

Context Switching pada Preemptive Multitasking: Penjelasan Alur Kerja dan Dampaknya terhadap Kinerja CPU

Christinauli Purba^{1*}, Josua Simangunsong^{2*},

Kezia Nindi Oktaviana Simorangkir^{3*}, Indra Gunawan^{4*},

¹ Program Studi Teknik Informatika, Sekolah Tinggi Manajemen Informatika dan Komputer Tunas Bangsa Pematangsiantar, Indonesia

^{1*}chrischristinauli@gmail.com · ^{2*}josuasimangunsong269@gmail.com · ^{3*}nindikezia10@gmail.com · ^{4*}indra@amiktunasbangsa.ac.id

Abstrak

Context switching memungkinkan satu prosesor melayani banyak proses secara bergantian pada sistem operasi *preemptive*. Mekanisme ini tidak gratis, sebab selama perpindahan berlangsung CPU tidak mengerjakan proses pengguna. Artikel ini bertujuan menjelaskan alur kerja *context switching* pada *preemptive multitasking* dan menelaah dampaknya terhadap kinerja CPU. Metode yang digunakan adalah kajian pustaka terhadap buku teks sistem operasi, artikel jurnal terbitan 2021–2026, makalah pengukuran biaya *context switch*, dan dokumentasi kernel Linux, ditambah perhitungan ilustratif berdasarkan parameter dari literatur. Hasil kajian menunjukkan bahwa perpindahan proses berlangsung dalam enam tahap, mulai dari *timer interrupt* sampai proses berikutnya kembali ke *user mode*, dan biayanya terbagi menjadi biaya langsung serta biaya tidak langsung akibat *cache* dan TLB. Perhitungan ilustratif dengan biaya langsung 3,4 μ s memberi fraksi *overhead* sekitar 0,03% pada *time quantum* 10 ms, tetapi sekitar 25% pada *time quantum* 10 μ s. Data simulasi penelitian terdahulu juga memperlihatkan bahwa *time quantum* 15 ms menghasilkan 17 *context switch*, sedangkan 25 ms menghasilkan 10. Pemilihan *time quantum* karena itu perlu menimbang responsivitas terhadap *overhead* yang ditanggung CPU.

Kata Kunci: *context switching*; *preemptive multitasking*; *time quantum*; *overhead* CPU; sistem operasi

PENDAHULUAN

Sistem operasi modern memberi kesan bahwa banyak program berjalan bersamaan, padahal satu inti prosesor pada satu saat hanya mengeksekusi satu aliran instruksi. Kesan itu lahir dari pergantian proses yang berlangsung sangat cepat, dan pergantian tersebut disebut *context switching* (Silberschatz et al., 2018). Pada sistem *preemptive*, sistem operasi tidak menunggu proses melepaskan CPU dengan sukarela. Perangkat *timer* memicu *interrupt* secara berkala, *kernel* mengambil alih kendali, lalu *scheduler* menentukan proses mana yang berhak melanjutkan eksekusi (Arpaci-Dusseau & Arpaci-Dusseau, 2018). Cara ini menjaga sistem tetap tanggap, termasuk ketika ada proses yang berjalan sangat lama atau terjebak dalam perulangan tanpa akhir.

Pergantian itu memakan waktu. Selama *kernel* menyimpan keadaan proses lama dan memuat keadaan proses baru, tidak ada instruksi milik pengguna yang dikerjakan, sehingga waktu tersebut murni *overhead* (Silberschatz et al., 2018). Besarnya tidak ditentukan oleh jumlah register yang disalin saja. Setelah proses baru berjalan, *cache* dan TLB masih berisi data milik proses sebelumnya, dan proses baru harus mengisinya kembali lewat sejumlah *miss* (David et al., 2007; Li et al., 2007). Dalam banyak pembahasan di kelas, hal ini disederhanakan menjadi asumsi bahwa *context switch* tidak membutuhkan waktu, atau hanya dicatat sebagai jumlah perpindahan. Akibatnya *context switching* lebih sering dipahami sebagai angka dalam tabel daripada sebagai rangkaian peristiwa di tingkat *kernel* yang punya harga.

Penelitian terkait dalam lima tahun terakhir sebagian besar berfokus pada algoritma penjadwalan. Putra dan Purnomo (2021) menganalisis algoritma Round Robin pada lima proses dengan dua nilai *time quantum* dan memperoleh 10 *context switch* pada *time quantum* 25 ms serta 17 pada 15 ms. Omar et al. (2021) membandingkan sepuluh algoritma penjadwalan berdasarkan kinerja, kompleksitas, dan waktu tunggu rata-rata. Abdelhafiz (2021) dan Sakshi et al. (2022) mengusulkan penentuan *time quantum* dinamis dari median serta rata-rata *burst time*, dengan jumlah *context switch* dan waktu tunggu sebagai pembanding. Hajjar et al. (2024) menjalankan simulasi berbasis skenario untuk melihat pengaruh ukuran *time quantum* dinamis terhadap waktu respons dan *overhead context switching*. Widiarto et al. (2024) membandingkan Round Robin dengan Priority Preemptive, sedangkan Ramadan et al. (2026) menemukan bahwa Round Robin dengan *time quantum* 3 membagi eksekusi lebih adil daripada FCFS tetapi menambah jumlah *context switching*. Ringkasan penelitian-penelitian tersebut disajikan pada Tabel 1.

Tabel 1. Ringkasan penelitian terkait 2021–2026 dan celah yang ditemukan

Penelitian	Fokus utama	Kaitan dengan <i>context switching</i>	Celah
Putra & Purnomo (2021)	Round Robin, dua nilai <i>time quantum</i> , lima proses	10 vs 17 <i>context switch</i>	Waktu tiap perpindahan tidak dimodelkan
Omar et al. (2021)	Perbandingan sepuluh algoritma penjadwalan	Waktu tunggu rata-rata sebagai ukuran utama	Alur dan biaya perpindahan bukan fokus

Penelitian	Fokus utama	Kaitan dengan <i>context switching</i>	Celah
Abdelhafiz (2021)	<i>Time quantum</i> dari median <i>burst time</i>	Jumlah <i>context switch</i> sebagai metrik	Berupa hitungan, bukan biaya waktu
Sakshi et al. (2022)	<i>Time quantum</i> dinamis (median dan rata-rata)	<i>Context switch</i> yang tinggi memengaruhi kinerja sistem	Komponen biaya tidak diuraikan
Hajjar et al. (2024)	Simulasi FCFS, RR, SJF berbasis skenario	Respons dan <i>overhead context switching</i>	Hasil simulasi, tanpa penjelasan mekanisme
Widiarto et al. (2024)	Round Robin vs Priority <i>Preemptive</i>	Perpindahan yang sering memperpanjang waktu tunggu	Mekanisme internal tidak dibahas
Ramadan et al. (2026)	FCFS vs Round Robin, sepuluh proses	Round Robin menambah jumlah <i>context switching</i>	Biaya per perpindahan tidak diukur

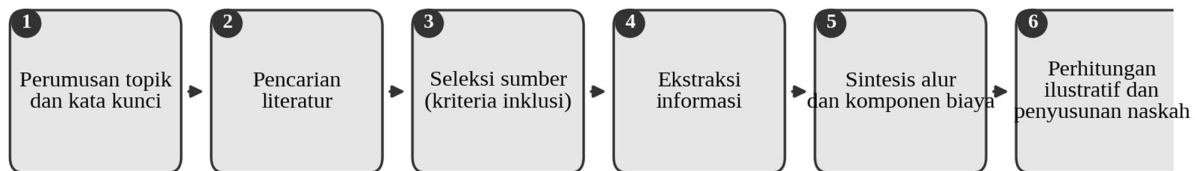
Dari Tabel 1 terlihat pola yang sama: *context switching* diperlakukan sebagai hitungan, sedangkan penjelasan tentang apa yang terjadi di dalamnya dan berapa harganya berada di literatur lain, yaitu makalah pengukuran seperti Li et al. (2007) dan David et al. (2007). Makalah-makalah itu terbit hampir dua dekade lalu dan ditulis untuk pembaca yang sudah akrab dengan arsitektur prosesor. Artikel ini mencoba menjembatani keduanya. Penulis tidak melakukan pengukuran baru, tetapi merangkum alur kerja dan komponen biaya dari sumber yang sudah ada, lalu menghubungkannya dengan data penjadwalan terbaru lewat perhitungan sederhana.

Tujuan penelitian ini ada dua. Pertama, menjelaskan alur kerja *context switching* pada sistem *preemptive multitasking*. Kedua, menelaah dampaknya terhadap kinerja CPU, terutama melalui hubungan antara *time quantum*, jumlah perpindahan, dan bagian waktu CPU yang habis sebagai *overhead*. Penulis berharap artikel ini dapat menjadi bahan rujukan singkat bagi mahasiswa dan pengajar sistem operasi yang membutuhkan penjelasan berbasis pustaka.

METODE

Tahapan Penelitian

Penelitian ini berupa kajian pustaka dengan pendekatan deskriptif-analitis. Tahapannya ditunjukkan pada Gambar 1, dimulai dari perumusan topik dan kata kunci, pencarian dan seleksi literatur, ekstraksi informasi, sintesis alur serta komponen biaya, lalu ditutup dengan perhitungan ilustratif dan penyusunan naskah.



Gambar 1. Tahapan penelitian

Sumber dan Seleksi Literatur

Pencarian dilakukan melalui mesin pencari akademik, portal jurnal berbasis OJS, laman penerbit, dan dokumentasi kernel Linux, dengan kata kunci *context switching*, *context switch overhead*, *preemptive scheduling*, *time quantum*, Round Robin, dan penjadwalan CPU. Sumber dipilih dengan tiga kriteria. Pertama, artikel jurnal tentang penjadwalan CPU yang menyinggung *context switching*. Kedua, makalah pengukuran biaya *context switch* dan buku teks sistem operasi, karena keduanya menjadi rujukan dasar untuk penjelasan konsep. Ketiga, sumber nonjurnal hanya, yaitu dokumentasi resmi kernel atau tulisan praktisi yang datanya dapat ditelusuri di internet.

Perhitungan Ilustratif

Untuk menghubungkan konsep dengan angka, penulis memakai tiga persamaan sederhana. Bagian waktu CPU yang hilang sebagai *overhead* pada setiap siklus *time quantum* dinyatakan oleh Persamaan (1).

$$O = \frac{s}{q+s} \quad (1)$$

dengan q adalah lama *time quantum* dan s adalah biaya satu kali *context switch*. Persamaan ini mengandaikan setiap proses selalu memakai *time quantum* secara penuh. Untuk Round Robin, jumlah irisan eksekusi pada n proses dengan *burst time* B dihitung dengan Persamaan (2), dengan asumsi semua proses tiba pada waktu yang sama.

$$N = \sum_{i=1}^n \left\lceil \frac{B_i}{q} \right\rceil \quad (2)$$

Total waktu yang terpakai sebagai *overhead* kemudian diperoleh dari Persamaan (3), yaitu jumlah irisan dikali biaya satu perpindahan.

$$T_o = N \times s \quad (3)$$

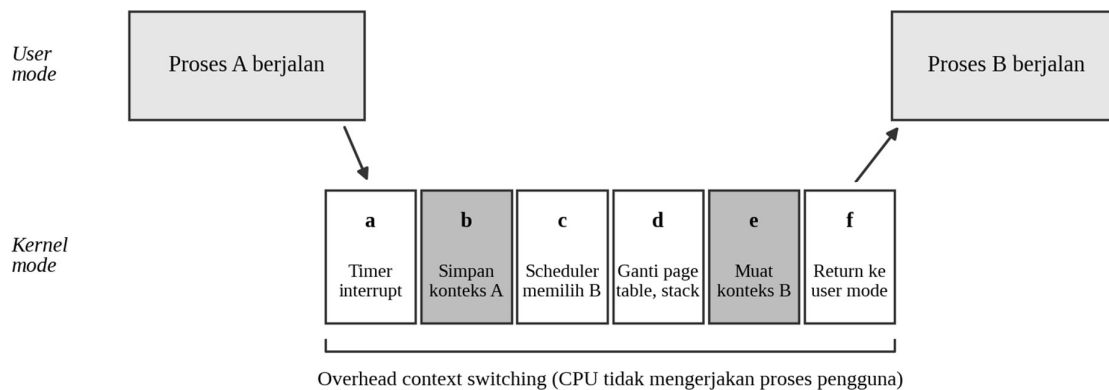
Nilai s diambil dari literatur. Penulis memakai $3,4 \mu s$ sebagai biaya langsung hasil pengukuran Li et al. (2007), dan $100 \mu s$ sebagai contoh biaya total (langsung ditambah tidak langsung) yang masih berada dalam rentang beberapa ratusan mikrodetik pada makalah yang sama. Angka $100 \mu s$ ini pilihan penulis untuk ilustrasi, bukan hasil pengukuran. Nilai q dibuat bervariasi dari $10 \mu s$ sampai $100 ms$.

HASIL DAN PEMBAHASAN

Bagian ini menguraikan alur kerja *context switching*, komponen biayanya, hubungan antara *time quantum* dan jumlah perpindahan, serta dampaknya terhadap kinerja CPU.

Alur Kerja Context Switching pada Preemptive Multitasking

Pada sistem *preemptive*, kendali atas CPU dipegang sistem operasi melalui *timer* yang dinyalakan saat *boot* dan memicu *interrupt* setiap beberapa milidetik (Arpaci-Dusseau & Arpac-Dusseau, 2018). Ketika *interrupt* terjadi, proses yang sedang berjalan dihentikan dan eksekusi berpindah ke penanganan *interrupt* di dalam *kernel*. Dari titik ini, perpindahan ke proses lain dapat dirangkum menjadi enam tahap seperti pada Gambar 2.



Gambar 2. Alur *context switching* dari proses A ke proses B pada sistem *preemptive*

- Timer interrupt*. Perangkat keras *timer* mengirim *interrupt*, CPU masuk ke *kernel mode*, dan perangkat keras menyimpan sebagian keadaan proses yang berjalan, misalnya *program counter*, ke *kernel stack* milik proses tersebut (Arpaci-Dusseau & Arpac-Dusseau, 2018).
- Penyimpanan konteks. *Kernel* menyalin sisa register dan informasi lain ke *Process Control Block* (PCB) proses A. PCB memuat antara lain keadaan proses, *program counter*, register CPU, dan informasi manajemen memori (Silberschatz et al., 2018).
- Keputusan *scheduler*. *Scheduler* memeriksa antrian proses siap dan memilih proses berikutnya sesuai algoritma, misalnya Round Robin. Bila tidak ada proses lain yang siap, proses A dapat langsung dilanjutkan.
- Pergantian *kernel stack* dan ruang alamat. *Kernel* beralih ke *kernel stack* proses B dan, bila ruang alamatnya berbeda, memuat *page table* milik proses B. Pada prosesor x86 yang mendukung PCID, tabel terjemahan alamat (TLB) tidak harus dibuang seluruhnya setiap *page table* diganti (The kernel development community, n.d.).
- Pemulihan konteks. Register milik proses B dimuat dari PCB-nya.
- Kembali ke *user mode*. Instruksi *return-from-trap* memulihkan keadaan yang tadi disimpan perangkat keras di *kernel stack*, sehingga proses B melanjutkan eksekusi dari instruksi terakhirnya (Arpaci-Dusseau & Arpac-Dusseau, 2018).

Bagian *kernel* yang menyerahkan CPU kepada proses terpilih disebut *dispatcher*, dan selang waktu yang dibutuhkannya untuk menghentikan satu proses lalu memulai proses lain disebut *dispatch latency* (Silberschatz et al., 2018). Perpindahan yang dipicu *timer* bersifat paksa. Ada pula perpindahan sukarela, misalnya ketika proses memanggil *system call* untuk operasi masukan-keluaran dan masuk ke keadaan menunggu; alur yang ditempuh *kernel* pada kasus ini serupa. Perpindahan antar-*thread* dalam satu proses umumnya lebih ringan karena ruang alamatnya sama sehingga tahap d tidak perlu dikerjakan penuh (Silberschatz et al., 2018).

Alur di atas mengasumsikan perpindahan paksa oleh *timer*. Alternatifnya adalah pendekatan kooperatif, ketika sistem operasi menunggu proses menyerahkan CPU sendiri lewat *system call* atau berhenti. Arpac-Dusseau dan Arpac-Dusseau (2018) mencatat kelemahannya: proses yang masuk ke perulangan tanpa akhir tidak akan pernah mengembalikan kendali, dan satu-satunya jalan keluar adalah memulai ulang mesin. *Timer interrupt* menutup celah itu karena kendali kembali ke *kernel* secara berkala, apa pun yang sedang dikerjakan proses. Harganya adalah perpindahan yang terjadi pada saat yang tidak dipilih proses itu sendiri.

Perlu pula dibedakan antara *context switch* dan perpindahan mode. Pada *system call* biasa, CPU hanya berpindah dari *user mode* ke *kernel mode* lalu kembali ke proses yang sama; *context switch* baru terjadi bila *kernel* memutuskan melanjutkan proses lain (Arpaci-Dusseau & Arpac-Dusseau, 2018). Keduanya sering disebut bersamaan karena sama-sama melewati *kernel*, dan

Gregg (2018) memperlakukan laju *context switch* sebagai tambahan terhadap laju *system call* ketika memperkirakan *overhead* KPTI. Dalam artikel ini, istilah *context switch* hanya dipakai untuk pergantian proses atau *thread* yang sedang berjalan.

Komponen Biaya Context Switching

Biaya *context switching* lazim dibagi dua. Biaya langsung adalah waktu yang dipakai kode *kernel* untuk tahap a sampai f, sedangkan biaya tidak langsung adalah penalti yang muncul setelah proses baru berjalan karena *cache* dan struktur terkait masih berisi data proses lama (David et al., 2007; Li et al., 2007). Li et al. (2007) mengukur biaya langsung rata-rata 3,4 μ s pada sistem uji mereka yang memiliki *cache* L1 16 KB dan L2 512 KB, serta mendapati biaya tidak langsung berkisar dari beberapa mikrodetik hingga ratusan mikrodetik. Besarnya bergantung pada ukuran data dan pola akses memori; pada pengukuran mereka, *stride* akses yang berbeda mengubah biaya perpindahan secara signifikan. David et al. (2007) menempuh cara serupa pada perangkat bergerak berbasis ARM, yakni menghitung *overhead* tidak langsung dengan mengurangi *overhead* langsung dari waktu jalan tugas yang mengalami *context switch*. Liu dan Solihin (2010) menyebut *miss* yang muncul akibat perpindahan ini sebagai *context switch misses*. Komponen-komponen biaya dirangkum pada Tabel 2.

Tabel 2. Komponen biaya *context switching*

Komponen	Jenis biaya	Penyebab	Sumber
Penyimpanan dan pemulihan register	Langsung	Salinan keadaan proses ke dan dari PCB serta <i>kernel stack</i>	Silberschatz et al. (2018); Arpaci-Dusseau & Arpaci-Dusseau (2018)
Eksekusi <i>scheduler</i> dan <i>dispatcher</i>	Langsung	Pemilihan proses dan <i>dispatch latency</i>	Silberschatz et al. (2018)
Pergantian <i>page table</i>	Langsung dan tidak langsung	Ruang alamat berbeda; TLB perlu dibuang atau dibedakan dengan PCID	The <i>kernel</i> development community (n.d.)
Miss pada <i>cache</i>	Tidak langsung	Isi <i>cache</i> masih milik proses sebelumnya	Li et al. (2007); David et al. (2007); Liu & Solihin (2010)
Miss pada TLB	Tidak langsung	Terjemahan alamat proses baru belum tersimpan di TLB	Gregg (2018); The <i>kernel</i> development community (n.d.)

Struktur lain, seperti *prediktor cabang*, secara teori ikut terpengaruh, tetapi sumber-sumber yang ditelaah dalam artikel ini tidak membahasnya sehingga tidak dimasukkan ke tabel.

Biaya tidak langsung sulit diukur karena ia tidak muncul pada satu titik waktu di dalam *kernel*. Ia tersebar sebagai *miss* selama proses baru berjalan, sehingga pengukuran yang dilakukan Li et al. (2007) dan David et al. (2007) memperkirakannya dari selisih dua kondisi uji. Li et al. (2007), misalnya, membandingkan waktu dua proses yang bergantian dengan waktu pada kondisi tanpa perpindahan, lalu membagi selisihnya dengan jumlah perpindahan. Hasilnya bergantung pada sistem uji, sehingga angka mutlak dari satu makalah sebaiknya dibaca sebagai orde besaran, bukan konstanta.

Pemicu Perpindahan pada Berbagai Algoritma Penjadwalan

Jumlah *context switch* ditentukan oleh algoritma penjadwalan, karena algoritma itulah yang menetapkan kapan CPU boleh diambil dari sebuah proses. Pada FCFS tanpa *preemption*, perpindahan hanya terjadi ketika proses selesai atau menunggu operasi masukan-keluaran, sehingga jumlahnya paling sedikit, tetapi proses pendek dapat tertahan di belakang proses panjang (Silberschatz et al., 2018). Pada Round Robin, setiap proses dipotong ketika *time quantum*-nya habis dan dipindahkan ke akhir antrian, sehingga jumlah perpindahan bergantung langsung pada *time quantum* (Putra & Purnomo, 2021). Pada Priority Preemptive, perpindahan dipicu kedatangan proses berprioritas lebih tinggi, bukan oleh waktu (Widiarto et al., 2024). Perbandingannya diringkas pada Tabel 3.

Tabel 3. Pemicu dan penentu jumlah perpindahan pada tiga algoritma penjadwalan

Algoritma	Pemicu perpindahan	Penentu jumlah perpindahan	Catatan dari literatur
FCFS (tanpa <i>preemption</i>)	Proses selesai atau menunggu masukan-keluaran	Jumlah proses dan frekuensi operasi masukan-keluaran	Perpindahan sedikit, tetapi proses pendek dapat tertahan (Silberschatz et al., 2018)
Round Robin	<i>Time quantum</i> habis	<i>Time quantum</i> dan <i>burst time</i> setiap proses	Jumlah irisan mengikuti Persamaan (2); <i>time quantum</i> kecil menambah perpindahan (Putra & Purnomo, 2021)
Priority Preemptive	Proses berprioritas lebih tinggi tiba	Pola kedatangan dan nilai prioritas	Proses berprioritas rendah dapat terhambat (Widiarto et al., 2024)

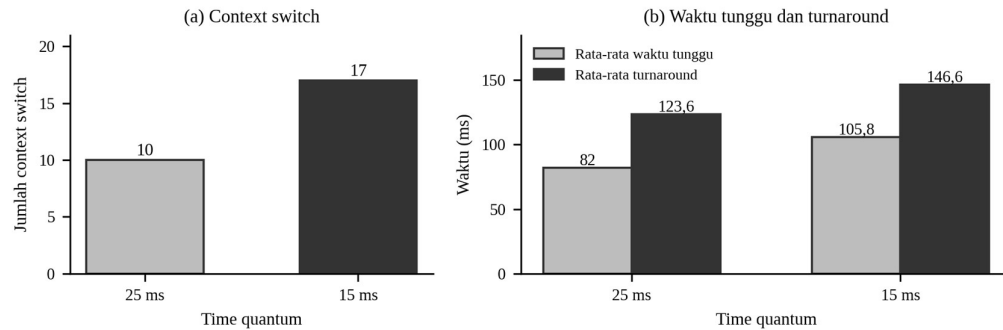
Time Quantum dan Jumlah Context Switch

Hubungan antara *time quantum* dan jumlah perpindahan paling jelas terlihat pada data Putra dan Purnomo (2021). Mereka menjalankan Round Robin pada lima proses dengan *burst time* 20, 34, 25, 62, dan 67 ms (total 208 ms). Pada *time quantum* 25 ms terjadi 10 *context switch*, rata-rata waktu tunggu 82 ms, dan rata-rata *turnaround* 123,6 ms. Ketika *time quantum* diturunkan menjadi 15 ms, jumlahnya naik menjadi 17, waktu tunggu menjadi 105,8 ms, dan *turnaround* menjadi 146,6 ms (Tabel 4 dan Gambar 3).

Tabel 4. Data penelitian terdahulu tentang *time quantum* dan kinerja Round Robin

Penelitian	Data uji	<i>Time quantum</i>	Hasil
Putra & Purnomo (2021)	5 proses; <i>burst time</i> 20, 34, 25, 62, 67 ms	25 ms	10 <i>context switch</i> ; waktu tunggu 82 ms; <i>turnaround</i> 123,6 ms

Penelitian	Data uji	Time quantum	Hasil
Putra & Purnomo (2021)	Data yang sama	15 ms	17 <i>context switch</i> ; waktu tunggu 105,8 ms; <i>turnaround</i> 146,6 ms
Ramadan et al. (2026)	10 proses (simulasi)	3 satuan waktu	Waktu tunggu rata-rata RR 17,4 vs FCFS 13,1; RR menambah <i>context switching</i>
Widiarto et al. (2024)	5 proses; <i>burst time</i> 7, 9, 5, 8, 12	4 satuan waktu	Waktu tunggu RR 14,6 vs Priority Preemptive 10,6; <i>turnaround</i> 22,8 vs 18,8



Gambar 3. Jumlah *context switch* serta waktu tunggu dan *turnaround* rata-rata pada dua nilai *time quantum* (data: Putra & Purnomo, 2021)

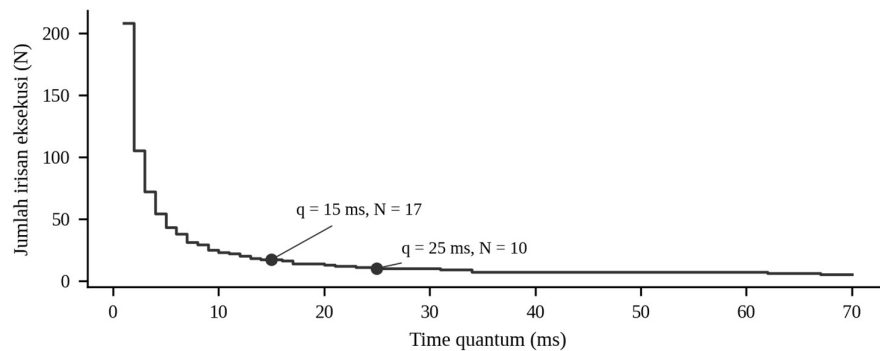
Angka 10 dan 17 tersebut dapat ditelusuri dengan Persamaan (2). Pada *time quantum* 25 ms, irisan eksekusi kelima proses masing-masing 1, 2, 1, 3, dan 3, sehingga jumlahnya 10. Pada 15 ms, irisannya 2, 3, 2, 5, dan 5, dengan jumlah 17. Kecocokan ini menunjukkan bahwa jumlah *context switch* dalam penelitian tersebut dihitung sebagai jumlah irisan eksekusi. Secara ketat, perpindahan antar-irisan berjumlah $N - 1$, sebab setelah irisan terakhir CPU tidak perlu berpindah lagi. Jumlah irisan naik 70% ketika *time quantum* diturunkan dari 25 ke 15 ms, sementara waktu tunggu rata-rata naik sekitar 29% dan *turnaround* rata-rata sekitar 19%.

Ada hal yang perlu dicermati. Total *burst time* kedua kasus sama, yaitu 208 ms, yang menandakan waktu perpindahan tidak ikut dihitung dalam simulasi tersebut. Kenaikan waktu tunggu itu karena itu bersumber dari perubahan urutan eksekusi, bukan dari biaya perpindahan. Temuan Ramadan et al. (2026) dan Widiarto et al. (2024) pada Tabel 4 searah: Round Robin memberi pembagian waktu yang adil, tetapi pada data yang mereka pakai waktu tunggu lebih besar daripada FCFS dan Priority Preemptive. Widiarto et al. (2024) mengaitkannya dengan seringnya perpindahan proses, sedangkan Sakshi et al. (2022) dan Abdelhafiz (2021) menunjuk jumlah *context switch* yang tinggi sebagai salah satu kelemahan Round Robin klasik.

Untuk melihat pola yang lebih utuh, Persamaan (2) dapat dijalankan pada data yang sama dengan nilai *time quantum* yang lebih beragam. Hasilnya pada Tabel 5 dan Gambar 4 adalah perhitungan penulis, bukan data dari makalah yang dirujuk.

Tabel 5. Jumlah irisan eksekusi (N) dan total *overhead* pada data Putra dan Purnomo (2021) menurut *time quantum* (perhitungan ilustratif penulis)

<i>Time quantum</i> (ms)	N	Total overhead, $s = 3,4 \mu s$ (μs)	Total overhead, $s = 100 \mu s$ (ms)	Persentase terhadap 208 ms ($s = 100 \mu s$)
5	43	146,2	4,3	2,07%
10	23	78,2	2,3	1,11%
15	17	57,8	1,7	0,82%
20	13	44,2	1,3	0,62%
25	10	34,0	1,0	0,48%
30	10	34,0	1,0	0,48%
40	7	23,8	0,7	0,34%
50	7	23,8	0,7	0,34%
62	6	20,4	0,6	0,29%
67	5	17,0	0,5	0,24%



Gambar 4. Jumlah irisan eksekusi (N) menurut *time quantum* pada data Putra dan Purnomo (2021), dihitung dengan Persamaan (2)

Jumlah irisan turun dari 43 pada *time quantum* 5 ms menjadi 5 pada 67 ms. Penurunannya tidak mulus. *Time quantum* 25 dan 30 ms sama-sama menghasilkan 10 irisan, karena tidak ada proses yang melewati batas irisan berikutnya pada rentang itu. Mulai 67 ms, yang sama dengan *burst time* terpanjang, setiap proses hanya mendapat satu irisan dan Round Robin berperilaku seperti FCFS, sejalan dengan catatan Silberschatz et al. (2018) bahwa *time quantum* yang sangat besar membuat Round Robin menyerupai FCFS. Di sisi lain, pada 5 ms dengan biaya total 100 μ s, *overhead* mencapai 2,07% dari seluruh *burst time*. Angka ini masih kecil, tetapi sudah berlipat-lipat dibanding kasus 67 ms yang hanya 0,24%.

Keuntungan *time quantum* kecil ada pada responsivitas. Silberschatz et al. (2018) menyatakan bahwa bila terdapat n proses dalam antrian siap dan *time quantum* bernilai q , setiap proses menunggu paling lama $(n - 1) \times q$ sebelum mendapat giliran berikutnya. Untuk lima proses seperti pada data di atas, batas itu 100 ms pada *time quantum* 25 ms, 60 ms pada 15 ms, dan 20 ms pada 5 ms. Bila biaya perpindahan s dimasukkan, batas tersebut menjadi $(n - 1) \times (q + s)$, yang dengan $s = 100 \mu$ s hanya menambah sekitar 0,4 ms. Pada skala ini, mempercepat giliran dari 100 ms menjadi 20 ms (*time quantum* 25 ke 5 ms) menaikkan *overhead* dari sekitar 0,48% menjadi 2,07% dari *burst time* untuk $s = 100 \mu$ s. Pilihan *time quantum* dengan demikian lebih ditentukan oleh sifat beban kerja daripada oleh biaya perpindahan.

Dampak terhadap Kinerja CPU

Untuk melihat harga perpindahan yang sebenarnya, Persamaan (1) dihitung pada beberapa nilai *time quantum* dengan dua nilai biaya perpindahan. Hasilnya tersaji pada Tabel 6.

Tabel 6. Fraksi waktu CPU yang terpakai sebagai *overhead* menurut Persamaan (1) (perhitungan ilustratif penulis)

Time quantum (q)	$s = 3,4 \mu$ s (biaya langsung)	$s = 100 \mu$ s (contoh biaya total)
10 μ s	25,4%	90,9%
100 μ s	3,29%	50,0%
1 ms	0,339%	9,09%
10 ms	0,034%	0,990%
100 ms	0,003%	0,100%

Pada *time quantum* 10–100 ms yang lazim dipakai sistem operasi (Silberschatz et al., 2018), *overhead* berada di bawah 1% untuk kedua nilai biaya. Ia mulai terasa ketika *time quantum* menyusut ke orde ratusan mikrodetik: pada 100 μ s, biaya langsung 3,4 μ s menyita 3,29% waktu CPU, tetapi biaya total 100 μ s menyita 50%. Contoh klasik dalam Tanenbaum dan Bos (2015) sejalan dengan hitungan ini: dengan *time quantum* 4 ms dan waktu perpindahan 1 ms, Persamaan (1) memberi 20% waktu CPU yang terbuang.

Persamaan (1) juga dapat dibalik untuk mencari *time quantum* terkecil yang menjaga *overhead* di bawah batas tertentu, yaitu Persamaan (4).

$$q_{\min} = \frac{s(1 - O_{\max})}{O_{\max}} \quad (4)$$

dengan $O_{\max}$ adalah batas *overhead* yang dapat diterima. Untuk batas 1%, *time quantum* minimum adalah 99 kali biaya perpindahan, yaitu 336,6 μ s bila $s = 3,4 \mu$ s dan 9,9 ms bila $s = 100 \mu$ s. Untuk batas 5%, nilainya 19 kali biaya perpindahan, yaitu 64,6 μ s dan 1,9 ms. Selisih sekitar dua puluh kali antara kedua hasil menunjukkan betapa besar peran biaya tidak langsung dalam menentukan batas bawah *time quantum*.

Bila hasil ini dikembalikan ke data Putra dan Purnomo (2021), gambarnya menjadi lebih jelas. Dengan biaya langsung 3,4 μ s, total *overhead* kasus 25 ms hanya 34 μ s dan kasus 15 ms 57,8 μ s, atau masing-masing sekitar 0,016% dan 0,028% dari 208 ms *burst time*. Dengan biaya total 100 μ s, angkanya menjadi 1,0 ms dan 1,7 ms, atau sekitar 0,48% dan 0,82%. Selisih waktu tunggu 23,8 ms antara kedua kasus (105,8 – 82) jauh lebih besar daripada selisih *overhead* tersebut, sehingga ia memang berasal dari urutan eksekusi. Pada skala milidetik, biaya perpindahan nyaris tidak terlihat. Dampaknya baru nyata pada beban kerja yang memaksa perpindahan sangat sering atau yang *working set*-nya besar.

Faktor *working set* ini ditunjukkan oleh Gregg (2018). Ia menyatakan bahwa laju *context switch* menambah *overhead* mitigasi Meltdown (KPTI) dengan cara yang mirip laju *system call*, dan bahwa *working set* di atas 10 MB dapat menaikkan

overhead dari sekitar 1% menjadi 7% karena TLB harus dibuang, dengan tambahan hingga 10% pada pola akses *cache* yang buruk. Dokumentasi *kernel* Linux mencatat bahwa PCID memungkinkan penggantian *page table* tanpa membuang seluruh isi TLB, sedangkan pada prosesor tanpa PCID setiap penulisan ke register CR3 membuangnya (The kernel development community, n.d.). Kedua sumber ini memperlihatkan bahwa harga perpindahan bergantung pada dukungan perangkat keras dan perilaku memori beban kerja, bukan hanya pada kode *kernel*.

Dari sisi desain, ukuran *time quantum* adalah *trade-off*. *Time quantum* kecil memperbaiki responsivitas proses interaktif tetapi menaikkan *overhead*, sementara *time quantum* besar menurunkan *overhead* namun membuat Round Robin mendekati FCFS sehingga proses pendek menunggu lebih lama (Silberschatz et al., 2018). Ramadan et al. (2026) menyimpulkan hal senada: Round Robin lebih cocok untuk sistem yang memerlukan keadilan dan responsivitas tinggi, sedangkan FCFS memadai untuk beban kerja ringan tanpa kebutuhan interaktif.

Cara Menekan *Overhead*

Dari sumber-sumber yang ditelaah, ada tiga arah yang lazim ditempuh.

- Memilih *time quantum* yang sepadan dengan beban kerja. Abdelhafiz (2021), Sakshi et al. (2022), dan Hajjar et al. (2024) mengusulkan atau mengkaji *time quantum* dinamis yang diturunkan dari *burst time* proses, dengan jumlah *context switch* dan waktu tunggu sebagai ukuran perbandingan.
- Mengurangi biaya TLB, misalnya dengan PCID dan *huge pages* (Gregg, 2018; The kernel development community, n.d.).
- Memakai *thread* dalam satu proses bila memungkinkan, karena ruang alamat yang sama membuat perpindahan lebih ringan (Silberschatz et al., 2018).

Kajian ini memiliki keterbatasan. Artikel ini tidak memuat pengukuran sendiri, dan nilai *s* berasal dari satu sistem uji yang sudah lama (Li et al., 2007), sehingga Tabel 6 hanya menunjukkan orde besaran. Perangkat keras sekarang dengan *cache* yang lebih besar dan dukungan PCID dapat memberi biaya yang berbeda. Penelitian lanjutan dapat mengukur biaya langsung dan tidak langsung pada perangkat serta *kernel* terkini dengan alat seperti *lmbench* atau *perf*.

KESIMPULAN

Context switching pada *preemptive multitasking* berlangsung dalam enam tahap, dimulai dari *timer interrupt* dan berakhir ketika proses berikutnya kembali ke *user mode*. Biayanya terdiri atas biaya langsung dari kode *kernel* dan biaya tidak langsung akibat *cache* serta TLB yang masih berisi data proses sebelumnya. Biaya tidak langsung inilah yang paling sulit diperkirakan karena bergantung pada ukuran data dan pola akses memori.

Perhitungan ilustratif dengan parameter dari literatur menunjukkan bahwa pada *time quantum* 10–100 ms *overhead* berada di bawah 1%, tetapi naik tajam ketika *time quantum* mengecil ke orde mikrodetik. Data Putra dan Purnomo (2021) memperlihatkan bahwa jumlah *context switch* naik 70% ketika *time quantum* turun dari 25 ke 15 ms, sementara kenaikan waktu tunggu pada data itu lebih tepat dijelaskan oleh perubahan urutan eksekusi daripada biaya perpindahan. Pemilihan *time quantum* karena itu perlu menimbang responsivitas, keadilan, dan *overhead* sekaligus. Penelitian selanjutnya disarankan mengukur biaya perpindahan secara langsung pada perangkat keras dan *kernel* terkini agar angka pada kajian konseptual ini dapat diuji.

UCAPAN TERIMA KASIH

Terima kasih disampaikan kepada dosen pengampu mata kuliah Sistem Operasi dan pihak-pihak yang telah memberikan dukungan dan masukan selama proses penyusunan jurnal ini.

DAFTAR PUSTAKA

- Abdelhafiz, A. (2021). VORR: A new round robin scheduling algorithm. *Al-Azhar Bulletin of Science*, 32(2), Article 12. <https://doi.org/10.21608/absb.2021.99340.1141>
- Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). *Operating systems: Three easy pieces*. Arpaci-Dusseau Books. <https://pages.cs.wisc.edu/~remzi/OSTEP/>
- David, F., Carlyle, J., & Campbell, R. (2007). Context switch overheads for Linux on ARM platforms. *Proceedings of the 2007 Workshop on Experimental Computer Science (ExpCS '07)*. ACM. <https://doi.org/10.1145/1281700.1281703>
- Gregg, B. (2018, February 9). *KPTI/KAISER Meltdown initial performance regressions*. <https://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html>
- Hajjar, O., Mekhallalati, E., Annwty, N., Alghayadh, F., Keshta, I., & Algabri, M. (2024). Performance assessment of CPU scheduling algorithms: A scenario-based approach with FCFS, RR, and SJF. *Journal of Computer Science*, 20(9), 972–985. <https://doi.org/10.3844/jcssp.2024.972.985>
- Li, C., Ding, C., & Shen, K. (2007). Quantifying the cost of context switch. *Proceedings of the 2007 Workshop on Experimental Computer Science (ExpCS '07)*. ACM. <https://doi.org/10.1145/1281700.1281702>

- Liu, F., & Solihin, Y. (2010). Understanding the behavior and implications of context switch misses. *ACM Transactions on Architecture and Code Optimization*, 7(4), 1–28.
- Omar, H. K., Jihad, K. H., & Hussein, S. F. (2021). Comparative analysis of the essential CPU scheduling algorithms. *Bulletin of Electrical Engineering and Informatics*, 10(5), 2742–2750. <https://doi.org/10.11591/eei.v10i5.2812>
- Putra, T. D., & Purnomo, R. (2021). Analisis algoritma Round Robin pada penjadwalan CPU: Algoritma scheduling CPU. *Jurnal Ilmiah Teknologi Informasi Asia*, 15(2), 85–90. <https://doi.org/10.32815/jitika.v15i2.481>
- Ramadan, A., Anraeni, S., & Sugiarti, S. (2026). Analisis perbandingan algoritma First Come First Served dan Round Robin pada proses penjadwalan. *LINIER: Literatur Informatika dan Komputer*, 3(2), 260–270. <https://jurnal.fikom.umi.ac.id/index.php/LINIER/article/view/3643>
- Sakshi, Sharma, C., Sharma, S., Kautish, S., Alsallami, S. A. M., Khalil, E. M., & Mohamed, A. W. (2022). A new median-average round Robin scheduling algorithm: An optimal approach for reducing turnaround and waiting time. *Alexandria Engineering Journal*, 61(12), 10527–10538. <https://doi.org/10.1016/j.aej.2022.04.006>
- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating system concepts* (10th ed.). Wiley.
- Tanenbaum, A. S., & Bos, H. (2015). *Modern operating systems* (4th ed.). Pearson.
- The kernel development community. (n.d.). *Page table isolation (PTI)*. The Linux Kernel documentation. <https://www.kernel.org/doc/html/v6.9/arch/x86/pti.html>
- Widiarto, W., Chaerunnisa, R. A., Tsaqif, R. A., & Nadia, S. S. (2024). Analisis perbandingan penjadwalan proses menggunakan algoritma Round Robin dan Priority Preemptive. *Progresif: Jurnal Ilmiah Komputer*, 20(2), 762–773. <https://ojs.stmik-banjarbaru.ac.id/index.php/progresif/article/view/2169>