# Seleksi Bagian A
# Laboratorium Sistem Terdistribusi

*"Warisan Fragmentasi"*

Dipersiapkan oleh:

妹ラボラトリー

*(Asisten Laboratorium Sistem Terdistribusi)*

Sister;Lab[22]


**Dikerjakan oleh**:

**M. Rayhan Farrukh - 13523035**


**Waktu Mulai :**

Minggu, 20 Juli 2025, 01.35 WIB

**Waktu Akhir :**

Kamis, 31 Juli 2024, 23.59 WIB

# Tahap I

## I. Organisasi dan Arsitektur Komputer


*Kaguya-sama, we have to write IA-32 Assembly, Kaguya-sama*

**1. (3.5 poin)**

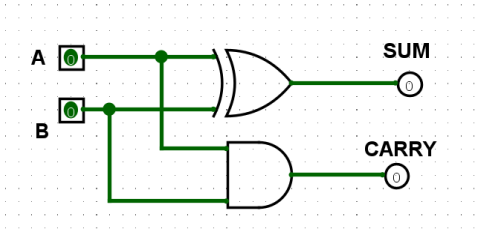
a. Komponen yang tepat untuk ini adalah *half adder*

Arsitektur
Dua input: 2 angka 1 bit yang akan dijumlahkan (misal A dan B)
Dua output: 1 bit SUM dan 1 bit CARRY

Half adder dibentuk oleh dua buah logic gate, XOR (untuk sum), dan AND (untuk carry). Kedua bit input dihubungkan pada dua logic gate ini sehingga memengaruhi hasil penjumlahan.

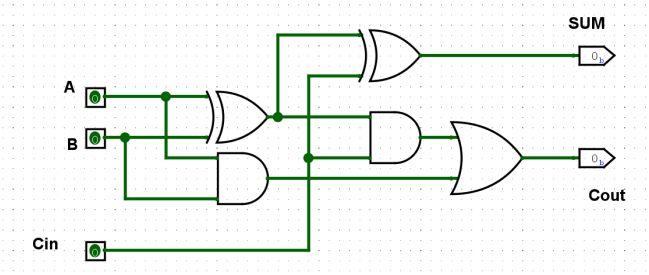
Cara Kerja
SUM hanya akan bernilai satu jika hanya salah satu dari A atau B bernilai 1. Jika keduanya bernilai satu maka XOR menghasilkan 0 sedangkan carry menghasilkan 1, artinya SUM bernilai 0 dan CARRY bernilai 1 (karena 1 + 1 = 10).

Ilustrasi *half adder*




b. Untuk menjumlahkan angka multibit, perlu menambahkan *full adder*, yaitu komponen seperti *half adder* namun dapat menjumlahkan 3 angka 1 bit. Ini berarti kita bisa menghitung 2 angka 1 bit beserta 1 bit *carry* dari hasil penjumlahan sebelumnya. Beberapa *full adder* kemudian bisa dirangkai menjadi *ripple-carry adder* atau lainnya yang mampu menghitung penjumlahan multi-bit.

Ilustrasi *full adder*




c. Komponen Memori

i. SR Latch

Set-Reset latch adalah bentuk memori paling dasar yang dapat menyimpan 1 bit informasi.

Arsitektur

Dua input: S (set) dan R (reset)

Dua output: Q dan Q'

Dua input dihubungkan ke dua NOR gate secara *cross coupled*, dimana output dari satu *gate* menjadi input dari *gate* lainnya. Lalu masing-masing *gate* terhubung ke output.

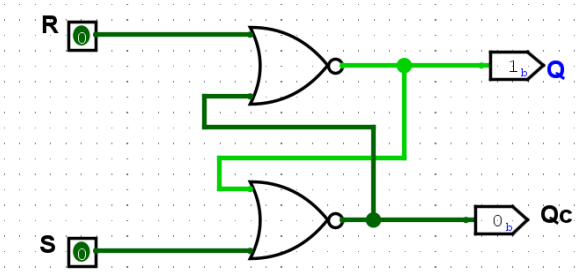
Cara Kerja

Ketika S = R = 0, output menyimpan nilai terakhir dari Q, ketika S = 1, R = 0, output Q akan menjadi 1 dan Q' menjadi 0. Ketika R = 1, Q ter-*reset* dan menjadi 0 sedangkan Q' menjadi 1. Namun, jika S = R = 1, kondisi ini tidak terdefinisi.

Kekurangan

Kekurangan SR latch terletak pada kondisi *undefined* yang disebabkan oleh input S dan R bernilai 1. Ini berarti ada pasangan input yang tidak diperbolehkan.

Ilustrasi SR Latch




ii. Gated SR Latch

Gated SR latch adalah SR latch dengan mekanisme *enable*, dimana hasil input dari S dan R bisa disimpan jika *enable* diaktifkan, namun tidak disimpan jika *enable* tidak aktif.

Arsitektur

Sama seperti SR latch, namun input dari R dan S akan masing-masing terhubung ke AND *gate*, dimana input dari AND *gate* tersebut adalah E (*enable*) serta masing-masing dari R dan S.

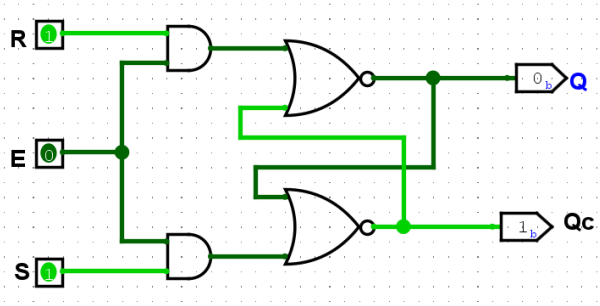
Cara Kerja

Cara kerja sama seperti SR latch, namun penyimpanan *state* hanya terjadi jika *enable* diaktifkan. Ini karena jika E nonaktif (0) maka AND gate yang dilalui R dan S akan menghasilkan nilai 0 juga.

Kekurangan

Sama seperti SR latch, Gated SR latch masih memiliki kondisi tidak terdefinisi, yaitu ketika R dan S hidup, **dan** E juga hidup.

Ilustrasi Gated SR Latch:




iii. Gated D Latch

D latch, atau Gated D latch juga dikenal sebagai *transparent latch*. D latch secara inti sama seperti Gated SR latch, namun input S dan R digabung menjadi satu input utama yaitu input D (data). *Transparent latch* artinya ketika *enable* aktif, output Q akan secara langsung mencerminkan input D tanpa hambatan.

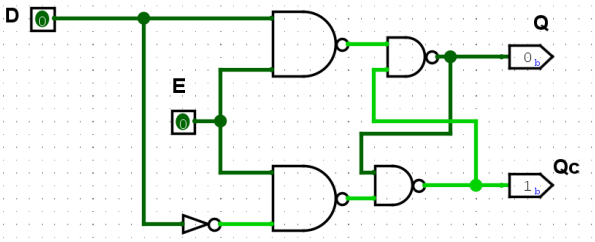
Arsitektur

Input: 1 input data (D) dan 1 input *trigger/enable* (E), input di olah menggunakan NOT *gate* pada jalur yang menyebabkan Qnot hidup sehingga mensimulasikan R.
Bagian lainnya kurang lebih sama dengan Gated SR latch.

Cara Kerja
Jika *enable* diaktifkan, ketika D mati, NOT gate akan menghasilkan output hidup. yang membuat Qnot menjadi hidup. Sebaliknya, saat D hidup, NOT *gate* menghasilkan 0 sehingga Qnot 0 sedangkan jalur lainnya membuat Q bernilai 1. Sifat ini mensimulasikan S dan R dengan satu input, sehingga tidak mungkin terjadi S dan R bernilai 1. Maka, kondisi *undefined* tidak ada pada D latch.

Kekurangan
Sifat *transparency* menjadi kekurangan karena perubahan langsung output sesuai input ini dapat menyebabkan ketidakstabilan output pada sirkuit kompleks yang disebabkan oleh *race condition* dari input.

Illustrasi Gated D Latch




[1] *"Binary Adder with Logic Gates," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/digital-logic/binary-adder-with-logic-gates*
[2] *"Latches in Digital Logic - GeeksforGeeks," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/digital-logic/latches-in-digital-logic/#sr-latch*

## 2. (5 poin)

Misprediksi pada *speculative execution* (dinamakan *transient execution*) dapat memiliki efek samping dimana perubahan cache tidak ter-*revert*, sehingga dapat menyebabkan kebocoran informasi.

Salah satu *vulnerability* yang berkaitan adalah Spectre. Spectre bekerja dengan cara mengelabui program agar melakukan *transient execution* membaca *byte* rahasia. Nilai *byte* ini dapat digunakan sebagai indeks untuk mengakses sebuah *probing array,* sehingga data di array tersebut masuk ke dalam cache. Setelah itu, kita bisa melakukan *brute-force* pada array tersebut untuk mencari indeks berapa yang ada di cache. Jika pengaksesan data indeks tertentu terjadi dengan cepat (*cache hit*) maka indeks tersebut adalah nilai *byte* yang kita cari. Proses ini kemudian dilakukan *byte*-per-*byte* hingga data secara penuh bocor.

Sebelum penggunaan, cache harus diisi agar hacker tahu bahwa cache digunakan. Mekanismenya bergantung pada *cache replacement policy* yang digunakan. Misalnya untuk LRU, hacker akan mengisi penuh cache dengan data yang **diketahui** sebelumnya. Ketika

data tersebut diakses dan ternyata membutuhkan waktu lama (cache miss), maka data *victim* telah masuk cache.

Salah satu mitigasinya adalah “LFENCE”, yang memberi jeda operasi CPU agar tidak terjadi *transient execution*. Misal untuk varian Spectre *Bounds Check Bypass,* pada pengecekan batas array, LFENCE ditaruh agar *transient execution* tidak sempat melewatinya sebelum CPU sadar bahwa misprediksi terjadi.


[1] *"Transient execution CPU vulnerability," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Transient_execution_CPU_vulnerability*

[2] *P. Kocher et al., "Spectre Attacks: Exploiting Speculative Execution," SpectreAttack.com, 2018. [Online]. Available: https://spectreattack.com/spectre.pdf*

[3] *P. Cordes, "How does Spectre attack read the cache it tricked CPU to load?," Stack Overflow, Jan. 14, 2018. [Online]. Available: https://stackoverflow.com/questions/48108572/how-does-spectre-attack-read-the-cache-it-tricked-cpu-to-load.*


**3. (3 Poin)**

a. constprop adalah *Constant Propagation*, yaitu teknik optimasi untuk nilai-nilai konstan pada kode. Karena argumen printf merupakan string literal (“Hello World\n”) yang bersifat konstan, maka *compiler* menggunakan printf spesial dimana argumen telah dimuat saat *compile time*.

b. Tidak. Karena untuk perhitungan seperti sin, optimasi yang digunakan (yang lebih baik) adalah *Constant Folding* dimana nilai secara langsung di-*assign* saat kompilasi tanpa ada pemanggilan fungsi pada *runtime*.

c. FItur ini penting karena merupakan teknik optimisasi yang dasar dan sangat umum digunakan oleh *compiler* modern. Pada dasarnya fitur ini tidak berbahaya, namun sebagai developer compiler, bisa saja terjadi *human error* ketika mengembagkan fitur ini sehingga menghasilkan *bug* yang berbahaya.


*[1] "Constant Propagation in Compiler Design" GeeksforGeeks, May 22, 2025. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/constant-propagation-in-complier-design*


**4. (3 Poin)**

*a.* *Locality of reference* adalah kecenderungan murid untuk membaca beberapa buku yang sama, sehingga perlu diletakkan di perpustakaan kecil agar meminimalkan kemungkinan berjalan ke perpustakaan umum.

b. *Cache miss* terjadi ketika buku yang ingin dibaca tidak ada di perpustakaan kecil, sehingga murid harus ke perpustakaan utama untuk mengambilnya dan meletakkannya di perpustakaan kecil (jika menggunakan LRU *policy*)

c. Sebelum diskusi kelas, murid memprediksi atau mengira buku apa yang mungkin akan dibutuhkan dan meletakkannya di perpustakaan kecil. Sehingga jika memang buku tersebut dibutuhkan murid tidak perlu mengambilnya jauh di perpustakaan utama

**5. (3 Poin)**

a. Mekanisme yang digunakan adalah *register allocation*, lebih spesifiknya *register spilling*. *Register spilling* adalah mekanisme dimana ketika *physical register* penuh, namun dibutuhkan tempat untuk menyimpan suatu nilai di *registeri*, maka terjadi pemindahan suatu nilai pada *physical register* ke RAM. Mekanisme ini serupa dengan *caching* dan memiliki aturan serupa dengan *cache replacement policy* yang diatur oleh *compiler*.

*b.* *Register spilling* berdampak negatif pada *Read after Write* (RAW) *hazard*. Ini karena ketika ada instruksi yang membutuhkan suatu *register value* yang sedang berada di RAM, untuk mengembalikan nilai tersebut ke *register* membutuhkan waktu yang lama. Sehingga durasi *stall* untuk meng-*handle* RAW menjadi sangat lama.

c. Optimasi pada kompilasi sekarang perlu meminimalkan terjadinya *register spilling* dengan memperhitungkan *spill cost* menggunakan aturan yang serupa dengan *cache replacement policy*. Performa kode menjadi tergantung pada kemampuan *compiler* untuk menghindari *register spilling*.


[1] *"Register allocation," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Register_allocation.*

[2] *GeeksforGeeks, "Data Hazards and its Handling Methods," GeeksforGeeks. Last updated Dec. 28 2024 [Online]. Available: https://www.geeksforgeeks.org/computer-organization-architecture/data-hazards-and-its-handling-methods/.*


**6. (2.5 poin)**

a. Byte alignment dan ukuran

i. hawk

a - 4 byte (sesuai iii)
b - 1 byte (char, walaupun list, tetap 1 byte)
c - 4 byte (int pada C)
Ukuran total:
12 byte. 4 + 4 + (3 * 1), ditambah 1 byte padding agar menjadi kelipatan alignment terketat, yaitu 4 byte.

ii. tuah

d - 1 byte (char)
e - 4 byte (int)
f - 4 byte (pointer)
g - 4 byte (pointer)
Ukuran Total:
28 byte. 1 + (4 * 4) + 4 + 4, ditambah 3 byte padding agar menjadi kelipatan 4.

iii. sybau

h - 1 byte (char)
i - 4 byte (pointer 32 bit)
j - 4 byte (pointer 32 bit)
Ukuran total:
4 byte. Union menyimpan data di tempat yang sama dengan tipe yang berbeda. Ukurannya harus dapat memuat atribut terbesar didalamnya dan merupakan kelipatan dari ukuran *alignment* terketat (atribut dengan alignment terbesar).

b. Return value fungsi

i. proc1

`movl 8(%ebp), %eax` - Memuat argumen pertama (x)
`movl 12(%eax), %eax` - Meletakkan 4 byte data pada offset 12 dari x
Offset 12 dari x adalah e[2] (1 d + 3 padding + 4 e[0] + 4 e[1])
Maka,

```
int proc1(struct tuah *x) { return x->e[2]}
```

ii. proc2

| | |
|---|---|
| `movl 8(%ebp), %eax` | Memuat argumen pertama (x) |
| `movl (%eax), %eax` | memuat pointer yang ada di atribut a (union) pada x, ini mungkin i atau j (x→a.?) |
| `movl 24(%eax), %eax` | karena offsetny 24, maka instruksi sebelumnya mengakses struct tuah (j) pada a (x→a.j), kemudian instruksi ini memuat pointer g yang ada di j (x→a.j→g) |
| `movl 20(%eax), %eax` | Pada g, akses atribut pada offset 20, yaitu pointer f (x→a.j→g→f) |
| `movzbl 6(%eax),%eax` | Mengakses offset 6 dari f, yaitu b pada indeks ketiga b[2] (x→a.j→g→f→b[2]) |

```
int proc2(struct hawk *x) { return x→a.j→g→f→b[2]}
```

**7.** (1 Poin)

Little endian menjadi populer karena alasan historis, yaitu arsitektur PC paling dominan (Intel x86) memilih menggunakannya sejak awal. Alasan penggunaan ini adalah adanya keuntungan performa pada operasi aritmatika tertentu serta cara mengakses memori, dan pembacaan little endian yang lebih natural untuk CPU.

## II. Sistem Operasi


*Linux my beloved*

**1. (1 poin)**

Ketika *multiprogramming* terlalu tinggi, RAM tidak mampu menampung semua proses tersebut sehingga sistem terlalu sibuk *paging*, yaitu memindahkan proses antara RAM dan disk. Pada kondisi ini, CPU lebih banyak menunggu *paging* selesai daripada mengeksekusi instruksi, sehingga menurunkan CPU *utilization*. Fenomena ini dalam manajemen memori disebut *thrashing*.

**2. (1.5 poin)**

a. *Virtualization* adalah teknologi yang memungkinkan satu komputer fisik menjalankan banyak komputer virtual yang masing-masing berjalan dengan sistem operasi dan *virtual hardware* sendiri, dan terisolasi dari komputer fisik.

   Cara kerja
   *Virtualization* bekerja karena adanya *hypervisor* yang menggunakan sumber daya komputer fisik dan membaginya untuk digunakan *virtual environment*. Selain itu, *hypervisor* melakukan semua komunikasi antara sistem operasi komputer fisik dengan sistem operasi komputer-komputer virtual yang berjalan di atasnya.

b. *Containerization* memungkinkan satu komputer menjalankan banyak *virtual environment* dengan *library* dan *dependency* tersendiri, namun hanya untuk aplikasi tertentu dan tidak menyertakan sistem operasi di dalamnya.

   Cara kerja
   *Containerization* menggunakan *container* yang merupakan *package* dari suatu aplikasi beserta *dependency*-nya. Masing-masing *container* terisolasi dari *container* lainnya

sehigga tidak ada konflik *dependency*, namun tetap pada kernel OS yang sama. Ini memastikan *runtime environment* yang konsisten untuk aplikasi.

c. *Virtualization vs Containerization*

Persamaan

Keduanya sama-sama memungkinkan beberapa lingkungan terisolasi untuk berjalan di atas satu mesin fisik tunggal dan memiliki tujuan sama untuk memudahkan *development* dan *deployment*

Perbedaan

Perbedaannya adalah pada tingkat abstraksi. *Virtualization* mensimulasikan *hardware* yang menjalankan operasi sistem penuh, sedangkan *containerization* hanya mensimulasikan *environment* untuk aplikasi dengan menggunakan kernel OS yang sama.

Kelebihan

*Virtualization* - keamanan yang lebih terjamin karena VM menjalankan OS yang terpisah.
*Containerization* - efisiensi sumber daya, lebih cepat.

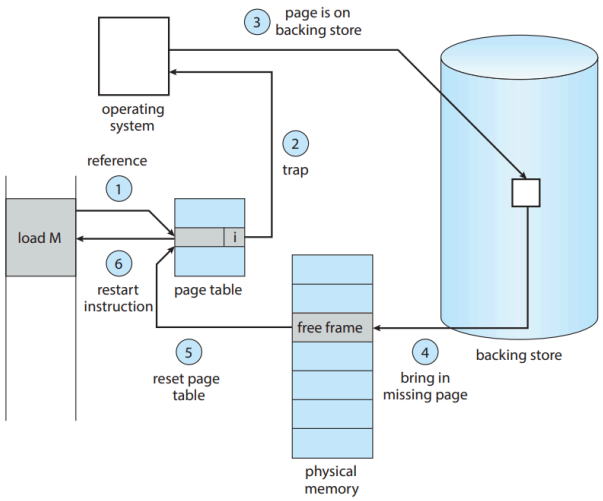
Kekurangan

*Virtualization* - konsumsi sumber daya yang tinggi dan lebih lambat.
*Containerization* - isolasi lemah yang membuat keamanannya lebih lemah dari *virtualization*


[1] *S. Musib, "Virtualization vs Containerization," Baeldung on Computer Science, 2024. [Online]. Available: https://www.baeldung.com/cs/virtualization-vs-containerization.*

[2] *"Virtualization vs. Containerization in System Design," GeeksforGeeks, 2024. [Online]. Available: https://www.geeksforgeeks.org/system-design/virtualization-vs-containerization.*


**3. (1.5 poin)**



Cara Kerja *Virtual Memory*

1. CPU mencoba mengakses memori dengan mencari referensi *page table*, namun *page table* menandakan bahwa *page* memori tersebut tidak ada di RAM.
2. Terjadi *trap* (*interrupt*) karena akses memori tidak valid.
3. Sistem operasi meng-*handle trap* dengan mencari *page* yang dibutuhkan di *backing store*.
4. Sistem operasi memuat *page* tersebut dari *backing store* ke dalam *frame* yang kosong pada RAM.
5. *Page table* diperbarui dengan entri lokasi baru *page* pada RAM.
6. Instruksi yang sebelumnya gagal akan dicoba kembali, yang kali ini akan berhasil karena *page* sudah ada di RAM.

**4. (6 poin)**

a. Algoritma Peterson bekerja dengan cara menandakan apakah suatu proses ingin masuk *critical section,* serta menandakan giliran proses mana. Algoritma Peterson hanya bekerja untuk dua proses saja.

   Ketika suatu proses (misal **i**) ingin masuk, ia mengganti `flag`-nya menjadi *true*, kemudian menandakan bahwa sekarang giliran proses lain (misal **j**). Lalu algoritma memasuki blok `while` *loop*, jika `flag` **j** *true* maka **i** menunggu hingga `flag` **j** *false* (artinya **j** telah keluar dari *critical section*). Jika `flag` **j** *false*, maka blok `while` dilewati dan **i** masuk (tanpa menandakan bahwa ini gilirannya). Setelah **i** keluar, `flag` **i** menjadi *false* kembali.

b. Variabel `flag` menandakan niat proses untuk memasuki *critical section*, memberi tahu proses lain tentang niat tersebut. `turn` menandakan giliran proses mana untuk masuk, berguna sebagai *tie-breaker* agar tidak terjadi konflik.

   JIka hanya `flag` yang ada, maka ketika **i** dan **j** sama-sama ingin masuk, mereka akan saling menunggu tanpa ada *progress* (*deadlock*). Jika hanya ada `turn`, proses yang tidak berniat masuk *critical section* bisa menahan proses yang ingin masuk, sehingga harus menunggu pergantian `turn` yang belum tentu akan terjad .

c. Dengan asumsi proses yang masuk ke *critical section* pasti akan keluar dalam waktu yang wajar, Peterson's Solution merupakan solusi benar karena memenuhi 3 syarat solusi *critical section problem:*
   - *Mutual exclusion,*
     Variabel `turn` tidak mungkin bernilai **i** dan **j** sekaligus, sehingga keduanya tidak akan berada di *critical section* bersamaan.
   - *Progress*
     Proses dijamin akan maju karena hanya akan menunggu jika proses lain berniat masuk. Dan jika masuk, pasti akan keluar
   - *Bounded waiting*

variabel `turn` tidak mungkin bernilai **i** dan **j** sekaligus, sehingga keduanya tidak akan berada di *critical section* bersamaan.

d. Kelemahan Peterson's Solution
   - Hanya untuk dua proses.
   - Rentan terhadap *out-of-order execution* karena bergantung pada asumsi bahwa instruksi dijalankan dalam urutan tertentu.
   - Menggunakan *busy-waiting*, sehingga tidak efisien.

e. Metode alternatif

i. `test_and_set()` digunakan pada sebuah variabel *lock* untuk mengecek status (*test*) *lock* tersebut. Sebuah proses menggunakan fungsi ini untuk mencoba mengambil kunci (*set*) dan masuk ke *critical section*. Jika hasil fungsi *false*, maka kunci berhasil diambil dan bisa lanjut ke *critical section*. Jika hasil fungsi *true*, kunci sudah diambil proses lain, dan harus menunggu kunci dilepaskan kembali.

ii. `compare_and_swap()` bekerja dengan mengecek nilai suatu variabel untuk memastikan pembaruan aman, memodifikasinya, kemudian mengembalikan nilai awal. Proses membaca sebuah variabel, kemudian `compare_and_swap()` mengecek kembali nilai variabel tersebut, jika nilainya masih sama seperti sebelumnya, maka penukaran nilai berhasil. Jika nilainya sudah diubah oleh proses lain, penukaran gagal dan proses mengulang dari awal.

Kedua fungsi tersebut bekerja dengan instruksi atomik, sehingga tidak akan terganggu *mid-instruction*.

f. *Out-of-order execution*, yaitu kelemahan kritis Peterson's Solution tidak berpengaruh pada instruksi atomik, sehingga membuat kedua cara andal dibanding Peterson's Solution. Selain itu, keduanya juga dapat meng-*handle* lebih dari dua proses.

[1] *"Peterson's Algorithm in Process Synchronization," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/dsa/petersons-algorithm-in-process-synchronization.*

[2] *Shahid Nihal, "Hardware Synchronization: Test and Set, Compare and Swap," *YouTube*. Uploaded: Aug. 31, 2024. Accessed: July 22, 2025. [Online]. Available: http://www.youtube.com/watch?v=Fxsd5k1xjd0*

**5. (2.5 poin)**

```
int putchar (int c)
{
  int result;
  _IO_acquire_lock (_IO_stdout);
  result = _IO_putc_unlocked (c, _IO_stdout);
  _IO_release_lock (_IO_stdout);
  return result;
}
```

Fungsi `putchar` merupakan *wrapper* yang *thread-safe* mekanisme *mutex lock* agar penulisan karakter ke *buffer* aman .

*Mutex lock* yang digunakan adalah *macro* `_IO_acquire_lock` dan `_IO_release_lock` pada *standard output* (stdout). Dua *macro* tersebut berada di file "libioP.h". `putchar` tidak langsung menuliskan karakter pada layar, tetapi mennyimpannya di dalam sebuah *buffer*. *Mutex locking* berguna untuk menjaga kestabilan *buffer* tersebut. Variabel stdout (`_IO_stdout`) adalah *pointer* pada sebuah *struct* `_IO_FILE` yang memiliki atribut *buffer* dimana karakter disimpan

Setelah *lock* didapatkan, mekanisme utama `putchar` akan dipanggil melalui *macro* `_IO_putc_unlocked`. *Macro* ini sendiri hanya memanggil *macro* lain yaitu `__putc_unlocked_body`. Disinilah mekanisme `putchar` yang sebenarnya.

*Macro* ini didefinisikan sebagai berikut.

```
#define __putc_unlocked_body(_ch, _fp)             \
  (__glibc_unlikely ((_fp)->_IO_write_ptr >= (_fp)->_IO_write_end)   \
   ? __overflow (_fp, (unsigned char) (_ch))          \
   : (unsigned char) (*(_fp)->_IO_write_ptr++ = (_ch)))
```

*Macro* ini menggunakan *ternary operator* untuk mengecek *buffer overflow*, dengan mengecek apakah posisi kosong pada *buffer* melebihi posisi akhir *buffer*. Jika iya, fungsi `__overflow` dipanggil yang akan menge-*flush* *buffer*, yaitu menulis sekaligus mengosongkan isi *buffer*. Jika overflow tidak terjadi, karakter disimpan pada *buffer* dan *pointer* dinaikkan untuk menyiapkan penulisan karakter berikutnya. Perlu dicatat bahwa *overflow* sangat jarang terjadi, dan karena itu ada *flag* `__glibc_unlikely`, agar *compiler* dapat mengoptimasi instruksi tersebut.

**6. (2.5 poin)**

Tahapan *kernel exploitation*:

- *Vulnerability Discovery/Analysis,* mengidentifikasi jenis *vulnerability* yang dapat digunakan.
- *Initial Exploitation*, eksploitasi awal untuk mendapat akses dasar pada *user mode* pada sistem target.
- *Privilege Escalation*, meningkatkan akses yang sudah didapat agar memiliki hak lebih tinggi, sehingga penyerang berada di *kernel mode* (ring 0). Setelah itu, penyerang bisa melakukan serangan yang lebih *powerful*.
- *Persistence Mechanisms*, membuat celah pada sistem untuk mempertahankan akses, seperti membuat *backdoor* agar mendapatkan akses menjadi sangat mudah.

*Kernel exploitation* membutuhkan penyerangan bertahap karena satu tahap saja tidak cukup. Eksploitasi pertama mungkin berhasil memberikan akses ke sistem, namun umumnya akses ini terbatas sehingga penyerang tidak bisa melakukan hal bermakna (misalnya membaca file tertentu). Oleh karena itu dibutuhkan eksploitasi lebih lanjut untuk mendapat akses lebih, khususnya akses *kernel mode*. Selain itu, akses ini bisa hilang ketika

sistem di-*shutdown* atau di-*restart*, sehingga dibutuhkan eksploitasi lagi agar bisa mempertahankannya walaupun sistem di-*shutdown*. Eksploitasi untuk mempertahankan akses ini disebut *persistence mechanism*.

Perbedaan tahapan *kernel exploitation*
- *Initial exploitation* - pintu masuk pertama.
- *Privilege escalation* - menaikkan level hak akses.
- *Persistence mechanisms* - membuat akses menjadi permanen.

**7. (2.5 poin)**

a. Komponen yang memungkinkan ini terjadi adalah *Virtual Filesystem Switch* (VFS). Ketika user mengakses/memodifikasi suatu file, *system call* yang dipanggil akan berkomunikasi dengan VFS, yang kemudian mentranslasikan operasi pada file sesuai *filesystem* dimana file tersebut disimpan.

b. Cara kerja VFS adalah seperti adapter USB yang menghubungkan colokan tipe tertentu (misal Type-C), yang dapat menerjemahkan koneksi pada port dengan banyak tipe lain yang berbeda (misal Type-A, Type-B, atau *lighning cable*)

[1] *"Virtual File System," GeeksforGeeks, 2025. [Online]. Available: https://www.geeksforgeeks.org/operating-systems/virtual-file-system.*

**8. (2.5 poin)**

a. Kernel itu seperti otak komputer, tugas dia lah yang ngatur bagian-bagian komputer untuk bekerja, sama seperti otak yang mengatur tubuh kita untuk bergerak.

b. RAM itu seperti meja bermain, tempat kamu narok mainan-mainan yang lagi kamu mainin biar gampang diambil. Tapi kalo kamu biarin mainan di meja dan nggak diberesin, nanti mainannya bakal hilang.

c. Filesystem itu seperti lemari mainan, di lemari itu diatur cara nyimpan mainan-mainan supaya nanti gampang dicari lagi dan nggak mudah hilang atau rusak.

d. Shell itu cara kita bisa bicara dengan komputer. Ketika kamu bicara dengan komputer dengan mengetik sesuatu, shell bakal nyampain pesan kamu ke komputer untuk dikerjakan

e. Multiprocessing, karena kamu punya dua tangan, kamu bisa main mobil-mobilan pake satu tangan, terus tangan lain kamu ngegambar sesuatu secara bersamaan. Komputer juga bisa ngelakuin itu dengan prosesor yang banyak.

**9. (2 poin)**

a. Kernel menyiapkan dan memuat sebuah *root filesystem* sementara pada RAM. Ini dilakukan untuk memuat modul dan driver penting yang dibutuhkan untuk *startup*, serta untuk memuat *filesystem* sebenarnya.

b. Untuk membaca *filesystem*, kernel perlu *driver* dari *filesystem* tersebut, sedangkan *driver* tersebut tersimpan di dalamnya (jika disimpan di kernel, akan terlalu besar). Oleh karena itu dibutuhkan initramfs/initrd untuk menjembatani kernel ke *filesystem*. Selain

itu, *filesystem* kompleks seperti LVM atau NFS membutuhkan penanganan spesial untuk me-*mount*.

c. Booting akan gagal karena kernel tidak dapat menemukan *filesystem* yang akan digunakan. Ini menyebabkan *kernel panic*.


[1] *Gentoo Wiki, "Initramfs," Gentoo Wiki, [Online]. Available: https://wiki.gentoo.org/wiki/Initramfs.*

[2] *"Why do we need initramfs/initrd to boot a Linux system? Can't we store all device and FS drivers directly into the Kernel?," Quora, May 15, 2017. [Online]. Available: https://www.quora.com/Why-do-we-need-initramfs-initrd-to-boot-a-Linux-system.*


**10. (2 poin)**

Keuntungan copy-on-write adalah efisiensi pada kecepatan dan penggunaan memori. Fork menjadi sangat cepat karena tidak perlu langsung menyalin ruang memori dari Induk ke Anak. Selain itu, jika proses anak hanya memanggil proses lain, maka tidak perlu menghabiskan waktu dan memori untuk menyalin ruang memori ke proses anak.


[1] *Simplyblock, "Copy-on-Write (CoW)," Simplyblock Glossary, [Online]. Available: https://www.simplyblock.io/glossary/copy-on-write.*

## III. Teknologi Platform


*POV: Me after ruling the cloud infrastructure*

**1. (6 poin)**

*a.* *Container* menggunakan isolasi level sistem operasi, dimana *container* berjalan sebagai sebuah proses di dalam sebuah *pod*, dann berbagi satu kernel dengan *node*.

*Unikernel* menggunakan isolasi level *hardware*, dimana setiap proses menggunakan *kernel* sendiri di dalam sebuah *pod*. *Kernel* di-*compile* seminimal mungkin khusus untuk suatu proses tertentu.

*Attack surface* akan menjadi lebih sedikit pada *unikernel* karena sistem operasi minimal yang digunakan untuk *auth service* tidak meng-*include* banyak *syscall* atau API yang dimiliki kernel pada umumnya, tetapi tidak dibutuhkun untuk *auth service*.

b. *Logging*, *monitoring* dan *debugging* dapat disebut sebagai *observability*. Kesulitan utama *observability unikernel* pada Kubernetes adalah hilangnya sistem operasi minimal yang tidak memiliki fitur-fitur yang diandalkan oleh *workflow* yang konvensional.

Untuk *logging*, tidak ada tempat standar menyimpan *log* atau *log* tidak mudah diakses dari luar *pod*. Untuk *monitoring*, sulit mendapat metrik kinerja karena agen pengumpul metrik mungkin tidak bekerja. Untuk *debugging*, tanpa adanya *shell* interaktif, tidak bisa masuk ke *pod* untuk menjalankan debugginng tools.

Beberapa solusi:
- *Logging*, menggunakan *logging* berbasis jaringan yang mengirim *log* ke *aggregator* melalui jaringan
- *Monitoring*, menggunakan metrik *hypervisor* bukan metrik dari aplikasi atau sistem operasi *unikernel*. Selain itu, aplikasinya sendiri dapat dikembangkan dengan kemampuan mengukur dan mengirim metrik.

- *Debugging*, memperbanyak *logging* atau mereproduksi masalah di *local environment*.

c. Sebaiknya tidak, karena implementasi *unikernel* tidak mudah dan ada cara yang lebih simpel untuk memperkuat keamanan dan mempermudah *scale-out*. Migrasi ke *unikernel* hanya kompleksitas dari migrasi jauh lebih besar ketimbang keuntungan yang ditawarkan dalam ekosistem Kubernetes yang sudah 'matang'.

   *Pros and Cons* migrasi *unikernel*:
   Pros:
   - Keamanan superior, karena isolasi kuat dan sistem operasi yang minimal
   - Skalabilitas cepat, sistem operasi yang minimal membuat ukuran *image* kecil dan waktu *bootup* yang cepat, sehingga proses *scaling* menjadi cepat.

   Cons:
   - Integrasi Kubernetes yang sulit, untuk dapat mengelola VM, Kubernetes membutuhkan lapisan tambahan seperi KubeVirt.
   - *Observability* menjadi rumit, *tools* dan metode untuk *logging*, *monitoring*, dan *debugging* pada *unikernel* berbeda dan lebih rumit dari *container*.
   - Ekosistem belum matang, *tools-tools* untuk mengembangkan *unikernel* masih terbatas dibandingkan ekosistem *container*.
   - Membutuhkan keahlian baru, penggunaan *unikernel* membutuhkan *workflow* dan keahlian teknis yang membutuhkan waktu *training*.

   Alternatif yang dapat digunakan adalah *image distroless* yang minimal namun masih kompatibel dengan ekosistem dan *workflow* Kubernetes/Docker yang digunakan FCC .inc.


[1] *PhoenixNAP, "Unikernel vs. Container," PhoenixNAP Knowledge Base, 2024. [Online]. Available: https://phoenixnap.com/kb/unikernel-vs-container.*

[2] *NanoVMs, "Security," NanoVMs, 2024. [Online]. Available: https://nanovms.com/security.*

[3] *GeeksforGeeks, "Kubernetes Monitoring and Logging," GeeksforGeeks, 2023. [Online]. Available: https://www.geeksforgeeks.org/devops/kubernetes-monitoring-and-logging.*


**2. (2.5 poin)**

Docker menjamin konsistensi dengan *containerization*, yaitu membungkus aplikasi beserta *dependency*-nya ke dalam *container* yang terisolasi. Isolasi ini membuat aplikasi dijalankan menggunakan *runtime environment* pada *container* yang dibuat, tanpa mempedulikan *environment* dari sistem *host*. Hal ini penting untuk menghindari kasus dimana aplikasi berjalan dengan baik pada satu *device* namun bermasalah pada *device* lain karena perbedaan *environment* yang sulit untuk di-*resolve*.

Risiko pengelolaan *environment* tidak baik adalah masalah seperti *bug* yang muncul di server namun tidak pada komputer *developer*, bahkan sampai membuat *deployment* gagal, sehingga menurunkan produktivitas karena harus memperbaiki masalah tersebut. Docker meminimalkan masalah melalui *containerization* dengan manfaat yang sudah dijelaskan di paragraf pertama.