Mutual Exclusion, Semaphore, dan Monitor Untuk Sinkronisasi Proses

MATA KULIAH : SISTEM OPERASI

21 MEI 2025

JURUSAN TEKNIK INFORMATIKA

STMIK TAZKIA BOGOR

2025

DAFTAR ISI

I. Pendahuluan Konkurensi dalam Sistem Operasi

Sistem operasi modern dirancang untuk mengelola berbagai tugas dan proses secara efisien, seringkali secara bersamaan. Kemampuan ini, yang dikenal sebagai konkurensi, adalah inti dari kinerja dan responsivitas sistem saat ini. Konkurensi memungkinkan pemanfaatan sumber daya sistem secara optimal, namun juga memperkenalkan tantangan kompleks yang memerlukan mekanisme sinkronisasi yang canggih.

A. Konsep Dasar Konkurensi: Interleaving dan Overlapping

Konkurensi dapat diwujudkan melalui dua prinsip dasar: *interleaving* dan *overlapping*.[1] *Interleaving* terjadi pada sistem uni-prosesor, di mana sistem operasi (OS) beralih antar proses dengan cepat, menciptakan ilusi eksekusi paralel. Penjadwal OS mengelola eksekusi proses berdasarkan kebijakan yang telah ditentukan, memungkinkan eksekusi sekuensial yang meniru konkurensi.[1] Sebaliknya, *overlapping* terjadi pada sistem multi-prosesor atau multi-core, di mana proses benar-benar berjalan secara simultan pada unit pemrosesan yang berbeda. Pendekatan ini memanfaatkan kapabilitas perangkat keras untuk mencapai paralelisme sejati, yang secara signifikan meningkatkan efisiensi komputasi dan *throughput* sistem.[1]

Konkurensi adalah fundamental dalam pemrograman modern dan ditemukan di mana-mana, mulai dari jaringan komputer, aplikasi yang berjalan pada sistem multi-prosesor, hingga antarmuka pengguna grafis (GUI).[2] Integrasi konkurensi, baik melalui *interleaving* maupun *overlapping*, memang meningkatkan efisiensi dan responsivitas sistem dengan memungkinkan banyak tugas berjalan "bersamaan." Namun, sifat "bersamaan" ini, terutama ketika proses berbagi sumber daya yang sama, secara inheren menimbulkan masalah. Apabila dua proses mencoba memodifikasi data yang sama secara bersamaan, urutan eksekusi instruksi tingkat rendah menjadi tidak dapat diprediksi, yang pada akhirnya mengarah pada hasil yang tidak konsisten. Situasi ini secara langsung memicu kebutuhan akan mekanisme sinkronisasi seperti *mutual exclusion*, *semaphores*, dan *monitors* untuk mengontrol akses dan memastikan integritas data. Dengan demikian, konkurensi bukan hanya tentang eksekusi paralel, tetapi juga tentang manajemen yang cermat terhadap interaksi antar proses untuk menjaga keandalan sistem.

B. Tantangan Utama dalam Konkurensi: Race Condition

Salah satu tantangan paling signifikan dalam konkurensi adalah *race condition*. Ini adalah situasi di mana dua atau lebih *thread* secara bersamaan membaca dan menulis variabel bersama, dengan hasil akhir yang bergantung pada urutan eksekusi

mereka.[1] *Race condition* bersifat intermiten dan sangat bergantung pada waktu, menjadikannya sangat sulit untuk di-debug.[1] Implikasi dari *race condition* bisa sangat serius, mulai dari *system crashes* dan *incorrect outputs* hingga *data corruption* dan bahkan *security breaches*, seperti *privilege escalation* atau *Denial of Service (DoS)*.[4] Penyebab utama *race condition* adalah kurangnya sinkronisasi yang memastikan urutan operasi yang terdefinisi dengan baik. Tanpa sinkronisasi yang memadai, operasi menjadi non-atomik, yang berarti mereka dapat diinterupsi atau disisipkan dengan operasi lain, menyebabkan keadaan yang tidak konsisten.[4]

Race condition bukan sekadar "bug" biasa; ia adalah manifestasi langsung dari kurangnya kontrol atas urutan eksekusi dalam lingkungan konkurensi. Ketika sistem memungkinkan banyak *thread* mengakses dan memanipulasi data bersama tanpa koordinasi yang tepat, hasil akhirnya menjadi tidak deterministik. Ini berarti bahwa tanpa mekanisme sinkronisasi yang tepat, integritas data tidak dapat dijamin. Oleh karena itu, *mutual exclusion*, *semaphores*, dan *monitors* adalah solusi yang dirancang secara fundamental untuk mengatasi *race condition* dengan memaksakan urutan atau eksklusivitas akses ke *critical section*, sehingga memastikan konsistensi data dan perilaku sistem yang dapat diprediksi.

C. Pentingnya Sinkronisasi Proses

Mengingat tantangan yang ditimbulkan oleh *race condition*, sinkronisasi proses menjadi sangat penting. Sinkronisasi memastikan bahwa proses atau *thread* yang konkuren mengakses sumber daya bersama dengan aman, secara efektif mencegah *race conditions* dan *data corruption*.[7] Tujuan utamanya adalah untuk mengontrol akses banyak proses ke sumber daya bersama, di mana setiap proses membutuhkan kontrol eksklusif terhadap sumber daya tersebut saat melakukan pekerjaannya.[3]

Di era sistem multi-core dan terdistribusi, konkurensi adalah norma, bukan pengecualian. Tanpa sinkronisasi yang efektif, sistem akan rentan terhadap *race conditions*, yang mengarah pada inkonsistensi data, *crash*, dan kerentanan keamanan. Oleh karena itu, sinkronisasi bukan hanya fitur tambahan, melainkan pilar fundamental yang memungkinkan sistem operasi modern beroperasi dengan andal, efisien, dan aman. Mekanisme seperti *mutual exclusion*, *semaphores*, dan *monitors* adalah fondasi teknis yang memungkinkan integritas ini, menjembatani kesenjangan antara potensi kinerja konkurensi dan kebutuhan akan hasil yang konsisten dan benar.

II. Mutual Exclusion (Saling Pengecualian)

A. Definisi dan Tujuan

Mutual exclusion adalah properti kontrol konkurensi yang esensial, yang mensyaratkan bahwa dua atau lebih proses atau *thread* yang berbagi sumber daya tidak dapat memasuki *critical section* secara bersamaan.[3] *Critical section* sendiri mengacu pada bagian program di mana *thread* mengakses atau memodifikasi sumber daya bersama, seperti data, memori, atau perangkat keras seperti printer.[3] Sumber daya yang hanya boleh diakses oleh satu proses pada satu waktu disebut *critical resource*, yang dicirikan oleh keunikan dan eksklusivitas.[8] Tujuan utama dari *mutual exclusion* adalah untuk memastikan bahwa hanya satu proses yang dapat berada di *critical section* pada satu waktu, sehingga mencegah *race conditions* dan menjaga konsistensi data.[3]

Untuk memastikan solusi *mutual exclusion* berfungsi dengan benar dan efisien, beberapa properti esensial harus dipenuhi:

- **Mutual Exclusion:** Hanya satu proses yang diizinkan berada di *critical section* pada satu waktu.[3]
- **Progress:** Jika tidak ada proses di *critical section* dan beberapa proses ingin masuk, hanya proses yang tidak sedang mengeksekusi bagian *remainder section* yang boleh berpartisipasi dalam keputusan proses mana yang akan masuk selanjutnya. Selain itu, seleksi ini tidak boleh ditunda tanpa batas waktu.[13]
- **Bounded Waiting:** Harus ada batasan jumlah kali proses lain dapat masuk ke *critical section* setelah suatu proses menyatakan keinginannya untuk masuk. Properti ini mencegah *starvation*, di mana suatu proses mungkin tertunda tanpa batas waktu.[3]
- **Deadlock-Freedom / Lockout-Freedom:** Setiap proses yang ingin masuk ke *critical section* pada akhirnya akan dapat melakukannya. Ini berbeda dari *deadlock avoidance* yang hanya menjamin bahwa *beberapa* proses yang menunggu akan mendapatkan akses.[3]

Konsep *mutual exclusion* adalah respons langsung terhadap masalah *race condition*. Dengan memastikan hanya satu proses yang dapat mengakses *critical section* pada satu waktu, ia secara fundamental menghilangkan ketidakpastian urutan eksekusi yang menyebabkan inkonsistensi data. Properti seperti *progress* dan *bounded waiting* kemudian melengkapi *mutual exclusion* dengan memastikan bahwa solusi tersebut tidak menyebabkan masalah *liveness* (misalnya, *starvation* atau *deadlock*), yang sama pentingnya untuk sistem yang responsif dan andal. Ini menunjukkan bahwa *mutual exclusion* tidak hanya tentang "mengunci" akses, tetapi juga tentang "mengelola"

akses secara adil dan efisien.

B. Mekanisme Implementasi Mutual Exclusion

Implementasi *mutual exclusion* dapat dilakukan melalui berbagai mekanisme, baik berbasis perangkat keras maupun perangkat lunak.

1. Solusi Berbasis Perangkat Keras (Hardware-based Solutions)

Solusi berbasis perangkat keras memanfaatkan instruksi khusus CPU untuk mencapai *mutual exclusion* secara atomik.

a. Disabling Interrupts (Sistem Uni-prosesor)

Pada sistem uni-prosesor, solusi paling sederhana untuk mencapai *mutual exclusion* adalah dengan menonaktifkan interupsi selama *critical section* proses.[3] Pendekatan ini mencegah *interrupt service routines* (ISR) berjalan, secara efektif menghentikan proses dari *preemption*.[3] Kelebihan utamanya adalah jaminan *mutual exclusion* yang efektif.[15] Namun, metode ini memiliki beberapa kekurangan signifikan. Pertama, tidak efektif untuk sistem multi-prosesor.[15] Kedua, jika *critical section* terlalu panjang, *system clock* dapat melenceng karena *timer interrupt* tidak dilayani, membuat pelacakan waktu menjadi tidak mungkin.[3] Terakhir, pendekatan ini dapat menurunkan efisiensi eksekusi secara signifikan.[15]

Menonaktifkan interupsi adalah solusi yang sangat sederhana dan efektif untuk *mutual exclusion* pada sistem uni-prosesor karena secara langsung mencegah *preemption* dan *interleaving* yang tidak diinginkan. Namun, kesederhanaan ini datang dengan biaya yang signifikan: hilangnya presisi waktu dan ketidakmampuan untuk berskala ke sistem multi-prosesor. Ini menyoroti *trade-off* mendasar dalam desain sistem operasi: solusi yang paling langsung mungkin tidak selalu yang paling praktis atau efisien di lingkungan yang lebih kompleks. Kondisi ini mendorong pengembangan instruksi perangkat keras atomik yang lebih canggih.

b. Instruksi Test-and-Set

Instruksi Test-and-Set adalah solusi perangkat keras yang lebih canggih yang efektif untuk sistem uni-prosesor maupun multi-prosesor.[3] Prinsipnya melibatkan penggunaan variabel *lock* bersama yang dapat bernilai 0 (bebas) atau 1 (digunakan).[11] Operasi test_and_set() secara atomik memeriksa nilai *lock*, menyimpannya ke variabel sementara, lalu mengatur *lock* menjadi 1, dan mengembalikan nilai lama.[11] Karena operasi ini atomik, tidak ada proses lain yang dapat mengganggu selama eksekusinya, memastikan hanya satu proses yang berhasil memperoleh *lock* pada satu waktu.[3] Kelebihan utama dari Test-and-Set adalah jaminan *mutual exclusion* dan

kesederhanaan implementasinya.[11]

Namun, instruksi ini memiliki kekurangan signifikan, terutama terkait dengan *busy waiting* dan *starvation*. Proses yang gagal memperoleh *lock* akan terus-menerus memeriksa *lock* dalam sebuah *loop* (busy-waiting), yang membuang siklus CPU secara tidak efisien.[3] Selain itu, Test-and-Set tidak menjamin *bounded waiting*, yang berarti beberapa proses mungkin tertunda tanpa batas waktu, menyebabkan *starvation*.[11]

Instruksi Test-and-Set adalah langkah maju dari menonaktifkan interupsi karena ia menyediakan atomicity yang diperlukan untuk *mutual exclusion* bahkan di sistem multi-prosesor (melalui *memory bus locking*). Namun, ketergantungannya pada *busy-waiting* mengungkapkan masalah efisiensi yang signifikan. Proses yang menunggu membuang siklus CPU alih-alih menyerahkan kontrol, yang sangat tidak efisien dalam kondisi *contention* tinggi. Ini menunjukkan bahwa atomicity saja tidak cukup; mekanisme harus juga efisien dalam mengelola proses yang menunggu, mendorong pengembangan solusi yang mengurangi *busy-waiting* melalui *context switch* dan antrian.

c. Instruksi Compare-and-Swap (CAS)

Instruksi Compare-and-Swap (CAS) adalah instruksi atomik yang digunakan dalam *multithreading* untuk mencapai sinkronisasi.[19] Prinsip kerjanya adalah membandingkan isi lokasi memori dengan nilai yang diberikan (expected_value). Hanya jika nilai-nilai ini sama, isi lokasi memori tersebut dimodifikasi menjadi nilai baru (new_value). Seluruh proses perbandingan dan potensi modifikasi ini terjadi sebagai operasi tunggal yang tidak terpisahkan dan atomik.[3] Atomicity CAS sangat penting karena menjamin bahwa nilai baru yang ditulis didasarkan pada informasi yang *up-to-date*. Jika *thread* lain telah memperbarui nilai di lokasi memori di antara waktu nilai expected_value dibaca dan operasi CAS dicoba, operasi tulis akan gagal, yang kemudian memicu percobaan ulang.[20]

CAS adalah primitif fundamental untuk mengimplementasikan berbagai mekanisme sinkronisasi, termasuk semaphores dan *mutexes*, serta algoritma bebas *lock* dan bebas *wait* yang lebih kompleks.[3] Kelebihannya adalah fleksibilitasnya yang lebih tinggi dibandingkan Test-and-Set, memungkinkan pengembangan algoritma bebas *lock* yang lebih canggih.[20] Namun, CAS memiliki kekurangan, salah satunya adalah masalah ABA. Ini terjadi jika nilai berubah dari A ke B lalu kembali ke A sebelum operasi CAS dieksekusi; dalam kasus ini, CAS akan berhasil meskipun ada perubahan yang tidak diinginkan.[20] Selain itu, implementasi sederhana CAS masih dapat

melibatkan *busy-waiting*.[3]

CAS adalah primitif perangkat keras yang jauh lebih kuat daripada Test-and-Set karena kemampuannya untuk memverifikasi kondisi sebelum melakukan pembaruan secara atomik. Ini adalah kunci untuk membangun algoritma bebas *lock* dan bebas *wait*, yang merupakan paradigma *concurrency control* yang lebih canggih. Kemampuan ini memungkinkan *thread* untuk mencoba operasi, dan jika gagal karena konflik, mencoba lagi tanpa harus memblokir *thread* lain. Ini adalah pergeseran dari pendekatan "kunci dan tunggu" ke "coba dan jika gagal, ulangi," yang dapat meningkatkan paralelisme dan ketahanan sistem terhadap kegagalan *thread*. Namun, masalah seperti ABA menunjukkan bahwa bahkan primitif yang kuat pun memerlukan desain algoritma yang cermat untuk memastikan kebenaran.

Tabel 1 merangkum perbandingan mekanisme *mutual exclusion* berbasis perangkat keras:

Tabel 1: Perbandingan Mekanisme Mutual Exclusion Berbasis Perangkat Keras

Mekanisme	Prinsip Kerja Utama	Kelebihan Kunci	Kekurangan Kunci	Kondisi Penerapan	Isu Utama
Disabling Interrupts	Menonaktifkan interupsi CPU selama *critical section* untuk mencegah *preemption*.	Menjamin *mutual exclusion* secara efektif.	Tidak efektif untuk multi-prosesor; *system clock* dapat melenceng; menurunkan efisiensi.	Uni-prosesor	Kehilangan presisi waktu, tidak skalabel.
Test-and-Set	Menggunakan variabel *lock* bersama (0/1). Atomik memeriksa dan mengatur *lock* menjadi 1.	Menjamin *mutual exclusion*; sederhana.	*Busy waiting*; tidak ada *bounded waiting* (potensi *starvation*).	Uni-prosesor & Multi-prosesor	*Busy waiting*, *starvation*.
Compare-and-Swap	Atomik membanding	Fleksibel untuk	Masalah ABA; masih	Uni-prosesor &	Masalah ABA, *busy*

(CAS)	kan nilai memori dengan nilai yang diharapkan, dan jika sama, memperbaru i ke nilai baru.	algoritma bebas *lock*; menjamin *up-to-date* data.	dapat melibatkan *busy waiting*.	Multi-proses or	*waiting*.

2. Algoritma Berbasis Perangkat Lunak (Software-based Algorithms)

Algoritma berbasis perangkat lunak adalah solusi *mutual exclusion* yang tidak bergantung pada instruksi perangkat keras khusus.

a. Algoritma Dekker (untuk 2 proses)

Algoritma Dekker adalah salah satu solusi berbasis perangkat lunak pertama untuk *mutual exclusion*, dirancang khusus untuk dua proses konkuren.[14] Algoritma ini menggunakan dua *flag* (menunjukkan niat masing-masing proses untuk memasuki *critical section*) dan satu variabel *turn* (menentukan proses mana yang memiliki prioritas jika keduanya ingin masuk secara bersamaan).[14] Dengan mengelola *flag* dan variabel *turn* ini, algoritma memastikan hanya satu proses yang dapat memasuki *critical section* pada satu waktu.[14] Kelebihan utamanya adalah bahwa ini adalah solusi pertama yang diketahui untuk *mutual exclusion* yang memenuhi properti *safety* (mutual exclusion), *liveness* (progress), dan *bounded bypass* (bounded waiting) tanpa bergantung pada instruksi perangkat keras atomik.[14] Namun, kekurangannya adalah keterbatasannya hanya untuk dua proses dan tidak berskala dengan baik untuk jumlah proses yang lebih banyak.[14] Selain itu, algoritma ini menggunakan *busy-waiting*, yang membuang siklus CPU saat proses menunggu.[14]

b. Algoritma Peterson (untuk 2 proses)

Algoritma Peterson, yang dirumuskan oleh Gary L. Peterson pada tahun 1981, juga merupakan algoritma pemrograman konkuren untuk *mutual exclusion* yang dirancang untuk dua proses.[13] Algoritma ini memungkinkan dua proses berbagi sumber daya tunggal tanpa konflik, hanya menggunakan memori bersama untuk komunikasi.[13] Ia menggunakan dua variabel: flag (menunjukkan niat proses untuk masuk *critical section*) dan turn (menentukan proses mana yang memiliki prioritas).[13] Misalnya, proses P0 diizinkan masuk jika P1 tidak ingin masuk atau P1 telah memberikan prioritas kepada P0 dengan mengatur turn ke 0.[13] Kelebihan Algoritma Peterson adalah

kemampuannya untuk memenuhi ketiga kriteria esensial masalah *critical section*: *mutual exclusion*, *progress*, dan *bounded waiting*. Selain itu, ia hanya membutuhkan memori minimal (tiga bit).[13] Namun, seperti Dekker, ia terbatas pada dua proses dan menggunakan *busy-waiting*.[13] Algoritma ini juga tidak berfungsi dengan benar jika eksekusi *out-of-order* digunakan pada platform yang mengeksekusinya, kecuali jika programmer secara eksplisit menentukan pengurutan ketat pada operasi memori.[3]

c. Algoritma Lamport's Bakery (untuk N proses)

Algoritma Lamport's Bakery adalah solusi berbasis perangkat lunak yang lebih umum untuk *mutual exclusion* yang dapat menangani N proses. Algoritma ini menggunakan sistem "nomor antrian" yang analog dengan sistem di toko roti. Setiap proses yang ingin memasuki *critical section* mengambil nomor, dan proses dengan nomor terkecil diizinkan masuk terlebih dahulu.[3] Kelebihan algoritma ini adalah kemampuannya untuk memenuhi *mutual exclusion*, *progress*, dan *bounded waiting* untuk sejumlah proses arbitrer. Namun, kekurangannya adalah kompleksitas implementasi yang lebih tinggi dibandingkan Dekker atau Peterson, dan ia masih menggunakan *busy-waiting*.

Algoritma perangkat lunak seperti Dekker dan Peterson adalah tonggak sejarah penting karena mereka menunjukkan bahwa *mutual exclusion* dapat dicapai tanpa dukungan perangkat keras khusus. Namun, mereka terbatas pada dua proses dan menderita masalah *busy-waiting*, yang tidak efisien. Lamport's Bakery memperluas konsep ini ke N proses, tetapi kompleksitasnya meningkat. Evolusi ini menunjukkan pergeseran dari solusi ad-hoc dan spesifik ke algoritma yang lebih umum dan terbukti benar, tetapi tetap menyoroti tantangan *busy-waiting*. Hal ini membuka jalan bagi primitif tingkat OS yang dapat memblokir proses alih-alih membuatnya *busy-wait*, seperti semaphores.

C. Contoh Penerapan Mutual Exclusion di Dunia Nyata

Kebutuhan akan *mutual exclusion* tidak hanya terbatas pada teori, tetapi sangat relevan dalam berbagai aplikasi dunia nyata:

- **Printer Spooling:** Dalam sistem operasi multi-pengguna, banyak proses atau individu mungkin meminta dokumen untuk dicetak secara bersamaan. *Mutual exclusion* digunakan untuk menjamin bahwa hanya satu proses pada satu waktu yang memiliki akses ke printer. Ini mencegah konflik dan memastikan bahwa pekerjaan cetak ditangani dalam urutan yang benar, seringkali menggunakan *lock* atau semaphore.[8]
- **Transaksi Rekening Bank:** Dalam sistem perbankan elektronik, banyak orang dapat secara bersamaan mencoba mengakses dan mengubah rekening bank mereka. *Mutual exclusion* sangat diperlukan untuk mencegah masalah seperti

overloading atau inkonsistensi data. Dengan *lock* atau primitif sinkronisasi lainnya, hanya satu transaksi yang diizinkan untuk mengakses rekening bank tertentu pada satu waktu, menjaga kerahasiaan informasi dan menghindari konflik.[9]

- **Sistem Kontrol Lalu Lintas:** Sinyal lalu lintas di persimpangan jalan harus dikoordinasikan untuk mengelola pergerakan kendaraan dengan aman. *Mutual exclusion* digunakan untuk menghindari sinyal yang saling bersaing ditampilkan secara bersamaan. Aturan *mutual exclusion* memastikan bahwa hanya satu indikator yang diizinkan aktif pada satu waktu, mempromosikan aliran lalu lintas yang efisien dan terorganisir.[9]

Contoh-contoh dunia nyata ini menunjukkan bahwa *mutual exclusion* bukan hanya konsep akademis, tetapi kebutuhan yang universal di berbagai sistem komputasi. Dari manajemen perangkat keras (printer) hingga sistem keuangan (bank) dan infrastruktur (lalu lintas), setiap skenario di mana sumber daya bersama dapat dimodifikasi oleh banyak entitas secara bersamaan memerlukan *mutual exclusion* untuk menjaga integritas dan fungsionalitas. Ini menegaskan bahwa pemahaman mendalam tentang *mutual exclusion* sangat penting bagi setiap *engineer* sistem.

III. Semaphore

A. Definisi dan Sejarah Singkat

Semaphore adalah variabel atau tipe data abstrak yang digunakan untuk mengontrol akses ke sumber daya umum oleh banyak *thread* dan menghindari masalah *critical section* dalam sistem konkuren, seperti sistem operasi *multitasking*.[18] Konsep semaphore diperkenalkan oleh ilmuwan komputer Belanda Edsger Dijkstra pada tahun 1962 atau 1963, saat ia dan timnya mengembangkan sistem operasi untuk Electrologica X8, yang kemudian dikenal sebagai sistem *multiprogramming* THE.[28] Semaphore merupakan salah satu *synchronization primitive* yang paling fundamental dan banyak digunakan dalam sistem operasi.[28]

Penciptaan semaphore oleh Dijkstra merupakan jembatan penting antara instruksi perangkat keras mentah dan abstraksi perangkat lunak tingkat tinggi. Ini memungkinkan abstraksi yang lebih baik untuk masalah sinkronisasi dibandingkan dengan algoritma *busy-waiting* murni. Semaphore memungkinkan sistem operasi untuk mengelola antrian proses yang menunggu secara efisien, mengurangi *busy-waiting* dan meningkatkan efisiensi CPU melalui *context switch*. Ini adalah langkah penting dalam evolusi *concurrency control*, menyediakan alat yang fleksibel namun tetap membutuhkan kehati-hatian programmer dalam penggunaannya.

B. Tipe Semaphore

Secara umum, terdapat dua tipe utama semaphore yang digunakan dalam sistem operasi:

1. Binary Semaphore (Mutex)

Binary semaphore, juga dikenal sebagai *mutex*, hanya dapat mengambil nilai 0 atau 1. Nilai 1 biasanya menunjukkan bahwa sumber daya tersedia atau tidak terkunci, sedangkan 0 menunjukkan bahwa sumber daya sedang digunakan atau terkunci.[18] Tipe semaphore ini digunakan untuk mengimplementasikan *lock* dan menyediakan *mutual exclusion* untuk satu instansi sumber daya.[18] Meskipun sering disebut *mutex*, perlu dicatat bahwa *binary semaphore* adalah jenis semaphore, sementara *mutex* adalah mekanisme penguncian yang menegakkan kepemilikan ketat, di mana hanya *thread* yang mengunci yang dapat membuka *lock* tersebut.[33]

2. Counting Semaphore

Counting semaphore dapat memiliki nilai bilangan bulat non-negatif yang lebih besar dari 1. Nilai ini merepresentasikan jumlah instansi sumber daya yang tersedia.[18] *Counting semaphore* digunakan untuk mengontrol akses ke kumpulan sumber daya

dengan jumlah instansi terbatas, seperti mengelola sepuluh ruang belajar identik di perpustakaan.[18]

Perbedaan antara semaphore biner dan *counting semaphore* menunjukkan fleksibilitas semaphore sebagai alat sinkronisasi. Semaphore biner secara langsung menangani *mutual exclusion* untuk sumber daya tunggal, sedangkan *counting semaphore* memperluas kemampuan ini untuk mengelola kumpulan sumber daya yang identik. Ini memungkinkan *engineer* untuk memilih mekanisme yang paling sesuai dengan granularitas sumber daya yang perlu dilindungi, dari akses eksklusif ke satu *critical section* hingga manajemen kapasitas untuk beberapa instansi sumber daya.

Tabel 2 menyajikan perbandingan antara *binary semaphore* dan *counting semaphore*:

Tabel 2: Perbandingan Binary Semaphore dan Counting Semaphore

Fitur	Binary Semaphore (Mutex)	Counting Semaphore
Nilai yang Diizinkan	0 atau 1	Bilangan bulat non-negatif (>= 0)
Tujuan Utama	Mutual exclusion (mengunci akses ke satu sumber daya/critical section)	Mengontrol akses ke kumpulan sumber daya dengan banyak instansi
Kasus Penggunaan Khas	Mengimplementasikan *lock*, melindungi *critical section*	Mengelola jumlah slot login, ruang belajar, *buffer* terbatas
Hubungan dengan Mutex	Sering disebut *mutex*, berfungsi sebagai *lock* sederhana	Tidak secara langsung terkait dengan *mutex*

C. Operasi Atomik Semaphore: P (Wait/Down) dan V (Signal/Up)

Semaphore beroperasi melalui dua operasi atomik utama: P (juga dikenal sebagai *Wait* atau *Down*) dan V (juga dikenal sebagai *Signal* atau *Up*).[18]

- **P (Wait/Down):** Operasi ini mengurangi nilai semaphore. Jika nilai semaphore menjadi negatif (atau 0, tergantung definisi implementasi), proses yang mengeksekusi operasi ini akan diblokir dan ditambahkan ke antrian semaphore. Proses tersebut akan tetap diblokir hingga nilai semaphore menjadi positif kembali.[15]

- **V (Signal/Up):** Operasi ini meningkatkan nilai semaphore. Jika ada satu atau lebih proses yang menunggu di antrian semaphore (karena nilai semaphore sebelumnya negatif atau 0), salah satu proses tersebut akan dipilih (biasanya dengan kebijakan FIFO) untuk di-unblock dan melanjutkan eksekusi.[15]

Sangat penting bahwa operasi P dan V dilakukan sebagai tindakan tunggal, tidak terpisahkan, dan atomik. Atomicity ini memastikan bahwa tidak ada proses lain yang dapat mengakses atau memodifikasi semaphore saat operasi P atau V sedang berlangsung, sehingga mencegah *race conditions* pada semaphore itu sendiri.[28] Untuk menghindari *starvation*, semaphore biasanya memiliki antrian proses terkait, yang seringkali diimplementasikan dengan semantik FIFO (First-In, First-Out).[28]

Atomicity dari operasi P dan V sangat krusial. Tanpa atomicity, dua proses dapat secara bersamaan membaca nilai semaphore, memutuskan untuk melanjutkan, dan kemudian memodifikasinya, yang mengarah pada nilai semaphore yang tidak konsisten atau pelanggaran *mutual exclusion* yang seharusnya dijamin oleh semaphore itu sendiri. Ini menegaskan kembali bahwa atomicity adalah prasyarat fundamental untuk setiap primitif sinkronisasi yang andal, bahkan pada tingkat abstraksi yang lebih tinggi seperti semaphore.

D. Penerapan Semaphore dalam Masalah Sinkronisasi Klasik

Semaphore sangat efektif dalam memecahkan masalah sinkronisasi klasik dalam sistem operasi.

1. Masalah Producer-Consumer

Deskripsi Masalah: Masalah *Producer-Consumer* melibatkan dua jenis proses: produsen yang menghasilkan data dan konsumen yang mengonsumsi data tersebut. Keduanya berbagi *buffer* berukuran terbatas. Produsen harus menunggu jika *buffer* penuh, dan konsumen harus menunggu jika *buffer* kosong.[28]

Solusi dengan Semaphore: Solusi klasik untuk masalah ini menggunakan tiga semaphore [30]:

- full: Menghitung jumlah slot *buffer* yang sudah terisi. Diinisialisasi ke 0.
- empty: Menghitung jumlah slot *buffer* yang kosong. Diinisialisasi ke N (ukuran total *buffer*).
- mutex: Untuk menjamin *mutual exclusion* saat produsen atau konsumen mengakses *buffer* secara bersamaan. Diinisialisasi ke 1.

Pseudocode (Producer):

```
Producer ( )
WHILE (true)
   produce-Item ( );  // Produsen membuat item baru
   P (empty);        // Menunggu slot kosong di buffer (decrement empty)
   P (mutex);        // Mengunci akses ke buffer (decrement mutex)
   enter-Item ( );    // Menambahkan item ke buffer (critical section)
   V (mutex);        // Melepaskan kunci buffer (increment mutex)
   V (full);         // Memberi sinyal bahwa ada item baru di buffer (increment full)
```

Pseudocode (Consumer):

```
Consumer ( )
WHILE (true)
   P (full);         // Menunggu item tersedia di buffer (decrement full)
   P (mutex);        // Mengunci akses ke buffer (decrement mutex)
   remove-Item ( );   // Mengambil item dari buffer (critical section)
   V (mutex);        // Melepaskan kunci buffer (increment mutex)
   V (empty);        // Memberi sinyal bahwa ada slot kosong di buffer (increment empty)
   consume-Item (Item) // Konsumen memproses item yang diambil
```

Solusi *Producer-Consumer* menunjukkan bagaimana beberapa semaphore dapat dikoordinasikan untuk memecahkan masalah sinkronisasi yang lebih kompleks daripada sekadar *mutual exclusion*. Semaphore mutex menjaga integritas *buffer* itu sendiri, sementara full dan empty mengelola kondisi "penuh" dan "kosong", mencegah *buffer overflow* dan *underflow*. Ini adalah contoh klasik dari bagaimana semaphore, meskipun primitif, dapat digabungkan untuk mencapai *concurrency control* yang canggih, namun juga menunjukkan potensi kompleksitas dalam penempatan operasi P dan V yang benar.

2. Masalah Readers-Writers

Deskripsi Masalah: Masalah *Readers-Writers* melibatkan banyak *thread* pembaca dan penulis yang mencoba mengakses sumber daya bersama. Aturannya adalah

banyak pembaca diizinkan untuk mengakses sumber daya secara bersamaan, tetapi penulis memerlukan akses eksklusif, artinya tidak boleh ada pembaca atau penulis lain saat seorang penulis sedang beroperasi.[8]

Variasi dan Solusi dengan Semaphore:

- **Readers-Preference (Solusi Pertama):** Prioritas diberikan kepada pembaca. Pembaca tidak akan menunggu jika sumber daya sedang dibaca.[8] Namun, isu yang mungkin timbul adalah penulis dapat mengalami *starvation* jika ada aliran pembaca yang terus-menerus, karena penulis tidak akan pernah mendapatkan kesempatan untuk menulis.[35]
- **Writers-Preference (Solusi Kedua):** Prioritas diberikan kepada penulis. Penulis tidak akan menunggu lebih lama dari yang diperlukan.[35] Dalam skenario ini, pembaca dapat mengalami *starvation* jika ada aliran penulis yang terus-menerus.[35]
- **Fair Solution (Solusi Ketiga):** Bertujuan mencegah *starvation* baik untuk pembaca maupun penulis, seringkali dengan menggunakan semaphore antrian layanan (FIFO) untuk memastikan urutan permintaan yang adil.[35]

Masalah *Readers-Writers* adalah contoh sempurna dari *trade-off* inheren dalam desain sistem konkuren. Memberikan preferensi kepada satu jenis proses (pembaca atau penulis) secara langsung menyebabkan potensi *starvation* untuk jenis proses lainnya. Solusi "adil" mencoba menyeimbangkan ini, tetapi seringkali dengan menambah kompleksitas atau *overhead*. Ini mengajarkan bahwa tidak ada solusi "satu ukuran cocok untuk semua" dalam sinkronisasi, dan pilihan desain harus mempertimbangkan persyaratan kinerja dan keadilan spesifik aplikasi.

E. Keterbatasan dan Potensi Masalah Penggunaan Semaphore

Meskipun semaphore adalah alat sinkronisasi yang kuat dan fleksibel, penggunaannya tidak lepas dari keterbatasan dan potensi masalah:

- **Kompleksitas Pemrograman:** Semaphore memerlukan penempatan operasi wait() (P) dan signal() (V) yang sangat cermat oleh programmer. Kesalahan kecil dalam penempatan ini dapat dengan mudah menyebabkan masalah serius seperti *deadlock* (proses menunggu tanpa batas) atau *starvation* (proses tidak pernah mendapatkan akses ke sumber daya).[18]
- **Busy Waiting:** Dalam implementasi sederhana, terutama pada sistem uni-prosesor atau ketika *contention* tinggi, proses yang gagal memperoleh semaphore mungkin akan terus-menerus memeriksa nilainya dalam *loop* (busy-waiting). Ini membuang siklus CPU yang berharga dan mengurangi efisiensi sistem.[3]

- **Priority Inversion:** Semaphore dapat menyebabkan masalah *priority inversion*, di mana proses dengan prioritas tinggi mungkin harus menunggu proses dengan prioritas rendah yang sedang memegang semaphore yang dibutuhkan.[3] Ini dapat mengganggu responsivitas sistem, terutama dalam sistem *real-time*.
- **Kesulitan Debugging:** Masalah sinkronisasi yang disebabkan oleh penggunaan semaphore yang tidak tepat seringkali sulit untuk di-debug karena sifatnya yang bergantung pada waktu dan intermiten.[18]

Meskipun semaphore sangat fleksibel dan kuat, keterbatasan utamanya terletak pada "beban" yang diletakkan pada programmer. Penempatan P dan V yang tidak tepat dapat dengan mudah menyebabkan *deadlock*, *starvation*, atau *race conditions* yang luput dari perhatian. Ini adalah masalah *correctness by construction* — semaphore tidak secara inheren menjamin kebenaran; ia hanya menyediakan primitif. Kondisi ini memotivasi pengembangan konstruksi sinkronisasi tingkat lebih tinggi seperti monitor yang mengotomatiskan banyak aspek sinkronisasi dan mengurangi ruang lingkup kesalahan programmer.

IV. Monitor

A. Definisi dan Konsep Abstraksi Tingkat Tinggi

Monitor adalah konstruk sinkronisasi tingkat tinggi yang dirancang untuk menyederhanakan sinkronisasi proses dengan menyediakan abstraksi tingkat tinggi untuk akses data dan sinkronisasi.[10] Konsep monitor diimplementasikan sebagai konstruksi bahasa pemrograman (misalnya, di Java, C#, Ada) dan mengintegrasikan *mutual exclusion*, variabel kondisi, serta enkapsulasi data dalam satu unit.[10] Monitor dapat dianggap sebagai generalisasi alami dari objek dalam pemrograman berorientasi objek, di mana deklarasi data dan operasi dienkapsulasi dalam sebuah kelas.[37]

Munculnya monitor adalah respons langsung terhadap kompleksitas dan kerentanan kesalahan dalam penggunaan semaphore. Dengan mengenkapsulasi data bersama dan prosedur yang memodifikasinya dalam satu unit, monitor menyediakan abstraksi yang lebih tinggi. Ini secara fundamental mengurangi beban pada programmer dengan mengotomatiskan banyak aspek *mutual exclusion* dan *condition synchronization*, memungkinkan programmer untuk fokus pada logika bisnis daripada detail sinkronisasi tingkat rendah. Ini adalah pergeseran paradigma dari "primitif" ke "struktur" dalam desain sinkronisasi.

B. Struktur Monitor

Struktur monitor dirancang untuk memfasilitasi sinkronisasi yang aman dan terstruktur:

- **Data Bersama dan Prosedur Terenkapsulasi:** Monitor adalah modul yang mengenkapsulasi sumber daya bersama dan menyediakan akses ke sumber daya tersebut melalui serangkaian prosedur. Proses yang berada di luar monitor tidak dapat mengakses variabel internal monitor secara langsung; mereka harus memanggil prosedur yang disediakan oleh monitor untuk berinteraksi dengan sumber daya bersama.[10]
- **Kunci Implisit (Implicit Lock) untuk Mutual Exclusion:** Setiap monitor memiliki *lock* implisit pada "pintu" masuknya, yang memastikan bahwa hanya satu proses yang dapat berada "di dalam" monitor pada satu waktu.[37] *Mutual exclusion* secara otomatis ditegakkan untuk semua operasi monitor, sehingga programmer tidak perlu secara eksplisit mengelola *lock* ini.
- **Variabel Kondisi (Condition Variables) dan Antrian Terkait:** Monitor menggunakan variabel kondisi untuk mengelola proses yang perlu menunggu kondisi tertentu menjadi benar sebelum melanjutkan eksekusi. Setiap variabel kondisi memiliki antrian blokir uniknya sendiri untuk proses-proses yang

menunggu.[10]

Struktur monitor, dengan enkapsulasi data dan *lock* implisit, secara proaktif mencegah *race conditions* pada data bersama. Ini berarti programmer tidak perlu secara manual menempatkan *lock* dan *unlock* di setiap *critical section*, mengurangi kemungkinan kesalahan. Kombinasi *mutual exclusion* otomatis dan variabel kondisi yang terkelola dengan baik membuat monitor menjadi konstruksi yang sangat kuat untuk menjaga konsistensi data dalam program konkuren.

C. Prinsip Kerja Monitor: Sinkronisasi Kondisi dengan wait() dan signal()

Monitor mencapai sinkronisasi kondisi melalui penggunaan operasi wait() dan signal() pada variabel kondisi:

- **waitC(condition) (atau x.wait()):** Ketika sebuah proses mengeksekusi waitC(condition) pada variabel kondisi x, proses tersebut diblokir pada kondisi tersebut. Proses yang diblokir ini kemudian ditambahkan ke antrian FIFO yang terkait dengan variabel kondisi x tersebut. Setelah diblokir, proses tersebut secara implisit meninggalkan monitor, melepaskan *lock mutual exclusion* yang dipegangnya. Ini memungkinkan proses lain untuk masuk ke monitor dan berpotensi mengubah kondisi yang ditunggu.[10]
- **signalC(condition) (atau x.signal()/notify()):** Ketika signalC(condition) dieksekusi, ini menandakan bahwa kondisi yang ditentukan telah menjadi benar. Jika antrian yang terkait dengan variabel kondisi x tidak kosong, salah satu proses yang menunggu di kepala antrian tersebut akan di-unblock dan menjadi siap untuk dieksekusi kembali. Namun, jika antrian kosong, sinyal tersebut diabaikan dan tidak memiliki efek.[10]

Mekanisme wait() dan signal() pada variabel kondisi adalah kunci efisiensi monitor. Alih-alih *busy-waiting*, proses yang tidak dapat melanjutkan karena suatu kondisi tidak terpenuhi dapat "tidur" dan melepaskan *lock* monitor, memungkinkan proses lain untuk masuk dan berpotensi mengubah kondisi yang dibutuhkan. Ini adalah peningkatan signifikan dibandingkan semaphore yang sederhana, yang mungkin memerlukan *busy-waiting* atau desain yang lebih rumit untuk mencapai sinkronisasi kondisi yang serupa.

D. Keunggulan Monitor dibandingkan Semaphore

Monitor menawarkan beberapa keunggulan signifikan dibandingkan semaphore, terutama dalam hal abstraksi dan kemudahan penggunaan:

- **Abstraksi Lebih Tinggi:** Monitor menyediakan tingkat abstraksi yang lebih tinggi,

menyembunyikan detail sinkronisasi yang rumit dari programmer.[10] Ini memungkinkan programmer untuk fokus pada logika aplikasi daripada detail implementasi sinkronisasi tingkat rendah.

- **Kurang Rentan Kesalahan:** Monitor membuat pemrograman paralel lebih mudah dan secara inheren kurang rentan terhadap kesalahan dibandingkan dengan menggunakan teknik seperti semaphore. Hal ini karena logika sinkronisasi dienkapsulasi di dalam monitor, mengurangi kemungkinan kesalahan penempatan operasi wait dan signal yang tidak tepat.[10]
- **Mutual Exclusion Otomatis:** *Mutual exclusion* secara otomatis ditegakkan untuk semua operasi monitor. Programmer tidak perlu secara manual menempatkan operasi wait dan signal untuk memastikan akses eksklusif ke data bersama, karena ini ditangani secara implisit oleh kompilator atau *runtime*.[37]
- **Enkapsulasi Data dan Operasi:** Monitor membungkus data bersama dan metode yang memodifikasinya dalam satu unit yang rapi. Ini meningkatkan modularitas kode dan membantu menjaga konsistensi data dengan membatasi akses hanya melalui prosedur monitor yang terkontrol.[26]

Keunggulan utama monitor adalah pergeseran beban dari "bagaimana" sinkronisasi diimplementasikan (detail P dan V pada semaphore) ke "apa" yang perlu disinkronkan (data bersama dan operasi yang memodifikasinya). Ini adalah contoh klasik dari bagaimana abstraksi yang lebih tinggi dalam ilmu komputer dapat menyederhanakan tugas-tugas yang kompleks, mengurangi kesalahan manusia, dan meningkatkan produktivitas pengembang, meskipun mungkin dengan sedikit *overhead* kinerja dibandingkan dengan implementasi tingkat terendah yang dioptimalkan secara manual.

Tabel 3 menyajikan perbandingan antara semaphore dan monitor:

Tabel 3: Perbandingan Semaphore dan Monitor

Fitur	Semaphore	Monitor
Tingkat Abstraksi	Primitif tingkat rendah	Konstruk tingkat tinggi
Enkapsulasi Data	Tidak ada enkapsulasi data bersama; hanya variabel integer	Mengenkapsulasi data bersama dan operasinya
Mutual Exclusion	Perlu diimplementasikan secara eksplisit dengan	Ditegakkan secara otomatis melalui kunci implisit

	P()/V()	
Sinkronisasi Kondisi	Perlu diimplementasikan secara manual dengan P()/V()	Menggunakan variabel kondisi (wait()/signal())
Potensi Kesalahan	Sangat rentan kesalahan (deadlock, starvation) karena penempatan P()/V() yang salah	Kurang rentan kesalahan karena abstraksi dan enkapsulasi
Dukungan Bahasa	Primitif OS, dapat diimplementasikan di berbagai bahasa	Konstruk bahasa pemrograman (mis. Java synchronized)
Kompleksitas Pemrograman	Lebih kompleks, membutuhkan kehati-hatian tinggi	Lebih sederhana, menyembunyikan detail sinkronisasi

E. Keterbatasan Monitor

Meskipun monitor menawarkan keuntungan signifikan dalam menyederhanakan pemrograman konkuren, mereka juga memiliki keterbatasan:

- **Dukungan Bahasa Pemrograman:** Monitor harus diimplementasikan sebagai bagian dari bahasa pemrograman itu sendiri. Ini berarti kompilator harus menghasilkan kode khusus untuk monitor, yang menambah beban pada kompilator dan mensyaratkan pengetahuan tentang fasilitas sistem operasi yang tersedia untuk mengontrol akses ke *critical section*.[10]
- **Overhead:** Monitor mungkin kurang efisien dibandingkan primitif sinkronisasi tingkat rendah seperti semaphore dan *lock* karena *overhead* tambahan yang terkait dengan abstraksi tingkat tingginya.[10] Dalam aplikasi yang sangat sensitif terhadap kinerja, *overhead* ini bisa menjadi faktor penentu.
- **Tidak Selalu Cocok:** Monitor mungkin tidak cocok untuk semua jenis masalah sinkronisasi. Dalam skenario tertentu, primitif tingkat rendah mungkin diperlukan untuk mencapai kinerja optimal atau untuk menangani pola sinkronisasi yang sangat spesifik yang tidak sesuai dengan model monitor.[10]
- **Potensi Starvation (jika antrian tidak FIFO):** Meskipun monitor memiliki antrian yang terkait dengan variabel kondisi, beberapa implementasi mungkin tidak menjamin urutan FIFO. Jika antrian tidak ketat FIFO, *starvation* masih mungkin terjadi, di mana proses tertentu mungkin tidak pernah mendapatkan giliran untuk

melanjutkan.[37]

Abstraksi datang dengan biaya dan batasan. Meskipun monitor menawarkan keuntungan signifikan dalam menyederhanakan pemrograman konkuren, mereka tidak tanpa batasan. Ketergantungan pada dukungan bahasa berarti fleksibilitas implementasi dapat dibatasi. Selain itu, *overhead* yang terkait dengan abstraksi tingkat tinggi dapat menjadi faktor dalam aplikasi yang sangat sensitif terhadap kinerja. Ini menggarisbawahi prinsip bahwa setiap abstraksi dalam ilmu komputer membawa *trade-off* antara kemudahan penggunaan dan kinerja/fleksibilitas.

F. Contoh Penerapan Monitor (e.g., Java synchronized methods)

Konsep monitor telah diintegrasikan langsung ke dalam banyak bahasa pemrograman modern, menjadikannya alat yang kuat dan mudah diakses untuk *concurrency control*. Contoh paling menonjol adalah di Java:

- **Java synchronized methods/blocks:** Setiap objek di Java memiliki monitor terkait yang dapat dikunci dan dibuka oleh *thread*. Ketika sebuah metode atau blok kode dideklarasikan sebagai synchronized, Java secara implisit memperoleh *lock* monitor objek di awal eksekusi dan melepaskannya di akhir.[10] Ini secara otomatis menjamin *mutual exclusion* untuk kode di dalam metode atau blok tersebut.
- **Masalah Producer-Consumer:** Implementasi *bounded buffer* menggunakan monitor seringkali lebih intuitif dan mudah dikelola dibandingkan solusi berbasis semaphore. Monitor dapat mengenkapsulasi *buffer* dan menyediakan metode put() dan get() yang disinkronkan, dengan variabel kondisi untuk menangani kasus *buffer* penuh atau kosong.[40]
- **Masalah Readers-Writers:** Monitor juga dapat digunakan untuk menyelesaikan masalah *Readers-Writers* dengan mengelola antrian pembaca dan penulis menggunakan variabel kondisi di dalam monitor. Ini memungkinkan kontrol yang lebih terstruktur atas akses ke sumber daya bersama.[40]

Contoh Java synchronized methods menunjukkan bagaimana konsep monitor telah diintegrasikan langsung ke dalam bahasa pemrograman modern, menjadikannya alat yang mudah diakses dan kuat untuk *concurrency control*. Ini menegaskan relevansi monitor dalam praktik pengembangan *software* saat ini, memungkinkan pengembang untuk menulis kode konkuren yang lebih bersih dan aman tanpa harus berurusan dengan detail sinkronisasi tingkat rendah secara eksplisit.

V. Isu Umum dalam Sinkronisasi: Deadlock dan Starvation

Meskipun mekanisme sinkronisasi seperti semaphore dan monitor dirancang untuk mengatasi *race conditions*, penggunaannya yang tidak tepat atau kondisi sistem tertentu dapat menimbulkan masalah serius lainnya, yaitu *deadlock* dan *starvation*.

A. Deadlock

1. Definisi dan Empat Kondisi yang Diperlukan

Deadlock adalah situasi dalam komputasi konkuren di mana tidak ada anggota dari suatu kelompok entitas yang dapat melanjutkan eksekusi karena masing-masing menunggu anggota lain, termasuk dirinya sendiri, untuk mengambil tindakan, seperti melepaskan *lock*.[44] *Deadlock* adalah masalah umum dalam sistem *multiprocessing*, komputasi paralel, dan sistem terdistribusi, di mana *lock* perangkat lunak atau perangkat keras sering digunakan untuk mengarbitrasi sumber daya bersama dan mengimplementasikan sinkronisasi proses.[44]

Deadlock dapat terjadi jika dan hanya jika keempat kondisi berikut terpenuhi secara bersamaan dalam sistem [44]:

- **Mutual Exclusion:** Setidaknya satu sumber daya harus dipegang dalam mode non-shareable, artinya hanya satu proses yang dapat menggunakan sumber daya tersebut pada satu waktu. Jika tidak, proses tidak akan terhalang untuk menggunakan sumber daya saat diperlukan.[44]
- **Hold and Wait:** Suatu proses sedang memegang setidaknya satu sumber daya dan meminta sumber daya tambahan yang sedang dipegang oleh proses lain.[27]
- **No Preemption:** Sumber daya hanya dapat dilepaskan secara sukarela oleh proses yang memegangnya. Sumber daya tidak dapat diambil secara paksa dari proses tersebut oleh proses lain atau sistem operasi.[27]
- **Circular Wait:** Terdapat serangkaian proses, P = {P1, P2,..., PN}, sedemikian rupa sehingga P1 menunggu sumber daya yang dipegang oleh P2, P2 menunggu sumber daya yang dipegang oleh P3, dan seterusnya hingga PN menunggu sumber daya yang dipegang oleh P1, membentuk siklus.[44]

Deadlock adalah masalah *liveness* yang sangat sulit dihindari dalam sistem konkurensi karena ia muncul dari kombinasi empat kondisi yang seringkali diinginkan (misalnya, *mutual exclusion* untuk integritas data). Strategi penanganannya mencerminkan *trade-off* antara konservatisme (pencegahan/penghindaran) dan *overhead*/kerugian (deteksi/pemulihan). Memahami empat kondisi ini adalah kunci untuk merancang sistem yang tahan *deadlock*, dan ini seringkali melibatkan kompromi antara kinerja

dan keandalan.

Tabel 4 meringkas empat kondisi yang diperlukan untuk *deadlock*:

Tabel 4: Empat Kondisi yang Diperlukan untuk Deadlock

Kondisi	Definisi Singkat	Contoh/Implikasi
Mutual Exclusion	Setidaknya satu sumber daya harus non-shareable.	Printer, *lock* pada struktur data bersama.
Hold and Wait	Proses memegang satu sumber daya dan menunggu yang lain.	Proses A memegang R1, menunggu R2 yang dipegang Proses B.
No Preemption	Sumber daya tidak dapat diambil paksa.	Memori yang sedang ditulis tidak dapat diambil.
Circular Wait	Setiap proses menunggu sumber daya yang dipegang oleh proses berikutnya dalam sebuah siklus.	P1 menunggu R2 (dipegang P2), P2 menunggu R1 (dipegang P1).

2. Strategi Penanganan Deadlock

Ada tiga strategi utama untuk menangani *deadlock* [46]:

- **Pencegahan (Prevention):** Strategi ini berfokus pada penghapusan salah satu dari empat kondisi yang diperlukan agar *deadlock* tidak pernah terjadi.[46] Contohnya termasuk:
  - Menghilangkan *hold and wait* dengan mengharuskan proses meminta semua sumber daya yang dibutuhkan di awal, atau melepaskan semua sumber daya yang dipegang sebelum meminta yang baru.[47]
  - Menghilangkan *circular wait* dengan menetapkan urutan numerik unik untuk setiap sumber daya dan mengharuskan proses meminta sumber daya hanya dalam urutan yang meningkat.[47]
- **Penghindaran (Avoidance):** Pendekatan ini lebih dinamis. Sistem memeriksa setiap permintaan sumber daya untuk memastikan bahwa alokasi tersebut tidak akan menyebabkan *deadlock* di masa depan. Algoritma Banker adalah contoh klasik dari strategi penghindaran *deadlock*, yang memerlukan informasi awal tentang kebutuhan sumber daya maksimum setiap proses.[44] Permintaan hanya

diberikan jika sistem tetap dalam "keadaan aman."
- **Deteksi & Pemulihan (Detection & Recovery):** Dalam strategi ini, *deadlock* diizinkan terjadi. Sistem secara berkala mendeteksi keberadaan *deadlock* (misalnya, dengan menganalisis *resource-allocation graph* untuk siklus) dan kemudian memulihkan diri. Pemulihan dapat melibatkan penghentian satu atau lebih proses yang terlibat dalam *deadlock*, atau *preemption* (pengambilan paksa) sumber daya dari proses tertentu.[44]

B. Starvation

1. Definisi dan Penyebab

Starvation adalah masalah di mana suatu proses secara terus-menerus ditolak sumber daya yang diperlukan untuk melanjutkan eksekusinya, meskipun sumber daya tersebut mungkin tersedia dari waktu ke waktu.[3] Berbeda dengan *deadlock* di mana tidak ada proses yang maju, dalam *starvation*, beberapa proses dapat maju, tetapi proses tertentu tidak pernah mendapatkan kesempatan.

Penyebab *starvation* meliputi:

- **Algoritma Penjadwalan yang Terlalu Sederhana:** Jika sistem menggunakan algoritma penjadwalan yang selalu memprioritaskan tugas prioritas tinggi, proses dengan prioritas rendah mungkin tidak pernah mendapatkan waktu CPU atau sumber daya lain.[39]
- **Pemilihan Proses Acak:** Jika sistem menggunakan pemilihan proses acak untuk alokasi sumber daya, beberapa proses mungkin menunggu untuk waktu yang sangat lama karena tidak pernah terpilih.[48]
- **Strategi Alokasi Sumber Daya yang Bias atau Rusak:** Kebijakan alokasi sumber daya yang terus-menerus menguntungkan tugas prioritas tinggi dapat mencegah proses tertentu memperoleh sumber daya yang mereka butuhkan.[48]
- **Implementasi Semaphore yang Tidak FIFO:** Jika antrian proses yang menunggu pada semaphore tidak dijamin FIFO, proses dengan prioritas lebih rendah mungkin tidak pernah mendapatkan giliran, bahkan jika semaphore dilepaskan.[28]

Starvation adalah isu keadilan dalam alokasi sumber daya. Solusi untuk *starvation* berfokus pada memastikan bahwa setiap proses pada akhirnya akan mendapatkan akses ke sumber daya yang dibutuhkan, seringkali melalui modifikasi pada algoritma penjadwalan atau mekanisme antrian. Memahami *starvation* penting untuk merancang sistem yang tidak hanya benar tetapi juga adil dan responsif bagi semua komponennya.

2. Mitigasi Starvation

Beberapa teknik dapat digunakan untuk memitigasi atau mencegah *starvation*:

- **Aging:** Ini adalah teknik di mana prioritas proses yang menunggu di sistem untuk waktu yang lama secara bertahap ditingkatkan. Dengan demikian, proses yang "lapar" pada akhirnya akan mencapai prioritas yang cukup tinggi untuk mendapatkan akses ke sumber daya.[39]
- **Bounded Waiting:** Memastikan bahwa suatu proses akan menunggu tidak lebih dari jumlah waktu atau giliran yang terbatas sebelum diizinkan masuk ke *critical section* atau mendapatkan sumber daya.[3]
- **Timeouts:** Mekanisme *timeout* dapat digunakan untuk membatasi jumlah waktu proses dapat menunggu sumber daya. Jika sumber daya tidak tersedia dalam periode *timeout* yang ditentukan, proses dapat dipaksa untuk melepaskan sumber daya yang sedang dipegangnya dan mencoba lagi nanti.[47] Ini mencegah penundaan tak terbatas.

VI. Perkembangan Terbaru dan Praktik Terbaik dalam Kontrol Konkurensi

Bidang kontrol konkurensi terus berkembang seiring dengan kemajuan arsitektur perangkat keras dan kebutuhan sistem yang semakin kompleks, terutama dalam konteks sistem multi-core dan terdistribusi.

A. Algoritma Bebas Kunci (Lock-Free Algorithms)

Algoritma bebas kunci (lock-free algorithms) adalah jenis algoritma non-blocking yang dirancang untuk memastikan bahwa kegagalan atau penundaan satu *thread* tidak menyebabkan kegagalan atau penundaan *thread* lain.[23] Algoritma ini merupakan alternatif dari implementasi *blocking* tradisional yang menggunakan *lock*. Ada dua tipe utama algoritma non-blocking:

- **Lock-free:** Algoritma ini menjamin *system-wide progress*, artinya jika *thread* program dijalankan cukup lama, setidaknya satu dari *thread* tersebut akan membuat kemajuan.[23] Ini berarti *thread* yang ditangguhkan tidak akan memblokir kemajuan *thread* lainnya.
- **Wait-free:** Ini adalah jaminan yang lebih kuat, menjamin *per-thread progress*. Setiap *thread* dijamin berhasil menyelesaikan operasinya dalam sejumlah langkah terbatas, terlepas dari kecepatan atau penundaan *thread* lain.[23]

Algoritma bebas kunci seringkali dibangun di atas instruksi atomik tingkat rendah seperti Compare-and-Swap (CAS).[3] Kelebihan utama dari algoritma ini adalah kemampuannya untuk meningkatkan paralelisme pada prosesor multi-core dan ketangguhannya terhadap kegagalan *thread*, karena *thread* yang ditangguhkan tidak akan memblokir kemajuan *thread* lain.[23] Namun, mereka datang dengan biaya kompleksitas desain yang tinggi, potensi *overhead* kinerja pada jumlah prosesor kecil, dan masih rentan terhadap masalah seperti ABA problem.[20]

Meskipun *lock* tradisional (mutex, semaphore) efektif untuk *mutual exclusion*, mereka dapat membatasi paralelisme dan rentan terhadap *deadlock* atau *priority inversion*. Algoritma bebas *lock* muncul sebagai respons terhadap batasan ini, terutama di arsitektur multi-core. Mereka mencoba mencapai konkurensi tanpa memblokir *thread*, mengandalkan operasi atomik tingkat rendah. Ini adalah tren penting dalam penelitian sistem operasi modern yang berusaha memaksimalkan pemanfaatan perangkat keras dan ketahanan sistem, meskipun dengan biaya kompleksitas yang signifikan.

B. Memori Transaksional (Transactional Memory - STM, HTM)

Memori Transaksional (TM) adalah mekanisme kontrol konkurensi yang analog dengan

transaksi basis data, digunakan untuk mengontrol akses ke memori bersama dalam komputasi konkuren.[52] TM bertujuan untuk menyederhanakan sinkronisasi dalam sistem *multi-threaded* dengan memungkinkan pengembang mengelola sumber daya bersama tanpa kompleksitas *lock* eksplisit.[52]

TM menjamin tiga properti penting untuk serangkaian operasi baca/tulis ke memori bersama [52]:

- **Atomicity:** Transaksi dieksekusi sebagai unit yang tidak terpisahkan.
- **Consistency:** Memori bersama tetap dalam keadaan yang valid.
- **Isolation:** Transaksi diisolasi satu sama lain, menghindari *race conditions*.

Ada beberapa tipe implementasi TM:

- **Software Transactional Memory (STM):** Diimplementasikan sepenuhnya dalam perangkat lunak. Ini lebih fleksibel dan adaptif untuk berbagai sistem.[52]
- **Hardware Transactional Memory (HTM):** Memanfaatkan dukungan perangkat keras asli (misalnya, Intel Transactional Synchronization Extensions/TSX). HTM menawarkan *overhead* yang lebih rendah dan kinerja tinggi untuk transaksi kecil dan sering.[52]
- **Hybrid Transactional Memory (HyTM):** Menggabungkan kekuatan STM dan HTM untuk fleksibilitas dan kinerja yang lebih besar.[52]

Kelebihan utama TM adalah kemampuannya untuk menyederhanakan pemrograman konkuren (tidak perlu *lock* eksplisit), meningkatkan skalabilitas, dan meningkatkan keandalan sistem.[52] Namun, TM juga memiliki kekurangan, seperti *overhead* yang signifikan dari konflik transaksi yang sering, dukungan perangkat keras yang terbatas untuk HTM, dan tantangan dalam mengintegrasikan operasi I/O yang non-deterministik dalam transaksi.[52]

Memori Transaksional adalah upaya untuk mengangkat abstraksi sinkronisasi ke tingkat yang lebih tinggi lagi, mirip dengan bagaimana transaksi basis data menyederhanakan operasi kompleks. Ini bertujuan untuk sepenuhnya menghilangkan kompleksitas *lock* dan semaphore dari programmer, memungkinkan mereka untuk mendeklarasikan "blok atomik" kode. Ini adalah tren yang sangat menarik yang menunjukkan keinginan untuk menyederhanakan pemrograman konkuren, terutama di lingkungan multi-core yang semakin kompleks, meskipun masih dalam tahap penelitian dan adopsi yang berkembang.

C. Kontrol Konkurensi dalam Sistem Terdistribusi (Optimistic vs. Pessimistic)

Kontrol konkurensi dalam sistem terdistribusi adalah konsep fundamental yang

memastikan banyak transaksi dapat mengakses atau memodifikasi data secara bersamaan tanpa menyebabkan kesalahan atau inkonsistensi, menjaga properti ACID (Atomicity, Consistency, Isolation, Durability) atau BASE (Basically Available, Soft state, Eventually consistent).[56] Tantangan di sini diperparah oleh latensi jaringan, kegagalan parsial, dan tidak adanya memori bersama global.

Dua model utama kontrol konkurensi dalam sistem terdistribusi adalah:

- **Pessimistic Concurrency Control (PCC):** Pendekatan ini berasumsi bahwa konflik sering terjadi. Oleh karena itu, ia mengunci sumber daya di awal transaksi untuk mencegah konflik. Meskipun ini menjamin isolasi yang ketat, hal itu dapat mengurangi konkurensi keseluruhan sistem.[57] Contoh metode PCC termasuk Two-Phase Locking (2PL) dan Distributed Lock Manager.[58]
- **Optimistic Concurrency Control (OCC):** Pendekatan ini berasumsi bahwa konflik jarang terjadi. Ia membiarkan transaksi berlanjut tanpa batasan dan hanya memeriksa konflik pada waktu *commit*. Ini memberikan *throughput* yang lebih tinggi dengan meminimalkan *lock contention*.[57] Contoh metode OCC meliputi *Timestamp-based concurrency control*, *Multi-Version Concurrency Control (MVCC)*, dan *Conflict-Free Replicated Data Types (CRDTs)*.[57]

Kontrol konkurensi di sistem terdistribusi memperkenalkan dimensi kompleksitas baru karena masalah latensi jaringan, kegagalan parsial, dan ketiadaan memori bersama global. Pergeseran antara pendekatan *pessimistic* dan *optimistic* mencerminkan *trade-off* antara konsistensi segera dan *throughput*/skalabilitas. Ini menunjukkan bahwa solusi sinkronisasi tidak statis; mereka harus beradaptasi dengan arsitektur sistem yang mendasarinya (misalnya, terpusat vs. terdistribusi) dan karakteristik beban kerja (misalnya, banyak konflik vs. sedikit konflik).

D. Praktik Terbaik Penggunaan Mekanisme Sinkronisasi

Mengelola konkurensi secara efektif memerlukan pemahaman yang mendalam tentang berbagai mekanisme sinkronisasi dan kapan harus menggunakannya. Berikut adalah beberapa praktik terbaik:

- **Hindari Berbagi Sumber Daya Jika Memungkinkan:** Seringkali, cara paling efektif untuk mencegah *race conditions* adalah dengan menghindari berbagi *state* sama sekali, melalui duplikasi atau partisi data. Pendekatan ini seringkali lebih cepat daripada perlindungan *shared-state* yang menggunakan *lock*.[1] Contohnya adalah penggunaan *Thread-Local Storage (TLS)*, yang mengalokasikan salinan unik variabel untuk setiap *thread*, secara efektif mengisolasi dari masalah akses konkuren.[1]
- **Pastikan Atomicity:** Penting untuk mencegah pembaruan bersamaan ke variabel

bersama oleh *thread* lain saat satu *thread* sedang secara aktif memodifikasinya. Ini memastikan bahwa pembaruan bersifat atomik, di mana *thread* akan mengamati seluruh nilai lama atau nilai baru, tanpa keadaan parsial atau perantara.[1]

- **Gunakan Abstraksi yang Tepat:**
  - **Mutexes:** Ideal untuk *mutual exclusion* sederhana pada satu sumber daya.[40]
  - **Semaphores:** Cocok untuk masalah sinkronisasi yang lebih kompleks seperti *Producer-Consumer* atau *Readers-Writers*, atau untuk mengelola sumber daya dengan banyak instansi.[40]
  - **Monitors:** Pilihan yang baik untuk sistem konkuren yang kompleks, karena menawarkan abstraksi dan enkapsulasi yang lebih tinggi, yang secara signifikan mengurangi kesalahan pemrograman.[40]
- **Desain Monitor dengan Hati-hati:** Desain *interface* monitor yang cermat sangat penting untuk mencegah panggilan monitor bersarang yang dapat menyebabkan *deadlock*. Penggunaan variabel kondisi yang tepat juga esensial untuk koordinasi *thread* yang benar.[40]
- **Strategi Sinyal yang Tepat:** Pertimbangkan strategi sinyal yang berbeda (misalnya, signal-and-continue vs. signal-and-wait) dan penggunaan operasi *broadcast* untuk membangunkan banyak *thread* yang menunggu secara bersamaan, tergantung pada kebutuhan aplikasi.[40]
- **Inisialisasi yang Benar:** Inisialisasi nilai semaphore yang tepat sangat krusial untuk perilaku sinkronisasi yang benar dan untuk menghindari masalah seperti *deadlock* atau *starvation*.[40]
- **Penempatan Operasi yang Strategis:** Penempatan operasi wait dan signal (atau P dan V) secara strategis sangat penting untuk mencegah *race conditions* dan *deadlock*.[40]

Praktik terbaik ini menunjukkan bahwa *concurrency control* adalah perpaduan antara seni dan ilmu. Tidak ada solusi tunggal yang "terbaik"; pilihan bergantung pada *trade-off* antara kinerja, kompleksitas, dan keandalan. Memahami kapan harus menggunakan primitif tingkat rendah (semaphore) vs. abstraksi tingkat tinggi (monitor), serta bagaimana merancang sistem untuk menghindari *race conditions* dan *deadlock* secara proaktif (misalnya, dengan menghindari berbagi *state*), adalah inti dari rekayasa sistem operasi yang efektif.

Tabel 5 memberikan ringkasan komprehensif dari berbagai mekanisme sinkronisasi yang dibahas, beserta kasus penggunaannya:

Tabel 5: Ringkasan Mekanisme Sinkronisasi dan Kasus Penggunaannya

Mekanisme	Tingkat Abstraksi	Tujuan Utama	Kelebihan Kunci	Kekurangan Kunci	Kasus Penggunaan Ideal
Disabling Interrupts	Sangat Rendah	Mutual Exclusion	Sederhana, efektif (uni-prosesor)	Tidak skalabel, *timer drift*	Sistem uni-prosesor sederhana, *kernel*
Test-and-Set	Rendah	Mutual Exclusion	Sederhana, *hardware-supported*	*Busy waiting*, *starvation*	Sistem multi-prosesor dengan *contention* rendah
Compare-and-Swap (CAS)	Rendah	Mutual Exclusion, Lock-Free	Fleksibel, *hardware-supported*	Masalah ABA, kompleksitas	Algoritma *lock-free*, struktur data konkuren
Algoritma Dekker	Software	Mutual Exclusion	Tidak butuh *hardware*, *bounded waiting*	Hanya 2 proses, *busy waiting*	Sinkronisasi 2 proses sederhana
Algoritma Peterson	Software	Mutual Exclusion	Tidak butuh *hardware*, *bounded waiting*	Hanya 2 proses, *busy waiting*	Sinkronisasi 2 proses sederhana
Lamport's Bakery	Software	Mutual Exclusion	Untuk N proses, *bounded waiting*	Kompleks, *busy waiting*	Sinkronisasi N proses (teoritis)
Binary Semaphore	Menengah	Mutual Exclusion	Sederhana, fleksibel	Rentan kesalahan, *priority inversion*	Melindungi *critical section* tunggal
Counting Semaphore	Menengah	Kontrol Akses Sumber	Mengelola banyak instansi	Rentan kesalahan, *priority*	Masalah *Producer-Consumer*,

		Daya	sumber daya	*inversion*	*Readers-Writers*
Monitor	Tinggi	Sinkronisasi Terstruktur	Abstraksi tinggi, kurang rentan kesalahan	Dukungan bahasa, *overhead*	Masalah *Producer-Consumer*, *Readers-Writers* kompleks
Lock-Free Algorithms	Sangat Tinggi	Paralelisme Maksimal	Tahan kegagalan, skalabel	Kompleks, *overhead* tinggi	Sistem multi-core berkinerja tinggi, *real-time*
Transactional Memory	Sangat Tinggi	Atomicity Transaksi	Menyederhanakan konkurensi, skalabel	*Overhead* konflik, dukungan *hardware* terbatas	Sistem multi-core, basis data terdistribusi

VII. Kesimpulan

Materi perkuliahan ini telah menguraikan secara komprehensif konsep fundamental *Mutual Exclusion*, *Semaphores*, dan *Monitors* sebagai pilar utama dalam sinkronisasi proses pada sistem operasi. Dimulai dengan pemahaman tentang konkurensi—baik melalui *interleaving* pada uni-prosesor maupun *overlapping* pada multi-prosesor—ditekankan bahwa kemampuan sistem untuk menjalankan banyak tugas secara bersamaan ini, meskipun meningkatkan efisiensi, secara inheren menimbulkan tantangan serius, terutama *race conditions*. *Race condition* adalah akar masalah yang menyebabkan inkonsistensi data dan ketidakandalan sistem, sehingga memicu kebutuhan akan mekanisme sinkronisasi.

Mutual exclusion menjadi fondasi utama untuk mengatasi *race conditions*, memastikan hanya satu proses yang mengakses *critical section* pada satu waktu. Evolusi implementasinya menunjukkan pergeseran dari solusi perangkat keras tingkat rendah seperti menonaktifkan interupsi dan instruksi Test-and-Set (yang efektif tetapi seringkali tidak efisien karena *busy-waiting* dan kurangnya *bounded waiting*) ke instruksi yang lebih canggih seperti Compare-and-Swap (CAS). CAS, dengan atomicity-nya, menjadi dasar bagi algoritma *non-blocking* yang lebih modern. Secara paralel, algoritma berbasis perangkat lunak seperti Dekker dan Peterson menunjukkan kemungkinan mencapai *mutual exclusion* tanpa dukungan perangkat keras khusus, meskipun terbatas pada dua proses dan masih menderita *busy-waiting*. Algoritma Lamport's Bakery memperluas ini ke N proses, namun dengan kompleksitas yang lebih tinggi.

Semaphore, yang diperkenalkan oleh Dijkstra, merepresentasikan langkah maju dalam abstraksi, menyediakan primitif yang lebih fleksibel untuk mengontrol akses ke sumber daya. Dengan operasi atomik P (Wait) dan V (Signal), semaphore dapat digunakan untuk *mutual exclusion* (binary semaphore) maupun manajemen sumber daya multi-instansi (*counting semaphore*). Penerapannya dalam masalah klasik seperti *Producer-Consumer* dan *Readers-Writers* menunjukkan kemampuannya yang luas. Namun, semaphore menuntut kehati-hatian programmer yang tinggi; kesalahan penempatan operasi P dan V dapat dengan mudah menyebabkan *deadlock* dan *starvation*, serta masalah *priority inversion*. Beban pada programmer inilah yang mendorong pengembangan abstraksi yang lebih tinggi.

Monitor muncul sebagai solusi untuk mengatasi keterbatasan semaphore, menyediakan konstruksi sinkronisasi tingkat tinggi yang mengintegrasikan data bersama, prosedur, dan variabel kondisi dalam satu unit yang dienkapsulasi. Dengan

mutual exclusion yang otomatis dan mekanisme wait()/signal() yang terstruktur, monitor secara signifikan mengurangi kompleksitas dan potensi kesalahan dalam pemrograman konkuren. Ini adalah pergeseran paradigma dari "bagaimana" sinkronisasi diimplementasikan ke "apa" yang perlu disinkronkan, memungkinkan pengembang untuk menulis kode yang lebih bersih dan aman, seperti yang terlihat pada synchronized methods di Java. Meskipun monitor mungkin memiliki *overhead* dan ketergantungan pada dukungan bahasa, keunggulannya dalam menyederhanakan tugas-tugas kompleks sangatlah signifikan.

Isu *deadlock* dan *starvation* tetap menjadi tantangan inheren dalam sistem konkuren. *Deadlock* terjadi ketika empat kondisi spesifik (mutual exclusion, hold and wait, no preemption, circular wait) terpenuhi, sementara *starvation* adalah masalah keadilan di mana proses tertentu terus-menerus ditolak akses ke sumber daya. Pemahaman mendalam tentang kondisi-kondisi ini dan strategi penanganannya (pencegahan, penghindaran, deteksi & pemulihan; serta teknik seperti *aging* dan *bounded waiting* untuk *starvation*) sangat penting untuk merancang sistem yang andal.

Perkembangan terbaru dalam kontrol konkurensi, seperti algoritma bebas kunci (*lock-free algorithms*) yang memanfaatkan instruksi atomik seperti CAS, dan *Transactional Memory* (STM/HTM) yang menawarkan model transaksi seperti basis data, menunjukkan upaya berkelanjutan untuk memaksimalkan paralelisme dan menyederhanakan pemrograman di lingkungan multi-core yang semakin kompleks. Selain itu, kontrol konkurensi dalam sistem terdistribusi, dengan pendekatan *optimistic* dan *pessimistic*, menyoroti adaptasi solusi sinkronisasi terhadap tantangan skala dan kegagalan parsial.

Secara keseluruhan, materi ini menegaskan bahwa pemahaman mendalam tentang *mutual exclusion*, *semaphores*, dan *monitors* adalah fundamental bagi mahasiswa teknik informatika. Konsep-konsep ini bukan hanya teori, melainkan fondasi teknis yang memungkinkan sistem operasi modern beroperasi dengan andal, efisien, dan aman di tengah kompleksitas konkurensi. Kemampuan untuk memilih dan menerapkan mekanisme sinkronisasi yang tepat, sambil memahami *trade-off* yang terlibat, adalah keterampilan krusial untuk membangun sistem yang tangguh dan efisien di masa depan.

Works cited

1. Concurrency in Operating Systems: Understanding Principles, Challenges, accessed May 20, 2025, https://vmhatre.com/concurrency-in-operating-systems
2. Reading 17: Concurrency - MIT, accessed May 20, 2025, https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/

3. Mutual exclusion - Wikipedia, accessed May 20, 2025, https://en.wikipedia.org/wiki/Mutual_exclusion
4. Race Condition Vulnerability | Causes, Impacts & Prevention - Imperva, accessed May 20, 2025, https://www.imperva.com/learn/application-security/race-condition/
5. Understanding Mutex in Android: Preventing Race Conditions - droidcon, accessed May 20, 2025, https://www.droidcon.com/2024/09/20/understanding-mutex-in-android-preventing-race-conditions/
6. What is a Race Condition? Vulnerability and Attack - Wallarm, accessed May 20, 2025, https://www.wallarm.com/what/what-is-a-race-condition
7. (PDF) Analysis of Synchronization Mechanisms in Operating Systems - ResearchGate, accessed May 20, 2025, https://www.researchgate.net/publication/384427937_Analysis_of_Synchronization_Mechanisms_in_Operating_Systems
8. Preliminary Analysis of Synchronization and Mutual Exclusion of Process in Operating System - VIBGYOR ePress, accessed May 20, 2025, https://vibgyorpublishers.org/content/ijspa/ijspa-2-003.php?jid=ijspa
9. Mutual Exclusion in Operating Systems - Tutorialspoint, accessed May 20, 2025, https://www.tutorialspoint.com/operating_system/os_mutual_exclusion_in_synchronization.htm
10. Monitors in Process Synchronization | GeeksforGeeks, accessed May 20, 2025, https://www.geeksforgeeks.org/monitors-in-process-synchronization/
11. Set Lock in Process Synchronization - Tutorialspoint, accessed May 20, 2025, https://www.tutorialspoint.com/operating_system/os_test_set_lock_in_process_synchronization.htm
12. Race Conditions, Critical Sections and Semaphores, accessed May 20, 2025, https://www.sjsu.edu/people/robert.chun/courses/cs159/s0/Day-8---Race-Semaphores.pdf
13. Peterson's algorithm - Wikipedia, accessed May 20, 2025, https://en.wikipedia.org/wiki/Peterson%27s_algorithm
14. Dekker's Algorithm - Glossary - DevX, accessed May 20, 2025, https://www.devx.com/terms/dekkers-algorithm/
15. Mutual Exclusion Flashcards - Quizlet, accessed May 20, 2025, https://quizlet.com/129260584/mutual-exclusion-flash-cards/
16. Mutual Exclusion, accessed May 20, 2025, https://cs.lmu.edu/~ray/notes/mutualexclusion/
17. Test and Set Lock - Khushali, accessed May 20, 2025, https://khushalip.github.io/OS-lab/deadlockAlgo/OS-Team%2048%20Semaphore,%20Deadlock%20and%20Concurrency/TSl/home.html
18. Semaphores in Process Synchronization - GeeksforGeeks, accessed May 20, 2025, https://www.geeksforgeeks.org/semaphores-in-process-synchronization/
19. en.wikipedia.org, accessed May 20, 2025, https://en.wikipedia.org/wiki/Compare-and-swap#:~:text=In%20computer%20science%2C%20compare%2Dand,to%20a%20new%20given%20value.

20. Exploring the Efficiency of Multi-Word Compare-and-Swap - Diva Portal, accessed May 20, 2025, https://uu.diva-portal.org/smash/get/diva2:1635674/FULLTEXT01.pdf
21. Compare-and-swap - Wikipedia, accessed May 20, 2025, https://en.wikipedia.org/wiki/Compare-and-swap
22. Operating System: mutual exclusion,deadlock problem, accessed May 20, 2025, https://gateoverflow.in/66383/mutual-exclusion-deadlock-problem?show=70080
23. Non-blocking algorithm - Wikipedia, accessed May 20, 2025, https://en.wikipedia.org/wiki/Non-blocking_algorithm
24. www.tutorialspoint.com, accessed May 20, 2025, https://www.tutorialspoint.com/dekker-s-algorithm-in-process-synchronization#:~:text=The%20algorithm%20achieves%20mutual%20exclusion,critical%20section%20at%20a%20time.
25. en.wikipedia.org, accessed May 20, 2025, https://en.wikipedia.org/wiki/Peterson%27s_algorithm#:~:text=Peterson's%20algorithm%20(or%20Peterson's%20solution,only%20shared%20memory%20for%20communication.
26. What is a monitor, and how does it differ from a semaphore? - Quora, accessed May 20, 2025, https://www.quora.com/What-is-a-monitor-and-how-does-it-differ-from-a-semaphore
27. Monitors Real-world Examples - LASS, accessed May 20, 2025, https://lass.cs.umass.edu/~shenoy/courses/fall08/lectures/Lec12.pdf
28. Semaphore (programming) - Wikipedia, accessed May 20, 2025, https://en.wikipedia.org/wiki/Semaphore_(programming)
29. Understanding Semaphore in Operating Systems: A Comprehensive Guide - Bito AI, accessed May 20, 2025, https://bito.ai/resources/semaphore-in-operating-systems/
30. Producer-Consumer Problem Using Semaphores - Operating ..., accessed May 20, 2025, http://personal.kent.edu/~rmuhamma/OpSystems/Myos/semaphore.htm
31. Semaphores in Os (Operating Systems) with example - BCA Labs, accessed May 20, 2025, https://bcalabs.org/subject/semaphores-in-operating-systems-with-example
32. Mutex vs.semaphore: What are the differences? - Shiksha Online, accessed May 20, 2025, https://www.shiksha.com/online-courses/articles/mutex-vs-semaphore-what-are-the-differences/
33. Mutex vs Semaphore | GeeksforGeeks, accessed May 20, 2025, https://www.geeksforgeeks.org/mutex-vs-semaphore/
34. Using Semaphores for Producer-Consumer Problems - DEV ..., accessed May 20, 2025, https://dev.to/kalkwst/using-semaphores-for-producer-consumer-problems-3d0p
35. Readers–writers problem - Wikipedia, accessed May 20, 2025,

https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem
36. 8.4. Readers-Writers Problem — Computer Systems Fundamentals, accessed May 20, 2025, https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/ReadWrite.html
37. Monitors, accessed May 20, 2025, https://www.philadelphia.edu.jo/academics/mbaniyounes/uploads/Monitors.ppt
38. What is the difference about Mutual Exclusion, between Monitor and Semaphore, accessed May 20, 2025, https://stackoverflow.com/questions/23612212/what-is-the-difference-about-mutual-exclusion-between-monitor-and-semaphore
39. Starvation (computer science) - Wikipedia, accessed May 20, 2025, https://en.wikipedia.org/wiki/Starvation_(computer_science)
40. Semaphores, mutexes, and monitors | Operating Systems Class Notes - Fiveable, accessed May 20, 2025, https://library.fiveable.me/operating-systems/unit-6/semaphores-mutexes-monitors/study-guide/S4jiSv0NbbMFRCtj
41. CS 537 Notes, Section #8: Monitors, accessed May 20, 2025, https://pages.cs.wisc.edu/~bart/537/lecturenotes/s8.html
42. Concurrent programming with monitors - Project Nayuki, accessed May 20, 2025, https://www.nayuki.io/page/concurrent-programming-with-monitors
43. Is it better to synchronize with semaphores or with monitors? - Stack Overflow, accessed May 20, 2025, https://stackoverflow.com/questions/5083818/is-it-better-to-synchronize-with-semaphores-or-with-monitors
44. Deadlock (computer science) - Wikipedia, accessed May 20, 2025, https://en.wikipedia.org/wiki/Deadlock_(computer_science)
45. 6.1: Concept and Principles of Deadlock - Engineering LibreTexts, accessed May 20, 2025, https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/06%3A_Deadlock/6.1%3A_Concept_and_Principles_of_Deadlock
46. Introduction of Deadlock in Operating System, accessed May 20, 2025, https://gwcet.ac.in/uploaded_files/Unit_3_OperatingSys_compressed.pdf
47. Deadlock Prevention And Avoidance | GeeksforGeeks, accessed May 20, 2025, https://www.geeksforgeeks.org/deadlock-prevention/
48. What Is Starvation In OS? Definition, Causes And Solution - Unstop, accessed May 20, 2025, https://unstop.com/blog/starvation-in-os
49. Deadlock in Operating System : Algorithms, Advanatges & Disadvantages - ElProCus, accessed May 20, 2025, https://www.elprocus.com/deadlock-in-operating-system/
50. About semaphore deadlock and idle task hook - Kernel - FreeRTOS Community Forums, accessed May 20, 2025, https://forums.freertos.org/t/about-semaphore-deadlock-and-idle-task-hook/22087
51. (PDF) Lock-Free Parallel Algorithms: An Experimental Study - ResearchGate, accessed May 20, 2025,

https://www.researchgate.net/publication/220727876_Lock-Free_Parallel_Algorithms_An_Experimental_Study
52. Transactional Memory: A Comprehensive Review of Implementation, Applications, Performance, Challenges, Framework Comparisons, and Future Prospects, accessed May 20, 2025, https://www.ijisae.org/index.php/IJISAE/article/view/7317
53. Software transactional memory - Wikipedia, accessed May 20, 2025, https://en.wikipedia.org/wiki/Software_transactional_memory
54. Understanding Transactional Memory: Evolution and Modern Applications, accessed May 20, 2025, https://hub.paper-checker.com/blog/transactional-memory-evolution-history-and-its-role-in-modern-computing/
55. (PDF) Hardware Transactional Memories - Advances in Systems - Amanote Research, accessed May 20, 2025, https://research.amanote.com/publication/hKhAAnQBKQvf0BhipsgS/hardware-transactional-memories
56. Concurrency Control in DBMS - GeeksforGeeks, accessed May 20, 2025, https://www.geeksforgeeks.org/concurrency-control-in-dbms/
57. Mastering Concurrency Control in Distributed Databases - TiDB, accessed May 20, 2025, https://www.pingcap.com/article/mastering-concurrency-control-in-distributed-databases/
58. Concurrency Control in Distributed Transactions | GeeksforGeeks, accessed May 20, 2025, https://www.geeksforgeeks.org/concurrency-control-in-distributed-transactions/
59. Concurrency Control Techniques - GeeksforGeeks, accessed May 20, 2025, https://www.geeksforgeeks.org/concurrency-control-techniques/