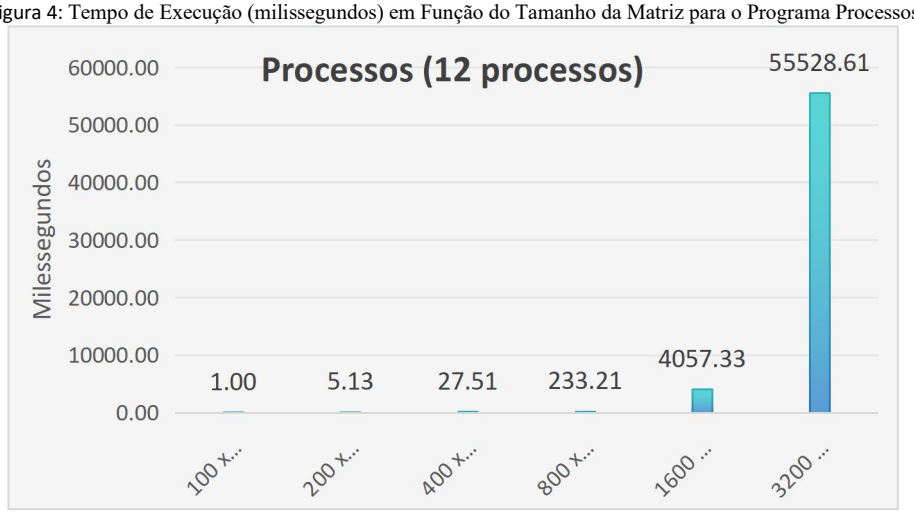
Na matriz de dimensão 100 × 100, observou-se um ganho de aproximadamente 3 milissegundos. Para a matriz 400 × 400, o ganho foi de cerca de 251,61 milissegundos, aproximadamente. Na matriz 1600 × 1600, a redução no tempo de execução foi de aproximadamente 20.905 milissegundos. Já para a maior matriz avaliada, de 3200 × 3200, enquanto a execução sequencial levou quase 6 minutos, a versão paralelizada concluiu a tarefa em menos de 1 minuto. Pode não parecer, mas o ganho de alguns milessegundos é muita coisa, já que o SO quer manter a CPU o menos ocioso possível. Dessa forma, mesmo pequenas reduções no tempo de execução contribuem significativamente para o melhor aproveitamento dos recursos do sistema, especialmente em aplicações que realizam diversas operações computacionalmente intensivas ou que são executadas repetidamente.

Esses resultados indicam que o ganho de desempenho tende a aumentar conforme cresce a complexidade do problema, já que os gráficos de tempo de execução apresentam um comportamento exponencial.

Universidade Federal do Rio Grande do Norte
(UFRN)
Lucas Marques dos Santos
(lucas.marques.111@ufrn.edu.br)
Kayo Gonçalves e Silva
(kayo@imd.ufrn.br)
Intituto Metrópole Digital

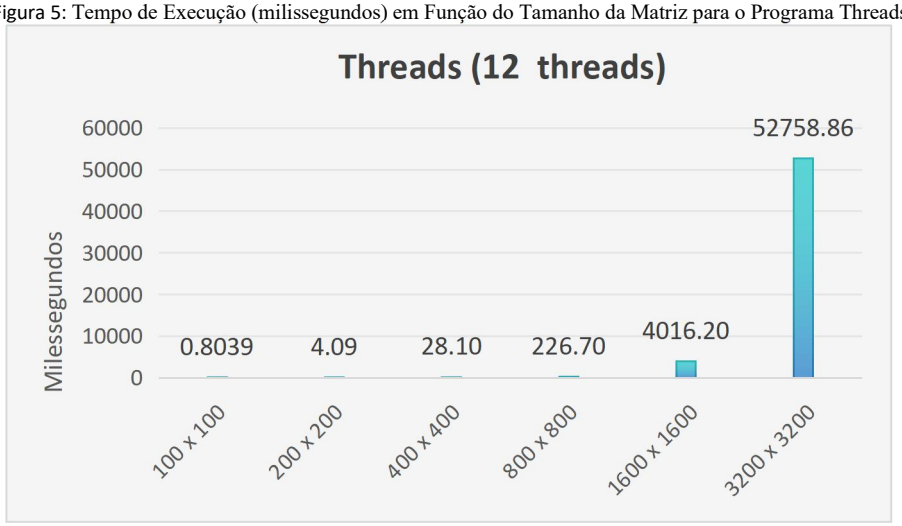
Seguindo agora para o último experimento(E2), aumentamos o valor de P de modo que o número total de processos/threads fosse 12. O resultados obtidos, estão apresentados logo abaixo.

Figura 4: Tempo de Execução (milissegundos) em Função do Tamanho da Matriz para o Programa Processos



Fonte: Autoria própria.

Figura 5: Tempo de Execução (milissegundos) em Função do Tamanho da Matriz para o Programa Threads



Fonte: Autoria própria.

Analisando os gráficos obtidos, percebe-se que, com o aumento do valor de *P* até atingir 12 processos/threads, o tempo de execução da paralelização diminuiu ainda mais. À primeira vista, pode parecer óbvio: “com mais processos/threads, há uma maior divisão do trabalho, o que naturalmente acelera a execução”. No entanto, é importante considerar que o computador utilizado nos testes possui 6 núcleos físicos, sendo que cada um é capaz de executar 2 threads simultaneamente. Isso é possível graças à tecnologia SMT (Simultaneous Multithreading)[4], da AMD, que permite que um único núcleo físico se comporte como dois

Universidade Federal do Rio Grande do Norte
(UFRN)
Lucas Marques dos Santos
(lucas.marques.111@ufrn.edu.br)
Kayo Gonçalves e Silva
(kayo@imd.ufrn.br)
Intituto Metrópole Digital

núcleos lógicos, aproveitando partes que estariam ociosas para executar duas threads ao mesmo tempo. Dessa forma, mesmo com apenas 6 núcleos físicos, o sistema operacional enxerga 12 núcleos lógicos, permitindo uma execução paralela mais eficiente e, consequentemente, uma redução significativa no tempo de processamento.

Os maiores ganhos de desempenho foram observados na multiplicação da matriz 1600 × 1600, em que a versão com processos obteve uma redução de 184 milissegundos no tempo de execução, enquanto a versão com threads apresentou um ganho de 30 milissegundos. Já para a matriz 3200 × 3200, os ganhos foram ainda mais expressivos: 2041 milissegundos para a versão com processos e 1256 milissegundos para a versão com threads.

Podemos observar que a performance não dobrou, o que é esperado pois estamos usando as partes ociosas do núcleo, mas houve sim, uma diminuição no tempo de execução. Desta forma o valor ideal de P, depende de qual tipo da tarefa e do seu computador. Se forem tarefas CPU-bound, ou seja, que passe mais tempo nos cores, que é o que enquadra em nosso caso, o valor de P ideal, para nosso programa, seria (número de colunas de M1 * Números de linhas de M2 / (cores * threads por core). Caso seja tarefas I/O-Bound, você pode usar mais, já que muitas threads ficaram ociosas esperando entrada/saída. É importante lembrar que, no caso de tarefas do tipo CPU-bound, criar um número excessivo de threads pode ser prejudicial. Isso ocorre porque o sistema operacional precisa criar e gerenciar todas essas threads, o que gera overhead, ou seja, um custo adicional de tempo e uso de memória, podendo, em vez de melhorar, reduzir o desempenho da aplicação.

Universidade Federal do Rio Grande do Norte
(UFRN)
Lucas Marques dos Santos
(lucas.marques.111@ufrn.edu.br)
Kayo Gonçalves e Silva
(kayo@imd.ufrn.br)
Intituto Metrópole Digital

Conclusão

Por fim, podemos concluir que sim: threads são a ser mais rápidas que processos. Isso pôde ser observado nos gráficos apresentados anteriormente, validando a teoria discutida em sala de aula. Esse resultado se deve ao fato de que as threads compartilham a memória do processo ao qual pertencem, o que reduz significativamente os custos de criação e comunicação entre elas. Além disso, foi possível perceber os benefícios da divisão de tarefas em múltiplos processos ou threads, resultando em uma execução mais eficiente e com menor tempo de processamento. Portanto, é fundamental verificar as especificações do seu computador a fim de adequar corretamente o número de processos e threads utilizados, otimizando assim o desempenho da aplicação de acordo com a natureza da tarefa.

Universidade Federal do Rio Grande do Norte
(UFRN)
Lucas Marques dos Santos
(lucas.marques.111@ufrn.edu.br)
Kayo Gonçalves e Silva
(kayo@imd.ufrn.br)
Intituto Metrópole Digital

Referências

OLIVEIRA, Rômulo Silva de; CARISSIMI, Alexandre da Silva; TOSCANI, Simão Sirineo. Sistemas operacionais. 4. ed. Porto Alegre: Bookman, 2010. ISBN: 9788577805211.[1]

TANENBAUM, Andrew S. Sistemas operacionais modernos. 3. ed. São Paulo: Prentice Hall, 2009. 653 p. ISBN: 9788576052371.[2]

MARQUES, Lucas. *Sistema_Operacionais-Processos_and_Threads*. 2025. Repositório no GitHub. Disponível em: https://github.com/LukasMarquess/Sistema_Operacionais-Processos_and_Threads. Acesso em: 28 set. 2025.[3]

AMD. *Simultaneous Multithreading: Driving Performance and Efficiency on AMD EPYC CPUs*. 2025. Disponível em: https://www.amd.com/en/blogs/2025/simultaneous-multithreading-driving-performance-a.html. Acesso em: 29 set. 2025[4]