https://jurnal.pustakagalerimandiri.co.id/index.php/pustakadata
DOI: https://doi.org/10.55382/jurnalpustakadata.v5i2.1504

Jurnal Pustaka Data
Pusat Akses Kajian Database, Analisa Teknologi, dan Arsitektur Komputer
Pustaka Galeri Mandiri
Vol. 5 No. 2 (2025) 431 – 437
E ISSN: 2809-7483

Pengaruh *Context Switching* terhadap Efisiensi *CPU* pada Sistem *Multicore*

Muhammad Aditya Dermawan[1], Veronica Aisyah Nabilla[2], Muhammad Ainul Yaqin[3]
Fakultas Sains dan Teknologi, Universitas Islam Negeri Maulana Malik Ibrahim Malang
[1]240605110172@student.uin-malang.ac.id, [2]240605110241@student.uin-malang.ac.id, [3]yaqinov@ti.uin-malang.ac.id

Abstract

Context switching is a fundamental mechanism in modern operating systems that enables the CPU to alternate rapidly between active processes, allowing concurrent execution in multitasking environments. Despite its essential role, context switching introduces computational overhead due to the need to save and restore process states. This study aims to empirically analyze execution efficiency by comparing multithreaded process execution, which involves context switching, with sequential process execution, which conceptually represents the absence of context switching. The experiments were conducted using a Java based simulation on a multicore CPU environment, where multiple CPU bound processes were executed under controlled conditions. The results indicate that the average total execution time for the multithreaded scenario was 909 milliseconds, while the sequential execution scenario required 5791.2 milliseconds. These findings demonstrate that, in a multicore and CPU intensive workload context, the overhead of context switching is outweighed by the benefits of parallel execution. This research provides empirical evidence of the trade-off between context switching overhead and execution efficiency, contributing to a clearer understanding of process management behavior in multitasking systems.

Keywords: context switching, multitasking, multithreading, CPU scheduling, Java simulation

Abstrak

Context switching merupakan mekanisme fundamental dalam sistem operasi modern yang memungkinkan CPU berpindah secara cepat antar proses aktif sehingga mendukung eksekusi paralel dalam lingkungan multitasking. Meskipun berperan penting, context switching menimbulkan overhead komputasi karena CPU harus menyimpan dan memulihkan konteks proses setiap kali terjadi perpindahan. Penelitian ini bertujuan untuk menganalisis efisiensi eksekusi proses dengan membandingkan skenario eksekusi multithreading yang melibatkan context switching dan skenario eksekusi sequential yang secara konseptual merepresentasikan kondisi tanpa perpindahan konteks. Eksperimen dilakukan melalui simulasi berbasis Java pada lingkungan CPU multicore dengan beban kerja bersifat CPU bound. Hasil simulasi menunjukkan bahwa rata-rata total waktu eksekusi pada skenario multithreading adalah 909 milidetik, sedangkan eksekusi sequential memerlukan waktu sebesar 5791,2 milidetik. Temuan ini menunjukkan bahwa pada lingkungan multicore dengan beban komputasi intensif, overhead context switching dapat dikompensasi oleh peningkatan efisiensi eksekusi melalui paralelisme. Penelitian ini memberikan pembuktian empiris melalui simulasi terkontrol mengenai trade-off antara overhead context switching dan efisiensi eksekusi dalam pengelolaan proses sistem operasi.

Kata kunci: context switching, multitasking, multithreading, penjadwalan CPU, simulasi Java

Submitted : 08-12-2025 | Reviewed : 14-12-2025 | Accepted : 19-12-2025

1. Pendahuluan

Sistem operasi merupakan komponen dasar yang bertanggung jawab mengelola perangkat keras serta menyediakan lingkungan bagi aplikasi untuk dapat berjalan dengan baik. [1], [2], [3] Salah satu fungsi utama yang menentukan kinerja sistem operasi adalah kemampuan dalam mengatur eksekusi proses. Proses didefinisikan sebagai program yang sedang berjalan dan memiliki konteks tersendiri, seperti nilai register, counter instruksi, serta informasi status eksekusi. [2], [3] Ketika terdapat lebih dari satu proses aktif, sistem operasi harus memastikan bahwa setiap proses memperoleh kesempatan untuk menggunakan CPU secara adil dan efisien. Di sinilah context switching menjadi mekanisme penting karena memungkinkan CPU berpindah dari satu proses ke proses lainnya.

Context switching membuat multitasking modern dapat terjadi. [1], [2], [7] Meskipun CPU hanya dapat mengerjakan satu instruksi pada satu waktu, perpindahan konteks yang dilakukan sangat cepat sehingga pengguna merasakan seolah olah banyak aplikasi berjalan secara paralel. Mekanisme ini menghasilkan ilusi paralelisme pada sistem berbasis satu inti dan meningkatkan kapasitas pemrosesan pada sistem dengan banyak inti. [1], [2], [21] Namun, perpindahan konteks tidak terjadi tanpa biaya. CPU perlu menyimpan status proses yang ditinggalkan dan memulihkan status proses yang akan dijalankan. [4], [18] Hal ini dikenal sebagai overhead context switching dan dapat mempengaruhi performa keseluruhan sistem.

Efek dari overhead ini menjadi perhatian utama dalam penelitian performa sistem operasi. [3], [19] Ketika jumlah proses bertambah atau ketika proses sering melakukan operasi yang memicu pergantian konteks, waktu yang dihabiskan untuk mengelola context switching akan meningkat. [3], [16] Kondisi ini dapat menurunkan throughput, meningkatkan total waktu eksekusi, dan mengurangi efisiensi CPU. Dalam beberapa kasus yang ekstrem, sistem dapat menjadi kurang responsif apabila frekuensi pergantian konteks terlalu tinggi. Meski demikian, tanpa context switching, sistem tidak dapat mendukung multitasking yang merupakan kebutuhan utama komputer modern. [1], [2]

Literatur menyebutkan bahwa perbandingan langsung antara sistem dengan dan tanpa context switching masih jarang dibahas dalam penelitian berbasis eksperimental. [1], [2], [3], [19], [23] Sebagian besar kajian berfokus pada teori manajemen proses, algoritma penjadwalan, atau analisis kinerja tanpa melibatkan simulasi aktual pada tingkat implementasi. Padahal pendekatan berbasis simulasi dapat memberikan gambaran kuantitatif yang lebih nyata mengenai pengaruh context switching terhadap performa eksekusi proses.

Penelitian ini berupaya memberikan kontribusi pada aspek tersebut melalui pembuatan simulasi berbasis Java yang menjalankan dua skenario berbeda. Skenario pertama menggunakan multithreading sehingga CPU memungkinkan terjadinya context switching ketika beberapa thread berjalan secara bersamaan. Skenario kedua menjalankan proses secara sequential tanpa perpindahan konteks sehingga CPU menyelesaikan satu proses sepenuhnya sebelum beralih ke proses berikutnya. Melalui pengukuran waktu eksekusi dari kedua skenario tersebut, penelitian ini bertujuan memberikan pemahaman yang lebih nyata mengenai peran context switching dalam efisiensi eksekusi proses.

Tujuan penelitian ini adalah untuk menganalisis dan membandingkan efisiensi eksekusi proses antara skenario multithreading yang melibatkan context switching dan skenario eksekusi sequential tanpa perpindahan konteks melalui simulasi terkontrol berbasis Java. Secara khusus, penelitian ini bertujuan mengukur dampak context switching terhadap total waktu eksekusi proses pada beban kerja bersifat CPU bound, serta memberikan pembuktian empiris mengenai trade-off antara overhead context switching dan manfaat paralelisme dalam sistem operasi multitasking.

2. Metode Penelitian

Penelitian ini dilakukan menggunakan pendekatan eksperimen langsung yang memanfaatkan dua model simulasi eksekusi proses pada bahasa pemrograman Java. [5], [6], [14] Seluruh eksperimen dirancang untuk mengukur dan membandingkan performa eksekusi proses pada dua kondisi utama, yaitu skenario eksekusi multithreading yang memungkinkan terjadinya context switching dan skenario eksekusi sequential yang secara konseptual merepresentasikan kondisi tanpa perpindahan konteks. Pendekatan ini dipilih karena mampu menghasilkan data empiris yang menggambarkan perbedaan waktu eksekusi secara kuantitatif pada beban kerja yang sama.

Proses penelitian dimulai dengan pengembangan dua program Java yang memiliki struktur tugas komputasi identik, yaitu operasi perkalian matriks dengan ukuran tertentu pada setiap proses. Operasi perkalian matriks dipilih karena termasuk komputasi bersifat CPU bound dan memiliki kompleksitas yang cukup tinggi, sehingga mampu merefleksikan kinerja CPU secara konsisten tanpa ketergantungan signifikan pada operasi input output. [8], [9], [15] Pada kedua program, setiap proses diberikan ukuran matriks yang sama dan waktu kedatangan yang dihasilkan secara acak

Submitted : 08-12-2025 | Reviewed : 14-12-2025 | Accepted : 19-12-2025

untuk mensimulasikan kondisi kedatangan proses yang tidak seragam seperti pada sistem nyata. [8], [16]

Program pertama dirancang menggunakan mekanisme multithreading, di mana seluruh proses dijalankan sebagai thread yang aktif secara bersamaan. [5], [6], [7] Dalam kondisi ini, sistem operasi dan Java Virtual Machine memiliki kebebasan untuk menjadwalkan eksekusi thread sehingga memungkinkan terjadinya context switching. Perlu ditekankan bahwa context switching yang diamati dalam penelitian ini merupakan context switching pada level thread Java, yang melibatkan penjadwalan oleh JVM dan sistem operasi, dan bukan context switching kernel secara murni. Meskipun demikian, mekanisme ini tetap relevan untuk merepresentasikan perilaku multitasking pada sistem operasi modern, khususnya pada aplikasi berbasis user level thread. [1], [2], [3], [17].

Program kedua menjalankan proses dengan cara sequential. [1], [2], [12] Pada skenario ini, setiap proses tetap diimplementasikan sebagai thread, tetapi proses berikutnya hanya dijalankan setelah proses sebelumnya selesai sepenuhnya. Dengan demikian, hanya terdapat satu thread aktif pada satu waktu, sehingga tidak terjadi context switching antar proses selama eksekusi berlangsung. Skenario ini digunakan sebagai pembanding untuk merepresentasikan kondisi eksekusi tanpa perpindahan konteks, meskipun secara konseptual kondisi tersebut bukan merupakan karakteristik sistem operasi nyata, melainkan pendekatan simulatif untuk mengisolasi biaya komputasi murni.

Eksperimen dijalankan pada lingkungan pengujian dengan spesifikasi yang dikontrol. Seluruh pengujian dilakukan pada sistem dengan CPU multicore, sistem operasi berbasis desktop, serta Java Virtual Machine versi modern yang mendukung optimasi Just In Time compilation. Kebijakan penjadwalan thread sepenuhnya diserahkan kepada JVM dan sistem operasi tanpa modifikasi eksplisit, sehingga hasil yang diperoleh mencerminkan perilaku penjadwalan standar pada lingkungan Java. Penelitian ini tidak membedakan secara eksplisit kebijakan scheduling internal JVM, namun mengasumsikan bahwa mekanisme tersebut berjalan secara konsisten selama seluruh percobaan. Untuk meminimalkan pengaruh optimasi awal JVM, setiap skenario dijalankan sebanyak lima kali, dan hasil pengukuran diambil setelah program memasuki fase eksekusi stabil. Pendekatan ini bertujuan mengurangi bias yang mungkin muncul akibat proses warm up JVM dan optimasi Just In Time compilation pada eksekusi awal. Dari setiap percobaan, dua parameter utama dicatat, yaitu total waktu eksekusi dan burst time masing masing proses. Total waktu eksekusi mencerminkan waktu keseluruhan yang dibutuhkan sejak program dimulai hingga seluruh proses selesai, sedangkan burst time menggambarkan waktu komputasi aktual yang digunakan oleh CPU untuk menyelesaikan tugas perkalian matriks.

Data hasil eksperimen dicatat dalam lembar kerja spreadsheet untuk memudahkan proses pengolahan dan analisis statistik. Nilai rata rata dan standar deviasi dari masing masing parameter dihitung untuk memperoleh gambaran performa yang lebih stabil dan reliabel. Data tersebut kemudian digunakan untuk menyusun tabel dan grafik perbandingan antara skenario multithreading dan skenario sequential.

Tahap akhir penelitian adalah analisis komparatif antara kedua skenario. Analisis dilakukan dengan membandingkan total waktu eksekusi, burst time, serta variabilitas hasil pengukuran. Hasil eksperimen dianalisis secara kuantitatif dan diinterpretasikan secara kualitatif dengan mengacu pada teori sistem operasi mengenai overhead context switching dan pemanfaatan paralelisme pada sistem multicore. Dengan pendekatan ini, penelitian diharapkan mampu memberikan pemahaman empiris mengenai trade off antara biaya perpindahan konteks dan peningkatan efisiensi eksekusi proses.

3. Hasil dan Pembahasan

3.1. Hasil Simulasi

Percobaan dilakukan menggunakan program Java yang dijalankan melalui NetBeans. Program mensimulasikan dua skenario:

1. Dengan Context Switching (multithreading)
2. Tanpa Context Switching (sequential)

Setiap skenario diuji menggunakan ukuran matriks yang berbeda (misalnya 90×90, 120×120, 150×150, dan seterusnya). Untuk setiap ukuran matriks, percobaan dijalankan 5 kali untuk memperoleh nilai rata-rata Total Waktu Eksekusi.

Tabel 1. Hasil Simulasi dengan Context Switching

Ukuran Matrix	Waktu Eksekusi (ms) ($\underline{x}$, SD)	Burst Time (ms) ($\underline{x}$, SD)
90	(25, 1)	(54.2, 6.4)
120	(24.4, 1.1)	(53.6, 3.2)
150	(28, 1.4)	(71, 7.4)
180	(28.4, 1.9)	(73.8, 8)
210	(30.4, 1.3)	(83.4, 6.1)
250	(34.8, 2)	(109.4, 12.1)
270	(44, 3.8)	(151.8, 16.6)
300	(44, 3)	(159.6, 14.3)
330	(51, 3.2)	(194.4, 16.3)
360	(61.2, 1.6)	(246.6, 16.1)

Berdasarkan tabel 1, total waktu eksekusi pada skenario dengan context switching berada pada

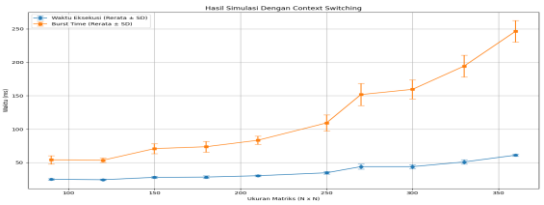
kisaran 24-63 ms, tergantung ukuran matriks yang diuji. Nilai ini menunjukkan bahwa ketika program dijalankan secara multithreaded, CPU dapat membagi waktu eksekusi ke beberapa proses secara bergantian sehingga beban kerja dapat tersebar dengan lebih efisien. Burst time tiap proses juga berada dalam rentang sekitar 52-250 ms sesuai ukuran matriks, dan variasi antar-run relatif kecil, menandakan bahwa perpindahan konteks tidak menimbulkan penalti signifikan terhadap performa. Secara keseluruhan, mekanisme context switching membuat waktu eksekusi tetap stabil walaupun ukuran matriks bertambah, karena CPU dapat melakukan interleaving eksekusi antar-thread.

Tabel 2. Hasil Simulasi Tanpa Context Switching

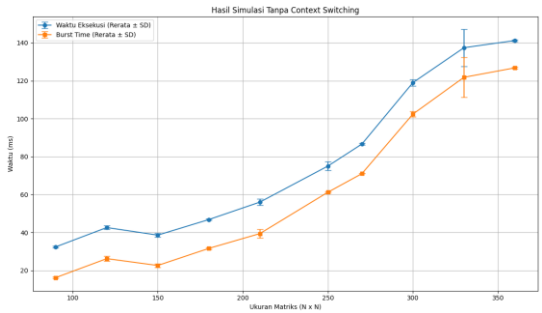
Ukuran Matrix	Waktu Eksekusi (ms) ($\underline{x}$, SD)	Burst Time (ms) ($\underline{x}$, SD)
90	(32.4, 0.5)	(16.2, 0.4)
120	(42.6, 0.9)	(26.2, 1.3)
150	(38.6, 0.9)	(22.6, 0.9)
180	(46.8, 0.4)	(31.6, 0.5)
210	(56, 1.7)	(39.4, 2.2)
250	(75, 2.3)	(61.2, 0.4)
270	(86.6, 0.5)	(71, 0)
300	(119, 1.7)	(102.4, 1.5)
330	(137.4, 9.8)	(121.8, 10.5)
360	(141.2, 0.4)	(126.8, 0.4)

Pada skenario tanpa context switching di tabel 2, total waktu eksekusi tercatat berada pada kisaran 32-142 ms, lebih tinggi dibandingkan skenario dengan context switching terutama pada ukuran matriks yang lebih besar. Hal ini terjadi karena setiap proses dijalankan secara murni serial tanpa pembagian waktu oleh CPU, sehingga seluruh burst time ditumpuk dan dijalankan satu per satu. Burst time tiap proses memang terlihat stabil, namun tidak adanya interleaving menyebabkan total waktu meningkat seiring bertambahnya ukuran matriks. Dampaknya terlihat jelas pada ukuran yang lebih besar, di mana waktu eksekusi bisa melonjak hingga lebih dari 140 ms, jauh di atas performa skenario dengan context switching. Ini menunjukkan bahwa tanpa mekanisme peralihan konteks, CPU tidak dapat memanfaatkan potensi paralelismenya secara optimal sehingga efisiensi eksekusi menurun.

Untuk memberikan gambaran visual mengenai perbedaan kedua skenario, disediakan grafik 2 dan 3.



Gambar 1. Grafik Hasil Simulasi Tanpa Context Switching



Gambar 2. Grafik Hasil Simulasi Tanpa Context Switching

3.2. Analisis Perbandingan

Perbedaan performa antara kedua skenario terlihat sangat jelas ketika hasil simulasi dibandingkan. Pada skenario dengan context switching, total waktu eksekusi berada pada rentang 24-63 ms, menunjukkan bahwa mekanisme pembagian waktu CPU antar-thread mampu mempercepat penyelesaian proses meskipun ukuran matriks bertambah. [1], [5], [7], [20] Hal ini terjadi karena eksekusi beberapa thread dapat saling tumpang-tindih (interleaved), sehingga CPU tidak pernah benar-benar menganggur. [5], [6], [7], [20] Sebaliknya, pada skenario tanpa context switching, total waktu eksekusi meningkat jauh lebih besar dan dapat mencapai lebih dari 140 ms pada ukuran matriks yang lebih besar. Karena seluruh proses berjalan secara serial, waktu eksekusi total menjadi akumulasi penuh dari masing-masing burst time, tanpa adanya peluang interleaving yang dapat mengurangi durasi penyelesaian. Kenaikan waktu pada skenario ini juga jauh lebih tajam dibandingkan skenario dengan context switching, sehingga gap performa makin terlihat ketika beban komputasi bertambah besar.

Dari sisi kestabilan, kedua skenario sama-sama memiliki burst time yang cenderung konsisten, namun variasi total waktu eksekusi pada skenario tanpa context switching lebih besar. [8], [9], [16], [3], [19], [23] Hal ini menunjukkan bahwa ketika proses berjalan sepenuhnya serial, perbedaan kecil pada kondisi run tertentu dapat berdampak langsung pada total waktu eksekusi. Sebaliknya, pada skenario dengan context switching, standar deviasi total waktu eksekusi lebih stabil karena adanya fleksibilitas CPU dalam mengatur eksekusi beberapa thread sekaligus. Mekanisme ini membuat perubahan kecil pada satu proses tidak terlalu mempengaruhi keseluruhan durasi penyelesaian.

Secara keseluruhan, context switching terbukti memberikan keuntungan paralelisme yang signifikan meskipun tetap memiliki overhead. [1], [5], [7], [21] Overhead tersebut terlihat dari burst time yang sedikit lebih bervariasi dibandingkan skenario tanpa context switching, namun peningkatan efisiensi total waktunya jauh lebih besar daripada biaya tambahan tersebut. [3], [4],

Submitted : 08-12-2025 | Reviewed : 14-12-2025 | Accepted : 19-12-2025

[18] Hasil ini memperlihatkan bahwa pada beban komputasi modern terutama operasi *CPU-bound* seperti perkalian matriks context switching bukan hanya tidak merugikan, tetapi justru menjadi mekanisme penting untuk mencapai efisiensi eksekusi yang lebih baik.

3.3. Pembahasan

Hasil simulasi menunjukkan perbedaan karakteristik kinerja yang sangat jelas antara eksekusi proses dengan context switching dan tanpa context switching. Context switching memungkinkan CPU menjalankan banyak thread secara bersamaan, sedangkan tanpa context switching proses dijalankan secara serial. [1], [5], [6], [7] Perbedaan mendasar ini menghasilkan gap performa yang besar, sebagaimana terlihat dari total waktu eksekusi masing-masing skenario.

Pada skenario dengan context switching, rata-rata total waktu eksekusi hanya mencapai 909 ms. Angka ini jauh lebih rendah dibandingkan skenario tanpa context switching yang mencapai 5791 ms. Perbedaan ini menunjukkan bahwa distribusi eksekusi antar thread memberikan manfaat besar dalam mempercepat penyelesaian keseluruhan tugas. Ketika banyak thread aktif, CPU dapat melakukan interleaving eksekusi secara cepat, memanfaatkan waktu idle pada thread yang menunggu arrival time, dan memaksimalkan penggunaan inti prosesor (cores). Dengan kata lain, context switching memperbolehkan proses berjalan secara overlapped, sehingga beban komputasi berat seperti perkalian matriks dapat terselesaikan lebih cepat secara keseluruhan.

Meskipun demikian, context switching tidak terlepas dari overhead. [3], [4], [18], [19] Setiap perpindahan konteks mengharuskan CPU menyimpan state proses yang sedang berjalan dan memulihkan state proses lain yang akan dilanjutkan. Secara teori, semakin sering context switching terjadi, semakin banyak waktu yang terbuang untuk operasi administratif tersebut. Hal ini biasanya terlihat pada burst time, karena burst time mewakili durasi eksekusi murni dari proses sebelum dan sesudah interupsi. Pada simulasi ini, burst time rata-rata pada skenario context switching sedikit lebih rendah tetapi memiliki variasi yang lebih besar dibandingkan skenario tanpa context switching. Hal tersebut wajar karena proses yang berjalan secara paralel sering kali terpengaruh oleh jadwal perpindahan CPU antar thread, sehingga durasi aktual penyelesaian suatu proses bisa bervariasi.

Namun, meskipun terdapat overhead tersebut, hasil menunjukkan bahwa dampaknya tidak signifikan. [1], [7], [20] Burst time rata-rata pada skenario context switching masih berada pada rentang yang hampir sama dengan skenario tanpa context switching. Standar deviasi burst time pada kedua skenario juga sangat kecil (sekitar 1 ms), menunjukkan bahwa perbedaan antar proses terkait penalti switching tidak terlalu besar. Dengan demikian, dapat dikatakan bahwa biaya context switching tidak mengganggu performa proses secara individual dalam tingkat yang berarti. Faktor yang lebih dominan justru terletak pada kemampuan paralelisme yang membuat total waktu penyelesaian jauh lebih cepat.

Sebaliknya, skenario tanpa context switching menunjukkan karakteristik yang berbeda. [1], [2], [12] Pada pendekatan ini, proses dijalankan satu per satu berdasarkan urutan pemanggilan *join()*, sehingga tidak ada dua proses yang berjalan bersamaan. Walaupun pendekatan ini sepenuhnya menghindari overhead switching, total waktu eksekusi menjadi sangat tinggi. Pada sistem tanpa context switching, CPU hanya dapat menjalankan satu proses pada satu waktu, sehingga waktu penyelesaian setiap proses menumpuk secara linear. Dengan ukuran matriks yang besar, penjumlahan burst time dari seluruh proses menghasilkan waktu penyelesaian total yang jauh lebih panjang.

Skenario tanpa context switching memang menghasilkan burst time yang lebih stabil karena proses tidak pernah terinterupsi. Hal ini menyebabkan waktu eksekusi individual tiap proses tampak lebih konsisten. Namun, stabilitas ini tidak memberikan keuntungan terhadap efisiensi keseluruhan. Justru, stabilitas ini menjadi bukti bahwa tanpa mekanisme paralelisme, performa sistem akan terjebak pada keterbatasan eksekusi sekuensial.

Perbedaan standar deviasi pada total waktu eksekusi kedua skenario juga memperkuat karakteristik ini. [8], [9], [16] Pada skenario context switching, standar deviasi total eksekusi hanya sekitar 88 ms, menunjukkan konsistensi performa eksekusi paralel. Sebaliknya, tanpa context switching, standar deviasi mencapai 624 ms, menandakan bahwa variasi durasi setiap proses sangat berpengaruh terhadap total waktu eksekusi. Karena proses dijalankan serial, perbedaan burst time sedikit saja dapat memperbesar total waktu penyelesaian secara drastis.

Temuan ini menegaskan peran penting context switching dalam sistem operasi modern. [1], [2], [3], [7], [21] Mekanisme ini memungkinkan CPU mengelola berbagai proses secara efisien melalui penjadwalan yang adaptif. Walaupun ada biaya penyimpanan dan pemulihan state proses, biaya tersebut merupakan trade-off yang kecil dibandingkan potensi peningkatan throughput yang sangat signifikan. Pada sistem multicore, manfaat context switching bahkan lebih terasa karena proses dapat benar-benar berjalan secara paralel pada inti

Submitted : 08-12-2025 | Reviewed : 14-12-2025 | Accepted : 19-12-2025

yang berbeda, bukan hanya di interleaving dalam satu inti.

Secara keseluruhan, penelitian ini memperlihatkan bahwa tanpa context switching, sistem akan mengalami penurunan efisiensi yang sangat besar, terutama ketika menangani beban kerja yang berulang dan berat seperti komputasi matriks. [1], [2], [3], [20] Hal tersebut menegaskan bahwa context switching merupakan komponen fundamental dalam desain sistem operasi, bukan sekadar fitur tambahan.

Dengan demikian, dapat disimpulkan bahwa context switching memberikan keuntungan substansial dalam hal efisiensi waktu eksekusi total, pemanfaatan sumber daya CPU, kemampuan menangani banyak proses secara paralel, dan konsistensi performa. Meskipun mengandung sedikit overhead, biaya tersebut jauh lebih kecil dibandingkan manfaat yang diberikan.

4. Kesimpulan

Berdasarkan hasil simulasi yang dilakukan menggunakan program Java di NetBeans, dapat disimpulkan bahwa mekanisme context switching memberikan peningkatan efisiensi yang signifikan terhadap waktu eksekusi proses. Skenario dengan context switching mampu menyelesaikan tugas dalam rentang 24-63 ms, jauh lebih cepat dibandingkan skenario tanpa context switching yang dapat mencapai lebih dari 140 ms pada ukuran matriks besar. Perpindahan konteks memang memiliki sedikit overhead, namun biaya tersebut tidak sebanding dengan keuntungan paralelisme yang diperoleh dari eksekusi multithreaded. Hal ini menunjukkan bahwa context switching berperan penting dalam meningkatkan performa sistem, terutama pada beban komputasi *CPU-bound* seperti perkalian matriks. Secara keseluruhan, penelitian ini memperlihatkan bahwa eksekusi multithreaded memberikan hasil yang lebih efisien dan stabil dibandingkan eksekusi sequential.

Untuk penelitian berikutnya, pengujian disarankan dilakukan pada lingkungan sistem operasi nyata agar perilaku scheduler, kebijakan kernel, dan karakteristik hardware multicore dapat diamati secara lebih komprehensif. Selain itu, beban komputasi yang digunakan dapat diperluas, misalnya dengan menambahkan skenario *I/O-bound* atau kombinasi beban campuran, sehingga dampak context switching dapat dianalisis dalam kondisi yang lebih beragam dan mendekati penggunaan dunia nyata.

Daftar Rujukan

[1] A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts, Hoboken: Wiley, 2018.

[2] A. S. Tanenbaum and H. Bos, Modern Operating Systems, Boston: Pearson, 2015.

[3] W. Stallings, Operating Systems: Internals and Design Principles, Boston: Pearson, 2018.

[4] D. P. Bovet and M. Cesati, Understanding the Linux Kernel, Sebastopol: O'Reilly Media , 2005.

[5] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes and D. Lea, Java Concurrency in Practice, Boston: Addison-Wesley, 2006.

[6] D. Lea, Concurrent Programming in Java: Design Principles and Patterns, Boston: Addison-Wesley, 2000.

[7] M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, San Francisco : Morgan Kaufmann, 2012.

[8] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, San Francisco: Morgan Kaufmann, 2017.

[9] P. Pacheco, An Introduction to Parallel Programming, Burlington: Morgan Kaufmann, 2011.

[10] Intel Corporation, Intel 64 and IA-32 Architectures Optimization Reference Manual, Santa Clara: Intel, 2022.

[11] J. Ousterhout, "Why Threads Are a Bad Idea (for Most Purposes)," *USENIX Annual Technical Conference,* p. 1-10, 1996.

[12] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal,* p. 1-4, 2005.

[13] B. McHale, Effective Multithreading and Performance Tuning in Java, Birmingham: Packt Publishing, 2018.

[14] Oracle Corporation, "Java Platform Standard Edition Documentation," Oracle Corporation, January 2024. [Online]. Available: https://docs.oracle.com/javase . [Accessed 24 November 2025].

[15] B. Wilkinson and M. Allen, Parallel Programming: Techniques and Applications, Boston: Pearson, 2005.

[16] D. G. Feitelson, Workload Characterization for Computer Systems Performance Evaluation, New York: Springer, 2015.

[17] Linux Kernel Organization, "Linux Kernel Scheduler Documentation," Linux Kernel Org, 2010.

[18] U. Drepper, "What Every Programmer Should Know About Memory," Red Hat, 2007.

[19] P. Manoj dan A. Kumar, "Performance Impact of Context Switching in Multicore Systems," *International Journal of Computer Applications,* vol. 178, no. 7, p. 12-18, 2019.

[20] P. E. McKenney, Is Parallel Programming Hard, And If So, What Can You Do About It?, Linux Foundation, 2020.

[21] M. J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE,* vol. 60, no. 2, p. 190-232, 1972.

[22] W.-m. Hwu, GPU Computing Gems, Amsterdam: Morgan Kaufmann , 2011.

Submitted : 08-12-2025 | Reviewed : 14-12-2025 | Accepted : 19-12-2025

[23] 2011, "Scheduling Mechanisms in Multicore Processors," *Journal of Computer Systems Engineering,* vol. 15, no. 4, p. 55-67, 2020.

Submitted : 08-12-2025 | Reviewed : 14-12-2025 | Accepted : 19-12-2025