Muhammad Adha Ridwan - 13523098

1 Computer Organization and Architecture

1.1 Logic Gates and Binary Arithmetic (3.5 points)

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.

1.1.1 1-bit Binary Addition (0.75 points)

Bagaimana kita menyusun sebuah komponen logic gate untuk melakukan penjumlahan dua angka biner berukuran 1-bit? Jelaskan arsitektur dan cara kerja dari komponen tersebut.

Answer:

Untuk penjumlahan bilangan biner 1-bit, dikenal luas suatu sirkuit bernama *half-adder*. Half-Adder adalah kombinasi sirkuit yang menerima input dua bilangan biner 1-bit, yang kemudian menghasilkan output *sum* dan *carry*. Hasil *output* tersebut didapatkan dari melakukan XOR kedua input dan hasil AND kedua input untuk *carry*.

Half Adder adalah komponen dasar dalam aritmatika biner yang terdiri dari dua logic gate:

1. XOR Gate - Untuk mendapatkan bit sum output.
2. AND Gate - Untuk mendapatkan bit carry output

Cara kerjanya adalah dengan berikut:

1. XOR Gate
   - Menghasilkan 1 ketika input berbeda (0-1/1-0).
   - Menghasilkan 0 ketika input sama (1-1/0-0).
   - Digunakan untuk merepresentasikan digit terakhir hasil perjumlahan.
2. AND Gate
   - Menghasilkan 1 ketika kedua input bernilai 1.
   - Menghasilkan 0 ketika kedua input salah satunya bernilai 0.
   - Digunakan untuk merepresentasikan "carry" dari hasil perjumlahan.

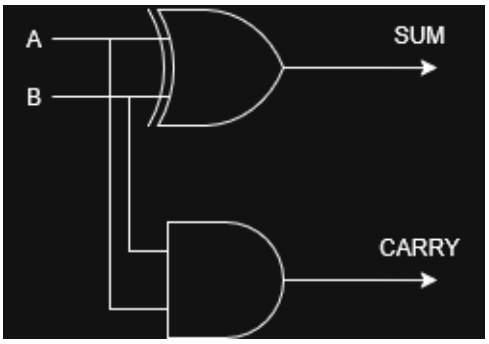


Figure 1: Sumber Pribadi

1.1.2 Multi-bit Binary Addition (0.75 points)

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

Answer:

Untuk penjumlahan bilangan biner yang lebih besar dari 1-bit atau multi-bit terdapat beberapa metode, kita akan menggunakan metode yang sederhana dan *straightforward* yaitu Ripple Carry Adder. Dari Half Adder sebelumnya yang hanya dapat mengatasi penjumlahan dua bilangan 1-bit, dikembangkan Full Adder yang dapat menerima bit carry dari tahap/penjumlahan sebelumnya.

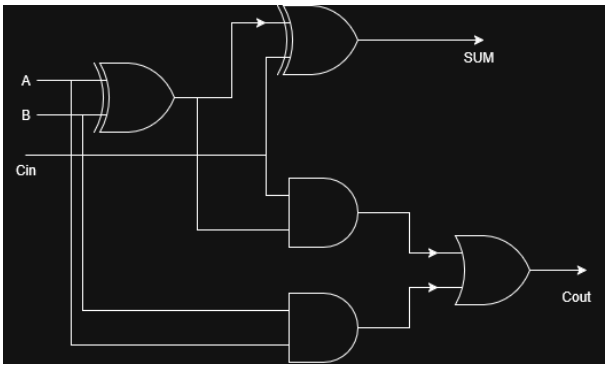


Figure 2: Sumber Pribadi

Masalah utama ketika kita ingin menjumlahkan bilangan dengan bit-bit yang lebih tinggi adalah diperlukannya untuk menghitung *carry-in* dari bit yang lebih rendah. Dengan Full Adder yang kita *stack*, didapat kombinasi logic gates yang dapat melakukan penjumlahan multi bit sesuai dengan jumlah Full Adder yang digunakan. Berikut Contoh Ripple Carry Adder untuk 4 bit.

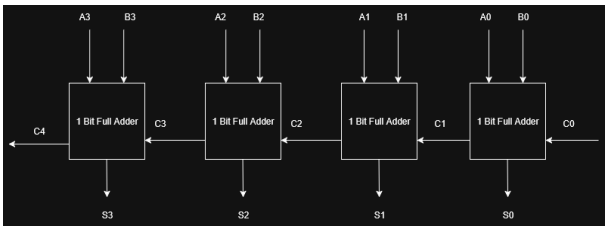


Figure 3: Sumber Pribadi

1.1.3 Memory Components (2 points)

BSelain 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.

SR Latch (1 point) Answer:

SR Latch adalah salah satu komponen memori yang dapat menyimpan informasi sebanyak 1 bit. Arsitektur dari SR Latch memiliki dua tipe, yaitu terdiri dari 2 NOR Gate ataupun 2 NAND Gate.

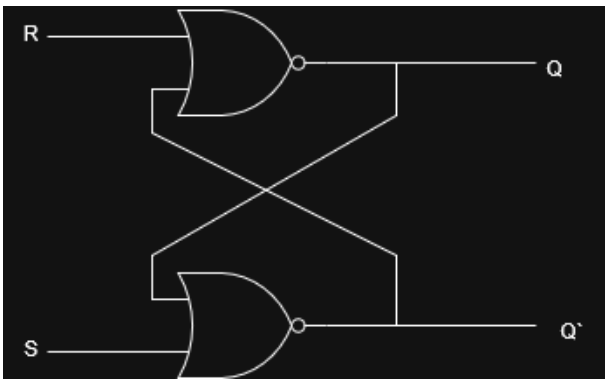


Figure 4: Sumber Pribadi

Cara Kerja dari SR Latch adalah seperti berikut.

- S=0, R=0: Kondisi hold/memory - output tetap mempertahankan nilai sebelumnya
- S=1, R=0: Set operation - Q menjadi 1, Q‘menjadi 0
- S=0, R=1: Reset operation - Q menjadi 0, Q‘menjadi 1
- S=1, R=1: Forbidden state - tidak boleh terjadi karena menghasilkan kondisi tidak stabil

Kelemahan dari SR Latch adalah:

- Race condition: Ketika S dan R berubah secara bersamaan, output bisa menjadi tidak predictable
- Forbidden state: Kombinasi S=1, R=1 harus dihindari
- Tidak ada kontrol timing: Latch langsung merespons perubahan input tanpa sinkronisasi clock

Gated SR Latch (0.5 points) Answer:

Merupakan perkembangan dari SR Latch, dimana ditambahkan *enable signal* yangg berupa dua AND Gate pada input S dan R.

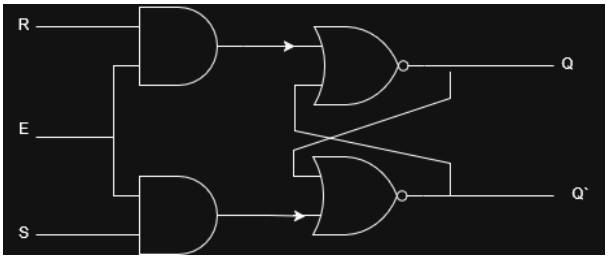


Figure 5: Sumber Pribadi

Cara Kerja:

- Latch hanya aktif ketika EN=1
- Ketika EN=0, latch dalam kondisi hold terlepas dari nilai S dan R
- Ketika EN=1, berperilaku seperti SR Latch normal

Kelemahan:

- Masih rentan terhadap transparency problem - output dapat berubah berkali-kali selama enable signal aktif
- Glitch sensitivity - noise pada input saat enable aktif dapat mengubah state

Gated D Latch (0.5 points) **Answer:** Merupakan perkembangan dari D Latch biasa, dimana menggunakan satu input data (D) yang terhubung langsung ke S dan melalui inverter ke R pada Gated SR Latch. Hal ini menghilangkan kemungkinan forbidden state.

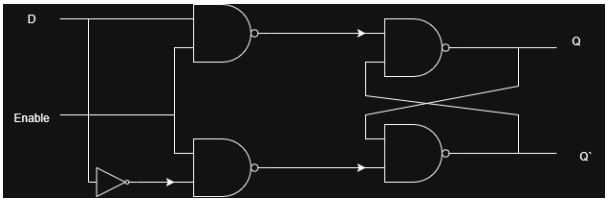


Figure 6: Enter Caption

Cara Kerja:

- Ketika EN=0: Latch dalam kondisi hold
- Ketika EN=1: Output Q mengikuti input D (Q = D)
- Tidak ada forbidden state karena R selalu complement dari S

Kelemahan:

- Transparency issue: Selama EN=1, output langsung mengikuti perubahan input D, yang dapat menyebabkan multiple transitions
- Setup dan hold time requirements: Input D harus stabil sebelum dan sesudah enable signal untuk operasi yang reliable

Bonus (0.3 points each): Provide visualization or illustration of each component (self-created).

1.2 Speculative Execution and Cache Attacks (5 points)

eberapa 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 mengeksploitasi speculative execution. Jelaskan juga bagaimana cara untuk memitigasi serangan tersebut?

Answer:

Timing side-channel attack yang mengeksploitasi speculative execution melalui cache replacement policy bekerja dengan memanfaatkan perbedaan waktu akses memori yang dapat diukur. Hacker pertama-tama mempersiapkan kondisi cache dengan melakukan flush pada cache lines tertentu menggunakan instruksi seperti 'clflush' atau mengisi cache dengan data yang dikontrol. Selanjutnya, attacker memicu speculative execution dengan membuat branch prediction yang salah, sehingga processor secara spekulatif mengeksekusi instruksi yang seharusnya tidak dijalankan. Instruksi spekulatif ini akan mengakses lokasi memori berdasarkan nilai data rahasia, misalnya menggunakan data rahasia sebagai index untuk mengakses array. Meskipun hasil spekulasi akan di-rollback ketika processor menyadari prediksi yang salah, jejak akses memori tetap tertinggal di cache.

Hacker kemudian menganalisis perubahan state cache dengan mengukur waktu akses ke berbagai lokasi memori. Jika suatu alamat memori dapat diakses dengan cepat (cache hit), berarti alamat tersebut telah di-load ke cache selama fase speculative execution, mengindikasikan bahwa data rahasia memiliki nilai tertentu yang digunakan sebagai index. Dengan mengulang proses ini berkali-kali dan menganalisis pola timing, attacker dapat merekonstruksi nilai data sensitif bit demi bit. Teknik seperti Flush+Reload, Prime+Probe, atau Evict+Time sering digunakan untuk mengoptimalkan pengukuran timing dan meningkatkan akurasi ekstraksi data.

Untuk memitigasi serangan ini, beberapa pendekatan dapat diterapkan. Pertama, implementasi software-based mitigation seperti array bounds checking, constant-time algorithms, dan memory layout randomization dapat mengurangi kemungkinan kebocoran informasi. Kedua, hardware-based solutions seperti cache partitioning, disabling speculative execution untuk operasi sensitif, atau menggunakan processor dengan desain yang lebih aman terhadap side-channel attack. Ketiga, sistem dapat menerapkan microcode updates yang disediakan vendor untuk menutup celah keamanan spesifik, meskipun ini dapat mengurangi performa. Selain itu, penggunaan teknik seperti kernel page-table isolation (KPTI), address space layout randomization (ASLR), dan control flow integrity (CFI) dapat mempersulit attacker dalam melakukan eksploitasi yang sukses.

1.3 Assembly Code Analysis (3 points)

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

```
0x0000000140007c10 <+0>: sub $0x28,%rsp
0x0000000140007c14 <+4>: call 0x140001550 <__main>
0x0000000140007c19 <+9>: lea 0x13e0(%rip),%rcx
0x0000000140007c20 <+16>: call 0x140001450 <printf.constprop.0>
0x0000000140007c25 <+21>: xor %eax,%eax
0x0000000140007c27 <+23>: add $0x28,%rsp
0x0000000140007c2b <+27>: ret
```

Original source code:

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

1.3.1 Compiler Optimization (0.5 points)

Kenapa bisa muncul printf.constprop.0 dan compiler tidak menggunakan printf biasa?
Answer:

Dikarenakan dilakukan optimisasi oleh compiler, pada *original source code* dilihat perintah printf tidak memberikan format untuk jenis data yang akan di print, tetapi langsung memberikan literal string sehingga tidak memerlukan runtime proccesing dan argumen atau data yang diprint telah diketahui saat compile time. GCC membuat version printf yang dioptimize untuk kasus ini, suffix "constprop.0" menandakan bahwa ini adalah 0-indexed constant propagated printf. Pada printf versi ini tidak diperlukan parsing untuk format specifies dan bisa langsung print sehingga lebih efisien.

1.3.2 Math Library Functions (0.5 points)

Apakah sin.constprop.0 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.

Answer:

Bisa saja muncul. Kondisi-kondisi yang memungkinkan compiler untuk melakukan optimisasi dengan *constant propagation* adalah seperti:

1. Fungsi sin memiliki argumen berupa konstanta literal.
2. Konstanta yang digunakan dapat di evaluate saat compile time.
3. Utilisasi flag compiler (O1, O2, O3, etc) untuk tingkat optimisasi

1.3.3 Compiler Design Analysis (2 points)

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)

Answer:

Ya, untuk compiler modern fitur optimisasi menjadi penting karena beberapa alasan, seperti:

1. Performa - penggunaan function specialization seperti constant propagation dapat meningkatkan performa kode secara signifikan dengan cara mengeliminasi overhead runtime.
2. Ukuran - Untuk kasus dengan ukuran yang besar dengan penggunaan berulang, optimisasi seperti ini dapat mengurangi ukuran dari binary.

3. Relevansi - dengan semakin *bloated*-nya *software* saat ini, optimisasi yang dilakukan oleh compiler semakin penting untuk mengurangi dampak tersebut.

Tetapi, apakah fitur optimisasi seperti ini berbahaya?, tentu, akan tetapi sangat tergantung dengan konteks, seperti:

1. Debugging - dikarenakan dilakukan optimisasi, assembly hasil decompiling binary yang dihasilkan akan tidak 1:1 mapping dengan source code asli, sehingga dapat memperkompleks debugging.

2. Analsis - dikarenakan hasil assembly sudah tidak 1:1 mapping dengan source code, *static analysis* untuk melakukan analisis keamanan menjadi lebih *tricky* serta *tools* dapat kesusahan menangkap *pattern* karena beberapa bagian sudah diganti.

3. Reproducibility - jika terjadi bug, akan menjadi lebih sulit untuk mereproduksinya pada komputer lain, karena optimisasi sangat bergantung dengan versi *compilter* dan *environment*-nya sehingga hasil kompilasi dapat berbeda di tiap *environment* mengakibatkan reproduksi bug lebih sulit.

1.4 Cache Memory Analogy (3 points)

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,

1.4.1 Locality of Reference (1 point)

Jelaskan prinsip "locality of reference" menggunakan analogi ini.

Answer:

1. Temporal Locality (Lokalitas Waktu)

   - Ketika suatu buku dibaca, kemungkinan besar buku tersebut akan dibaca lagi, misal: seorang murid membaca Halliday untuk belajar Fisdas, maka kemungkinan besar murid tersebut akan membaca Halliday lagi untuk mengerjakan tugas, sehingga lebih baik buku tersebut diletakkan di perpustakaan kecil didekat ruang kelas.

2. Spatial Locality (Lokalitas Ruang)

   - Ketika suatu buku berjenis tertentu dibaca, kemungkinan besar buku dengan jenis yang sama akan dibaca, misal: seorang murid membaca buku Uma Musume Cinderella Gray Volume 1, maka kemungkinan besar murid tersebut akan membaca Uma Musume Cinderella Gray Volume 2, sehingga buku jenis/series tersebut lebih baik diletakkan di perpustakaan kecil.

1.4.2 Cache Miss Impact (1 point)

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

Answer:

Dalam analogi diatas cache miss dapat terjadi saat seorang murid ingin mencari buku dengan judul tertentu di perpustakaan kecil, akan tetapi perpustakaan kecil tidak memilikinya, sehingga murid harus:

1. Memberhentikan aktivitasnya, seperti belajar di kelas.
2. Berjalan menuju perputaskaan utama yang jaraknya lebih jauh dari perpustakaan kecil.
3. Melakukan prosedur peminjaman buku di perpustakaan utama
4. Berjalan kembali ke kelas

Sehingga proses ini jauh lebih lama dibanding jika suatu buku tersedia di perpustakaan kecil.

1.4.3 Prefetching Strategy (1 point)

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

Answer:

Dengan asumsi "terjebak" adalah ketika murid berhenti (*halt*/stuck) ketika diskusi. Strategi prefetching dapat membantu murid tidak terjebak dalam diskusi dengan menyediakan buku buku yang kemungkinan terpakai dalam diskusi, misal: seorang murid sedang dalam diskusi tentang kalkulus, anggaplah ada guru yang mengamati apa saja yang didiskusikan, sehingga ketika guru tersebut mengetahui apa yang dibahas dalam diskusi dan mempersiapkan buku yang berkaitan dengan bahan diskusi tersebut, sehingga seorang murid jika tidak mengetahui sesuatu, tidak perlu pergi ke perpustakaan utama untuk mencari buku, akan tetapi cukup ke perpustakaan kecil yang isinya sudah di siapkan oleh guru tersebut, sehingga tidak terjadi "stuck".

1.5 Register Renaming (3 points)

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).

1.5.1 Implementation Mechanism (2 points)

Jelaskan cara/mekanisme yang memungkinkan agar arsitektur tersebut dapat diimplementasikan!

Answer:

Untuk arsitektur ini bekerja, kita akan melakukan beberapa hal, untuk implementasinya kita akan menggunakan:

1. RAT - Register Alias Table (RAT) adalah suatu struktur data yang akan memetakan register fisik (P0-P7) ke register logis (R0-R15).
2. Free List - list yang akan menyimpan informasi register fisik mana saja yang kosong dan dapat digunakan.

3. Renaming Logic - logic untuk melakukan penamaan ulang untuk register yang akan dipetakan

4. Register Spiling - karena register fisik yang lebih sedikit dari register logis, maka jika terjadi konflik, suatu data harus dipindahkan ke storage

Cara kerjanya adalah seperti berikut:

1. ketika instruksi di-decode register logis akan dipetakan ke register fisik melalui RAT

2. jika ada operasi write, register fisik kosong yang berada di free list akan dialokasikan untuk melakukan operasi tersebut.

3. RAT diperbarui sesuai dengan perubahan/penambahan yang terjadi.

4. jika ada register yang tidak digunakan atau dipanggil lagi register tersebut akan ditambahkan ke free list.

1.5.2 Pipeline Hazards Impact (0.5 points)

Apa dampak pada pipeline hazard (specifically data hazards)?

Answer:

Dampak Positif:

1. Mengurangi WAR (Write After Read) hazards: Register renaming menghilangkan false dependencies karena setiap write menggunakan register fisik baru

2. Mengurangi WAW (Write After Write) hazards: Multiple writes ke register logis yang sama menggunakan register fisik berbeda

Dampak Negatif:

1. RAW (Read After Write) hazards masih ada: True dependencies tetap harus ditangani dengan forwarding atau stalling

2. Kompleksitas tambahan: Perlu mekanisme untuk tracking kapan register fisik bisa di-dealokasi

1.5.3 Compilation Optimization Impact (0.5 points)

Apa dampak pada optimasi kompilasi?

Answer:

Dampak Positif:

1. Mengurangi beban register allocation: Compiler tidak perlu terlalu agresif dalam mengoptimalkan penggunaan register karena hardware dapat menangani lebih banyak "virtual" register

2. Memungkinkan optimasi yang lebih agresif: Compiler bisa melakukan transformasi yang menghasilkan lebih banyak temporary values tanpa khawatir keterbatasan register

1.6 IA32 Data Structures (2.5 points)

Diberikan struktur data sebagai berikut pada mesin IA32.

```
struct hawk {
    union sybau a;
    char b[3];
    int c;
};

struct tuah {
    char d;
    int e[4];
    struct hawk* f;
    struct tuah* g;
};

union sybau {
    char h;
    struct hawk* i;
    struct tuah* j;
};
```

1.6.1 Memory Alignment and Size (1 point)

Tentukan nomor byte alignment untuk atribut pada masing-masing struktur. Lalu tentukan ukuran byte total dari masing-masing struktur. Jelaskan secara singkat.

struct hawk (0.3 points) Answer:

untuk struct hawk
Byte Alignment:

1. union sybau a - offset 0, 4 bytes
2. char b[3] - offset 4, 3 bytes maka dipadding 1 bytes
3. int c - offset 8, 4 bytes

Total sizes: 12 Bytes

struct tuah (0.3 points) Answer:

Untuk struct tuah
Byte Alignment:

1. char d - offset 0, dengan size 1, akan dipadding 3 byte.
2. int e[4] - offset 4, dengan 4 element, maka 4 - 19
3. struct hawk* f - offset 20 , karena IA32, maka pointer memiliki 4 byte
4. struct tuah* g - offset 24, karena IA32, maka pointer memiliki 4 byte.

Size Total = 28 Bytes

union sybau (0.4 points) **Answer:**

Union merupakan kasus dimana semua attribute berbagi tempat memory yang sama, sehingga:

1. char h - offset 0
2. struct hawk* i - offset 0
3. struct tuah* j - offset 0

Total size bergantung dengan attribute yang memiliki size terbesar sehingga: 4 bytes (pointer 32-bit)

1.6.2 Assembly Analysis (1.5 points)

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.

proc1 (0.25 points) `int proc1(struct tuah *x) { return ______ }`

Answer:

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

```
    movl 8(%ebp), %eax
```

Meload parameter pertama yaitu struct tuah *x

```
    movl 12(%eax), %eax
```

dari pointer tersebut direference dan diambil di offset 12, jika di cek dari soal sebelumnya, yaitu e[2]

proc2 (1.25 points) `int proc2(struct hawk *x) { return _____ }`

Answer:

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

```
movl 8(%ebp), %eax
```

Load parameter pertama

```
movl (%eax), %eax
movl 24(%eax), %eax
```

Dari 2 command tersebut kita tahu bahwa union ditreat sebaga struct tuah* j, karena diperlakukan sebagai pointer dan menggunakan offset 24 byte. disini dapat diakses field g dari struct tuah

```
movl 20(%eax), %eax
movzbl 6(%eax), %eax
```

Kemudian direference lgi dan diambil offset ke 20 dari field g yaitu field f, kemudian diakses offset ke 6 dari hasil sebelumnya dan didapat yaitu b[2] setelah berbagai reference. digunakan movzbl diakhir karena return type berupa integer sedangkan b[2] adalah char, sehingga diextend byte nya menjadi long, atau sebuah unsigned char, agar tidak ada garbage byte yang ikut direturn. return akhir kurang lebih setara: (unsigned char)x→ $a.j \rightarrow g \rightarrow f \rightarrow b[2]$

1.7 Endianness (1 point)

Kita mengenal beberapa cara untuk menyimpan data ada little endian dan big endian. Mengapa beberapa device lebih memilih untuk menyimpan data secara little endian?

Answer:

Sebagian besar device memilih menggunakan little endian karena kompabilitas dan legacy. Pemain besar sektor prosesor dan semikonduktor seperti Intel dan AMD sama2 menggunakan arsitektur x86 yang menggunakan little endian byte ordering, sehingga kebanyakan perusahaan yang ingin membuat device/software yang menggunakan /kompatibel dengan chip semikonduktor tersebut mau tidak mau menggunakan little endian byte ordering. Alasan lainnya adalah little endian lebih "natural" untuk operasi yang dilakukan oleh prosesor, seperti aritmatika karena diakses mulai dari bit terkecil, penambahan untuk alamat memori, dan typecasting suatu bilangan, misal dari 32 bit ke 8 bit.

2 Operating Systems

"Linux my beloved"

2.1 Multiprogramming Performance (1 point)

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.

Answer:

Penurunan kinerja prosesor yang drastis disebabkan oleh fenomena yang bernama *trashing*. Thrashing adalah saat dimana *in nutshell* prosesor mengerjakan tugas lain yang tidak produktif dibandingkan tugas utama yang produktif, dalam konteks ini adalah prosesor lebih banyak mengurus page fault dibanding menjalankan proses/instruksi utama. Dengan semakin tingginya derajat multiprogramming, akan semakin banyak proses yang berjalan, mengakibatkan semakin banyak proses yang membutuhkan sumber daya sehingga terjadi kompetisi, untuk menanganinya OS melakukan page swapping, akan tetapi karena terlalu banyak swapping dan kita tahu page swapping adalah operasi yang cukup "mahal" mengakibatkan kinerja prosesor menurun karena lebih sering mengerjakan swapping dibanding proses/intruksi asli.

2.2 Virtualization vs Containerization (1.5 points)

2.2.1 Virtualization (0.5 points)

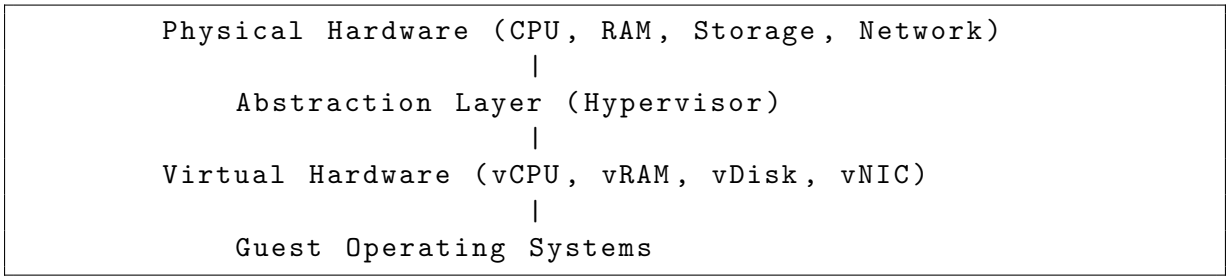
Jelaskan konsep virtualization dan cara kerjanya.

Answer:

Dalam konteks OS, Virtualization adalah teknologi yang membuat suatu mesin komputer dapat menjalankan banyak sistem yang diemulasi dalam satu mesin komputer seperti, operating system, network, storage, etc. Dilakukan dengan membuat layer abstraksi antara physical hardware dan operating system.

Komponen yang digunakan dalam Virtualization adalah Hypervisor, dimana Hypervisor ini bertindak sebagai layer abstraksi antara physical hardware dan virtual machine. Fungsi utamanya adalah mengurus manajemen sumber daya hardware, selain itu juga melakukan tugas seperti schedulibg, I/O abstraction, dan memory management.

Cara kerjanya adalah:



Dengan melakukan proses seperti

1. CPU Virtualization: CPU time dibagi antar VM,CPU scheduling algorithms untuk fairness

2. Memory Virtualization: Reclaim unused memory dari VM, Alokasi memory virtual ¿ physical, Deduplication pages yang identik

3. Storage Virtualization: Virtual disk Snapshots dan cloning capabilities

   3. Network Virtualization

   Virtual switches: Software-based network switching VLANs: Network segmentation dalam environment virtual Port groups: Konfigurasi network policies

2.2.2 Containerization (0.5 points)

Jelaskan konsep containerization dan cara kerjanya.

Answer:

Cointainerization adalah salah satu penerapan Virtualization, lebih tepatnya di tingkat operating system. Containerization memungkinkan kita untuk memaketkan suatu kode dengan semua file dan library yang dibutuhkan untuk proses dari kode tersebut berjalan. Ini membuat proses tersebut menjadi user space isolated instance yang berjalan pada host kernel dan dapat berjalan sekali banyak (multiple) sekaligus. Cara kerja containerization adalah dengan memanfaatkan Linux namespaces dan control groups (cgroups) untuk menciptakan isolasi process tanpa virtualisasi hardware penuh.

Container runtime membaca image berlapis, kemudian menggunakan system call clone() dengan *namespace flags* untuk membuat process terisolasi. Namespaces memberikan isolasi pada PID (process tree yang terpisah), network (stack jaringan virtual), mount (filesystem view terisolasi), dan user (mapping user ID), sementara cgroups mengontrol resource manajemen seperti CPU, memory, dan I/O. Union filesystem seperti OverlayFS menggabungkan read-only image layers dengan writable container layer, memungkinkan sharing base layers antar containers untuk efisiensi storage. Container kemudian berjalan dalam lingkungan terisolasi ini dengan virtual network interface untuk komunikasi dan volume mounts untuk persistent data, menghasilkan isolated instances yang ringan dan cepat dibandingkan virtualisasi dengan Virtual Machine biasa.

2.2.3 Comparison (0.5 points)

Bandingkan kedua konsep tersebut. Apa perbedaannya? Apa persamaannya? Apa kelebihan dan kekurangan dari menggunakan masing-masing metode?

Answer:

Perbedaan: Virtualization dan containerization berbeda secara fundamental dalam level abstraksi dan mekanisme isolasi yang digunakan. Virtualization bekerja di hardware level dengan menciptakan multiple virtual machines yang masing-masing memiliki operating system lengkap dan terisolasi secara hardware melalui hypervisor, sementara containerization bekerja di OS level dengan berbagi kernel host yang sama namun menciptakan isolated user space instances menggunakan Linux namespaces dan cgroups. Perbedaan ini mengakibatkan virtualization membutuhkan resource overhead yang jauh lebih besar dengan minimum 2-4GB RAM dan 20-50GB storage per VM serta startup time 30 detik hingga beberapa menit, sedangkan container hanya memerlukan 10-100MB RAM dan ratusan MB storage dengan startup time 1-5 detik. Dari segi performance, virtualization memiliki overhead 5-15% karena layer hypervisor, sementara container memberikan near-native performance karena direct access ke host kernel.

Persamaan: Kedua teknologi memiliki tujuan fundamental yang sama yaitu menyediakan isolasi workload dan optimasi resource utilization. Keduanya memungkinkan multiple aplikasi atau sistem berjalan secara terisolasi pada single physical hardware, mendukung portability workload antar different environments, menyediakan mekanisme untuk resource allocation dan management, serta memungkinkan scalability baik vertical maupun horizontal. Virtualization dan containerization sama-sama mendukung automation dalam deployment dan management, menyediakan snapshot dan backup capabilities untuk disaster recovery, serta memiliki ecosystem tools yang mature untuk monitoring, orchestration, dan security management.

Kelebihan dan Kekurangan: Virtualization memberikan kelebihan berupa strong security isolation yang ideal untuk multi-tenant environments, kemampuan menjalankan multiple OS yang berbeda pada single hardware, mature ecosystem dengan extensive vendor support, serta compliance dengan enterprise security standards. Namun kekurangannya adalah resource consumption yang tinggi, management complexity yang lebih besar, licensing costs untuk multiple OS instances, dan slower deployment cycles karena boot time yang lama. Containerization unggul dalam efficiency dengan resource utilization yang optimal, fast deployment dan scaling capabilities, perfect integration dengan DevOps workflows dan CI/CD pipelines, serta ideal untuk microservices architecture dan cloud-native applications. Kelemahannya adalah security isolation yang lebih lemah karena shared kernel, keterbatasan pada single OS family (umumnya Linux), kompleksitas dalam persistent storage management, dan potential kernel-level vulnerabilities yang dapat mempengaruhi semua containers

2.3 Virtual Memory (1.5 points)

Jelaskan bagaimana virtual memory bekerja berdasarkan diagram tersebut

Answer:

Virtual Memory bekerja sebagai manajemen memori dimana dia memberikan ilusi kepada operating system bahwa mereka memiliki memory yang lebih besar dibandingkan physical memory, ini dilakukan dengan memanfaatkan storage. Berdasarkan diagram, suatu proses membutuhkan data pada page table, kemudian melakukan reference ke page table untuk mendapatkan alamat dari page yang dibutuhkan, namun page yang diminta tidak berada pada physical memory sehingga terjadi page fault yang melakukan trap kepada operating system. Kemudian, operating system mengecek apakah page tersebut ada distorage, kemudian OS akan mencari free page pada physical memory, jika tidak ada algoritma page replacement akan dilakukan. Kemudian, dilakukan pemuatan missing page ke free page pada physical memory dan operating system melakukan page table reset untuk update mapping. Terakhir, akan diulang intruction yang menyebabkan trap tersebut dan sekarang berhasil.

2.4 Peterson's Solution (6 points)

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, P0 dan P1, 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.)

2.4.1 Algorithm Explanation (1 point)

Jelaskan bagaimana algoritma Peterson's Solution bekerja.

Answer:

Algoritma solusi Peterson bekerja dengan utilize dua variabel shared turn dan flag[2] untuk menentukan apakah aman atau tidak, sehingga dapat memastikan mutual exclusion. Tetapi, Solusi Peterson merupakan solusi teoritis yang tidak applicable.

Cara kerjanya sebagai berikut:

1. Entry - Ketika suatu proses ingin masuk ke critical section, ia akan mengeset flag true dan turn untuk proses lainnya.
2. Waiting - kemudian proses tersebut akan menunggu di loop while (flag [j] && turn == j) hingga antara flag[j] = false menandakan proses lain tidak ingin masuk atau turn != j menandakan proses lain sudah selesai akses dan memindahkan turn.
3. Critical Section - Proses mengakses bagian critical section
4. Exit - Setelah selesai, flag di set false menandakan sudah tidak ingin masuk ke critical section.

2.4.2 Variable Roles (0.5 points)

Apa peran dari variabel turn dan flag pada algoritma? Mengapa diperlukan kedua variabel tersebut agar solusi bekerja dengan benar? Jelaskan

Answer:

flag[i]: Menandakan apakah proses Pi ingin masuk ke critical section

turn: Menandakan giliran proses mana yang diizinkan masuk jika kedua proses sama-sama ingin masuk

Kedua variabel diperlukan karena:

- Hanya menggunakan flag saja dapat menyebabkan deadlock (kedua proses menunggu satu sama lain)
- Hanya menggunakan turn saja tidak memungkinkan proses yang lebih cepat untuk masuk berkali-kali jika proses lain sedang tidak aktif
- Kombinasi keduanya memastikan tidak ada deadlock dan memberikan alternatif masuk

2.4.3 Correctness Proof (1.5 points)

Buktikan bahwa Peterson's Solution sudah merupakan solusi yang benar untuk mengatasi critical section problem.

Answer:

Algoritma Peterson Solution menyelesaikan 3 masalah untuk Critical Section

- Mutual Exclusion - Anggap P0 dan P1 sama-sama berada di critical section. Ini berarti:
  - flag[0] = true dan flag[1] = true
  - P0 telah melewati while (flag[1] turn == 1)
  - P1 telah melewati while (flag[0] turn == 0)

  Untuk P0 melewati loop, haruslah turn != 1, artinya turn = 0 Untuk P1 melewati loop, haruslah turn != 0, artinya turn = 1 Ini kontradiksi karena turn tidak bisa bernilai 0 dan 1 secara bersamaan

- Progress - Jika tidak ada proses di critical section dan ada proses yang ingin masuk:

  - Jika hanya satu proses ingin masuk (misal P0), maka flag[1] = false, sehingga P0 langsung dapat masuk

  - Jika kedua proses ingin masuk, variabel turn akan menentukan siapa yang masuk terlebih dahulu

- Bounded Waiting - Setelah P0 merequest masuk critical section, P1 maksimal dapat masuk satu kali sebelum P0 mendapat giliran, karena setelah P1 keluar, flag[1] = false dan P0 dapat langsung masuk.

2.4.4 Modern OS Limitations (0.5 points)

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.

Answer:

Alasan Solusi Peterson tidak digunakan pada operating system modern adalah:

- Busy Waiting - Proses yang menunggu terus-menerus mengecek kondisi dalam loop, membuang CPU cycles.

- Tidak Scalable - Solusi elegan untuk 2 proses, tetapi susah discale untuk n proses.

- Memory Ordering Issues - Pada multiprocessor modern dengan cache dan optimisasi compiler, urutan eksekusi instruksi bisa berubah-berubah,sehingga dapat merusak asumsi algoritma.

- Tidak Efisien - Menggunakan CPU secara intensif meskipun tidak melakukan pekerjaan produktif.

2.4.5 Alternative Methods (2 points)

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.

test_and_set() (1 point) **Answer:**

test_and_set() adalah operasi atomic yang melakukan dua aksi sekaligus dalam satu instruksi yang tidak dapat diinterupsi:

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;     // 1. Baca nilai lama
    *target = true;           // 2. Set ke true
    return rv;                // 3. Return nilai lama
}
```

Contoh pemakain:

```
// Inisialisasi
boolean lock = false;

// Process i
do {
    while (test_and_set(&lock))
        ; /* busy wait - spin until lock available */

    /* CRITICAL SECTION */

    lock = false;  /* release lock */

    /* REMAINDER SECTION */
} while (true);
```

Cara Kerja:

- Jika lock = false (tersedia): test_and_set() return false, keluar dari loop, masuk critical section
- Jika lock = true (terkunci): test_and_set() return true, tetap dalam loop waiting
- Setelah selesai: set lock = false untuk release

compare_and_swap() (1 point) Answer:

compare_and_swap() (CAS) melakukan operasi conditional assignment secara atomic:

```
    int compare_and_swap(int *value, int expected, int new_value)
        {
    int temp = *value;                  // 1. Simpan nilai current
    if (*value == expected)             // 2. Bandingkan dengan
       expected
        *value = new_value;             // 3. Jika sama, update ke
           new_value
    return temp;                        // 4. Return nilai original
}
```

Contoh Pemakaian:

```
// Inisialisasi
int lock = 0;  // 0 = available, 1 = locked

// Process i
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* busy wait until successfully acquire lock */

    /* CRITICAL SECTION */

    lock = 0;  /* release lock */

    /* REMAINDER SECTION */
} while (true);
```

Cara Kerja:

- Acquire: compare_and_swap(&lock, 0, 1)
  - Jika lock = 0: set ke 1, return 0 → berhasil acquire
  - Jika lock = 1: tidak berubah, return 1 → gagal, terus loop
- Release: Set lock = 0 langsung

2.4.6 Hardware Solutions Advantage (0.5 points)

Mengapa test_and_set() dan compare_and_swap() dapat digunakan sebagai solusi sinkronisasi pada sistem operasi nyata, sedangkan Peterson's Solution tidak?

Answer:

test_and_set() dan compare_and_swap() dapat digunakan pada sistem nyata karena:

- Hardware Support - Diimplementasikan langsung di level hardware sebagai instruksi atomic
- Memory Consistency - Hardware menjamin bahwa operasi ini tidak terpengaruh oleh cache coherency atau memory reordering
- Portability - Dapat digunakan untuk n proses, tidak terbatas pada 2 proses

Meskipun masih menggunakan busy waiting, hardware solutions ini lebih reliable dan dapat dikombinasikan dengan teknik lain (seperti blocking) untuk mengurangi pemborosan CPU.

2.5 glibc putchar Analysis (2.5 points)

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).

Answer:

Dimulai dengan locking untuk thread safety

```
  _IO_acquire_lock (_IO_stdout);
```

Kode diatas menacquire lock pada _IO_stdout, ini untuk memastikan thread safety ketika multiple threat mengakses output bersamaan.

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

Menulis karakter pada variabel c ke output, digunakan unlocked karena kita sudah mengakuisisi lock secara manual.

```
  _IO_release_lock (_IO_stdout);
```

Melepas lock pada _IO_stdout karena kita sudah menulis keoutput.

Mackro:

```
_IO_acquire_lock (_IO_stdout);  // Acquire lock
_IO_release_lock (_IO_stdout);  // Release lock
```

Makro ini menghandle:

- Mutex locking/unlocking dalam environment multithreaded
- No-op dalam single-threaded environment

Weak Alias

```
#if defined weak_alias && !defined _IO_MTSAFE_IO
#undef putchar_unlocked
weak_alias (putchar, putchar_unlocked)
#endif
```

- Dalam build single-threaded (!_IO_MTSAFE_IO), putchar_unlocked menjadi alias untuk putchar
- Ini karena tidak ada locking yang diperlukan dalam environment single-threaded
- weak_alias memungkinkan symbol yang sama digunakan dengan nama berbeda

Fungsi _IO_putc_unlocked:

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

Fungsi ini melakukan operasi penulisan aktual, karena mengamankan lock, operasi actual dilakukan oleh _IO_putc_unlocked yang lebih efisien karena tidak perlu menghandle locking lagi.

2.6 Kernel Exploitation (2.5 points)

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?

Answer:

Kernel exploitation adalah proses yang riweuh, melibatkan beberapa tahapan mendapat akses melalui exploit. Berikut penjelasan detail tahapan-tahapannya:

1. **Tahapan-Tahapan Kernel Exploitation**

   (a) **Initial Exploitation (Eksploitasi Awal)**

   Tahap ini melibatkan penemuan dan pemanfaatan vulnerability untuk mendapatkan akses awal ke sistem:

   - **Information Gathering**: Reconnaissance untuk memperoleh informasi target, versi kernel, dan service yang berjalan
   - **Vulnerability Discovery**: Mencari kelemahan dalam sistem target atau celah yang bisa diexploit.
   - **Exploit Development**: Mengembangkan atau mengadaptasi exploit untuk memanfaatkan vulnerability yang ditemukan
   - **Initial Access**: Mendapatkan eksekusi kode dengan privilege terbatas (biasanya user-level)

   (b) **Privilege Escalation (Eskalasi Privilese)**

   Setelah mendapat akses awal, penyerang perlu meningkatkan privilege mereka:

   - **Local Privilege Escalation**: Memanfaatkan vulnerability kernel untuk meningkatkan privilege dari user ke root/administrator
   - **Kernel Space Access**: Mendapatkan kemampuan untuk mengeksekusi kode dalam kernel space
   - **Security Bypass**: Melewati mekanisme keamanan seperti ASLR, DEP, SMEP/SMAP, atau Control Flow Integrity
   - **Ring 0 Access**: Mencapai level privilege tertinggi dalam arsitektur sistem (ex: root)

   (c) **Persistence Mechanisms (Mekanisme Persistensi)**

   Mempertahankan akses jangka panjang setelah mendapat privilege kernel:

   - **Rootkit Installation**: Menanamkan kode tersembunyi dalam kernel space
   - **System Modification**: Mengubah struktur data kernel, system calls, atau interrupt handlers
   - **Stealth Techniques**: Menyembunyikan proses, file, dan aktivitas dari deteksi sistem
   - **Backdoor Creation**: Membuat jalur akses alternatif yang sulit dideteksi

2. **Mengapa Multiple Stages Diperlukan?**

   (a) **Kompleksitas Keamanan Modern**
   Sistem operasi modern memiliki banyak lapisan pertahanan:

   - **Defense in Depth**: Banyak lapisan security memerlukan multiple bypass techniques
   - **Mitigation Technologies**: ASLR, DEP, Stack Canaries, dan CFI memerlukan teknik bypass yang berbeda
   - **Privilege Separation**: Pemisahan privilege user dan kernel memerlukan eskalasi bertahap

   (b) **Keterbatasan Exploit Tunggal**

   - **Single Point of Failure**: Satu vulnerability jarang memberikan kontrol penuh sistem
   - **Reliability Issues**: Multiple stages meningkatkan reliabilitas serangan
   - **Detection Avoidance**: Serangan bertahap lebih sulit dideteksi daripada serangan frontal

   (c) **Persistensi dan Stealth**

   - **Temporary Access**: Akses awal sering bersifat sementara atau tidak stabil
   - **Detection Evasion**: Memerlukan teknik khusus untuk menghindari deteksi jangka panjang
   - **Maintaining Control**: Sistem dapat di-restart atau di-patch, memerlukan mekanisme persistensi

3. **Cara Penyerang Mempertahankan Akses Setelah Mendapat Privilege Kernel**

   (a) **Rootkit Installation dan Modifikasi Kernel**
   Kernel-Level Rootkits:

   - **System Call Hooking**: Memodifikasi system call table untuk mengintercept dan memanipulasi panggilan sistem
   - **Inline Hooking**: Mengganti instruksi awal fungsi kernel dengan jump ke kode malicious
   - **Direct Kernel Object Manipulation (DKOM)**: Memodifikasi struktur data kernel secara langsung untuk menyembunyikan proses, file, atau network connections
   - **Interrupt Descriptor Table (IDT) Hooking**: Memodifikasi IDT untuk mengintercept interrupt handlers

   (b) **Persistence Mechanisms yang Tahan Reboot**
   Bootkit Implementation:

   - **Master Boot Record (MBR) Modification**: Menginfeksi MBR untuk memuat kode malicious sebelum OS
   - **UEFI Rootkit**: Menginfeksi UEFI firmware untuk persistensi level firmware

- **Boot Chain Infection**: Menyisipkan kode dalam bootloader (GRUB, Windows Boot Manager)

Kernel Module Persistence:

- **Legitimate Driver Abuse**: Menggunakan signed driver yang legitimate untuk memuat kode malicious
- **Driver Signing Bypass**: Memanfaatkan vulnerability dalam driver signing enforcement
- **Kernel Module Hiding**: Menghapus module dari linked list kernel untuk menghindari deteksi

4. **Perbedaan Antara Initial Exploitation, Privilege Escalation, dan Persistence Mechanisms**

   (a) **Initial Exploitation vs Privilege Escalation**
   Initial Exploitation:

   - Fokus pada mendapatkan foothold dalam sistem
   - Biasanya menghasilkan akses dengan privilege terbatas
   - Memanfaatkan vulnerability dalam aplikasi atau layanan
   - Tujuan: Mendapatkan eksekusi kode dalam sistem target

   Privilege Escalation:

   - Fokus pada peningkatan level akses
   - Memanfaatkan vulnerability dalam kernel atau sistem operasi
   - Mengubah konteks security dari user ke administrator/root
   - Tujuan: Mendapatkan kontrol penuh sistem

   (b) **Privilege Escalation vs Persistence Mechanisms**
   Privilege Escalation:

   - Bersifat sementara dan fokus pada peningkatan akses
   - Terjadi selama sesi aktif serangan
   - Memanfaatkan vulnerability untuk bypass security controls
   - Hasil: Akses administratif sementara

   Persistence Mechanisms:

   - Bersifat jangka panjang dan fokus pada pemeliharaan akses
   - Dirancang untuk bertahan setelah reboot atau pembaruan sistem
   - Menggunakan teknik stealth dan rootkit
   - Hasil: Akses tersembunyi yang berkelanjutan

2.7 File System Abstraction (2.5 points)

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.

2.7.1 Abstraction Component (1 point)

Jelaskan komponen abstraksi utama di dalam kernel sistem operasi yang memungkinkan ini terjadi.

Answer:

Komponen abstraksi utama yang memungkinkan hal ini terjadi adalah Virtual File System (VFS) di dalam kernel Linux. VFS bertindak sebagai lapisan abstraksi yang menyediakan interface yang seragam untuk berbagai jenis file system yang berbeda. VFS memungkinkan kernel Linux untuk:

- Mengelola berbagai file system (ext4, NTFS, FAT32, dll.) dengan cara yang konsisten
- Menyediakan system call yang sama (open, read, write, close) untuk semua jenis file system
- Melakukan mounting dan unmounting device dengan file system yang berbeda
- Menerjemahkan operasi file generik menjadi operasi spesifik untuk setiap file system melalui driver yang sesuai

2.7.2 Analogy (1.5 points)

Buatlah sebuah analogi yang menarik untuk menjelaskan cara kerja komponen ini..

Answer: Pertama didefinisikan analogi dengan berikut:

1. Operating System = Tracen Academy.
2. Virtual Filesystem(VFS) = Trainer
3. User = Pengguna.
4. Tipe FS = Uma Musume.

Di Tracen Academy (Sistem Operasi) terdapat berbagai Uma Musume dengan berbagai gaya berlari yang berbeda:

- Agnes Tachyon (Pace Chaser) = EXT4
- Sakura Bakushin 0 (Front Runner) = NTFS
- Symboli Rudolf (Late Surger) = FAT32

Sebagai player kita tidak tahu cara train para uma karena tiap uma memiliki gaya lari yang berbeda, namun karena Tracen Academy memiliki Trainer(VFS) yang memahami gaya lari tiap uma dan cara melatihnya, sehingga para player hanya perlu memberikan instruksi umum, seperti train speed, train power, etc. Tanpa perlu mengetahui secara detil caranya karena translasinya dihandle dengan Trainer. Sama dengan VFS dimana ketika ada pengguna yang menggunakan flashdisk dengan format NTFS dicolokkan ke Linux dengan format NTFS, VFS menjadi layer abstraksi dimana pengguna tidak perlu tahu implementasi detil karena sudah di handle VFS.

2.8 OS Concepts for 5-Year-Olds (2.5 points)

Jelaskan konsep - konsep sistem operasi berikut dalam bahasa yang dimengerti anak berumur 5 tahun!

2.8.1 Kernel (0.5 points)

Answer:

Kernel Bayangkan komputer seperti sebuah rumah besar. **Kernel** itu seperti Bapakmu yang mengatur semua hal di rumah. Dia yang memutuskan siapa boleh memakai kamar mandi dulu, siapa yang boleh menonton TV, dan memastikan semua orang di rumah mendapat makanan. Kernel mengatur semua program di komputer agar tidak berebutan dan semuanya berjalan dengan baik.

2.8.2 RAM (0.5 points)

Answer:

RAM itu seperti mejamu di kamar. Ketika kamu mau menggambar, kamu taruh kertas, krayon, dan pensil di atas meja supaya cepat diambil. RAM juga begitu, dia tempat komputer menaruh hal-hal yang sedang dipakai supaya bisa diambil dengan cepat. Kalau mejanya kecil, kamu cuma bisa taruh sedikit barang. Kalau mejanya besar, kamu bisa taruh banyak barang sekaligus sama seperti RAM, semakin besar RAM, komputer bisa naruh lebih banyak dan sebaliknya.

2.8.3 Filesystem (0.5 points)

Answer:

Filesystem itu seperti lemari yang punya banyak tempat kosong. Setiap barang punya tempat khusus, maian di laci atas, baju di laci tengah, celana di kotak bawah. Filesystem mengatur dimana semua file dan foto disimpan di komputer, jadi komputer tahu dimana harus mencari kalau kamu mau buka foto atau lagu.

2.8.4 Shell (0.5 points)

Answer:

Shell itu seperti remote control TV, Kamu bisa pencet tombol di remote untuk ganti channel atau naik-turunin volume. Shell adalah cara kita ”ngomong” sama komputer dengan menulis perintah, seperti ”buka folder A (cd A)” atau ”bikin file B (touch A)”. Komputer akan melakukan apa yang kita minta lewat shell ini.

2.8.5 Multiprocessing (0.5 points)

Answer:

Multiprocessing itu seperti mamak GOAT yang bisa masak sambil nyuci baju sambil nyapu rumah dalam waktu bersamaan! Komputer juga bisa melakukan banyak hal sekaligus, seperti putar musik sambil buka internet sambil main game.

2.9 Linux Boot Process (2 points)

Pada saat booting sebuah mesin dengan sistem operasi Linux, seringkali akan terlihat kata-kata “initramfs” atau “initial ramdisk”

2.9.1 Boot Stage Process (0.5 points)

Jelaskan proses apa yang sedang terjadi pada tahap booting ini!

Answer:

Pada tahap ini, kernel sedang memuat dan menjalankan initramfs (initial RAM filesystem) atau initial ramdisk. Ini adalah filesystem sementara yang dimuat ke dalam RAM yang berisi file-file dan program minimal yang diperlukan untuk melanjutkan proses boot seperti mouting root filesystem. Initramfs berperan sebagai ”jembatan” antara kernel yang baru dimuat dengan filesystem root yang sebenarnya.

2.9.2 Why Initial Ramdisk? (1 point)

Mengapa sistem harus menggunakan initial ramdisk atau initramfs? Mengapa sistem tidak langsung me-mount filesystem sebenarnya?

Answer:

Digunakan initramfs karena beberapa hal:

- Kernel Linux yang dimuat pertama kali biasanya adalah kernel generik yang tidak memiliki semua driver yang diperlukan untuk mengakses berbagai jenis storage device (SATA, SCSI, RAID, LVM, encrypted filesystems, dll). Initramfs berisi modul-modul driver yang diperlukan untuk mengenali dan mengakses perangkat penyimpanan tempat filesystem root berada.

- Memasukkan semua driver langsung pada kernel dapat membuat kernel bloating dan menambah waktu booting dan semakin besar memory usage, iniramfs membuat driver hanya dimasukkan ketika perlu saja.

- Initramfs memungkinkan untuk melakukan hardware detection, sehingga tidak perlu manual menghidupkan/menambahkan driver yang diperlukan.

2.9.3 Corruption Consequences (0.5 points)

Apa yang biasanya akan terjadi jika initial ramdisk atau initramfs rusak atau tidak ditemukan?

Answer:

Jika initramfs rusak atau tidak ditemukan, sistem akan mengalami kernel panic dengan pesan error seperti:

- ”VFS: Cannot open root device” - kernel tidak dapat mengakses perangkat root

- ”Kernel panic - not syncing: VFS: Unable to mount root fs” - kernel gagal mount filesystem root Sistem akan berhenti booting dan menampilkan pesan error

Dalam kondisi ini, sistem tidak akan bisa melanjutkan proses boot karena kernel tidak memiliki driver atau tools yang diperlukan untuk mengakses filesystem root yang sebenarnya. Sehingga kernel menjadi useless karena tidak bisa melakukan apa-apa untuk melanjutkan sistem booting dan boot gagal.

2.10 Copy-on-Write in fork() (2 points)

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?"

Answer:

Misal Ghana dan Fariz patungan untuk membuat perpustakaan lengkap. Tiba-tiba Ghana ingin memiliki perpustakaan sendiri yang identik. Dengan cara tradisional, Ghana harus memfotokopi seluruh perpustakaan dari awal sampai akhir yang membutuhkan waktu berhari-hari dan biaya sangat mahal.

Dengan sistem copy-on-write, Ghana langsung bisa mengklaim "punya perpustakaan sendiri" dalam hitungan detik. Kedua perpustakaan sebenarnya masih berbagi buku-buku fisik yang sama, hanya ada papan nama berbeda di pintu masuk. Ketika Ghana ingin merubah atau menambah sesuatu pada suatu halaman pada buku, baru saat itu halaman tersebut benar-benar difotokopi dan menjadi miliknya. Buku-buku yang hanya dibaca tetap shared dengan Fariz.

Jadi Ghana mendapat perpustakaan "pribadi" secara instan tanpa biaya fotokopi di awal, dan hanya membayar untuk halaman yang benar-benar dia ubah.

3 Platform Technology

"POV: Me after ruling the cloud infrastructure"

3.1 Unikernel vs Container Security (6 points)

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.

3.1.1 Isolation Model Differences (2 points)

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?

Answer:

Model isolasi container menggunakan isolasi pada level *namespace dan cgroups* di atas kernel host yang sama. Dalam Kubernetes, auth service berjalan dalam container yang berbagi kernel Linux dengan container lain di node yang sama. Isolasi dicapai melalui process namespaces (PID, network, mount), resource limits via cgroups, dan security contexts dengan capabilities.

Sebaliknya, unikernel menggunakan isolasi pada level *hypervisor/VM*. Auth service akan dikompilasi bersama minimal OS kernel menjadi single bootable image yang berjalan langsung di atas hypervisor tanpa traditional OS layer.

Dampaknya adalah, container memiliki permukaan serangan yang lebih luas karena karena berjalan pada kernel yang sama, sehingga shared kernel vulnerabilities dapat

mempengaruhi semua container, container runtime (Docker/containerd) menjadi attack vector, syscall interface yang luas (300+ syscalls Linux), dan risiko privilege escalation melalui kernel exploits.

Unikernel memiliki permukaan serangan minimal dengan hanya syscalls yang dibutuhkan aplikasi (biasanya < 50), tidak ada shell, package manager, atau utilities, VM-level isolation yang melindungi dari shared kernel attacks, dan minimal attack surface karena hanya berisi aplikasi code plus minimal runtime.t

3.1.2 Integration Challenges (2 points)

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.

Answer:

Challenges & Solutions

1. Logging: Biasanya unikernel tidak memiliki mekanisme logging seperti syslog atau filed-based logging, karena unikernel dirancang lightweight untuk mengurangi overhead. Sedangkan k8s mengharapkan pod untuk mengeluarkan log ke/stderr yang kemudian di collect on runtime container. Jika stream tersebut tidak dijembatani maka log unikernel tidak bisa diakses.

   - Solusi:

   - Gunakan Logging Kustom Kustom dalam kode Unikernel
   - Gunakan Logging eksternal seperti Loki atau ELK dengan mengirim langsung log ke layanan tersebut dengan UDP atau HTTP.

2. Monitoring: Unikernel tidak memiliki antarmuka OS tradisional seperti /proc atau panggilan sistem untuk pengumpulan metrik yang dibutuhkan k8s.

   - Solusi:

   - Kustom monitoring dengan membuat endpoint (ex:/metrics) yang mengirimkan data seperti jumlah request atau memory usage.

3. Debugging: Unikernel karena sangat lightweight tidak memiliki layanan interface shell interaktif, debugger, bahkan dynamic lingking mengakibatkan debugging kurbenetes biasa seperti kubectl exec/debug tidak bisa digunakan.

   - Solusi:

   - Buat versi untuk debugging dari unikernel dengan menambahkan kemapuan untuk debugging (ex: verbose, memory dump) yang bisa digunakan di luar production.

3.1.3 Migration Decision (2 points)

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.

Answer:

Bagi saya belum perlu, karena beberapa hal:

Pros jika migrate:

- Keamanan yang tinggi - seperti dijelaskan sebelumnya, attack surface pada unikernel lebih sedikit.
- High Performance - karena lightweight mengakibatkan performa juga lebih cepat karena minimnya overhead.
- Small Size - karena lightweight size dari image lebih kecil sehingga jika memerlukan envinronment k8s yang membutuhkan banyak instance sangat baik.
- Isolasi - karena sebagai VM terisolasi, meberikan isolasi yang lebih baik dibanding kontainer. bagus untuk critical service.

Cons jika Migrate:

- Kompleksitas Pengembangan dan Debugging - karena unikernel tidak memiliki shell interactive akan sulit melakukan debugging pada auth service nanti yang akan menyulitkan proses development serta maintenance
- Ekosistem Minim - Dukungan untuk pustaka atau framework autentikasi (misalnya, Keycloak, OpenID Connect) mungkin terbatas atau perlu manual port.
- Overhead Operasional - team lain seperti DevOps perlu belajar mengelola stack yang baru.

Dengan berbagai pros dan cons, saya merasa consnya lebih berasa dibanding pros, selain itu diperlukan juga waktu yang tidak cepat untuk melakukan migrating. Sehingga, lebih baik tidak migrate.

3.2 Docker Environment Consistency (2.5 points)

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!

Answer:

Docker menjamin konsistensi environment aplikasi melalui containerization yang mengemas aplikasi beserta semua dependensi, library, dan konfigurasi ke dalam sebuah container yang terisolasi dan portable. Container image yang bersifat immutable berisi snapshot lengkap dari environment aplikasi, mencakup sistem operasi, runtime, library, dependensi, dan kode aplikasi dalam versi yang tepat sama

Konsistensi environment dibutuhkan dalam software developement modern karena beberapa faktor. Tim developer yang bekerja di berbagai laptop/pc dengan OS dan konfigurasi berbeda membutuhkan environment yang seragam untuk menghindari masalah "works on my machine" yang sering menghambat development. Proses CI/CD memerlukan environment yang konsisten dari development, testing, hingga production untuk memastikan behavior yang sama untuk memastikan kualitas. Dalam arsitektur microservices dan aplikasi terdistribusi, setiap service harus dapat berjalan secara konsisten di berbagai node dan cluster untuk mendukung scalability dan maintainability sistem.

Manajemen environment yang buruk menimbulkan risiko dalam pengembangan perangkat lunak, termasuk *environment drift* yang menyebabkan perbedaan konfigurasi tak terdokumentasi dan bug pada production yang sulit dideteksi, *dependency hell* akibat konflik versi

library/package terutama di ekosistem Python/Node.js, configuration management yang rumit dengan setting berbeda antar environment, serta *deployment inconsistency* yang membuat aplikasi gagal di staging/production meski berjalan sempurna di development.

Docker mengatasi masalah-masalah tersebut dengan menyediakan standardisasi environment melalui container yang identik dari image yang sama, mengisolasi dependencies untuk menghindari konflik dengan system libraries, mengoptimalkan image size dengan multi-stage builds, memfasilitasi orchestration multi-container via Docker Compose, dan memungkinkan sharing serta versioning image terpusat melalui registry system.