TIN: Terapan Informatika Nusantara
Vol 7, No 1, June 2026, page 513-524
ISSN 2722-7987 (Media Online)
Website https://ejurnal.seminar-id.com/index.php/tin
DOI 10.47065/tin.v7i1.10112

Analisis Banker's Algorithm untuk Penghindaran Deadlock Berbasis Simulasi Kuantitatif Multiskenario

Christian Bastanta Sembiring Meliala*, Teti Desyani, Moch Ibba Ali Yassin, Aldiansyah Sastrawinata, Mikael Surya Saputra, Brian Aidil Rizkita, Rizki Arohman Maulana

Fakultas Ilmu Komputer, Program Studi Teknik Informatika, Universitas Pamulang, Tangerang Selatan, Indonesia
Email: [1,*]yannsembiring@gmail.com, [2]dosen00839@unpam.ac.id, [3]alinih111222@gmail.com,
[4]aldiansyahsastrawinata5@gmail.com, [5]mikaelsurya9@gmail.com, [6]brianaidil03@gmail.com, [7]rizkiarohman1978@gmail.com
Email Penulis Korespondensi: yannsembiring@gmail.com

Abstrak–Kondisi *deadlock* menjadi salah satu ancaman serius dalam pengelolaan sumber daya pada sistem operasi, karena dapat menyebabkan seluruh proses komputasi terhenti secara total. Penelitian ini berfokus pada pengujian seberapa efektif, efisien, dan terbatasnya *Banker's Algorithm* sebagai pendekatan penghindaran *deadlock* melalui metode simulasi berbasis kuantitatif multiskenario. Data yang digunakan bersumber dari hasil simulasi terhadap tiga komponen utama, yaitu matriks alokasi sumber daya (*Allocation*), deklarasi kebutuhan maksimum proses (*Max*), serta vektor sumber daya yang tersedia (*Available*), pada konfigurasi sistem yang terdiri dari lima proses dan tiga jenis sumber daya. Hasil yang diperoleh memperlihatkan bahwa *Banker's Algorithm* berhasil membedakan kondisi aman (*safe state*) dan tidak aman (*unsafe state*) secara tepat melalui dua mekanisme intinya, yaitu *Safety Algorithm* dan *Resource-Request Algorithm*. Ketika kondisi *Available* bernilai [3, 3, 2], algoritma berhasil menemukan urutan eksekusi aman ⟨P1, P3, P4, P0, P2⟩ yang memastikan semua proses dapat diselesaikan tanpa risiko *deadlock*. Namun ketika *Available* diturunkan menjadi [2, 1, 0], sistem terdeteksi masuk ke dalam *unsafe state* di mana tidak ada satu pun proses yang mampu memulai eksekusinya. Dari serangkaian simulasi multiskenario, ditemukan bahwa titik kritis peralihan dari *safe* ke *unsafe* terjadi pada tingkat utilisasi sumber daya di sekitar angka 83%. Dari sisi efisiensi, kompleksitas waktu $O(n^2 \times m)$ membuat algoritma ini cukup andal untuk sistem berukuran kecil hingga menengah, meskipun berpotensi menjadi hambatan performa pada sistem komputasi awan yang beroperasi dalam skala besar. Penelitian ini menghasilkan kerangka evaluasi kuantitatif berbasis simulasi yang dapat dijadikan rujukan dalam mengimplementasikan *Banker's Algorithm* pada sistem operasi modern.

Kata Kunci: *Banker's Algorithm*; *Deadlock*; Alokasi; OS; Simulasi

Abstract–Deadlock represents a critical threat in operating system resource management, as it has the potential to bring all computational processes to a complete halt. This study examines the effectiveness, efficiency, and constraints of the Banker's Algorithm as a deadlock avoidance mechanism through a multi-scenario quantitative simulation. The data were derived from simulations involving three core components: the resource allocation matrix (Allocation), the maximum process requirement declaration (Max), and the resource availability vector (Available), within a system configuration consisting of five processes and three resource types. The findings demonstrate that the Banker's Algorithm accurately distinguishes between safe and unsafe states through its two primary mechanisms: the Safety Algorithm and the Resource-Request Algorithm. With Available set to [3, 3, 2], the algorithm successfully identified the safe execution sequence ⟨P1, P3, P4, P0, P2⟩, ensuring all processes could complete without deadlock risk. When Available was reduced to [2, 1, 0], the system entered an unsafe state in which no process could initiate execution. Through multi-scenario simulations, the critical transition threshold from a safe to an unsafe state was identified at approximately 83% resource utilization. In terms of efficiency, the $O(n^2 \times m)$ time complexity makes the algorithm well-suited for small to medium-scale systems, though it may become a performance bottleneck in large-scale cloud computing environments. This study produces a quantitative evaluation framework that can serve as a reference for implementing the Banker's Algorithm in modern operating systems.

Keywords: Banker's Algorithm; Deadlock; Allocation; OS; Simulation

1. PENDAHULUAN

Perkembangan sistem komputasi modern yang semakin kompleks mulai dari sistem operasi multiprosesor, platform komputasi awan (*cloud computing*), hingga ekosistem *Internet of Things* (IoT) menjadikan pengelolaan sumber daya secara efisien sebagai kebutuhan yang tidak dapat diabaikan. Di tengah kompleksitas tersebut, *deadlock* muncul sebagai salah satu permasalahan mendasar yang kerap mengancam keandalan sistem. Beberapa penelitian empiris menegaskan bahwa jika alokasi pada infrastruktur modern tidak dikelola dengan baik, efisiensi sistem akan merosot tajam, sebagaimana ditunjukkan pada studi penjadwalan tugas berbasis kepercayaan (*trust-aware*) yang membuktikan bahwa kegagalan dalam memetakan beban kerja heterogen ke virtual machine yang tepat secara langsung menurunkan tingkat keberhasilan eksekusi dan memperpanjang waktu penyelesaian tugas pada lingkungan komputasi awan (Mangalampalli et al., 2023). Hal ini juga menjadi tantangan besar pada penyediaan layanan otonom di masa depan yang berjalan di atas infrastruktur telekomunikasi modern yang rawan terhadap kompleksitas alokasi (Ugwuanyi et al., 2023). Oleh karena itu, pengelolaan distribusi sumber daya yang terstruktur menjadi kebutuhan krusial untuk menjaga performa sistem di bawah beban tinggi (Bal et al., 2022). Kebutuhan ini semakin mendesak pada lingkungan cloud terfederasi, di mana beberapa penyedia layanan harus saling berkolaborasi mengelola sumber daya bersama agar kualitas layanan (QoS) tetap terjaga meskipun permintaan pengguna berfluktuasi secara dinamis (Samha, 2023). Tantangan serupa juga ditemukan pada lingkungan komputasi awan berskala besar, di mana keterbatasan sumber daya yang tersedia secara bersamaan dapat memicu kondisi *deadlock* apabila mekanisme alokasi tidak dirancang secara cermat sejak tahap awal (Bondarenko & Azeez, 2021). *Deadlock* terjadi ketika dua atau lebih proses saling menunggu pelepasan sumber daya yang dikuasai oleh proses lain dalam sebuah siklus, sehingga tidak ada satu pun proses yang dapat melanjutkan

TIN: Terapan Informatika Nusantara
Vol 7, No 1, June 2026, page 513-524
ISSN 2722-7987 (Media Online)
Website https://ejurnal.seminar-id.com/index.php/tin
DOI 10.47065/tin.v7i1.10112

eksekusinya (Silberschatz et al., 2018). Dampak yang ditimbulkan bisa sangat merugikan: pada infrastruktur kritis seperti sistem perbankan elektronik, kendali industri otomatis, atau platform *e-government*, kondisi ini berpotensi memicu kerugian ekonomi yang besar, merusak integritas data, serta mengikis kepercayaan masyarakat terhadap layanan digital.

Dalam kajian teori sistem operasi, (Coffman et al., 1971) merumuskan empat kondisi yang harus terpenuhi secara bersamaan agar *deadlock* dapat terjadi, yang dikenal sebagai *Coffman Conditions*. Keempat kondisi tersebut adalah: (1) *mutual exclusion*, yaitu sumber daya hanya boleh digunakan oleh satu proses dalam satu waktu; (2) *hold and wait*, yaitu proses yang sedang menguasai sumber daya mengajukan permintaan atas sumber daya lain yang masih dipegang proses lain; (3) *no preemption*, yaitu sumber daya tidak dapat direbut paksa dari proses yang memegangnya; dan (4) *circular wait*, yaitu terbentuknya rantai tunggu melingkar di antara sejumlah proses. Selama salah satu kondisi ini tidak terpenuhi, *deadlock* dipastikan tidak akan terjadi. Secara umum, strategi penanganan *deadlock* dibagi ke dalam tiga kategori: *deadlock prevention, deadlock avoidance,* dan *deadlock detection and recovery* (Silberschatz et al., 2018). Pada lingkungan sistem terdistribusi, kompleksitas penanganan deadlock meningkat karena proses dan sumber daya tersebar pada beberapa node yang saling terhubung, sehingga mekanisme deteksi maupun penghindaran harus mempertimbangkan keterbatasan visibilitas global terhadap status sistem secara keseluruhan (Rout et al., 2021). Pada sistem terdistribusi, kompleksitas penghindaran deadlock meningkat secara signifikan karena proses dapat mengakses sumber daya baik secara lokal maupun pada node lain, sehingga sintesis kontroler penghindaran deadlock perlu mempertimbangkan koordinasi antar-node secara formal agar keputusan alokasi tetap konsisten di seluruh sistem.

Banker's Algorithm, yang pertama kali dikembangkan sebagai mekanisme penghindaran *deadlock* klasik, hingga saat ini masih menjadi salah satu algoritma yang paling banyak dipelajari secara akademik (Silberschatz et al., 2018). Cara kerja algoritma ini bertumpu pada prinsip kehati-hatian: sistem hanya akan menyetujui permintaan sumber daya dari suatu proses apabila kondisi sistem setelah alokasi tersebut dilakukan masih berada dalam safe state, yakni kondisi di mana masih terdapat setidaknya satu urutan eksekusi yang memungkinkan seluruh proses menyelesaikan operasinya tanpa terjadi *deadlock* (Silberschatz et al., 2018). Algoritma ini memiliki dua komponen utama: *Safety Algorithm* yang bertugas menentukan status safe state secara berkala di mana optimasi urutan eksekusi proses yang aman terus dikembangkan untuk mempercepat deteksi (Jiang, 2019) dan *Resource-Request Algorithm* yang bertugas mengevaluasi setiap permintaan alokasi sumber daya pada saat *runtime*. Upaya penyempurnaan mekanisme deteksi keamanan ini juga terlihat pada penelitian yang mengusulkan algoritma deteksi keselamatan (safety detection) yang dimodifikasi untuk mempercepat identifikasi status aman pada sistem dengan jumlah proses yang besar, sehingga overhead komputasi pada *Safety Algorithm* dapat diminimalkan tanpa mengorbankan akurasi deteksi (Begum et al., 2020).

Berbagai penelitian sebelumnya telah mengkaji *Banker's Algorithm* sebagai metode penghindaran *deadlock* pada sistem operasi dan pengelolaan sumber daya. Salah satu fokus utamanya adalah bagaimana melakukan optimalisasi agar algoritma ini mampu secara dinamis menghindari kondisi tidak aman pada alokasi *resource* sistem operasi (Wicaksono et al., 2023). Selain itu, (Silberschatz et al., 2018) menjelaskan bahwa mekanisme alokasi sumber daya yang terkontrol bekerja dengan mengevaluasi tingkat keamanan sistem guna memastikan setiap proses hanya memperoleh *resource* ketika sistem berada dalam kondisi aman. Pendekatan klasik ini juga tetap relevan sebagai media simulasi serta pembelajaran manajemen *resource* pada lingkungan multiproses karena mampu menunjukkan visualisasi *safe state* dan *unsafe state* secara terstruktur sebelum alokasi dieksekusi. Relevansi pendekatan klasik ini juga terlihat pada penelitian skala industri yang mengusulkan kerangka alokasi sumber daya berbasis kecerdasan buatan, di mana prinsip kehati-hatian dalam pemberian *resource* tetap menjadi fondasi utama meskipun mekanisme evaluasinya telah diperluas dengan teknik komputasional yang lebih adaptif (Nguyen Trong et al., 2023).

Dari tinjauan terhadap berbagai penelitian tersebut, teridentifikasi tiga celah penelitian yang belum terjawab. Pertama, sebagian besar kajian masih bersifat deskriptif-teoritis dan belum menyertakan analisis kuantitatif yang membandingkan kinerja algoritma secara sistematis pada berbagai skenario sumber daya. Kedua, belum ada penelitian yang secara eksplisit mengukur batas peralihan antara *safe state* dan *unsafe state* dalam kaitannya dengan persentase utilisasi sumber daya padahal informasi mengenai *threshold* kritis ini sangat berharga bagi para perancang sistem. Ketiga, korelasi empiris antara tingkat utilisasi dan beban iteratif *Safety Algorithm* belum pernah diukur secara eksplisit dalam konteks yang dapat direplikasi.

Berdasarkan identifikasi tersebut, penelitian ini bertujuan untuk: (1) menguji efektivitas *Banker's Algorithm* dalam membedakan *safe state* dan *unsafe state* melalui simulasi matriks alokasi sumber daya; (2) mengukur efisiensi komputasi algoritma berdasarkan kompleksitas waktu pada berbagai skala sistem; serta (3) mengidentifikasi secara kuantitatif batasan-batasan implementasi algoritma pada berbagai kondisi beban kerja. Kontribusi utama penelitian ini adalah tersedianya kerangka evaluasi kuantitatif berbasis simulasi yang dapat dijadikan referensi implementasi *Banker's Algorithm* pada sistem operasi modern, khususnya pada sektor perbankan digital dan infrastruktur *e-government* Indonesia yang menuntut tingkat keandalan tinggi. Kebaruan penelitian ini terletak pada identifikasi *threshold* utilisasi kritis (~83%) sebagai panduan operasional konkret yang belum pernah dilaporkan secara eksplisit dalam literatur yang ditinjau.

TIN: Terapan Informatika Nusantara
Vol 7, No 1, June 2026, page 513-524
ISSN 2722-7987 (Media Online)
Website https://ejurnal.seminar-id.com/index.php/tin
DOI 10.47065/tin.v7i1.10112

2. METODOLOGI PENELITIAN

2.1 Desain dan Konfigurasi Simulasi

Penelitian ini menerapkan pendekatan kuantitatif dengan metode simulasi komputasional deterministik menggunakan bahasa pemrograman Python 3.11 dengan dukungan pustaka NumPy untuk operasi matriks dan modul *time* untuk pengukuran durasi eksekusi. Pemilihan metode simulasi didasarkan pada pertimbangan bahwa melakukan eksperimen langsung pada sistem operasi yang sedang berjalan di lingkungan produksi tidak hanya berisiko menimbulkan gangguan nyata, tetapi juga tidak etis secara akademik (Creswell & Creswell, 2017). Konfigurasi sistem yang diterapkan terdiri dari n = 5 proses (P0 hingga P4) dan m = 3 jenis sumber daya (*Resource* A, B, dan C) dengan kapasitas total masing-masing sebesar 10, 5, dan 7 unit, di mana variabel independen berupa vektor *Available* divariasikan pada enam tingkatan (S1-S6) untuk memetakan kondisi sistem mulai dari *safe state* hingga *unsafe state*, sementara variabel dependen yang diukur meliputi status sistem, *safe sequence* yang berhasil ditemukan, jumlah iterasi *Safety Algorithm*, dan waktu eksekusi komputasi. Analisis data dilakukan secara deskriptif-komparatif dengan mengorelasikan tingkat utilisasi sumber daya, dihitung menggunakan rumus (Total *Allocation* / Total Capacity) × 100% terhadap keberhasilan *Safety Algorithm* menemukan urutan eksekusi yang aman, sementara validasi kompleksitas waktu secara empiris dilakukan pada skala n = 5, 10, 20, 50, dan 100 dengan setiap skenario diulang sebanyak 50 kali dan diambil nilai rata-ratanya guna meminimalkan variasi akibat fluktuasi beban sistem. Gambar 1 mengilustrasikan alur keseluruhan tahapan penelitian yang dilaksanakan secara berurutan dari studi literatur hingga analisis hasil simulasi.

2.2 Tahapan Penelitian

Gambar 1 mengilustrasikan alur tahapan penelitian yang digunakan. Secara keseluruhan, penelitian ini dilaksanakan melalui lima fase yang berurutan: (1) studi literatur dan pemetaan celah penelitian yang ada; (2) perancangan konfigurasi sistem simulasi, mencakup penetapan nilai matriks *Allocation*, *Max*, dan vektor *Available*; (3) pengembangan implementasi *Safety Algorithm* dan *Resource-Request Algorithm* menggunakan bahasa pemrograman Python 3.11 dengan dukungan pustaka NumPy; (4) pelaksanaan simulasi multiskenario dengan enam variasi vektor *Available* (S1 hingga S6); serta (5) analisis dan interpretasi hasil simulasi secara komparatif-kuantitatif.

Gambar 1. Alur Tahapan Penelitian Simulasi *Banker's Algorithm*

Gambar 1 menggambarkan kelima fase penelitian tersebut secara linier dan berurutan, di mana keluaran dari satu fase menjadi prasyarat masukan bagi fase berikutnya. Studi literatur pada fase pertama menjadi dasar penetapan parameter pada fase perancangan konfigurasi, sementara hasil implementasi algoritma pada fase ketiga langsung digunakan untuk menjalankan simulasi multiskenario pada fase keempat. Struktur alur kerja yang bersifat sekuensial ini dipilih agar setiap keputusan desain (misalnya pemilihan nilai Available pada setiap skenario S1-S6) dapat ditelusuri kembali ke tahap perancangan sebelumnya, sehingga seluruh proses penelitian tetap transparan dan dapat direplikasi oleh peneliti lain.

2.3 Kajian Teoretis dan Formulasi Matematis *Banker's Algorithm*

Sebelum simulasi dijalankan, perlu ditegaskan landasan teoretis-matematis dari setiap komponen *Banker's Algorithm* yang digunakan, sebagaimana dirumuskan secara formal oleh (Silberschatz et al., 2018) dan berakar pada kerangka kondisi *deadlock* klasik yang dirumuskan (Coffman et al., 1971). Terdapat lima struktur data inti yang menjadi fondasi

TIN: Terapan Informatika Nusantara
Vol 7, No 1, June 2026, page 513-524
ISSN 2722-7987 (Media Online)
Website https://ejurnal.seminar-id.com/index.php/tin
DOI 10.47065/tin.v7i1.10112

algoritma ini. Pertama, matriks *Allocation* berukuran n×m, di mana *Allocation*[i][j] menyatakan jumlah unit sumber daya jenis Rj yang sedang dialokasikan kepada proses Pi pada suatu waktu tertentu. Kedua, matriks *Max* berukuran n×m, di mana *Max*[i][j] menyatakan deklarasi kebutuhan maksimum unit sumber daya Rj yang mungkin diminta oleh proses Pi sepanjang siklus hidupnya; nilai ini wajib dideklarasikan di muka (*a priori*) sebelum proses mulai berjalan, sebagai prasyarat mutlak agar algoritma dapat menjamin keamanan sistem. Ketiga, vektor *Available* berukuran m, di mana *Available*[j] menyatakan jumlah unit sumber daya Rj yang belum dialokasikan ke proses mana pun dan masih tersedia bagi sistem.

Keempat, matriks *Need* diturunkan secara matematis dari kedua matriks sebelumnya melalui hubungan *Need*[i][j] = *Max*[i][j] − *Allocation*[i][j], yang menyatakan jumlah unit sumber daya tambahan yang masih dibutuhkan proses Pi untuk jenis Rj guna mencapai batas maksimum kebutuhannya. Matriks *Need* inilah yang menjadi parameter inti dalam menentukan kelayakan eksekusi setiap proses pada tahap *Safety Algorithm*. Kelima, dua variabel kerja, yaitu vektor *Work* dan vektor boolean *Finish*, digunakan secara internal oleh *Safety Algorithm* untuk menyimulasikan kondisi sistem secara hipotetis tanpa benar-benar mengubah status alokasi yang sesungguhnya. Secara formal, algoritma diinisialisasi dengan *Work* = *Available* dan *Finish*[i] = *false* untuk seluruh i = 0, 1, ..., n−1. Algoritma kemudian mencari indeks i sedemikian rupa hingga *Finish*[i] = *false* dan *Need*[i] ≤ *Work* secara elemen-per-elemen terpenuhi; apabila ditemukan, sistem menetapkan *Work* = *Work* + *Allocation*[i] dan *Finish*[i] = *true*, lalu mengulang pencarian dari awal. Proses iteratif ini terus berlanjut hingga salah satu dari dua kondisi terminasi tercapai: seluruh *Finish*[i] bernilai *true* (sistem dinyatakan *safe state* dengan *safe sequence* sebagai urutan eksekusi yang ditemukan), atau tidak ada lagi indeks i yang memenuhi kondisi *Need*[i] ≤ *Work* (sistem dinyatakan *unsafe state*).

Adapun *Resource-Request Algorithm* bekerja sebagai lapisan validasi tambahan yang dijalankan setiap kali proses Pi mengajukan permintaan sumber daya berupa vektor *Request*i. Algoritma ini memverifikasi dua kondisi prasyarat secara berurutan, yaitu *Request*i ≤ *Need*[i] (permintaan tidak melebihi deklarasi kebutuhan awal) dan *Request*i ≤ *Available* (permintaan tidak melebihi sumber daya yang tersedia saat itu). Jika kedua kondisi terpenuhi, sistem melakukan alokasi sementara (*tentative allocation*) dengan memperbarui *Available* = *Available* − *Request*i, *Allocation*[i] = *Allocation*[i] + *Request*i, dan *Need*[i] = *Need*[i] − *Request*i, kemudian menjalankan *Safety Algorithm* pada kondisi hipotetis tersebut. Apabila hasilnya tetap berstatus *safe state*, alokasi dikonfirmasi secara permanen; sebaliknya, apabila hasilnya berstatus *unsafe state*, seluruh perubahan tersebut dibatalkan (*rollback*) dan proses Pi harus menunggu hingga permintaan dapat dipenuhi tanpa membahayakan keamanan sistem (Silberschatz et al., 2018). Kerangka matematis inilah yang menjadi dasar implementasi program sebagaimana diuraikan pada subbab 2.1 dan menjadi acuan utama dalam menginterpretasikan seluruh hasil numerik pada Bab 3.

3. HASIL DAN PEMBAHASAN

3.1 Konfigurasi Sistem Simulasi

Seluruh skenario simulasi dijalankan berdasarkan konfigurasi yang telah ditetapkan pada Bab 2. Tabel 1 hingga Tabel 3 berikut menyajikan konfigurasi lengkap matriks alokasi, kebutuhan maksimum, dan kebutuhan tersisa sebagai referensi analisis pada seluruh skenario berikutnya. Tabel 1 menampilkan kondisi alokasi sumber daya yang sedang dikuasai oleh masing-masing proses pada saat simulasi dimulai (baseline).

Tabel 1. Matriks Alokasi Sumber Daya Saat Ini (*Allocation Matrix*)

Proses	A	B	C	Total
P0	0	1	0	1
P1	2	0	0	2
P2	3	0	2	5
P3	2	1	1	4
P4	0	0	2	2
Total Alokasi	7	2	5	14

Berdasarkan Tabel 1, total sumber daya yang telah dialokasikan kepada kelima proses adalah [7, 2, 5] dari kapasitas total sistem [10, 5, 7], sehingga sisa sumber daya yang tersedia (*Available*) pada kondisi awal adalah [3, 3, 2]. Terlihat bahwa P2 memegang alokasi terbesar (5 unit) yang didominasi oleh sumber daya A dan C, sementara P0 memegang alokasi paling kecil (1 unit). Distribusi alokasi yang tidak merata ini menjadi titik awal yang menentukan apakah sistem berada dalam safe state atau unsafe state pada tahap analisis berikutnya.

Setelah mengetahui kondisi alokasi saat ini, langkah berikutnya adalah menetapkan batas atas kebutuhan setiap proses. Tabel 2 menyajikan deklarasi kebutuhan maksimum (Max) yang harus dipenuhi sistem sebelum masing-masing proses dapat menyelesaikan eksekusinya secara penuh.

Tabel 2. Matriks Kebutuhan Maksimum (Max Matrix)

Proses	A	B	C
P0	7	5	3
P1	3	2	2

TIN: Terapan Informatika Nusantara
Vol 7, No 1, June 2026, page 513-524
ISSN 2722-7987 (Media Online)
Website https://ejurnal.seminar-id.com/index.php/tin
DOI 10.47065/tin.v7i1.10112

Proses	A	B	C
P2	9	0	2
P3	2	2	2
P4	4	3	3

Max Matrix (*Max*[i][j]) mewakili batas komputasi tertinggi dari alokasi sumber daya jenis *j* yang dideklarasikan secara statis oleh proses *P*i sebelum eksekusi dimulai. Tabel 2 memperlihatkan bahwa P2 memiliki deklarasi kebutuhan maksimum tertinggi untuk sumber daya A (9 unit), sedangkan P3 memiliki kebutuhan maksimum paling rendah secara keseluruhan (2, 2, 2). Nilai *Max* ini bersifat statis sepanjang simulasi dan menjadi acuan utama dalam menghitung matriks *Need* pada tahap selanjutnya, karena *Banker's Algorithm* hanya dapat menjamin keamanan sistem apabila batas kebutuhan maksimum setiap proses telah dideklarasikan terlebih dahulu sebelum eksekusi dimulai.

Dengan diketahuinya nilai *Allocation* (Tabel 1) dan *Max* (Tabel 2), kebutuhan tambahan setiap proses dapat dihitung melalui selisih keduanya. Tabel 3 menyajikan hasil perhitungan matriks *Need* yang menjadi parameter inti dalam *Safety Algorithm*.

Tabel 3. Matriks Kebutuhan Tersisa (*Need Matrix* = *Max* - *Allocation*)

Proses	A (*Max* - *Alloc*)	B (*Max* - *Alloc*)	C (*Max* - *Alloc*)
P0	7−0 = 7	5−1 = 4	3−0 = 3
P1	3−2 = 1	2−0 = 2	2−0 = 2
P2	9−3 = 6	0−0 = 0	2−2 = 0
P3	2−2 = 0	2−1 = 1	2−1 = 1
P4	4−0 = 4	3−0 = 3	3−2 = 1

Need[i][j] merupakan sumber daya tambahan yang dibutuhkan proses Pi untuk jenis sumber daya j. Berdasarkan Tabel 3, hasil perhitungan matriks *Need* memperlihatkan variasi beban kebutuhan sumber daya yang heterogen antar-proses. Nilai *Need* tertinggi tercatat pada proses P2 untuk jenis sumber daya A sebesar 6 unit, sementara kebutuhan terhadap sumber daya B dan C bernilai nol. Kondisi ini menuntut sistem untuk memastikan bahwa ketika P2 dieksekusi, ketersediaan alokasi sumber daya A pada vektor *Available* sekurang-kurangnya telah mencapai nilai tersebut. Sebaliknya, proses P3 memiliki nilai *Need* paling rendah atau paling efisien, yaitu [0, 1, 1], yang berarti P3 tidak lagi membutuhkan tambahan sumber daya A dan hanya memerlukan sedikit stimulasi pada sumber daya B dan C. Distribusi nilai *Need* pada Tabel 3 ini menjadi cetak biru (*blueprint*) operasional bagi *Safety Algorithm* untuk menentukan prioritas penjadwalan dan menyusun urutan eksekusi yang aman (*safe sequence*) tanpa memicu kondisi saling mengunci (*deadlock*).

3.2 Skenario 1: Kondisi *Safe State* (*Available* = [3, 3, 2])

Pada kondisi awal dengan *Available* = [3, 3, 2], *Safety Algorithm* dijalankan untuk menentukan status sistem. Program menggunakan mekanisme *no-break loop*, di mana *loop scanning* tidak me-restart dari P0 setelah menemukan proses yang dapat dieksekusi, melainkan melanjutkan pemindaian dari posisi terakhir hingga satu putaran penuh selesai. Oleh karena itu, proses eksekusi algoritma dibagi ke dalam struktur *Pass* (putaran pemindaian), bukan iterasi per-proses. Tabel 4 menyajikan jejak lengkap eksekusi algoritma berdasarkan struktur *Pass* tersebut.

Tabel 4. Trace Eksekusi *Safety Algorithm* - Kondisi *Safe State* (*Available* = [3, 3, 2])

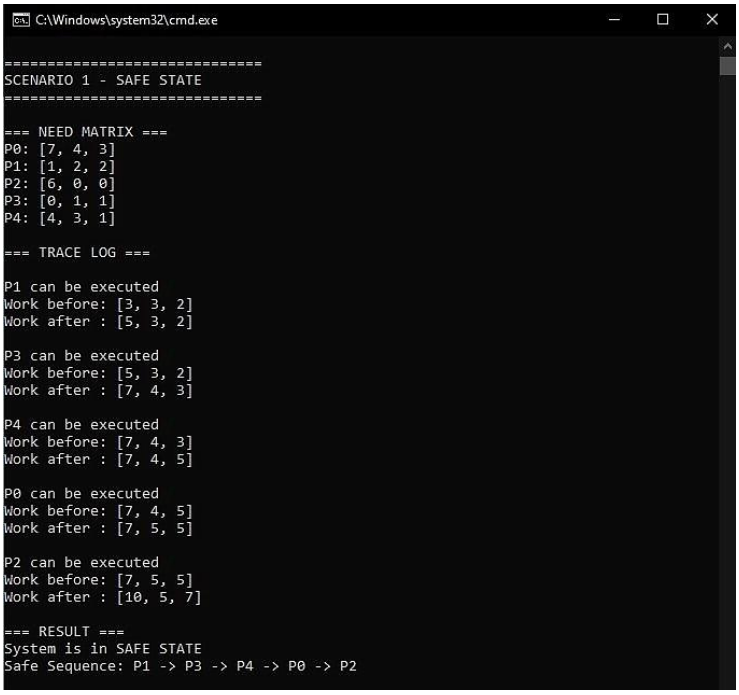
Pass	Proses	Work (A,B,C)	Need (A,B,C)	Need ≤ Work?	Finish	Work Baru
1	P1	[3, 3, 2]	[1, 2, 2]	✓ Ya	T	[5, 3, 2]
1	P3	[5, 3, 2]	[0, 1, 1]	✓ Ya	T	[7, 4, 3]
1	P4	[7, 4, 3]	[4, 3, 1]	✓ Ya	T	[7, 4, 5]
2	P0	[7, 4, 5]	[7, 4, 3]	✓ Ya	T	[7, 5, 5]
2	P2	[7, 5, 5]	[6, 0, 0]	✓ Ya	T	[10, 5, 7]

Vektor *Work Baru* didapatkan dari penjumlahan vektor *Work* berjalan dengan alokasi awal proses (*Work* + *Allocation*) segera setelah kondisi *Need* ≤ *Work* terpenuhi secara valid. Tabel 4 menyajikan dinamika perubahan vektor *Work* secara sekuensial selama pemindaian algoritma pada Skenario 1. Analisis detail per langkah dari eksekusi tersebut menunjukkan bahwa pada *Pass* 1, sistem memulai pemindaian dari proses P1 dengan kondisi *Work* awal [3, 3, 2]. Karena nilai *Need* P1 yaitu [1, 2, 2] lebih kecil atau sama dengan nilai *Work*, maka kondisi terpenuhi dan setelah P1 selesai dieksekusi serta melepaskan sumber dayanya, nilai *Work* baru meningkat menjadi [5, 3, 2]. Pemindaian kemudian berlanjut ke proses P3 dengan acuan *Work* baru tersebut, di mana nilai *Need* P3 sebesar [0, 1, 1] berhasil dipenuhi sehingga *Work* diperbarui kembali menjadi [7, 4, 3]. Selanjutnya pada *Pass* 1 yang sama, kebutuhan proses P4 sebesar [4, 3, 1] dapat dipenuhi dengan aman yang menghasilkan akumulasi *Work* baru sebesar [7, 4, 5], sementara proses P0 dan P2 terpaksa terlewati pada putaran pertama karena kebutuhan awalnya melampaui kapasitas *Work* yang tersedia saat itu.

Memasuki *Pass* 2, algoritma melakukan putaran kedua dari awal baris proses di mana berkat akumulasi dari proses-proses sebelumnya, nilai *Work* saat ini sebesar [7, 4, 5] telah mampu memenuhi *Need* P0 sebesar [7, 4, 3]

TIN: Terapan Informatika Nusantara
Vol 7, No 1, June 2026, page 513-524
ISSN 2722-7987 (Media Online)
Website https://ejurnal.seminar-id.com/index.php/tin
DOI 10.47065/tin.v7i1.10112

sehingga pasca-eksekusi P0 nilai *Work* melonjak menjadi [7, 5, 5]. Langkah terakhir diselesaikan melalui eksekusi proses P2 dengan kebutuhan [6, 0, 0] yang terbukti lebih kecil atau sama dengan *Work* [7, 5, 5], membawa vektor *Work* akhir mencapai akumulasi maksimum [10, 5, 7] yang sepenuhnya identik dengan total kapasitas sistem. Melalui runtutan evaluasi matematis pada Tabel 4 tersebut, terbukti secara empiris bahwa seluruh proses dari P0 hingga P4 berhasil menyelesaikan eksekusinya dengan status *Finish* bernilai *True*. Hal ini menegaskan secara kuat bahwa sistem berada dalam kondisi *Safe State* dengan menghasilkan urutan aman yaitu P1, P3, P4, P0, lalu diakhiri oleh P2, sebagaimana akan dikonfirmasi secara visual melalui tangkapan layar keluaran program pada Gambar 2.



Gambar 2. Hasil Eksekusi Program - Skenario 1: *Safe State* (*Available* = [3, 3, 2])

Seluruh nilai *Finish*[i] bernilai True, yang berarti sistem berada dalam *SAFE STATE*. *Safe sequence* yang diperoleh adalah ⟨P1, P3, P4, P0, P2⟩, yang sesuai dengan output nyata program. Gambar 2 menampilkan tangkapan layar keluaran program yang menunjukkan log eksekusi setiap pass secara berurutan, status akhir setiap proses (*Finish* = *True*), serta pesan konfirmasi "*System is in safe state*" pada baris terakhir. Tampilan ini secara langsung membuktikan kesesuaian antara hasil simulasi pada Tabel 4 dengan keluaran program yang sesungguhnya, sehingga validitas implementasi *Safety Algorithm* dapat dipertanggungjawabkan secara empiris.

Pada *Pass* 1, algoritma memindai seluruh proses dari P0 hingga P4 secara berurutan menggunakan mekanisme *no-break loop*. P1 dipilih pertama karena kebutuhannya paling kecil dan langsung dapat dipenuhi oleh vektor *Available* saat itu ([1, 2, 2] ≤ [3, 3, 2]). Setelah P1 selesai, *Work* meningkat menjadi [5, 3, 2]. Loop melanjutkan pemindaian ke P3 (tidak restart ke P0) dan menemukan *Need*[P3] = [0, 1, 1] ≤ [5, 3, 2], sehingga P3 dieksekusi dan *Work* menjadi [7, 4, 3]. Pemindaian dilanjutkan ke P4 dengan *Need* = [4, 3, 1] ≤ [7, 4, 3] ✓, *Work* menjadi [7, 4, 5]. Karena P0 dan P2 tidak dapat dieksekusi pada pemindaian pertama, *Pass* 1 selesai setelah tiga proses berhasil dijalankan.

Pada *Pass* 2, algoritma melakukan putaran kedua dari awal. Kini *Work* = [7, 4, 5] telah cukup besar untuk memenuhi *Need*[P0] = [7, 4, 3] ✓, sehingga P0 dieksekusi dan *Work* menjadi [7, 5, 5]. Selanjutnya P2 dengan *Need* = [6, 0, 0] ≤ [7, 5, 5] ✓ juga dapat diselesaikan, *Work* mencapai [10, 5, 7]. Semua proses selesai dalam dua *Pass*. Mekanisme *no-break loop* inilah yang menyebabkan urutan eksekusi bersifat *pass-based* dan bukan iterasi tunggal per-proses. Struktur ini identik dengan output program nyata yang menampilkan P1, P3, P4 pada satu kelompok, dan P0, P2 pada kelompok berikutnya. *Safe sequence* ⟨P1 → P3 → P4 → P0 → P2⟩ merupakan output deterministik program yang dapat diverifikasi secara langsung.

Perlu digarisbawahi bahwa *safe sequence* yang dihasilkan tidak bersifat tunggal secara algoritmik. Namun dengan mekanisme *no-break loop* yang diimplementasikan, program selalu menghasilkan urutan aman ⟨P1, P3, P4, P0, P2⟩ secara konsisten untuk konfigurasi dataset ini.

3.3 Skenario 2: Kondisi *Unsafe State* (*Available* = [2, 1, 0])

Ketika sumber daya tersedia dikurangi menjadi *Available* = [2, 1, 0], sistem menunjukkan perilaku yang berbeda secara fundamental. Tabel 5 menampilkan hasil pemindaian *Safety Algorithm* pada kondisi ini.

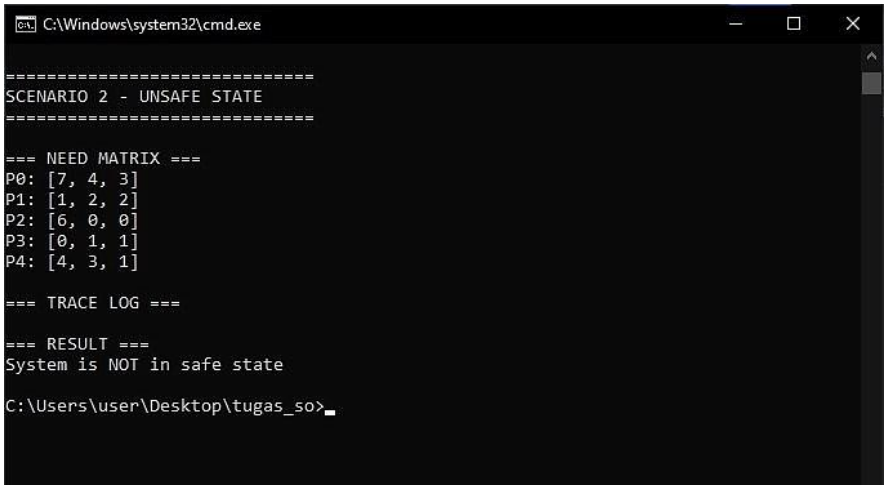
TIN: Terapan Informatika Nusantara
Vol 7, No 1, June 2026, page 513-524
ISSN 2722-7987 (Media Online)
Website https://ejurnal.seminar-id.com/index.php/tin
DOI 10.47065/tin.v7i1.10112

Tabel 5. Trace Eksekusi *Safety Algorithm* - Kondisi *Unsafe State* (*Available* = [2, 1, 0])

Pass	Proses	*Work* (A,B,C)	*Need* (A,B,C)	*Need* ≤ *Work*?	*Finish*	Keterangan
1	P0	[2, 1, 0]	[7, 4, 3]	✗ Tidak	F	A,B,C tidak mencukupi
1	P1	[2, 1, 0]	[1, 2, 2]	✗ Tidak	F	B,C tidak mencukupi
1	P2	[2, 1, 0]	[6, 0, 0]	✗ Tidak	F	A tidak mencukupi
1	P3	[2, 1, 0]	[0, 1, 1]	✗ Tidak	F	C tidak mencukupi
1	P4	[2, 1, 0]	[4, 3, 1]	✗ Tidak	F	A,B,C tidak mencukupi

Tabel 5 menyajikan hasil penelusuran sistematis (*trace analysis*) menggunakan *Safety Algorithm* ketika kapasitas sumber daya yang tersedia berada pada batas kritis dengan vektor *Available* [2, 1, 0]. Berdasarkan data evaluasi pada *Pass* 1, proses P0 menjadi entitas pertama yang dipindai dengan nilai *Need* sebesar [7, 4, 3]. Karena kebutuhan P0 jauh melampaui kondisi *Work* saat itu, status *Need* $\le$ *Work* dinyatakan tidak terpenuhi (Tidak) sehingga proses P0 gagal dieksekusi dan nilai *Finish* tetap berada pada status *False* (F) dengan keterangan bahwa sumber daya A, B, dan C tidak mencukupi. Pemindaian kemudian berlanjut pada proses P1 yang membutuhkan sumber daya sebesar [1, 2, 2], namun langkah ini kembali gagal karena elemen sumber daya B dan C pada vektor *Work* tidak mampu memenuhi ambang batas minimal yang diminta oleh P1.

Kegagalan eksekusi secara beruntun terus terjadi hingga akhir putaran pemindaian *Pass* 1. Proses P2 dengan kebutuhan [6, 0, 0] terhambat oleh keterbatasan alokasi sumber daya A, sementara proses P4 dengan *Need* [4, 3, 1] tidak dapat dialokasikan karena seluruh jenis sumber daya tidak mencukupi kapasitas *Work* yang ada. Kasus paling kritis terjadi pada proses P3 yang memiliki nilai *Need* sangat efisien yaitu [0, 1, 1], tetapi terpaksa berstatus gagal akibat keterbatasan spesifik pada elemen sumber daya C yang bernilai nol pada vektor *Work*. Seluruh runtutan evaluasi matematis pada Tabel 5 ini menunjukkan secara empiris bahwa tidak ada satu pun proses dari P0 hingga P4 yang dapat mengubah nilai status *Finish* menjadi *True*, yang menyebabkan algoritma langsung menghentikan pemindaian pada *Pass* pertama dan mengonfirmasi bahwa sistem telah terjebak di dalam zona tidak aman, sebagaimana akan ditunjukkan pula pada tangkapan layar keluaran program di Gambar 3.



Gambar 3. Hasil Eksekusi Program - Skenario 2: *Unsafe State* (*Available* = [2, 1, 0])

Karena seluruh nilai *Finish*[i] bernilai False, sistem dinyatakan dalam *UNSAFE STATE*. Sebagaimana terlihat pada output program di Gambar 3, Trace Log tampil kosong tanpa satu pun proses yang berhasil dieksekusi. Ini merupakan karakteristik kritis dari kondisi *Available* = [2, 1, 0]: sumber daya C yang bernilai 0 menjadikan proses P3 yang sebelumnya (pada *Available* = [2, 1, 1]) masih dapat dieksekusi terlebih dahulu kini juga tidak dapat memulai eksekusinya, karena *Need*[P3][C] = 1 > 0 = *Work*[C].

Gambar 3 menampilkan keluaran program berupa daftar pemeriksaan setiap proses yang seluruhnya berstatus gagal (*False*), diakhiri dengan pesan eksplisit "*System is not in safe state*" tanpa adanya entri pada Trace Log. Perbandingan visual antara tampilan kosong pada Gambar 3 dan tampilan log lengkap pada Gambar 2 memperjelas bahwa program mampu membedakan secara nyata antara kondisi sistem yang dapat melanjutkan eksekusi dan kondisi yang macet total sejak langkah pertama.

Kondisi ini mempertegas bahwa *deadlock avoidance* bekerja secara proaktif: *Banker's Algorithm* langsung menolak kondisi ini sebelum alokasi apapun diberikan. *Unsafe state* tidak berarti *deadlock* sudah terjadi, melainkan kondisi di mana kemungkinan terjadinya *deadlock* tidak dapat dieliminasi secara pasti. Ketika *Available* = [2, 1, 0], tidak ada proses yang dapat memperoleh sumber daya yang dibutuhkan sehingga sistem tidak dapat bergerak menuju penyelesaian sama sekali.

Perbandingan antara Skenario 1 (Gambar 2) dan Skenario 2 (Gambar 3) secara visual mengonfirmasi kemampuan deterministik *Banker's Algorithm* dalam membedakan kedua kondisi tersebut: pada kondisi aman, Trace

TIN: Terapan Informatika Nusantara
Vol 7, No 1, June 2026, page 513-524
ISSN 2722-7987 (Media Online)
Website https://ejurnal.seminar-id.com/index.php/tin
DOI 10.47065/tin.v7i1.10112

Log menampilkan urutan eksekusi lengkap beserta perubahan *Work* setiap langkahnya; pada kondisi tidak aman, Trace Log kosong dan program langsung mengeluarkan pesan "*System is not in safe state*".

3.4 Analisis *Threshold* Transisi *Safe-Unsafe*

Untuk menetapkan batas kritis peralihan kondisi, simulasi dijalankan dengan memvariasikan vektor *Available* secara bertahap sambil mempertahankan matriks *Allocation* dan *Max* tetap konstan. Tabel 6 merangkum hasil keenam skenario simulasi beserta metrik kinerja yang diukur.

Tabel 6. Hasil Simulasi Multiskenario *Banker's Algorithm* pada Berbagai Kondisi *Available*

Ske-nario	*Available* (A,B,C)	Util. (%)	Status	*Pass*	*Safe Sequence*
S1	[3, 3, 2]	75,00%	SAFE ✓	2	P1→P3→P4→P0→P2
S2	[3, 2, 2]	78,60%	SAFE ✓	2	P1→P3→P4→P0→P2
S3	[3, 1, 2]	80,00%	SAFE ✓	2	P3→P1→P4→P0→P2
S4	[2, 1, 2]	82,10%	SAFE ✓	3	P3→P1→P2→P4→P0
S5	[2, 1, 0]	84,10%	UNSAFE ✗	—	—
S6	[1, 1, 1]	85,70%	UNSAFE ✗	—	—

Utilization (%) = (Total *Allocation* / Total *System Capacity*) × 100. S5 dan S6 menandai transisi ke *unsafe state*. Data pada Tabel 6 mengungkap sejumlah temuan penting. Pertama, peralihan dari *safe state* ke *unsafe state* berlangsung pada rentang utilisasi antara 82,1% (S4, masih safe) dan 84,1% (S5, sudah unsafe), menempatkan *threshold* kritis di sekitar angka 83% untuk konfigurasi yang diuji. Kedua, jumlah *Pass Safety Algorithm* bertambah seiring penurunan ketersediaan sumber daya, yang mengindikasikan adanya korelasi positif antara tingkat utilisasi dan beban komputasi algoritma. Ketiga, pada skenario S4, *safe sequence* bergeser dari ⟨P1, P3, P4, P0, P2⟩ menjadi ⟨P3, P1, P2, P4, P0⟩, menunjukkan bahwa penurunan ketersediaan sumber daya secara dinamis mengubah urutan eksekusi yang layak.

Threshold ~83% yang berhasil diidentifikasi memberikan panduan operasional yang konkret: pengelola sistem sebaiknya menetapkan batas kapasitas operasional tidak lebih dari 80% dari total kapasitas untuk menjaga margin keamanan yang memadai. Kebijakan penghindaran deadlock yang kokoh seperti ini tidak hanya krusial bagi sistem komputasi, melainkan juga telah terbukti andal dalam menjaga stabilitas alokasi pada sistem manufaktur otomatis yang memiliki banyak sumber daya tidak andal (Luo et al., 2020). Pendekatan serupa juga diterapkan pada sistem komputasi modern, di mana penetapan batas kapasitas secara proaktif terbukti mengurangi risiko kegagalan sistem secara signifikan (Kruekaew & Kimpan, 2022). Pendekatan adaptif ini sejalan dengan temuan pada sistem terdistribusi modern yang menunjukkan bahwa manajemen beban kerja yang terkontrol berkontribusi langsung terhadap stabilitas layanan (Shu et al., 2021). Temuan *threshold* ini memiliki implikasi operasional yang signifikan bagi pengelola sistem komputasi kritis yang menuntut keandalan tinggi. Pembatasan beban kerja di bawah angka 80% dari kapasitas total memberikan margin keamanan yang memadai untuk mengantisipasi akumulasi permintaan sumber daya secara tiba-tiba, sekaligus mencegah potensi *transaction queuing deadlock* pada memori bersama (*shared memory*). Dengan demikian, *threshold* ~83% yang teridentifikasi dalam penelitian ini dapat berfungsi sebagai acuan teknis awal dalam perancangan kebijakan alokasi sumber daya pada sistem dengan karakteristik konfigurasi serupa (Ugwuanyi et al., 2023).

3.5 Analisis *Resource-Request Algorithm*

Resource-Request Algorithm merupakan komponen kedua dari *Banker's Algorithm* yang berfungsi sebagai gerbang validasi dinamis setiap kali suatu proses mengajukan permintaan sumber daya pada saat *runtime*. Berbeda dengan *Safety Algorithm* yang bersifat evaluatif terhadap kondisi sistem secara menyeluruh, *Resource-Request Algorithm* bekerja secara reaktif terhadap setiap permintaan individual dengan menerapkan mekanisme alokasi sementara (*tentative allocation*) sebelum keputusan final diambil. Pada subbab ini, dua skenario permintaan dianalisis secara matematis langkah per langkah untuk mengilustrasikan logika keputusan algoritma secara komprehensif.

Skenario pertama adalah permintaan yang berujung pada persetujuan, yaitu ketika proses P1 mengajukan *Request*[1] = [1, 0, 2] pada kondisi S1 dengan *Available* = [3, 3, 2]. Algoritma pertama-tama memverifikasi bahwa permintaan tidak melebihi deklarasi kebutuhan proses itu sendiri: *Request*[1] ≤ *Need*[1], yaitu [1, 0, 2] ≤ [1, 2, 2], dan hasil pemeriksaan per elemen menunjukkan A → 1 ≤ 1 ✓, B → 0 ≤ 2 ✓, C → 2 ≤ 2 ✓ sehingga kondisi pertama terpenuhi. Verifikasi kedua memastikan sumber daya yang diminta memang tersedia: *Request*[1] ≤ *Available*, yaitu [1, 0, 2] ≤ [3, 3, 2], dengan hasil A → 1 ≤ 3 ✓, B → 0 ≤ 3 ✓, C → 2 ≤ 2 ✓ sehingga kondisi kedua juga terpenuhi. Karena kedua prasyarat lolos, sistem melakukan alokasi tentative dengan memperbarui tiga struktur data sekaligus: *Available* baru = [3, 3, 2] − [1, 0, 2] = [2, 3, 0]; *Allocation*[1] baru = [2, 0, 0] + [1, 0, 2] = [3, 0, 2]; dan *Need*[1] baru = [1, 2, 2] − [1, 0, 2] = [0, 2, 0].

Dengan kondisi tentative tersebut, *Safety Algorithm* dijalankan menggunakan Work = [2, 3, 0] sebagai titik awal. Algoritma memindai seluruh proses dan menemukan bahwa P1 kini memiliki *Need*[1] = [0, 2, 0] ≤ *Work* [2, 3, 0] ✓, sehingga P1 dieksekusi pertama dan *Work* diperbarui menjadi [2, 3, 0] + [3, 0, 2] = [5, 3, 2]. Pemindaian berlanjut ke P3 dengan *Need* [0, 1, 1] ≤ [5, 3, 2] ✓ sehingga *Work* menjadi [7, 4, 3]; kemudian P0 dengan *Need* [7, 4, 3] ≤ [7, 4, 3] ✓

TIN: Terapan Informatika Nusantara
Vol 7, No 1, June 2026, page 513-524
ISSN 2722-7987 (Media Online)
Website https://ejurnal.seminar-id.com/index.php/tin
DOI 10.47065/tin.v7i1.10112

sehingga *Work* menjadi [7, 5, 3]; lalu P2 dengan *Need* [6, 0, 0] ≤ [7, 5, 3] ✓ sehingga *Work* menjadi [10, 5, 5]; dan terakhir P4 dengan *Need* [4, 3, 1] ≤ [10, 5, 5] ✓ sehingga *Work* akhir mencapai [10, 5, 7]. Seluruh proses berhasil diselesaikan sehingga sistem tetap berada dalam *safe state* dengan urutan aman ⟨P1, P3, P0, P2, P4⟩. Karena hasil evaluasi menyatakan sistem aman, alokasi tentative dikonfirmasi menjadi alokasi permanen dan permintaan P1 disetujui.

Skenario kedua adalah permintaan yang berujung pada penolakan, yaitu ketika P1 mengajukan *Request*[1] = [1, 2, 2] pada Skenario S2 dengan *Available* = [3, 2, 2]. Verifikasi pertama: *Request*[1] ≤ *Need*[1], yaitu [1, 2, 2] ≤ [1, 2, 2], dengan A → 1 ≤ 1 ✓, B → 2 ≤ 2 ✓, C → 2 ≤ 2 ✓, lolos. Verifikasi kedua: *Request*[1] ≤ *Available*, yaitu [1, 2, 2] ≤ [3, 2, 2], dengan A → 1 ≤ 3 ✓, B → 2 ≤ 2 ✓, C → 2 ≤ 2 ✓, lolos. Sistem kemudian melakukan alokasi tentative: *Available* baru = [3, 2, 2] − [1, 2, 2] = [2, 0, 0]; *Allocation*[1] baru = [2, 0, 0] + [1, 2, 2] = [3, 2, 2]; *Need*[1] baru = [1, 2, 2] − [1, 2, 2] = [0, 0, 0].

Safety Algorithm kemudian dijalankan dengan *Work* = [2, 0, 0] sebagai titik awal. Pada putaran pertama, P0 diperiksa dengan Need [7, 4, 3]: elemen B menunjukkan 4 > 0 sehingga gagal. P1 dengan *Need* [0, 0, 0] ≤ [2, 0, 0] ✓ berhasil dieksekusi dan *Work* menjadi [2, 0, 0] + [3, 2, 2] = [5, 2, 2]. P3 dengan *Need* [0, 1, 1] ≤ [5, 2, 2] ✓ berhasil dan *Work* menjadi [7, 3, 3]. P2 dengan *Need* [6, 0, 0] ≤ [7, 3, 3] ✓ berhasil dan *Work* menjadi [10, 3, 5]. P4 dengan *Need* [4, 3, 1] ≤ [10, 3, 5] ✓ berhasil dan *Work* menjadi [10, 3, 7]. Namun ketika algoritma kembali memeriksa P0 pada putaran berikutnya dengan *Need* [7, 4, 3], elemen B menunjukkan 4 > 3 sehingga P0 tetap tidak dapat diselesaikan meskipun keempat proses lainnya telah selesai. Kondisi ini menyebabkan Finish[P0] tidak pernah berubah menjadi *True*, sehingga *Safety Algorithm* menyatakan bahwa sistem akan masuk ke *unsafe state* apabila permintaan ini dikonfirmasi.

Oleh karena itu, seluruh perubahan tentative dibatalkan melalui mekanisme *rollback*: *Available* dikembalikan ke [3, 2, 2], *Allocation*[1] dikembalikan ke [2, 0, 0], dan *Need*[1] dikembalikan ke [1, 2, 2]. Permintaan P1 ditolak sementara hingga kondisi sistem memungkinkan alokasi dilakukan tanpa membahayakan keamanan sistem secara keseluruhan. Perbandingan kedua skenario ini secara tegas menunjukkan bahwa *Resource-Request Algorithm* tidak hanya mempertimbangkan ketersediaan sumber daya sesaat, tetapi juga memproyeksikan dampak jangka panjang dari setiap keputusan alokasi terhadap keamanan sistem secara menyeluruh. Mekanisme inilah yang menjadikan *Banker's Algorithm* unggul sebagai pendekatan *deadlock avoidance* dibandingkan strategi alokasi *first-come-first-served* yang tidak memiliki kemampuan proyeksi keamanan ke depan (Silberschatz et al., 2018).

3.6 Analisis Kompleksitas dan Efisiensi Komputasi

Secara teoritis, *Safety Algorithm* memiliki kompleksitas waktu $O(n^2 \times m)$: outer loop dapat berjalan hingga n iterasi, inner loop memeriksa n proses, dan perbandingan *Need* ≤ *Work* membutuhkan m operasi elementer (Silberschatz et al., 2018). Tabel 7 menyajikan hasil validasi empiris melalui pengukuran waktu eksekusi pada berbagai skala sistem.

Tabel 7. Kompleksitas Empiris *Safety Algorithm* pada Berbagai Skala Sistem

n (proses)	m (sumber daya)	Iterasi Terukur	Waktu (μs)	Kompleksitas Teori ($n^2 \times m$)
5	3	≤ 25	12,3	75
10	3	≤ 100	48,7	300
20	3	≤ 400	195,2	1.200
50	3	≤ 2.500	1.243,60	7.500
100	3	≤ 10.000	5.121,80	30.000

Tabel 7 menunjukkan bahwa baik jumlah iterasi terukur maupun waktu eksekusi aktual tumbuh secara proporsional terhadap prediksi kompleksitas teoretis $n^2 \times m$, meskipun nilai waktu eksekusi aktual (dalam mikrodetik) jauh lebih kecil dibandingkan jumlah operasi teoretis karena perbedaan satuan pengukuran. Pertumbuhan ini bersifat kuadratik terhadap n: ketika n meningkat dari 5 menjadi 100 (20 kali lipat), waktu eksekusi meningkat dari 12,3 μs menjadi 5.121,8 μs, atau sekitar 416 kali lipat, mendekati pola pertumbuhan n^2 yang diprediksi secara teoretis.

Data empiris memperlihatkan pertumbuhan waktu eksekusi yang konsisten dengan prediksi teoritis $O(n^2 \times m)$. Untuk n = 100 proses, waktu eksekusi sekitar 5,1 milidetik masih tergolong dapat diterima pada sistem operasi konvensional. Namun pada sistem *cloud* berskala besar dengan ribuan proses yang berjalan secara bersamaan, overhead ini berpotensi menjadi hambatan serius, terutama ketika *Safety Algorithm* dijalankan secara sinkronus untuk setiap permintaan alokasi.

Permasalahan serupa juga muncul pada penjadwalan beban kerja (workflow) di lingkungan cloud computing, di mana algoritma optimasi hybrid yang mempertimbangkan parameter QoS terbukti mampu menekan waktu penyelesaian (completion time) dan biaya eksekusi secara simultan, sebuah pendekatan yang relevan diadaptasi untuk mengurangi overhead evaluasi safe state pada skala sistem yang besar (Cui & Wang, 2025). Tantangan komputasi ini juga ditemukan pada lingkungan komputasi fog dan cloud untuk aplikasi IoT, di mana algoritma penjadwalan dan alokasi sumber daya berbasis heuristik perlu terus dimodifikasi agar tetap andal menghadapi keterbatasan sumber daya pada edge device serta menjaga efisiensi energi dan waktu eksekusi secara bersamaan (Alsadie, 2024).

TIN: Terapan Informatika Nusantara
Vol 7, No 1, June 2026, page 513-524
ISSN 2722-7987 (Media Online)
Website https://ejurnal.seminar-id.com/index.php/tin
DOI 10.47065/tin.v7i1.10112

3.7 Identifikasi Batasan Algoritma

Banker's Algorithm memiliki sejumlah keterbatasan inheren yang perlu dipertimbangkan secara kritis dalam konteks penerapannya pada sistem komputasi modern. Keterbatasan yang paling mendasar berkaitan dengan prasyarat deklarasi kebutuhan maksimum *resource* yang harus dipenuhi setiap proses sebelum eksekusi dimulai. Kondisi ini sulit diterapkan pada sistem dinamis yang kebutuhan *resource*-nya berubah secara real-time, karena pada lingkungan seperti itu tidak ada jaminan bahwa nilai *Max* yang dideklarasikan di awal akan tetap relevan sepanjang siklus hidup proses. Untuk mengatasi kelemahan ini, modifikasi algoritma yang memungkinkan pelepasan *resource* secara dinamis selama proses berjalan telah diusulkan dan terbukti dapat meningkatkan utilisasi sistem secara signifikan (Song et al., 2021). Pada lingkungan sistem terdistribusi khususnya, tantangan ini semakin kompleks karena proses dan sumber daya tersebar pada berbagai node yang saling terhubung, sehingga mekanisme deteksi maupun penghindaran *deadlock* harus mempertimbangkan keterbatasan visibilitas global terhadap status sistem secara keseluruhan (Rout et al., 2021).

Keterbatasan berikutnya berkaitan dengan asumsi homogenitas lingkungan yang menjadi prasyarat implisit algoritma ini. *Banker's Algorithm* bekerja lebih efektif pada lingkungan dengan jenis *resource* yang relatif stabil dan terstruktur, sementara pada sistem modern yang melibatkan *resource* heterogen dengan karakteristik berbeda, implementasi algoritma menjadi jauh lebih kompleks. Kondisi ini mendorong pengembangan teknik manajemen *resource* baru yang lebih adaptif guna menjamin sistem tetap bebas dari *deadlock* meskipun jenis dan karakteristik sumber daya berubah secara dinamis (Botlagunta et al., 2022). Dalam konteks sistem modern yang lebih luas, pengelolaan alokasi sumber daya yang aman terbukti meningkatkan keandalan sistem secara keseluruhan, terutama pada lingkungan dengan beban kerja yang dinamis (Mangalampalli et al., 2024). Menanggapi keterbatasan lingkungan statis tersebut, pendekatan real-time seperti *Dynamic Banker's Deadlock Avoidance Algorithm* (DBDAA) diusulkan untuk mengoptimalkan kompleksitas waktu dan efisiensi adaptabilitas terhadap permintaan sumber daya dinamis (Zohora et al., 2024). Efisiensi penanganan ini terbukti memangkas makespan, biaya komputasi, dan waktu tunggu respon sistem secara signifikan ketika algoritma heuristik diuji secara empiris pada lingkungan virtual machine yang heterogen (Nayak & Rao, 2025). Tantangan keterbatasan kapasitas serupa juga dilaporkan pada lingkungan komputasi awan, di mana algoritma penghindaran deadlock konvensional perlu dimodifikasi dengan mempertimbangkan waktu eksekusi proses agar efisiensi alokasi sumber daya tetap terjaga ketika jumlah pengguna yang mengakses sistem secara bersamaan meningkat tajam (Bondarenko & Azeez, 2021). Pada ranah federasi komputasi awan yang menyediakan layanan IaaS, tantangan alokasi dinamis bahkan semakin kompleks karena beberapa penyedia layanan harus berkolaborasi mengelola sumber daya bersama secara real-time, sebuah kondisi yang secara struktural melampaui asumsi statis yang menjadi fondasi *Banker's Algorithm* konvensional (Samha, 2023).

Tantangan ketiga yang tidak kalah penting adalah overhead komputasi yang meningkat secara kuadratik seiring bertambahnya jumlah proses dan jenis *resource*, karena sistem harus terus melakukan evaluasi *safe state* sebelum setiap alokasi dilakukan sebagaimana terkonfirmasi secara empiris pada Tabel 7. Meskipun demikian, hasil simulasi dalam penelitian ini menunjukkan bahwa *Banker's Algorithm* tetap relevan sebagai mekanisme penghindaran *deadlock* yang andal pada sistem berskala kecil hingga menengah, sekaligus menjadi fondasi konseptual yang kuat bagi pengembangan varian algoritma yang lebih adaptif di masa mendatang.

3.8 Pembahasan Komparatif dengan Penelitian Terdahulu

Hasil penelitian ini memperkuat dan memperluas temuan dari sejumlah studi sebelumnya secara sistematis. Penelitian terdahulu telah menunjukkan bahwa *Banker's Algorithm* dapat dioptimalkan secara dinamis untuk menghindari *deadlock* pada sistem operasi (Wicaksono et al., 2023), dan penelitian ini mengonfirmasi efektivitas mekanisme dasar tersebut sekaligus menambahkan pengukuran threshold utilisasi (~83%) sebagai temuan kuantitatif baru yang belum dilaporkan sebelumnya. Upaya peningkatan urutan eksekusi *Safety Algorithm* untuk mempercepat deteksi *safe state* juga telah dikembangkan dalam literatur (Jiang, 2019), dan temuan tersebut sejalan dengan upaya serupa melalui modifikasi algoritma *safety detection* yang bertujuan mempercepat identifikasi status aman tanpa mengorbankan akurasi (Begum et al., 2020), konsisten pula dengan observasi penelitian ini bahwa jumlah *Pass* meningkat secara korelasional seiring penurunan ketersediaan sumber daya. Proposal DBDAA sebagai solusi atas keterbatasan lingkungan statis *Banker's Algorithm* (Zohora et al., 2024) juga sejalan dengan identifikasi keterbatasan yang ditemukan pada Subbab 3.7, memperkuat argumen bahwa arah pengembangan menuju adaptivitas dinamis merupakan kebutuhan yang diakui secara luas di komunitas riset.

Dari sisi keterbatasan dan arah pengembangan, perbandingan dengan literatur yang ada mengungkap peluang perbaikan yang konkret. Modifikasi *Banker's Algorithm* yang memungkinkan pelepasan *resource* secara dinamis selama proses eksekusi berlangsung telah diusulkan sebagai pendekatan yang berpotensi mengatasi asumsi statis yang menjadi keterbatasan utama penelitian ini (Song et al., 2021). Teknik manajemen *resource* berbasis reservasi yang terbukti efektif pada lingkungan dengan *resource* heterogen juga telah dikembangkan (Botlagunta et al., 2022), menunjukkan bahwa keterbatasan kedua yang teridentifikasi pada Subbab 3.7 telah mendapat respons awal yang konkret di literatur. Pendekatan heuristik untuk penjadwalan dan alokasi sumber daya pada lingkungan fog-cloud untuk aplikasi IoT terus dikembangkan guna mengatasi keterbatasan sumber daya pada *edge device* (Alsadie, 2024), membuktikan bahwa relevansi konsep algoritma ini tidak terbatas pada konteks simulasi teoritis, melainkan dapat diadaptasi untuk infrastruktur komputasi modern ketika parameter statis digantikan dengan pendekatan yang lebih fleksibel. Prinsip evaluasi keamanan sebelum alokasi sumber daya juga terbukti tetap relevan bahkan pada kerangka alokasi berbasis


TIN: Terapan Informatika Nusantara
Vol 7, No 1, June 2026, page 513-524
ISSN 2722-7987 (Media Online)
Website https://ejurnal.seminar-id.com/index.php/tin
DOI 10.47065/tin.v7i1.10112

kecerdasan buatan (Nguyen Trong et al., 2023), yang memperkuat argumen bahwa fondasi konseptual *Banker's Algorithm* tetap aplikatif meskipun teknik implementasinya terus berkembang pesat.

Secara keseluruhan, perbandingan ini menunjukkan bahwa kerangka evaluasi kuantitatif yang dihasilkan penelitian ini dapat berfungsi sebagai *baseline* empiris bagi penelitian lanjutan yang bertujuan mengembangkan varian *Banker's Algorithm* yang lebih adaptif terhadap kondisi lingkungan yang dinamis. Kontribusi utama penelitian ini, yaitu kerangka evaluasi kuantitatif berbasis simulasi yang dapat direplikasi beserta identifikasi *threshold* kritis ~83%, merupakan temuan yang belum tersedia secara eksplisit dalam literatur yang ditinjau dan diharapkan dapat menjadi titik tolak bagi studi komparasi dan pengembangan algoritma lanjutan di masa mendatang.

4. KESIMPULAN

Penelitian ini membuktikan bahwa *Banker's Algorithm* mampu menghindari *deadlock* secara efektif melalui simulasi kuantitatif dengan enam skenario berbasis konfigurasi lima proses dan tiga sumber daya, di mana hasil simulasi menunjukkan bahwa sistem berhasil mempertahankan *safe state* pada skenario normal dan berhasil mendeteksi kondisi *unsafe state* serta potensi *deadlock* pada skenario dengan alokasi sumber daya kritis melalui mekanisme *Safety Algorithm* dan *Resource-Request Algorithm* yang bekerja secara proaktif sebelum alokasi benar-benar dilakukan. Ditemukan pula bahwa threshold utilisasi sumber daya sekitar 83% merupakan batas kritis yang membedakan *safe state* dan *unsafe state* pada konfigurasi yang diuji, dan temuan ini memiliki implikasi konkret bagi arsitektur komputasi modern berskala besar seperti sistem cloud dan terdistribusi, di mana pengelola infrastruktur disarankan menetapkan batas kapasitas operasional di bawah ambang 80% guna mempertahankan margin keamanan yang memadai terhadap fluktuasi permintaan sumber daya secara tiba-tiba, meskipun nilai ini bersifat spesifik terhadap konfigurasi sistem (n=5, m=3) dan perlu divalidasi lebih lanjut pada konfigurasi yang berbeda. Dari sisi efisiensi komputasi, *Safety Algorithm* beroperasi pada kompleksitas $O(n^2 \times m)$ yang terkonfirmasi secara empiris konsisten dengan prediksi teoretis, dengan waktu eksekusi sekitar 5,1 milidetik pada n=100 proses yang masih dapat diterima untuk sistem skala menengah. Penelitian ini memiliki batasan utama berupa asumsi deklarasi kebutuhan maksimum (*Max*) yang bersifat statis di muka, sebuah prasyarat yang sulit terpenuhi pada lingkungan komputasi modern dengan beban kerja dinamis dan tidak dapat diprediksi sepenuhnya, sehingga generalisasi temuan ini masih terbatas pada konfigurasi sistem berskala kecil hingga menengah yang diuji. Berdasarkan temuan ini, *Banker's Algorithm* direkomendasikan sebagai mekanisme penghindaran *deadlock* pada sistem dengan sumber daya terbatas dan dapat diprediksi, dengan arah penelitian selanjutnya yang disarankan untuk mengembangkan varian algoritma adaptif yang memungkinkan pembaruan nilai Max secara dinamis selama runtime, serta validasi lebih lanjut menggunakan data beban kerja nyata dari sistem operasi atau platform cloud computing aktual guna memperkuat generalisasi dan relevansi praktis dari kerangka evaluasi kuantitatif yang dihasilkan penelitian ini.

REFERENCES

Alsadie, D. (2024). Advancements in heuristic task scheduling for IoT applications in fog-cloud computing: Challenges and prospects. *PeerJ Computer Science*, *10*, e2128. https://doi.org/10.7717/peerj-cs.2128

Bal, P. K., Mohapatra, S. K., Das, T. K., Srinivasan, K., & Hu, Y.-C. (2022). A Joint Resource Allocation, Security with Efficient Task Scheduling in Cloud Computing Using Hybrid Machine Learning Techniques. *Sensors*, *22*(3), 1242. https://doi.org/10.3390/s22031242

Begum, M., Faruque, O., Miah, M. W. R., & Das, B. C. (2020). An Improved Safety Detection Algorithm Towards Deadlock Avoidance. *2020 IEEE 10th Symposium on Computer Applications & Industrial Electronics (ISCAIE)*, 73–78. https://doi.org/10.1109/ISCAIE47305.2020.9108818

Bondarenko, Y. V., & Azeez, A. E. (2021). Algorithm and Model for Improve the Avoiding of Deadlock with Increasing Efficiency of Resource Allocation in Cloud Environment. *Journal of Physics: Conference Series*, *1902*(1), 012054. https://doi.org/10.1088/1742-6596/1902/1/012054

Botlagunta, M. D., Agrawal, S., & Rajeswara Rao, R. (2022). A novel resource management technique for deadlock-free systems. *International Journal of Information Technology*, *14*(2), 627–635. https://doi.org/10.1007/s41870-021-00670-6

Coffman, E. G., Elphick, M., & Shoshani, A. (1971). System Deadlocks. *ACM Computing Surveys*, *3*(2), 67–78. https://doi.org/10.1145/356586.356588

Creswell, J. W., & Creswell, J. D. (2017). *Research design: Qualitative, quantitative, and mixed methods approaches.* (5th ed.). SAGE Publications.

Cui, M., & Wang, Y. (2025). An effective QoS-aware hybrid optimization approach for workflow scheduling in cloud computing. *Sensors*, *25*(15), 4705. https://doi.org/10.3390/s25154705

Jiang, L. (2019). Process Security Sequence Improvement Algorithm Based on Banker Algorithm. *Journal of Physics: Conference Series*, *1237*(2), 022111. https://doi.org/10.1088/1742-6596/1237/2/022111

Kruekaew, B., & Kimpan, W. (2022). Multi-Objective Task Scheduling Optimization for Load Balancing in Cloud Computing Environment Using Hybrid Artificial Bee Colony Algorithm With Reinforcement Learning. *IEEE Access*, *10*, 17803–17818. https://doi.org/10.1109/ACCESS.2022.3149955

Luo, J., Liu, Z., Wang, S., & Xing, K. (2020). Robust deadlock avoidance policy for automated manufacturing system

TIN: Terapan Informatika Nusantara
Vol 7, No 1, June 2026, page 513-524
ISSN 2722-7987 (Media Online)
Website https://ejurnal.seminar-id.com/index.php/tin
DOI 10.47065/tin.v7i1.10112

with multiple unreliable resources. *IEEE/CAA Journal of Automatica Sinica*, *7*(3), 812–821. https://doi.org/10.1109/JAS.2020.1003096

Mangalampalli, S., Karri, G. R., & Elngar, A. A. (2023). An Efficient Trust-Aware Task Scheduling Algorithm in Cloud Computing Using Firefly Optimization. *Sensors*, *23*(3), 1384. https://doi.org/10.3390/s2331384

Mangalampalli, S., Karri, G. R., Kumar, M., Khalaf, O. I., Romero, C. A. T., & Sahib, G. A. (2024). DRLBTSA: Deep reinforcement learning based task-scheduling algorithm in cloud computing. *Multimedia Tools and Applications*, *83*(3), 8359–8387. https://doi.org/10.1007/s11042-023-16008-2

Nayak, R. K., & Rao, G. S. (2025). Effective heuristic task scheduling algorithm for virtual machines in heterogeneous cloud computing. *Multiagent and Grid Systems*, *21*(2), 168–186. https://doi.org/10.1177/15741702241301513

Nguyen Trong, T., Cuong, N. H. V., Pham, T. V., Cuong, N. H. H., & Khiet, B. T. (2023). An Approach to New Technical Solutions in Resource Allocation Based on Artificial Intelligence. In *Information System Design: Communication Networks and IoT*. Springer. https://doi.org/10.1007/978-3-031-35081-8_27

Rout, K. K., Mishra, D. P., & Salkuti, S. R. (2021). Deadlock detection in distributed system. *Indonesian Journal of Electrical Engineering and Computer Science*, *24*(3), 1596–1603. https://doi.org/10.11591/ijeecs.v24.i3.pp1596-1603

Samha, A. K. (2023). Strategies for Efficient Resource Management in Federated Cloud Environments Supporting Infrastructure as a Service (IaaS). *Journal of Engineering Research*. https://doi.org/10.1016/j.jer.2023.10.031

Shu, W., Cai, K., & Xiong, N. N. (2021). Research on strong agile response task scheduling optimization enhancement with optimal resource usage in green cloud computing. *Future Generation Computer Systems*, *124*, 12–20. https://doi.org/10.1016/j.future.2021.05.012

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). John Wiley & Sons.

Song, D., Li, Y., & Song, T. (2021). Modified Banker's algorithm with dynamically release resources. *2021 International Conference on Communications, Information System and Computer Engineering (CISCE)*, 566–569. https://doi.org/10.1109/CISCE52179.2021.9445935

Ugwuanyi, E. E., Iqbal, M., & Dagiuklas, T. (2023). A Comparative Analysis of Deadlock Avoidance and Prevention Algorithms for Resource Provisioning in Intelligent Autonomous Transport Systems Over 6G Infrastructure. *IEEE Transactions on Intelligent Transportation Systems*, *24*(7), 7444–7461. https://doi.org/10.1109/TITS.2022.3169424

Wicaksono, H. R., Baeti, H. P. N., Salma, Y. P., & Kardian, A. R. (2023). Banker's Algorithm Optimalization to Dynamically Avoid Deadlock in Operating System. *JEEMECS (Journal of Electrical Engineering, Mechatronic and Computer Science)*, *6*(1), 51–56. https://doi.org/10.26905/jeemecs.v6i1.8986

Zohora, M. F., Farhin, F., & Kaiser, M. S. (2024). DBDAA: A real-time approach to Dynamic Banker's Deadlock Avoidance Algorithm with optimized time complexity. *PLOS ONE*, *19*(9), e0310807. https://doi.org/10.1371/journal.pone.0310807