HỆ ĐIỀU HÀNH - CHEATSHEET HOÀN CHỈNH

Trần Minh Huy

1. Khái niệm Hệ Điều Hành

Định nghĩa: Phần mềm hệ thống điều khiển, quản lý phần cứng, cung cấp môi trường cho các chương trình hoạt động.

Chức năng chính:

- **Quản lý tiến trình:** Tạo/xóa process, đình chỉ/tiếp tục, đồng bộ, truyền thông, xử lý deadlock
- **Quản lý bộ nhớ:** Theo dõi, phân bổ/giải phóng, chuyển vào/ra bộ nhớ
- **Quản lý file:** Tạo/xóa file/thư mục, thao tác file, ánh xạ file, backup
- **Quản lý I/O:** Buffering, caching, spooling, device driver interfaces
- **Bảo mật:** Authentication, authorization, protection

Chế độ hoạt động:

- **User mode:** bit mode = 1, hạn chế truy cập
- **Kernel mode:** bit mode = 0, truy cập toàn bộ

Interrupts:

- **Hardware:** Từ thiết bị (keyboard, timer, network)
- **Software:** Trap/Exception (zero division, invalid memory)
- **Interrupt vector:** Bảng địa chỉ các ISR

2. Cấu trúc Hệ Điều Hành

Kiến trúc	Mô tả
Monolithic	Toàn bộ chức năng trong một khối. VD: Unix, Linux
Microkernel	Chỉ giữ chức năng cơ bản (IPC, memory, scheduling). VD: MINIX
Layered	Chia theo lớp, dễ kiểm thử, nhưng hiệu suất kém
Module-based	Mô-đun nạp/rút động. VD: Linux kernel modules
Hybrid	Kết hợp monolithic và microkernel. VD: Windows NT

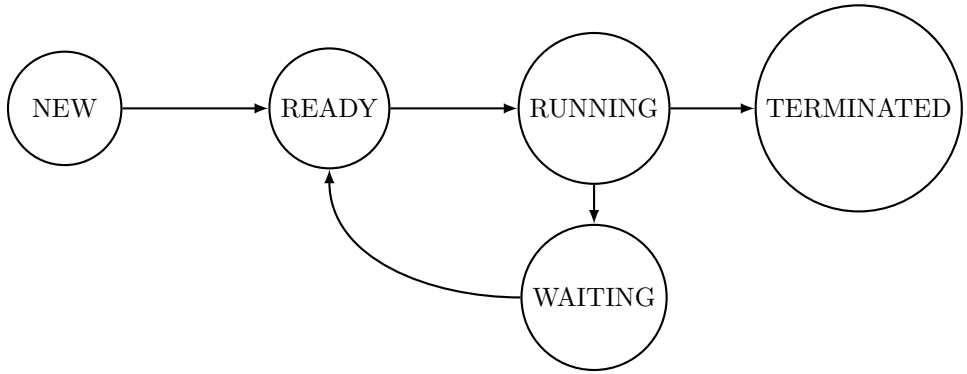
System Calls (6 loại):

1. **Process control:** `fork()`, `exit()`, `wait()`, `exec()`
2. **File management:** `open()`, `read()`, `write()`, `close()`
3. **Device management:** `ioctl()`, `read()`, `write()`
4. **Information:** `getpid()`, `alarm()`, `sleep()`
5. **Communications:** `pipe()`, `shmget()`, `mmap()`
6. **Protection:** `chmod()`, `chown()`, `umask()`

3. Tiến trình (Process)

Định nghĩa: Chương trình đang thực thi với không gian địa chỉ riêng.

Process States:



Process Control Block (PCB):

- Process state
- Program counter
- CPU registers
- CPU scheduling info
- Memory management info
- Accounting info
- I/O status info

Memory Layout:

```
+------------------+
|      Stack       | <- Local variables
+------------------+
|                    |
|                  |
|                    |
+------------------+
|       Heap       | <- Dynamic allocation
+------------------+
|       Data       | <- Global variables
+------------------+
|       Text       | <- Code
+------------------+
```

4. THUẬT TOÁN ĐỊNH THỜI CPU

Scheduling Criteria:

- **CPU utilization:** Maximize
- **Throughput:** Maximize
- **Turnaround time:** Minimize
- **Waiting time:** Minimize
- **Response time:** Minimize

First-Come First-Served (FCFS)

Cách hoạt động:

- Process đến trước được phục vụ trước
- Non-preemptive (không ngắt)
- Dùng queue FIFO

Ưu điểm: Đơn giản, công bằng theo thứ tự đến

Nhược điểm: Convoy effect - process ngắn chờ process dài

Ví dụ chi tiết:

```
Process  Arrival  Burst  Start  Finish  Wait
P1       0        24     0      24      0
P2       0        3      24     27      24
P3       0        3      27     30      27

Gantt Chart: |----P1----|P2|P3|
             0          24 27 30

Average Waiting = (0+24+27)/3 = 17
Average Turnaround = (24+27+30)/3 = 27
```

Shortest Job First (SJF)

Cách hoạt động:

- Chọn process có burst time nhỏ nhất
- Non-preemptive hoặc preemptive (SRTF)
- Optimal cho average waiting time

Ví dụ SJF (Non-preemptive):

```
Process  Arrival  Burst  Start  Finish  Wait  Turnaround
P1       0        6      0      6       0     6
P2       2        8      14     22      12    20
P3       4        7      6      13      2     9
P4       6        3      3      9       0     3

Gantt Chart: |--P1--|--P4--|---P3---|----P2----|
             0      6      9        16         24

Average Waiting = (0+0+2+12)/4 = 3.5
Average Turnaround = (6+3+9+20)/4 = 9.5
```

Ví dụ SRTF (Preemptive SJF):

```
Process  Arrival  Burst  Completion  Wait  Turnaround
P1       0        8      17          9     17
P2       1        4      5           0     4
P3       2        9      26          15    24
P4       3        5      10          2     7

Gantt Chart: |P1|P2|--P2--|P4|--P4--|--P1--|---P3---|
             0  1  2      5  7     10     17      26

Average Waiting = (9+0+15+2)/4 = 6.5
Average Turnaround = (17+4+24+7)/4 = 13
```

Dự đoán burst time: $\tau_{n+1} = \alpha \cdot t_n + (1-\alpha) \cdot \tau_n$ Với: t_n = burst time thực, τ_n = dự đoán, α = weight (0-1)

Ví dụ dự đoán với $\alpha = 0.5$:

```
Burst  Actual  Predicted
1      6       10        // Initial guess
2      4       8         // 0.5*6 + 0.5*10 = 8
3      6       6         // 0.5*4 + 0.5*8 = 6
4      4       6         // 0.5*6 + 0.5*6 = 6
5      5       5         // 0.5*4 + 0.5*6 = 5
```

Ưu điểm:

- Tối ưu average waiting time
- Throughput cao

Nhược điểm:

- Khó xác định burst time
- Starvation cho process dài
- Overhead của preemption (SRTF)

Round Robin (RR)

Cách hoạt động:

- Mỗi process được cấp time quantum (q)
- Sau q đơn vị thời gian, process bị ngắt
- Process ngắt được đưa vào cuối ready queue

Chọn quantum:

- q lớn → FCFS
- q nhỏ → overhead context switch cao
- Rule of thumb: 80% process nên kết thúc trong q

Ví dụ chi tiết (q=4):

```
Process  Arrival  Burst  Timeline
P1       0        24     0-4,12-16,20-24,25-29,33-37
P2       0        3      4-7
P3       0        3      7-10

Gantt: |P1|P2|P3|P1|P1|P1|P1|P1|
       0  4  7 10 14 18 22 26 30
```

5. THUẬT TOÁN ĐỒNG BỘ

Critical Section Problem (3 điều kiện):

1. **Mutual Exclusion:** Chỉ 1 process trong CS
2. **Progress:** Process ngoài CS không block
3. **Bounded Waiting:** Giới hạn thời gian chờ

Peterson's Algorithm

Giải quyết critical section cho 2 process:

```
// Bien chia se
int turn;
boolean flag[2];

// Process Pi (i=0 hoac 1, j=1-i)
do {
    flag[i] = true;          // Toi muon vao CS
    turn = j;                // Nhuong quyen uu tien
    while (flag[j] && turn == j); // Doi

    // CRITICAL SECTION

    flag[i] = false;         // Toi ra khoi CS

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

Cách hoạt động:

- **flag[i] = true:** Process i thông báo muốn vào CS
- **turn = j:** Process i nhường quyền ưu tiên cho process j
- **while loop:** Kiểm tra xem process j có muốn vào CS không
- Nếu flag[j]=true và turn=j: process i phải đợi
- Khi process j ra khỏi CS hoặc từ bỏ quyền ưu tiên, process i được vào
- **flag[i] = false:** Process i thông báo đã rời khỏi CS

Tính chất:

- Đảm bảo mutual exclusion
- Đảm bảo progress
- Đảm bảo bounded waiting
- Chỉ dùng cho 2 processes

Semaphores

Định nghĩa:

```
// Cac phep toan semaphore co ban
wait(S) {
    while (S <= 0); // busy wait
    S--;
}

signal(S) {
    S++;
}
```

Cách hoạt động cơ bản:

- **wait(S):** Giảm giá trị semaphore
- Nếu S $\leq$ 0: process phải đợi (busy wait)
- **signal(S):** Tăng giá trị semaphore
- Cho phép 1 process đang chờ tiếp tục

Implementation không busy wait:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;

wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Cách hoạt động (không busy wait):

- **wait():**
  - Giảm value trước
  - Nếu value $<$ 0: có process đang chờ
  - Thêm process hiện tại vào hàng đợi
  - Block process hiện tại
- **signal():**
  - Tăng value
  - Nếu value $\leq$ 0: có process trong hàng đợi
  - Lấy 1 process từ hàng đợi
  - Đánh thức process đó
- **Ý nghĩa value:**
  - value $>$ 0: số lượng tài nguyên còn trống
  - value $\leq$ 0: số process đang chờ

Producer-Consumer (Bounded Buffer)

```
// Bien chia se
semaphore mutex = 1;  // Bao ve buffer
semaphore empty = n;  // So slot trong
semaphore full = 0;   // So slot day

// Producer
do {
    produce an item;

    wait(empty);      // Giam slot trong
    wait(mutex);      // Vao critical section
    add item to buffer;
    signal(mutex);    // Ra critical section
    signal(full);     // Tang slot day
```

```
} while (true);

// Consumer
do {
    wait(full);        // Giam slot day
    wait(mutex);       // Vao critical section
    remove item from buffer;
    signal(mutex);     // Ra critical section
    signal(empty);     // Tang slot trong

    consume the item;
} while (true);
```

Cách hoạt động:

- **3 semaphores:**
  - **mutex:** Bảo vệ truy cập buffer (mutual exclusion)
  - **empty:** Đếm số slot trống trong buffer
  - **full:** Đếm số slot đã có dữ liệu
- **Producer:**
  1. wait(empty): Chờ có slot trống
  2. wait(mutex): Lấy quyền truy cập buffer
  3. Thêm item vào buffer
  4. signal(mutex): Trả lại quyền truy cập
  5. signal(full): Tăng số slot đầy
- **Consumer:**
  1. wait(full): Chờ có dữ liệu
  2. wait(mutex): Lấy quyền truy cập buffer
  3. Lấy item từ buffer
  4. signal(mutex): Trả lại quyền truy cập
  5. signal(empty): Tăng số slot trống
- **Tránh deadlock:** wait empty/full trước, wait mutex sau

6. Deadlock

4 điều kiện gây deadlock:

1. **Mutual Exclusion:** Tài nguyên không chia sẻ được
2. **Hold and Wait:** Giữ và chờ thêm tài nguyên
3. **No Preemption:** Không thu hồi được
4. **Circular Wait:** Chờ đợi vòng tròn

Xử lý Deadlock:

- **Prevention:** Ngăn 1 trong 4 điều kiện
- **Avoidance:** Safe state, Banker's algorithm
- **Detection:** Wait-for graph, detection algorithm
- **Recovery:** Abort process, preempt resources

7. BANKER'S ALGORITHM

Data Structures:

```
Available[m]: Ta nguyen co san
Max[n][m]: Nhu cau toi da
Allocation[n][m]: Da cap phat
Need[n][m] = Max - Allocation
```

Ý nghĩa:

- **Available:** Số lượng tài nguyên mỗi loại còn sẵn sàng
- **Max:** Nhu cầu tối đa của mỗi process cho mỗi loại tài nguyên
- **Allocation:** Số tài nguyên mỗi loại đã cấp cho mỗi process
- **Need:** Số tài nguyên mỗi loại còn cần để hoàn thành

Safety Algorithm:

```
1. Work = Available
   Finish[i] = false for all i

2. Find i such that:
   Finish[i] == false AND Need[i] <= Work
   If no such i exists, go to step 4

3. Work = Work + Allocation[i]
   Finish[i] = true
   Go to step 2

4. If Finish[i] == true for all i:
      System is SAFE
   Else:
      System is UNSAFE
```

Cách hoạt động Safety Algorithm:

1. Khởi tạo Work = tài nguyên có sẵn
2. Tìm process chưa hoàn thành và có thể thỏa mãn nhu cầu
3. Giả sử process đó hoàn thành và trả lại tài nguyên
4. Lặp lại cho đến khi không tìm được process nào
5. Nếu tất cả process hoàn thành: SAFE, ngược lại: UNSAFE

Resource Request Algorithm:

```
1. If Request[i] <= Need[i], go to step 2
   Else raise error

2. If Request[i] <= Available, go to step 3
   Else Pi must wait

3. Pretend to allocate:
   Available = Available - Request[i]
   Allocation[i] = Allocation[i] + Request[i]
   Need[i] = Need[i] - Request[i]

4. Run safety algorithm
   If SAFE: allocate resources
   If UNSAFE: restore old state, Pi must wait
```

Cách hoạt động Request Algorithm:

1. Kiểm tra request hợp lệ (không vượt quá Need)

2. Kiểm tra có đủ tài nguyên sẵn sàng không
3. Giả định cấp phát tài nguyên (thay đổi tạm thời)
4. Chạy Safety Algorithm để kiểm tra
5. Nếu SAFE: cấp phát thật; nếu UNSAFE: hoàn tác và chờ

8. QUẢN LÝ BỘ NHỚ

Address Binding:

- **Compile time:** Absolute code
- **Load time:** Relocatable code
- **Execution time:** Dynamic binding

Swapping

Định nghĩa: Di chuyển process giữa main memory và disk

Cách hoạt động:

- **Swap out:** Di chuyển process từ memory ra backing store (disk)
- **Swap in:** Di chuyển process từ backing store vào memory
- Thường xảy ra khi memory không đủ cho tất cả process
- Priority-based: swap out process có priority thấp

Backing Store:

- Vùng disk đủ lớn để chứa memory images của tất cả process
- Phải cung cấp direct access
- Thường là một phân vùng riêng (swap partition)

Swap Time:

- Transfer time = kích thước process / tốc độ transfer disk
- Ví dụ: Process 100MB, disk 50MB/s → 2 giây
- Thời gian swap = seek time + transfer time
- Swap time là yếu tố chính ảnh hưởng hiệu năng

Các vấn đề với Swapping:

- **Performance:** Swapping rất chậm
- **Pending I/O:** Không swap process đang chờ I/O
- **Modified swapping:** Chỉ swap một phần process (demand paging)

Memory Allocation Algorithms

1. First Fit:

```
// Tim hole DAU TIEN du lon
for each hole in memory:
    if (hole.size >= process.size):
        allocate(hole, process)
        break
```

Cách hoạt động:

- Duyệt các hole theo thứ tự địa chỉ
- Chọn hole đầu tiên có kích thước đủ lớn
- Cấp phát và tạo hole mới nếu còn dư
- **Ưu điểm:** Nhanh, không cần duyệt hết danh sách
- **Nhược điểm:** Tạo nhiều hole nhỏ ở đầu bộ nhớ

2. Best Fit:

```
// Tim hole NHO NHAT vua du
best_hole = null
min_leftover = INFINITY

for each hole in memory:
    if (hole.size >= process.size):
        leftover = hole.size - process.size
        if (leftover < min_leftover):
            min_leftover = leftover
            best_hole = hole
```

Cách hoạt động:

- Duyệt toàn bộ danh sách hole
- Chọn hole nhỏ nhất mà vẫn đủ lớn
- Mục tiêu: giảm thiểu phần dư thừa
- **Ưu điểm:** Tiết kiệm bộ nhớ
- **Nhược điểm:** Chậm, tạo nhiều hole rất nhỏ

3. Worst Fit:

```
// Tim hole LON NHAT
max_hole = null
max_size = 0

for each hole in memory:
    if (hole.size >= process.size &&
        hole.size > max_size):
        max_size = hole.size
        max_hole = hole
```

Cách hoạt động:

- Duyệt toàn bộ danh sách hole
- Chọn hole lớn nhất
- Mục tiêu: tạo hole dư lớn có thể dùng tiếp
- **Ưu điểm:** Hole dư thường đủ lớn để dùng
- **Nhược điểm:** Chậm, lãng phí bộ nhớ lớn

Tiêu chí	First-Fit	Best-Fit	Worst-Fit
Nguyên lý	Cấp phát khối trống đầu tiên vừa đủ	Cấp phát khối trống nhỏ nhất đủ dùng	Cấp phát khối trống lớn nhất
Tốc độ tìm kiếm	Nhanh (tìm từ đầu)	Chậm (phải tìm toàn bộ)	Chậm (phải tìm toàn bộ)
Hiệu quả sử dụng bộ nhớ	Trung bình	Tốt, ít lãng phí	Thường kém, gây phân mảnh lớn
Khả năng phân mảnh ngoài	Trung bình	Cao (nhiều mảnh nhỏ dư ra)	Thấp hơn Best-Fit

Paging

Address Translation:

```
Logical Address = Page Number + Page Offset
Physical Address = Frame Number + Page Offset

Given: logical address, page size
Page number = logical address / page size
Page offset = logical address % page size
Frame number = page_table[page number]
Physical address = frame number * page size + offset
```

Cách hoạt động:

- Chia logical address thành 2 phần: page number và offset
- Page number dùng để tra page table
- Page table trả về frame number
- Frame number kết hợp với offset tạo physical address
- Offset không thay đổi trong quá trình translation

Tiêu chí	**External Fragmentation**	**Internal Fragmentation**
Vị trí phân mảnh	Bên ngoài khối cấp phát	Bên trong khối cấp phát
Nguyên nhân	Khối trống không liền kề	Cấp phát thừa so với nhu cầu
Ảnh hưởng	Không cấp phát được cho tiến trình lớn	Lãng phí bộ nhớ bên trong khối
Giải pháp	Nén bộ nhớ (compaction)	Cấp phát chính xác, chia nhỏ khối
Thường gặp trong	Cấp phát động (First/Best/Worst-Fit)	Cấp phát cố định, paging

TLB (Translation Lookaside Buffer):

$$\text{EAT} = \alpha \times (t_{TLB} + t_{mem}) + (1 - \alpha) \times (t_{TLB} + 2t_{mem})$$

Cách hoạt động TLB:

- TLB là cache cho page table entries
- Khi cần translation:
  1. Kiểm tra TLB trước (nhanh)
  2. Nếu hit: lấy frame number từ TLB
  3. Nếu miss: truy cập page table trong memory
  4. Cập nhật TLB với entry mới

- **EAT formula:**
  - α: TLB hit ratio
  - t_{TLB}: TLB access time
  - t_{mem}: Memory access time
  - Hit: 1 TLB + 1 memory access
  - Miss: 1 TLB + 2 memory access (page table + data)

9. VIRTUAL MEMORY

Demand Paging: Load page khi cần

Effective Access Time:

$$\text{EAT} = (1 - p) \times ma + p \times \text{page\_fault\_time}$$

Mối quan hệ với Swapping:

- Virtual memory là dạng swapping ở mức page
- Chỉ swap những page cần thiết (demand paging)
- Hiệu quả hơn full process swapping
- Cho phép process lớn hơn physical memory

Page Replacement Algorithms

1. FIFO:

```
// Cai dat hang doi
on page_fault(page):
    if queue.is_full():
        victim = queue.dequeue()  // Xoa trang cu nhat
        free_frame(victim)
    queue.enqueue(page)           // Them trang moi nhat
    load_page(page)
```

Cách hoạt động:

- Dùng queue FIFO để quản lý thứ tự page
- Page vào trước sẽ bị thay thế trước
- Khi page fault và không còn frame trống:
  1. Lấy page cũ nhất từ đầu queue
  2. Giải phóng frame của page đó
  3. Load page mới vào frame
  4. Thêm page mới vào cuối queue
- **Vấn đề:** Belady's anomaly - tăng frame có thể tăng page fault

2. LRU (Least Recently Used):

```
// Cai dat voi counter
page_table[page].counter = current_time

on page_fault(page):
    if no_free_frame():
        victim = find_LRU_page()
        free_frame(victim)
    load_page(page)
```

Cách hoạt động:

- Ghi nhận thời gian sử dụng cuối của mỗi page
- Khi cần thay thế: chọn page lâu nhất chưa dùng
- Implementation:
  1. Counter: mỗi page có counter lưu thời gian
  2. Stack: page được dùng đẩy lên top
  3. Matrix: ma trận reference bit
- **Ưu điểm:** Không có Belady's anomaly
- **Nhược điểm:** Overhead cao

3. Clock (Second Chance):

```
// Danh sach vong voi reference bit
clock_hand = 0

on page_fault(page):
    while (true):
        if (ref_bit[clock_hand] == 0):
            // Tim thay trang bi loai
            replace page
            break
        else:
            // Cho co hoi thu hai
            ref_bit[clock_hand] = 0
            clock_hand = (clock_hand + 1) % num_frames
```

Cách hoạt động:

- Sắp xếp các frame thành vòng tròn
- Mỗi frame có reference bit (0 hoặc 1)
- Clock hand quay quanh vòng tròn:
  1. Nếu ref_bit = 0: thay thế page này
  2. Nếu ref_bit = 1: cho cơ hội thứ 2
  3. Set ref_bit = 0 và di chuyển tiếp
- Page được truy cập: set ref_bit = 1
- **Ưu điểm:** Hiệu quả gần bằng LRU, chi phí thấp hơn

Working Set Model:

$$WS(t, \Delta) = \{\text{pages referenced in } [t - \Delta, t]\}$$

Thrashing: $\sum$ (working set sizes) $>$ physical memory

10. File System

File Attributes: Name, type, location, size, protection, time

Directory Structure:

- Single-level
- Two-level
- Tree-structured
- Acyclic graph

Allocation Methods:

Method	Pros	Cons
Contiguous	Fast	External fragmentation
Linked	No fragmentation	Slow random access
Indexed	Fast access	Index overhead

10. File System

File Attributes: Name, type, location, size, protection, time

Directory Structure:

- Single-level
- Two-level
- Tree-structured
- Acyclic graph

Allocation Methods:

Method	Pros	Cons
Contiguous	Fast	External fragmentation
Linked	No fragmentation	Slow random access
Indexed	Fast access	Index overhead

11. I/O Systems

I/O Techniques:

- **Polling:** CPU checks device status
- **Interrupt-driven:** Device interrupts CPU
- **DMA:** Direct Memory Access

Chi tiết các kỹ thuật I/O:

1. Polling (Programmed I/O):

- CPU liên tục kiểm tra trạng thái thiết bị
- Vòng lặp: while(device_not_ready) {}
- CPU thực hiện mọi transfer dữ liệu
- **Ưu điểm:** Đơn giản, dễ implement
- **Nhược điểm:** Lãng phí CPU (busy waiting)

2. Interrupt-Driven I/O:

- CPU khởi tạo I/O rồi làm việc khác
- Thiết bị gửi interrupt khi sẵn sàng
- CPU xử lý interrupt, transfer dữ liệu
- **Ưu điểm:** CPU không phải chờ
- **Nhược điểm:** Interrupt overhead cho mỗi byte

3. DMA (Direct Memory Access):

- DMA controller điều khiển transfer
- CPU chỉ khởi tạo và nhận thông báo khi xong
- Data transfer trực tiếp giữa device và memory
- **Ưu điểm:** Giải phóng CPU, transfer nhanh
- **Nhược điểm:** Cần hardware đặc biệt

Disk Scheduling:

Algorithm	Description
FCFS	First-Come First-Served
SSTF	Shortest Seek Time First
SCAN	Elevator algorithm
C-SCAN	Circular SCAN

Chi tiết Disk Scheduling:

1. FCFS (First-Come First-Served):

- Xử lý requests theo thứ tự đến
- Đơn giản, công bằng
- Không tối ưu seek time
- Wild swings có thể xảy ra

2. SSTF (Shortest Seek Time First):

- Chọn request gần vị trí hiện tại nhất
- Giảm thiểu seek time trung bình
- Có thể gây starvation cho requests xa
- Không công bằng

3. SCAN (Elevator):

- Đầu đọc di chuyển theo một hướng
- Đến cuối đĩa thì đổi hướng
- Phục vụ requests trên đường đi
- Công bằng hơn SSTF
- Uniform wait time

4. C-SCAN (Circular SCAN):

- Chỉ phục vụ theo một hướng
- Khi đến cuối, quay lại đầu mà không phục vụ
- Đảm bảo wait time đồng đều hơn
- Treats cylinders như circular list

Ví dụ: Disk queue = 98, 183, 37, 122, 14, 124, 65, 67

Head starts at 53:

- FCFS: 53→98→183→37→122→14→124→65→67 (640 cylinders)
- SSTF: 53→65→67→37→14→98→122→124→183 (236 cylinders)
- SCAN: 53→37→14→0→65→67→98→122→124→183 (236 cylinders)

12. Security & Protection

Protection: Internal access control

Security: External threat protection

Chi tiết:

- **Protection:** Cơ chế kiểm soát truy cập tài nguyên trong hệ thống
- **Security:** Bảo vệ hệ thống khỏi các mối đe dọa bên ngoài

Access Control:

- **Access Matrix:** Objects × Subjects
- **ACL:** List per object
- **Capability List:** List per subject

Chi tiết Access Control:

1. Access Matrix:

- Ma trận 2 chiều: Subjects (users/processes) × Objects (files/resources)
- Mỗi ô chứa quyền truy cập (read, write, execute)
- Ví dụ:

```
          File1  File2  Printer
User1     R,W    R      -
User2     R      R,W    Print
Process1  R,X    -      -
```

- **Vấn đề:** Ma trận thường rất thưa, lãng phí bộ nhớ

2. Access Control List (ACL):

- Mỗi object có danh sách các subjects và quyền
- Lưu theo cột của Access Matrix
- Ví dụ cho File1: [(User1, R,W), (User2, R), (Process1, R,X)]
- **Ưu điểm:** Dễ xác định ai có thể truy cập object
- **Nhược điểm:** Khó xác định user có thể truy cập gì

3. Capability List:

- Mỗi subject có danh sách các objects và quyền
- Lưu theo hàng của Access Matrix
- Ví dụ cho User1: [(File1, R,W), (File2, R)]
- **Ưu điểm:** Dễ xác định subject có thể truy cập gì
- **Nhược điểm:** Khó thu hồi quyền truy cập

Authentication Methods:

- Something you know (password)
- Something you have (token)
- Something you are (biometric)

Chi tiết Authentication:

1. Something you know:

- Password, PIN, security questions
- **Ưu điểm:** Đơn giản, không cần thiết bị
- **Nhược điểm:** Có thể quên, bị đoán, bị đánh cắp
- **Best practices:** Hash + salt, password policies

2. Something you have:

- Smart card, token device, phone
- **Ưu điểm:** Khó giả mạo, có thể thu hồi
- **Nhược điểm:** Có thể mất, bị đánh cắp
- **Ví dụ:** RSA token, Google Authenticator

3. Something you are:

- Fingerprint, face, retina, voice
- **Ưu điểm:** Không thể quên hoặc mất
- **Nhược điểm:** Đắt tiền, false positives/negatives
- **Issues:** Privacy concerns, không thể thay đổi

Two-Factor Authentication (2FA):

- Kết hợp 2 trong 3 phương pháp trên
- Tăng cường bảo mật đáng kể
- Ví dụ: Password + SMS code

13. Important Formulas

Formula	Description
CPU Utilization $= 1 - p^n$	p: I/O time, n: processes
Throughput $= \frac{\text{processes}}{\text{time}}$	Processes completed
Turnaround = completion − arrival	Total time
Waiting = turnaround − burst	Time in ready queue
Response = first run − arrival	Time to first response
TLB Reach = entries × page size	TLB coverage
Speedup $= \frac{1}{S+\frac{1-S}{N}}$	Amdahl's Law