

Disclaimer: Soal ini dikerjakan dengan bantuan Gemini. Bismillah!

I. Organisasi dan Arsitektur Komputer



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

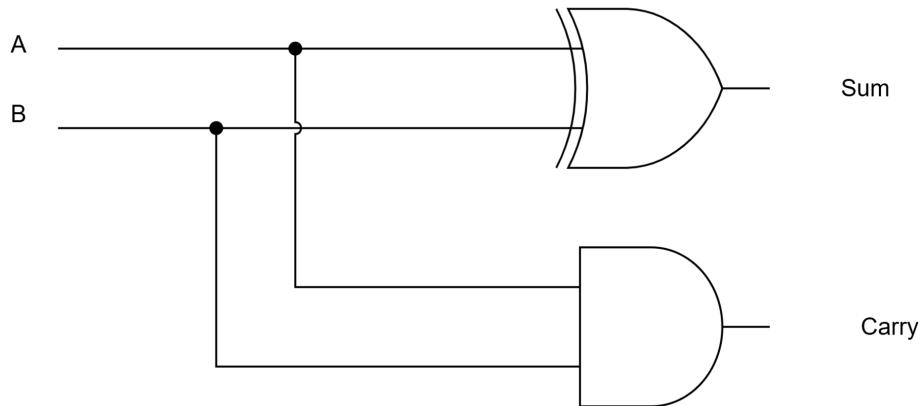
1. (3.5 poin) Pada level sistem paling rendah, sebuah komputer hanyalah sekumpulan transistor yang disusun sedemikian rupa sehingga menghasilkan berbagai *logic gates* yang dapat digunakan untuk membuat komponen kalkulasi input-output pada angka biner.
 - a. (0.75 poin) Bagaimana kita menyusun sebuah komponen *logic gate* untuk melakukan penjumlahan dua angka berukuran 1-bit? Jelaskan arsitektur dan cara kerja dari komponen tersebut.

Kalau kita mau nambahin dua angka super kecil (cuma 1-bit), kita pakai sirkuit namanya Half-Adder.

Dia nerima dua input (misalnya A dan B) dan ngeluarin dua output: SUM (hasilnya) dan CARRY (sisaannya, kalau ada).

- $0 + 0 = 0$ (SUM=0, CARRY=0)
- $1 + 0 = 1$ (SUM=1, CARRY=0)

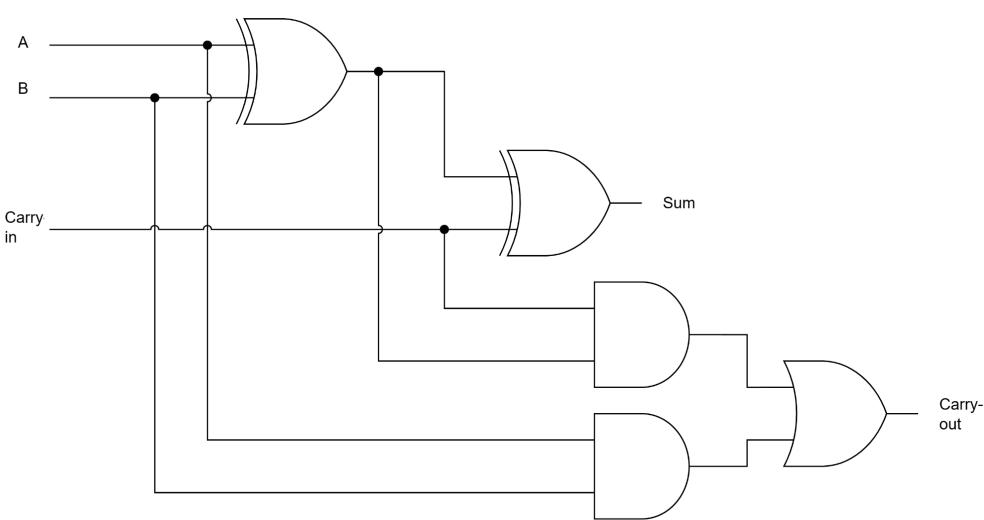
- $1+1=10$ dalam biner, jadi 0 dengan sisaan 1(SUM=0, CARRY=1)



Half-Adder gabisa nerima "sisaan" (carry) dari penjumlahan sebelumnya. Makanya, dia cuma cocok buat nambahin satu bit doang dan gabisa dipakai buat matematika yang lebih rumit.

- b. (0.75 poin) Bagaimana jika kita ingin menjumlahkan dua angka yang berukuran lebih besar dari 1-bit? Perubahan apa atau komponen apa yang perlu kita buat? Jelaskan.

Buat nambahin angka yang lebih gede (misalnya 8-bit), kita butuh Full -Adder. Bedanya, dia punya tiga input: A, B, dan Carry-In (sisaan dari penjumlahan bit di sebelahnya). Jadi, dia bisa nerima "operan" dari temennya.



- Ripple-Carry Adder: Kalau mau nambahin angka 8-bit, kita tinggal jejerin aja 8 Full-Adder. Rangkaian ini disebut Ripple-Carry Adder. Tapi masalahnya, agak lambat. Hasil di ujung harus nunggu "sisaan" dari awal merambat satu per satu. Makin panjang angkanya, makin lama nunggunya.

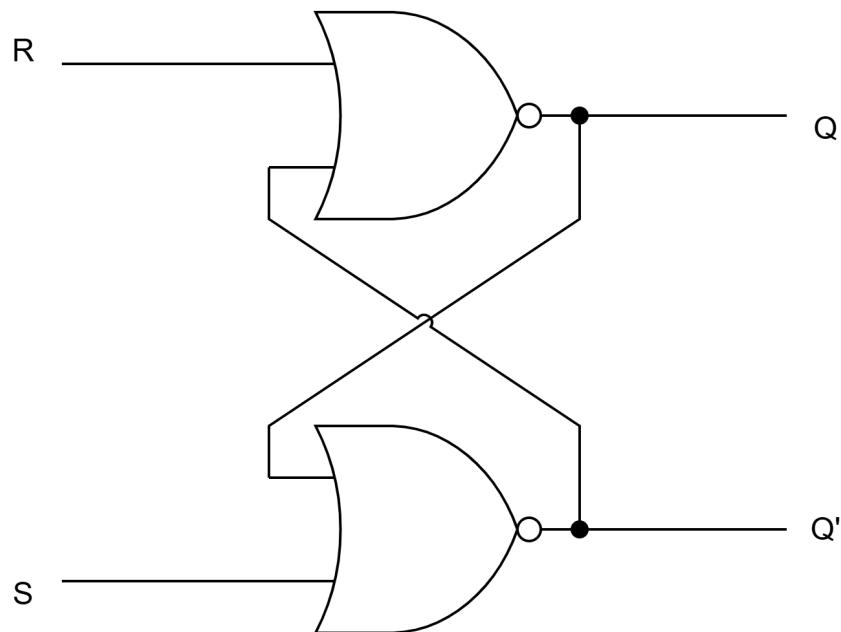
- c. (2 poin) Selain kalkulasi, sifat penting yang perlu dimiliki sebuah komputer adalah dapat menyimpan informasi, baik secara *volatile* (*memory*) ataupun *non-volatile*. Beberapa komponen yang dapat digunakan untuk mengimplementasikan memori adalah sebagai berikut. Untuk masing-masing komponen, jelaskan arsitektur, cara kerja, dan kekurangan dari menggunakan komponen tersebut jika ada.

- i. (1 poin) SR Latch

SR Latch (Set-Reset Latch) adalah elemen memori paling fundamental, biasanya dibikin dari dua gerbang NOR atau dua gerbang NAND yang dihubungkan secara silang (cross-coupled). Outputnya dari setiap gerbang akan menjadi input bagi gerbang lainnya. Ini bakal ngebikin sebuah *feedback loop* yang memungkinkannya untuk menyimpan satu bit informasi. Outputnya adalah Q (nilai yang disimpan) dan Q-bar (komplemennya).

Operasinya:

- **Set (S):** Kalau input $S = 1$, output Q jadi 1.
- **Reset (R):** Kalau input $R = 1$, output Q jadi 0.
- **Hold:** Kalau S dan R keduanya gak aktif (misalnya, 0 untuk gerbang NOR), feedback loop akan mempertahankan keadaan terakhir, "mengunci" (*latching*) nilainya.



Kekurangan

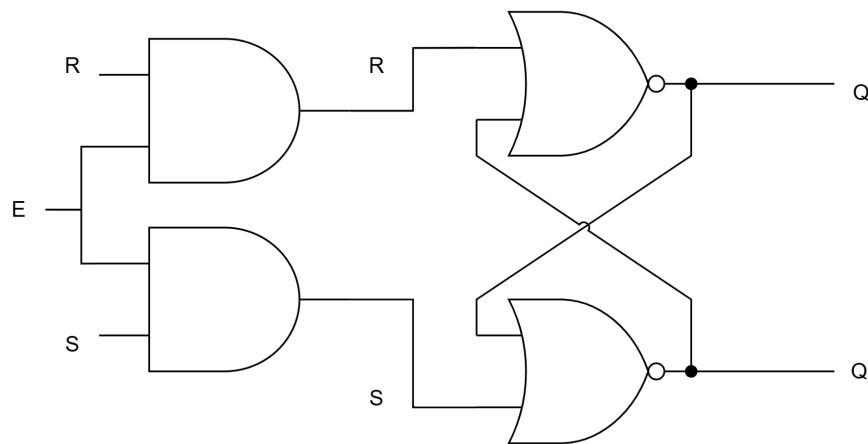
Kelemahan utama dari SR Latch adalah adanya **keadaan terlarang** (*invalid/forbidden state*). Jika S dan R diaktifkan secara bersamaan (misalnya, $S=1$ dan $R=1$ untuk latch berbasis NOR), kedua output Q dan Q -bar akan menjadi 0. Padahal, harusnya $Q \neq Q$ -bar. Nanti, kalau inputnya kembali ke keadaan *hold* dari keadaan terlarang ini, keadaan akhir latch menjadi tidak dapat diprediksi dan bergantung pada gerbang mana yang bereaksi lebih cepat \rightarrow Race Condition

ii. (0.5 poin) Gated SR Latch

Arsitektur dan Cara Kerja

Gated SR Latch (atau Clocked SR Latch) sebenarnya sama kayak SR Latch, tapi ditambah input ketiga yang disebut **Enable** (E) atau Clock (Clk). Kalau tadi kita pakai dua gerbang NOR, kita tinggal tambah dua gerbang AND.

Input S dan R hanya akan memengaruhi output latch ketika input E aktif (misalnya, bernilai 1). Kalau E tidak aktif (bernilai 0), gerbang AND akan memblokir sinyal S dan R, sehingga tetap dalam keadaan *hold* -> Bisa bikin sinkronisasi



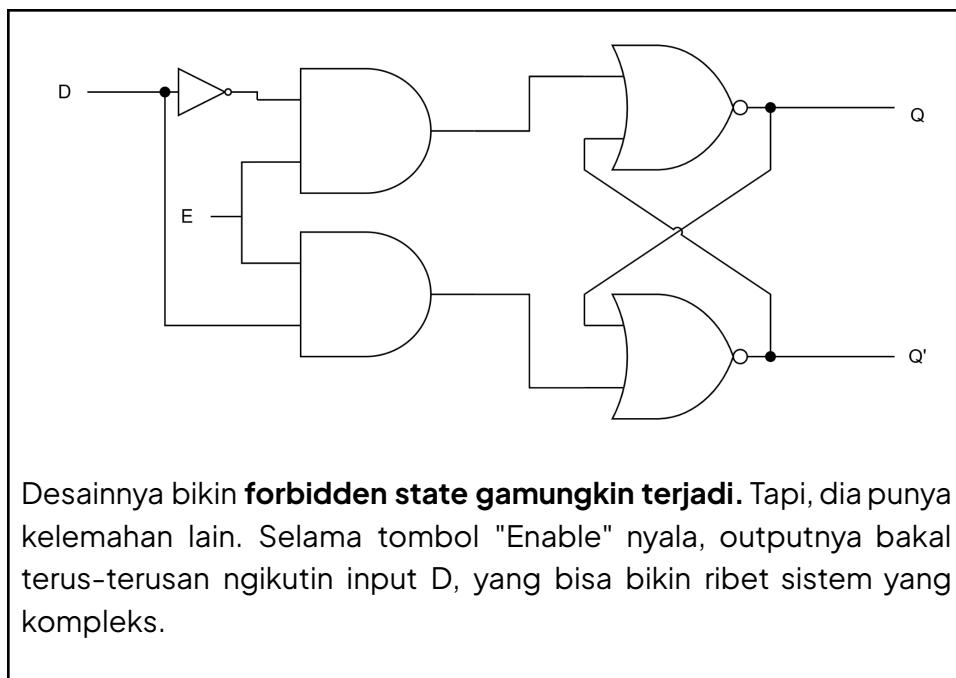
Kekurangan

Masih sama kayak SR Latch, masih mungkin terjadi forbidden state.

iii. (0.5 poin) Gated D Latch

Gated D Latch ini hasil "perbaikan" dari SR Latch. Dia lebih pinterr karena *instead of make S and R*, dia punya satu input data (D) dan satu input "Enable" (E).

- Ketika E bernilai HIGH (aktif), latch jadi "transparan": output Q akan secara langsung mengikuti nilai input D. Setiap perubahan pada D akan segera tercermin pada Q.
- Ketika E bernilai LOW (tidak aktif), ia akan mengunci dan menahan nilai terakhir dari D yang ada sesaat sebelum E menjadi LOW.



Bonus: (0.3 poin/komponen) Berikan visualisasi atau ilustrasi masing-masing komponen. Visualisasi dibuat sendiri.

Ibu semua bikin sendiri, ada di
<https://drive.google.com/file/d/1k3TlgTbpQTQ5MEalr5PmeilP2d0lAUx/view?usp=sharing> (Pakai draw io)

Referensi:

GeeksforGeeks, "Difference between Half Adder and Full Adder," GeeksforGeeks, [Online]. Available:

<https://www.geeksforgeeks.org/gate/difference-between-half-adder-and-full-adder/>.

T. Kuphaldt, "S-R Latch," All About Circuits, [Online]. Available:
<https://www.allaboutcircuits.com/textbook/digital/chpt-10/s-r-latch/>.

T. Kuphaldt, "The Gated S-R Latch," All About Circuits, [Online]. Available:
<https://www.allaboutcircuits.com/textbook/digital/chpt-10/the-gated-s-r-latch/>

T. Kuphaldt, "The D Latch," All About Circuits, [Online]. Available:
<https://www.allaboutcircuits.com/textbook/digital/chpt-10/d-latch/>.

2. (5 poin) Beberapa CPU intel seperti Intel Core i7/i5/i3 generasi 1 sampai 8 memiliki fitur bernama speculative execution. Speculative execution adalah ketika prosesor

menebak instruksi apa yang akan dieksekusi berikutnya dan menjalankannya lebih awal sebelum dipastikan apakah tebakan itu benar untuk meningkatkan kecepatan eksekusi dan meningkatkan efektivitas pada pipelining.

Cache Replacement Policy adalah strategi yang digunakan oleh cache memory untuk memilih data mana yang harus dikeluarkan dari cache ketika cache sudah penuh dan data baru perlu dimasukkan. **Deskripsikan bagaimana seorang hacker dapat menggunakan cache replacement policy tertentu untuk melakukan timing side-channel attack agar dapat mengeksplorasi speculative execution. Jelaskan juga bagaimana cara untuk memitigasi serangan tersebut?**

Eksplorasi speculative execution dapat dilakukan untuk membocorkan data rahasia dengan menggabungkan manipulasi prediktor cabang CPU dengan serangan side-channel berbasis waktu pada cache. Proses ini, yang dikenal sebagai Spectre Variant 1 (Bounds Check Bypass), berjalan dalam beberapa tahap:

1. Membiasakan CPU dengan Pola Tertentu

Pertama, peretas "melatih" branch predictor CPU. Caranya, dia menjalankan sebuah kode berulang-ulang dengan input yang aman. Misalnya, kode yang mengecek kondisi if (indeks < 100). Dengan input yang selalu di bawah 100, CPU jadi kebiasa, dan nganggap kondisionalnya benar terus.

2. Langkah 2: Menjebak dengan Data di Luar Batas

Setelah CPU terbiasa, peretas memberikan jebakan. Dia memanggil kode yang sama, tapi kali ini dengan indeks = 500, yang jelas-jelas di luar batas. Karena udah "terlatih" dan pengen cepet, CPU secara spekulatif langsung jalanin kode di dalam blok if itu, sebelum dia sadar kalau pengecekan indeks < 100 itu sebenarnya gagal.

3. Langkah 3: Membocorkan Rahasia Lewat Cache

Di dalam blok if yang salah jalan tadi, peretas sudah menyiapkan kode "gadget". Kode ini melakukan dua hal:

- Membaca satu byte data rahasia dari memori yang seharusnya tidak boleh diakses. Anggap saja byte rahasia ini nilainya 83 (huruf 'S').
- Menggunakan nilai rahasia itu untuk mengakses sebuah array lain yang dikendalikan peretas. Contohnya: akses_array[83 * 4096].

Aksi ini menyebabkan data dari alamat akses_array yang spesifik itu (yang lokasinya ditentukan oleh nilai byte rahasia) jadi masuk ke cache CPU

4. Langkah 4: Mengukur Jejak di Cache

Walaupun akhirnya CPU sadar kalau tebakannya salah dan semua instruksi spekulatif tadi dibatalkan, ada satu hal yang tidak bisa kembali lagi: jejak data di cache masih ada.

Gampangnya gini, kalau pake analogi:

1. Bayangin kita "ngelatih" seorang pustakawan (CPU). Kita bolak-balik minta buku dari rak A. Si pustakawan jadi terbiasa dan mikir, "Ah, pasti dia minta dari rak A lagi."
2. Terus, tiba-tiba kita minta buku dari rak Z. Anggep aja si rak Z nih buku NSFW (apa aja lah bingung). Karena udah "terlatih" buat diminta tolong ngambilin buku, si pustakawan secara spekulatif (nebak-nebak) ngambilin buku dari rak Z itu sebelum sadar kalau itu rak terlarang.
3. Walaupun akhirnya dia sadar dan gajadi kasih bukunya ke kita, jejak kakinya ke rak Z itu masih ada (perubahan di cache). Nah, kita bisa "ngintip" jejak kaki itu buat tahu isi rak terlarang.

Biasanya compiler udah pake instruksi buat ngalangin spekulasi (kayak lfence) habis meriksa batas index buat mencegah CPU mengeksekusi kode lebih lanjut sampai hasil pemeriksaan dipastikan.

Teknik lain adalah *index masking*, yakni compiler mastiin bahwa setiap akses di luar batas secara otomatis dijepit ke nilai yang aman

Referensi:

J. Horn, "Reading privileged memory with a side channel," Google Project Zero, 2018. [Online]. Available: <https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html>.

3. (3 poin) Perhatikan decompiled code berikut.

```
0x0000000140007c10 <+0>: sub $0x28,%rsp  
0x0000000140007c14 <+4>: call 0x140001550 <__main>  
0x0000000140007c19 <+9>: lea 0x13e0(%rip),%rcx
```

```
0x0000000140007c20 <+16>: call 0x140001450 <printf.constprop.O>
0x0000000140007c25 <+21>: xor %eax,%eax
0x0000000140007c27 <+23>: add $0x28,%rsp
0x0000000140007c2b <+27>: ret
```

Kode di atas didapat dari menjalankan `disas main` dengan `gdb`, kode sumber originalnya adalah seperti berikut.

```
#include <stdio.h>
int main(){
    printf("Hello, World!\n");
    return 0;
}
```

Jawablah pertanyaan-pertanyaan berikut.

- a. (0.5 poin) Kenapa bisa muncul `printf.constprop.O` dan compiler tidak menggunakan `printf` biasa?

Itu soalnya hasil optimisasi compiler. Compiler GCC menghasilkan versi khusus dari sebuah fungsi ketika mendeteksi bahwa fungsi tersebut dipanggil dengan satu atau lebih argumen yang merupakan konstanta waktu kompilasi (*compile-time constant*).

Ketika kode memanggil `printf("Hello, World!\n")`, itu jelas merupakan *string* konstanta. GCC mengidentifikasi ini dan membuat "klon" dari fungsi `printf`. Klon ini dikhususkan untuk menangani *string* konstan tersebut. Di dalam fungsi kloningan ini, compiler dapat melakukan optimisasi lebih lanjut. Misalnya, daripada melalui logika penguraian format *string*, compiler dapat menggantinya dengan panggilan langsung ke fungsi yang lebih sederhana dan efisien seperti `puts` atau serangkaian `putchar`. Nama `printf.constprop.O` adalah konvensi penamaan yang digunakan oleh GCC untuk menandai fungsi yang merupakan hasil kloning dari *constant propagation*.

- b. (0.5 poin) Apakah `sin.constprop.O` sebagai pengganti fungsi `sin` pada library `math.h` mungkin dihasilkan oleh compiler? Jika tidak, sebutkan alasannya. Jika iya, sebutkan dalam kondisi apa hal tersebut mungkin terjadi.

Sangat mungkin bagi compiler untuk menghasilkan fungsi `sin.constprop.0`, tetapi hanya dalam kondisi yang spesifik. Kondisi ini terpenuhi jika fungsi `sin` dari `math.h` dipanggil dengan argumen yang merupakan konstanta waktu kompilasi, misalnya, `y = sin(0.0);`.

Dalam kasus ini, compiler dapat melakukan optimisasi yang disebut **constant folding**, di mana ekspresi dengan input konstan dievaluasi pada saat kompilasi, bukan saat *runtime*.

Compiler akan menghitung nilai `sin(0.0)` (yaitu 0.0) dan secara efektif mengubah kode menjadi `y = 0.0;`. Jika panggilan ini terjadi di dalam fungsi yang sering dipanggil, compiler mungkin membuat versi kloningan `sin.constprop.0` yang isinya hanya mengembalikan nilai 0.0 yang sudah dihitung sebelumnya, sepenuhnya menghilangkan kalkulasi trigonometri yang mahal saat program berjalan. Optimisasi ini tidak akan berlaku jika `sin` dipanggil dengan variabel yang nilainya tidak diketahui saat kompilasi, seperti `sin(x)` di mana `x` adalah input dari pengguna.

- c. (2 poin) Hasil compile seperti di atas bersifat unik ke GCC. Apabila Anda seorang developer compiler, menurut Anda apakah teknik yang dilakukan GCC pada fenomena ini sebuah fitur yang penting bagi sebuah compiler? Kemudian menurut Anda, apakah fitur tersebut berbahaya? (Elaborate your answers! Jangan hanya jawab ya atau tidak saja)

Apakah ini Fitur yang Penting?

Ya penting si, jelas ngaruh ke performa soalnya. Karena fungsinya dimodifikasi khusus, performa jadi lebih cepet. *runtime* komputasi dikurangi dengan memindahkannya ke waktu kompilasi (**constant folding**) dan memungkinkan optimisasi lebih lanjut di dalam fungsi yang dikloning, seperti **dead code elimination**.

Apakah Fitur Ini Berbahaya?

Secara umum, fitur ini gak bahaya dalam penggunaan normal, tetapi punya potensi risiko dan kelemahan. Salah satunya **Cross-Compilation**. Jika arsitektur host (tempat compiler berjalan) dan arsitektur target (tempat program akan berjalan) memiliki perilaku *floating-point* yang beda, **constant folding** dapat menghasilkan nilai pada waktu kompilasi yang

berbeda dari hasil yang akan dihitung pada *runtime* di arsitektur target. Jadinya bisa muncul bug.

Referensi:

Wikipedia, “Constant folding,” Wikipedia, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Constant_folding.

M. M., “What does the compiler optimization ‘constant propagation’ mean?,” Stack Overflow, 2011. [Online]. Available: <https://stackoverflow.com/questions/4934926/what-does-the-compiler-optimization-constant-propagation-mean>.

R., “Influencing function cloning/duplication/constant propagation in gcc,” Stack Overflow, 2013. [Online]. Available: <https://stackoverflow.com/questions/15439890/influencing-function-cloning-duplication-constant-propagation-in-gcc>.

4. (3 poin) Bayangkan cache memory seperti perpustakaan kecil di dekat ruang kelas. Jika buku yang sering dibaca disimpan di sini, sedangkan buku langka disimpan di perpustakaan utama
 - a. Jelaskan prinsip "locality of reference" menggunakan analogi ini.

Prinsip "locality of reference" menyatakan bahwa CPU cenderung mengakses lokasi memori yang sama atau yang berdekatan secara berulang kali dalam periode waktu yang singkat. Dengan menggunakan analogi perpustakaan:

- **Temporal Locality (Lokalitas Temporal):** Jika seorang "murid" (CPU) mengambil sebuah buku (misalnya, kode fungsi `printf`), kemungkinan besar ia akan membutuhkan buku yang sama itu lagi dalam waktu dekat. Menyimpan "buku yang sering dibaca" ini di meja kecil di dalam kelas (cache CPU) jauh lebih cepat daripada harus berjalan kembali ke perpustakaan utama (RAM) setiap kali membutuhkannya
- **Spatial Locality (Lokalitas Spasial):** Ketika murid mengambil buku dan membaca halaman 50, sangat mungkin ia akan segera membutuhkan halaman 51. Oleh karena itu, saat mengambil buku

dari perpustakaan utama, akan lebih efisien untuk mengambil seluruh bab (sebuah *cache line*) daripada hanya satu halaman. Ini analog dengan CPU yang memuat satu blok alamat memori yang berdekatan ke dalam cache sekaligus, mengantisipasi bahwa data di dekatnya akan segera dibutuhkan.

- b. Mengapa "cache miss" bisa membuat "murid" (CPU) harus berjalan jauh ke perpustakaan utama?

Terjadi pas murid butuh buku yang ternyata gaada di meja. Murid tsb terpaksa harus jalan ke rak buku (akses RAM), yang pastinya lebih lama. Proses nunggu ini bikin CPU "bengong" dan performa turun.

- c. Bagaimana strategi "prefetching" bisa membantu murid tidak "terjebak" di tengah diskusi kelas?

Ini kayak punya temen yang nemenin murid tsb, terus dia ngeliatin temennya baca. Pas murid selesai baca satu bab, dia udah inisiatif ngambilin bab selanjutnya buat jadi bahan diskusi dan naruh di mejanya. Jadi, pas murid butuh, barangnya udah siap.

Referensi

TechGeekBuzz, "Locality of Reference in Computer Architecture," TechGeekBuzz, 2023. [Online]. Available: <https://www.techgeekbuzz.com/blog/locality-of-reference/>.

5. (3 poin) Misal ada sebuah arsitektur komputer yang memiliki 8 buah register secara fisik (*let's say P0-P7*), tetapi ternyata arsitektur tersebut harus bisa diakses oleh programmer/user (*let's say R0-R15*).
- (2 poin) Jelaskan cara/mekanisme yang memungkinkan agar arsitektur tersebut dapat diimplementasikan
 - (0.5 poin) Apa dampak pada pipeline hazard (specifically data hazards)
 - (0.5 poin) Apa dampak pada optimasi kompilasi

a. Mekanisme yang memungkinkan implementasi ini adalah **Register Renaming**. Prosesor memelihara sebuah tabel pemetaan (mapping table) atau alias table yang melacak asosiasi antara register arsitektural dan register fisik. Prosesnya berjalan sebagai berikut:

1. Ketika sebuah instruksi yang akan menulis hasil ke register arsitektural (misalnya, ADD R5, R1, R2) memasuki tahap decode/ rename pada pipeline, CPU mengalokasikan sebuah register fisik yang sedang tidak terpakai dari pool.
2. Tabel pemetaan diperbarui untuk mencatat perujukan.
3. Setiap instruksi berikutnya dalam aliran program yang perlu membaca R5 akan diarahkan oleh perangkat keras untuk membaca dari register fisik.
4. Jika instruksi lain di kemudian hari juga menulis ke R5 (misalnya, MUL R5, R3, R4), CPU akan mengalokasikan register fisik baru lainnya dan memperbarui pemetaan R5.

Dengan cara ini, beberapa "versi" dari register arsitektural R5 dapat ada secara bersamaan di dalam register fisik yang berbeda. Hal ini secara efektif memecah dependensi data palsu yang timbul dari penggunaan kembali nama register yang sama.

b. Register renaming secara langsung mengatasi **false data dependencies**, yang merupakan sumber utama dari *data hazards* struktural dalam *pipeline*. Secara spesifik, ia menyelesaikan:

- **Write-After-Read (WAR) Hazard (Anti-dependency):** Terjadi ketika sebuah instruksi mencoba menulis ke sebuah register sebelum instruksi sebelumnya selesai membacanya (misalnya, I1: read R5, diikuti I2: write R5). Tanpa renaming, I2 harus menunggu I1 selesai untuk menghindari penimpaan nilai yang masih dibutuhkan. Dengan renaming, I2 menulis ke register fisik baru yang berbeda, sehingga I1 dan I2 dapat dieksekusi secara paralel atau bahkan di luar urutan (*out-of-order*).
- **Write-After-Write (WAW) Hazard (Output dependency):** Terjadi ketika dua instruksi menulis ke register yang sama (misalnya, I1: write R5, diikuti I2: write R5). Tanpa renaming, keduanya harus dieksekusi secara berurutan untuk memastikan hasil akhir yang benar. Dengan renaming, mereka menulis ke register fisik yang berbeda, sehingga urutan eksekusinya dapat diubah.

Penting untuk dicatat bahwa register renaming **tidak** menyelesaikan dependensi data sejati, yaitu **Read-After-Write (RAW) Hazard (Flow dependency)**. Jika I₂ perlu membaca hasil yang ditulis oleh I₁, I₂ tetap harus menunggu I₁ selesai. Namun, dengan menghilangkan hazard palsu, register renaming secara drastis meningkatkan peluang untuk **out-of-order execution**, yang merupakan kunci untuk mencapai *Instruction-Level Parallelism* (ILP) yang tinggi dan memaksimalkan throughput pipeline.

c. Register renaming secara signifikan **menyederhanakan tugas register allocator** pada compiler. Tanpa renaming, compiler harus sangat berhati-hati dalam mengalokasikan register arsitektural yang terbatas untuk menghindari terciptanya *false dependencies* yang akan menyebabkan *stall* pada pipeline. Ini adalah masalah optimisasi yang sangat kompleks.

Referensi: Wikipedia, “Register renaming,” Wikipedia, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Register_renaming.

6. (2.5 poin) Diberikan struktur data sebagai berikut pada mesin IA32.

<pre>struct hawk{ union sybau a; char b[3]; int c; };</pre>	<pre>struct tuah{ char d; int e[4]; struct hawk* f; struct tuah* g; };</pre>	<pre>union sybau{ char h; struct hawk* i; struct tuah* j; };</pre>
---	--	--

- a. (1 poin) Tentukan nomor byte *alignment* untuk atribut pada masing-masing struktur. Lalu tentukan ukuran byte total dari masing-masing struktur. Jelaskan secara singkat.

-> Untuk IA32, ukuran tipe data yang relevan adalah: **char = 1 byte, int = 4 byte, dan pointer (*) = 4 byte**.

- i. (0.3 poin) struct hawk

- **Alignment:** Anggota dengan alignment terbesar adalah **union sybau a** dan **int c**, keduanya memerlukan *alignment* 4 byte. Jadi, *alignment* struct hawk adalah **4 byte**.
- **Perhitungan Ukuran:**

- union sybau a: offset=0, size=4.
- char b: offset=4, size=3.
- int c: Memerlukan *alignment* 4 byte. Offset berikutnya yang tersedia adalah 7 (4+3), yang bukan kelipatan 4. Oleh karena itu, compiler menambahkan **1 byte padding**. Offset c menjadi 8. size=4.
- Ukuran total saat ini: $8 + 4 = 12$.
- Ukuran total struktur (12) harus merupakan kelipatan dari *alignment*-nya (4). Karena 12 adalah kelipatan 4, tidak ada *padding* tambahan di akhir.
- Ukuran total struct hawk adalah **12 byte**.

ii. (0.3 poin) struct tuah

- **Alignment:** Anggota dengan *alignment* terbesar adalah int dan pointer, keduanya memerlukan *alignment* 4 byte. Jadi, *alignment* struct tuah adalah **4 byte**.
- **Perhitungan Ukuran:**
 - char d: offset=0, size=1.
 - int e: Memerlukan *alignment* 4 byte. Offset berikutnya yang tersedia adalah 1, yang bukan kelipatan 4. Compiler menambahkan **3 byte padding**. Offset e menjadi 4. size=16 (4 * 4 byte).
 - struct hawk* f: offset=20 (4+16), size=4.
 - struct tuah* g: offset=24 (20+4), size=4.
 - Ukuran total saat ini: $24 + 4 = 28$.
 - Ukuran total struktur (28) harus merupakan kelipatan dari *alignment*-nya (4). Karena 28 adalah kelipatan 4, tidak ada *padding* tambahan di akhir.
- Ukuran total struct tuah adalah **28 byte**.

iii. (0.4 poin) union sybau

- **Alignment:** Alignment sebuah union adalah alignment dari anggota terbesarnya. Di sini, anggota terbesarnya adalah pointer dengan alignment 4 byte. Jadi, alignment union sybau adalah **4 byte**.
- **Ukuran:** Ukuran sebuah union adalah ukuran dari anggota terbesarnya. Dalam hal ini, ukurannya adalah **4 byte**.

b. (1.5 poin) Diberikan hasil kompilasi dari dua buah fungsi sebagai berikut.

proc1:

```
pushl %ebp  
movl %esp, %ebp  
movl 8(%ebp), %eax  
movl 12(%eax), %eax  
movl %ebp, %esp  
popl %ebp  
ret
```

proc2:

```
pushl %ebp  
movl %esp, %ebp  
movl 8(%ebp), %eax  
movl (%eax), %eax  
movl 24(%eax), %eax  
movl 20(%eax), %eax  
movzbl 6(%eax), %eax  
movl %ebp, %esp  
popl %ebp  
ret
```

Tentukan apa yang dikembalikan oleh masing-masing fungsi. Jelaskan secara singkat.

i. (0.25 poin) proc1

```
int proc1(struct tuah *x){ return x->e[3]; }
```

// movl 8(%ebp), %eax => ambil argumen pertama fungsi, yakni x
// movl 12(%eax), %eax => dereference x, ambil offset ke 12. Karena
0-4 char d, 4-16 e[4], maka yang diambil adalah 12-16 -> e[3]
(elemen terakhir)

ii. (1.25 poin) proc2

```
int proc2(struct hawk *x){ return x->a.j->g->f->b; }
```

```
/*
```

- movl 8(%ebp), %eax: Memuat argumen pertama, struct

hawk *x, ke dalam %eax.

- movl (%eax), %eax: Mengakses offset 0 dari x, yaitu x->a. Karena instruksi berikutnya adalah dereferensi, union a haruslah berisi pointer. Berdasarkan struktur, ini bisa i atau j. Mari kita ikuti alurnya.
- movl 24(%eax), %eax: Mengakses offset 24 dari pointeryang baru dimuat. Jika kita melihat struct tuah, offset 24 adalah anggota g (struct tuah*). Jadi, langkah sebelumnya pasti mengakses x->a.j. Langkah ini adalah x->a.j->g.
- movl 20(%eax), %eax: Mengakses offset 20 dari pointer struct tuah* yang baru. Ini adalah anggota f (struct hawk*). Jadi, ini adalah x->a.j->g->f.
- movzbl 6(%eax), %eax: Mengakses offset 6 dari pointer struct hawk* yang baru. Berdasarkan struct hawk, offset 6 adalah byte kedua dari array b (karena b dimulai di offset 4, maka b di 4, b di 5, b di 6). Instruksi movzbl memuat satu byte dan memperluasnya menjadi 32-bit dengan mengisi bit atas dengan nol. Ini mengakses x->a.j->g->f->b.

*/

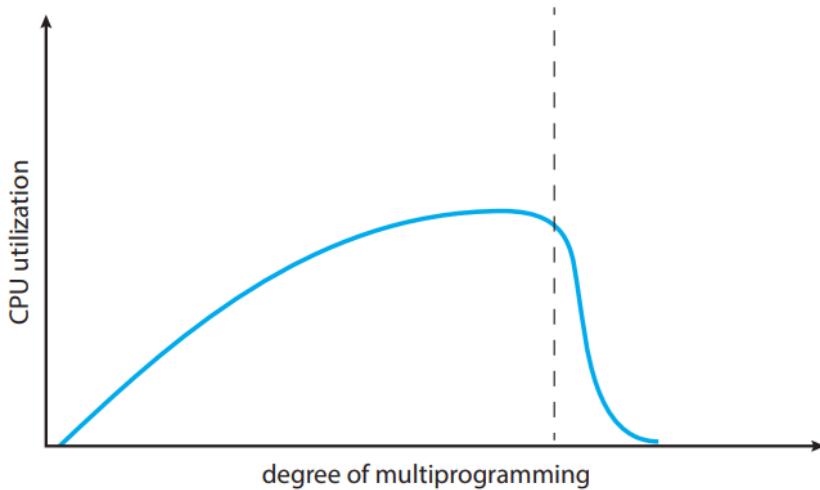
7. (1 poin) Kita mengenal beberapa cara untuk menyimpan data ada little endian dan big endian. Mengapa beberapa device lebih memilih untuk menyimpan data secara little endian?

- Di little-endian, ngubah tipe data dari yang besar ke yang kecil lebih gampang. Misal kita punya angka 32-bit (integer) dan kita cuma mau baca 16-bit pertamanya (short). Tinggal ambil aja dua byte pertama dari alamat memori yang sama.
- Dulu, mikroprosesor awal dari Intel (kayak 8080 dan 8086 yang super populer) udah make format little-endian. Karena arsitektur Intel ini kemudian mendominasi pasar komputer pribadi, semua ekosistem—mulai dari software, compiler, sampai sistem operasi—akhirnya dibangun mengikuti standar ini.

II. Sistem Operasi



1. (1 poin) Perhatikan gambar berikut.



Mengapa terdapat penurunan drastis kinerja prosesor ketika derajat multiprogramming terlalu tinggi pada sistem? Apakah terdapat hubungan dari fenomena tersebut dengan salah satu konsep manajemen sumber daya utama pada sistem operasi? Jelaskan.

Thrashing -> kondisi di mana sistem menghabiskan lebih banyak waktu untuk memindahkan halaman data antara memori utama (RAM) dan penyimpanan sekunder (*disk*) daripada melakukan pekerjaan produktif.

Berkaitan erat dengan **manajemen memori virtual**.

- Setiap proses membutuhkan sejumlah halaman memori untuk berjalan secara efisien, yang dikenal sebagai *working set*.
- Pas derajat *multiprogramming* terlalu tinggi, total *working set* dari semua proses yang aktif melebihi kapasitas RAM fisik yang tersedia.
- Jadinya, sistem operasi harus terus-menerus menukar halaman (*swapping*) ke dan dari *disk* untuk mengakomodasi semua proses.
- Karena akses *disk* ribuan kali lebih lambat daripada akses RAM, CPU sering kali menganggur, menunggu operasi I/O paging selesai.
- **Hal ini ngebuat utilitas CPU yang tinggi (karena OS sibuk melakukan paging) tetapi throughput sistem (pekerjaan aktual yang diselesaikan) sangat rendah.** (Ini intinya)

Terus juga bisa makin parah akibat *feedback loop*. Karena keliatannya gabut, OS ngira CPU perlu kerja tambahan, jadi multiprogrammingnya dinaikin. Akhirnya malah makin parah masalahnya.

Referensi: PPT Pak Monte :D

2. (1.5 poin) Jawablah pertanyaan-pertanyaan berikut.

a. (0.5 poin) Jelaskan konsep *virtualization* dan cara kerjanya.

Virtualisasi adalah teknologi yang memungkinkan pembuatan versi virtual dari sebuah komputer fisik, lengkap dengan hardware virtual (CPU, RAM, storage). Di atas VM ini, guest OS yang lengkap dapat diinstal dan dijalankan.

Cara Kerja: Komponen inti dari virtualisasi adalah **hypervisor** atau *Virtual Machine Monitor* (VMM). Hypervisor adalah lapisan perangkat lunak yang berjalan di atas perangkat keras fisik (*host*). Ia bertugas mengabstraksikan sumber daya fisik dan menyajikannya sebagai perangkat keras virtual

kepada satu atau lebih **VM**. Hypervisor ada dua tipe, I dan II. Kalau I, langsung diinstal di hardware. Kalau II, butuh Host OS.

Referensi: Dijelaskan temen STI (Aidan, rispek)

- b. **(0.5 poin)** Jelaskan konsep **containerization** dan cara kerjanya.

Containerization adalah bentuk virtualisasi tingkat sistem operasi yang lebih ringan. Sebuah aplikasi beserta semua dependensinya dikemas ke dalam satu unit terisolasi yang disebut **container**.

Cara Kerja: Containerization menggunakan **container engine** (seperti Docker) yang berjalan di atas sistem operasi host. Semua container yang berjalan di host yang sama **berbagi kernel OS host yang sama**.

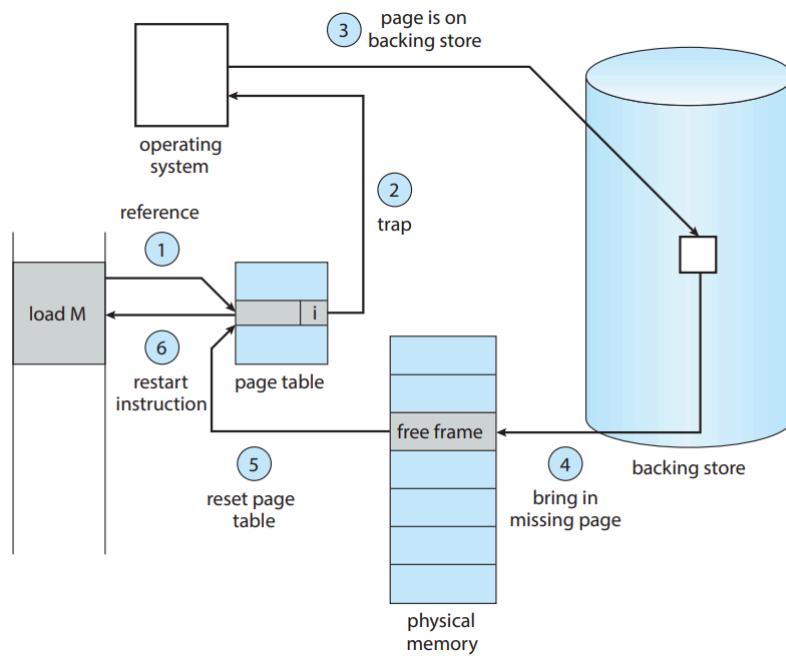
Referensi: Dijelaskan temen STI (Aidan, rispek)

- c. **(0.5 poin)** Bandingkan kedua konsep tersebut. Apa perbedaannya? Apa persamaannya? Apa kelebihan dan kekurangan dari menggunakan masing-masing metode?

Fitur	Virtualisasi (VM)	Containerization (Container)
Tingkat Isolasi	Penuh (tingkat perangkat keras). Setiap VM memiliki kernel sendiri. Sangat aman.	Proses. Berbagi kernel OS host. Kurang aman jika kernel host terganggu.
Overhead Sumber Daya	Tinggi. Setiap VM menyertakan OS tamu lengkap.	Rendah. Hanya mengemas aplikasi dan dependensi.
Waktu Startup	Lambat (menit). Perlu me-boot OS lengkap.	Cepat (detik atau milidetik). Hanya memulai proses.
Ukuran	Besar (beberapa Gigabyte).	Kecil (beberapa Megabyte).

Portabilitas	Kurang portabel. Terikat pada <i>hypervisor</i> .	Sangat portabel. Dapat berjalan di mana saja <i>container engine</i> terinstal.
OS Tamu	Dapat menjalankan OS yang berbeda dari host (misalnya, Windows di host Linux).	Harus menggunakan kernel OS yang sama dengan host (misalnya, container Linux di host Linux).
Kasus Penggunaan	Isolasi keamanan tinggi, menjalankan aplikasi lawas, lingkungan multi-OS.	Arsitektur <i>microservices</i> , aplikasi <i>cloud-native</i> , CI/CD, pengembangan perangkat lunak.

3. (1.5 poin) Perhatikan gambar berikut.



Jelaskan bagaimana *virtual memory* bekerja berdasarkan diagram tersebut.

Proses ini bekerja melalui interaksi antara perangkat keras dan perangkat lunak sebagai berikut:

- **CPU Menghasilkan Alamat Virtual:** Ketika CPU menjalankan sebuah proses, ia menghasilkan alamat memori yang disebut **alamat virtual** untuk mengakses data atau instruksi. Proses tidak mengetahui alamat fisik yang sebenarnya.
- **Penerjemahan oleh MMU:** Alamat virtual ini dikirim ke unit perangkat keras khusus yang disebut **Memory Management Unit (MMU)**. MMU bertugas menerjemahkan VA menjadi alamat fisik (PA). Untuk melakukan ini, MMU menggunakan sebuah struktur data yang dikelola oleh sistem operasi yang disebut **Page Table**.
- **Penggunaan Page Table:** VA dibagi menjadi dua bagian: nomor halaman virtual dan offset. Nomor halaman digunakan sebagai indeks untuk mencari entri yang sesuai di dalam **Page Table**.

Kemudian, untuk setiap angka pada diagram:

1. **Kasus 1: Page Hit:** Jika entri **Page Table** untuk VPN tersebut valid (ditandai dengan **valid bit** bernilai 1), itu berarti halaman yang dicari ada di dalam memori fisik (RAM). Entri tersebut berisi nomor bingkai fisik di mana halaman itu disimpan. MMU menggabungkannya dengan offset dari VA untuk membentuk alamat fisik akhir. Data kemudian diambil dari alamat fisik tersebut di RAM.

Kasus 2: Page Fault (Page Miss): Jika entri **Page Table** tidak valid (**valid bit** bernilai 0), ini berarti halaman yang dibutuhkan tidak ada di RAM. MMU akan memicu sebuah **page fault**, yang merupakan sebuah *hardware exception* (interupsi). Interupsi ini menghentikan proses sejenak dan mentransfer kontrol ke sistem operasi.

2. **Penanganan Page Fault oleh OS:** Sistem operasi mengambil alih.
3. **Pencarian Page:** OS akan mencari halaman yang dibutuhkan di area penyimpanan sekunder yang disebut *swap space* atau *page file*.
4. **Pembawaan Page:** Setelah ditemukan, OS akan memuat halaman tersebut dari disk ke sebuah bingkai kosong di RAM. Jika tidak ada bingkai yang kosong, OS akan menggunakan **algoritma penggantian halaman** (seperti LRU - Least Recently Used) untuk memilih halaman korban yang akan

dikeluarkan dari RAM (jika halaman korban telah dimodifikasi, ia harus ditulis kembali ke disk terlebih dahulu).

5. **Pembaruan Page Table:** Setelah halaman dimuat ke RAM, OS memperbarui *Page Table* dengan PFN yang baru dan mengatur *valid bit* menjadi 1. Kemudian, kontrol dikembalikan ke proses yang terinterupsi.
6. **Eksekusi Ulang:** Instruksi yang menyebabkan *fault* akan dieksekusi ulang. Kali ini, MMU akan menemukan entri yang valid di *Page Table* dan proses akan berlanjut sebagai *page hit*.

Referensi

GeeksforGeeks, “Virtual Memory in Operating System,” GeeksforGeeks, 2025. [Online]. Available:

<https://www.geeksforgeeks.org/operating-systems/virtual-memory-in-operating-system/>.

Buku OS Abraham ([Operating System Concepts, 10th ed](#))

4. (6 poin) Salah satu permasalahan yang umum ditemukan pada lingkup sistem operasi adalah masalah sinkronisasi, yaitu bagaimana banyak proses memastikan bahwa data yang digunakan secara bersamaan selalu konsisten dan benar. Salah satu metode yang telah dirancang untuk mengatasi masalah tersebut berupa Peterson’s Solution, dengan algoritma secara kasar sebagai berikut:

```
int turn;
boolean flag[2];

while (true){
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
        /* critical section */
    flag[i] = false;
    /* remainder section */
}
```

Contoh kasus terkait Peterson’s Solution umumnya mencakup dua buah proses, P_1 dan P_2 , yang keduanya menjalankan kode seperti di atas. Variabel i menandakan

nomor dari proses yang menjalankan, sedangkan variabel `j` menandakan nomor dari proses lawan. (Read: [Operating System Concepts, 10th ed.](#))

- a. (1 poin) Jelaskan bagaimana algoritma Peterson's Solution bekerja.

Cara kerjanya adalah sebagai berikut untuk proses Pi:

1. Pi menyatakan niatnya untuk masuk ke *critical section* dengan mengatur benderanya sendiri menjadi true: `flag[i] = true;`
2. Pi kemudian memberikan giliran kepada proses lain (Pj): `turn = j;`. Hal ini jadi langkah kunci untuk memecahkan kebuntuan jika kedua proses mencoba masuk secara bersamaan.
3. Pi masuk ke dalam *loop* tunggu aktif (*busy-wait*): `while (flag[j] && turn == j);`. Proses Pi akan terus menunggu selama Pj juga ingin masuk (`flag[j] == true`) **DAN** saat ini adalah giliran Pj (`turn == j`). Pi hanya bisa keluar dari *loop* dan masuk ke *critical section* jika salah satu dari dua kondisi ini tidak terpenuhi: entah Pj tidak tertarik (`flag[j]` adalah `false`), atau Pj tertarik tetapi bukan gilirannya (karena `turn == i`).

- b. (0.5 poin) Apa peran dari variabel `turn` dan `flag` pada algoritma? Mengapa diperlukan kedua variabel tersebut agar solusi bekerja dengan benar? Jelaskan.

Kedua variabel, `turn` dan `flag`, sangat penting dan saling melengkapi untuk memastikan solusi ini bekerja dengan benar.

- **Variabel flag:** `flag[i]` menunjukkan **niat** atau keinginan proses Pi untuk masuk ke *critical section*. Tanpa `flag`, tidak ada cara bagi satu proses untuk mengetahui apakah proses lain sedang tertarik untuk masuk.
- **Variabel turn:** `turn` berfungsi sebagai **pemecah kebuntuan**. Ia digunakan untuk menentukan proses mana yang mendapatkan prioritas jika kedua proses menyatakan niat untuk masuk pada saat yang bersamaan.

Mengapa keduanya diperlukan?

- Jika hanya menggunakan `flag`, bisa terjadi *race condition*. Kedua proses (P0 dan P1) bisa saja mengatur `flag` mereka menjadi `true` secara bersamaan. Kemudian, keduanya memeriksa `flag` lawannya, menemukan nilainya `true`, dan keduanya masuk ke *critical section*, sehingga melanggar *mutual exclusion*.
- Jika hanya menggunakan `turn`, sistem akan dipaksa ke dalam **alternasi ketat**. Misalnya, jika `turn` diatur ke 0, P0 masuk. Setelah selesai, ia harus mengatur `turn` ke 1. Sekarang, P0 tidak bisa masuk lagi sampai P1 masuk dan mengatur `turn` kembali ke 0. Jika P1 tidak perlu masuk ke *critical section*, P0 akan diblokir tanpa batas, yang melanggar syarat *progress*.

- c. (1.5 poin) Buktikan bahwa Peterson's Solution sudah merupakan solusi yang benar untuk mengatasi *critical section problem*.

Untuk dikatakan mengatasi *critical section problem*, sebuah algoritma sinkronisasi harus memenuhi ketiga syarat berikut:

- Mutual Exclusion : Hanya satu proses/thread yang mampu mengakses *critical section*.
- Progress : Kalau gaada proses yang dieksekusi pada *critical section* dan beberapa proses ingin memasuki *critical section*, cuma proses yang tidak dieksekusi pada *remainder section* yang dapat berpartisipasi dalam menentukan proses mana yang akan memasuki *critical section* berikutnya, dan pemilihan ini tidak dapat ditunda.
- Bounded Waiting: Harus ada batasan berapa kali proses lain diizinkan memasuki bagian kritis sebelum proses yang menunggu mendapat giliran (Hindari Starvation!!)

Kalau dilihat lebih lanjut:

1. **Mutual Exclusion:** Dibuktikan dengan kontradiksi. Asumsikan kedua proses, P0 dan P1, berada di dalam *critical section* mereka secara bersamaan.

- o Jika P0 berada di dalam *critical section*, maka kondisi while-nya (`flag && turn == 1`) pasti salah.
 - o Jika P1 berada di dalam *critical section*, maka kondisi while-nya (`flag && turn == 0`) pasti salah.
 - o Karena kedua proses berada di dalam *critical section*, maka `flag` dan `flag` keduanya harus true.
 - o Agar kondisi while P0 salah (dengan `flag` true), maka `turn == 1` harus salah, yang berarti `turn` harus 0.
 - o Agar kondisi while P1 salah (dengan `flag` true), maka `turn == 0` harus salah, yang berarti `turn` harus 1.
 - o Ini mengarah pada kontradiksi bahwa `turn` harus bernilai 0 dan 1 secara bersamaan, yang tidak mungkin. Oleh karena itu, asumsi awal salah, dan *mutual exclusion* terjamin.
2. **Progress:** Jika tidak ada proses yang berada di *critical section* dan ada proses yang ingin masuk, maka proses tersebut akan diizinkan masuk. Jika Pj tidak tertarik, `flag[j]` akan false, sehingga Pi dapat langsung masuk. Jika keduanya tertarik, salah satu pasti akan diizinkan masuk karena variabel `turn` akan memecah kebuntuan.
3. **Bounded Waiting:** Sebuah proses ga akan menunggu selamanya. Proses Pi akan menunggu paling lama untuk satu giliran dari proses Pj. Setelah Pj keluar dari *critical section*-nya, ia akan mengatur `flag[j]` menjadi false, yang memungkinkan Pi untuk melanjutkan. Jika Pj mencoba masuk lagi, ia akan mengatur `turn = i`, yang secara eksplisit memberikan giliran kepada Pi, sehingga menjamin Pi tidak akan kelaparan (starvation).

- d. (0.5 poin) Walaupun sudah merupakan solusi yang benar secara teoritis, Peterson's Solution umumnya tidak digunakan pada implementasi sistem operasi modern. Mengapa demikian? Apa kelemahan dari solusi ini yang menyebabkannya tidak dapat digunakan pada sistem yang nyata? Jelaskan.

Alasannya adalah karena ia bergantung pada model **sequential consistency** memori, di mana operasi tulis ke memori oleh satu prosesor

terlihat oleh prosesor lain dalam urutan yang sama seperti saat ditulis dalam program.

Arsitektur komputer modern tidak menjamin ini. Untuk optimisasi performa, baik *compiler* maupun prosesor dapat melakukan **memory reordering**. Misalnya, *compiler* bisa saja menukar urutan `flag[i] = true;` dan `turn = j;`. Prosesor dengan *out-of-order execution* juga bisa melakukan hal yang sama. Pengurutan ulang ini dapat merusak logika algoritma dan menyebabkan kegagalan sinkronisasi, di mana kedua proses bisa masuk ke *critical section* secara bersamaan.

- e. (2 poin) Untuk mengatasi kelemahan dari Peterson's Solution, telah dirancang beberapa metode alternatif untuk mengatasi masalah sinkronisasi. Jelaskan cara kerja dan cara pemakaian dari kedua metode alternatif berikut.

- i. (1 poin) `test_and_set()`

Instruksi ini melakukan dua hal: membaca nilai asli dari sebuah lokasi memori (sebuah *lock*), kemudian menulis nilai `true` (atau `1`) ke lokasi tersebut, dan akhirnya mengembalikan nilai asli yang dibacanya.

Pemakaian: Sebuah *lock* sederhana dapat diimplementasikan dengan `while (test_and_set(&lock) == true);`. Proses pertama yang mengeksekusi ini akan mendapatkan `false` (nilai awal *lock*), mengatur *lock* menjadi `true`, dan masuk ke *critical section*. Proses lain yang mencoba akan mendapatkan `true` dan terjebak dalam loop tunggu. Untuk membuka kunci (*unlock*), proses hanya perlu mengatur `lock = false;`

- ii. (1 poin) `compare_and_swap()`

Instruksi ini menerima tiga argumen: alamat memori (`reg`), nilai yang diharapkan (`expected_val`), dan nilai baru (`new_val`). Secara atomik, ia membandingkan isi `reg` dengan `expected_val`. Jika sama, ia memperbarui `reg` dengan `new_val` dan mengembalikan status

berhasil. Jika tidak sama (artinya proses lain telah mengubahnya), ia tidak melakukan apa-apa dan mengembalikan status gagal.

Pemakaian: `compare_and_swap()` sangat kuat dan digunakan untuk membangun struktur data *lock-free* yang canggih. Misalnya, untuk menambahkan elemen ke *linked list* tanpa *lock*, sebuah *thread* dapat membaca *head* saat ini, membuat *node* baru yang menunjuk ke *head* tersebut, lalu menggunakan `compare_and_swap` untuk mengganti *head* dengan *node* baru, **hanya jika** *head* belum diubah oleh *thread* lain sementara itu.

- f. (0.5 poin) Mengapa `test_and_set()` dan `compare_and_swap()` dapat digunakan sebagai solusi sinkronisasi pada sistem operasi nyata, sedangkan Peterson's Solution tidak?
- Operasinya dijamin **atomik oleh perangkat keras**. Jaminan ini menghilangkan masalah *memory reordering* yang mengganggu solusi perangkat lunak seperti Peterson's pada sistem multiprosesor modern.
5. (2.5 poin) Clone repository [glibc](#) dan masuklah ke folder `libio`. Jelaskan cara kerja fungsi `putchar` yang kita kenal berdasarkan kode yang Anda temui di sana. Sertakan juga penjelasan terkait beberapa *file*, *macro*, fungsi, atau *struct* yang relevan (berikan code snippet jika diperlukan untuk mempermudah penjelasan).

Cara kerja `putchar` adalah sebagai berikut:

```
#include "libioP.h"
#include "stdio.h"

#undef putchar

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;
}

#ifndef defined weak_alias && !defined _IO_MTSAFE_IO
#define putchar_unlocked
weak_alias (putchar, putchar_unlocked)
#endif
```

1. Proses:

```
_IO_acquire_lock(_IO_stdout);
```

- Sebelum sebuah *thread* bisa mencetak karakter, ia harus "mengunci" akses ke layar. *Thread* lain yang ingin mencetak harus antre dan menunggu.

```
result = _IO_putc_unlocked(c, _IO_stdout);
```

- Ini adalah fungsi inti yang benar-benar menuliskan karakter `c` ke *buffer* layar. Nama `_unlock` di belakangnya adalah petunjuk penting: fungsi ini *tidak* punya mekanisme pengaman sendiri. Ia berasumsi "kunci" sudah dipegang oleh pemanggilnya.

```
_IO_release_lock(_IO_stdout);
```

- Setelah selesai mencetak satu karakter, *thread* tersebut akan melepaskan kuncinya.

2. Struktur dan File Relevan:

- `libio/libio.h`: Mendefinisikan struktur inti `_IO_FILE` dan makro-makro terkait.
- `libio/putc.c` (atau file serupa): Berisi implementasi `_IO_putc`.
- `stdio.h`: Menyediakan makro `putchar` yang dilihat oleh pengguna.
- `struct _IO_FILE`: Struktur data krusial yang menyimpan semua keadaan sebuah aliran data, termasuk penunjuk ke *buffer*, penunjuk posisi, dan penunjuk fungsi untuk berbagai operasi I/O seperti `_IO_OVERFLOW`.

6. (2.5 poin) Jelaskan apa saja tahapan-tahapan dalam kernel exploitation, mengapa penyerang memerlukan multiple stages untuk mencapai tujuan mereka, dan bagaimana mereka mempertahankan akses setelah mendapatkan privilege kernel. Apa perbedaan antara *initial exploitation*, *privilege escalation*, dan *persistence mechanisms*?

1. **Initial Exploitation (Eksloitasi Awal):** Penyerang, dari program dengan hak istimewa rendah di user-space, memicu kerentanan kernel. Contoh kerentanan termasuk buffer overflow, use-after-free, atau race condition. Tujuan langsung dari tahap ini bukanlah untuk mengambil alih sistem, melainkan untuk mendapatkan **kemampuan primitif** pertama. Kemampuan ini bisa berupa:
 - **Information Leak:** Membocorkan alamat memori kernel untuk mengalahkan proteksi seperti KASLR (Kernel Address Space Layout Randomization), yang mengacak lokasi kode kernel di memori.
 - **Limited Read/Write:** Mendapatkan kemampuan untuk membaca atau menulis ke area memori kernel yang terbatas dan terkontrol.
2. **Privilege Escalation (Eskalasi Hak Istimewa):** Setelah mendapatkan kemampuan primitif, penyerang menggunakan其nya untuk mencapai tujuan akhir: eksekusi kode sewenang-wenang dalam mode kernel atau memodifikasi struktur data kernel yang kritis untuk memberikan dirinya hak akses tertinggi (root/SYSTEM).
 - **Code Execution:** Teknik seperti Return-Oriented Programming (ROP) digunakan untuk merangkai potongan-potongan kode yang sudah ada di kernel ("gadget") untuk melakukan tindakan jahat. Ini dilakukan untuk melewati proteksi seperti SMEP/SMAP, yang mencegah kernel mengeksekusi kode dari user-space secara langsung.
 - **Data-Only Attack:** Pendekatan yang lebih modern dan seringkali lebih sulit dideteksi adalah dengan tidak mengeksekusi kode sama sekali. Sebaliknya, penyerang menggunakan kemampuan tulis primitifnya untuk secara langsung memodifikasi struktur data kredensial prosesnya sendiri (`struct cred` di Linux) di memori, mengubah User ID-nya menjadi 0 (root). Target populer lainnya adalah `modprobe_path`, yang jika diubah dapat digunakan untuk mengeksekusi skrip jahat dengan hak akses root.

3. **Persistence (Mempertahankan Akses):** Akses root yang diperoleh melalui eskalasi hak istimewa seringkali bersifat sementara dan akan hilang saat sistem di-reboot. Untuk mempertahankan akses, penyerang perlu menginstal mekanisme *backdoor* yang permanen. **Contoh:** Menginstal rootkit, mengganti biner sistem penting dengan versi yang telah ditrojan, membuat akun pengguna tersembunyi, atau memodifikasi skrip startup sistem.

Initial exploitation gunanya untuk mendapatkan **pijakan awal**. *Privilege escalation* adalah gunanya untuk mengubah pijakan tersebut menjadi **kontrol penuh**. **Persistence** adalah tentang memastikan **kontrol tersebut tidak hilang**.

7. (2.5 poin) Seorang pengguna memiliki laptop yang menjalankan Linux (dengan file system ext4). Ia kemudian menyambungkan sebuah USB drive yang diformat di komputer Windows (dengan file system NTFS). Ajaibnya, ia bisa langsung membuka, menyalin, dan menyimpan file di USB drive tersebut seolah-olah itu adalah bagian dari sistemnya sendiri.
- a. (1 poin) Jelaskan komponen abstraksi utama di dalam kernel sistem operasi yang memungkinkan ini terjadi

Komponen abstraksi utama yang memungkinkan hal ini terjadi adalah **Virtual File System (VFS)**, yang juga dikenal sebagai *Virtual Filesystem Switch*.

VFS menyediakan sebuah antarmuka (API) yang seragam dan tunggal untuk semua panggilan sistem yang berhubungan dengan file, seperti `open()`, `read()`, `write()`, dan `close()`.

Ketika sebuah aplikasi pengguna melakukan operasi file, komunikasinya dilakukan dengan VFS. VFS kemudian bertindak sebagai sebuah **dispatcher**, memeriksa tipe sistem file dari target operasi dan meneruskan panggilan tersebut ke driver sistem file konkret yang sesuai yang telah terdaftar di kernel. Dengan demikian, VFS menyembunyikan detail implementasi dari setiap sistem file, sehingga bagi aplikasi, seluruh ruang file terlihat sebagai satu kesatuan.

- b. (1.5 poin) Buatlah sebuah analogi yang menarik untuk menjelaskan cara kerja komponen ini.

Bayangan **perusahaan multinasional besar** yang memiliki banyak departemen yang tersebar di seluruh dunia, dan setiap departemen berbicara dalam bahasa yang berbeda (misalnya, departemen Teknik di Jerman, Pemasaran di Jepang, dan Keuangan di Spanyol). Perusahaan ini ingin memiliki satu **ruang surat pusat (Mailroom)** untuk menangani semua komunikasi internal dan eksternal secara efisien.

- **Ruang Surat (VFS): Pusat operasional.** Semua karyawan (aplikasi pengguna) mengirimkan permintaan mereka (panggilan sistem seperti `open`, `read`) ke ruang surat menggunakan formulir standar perusahaan (API VFS). Karyawan tidak perlu tahu atau peduli dengan bahasa yang digunakan di departemen tujuan.
 - **Departemen-departemen (Sistem File: ext4, NTFS, FAT32)**
 - **Para Penerjemah (Driver Sistem File): Ruang surat** mempekerjakan tim **penerjemah ahli** untuk setiap bahasa.
 - Ketika sebuah permintaan tiba yang ditujukan untuk departemen Teknik di Jerman (sebuah file di partisi `ext4`), ruang surat memberikannya kepada penerjemah bahasa Jerman (`driver ext4`).
 - Ketika sebuah permintaan datang untuk departemen Pemasaran di Jepang (sebuah file di USB drive `NTFS`), permintaan itu diserahkan kepada penerjemah bahasa Jepang (`driver ntfs`).
1. Penerjemah menangani semua detail spesifik dalam berkomunikasi dengan departemennya masing-masing, mendapatkan hasilnya (misalnya, isi file), dan membawanya kembali ke ruang surat.
 2. Ruang surat kemudian meneruskan hasilnya kembali kepada karyawan yang meminta.
 3. Dari sudut pandang karyawan, mereka cuma ikut SOP dan menerima respons standar, sama sekali tidak menyadari proses penerjemahan yang rumit dan prosedur departemen yang berbeda yang terjadi di belakang layar.

8. (2.5 poin) Jelaskan konsep - konsep sistem operasi berikut dalam bahasa yang dimengerti anak berumur 5 tahun!

- a. (0.5 poin) *Kernel* -> Bos (Otaknya) komputer, tukang nyuruh-nyuruh bagian komputer lain.
- b. (0.5 poin) *RAM* -> Papan tulis magnet. Kalo dipake, datanya ada buat diliat langsung, biar kita bisa inget apa yang terakhir digambar. Kalo digeser (= dimatiin), isinya hilang. Hapusnya (hilang datanya) juga gampang, digeser dikit langsung hilang.



Maaf dicomot

- c. (0.5 poin) *Filesystem* -> Lemari yang isinya macem-macem barang anak 5 tahun. Mau itu kaos, celana, mainan, disatuin di satu tempat penyimpanan.
- d. (0.5 poin) *Shell* -> Remote TV. Kalau kita mencet tombol-tombol, hasilnya bisa beda. Bahkan kita bisa campur-campur tombol dan hasilnya bakal beda (dan bisa aja bekerja maupun engga)
- e. (0.5 poin) *Multiprocessing* dan cara kerjanya -> Coba liat ini orang



Tangannya ada banyak. Jadi, dia bisa sambil makan, sambil ngelus kucing, sambil dorong ayunan (multiproses). Semuanya dilakuin sama satu orang (komputer) yang sama, dalam satu waktu yang sama.

9. (2 poin) Pada saat *booting* sebuah mesin dengan sistem operasi Linux, seringkali akan terlihat kata-kata "initramfs" atau "initial ramdisk"

- a. (0.5 poin) Jelaskan proses apa yang sedang terjadi pada tahap *booting* ini!

Kernel yang baru saja dimuat oleh *boot loader* (seperti GRUB) sedang melakukan dekompresi dan memuat sebuah **sistem file root sementara** ke dalam memori (RAM). Sistem file sementara ini adalah `initramfs` (initial RAM file system). Kalau udah, kernel akan me-mount-nya sebagai sistem file root pertama dan menjalankan program `/init` yang ada di dalamnya.

- b. (1 poin) Mengapa sistem harus menggunakan initial ramdisk atau initramfs? Mengapa sistem tidak langsung me-mount filesystem sebenarnya?

Masalahnya tuh disini:

- Kernel butuh driver buat baca hard disk
- Tapi drivernya ada di dalam hard disk (Walawee)

Yaudah gabisa langsung di mount kalau kayak gitu. `Initramfs` disini sebenarnya adalah arsip `cpio` kecil yang berisi satu set minimal driver dan utilitas yang diperlukan untuk "membuka kunci" sistem file root yang sebenarnya. Prosesnya adalah:

1. *Boot loader* memuat kernel dan `initramfs` ke dalam RAM.
2. Kernel me-mount `initramfs` dari RAM.
3. Skrip `/init` di dalam `initramfs` berjalan dan memuat modul kernel yang diperlukan (misalnya, `nvme.ko`, `dm-mod.ko`, `ext4.ko`).
4. Setelah driver yang benar dimuat, skrip tersebut dapat menemukan, memeriksa, dan me-mount sistem file root yang sebenarnya.
5. Terakhir, sistem akan "berpindah root" (*switch root*) dari `initramfs` ke sistem file yang sebenarnya dan melanjutkan proses *booting* dengan menjalankan `/sbin/init` (`systemd`) yang asli.

- c. (0.5 poin) Apa yang biasanya akan terjadi jika initial ramdisk atau initramfs rusak atau tidak ditemukan?

Kalau initramfs rusak atau hilang, proses booting bakal gagal di tengah jalan karena kernel gatahu cara ngakses "rumahnya" sendiri.

Biasanya, sistem akan berhenti dan menjatuhkan pengguna ke sebuah **shell darurat** yang sangat terbatas (seperti prompt initramfs atau dracut), seringkali dengan pesan kesalahan seperti "kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)".

10. (2 poin) Apa keuntungan copy-on-write saat proses **fork()**?

"Bayangkan kamu punya buku catatan raksasa. Mengapa lebih efisien menyalin hanya halaman yang diubah daripada seluruh buku?"

Jawaban tidak perlu terpaku dengan analogi diatas.

Keuntungan utama dari penggunaan teknik **Copy-on-Write (CoW)** selama panggilan sistem `fork()` adalah **efisiensi**, baik dari segi **kecepatan** maupun **penggunaan memori**.

Daripada secara boros menyalin seluruh ruang memori proses parent (yang bisa berukuran gigabyte) untuk proses child yang baru, CoW memungkinkan proses parent dan child untuk berbagi halaman memori yang sama sampai salah satu dari mereka perlu memodifikasinya.

1. **Panggilan `fork()` Tanpa CoW:** Tanpa CoW, ketika `fork()` dipanggil, sistem operasi harus segera mengalokasikan memori baru untuk proses child dan menyalin setiap halaman memori dari proses parent. Ini lambat dan makan memori.
2. **Mekanisme CoW:** Dengan CoW, prosesnya jauh lebih cerdas:
 - o Ketika `fork()` dipanggil, OS tidak menyalin halaman memori fisik. Sebaliknya, ia hanya membuat tabel halaman (page table) baru untuk proses child dan memetakannya ke **halaman memori fisik yang sama** dengan proses parent.
 - o Untuk mencegah modifikasi yang tidak disengaja, OS menandai semua halaman yang dibagikan ini sebagai **read-only** di tabel

halaman kedua proses. Proses pembuatan proses child ini menjadi sangat cepat karena hanya melibatkan penyalinan tabel halaman, bukan data sebenarnya.

3. **Pemicu "Write":** Jika proses parent atau child mencoba untuk **menulis** ke salah satu halaman yang dibagikan (yang ditandai *read-only*), perangkat keras MMU akan memicu *page fault*.
4. **Proses "Copy":** Kernel menangani *fault* ini. Nanti dia bakal mengalokasikan halaman memori fisik yang baru, menyalin konten dari halaman bersama asli ke halaman baru ini, dan kemudian memperbarui tabel halaman dari proses yang mencoba menulis untuk menunjuk ke salinan baru yang sekarang bersifat pribadi dan dapat ditulisi (*writable*). Proses lainnya (yang tidak melakukan modifikasi) terus berbagi halaman asli.

III. Teknologi Platform



POV: Me after ruling the cloud infrastructure

1. (6 poin) Sebuah perusahaan bernama Furina Courthouse Corporated .inc menggunakan arsitektur microservices yang berjalan diatas kubernetes dan docker. Salah satu service yang digunakan adalah auth service berbasis Go yang menangani user authentication & authorization. Karena sifatnya yang sangat sensitif dan sering di scale out, developer pada FCC ingin mengevaluasi apakah service tersebut sebaiknya dijalankan sebagai unikernel demi meningkatkan keamanan dan efisiensi kinerja.
 - a. (2 poin) Jelaskan bagaimana model isolasi unikernel berbeda dibandingkan container dalam konteks execution di dalam cluster kubernetes, bagaimana perbedaan tersebut dapat mengubah permukaan serangan untuk auth service?

- **Container (di Kubernetes):** Sebuah container berjalan sebagai proses terisolasi di atas **kernel sistem operasi host yang sama**. Isolasi ini berada pada tingkat sistem operasi.
- **Unikernel (di Kubernetes):** Sebuah unikernel adalah *image single-address-space* yang terspesialisasi, yang menggabungkan aplikasi dengan *library OS* minimal. Ia tidak berjalan di atas kernel OS host; sebaliknya, ia **adalah OS itu sendiri**. Untuk dapat berjalan di dalam klaster Kubernetes, setiap unikernel harus dibungkus di

dalam sebuah mesin virtual (VM) ringan yang dikelola oleh hypervisor (seperti KVM/QEMU atau AWS Firecracker). Dengan demikian, isolasi yang diberikan berada pada **tingkat virtualisasi perangkat keras**, yang secara fundamental lebih kuat daripada isolasi tingkat OS pada container.

- b. (2 poin) Apa saja kesulitan yang mungkin muncul ketika mengintegrasikan unikernel ke dalam ekosistem server kubernetes, terutama dalam hal logging, monitoring, dan debugging, serta jelaskan cara untuk mengatasi kesulitan tersebut.

- Logging & Monitoring -> gabisa pasang agen pemantau di dalam bunker. Pemantauan harus dilakukan "dari luar" pakai hypervisor.
- Debugging: gabisa kubectl exec buat masuk dan ngoprek. Debugging jadi lebih susah.

- c. (2 poin) Apakah auth service sebaiknya di migrasikan ke unikernel? Jika ya, jelaskan pendekatan migrasi secara bertahap, jika tidak jelaskan alasannya dengan pros and cons yang jelas.

Buat layanan super sensitif kayak otentikasi, pindah ke unikernel itu langkah strategis yang bagus banget buat keamanan. Tapi, tim harus siap dengan cara kerja operasional yang baru dan berbeda.

2. (2.5 poin) Bagaimana cara docker dapat menjamin konsistensi environment aplikasi di berbagai mesin dan mengapa hal tersebut dibutuhkan dalam pengembangan perangkat lunak? Jelaskan pula apa risiko jika environment aplikasi tidak dikelola dengan baik, dan bagaimana docker membantu meminimalkan masalah tersebut!

Semua kode, library, dan dependency ver. digabungin jadi satu dalam sebuah *immutable* file bernama Docker Image. Ini ngejamin semua isi aplikasi konsisten di berbagai mesin.

Kenapa dibutuhkan? Karena emang beberapa library biasanya deprecated untuk beberapa versi. Dan beberapa "alat"/library yang hanya tersedia di atas, atau

bahkan spesifik untuk sebuah versi. Kalau ga konsisten, ini memicu munculnya bug, yang bisa ngebikin sistem rentan.

Dengan ngebikin **lingkungan yang terstandarisasi, portabel, dan terkontrol versinya** (yaitu, Docker image), Docker mastiin lingkungan yang persis sama digunakan di seluruh *pipeline* CI/CD. Image yang diuji adalah image yang sama persis dengan yang di-deploy ke produksi.

Referensi: Dijelaskan temen STI (Aidan)