

Analisis Manajemen Memori Pada Sistem Operasi Dalam Mendukung Kinerja Komputer

Yonatan S^{1*}, Glenn D M Sihombing², Jazluth Thoan Alriyadh³, Indra Gunawan⁴

Teknik Informatika, STIKOM Tunas Bangsa

1*yonatansmjtk@gmail.com, 2glennsihombing7@gmail.com, 3jazluththoan@gmail.com, 4indra@amiktunasbangsa.ac.id

Abstrak

Manajemen memori merupakan salah satu fungsi penting sistem operasi yang berperan dalam mengatur penggunaan memori utama, mengalokasikan ruang bagi proses, dan menjaga efisiensi pelaksanaan program. Keterbatasan kapasitas memori fisik dapat menyebabkan terjadinya *page fault*, yaitu kondisi ketika halaman yang dibutuhkan tidak tersedia di memori utama. Oleh karena itu, pemilihan algoritma penggantian halaman menjadi salah satu aspek penting dalam pengelolaan memori virtual. Penelitian ini bertujuan menganalisis pengaruh algoritma penggantian halaman terhadap efisiensi penggunaan memori melalui studi literatur dan simulasi perhitungan. Algoritma yang dianalisis meliputi *First-In First-Out* (FIFO), *Least Recently Used* (LRU), dan *Optimal*. Simulasi menggunakan tiga bingkai memori dan 13 referensi halaman dengan urutan akses yang sama pada seluruh algoritma. Hasil simulasi menunjukkan bahwa FIFO menghasilkan 10 *page fault*, LRU menghasilkan 9 *page fault*, dan Optimal menghasilkan 7 *page fault*. Rasio keberhasilan akses atau *hit ratio* masing-masing adalah 23,08%, 30,77%, dan 46,15%. Hasil tersebut menunjukkan bahwa algoritma Optimal memberikan jumlah *page fault* terendah pada skenario yang diuji, sedangkan LRU lebih baik daripada FIFO. Meskipun demikian, Optimal memerlukan informasi mengenai akses halaman pada masa mendatang sehingga lebih tepat digunakan sebagai pembandingan teoretis. Penelitian ini menunjukkan bahwa pemilihan algoritma penggantian halaman memengaruhi efisiensi pengelolaan memori, tetapi pengukuran kinerja komputer secara menyeluruh juga memerlukan pengamatan terhadap waktu akses, aktivitas penyimpanan, dan beban kerja sistem.

Kata Kunci: Algoritma Penggantian Halaman, Kinerja Komputer, Manajemen Memori, Memori Virtual

PENDAHULUAN

Sistem operasi merupakan perangkat lunak utama yang berfungsi mengelola sumber daya komputer dan menyediakan lingkungan bagi aplikasi untuk dijalankan. Sistem operasi mengatur penggunaan prosesor, memori, perangkat masukan dan keluaran, sistem berkas, serta mekanisme keamanan. Tanpa pengelolaan tersebut, setiap aplikasi harus berinteraksi secara langsung dengan perangkat keras dan mengatur sumber dayanya sendiri. Kondisi ini dapat meningkatkan kompleksitas pemrograman serta menimbulkan konflik penggunaan sumber daya. Oleh karena itu, sistem operasi diperlukan untuk mengoordinasikan pelaksanaan program agar sumber daya komputer dapat digunakan secara teratur dan efisien.

Salah satu sumber daya yang memiliki peran penting adalah memori utama atau *Random Access Memory* (RAM). Memori utama menyimpan instruksi dan data yang sedang dibutuhkan oleh prosesor. Kecepatan akses memori berpengaruh terhadap kelancaran eksekusi instruksi, sedangkan kapasitasnya menentukan seberapa banyak data dan bagian program yang dapat ditampung pada suatu waktu. Ketika beberapa aplikasi berjalan secara bersamaan, sistem operasi harus membagi kapasitas memori yang tersedia dengan mempertimbangkan kebutuhan setiap proses. Pengelolaan yang kurang tepat dapat menyebabkan tekanan memori, peningkatan aktivitas pemindahan data, dan penurunan respons sistem [1], [3].

Proses merupakan program yang sedang dieksekusi beserta keadaan dan sumber daya yang berkaitan dengan pelaksanaannya. Setiap proses memerlukan ruang alamat untuk menyimpan instruksi, variabel, struktur data, dan informasi lain yang dibutuhkan selama eksekusi. Dalam sistem multiprogramming, beberapa proses dapat berada dalam keadaan aktif, menunggu, atau siap dijalankan secara bergantian. Sistem operasi perlu menyediakan ruang memori bagi proses-proses tersebut tanpa membiarkan satu proses mengakses wilayah memori proses lain secara tidak sah. Dengan demikian, manajemen memori tidak hanya berkaitan dengan kapasitas penyimpanan, tetapi juga dengan alokasi, pemetaan alamat, perlindungan, dan isolasi proses [1], [4].

Permasalahan muncul ketika kebutuhan memori seluruh proses melebihi kapasitas memori fisik yang tersedia. Jika seluruh bagian program harus selalu berada di RAM, jumlah proses yang dapat ditangani secara bersamaan akan dibatasi oleh ukuran memori. Untuk mengatasi keterbatasan tersebut, sistem operasi modern menggunakan konsep memori virtual. Memori virtual memungkinkan setiap proses memiliki ruang alamat logis yang dikelola secara terpisah dari memori fisik. Melalui mekanisme pemetaan alamat, halaman virtual dapat dihubungkan dengan bingkai memori fisik ketika diperlukan. Bagian program atau data yang belum diperlukan tidak harus selalu berada di memori utama [1], [3].

Salah satu teknik yang banyak digunakan dalam memori virtual adalah *paging*. Teknik ini membagi ruang alamat virtual menjadi unit berukuran tetap yang disebut halaman atau *page*, sedangkan memori fisik dibagi menjadi unit berukuran yang disebut bingkai atau *frame*. Ukuran halaman dan bingkai yang bersesuaian memungkinkan sistem operasi memetakan halaman ke bingkai tanpa harus menempatkan seluruh ruang alamat proses secara berurutan di memori fisik. Pemetaan tersebut dikelola melalui struktur tabel halaman dan didukung oleh mekanisme perangkat keras untuk menerjemahkan alamat [1], [4].

Meskipun memberikan fleksibilitas, sistem paging menghadapi persoalan ketika suatu proses mengakses halaman yang belum tersedia di RAM. Kondisi tersebut disebut *page fault*. Sistem operasi harus menangani permintaan tersebut dengan memeriksa validitas akses, menyediakan bingkai yang sesuai, dan memuat halaman jika diperlukan. Apabila tidak tersedia bingkai kosong, sistem perlu menentukan halaman yang akan dikeluarkan dari memori. Proses penggantian ini menjadi penting karena halaman yang dikeluarkan mungkin akan dibutuhkan kembali sehingga dapat menimbulkan *page fault* berikutnya [1], [3].

Penggantian halaman yang tidak efisien dapat meningkatkan jumlah *page fault* dan menambah aktivitas pemindahan data antara memori utama dan penyimpanan sekunder. Jika aktivitas tersebut berlangsung berlebihan, sistem dapat mengalami *thrashing*, yaitu keadaan ketika sebagian besar waktu digunakan untuk menangani perpindahan halaman daripada menjalankan pekerjaan yang berguna. Konsep *working set* yang dikembangkan dalam kajian memori virtual menjelaskan pentingnya mempertahankan halaman-halaman yang secara aktif digunakan oleh proses agar kebutuhan memori dapat dipenuhi secara lebih efektif [7].

Berbagai algoritma penggantian halaman dikembangkan untuk menentukan halaman yang sebaiknya dikeluarkan ketika memori penuh. Algoritma *First-In First-Out* (FIFO) memilih halaman yang paling dahulu dimuat. Algoritma *Least Recently Used* (LRU) memilih halaman yang paling lama tidak digunakan. Sementara itu, algoritma *Optimal* memilih halaman yang akses berikutnya paling jauh pada masa mendatang. Ketiga algoritma tersebut menggunakan dasar pengambilan keputusan yang berbeda sehingga dapat menghasilkan jumlah *page fault* yang berbeda pada urutan akses yang sama [1], [3].

Penelitian Belady menunjukkan bahwa algoritma FIFO dapat mengalami anomali ketika penambahan bingkai justru meningkatkan jumlah *page fault* pada urutan referensi tertentu [5]. Selanjutnya, Mattson dan rekan-rekan mengembangkan teknik evaluasi untuk menganalisis hierarki penyimpanan dan karakteristik algoritma penggantian [6]. Denning menjelaskan pentingnya memori virtual dan konsep *working set* dalam memahami hubungan antara perilaku proses dan kebutuhan memori [7]. Karya-karya tersebut menjadi dasar penting untuk mengevaluasi kebijakan penggantian halaman dan memahami bahwa kapasitas memori bukan satu-satunya faktor yang menentukan hasil pengelolaan memori.

Dalam konteks sistem operasi modern, manajemen memori juga melibatkan mekanisme yang lebih kompleks daripada penggantian halaman sederhana. Dokumentasi kernel Linux membahas pengelolaan memori dan mekanisme *page reclaim*, sedangkan dokumentasi Microsoft menjelaskan pengelolaan ruang alamat virtual pada Windows [8]–[11]. Implementasi nyata dapat mempertimbangkan halaman anonim, cache berkas, halaman yang telah dimodifikasi, tekanan memori, serta karakteristik beban kerja. Oleh sebab itu, hasil simulasi algoritma dasar perlu ditafsirkan sesuai dengan batasan model yang digunakan.

Permasalahan yang menjadi fokus penelitian ini adalah bagaimana perbedaan algoritma penggantian halaman memengaruhi jumlah *page fault* dan rasio keberhasilan akses ketika kapasitas memori serta urutan referensi dibuat sama. Simulasi yang tidak menjelaskan jumlah bingkai, urutan referensi, dan langkah penggantian akan sulit direproduksi. Selain itu, jumlah *page fault* tidak secara langsung menunjukkan waktu eksekusi total karena biaya penanganan setiap kejadian dapat berbeda menurut kondisi sistem.

Berdasarkan permasalahan tersebut, penelitian ini menggunakan studi literatur dan simulasi untuk membandingkan FIFO, LRU, dan Optimal dengan satu dataset yang konsisten. Kebaruan artikel terletak pada penyajian perhitungan yang dapat ditelusuri, tabel status bingkai pada setiap akses, perbandingan hasil, serta pembahasan mengenai keterbatasan interpretasi *page fault* sebagai indikator kinerja. Penelitian ini bertujuan menjelaskan prinsip manajemen memori dalam sistem operasi, menganalisis hasil ketiga algoritma pada skenario yang ditentukan, dan memberikan pemahaman kepada mahasiswa mengenai hubungan antara kebijakan penggantian halaman dan efisiensi penggunaan memori komputer.

METODE

Tahapan Penelitian

Penelitian ini menggunakan pendekatan deskriptif melalui studi literatur dan simulasi perhitungan algoritma penggantian halaman. Studi literatur dilakukan untuk memperoleh pemahaman mengenai konsep manajemen memori, memori virtual, *paging*, *page fault*, dan algoritma penggantian halaman. Sumber yang digunakan meliputi buku teks sistem operasi, artikel ilmiah mengenai algoritma penggantian halaman, serta dokumentasi resmi sistem operasi. Simulasi digunakan untuk membandingkan hasil algoritma berdasarkan urutan referensi dan kapasitas bingkai yang sama.

Tahapan penelitian disusun secara sistematis agar proses analisis dapat direproduksi. Urutan penelitian dimulai dari identifikasi permasalahan, pengumpulan literatur, penentuan parameter simulasi, pelaksanaan simulasi untuk setiap algoritma, pencatatan hasil, perhitungan metrik, hingga interpretasi hasil.

Alur penelitian dapat digambarkan dalam bentuk teks sebagai berikut:

Mulai → Identifikasi masalah → Studi literatur → Penentuan string referensi dan jumlah bingkai → Simulasi FIFO, LRU, dan Optimal → Pencatatan status bingkai → Perhitungan jumlah *page fault* dan *hit ratio* → Perbandingan hasil → Pembahasan → Kesimpulan → Selesai.

Pada tahap identifikasi masalah, perhatian diarahkan pada keterbatasan kapasitas memori fisik dan kebutuhan menentukan halaman yang akan dikeluarkan ketika seluruh bingkai telah terisi. Tahap studi literatur digunakan untuk menentukan prinsip kerja algoritma yang dianalisis. Selanjutnya, dataset simulasi ditetapkan agar seluruh algoritma diuji menggunakan kondisi yang sama.

Pada tahap simulasi, setiap referensi halaman diperiksa berdasarkan status bingkai saat itu. Jika halaman telah berada di memori, akses dicatat sebagai *hit*. Jika halaman belum tersedia, akses dicatat sebagai *page fault*. Ketika masih terdapat bingkai kosong, halaman dimasukkan ke bingkai tersebut. Jika semua bingkai telah terisi, algoritma menentukan halaman korban sesuai dengan kebijakannya masing-masing.

Hasil simulasi kemudian digunakan untuk menghitung jumlah *page fault*, jumlah *hit*, rasio keberhasilan akses, dan rasio kegagalan akses. Tahap terakhir adalah membandingkan hasil dan menjelaskan kelebihan serta keterbatasan setiap algoritma. Penelitian ini tidak melakukan eksperimen langsung pada komputer fisik sehingga tidak mengukur waktu akses aktual, konsumsi energi, atau utilisasi prosesor.

Data Simulasi

Dataset yang digunakan terdiri atas lima jenis nomor halaman berbeda dan 13 kali akses halaman. Jumlah bingkai ditetapkan sebanyak tiga untuk seluruh pengujian utama. Pada awal simulasi, seluruh bingkai berada dalam keadaan kosong.

Tabel 1. Data dan parameter simulasi

Parameter	Nilai	Keterangan
Jumlah akses	13	Banyaknya referensi halaman
Jumlah bingkai	3	Kapasitas memori fisik dalam simulasi
String referensi	7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2	Urutan akses halaman
Algoritma	FIFO, LRU, Optimal	Algoritma yang dibandingkan
Kondisi awal	Kosong	Tidak ada halaman yang telah dimuat
Parameter utama	Jumlah <i>page fault</i>	Indikator hasil simulasi

String referensi yang digunakan adalah:

R = 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2.

Penggunaan string referensi yang sama memastikan bahwa perbedaan hasil berasal dari kebijakan penggantian halaman, bukan dari perbedaan beban akses. Setiap algoritma dijalankan dari kondisi awal kosong dan menggunakan tiga bingkai.

Algoritma yang Dianalisis

Algoritma FIFO mengganti halaman yang paling dahulu dimuat ke memori. Algoritma ini mempertahankan urutan pemuatan halaman dan tidak memperhitungkan frekuensi atau kebaruan akses. Kesederhanaan implementasinya menjadi kelebihan utama, tetapi halaman yang masih sering digunakan dapat dikeluarkan hanya karena telah berada di memori lebih lama [1], [5].

Algoritma LRU mengganti halaman yang paling lama tidak diakses. Algoritma ini menggunakan riwayat akses sebagai dasar perkiraan kebutuhan halaman pada waktu berikutnya. Halaman yang baru saja digunakan dipertahankan selama terdapat halaman lain yang lebih lama tidak digunakan. Meskipun lebih peka terhadap lokalitas temporal, implementasi LRU yang tepat membutuhkan pencatatan informasi akses sehingga dapat menimbulkan overhead [1], [3].

Algoritma Optimal mengganti halaman yang akses berikutnya paling jauh pada masa mendatang. Jika suatu halaman tidak akan digunakan lagi dalam sisa string referensi, halaman tersebut dapat dipilih sebagai korban. Karena keputusan ini memerlukan pengetahuan tentang akses masa depan, Optimal tidak dapat diterapkan secara langsung sebagai kebijakan umum untuk program yang perilaku akses berikutnya belum diketahui. Algoritma ini digunakan sebagai pembandingan teoretis untuk menilai hasil algoritma lainnya [1], [6].

Parameter dan Teknik Perhitungan

Parameter utama yang digunakan adalah jumlah *page fault*, jumlah *hit*, *hit ratio*, dan *page fault ratio*. *Page fault* terjadi ketika halaman yang diminta tidak ditemukan pada bingkai memori. Sebaliknya, *hit* terjadi ketika halaman yang diminta telah berada di memori.

Rasio keberhasilan akses dihitung menggunakan Persamaan (1).

Rumus Hit Ratio

$$\text{Hit Ratio} = \frac{\text{Jumlah Hit}}{\text{Jumlah Akses}} \times 100\%$$

Rumus Page Fault Ratio

$$\text{Page Fault Ratio} = \frac{\text{Jumlah Page Fault}}{\text{Jumlah Akses}} \times 100\%$$

Persamaan Hubungan Jumlah Akses

$$\text{Jumlah Hit} + \text{Jumlah Page Fault} = \text{Jumlah Akses}$$

Dalam simulasi ini, jumlah akses adalah 13. Oleh karena itu, jumlah *hit* ditambah jumlah *page fault* harus sama dengan 13 pada setiap algoritma. Seluruh perhitungan dilakukan berdasarkan status bingkai setelah setiap referensi diproses.

Penelitian ini tidak menghitung *CPU utilization* karena simulasi tidak mencakup durasi eksekusi proses, waktu penanganan *page fault*, waktu akses penyimpanan, atau interval ketika prosesor tidak menjalankan pekerjaan. Jumlah *page fault* hanya digunakan sebagai indikator algoritmik dalam skenario simulasi, bukan sebagai pengukuran langsung terhadap waktu eksekusi komputer.

HASIL DAN PEMBAHASAN

Hasil Simulasi FIFO

FIFO merupakan algoritma penggantian halaman yang menggunakan urutan pemuatan sebagai dasar pengambilan keputusan. Ketika bingkai belum penuh, halaman baru ditempatkan pada bingkai kosong. Setelah seluruh bingkai terisi, halaman yang paling dahulu dimuat dikeluarkan ketika terjadi *page fault*.

Pada tiga akses pertama, halaman 7, 0, dan 1 dimasukkan ke memori. Ketika halaman 2 diminta pada akses keempat, tidak tersedia bingkai kosong sehingga halaman 7 yang paling dahulu dimuat dikeluarkan. Proses penggantian berikutnya mengikuti urutan pemuatan halaman yang tersisa.

Tabel 2. Simulasi algoritma FIFO

Langkah	Halaman	F1	F2	F3	Status	Total Page Fault
1	7	7	–	–	Fault	1
2	0	7	0	–	Fault	2
3	1	7	0	1	Fault	3
4	2	2	0	1	Fault	4
5	0	2	0	1	Hit	4
6	3	2	3	1	Fault	5
7	0	2	3	0	Fault	6
8	4	4	3	0	Fault	7
9	2	4	2	0	Fault	8
10	3	4	2	3	Fault	9
11	0	0	2	3	Fault	10
12	3	0	2	3	Hit	10
13	2	0	2	3	Hit	10

Berdasarkan Tabel 2, urutan akses FIFO dapat dituliskan sebagai berikut:

7(F) → 0(F) → 1(F) → 2(F) → 0(H) → 3(F) → 0(F) → 4(F) → 2(F) → 3(F) → 0(F) → 3(H) → 2(H).

Keterangan: F menunjukkan *page fault*, sedangkan H menunjukkan *hit*. Dari 13 akses, terdapat 10 *page fault* dan 3 *hit*.

Perhitungan *hit ratio* FIFO adalah:

Rumus Hit Ratio:

$$\text{Hit Ratio} = \frac{3}{13} \times 100\% = 23,08\%$$

Rumus Page Fault Ratio:

$$\text{Page Fault Ratio} = \frac{10}{13} \times 100\% = 76,92\%$$

Hasil tersebut menunjukkan bahwa FIFO menghasilkan 10 *page fault* pada skenario yang diuji. Sebagian besar akses membutuhkan pemuatan halaman baru karena kebijakan FIFO tidak mempertimbangkan seberapa sering atau seberapa baru suatu halaman digunakan.

Kelemahan tersebut terlihat ketika halaman yang berpotensi digunakan kembali dikeluarkan hanya karena lebih dahulu dimuat. FIFO tidak membedakan halaman yang masih memiliki relevansi tinggi dengan halaman yang sudah jarang digunakan. Akibatnya, halaman yang kemudian diminta kembali dapat menimbulkan *page fault* tambahan. Walaupun mudah diterapkan, FIFO belum tentu menghasilkan penggunaan bingkai yang efisien untuk semua pola akses [1], [5].

Hasil Simulasi LRU

Algoritma LRU memilih halaman yang paling lama tidak digunakan. Setiap kali halaman ditemukan dalam memori, informasi mengenai akses terakhirnya diperbarui. Ketika terjadi *page fault* pada kondisi semua bingkai terisi, halaman dengan waktu akses terakhir paling lama dipilih sebagai korban.

Pada awal simulasi, halaman 7, 0, dan 1 dimasukkan ke tiga bingkai. Ketika halaman 2 diminta, halaman 7 dikeluarkan karena merupakan halaman yang paling lama berada tanpa akses sejak pemuatan awal. Selanjutnya, akses terhadap halaman 0 pada langkah kelima dan ketujuh memperbarui riwayat penggunaannya sehingga halaman tersebut tetap berada di memori.

Tabel 3. Simulasi algoritma LRU

Langkah	Halaman	F1	F2	F3	Status	Total Page Fault
1	7	7	–	–	Fault	1
2	0	7	0	–	Fault	2
3	1	7	0	1	Fault	3
4	2	2	0	1	Fault	4
5	0	2	0	1	Hit	4
6	3	2	0	3	Fault	5
7	0	2	0	3	Hit	5
8	4	4	0	3	Fault	6
9	2	4	0	2	Fault	7
10	3	4	3	2	Fault	8
11	0	0	3	2	Fault	9
12	3	0	3	2	Hit	9
13	2	0	3	2	Hit	9

Urutan akses LRU adalah:

7(F) → 0(F) → 1(F) → 2(F) → 0(H) → 3(F) → 0(H) → 4(F) → 2(F) → 3(F) → 0(F) → 3(H) → 2(H).

Berdasarkan tabel, LRU menghasilkan 9 *page fault* dan 4 *hit*. Perhitungan *hit ratio* LRU adalah:

Perbaikan Rumus Hit Ratio LRU:

$$\text{Hit Ratio} = \frac{4}{13} \times 100\% = 30,77\%$$

Perbaikan Rumus Page Fault Ratio LRU:

$$\text{Page Fault Ratio} = \frac{9}{13} \times 100\% = 69,23\%$$

Hasil simulasi menunjukkan bahwa LRU menghasilkan satu *page fault* lebih sedikit daripada FIFO. Pengurangan tersebut setara dengan 10% dari jumlah *page fault* FIFO pada skenario ini.

Keuntungan LRU berkaitan dengan penggunaan riwayat akses untuk menentukan halaman yang dipertahankan. Halaman yang baru digunakan tidak langsung dikeluarkan selama terdapat halaman lain yang lebih lama tidak digunakan. Pendekatan tersebut sesuai dengan prinsip lokalitas temporal, yaitu kecenderungan sebagian program untuk mengakses kembali data atau instruksi yang baru digunakan [3], [7].

Meskipun demikian, LRU tidak menjamin jumlah *page fault* paling rendah untuk setiap string referensi. Efektivitasnya bergantung pada pola akses dan kemampuan riwayat akses untuk memperkirakan kebutuhan berikutnya. Selain itu, pencatatan akses membutuhkan mekanisme tambahan yang dapat meningkatkan overhead implementasi.

Hasil Simulasi Optimal

Algoritma Optimal mengganti halaman yang akses berikutnya paling jauh dalam sisa string referensi. Jika suatu halaman tidak akan digunakan lagi, halaman tersebut dapat dipilih sebagai korban karena pengeluarannya tidak akan menimbulkan *page fault* tambahan pada sisa urutan tersebut.

Pada simulasi ini, tiga halaman pertama yang dimuat adalah 7, 0, dan 1. Ketika halaman 2 diminta pada langkah keempat, halaman 7 dapat dikeluarkan karena tidak digunakan kembali dalam sisa string. Pada langkah keenam, halaman 1 dikeluarkan karena tidak muncul lagi dalam sisa urutan. Keputusan penggantian berikutnya ditentukan dengan memeriksa posisi kemunculan halaman pada akses selanjutnya.

Tabel 4. Simulasi algoritma Optimal

Langkah	Halaman	F1	F2	F3	Status	Total Page Fault
1	7	7	–	–	Fault	1
2	0	7	0	–	Fault	2
3	1	7	0	1	Fault	3

4	2	2	0	1	Fault	4
5	0	2	0	1	Hit	4
6	3	2	0	3	Fault	5
7	0	2	0	3	Hit	5
8	4	2	4	3	Fault	6
9	2	2	4	3	Hit	6
10	3	2	4	3	Hit	6
11	0	2	0	3	Fault	7
12	3	2	0	3	Hit	7
13	2	2	0	3	Hit	7

Urutan akses Optimal adalah:

7(F) → 0(F) → 1(F) → 2(F) → 0(H) → 3(F) → 0(H) → 4(F) → 2(H) → 3(H) → 0(F) → 3(H) → 2(H).

Dari 13 akses, algoritma Optimal menghasilkan 7 *page fault* dan 6 *hit*. Perhitungan *hit ratio* adalah:

Rumus Hit Ratio Optimal:

$$\text{Hit Ratio} = \frac{6}{13} \times 100\% = 46,15\%$$

Rumus Page Fault Ratio Optimal:

$$\text{Page Fault Ratio} = \frac{7}{13} \times 100\% = 53,85\%$$

Optimal menghasilkan jumlah *page fault* terendah di antara ketiga algoritma pada skenario ini. Hasil tersebut diperoleh karena algoritma dapat memilih halaman korban dengan mengetahui kapan halaman lain akan digunakan kembali. Dengan demikian, halaman yang dibutuhkan dalam waktu dekat dapat dipertahankan di memori.

Namun, informasi mengenai akses masa depan tidak tersedia secara sempurna pada sistem operasi ketika program sedang berjalan. Oleh sebab itu, Optimal lebih tepat digunakan sebagai batas pembandingan teoretis untuk mengevaluasi algoritma praktis, bukan sebagai kebijakan penggantian yang dapat diterapkan secara langsung pada seluruh beban kerja nyata [1], [6].

Perbandingan Algoritma

Perbandingan dilakukan dengan mempertahankan jumlah bingkai sebanyak tiga dan menggunakan string referensi yang sama. Dengan demikian, jumlah *page fault* dan *hit ratio* dapat dibandingkan secara langsung.

Tabel 5. Perbandingan hasil simulasi algoritma penggantian halaman

Algoritma	Page Fault	Hit	Hit Ratio	Fault Ratio
FIFO	10	3	23,08%	76,92%
LRU	9	4	30,77%	69,23%
Optimal	7	6	46,15%	53,85%

Berdasarkan Tabel 5, FIFO menghasilkan 10 *page fault*, LRU menghasilkan 9 *page fault*, dan Optimal menghasilkan 7 *page fault*. Jumlah *hit* meningkat dari 3 pada FIFO menjadi 4 pada LRU dan 6 pada Optimal. Perbedaan tersebut menunjukkan bahwa keputusan penggantian halaman memengaruhi keberhasilan akses pada urutan referensi yang sama.

Optimal menghasilkan tiga *page fault* lebih sedikit daripada FIFO. Jika dibandingkan dengan jumlah *page fault* FIFO, pengurangan tersebut adalah:

Rumus Perhitungan Pengurangan Page Fault Optimal terhadap FIFO:

$$\frac{10 - 7}{10} \times 100\% = 30\%$$

Rumus Perhitungan Pengurangan Page Fault LRU terhadap FIFO:

$$\frac{10 - 9}{10} \times 100\% = 10\%$$

Dengan demikian, Optimal merupakan algoritma yang paling efisien berdasarkan indikator jumlah *page fault* untuk data simulasi ini. LRU berada di posisi berikutnya, sedangkan FIFO menghasilkan jumlah *page fault* tertinggi. Kesimpulan tersebut hanya berlaku untuk string referensi dan kapasitas bingkai yang diuji, bukan untuk semua kondisi penggunaan komputer.

Kelebihan FIFO adalah mekanismenya sederhana dan mudah dipahami. Namun, FIFO tidak mempertimbangkan riwayat akses sehingga dapat mengeluarkan halaman yang masih dibutuhkan. LRU lebih peka terhadap pola akses karena menggunakan informasi penggunaan terakhir, tetapi membutuhkan mekanisme pencatatan tambahan. Optimal menghasilkan jumlah *page fault* terendah pada skenario ini, tetapi membutuhkan informasi masa depan yang tidak tersedia secara langsung dalam sistem operasi umum.

CPU utilization tidak dimasukkan dalam tabel perbandingan karena penelitian ini tidak mengukur waktu pemrosesan, durasi penanganan *page fault*, waktu tunggu perangkat penyimpanan, ataupun waktu ketika CPU berada dalam keadaan idle. Jumlah *page fault* yang lebih rendah dapat menunjukkan lebih sedikit kejadian halaman yang tidak tersedia pada simulasi, tetapi tidak cukup untuk menghitung persentase utilisasi prosesor.

Pembahasan

Hasil simulasi memperlihatkan bahwa setiap algoritma menghasilkan jumlah *page fault* berbeda karena menggunakan dasar pengambilan keputusan yang berbeda. FIFO berfokus pada urutan pemuatan, LRU menggunakan riwayat akses terakhir, sedangkan Optimal mempertimbangkan akses halaman pada masa mendatang. Perbedaan informasi yang digunakan tersebut memengaruhi halaman mana yang dikeluarkan ketika seluruh bingkai terisi.

Pada FIFO, halaman yang lebih dahulu dimuat menjadi kandidat penggantian meskipun mungkin masih relevan bagi proses. Hal tersebut menjelaskan mengapa FIFO menghasilkan jumlah *page fault* tertinggi pada simulasi ini. Karakteristik FIFO juga berkaitan dengan anomali Belady, yaitu keadaan ketika penambahan bingkai justru dapat meningkatkan jumlah *page fault* untuk string referensi tertentu [5]. Oleh karena itu, kapasitas memori yang lebih besar tidak selalu menjamin hasil yang lebih baik apabila kebijakan penggantian tidak sesuai dengan pola akses.

LRU menghasilkan jumlah *page fault* yang lebih rendah daripada FIFO karena mempertimbangkan riwayat penggunaan halaman. Dalam simulasi, halaman yang baru diakses kembali dapat dipertahankan di memori sehingga beberapa akses berikutnya menjadi *hit*. Prinsip tersebut berkaitan dengan lokalitas temporal, yaitu kecenderungan program untuk menggunakan kembali data atau instruksi yang baru saja diakses [3], [7]. Meskipun demikian, LRU tetap dapat menghasilkan banyak *page fault* apabila pola akses berubah dan halaman yang sebelumnya jarang digunakan tiba-tiba diperlukan kembali.

Optimal memberikan hasil terbaik dalam simulasi karena mengetahui urutan akses halaman berikutnya. Informasi tersebut memungkinkan algoritma mengeluarkan halaman yang tidak segera diperlukan atau tidak digunakan lagi dalam sisa urutan. Akan tetapi, sistem operasi tidak dapat mengetahui seluruh akses masa depan secara pasti untuk program umum. Dengan demikian, Optimal berfungsi sebagai pembanding untuk menilai seberapa dekat algoritma praktis terhadap hasil ideal [1], [6].

Jumlah *page fault* berhubungan dengan potensi biaya akses memori karena halaman yang tidak tersedia harus ditangani sebelum proses dapat melanjutkan akses tersebut. Jika penanganan memerlukan pembacaan dari penyimpanan sekunder, proses dapat mengalami tambahan latensi. Namun, biaya setiap *page fault* tidak selalu sama. Sebagian kejadian dapat diselesaikan tanpa membaca data dari perangkat penyimpanan, sedangkan kejadian lainnya memerlukan operasi yang lebih mahal. Oleh karena itu, jumlah *page fault* tidak dapat langsung dikonversi menjadi waktu eksekusi tanpa model biaya dan pengukuran tambahan [1], [2].

Pada sistem operasi modern, pengelolaan memori melibatkan banyak mekanisme selain algoritma penggantian halaman sederhana. Dokumentasi kernel Linux menjelaskan pengelolaan memori dan *page reclaim*, yang berkaitan dengan upaya sistem memperoleh kembali halaman ketika terdapat tekanan memori [8], [9]. Sistem juga perlu mempertimbangkan jenis halaman, cache berkas, halaman anonim, serta kondisi halaman yang telah dimodifikasi. Hal tersebut menunjukkan bahwa hasil simulasi FIFO, LRU, dan Optimal tidak dapat disamakan secara langsung dengan perilaku keseluruhan kernel sistem operasi produksi.

Windows juga menyediakan mekanisme ruang alamat virtual untuk mengelola pemetaan dan akses memori oleh proses. Ruang alamat virtual memungkinkan proses bekerja dengan alamat logis yang diterjemahkan melalui mekanisme sistem dan perangkat keras. Pengelolaan tersebut mendukung isolasi antarp proses, pengaturan izin akses, dan pemanfaatan memori yang tersedia [10], [11]. Dengan demikian, keberhasilan manajemen memori perlu dinilai tidak hanya dari jumlah *page fault*, tetapi juga dari aspek keamanan, proteksi, alokasi, dan overhead pengelolaan.

Manajemen memori juga berkaitan dengan penerjemahan alamat dan perlindungan data. Tabel halaman digunakan untuk memetakan halaman virtual ke bingkai fisik, sedangkan mekanisme proteksi membantu membatasi akses baca, tulis, dan eksekusi sesuai dengan hak yang diberikan. Ruang alamat yang terpisah memungkinkan proses berjalan tanpa harus berbagi seluruh memori secara langsung. Di sisi lain, pemetaan bersama untuk halaman tertentu dapat membantu mengurangi penggunaan memori ketika beberapa proses memanfaatkan konten yang sama [1], [3], [12].

Perbedaan hasil simulasi dengan penelitian atau pengujian lain dapat disebabkan oleh perbedaan string referensi, jumlah bingkai, kondisi awal, aturan pemilihan korban ketika terdapat beberapa kandidat, dan karakteristik beban kerja. Studi Belady berfokus pada sifat algoritma FIFO, sedangkan kajian Mattson dan rekan-rekan membahas evaluasi hierarki penyimpanan [5], [6]. Sementara itu, dokumentasi sistem operasi modern mencakup mekanisme yang lebih luas daripada model penggantian halaman dalam penelitian ini [8]–[11]. Karena itu, perbandingan hasil perlu memperhatikan konteks dan asumsi setiap penelitian.

Penelitian ini memiliki beberapa keterbatasan. Pertama, simulasi hanya menggunakan satu string referensi dengan 13 akses halaman. Kedua, jumlah bingkai pada pengujian utama ditetapkan sebanyak tiga. Ketiga, penelitian tidak melakukan pengukuran langsung terhadap waktu akses RAM, aktivitas penyimpanan, penggunaan CPU, ataupun waktu respons aplikasi. Keempat, simulasi tidak memperhitungkan halaman yang dimodifikasi, *prefetching*, cache, dan interaksi beberapa proses secara bersamaan.

Keterbatasan tersebut menunjukkan bahwa hasil penelitian lebih tepat digunakan untuk memahami prinsip algoritma penggantian halaman daripada untuk menyimpulkan kinerja sistem operasi secara menyeluruh. Penelitian selanjutnya dapat menggunakan beberapa string referensi dengan panjang dan pola lokalitas yang berbeda, variasi jumlah bingkai, serta pengujian pada sistem nyata. Pengukuran jumlah *page fault*, waktu respons, aktivitas penyimpanan, dan penggunaan memori secara bersamaan dapat memberikan gambaran yang lebih lengkap mengenai hubungan kebijakan manajemen memori dengan kinerja komputer.

KESIMPULAN

Manajemen memori merupakan fungsi penting sistem operasi yang mendukung efisiensi komputer melalui pengalokasian ruang memori, pemetaan alamat, perlindungan proses, dan pengelolaan memori virtual. Berdasarkan simulasi menggunakan tiga bingkai dan 13 referensi halaman, algoritma FIFO menghasilkan 10 *page fault* dengan *hit ratio* 23,08%, LRU menghasilkan 9 *page fault* dengan *hit ratio* 30,77%, sedangkan Optimal menghasilkan 7 *page fault* dengan *hit ratio* 46,15%. Hasil tersebut menunjukkan bahwa Optimal memberikan hasil terbaik berdasarkan jumlah *page fault* pada skenario yang diuji, sementara LRU menghasilkan hasil yang lebih baik daripada FIFO. Meskipun demikian, Optimal memerlukan informasi akses masa depan sehingga berfungsi terutama sebagai pembanding teoretis. Penelitian ini memberikan contoh perhitungan yang

dapat ditelusuri untuk membantu mahasiswa memahami hubungan antara algoritma penggantian halaman, pola akses, dan penggunaan memori. Karena jumlah *page fault* tidak secara langsung menggambarkan seluruh aspek kinerja komputer, penelitian selanjutnya disarankan menggunakan lebih banyak pola referensi dan melakukan pengukuran langsung terhadap waktu akses, aktivitas penyimpanan, serta penggunaan sumber daya sistem.

UCAPAN TERIMA KASIH

Ucapan terima kasih disampaikan kepada dosen pengampu mata kuliah Sistem Operasi atas arahan akademik dalam memahami konsep manajemen memori serta kepada pihak-pihak yang mendukung proses pembelajaran dan penyusunan kajian ini. Bagian ini dapat disesuaikan dengan kondisi sebenarnya atau dihapus apabila tidak diperlukan.

DAFTAR PUSTAKA

- [1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ, USA: John Wiley & Sons, 2018.
- [2] W. Stallings, *Operating Systems: Internals and Design Principles*, 9th ed. Hoboken, NJ, USA: Pearson, 2018.
- [3] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2018. [Online]. Available: <https://pages.cs.wisc.edu/~remzi/OSTEP/>
- [4] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Boston, MA, USA: Pearson, 2015.
- [5] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966, doi: 10.1147/sj.52.0078.
- [6] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970, doi: 10.1147/sj.92.0078.
- [7] P. J. Denning, "Virtual memory," *ACM Computing Surveys*, vol. 2, no. 3, pp. 153–189, 1970, doi: 10.1145/356571.356573.
- [8] The Linux Kernel Documentation, "Memory Management." [Online]. Available: <https://docs.kernel.org/admin-guide/mm/index.html>
- [9] The Linux Kernel Documentation, "Page Reclaim." [Online]. Available: https://docs.kernel.org/mm/page_reclaim.html
- [10] Microsoft, "Virtual Address Spaces." [Online]. Available: <https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/virtual-address-spaces>
- [11] Microsoft, "Memory Management." [Online]. Available: <https://learn.microsoft.com/en-us/windows/win32/memory/memory-management>
- [12] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>