HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
KHOA ĐÀO TẠO SAU ĐẠI HỌC

BÁO CÁO BÀI TẬP CÁC HỆ THỐNG PHÂN TÁN

NỘI DUNG:

BÀI TẬP CHƯƠNG 2, 3

NGÀNH:	**HỆ THỐNG THÔNG TIN**
LỚP:	**M25CQHT01-B**
GIẢNG VIÊN:	**T.S KIM NGỌC BÁCH**
HỌC VIÊN THỰC HIỆN:	**B25CHHT056 - VŨ MINH TOÀN**

Hà Nội, 12/2025

Mục Lục

Câu 1:

Một công ty đang sử dụng một hệ thống quản lý kho (Warehouse Management System – WMS) cũ, trong đó giao diện API không tương thích với hệ thống thương mại điện tử mới của họ.

Hãy áp dụng khái niệm wrapper để đề xuất cách tích hợp hai hệ thống này.

Mô tả cách wrapper giúp giải quyết vấn đề tương thích giao diện và so sánh chi phí phát triển nếu sử dụng $O(N^2)$ wrappers trực tiếp giữa các hệ thống so với giải pháp message broker.

Trả lời:

1. Đề xuất giải pháp sử dụng Wrapper

Wrapper Pattern là một design pattern cho phép chuyển đổi interface của một hệ thống thành giao diện mà hệ thống khác mong đợi, hoạt động như một lớp trung gian (adapter) giữa hai hệ thống không tương thích.

Kiến trúc đề xuất:

```
[E-commerce System] <---> [WMS Wrapper] <---> [Legacy WMS]
```

Các thành phần:

1. **WMS Wrapper Service**: Một service trung gian đóng vai trò adapter

- Nhận request từ hệ thống e-commerce (format mới)
- Chuyển đổi request sang format của WMS cũ
- Gọi API của WMS cũ
- Nhận response từ WMS cũ
- Chuyển đổi response về format mà e-commerce hiểu được
- Trả về cho hệ thống e-commerce

2. **Interface Mapping Layer**: Ánh xạ giữa các API endpoints

```
E-commerce API → WMS Wrapper → Legacy WMS API
POST /orders/inventory → Transform → GET /stock/check?sku=xxx
GET /products/stock → Transform → POST /inventory/query
```

3. **Data Transformation Layer**: Chuyển đổi cấu trúc dữ liệu

```
// E-commerce format (input)
{
"productId": "PROD-123",
"quantity": 10
}
// WMS format (output after transformation)
{
"sku": "PROD-123",
"qty": 10,
"warehouse_id": "WH-01"
}
```

2. Cách Wrapper giải quyết vấn đề tương thích giao diện

a) Protocol Adaptation:

- WMS cũ có thể dùng SOAP, XML-RPC
- E-commerce mới dùng RESTful JSON
- Wrapper chuyển đổi giữa các protocol khác nhau

b) Data Format Transformation:

- Chuyển đổi cấu trúc dữ liệu (JSON XML)
- Mapping tên trường (productId → sku)

• Chuyển đổi kiểu dữ liệu (string integer)

c) Business Logic Translation:

• WMS cũ có thể yêu cầu nhiều API calls để hoàn thành một nghiệp vụ
• Wrapper có thể gộp nhiều calls thành một endpoint duy nhất cho e-commerce

d) Error Handling:

• Chuẩn hóa error codes và messages
• Retry logic cho các lỗi tạm thời
• Fallback mechanisms

e) Security & Authentication:

• E-commerce dùng OAuth 2.0/JWT
• WMS cũ dùng Basic Auth hoặc API Key
• Wrapper xử lý việc chuyển đổi authentication

3. So sánh chi phí xử lý: O(N^2) Wrappers vs Message Broker

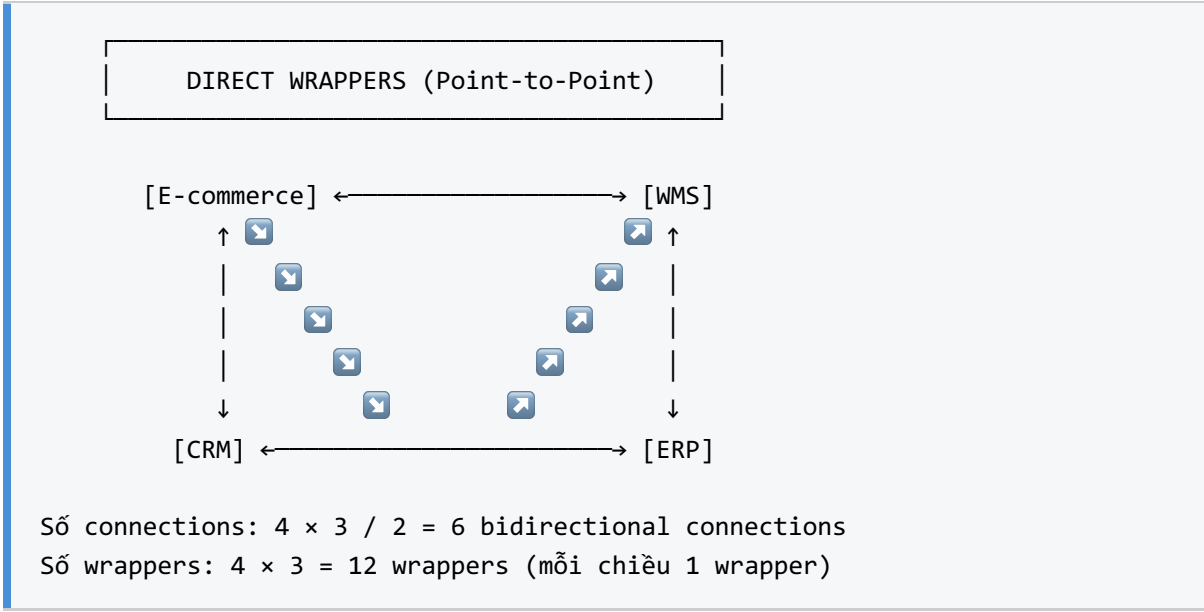
Giải pháp 1: Direct Wrappers - O(N^2)

Mô hình:

```
Với N hệ thống, mỗi hệ thống cần wrapper riêng để giao tiếp với (N-1) hệ thống khác
→ Tổng số wrappers = N × (N-1) = O(N²)
→ Tổng số connections (kết nối) = N × (N-1) / 2 = O(N²)
```

Ví dụ với 4 hệ thống:

• E-commerce System
• WMS (Warehouse)
• CRM (Customer)
• ERP (Enterprise Resource Planning)



Chi phí xử lý (Processing Cost):

Metric	Giá trị	Công thức
Số wrappers	12	N × (N-1) = 4 × 3
Số connections	6	N × (N-1) / 2

Transformations per message	1	Direct A→B
Hops per message	1	Source → Destination

Phân tích chi phí xử lý:

1. **Message Overhead per Request:**

```
E-commerce → WMS:
┌──────────────┐   ┌──────────────┐   ┌──────────┐
│ E-commerce │ → │ WMS Wrapper │ → │ WMS      │
└──────────────┘   └──────────────┘   └──────────┘
Processing: 1 transformation + 1 hop
Latency: ~5-10ms (direct connection)
```

2. **Tổng số processing units khi scale:**

```
N = 4: 4 × 3 = 12 wrappers
N = 5: 5 × 4 = 20 wrappers (+67%)
N = 6: 6 × 5 = 30 wrappers (+50%)
N = 10: 10 × 9 = 90 wrappers
N = 20: 20 × 19 = 380 wrappers
```

3. **Độ phức tạp khi thay đổi API:**

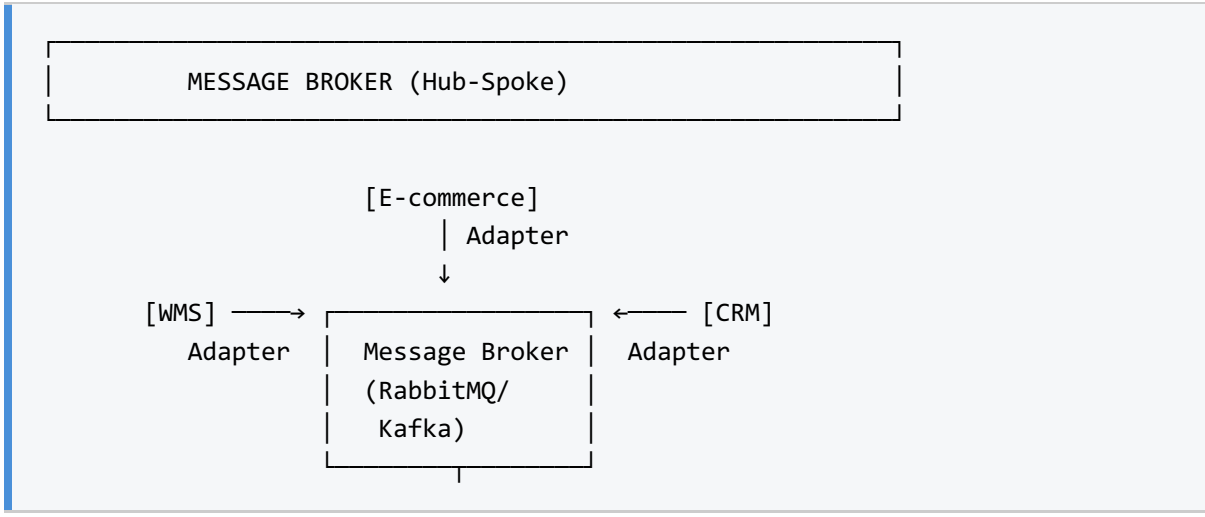
```
Khi WMS thay đổi API:
→ Phải update 3 wrappers (từ E-commerce, CRM, ERP)
→ Chi phí xử lý: O(N-1) modifications
```

Giải pháp 2: Message Broker - O(N)

Mô hình:

```
Mỗi hệ thống chỉ cần 1 adapter để kết nối với Message Broker
→ Tổng số adapters = N = O(N)
→ Tổng số connections = N (mỗi system → broker)
```

Kiến trúc:



```
                          |
                          ↓ Adapter
                        [ERP]

      Số connections: 4 (mỗi system 1 connection)
      Số adapters: 4 (mỗi system 1 adapter)
```

Chi phí xử lý (Processing Cost):

Metric	Giá trị	Công thức
Số adapters	4	N
Số connections	4	N
Transformations per message	2	Source adapter + Dest adapter
Hops per message	2	Source → Broker → Destination

Phân tích chi phí xử lý:

1. **Message Overhead per Request:**

```
E-commerce → WMS:

┌──────────┐  ┌──────────┐  ┌────────┐  ┌──────────┐  ┌────────┐
│ E-commerce │ → │Adapter 1│ → │ Broker │ → │Adapter 2│ → │ WMS    │
└──────────┘  └──────────┘  └────────┘  └──────────┘  └────────┘

Processing: 2 transformations + 2 hops + queue operations
Latency: ~10-50ms (qua broker)
```

2. **Tổng số processing units khi scale:**

```
N = 4: 4 adapters + 1 broker = 5 components
N = 5: 5 adapters + 1 broker = 6 components (+20%)
N = 6: 6 adapters + 1 broker = 7 components (+17%)
N = 10: 10 adapters + 1 broker = 11 components
N = 20: 20 adapters + 1 broker = 21 components
```

3. **Độ phức tạp khi thay đổi API:**

```
Khi WMS thay đổi API:
→ Chỉ cần update 1 adapter (WMS adapter)
→ Chi phí xử lý: O(1) modification
```

Bảng so sánh chi phí xử lý

Tiêu chí	O(N²) Direct Wrappers	O(N) Message Broker
Số components (N=4)	12 wrappers	4 adapters + 1 broker
Số connections	N×(N-1)/2 = 6	N = 4
Hops per message	1	2
Transformations/msg	1	2
Latency per request	Thấp (5-10ms)	Cao hơn (10-50ms)
Khi thêm 1 system	+2N wrappers	+1 adapter

Khi sửa 1 API	O(N-1) updates	O(1) update
Scaling complexity	$O(N^2)$	O(N)
Bottleneck	Không	Broker (cần cluster)

Phân tích chi tiết theo N

```
CHI PHÍ XỬ LÝ THEO SỐ HỆ THỐNG:
─────────────────────────────────

N (systems) │ Direct Wrappers │ Message Broker │ Winner
────────────┼─────────────────┼────────────────┼──────────────
     2      │     2 wrappers  │   2 + broker   │ Direct
     3      │     6 wrappers  │   3 + broker   │ ~Same
     4      │    12 wrappers  │   4 + broker   │ Broker
     5      │    20 wrappers  │   5 + broker   │ Broker
    10      │    90 wrappers  │  10 + broker   │ Broker
    20      │   380 wrappers  │  20 + broker   │ Broker

Break-even point: N ≈ 3-4 systems
```

```
BIỂU ĐỒ SO SÁNH:
────────────────

Components
    │
400 ┤                                          ╱ O(N²)
    │                                        ╱
300 ┤                                      ╱
    │                                    ╱
200 ┤                                  ╱
    │                                ╱
100 ┤                    ╱────────╱
    │          ╱───────╱
 50 ┤     ╱──╱
    │  ╱╱       ──────────────────────────────── O(N)
    └──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──→ N
    0  2  4  6  8  10 12 14 16 18 20
```

Trade-offs chi phí xử lý

Aspect	Direct Wrappers	Message Broker
Per-message latency	Thấp	Cao hơn (extra hop)
Per-message throughput	Cao	Phụ thuộc broker
Total system complexity	$O(N^2)$	O(N)
API change impact	O(N-1)	O(1)
New system integration	O(2N)	O(1)
Single point of failure	Không	Có (broker)

Message ordering	Khó đảm bảo	Dễ (queue)
Async processing	Khó	Native support

Kết luận và Khuyến nghị

Về chi phí xử lý:

Số hệ thống	Khuyến nghị	Lý do
N = 2	Direct Wrapper	Đơn giản, latency thấp
N = 3-4	Tùy yêu cầu	Break-even point
N ≥ 5	Message Broker	O(N) << O(N²)

Cho bài toán hiện tại (2 hệ thống: E-commerce + WMS):

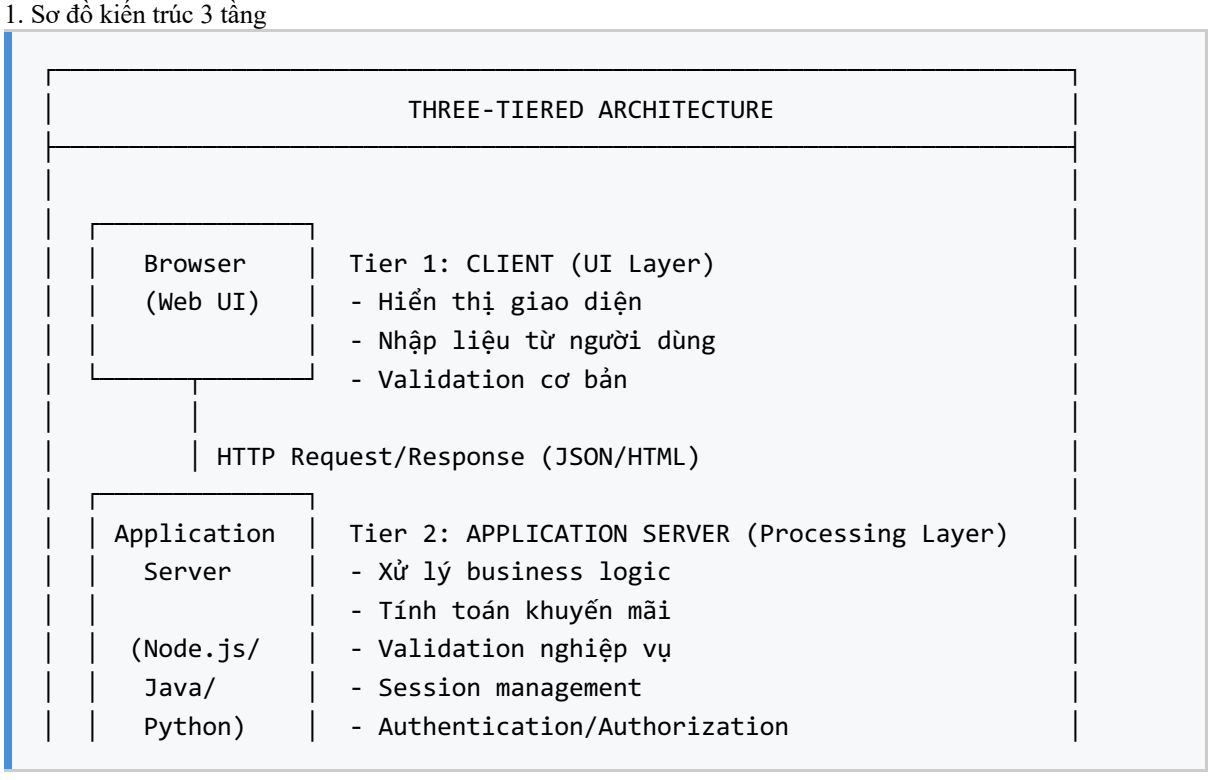
- **Khuyến nghị: Direct Wrapper**
- Chi phí xử lý: 1 wrapper, 1 hop, latency thấp
- Đơn giản, không cần infrastructure phức tạp

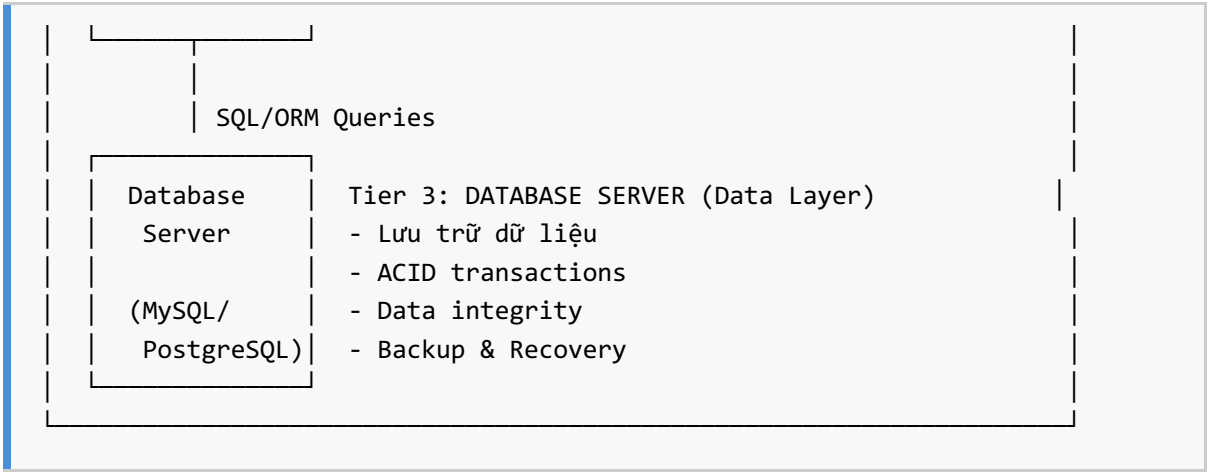
Khi mở rộng (≥5 hệ thống):

- **Khuyến nghị: Message Broker**
- Chi phí xử lý giảm từ O(N²) xuống O(N)
- Trade-off: latency cao hơn (2 hops) nhưng scalability tốt hơn
- Khi API thay đổi: chỉ update O(1) adapter thay vì O(N-1) wrappers

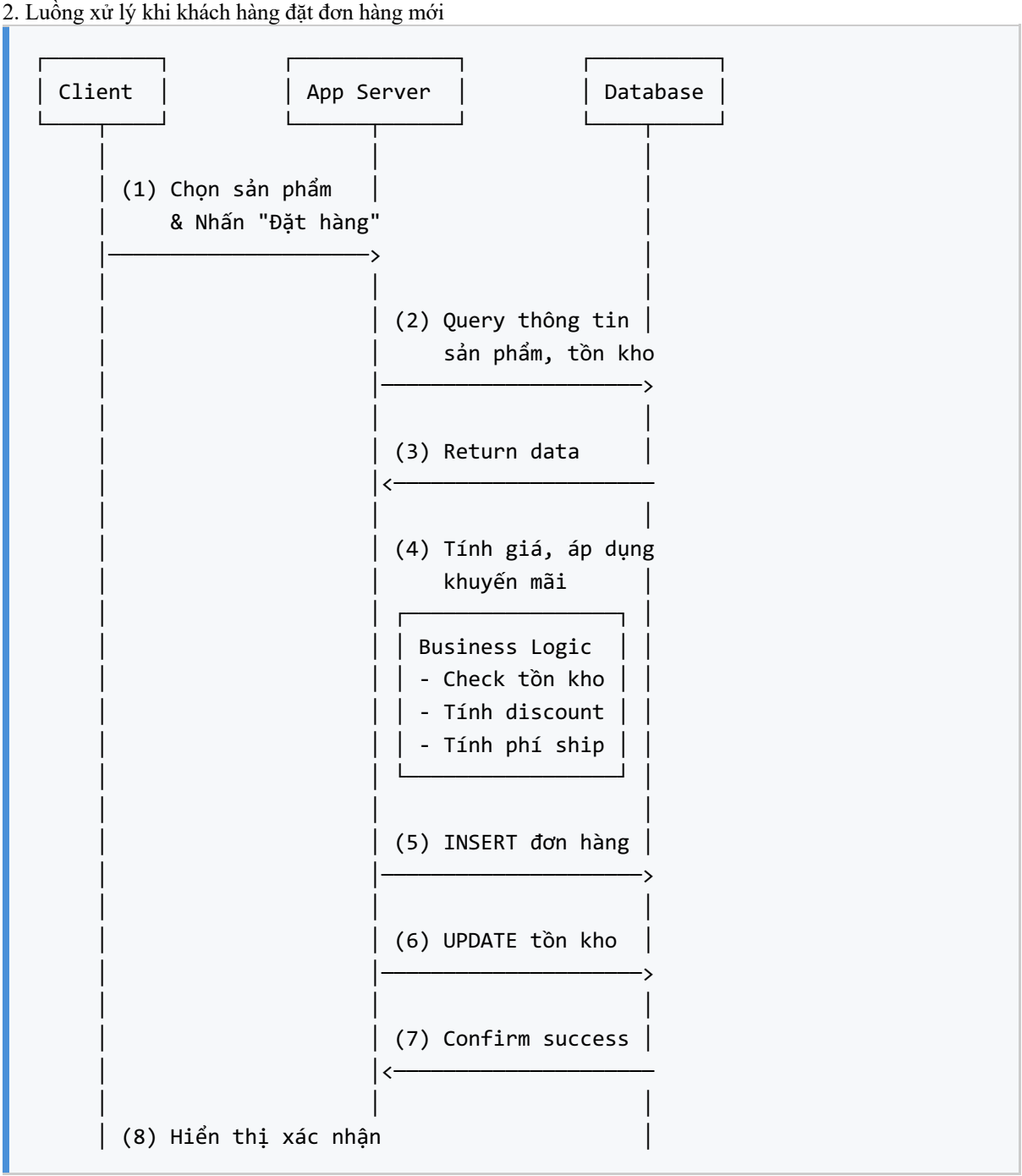
Câu 2:

Một hệ thống bán hàng trực tuyến được thiết kế theo kiến trúc 3 tầng (three-tiered architecture) gồm:
Client (UI layer): giao diện web cho khách hàng đặt hàng.
Application server (Processing layer): xử lý đơn hàng, tính toán khuyến mãi.
Database server (Data layer): lưu trữ sản phẩm và giao dịch.
Hãy mô tả:
Luồng xử lý khi khách hàng đặt một đơn hàng mới.
Ưu điểm của việc tách application server ra thành một tầng riêng thay vì để toàn bộ xử lý ở client hoặc database server.
Trả lời:
1. Sơ đồ kiến trúc 3 tầng





2. Luồng xử lý khi khách hàng đặt đơn hàng mới



Chi tiết các bước:

Bước	Tầng	Mô tả
(1)	Client → App Server	Khách hàng điền thông tin đơn hàng (sản phẩm, số lượng, địa chỉ) và nhấn "Đặt hàng". Browser gửi HTTP POST request đến App Server
(2)	App Server → Database	App Server query thông tin sản phẩm, giá, số lượng tồn kho
(3)	Database → App Server	Database trả về dữ liệu sản phẩm và inventory
(4)	App Server (internal)	Xử lý business logic: Kiểm tra tồn kho đủ không, tính tổng tiền, áp dụng mã khuyến mãi (giảm 10%, free ship), tính phí vận chuyển
(5)	App Server → Database	INSERT đơn hàng mới vào bảng orders và order_items
(6)	App Server → Database	UPDATE giảm số lượng tồn kho trong bảng inventory
(7)	Database → App Server	Xác nhận transaction thành công
(8)	App Server → Client	Trả về trang xác nhận đơn hàng với mã đơn, tổng tiền, thời gian giao hàng dự kiến

3. Ưu điểm của việc tách Application Server thành tầng riêng

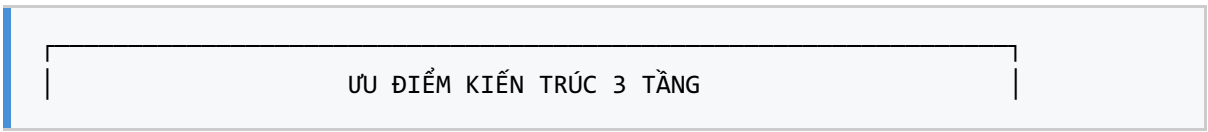
So sánh với phương án để xử lý ở Client:

Tiêu chí	Xử lý ở Client	Tách App Server riêng
Bảo mật	Kém - logic lộ ra ngoài, dễ bị hack giá, fake discount	Tốt - logic ẩn phía server, client không can thiệp được
Validation	Dễ bypass - user có thể sửa JavaScript	Đáng tin cậy - server kiểm tra lại mọi thứ
Hiệu năng client	Nặng - browser phải xử lý nhiều	Nhẹ - chỉ render UI
Cập nhật logic	Khó - phải chờ user refresh/clear cache	Dễ - deploy server là xong
Đa nền tảng	Duplicate code cho web, mobile, API	Một backend phục vụ tất cả

So sánh với phương án để xử lý ở Database (Stored Procedures):

Tiêu chí	Xử lý ở Database	Tách App Server riêng
Khả năng mở rộng	Khó - database là bottleneck	Dễ - scale horizontal nhiều app servers
Bảo trì code	Khó - SQL/PL khó test, debug	Dễ - dùng ngôn ngữ phổ biến, có IDE hỗ trợ
Tích hợp bên ngoài	Rất khó - gọi API từ stored proc phức tạp	Dễ - call REST API, message queue
Vendor lock-in	Cao - phụ thuộc DB vendor	Thấp - có thể đổi database
Tài nguyên DB	Lãng phí CPU cho logic	DB tập trung vào I/O, query

Tổng hợp ưu điểm của kiến trúc 3 tầng:



```
┌──────────────────────────────────────────────────────────────────┐
│                                                                  │
│  1. SEPARATION OF CONCERNS (Phân tách trách nhiệm)               │
│     - UI: chỉ lo hiển thị                                        │
│     - App Server: chỉ lo business logic                          │
│     - Database: chỉ lo lưu trữ                                   │
│                                                                  │
│  2. SCALABILITY (Khả năng mở rộng)                               │
│     - Scale từng tầng độc lập                                    │
│     - Thêm app server khi traffic cao                            │
│     - Thêm read replica cho database                             │
│                                                                  │
│  3. MAINTAINABILITY (Dễ bảo trì)                                 │
│     - Sửa UI không ảnh hưởng backend                             │
│     - Đổi database không cần sửa UI                              │
│     - Team chuyên biệt cho từng tầng                             │
│                                                                  │
│  4. SECURITY (Bảo mật)                                           │
│     - Database không expose trực tiếp ra internet                │
│     - Business logic được bảo vệ phía server                     │
│     - Dễ implement authentication/authorization                  │
│                                                                  │
│  5. REUSABILITY (Tái sử dụng)                                    │
│     - Một App Server phục vụ: Web, Mobile App, API               │
│     - Shared business logic across platforms                     │
│                                                                  │
│  6. TECHNOLOGY FLEXIBILITY                                       │
│     - Mỗi tầng có thể dùng công nghệ phù hợp nhất                │
│     - Frontend: React/Vue, Backend: Java/Node, DB: PostgreSQL    │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

4. Ví dụ thực tế với hệ thống bán hàng

```
Ví dụ: Khách hàng mua 2 áo với mã giảm giá "SALE20"
 CLIENT (Browser):
┌──────────────────────────────────┐
│  Giỏ hàng:                       │
│  - Áo thun x2: hiển thị 200.000đ │
│  - Mã giảm giá: [SALE20]         │
│  - [Đặt hàng]                    │
└──────────────────────────────────┘
                  │
            POST /api/orders
APPLICATION SERVER:
┌──────────────────────────────────┐
│  1. Validate mã SALE20 còn hạn?  │
│  2. Check tồn kho >= 2?          │
```

```
│  3. Tính: 200.000 - 20% = 160.000đ  │
│  4. Phí ship: 30.000đ               │
│  5. Tổng: 190.000đ                  │
│  6. Tạo đơn hàng                    │
└─────────────────────────────────────┘
                    │
            SQL Transaction
DATABASE:
┌─────────────────────────────────────┐
│  BEGIN TRANSACTION;                 │
│  INSERT INTO orders (...);          │
│  INSERT INTO order_items (...);     │
│  UPDATE inventory SET qty = qty - 2;│
│  UPDATE promotions SET used = +1;   │
│  COMMIT;                            │
└─────────────────────────────────────┘
```

5. Kết luận

Kiến trúc 3 tầng là pattern phổ biến và được chứng minh hiệu quả cho các hệ thống web, đặc biệt là e-commerce. Việc tách **Application Server** thành tầng riêng mang lại:

- **Bảo mật**: Logic nghiệp vụ được bảo vệ
- **Khả năng mở rộng**: Scale horizontal dễ dàng
- **Bảo trì**: Code dễ đọc, test, và deploy
- **Linh hoạt**: Phục vụ nhiều loại client (web, mobile, API)

Câu 3:

Trong vòng Chord với m = 5 và các nút hiện có {1, 4, 9, 11, 14, 18, 20, 21, 28}, hãy:

Xác định succ(7), succ(22) và succ(30).

Giả sử node 9 thực hiện tra cứu key = 3, hãy mô tả chi tiết các bước chuyển tiếp yêu cầu qua các shortcut (theo hình 2.19) cho đến khi tìm được node chịu trách nhiệm.

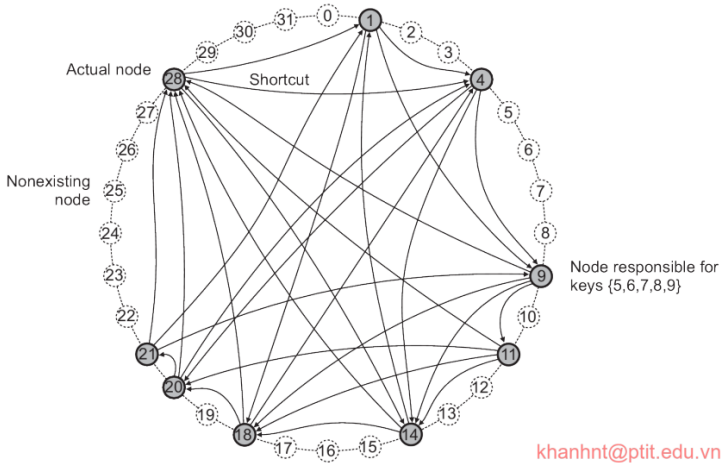


Figure 2.19: The organization of nodes and data items in Chord.

Trả lời:

1. Tổng quan về Chord DHT

Chord là một giao thức Distributed Hash Table (DHT) có cấu trúc, sử dụng consistent hashing để phân phối keys cho các nodes trong một vòng tròn logic.

Các thông số:

- **m = 5** → Không gian định danh: 2^5 = **32** (từ 0 đến 31)
- **Nodes hiện có:** {1, 4, 9, 11, 14, 18, 20, 21, 28}
- **Số nodes:** 9

2. Biểu diễn vòng Chord (m=5)

```
                          0
                     31       1 ←[NODE]
                  30            2
               29                  3
             28 ←[NODE]              4 ←[NODE]
            27                        5
           26                          6
          25                            7
         24                              8
         23                              9 ←[NODE]
          22                            10
           21 ←[NODE]                   11 ←[NODE]
            20 ←[NODE]                12
             19                      13
               18 ←[NODE]          14 ←[NODE]
                  17             15
                     16      16

Nodes: {1, 4, 9, 11, 14, 18, 20, 21, 28}
```

3. Xác định Successor

Định nghĩa: succ(k) = node đầu tiên có ID ≥ k khi đi theo chiều kim đồng hồ trên vòng Chord.

succ(7) = ?

Đi từ vị trí 7 theo chiều kim đồng hồ, tìm node đầu tiên:

- Vị trí 7 → không có node
- Vị trí 8 → không có node
- Vị trí 9 → **có NODE 9**

→ **succ(7) = 9**

succ(22) = ?

Đi từ vị trí 22 theo chiều kim đồng hồ:

- Vị trí 22 → không có node
- Vị trí 23 → không có node
- ...
- Vị trí 28 → **có NODE 28**

→ **succ(22) = 28**

succ(30) = ?

Đi từ vị trí 30 theo chiều kim đồng hồ:

- Vị trí 30 → không có node
- Vị trí 31 → không có node
- Vị trí 0 → không có node (wrap around)
- Vị trí 1 → **có NODE 1**

→ **succ(30) = 1** (wrap around qua 0)

4. Bảng tóm tắt Successor

Key (k)	Tìm kiếm	Kết quả
succ(7)	7 → 8 → 9	9
succ(22)	22 → 23 → ... → 28	28
succ(30)	30 → 31 → 0 → 1	1

5. Finger Table của Node 9

Để tra cứu key = 3 từ node 9, cần xây dựng **Finger Table** của node 9.

Công thức: finger[i] = succ((n + 2^(i-1)) mod 2^m) với i = 1, 2, ..., m

Với n = 9, m = 5:

i	start = (9 + 2^(i-1)) mod 32	finger[i] = succ(start)
1	(9 + 1) mod 32 = 10	succ(10) = 11
2	(9 + 2) mod 32 = 11	succ(11) = 11
3	(9 + 4) mod 32 = 13	succ(13) = 14
4	(9 + 8) mod 32 = 17	succ(17) = 18
5	(9 + 16) mod 32 = 25	succ(25) = 28

Finger Table của Node 9:

```
┌─────┬───────┬─────────────┐
│  i  │ start │ finger[i]   │
├─────┼───────┼─────────────┤
│  1  │  10   │     11      │
│  2  │  11   │     11      │
│  3  │  13   │     14      │
│  4  │  17   │     18      │
│  5  │  25   │     28      │
└─────┴───────┴─────────────┘
```

6. Tra cứu Key = 3 từ Node 9

Mục tiêu: Tìm node chịu trách nhiệm cho key = 3 (tức là succ(3) = **4**)

Thuật toán Chord Lookup:

1. Nếu key nằm trong khoảng (n, successor], trả về successor
2. Ngược lại, chuyển tiếp đến node gần key nhất trong finger table

Bước 1: Tại Node 9

- Key = 3
- Node 9 kiểm tra: key = 3 có nằm trong (9, succ(9)] = (9, 11] không?
- Vì không gian là vòng tròn, khoảng (9, 11] không chứa 3
- **Cần chuyển tiếp request**

Tìm finger lớn nhất < key = 3:

- Duyệt finger table từ i=5 xuống i=1:

- finger[5] = 28: 28 > 3? Trong không gian vòng, cần xét khoảng (9, 3)
- Khoảng (9, 3) theo chiều kim đồng hồ: 10, 11, ..., 31, 0, 1, 2
- finger[5] = 28 ∈ (9, 3)? **Có** → Chuyển đến **node 28**

Bước 2: Tại Node 28

Cần xây dựng Finger Table của Node 28:

i	start = (28 + 2^(i-1)) mod 32	finger[i]
1	29	succ(29) = 1
2	30	succ(30) = 1
3	0	succ(0) = 1
4	4	succ(4) = 4
5	12	succ(12) = 14

- Node 28 kiểm tra: key = 3 ∈ (28, succ(28)] = (28, 1]?
- Khoảng (28, 1] theo chiều kim đồng hồ: 29, 30, 31, 0, 1
- 3 ∉ (28, 1] → **Cần chuyển tiếp**

Tìm finger gần key nhất trong khoảng (28, 3):

- finger[5] = 14: 14 ∈ (28, 3)? Không (14 không nằm trong 29,30,31,0,1,2)
- finger[4] = 4: 4 ∈ (28, 3)? Không
- finger[3] = 1: 1 ∈ (28, 3)? **Có** (1 nằm trong 29,...,0,1,2)
- Chuyển đến **node 1**

Bước 3: Tại Node 1

Finger Table của Node 1:

i	start = (1 + 2^(i-1)) mod 32	finger[i]
1	2	succ(2) = 4
2	3	succ(3) = 4
3	5	succ(5) = 9
4	9	succ(9) = 9
5	17	succ(17) = 18

- Node 1 kiểm tra: key = 3 ∈ [1, succ(1)] = [1, 4]
- 3 ∈ [1, 4]
- **Trả về successor = 4**

7. Tóm tắt quá trình tra cứu

```
Node 9 ──[finger[5]=28]──→ Node 28 ──[finger[3]=1]──→ Node 1 ──→ Found: Node 4
```

Bước	Tại Node	Hành động	Chuyển đến
1	9	key=3 ∉ (9,11], dùng finger[5]=28	Node 28
2	28	key=3 ∉ (28,1], dùng finger[3]=1	Node 1
3	1	key=3 ∈ (1,4], trả về succ=4	Node 4

8. Kết luận

- **succ(7) = 9**
- **succ(22) = 28**
- **succ(30) = 1**
- **Key 3 được lưu tại Node 4**
- Số bước lookup: **3 hops** (9 → 28 → 1 → 4)
- Độ phức tạp: O(log N) với N = 9 nodes, $\log_2(9) \approx 3.17$

Câu 4:

Giả sử bạn xây dựng một ứng dụng chia sẻ tệp tin P2P không cấu trúc.
Áp dụng cơ chế flooding, hãy mô tả cách tìm kiếm một tệp tin khi nút phát yêu cầu có TTL = 3.
Phân tích tình huống: Nếu hệ thống có mật độ nút cao, việc chọn TTL = 3 có thể dẫn đến hệ quả gì về độ bao phủ và chi phí truyền thông?
Trả lời:
1. Tổng quan về P2P không cấu trúc và Flooding

P2P không cấu trúc (Unstructured P2P):

- Các node kết nối với nhau một cách ngẫu nhiên
- Không có quy tắc về vị trí lưu trữ dữ liệu
- Không có DHT hay cấu trúc định tuyến cố định
- Ví dụ: Gnutella, Kazaa, Freenet

Flooding (Lan truyền tràn ngập):

- Cơ chế tìm kiếm đơn giản nhất trong P2P không cấu trúc
- Node gửi query đến TẤT CẢ các neighbors
- Mỗi neighbor tiếp tục gửi đến tất cả neighbors của nó
- Sử dụng **TTL (Time-To-Live)** để giới hạn phạm vi lan truyền

2. Mô tả cơ chế Flooding với TTL = 3

Cấu trúc Query Message:

```
{
  "message_id": "unique-uuid-12345",
  "type": "QUERY",
  "filename": "movie.mp4",
  "ttl": 3,
  "origin_node": "NodeA",
  "hops": 0
}
```

Thuật toán Flooding:

```
FUNCTION flood_search(query, ttl):
    IF ttl <= 0:
        RETURN  // Dừng lan truyền
    // Kiểm tra xem đã xử lý query này chưa (tránh loop)
    IF query.message_id IN processed_queries:
        RETURN

    // Đánh dấu đã xử lý
    ADD query.message_id TO processed_queries

    // Kiểm tra local: có file không?
    IF file_exists(query.filename):
        SEND query_hit TO query.origin_node

    // Forward đến tất cả neighbors (trừ node gửi đến)
    FOR EACH neighbor IN neighbors:
        new_query = copy(query)
        new_query.ttl = ttl - 1
```

```
        new_query.hops = query.hops + 1
        SEND new_query TO neighbor
```

3. Ví dụ minh họa với TTL = 3

Mạng P2P mẫu:

```
                [D]
               / |
              /  |
        [B]-----[A]--[E]
         |       |    |
         |       |    |
        [C]     [F]--[G]
         |            |
        [H]          [I]
```

Giả sử: Node A tìm kiếm file "movie.mp4"

Bước 1: TTL = 3 (tại Node A - Origin)

Node A gửi query đến tất cả neighbors: B, D, E, F

```
A ──→ B (TTL=2)
A ──→ D (TTL=2)
A ──→ E (TTL=2)
A ──→ F (TTL=2)
```

Trạng thái: 4 messages được gửi

Bước 2: TTL = 2 (tại các Node B, D, E, F)

Node B (nhận từ A) → gửi đến neighbors trừ A:

```
B ──→ C (TTL=1)
```

Node D (nhận từ A) → gửi đến neighbors trừ A:

```
D ──→ E (TTL=1)  // E cũng là neighbor của D
```

Node E (nhận từ A) → gửi đến neighbors trừ A:

```
E ──→ D (TTL=1)  // Nhưng D đã nhận từ A, sẽ bị drop
E ──→ G (TTL=1)
```

Node F (nhận từ A) → gửi đến neighbors trừ A:

```
F ──→ G (TTL=1)  // G cũng nhận từ E
```

Trạng thái: Thêm ~5 messages (một số bị drop do duplicate)

Bước 3: TTL = 1 (tại các Node C, G)

Node C (nhận từ B):

```
C ──→ H (TTL=0)  // H nhận nhưng không forward tiếp
```

Node G (nhận từ E hoặc F, cái nào đến trước):

```
G ──→ I (TTL=0)  // I nhận nhưng không forward tiếp
G ──→ F (TTL=0)  // Có thể bị drop nếu F đã xử lý
```

Bước 4: TTL = 0 (Dừng)

Các node H, I nhận query với TTL=0:

- Kiểm tra local có file không
- **KHÔNG forward tiếp** (TTL hết)

4. Sơ đồ lan truyền

```
Thời gian →
T=0:     [A] Origin
           ↓ TTL=3
T=1:   [B] [D] [E] [F]
           ↓ TTL=2
T=2:   [C]    [G]
           ↓ TTL=1
T=3:   [H]    [I]
           ↓ TTL=0
          STOP
```

5. Xử lý Query Hit (Tìm thấy file)

Giả sử Node G có file "movie.mp4":

```
1. Node G nhận query từ E
2. G kiểm tra local → TÌM THẤY "movie.mp4"
3. G gửi QUERY_HIT ngược về theo đường đi:
   G → E → A (hoặc G → F → A)
4. Node A nhận được thông tin:
   - Node G có file
   - Địa chỉ IP/Port của G
5. A kết nối trực tiếp với G để download file
```

Query Hit Message:

```
{
  "message_id": "unique-uuid-12345",
  "type": "QUERY_HIT",
  "filename": "movie.mp4",
  "file_size": 1500000000,
  "node_address": "192.168.1.50:6346",
  "hops": 2
}
```

6. Phân tích: Mật độ nút cao + TTL = 3

Giả định:

- Mỗi node có trung bình **k neighbors** (degree)
- TTL = 3
- Mật độ cao: k lớn (ví dụ k = 10)

Ước tính số messages:

Công thức (worst case):

> Messages ≈ $k + k^2 + k^3 = k(k^2 + k + 1)$

TTL	Nodes reached (worst case)	Với k=5	Với k=10
1	k	5	10
2	$k + k^2$	30	110
3	$k + k^2 + k^3$	155	1,110

Phân tích độ bao phủ (Coverage):

Ưu điểm:

Khía cạnh	Mô tả
Bao phủ rộng	Với k=10, TTL=3 có thể đạt ~1000 nodes
Tìm nhanh file phổ biến	File có nhiều bản sao sẽ được tìm thấy nhanh
Đơn giản	Không cần cấu trúc phức tạp

Nhược điểm:

Khía cạnh	Mô tả
Giới hạn phạm vi	Chỉ tìm trong 3 hops, có thể bỏ sót file ở xa
Không đảm bảo	Không chắc chắn tìm thấy dù file tồn tại

Phân tích chi phí truyền thông (Communication Cost):

Vấn đề nghiêm trọng với mật độ cao:

Vấn đề	Mô tả	Mức độ
Message Explosion	Số message tăng theo cấp số nhân O(k^TTL)	Nghiêm trọng
Bandwidth Consumption	Mỗi node nhận/gửi hàng trăm messages	Nghiêm trọng
Duplicate Messages	Cùng query đến 1 node qua nhiều đường	Trung bình
Processing Overhead	CPU xử lý nhiều queries	Trung bình

Ví dụ cụ thể:

```
Mạng: 10,000 nodes, k=10, TTL=3

Nếu 100 nodes cùng search trong 1 phút:
- Messages/search ≈ 1,000
- Total messages = 100 × 1,000 = 100,000 messages/phút
- Bandwidth: ~10MB/phút (giả sử 100 bytes/message)
```

7. Bảng so sánh TTL values

TTL	Coverage	Messages (k=10)	Trade-off
1	Rất thấp	10	Tiết kiệm nhưng kém hiệu quả
2	Thấp	110	Cân bằng cho mạng nhỏ
3	Trung bình	1,110	Phổ biến, nhưng tốn kém
4	Cao	11,110	Quá tốn kém
5	Rất cao	111,110	Không khả thi

8. Giải pháp cải thiện

a) Random Walk thay vì Flooding:

```
Thay vì gửi đến TẤT CẢ neighbors:
→ Chọn NGẪU NHIÊN 1-2 neighbors để forward
→ Giảm messages nhưng vẫn có cơ hội tìm thấy
```

b) Expanding Ring Search:

```
Bắt đầu với TTL=1
Nếu không tìm thấy → tăng TTL=2
Tiếp tục cho đến khi tìm thấy hoặc TTL max
```

c) Supernode Architecture (Kazaa-style):

```
- Một số node mạnh làm "supernode"
- Query chỉ flood giữa các supernodes
- Giảm đáng kể số messages
```

d) Bloom Filters:

```
- Mỗi node lưu Bloom filter của neighbors
- Chỉ forward query đến neighbor có khả năng có file
```

9. Kết luận

Khía cạnh	Đánh giá với TTL=3 và mật độ cao
Độ bao phủ	Tốt - đạt được nhiều nodes trong 3 hops
Chi phí truyền thông	Rất cao - $O(k^3)$ messages
Scalability	Kém - không phù hợp mạng lớn
Khuyến nghị	Cần kết hợp với các kỹ thuật tối ưu (random walk, supernodes)

Tóm lại: TTL=3 trong mạng mật độ cao tạo ra sự đánh đổi giữa **coverage** và **cost**. Độ bao phủ tốt nhưng chi phí truyền thông rất cao (message explosion), có thể gây quá tải mạng. Cần áp dụng các kỹ thuật tối ưu để cân bằng.

Câu 5:

Trong một mạng P2P không cấu trúc, một nút cần tìm kiếm dữ liệu hiếm (rare data).

Hãy áp dụng phương pháp random walk để mô tả cách tìm kiếm dữ liệu này.

Đề xuất và giải thích cách cải tiến (ví dụ: khởi động nhiều random walks đồng thời) để giảm thời gian tìm thấy dữ liệu, và phân tích sự đánh đổi giữa thời gian tìm kiếm và lưu lượng mạng.

Trả lời:

1. Tổng quan về Random Walk trong P2P

Random Walk là phương pháp tìm kiếm trong mạng P2P không cấu trúc, trong đó query message di chuyển ngẫu nhiên từ node này sang node khác cho đến khi tìm thấy dữ liệu hoặc đạt giới hạn.

So sánh với Flooding:

Tiêu chí	Flooding	Random Walk
Số messages	O(k^TTL) - rất cao	O(TTL) - tuyến tính
Coverage	Rộng, đồng thời	Hẹp, tuần tự
Phù hợp	Dữ liệu phổ biến	Dữ liệu hiếm (với cải tiến)

2. Cơ chế Random Walk cơ bản

Thuật toán:

```
def random_walk_search(query, ttl, max_ttl):
    """
    query: thông tin tìm kiếm (filename, keywords)
    ttl: số bước còn lại
    max_ttl: giới hạn tối đa
    """

    # Điều kiện dừng
    if ttl <= 0:
        return NOT_FOUND

    # Kiểm tra local
    if has_data(query.filename):
        return FOUND(current_node)

    # Chọn NGẪU NHIÊN 1 neighbor
    next_node = random.choice(neighbors)

    # Forward query
    query.ttl = ttl - 1
    query.path.append(current_node)

    return forward(query, next_node)
```

Ví dụ minh họa:

```
Mạng P2P:
    [A]---[B]---[C]---[D]
     |     |     |     |
    [E]---[F]---[G]---[H]
     |     |     |     |
    [I]---[J]---[K]---[L]  ← L có dữ liệu hiếm

Node A tìm kiếm dữ liệu, TTL = 10
```

Random Walk có thể đi theo đường:

```
A → B → F → G → C → D → H → L (FOUND)
     ↓
   7 hops, 7 messages
```

So với Flooding TTL=3:

```
Flooding: ~100+ messages nhưng có thể không đến được L
Random Walk: 7 messages, đến được L
```

3. Vấn đề với dữ liệu hiếm (Rare Data)

Thách thức:

• Dữ liệu hiếm chỉ có ở 1-2 nodes trong mạng lớn
• Random walk đơn lẻ có thể mất rất lâu để tìm thấy
• Xác suất tìm thấy trong k bước: P = 1 - (1 - r)^k

- Với r = tỷ lệ nodes có dữ liệu (rất nhỏ với rare data)

Ví dụ:

```
Mạng 10,000 nodes, chỉ 10 nodes có dữ liệu
r = 10/10,000 = 0.001 (0.1%)

Xác suất tìm thấy trong 100 bước:
P = 1 - (1-0.001)^100 ≈ 9.5%

Xác suất tìm thấy trong 1000 bước:
P = 1 - (1-0.001)^1000 ≈ 63%
```

→ Cần cải tiến để tìm nhanh hơn

4. Các phương pháp cải tiến

4.1. Multiple Random Walks (k-Random Walks)

Ý tưởng: Khởi động k random walks đồng thời từ node nguồn.

```
def k_random_walks(query, k, ttl):
    """
    k: số random walks đồng thời
    """
    results = []

    # Chọn k neighbors khác nhau (hoặc có thể trùng)
    selected_neighbors = random.sample(neighbors, min(k, len(neighbors)))

    # Khởi động k walks song song
    for i, neighbor in enumerate(selected_neighbors):
        walk_query = copy(query)
        walk_query.walk_id = i
        walk_query.ttl = ttl

        # Gửi đồng thời
        async_send(walk_query, neighbor)

    # Chờ kết quả đầu tiên (hoặc timeout)
    return wait_for_first_result(timeout=30s)
```

Sơ đồ:

```
          ┌──→ Walk 1: A→B→F→J→...
          │
Node A ──┼──→ Walk 2: A→E→I→J→K→L (FOUND)
          │
```

```
          └──→ Walk 3: A→B→C→G→...
```

Phân tích:

- **Thời gian:** Giảm đáng kể (chia cho k nếu độc lập)
- **Messages:** Tăng k lần so với single walk
- **Xác suất tìm thấy:** P_k = 1 - (1-r)^(k×steps)

4.2. Biased Random Walk

Ý tưởng: Không chọn neighbor hoàn toàn ngẫu nhiên, mà ưu tiên nodes có khả năng cao hơn.

```
def biased_random_walk(query, ttl):
    if ttl <= 0 or has_data(query):
        return result

    # Tính điểm cho mỗi neighbor
    scores = {}
    for neighbor in neighbors:
        scores[neighbor] = calculate_score(neighbor, query)

    # Chọn neighbor với xác suất tỷ lệ với score
    next_node = weighted_random_choice(neighbors, scores)

    return forward(query, next_node)

def calculate_score(neighbor, query):
    """Các tiêu chí đánh giá"""
    score = 1.0

    # Ưu tiên node có nhiều connections (high degree)
    score *= neighbor.degree / avg_degree

    # Ưu tiên node chưa được visit gần đây
    if neighbor not in recent_visits:
        score *= 2.0

    # Ưu tiên dựa trên metadata (nếu có)
    if query.category in neighbor.content_categories:
        score *= 3.0

    return score
```

4.3. Random Walk với Checkpointing

Ý tưởng: Lưu lại các nodes đã visit, tránh lặp lại.

```
def random_walk_with_history(query, ttl, visited_set):
    if ttl <= 0:
        return NOT_FOUND
```

```
    if has_data(query):
        return FOUND

    # Thêm current node vào visited
    visited_set.add(current_node)

    # Chọn neighbor CHƯA VISIT
    unvisited = [n for n in neighbors if n not in visited_set]

    if not unvisited:
        # Backtrack hoặc random restart
        return random_restart(query, ttl)

    next_node = random.choice(unvisited)
    return forward(query, next_node, visited_set)
```

4.4. Adaptive Random Walk

Ý tưởng: Điều chỉnh số walks dựa trên độ hiếm của dữ liệu.

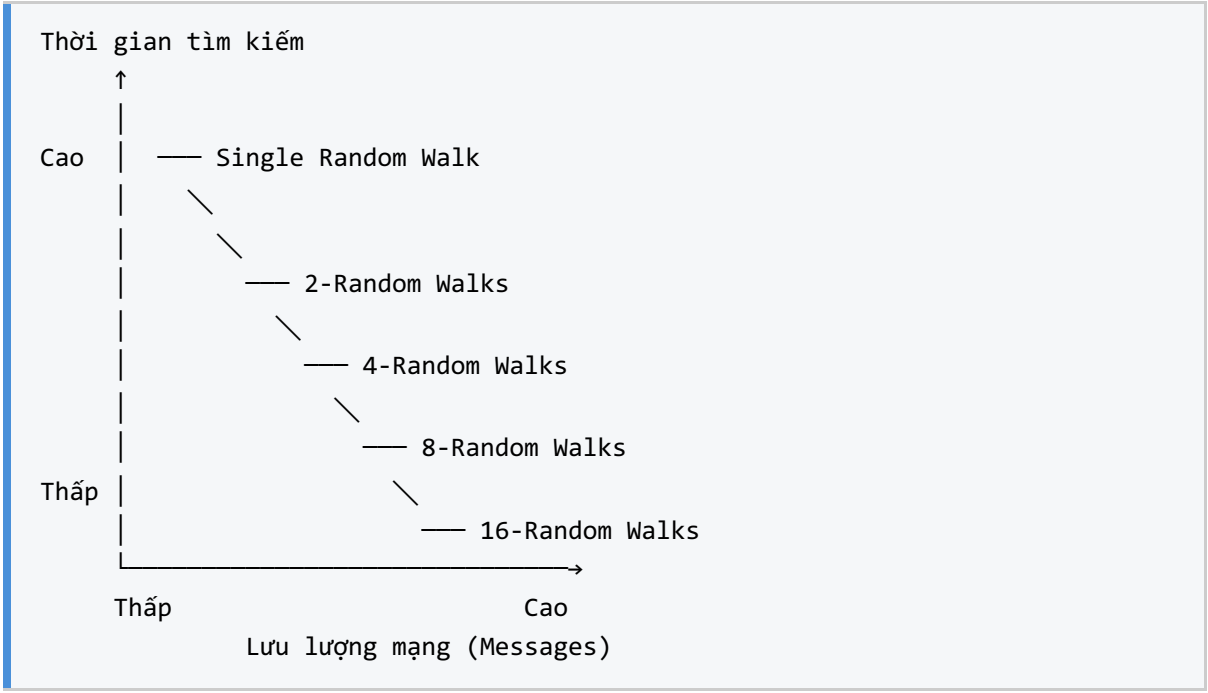
```
def adaptive_random_walks(query):
    # Bắt đầu với ít walks
    k = 2
    ttl = 50

    while not found and k <= MAX_WALKS:
        # Thử với k walks
        result = k_random_walks(query, k, ttl)

        if result == FOUND:
            return result

        # Tăng số walks
        k = k * 2

        # Tùy chọn: tăng TTL
        ttl = min(ttl + 20, MAX_TTL)

    return NOT_FOUND
```

5. Phân tích đánh đổi: Thời gian vs Lưu lượng mạng

Bảng so sánh các phương pháp:

Phương pháp	Messages	Thời gian	Trade-off
Single Walk	TTL	Cao	Tiết kiệm bandwidth, chậm
k-Walks (k=4)	4×TTL	Giảm ~4x	Cân bằng tốt
k-Walks (k=16)	16×TTL	Giảm ~16x	Tốn bandwidth
Flooding TTL=3	$O(k^3)$	Thấp nhất	Rất tốn, không scale

Đồ thị Trade-off:



Phân tích chi tiết:

Số walks (k)	Messages (TTL=100)	Thời gian trung bình	Xác suất (r=0.1%)
1	100	1000 steps	9.5%
2	200	500 steps	18.1%
4	400	250 steps	33.0%
8	800	125 steps	55.1%
16	1,600	62 steps	79.8%
32	3,200	31 steps	95.9%

6. Khuyến nghị cho Rare Data

```
┌─────────────────────────────────────────────────┐
│  KHUYẾN NGHỊ: Kết hợp nhiều kỹ thuật            │
├─────────────────────────────────────────────────┤
│  1. Sử dụng k-Random Walks với k = 4-8          │
│  2. Áp dụng Biased Walk (ưu tiên high-degree nodes)  │
│  3. Tracking visited nodes để tránh lặp         │
│  4. Adaptive: tăng k nếu chưa tìm thấy          │
│  5. TTL hợp lý: 50-100 cho mạng lớn             │
└─────────────────────────────────────────────────┘
```

7. Kết luận

Tiêu chí	Đánh giá
Random Walk cơ bản	Tiết kiệm bandwidth nhưng chậm với rare data
k-Random Walks	Giải pháp hiệu quả, cân bằng tốt
Trade-off	Tăng k → giảm thời gian, tăng messages (tuyến tính)
So với Flooding	Tốt hơn nhiều về scalability (O(k×TTL) vs O(degree^TTL))

Công thức tối ưu:

```
k_optimal ≈ √(N/n)
Trong đó:
- N = tổng số nodes
- n = số nodes có dữ liệu
```

Câu 6:

• Hãy định nghĩa khái niệm luồng (thread) và nêu sự khác biệt cơ bản giữa luồng và tiến trình (process) trong hệ điều hành.

• Giải thích tại sao việc chia tiến trình thành nhiều luồng có thể cải thiện hiệu năng trên máy tính đa bộ vi xử lý.

Trả lời:

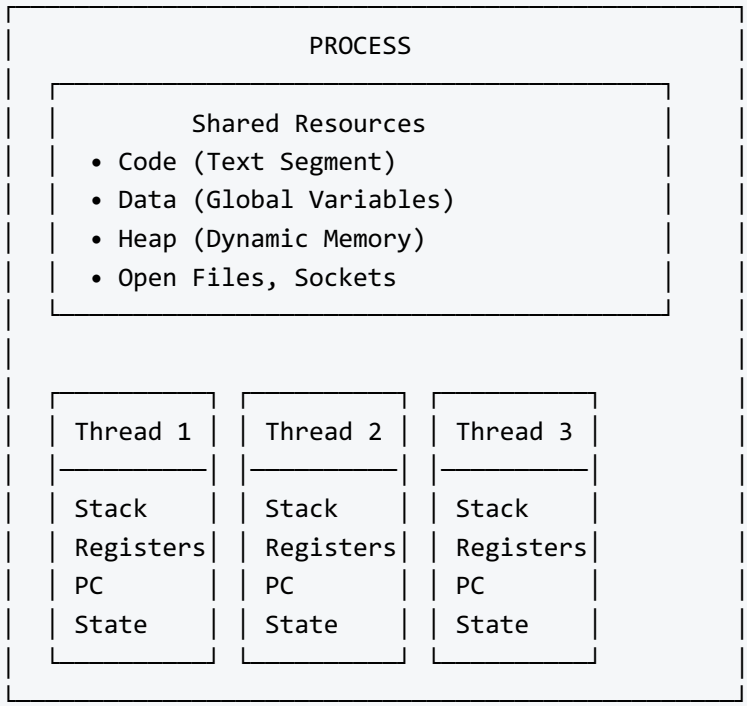
1. Định nghĩa Thread (Luồng)

Thread (Luồng) là đơn vị thực thi nhỏ nhất trong một tiến trình, đại diện cho một dòng điều khiển (flow of control) độc lập có thể được lập lịch bởi hệ điều hành.

Đặc điểm của Thread:

- Là một "lightweight process" (tiến trình nhẹ)
- Chia sẻ không gian địa chỉ và tài nguyên với các thread khác trong cùng process
- Có stack, registers, program counter riêng
- Có thể chạy song song trên nhiều CPU cores

Cấu trúc Thread:



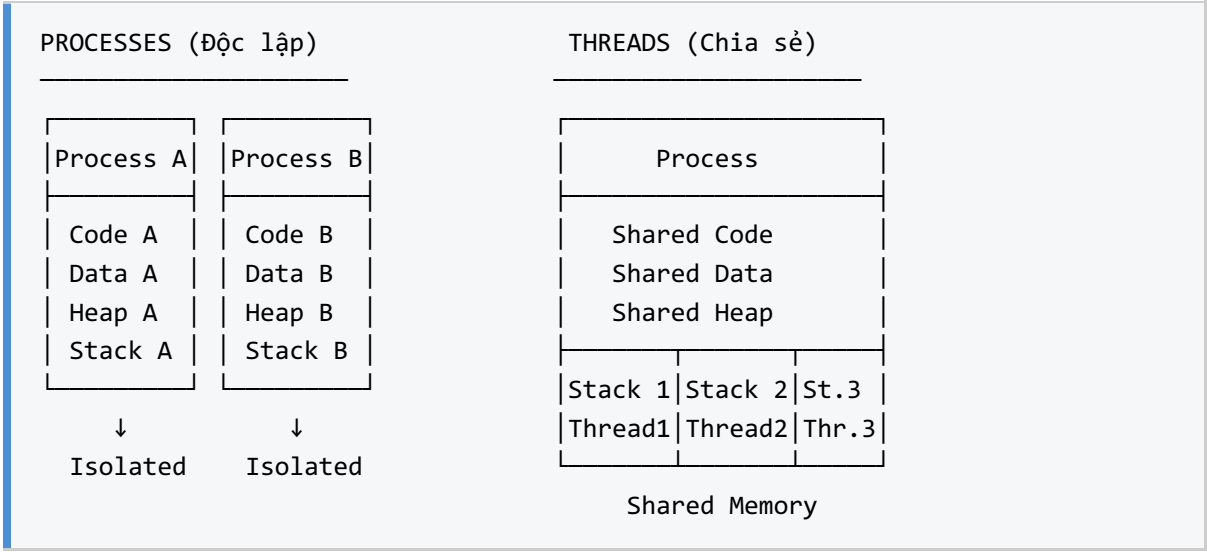
2. So sánh Thread và Process

Bảng so sánh chi tiết:

Tiêu chí	Process (Tiến trình)	Thread (Luồng)
Định nghĩa	Chương trình đang thực thi, có không gian địa chỉ riêng	Đơn vị thực thi trong process, chia sẻ không gian địa chỉ

Không gian địa chỉ	Riêng biệt, độc lập	Chia sẻ với các threads khác trong process
Bộ nhớ	Code, Data, Heap, Stack riêng	Chỉ có Stack riêng; chia sẻ Code, Data, Heap
Tạo mới	Tốn kém (fork, clone)	Nhẹ hơn nhiều
Context Switch	Chậm (phải đổi page table, TLB flush)	Nhanh (chỉ đổi registers, stack)
Giao tiếp	IPC (pipes, sockets, shared memory) - phức tạp	Trực tiếp qua shared memory - đơn giản
Cô lập lỗi	Cao (crash 1 process không ảnh hưởng process khác)	Thấp (crash 1 thread có thể crash cả process)
Overhead	Cao	Thấp
Đồng bộ hóa	Ít cần (độc lập)	Cần nhiều (mutex, semaphore)

Sơ đồ so sánh bộ nhớ:

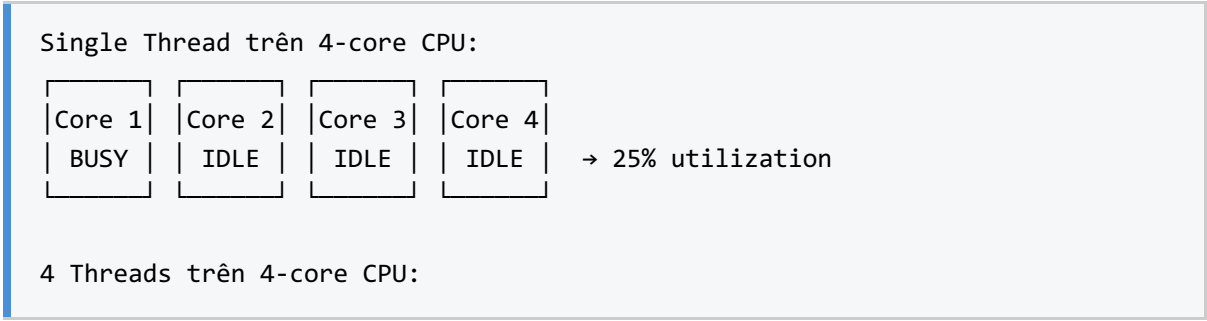


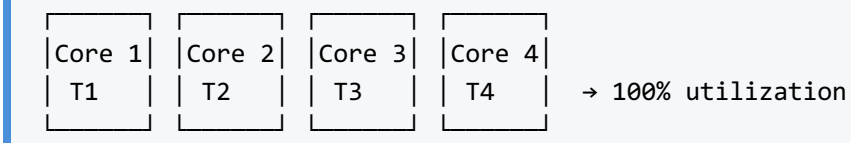
Chi phí Context Switch:

Thao tác	Process Switch	Thread Switch
Save/Restore Registers		
Save/Restore Stack Pointer		
Switch Page Tables		
Flush TLB		
Flush CPU Cache	Có thể	Không
Thời gian (ước tính)	~1000-10000 cycles	~100-1000 cycles

3. Tại sao Multi-threading cải thiện hiệu năng trên Multi-processor

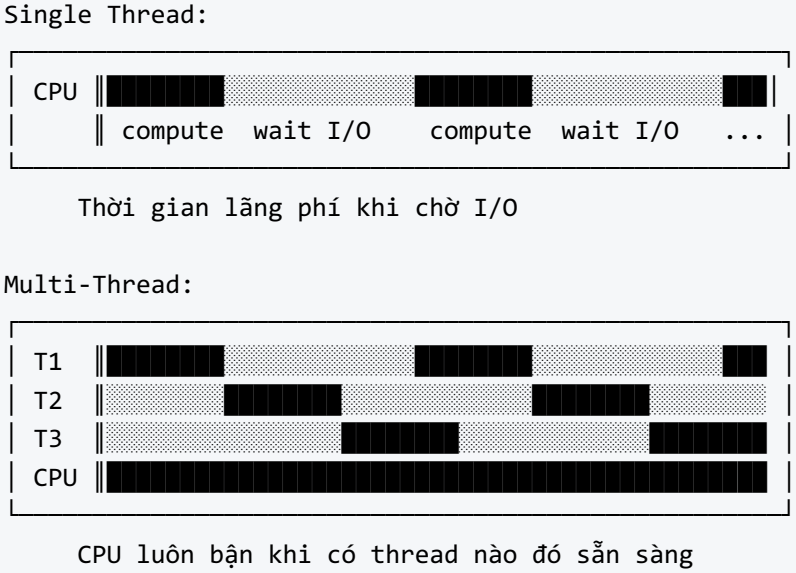
3.1. Tận dụng song song thực sự (True Parallelism)





Speedup lý tưởng: S = N (với N cores)
Speedup thực tế: S < N (do overhead, Amdahl's Law)

3.2. Che giấu độ trễ I/O (Latency Hiding)



3.3. Ví dụ cụ thể: Web Server

Scenario: Web server xử lý 1000 requests/giây

Single-threaded:

```
while True:
    request = accept_connection()      # 1ms
    data = read_from_disk(request)     # 10ms (I/O wait)
    response = process(data)           # 2ms
    send_response(response)            # 1ms
    # Total: 14ms/request → Max 71 requests/sec
```

Multi-threaded (100 threads):

```
def worker_thread():
    while True:
        request = accept_connection()
        data = read_from_disk(request)  # Thread khác chạy trong khi wait
        response = process(data)
        send_response(response)

# 100 threads xử lý song song
# Throughput: ~1000+ requests/sec
```

3.4. Phân tích theo Amdahl's Law

```
Speedup = 1 / ((1-P) + P/N)

Trong đó:
- P = phần có thể song song hóa
- N = số processors/threads
- (1-P) = phần tuần tự
```

Ví dụ:

P (Parallel)	N=2	N=4	N=8	N=∞
50%	1.33x	1.60x	1.78x	2.0x
75%	1.60x	2.29x	2.91x	4.0x
90%	1.82x	3.08x	4.71x	10.0x
95%	1.90x	3.48x	5.93x	20.0x

4. Các lợi ích khác của Multi-threading

Lợi ích	Mô tả
Responsiveness	UI thread riêng, không bị block bởi computation
Resource Sharing	Threads chia sẻ memory, giảm overhead
Economy	Tạo thread rẻ hơn tạo process
Scalability	Dễ scale với số cores tăng
Modularity	Chia task phức tạp thành các threads độc lập

5. Thách thức của Multi-threading

Thách thức	Giải pháp
Race Conditions	Mutex, Locks, Atomic operations
Deadlocks	Lock ordering, Timeout, Deadlock detection
Starvation	Fair scheduling, Priority inheritance
Debugging khó	Thread-safe logging, Debugger hỗ trợ threads
Memory Consistency	Memory barriers, Volatile, Atomic

6. Kết luận

```
┌─────────────────────────────────────────────────────────┐
│  THREAD vs PROCESS                                      │
├─────────────────────────────────────────────────────────┤
│  • Thread = lightweight, chia sẻ memory, context switch │
│           nhanh, giao tiếp dễ                           │
│  • Process = heavyweight, cô lập, an toàn hơn           │
├─────────────────────────────────────────────────────────┤
│  MULTI-THREADING TRÊN MULTI-PROCESSOR                   │
├─────────────────────────────────────────────────────────┤
│  1. Tận dụng tất cả CPU cores (true parallelism)        │
│  2. Che giấu I/O latency (1 thread wait, thread khác chạy) │
│  3. Tăng throughput đáng kể cho workloads phù hợp       │
│  4. Giới hạn bởi Amdahl's Law (phần tuần tự)            │
└─────────────────────────────────────────────────────────┘
```

Câu 7:

So sánh hai cách cài đặt luồng: (a) hoàn toàn ở mức người dùng (user level threads) và (b) kết hợp luồng ở mức người dùng với lightweight process (LWP) ở mức kernel. Phân tích ưu, nhược điểm của từng cách về chi phí chuyển ngữ cảnh, khả năng phong tỏa và độ phức tạp triển khai.

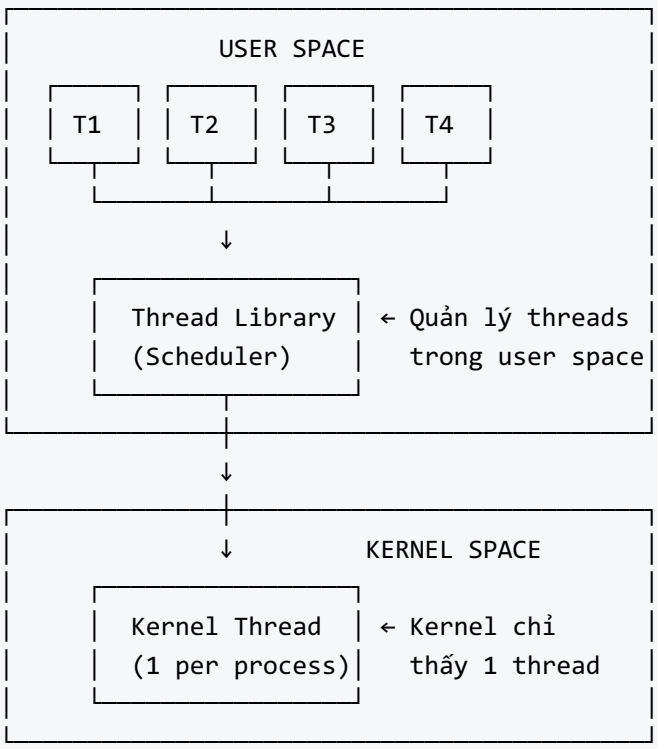
Trả lời:

1. Tổng quan các mô hình Thread

```
┌─────────────────────────────────────────────────────────────────────┐
│                    CÁC MÔ HÌNH THREAD                               │
├──────────────────┬──────────────────────┬───────────────────────────┤
│  User-Level      │   Kernel-Level       │   Hybrid (M:N)            │
│  Threads (ULT)   │   Threads (KLT)      │   User + LWP              │
│  Many-to-One     │   One-to-One         │   Many-to-Many            │
└──────────────────┴──────────────────────┴───────────────────────────┘
```

2. Mô hình (a): User-Level Threads (ULT) - Many-to-One

Kiến trúc:



Đặc điểm:

- Thread được quản lý hoàn toàn bởi **user-space library** (pthreads, green threads)
- Kernel **không biết** về sự tồn tại của các threads
- Tất cả user threads map vào **1 kernel thread**
- Ví dụ: GNU Portable Threads, early Java Green Threads

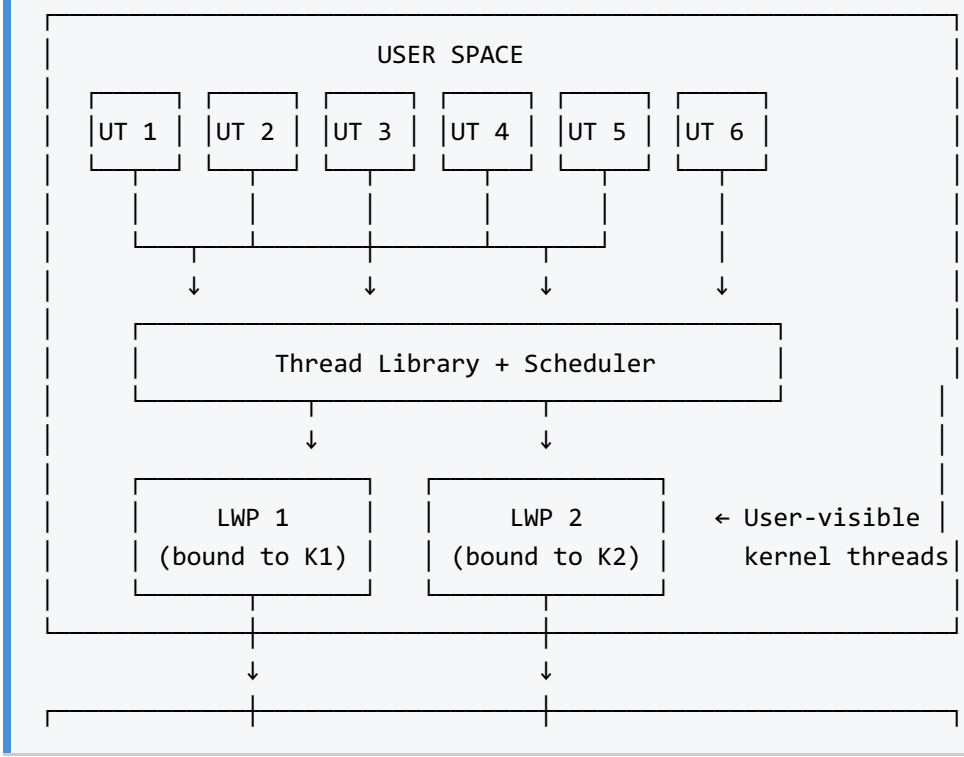
Cơ chế hoạt động:

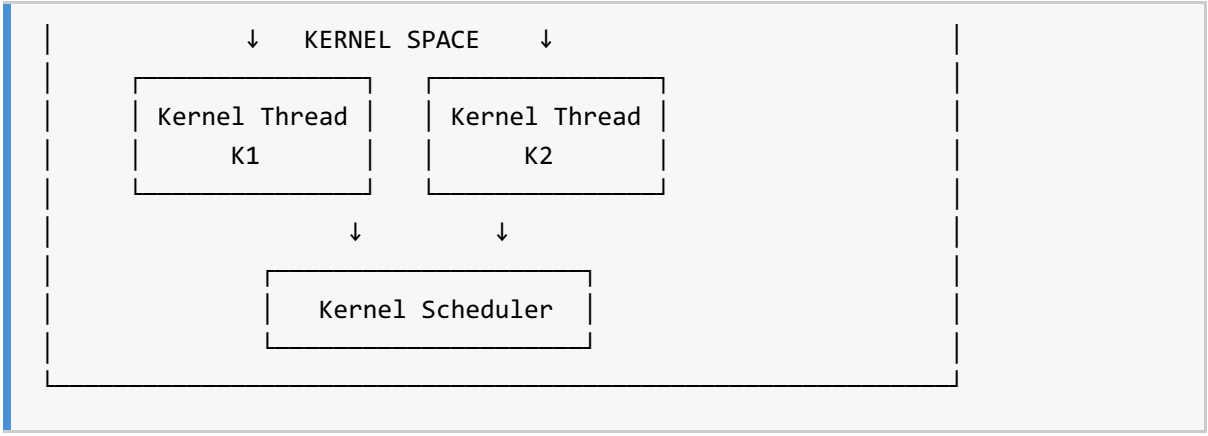
```
# Thread Library (User Space)
```

```
class UserThreadScheduler:
    def __init__(self):
        self.threads = []
        self.current = None

    def create_thread(self, func):
        thread = UserThread(func)
        self.threads.append(thread)

    def yield_thread(self):
        # Lưu context của current thread
        save_context(self.current)

        # Chọn thread tiếp theo (round-robin)
        self.current = self.pick_next()

        # Restore context và chạy
        restore_context(self.current)

    def schedule(self):
        # Cooperative scheduling
        # Thread phải tự gọi yield()
        while self.threads:
            self.current.run_until_yield()
```

3. Mô hình (b): Hybrid Model - User Threads + LWP (Many-to-Many)

Kiến trúc:





Đặc điểm:

- **M user threads** được map vào **N kernel threads (LWPs)** với M ≥ N
- LWP (Lightweight Process) = kernel thread có thể được lập lịch bởi kernel
- Thread library quản lý việc gán user threads vào LWPs
- Ví dụ: Solaris Threads, IRIX, HP-UX

Cơ chế Scheduler Activation:

```
┌──────────────────────────────────────────────────────┐
│  SCHEDULER ACTIVATIONS                               │
├──────────────────────────────────────────────────────┤
│  1. Kernel thông báo cho user-space khi:             │
│     - LWP bị block (blocked activation)              │
│     - LWP được unblock (unblocked activation)        │
│     - Có thêm processor available                    │
│                                                      │
│  2. User-space scheduler quyết định:                 │
│     - Gán user thread nào cho LWP nào                │
│     - Khi nào preempt user thread                    │
└──────────────────────────────────────────────────────┘
```

4. So sánh chi tiết

4.1. Chi phí chuyển ngữ cảnh (Context Switch Cost)

Tiêu chí	User-Level Threads	Hybrid (ULT + LWP)
Intra-process switch	Rất nhanh (~100 cycles)	Nhanh (~100-500 cycles)
Cần kernel call?	Không	Không (giữa ULTs trên cùng LWP)
Thao tác	Save/restore: PC, SP, registers	Tương tự + có thể đổi LWP
TLB/Cache flush	Không	Không
Cross-LWP switch	N/A	Cần kernel intervention

Chi tiết:

```
USER-LEVEL THREAD SWITCH (ULT):
┌──────────────────────────────────────┐
│ 1. Save registers của T1 (user space)  │ ~20 cycles
│ 2. Update thread table                 │ ~10 cycles
│ 3. Select next thread T2               │ ~20 cycles
```

```
│ 4. Restore registers của T2           │ ~20 cycles
│ ─────────────────────────────────────  │
│ TOTAL: ~70-100 cycles                  │
│ NO KERNEL INVOLVEMENT                  │
└────────────────────────────────────────┘

HYBRID MODEL SWITCH:
┌────────────────────────────────────────────────┐
│ Case 1: Switch giữa ULTs trên cùng LWP         │
│ → Giống ULT, ~100 cycles                       │
│                                                │
│ Case 2: Switch khi ULT block (syscall)         │
│ 1. ULT1 calls blocking syscall                 │
│ 2. Kernel blocks LWP1                          │
│ 3. Scheduler activation notifies user library  │
│ 4. Library schedules ULT2 on LWP2              │
│ → ~1000-5000 cycles (kernel involved)          │
└────────────────────────────────────────────────┘
```

4.2. Khả năng phong tỏa (Blocking Behavior)

Tiêu chí	User-Level Threads	Hybrid (ULT + LWP)
Khi 1 thread gọi blocking I/O	Toàn bộ process bị block	Chỉ LWP đó bị block
Các threads khác	Không chạy được	Chạy trên LWPs khác
Page fault	Block toàn bộ	Chỉ block LWP liên quan
Giải pháp cho ULT	Non-blocking I/O, async I/O	Không cần

Minh họa vấn đề blocking:

```
USER-LEVEL THREADS - Blocking Problem:
─────────────────────────────────────
Time →
T1: ████████░░░░░░░░░░░░░░░░░░░░████████
T2: ────────░░░░░░░░░░░░░░░░░░░░────────  BLOCKED
T3: ────────░░░░░░░░░░░░░░░░░░░░────────  BLOCKED
         ↑                    ↑
    T1 calls               I/O done
    read()

Kernel chỉ thấy 1 thread → block cả process
HYBRID MODEL - No Blocking Problem:
───────────────────────────────────
T1 on LWP1: ████████░░░░░░░░░░░░████████
T2 on LWP2: ────────████████████────────  RUNNING
T3 on LWP2: ────────────────────████████  RUNNING
                ↑           ↑
           LWP1 blocked   LWP2 continues
```

4.3. Độ phức tạp triển khai (Implementation Complexity)

Tiêu chí	User-Level Threads	Hybrid (ULT + LWP)
Độ phức tạp	Thấp	Cao
Cần sửa kernel?	Không	Có (scheduler activations)
Thread library	Đơn giản	Phức tạp
Debugging	Dễ hơn	Khó hơn
Portability	Cao (pure user space)	Thấp (kernel-dependent)

Chi tiết implementation:

```
USER-LEVEL THREADS:

┌─────────────────────────────────────────┐
│ Components needed:                      │
│ • Thread control blocks (TCB) in user space  │
│ • Simple round-robin or priority scheduler   │
│ • setjmp/longjmp hoặc assembly cho switching │
│ • Mutex/Condition variables (user space)     │
│                                         │
│ LOC estimate: ~2000-5000 lines          │
└─────────────────────────────────────────┘


HYBRID MODEL:

┌─────────────────────────────────────────┐
│ Components needed:                      │
│ • Everything from ULT                   │
│ • Kernel support for LWP creation       │
│ • Scheduler activations in kernel       │
│ • Upcall mechanism (kernel → user)      │
│ • LWP pool management                   │
│ • Complex synchronization               │
│                                         │
│ LOC estimate: ~10000-50000 lines        │
│ (kernel + user space)                   │
└─────────────────────────────────────────┘
```

5. Bảng tổng hợp so sánh

Tiêu chí	User-Level Threads	Hybrid (ULT + LWP)
Context Switch Cost	Rất nhanh	Nhanh
Blocking Behavior	Tệ (block all)	Tốt
True Parallelism	Không	Có
Implementation	Đơn giản	Phức tạp
Portability	Cao	Thấp
Kernel Dependency	Không	Có
Scalability	Thấp (1 core)	Cao (multi-core)
Use Cases	Cooperative tasks, coroutines	General purpose

6. Ví dụ thực tế

Mô hình	Implementations

User-Level	GNU Pth, Early Java Green Threads, Stackless Python, Goroutines (Go - hybrid)
Hybrid M:N	Solaris Threads, FreeBSD KSE, Windows Fibers + Threads
1:1 (so sánh)	Linux NPTL, Windows Threads, Modern Java

7. Kết luận và Khuyến nghị

```
┌─────────────────────────────────────────────────────────────┐
│                        KHUYẾN NGHỊ                          │
├─────────────────────────────────────────────────────────────┤
│  Chọn USER-LEVEL THREADS khi:                               │
│  • Ứng dụng CPU-bound, ít I/O blocking                      │
│  • Cần context switch cực nhanh                             │
│  • Portability quan trọng                                   │
│  • Cooperative multitasking đủ dùng                         │
│                                                             │
│  Chọn HYBRID (ULT + LWP) khi:                               │
│  • Ứng dụng I/O-bound với nhiều blocking calls              │
│  • Cần true parallelism trên multi-core                     │
│  • Có thể chấp nhận kernel dependency                       │
│  • Performance và scalability quan trọng                    │
├─────────────────────────────────────────────────────────────┤
│  Xu hướng hiện đại:                                         │
│  • Hầu hết OS dùng 1:1 (NPTL) vì đơn giản và hiệu quả       │
│  • M:N phức tạp nhưng vẫn dùng ở Go (goroutines), Erlang    │
│  • User-level coroutines/fibers phổ biến trong async I/O    │
└─────────────────────────────────────────────────────────────┘
```

Câu 8:

Trong một hệ thống phân tán phục vụ đọc nhiều – ghi ít (read heavy, write light) như dịch vụ tập tin, bạn sẽ chọn mô hình đa luồng, đơn luồng, hay máy trạng thái hữu hạn (event driven FSM) cho thành phần xử lý trên máy chủ? Hãy đánh giá dựa trên các tiêu chí: hiệu năng, khả năng mở rộng, độ phức tạp phát triển và khả năng chịu lỗi.

Trả lời:

1. Đặc điểm workload "Read Heavy, Write Light"

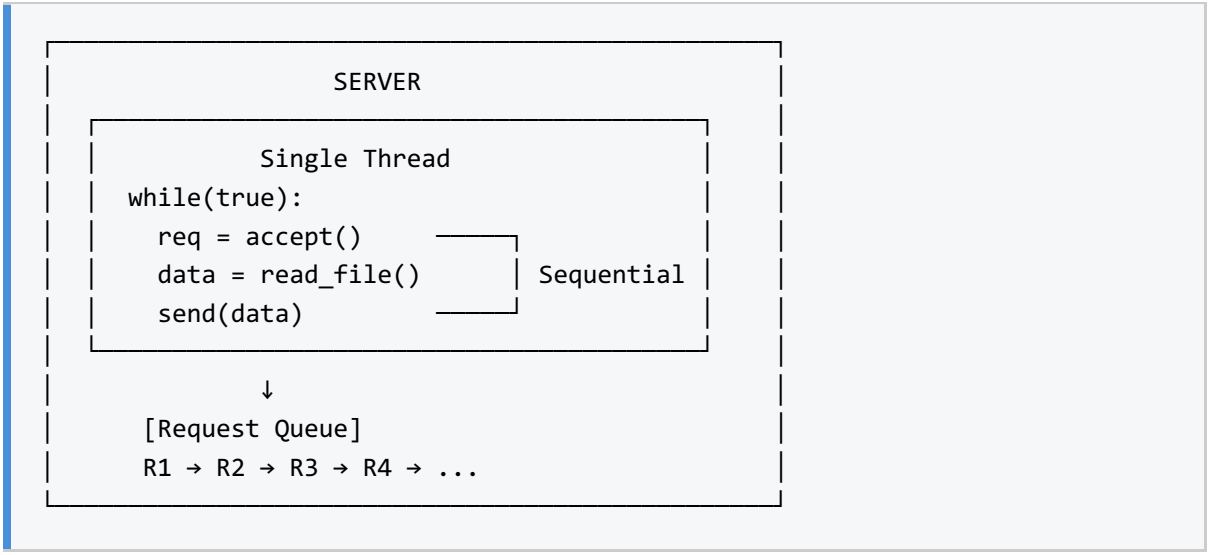
Dịch vụ tập tin (File Service) điển hình:

- **90-99% operations là READ** (đọc file, list directory)
- **1-10% operations là WRITE** (upload, update, delete)
- I/O-bound: phần lớn thời gian chờ disk/network
- Concurrent clients: hàng nghìn đến hàng triệu

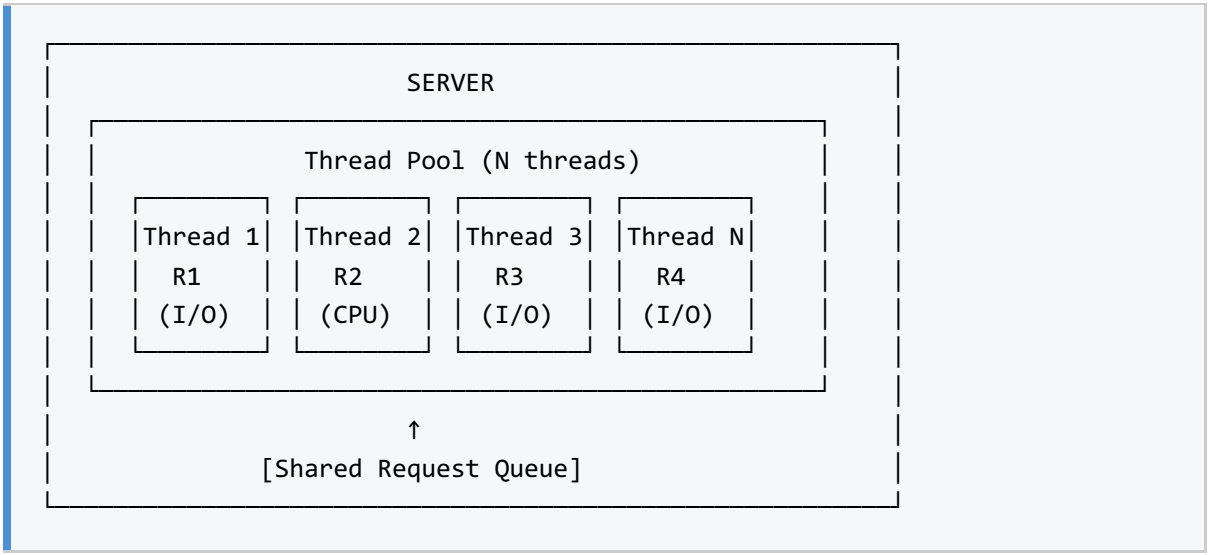
```
Workload Profile:

┌──────────────────────────────────────────────┐
│  READ  ████████████████████████████ 95%      │
│  WRITE ██ 5%                                 │
└──────────────────────────────────────────────┘


Time breakdown per request:

┌──────────────────────────────────────────────┐
│  CPU Processing  ██ 5%                       │
```

2. Ba mô hình xử lý

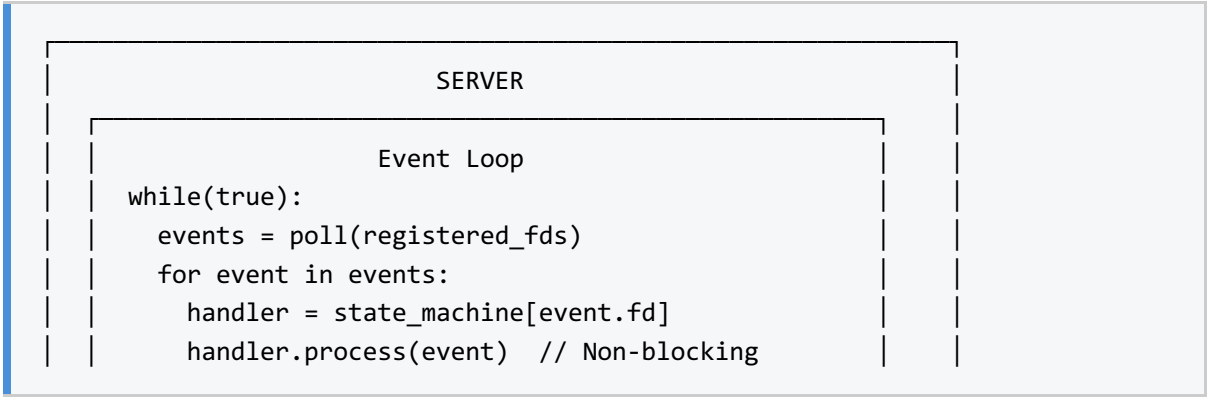
2.1. Mô hình Đơn luồng (Single-Threaded)



2.2. Mô hình Đa luồng (Multi-Threaded)



2.3. Mô hình Event-Driven FSM (Finite State Machine)



```
│  └──────────────────────────────────────────────────────┘    │
│                                                             │
│  State Machine per connection:                              │
│  ┌──────────┐    ┌──────────┐    ┌──────────┐               │
│  │ READING  │ → │PROCESSING│ → │ WRITING  │ → Done        │
│  │ request  │    │  data    │    │ response │               │
│  └──────────┘    └──────────┘    └──────────┘               │
└─────────────────────────────────────────────────────────────┘
```

3. Đánh giá theo các tiêu chí

3.1. Hiệu năng (Performance)

Mô hình	Throughput	Latency	I/O Efficiency
Đơn luồng	Thấp	Cao (blocking)	Kém
Đa luồng	Cao	Thấp	Tốt
Event-Driven	Rất cao	Thấp	Rất tốt

Phân tích chi tiết:

```
ĐƠN LUỒNG - Performance Analysis:
─────────────────────────────────

Request timeline:
R1: [accept][read file.....][send]
R2:                                [accept][read][send]
R3:                                                    [...]

Problem: CPU idle 95% thời gian chờ I/O
Throughput: ~50-100 requests/sec (limited by I/O latency)
ĐA LUỒNG - Performance Analysis:
─────────────────────────────────

Thread 1: [R1: read file.....][R5: read.....]
Thread 2: [R2: read file.....][R6: read.....]
Thread 3: [R3: read file.....][R7: read.....]
Thread 4: [R4: read file.....][R8: read.....]

Improvement: Multiple I/O operations in parallel
Throughput: ~500-5000 requests/sec
Overhead: Context switching, memory per thread


EVENT-DRIVEN - Performance Analysis:
─────────────────────────────────────

Single thread multiplexing:
Event loop: [R1:accept][R2:accept][R3:accept][R1:data_ready][...]
            └─ Non-blocking, immediate return ─┘

Benefits:
```

```
• No context switch overhead
• No thread stack memory (~1MB/thread saved)
• Handles 10,000+ concurrent connections easily

Throughput: ~10,000-100,000 requests/sec
```

Benchmark ước tính (File Service, 10KB files):

Mô hình	Concurrent Connections	Requests/sec	Memory
Đơn luồng	1	100	10 MB
Đa luồng (100 threads)	100	2,000	100 MB
Đa luồng (1000 threads)	1,000	10,000	1 GB
Event-Driven	10,000+	50,000	50 MB

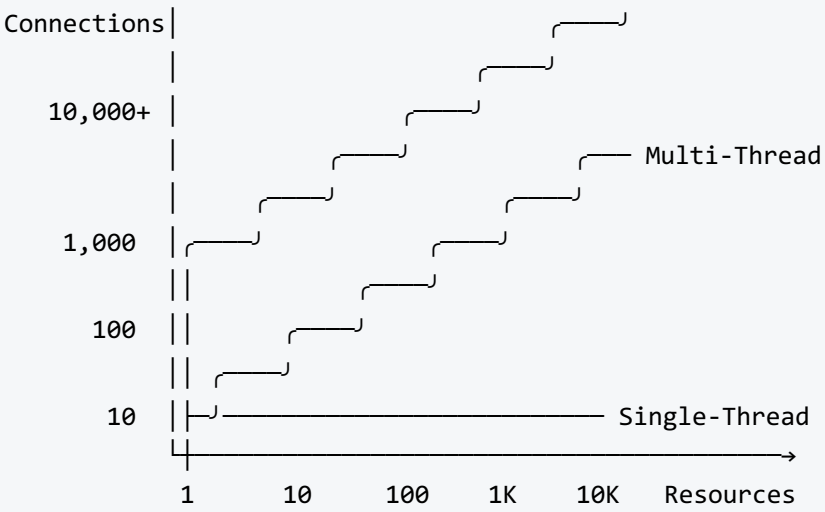
3.2. Khả năng mở rộng (Scalability)

Mô hình	Horizontal	Vertical	Connection Scaling
Đơn luồng	Không	Không	O(1)
Đa luồng	Có	Có	O(threads)
Event-Driven	Tốt	Tốt	O(10K+)

Phân tích:

```
ĐƠN LUỒNG:
• Không scale - 1 request tại 1 thời điểm
• Thêm cores không giúp ích
• Bottleneck: single thread

ĐA LUỒNG:
• Scale với số cores (đến giới hạn)
• Giới hạn:
  - Thread pool size (1000s threads → overhead)
  - Memory: 1MB stack/thread
  - Context switching cost
• C10K problem: 10,000 connections = 10GB RAM

EVENT-DRIVEN:
• Handles C10K, even C100K easily
• Minimal memory per connection (~KB)
• Single thread per core is efficient
• Scale out: multiple event loops on multiple cores
```

```
Connection Scaling Comparison:

Concurrent │                                ┌─ Event-Driven
```



3.3. Độ phức tạp phát triển (Development Complexity)

Mô hình	Code Complexity	Debugging	Reasoning
Đơn luồng	Rất đơn giản	Dễ	Sequential
Đa luồng	Phức tạp	Khó	Concurrent
Event-Driven	Trung bình	Trung bình	State machine

So sánh code:

```
# ĐƠN LUỒNG - Simple, sequential
def handle_request(socket):
    request = socket.recv()        # Blocking
    data = read_file(request.path) # Blocking
    socket.send(data)              # Blocking
    socket.close()

while True:
    client = server.accept()       # Blocking
    handle_request(client)
```

```
# ĐA LUỒNG - Concurrent, needs synchronization
from threading import Thread, Lock

file_cache = {}
cache_lock = Lock()

def handle_request(socket):
    request = socket.recv()

    with cache_lock:               # Critical section
```

```
        if request.path in file_cache:
            data = file_cache[request.path]
        else:
            data = read_file(request.path)
            file_cache[request.path] = data

    socket.send(data)
    socket.close()

while True:
    client = server.accept()
    Thread(target=handle_request, args=(client,)).start()

# Challenges: Race conditions, deadlocks, debugging
```

```
# EVENT-DRIVEN - State machine, callback-based
import select

class ConnectionState:
    READING_REQUEST = 1
    READING_FILE = 2
    SENDING_RESPONSE = 3

connections = {}  # fd -> state

def event_loop():
    while True:
        readable, writable, _ = select.select(read_fds, write_fds, [])

        for fd in readable:
            state = connections[fd]
            if state.phase == READING_REQUEST:
                data = fd.recv(1024)  # Non-blocking
                if complete_request(data):
                    state.phase = READING_FILE
                    start_async_read(state.path)
            elif state.phase == READING_FILE:
                # File data ready
                state.response = state.file_data
                state.phase = SENDING_RESPONSE

        for fd in writable:
            if connections[fd].phase == SENDING_RESPONSE:
                sent = fd.send(state.response)  # Non-blocking
                if all_sent:
```

```
                    cleanup(fd)

# Challenges: Callback hell, state management
```

3.4. Khả năng chịu lỗi (Fault Tolerance)

Mô hình	Error Isolation	Recovery	Partial Failure
Đơn luồng	Tốt nhất	Đơn giản	Crash all
Đa luồng	Kém	Phức tạp	Có thể crash process
Event-Driven	Tốt	Tốt	Isolated per connection

Phân tích:

```
ĐƠN LUỒNG:
• Error trong 1 request → chỉ request đó fail
• Dễ recover: try-catch, restart loop
• Nhược điểm: 1 request treo → cả server treo

ĐA LUỒNG:
• Error trong 1 thread có thể:
  - Corrupt shared state
  - Deadlock các threads khác
  - Crash toàn bộ process (uncaught exception)
• Khó debug race conditions
• Cần careful exception handling

EVENT-DRIVEN:
• Error trong 1 connection handler:
  - Cleanup connection đó
  - Các connections khác không bị ảnh hưởng
• Single point of failure: event loop crash → all down
• Solution: Multiple event loops, process supervision
```

4. Bảng tổng hợp đánh giá

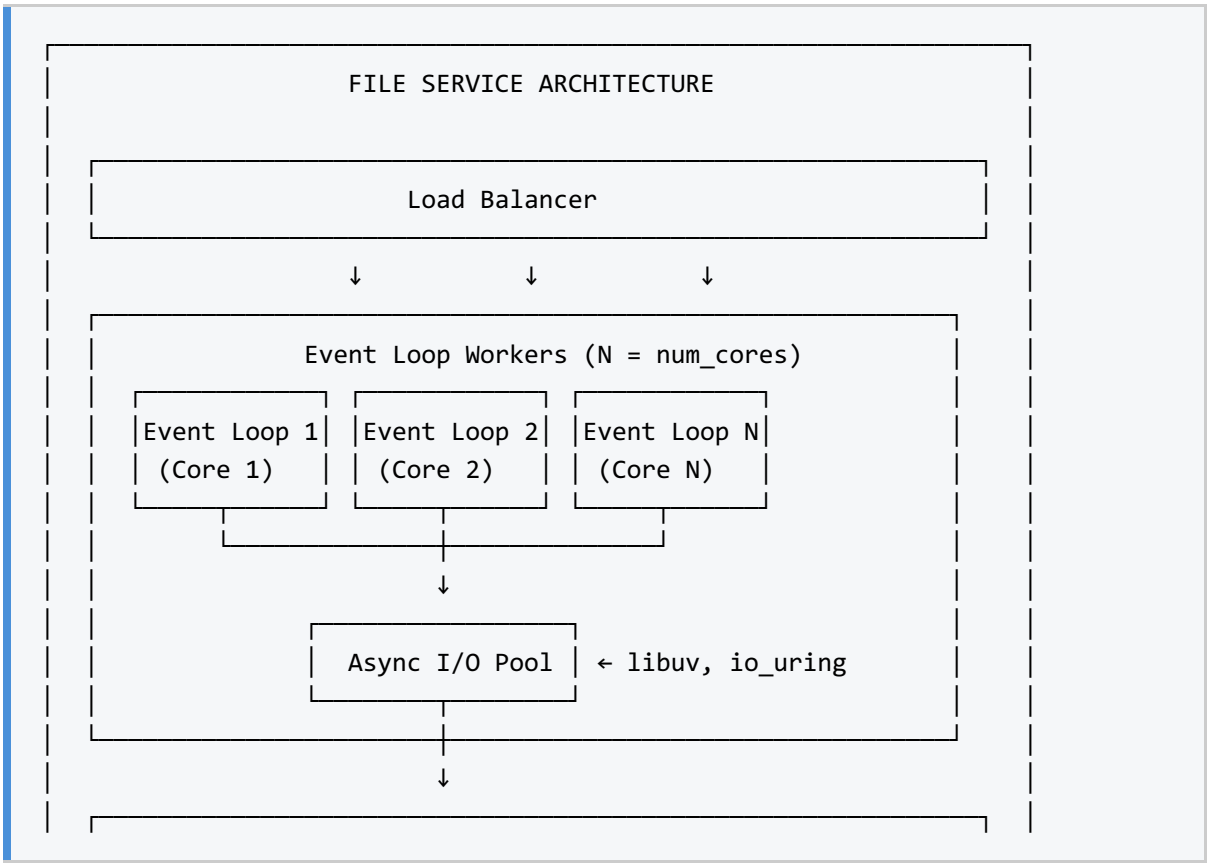
Tiêu chí	Đơn luồng	Đa luồng	Event-Driven
Hiệu năng	1/5	4/5	5/5
Khả năng mở rộng	1/5	3/5	5/5
Độ phức tạp phát triển	5/5	2/5	3/5
Khả năng chịu lỗi	3/5	2/5	4/5
TỔNG	10/20	11/20	17/20

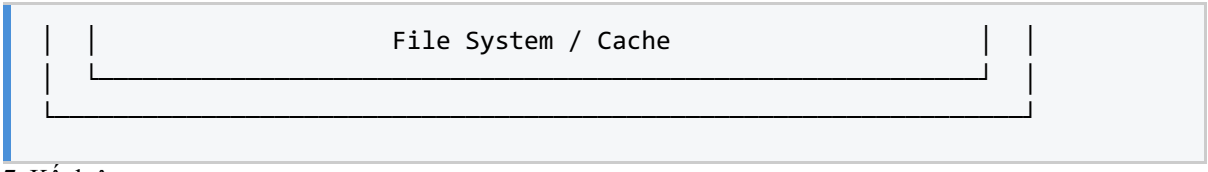
5. Khuyến nghị cho File Service

```
┌─────────────────────────────────────────────────────────┐
│  KHUYẾN NGHỊ: EVENT-DRIVEN FSM                          │
├─────────────────────────────────────────────────────────┤
│                                                         │
│  Lý do chọn Event-Driven cho Read-Heavy File Service:   │
```

```
┌                                                                    ┐
│                                                                    │
│  1.  I/O-bound workload phù hợp hoàn hảo                           │
│     - 95% thời gian chờ I/O → event-driven tận dụng tốt            │
│                                                                    │
│  2.  Handles thousands of concurrent readers                       │
│     - File service cần serve nhiều clients đồng thời              │
│     - Event-driven: C10K+ với minimal resources                    │
│                                                                    │
│  3.  Read operations are independent                               │
│     - Không cần synchronization phức tạp như write                 │
│     - State machine đơn giản: ACCEPT → READ → SEND                 │
│                                                                    │
│  4.  Memory efficient                                              │
│     - Quan trọng khi có nhiều connections                          │
│                                                                    │
│  5.  Fault isolation per connection                                │
│     - 1 connection lỗi không ảnh hưởng các connections khác        │
│                                                                    │
├────────────────────────────────────────────────────────────────────┤
│  Hybrid approach (Production):                                     │
│  • Event loop + Thread pool for file I/O                           │
│  • Multiple event loops (1 per core) + work stealing               │
│  • Examples: Nginx, Redis, Node.js                                 │
└                                                                    ┘
```

6. Kiến trúc đề xuất





7. Kết luận

Với đặc điểm **read-heavy, write-light** và **I/O-bound** của dịch vụ tập tin:

Lựa chọn	Đánh giá
Đơn luồng	Không phù hợp - quá chậm
Đa luồng	Có thể dùng - nhưng tốn resources
Event-Driven FSM	Phù hợp nhất - hiệu năng cao, scale tốt

Ví dụ thực tế: Nginx (web server, file serving) sử dụng event-driven architecture và có thể xử lý hàng chục nghìn connections đồng thời với minimal resources.

Câu 9:

• Hãy định nghĩa ảo hóa (virtualization) và nêu vai trò chính của kỹ thuật ảo hóa trong các hệ thống phân tán.

• Phân loại các hình thức ảo hóa dựa trên lớp giao diện (instruction set virtualization vs. system level virtualization):

o Máy ảo tiến trình (process level VM, ví dụ Java Runtime).

o Giám sát máy ảo (hypervisor based VM).

Mô tả cơ chế hoạt động và điểm khác biệt cơ bản giữa hai hình thức này.

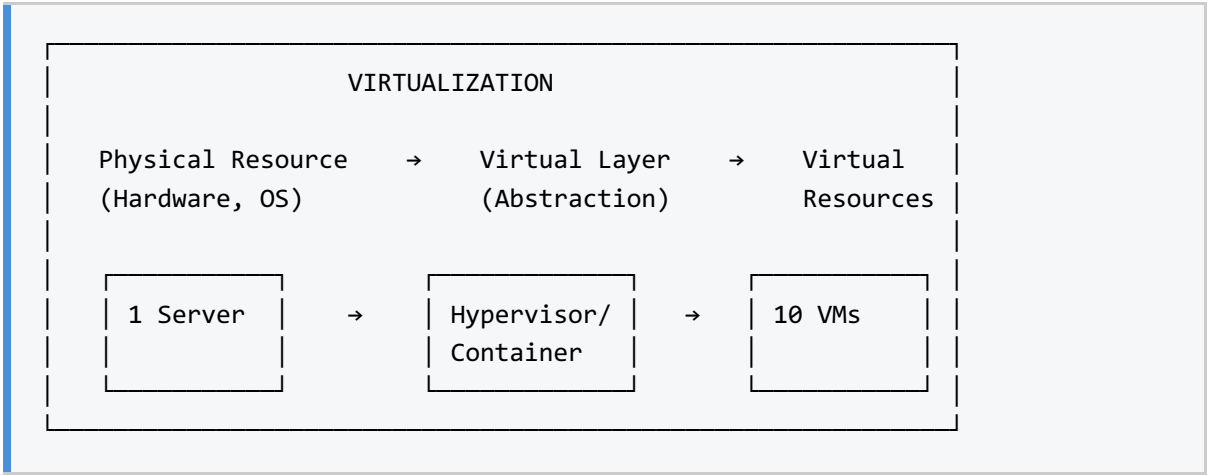
Trả lời:

1. Định nghĩa Ảo hóa (Virtualization)

Ảo hóa (Virtualization) là kỹ thuật tạo ra một phiên bản ảo (virtual) của tài nguyên máy tính, bao gồm phần cứng, hệ điều hành, thiết bị lưu trữ, hoặc tài nguyên mạng, cho phép nhiều hệ thống hoặc ứng dụng chia sẻ cùng một tài nguyên vật lý một cách độc lập và cô lập.

Định nghĩa hình thức:

> Virtualization = Mở rộng hoặc thay thế một **interface** hiện có để mô phỏng hành vi của một hệ thống khác.



2. Vai trò của Ảo hóa trong Hệ thống Phân tán

2.1. Các vai trò chính:

Vai trò	Mô tả	Ví dụ
Resource Sharing	Chia sẻ tài nguyên vật lý giữa nhiều tenants	Cloud multi-tenancy
Isolation	Cô lập các ứng dụng/users khỏi nhau	Security, fault containment
Portability	Chạy ứng dụng trên nhiều nền tảng khác nhau	Java "Write Once, Run Anywhere"

Migration	Di chuyển workloads giữa các servers	Live VM migration
Scalability	Tạo/xóa resources theo nhu cầu	Auto-scaling
Resource Efficiency	Tận dụng tối đa phần cứng	Server consolidation

2.2. Minh họa trong Distributed Systems:

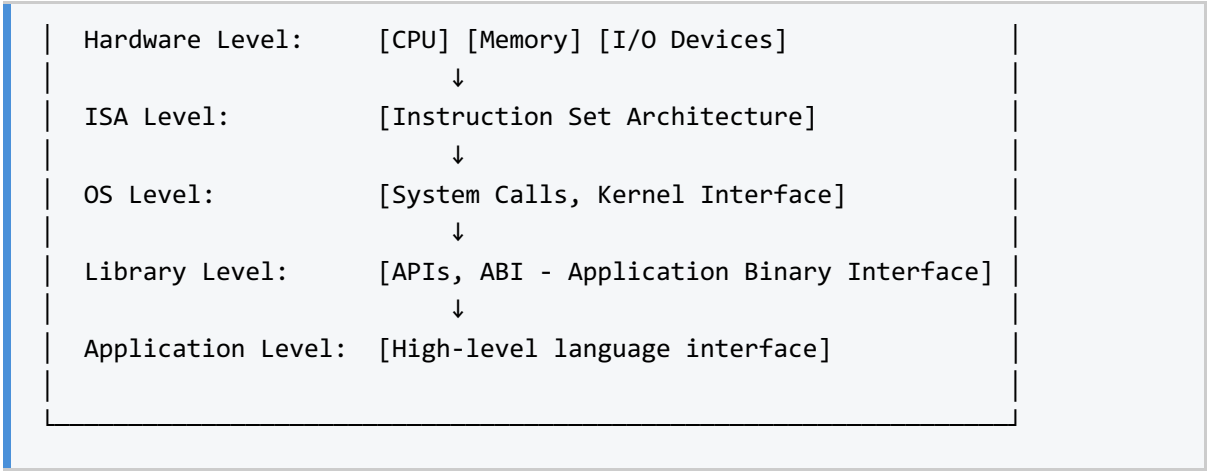
```
TRƯỚC ẢO HÓA:
┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐
│Server 1  │ │Server 2  │ │Server 3  │ │Server 4  │
│ App A    │ │ App B    │ │ App C    │ │ App D    │
│ 10% CPU  │ │ 15% CPU  │ │ 20% CPU  │ │  5% CPU  │
└──────────┘ └──────────┘ └──────────┘ └──────────┘

→ 4 servers, average 12.5% utilization = WASTEFUL
 SAU ẢO HÓA:
┌──────────────────────────────────────────────────┐
│                    Server 1                      │
│  ┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────┐ │
│  │  VM A    │ │  VM B    │ │  VM C    │ │VM D  │ │
│  │ App A    │ │ App B    │ │ App C    │ │App D │ │
│  └──────────┘ └──────────┘ └──────────┘ └──────┘ │
│                   50% CPU                        │
└──────────────────────────────────────────────────┘

→ 1 server, 50% utilization = EFFICIENT
```

2.3. Vai trò trong Cloud Computing:

```
┌──────────────────────────────────────────────────────────────┐
│                    CLOUD INFRASTRUCTURE                      │
│                                                              │
│  ┌──────────────────────────────────────────────────────┐    │
│  │                  Virtualization Layer                │    │
│  │  ┌──────┐ ┌──────────┐ ┌──────────┐ ┌──────────────┐ │    │
│  │  │ VMs  │ │Containers│ │Serverless│ │Virtual Networks│ │    │
│  │  └──────┘ └──────────┘ └──────────┘ └──────────────┘ │    │
│  └──────────────────────────────────────────────────────┘    │
│                          ↓                                   │
│  Benefits:                                                   │
│  • IaaS: Cung cấp VMs theo yêu cầu                           │
│  • PaaS: Isolate applications                                │
│  • SaaS: Multi-tenant deployment                             │
│  • Elasticity: Scale up/down dynamically                     │
└──────────────────────────────────────────────────────────────┘
```

3. Phân loại ảo hóa theo lớp giao diện

```
┌──────────────────────────────────────────────────────────────┐
│                  VIRTUALIZATION TAXONOMY                     │
├──────────────────────────────────────────────────────────────┤
│                                                              │
```



3.1. Process-Level VM (Máy ảo tiến trình)

Ví dụ: Java Virtual Machine (JVM), .NET CLR, Python interpreter

Đặc điểm:

- Ảo hóa tại mức **ứng dụng/tiến trình**
- Mỗi process có một VM instance riêng
- Thường sử dụng **bytecode interpretation** hoặc **JIT compilation**

3.2. System-Level VM (Hypervisor-based)

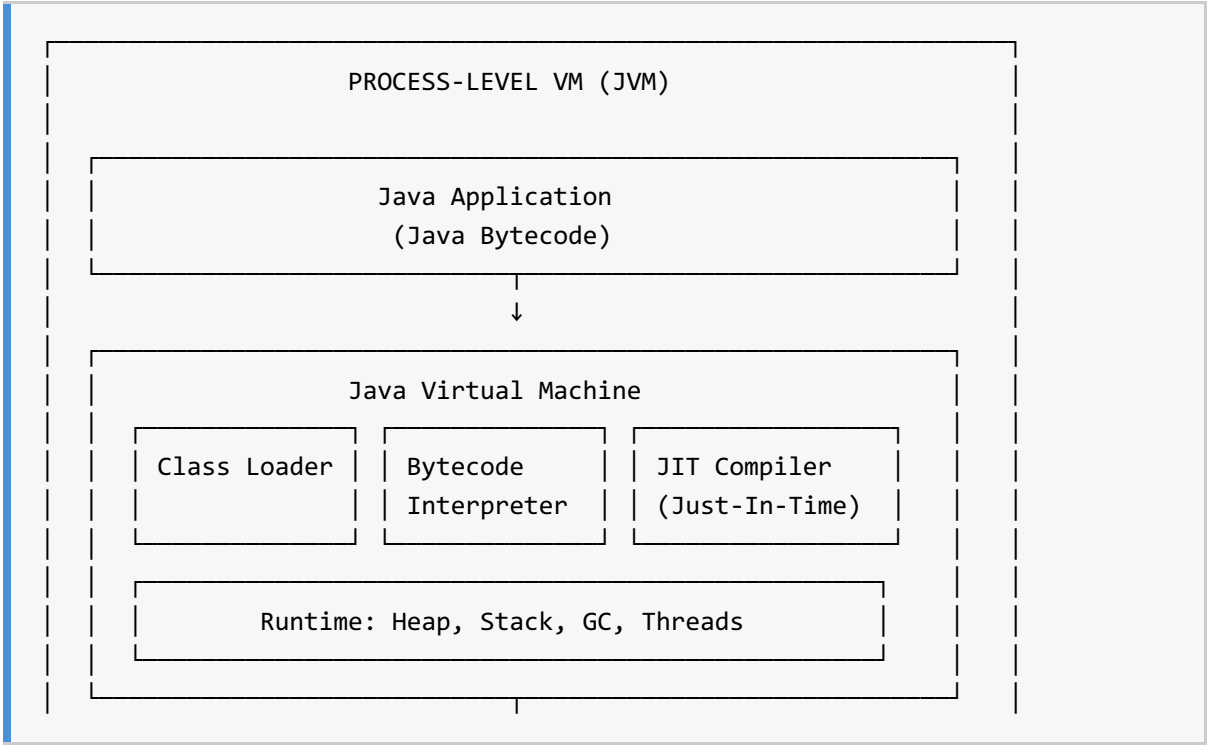
Ví dụ: VMware ESXi, KVM, Hyper-V, VirtualBox

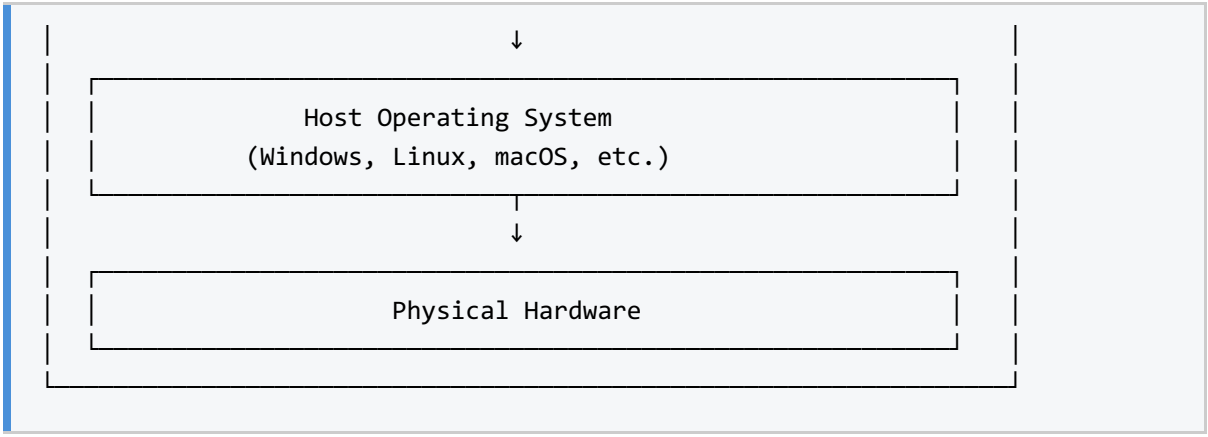
Đặc điểm:

- Ảo hóa toàn bộ **hệ thống phần cứng**
- Chạy complete OS trong mỗi VM
- Cô lập hoàn toàn giữa các VMs

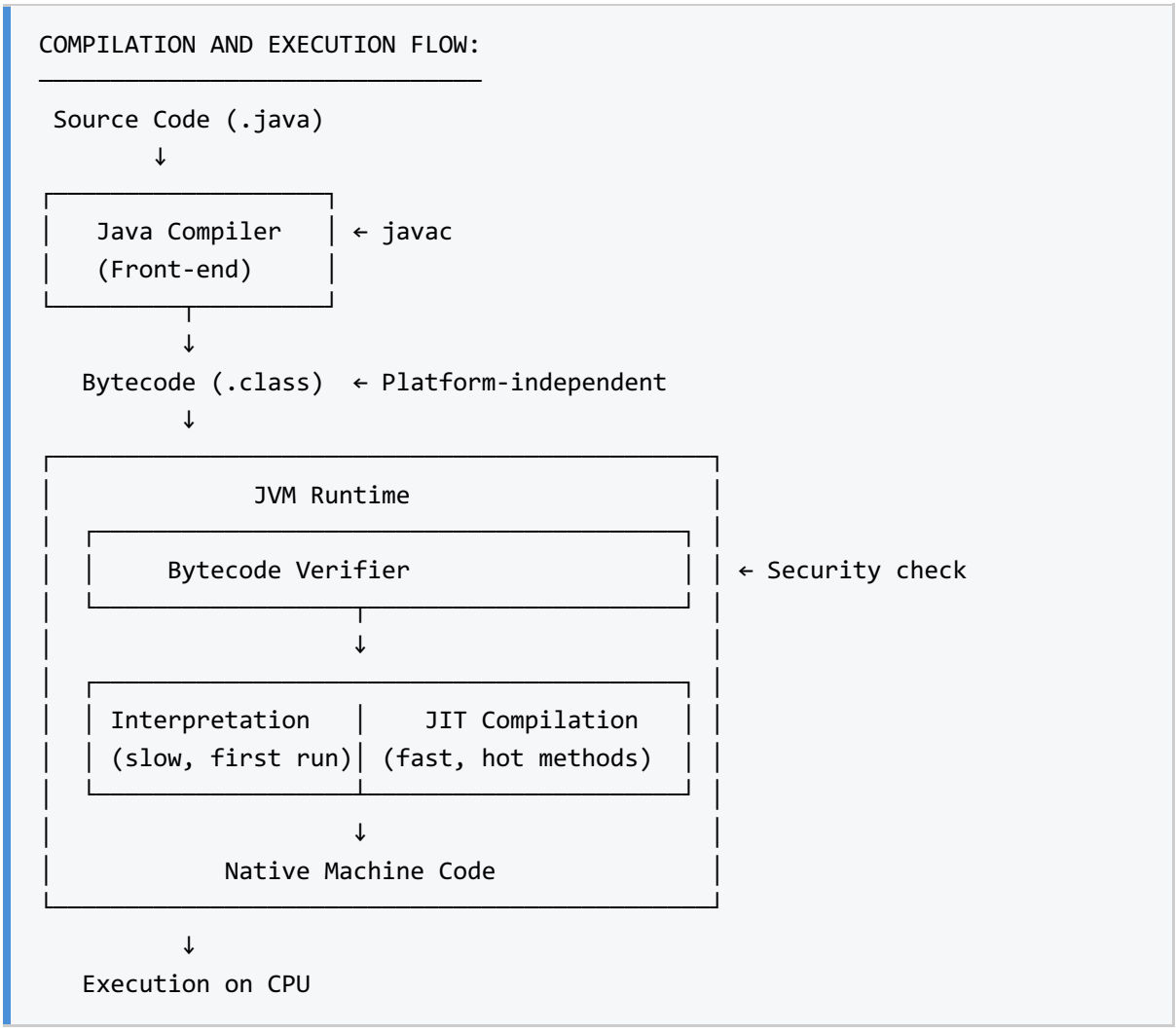
4. Máy ảo tiến trình (Process-Level VM)

4.1. Kiến trúc:





4.2. Cơ chế hoạt động:



Key mechanisms:

1. **Bytecode Interpretation:**

Bytecode: iconst_1, iconst_2, iadd, ireturn

↓

Interpreter:

- Read iconst_1 → push 1 to stack
- Read iconst_2 → push 2 to stack
- Read iadd → pop 2 values, add, push result

- Read ireturn → return value

2. **JIT Compilation (Just-In-Time):**

Hot method detected (called frequently)

↓

JIT Compiler compiles to native code

↓

Native code cached for future calls

↓

Subsequent calls → direct native execution (fast)

3. **Garbage Collection:**

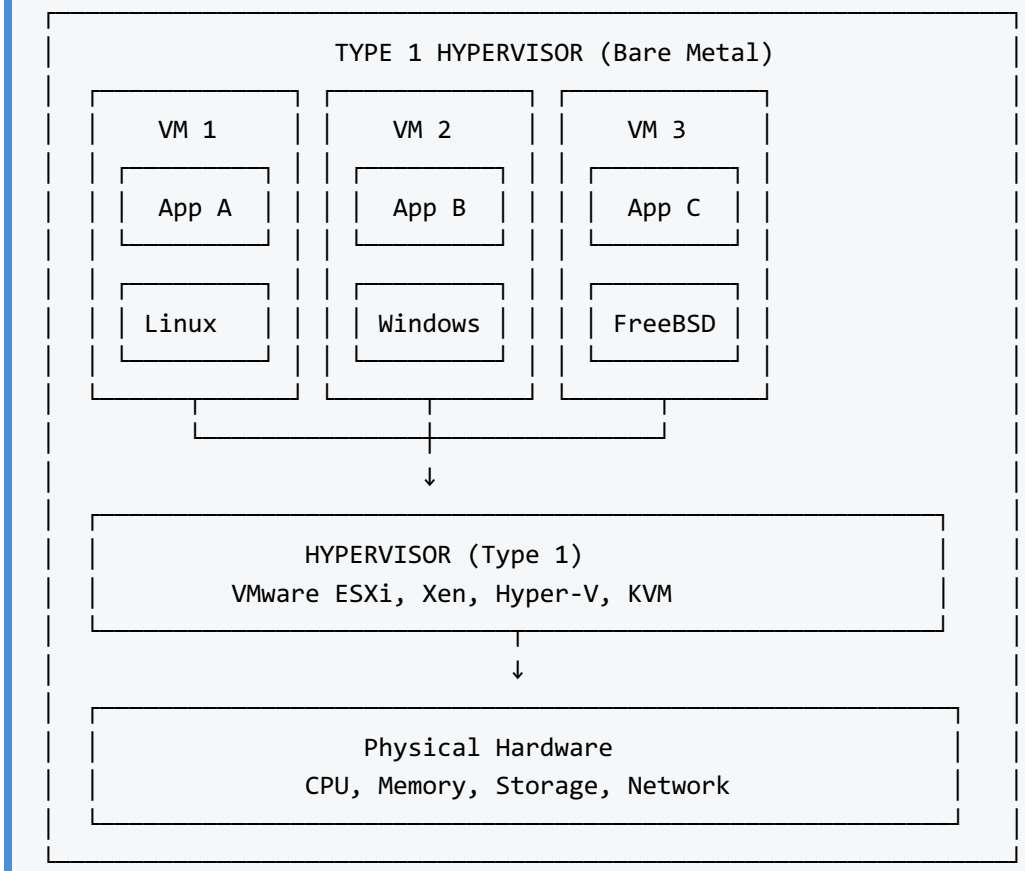
Automatic memory management

- Track object references
- Reclaim unreachable objects
- No manual free() needed

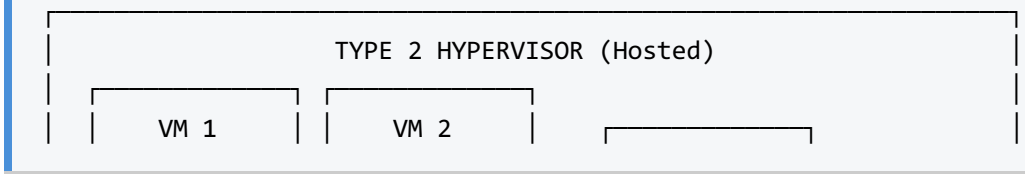
5. Giám sát máy ảo (Hypervisor-based VM)

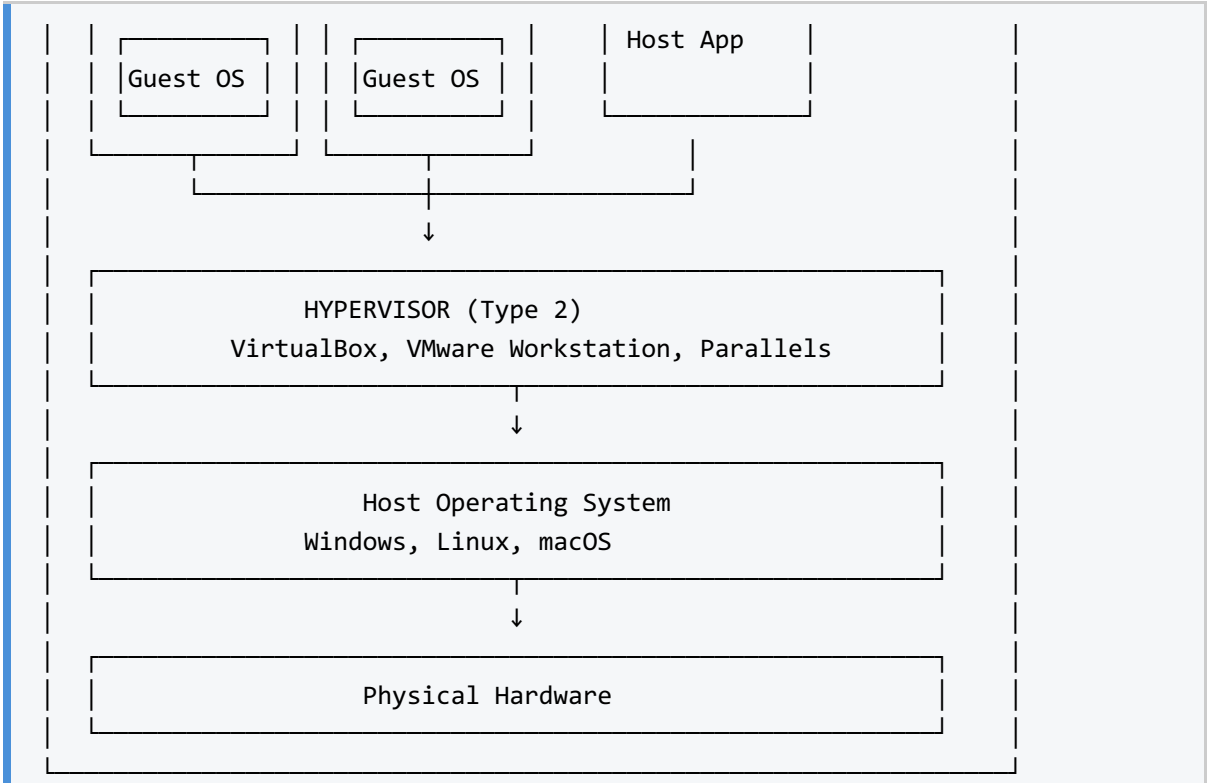
5.1. Kiến trúc:

Type 1 Hypervisor (Bare Metal):



Type 2 Hypervisor (Hosted):





5.2. Cơ chế hoạt động:

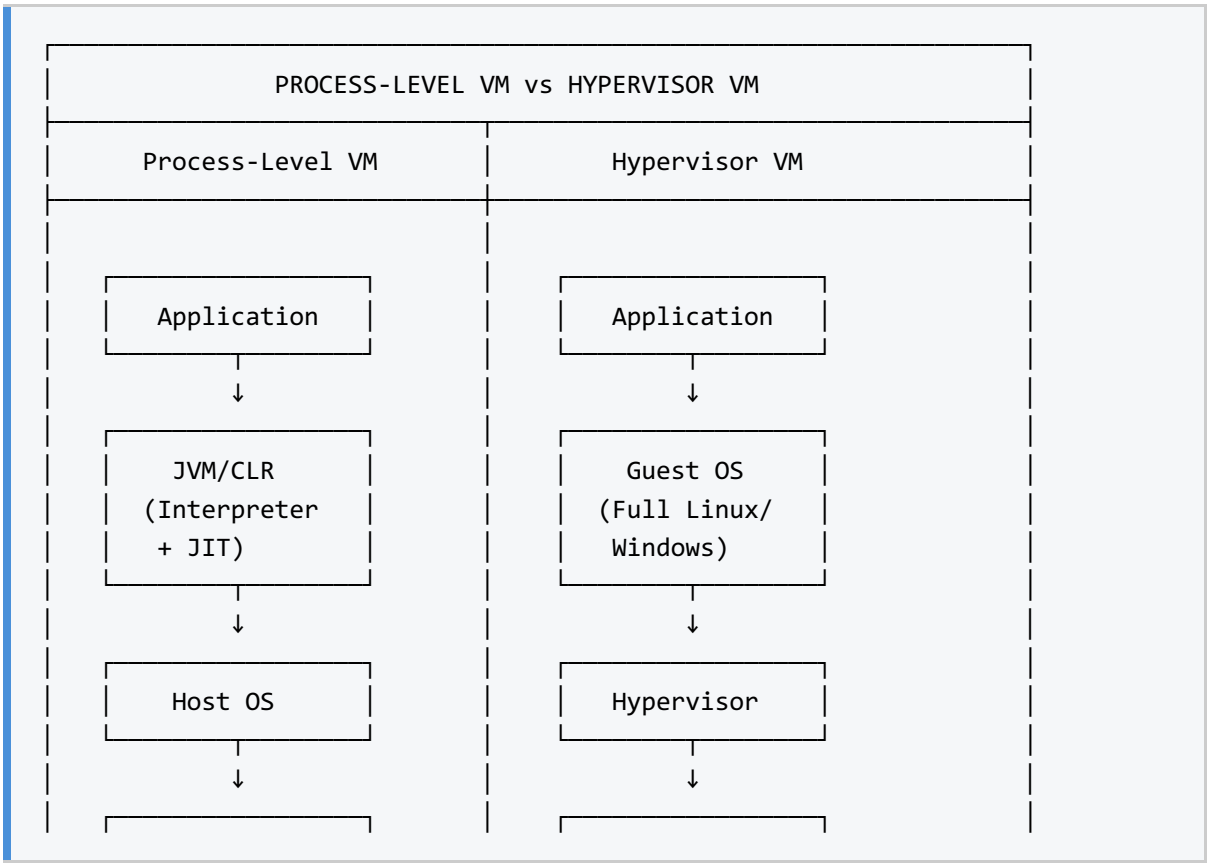
```
HARDWARE VIRTUALIZATION MECHANISMS:
═══════════════════════════════════

1. CPU VIRTUALIZATION:

    ┌─────────────────────────────────────────┐
    │  Guest OS thinks it's running in Ring 0 │
    │                    ↓                    │
    │  Hypervisor intercepts privileged ops   │
    │                    ↓                    │
    │  - Binary Translation (old method)      │
    │  - Hardware-assisted (VT-x, AMD-V)      │
    │                    ↓                    │
    │  Emulate or execute on real hardware    │
    └─────────────────────────────────────────┘

2. MEMORY VIRTUALIZATION:

    ┌─────────────────────────────────────────┐
    │  Guest Physical Address (GPA)           │
    │              ↓                          │
    │  Hypervisor maintains page tables       │
    │              ↓                          │
    │  Host Physical Address (HPA)            │
    │                                         │
    │  Technologies: Shadow Page Tables, EPT  │
    └─────────────────────────────────────────┘

3. I/O VIRTUALIZATION:
```

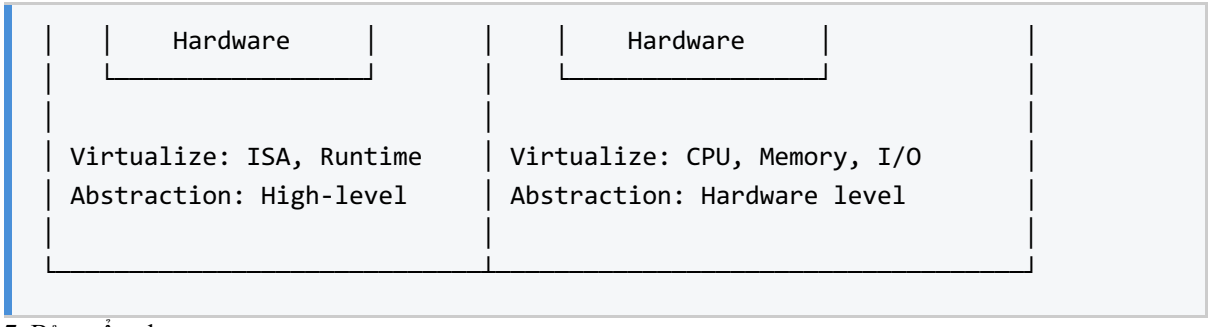
```
┌─────────────────────────────────────────┐
│  Guest I/O request                      │
│              ↓                          │
│  Hypervisor intercepts                  │
│              ↓                          │
│  - Emulation (slow)                     │
│  - Para-virtualization (fast, modified) │
│  - Direct assignment (fastest)          │
│              ↓                          │
│  Real hardware I/O                      │
└─────────────────────────────────────────┘
```

6. So sánh hai hình thức ảo hóa

Tiêu chí	Process-Level VM (JVM)	Hypervisor-based VM
Mức ảo hóa	Application/Runtime	Full Hardware
Guest OS	Không cần	Cần complete OS
Isolation	Process-level	Complete system
Performance	Gần native (với JIT)	2-10% overhead
Portability	Bytecode portable	VM image portable
Startup time	Milliseconds	Seconds to minutes
Memory overhead	~50-200 MB	~512 MB - GB
Use case	Cross-platform apps	Server consolidation

Chi tiết so sánh:

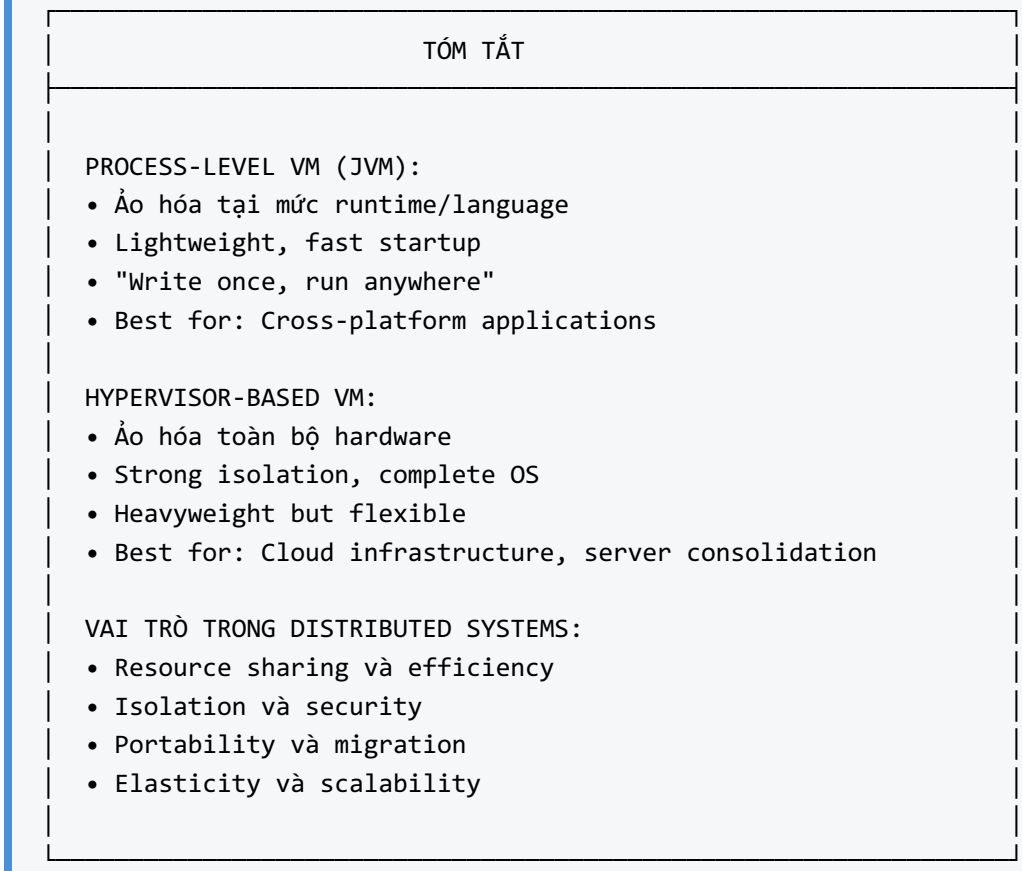




7. Bảng tổng hợp

Aspect	Process VM (JVM)	Hypervisor VM
Virtualized	Language runtime, bytecode	Complete hardware
Goal	Portability, managed runtime	Isolation, consolidation
Examples	JVM, CLR, Python, V8	VMware, KVM, Hyper-V
OS needed	Shared host OS	Each VM has own OS
Security	Sandboxed process	Strong isolation
Resources	Lightweight	Heavyweight
Boot time	Instant	Slow (minutes)
Use case	Enterprise apps, web servers	Cloud, data centers

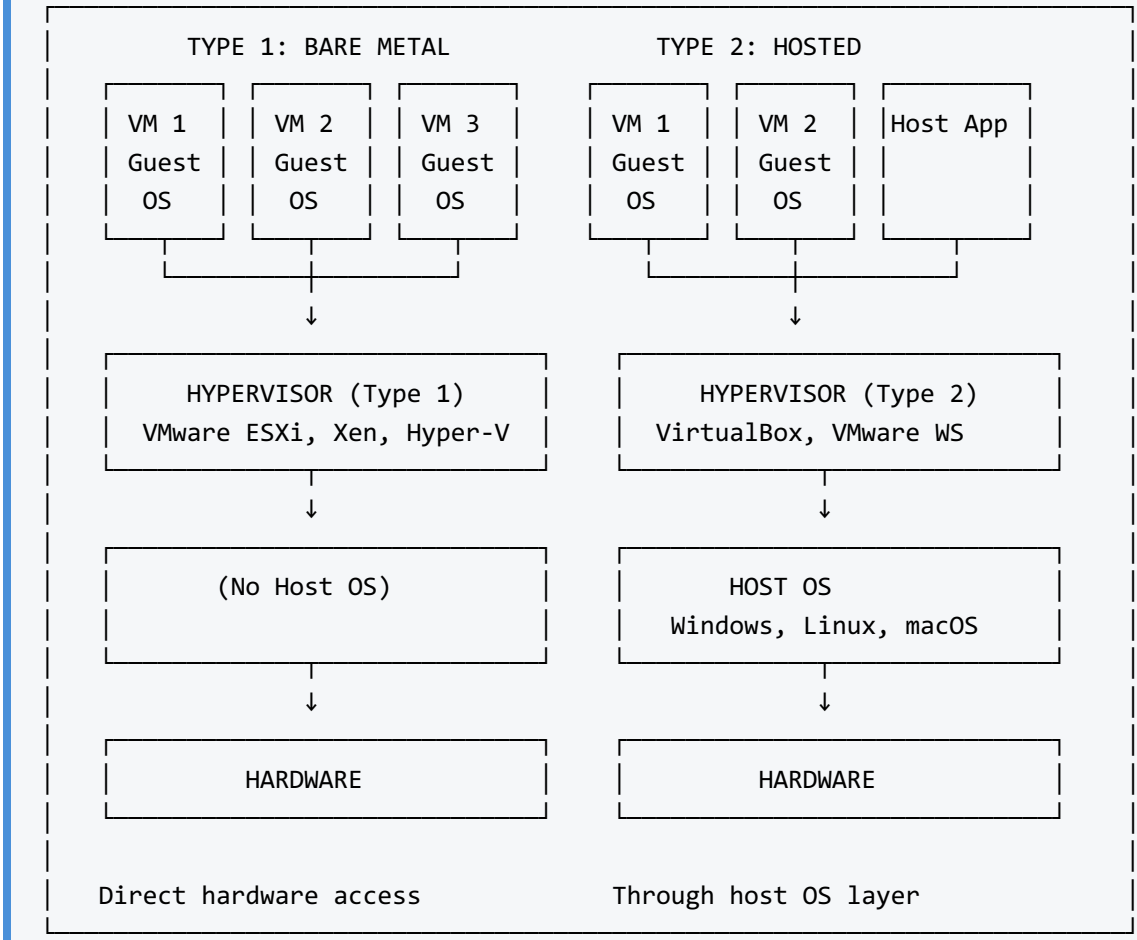
8. Kết luận



Câu 10:

So sánh hypervisor thuần (bare metal) với hypervisor lưu trữ (hosted hypervisor) về các khía cạnh:

o hiệu năng I/O và CPU
o chi phí phát triển, vận hành
o khả năng sử dụng lại trình điều khiển thiết bị (device drivers)
Trình bày ưu – nhược điểm của mỗi mô hình.
Trong bối cảnh điện toán đám mây công cộng (public cloud), hãy đánh giá mức độ phù hợp của ảo hóa dựa trên hypervisor truyền thống so với ảo hóa cấp container (container based virtualization) cho các dịch vụ:
o (i) nền tảng PaaS (Platform as a Service) với tính năng mở rộng nhanh,
o (ii) hạ tầng IaaS cho phép người dùng tùy chỉnh kernel.
Đưa ra khuyến nghị và giải thích trade off về bảo mật, hiệu năng, và tính di động.
Trả lời:
PHẦN 1: So sánh Bare Metal vs Hosted Hypervisor
1. Kiến trúc hai loại Hypervisor



2. So sánh hiệu năng I/O và CPU

2.1. Hiệu năng CPU

Tiêu chí	Bare Metal (Type 1)	Hosted (Type 2)
Overhead	2-5%	5-15%
Direct hardware access	Có	Qua Host OS
Hardware-assisted virt	Tối ưu	Có nhưng qua layer
Context switch	Nhanh	Chậm hơn

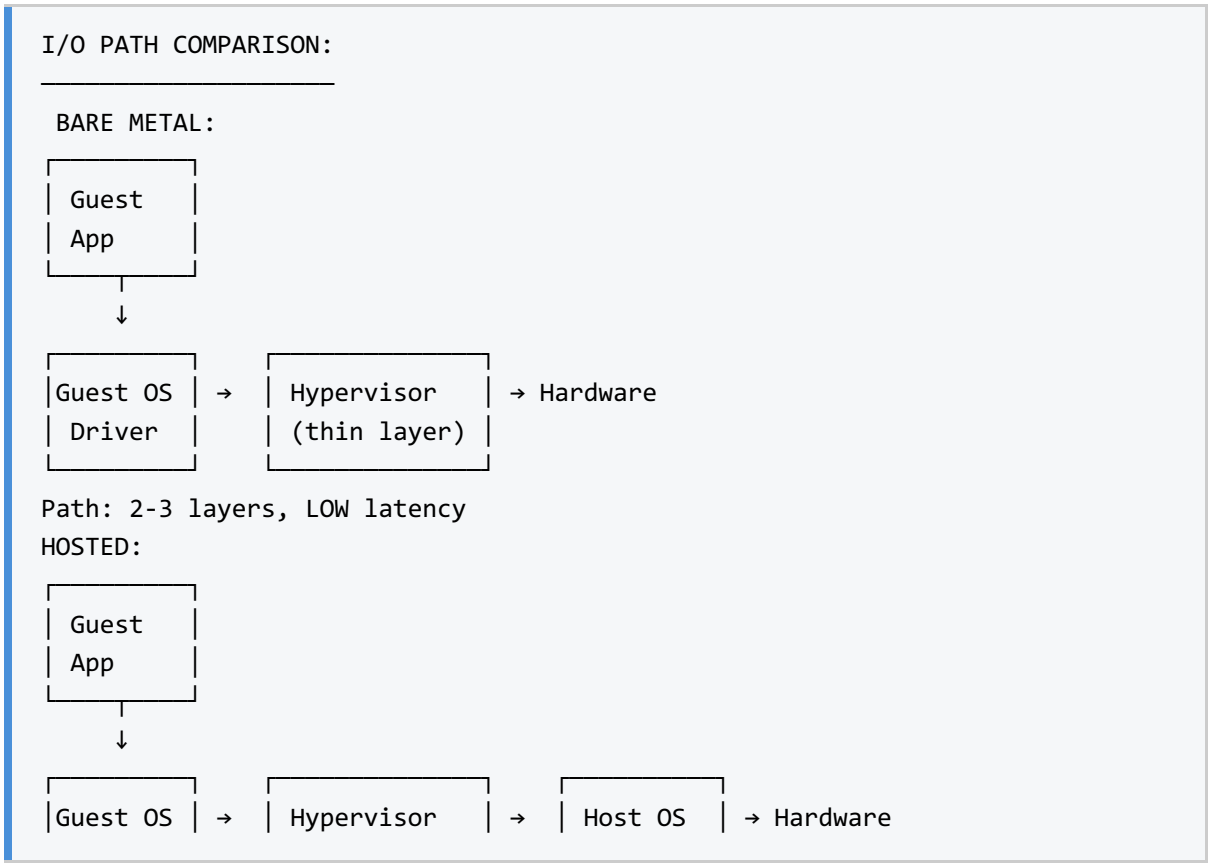
```
CPU EXECUTION PATH:
────────────────────

BARE METAL:
Guest App → Guest OS → Hypervisor → CPU
                            ↓
               (Direct execution với VT-x/AMD-V)
HOSTED:
Guest App → Guest OS → Hypervisor → Host OS → CPU
                                        ↓
               (Extra layer = extra overhead)
```

Benchmark ước tính:

Workload	Bare Metal	Hosted	Native
CPU-intensive	97-98%	90-95%	100%
Memory-intensive	95-97%	88-93%	100%
Mixed	95-98%	85-92%	100%

2.2. Hiệu năng I/O

Tiêu chí	Bare Metal (Type 1)	Hosted (Type 2)
Disk I/O	90-98% native	60-85% native
Network I/O	95-99% native	70-90% native
Latency	Thấp	Cao hơn
Direct device access	Có (SR-IOV)	Hạn chế



```
| Driver  |    |              |    | Driver   |
└─────────┘    └──────────────┘    └──────────┘

 Path: 4-5 layers, HIGH latency
```

3. Chi phí phát triển và vận hành

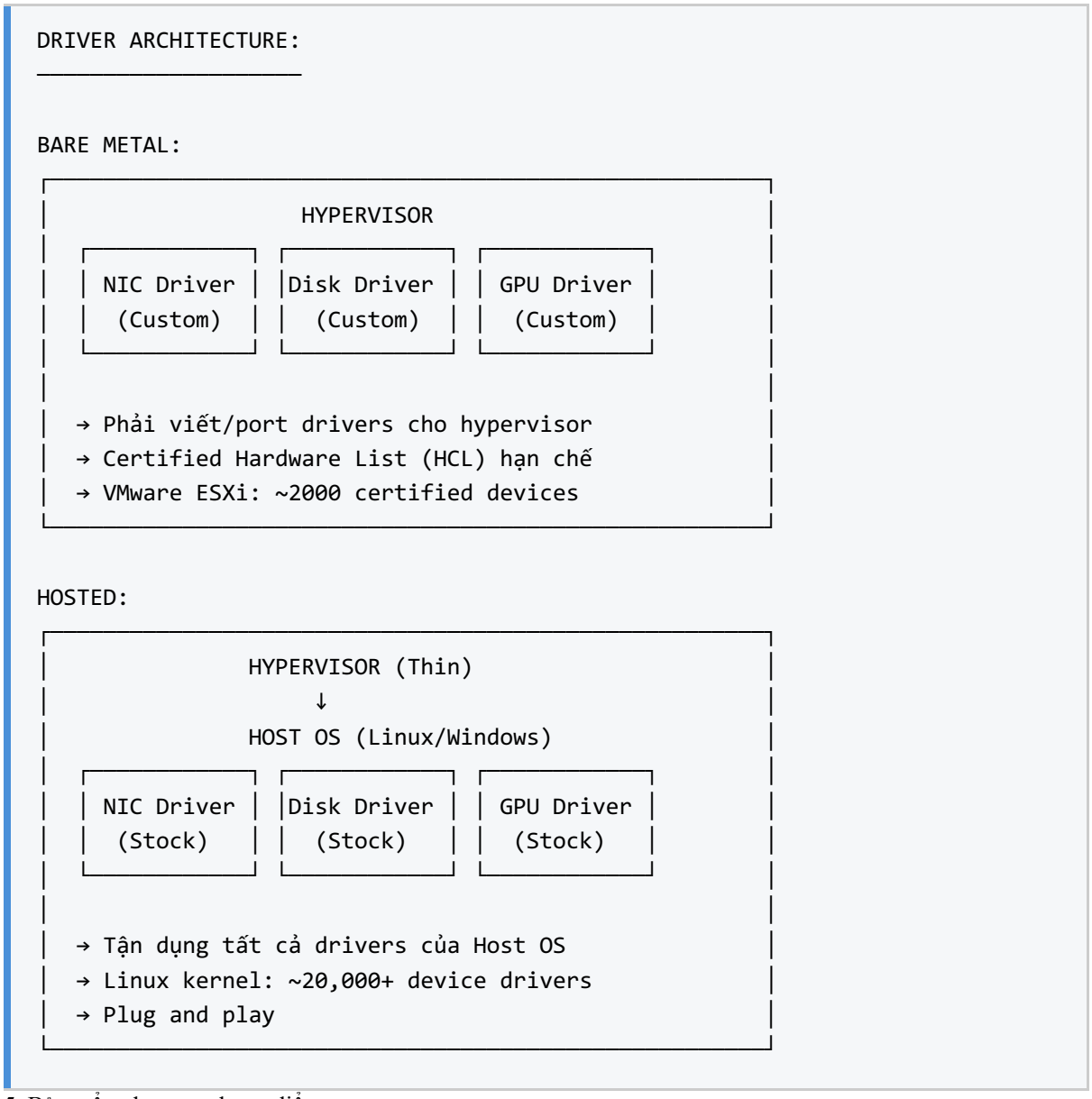
Tiêu chí	Bare Metal (Type 1)	Hosted (Type 2)
Development cost	Cao	Thấp
Hardware compatibility	Hạn chế	Rộng (dùng Host drivers)
Deployment complexity	Phức tạp	Đơn giản
Management tools	Enterprise-grade	Basic
Licensing cost	Cao (VMware, etc.)	Thường miễn phí
Operational expertise	Cần chuyên gia	Dễ sử dụng
Scalability	Tốt cho data center	Giới hạn

Chi phí ước tính:

```
BARE METAL (Enterprise):
┌──────────────────────────────────────────┐
│ VMware vSphere Enterprise Plus:          │
│ • License: ~$5,000/CPU                   │
│ • Support: ~$1,200/year                  │
│ • Training: ~$3,000                      │
│ • Hardware certified: +20% cost          │
│ • Dedicated admin: $80,000+/year         │
│                                          │
│ TCO (3 years, 10 servers): ~$150,000+    │
└──────────────────────────────────────────┘

HOSTED (Desktop/Dev):
┌──────────────────────────────────────────┐
│ VirtualBox:                              │
│ • License: FREE (GPL)                    │
│ • Support: Community                     │
│ • Training: Minimal                      │
│ • Any hardware                           │
│ • No dedicated admin needed              │
│                                          │
│ TCO (3 years): ~$0 - $1,000              │
└──────────────────────────────────────────┘
```

4. Khả năng sử dụng lại Device Drivers

Tiêu chí	Bare Metal (Type 1)	Hosted (Type 2)
Driver reuse	Cần drivers riêng	Dùng Host OS drivers
Hardware support	Certified list only	Rất rộng
Driver development	Tốn kém	Không cần
New hardware	Chờ vendor support	Ngay khi Host OS support



5. Bảng tổng hợp ưu nhược điểm

Bare Metal Hypervisor (Type 1):

Ưu điểm	Nhược điểm
Hiệu năng cao nhất	Chi phí cao
Bảo mật tốt (no Host OS)	Hardware compatibility hạn chế
Scalable cho data center	Phức tạp để setup
Enterprise management tools	Cần chuyên gia
Live migration, HA	Cần drivers riêng

Best for: Data centers, production workloads, cloud infrastructure

Hosted Hypervisor (Type 2):

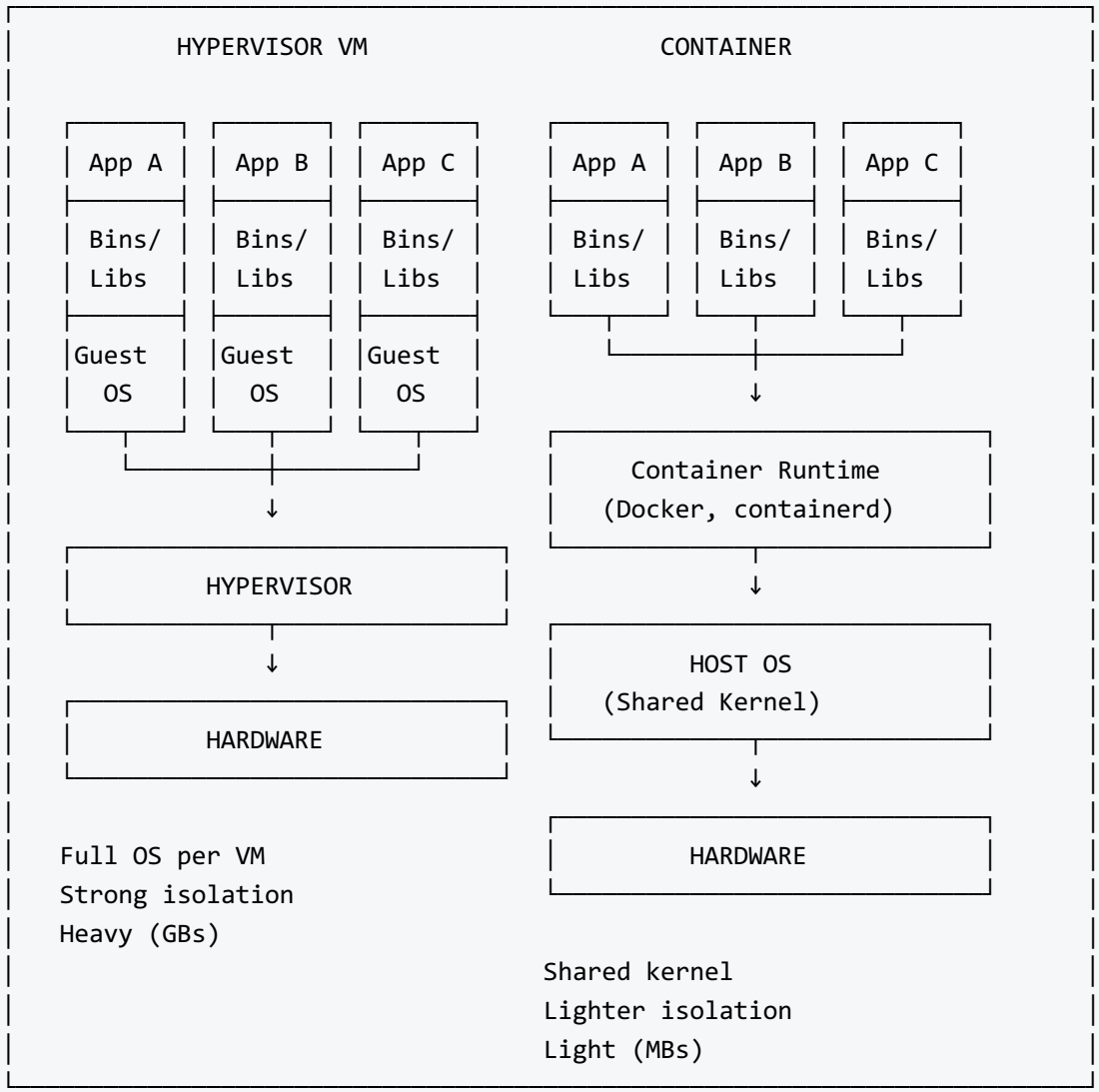
Ưu điểm	Nhược điểm
Dễ cài đặt và sử dụng	Hiệu năng thấp hơn
Chi phí thấp/miễn phí	Host OS overhead
Hardware compatibility rộng	Bảo mật yếu hơn

Tận dụng Host OS drivers	Không phù hợp production
Chạy song song với Host apps	Scalability hạn chế

Best for: Development, testing, desktop virtualization, learning

PHẦN 2: Hypervisor vs Container trong Public Cloud

6. So sánh Hypervisor và Container



Bảng so sánh chi tiết:

Tiêu chí	Hypervisor VM	Container
Isolation	Complete	Process-level
Startup time	30s - 5min	100ms - 1s
Resource overhead	500MB - GBs	10-100 MBs
Density	10s VMs/server	100s-1000s/server
Portability	VM images (large)	Container images (small)
Kernel customization	Yes (own kernel)	No (shared kernel)
Security	Strong	Good (improving)

Performance	95-98% native	99%+ native

7. Đánh giá cho từng use case

7.1. PaaS với tính năng mở rộng nhanh

Yêu cầu PaaS:

- Scale up/down nhanh (seconds)
- □ Deploy ứng dụng thường xuyên
- □ Chi phí hiệu quả (high density)
- CI/CD integration
- Resource efficiency

```
PAAS SCALING SCENARIO:
──────────────────────

Traffic Spike: 100 → 1000 requests/sec

HYPERVISOR VM:
┌──────────────────────────────────────────────────┐
│ T=0:  [VM1] [VM2] [VM3]                          │
│ T=30s: Starting new VMs...                       │
│ T=60s: [VM1] [VM2] [VM3] [VM4] [VM5]             │
│ T=90s: [VM1] [VM2] [VM3] [VM4] [VM5] [VM6] [VM7] │
│                                                  │
│  Scaling time: 30-60 seconds per VM              │
│  During scaling: requests may timeout/drop       │
└──────────────────────────────────────────────────┘

CONTAINER:
┌──────────────────────────────────────────────────┐
│ T=0:    [C1] [C2] [C3]                           │
│ T=1s:   [C1] [C2] [C3] [C4] [C5] [C6] [C7]       │
│ T=2s:   [C1] [C2] [C3] [C4] [C5] [C6] [C7] [C8]...[C20]│
│                                                  │
│  Scaling time: <1 second per container           │
│  Near-instant response to traffic                │
└──────────────────────────────────────────────────┘
```

Đánh giá cho PaaS:

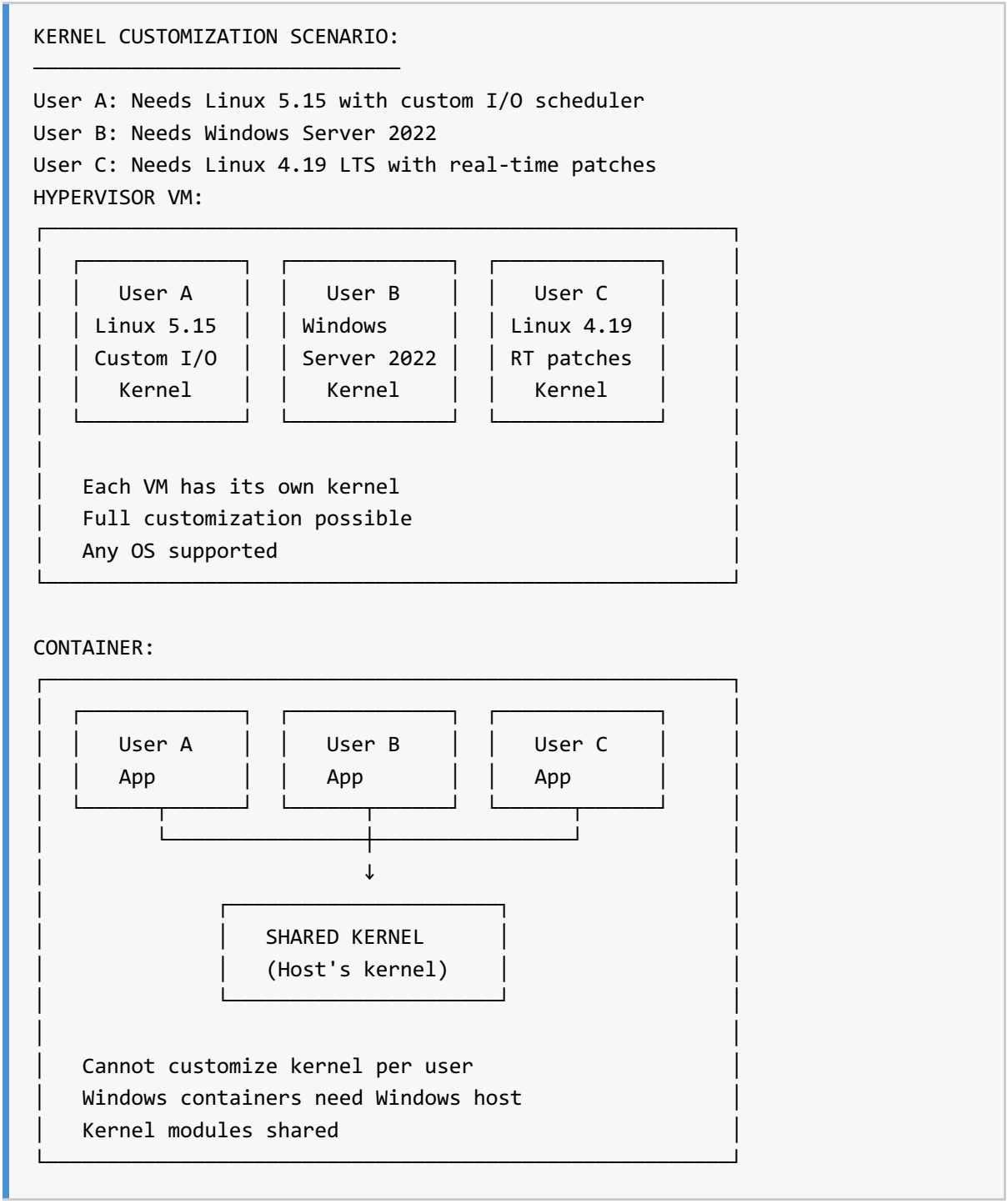
Tiêu chí	Hypervisor	Container	Winner
Scale speed			Container
Density			Container
Deployment			Container
Resource efficiency			Container
CI/CD integration			Container

Khuyến nghị: CONTAINER cho PaaS

7.2. IaaS cho phép người dùng tùy chỉnh kernel

Yêu cầu IaaS:

- Custom kernel (versions, modules, patches)
- □ Strong isolation giữa tenants
- Full OS control cho user
- Security compliance
- Multiple OS support (Linux, Windows)



Đánh giá cho IaaS (kernel customization):

Tiêu chí	Hypervisor	Container	Winner

Custom kernel			Hypervisor
Multi-OS			Hypervisor
Tenant isolation			Hypervisor
Kernel modules			Hypervisor
Compliance (SOC2, etc.)			Hypervisor

Khuyến nghị: HYPERVISOR VM cho IaaS với kernel customization

8. Trade-offs: Bảo mật, Hiệu năng, Tính di động

8.1. Bảo mật (Security)

```
SECURITY COMPARISON:
____________________


HYPERVISOR:
┌──────────────────────────────────────────────────┐
│ Attack Surface:                                  │
│ • Hypervisor + Guest OS                          │
│ • Hardware-level isolation (VT-x)                │
│ • Separate kernel per VM                         │
│                                                  │
│ Threats:                                         │
│ • VM escape (rare, critical)                     │
│ • Side-channel attacks (Spectre, Meltdown)       │
│                                                  │
│ Security Level:  (Industry standard)             │
└──────────────────────────────────────────────────┘

CONTAINER:
┌──────────────────────────────────────────────────┐
│ Attack Surface:                                  │
│ • Container runtime + Shared kernel              │
│ • Namespace/cgroup isolation (software)          │
│ • Shared kernel = shared vulnerabilities         │
│                                                  │
│ Threats:                                         │
│ • Container escape (more common than VM escape)  │
│ • Kernel exploits affect all containers          │
│ • Privileged containers = root on host           │
│                                                  │
│ Security Level:  (Improving with gVisor, Kata)   │
└──────────────────────────────────────────────────┘
```

Security Aspect	Hypervisor	Container
Isolation strength		
Kernel vulnerabilities	Isolated	Shared risk

Escape difficulty	Very hard	Easier
Multi-tenant trust	High	Medium
Compliance ready	Yes	Depends

8.2. Hiệu năng (Performance)

```
PERFORMANCE COMPARISON:
──────────────────────

                    Hypervisor      Container
CPU Performance:    95-98%          99-100%
Memory Overhead:    500MB-2GB/VM    10-50MB/container
Startup Time:       30-120 sec      0.1-1 sec
I/O Performance:    90-95%          98-100%
Network Latency:    +50-100µs       +10-20µs
Density:            10-50 VMs       100-1000 containers
```

Performance Aspect	Hypervisor	Container	Difference
CPU overhead	2-5%	<1%	Container +4%
Memory per instance	512MB+	10MB+	Container 50x better
Startup	30-120s	<1s	Container 100x faster
I/O throughput	90-95%	98%+	Container +5%

8.3. Tính di động (Portability)

```
PORTABILITY COMPARISON:
──────────────────────

HYPERVISOR:
┌──────────────────────────────────────────────────────┐
│ VM Image:                                            │
│ • Size: 10-100 GB                                    │
│ • Format: VMDK, VHD, QCOW2 (not standardized)        │
│ • Contains: Full OS + Apps                           │
│ • Transfer time: Minutes to hours                    │
│ • Cross-platform: Limited (hypervisor-specific)      │
│                                                      │
│ Migration: OVF format helps, but still complex       │
└──────────────────────────────────────────────────────┘

CONTAINER:
┌──────────────────────────────────────────────────────┐
│ Container Image:                                     │
│ • Size: 10-500 MB (layered)                          │
│ • Format: OCI standard (universal)                   │
```

```
│ • Contains: App + dependencies only                  │
│ • Transfer time: Seconds to minutes                  │
│ • Cross-platform: Any Linux host (mostly)            │
│                                                      │
│ Migration: docker push/pull anywhere                 │
└──────────────────────────────────────────────────────┘
```

Portability Aspect	Hypervisor	Container
Image size	10-100 GB	10-500 MB
Standardization	OVF/OVA	OCI
Registry ecosystem	Limited	Docker Hub, etc.
Build reproducibility	Harder	Dockerfile
Cross-cloud	Complex	Easy

9. Bảng tổng hợp khuyến nghị

Use Case	Khuyến nghị	Lý do
PaaS (fast scaling)	Container	Startup <1s, high density, CI/CD friendly
IaaS (custom kernel)	Hypervisor	Full OS control, strong isolation
Multi-tenant SaaS	Hypervisor hoặc Kata Containers	Security isolation critical
Microservices	Container	Lightweight, scalable
Legacy Windows apps	Hypervisor	Full Windows support
Development/Testing	Container	Fast, reproducible

10. Kết luận

```
┌──────────────────────────────────────────────────────────────────────┐
│                              SUMMARY                                 │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  BARE METAL vs HOSTED HYPERVISOR:                                    │
│  ├─ Bare Metal: Production, performance, enterprise                  │
│  └─ Hosted: Development, testing, desktop                            │
│                                                                      │
│  HYPERVISOR vs CONTAINER (Public Cloud):                             │
│  ├─ PaaS (fast scaling): → CONTAINER                                 │
│  │   • Startup <1s, high density, perfect for auto-scaling           │
│  │                                                                   │
│  └─ IaaS (custom kernel): → HYPERVISOR                               │
│      • Full kernel control, strong isolation, compliance             │
│                                                                      │
│  TRADE-OFFS:                                                         │
│  ├─ Security:    Hypervisor > Container                              │
│  ├─ Performance: Container > Hypervisor                              │
│  └─ Portability: Container > Hypervisor                              │
│                                                                      │
│  HYBRID APPROACH (Best practice):                                    │
```

```
│  ├─ Run containers inside VMs for security                          │
│  ├─ Example: GKE, EKS run on VMs underneath                         │
│  └─ Kata Containers: VM-level isolation + container UX              │
│                                                                     │
└─────────────────────────────────────────────────────────────────────┘
```