Các thuật toán lập lịch như: FCFS, SJF, Round Robin, Priority,... sẽ được học kỹ hơn trong **Chương CPU Scheduling**.

## ISSUES IN MULTITHREADING ( vấn đề trong đa luồng )

**Race Condition (Điều kiện tranh chấp)**

* Occurs when two or more threads access shared data concurrently and try to modify it at the same time.
* Xảy ra khi hai luồng cùng truy cập (đọc/ghi) dữ liệu chung **đồng thời**, và kết quả phụ thuộc **thứ tự thực thi** → dễ bị sai lệch dữ liệu.
* Giải pháp: dùng **mutex (mutual exclusion)** – khóa vùng truy cập chung lại

**Deadlock (Bế tắc)**

* Hai luồng chờ nhau mãi mãi, không ai nhả tài nguyên
* Cần 4 điều kiện để xảy ra (sẽ học kỹ ở chương **Deadlock**): Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait

**Livelock**

* Các luồng vẫn hoạt động, nhưng liên tục nhường nhau, không ai thực sự làm được việc

**Starvation**

* Một luồng không bao giờ được CPU cấp phát tài nguyên do các luồng ưu tiên cao hơn chiếm liên tục

## Scheduling ( Lập lịch cpu )

### CÁC CHỈ SỐ ĐÁNH GIÁ THUẬT TOÁN LẬP LỊCH

| **Thuật ngữ** | **Ý nghĩa** |
| --- | --- |
| **Turnaround Time** | Tổng thời gian tiến trình tồn tại trong hệ thống |
| **Waiting Time** | Tổng thời gian tiến trình chờ trong hàng đợi |
| **Response Time** | Thời gian từ khi nộp tiến trình đến khi nó bắt đầu chạy |
| **Throughput** | Số tiến trình hoàn thành trong một đơn vị thời gian |
| **CPU Utilization** | % thời gian CPU đang bận xử lý (không bị rảnh) |

### CÁC THUẬT TOÁN LẬP LỊCH CPU PHỔ BIẾN

| **STT** | **Thuật toán** | **Viết tắt** | **Loại** | **Preemptive?** |
| --- | --- | --- | --- | --- |
| 1 | First-Come First-Served | FCFS | Cơ bản | ❌ Không |
| 2 | Shortest Job First | SJF | Tối ưu chờ | ✅ Có bản preemptive (SRTF) |
| 3 | Round Robin | RR | Công bằng | ✅ Có |
| 4 | Priority Scheduling | Priority | Linh hoạt | ✅/❌ Cả hai |
| 5 | Multilevel Queue | MQ | Hệ thống phân lớp | ✅/❌ Tùy lớp |

#### 1. FCFS – First Come, First Served

* Xử lý tiến trình theo **thứ tự đến trước**
* **Non-preemptive**: không bị ngắt giữa chừng

FCFS is a **non-preemptive** scheduling algorithm that serves processes in the **order of their arrival**.

| **Ưu điểm (Advantages)** | **Nhược điểm (Disadvantages)** |
| --- | --- |
| Dễ hiểu, đơn giản để cài đặt | **Convoy effect**: tiến trình ngắn phải chờ tiến trình dài |
| Công bằng về mặt "ai đến trước, xử lý trước" | Thời gian chờ trung bình cao nếu tiến trình đầu có burst dài |
| Không gây starvation nếu hệ thống có đủ tài nguyên | Không tối ưu tài nguyên CPU và thiết bị ngoại vi |
| Ít context switch (không bị ngắt giữa chừng) → **overhead thấp** | Không thích hợp cho hệ thống đa chương trình cần phản hồi nhanh |

#### 2. SJF – Shortest Job First

Chọn tiến trình có **CPU burst time nhỏ nhất**

Có 2 dạng:

* **Non-preemptive** (SJF) ( chạy đến hết)
* **Preemptive** (Shortest Remaining Time First – SRTF)

| **Ưu điểm (Advantages)** | **Nhược điểm (Disadvantages)** |
| --- | --- |
| Tối ưu hóa **waiting time trung bình** (có bằng chứng toán học) | Dễ gây **starvation** nếu luôn có tiến trình ngắn hơn mới đến |
| Ưu tiên tiến trình ngắn → phản hồi nhanh hơn | Khó ước lượng trước chính xác CPU burst time |
| Thích hợp cho batch systems không có ràng buộc thời gian thực | Không dùng được nếu không biết trước burst time |

#### 3. Round Robin (RR)

Cấp CPU theo **vòng quay**, mỗi lần cấp **1 lượng thời gian (time quantum)**

Mỗi tiến trình được cấp CPU trong time quantum cố định (ví dụ: 4ms)

Preemptive: nếu chưa chạy xong trong quantum → bị ngắt và đưa về cuối hàng **đợi**

| **Ưu điểm (Advantages)** | **Nhược điểm (Disadvantages)** |
| --- | --- |
| Công bằng – mọi tiến trình đều có cơ hội chạy | Nếu time quantum nhỏ → context switch nhiều, giảm hiệu năng |
| Không có starvation | Nếu time quantum quá lớn → giống FCFS → mất ý nghĩa “chia đều CPU” |
| Phù hợp cho hệ thống interactive (người dùng) | Waiting time cao nếu nhiều tiến trình nhỏ, ngắt nhau liên tục |

#### 4. Priority Scheduling

Mỗi tiến trình có độ ưu tiên (priority number)

CPU cấp cho tiến trình có độ ưu tiên cao nhất

Có thể:

* Preemptive: khi có tiến trình ưu tiên cao hơn đến
* Non-preemptive: đang chạy thì không bị ngắt

| **Ưu điểm (Advantages)** | **Nhược điểm (Disadvantages)** |
| --- | --- |
| Cho phép xác định **mức độ quan trọng** của tiến trình | Dễ xảy ra **starvation** nếu các tiến trình ưu tiên thấp không được chạy |
| Phù hợp với hệ thống real-time cần xử lý ưu tiên trước | Cần cơ chế “aging” để tăng dần độ ưu tiên tránh starvation |

*\*Cơ chế aging : tăng độ ưu tiên theo thời gian chờ 🡪 tránh starvation*

#### 5. Multilevel Queue Scheduling

Chia tiến trình theo **nhóm loại**:

* foreground (người dùng)
* background (batch, system...)
* System → FCFS
* Interactive → RR
* Batch → SJF

Mỗi hàng có thuật toán riêng

Cố định vị trí tiến trình trong hàng (no movement)

| **Ưu điểm (Advantages)** | **Nhược điểm (Disadvantages)** |
| --- | --- |
| Phân loại tiến trình rõ ràng, dễ điều chỉnh hệ thống | **Không linh hoạt**: tiến trình không thể đổi nhóm (ví dụ từ batch sang user) |
| Tối ưu mỗi nhóm với thuật toán phù hợp | Nhóm thấp nhất dễ **starvation** nếu nhóm cao luôn chiếm CPU |

### CÁCH TÍNH TOÁN CÁC CHỈ SỐ HIỆU SUẤT

| **Chỉ số** | **Công thức** | **Ý nghĩa** |
| --- | --- | --- |
| Turnaround Time | Completion Time – Arrival Time | Tổng thời gian tiến trình tồn tại |
| Waiting Time | Turnaround Time – Burst Time | Tổng thời gian chờ không làm gì |
| Response Time | First Response Time – Arrival Time | Thời gian từ khi vào hàng đợi đến khi bắt đầu chạy |

### TỔNG HỢP TẤT CẢ THUẬT TOÁN

| **Thuật toán** | **Preemptive?** | **Công bằng** | **Tối ưu chờ?** | **Starvation?** | **Phù hợp với** |
| --- | --- | --- | --- | --- | --- |
| FCFS | ❌ | ❌ | ❌ | ❌ | Batch |
| SJF | ✅/❌ | ❌ | ✅ (best) | ✅ | Short tasks |
| RR | ✅ | ✅ | ❌ | ❌ | Interactive |
| Priority | ✅/❌ | ❌ | Có thể tốt | ✅ | Real-time |
| MQ | ✅/❌ | Có thể | Có thể | Có thể | System-level |

### Các khái niệm và công thức

| **Tên chỉ số** | **Ký hiệu thường dùng** | **Ý nghĩa đơn giản** |
| --- | --- | --- |
| Arrival Time | AT | Thời điểm tiến trình "đến hàng đợi" |
| Burst Time | BT | Thời gian CPU cần để hoàn thành tiến trình |
| Completion Time | CT | Thời điểm tiến trình kết thúc |
| Turnaround Time | TAT = CT – AT | Tổng thời gian tiến trình tồn tại trong hệ thống |
| Waiting Time | WT = TAT – BT | Thời gian tiến trình ngồi chờ |
| Response Time | RT = Start – AT | Thời gian từ lúc đến cho đến khi được chạy lần đầu tiên |

### Ví dụ về bài tập các thuật toán

#### 1. FCFS = First Come First Serve

Ví dụ đề bài

| **Process** | **Arrival Time (AT)** | **Burst Time (BT)** |
| --- | --- | --- |
| P1 | 0 | 5 |
| P2 | 1 | 3 |
| P3 | 6 | 4 |

***Bước 1 : Tính Start Time***

P1:

* Đến lúc 0, không ai trước nó → Start = 0
* Completion = 0 + 5 = 5

P2:

* Đến lúc 1
* CPU rảnh lúc 5  
  → Start = max(5, 1) = 5  
  → Completion = 5 + 3 = 8

P3:

* Đến lúc 6
* CPU rảnh lúc 8  
  → Start = max(8, 6) = 8  
  → Completion = 8 + 4 = 12

***Bước 2 : Sắp xếp theo Arrival Time***

***Bước 3 : Tính Completion Time (CT)***

| **Process** | **CT** | **AT** | **TAT = CT – AT** |
| --- | --- | --- | --- |
| P1 | 5 | 0 | 5 |
| P2 | 8 | 1 | 7 |
| P3 | 16 | 2 | 14 |

***Bước 4 : Tính Turnaround Time (TAT)***

| **Process** | **CT** | **AT** | **TAT = CT – AT** |
| --- | --- | --- | --- |
| P1 | 5 | 0 | 5 |
| P2 | 8 | 1 | 7 |
| P3 | 16 | 2 | 14 |

***Bước 5 : Tính Waiting Time (WT)***

| **Process** | **TAT** | **BT** | **WT = TAT – BT** |
| --- | --- | --- | --- |
| P1 | 5 | 5 | 0 |
| P2 | 7 | 3 | 4 |
| P3 | 14 | 8 | 6 |

***Bước 6 : Tính Response Time (RT)***

| **Process** | **Start** | **AT** | **RT = Start – AT** |
| --- | --- | --- | --- |
| P1 | 0 | 0 | 0 |
| P2 | 5 | 1 | 4 |
| P3 | 8 | 2 | 6 |

**Bảng kết quả cuối :**

| **Process** | **AT** | **BT** | **CT** | **TAT** | **WT** | **RT** |
| --- | --- | --- | --- | --- | --- | --- |
| P1 | 0 | 5 | 5 | 5 | 0 | 0 |
| P2 | 1 | 3 | 8 | 7 | 4 | 4 |
| P3 | 2 | 8 | 16 | 14 | 6 | 6 |

#### 2. SJF – Shortest Job First

CÁCH GIẢI BÀI SJF NON-PREEMPTIVE

**🔹 BƯỚC 1: Xem đề và sắp xếp tiến trình theo Arrival Time (AT)**

**🔹 BƯỚC 2: Tại mỗi thời điểm CPU rảnh, chọn tiến trình có BT nhỏ nhất**

**🔹 BƯỚC 3: Tính Start Time**

Start\_Time[i] = max(Current CPU time, Arrival Time)

**🔹 BƯỚC 4: Tính Completion Time (CT = Start + Burst)**

**🔹 BƯỚC 5: Tính Turnaround Time (TAT = CT – AT)**

**🔹 BƯỚC 6: Tính Waiting Time (WT = TAT – BT), Response Time (RT = Start – AT)**

Đề bài :

| **Process** | **Arrival Time (AT)** | **Burst Time (BT)** |
| --- | --- | --- |
| P1 | 0 | 7 |
| P2 | 2 | 4 |
| P3 | 4 | 1 |
| P4 | 5 | 4 |

**BƯỚC 1: Khởi đầu**

* **Thời gian hiện tại = 0**
* Chỉ có P1 đến lúc 0 → chỉ có thể chọn **P1**  
  → **Start(P1) = 0**, chạy đến 0+7 = **7**

**BƯỚC 2: Thời gian = 7**

* Ai đang chờ? P2, P3, P4 đều đến rồi:
* Chọn P3 (BT = 1) → Start = 7, CT = 7+1 = 8

**BƯỚC 3: Thời gian = 8**

* Tiến trình còn lại: P2, P4
* → Cả 2 đều BT = 4 → chọn ai đến trước  
  ✅ Chọn **P2 (đến trước)** → Start = 8, CT = 8+4 = 12

**BƯỚC 4: Thời gian = 12**

Chỉ còn P4 → Start = 12, CT = 12+4 = 16

**Bảng kết quả :**

| **P** | **AT** | **BT** | **Start** | **CT** | **TAT = CT–AT** | **WT = TAT–BT** | **RT = Start – AT** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | 0 | 7 | 0 | 7 | 7 | 0 | 0 |
| P2 | 2 | 4 | 8 | 12 | 10 | 6 | 6 |
| P3 | 4 | 1 | 7 | 8 | 4 | 3 | 3 |
| P4 | 5 | 4 | 12 | 16 | 11 | 7 | 7 |

#### 3. ROUND ROBIN (RR)

RR là thuật toán Preemptive → CPU có thể ngắt tiến trình khi hết time quantum!

**Giải bài RR hơi khác các thuật toán khác, nhưng quy tắc vẫn là:**

1. Sắp xếp tiến trình theo Arrival Time
2. Giữ hàng đợi Ready Queue cập nhật theo thời gian
3. Mỗi vòng quay:
   * Nếu tiến trình còn BT > Q → chạy Q đơn vị, trừ BT, đưa về cuối hàng đợi
   * Nếu BT ≤ Q → chạy xong luôn, tính Completion Time
4. Ghi lại Start Time lần đầu tiên để tính Response Time

**Đề bài :**

| **Tên tiến trình** | **Đến lúc nào (Arrival Time)** | **Cần bao lâu để hoàn tất (Burst Time)** |
| --- | --- | --- |
| P1 | 0 | 5 |
| P2 | 1 | 4 |
| P3 | 2 | 2 |
| P4 | 4 | 1 |

Time Quantum = 2 (Nghĩa là **mỗi tiến trình được chạy 2 đơn vị**, nếu chạy chưa xong thì nhường chỗ cho người tiếp theo)

**Ý tưởng giải :**

* Mỗi người được "chơi" 2 phút
* Chưa chơi xong → quay lại cuối hàng, chờ lượt tiếp
* Ai chơi xong rồi → ra khỏi hàng

Ban đầu: Time = 0

BT ban đầu: P1(5), P2(4), P3(2), P4(1)

Hàng đợi: Cập nhật theo thời gian + ai chưa xong thì đưa về cuối

#### 4. SJF – Shortest Job First – preemtive ( SRTF – Shortest Remaining Time First)

| **Nội dung** | **SJF Non-Preemptive** | **SJF Preemptive (SRTF)** |
| --- | --- | --- |
| Xét Arrival Time đầu tiên | ✅ Có | ✅ Có |
| Chọn BT nhỏ nhất từ các tiến trình đã đến | ✅ Có | ✅ Có |
| Khi đang chạy, tiến trình mới có BT nhỏ hơn thì sao? | ❌ Không ngắt, chờ chạy xong | ✅ Ngắt ngay và chuyển sang tiến trình mới |
| Tính Start/CT/TAT/WT/RT | ✅ Có | ✅ Có (nhưng khó hơn) |
| Dễ làm hơn | ✅ Dễ hơn | ❌ Phức tạp hơn |

BƯỚC 1: Nhập và sắp xếp tiến trình theo Arrival Time (AT)

* Ghi lại tất cả tiến trình theo AT, BT

BƯỚC 2: Lặp qua từng đơn vị thời gian (giây 0 → hết)

* Tại mỗi thời điểm t:
  + Xem các tiến trình nào đã đến (AT ≤ t) và chưa hoàn thành
  + Chọn tiến trình có Burst Time còn lại nhỏ nhất
  + Nếu đó là tiến trình mới → bắt đầu chạy (ghi lại Start Time nếu lần đầu)

**Tại mỗi thời điểm**, nếu có một tiến trình **mới đến** mà **Burst Time còn lại của nó nhỏ hơn** thời gian còn lại của tiến trình đang chạy → **ngắt ngay lập tức** và **chuyển CPU cho tiến trình mới**.

BƯỚC 3: Mỗi đơn vị thời gian → giảm Remaining Time của tiến trình đang chạy

* Nếu Remaining Time của tiến trình = 0 → đánh dấu hoàn thành → ghi Completion Time (CT)
* Cập nhật thời gian hiện tại t = t + 1

**BƯỚC 4: Sau khi hoàn thành tất cả → tính TAT, WT, RT**

| **Tham số** | **Công thức** |
| --- | --- |
| CT (Completion Time) | Khi tiến trình kết thúc thực sự |
| TAT (Turnaround Time) | TAT = CT - AT |
| WT (Waiting Time) | WT = TAT - BT |
| RT (Response Time) | RT = Start - AT (thời gian phản hồi đầu tiên) |

## PROCESS SYNCHRONIZATION ( đồng bộ tiến trình )

Tóm tắt :

* Khi nhiều tiến trình/luồng cùng truy cập dữ liệu → cần đồng bộ
* **Critical Section** là vùng cần bảo vệ
* Dùng **mutex, semaphore** để đảm bảo chỉ một luồng được vào
* **Peterson’s Algorithm** là mô hình lý thuyết để học
* **Deadlock** là nguy cơ nếu không kiểm soát tốt việc khóa

**Deadlock :**

Xảy ra nếu 2 tiến trình giữ tài nguyên, và **chờ tài nguyên của nhau mãi mãi**

Ví dụ:

* Tiến trình A giữ khóa 1, chờ khóa 2
* Tiến trình B giữ khóa 2, chờ khóa 1

→ Mắc kẹt!

**race condition:** khi 2 tiến trình/luồng truy cập cùng dữ liệu cùng lúc, và kết quả phụ thuộc vào thứ tự chạy ngẫu nhiên của CPU.

GIẢI PHÁP:

**CRITICAL SECTION :** Là đoạn mã mà chỉ một tiến trình được phép chạy tại một thời điểm, để tránh race condition.

**CÁC CƠ CHẾ ĐỒNG BỘ :**

**Mutex (Mutual Exclusion Object) ( khoá độc quyền )**

* Tiến trình A vào vùng quan trọng → lock
* Tiến trình B đến → bị chặn nếu chưa unlock
* Sau khi A xong → unlock → B mới vào

**Semaphore**

Semaphore là biến đếm (counter) cho biết có bao nhiêu tiến trình được vào

**Có 2 loại :**

| **Loại** | **Ý nghĩa** |
| --- | --- |
| Binary Semaphore | Giống mutex (chỉ 0 hoặc 1) |
| Counting Semaphore | Cho phép nhiều tiến trình đồng thời |

**PETERSON'S ALGORITHM (chỉ dùng cho 2 tiến trình)**

* Sử dụng 2 biến flag[i] và turn
* Bảo đảm tính chất:
  1. Tương hỗ (Mutual Exclusion)
  2. Tiến trình nào cũng có cơ hội
  3. Không bị chờ vô hạn

**📌 Học để hiểu logic phần mềm, không dùng trong thực tế hiện đại (do hỗ trợ phần cứng tốt hơn)**

**ĐIỀU KIỆN CỦA CRITICAL SECTION TỐT**

1. Mutual Exclusion: không có 2 tiến trình vào cùng lúc
2. Progress: tiến trình nào không ở trong CS thì không cản trở tiến trình khác
3. Bounded Waiting: không ai bị chờ vô hạn

## Deadlock

**Deadlock** là tình huống mà **nhiều tiến trình bị kẹt mãi mãi**, vì mỗi tiến trình đang giữ tài nguyên và **chờ tài nguyên do tiến trình khác giữ**.

Ví dụ dễ hiểu :

* Tiến trình A **giữ máy in**, chờ **máy scan**
* Tiến trình B **giữ máy scan**, chờ **máy in**

→ Cả 2 đều **chờ nhau mãi mãi** → **deadlock**

### 4 ĐIỀU KIỆN GÂY DEADLOCK (THE FOUR NECESSARY CONDITIONS)

| **Điều kiện** | **Diễn giải dễ hiểu** | **Ví dụ minh họa thực tế** |
| --- | --- | --- |
| 1. Mutual Exclusion (Loại trừ lẫn nhau) | Tài nguyên chỉ dùng được cho một tiến trình tại một thời điểm. Không thể chia sẻ. | Máy in chỉ in được cho 1 tiến trình, không thể chia sẻ |
| 2. Hold and Wait (Giữ và chờ) | Tiến trình đã giữ tài nguyên, nhưng vẫn yêu cầu thêm tài nguyên khác và phải chờ | Tiến trình A giữ máy in, chờ máy scan |
| 3. No Preemption (Không bị thu hồi) | Hệ điều hành không được cưỡng bức lấy lại tài nguyên đang cấp → chỉ chờ tiến trình tự trả lại | Không thể bắt A trả máy in, phải chờ A tự giải phóng |
| 4. Circular Wait (Vòng tròn chờ nhau) | Có chuỗi tiến trình mỗi cái giữ 1 tài nguyên, và chờ tài nguyên mà tiến trình khác giữ → thành vòng tròn | A giữ R1, chờ R2; B giữ R2, chờ R3; C giữ R3, chờ R1 |

* **Nếu tất cả 4 điều kiện đúng cùng lúc → deadlock có thể xảy ra**

### CÁCH GIẢI QUYẾT

**1. Deadlock Prevention – Phòng ngừa**

| **Ngăn điều kiện nào?** | **Cách làm cụ thể** |
| --- | --- |
| Hold and Wait | Bắt tiến trình yêu cầu tất cả tài nguyên cùng lúc trước khi chạy |
| No Preemption | Nếu không có đủ tài nguyên → thu hồi tài nguyên đã cấp, buộc tiến trình chờ |
| Circular Wait | Gán số thứ tự cho tài nguyên, chỉ cho yêu cầu theo thứ tự tăng dần |

**Nhược điểm:** Không linh hoạt, dễ gây lãng phí tài nguyên

**2. Deadlock Avoidance – Tránh bế tắc**

ho phép cấp phát tài nguyên **chỉ khi biết chắc không gây deadlock**

* Cần biết trước:
  + Tổng số tài nguyên
  + Mức tối đa từng tiến trình có thể yêu cầu
* Sử dụng: **Banker’s Algorithm** (thuật toán ngân hàng)

**3. Deadlock Detection & Recovery – Phát hiện và xử lý**

Cứ cấp phát bình thường → nếu deadlock xảy ra thì:

* Dò tìm deadlock (dùng biểu đồ tài nguyên, thuật toán)
* **Xử lý**:
  + Giết tiến trình (terminate tiến trình đó)
  + Thu hồi tài nguyên rồi cấp lại
  + **Rollback** (quay về trạng thái an toàn trước đó)

### BANKER’S ALGORITHM (Thuật toán ngân hàng)

Dựa trên nguyên tắc giống như ngân hàng: chỉ cho vay nếu đảm bảo khách có khả năng trả.

Dùng để tránh deadlock trong môi trường nhiều tài nguyên và tiến trình

| **Thành phần** | **Ý nghĩa** |
| --- | --- |
| **Available** | Số lượng tài nguyên còn lại |
| **Max** | Nhu cầu tối đa của mỗi tiến trình |
| **Allocation** | Tài nguyên đã cấp |
| **Need** = Max – Allocation | Nhu cầu còn lại của tiến trình |

Hệ thống chỉ cấp tài nguyên nếu sau khi cấp, **vẫn còn đủ tài nguyên cho tiến trình khác hoàn tất**

Nếu tất cả tiến trình có thể hoàn tất theo 1 thứ tự nào đó → hệ thống **an toàn**

| **Bước** | **Việc cần làm** |
| --- | --- |
| 1. Tính Available | Tổng – đã cấp |
| 2. Tính Need | Max – Allocation |
| 3. Lặp: tìm tiến trình có Need ≤ Available |  |
| 4. Nếu có → giả lập chạy xong → cộng lại vào Available |  |
| 5. Lặp đến khi: |  |
| • tất cả chạy được → Safe |  |
| • không còn ai chạy được → Unsafe |  |

Bài tập ví dụ :

**Giả sử hệ thống có 3 loại tài nguyên: A, B, C**

* Tổng số tài nguyên: A = 10, B = 5, C = 7

Max – Nhu cầu tối đa:

| **Process** | **A** | **B** | **C** |
| --- | --- | --- | --- |
| P0 | 7 | 5 | 3 |
| P1 | 3 | 2 | 2 |
| P2 | 9 | 0 | 2 |
| P3 | 2 | 2 | 2 |
| P4 | 4 | 3 | 3 |

Allocation – Đã cấp:

| **Process** | **A** | **B** | **C** |
| --- | --- | --- | --- |
| P0 | 0 | 1 | 0 |
| P1 | 2 | 0 | 0 |
| P2 | 3 | 0 | 2 |
| P3 | 2 | 1 | 1 |
| P4 | 0 | 0 | 2 |

**Bước 1: Tính Available**

Tổng – tổng cấp:

* A = 10 – (0+2+3+2+0) = **3**
* B = 5 – (1+0+0+1+0) = **3**
* C = 7 – (0+0+2+1+2) = **2**

➡️ Available = **[3, 3, 2]**

**Bước 2: Tính Need = Max – Allocation**

| **Process** | **A** | **B** | **C** |
| --- | --- | --- | --- |
| P0 | 7−0 = 7 | 5−1 = 4 | 3−0 = 3 |
| P1 | 3−2 = 1 | 2−0 = 2 | 2−0 = 2 |
| P2 | 9−3 = 6 | 0−0 = 0 | 2−2 = 0 |
| P3 | 2−2 = 0 | 2−1 = 1 | 2−1 = 1 |
| P4 | 4−0 = 4 | 3−0 = 3 | 3−2 = 1 |

**Bước 3: Xét từng tiến trình xem có chạy được không**

**📌 Ban đầu:**

Available = **[3, 3, 2]**

**📌 Tiến trình nào có Need ≤ Available?**

* P0: [7, 4, 3] ❌ (không đủ A)
* P1: [1, 2, 2] ✅ → **chạy được**

➡️ **Giả sử P1 chạy xong** → trả lại tài nguyên: [2, 0, 0]  
→ New Available = [3+2, 3+0, 2+0] = **[5, 3, 2]**

**→ Safe Sequence so far: [P1]**

**📌 Lần 2: Available = [5, 3, 2]**

Xét lại các tiến trình còn lại:

* P0: [7,4,3] ❌
* P2: [6,0,0] ❌
* P3: [0,1,1] ✅ → chạy được

➡️ P3 giải phóng [2,1,1] → Available = [7, 4, 3]

**→ Safe Sequence: [P1, P3]**

**📌 Lần 3: Available = [7, 4, 3]**

Xét tiếp:

* P0: [7,4,3] ✅  
  ➡️ P0 giải phóng [0,1,0] → Available = [7,5,3]

**→ Safe Sequence: [P1, P3, P0]**

**📌 Lần 4: Available = [7,5,3]**

* P2: [6,0,0] ✅ → giải phóng [3,0,2] → Available = [10,5,5]

**→ Safe Sequence: [P1, P3, P0, P2]**

**📌 Lần 5: Available = [10,5,5]**

* P4: [4,3,1] ✅ → giải phóng [0,0,2] → Available = [10,5,7]

**→ Safe Sequence: [P1, P3, P0, P2, P4]**

✅ ✅ ✅ **Tất cả tiến trình đều chạy được theo thứ tự nào đó → Hệ thống đang ở trạng thái AN TOÀN (safe state)**

Vì **tất cả tiến trình đều chạy được theo một thứ tự nào đó**, nên hệ thống đang ở **safe state** → có thể cấp phát!

## Contiguous Memory Allocation – Cấp phát bộ nhớ liên tục

Là kỹ thuật cấp phát bộ nhớ trong đó **mỗi tiến trình được cấp một vùng liên tiếp (liền nhau)** trong RAM.

Lý do dùng :

* Trước khi có paging, segmentation,... hệ điều hành đơn giản dùng cách này để quản lý bộ nhớ
* Dễ triển khai (vì địa chỉ logic = địa chỉ vật lý + offset)

| **Ưu điểm** | **Nhược điểm** |
| --- | --- |
| Đơn giản, dễ cài đặt | Gây **external fragmentation** (phân mảnh ngoài) |
| Truy cập nhanh (do liên tục) | Khó cấp phát cho tiến trình lớn khi RAM bị chia nhỏ |
| Ít overhead (không cần Page Table) | Không linh hoạt, khó hỗ trợ đa chương trình lớn |

**External fragmentation :**

Khi có nhiều **vùng trống nhỏ lẻ** nằm rải rác trong RAM, tổng dung lượng đủ lớn, nhưng **không vùng nào đủ lớn liên tiếp** để cấp phát cho tiến trình.

**CÁC CHIẾN LƯỢC CẤP PHÁT**