Jurnal Komputer, Informasi dan Teknologi Vol: 5, No 1, 2025, Page: 1-15

Optimalisasi Sinkronisasi Proses: Analisis Efektivitas *Semaphore*, *Mutex*, dan *Monitor*

Almira Rahma Fadhila*, Alvyn Hadrian Nugraha, Cynthia Hasna Mazaya, Mochammad Syahrul Ramadhan, Jajang Kusnendar

Universitas Pendidikan Indonesia, Bandung, Indonesia

Abstrak: Penelitian ini bertujuan untuk menganalisis dan membandingkan efektivitas tiga mekanisme sinkronisasi dalam sistem operasi yaitu *semaphore*, *mutex*, dan *monitor* dalam mencegah *race condition* serta menjaga integritas data. Metode yang digunakan adalah studi pustaka dengan pendekatan kualitatif deskriptif komparatif. Data dikumpulkan dari literatur ilmiah yang relevan dalam lima tahun terakhir dan dianalisis menggunakan teknik analisis isi. Penilaian dilakukan berdasarkan aspek kemudahan implementasi, efektivitas dalam menjaga konsistensi, risiko *deadlock* dan *starvation*, serta kesesuaian terhadap konteks sistem operasi. Hasil penelitian menunjukkan bahwa *monitor* paling stabil dan mudah digunakan dalam sistem yang kompleks, *mutex* unggul dalam efisiensi pada sistem dengan latensi rendah, sementara *semaphore* memberikan fleksibilitas lebih tinggi tetapi lebih rentan terhadap kesalahan logika. Simpulan dari penelitian ini adalah bahwa tidak ada mekanisme yang paling unggul secara mutlak, sehingga pemilihan mekanisme sinkronisasi harus disesuaikan dengan kebutuhan teknis, karakteristik sistem, dan konteks implementasi. Temuan ini memberikan panduan konseptual bagi pengembang dalam memilih strategi sinkronisasi yang tepat untuk meningkatkan keandalan dan efisiensi sistem operasi.

Kata Kunci: *Mutual Exclusion, Mutex, Monitor, Sinkronisasi* Proses, *Semaphore*

DOI: https://doi.org/10.53697/jkomitek.v5i1.2595
*Correspondence: Almira Rahma Fadhila
Email: almira@upi.edu

Received:27-04-2025
Accepted: 27-05-2025
Published: 27-06-2025

Abstract: *This study aims to analyze and compare the effectiveness of three synchronization mechanisms in operating systems, namely semaphores, mutexes, and monitors, in preventing race conditions and maintaining data integrity. The method used is a literature study with a descriptive comparative qualitative approach. Data was collected from relevant scientific literature in the last five years and analyzed using content analysis techniques. The evaluation was based on aspects of ease of implementation, effectiveness in maintaining consistency, risk of deadlock and starvation, and suitability for the operating system context. The results of the study show that monitors are the most stable and easy to use in complex systems, mutexes excel in efficiency in low-latency systems, while semaphores provide greater flexibility but are more prone to logical errors. The conclusion of this study is that no single mechanism is absolutely superior, so the selection of synchronization mechanisms must be tailored to technical requirements, system characteristics, and implementation context. These findings provide conceptual guidance for developers in selecting the appropriate synchronization strategy to enhance the reliability and efficiency of operating systems.*

Keywords: *Mutual Exclusion, Mutex, Monitor, Process Synchronization, semaphore*

Pendahuluan

Dalam era komputasi modern, sistem komputer semakin mengandalkan eksekusi paralel melalui *multitasking* dan *multithreading* untuk meningkatkan kinerja dan efisiensi. *Multitasking* memungkinkan sistem menjalankan beberapa proses secara bergantian dalam waktu yang sangat singkat, sehingga meningkatkan pemanfaatan CPU dan responsivitas sistem. Sementara itu, *multithreading* memungkinkan sebuah proses dibagi menjadi beberapa *thread* yang dapat dieksekusi secara bersamaan, mempercepat penyelesaian tugas-tugas komputasi kompleks dan meningkatkan performa aplikasi.

Meskipun memberikan keuntungan signifikan, eksekusi paralel juga menimbulkan tantangan serius, terutama dalam menjaga konsistensi data dan integritas sistem. Salah satu permasalahan yang sering muncul adalah *race condition*, yaitu kondisi ketika hasil akhir eksekusi bergantung pada urutan serta waktu akses *thread* terhadap sumber daya bersama. Keadaan ini dapat menyebabkan perilaku sistem yang tidak deterministik dan sulit direproduksi (Wang et al., 2022). Dalam sistem paralel, setiap *thread* idealnya mengakses *critical section* secara terkoordinasi guna memastikan konsistensi data dan keluaran yang dapat diprediksi. Namun dalam praktiknya, *race condition* sering terjadi ketika dua atau lebih *thread* memasuki *critical section* secara bersamaan, yang berpotensi memicu kesalahan sulit dilacak dan membahayakan integritas data (Chaturvedi et al., 2025).

Untuk mengatasi masalah tersebut, diperlukan mekanisme sinkronisasi yang mampu menjamin bahwa hanya satu proses atau *thread* yang dapat mengakses sumber daya bersama pada satu waktu tertentu, atau yang dikenal dengan konsep *mutual exclusion* (Dhoked & Mittal, 2022). Tiga mekanisme yang lazim digunakan dalam penerapan *mutual exclusion* adalah *semaphore*, *mutex*, dan *monitor*. Masing-masing memiliki karakteristik tersendiri dalam hal kemudahan implementasi, efisiensi, dan efektivitas dalam mencegah konflik akses. Implementasi *mutual exclusion* melalui ketiga mekanisme tersebut telah menjadi fokus utama dalam pengembangan sistem operasi dan aplikasi paralel untuk menjaga stabilitas sistem dan menghindari kondisi konflik (Chan & Woelfel, 2021). Oleh karena itu, pemilihan mekanisme sinkronisasi yang tepat tidak hanya memengaruhi konsistensi data, tetapi juga berdampak pada kinerja sistem secara keseluruhan, termasuk efisiensi waktu eksekusi dan pemanfaatan sumber daya (Kode & Oyemade, 2024).

Sejumlah penelitian sebelumnya telah mengkaji mekanisme sinkronisasi ini dalam berbagai konteks. Salah satunya (Sakthivel & Sreeja, 2024) yang mengevaluasi penerapan *semaphore* dan *mutex* pada sistem *real-time* berbasis FreeRTOS di Arduino. Hasil eksperimen menunjukkan bahwa kedua mekanisme tersebut mampu meningkatkan reliabilitas dan konsistensi data, namun memiliki *trade-off* berbeda, seperti dalam hal latensi, *blocking*, dan pengaturan prioritas *task*. Sementara itu, Suveetha et al. (2020) menekankan dasar konseptual serta praktik implementasi *semaphore* melalui penyelesaian masalah klasik seperti *producer-consumer* dan *reader-writer*, yang tetap menjadi relevan dalam pengembangan sistem paralel modern.

Dari sisi performa berbasis simulasi, Ehis (2024) menemukan bahwa *mutex* lebih unggul dalam *throughput* pada sistem dengan latensi rendah, sedangkan *semaphore* lebih stabil ketika terjadi delay komunikasi antar-*thread*. Di sisi lain, *monitor* dinilai paling

efektif dalam sistem *multithreaded* kompleks, terutama dalam lingkungan basis data, karena kemampuannya menangani pola interaksi *read-modify-write* secara berulang dengan lebih konsisten (Turck, 2022).

Penelitian ini menghadirkan kebaruan ilmiah dengan menyajikan analisis komparatif menyeluruh terhadap tiga mekanisme sinkronisasi utama, yaitu *semaphore*, *mutex*, dan *monitor*, dalam satu kerangka evaluatif yang terstandar. Analisis dilakukan dengan mempertimbangkan berbagai aspek seperti kemudahan implementasi, efektivitas dalam mencegah *race condition*, risiko *deadlock* dan *starvation*, serta kesesuaian terhadap konteks sistem operasi yang berbeda, termasuk sistem *embedded*, *real-time*, dan *multithreaded*.

Berbeda dari penelitian terdahulu yang umumnya hanya membandingkan dua mekanisme atau menitikberatkan pada studi kasus terbatas, kajian ini mengintegrasikan tinjauan pustaka konseptual dan analisis deskriptif-komparatif secara sistematis untuk memberikan rekomendasi yang aplikatif. Kebaruan lainnya terletak pada fokus evaluasi terhadap resiliensi terhadap kesalahan implementasi seperti penguncian tidak seimbang, yang jarang diulas secara eksplisit dalam studi sebelumnya.

Permasalahan utama yang diangkat dalam penelitian ini berkaitan dengan efektivitas mekanisme sinkronisasi dalam mengelola akses ke *critical section* pada sistem *multitasking* dan *multithreading*. Meskipun berbagai mekanisme seperti *semaphore*, *mutex*, dan *monitor* telah digunakan secara luas dalam sistem operasi modern, masing-masing memiliki karakteristik, kelebihan, dan kelemahan yang berbeda. Oleh karena itu, penting untuk mengetahui bagaimana ketiga mekanisme tersebut mampu mencegah *race condition* dan menjaga integritas data secara konsisten, khususnya dalam lingkungan sistem dengan beban tinggi atau tingkat kontensi yang kompleks. Selain itu, perlu ditelaah seberapa efisien masing-masing mekanisme dalam konteks performa teknis, termasuk *throughput*, latensi akses, dan efisiensi manajemen sumber daya. Permasalahan lainnya adalah bagaimana tingkat ketahanan mekanisme-mekanisme tersebut terhadap kesalahan implementasi seperti penguncian tidak seimbang (*unbalanced locking*), yang sering terjadi dalam pengembangan sistem *multithreaded*. Penelitian ini juga mempertanyakan mekanisme mana yang paling optimal untuk diterapkan dalam berbagai konteks sistem nyata, seperti sistem *embedded*, *real-time*, maupun sistem multiprosesor skala besar.

Hipotesis awal yang diajukan dalam penelitian ini adalah bahwa *monitor* akan menunjukkan stabilitas terbaik dalam kondisi kontensi tinggi karena struktur internalnya yang terintegrasi. *Mutex* diperkirakan unggul dalam lingkungan latensi rendah, sementara *semaphore* dinilai fleksibel namun lebih rentan terhadap kesalahan logika. *Monitor* juga diasumsikan paling tangguh terhadap kesalahan implementasi seperti *misuse unlock*.

Penelitian ini bertujuan untuk menganalisis dan membandingkan efektivitas *semaphore*, *mutex*, dan *monitor* dalam hal performa, kestabilan, dan resiliensi. Selain itu, penelitian ini juga bertujuan memberikan rekomendasi mekanisme sinkronisasi yang paling sesuai berdasarkan konteks sistem dan kebutuhan teknis aplikasi.

Metodologi

Penelitian ini merupakan penelitian kualitatif dengan pendekatan *literature review* atau studi pustaka yang mencakup penelaahan secara mendalam serta krisis terhadap berbagai sumber literatur yang relevan. Pendekatan ini dipilih karena memungkinkan peneliti untuk mengumpulkan, mengkaji, dan menganalisis berbagai sumber ilmiah untuk memperoleh pemahaman yang menyeluruh mengenai topik yang dikasi. Fokus utama dari kajian ini adalah pada mekanisme *mutual exclusion* seperti *Semaphore*, *Mutex*, dan *Monitor* dalam konteks pengelolaan sinkronisasi pada sistem komputer.

Sumber data penelitian ini diperoleh dari artikel jurnal nasional dan internasional, buku digital, dan sumber lainnya yang terpercaya. Adapun kriteria inklusi dalam pemilihan literatur meliputi publikasi yang diterbitkan dalam lima tahun terakhir (2021-2025) yang memiliki relevansi dengan topik *mutual exclusion* dan sinkronisasi, serta menyajikan informasi teknis atau hasil analisis terhadap mekanisme *Semaphore*, *Mutex*, dan *Monitor*.

Prosedur pengumpulan data dilakukan melalui beberapa tahapan. Pertama, mengidentifikasi kata kunci yang relevan untuk mencari literatur dari berbagai sumber data. Selanjutnya, menyeleksi hasil pencarian dengan menerapkan kriteria inklusi dan eksklusi untuk memastikan relevansi dan kualitas sumber data yang digunakan. Literatur yang terpilih selanjutnya diklasifikasikan berdasarkan topik, pendekatan, serta hasil temuan yang relevan, lalu dianalisis secara sistematis.

Teknik analisis data yang digunakan adalah *content analysis* atau analisis isi, dengan pendekatan deskriptif-komparatif. Masing-masing mekanisme *mutual exclusion* dibandingkan berdasarkan beberapa aspek, yaitu efektivitas dalam mencegah *race condition,* konsistensi dan integritas data, kemudahan implementasi, besar sumber daya sistem (seperti penggunaan CPU dan memori), serta dukungan terhadap berbagai sistem operasi. Hasil analisis akan dirangkum dalam bentuk narasi dan tabel komparatif untuk memudahkan pembacaan dan penarikan kesimpulan. Untuk menjaga validitas data, peneliti hanya menggunakan sumber-sumber yang kredibel. Selain itu, dilakukan triangulasi data dengan membandingkan informasi dari berbagai referensi untuk memastikan konsistensi dan keakuratan temuan yang diperoleh.

Hasil dan Pembahasan

Konsep Sinkronisasi dan *Mutual Exclusion*

Sinkronisasi adalah mekanisme penting dalam sistem komputer yang digunakan untuk mengelola akses urutan eksekusi dan interaksi beberapa unit eksekusi, seperti proses atau *thread,* yang secara bersamaan (konkuren) mengakses sumber daya bersama (Silberschatz et al., 2018). Tujuan utama sinkronisasi untuk menghindari ketidakkonsistenan data akibat akses bersamaan terhadap sumber daya dan memastikan setiap tahapan berlangsung secara terkoordinasi dalam lingkungan *multiprocessing* (Apsiswanto & Muharni, 2022). Selain itu, sinkronisasi juga berperan dalam pengelolaan penyimpanan data, baik sementara maupun permanen, guna mendukung efisiensi dan kelancaran proses kerja (Ratna, 2023).

Tidak hanya menjaga konsistensi data, sinkronisasi juga digunakan untuk mencegah *race condition* dan *deadlock* (Fajar et al., 2022). *Race condition* adalah kondisi yang terjadi ketika dua atau lebih proses atau *thread* mengakses dan memanipulasi data secara bersamaan, sehingga hasil akhir aksekusi bergantung pada urutan akses yang tidak dapat diprediksi (Syakur, 2020). Situasi ini dapat mengakibatkan perilaku program yang tidak konsisten atau tidak diinginkan. Sementara itu, *deadlock* merupakan suatu kondisi dalam sistem operasi ketika dua atau lebih proses atau *thread* saling menunggu pelepasan sumber daya yang sedang digunakan oleh proses lain, sehingga tidak ada proses yang dapat melanjutkan eksekusi (Prasetiyo et al, 2024). Keadaan ini menyebabkan kebuntuan (*stalemate*) dalam sistem dan menjadi permasalahan penting dalam manajemen sumber daya pada sistem operasi, karena dapat berdampak pada penurunan kinerja bahkan kegagalan sistem secara keseluruhan.

Salah satu permasalahan utama dalam sinkronisasi adalah *critical section problem*, yaitu kebutuhan untuk memastikan bahwa hanya satu proses yang dapat mengakses bagian kode yang mengelola sumber daya bersama secara eksklusif pada satu waktu (Silberschatz et al., 2018). Menurut (Silberschatz et al., 2018), sebelum sebuah proses dapat memasuki *critical section*, proses tersebut harus terlebih dahulu melalui tahapan permintaan akses yang dikenal sebagai *entry section*. Setelah menyelesaikan eksekusi di dalam *critical section,* proses tersebut memasuki *exit section* sebagai langkah keluar dari *critical section*. Adapun kode lainnya yang tidak berkaitan langsung dengan pengelolaan sumber daya bersama disebut sebagai *remainder section*.

Untuk mencegah berbagai permasalahan tersebut, termasuk *race condition* dan *deadlock,* maka diterapkan konsep *mutual exclusion* sebagai bagian inti dari mekanisme sinkronisasi. *Mutual exclusion* adalah kondisi ketika hanya satu proses atau *thread* yang diizinkan mengakses sumber daya kritis (*critical section*) pada satu waktu (Silberschatz et al., 2018). Tujuan utamanya adalah untuk memastikan bahwa akses terhadap sumber daya bersama dilakukan secara ekslusif, sehingga tidak terjadi konflik akibat eksekusi paralel yang tidak terkendali (Raynal & Taubenfeld, 2022).

Menurut Stallings (2018), *mutual exclusion* merupakan prinsip fundamental dalam pengendalian akses ke *critical section* dan solusi terhadap permasalahan ini harus memenuhi tiga syarat utama. Pertama, *mutual exclusion,* yaitu memastikan bahwa hanya satu proses yang dapat berada dalam *critical section* pada satu waktu untuk mencegah konflik dalam akses terhadap sumber daya bersama. Kedua, *progress,* yaitu jika tidak ada proses yang sedang berada di dalam *critical section*, maka proses-proses lain yang ingin masuk tidak boleh mengalami penundaan yang tidak perlu. Ketika, *bounded waiting,* yaitu menjamin bahwa terdapat batas maksimum jumlah proses lain yang dapat mengakses *critical section* sebelum suatu proses yang sedang menunggu diberikan giliran, guna mencegah terjadinya *starvation*, yaitu kondisi ketika proses berprioritas rendah terblokir tanpa batas waktu karena terus-menerus didahului oleh proses berprioritas lebih tinggi.

Sedangkan Tanenbaum & Bos (2015) menekankan bahwa solusi sinkronisasi yang efektif terhadap masalah *critical section* harus memenuhi empat kriteria penting. Pertama, sistem harus menjamin bahwa tidak ada dua proses yang berada dalam *critical section* secara bersamaan. Kedua, mekanisme sinkronisasi tidak boleh bergantung pada asumsi

tertentu mengenai kecepatan eksekusi proses atau jumlah prosesor yang tersedia. Ketiga, proses yang tidak sedang mengakses *critical section* tidak seharusnya menghalangi proses lain yang ingin masuk. Keempat, setiap proses yang ingin mengakses *critical section* harus dijamin tidak mengalami penundaan tanpa batas waktu. Pemenuhan keempat prinsip ini bertujuan untuk mencegah konflik dalam akses data bersama serta mendukung koordinasi proses yang efisien dan adil dalam lingkungan komputasi paralel atau multiprosesor

Mekanisme *Mutual Exclusion*

Seiring perkembangan ilmu komputer, berbagai pendekatan telah dikembangkan untuk mengatasi permasalahan *critical section* dalam sistem multiproses. Pada tahap awal, solusi dilakukan menggunakan algoritma perangkat lunak seperti *Dekker's Algorithm* dan *Peterson's Algorithm*, yang membuktikan bahwa sinkronisasi dapat dicapai tanpa bantuan perangkat keras (Nigro & Cicirelli, 2025). Menurut (Nigro & Cicirelli, 2025), *Dekker's Algorithm* yang diperkenalkan pada tahun 1960 merupakan solusi *mutual exclusion* pertama yang sepenuhnya bergantung pada mekanisme perangkat lunak dua proses. Algoritma ini membuktikan bahwa pengaturan akses ke *critical section* dapat dilakukan tanpa dukungan operasi atomic dari perangkat keras. Sementara itu, menurut (Nigro & Cicirelli, 2024a), *Peterson's Algorithm* dapat memberikan solusi yang lebih ringkas dan rapi dibandingkan dengan *Dekker's Algorithm*. Algoritma ini juga dapat diperluas untuk digunakan pada lebih dari dua proses (*N-process solution*), sehingga menjadikannya lebih fleksibel dalam lingkungan multiproses yang lebih kompleks. Namun, algoritma-algoritma ini memiliki kompleksitas implementasi yang tinggi dan kurang efisien untuk diterapkan dalam sistem nyata berskala besar.

Untuk meningkatkan efisiensi dan kemudahan implementasi, para pengembang saat ini lebih banyak menggunakan mekanisme sinkronisasi tingkat tinggi yang didukung oleh sistem operasi atau bahasa pemrograman. Contoh dari mekanisme ini adalah *semaphore, mutex*, dan *monitor* yang secara internal menangani permasalahan *mutual exclusion* dan menyediakan antarmuka yang lebih sederhana dan andal bagi pengembang aplikasi, terutama dalam sistem multiproses dan sistem terdistribusi. Berikut ini menyajikan penjelasan untuk masing-masing mekanisme *mutual exclusion.*

1. *Semaphore*

Semaphore merupakan salah satu mekanisme penting dalam sistem operasi yang digunakan untuk mengatur sinkronisasi dan pengendalian akses terhadap sumber daya bersama (*shared resources*). Dalam lingkungan komputasi modern yang melibatkan banyak proses atau *thread* yang berjalan secara bersamaan (*concurrent execution*), penggunaan *semaphore* menjadi krusial untuk menghindari kondisi yang tidak diinginkan seperti *race condition*, *deadlock*, atau *starvation* (Silberschatz et al, 2018).

Dalam konteks sistem operasi, *semaphore* diperkenalkan oleh Edsger Dijkstra sebagai alat bantu sinkronisasi yang bersifat abstrak, dan secara umum dibedakan menjadi dua jenis: *semaphore* biner (mirip dengan *mutex*) dan *semaphore* menghitung (*counting semaphore*). *Semaphore* biner memiliki dua nilai, 0 dan 1, dan sering digunakan untuk mengontrol akses ke *critical section*, sedangkan *semaphore* menghitung dapat

digunakan untuk mengatur akses ke sejumlah terbatas dari sumber daya tertentu, seperti koneksi *database* atau slot *buffer* (Silberschatz et al, 2018).

Menurut Apsiswanto & Muharni (2022), mekanisme kerja semaphore dalam sistem operasi melibatkan dua operasi dasar, yaitu *wait()* atau *P()* dan *signal()* atau *V()*. Operasi *wait()* berfungsi untuk mengurangi nilai semaphore. Apabila nilai semaphore menjadi negatif, maka proses yang memanggil operasi ini akan diblokir dan dimasukkan ke dalam antrian. Sebaliknya, operasi *signal()* berfungsi untuk menambah nilai semaphore. Jika terdapat proses yang sedang menunggu di antrian, maka salah satu proses tersebut akan dibangunkan dan diberi izin untuk melanjutkan eksekusi.

Fungsi utama dari *semaphore* dalam sistem operasi adalah sebagai alat untuk mencapai mutual exclusion dan proses sinkronisasi, terutama pada model komunikasi antar-proses (*inter-process communication*, IPC) dan pada sistem multiprosesor serta *multi-core* yang menuntut eksekusi paralel secara aman dan efisien (Tanenbaum & Bos, 2015).

Dalam bahasa pemrograman Java, kelas *Semaphore* yang tersedia dalam paket *java.util.concurrent* menyediakan fasilitas untuk implementasi sinkronisasi. Salah satu konfigurasi yang umum digunakan adalah *semaphore* biner, yaitu *semaphore* dengan satu izin, yang digunakan untuk mengatur akses ke bagian kritis, memastikan hanya satu *thread* dapat masuk pada satu waktu. Dua metode utama yang digunakan adalah *acquire()* dan *release()*. Metode *acquire()* digunakan oleh *thread* untuk meminta izin sebelum mengakses sumber daya bersama dan apabila izin tidak tersedia, *thread* akan diblokir hingga izin diberikan kembali. Setelah operasi selesai, *thread* akan memanggil metode *release()* untuk melepaskan izin tersebut, sehingga dapat digunakan oleh *thread* lain Oracle (2025).

Mekanisme ini menjamin *mutual exclusion* dengan memblokir *thread* yang tidak memperoleh izin akses (Kode & Oyemade, 2024). Penggunaan *semaphore* dalam konteks ini memungkinkan kontrol yang efektif terhadap akses paralel terhadap sumber daya bersama, sekaligus memberikan dasar untuk penerapan kontrol konkurensi yang aman (Goetz et al, 2009).

2. *Mutex*

Mutex (*mutual exclusion*) merupakan salah satu mekanisme sinkronisasi yang paling mendasar dalam sistem operasi, dirancang untuk menjamin bahwa hanya satu proses atau utas (*thread*) yang dapat mengakses sumber daya bersama (*shared resource*) pada suatu waktu. Mekanisme ini sangat penting dalam pengelolaan *critical section* atau bagian kode yang mengakses sumber daya bersama untuk mencegah terjadinya *race condition*, inkonsistensi data, dan gangguan eksekusi akibat akses simultan.

Dalam sistem operasi, *mutex* berfungsi sebagai alat sinkronisasi yang diimplementasikan untuk memastikan bahwa eksekusi kode dalam *critical section* dilakukan secara eksklusif. Mekanisme ini melibatkan dua operasi dasar, yaitu *lock()* atau *acquire()* dan *unlock()* atau *release()*. Operasi *lock()* digunakan oleh proses atau *thread* untuk mencoba memperoleh kunci *mutex* sebelum memasuki bagian kritis. Setelah selesai menjalankan kode di dalam *critical section*, proses atau *thread* tersebut harus melepaskan

kunci dengan menggunakan operasi *unlock()*, agar proses atau *thread* lain dapat memperoleh akses secara bergiliran.

Mutex biasanya digunakan dalam sistem *multi-threaded* dan *multiprocessing* untuk mendukung koordinasi antar-proses maupun antar-utas. Jika tidak digunakan dengan benar, *mutex* dapat menyebabkan masalah seperti *deadlock* (kebuntuan) atau *starvation* (kelaparan sumber daya), yang merupakan isu penting dalam desain dan implementasi sistem operasi (Silberschatz et al., 2018; Tanenbaum & Bos, 2015).

Dalam sistem operasi modern yang menjalankan banyak proses dan utas secara bersamaan, performa *mutex* menjadi faktor krusial dalam menjaga efisiensi sistem secara keseluruhan. Kontensi yang tinggi, yaitu kondisi ketika banyak *thread* bersaing untuk memperoleh *mutex* yang sama, dapat menyebabkan penurunan performa sistem secara signifikan. Oleh karena itu, banyak penelitian yang fokus pada optimalisasi algoritma *mutex* untuk mengurangi latensi dan overhead dalam sistem multi-utas (Yu et al., 2023).

Selain itu, sistem operasi yang memanfaatkan memori non-volatil (*Non-Volatile Memory*, NVM) memerlukan desain *mutex* yang tahan terhadap gangguan seperti pemadaman listrik atau *system crash*. Dalam konteks ini, dikembangkan algoritma *recoverable mutex*, yaitu algoritma yang memungkinkan proses untuk melanjutkan eksekusi setelah kegagalan dengan tetap menjaga konsistensi sumber daya yang dilindungi *mutex*. Pendekatan ini penting untuk memastikan bahwa status sinkronisasi tetap terjaga meskipun sistem mengalami pemulihan dari kegagalan (Jayanti et al., 2023).

3. *Monitor*

Monitor merupakan abstraksi tingkat tinggi untuk sinkronisasi antar proses yang pertama kali diperkenalkan oleh C.A.R. Hoare dan Per Brinch Hansen pada awal 1970-an. Sebagai struktur data abstrak, *monitor* membungkus variabel-variabel bersama serta prosedur-prosedur yang mengaksesnya dalam satu unit program yang tertutup (*encapsulated*). Hanya satu proses yang diperbolehkan menjalankan prosedur dalam *monitor* pada satu waktu, sehingga prinsip *mutual exclusion* dijamin secara otomatis (Hansen, 1973).

Monitor biasanya terdiri dari tiga elemen utama, yaitu variabel lokal, prosedur, dan variabel kondisi. Variabel lokal merupakan data yang hanya dapat diakses dari dalam *monitor* dan tidak dapat dijangkau langsung oleh proses di luar *monitor*, sehingga menjaga enkapsulasi dan keamanan data. Prosedur-prosedur di dalam *monitor* berfungsi untuk mengatur akses terhadap data tersebut, memastikan bahwa hanya satu proses yang dapat menjalankan satu prosedur dalam satu waktu. Sementara itu, variabel kondisi (condition variables) digunakan untuk mengelola proses-proses yang harus menunggu hingga syarat tertentu terpenuhi sebelum dapat melanjutkan eksekusi (Silberschatz et al, 2018). Dengan struktur ini, *monitor* mengurangi kompleksitas pemrograman sinkronisasi dibandingkan pendekatan tingkat rendah seperti *semaphore*.

Monitor menyediakan variabel kondisi untuk mendukung sinkronisasi yang lebih kompleks. Variabel ini digunakan ketika proses tidak dapat melanjutkan eksekusi karena syarat tertentu belum terpenuhi. Dalam kasus ini, proses tersebut akan menunggu

dengan memanggil operasi *wait()*, yang secara otomatis melepaskan kontrol atas *monitor* dan memblok proses tersebut hingga *signal()* dipanggil oleh proses lain. Dengan pendekatan ini, pengelolaan proses menjadi lebih terstruktur dan mencegah kesalahan umum yang sering terjadi saat menggunakan *semaphore* secara langsung (Silberschatz et al., 2018; Stallings, 2018).

Monitor tidak hanya merupakan konsep teoretis, tetapi juga telah banyak diimplementasikan dalam berbagai sistem dan bahasa pemrograman. Dalam bahasa Java, *monitor* diimplementasikan secara implisit pada setiap objek melalui kata kunci *synchronized*. Setiap metode atau blok kode yang diberi anotasi *synchronized* akan menjamin bahwa hanya satu *thread* yang dapat mengaksesnya dalam satu waktu. Di sisi lain, meskipun sistem operasi Linux tidak secara eksplisit menggunakan istilah *monitor*, konsep serupa diterapkan melalui mekanisme *mutex locks*, *condition variables*, dan *wait queues* untuk mendukung sinkronisasi antara kernel dan proses pengguna. Sementara itu, bahasa pemrograman seperti C# menyediakan konstruk *lock* sebagai abstraksi dari *monitor*, yang memudahkan pengelolaan sinkronisasi dalam lingkungan pengembangan berorientasi objek.

Penerapan *monitor* yang tepat dapat meningkatkan modularitas, keamanan, dan keterbacaan kode, sehingga mempermudah pemeliharaan dan debugging sistem yang kompleks (Bovet & Cesati, 2006) (Tanenbaum & Bos, 2015).

Analisis Efektifitas Mekanisme *Mutual Exclusion*

Setelah dijelaskan karakteristik konseptual dari *semaphore, mutex,* dan *monitor* pada subbab sebelumnya, bagian ini bertujuan untuk mengevaluasi efektivitas masing-masing secara komprehensif. Analisis dilakukan dengan mempertimbangkan lima aspek utama yang penting dalam sinkronisasi sebagaimana dikemukakan oleh Bueso (2015), yaitu: (1) kemudahan implementasi, (2) perlindungan terhadap *race condition*, (3) risiko terjadinya *deadlock* dan *starvation,* dan (4) kesesuaian terhadap berbagai konteks aplikasi. Berdasarkan kelima aspek tersebut, berikut adalah uraian analisis efektifitas masing-masing mekanisme:

1. Kemudahan Implementasi

Dalam sistem operasi, kemudahan implementasi menjadi pertimbangan penting, mengingat kompleksitas arsitektur sistem dan kebutuhan efisiensi waktu pengembangan. Mekanisme yang sederhana dan minim risiko kesalahan akan lebih mudah diintegrasikan dalam sistem. Tabel berikut ini membandingkan *semaphore, mutex,* dan *monitor* dari segi kemudaham implementasinya

Table 1. Analisis Kemudahan Implementasi Mekanisme *Mutual Exclusion*

Metode	Penjelasan
Semaphore	*Semaphore* relatif fleksibel karena dapat digunakan untuk mengelola akses ke banyak sumber daya secara bersamaan (*counting semaphore*) atau hanya satu sumber daya (*binary semaphore*). Namun, implementasinya lebih kompleks karena tidak memiliki batasan eksplisit terhadap siapa yang bisa melepaskan (*release*) *semaphore*. Hal ini dapat menyebabkan kesalahan logika dalam program jika tidak dikelola dengan baik (Silberschatz et al., 2018).
Mutex	*Mutex* secara khusus dirancang untuk *mutual exclusion* dan hanya dapat dibuka oleh proses atau *thread* yang telah menguncinya. Hal ini membuat *mutex* lebih mudah diimplementasikan dan lebih aman karena mencegah proses atau *thread* lain melepas kunci secara tidak sah. *Mutex* banyak digunakan dalam sistem operasi modern karena desainnya yang intuitif (Stallings, 2018).
Monitor	*Monitor* memberikan abstraksi tingkat tinggi yang menyederhanakan pengelolaan sinkronisasi melalui penggabungan otomatis antara *mutual exclusion* dan manajemen kondisi. Keunggulan utamanya adalah kemudahan penggunaan, yaitu programmer tidak perlu mengatur penguncian dan pembebasan secara manual, sehingga risiko kesalahan lebih kecil. Namun, *monitor* tidak tersedia secara eksplisit di semua bahasa atau kernel OS, sehingga dalam sistem operasi yang tidak mendukungnya secara *native*, implementasinya menjadi lebih kompleks dan harus disimulasikan menggunakan mekanisme lain seperti *mutex* dan variabel kondisi (Tanenbaum & Bos, 2015; Silberschatz et al., 2018)

2. Perlindungan Terhadap *Race Condition*

Race condition merupakan salah satu permasalahan penting dalam sistem operasi yang dapat menyebabkan ketidakstabilan sistem. Oleh karena itu, mekanisme sinkronisasi harus mampu mencegah kondisi ini secara konsisten. Tabel berikut menyajikan evaluasi terhadap kemampuan masing-masing mekanisme dalam mencegah *race condition* pada lingkungan sistem operasi.

Table 2. Analisis Perlindungan Terhadap Race Condition Mekanisme *Mutual Exclusion*

Metode	Penjelasan
Semaphore	*Semaphore* dapat efektif dalam mencegah *race condition* melalui mekanisme *counting* yang memungkinkan pengaturan akses ke satu atau lebih *resource* secara bersamaan. Namun, efektivitas ini sangat bergantung pada implementasi yang tepat dari operasi P (*wait*) dan V (*signal*). Tanpa kontrol yang ketat, kesalahan seperti lupa melakukan *release* atau melepaskan terlalu banyak dapat menyebabkan pelanggaran prinsip *mutual exclusion* dan justru memperburuk *race condition* (Tanenbaum & Bos, 2015).
Mutex	*Mutex* memberikan perlindungan yang kuat karena hanya satu proses atau *thread* yang dapat mengunci dan membuka akses ke *critical section*. Mekanisme kepemilikan membuatnya lebih aman dibanding *semaphore*, dan sangat efektif dalam mencegah *race condition* pada sistem operasi modern (Silberschatz et al., 2018).
Monitor	*Monitor* sangat efektif dalam mencegah *race condition* karena seluruh akses ke data bersama dilakukan melalui prosedur sinkron yang dibungkus dalam struktur *monitor*. Dengan kombinasi otomatis antara *mutual exclusion* dan manajemen kondisi, *race condition* dapat dihindari tanpa kontrol manual yang kompleks (Tanenbaum & Bos, 2015).

3. Risiko Terjadinya *Deadlock* dan *Starvation*

Dalam sistem operasi, *deadlock* merupakan kondisi kritis yang dapat menyebabkan proses-proses penting terhenti secara permanen karena saling menunggu sumber daya yang tidak kunjung tersedia. Kondisi ini tidak hanya menurunkan performa sistem, tetapi juga dapat menyebabkan kegagalan total pada layanan yang berjalan. Di sisi lain, *starvation* terjadi ketika suatu proses terus-menerus gagal memperoleh akses ke sumber daya karena selalu didahului oleh proses lain, sehingga menyebabkan kelaparan sumber daya (*resource starvation*). Oleh karena itu, penting untuk mengevaluasi tingkat risiko *deadlock* dan *starvation* yang mungkin ditimbulkan oleh setiap mekanisme sinkronisasi. Tabel berikut menyajikan perbandingan risiko kedua kondisi tersebut pada *semaphore*, *mutex*, dan *monitor*.

Tabel 3. Analisis Risiko Terjadinya *Deadlock* dan *Starvation* Mekanisme *Mutual Exclusion*

Metode	Penjelasan
Semaphore	*Semaphore* memiliki risiko *deadlock* dan *starvation* yang lebih tinggi karena tidak ada mekanisme bawaan untuk mengatur antrian atau keadilan akses. Jika dua proses atau *thread* saling menunggu *release* yang tidak pernah datang, *deadlock* dapat terjadi. Selain itu, proses atau *thread* dengan prioritas lebih rendah bisa terus-menerus tertunda, menyebabkan *starvation* jika tidak ada manajemen akses yang adil (Shakor, 2021).
Mutex	*Mutex* lebih aman dari segi *starvation*, terutama jika sistem mendukung fitur seperti *priority inheritance* atau *fair scheduling*. Namun, risiko *deadlock* tetap ada, terutama jika terjadi *nested locking* tanpa urutan penguncian yang konsisten antar proses atau *thread (Stallings, 2018)*.
Monitor	*Monitor* cenderung lebih aman terhadap *deadlock* karena kontrol akses terpusat melalui prosedur sinkron. Akan tetapi, *starvation* masih bisa terjadi jika sinyal ke kondisi (*condition variable*) tidak diberikan secara adil atau tidak tepat waktu, membuat proses atau *thread* terus menunggu tanpa dibangunkan (Silberschatz et al., 2018).

4. Kesesuaian Terhadap Beberapa Konteks Aplikasi

Tidak semua mekanisme sinkronisasi cocok untuk semua skenario. Dalam sistem operasi, kesesuaian ini sangat tergantung pada jenis aplikasi, tingkat paralelisme, dan kebutuhan performa. alam konteks sistem operasi, pemilihan antara *semaphore*, *mutex*, dan *monitor* harus mempertimbangkan lingkungan implementasi, seperti sistem *real-time*, *multiprosesor*, maupun sistem terdistribusi. Tabel berikut menyajikan perbandingan kesesuaian ketiga mekanisme sinkronisasi tersebut dalam berbagai skenario sistem operasi.

Tabel 4. Analisis Kesesuaian Terhadap Beberapa Konteks Aplikasi Mekanisme *Mutual Exclusion*

Metode	Penjelasan
Semaphore	*Semaphore* cocok digunakan dalam sistem operasi untuk mengelola akses ke beberapa unit sumber daya secara bersamaan, seperti pada manajemen buffer di masalah produsen-konsumen. Namun, dalam skala besar, penggunaan *semaphore* rawan kesalahan karena tidak adanya kontrol kepemilikan yang ketat, sehingga memerlukan pengelolaan yang sangat hati-hati (Nigro & Cicirelli, 2024).

Metode	Penjelasan
Mutex	*Mutex* sangat sesuai untuk kebutuhan akses eksklusif tunggal terhadap *resource*, seperti sinkronisasi akses *file system*, memori bersama, atau perangkat I/O dalam sistem operasi. Karena mendukung kepemilikan kunci, *mutex* juga banyak dipakai dalam sistem *embedded* dan *real-time* yang membutuhkan penguncian deterministik (Stallings, 2018).
Monitor	*Monitor* ideal untuk sistem operasi atau pustaka yang dibangun dengan paradigma berorientasi objek, khususnya dalam lingkungan seperti Java *Virtual Machine*. Namun, dalam OS yang ditulis dengan bahasa seperti C, penerapannya kurang fleksibel karena tidak tersedia secara eksplisit dan harus disimulasikan dengan mekanisme lain (Tanenbaum & Bos, 2015).

Penelitian ini bertujuan untuk mengevaluasi efektivitas tiga mekanisme utama dalam sinkronisasi proses, yaitu *semaphore*, *mutex*, dan *monitor*, dengan mempertimbangkan berbagai aspek fungsional dan kontekstual dalam sistem operasi modern. Hasil analisis yang telah dilakukan memperkuat serta memperluas wawasan yang telah dibahas dalam literatur sebelumnya, serta menghadirkan temuan baru yang signifikan, terutama dalam hal kesesuaian penggunaan terhadap karakteristik sistem dan risiko kesalahan implementasi.

Salah satu temuan utama dalam studi ini adalah keunggulan *monitor* dalam menjaga stabilitas sistem dan meminimalisasi kesalahan implementasi. Hal ini sejalan dengan pandangan Turck (2022), yang menunjukkan bahwa monitor secara konsisten unggul dalam lingkungan sistem *multithreaded* yang kompleks, khususnya dalam skenario *read-modify-write* yang sering ditemukan di basis data. Keunggulan ini diperkuat oleh struktur internal *monitor* yang mengintegrasikan *mutual exclusion* dan pengelolaan kondisi secara otomatis. Dengan demikian, *monitor* tidak hanya menjamin keamanan akses terhadap *critical section*, tetapi juga mengurangi kompleksitas pengembangan dan *debugging*.

Di sisi lain, *mutex* terbukti paling efisien dalam konteks sistem deterministik dan latensi rendah. Temuan ini mendukung hasil eksperimen oleh Ehis (2024), yang menemukan bahwa *mutex* mampu mencapai *throughput* tinggi saat latensi antar-*thread* minimal. Hal ini menjadikan *mutex* sebagai pilihan ideal untuk sistem *real-time* atau *embedded*, sebagaimana juga disampaikan oleh Sakthivel & Sreeja (2024), yang mengevaluasi kinerja *mutex* pada sistem FreeRTOS berbasis Arduino. Mekanisme kepemilikan yang ketat pada *mutex* menjadi faktor utama dalam menjaga determinisme dan menghindari pelepasan kunci oleh *thread* yang tidak berwenang.

Sebaliknya, *semaphore* menunjukkan fleksibilitas tinggi terutama dalam mengelola beberapa sumber daya secara bersamaan melalui *counting semaphore*. Namun, hasil analisis ini mengonfirmasi bahwa fleksibilitas tersebut juga menjadi titik rawan, karena *semaphore* lebih rentan terhadap kesalahan implementasi seperti unbalanced locki*ng* dan *release* yang tidak sesuai. Ini memperkuat kritik dari Shakor (2021) dan Suveetha et al. (2020), yang menunjukkan bahwa penggunaan *semaphore* sering kali mengakibatkan *race condition* dan *starvation* jika tidak dirancang dengan hati-hati.

Perbandingan dengan literatur terdahulu juga menunjukkan kontribusi penelitian ini dalam menyoroti aspek resiliensi terhadap kesalahan implementasi, yang jarang dibahas secara eksplisit dalam penelitian sebelumnya. Penekanan pada potensi *deadlock* dan *starvation* serta pencegahannya dalam tiap mekanisme menjadi landasan penting dalam pengambilan keputusan teknis saat membangun sistem paralel. Dalam hal ini, monitor kembali menunjukkan performa terbaik karena manajemen aksesnya yang terenkapsulasi dan terkontrol penuh oleh sistem, meskipun pada sistem yang tidak mendukungnya secara *native* (misalnya C atau kernel Linux), implementasinya menjadi kurang fleksibel dan memerlukan simulasi tambahan (Tanenbaum & Bos, 2015)

Lebih jauh, kajian ini memetakan hubungan antara konteks sistem dan efektivitas mekanisme, seperti yang dijabarkan dalam *state of the art* penelitian. Misalnya, pada sistem multiprosesor dengan kontensi tinggi, monitor lebih mampu menjaga stabilitas dan performa. Sebaliknya, dalam sistem *embedded* atau *real-time*, *mutex* menjadi pilihan utama karena *overhead*-nya yang rendah dan kemampuannya dalam mengatur prioritas *task*. Dalam sistem dengan komunikasi antar-*thread* yang tidak stabil, *semaphore* masih dapat diandalkan karena skalabilitasnya, meskipun harus diimbangi dengan disiplin pemrograman yang tinggi.

Dengan demikian, temuan baru dalam penelitian ini tidak hanya mempertegas efektivitas masing-masing mekanisme dalam konteks yang berbeda, tetapi juga menawarkan kerangka evaluatif standar yang dapat digunakan sebagai acuan dalam pemilihan teknik sinkronisasi yang optimal. Tidak seperti studi sebelumnya yang lebih bersifat fragmentatif atau studi kasus terbatas, analisis ini memberikan kontribusi berupa sintesis komparatif menyeluruh terhadap efektivitas, resiliensi, dan risiko teknis dari ketiga mekanisme sinkronisasi tersebut. Rekomendasi utama yang dapat disimpulkan adalah bahwa pemilihan mekanisme sinkronisasi harus bersifat kontekstual, mempertimbangkan kebutuhan performa sistem, kompleksitas pengembangan, dan potensi kesalahan implementasi.

Simpulan

Penelitian ini bertujuan untuk menganalisis dan membandingkan efektivitas tiga mekanisme sinkronisasi dalam sistem operasi, yaitu *semaphore*, *mutex*, dan *monitor*. Berdasarkan hasil analisis, dapat disimpulkan bahwa setiap mekanisme memiliki keunggulan dan keterbatasan yang bersifat kontekstual. *Monitor* menunjukkan stabilitas dan struktur yang unggul dalam menangani interaksi kompleks antar *thread*, *mutex* menawarkan efisiensi tinggi dalam sistem dengan latensi rendah, sedangkan *semaphore* menonjol dari sisi fleksibilitas meskipun lebih rawan kesalahan implementasi.

Dengan demikian, temuan ini menguatkan hipotesis bahwa *monitor* paling efektif dalam kondisi kontensi tinggi, *mutex* optimal dalam sistem deterministik dan *real time*, sementara *semaphore* cocok untuk pengelolaan akses serentak yang lebih bebas namun memerlukan kehati-hatian tinggi dalam penggunaannya. Pemilihan mekanisme sinkronisasi yang tepat sangat bergantung pada karakteristik sistem operasi yang digunakan, termasuk beban kerja, tingkat paralelisme, dan kebutuhan terhadap keandalan serta efisiensi.

Penelitian selanjutnya disarankan untuk memperluas kajian pada implementasi nyata ketiga mekanisme tersebut di berbagai distribusi sistem operasi modern. Fokus penelitian dapat diarahkan pada analisis performa sinkronisasi dalam lingkungan kernel, pengaruh terhadap efisiensi manajemen thread dan proses, serta integrasi mekanisme sinkronisasi dengan fitur keamanan sistem operasi. Selain itu, penting untuk mengeksplorasi pendekatan adaptif dalam pemilihan mekanisme sinkronisasi berdasarkan konteks runtime, guna meningkatkan efisiensi dan keandalan sistem secara dinamis.

Daftar Pustaka

Apsiswanto, U., & Muharni, S. (2022). *Sistem Operasi*. Literasi Nusantara Abadi.

Bovet, D. P., & Cesati, M. (2006). *Understanding the Linux Kernel (3rd ed.)*. O'Reilly Media.

Bueso, D. (2015). Scalability Techniques for Practical Synchronization Primitives. *Communications of the ACM*, *58*(1), 66–74. https://doi.org/10.1145/2687882

Chan, D. Y. C., & Woelfel, P. (2021). Tight Lower Bound for the RMR Complexity of Recoverable Mutual Exclusion. *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 533–543. https://doi.org/10.1145/3465084.3467938

Chaturvedi, A., Daymude, J. J., & Richa, A. W. (2025). On the Runtime of Local Mutual Exclusion for Anonymous Dynamic Networks. *Computing and Augmented Intelligence*, 1–16. https://doi.org/10.4230/LIPIcs.SAND.2025.15

Dhoked, S., & Mittal, N. (2022). Adaptive and Fair Transformation for Recoverable Mutual Exclusion. *IEEE International Conference on Program Comprehension*, *2022-March*, 36–47.

Ehis, A. T. (2024). Analysis of Multi-Threading and Cache Memory Latency Masking on Processor Performance Using Thread Synchronization Technique. *Brazilian Journal of Science*, *3*(1), 159–174. https://doi.org/10.14295/bjs.v3i1.458

Fajar, A., Dhika, D. S., & Febriansyah, R. P. (2022). Strategi penanganan Deadlock yang Efektif dalam Sistem Operasi Berbasis Windows: Pencegahan Deadlock, Identifikasi Faktor Penyebab, dan Dampak dari Deadlock. *SINTESIA: Jurnal Sistem Dan Teknologi Informasi Indonesia*, *2*(1), 6–11.

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2009). *Java concurrency in practice*. Pearson Education.

Hansen, P. B. (1973). *Operating System Principles*. Prentice-Hall.

Jayanti, P., Jayanti, S., & Joshi, A. (2023). *Constant RMR Recoverable Mutex under System-wide Crashes*. https://doi.org/10.48550/arXiv.2302.00748

Kode, O., & Oyemade, T. (2024). Analysis of Synchronization Mechanism in Operating Systems. *IEEE International Conference on Program Comprehension*.

Nigro, L., & Cicirelli, F. (2024a). Correctness Verification of Mutual Exclusion Algorithms by Model Checking. *Modelling*, *5*(3), 694–719. https://doi.org/10.3390/modelling5030037

Nigro, L., & Cicirelli, F. (2024b). Property Assessment of Peterson's Mutual Exclusion Algorithms. *Applied Computing and Intelligence*, *4*(1), 66–92. https://doi.org/10.3934/aci.2024005

Nigro, L., & Cicirelli, F. (2025). Proving Properties of Dekker's Algorithm for Mutual Exclusion of N Processes. *Algorithms*, *18*(4), 1–19. https://doi.org/10.3390/a18040226

Oracle. (n.d.). *Semaphore*.

Prasetiyo, S. M., Gustiawan, R., Farhat, & Albani, F. R. (2024). Penanganan Deadlock Yang Optimal Dalam Sistem Operasi Windows: Pencegahan, Identifikasi Penyebab, Dan Konsekuensi Deadlock. *BIIKMA : Buletin Ilmiah Ilmu Komputer Dan Multimedia*, *2*(1), 60–64. https://jurnalmahasiswa.com/index.php/biikma/article/view/1030/691

Ratna, S. (2023). *Sistem Operasi*. Yayasan Kita Menulis.

Raynal, M., & Taubenfeld, G. (2022). A visit to mutual exclusion in seven dates. *Theoretical Computer Science*, *919*, 47–65. https://doi.org/10.1016/j.tcs.2022.03.030

Sakthivel, & Sreeja. (2024). Synchronization Techniques in Real-Time Operating Systems: Implementation and Evaluation on Arduino with FreeRTOS. *IJARCCE: International Journal of Advanced Research in Computer and Communication Engineering*, *13*(2), 279–284. https://doi.org/10.17148/IJARCCE.2024.13246

Shakor, M. Y. (2021). Scheduling and Synchronization Algorithms in Operating System: A Survey. *Journal of Studies in Science and Engineering*, *1*(2), 1–16. https://doi.org/10.53898/josse2021121

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating system concepts (10th edition)*. John Wiley & Sons.

Stallings, W. (2018). *Operating Systems: Internals and Design Principles (9th Edition)*. Pearson Education.

Suveetha, Sree Dharshni V, Akhyara, Sujithra, & Chitra. (2020). A Theory of Synchronisation using Semophores. *International Journal of Advances in Engineering and Management (IJAEM*, *2*(9), 256–263. https://doi.org/10.35629/5252-0209256263

Syakur, M. A. (2020). *Buku Ajar Teknik Informatika: Sistem Operasi*. Media Nusa Creative.

Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Syatems (4th Edition)*. Pearson.

Turck, F. De. (2022). Methodology for Simulation-based Comparison of Algorithms for Distributed Mutual Exclusion. *Ghent*, 1–5. https://doi.org/10.48550/arXiv.2211.01747

Wang, Y., Gao, F., Wang, L., Yu, T., Zhao, J., & Li, X. (2022). Automatic Detection, Validation, and Repair of Race Conditions in Interrupt-Driven Embedded Software. *IEEE Trans. Softw. Eng.*, *48*(1), 346–363. https://doi.org/10.1109/TSE.2020.2989171

Yu, Z., Gu, J., Wu, Z., Liu, N., & Guo, J. (2023). HTLL: Latency-Aware Scalable Blocking Mutex. *IEEE Transactions on Parallel & Distributed Systems*, *36*(03), 471–486. https://doi.org/10.1109/TPDS.2025.3526859