Sinkronisasi Proses dan Penanganan Race Condition

MATA KULIAH : SISTEM OPERASI

14 MEI 2025

JURUSAN TEKNIK INFORMATIKA

STMIK TAZKIA BOGOR

2025

DAFTAR ISI

I. Pendahuluan

Konteks Proses Konkuren

Sistem operasi modern dirancang untuk melakukan banyak hal secara bersamaan. Kemampuan ini dikenal sebagai konkurensi (*concurrency*). Sistem operasi dapat menjalankan banyak program atau bagian dari program, yang disebut proses atau *thread*, seolah-olah pada waktu yang sama.[1] Tujuannya adalah untuk meningkatkan penggunaan sumber daya komputasi, seperti CPU, dan membuat sistem terasa lebih responsif terhadap pengguna. Proses-proses yang berjalan secara konkuren ini bisa saja sepenuhnya independen satu sama lain, namun seringkali mereka perlu berinteraksi atau bekerja sama untuk menyelesaikan tugas yang lebih besar.[1] Fokus kita pada pertemuan ini adalah pada proses-proses yang saling berinteraksi, khususnya yang memerlukan akses ke sumber daya yang sama.

Berbagi Sumber Daya

Ketika proses atau *thread* berinteraksi, mereka seringkali perlu menggunakan atau memodifikasi sumber daya yang sama. Sumber daya ini dikenal sebagai sumber daya bersama (*shared resources*). Contoh sumber daya bersama meliputi:

- Variabel atau struktur data di memori utama yang dapat diakses oleh beberapa *thread* dalam satu proses.
- File pada sistem penyimpanan yang perlu dibaca atau ditulis oleh beberapa proses berbeda.
- Perangkat keras seperti printer atau *port* komunikasi.

Sebagai contoh sederhana, bayangkan sebuah aplikasi perbankan di mana beberapa *thread* mungkin mencoba mengakses dan mengubah saldo dari rekening yang sama secara bersamaan, atau beberapa proses mencoba menulis log aktivitas ke dalam satu file log yang sama.[2]

Masalah Akses Bersamaan (Concurrent Access Problem)

Masalah muncul ketika beberapa proses atau *thread* mencoba mengakses dan memanipulasi (*read/write*) data bersama pada saat yang bersamaan tanpa adanya mekanisme pengaturan atau koordinasi.[2] Jika akses ini tidak dikendalikan, urutan operasi antar proses/thread menjadi tidak pasti, dan hasil akhir dari data bersama bisa menjadi salah, tidak konsisten, atau tidak terduga.[7] Ini adalah inti dari masalah

akses bersamaan yang akan kita bahas lebih lanjut. Pengelolaan akses ke sumber daya bersama menjadi sangat penting untuk menjaga integritas data dan memastikan program berjalan sesuai dengan yang diharapkan.

II. Tujuan Pembelajaran

Setelah mempelajari materi dalam bab ini, mahasiswa diharapkan mampu:

- Memahami konsep proses konkuren dan mengapa sinkronisasi diperlukan dalam lingkungan seperti itu.[1]
- Menjelaskan apa yang dimaksud dengan *race condition* dan bagaimana hal itu dapat menyebabkan inkonsistensi data serta dampak negatif lainnya.[1]
- Mengidentifikasi bagian kode program yang merupakan *critical section*, yaitu area di mana sumber daya bersama diakses.[1]
- Memahami tiga syarat fundamental yang harus dipenuhi oleh solusi yang baik untuk masalah *critical section*, yaitu: *Mutual Exclusion*, *Progress*, dan *Bounded Waiting*.[9]
- Menjelaskan prinsip kerja dan operasi dasar dari dua mekanisme sinkronisasi fundamental: *Mutex* dan *Semaphore*.[1]
- Menganalisis contoh kode C sederhana yang mendemonstrasikan terjadinya *race condition* ketika menggunakan Pthreads (POSIX Threads) di sistem operasi Linux.[12]
- Mengimplementasikan solusi sederhana untuk mengatasi *race condition* menggunakan *Mutex* atau *Semaphore* dalam program C/Pthreads yang dapat dijalankan di Linux.[12]

III. Konsep Dasar: Race Condition dan Critical Section

Definisi Race Condition

Race condition adalah suatu kondisi yang terjadi ketika dua atau lebih proses atau *thread* mengakses dan memanipulasi data bersama (*shared data*) secara bersamaan, dan hasil akhir dari operasi pada data tersebut sangat bergantung pada urutan waktu atau *timing* eksekusi dari proses/thread tersebut – khususnya, proses mana yang terakhir kali melakukan penulisan.[3] Istilah "race" (balapan) digunakan karena proses-proses tersebut seolah-olah sedang "berlomba" untuk mengakses atau mengubah data.[7] Race condition umumnya terjadi di dalam bagian kode yang disebut *critical section* jika akses ke bagian tersebut tidak diatur atau dilindungi dengan benar.[7]

Ilustrasi Sederhana

Untuk memahami konsep *race condition*, kita bisa menggunakan analogi sederhana. Bayangkan dua orang (sebagai *thread*) mencoba mengambil uang dari rekening bank yang sama melalui dua ATM yang berbeda pada waktu yang hampir bersamaan.[7] Atau, bayangkan dua orang mencoba menulis catatan pada halaman yang sama di buku catatan yang sama tanpa saling berkoordinasi terlebih dahulu.[7] Hasilnya bisa jadi kacau atau tidak sesuai harapan.

Secara teknis, mari kita lihat contoh umum menggunakan variabel penghitung (counter) yang dibagi antara dua *thread*, sebut saja Thread A dan Thread B. Keduanya bertugas menambahkan nilai 1 ke counter. Misalkan nilai awal counter adalah 5. Operasi counter++ (increment) sebenarnya tidak terjadi dalam satu langkah tunggal (bukan operasi atomik), melainkan melibatkan beberapa langkah di tingkat mesin:

1. Baca nilai counter saat ini dari memori ke register CPU.
2. Tambahkan 1 pada nilai di register.
3. Tulis kembali nilai dari register ke memori (counter).

Berikut skenario *race condition* yang mungkin terjadi:

1. **Thread A**: Baca counter (nilai 5) ke registernya.
2. *(Context Switch)* Sistem operasi menghentikan Thread A sejenak dan menjalankan Thread B.
3. **Thread B**: Baca counter (nilai masih 5) ke registernya.
4. **Thread B**: Tambahkan 1 di registernya (5 + 1 = 6).
5. **Thread B**: Tulis nilai 6 kembali ke counter di memori.
6. *(Context Switch)* Sistem operasi menghentikan Thread B dan melanjutkan Thread

A.
7. **Thread A**: Tambahkan 1 di registernya (yang masih berisi nilai 5, menjadi 6).
8. **Thread A**: Tulis nilai 6 kembali ke counter di memori.

Hasil akhir counter adalah 6. Padahal, karena ada dua operasi increment yang seharusnya terjadi, nilai yang benar seharusnya adalah 7. Satu operasi increment telah "hilang" (*lost update*) akibat urutan eksekusi yang tidak terkontrol ini.[12]

Dampak Race Condition

Terjadinya *race condition* dapat menimbulkan berbagai masalah serius, antara lain:

1. **Inkonsistensi Data:** Seperti contoh di atas, nilai akhir dari data bersama menjadi salah atau tidak mencerminkan operasi yang seharusnya terjadi.[2] Ini adalah dampak paling langsung dan umum.
2. **Perilaku Program Tidak Terduga:** Program dapat menghasilkan output yang berbeda-beda setiap kali dijalankan, tergantung pada faktor *timing* yang sulit diprediksi, seperti beban sistem atau penjadwalan *thread*.[8] Hal ini membuat *debugging* menjadi sangat sulit.
3. **Crash Program:** Dalam beberapa kasus, terutama jika *race condition* terjadi pada struktur data internal yang kompleks atau pointer, hal ini dapat menyebabkan kesalahan fatal seperti *segmentation fault* dan membuat program berhenti bekerja secara tiba-tiba (*crash*).[8]
4. **Kerentanan Keamanan (Security Vulnerability):** Ini adalah dampak yang seringkali diabaikan namun sangat berbahaya. *Race condition* dapat dieksploitasi oleh pihak jahat. Misalnya, penyerang dapat mencoba mengatur *timing* akses untuk melewati pemeriksaan keamanan (kondisi *Time-of-Check to Time-of-Use* / TOCTOU), mendapatkan hak akses yang lebih tinggi (*privilege escalation*), merusak data penting, menyebabkan penolakan layanan (*Denial of Service* / DoS) dengan membuat sistem macet atau menghabiskan sumber daya, atau bahkan membocorkan informasi sensitif.[7] Contoh nyata eksploitasi adalah penggunaan satu kode promo diskon berkali-kali sebelum sistem sempat menandainya sebagai sudah terpakai.[18] Oleh karena itu, penanganan *race condition* bukan hanya soal kebenaran fungsional program, tetapi juga merupakan aspek krusial dalam keamanan sistem.

Definisi Critical Section

Critical section (bagian kritis) adalah segmen atau bagian dari kode program di mana suatu proses atau *thread* perlu mengakses atau memanipulasi sumber daya bersama (*shared resource*).[1] Sumber daya bersama ini bisa berupa variabel global, buffer data, file, tabel dalam database, atau perangkat keras tertentu. Karena akses bersamaan ke

critical section inilah yang dapat menyebabkan *race condition*, maka bagian kode ini memerlukan perlindungan khusus. Tujuannya adalah untuk memastikan bahwa pada satu waktu, hanya ada satu proses/thread yang diizinkan untuk mengeksekusi *critical section* tersebut.[5]

Masalah Critical Section (The Critical Section Problem)

Inti dari masalah *critical section* adalah bagaimana merancang sebuah protokol atau mekanisme yang dapat digunakan oleh proses-proses yang bekerja sama (kooperatif) untuk memastikan akses yang aman ke *critical section* mereka.[1] Protokol ini harus menjamin bahwa jika satu proses sedang berada di dalam *critical section*-nya, tidak ada proses lain yang dapat masuk ke *critical section* yang sama (yang mengakses sumber daya bersama yang sama) pada saat itu.

Secara umum, struktur program sebuah proses yang melibatkan *critical section* dapat dibagi menjadi beberapa bagian:

- **Entry Section:** Kode yang digunakan oleh proses untuk meminta izin masuk ke *critical section*. Di sinilah mekanisme sinkronisasi (seperti meminta kunci) diterapkan.
- **Critical Section:** Bagian kode di mana akses ke sumber daya bersama terjadi. Hanya satu proses yang boleh berada di sini pada satu waktu.
- **Exit Section:** Kode yang dieksekusi setelah proses keluar dari *critical section*. Di sini, proses biasanya melepaskan kunci atau memberi sinyal kepada proses lain.
- **Remainder Section:** Bagian kode lainnya dari proses yang tidak melibatkan akses ke sumber daya bersama tersebut.

```
do {
   // Entry Section: Meminta izin masuk
   // <kode untuk mendapatkan akses eksklusif>

      // Critical Section: Mengakses sumber daya bersama
      // <kode yang memanipulasi data bersama>

   // Exit Section: Melepaskan izin
   // <kode untuk melepaskan akses eksklusif>
```

```
    // Remainder Section: Kode lainnya
    // <kode yang tidak mengakses data bersama>

} while (TRUE);
```

Tantangannya adalah merancang kode untuk *Entry Section* dan *Exit Section* yang memenuhi syarat-syarat tertentu agar akses ke *Critical Section* berjalan dengan benar dan efisien.

IV. Kebutuhan Akan Sinkronisasi

Mengapa Sinkronisasi Diperlukan?

Sinkronisasi proses atau *thread* adalah mekanisme pengaturan jalannya beberapa proses/thread pada saat yang bersamaan.[19] Kebutuhan utama akan sinkronisasi muncul dari adanya proses-proses konkuren yang perlu mengakses atau memodifikasi data bersama.[2] Tanpa adanya sinkronisasi, akses yang tidak terkoordinasi ini akan mengarah pada *race condition*, yang dapat menyebabkan data menjadi tidak konsisten atau rusak.[2] Hasil akhir dari operasi pada data bersama menjadi tidak dapat diprediksi karena bergantung pada urutan eksekusi yang kebetulan terjadi.[3] Oleh karena itu, sinkronisasi mutlak diperlukan untuk menjaga integritas data dan memastikan kebenaran hasil dalam sistem konkuren.

Tujuan Sinkronisasi

Tujuan utama dari sinkronisasi adalah untuk mengelola interaksi antar proses/thread yang konkuren. Secara lebih spesifik, tujuan sinkronisasi meliputi:

1. **Menjaga Konsistensi Data:** Ini adalah tujuan paling fundamental. Sinkronisasi memastikan bahwa meskipun data diakses oleh banyak entitas secara bersamaan, data tersebut tetap berada dalam keadaan yang valid, akurat, dan konsisten sesuai dengan aturan yang diharapkan.[2]
2. **Mengatur Urutan Eksekusi:** Terkadang, urutan eksekusi antar proses/thread sangat penting. Misalnya, sebuah proses 'konsumen' mungkin harus menunggu sampai proses 'produsen' selesai menghasilkan data sebelum dapat mengolahnya. Sinkronisasi menyediakan mekanisme untuk memastikan urutan ketergantungan ini terpenuhi.[1]
3. **Memfasilitasi Kolaborasi:** Sinkronisasi tidak hanya berfungsi sebagai pembatas untuk mencegah masalah, tetapi juga sebagai fondasi yang memungkinkan proses/thread untuk bekerja sama secara efektif. Dengan adanya mekanisme sinkronisasi yang aman, proses/thread dapat berbagi informasi dan sumber daya untuk menyelesaikan tugas-tugas kompleks secara paralel atau terdistribusi.[19] Ini memungkinkan sistem konkuren untuk benar-benar memanfaatkan kekuatannya dalam kolaborasi, bukan hanya berjalan secara terpisah.

Syarat Solusi Masalah Critical Section

Untuk memastikan bahwa akses ke *critical section* dikelola dengan baik, setiap solusi atau mekanisme sinkronisasi yang diusulkan harus memenuhi tiga kriteria penting berikut [9]:

1. **Mutual Exclusion (Saling Meniadakan):** Ini adalah syarat paling dasar. Jika

suatu proses sedang mengeksekusi *critical section*-nya, maka tidak boleh ada proses lain yang dapat mengeksekusi *critical section* mereka (yang terkait dengan sumber daya bersama yang sama) pada saat yang bersamaan.[5] Hanya satu proses yang diizinkan 'masuk' pada satu waktu.

2. **Progress (Kemajuan):** Jika tidak ada proses yang sedang berada di dalam *critical section*, dan ada satu atau lebih proses yang ingin masuk ke *critical section* mereka, maka pemilihan proses mana yang akan masuk berikutnya tidak boleh ditunda tanpa batas waktu. Selain itu, keputusan pemilihan ini hanya boleh melibatkan proses-proses yang sedang menunggu untuk masuk (berada di *entry section*) dan tidak sedang menjalankan *remainder section* mereka.[9] Syarat ini memastikan bahwa sumber daya tidak dibiarkan menganggur jika ada yang membutuhkannya dan sistem tidak mengalami kebuntuan (*deadlock*) hanya karena proses-proses ragu-ragu untuk masuk. Sebagai contoh, solusi yang memaksakan giliran secara ketat antara dua proses bisa melanggar *progress* jika satu proses yang jarang membutuhkan akses tetapi gilirannya tiba, sementara proses lain yang sering membutuhkan akses harus menunggu giliran meskipun *critical section* sedang kosong.[9]
3. **Bounded Waiting (Tunggu Berbatas):** Harus ada batasan (*bound*) pada jumlah berapa kali proses-proses lain diizinkan untuk masuk ke *critical section* mereka setelah suatu proses membuat permintaan untuk masuk ke *critical section*-nya dan sebelum permintaan tersebut akhirnya dikabulkan.[9] Dengan kata lain, sebuah proses yang ingin masuk tidak boleh menunggu selamanya. Syarat ini mencegah terjadinya *starvation* (kelaparan), di mana sebuah proses secara efektif tidak pernah mendapatkan kesempatan untuk masuk ke *critical section* meskipun ia terus mencoba.

Penting untuk dipahami bahwa *Progress* dan *Bounded Waiting* adalah dua konsep yang berbeda dan saling melengkapi. *Progress* memastikan bahwa *keputusan* untuk memilih siapa yang masuk berikutnya akan dibuat dalam waktu yang terbatas jika ada kandidat yang layak. *Bounded Waiting* memastikan bahwa *setiap* proses yang menunggu *akhirnya* akan terpilih. Keduanya diperlukan untuk menciptakan solusi *critical section* yang tidak hanya benar (menjamin *mutual exclusion*) tetapi juga efisien (tidak ada penundaan yang tidak perlu) dan adil (tidak ada *starvation*).

Contoh perbedaannya, dimana Progress terpenuhi tapi Bounded Waiting tidak terpenuhi:

- Ada 3 proses: A, B, C
- Proses A dan B terus bergantian masuk critical section
- Proses C tidak pernah kebagian giliran (starvation)
- Progress terpenuhi karena selalu ada yang masuk
- Bounded Waiting tidak terpenuhi karena C menunggu tanpa batas

Intinya: Progress memastikan sistem tetap jalan, Bounded Waiting memastikan semua proses diperlakukan adil.

V. Mekanisme Sinkronisasi Dasar

Pengantar

Ada berbagai teknik dan mekanisme perangkat keras maupun perangkat lunak yang dapat digunakan untuk mengatasi masalah *critical section* dan mencapai sinkronisasi. Beberapa pendekatan melibatkan instruksi perangkat keras khusus yang bersifat atomik (tidak dapat diinterupsi) seperti TestAndSet atau CompareAndSwap.[1] Namun, solusi berbasis perangkat lunak yang lebih umum digunakan dan menjadi fondasi bagi banyak sistem operasi adalah *Mutex* dan *Semaphore*.[1] Kita akan membahas keduanya.

Mutex (Mutual Exclusion) Locks

Mutex adalah singkatan dari *Mutual Exclusion*. Ini adalah salah satu mekanisme sinkronisasi paling dasar dan sering digunakan.

- **Konsep:** Mutex bekerja seperti sebuah kunci (*lock*) untuk sebuah sumber daya bersama. Hanya *thread* yang berhasil mendapatkan (memegang) kunci tersebut yang diizinkan untuk mengakses sumber daya (masuk ke *critical section*). Jika *thread* lain mencoba mendapatkan kunci yang sama saat sedang dipegang, *thread* tersebut harus menunggu sampai kunci dilepaskan oleh pemegangnya.[3] Analogi sederhananya adalah kunci pintu toilet umum: hanya satu orang yang bisa masuk pada satu waktu dengan mengunci pintu dari dalam.
- **Operasi:** Dua operasi utama pada mutex adalah:
    - **Lock (Kunci) / Acquire:** Operasi ini digunakan oleh *thread* untuk mencoba mendapatkan kepemilikan mutex.
        - Jika mutex sedang dalam keadaan *unlocked* (tidak ada yang memegang), *thread* tersebut berhasil mendapatkan mutex, mengubah statusnya menjadi *locked*, dan melanjutkan eksekusi ke dalam *critical section*.
        - Jika mutex sedang dalam keadaan *locked* (dipegang oleh *thread* lain), *thread* yang mencoba melakukan *lock* akan diblokir (masuk ke state *waiting* atau *sleep*) sampai mutex tersebut dilepaskan (*unlocked*).[12]
    - **Unlock (Buka Kunci) / Release:** Operasi ini digunakan oleh *thread* yang sedang memegang mutex untuk melepaskan kepemilikannya. Setelah mutex dilepaskan, statusnya kembali menjadi *unlocked*. Jika ada *thread* lain yang sedang menunggu (diblokir) untuk mendapatkan mutex ini, salah satu dari mereka akan 'dibangunkan' dan diberi kesempatan untuk mendapatkan mutex.[12]
- **Kepemilikan (Ownership):** Fitur penting dari mutex adalah konsep kepemilikan yang ketat. Artinya, hanya *thread* yang berhasil melakukan operasi *lock* pada suatu mutex yang diizinkan untuk melakukan operasi *unlock* pada mutex yang

sama.[12] Jika *thread* lain mencoba membuka kunci yang tidak dimilikinya, atau jika *thread* mencoba membuka kunci yang sudah dalam keadaan *unlocked*, biasanya akan terjadi kesalahan (*error*) atau perilaku yang tidak terdefinisi (*undefined behavior*). Aturan kepemilikan ini menyederhanakan pengelolaan akses karena jelas siapa yang bertanggung jawab atas sumber daya pada suatu waktu, namun juga membatasi penggunaannya pada skenario di mana entitas yang mengunci dan membuka kunci adalah sama.

- **Penggunaan Umum:** Mutex sangat cocok digunakan untuk melindungi *critical section* di mana aturan *mutual exclusion* yang ketat diperlukan, yaitu memastikan hanya satu *thread* yang dapat mengakses data atau kode tertentu pada satu waktu untuk mencegah *race condition*.[22]

Semaphores

Semaphore adalah mekanisme sinkronisasi yang lebih umum dan fleksibel dibandingkan mutex, diperkenalkan oleh Edsger Dijkstra pada tahun 1960-an.

- **Konsep:** Semaphore pada dasarnya adalah sebuah variabel integer non-negatif yang nilainya hanya dapat dimanipulasi melalui dua operasi khusus yang bersifat atomik (tidak dapat diinterupsi): wait dan signal.[1] Semaphore dapat diibaratkan sebagai penjaga gerbang yang memiliki sejumlah token (izin masuk). Untuk bisa melewati gerbang (*critical section* atau menggunakan sumber daya), sebuah *thread* harus mendapatkan satu token dari penjaga. Jika tidak ada token tersisa, *thread* harus menunggu.
- **Operasi Atomik:**
  - **wait(S)** (juga dikenal sebagai P(S) dari bahasa Belanda *Proberen* yang berarti 'mencoba', atau down(), atau sem_wait() di POSIX): Operasi ini mencoba untuk mengurangi nilai semaphore S.
    - Jika S > 0, maka nilai S dikurangi 1 (seperti mengambil satu token), dan *thread* melanjutkan eksekusinya.
    - Jika S == 0, maka *thread* tidak dapat melanjutkan. Ia akan diblokir (dimasukkan ke dalam antrian tunggu untuk semaphore S) sampai nilai S menjadi lebih besar dari 0 (yaitu, sampai ada *thread* lain yang melakukan operasi signal).[11]
  - **signal(S)** (juga dikenal sebagai V(S) dari bahasa Belanda *Verhogen* yang berarti 'meningkatkan', atau up(), atau sem_post() di POSIX): Operasi ini meningkatkan nilai semaphore S sebanyak 1 (seperti mengembalikan satu token).
    - Setelah nilai S ditingkatkan, jika ada *thread* lain yang sedang menunggu (diblokir) pada semaphore S tersebut, maka salah satu dari *thread* yang

menunggu itu akan 'dibangunkan' dan diizinkan untuk mencoba menyelesaikan operasi wait-nya.[11]

- **Jenis Semaphore:**
  - **Binary Semaphore:** Nilai semaphore ini hanya dapat berupa 0 atau 1. Binary semaphore seringkali digunakan untuk mengimplementasikan *mutual exclusion*, mirip dengan cara kerja mutex. Jika nilai awalnya 1, hanya satu *thread* yang bisa melakukan wait (mengubah nilai jadi 0) sebelum harus menunggu signal (yang mengembalikan nilai ke 1).[11]
  - **Counting Semaphore:** Nilai semaphore ini dapat berupa integer non-negatif berapapun. Counting semaphore digunakan untuk mengontrol akses ke sekumpulan sumber daya yang jumlahnya terbatas (misalnya, ada N buah printer). Nilai awal semaphore diatur ke N. Setiap kali sebuah *thread* ingin menggunakan satu sumber daya, ia melakukan wait. Jika berhasil (nilai > 0), nilai semaphore berkurang. Jika tidak (nilai == 0), *thread* menunggu. Setelah selesai menggunakan sumber daya, *thread* melakukan signal, yang menambah nilai semaphore, memungkinkan *thread* lain (jika ada yang menunggu) untuk menggunakan sumber daya.[11]
- **Penggunaan Umum:** Semaphore sangat fleksibel dan dapat digunakan untuk:
  - **Mutual Exclusion:** Menggunakan binary semaphore dengan nilai awal 1.[11]
  - **Mengontrol Akses ke Resource Pool:** Menggunakan counting semaphore dengan nilai awal N, di mana N adalah jumlah resource yang tersedia.[11]
  - **Sinkronisasi Urutan Eksekusi:** Memastikan satu *thread* (misal, B) hanya berjalan setelah *thread* lain (misal, A) mencapai titik tertentu. Caranya: inisialisasi semaphore sync_sem dengan nilai 0. Thread B melakukan wait(sync_sem) di awal. Thread A melakukan signal(sync_sem) setelah menyelesaikan tugasnya. Ini memastikan B tidak akan lanjut sebelum A memberi sinyal.[1]

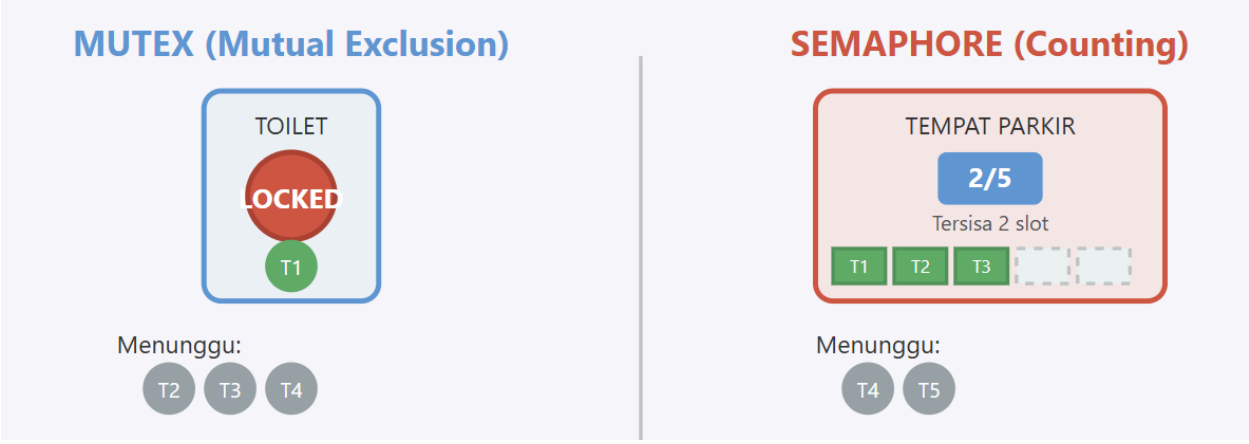
Tabel Perbandingan Mutex vs Semaphore

Untuk memperjelas perbedaan antara Mutex dan Semaphore, berikut adalah tabel perbandingannya:

Fitur	**Mutex (Mutual Exclusion Lock)**	**Semaphore**
Tujuan Utama	Menjamin akses eksklusif (hanya satu thread) ke critical section	Mekanisme sinkronisasi umum: mutual exclusion, resource counting, signaling

Nilai	Biasanya dianggap sebagai terkunci (*locked*) atau tidak (*unlocked*)	Integer non-negatif (0, 1, 2,...)
Operasi Utama	lock() / acquire(), unlock() / release()	wait() / P() / down(), signal() / V() / up()
Kepemilikan	Ya, hanya thread yang mengunci yang bisa membuka kunci	Tidak ada konsep kepemilikan; thread manapun bisa melakukan signal
Contoh Penggunaan	Melindungi critical section sederhana	Melindungi critical section (binary), mengelola resource pool (counting), sinkronisasi urutan
Sumber Data	12	11

Tabel ini menyoroti perbedaan fundamental, terutama terkait nilai dan kepemilikan. Mutex lebih spesifik untuk mutual exclusion dengan aturan kepemilikan yang ketat, sementara semaphore lebih general dengan mekanisme berbasis hitungan (counter) dan tidak memiliki aturan kepemilikan seketat mutex.



VI. Studi Kasus: Penanganan Race Condition di Linux (C/Pthreads)

Pengantar Studi Kasus

Untuk memberikan pemahaman praktis tentang bagaimana *race condition* terjadi dan bagaimana mengatasinya, kita akan melihat contoh kode sederhana menggunakan bahasa C dan Pthreads (POSIX Threads) di lingkungan Linux. Pthreads adalah standar API (Application Programming Interface) untuk membuat dan mengelola *thread* yang umum digunakan di sistem operasi mirip Unix, termasuk Linux.

Skenario yang akan kita gunakan adalah yang telah dibahas sebelumnya: dua *thread* secara bersamaan mencoba untuk menambah (increment) nilai sebuah variabel global yang sama. Kita akan lihat kode yang mengalami *race condition*, lalu memodifikasinya menggunakan Mutex dan Semaphore untuk mendapatkan hasil yang benar.

A. Contoh Kode Race Condition (Tanpa Sinkronisasi)

Berikut adalah kode C yang mendemonstrasikan *race condition*:

```c
#include <pthread.h> // Untuk fungsi-fungsi Pthreads
#include <stdio.h>   // Untuk printf
#include <stdlib.h>  // Untuk exit

// Variabel global yang akan diakses bersama oleh thread
long long counter = 0;
#define NUM_ITERATIONS 1000000 // Jumlah iterasi per thread

// Fungsi yang akan dijalankan oleh setiap thread
void* worker_thread(void* arg) {
    for (long long i = 0; i < NUM_ITERATIONS; i++) {
        // CRITICAL SECTION (tidak dilindungi)
        counter++; // Operasi increment yang rentan race condition
    }
    return NULL;
}

int main() {
    pthread_t thread1_id, thread2_id; // Variabel untuk menyimpan ID thread

    printf("Memulai program...\n");
    printf("Nilai counter awal: %lld\n", counter);
    printf("Setiap thread akan melakukan %lld iterasi increment.\n", NUM_ITERATIONS);
```

```
    printf("Nilai counter yang diharapkan: %lld\n", 2 * NUM_ITERATIONS);

    // Membuat thread pertama
    if (pthread_create(&thread1_id, NULL, worker_thread, NULL)!= 0) {
        perror("Gagal membuat thread 1");
        return 1;
    }

    // Membuat thread kedua
    if (pthread_create(&thread2_id, NULL, worker_thread, NULL)!= 0) {
        perror("Gagal membuat thread 2");
        return 1;
    }

    // Menunggu thread pertama selesai
    if (pthread_join(thread1_id, NULL)!= 0) {
        perror("Gagal menunggu thread 1");
        return 1;
    }

    // Menunggu thread kedua selesai
    if (pthread_join(thread2_id, NULL)!= 0) {
        perror("Gagal menunggu thread 2");
        return 1;
    }

    printf("Kedua thread telah selesai.\n");
    printf("Nilai counter akhir (aktual): %lld\n", counter);

    // Memeriksa apakah terjadi race condition
    if (counter == 2 * NUM_ITERATIONS) {
            printf("Hasil sesuai harapan. Tidak terdeteksi race condition (mungkin perlu iterasi lebih
banyak).\n");
    } else {
        printf("HASIL TIDAK SESUAI HARAPAN! Race condition kemungkinan besar terjadi.\n");
    }

    return 0;
}
```

- **Penjelasan Kode:**
  - Kode ini mendefinisikan variabel global counter yang diinisialisasi ke 0.
  - Fungsi worker_thread berisi loop yang mengincrement counter sebanyak NUM_ITERATIONS kali. Operasi counter++ adalah *critical section* yang tidak dilindungi.
  - Fungsi main membuat dua *thread* (thread1_id, thread2_id) yang keduanya menjalankan fungsi worker_thread.[16]
  - pthread_join digunakan untuk memastikan program utama menunggu kedua *thread* selesai sebelum mencetak nilai akhir counter.[16]
- **Kompilasi & Eksekusi di Linux:**
  1. Simpan kode di atas sebagai file, misalnya race_condition.c.
  2. Buka terminal Linux.
  3. Kompilasi menggunakan GCC dengan menyertakan library Pthread:
     Bash
     gcc race_condition.c -o race_condition -pthread
     Flag -pthread sangat penting untuk memberitahu compiler dan linker agar menyertakan dukungan untuk Pthreads.[12]
  4. Jalankan program hasil kompilasi:

./race_condition```
- Analisis Output:
  Anda kemungkinan besar akan melihat output seperti ini (nilai aktual bisa bervariasi):
  Memulai program...
  Nilai counter awal: 0
  Setiap thread akan melakukan 1000000 iterasi increment.
  Nilai counter yang diharapkan: 2000000
  Kedua thread telah selesai.
  Nilai counter akhir (aktual): 1374582 <-- Nilai lebih kecil dari yang diharapkan!
  HASIL TIDAK SESUAI HARAPAN! Race condition kemungkinan besar terjadi.

  Hasil akhir counter jauh lebih kecil dari 2.000.000 yang diharapkan. Ini terjadi karena banyak operasi increment yang "hilang" akibat *interleaving* (selang-seling) eksekusi operasi baca-modifikasi-tulis pada counter oleh kedua *thread* tanpa adanya perlindungan.[12] Setiap kali Anda menjalankan program, hasilnya mungkin berbeda, menunjukkan sifat tidak deterministik dari *race condition*.

B. Solusi Menggunakan Mutex

Sekarang, mari kita modifikasi kode sebelumnya untuk menggunakan Pthread Mutex

guna melindungi *critical section*.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h> // Untuk perror

long long counter = 0;
#define NUM_ITERATIONS 1000000

// Deklarasi variabel Mutex
pthread_mutex_t counter_mutex;

void* worker_thread_mutex(void* arg) {
    for (long long i = 0; i < NUM_ITERATIONS; i++) {
        // Mengunci Mutex sebelum masuk Critical Section
        if (pthread_mutex_lock(&counter_mutex)!= 0) {
            perror("Gagal mengunci mutex");
            exit(EXIT_FAILURE); // Keluar jika gagal lock
        }

        // CRITICAL SECTION (sekarang dilindungi)
        counter++;

        // Membuka kunci Mutex setelah keluar Critical Section
        if (pthread_mutex_unlock(&counter_mutex)!= 0) {
            perror("Gagal membuka mutex");
            exit(EXIT_FAILURE); // Keluar jika gagal unlock
        }
    }
    return NULL;
}

int main() {
    pthread_t thread1_id, thread2_id;

    printf("Memulai program (dengan Mutex)...\n");
    printf("Nilai counter awal: %lld\n", counter);
    printf("Setiap thread akan melakukan %lld iterasi increment.\n", NUM_ITERATIONS);
    printf("Nilai counter yang diharapkan: %lld\n", 2 * NUM_ITERATIONS);
```

```
    // Inisialisasi Mutex sebelum membuat thread
    if (pthread_mutex_init(&counter_mutex, NULL)!= 0) {
        perror("Gagal inisialisasi mutex");
        return 1;
    }

    // Membuat thread pertama
    if (pthread_create(&thread1_id, NULL, worker_thread_mutex, NULL)!= 0) {
        perror("Gagal membuat thread 1");
        pthread_mutex_destroy(&counter_mutex); // Cleanup mutex jika create gagal
        return 1;
    }

    // Membuat thread kedua
    if (pthread_create(&thread2_id, NULL, worker_thread_mutex, NULL)!= 0) {
        perror("Gagal membuat thread 2");
        // Mungkin perlu menunggu thread 1 selesai sebelum destroy, atau cara lain
        pthread_mutex_destroy(&counter_mutex); // Cleanup mutex
        return 1;
    }

    // Menunggu thread pertama selesai
    pthread_join(thread1_id, NULL); // Abaikan error check untuk join demi singkatnya contoh

    // Menunggu thread kedua selesai
    pthread_join(thread2_id, NULL); // Abaikan error check

    printf("Kedua thread telah selesai.\n");
    printf("Nilai counter akhir (aktual): %lld\n", counter);

    // Hancurkan Mutex setelah tidak digunakan lagi
    if (pthread_mutex_destroy(&counter_mutex)!= 0) {
        perror("Gagal menghancurkan mutex");
        // Lanjutkan saja untuk contoh ini
    }

    // Memeriksa hasil
    if (counter == 2 * NUM_ITERATIONS) {
```

```
        printf("Hasil sesuai harapan. Mutex berhasil mencegah race condition.\n");
    } else {
        printf("HASIL TIDAK SESUAI HARAPAN! Masalah terjadi meskipun menggunakan mutex.\n");
    }

    return 0;
}
```

- **Penjelasan Kode:**
  - Variabel pthread_mutex_t counter_mutex; dideklarasikan secara global.
  - Di main, sebelum *thread* dibuat, mutex di-inisialisasi menggunakan pthread_mutex_init(&counter_mutex, NULL);. Argumen NULL berarti menggunakan atribut default.[12] Inisialisasi ini penting untuk menyiapkan mutex sebelum digunakan.
  - Di worker_thread_mutex, sebelum baris counter++;, pthread_mutex_lock(&counter_mutex); dipanggil. Ini akan memastikan hanya satu *thread* yang bisa melewati titik ini pada satu waktu. Jika mutex sudah terkunci, *thread* lain akan menunggu.[12]
  - Setelah counter++;, pthread_mutex_unlock(&counter_mutex); dipanggil untuk melepaskan kunci, memungkinkan *thread* lain (jika ada yang menunggu) untuk masuk.[12]
  - Di main, setelah kedua *thread* selesai (pthread_join), pthread_mutex_destroy(&counter_mutex); dipanggil untuk membersihkan sumber daya yang terkait dengan mutex.[12] Melakukan inisialisasi dan penghancuran (*destroy*) mutex adalah praktik penting untuk manajemen sumber daya yang benar dan menghindari masalah seperti menggunakan mutex yang belum siap atau mencoba menghancurkan mutex yang masih terkunci.[28]
- **Kompilasi & Eksekusi:**
  1. Simpan sebagai mutex_solution.c.
  2. Kompilasi: gcc mutex_solution.c -o mutex_solution -pthread
  3. Jalankan: ./mutex_solution
- Analisis Output:
  Sekarang, Anda akan secara konsisten melihat output seperti ini:
  Memulai program (dengan Mutex)...
  Nilai counter awal: 0
  Setiap thread akan melakukan 1000000 iterasi increment.
  Nilai counter yang diharapkan: 2000000
  Kedua thread telah selesai.

```
Nilai counter akhir (aktual): 2000000  <-- Hasil benar!
Hasil sesuai harapan. Mutex berhasil mencegah race condition.
```

Nilai akhir counter sekarang selalu benar (2.000.000). Ini karena pthread_mutex_lock dan pthread_mutex_unlock memastikan bahwa operasi counter++ dieksekusi secara eksklusif oleh satu *thread* pada satu waktu, sehingga tidak ada lagi *update* yang hilang.

C. (Alternatif) Solusi Menggunakan Semaphore

Sebagai alternatif, kita juga bisa menggunakan POSIX Semaphore untuk mencapai *mutual exclusion*. Kita akan menggunakan *unnamed semaphore* yang cocok untuk sinkronisasi antar *thread* dalam satu proses.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h> // Untuk fungsi-fungsi Semaphore
#include <errno.h>

long long counter = 0;
#define NUM_ITERATIONS 1000000

// Deklarasi variabel Semaphore
sem_t counter_sem;

void* worker_thread_sem(void* arg) {
    for (long long i = 0; i < NUM_ITERATIONS; i++) {
        // Menunggu (wait) / Mengurangi nilai semaphore (P operation)
        if (sem_wait(&counter_sem) == -1) {
            perror("sem_wait gagal");
            exit(EXIT_FAILURE);
        }

        // CRITICAL SECTION (dilindungi oleh semaphore)
        counter++;

        // Memberi sinyal (signal) / Menambah nilai semaphore (V operation)
        if (sem_post(&counter_sem) == -1) {
            perror("sem_post gagal");
            exit(EXIT_FAILURE);
```

```
        }
    }
    return NULL;
}

int main() {
    pthread_t thread1_id, thread2_id;

    printf("Memulai program (dengan Semaphore)...\n");
    printf("Nilai counter awal: %lld\n", counter);
    printf("Setiap thread akan melakukan %lld iterasi increment.\n", NUM_ITERATIONS);
    printf("Nilai counter yang diharapkan: %lld\n", 2 * NUM_ITERATIONS);

    // Inisialisasi Semaphore sebelum membuat thread
    // Argumen kedua 0: semaphore dibagi antar thread dalam proses ini
    // Argumen ketiga 1: nilai awal semaphore (menjadikannya binary semaphore)
    if (sem_init(&counter_sem, 0, 1) == -1) {
        perror("Gagal inisialisasi semaphore");
        return 1;
    }

    // Membuat thread pertama
    if (pthread_create(&thread1_id, NULL, worker_thread_sem, NULL)!= 0) {
        perror("Gagal membuat thread 1");
        sem_destroy(&counter_sem); // Cleanup
        return 1;
    }

    // Membuat thread kedua
    if (pthread_create(&thread2_id, NULL, worker_thread_sem, NULL)!= 0) {
        perror("Gagal membuat thread 2");
        // Mungkin perlu menunggu thread 1 selesai
        sem_destroy(&counter_sem); // Cleanup
        return 1;
    }

    // Menunggu thread pertama selesai
    pthread_join(thread1_id, NULL);
```

```
    // Menunggu thread kedua selesai
    pthread_join(thread2_id, NULL);

    printf("Kedua thread telah selesai.\n");
    printf("Nilai counter akhir (aktual): %lld\n", counter);

    // Hancurkan Semaphore setelah tidak digunakan lagi
    if (sem_destroy(&counter_sem) == -1) {
        perror("Gagal menghancurkan semaphore");
    }

    // Memeriksa hasil
    if (counter == 2 * NUM_ITERATIONS) {
        printf("Hasil sesuai harapan. Semaphore berhasil mencegah race condition.\n");
    } else {
        printf("HASIL TIDAK SESUAI HARAPAN! Masalah terjadi meskipun menggunakan semaphore.\n");
    }

    return 0;
}
```

- **Penjelasan Kode:**
  - Header <semaphore.h> disertakan.
  - Variabel sem_t counter_sem; dideklarasikan.
  - Di main, sem_init(&counter_sem, 0, 1); digunakan untuk inisialisasi. Argumen kedua 0 menandakan semaphore ini untuk *thread* dalam satu proses. Argumen ketiga 1 adalah nilai awal, menjadikannya *binary semaphore* yang efektif berfungsi sebagai mutex.[13]
  - Di worker_thread_sem, sem_wait(&counter_sem); digunakan sebelum counter++. Ini adalah operasi P (wait): jika nilai semaphore > 0, ia akan dikurangi menjadi 0 dan *thread* lanjut; jika nilai 0, *thread* akan menunggu.[13]
  - Setelah counter++, sem_post(&counter_sem); digunakan. Ini adalah operasi V (signal): nilai semaphore ditambah (menjadi 1). Jika ada *thread* lain yang menunggu di sem_wait, salah satunya akan dibangunkan.[13]
  - Di main, sem_destroy(&counter_sem); digunakan untuk membersihkan semaphore setelah selesai.[13]
- **Kompilasi & Eksekusi:**
  1. Simpan sebagai semaphore_solution.c.
  2. Kompilasi: gcc semaphore_solution.c -o semaphore_solution -pthread

(Catatan: Pada beberapa sistem Linux yang lebih tua atau konfigurasi tertentu, Anda mungkin perlu menambahkan flag -lrt untuk library real-time, tapi biasanya -pthread sudah cukup).

3. Jalankan: ./semaphore_solution

- Analisis Output:
  Hasilnya akan sama konsisten dan benarnya dengan solusi Mutex:
  Memulai program (dengan Semaphore)...
  Nilai counter awal: 0
  Setiap thread akan melakukan 1000000 iterasi increment.
  Nilai counter yang diharapkan: 2000000
  Kedua thread telah selesai.
  Nilai counter akhir (aktual): 2000000 <-- Hasil benar!
  Hasil sesuai harapan. Semaphore berhasil mencegah race condition.

  Ini menunjukkan bahwa *binary semaphore* dapat digunakan secara efektif untuk mencapai *mutual exclusion* dengan cara yang mirip dengan mutex. Pola dasar perlindungan *critical section* tetap sama: lakukan operasi 'acquire' (sem_wait) sebelum masuk, dan lakukan operasi 'release' (sem_post) setelah keluar. Baik Mutex maupun Semaphore mengikuti pola fundamental Acquire -> Critical Section -> Release ini untuk memastikan akses yang aman ke sumber daya bersama.

VII. Kesimpulan

Ringkasan Konsep Kunci

Pada pertemuan ini, kita telah membahas konsep-konsep fundamental terkait sinkronisasi dalam sistem operasi:

- **Konkurensi:** Kemampuan sistem operasi menjalankan banyak proses atau *thread* secara bersamaan, yang seringkali memerlukan **berbagi sumber daya**.
- **Race Condition:** Masalah yang timbul ketika akses bersamaan ke sumber daya bersama tidak diatur, menyebabkan **inkonsistensi data**, perilaku tak terduga, dan bahkan **kerentanan keamanan**.
- **Critical Section:** Bagian kode yang mengakses sumber daya bersama dan memerlukan perlindungan khusus untuk mencegah *race condition*.
- **Sinkronisasi:** Kumpulan teknik dan mekanisme (seperti **Mutex** dan **Semaphore**) yang digunakan untuk mengoordinasikan akses ke *critical section* dan sumber daya bersama, memastikan terpenuhinya syarat **Mutual Exclusion**, **Progress**, dan **Bounded Waiting**.

Pentingnya Sinkronisasi

Memahami dan menerapkan mekanisme sinkronisasi dengan benar adalah keterampilan yang sangat penting dalam pengembangan perangkat lunak modern. Di dunia di mana aplikasi semakin kompleks dan memanfaatkan konkurensi untuk kinerja dan responsivitas, kegagalan dalam mengelola akses bersamaan dapat menyebabkan *bug* yang sulit dideteksi, data yang korup, dan sistem yang tidak stabil atau tidak aman. Oleh karena itu, penguasaan konsep sinkronisasi merupakan fondasi untuk membangun perangkat lunak konkuren yang andal, benar, dan aman.

Pengantar ke Topik Lanjutan (Opsional)

Meskipun Mutex dan Semaphore adalah alat yang ampuh untuk mengatasi *race condition*, penggunaannya juga harus hati-hati. Jika tidak dirancang dengan benar, penggunaan mekanisme sinkronisasi ini dapat menimbulkan masalah baru yang kompleks, salah satu yang paling terkenal adalah **Deadlock**. *Deadlock* adalah situasi di mana dua atau lebih proses/thread saling menunggu sumber daya yang sedang dipegang oleh proses/thread lain dalam siklus tunggu, sehingga tidak ada satupun dari mereka yang dapat melanjutkan eksekusi.[7] Masalah *deadlock* dan cara penanganannya akan menjadi topik bahasan pada pertemuan-pertemuan selanjutnya.

Works cited

1. Sinkronisasi Proses, accessed May 13, 2025, https://arna.lecturer.pens.ac.id/Diktat_SO/5.Sinkronisasi%20Proses.pdf
2. Sinkronisasi Proses, accessed May 13, 2025, https://repository.dinus.ac.id/docs/ajar/6.-Sinkronisasi-Proses_.ppt
3. Pertemuan Ke-5 - Sistem Operasi - Sinkronisasi Proses.pptx - SlideShare, accessed May 13, 2025, https://www.slideshare.net/slideshow/pertemuan-ke5-sistem-operasi-sinkronisasi-prosespptx/257400553
4. PERTEMUAN – 5 KULIAH SISTEM OPERASI - SINKRONISASI & DEADLOCK -, accessed May 13, 2025, https://dahlan.unimal.ac.id/files/diktat/os/PERTEMUAN6.ppt
5. Critical section - Wikipedia, accessed May 13, 2025, https://en.wikipedia.org/wiki/Critical_section
6. Critical Section Problem in OS (Operating System) | Hero Vired, accessed May 13, 2025, https://herovired.com/learning-hub/topics/critical-section-problem-in-os/
7. Race Condition Vulnerability | GeeksforGeeks, accessed May 13, 2025, https://www.geeksforgeeks.org/race-condition-vulnerability/
8. Race Condition Vulnerability | Causes, Impacts & Prevention - Imperva, accessed May 13, 2025, https://www.imperva.com/learn/application-security/race-condition/
9. CSE 120 Lecture 6 - andrew.cmu.ed, accessed May 13, 2025, https://www.andrew.cmu.edu/user/gkesden/olducsdstuff/classes/sp16/cse120-a/applications/ln/lecture6.html
10. Critical Section and Mutual Exclusion, accessed May 13, 2025, https://www.cs.mtu.edu/~shene/FORUM/Taiwan-Forum/ComputerScience/004-Concurrency/WWW/SLIDES/05-Sync-Basics.pdf
11. SEMAPHORE | Riza Arifudin - WordPress.com, accessed May 13, 2025, https://rizaarifudin.wordpress.com/2011/02/15/semaphore/
12. Mutex lock for Linux Thread Synchronization | GeeksforGeeks, accessed May 13, 2025, https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/
13. POSIX Semaphores in Linux - SoftPrayog, accessed May 13, 2025, https://www.softprayog.in/programming/posix-semaphores
14. Race condition - Wikipedia, accessed May 13, 2025, https://en.wikipedia.org/wiki/Race_condition
15. Semaphores: Race Condition Problem. For Example, Two Processes Both Increment, accessed May 13, 2025, https://www.scribd.com/document/410554960/semaphore-doc
16. Multithreading in C | GeeksforGeeks, accessed May 13, 2025, https://www.geeksforgeeks.org/multithreading-in-c/
17. What is Race Condition in OS? - Scaler Topics, accessed May 13, 2025, https://www.scaler.com/topics/race-condition-in-os/
18. Studi Web Race Condition Sebagai Acuan Pembuatan Modul Praktikum - Info Kripto, accessed May 13, 2025, https://infokripto.poltekssn.ac.id/index.php/infokripto/article/download/4/12/133

19. SINKRONISASI TABEL BERBASIS RECORD MENGGUNAKAN SISTEM KEAMANAN AUTHENTICATION, AUTHORIZATION, ACCOUNTING (AAA) (STUDI KASUS DI S - Jurnal ITDA, accessed May 13, 2025, https://ejournals.itda.ac.id/index.php/compiler/article/download/22/22
20. Sinkronisasi, accessed May 13, 2025, https://repository.dinus.ac.id/docs/ajar/Sister_6_Sinkronasasi.pdf
21. Sinkronisasi Data Sistem: Manfaatkan Data Demi Bisnis Andal! - Arkatama, accessed May 13, 2025, https://arkatama.id/sinkronisasi-data-sistem-manfaatkan-data-demi-bisnis-andal/
22. Understanding Mutex and Semaphore: Key Differences and Use Cases | NailYourInterview, accessed May 13, 2025, https://nailyourinterview.org/interview-resources/operating-systems/mutex-vs-semaphore/
23. Mutex vs Semaphore | GeeksforGeeks, accessed May 13, 2025, https://www.geeksforgeeks.org/mutex-vs-semaphore/
24. Mutex in Linux Kernel – Linux Device Driver Tutorial Part 22 - EmbeTronicX, accessed May 13, 2025, https://embetronicx.com/tutorials/linux/device-drivers/linux-device-driver-tutorial-mutex-in-linux-kernel/
25. Python Thread Safety: Using a Lock and Other Techniques, accessed May 13, 2025, https://realpython.com/python-thread-lock/
26. Multithreading and Synchronization - Jyotiprakash's Blog, accessed May 13, 2025, https://blog.jyotiprakash.org/multithreading-and-synchronization
27. C Mutex not preventing race condition - Stack Overflow, accessed May 13, 2025, https://stackoverflow.com/questions/53454350/c-mutex-not-preventing-race-condition
28. CON31-C. Do not destroy a mutex while it is locked - CERT, accessed May 13, 2025, https://wiki.sei.cmu.edu/confluence/display/c/CON31-C.+Do+not+destroy+a+mutex+while+it+is+locked