ĐẠI HỌC BÁCH KHOA HÀ NỘI
TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

BÁO CÁO BÀI TẬP LỚN

Chủ đề: Tìm hiểu một số thuật toán điều độ tài nguyên găng

Thành viên:
Nguyễn Vi Thái Sơn — Kim Việt Tiến — Phạm Quang Vinh — Nông Đức Toàn

Học phần:	Nguyên lý Hệ điều hành
Người hướng dẫn:	TS.Phạm Đăng Hải
Ngày thực hiện:	November 1, 2025

LỜI NÓI ĐẦU

Trong nguyên lí hệ điều hành, **tiến trình** được hiểu là một chương trình đang trong trạng thái thực thi. Để hoạt động, mỗi tiến trình đều đòi hỏi được cấp phát các tài nguyên thiết yếu như CPU, bộ nhớ, hay các thiết bị ngoại vi. Trong số đó, tồn tại những tài nguyên hạn chế về khả năng sử dụng chung và cần thiết cho nhiều tiến trình cùng lúc, được gọi là **tài nguyên găng (Critical Resource)**.
Nếu các tiến trình tranh nhau truy cập vào tài nguyên này một cách tự do, sự xung đột dữ liệu là điều tất yếu. Để dễ hình dung, hãy tưởng tượng một ngã tư đường hẹp không có đèn tín hiệu (tài nguyên găng). Nếu các phương tiện (tiến trình) cứ lao vào cùng lúc mà không có sự điều phối, tai nạn và ùn tắc sẽ xảy ra. Tương tự, trong máy tính, điều này dẫn đến sai lệch thông tin và mất tính toàn vẹn dữ liệu.
Do đó, hệ điều hành bắt buộc phải có chương trình điều độ tài nguyên găng để đồng bộ hóa các tiến trình. Một giải pháp điều độ hiệu quả cần phải thỏa mãn ba điều kiện sau:

- **Loại trừ lẫn nhau (Mutual Exclusion):** Mỗi thời điểm, tài nguyên găng không phải phục vụ một số lượng tiến trình vượt quá khả năng của nó.
- **Tiến triển (Progress):** Tài nguyên găng còn khả năng phục vụ và tồn tại tiến trình muốn vào đoạn găng, thì tiến trình đó phải được sử dụng tài nguyên găng.
- **Chờ đợi hữu hạn (Bounded Waiting):** Nếu tài nguyên găng hết khả năng phục vụ và vẫn tồn tại tiến trình muốn vào đoạn găng, thì tiến trình đó phải được xếp hàng chờ đợi và chờ đợi là hữu hạn.

Trong báo cáo này, chúng ta cùng đi sâu vào phân tích các thuật toán giải quyết vấn đề trên, đi từ **công cụ điều độ cấp thấp đến các công cụ điều độ cấp cao**, đồng thời chứng minh tính hợp lý của chúng theo đúng các nguyên tắc đã đề ra.

Mục lục

1. Công cụ điều độ cấp thấp

1.1 Không có sự hỗ trợ phần cứng

1.1.1 Peterson

Nguồn gốc: Giải thuật Peterson do Gary Peterson đề xuất năm 1981 cho bài toán đoạn găng. Giải thuật Peterson thuộc nhóm giải pháp phần mềm, không đòi hỏi sự hỗ trợ từ phía phần cứng hay hệ điều hành và được xem là đại diện tiêu biểu cho nhóm giải pháp phần mềm nhờ ưu điểm đơn giản và dễ cài đặt.

Mô tả bài toán

Cho hai tiến trình P_0, P_1 cùng truy cập một tài nguyên găng duy nhất. Thiết kế thuật toán điều độ đoạn găng thỏa mãn các điều kiện tiên quyết

Nguyên tắc

- Tiến trình P_i dùng biến want[i] trong bộ nhớ nhớ chung làm khóa và một biến turn (dùng chung) để chỉ ra tiến trình có quyền ưu tiên.
- Nếu cả hai cùng xin, ai ghi turn sau cùng sẽ nhường.
- Nếu tiến trình muốn vào đoạn găng thì phải kiểm tra khóa của tiến trình còn lại và quyền ưu tiên đang thuộc về tiến trình nào.
- Ban đầu `wait[0]` $\leftarrow$ `false, wait[1]` $\leftarrow$ `false`

Thuật toán điều độ

Algorithm 1 Peterson cho tiến trình P_i ($j = 1 - i$)

1: **Entry:**
2: `want[i]` $\leftarrow$ `true;` ▷ P_i thông báo đang muốn vào đoạn găng
3: `turn` $\leftarrow$ `j;` ▷ Nhường quyền ưu tiên cho P_j nếu hai bên cùng xin
4: **while** `want[j]` **and** `turn == j` **do** */*busy-wait*/* ▷ Nếu P_j cũng muốn vào & đang được ưu tiên thì chờ
5: **Đoạn găng của tiến trình** P_i ▷ Chỉ P_i được vào tại thời điểm này
6: **Exit:** `want[i]` $\leftarrow$ `false;` ▷ P_i rời đoạn găng, bỏ yêu cầu
7: **Phần còn lại của tiến trình** P_i ▷ Thực hiện công việc ngoài đoạn găng

Chứng minh tính hợp lý của điều độ

Yêu cầu 1 (Loại trừ lẫn nhau)**.** *Không thể xảy ra việc cả P_0 và P_1 cùng ở đoạn găng.*

Chứng minh. Giả sử ngược lại cả hai cùng vào Đoạn găng. Khi vào Đoạn găng, mỗi tiến trình P_i đã thoát khỏi điều kiện `want[j] && turn == j`. Do đó với P_0 vào Đoạn găng hoặc `want[1] = false` hoặc `turn = 0`. Tương tự, với P_1 vào Đoạn găng, hoặc `want[0] = false` hoặc `turn = 1`. Nhưng tại thời điểm cả hai cùng vào Đoạn găng ta có `want[0] = want[1] = true`. Vậy ta phải có `turn = 0` và `turn = 1`, mâu thuẫn. □

Yêu cầu 2 (Tiến triển)**.** *Nếu tài nguyên găng còn khả năng phục vụ và có tiến trình muốn vào Đoạn găng thì tiến trình đó phải được sử dụng tài nguyên găng.*

Chứng minh. Giả sử P_i là tiến trình đang muốn vào Đoạn găng. Ta xét 3 trường hợp sau.

- **TH1:** P_j đang dừng ở vòng lặp while.
  Lúc này `want[i] == true` và `turn == i` Do đó P_i sẽ vào Đoạn găng.
- **TH2:** P_j đang ở trong Đoạn găng. Sau khi P_j ra khỏi Đoạn găng, nó sẽ thực hiện `want[j]` $\leftarrow$ `false`. Lúc này điều kiện lặp while vi phạm nên P_i cũng sẽ thoát khỏi while và vào Đoạn găng.
- **TH3:** P_j đang ở trong phần còn lại của tiến trình. Sau khi thực hiện xong phần còn lại, P_j sẽ quay lại **Entry** và thực hiện `turn` $\leftarrow i$. Do đó P_i cũng sẽ vào Đoạn găng.

□

Yêu cầu 3 (Chờ đợi hữu hạn)**.** *Mỗi lần P_i yêu cầu vào Đoạn găng, nó bị vượt qua bởi P_j nhiều nhất một lần.*

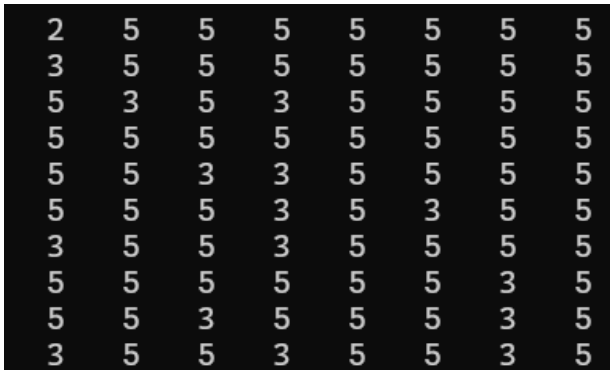
Chứng minh. Khi P_i vào **Entry**, nó gán `turn = j`, nhường ưu tiên *một lần* cho P_j. Sau khi P_j vào và rời Đoạn găng, `want[j]` trở thành `false`; điều kiện chờ của P_i trở thành sai, nên P_i sẽ vào. Vì `turn` chỉ cho phép P_j ưu tiên tối đa một lượt, P_i không thể bị vượt quá hơn một lần. □

Kết quả và thảo luận

Thiết lập đơn giản với 2 luồng. Luồng `T1` in ra $x = y + 1$ liên tục, luồng `T2` cập nhật $y = 2; y = y * 2;$. Ban đầu y = 1.

Khi không đồng bộ

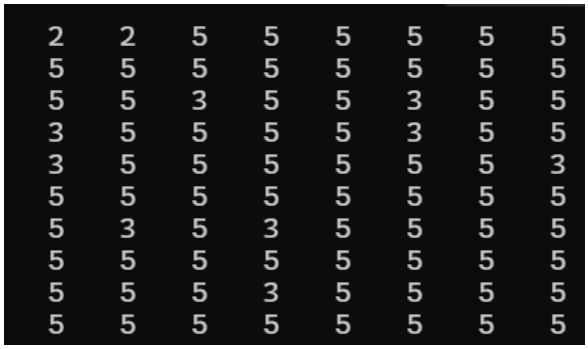
- `T1` có thể đọc y ngay sau lệnh y = 2; của `T2`, trước khi `T2` kịp thực hiện lệnh $y = y * 2;$. Lúc đó, `T1` thấy $y = 2$, và x in ra là 3. Do đó các giá trị có thể được in ra là $\{2, 3, 5\}$



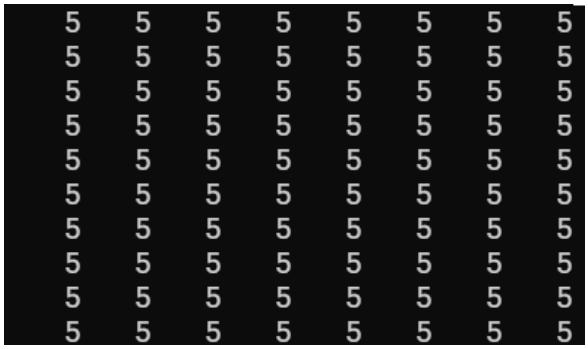
Hình 1: Không đồng bộ

Khi đồng bộ

- Toàn bộ quá trình cập nhật của `T2` ($y = 2; y = y * 2;$) được bảo vệ trong một vùng găng. Do đó `T1` phải đợi cho tới khi `T2` đã hoàn thành. Lúc này $y = 4$ nên $x = 5$.
- Kết quả từ thực nghiệm cho thấy ở Hình 1.1 có hai số 2 đầu tiên. Lý do: lúc khởi động, `T2` chưa đặt `want[1]` $\leftarrow$ `true` (chưa yêu cầu vào đoạn găng) nên điều kiện chờ của `T1` là `false` và `T1` có thể vào lại ngay, in thêm một lần 2 nữa. Khi `T2` bắt đầu yêu cầu và vào đoạn găng thì `T1` phải đợi. Từ đó về sau chỉ còn 5. Số lần xuất hiện 2 ban đầu phụ thuộc Scheduler và có thể khác nhau, nhưng thường rất ít.



Hình 1.1: `T1` thực hiện trước



Hình 1.2: `T2` thực hiện trước

1.1.2 Lamport's Bakery Algorithm (Thuật toán tiệm bánh của Lamport)

Nguồn gốc: Thuật toán Bakery do Leslie Lamport đề xuất năm 1974 để giải bài toán đoạn găng cho nhiều tiến trình ($n > 2$) chỉ sử dụng thao tác đọc/ghi biến chia sẻ. Giống như Peterson, Bakery là một giải pháp phần mềm thuần túy, không yêu cầu các lệnh đặc biệt từ phần cứng. Ý tưởng mô phỏng việc phát số thứ tự trong tiệm bánh, đảm bảo thứ tự công bằng giữa các tiến trình.

Mô tả bài toán

Có n tiến trình $P_1, P_2, \ldots, P_n$ cùng truy cập một tài nguyên găng duy nhất. Thiết kế thuật toán điều độ sao cho thỏa mãn các điều kiện tiên quyết

Nguyên tắc

- Mỗi tiến trình P_i có:
  - `choosing[i]`: đang chọn số thứ tự,
  - `number[i]`: số thứ tự hiện tại (vé xếp hàng).
- Khi muốn vào Đoạn găng, P_i:
  1. Đánh dấu đang chọn số: `choosing[i] ← true;`
  2. Lấy số lớn nhất hiện có và cộng 1:

     `number[i] ← 1 + max(number[1..n]);`

  3. Kết thúc chọn số: `choosing[i] ← false;`
- Thứ tự ưu tiên được so sánh theo cặp $(\texttt{number[i]}, i)$ theo thứ tự từ điển: tiến trình có số nhỏ hơn được ưu tiên; nếu số bằng nhau thì tiến trình có chỉ số i nhỏ hơn được ưu tiên.

Thuật toán điều độ

Biến dùng chung:

- `boolean choosing[1..n] ← {false};`
- `int number[1..n] ← {0};`

Algorithm 2 Thuật toán Bakery cho tiến trình P_i

```
1:  while true do
2:      Entry:
3:      choosing[i] ← true;
4:      number[i] ← 1 + max(number[1..n]);
5:      choosing[i] ← false;
6:      for j := 1 to n, j != i do
7:          while choosing[j] do /*busy-wait*/          ▷ Đợi P_j chọn số xong
8:          while number[j] != 0 and (number[j], j) < (number[i], i) do    ▷ Đợi nếu P_j có quyền ưu tiên hơn
9:      Đoạn găng của tiến trình P_i
10:     Exit: number[i] ← 0;
11:     Phần còn lại của tiến trình P_i
```

Chứng minh tính hợp lý của điều độ

Yêu cầu 1 (Loại trừ lẫn nhau)**.** *Không thể có hai tiến trình cùng ở trong Đoạn găng.*

Chứng minh. Giả sử ngược lại P_i và P_k cùng ở Đoạn găng. Khi vào Đoạn găng, mỗi tiến trình đã:

- Hoàn tất chọn số (`choosing[*] = false`),
- Vượt qua vòng kiểm tra với mọi tiến trình khác.

Xét hai cặp $(\texttt{number[i]}, i)$ và $(\texttt{number[k]}, k)$. Theo quy tắc so sánh từ điển, phải có:

$$(\texttt{number[i]}, i) < (\texttt{number[k]}, k) \quad \text{hoặc} \quad (\texttt{number[k]}, k) < (\texttt{number[i]}, i)$$

Giả sử $(\texttt{number[i]}, i) < (\texttt{number[k]}, k)$. Khi đó, ở bước kiểm tra, P_k phải đợi cho đến khi `number[i] = 0`, tức là P_i đã rời Đoạn găng. Mâu thuẫn với giả thiết cả hai cùng ở Đoạn găng. Trường hợp ngược lại tương tự. □

Yêu cầu 2 (Tiến triển)**.** *Nếu tài nguyên găng còn khả năng phục vụ và có tiến trình muốn vào Đoạn găng thì tiến trình đó phải được sử dụng tài nguyên găng.*

Chứng minh. Mọi tiến trình muốn vào đều lấy một số hữu hạn `number[i]`. Tập các số khác 0 là hữu hạn nên tồn tại tiến trình P_m với cặp $(\texttt{number[m]}, m)$ nhỏ nhất. Trong vòng kiểm tra, với mọi $j \neq m$, điều kiện

$$(\texttt{number[j]}, j) < (\texttt{number[m]}, m)$$

đều sai, nên P_m không phải chờ các tiến trình có ưu tiên thấp hơn. Do đó, P_m sẽ tiến vào Đoạn găng. □

Yêu cầu 3 (Chờ đợi hữu hạn)**.** *Mỗi lần P_i yêu cầu vào Đoạn găng, số lần bị vượt mặt là hữu hạn.*

Chứng minh. Khi P_i chọn xong `number[i]`, mọi tiến trình đến sau chỉ có thể nhận số lớn hơn hoặc bằng `number[i]`. Theo quy tắc so sánh:

- Các tiến trình có `number[j]` nhỏ hơn có thể vào trước P_i, nhưng số này hữu hạn.
- Các tiến trình đến sau có `number[j]` lớn hơn do đó không thể chen vào.

Vì vậy, P_i chỉ phải chờ một lượng hữu hạn tiến trình có số ưu tiên hơn, đảm bảo điều kiện chờ đợi hữu hạn. □

Kết quả và thảo luận

Thiết lập đơn giản với 3 luồng. Luồng `T1` in ra $x = y + 1$ liên tục, luồng `T2` cập nhật $y = 2; y = y * 2$; luồng `T3` cập nhật $y = 5; y = y + 10;$. Ban đầu y = 1.

Khi không đồng bộ:

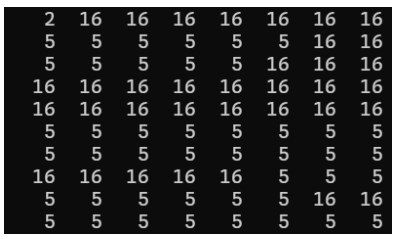
- `T1` có thể đọc y ngay sau lệnh y = 2; của `T2` hoặc ngay sau lệnh $y = 5$ của `T3`, trước khi `T2` kịp thực hiện lệnh $y = y * 2;$ hoặc lệnh $y = y + 10;$ của `T3`. Do đó các giá trị có thể được in ra là $\{2, 3, 5, 6, 16\}$

```
 2  16   3   5  16   5  16   5
 5   5   5  16   5  16   3  16
 5   5   5   5   5  16   5   5
 5   6  16   5   5  16  16   5
16   3   5  16   5   3  16   5
16   6   6  16   5   5  16   5
16   6  16   5   5  16  16   5
 5   5  16   5   5  16  16   3
 5   5   5   5  16  16  16   5
16  16   3  16   5   5   3  16
```

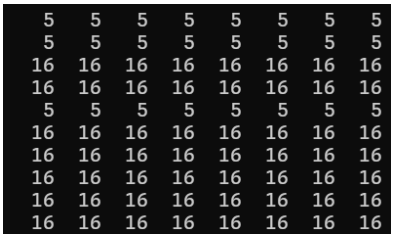
Hình 2: Không đồng bộ

Khi đồng bộ:

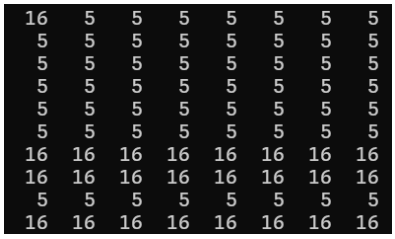
- Toàn bộ quá trình cập nhật của `T2` và `T3` được bảo vệ trong một vùng găng. Do đó `T1` phải đợi cho tới khi `T2` và `T3` đã hoàn thành. Lúc này $y = 4$ và $y = 15$ nên các giá trị có thể được in ra là $\{2, 5, 16\}$.
- Số đầu tiên được in ra sẽ phụ thuộc vào luồng nào sẽ được thực hiện đầu tiên.



Hình 2.1: `T1` thực hiện trước



Hình 2.2: `T2` thực hiện trước



Hình 2.3: `T3` thực hiện trước

1.2 Có sự hỗ trợ phần cứng

1.2.1 Cigarette Smokers Problem (Bài toán người hút thuốc)

Nguồn gốc: Bài toán được Donald E. Knuth nêu ra và được trình bày trong giáo trình **Operating Systems** của **A. Silberschatz** và **William Stallings**. Đây là một ví dụ kinh điển về **đồng bộ hoá nhiều tiến trình**, minh họa cách sử dụng cơ chế chờ có điều kiện (*condition synchronization*) và semaphore để điều phối hoạt động giữa các tiến trình có quan hệ ràng buộc.

Mô tả bài toán

Có ba người hút thuốc ngồi quanh một chiếc bàn. Để hút thuốc, mỗi người cần ba thành phần: **thuốc lá (tobacco)**, **giấy (paper)**, và **diêm (matches)**. Mỗi người hút thuốc chỉ có vô hạn **một loại nguyên liệu**:

- Người hút thuốc A có vô hạn *thuốc lá (tobacco)*.
- Người hút thuốc B có vô hạn *giấy (paper)*.
- Người hút thuốc C có vô hạn *diêm (matches)*.

Ngoài ra, có một **người cung cấp (Agent)** ngồi giữa bàn, người này liên tục chọn ngẫu nhiên **2 trong 3 nguyên liệu** và đặt chúng lên bàn. Sau đó, Agent chờ đến khi người hút thuốc có nguyên liệu còn lại lấy được chúng, cuốn thuốc và hút xong, rồi mới tiếp tục.

Ví dụ:

- Nếu Agent đặt *giấy* và *diêm*, người có *thuốc lá* sẽ hút.
- Khi người hút thuốc đó hút xong, họ báo hiệu để Agent đặt cặp nguyên liệu mới.

Nguyên tắc

- Chỉ một người hút thuốc có thể hút tại một thời điểm.
- Agent không được đặt nguyên liệu mới cho đến khi người hút thuốc hiện tại hoàn tất.
- Mỗi người hút thuốc chỉ được hoạt động khi đủ 3 nguyên liệu trên bàn.

Phân tích

(a) **Agent (Người cung cấp):** chọn ngẫu nhiên 2 nguyên liệu và đặt lên bàn.

(b) **Smokers (Người hút thuốc):** có 3 tiến trình tương ứng với 3 loại nguyên liệu riêng biệt.

(c) **Tuỳ chọn (Giám sát viên):** dùng để kiểm tra trạng thái bàn (trống / có nguyên liệu), không bắt buộc.

Mô hình hoạt động tổng quát

Agent $\longrightarrow$ Đặt 2 nguyên liệu ngẫu nhiên lên bàn
$\downarrow$
Smoker phù hợp $\longrightarrow$ Lấy nguyên liệu còn thiếu, cuốn và hút thuốc
$\downarrow$
Smoker báo hiệu $\longrightarrow$ Cho phép Agent tiếp tục vòng mới

Đề xuất giải pháp bằng Semaphore

Table 1.1: Các Semaphore trong bài toán Người hút thuốc

Tên	Giá trị khởi tạo	Chức năng / Mục đích
`agentSem`	1	Đảm bảo Agent chỉ hoạt động khi bàn trống.
`tobaccoSem`	0	Báo hiệu cho người hút thuốc có thuốc lá khi 2 nguyên liệu còn lại có sẵn.
`paperSem`	0	Báo hiệu cho người hút thuốc có giấy.
`matchSem`	0	Báo hiệu cho người hút thuốc có diêm.
`doneSem`	0	Báo hiệu người hút thuốc đã hoàn tất một lượt.
`mutex`	1	Bảo vệ truy cập vùng nhớ dùng chung (bàn, biến trạng thái).

Thuật toán điều độ

Algorithm 3 Tiến trình Người cung cấp (Agent)

```
1: while true do                                  ▷ Lặp vô hạn — Agent luôn hoạt động liên tục
2:     wait(agentSem)                             ▷ Đảm bảo chỉ hoạt động khi bàn trống
3:     (item1, item2) = random(2 ingredients)     ▷ Chọn ngẫu nhiên nguyên liệu
4:     if (item1, item2) == (paper, match) then
5:         signal(tobaccoSem)  ▷ Báo hiệu cho người có thuốc lá khi thiếu nguyên liệu
6:     else if (item1, item2) == (tobacco, match) then
7:         signal(paperSem)    ▷ Báo hiệu cho người có giấy (thiếu 2 thành phần này)
8:     else if (item1, item2) == (tobacco, paper) then
9:         signal(matchSem)    ▷ Báo hiệu cho người có diêm (thiếu 2 thành phần này)
10:    wait(doneSem)                              ▷ Đợi người hút thuốc báo hiệu rằng đã hút xong
```

Algorithm 4 Tiến trình Người hút thuốc có `tobacco` (Smoker with Tobacco)

```
1: while true do                ▷ Tiến trình lặp vô hạn — người hút thuốc luôn sẵn sàng
2:     wait(tobaccoSem)         ▷ Chờ tín hiệu từ Agent khi có (giấy, diêm) trên bàn
3:     makeCigarette()          ▷ Lấy nguyên liệu, cuốn điếu thuốc hoàn chỉnh
4:     smoke()    ▷ Mô phỏng hành động hút thuốc — chiếm dụng CPU một thời gian
5:     signal(doneSem)          ▷ Báo hiệu cho Agent biết rằng đã hút xong
6:     signal(agentSem)  ▷ Cho phép Agent tiếp tục đặt nguyên liệu cho vòng kế tiếp
```

Algorithm 5 Tiến trình Người hút thuốc có `paper` (Smoker with Paper)

```
1: while true do ▷ Vòng lặp vô hạn — người hút thuốc luôn trong trạng thái sẵn sàng
2:     wait(paperSem)           ▷ Đợi tín hiệu từ Agent khi trên bàn có (thuốc lá, diêm)
3:     makeCigarette()          ▷ Kết hợp giấy, thuốc lá và diêm để cuốn điếu thuốc
4:     smoke()        ▷ Mô phỏng hành động hút thuốc — tạm chiếm tài nguyên CPU
5:     signal(doneSem)          ▷ Báo hiệu rằng người hút đã hoàn tất quá trình hút
6:     signal(agentSem)         ▷ Cho phép Agent tiếp tục đặt nguyên liệu mới lên bàn
```

Algorithm 6 Tiến trình Người hút thuốc có `match` (Smoker with Match)

```
1: while true do              ▷ Vòng lặp vô hạn — người hút thuốc luôn chờ nguyên liệu
2:     wait(matchSem)         ▷ Đợi tín hiệu từ Agent khi trên bàn có (thuốc lá, giấy)
3:     makeCigarette()        ▷ khi đủ nguyên liệu thì cuốn điếu thuốc hoàn chỉnh
4:     smoke()  ▷ hành động hút thuốc — tạm chiếm CPU trong một khoảng thời gian
5:     signal(doneSem)  ▷ Báo cho Agent biết rằng người hút đã hút xong, bàn trống
6:     signal(agentSem)  ▷ Cho phép Agent đặt nguyên liệu mới cho vòng lặp kế tiếp
```

Chứng minh tính hợp lý của điều độ

Yêu cầu 1 (Loại trừ lẫn nhau)**.** *Tại mọi thời điểm, chỉ một người hút thuốc được phép hút.*

Chứng minh. Trong mỗi vòng lặp, Agent chỉ phát tín hiệu `signal()` cho đúng một semaphore tương ứng với người hút thuốc phù hợp. Các semaphore khác vẫn ở trạng thái chờ, do đó chỉ một tiến trình được đánh thức. □

Yêu cầu 2 (Tiến triển)**.** *Khi Agent đặt đúng 2 món nguyên liệu lên bàn, thì người hút thuốc sở hữu món thứ 3 tương ứng PHẢI được đánh thức để hút thuốc.*

Chứng minh. Khi Agent cung cấp nguyên liệu (ví dụ: paper, match), Agent gọi lệnh signal(tobaccoSem) (Dòng 5, Algo 3). Lệnh này nhắm chính xác và duy nhất vào Smoker có tobacco. Smoker có tobacco đang chờ tại wait(tobaccoSem) (Dòng 2, Algo 4) sẽ nhận tín hiệu và được đánh thức ngay lập tức. Các Smoker khác vẫn tiếp tục ngủ. □

Yêu cầu 3 (Chờ đợi hữu hạn)**.** *Mỗi người hút thuốc không bị chờ lâu vô hạn.*

Chứng minh. Do Agent chọn ngẫu nhiên 2 nguyên liệu trong mỗi vòng, mỗi người hút thuốc đều có xác suất hữu hạn để được đánh thức. Hệ thống tuần hoàn đảm bảo không có tiến trình nào bị đói tài nguyên. □

1.2.2 The Roller Coaster Problem (Bài toán Tàu lượn siêu tốc)

Nguồn gốc: Bài toán Tàu lượn siêu tốc (The Roller Coaster Problem) là một bài toán cổ điển trong lập trình đồng thời và hệ điều hành, được đề xuất bởi **Max Hailperin** cho giáo trình *Operating Systems and Middleware*. Bài toán này được sử dụng để minh họa việc điều phối hoạt động phức tạp giữa hai nhóm tiến trình (Luồng): nhóm **Toa tàu (Car)** và nhóm **Hành khách (Passenger)**, sử dụng các cơ chế đồng bộ hóa như Semaphore và Barrier.

Mô tả bài toán

Mô hình bao gồm hai loại tiến trình hoạt động theo chu kỳ:

(a) **Toa tàu (Car, 1 tiến trình):** Chịu trách nhiệm chở hành khách.

(b) **Hành khách (Passengers, N tiến trình):** Muốn đi tàu lượn.

Toa tàu có sức chứa tối đa là C hành khách (C là hằng số, $C < N$). Các yêu cầu về hoạt động của hệ thống là:

(a) **Đón khách:** Toa tàu phải đợi cho đến khi có **đủ C hành khách** lên tàu mới có thể bắt đầu chuyến đi.

(b) **Hành trình:** Khi toa tàu đang chạy, hành khách không thể lên hay xuống.

(c) **Trả khách:** Sau khi chuyến đi kết thúc, toa tàu phải đợi cho đến khi **tất cả C hành khách** đã xuống hết trước khi quay lại ga để đón chuyến tiếp theo.

(d) **Hành khách:** Hành khách phải đợi cho đến khi có chỗ trên toa tàu để lên, và phải đợi cho đến khi chuyến đi kết thúc mới được xuống.

Đề xuất giải quyết bằng kỹ thuật Semaphore

Ta định nghĩa các biến đếm và semaphore sau:

Table 1.2: Các biến và Semaphore cho Bài toán Tàu lượn siêu tốc

Tên	Khởi tạo	Mục đích
`mutex`	1	Bảo vệ biến đếm chung `passengersOnBoard`.
`boardQueue`	0	Toa tàu chờ ở đây (đủ hành khách).
`unboardQueue`	0	Hành khách chờ ở đây (kết thúc chuyến đi).
`allAboard`	0	Rào cản để C hành khách chờ toa tàu chạy (Broadcast Signal).
`allUnboard`	0	Rào cản cho toa tàu chờ C hành khách xuống hết (Final Signal).
`passengersOnBoard`	0	Biến đếm số hành khách đã lên tàu.

Thuật toán điều độ

Algorithm 7 Tiến trình Toa tàu (`CarThread`)

```
1: while true do
2:     wait(boardQueue)                          ▷ Chờ hành khách cuối cùng đánh thức
3:     loadPassengers()                          ▷ Hành khách đã lên, tàu chuẩn bị
4:     // Tàu báo hiệu cho C khách lên tàu đi tiếp (Rào cản 1)
5:     for i ← 1 to C do
6:         signal(allAboard)
7:     runRide()                                 ▷ Tàu chạy chuyến đi
8:     // Tàu dừng, báo hiệu cho C khách được xuống
9:     for i ← 1 to C do
10:        signal(unboardQueue)
11:    wait(allUnboard)                          ▷ Chờ cho khách thứ C xác nhận xuống hết
12:    unloadPassengers()                        ▷ Khách đã xuống hết, tàu sẵn sàng đón khách mới
```

Algorithm 8 Tiến trình Hành khách (`PassengerThread`) - N tiến trình (Đã hiệu chỉnh)

```
1:  while true do
2:      wait(mutex)
3:      // 1. Lên tàu (Vùng găng)
4:      passengersOnBoard ← passengersOnBoard + 1
5:      if passengersOnBoard == C then
6:          signal(boardQueue)        ▷ Khách thứ C đánh thức tàu → Tàu khởi hành
7:      signal(mutex)
8:      wait(allAboard)                  ▷ Chờ tín hiệu từ tàu báo hiệu đã đủ khách
9:      ride()                                                  ▷ Tham gia chuyến đi
10:     // 2. Xuống tàu
11:     wait(unboardQueue)                ▷ Chờ tàu dừng để được phép xuống
12:     getOff()                                              ▷ Hành động xuống tàu
13:     // 3. Báo hiệu xuống xong (Vùng găng)
14:     wait(mutex)
15:     passengersOnBoard ← passengersOnBoard - 1
16:     if passengersOnBoard == 0 then
17:         signal(allUnboard)          ▷ Khách cuối cùng báo hiệu cho tàu về ga
18:     signal(mutex)
```

Chứng minh tính hợp lý của điều độ

Yêu cầu 1 (Loại trừ lẫn nhau)**.** *Biến trạng thái chung* ***passengersOnBoard*** *luôn được truy cập một cách độc quyền.*

Chứng minh. Trong các tiến trình, việc truy cập và thay đổi biến `passengersOnBoard` luôn được đặt trong vùng loại trừ hỗ tương được bảo vệ bởi `mutex`. Điều này ngăn ngừa tình trạng tranh chấp (race condition). □

Yêu cầu 2 (Tiến triển)**.** *Toa tàu chắc chắn sẽ khởi hành ngay khi điều kiện về số lượng hành khách được thỏa mãn, và hành khách chắc chắn được xuống khi tàu dừng.*

Chứng minh: **Giai đoạn Đón khách (Loading):** Giả sử Toa tàu đang chờ tại `wait(boardQueue)` và có đủ C hành khách muốn đi. Mỗi hành khách khi đến đều thực hiện tăng biến đếm `passengersOnBoard` trong vùng độc quyền (`mutex`). Hành khách thứ C chắc chắn sẽ thỏa mãn điều kiện `passengersOnBoard == C` và thực hiện lệnh `signal(boardQueue)`. Không xảy ra trường hợp tàu vẫn còn có chỗ mà hành khách không được lên tàu.

Giai đoạn Trả khách (Unloading): Sau khi tàu chạy xong, Toa tàu phát C tín hiệu `signal(unboardQueue)`, mỗi 1 hành khách xuống tàu sẽ sử dụng đèn báo mutex, biến passengerOnBoard được trừ đi 1. Điều này đảm bảo tất cả C hành khách đang chờ xuống tàu đều được đánh thức để thực hiện `getOff()`. Hành khách cuối cùng rời đi (khi `passengersOnBoard == 0`) sẽ gửi tín hiệu `signal(allUnboard)`, cho phép Toa tàu quay lại trạng thái đón khách mới.

Do đó, chu trình hoạt động của hệ thống là liên tục và phụ thuộc chặt chẽ vào số lượng tiến trình tham gia, thỏa mãn yêu cầu Tiến triển. □

Yêu cầu 3 (Chờ đợi hữu hạn)**.** *Mỗi hành khách đều có thể hoàn thành chuyến đi trong thời gian hữu hạn.*

Chứng minh. Bởi vì số lượng hành khách là hữu hạn và toa tàu liên tục hoạt động theo chu kỳ, mọi hành khách đã lên tàu đều được giải phóng bằng `signal(unboardQueue)`. Cơ chế đếm `passengersOnBoard` đảm bảo đúng C hành khách tham gia và C hành khách rời đi, từ đó ngăn chặn tình trạng đói tài nguyên cho các tiến trình Hành khách. □

1.2.3 Santa Claus (Bài toán Ông già Noel)

Nguồn gốc: Bài toán Ông già Noel (Santa Claus Problem) xuất phát từ cuốn **Operating Systems của William Stallings** , nhưng ông cho rằng nó được tạo ra bởi **John Trono** của St. Michael's College ở Vermont. Đây là một bài toán cổ điển về đồng bộ hoá được đề xuất nhằm minh họa các vấn đề điều độ phức tạp giữa nhiều nhóm tiến trình có ưu tiên khác nhau.

Mô tả bài toán

Ông già Noel (**Santa**) ngủ trong cửa hàng của mình ở Bắc Cực và chỉ có thể được đánh thức bởi 9 con tuần lộc trở về từ kỳ nghỉ của chúng ở Nam Thái Bình Dương, hoặc 3 chú lùn gặp khó khăn khi làm đồ chơi; để cho ông Santa có thể được ngủ một chút, các chú lùn chỉ được đánh thức khi có đủ ba chú lùn gặp vấn đề. Khi ba chú lùn đang giải quyết vấn đề của họ, bất kỳ chú lùn khác muốn thăm ông Santa đều phải đợi cho đến khi ba chú lùn đó trở về. Nếu ông Santa thức dậy và thấy ba chú lùn đang đợi ở cửa hàng của mình, cùng với con tuần lộc cuối cùng trở về từ vùng nhiệt đới, ông Santa đã quyết định rằng các chú lùn có thể đợi cho đến sau ngày Giáng sinh,vì việc chuẩn bị chiếc xe trượt của ông quan trọng hơn. (Giả định rằng các con tuần lộc không muốn rời khỏi vùng nhiệt đới, và do đó chúng ở lại đó cho đến phút cuối cùng.) Con tuần lộc cuối cùng đến phải tìm ông Santa trong khi những con khác đợi trong một căn nhà ấm.

Ông già Noel (Santa, 1 tiến trình) là tài nguyên chung, phải ngủ cho đến khi được đánh thức bởi một trong hai nhóm sau:

(a) **Tuần lộc (Reindeer,** 9 **tiến trình):** Khi cả 9 tuần lộc đã về sau kỳ nghỉ.

(b) **Chú lùn (Elf,** 10 **tiến trình):** Khi 3 chú lùn tập hợp lại để xin tư vấn.

(c) **Yêu cầu Ưu tiên:** Santa phải **luôn ưu tiên** phục vụ nhóm Tuần lộc nếu cả hai nhóm cùng chờ. Santa không tham gia vào quá trình tập hợp nhóm (tự đồng bộ hoá nhóm).

Yêu cầu về thuật toán đồng bộ:

(a) **Đồng bộ hóa Tuần lộc:** Sau khi con tuần lộc thứ chín (thành viên cuối cùng) đến và đánh thức Santa, tiến trình Santa phải gọi hàm `prepareSleigh()`. Ngay sau đó, tất cả chín con tuần lộc phải đồng thời thực hiện hàm `getHitched()` trước khi chuyển sang trạng thái nghỉ tiếp.

(b) **Đồng bộ hóa Chú lùn:** Sau khi ba chú lùn (thành viên thứ ba) tập hợp và đánh thức Santa, tiến trình Santa phải gọi hàm `helpElves()`. Đồng thời, cả ba chú lùn đó phải gọi hàm `getHelp()`.

(c) **Loại trừ Nhóm (Elves Group Exclusion):** Tất cả ba chú lùn được phục vụ phải gọi hàm `getHelp()` và rời khỏi khu vực chờ trước khi bất kỳ chú lùn nào khác được phép vào (tức là tăng biến đếm số lượng chú lùn đang chờ - `elfCount` - cho nhóm tiếp theo). Điều này đảm bảo Santa chỉ giải quyết các vấn đề theo từng nhóm 3 người.

(d) **Cơ chế Đếm và Rào cản:** Cần cơ chế đếm số lượng Tuần lộc (`reindeerCount`) hoặc Chú lùn (`elfCount`) đã sẵn sàng.

- Tuần lộc cần một **rào cản (barrier)** để đảm bảo rằng **9 con** đều đã đến trước khi tiến trình của chúng đi tiếp và đánh thức Santa.
- Tương tự, Chú lùn cần một **rào cản** chỉ cho phép **3 chú lùn** tiếp theo đi vào và đánh thức Santa.

(e) **Cơ chế Báo hiệu (Signaling):** Cần cơ chế báo hiệu (`santaSem`) để Santa biết rằng một nhóm (Tuần lộc thứ 9 hoặc Chú lùn thứ 3) đã hoàn chỉnh và sẵn sàng được phục vụ.

Đề xuất giải quyết bằng kỹ thuật Semaphore

Để giải quyết bài toán bằng **Semaphore**, ta định nghĩa các biến đếm và semaphore sau:

Table 1.3: Các biến và Semaphore cho Bài toán Santa Claus

Tên	**Khởi tạo**	**Mục đích**
`mutex`	1	Bảo vệ các biến đếm chung (`reindeerCount`, `elfCount`).
`santaSem`	0	Santa chờ ở đây (ngủ). Được `signal` khi một nhóm hoàn chỉnh.
`reindeerSem`	0	Rào cản cho 9 Tuần lộc đợi Santa hoàn thành `prepareSleigh()`.
`elfSem`	0	Rào cản cho 3 Chú lùn đợi Santa hoàn thành `helpElves()`.
`elfMutex`	0	Ngăn chú lùn mới vào cho đến khi nhóm hiện tại rời đi.
`reindeerCount`	0	Biến đếm số Tuần lộc đã trở về.
`elfCount`	0	Biến đếm số Chú lùn đang chờ sự giúp đỡ.
`elvesReady`	0	Biến đếm số chú lùn đã hoàn thành `getHelp()` và sẵn sàng rời khỏi khu vực chờ.

Thuật toán điều độ

Thuật toán bao gồm logic cho ba loại tiến trình. Hàm `wait(S)` và `signal(S)` được dùng cho semaphore S.

Algorithm 9 Tiến trình Santa

```
while true do
    wait(santaSem)                          ▷ Santa ngủ và chờ được đánh thức
    wait(mutex)
    if reindeerCount == 9 then              ▷ Kiểm tra ưu tiên Tuần lộc
        prepareSleigh()
        for i ← 1 to 9 do
            signal(reindeerSem)             ▷ Giải phóng 9 Tuần lộc
        reindeerCount ← 0
    else if elfCount > 0 then               ▷ Santa được đánh thức bởi Chú lùn (≥ 3)
        helpElves()
        for i ← 1 to 3 do
            signal(elfSem)                  ▷ Giải phóng 3 Chú lùn
    signal(mutex)
```

Algorithm 10 Tiến trình Tuần lộc (9 tiến trình)

```
wait(mutex)
reindeerCount ← reindeerCount + 1
if reindeerCount == 9 then
    signal(santaSem)                        ▷ Tuần lộc cuối cùng đánh thức Santa
signal(mutex)
wait(reindeerSem)                           ▷ Chờ Santa chuẩn bị xe trượt
getHitched()                                ▷ Tham gia đoàn xe
```

Algorithm 11 Tiến trình Chú lùn (10 tiến trình)

```
wait(elfMutex)                  ▷ Khóa cổng vào để chỉ 3 chú lùn đầu tiên đi qua
wait(mutex)
elfCount ← elfCount + 1
if elfCount == 3 then
    signal(santaSem)            ▷ Chú lùn thứ 3 đánh thức Santa
signal(mutex)
signal(elfMutex)    ▷ Mở khóa cổng: Cho phép chú lùn thứ 4... vào và chờ nhóm tiếp theo
wait(elfSem)                    ▷ Chờ Santa tư vấn
getHelp()                       ▷ Nhận tư vấn
wait(mutex)
elfCount ← elfCount - 1
elvesReady ← elvesReady + 1     ▷ Đánh dấu chú lùn này đã xong
if elvesReady == 3 then         ▷ Nếu là chú lùn thứ 3 trong nhóm đã rời đi
    elvesReady ← 0              ▷ Đặt lại biến đếm nhóm
    wait(elfMutex)  ▷ Khóa cổng (elfMutex) lần nữa, ngăn nhóm mới đếm cho đến khi họ sẵn sàng
signal(mutex)
```

Chứng minh tính hợp lý của điều độ

Yêu cầu 1 (Loại trừ lẫn nhau)**.** *Santa chỉ phục vụ một nhóm (hoặc Tuần lộc hoặc Chú lùn) tại một thời điểm.*

Chứng minh. Việc kiểm tra điều kiện và thực hiện hành động phục vụ của Santa được đặt bên trong vùng loại trừ lẫn nhau được bảo vệ bởi `mutex` (Dòng 3 - 17 trong *Tiến trình Santa*). Điều kiện `If...ElseIf` đảm bảo chỉ một nhánh được thực thi cho mỗi lần Santa thức dậy, từ đó ngăn chặn việc phục vụ đồng thời. □

Yêu cầu 2 (Tiến triển)**.** *Khi Santa đang ngủ, nếu một nhóm Tuần lộc (9 con) hoặc một nhóm Chú lùn (3 người) đã tập hợp đủ, Santa chắc chắn sẽ thức giấc và thực hiện hành động phục vụ tương ứng mà không bị trì hoãn.*

Chứng minh. Ta chứng minh hệ thống luôn tiến triển dựa trên sự khớp nối tuyệt đối giữa *Điều kiện đánh thức* và *Điều kiện thực thi*:

Đảm bảo Santa được đánh thức (Wake-up Guarantee) Santa chỉ bị chặn tại lệnh `wait(santaSem)`. Trạng thái ngủ này chắc chắn bị phá vỡ khi:

Con Tuần lộc cuối cùng đến: Nó tăng `reindeerCount` lên 9 và gọi `signal(santaSem)`.

Chú lùn thứ 3 đến: Nó tăng `elfCount` lên 3 và gọi `signal(santaSem)`.

⇒ Vậy, bất cứ khi nào một nhóm hình thành đủ, Santa sẽ nhận được tín hiệu để chuyển từ trạng thái *Blocked* sang *Ready*.

Đảm bảo Santa thực hiện hành động (Execution Guarantee) Ngay khi thức dậy và giữ `mutex`, Santa thực hiện chuỗi kiểm tra logic:

$$\textbf{If } (\texttt{reindeerCount} == 9) \quad \ldots \quad \textbf{ElseIf } (\texttt{elfCount} \geq 3)$$

Do tín hiệu `santaSem` chỉ được phát ra khi các biến đếm này đã đạt ngưỡng, nên khi Santa tỉnh dậy, ít nhất một trong hai điều kiện trên chắc chắn là đúng.

Nếu Tuần lộc gọi: Điều kiện `If` đúng → Santa chuẩn bị xe trượt.

Nếu Chú lùn gọi (và không có Tuần lộc): Điều kiện `ElseIf` đúng → Santa giúp đỡ.

Nếu cả hai cùng gọi: Điều kiện `If` đúng (do ưu tiên) → Santa phục vụ Tuần lộc.

Santa không bao giờ rơi vào trường hợp thức dậy nhưng không lọt vào nhánh nào (Deadlock logic), do đó hệ thống luôn có sự tiến triển. □

Yêu cầu 3 (Chờ đợi hữu hạn)**.** *Không tiến trình nào bị trì hoãn vô hạn.*

Chứng minh. **Tuần lộc:** Sau khi 9 con đã về, `santaSem` được `signal`. Santa thức dậy và phục vụ (do ưu tiên). Santa thực hiện 9 lần `signal(reindeerSem)`, giải phóng tất cả 9 Tuần lộc đang chờ trên `reindeerSem`.

Chú lùn: Sau khi 3 chú lùn đã tập hợp, `santaSem` được `signal`. Mặc dù nhóm này có thể phải đợi nếu nhóm Tuần lộc đang chờ phục vụ, nhưng sau khi Tuần lộc xong, Santa sẽ phục vụ nhóm Chú lùn (nếu `elfCount` vẫn ≥ 3). Santa thực hiện 3 lần `signal(elfSem)`, giải phóng 3 chú lùn đang chờ. Vì số lượng tiến trình Tuần lộc là hữu hạn và sự kiện chúng quay lại là không liên tục, sự chờ đợi của Chú lùn là hữu hạn. □

2. Công cụ điều độ cấp cao

2.1 Traffic Light Intersection Problem (Bài toán Ngã tư đèn giao thông)

Nguồn gốc: Bài toán **Ngã tư đèn giao thông (Traffic Light Intersection)** được sử dụng phổ biến trong lĩnh vực **Đồng bộ hoá tiến trình (Process Synchronization)** để minh hoạ cách áp dụng cơ chế **Monitor** và **Condition Variables** trong việc điều phối tài nguyên chia sẻ.

Bài toán mô phỏng cách điều khiển luồng xe di chuyển tại một giao lộ sao cho **an toàn, không xung đột** và **tuân thủ tín hiệu đèn**. Mô hình này được giới thiệu và phân tích trong các tài liệu học thuật sau:

- *Modelling and Simulation of a Multi-Phase Traffic Light Controlled Cross-Type Intersection using Timed Coloured Petri Nets*, G. R. Adesina et al., **Journal of Scientific & Industrial Research**, 2011.
- *Self-Control of Traffic Lights and Vehicle Flows in Urban Road Networks*, Stefan Lämmer & Dirk Helbing, 2008.
- Ví dụ mô phỏng "*Traffic Lights and Intersection*" trong bài giảng **Monitors & Condition Synchronization** của **Magee & Kramer (2015)**, Imperial College London.

Mô tả bài toán

Tại một ngã tư có bốn hướng giao nhau: Bắc (N), Nam (S), Đông (E), và Tây (W). Mỗi hướng có dòng xe chờ đèn giao thông. Mỗi xe chỉ được phép đi khi:

- Đèn của hướng đó đang **xanh**.
- Không có xe nào từ hướng vuông góc đang băng qua ngã tư.

Đèn giao thông luân phiên bật theo chu kỳ:

$$\text{(N, S)} \rightarrow \text{Xanh, (E, W)} \rightarrow \text{Đỏ}$$
$$\text{(E, W)} \rightarrow \text{Xanh, (N, S)} \rightarrow \text{Đỏ}$$

Nguyên tắc

- Tại mọi thời điểm, chỉ các xe từ một cặp hướng đối diện được phép đi (N–S hoặc E–W).

- Các xe phải chờ nếu đèn hướng của mình đang đỏ.
- Khi đèn đổi, các xe ở hướng đang xanh được đánh thức để đi tiếp.

Phân tích

(a) **TrafficLightController (Bộ điều khiển đèn):** Theo chu kỳ thời gian, chuyển đổi trạng thái đèn giữa hai nhóm hướng.

(b) **Car (Xe cộ):** Mỗi tiến trình xe thuộc về một hướng (N, S, E hoặc W), sẽ chờ đèn xanh rồi băng qua ngã tư.

(c) **Monitor (Giao lộ):** Quản lý trạng thái đèn và đồng bộ việc các xe chờ / đi qua dựa trên điều kiện đèn.

Mô hình hoạt động tổng quát

TrafficLightController	$\longrightarrow$	Đổi trạng thái đèn (N–S $\leftrightarrow$ E–W)
		$\downarrow$
Monitor (Intersection)	$\longrightarrow$	Đánh thức các xe có đèn xanh, chặn các xe đèn đỏ
		$\downarrow$
Car tiến trình	$\longrightarrow$	Khi được phép, đi qua ngã tư, sau đó thoát khỏi hệ thống

Đề xuất giải pháp bằng Monitor & Condition Variables

Algorithm 12 Monitor `Intersection`

```
1:  shared variable: lightState = NS_GREEN          ▷ Trạng thái đèn hiện tại
2:  condition condNS, condEW                        ▷ Điều kiện chờ cho 2 hướng
3:  function ARRIVE(direction)
4:      if direction ∈ {N, S} then                  ▷ Hướng Bắc/Nam
5:          while lightState != NS_GREEN do         ▷ Chờ đèn xanh NS
6:              wait(condNS)                        ▷ Tạm dừng xe
7:      else if direction ∈ {E, W} then             ▷ Hướng Đông/Tây
8:          while lightState != EW_GREEN do         ▷ Chờ đèn xanh EW
9:              wait(condEW)                        ▷ Tạm dừng xe
10:     crossIntersection()                         ▷ Đi qua ngã tư
11: function SWITCHLIGHT
12:     if lightState == NS_GREEN then              ▷ Nếu đèn đang xanh NS
13:         lightState = EW_GREEN                   ▷ Chuyển sang đèn xanh EW
14:         signalAll(condEW)                       ▷ Đánh thức xe hướng EW
15:     else
16:         lightState = NS_GREEN                   ▷ Chuyển sang đèn xanh NS
17:         signalAll(condNS)                       ▷ Đánh thức xe hướng NS
```

Tiến trình Bộ điều khiển đèn (Controller)

Algorithm 13 Tiến trình `TrafficLightController`

```
1: while true do
2:     wait(timeQuantum)      ▷ Giữ trạng thái đèn trong một khoảng thời gian cố định
3:     Intersection.switchLight()      ▷ Đổi đèn, cho phép hướng còn lại di chuyển
```

Tiến trình Xe (Car)

Algorithm 14 Tiến trình `Car(direction)`

```
1: Intersection.arrive(direction)          ▷ Xe đến ngã tư và chờ nếu đèn đỏ
2: exitIntersection()                      ▷ Thoát khỏi vùng giao nhau sau khi đi qua
```

Chứng minh tính hợp lý của điều độ

Yêu cầu 1 (Loại trừ lẫn nhau)**.** *Không có hai xe từ các hướng vuông góc đi qua cùng lúc.*

Chứng minh. Monitor đảm bảo rằng chỉ các xe có cùng hướng đèn xanh mới được đánh thức. Các xe hướng còn lại đều chờ trên điều kiện riêng (`condNS` hoặc `condEW`), do đó không thể giao cắt. □

Yêu cầu 2 (Tiến triển)**.** *Hệ thống luôn tiến triển, không xảy ra deadlock.*

Chứng minh. Bộ điều khiển đèn luân phiên chuyển trạng thái `lightState` theo chu kỳ thời gian. Mỗi nhóm xe đều có thời điểm được đánh thức và đi qua, nên không có tiến trình nào chờ vô hạn. □

Yêu cầu 3 (Chờ đợi hữu hạn)**.** *Mỗi xe đều có cơ hội đi qua trong thời gian hữu hạn.*

Chứng minh. Vì đèn giao thông luôn luân phiên giữa hai nhóm hướng, nên mọi xe đều được phục vụ trong giới hạn thời gian cố định của chu kỳ đèn. □

Kết quả và Thảo luận

- Hệ thống đảm bảo rằng chỉ có xe từ một cặp hướng đối diện (N–S hoặc E–W) được phép đi qua ngã tư tại một thời điểm, đảm bảo an toàn giao thông và tránh va chạm.
- Việc sử dụng Monitor kết hợp với các biến điều kiện (condition variables) giúp đồng bộ hóa việc chờ đèn và cho phép xe đi khi đèn xanh một cách hiệu quả, tránh tình trạng tranh chấp tài nguyên.
- TrafficLightController theo chu kỳ thời gian cố định đảm bảo tính tuần hoàn và công bằng giữa các hướng, không để hướng nào bị bỏ quên hay chờ đợi vô thời hạn.
- Hệ thống không xảy ra deadlock hay starvation, vì mỗi nhóm xe được đánh thức theo chu kỳ đều đặn.

- Tuy nhiên, việc chu kỳ đèn cố định có thể chưa tối ưu trong các tình huống giao thông thực tế khi lưu lượng các hướng không đều nhau; trong thực tế có thể cần mở rộng bằng cách áp dụng các thuật toán điều khiển đèn thông minh hơn dựa trên cảm biến và lưu lượng thực tế.

2.2 Readers - Writers(Bài toán Người đọc - Người ghi)

Mô tả bài toán

Cho một đối tượng dữ liệu (như một tệp tin) được chia sẻ giữa nhiều tiến trình đồng thời. Các tiến trình này được chia thành hai loại:

- **Reader (Người đọc):** Chỉ đọc dữ liệu, không sửa đổi.
- **Writer (Người ghi):** Có thể đọc hoặc sửa đổi dữ liệu.

Thiết kế thuật toán điều phối truy cập để đảm bảo tính nhất quán của dữ liệu.

Nguyên tắc

Việc truy cập vùng dữ liệu dùng chung phải tuân thủ các quy tắc sau:

1. Nhiều Reader có thể truy cập đồng thời vào vùng dữ liệu.
2. Một Writer phải có quyền truy cập **độc quyền** vào vùng dữ liệu. Khi một Writer đang truy cập, không một tiến trình nào khác (cả Reader và Writer khác) được phép truy cập.

Giải pháp 1: Ưu tiên Reader (Sử dụng Semaphore)

Giải pháp này sử dụng hai đèn báo và một biến đếm toàn cục:

- **mutex:** Một đèn báo nhị phân dùng để bảo vệ biến `readcount` (khởi tạo bằng 1).
- **wrt:** Một đèn báo dùng để đảm bảo quyền truy cập độc quyền cho Writer. Nó cũng được Reader đầu tiên sử dụng để khóa Writer (khởi tạo bằng 1).
- **readcount:** Một biến để đếm số lượng Reader hiện đang ở trong vùng găng (khởi tạo bằng 0).

Thuật toán điều độ

Algorithm 15 Thuật toán cho tiến trình Writer (Ưu tiên Reader)

```
1: while true do
2:     // Vùng không găng
3:     wait(wrt)
4:     // Đoạn găng (Ghi dữ liệu)
5:     signal(wrt)
```

Algorithm 16 Thuật toán cho tiến trình Reader (Ưu tiên Reader)

```
1:  while true do
2:      // Vùng không găng
3:      wait(mutex)
4:      readcount ← readcount + 1
5:      if readcount == 1 then                    ▷ Là Reader đầu tiên
6:          wait(wrt)                             ▷ Khóa Writer
7:      signal(mutex)
8:      // Đoạn găng (Đọc dữ liệu)
9:      wait(mutex)
10:     readcount ← readcount - 1
11:     if readcount == 0 then                    ▷ Là Reader cuối cùng
12:         signal(wrt)                           ▷ Mở khóa cho Writer
13:     signal(mutex)
```

Chứng minh tính hợp lý của điều độ

Yêu cầu 1 (Loại trừ lẫn nhau)**.** *Nếu một Writer ở trong đoạn găng, không tiến trình nào khác ở trong đoạn găng. Nếu một hoặc nhiều Reader đang ở trong đoạn găng thì Writer không được vào đoạn găng.*

Chứng minh.

- **TH1: Một Writer khác (W2) muốn vào đoạn găng trong khi có một Writer(W1) trong đoạn găng.** W2 sẽ thực hiện **wait(wrt)** (dòng 3) và bị chặn, vì W1 đã gọi wait(wrt) trước đó.
- **TH2: Một Reader (R) muốn vào đoạn găng trong khi đã có W1 trong đoạn găng.** R sẽ thực hiện **wait(mutex)** (thành công), tăng `readcount` lên 1 (dòng 4). Vì W1 đang ở trong đoạn găng nên `readcount` trước đó phải bằng 0. Do `readcount == 1`, R sẽ thực hiện **wait(wrt)** (dòng 6) và bị chặn, vì `wrt == 0`.
- **TH3: Một Writer (W) muốn vào đoạn găng trong khi đã có một hoặc nhiều Reader trong đoạn găng.** W sẽ thực hiện **wait(wrt)** (dòng 3) và bị chặn, vì `wrt` đang bị giữ bởi Reader.

Chỉ khi Reader cuối cùng (dòng 12) thực hiện **signal(wrt)**, một Writer đang chờ mới có thể vào. Rõ ràng, nhiều Reader có thể cùng ở trong đoạn găng vì họ không bị chặn bởi `wrt` (trừ người đầu tiên) và `mutex` chỉ được giữ trong thời gian rất ngắn (khi cập nhật `readcount`). □

Yêu cầu 2 (Tiến triển)**.** *Nếu tập tin sẵn sàng mà có Writter yêu cầu ghi hoặc Reader yêu cầu đọc thì phải Writter, Reader phải được đáp ứng.*

Chứng minh. Giả sử không có tiến trình nào sử dụng tài nguyên găng (`readcount == 0` và không có Writer nào). Lúc này, `wrt == 1` và `mutex == 1`.

- **TH1: Một Writer (W) đến.** W gọi **wait(wrt)** (thành công) và vào đoạn găng.
- **TH2: Một Reader (R) đến.** R gọi **wait(mutex)** (thành công), `readcount` thành 1, gọi **wait(wrt)** (thành công), gọi **signal(mutex)** và vào đoạn găng.

Nếu cả R và W cùng đến, tùy thuộc vào lịch trình, một trong hai sẽ lấy được đèn báo (hoặc `wrt` hoặc `mutex`) trước và được vào. Quyết định được đưa ra, không bị trì hoãn. □

Yêu cầu 3 (Chờ đợi hữu hạn)**.** *Một tiến trình không bị "chết đói" .*

Chứng minh. Giải pháp này **không thỏa mãn** Chờ đợi hữu hạn cho Writer.

Giả sử một Writer (W) đang chờ tại **wait(wrt)** (dòng 3). Cùng lúc đó, có một Reader (R1) đang ở trong đoạn găng.

- **TH1:** Một Reader mới (R2) đến. R2 thực hiện **wait(mutex)**, tăng `readcount` lên 2. R2 không cần gọi **wait(wrt)** vì `readcount != 1`. R2 vào đoạn găng.
- **TH2:** R1 rời khỏi đoạn găng. R1 thực hiện **wait(mutex)**, giảm `readcount` còn 1. R1 không gọi **signal(wrt)** vì `readcount != 0`.
- **TH3:** Một Reader mới (R3) đến. R3 vào đoạn găng.

Nếu các Reader mới liên tục đến trước khi Reader cuối cùng rời đi, `readcount` sẽ không bao giờ bằng 0. Do đó, lời gọi **signal(wrt)** (dòng 13) sẽ không bao giờ được thực thi, và Writer W sẽ bị "chết đói", chờ đợi vô hạn. Đây là lý do giải pháp này được gọi là "ưu tiên Reader". □

Giải pháp 2: Ưu tiên Writer (Sử dụng Semaphore)

Giải pháp này đảm bảo rằng một khi Writer báo hiệu muốn vào, nó sẽ được ưu tiên trước tất cả các Reader mới đến. Điều này ngăn chặn Writer bị "chết đói" nhưng vẫn có thể xảy ra việc Reader bị chờ đợi vô hạn.

Các đèn báo và biến sử dụng:

- `mutex_r, mutex_w`: Đèn báo nhị phân để bảo vệ 2 biến đếm (khởi tạo bằng 1).
- `readcount, writecount`: Biến đếm số Reader/Writer (khởi tạo bằng 0).
- `read_try`: Đèn báo cho phép Reader thử vào (khởi tạo bằng 1).
- `resource`: Đèn báo bảo vệ tài nguyên găng (khởi tạo bằng 1).

Thuật toán điều độ

Algorithm 17 Thuật toán cho tiến trình Writer (Ưu tiên Writer)

```
1:  while true do
2:      wait(mutex_w)
3:      writecount ← writecount + 1
4:      if writecount == 1 then                    ▷ Là Writer đầu tiên
5:          wait(read_try)                         ▷ Chặn Reader mới
6:      signal(mutex_w)
7:      wait(resource)
8:      // Đoạn găng (Ghi dữ liệu)
9:      signal(resource)
10:     wait(mutex_w)
11:     writecount ← writecount - 1
12:     if writecount == 0 then                    ▷ Là Writer cuối cùng
13:         signal(read_try)                       ▷ Cho phép Reader vào
14:     signal(mutex_w)
```

Algorithm 18 Thuật toán cho tiến trình Reader (Ưu tiên Writer)

```
1:  while true do
2:      wait(read_try)                 ▷ Kiểm tra Writer có đang chờ không
3:      wait(mutex_r)
4:      readcount ← readcount + 1
5:      if readcount == 1 then                     ▷ Là Reader đầu tiên
6:          wait(resource)                         ▷ Chặn Writer
7:      signal(mutex_r)
8:      signal(read_try)                           ▷ Cho phép Reader khác vào
9:      // Đoạn găng (Đọc dữ liệu)
10:     wait(mutex_r)
11:     readcount ← readcount - 1
12:     if readcount == 0 then                     ▷ Là Reader cuối cùng
13:         signal(resource)                       ▷ Cho phép Writer vào
14:     signal(mutex_r)
```

Giải pháp 3: Điều độ công bằng (Sử dụng Semaphore)

Giải pháp này sử dụng thêm một đèn báo `controller` có vai trò như một "hàng đợi" để đảm bảo tính công bằng giữa Reader và Writer.

Các đèn báo và biến sử dụng:

- `controller`: Đèn báo đóng vai trò như một hàng đợi chung (khởi tạo bằng 1).
- `wrt`: Một đèn báo dùng để đảm bảo quyền truy cập độc quyền cho Writer (như giải pháp 1).
- `mutex`: Đèn báo nhị phân để bảo vệ `readcount` (khởi tạo bằng 1).
- `readcount`: Biến đếm số Reader (khởi tạo bằng 0).

Thuật toán điều độ

Algorithm 19 Thuật toán cho tiến trình Writer (Công bằng)

```
1: while true do
2:     wait(controller)                          ▷ Vào hàng đợi chung
3:     wait(wrt)                                 ▷ Yêu cầu tài nguyên độc quyền
4:     signal(controller)                        ▷ Ra khỏi hàng đợi
5:     // Đoạn găng (Ghi dữ liệu)
6:     signal(wrt)                               ▷ Giải phóng tài nguyên
```

Algorithm 20 Thuật toán cho tiến trình Reader (Công bằng)

```
1: while true do
2:     wait(controller)                          ▷ Vào hàng đợi chung
3:     wait(mutex)
4:     readcount ← readcount + 1
5:     if readcount == 1 then                    ▷ Là Reader đầu tiên
6:         wait(wrt)                             ▷ Yêu cầu tài nguyên
7:     signal(mutex)
8:     signal(controller)                        ▷ Ra khỏi hàng đợi, cho phép người kế tiếp
9:     // Đoạn găng (Đọc dữ liệu)
10:    wait(mutex)
11:    readcount ← readcount - 1
12:    if readcount == 0 then                    ▷ Là Reader cuối cùng
13:        signal(wrt)                           ▷ Giải phóng tài nguyên
14:    signal(mutex)
```

Giải pháp 4: Điều độ công bằng (Sử dụng Monitor)

Giải pháp này sử dụng **Monitor**, một công cụ điều độ cấp cao. Monitor tự động cung cấp một khóa (mutex) nội tại, đảm bảo rằng chỉ một tiến trình có thể thực thi mã *bên trong* Monitor tại một thời điểm. Thay vì dùng `wait/signal` trên đèn báo, các tiến trình sử dụng các *biến điều kiện* (Condition Variables) để chờ (bằng cách tạm thời rời khỏi Monitor) và đánh thức lẫn nhau.

Mô tả Monitor

Monitor đóng gói các biến trạng thái và biến điều kiện:

- **Khóa nội tại (implicit lock):** Tự động được Monitor quản lý.
- `active_readers (int)`: Số Reader đang đọc (khởi tạo bằng 0).
- `active_writer (bool)`: `true` nếu có Writer đang ghi (khởi tạo bằng `false`).
- `waiting_readers (int)`: Số Reader đang chờ (khởi tạo bằng 0).
- `waiting_writers (int)`: Số Writer đang chờ (khởi tạo bằng 0).
- `can_read (Condition)`: Hàng đợi cho các Reader bị chặn.
- `can_write (Condition)`: Hàng đợi cho các Writer bị chặn.

Thuật toán điều độ

Monitor cung cấp 4 hàm (procedures) mà các tiến trình sẽ gọi.

Algorithm 21 Định nghĩa Monitor FairReadWrite

Require:

```
1: active_readers ← 0, active_writer ← false
2: waiting_readers ← 0, waiting_writers ← 0
3: can_read, can_write : Biến điều kiện (Condition)
4: procedure StartRead                                  ▷ Reader gọi khi muốn vào
5:                                          ▷ Lock nội tại được Monitor tự động lấy
6:     waiting_readers ← waiting_readers + 1
7:     while active_writer OR waiting_writers > 0 do
8:         can_read.wait()                              ▷ Ngủ, tự động nhả lock
9:     waiting_readers ← waiting_readers - 1
10:    active_readers ← active_readers + 1
11:                          ▷ Lock nội tại được Monitor tự động nhả khi thoát

12: procedure EndRead                                   ▷ Reader gọi khi rời đi
13:                                          ▷ Lock nội tại được tự động lấy
14:    active_readers ← active_readers - 1
15:    if active_readers == 0 then
16:        can_write.signal()                    ▷ Đánh thức 1 Writer (nếu có)
17:                                          ▷ Lock nội tại được tự động nhả

18: procedure StartWrite                                ▷ Writer gọi khi muốn vào
19:                                          ▷ Lock nội tại được tự động lấy
20:    waiting_writers ← waiting_writers + 1
21:    while active_writer OR active_readers > 0 do
22:        can_write.wait()                             ▷ Ngủ, tự động nhả lock
23:    waiting_writers ← waiting_writers - 1
24:    active_writer ← true
25:                                          ▷ Lock nội tại được tự động nhả

26: procedure EndWrite                                  ▷ Writer gọi khi rời đi
27:                                          ▷ Lock nội tại được tự động lấy
28:    active_writer ← false
29:    if waiting_writers > 0 then              ▷ Ưu tiên Writer đang chờ (FIFO)
30:        can_write.signal()
31:    else if waiting_readers > 0 then
32:        can_read.broadcast()                  ▷ Đánh thức TẤT CẢ Reader
33:                                          ▷ Lock nội tại được tự động nhả
```

Cách các tiến trình sử dụng Monitor: Các tiến trình Reader và Writer giờ đây chỉ cần gọi các hàm của Monitor.

Algorithm 22 Thuật toán cho tiến trình Writer (sử dụng Monitor)

```
1: while true do
2:     // Vùng không găng
3:     FairReadWrite.StartWrite()          ▷ Yêu cầu vào đoạn găng
4:     // Đoạn găng (Ghi dữ liệu)
5:     FairReadWrite.EndWrite()            ▷ Rời khỏi đoạn găng
```

Algorithm 23 Thuật toán cho tiến trình Reader (sử dụng Monitor)

```
1: while true do
2:     // Vùng không găng
3:     FairReadWrite.StartRead()           ▷ Yêu cầu vào đoạn găng
4:     // Đoạn găng (Đọc dữ liệu)
5:     FairReadWrite.EndRead()             ▷ Rời khỏi đoạn găng
```

2.3 The Elevator Problem (Bài toán Thang máy)

Nguồn gốc: Bài toán này được lấy ý tưởng trực tiếp từ các thuật toán lập lịch ổ đĩa (Disk Scheduling) trong hệ điều hành, cụ thể là thuật toán **SCAN**, hay còn có biệt danh là **Thuật toán Thang máy (Elevator Algorithm)**.

Mô tả bài toán

Thiết kế phần mềm điều khiển cho một thang máy trong một tòa nhà có **M** tầng để phục vụ **N** hành khách (People).

- **Thang máy (Elevator):** Một tiến trình (luồng) điều khiển duy nhất.
- **Hành khách (People):** **N** tiến trình (luồng) client, mỗi người muốn đi từ tầng 'A' đến tầng 'B'.
- **Tài nguyên găng:** Trạng thái chung của hệ thống, bao gồm:
  - Các nút bấm yêu cầu bên ngoài.
  - Các nút bấm chọn tầng bên trong.
  - Trạng thái của thang máy: tầng hiện tại, hướng đi, đang chạy, đang dừng, cửa mở/đóng.

Nguyên tắc

1. **Hành khách gọi (Bên ngoài):** Một hành khách ở tầng 'i' bấm nút 'Lên' (hoặc 'Xuống'). Tiến trình hành khách này sau đó phải **chờ** thang máy đến.
2. **Hành khách chọn (Bên trong):** Khi hành khách đã vào trong, họ bấm nút tầng 'j'.
3. **Thang máy di chuyển:** Tiến trình thang máy phải quyết định di chuyển (lên/xuống) hoặc dừng lại.

4. **Vào/Ra (Board/Unboard):** Khi thang máy dừng ở một tầng, nó phải mở cửa, **chờ** một khoảng thời gian (hoặc chờ tín hiệu) để hành khách ra (nếu đây là tầng đích) và hành khách mới vào (nếu có người đang chờ), sau đó đóng cửa.

Phân tích và đề xuất giải pháp: Thuật toán SCAN

Thang máy sẽ đi theo một hướng (giả sử đi lên), phục vụ tất cả các yêu cầu có thể trên đường đi (cả gọi bên ngoài và chọn bên trong). Chỉ khi không còn yêu cầu nào ở phía trên, nó mới đảo hướng thành xuống và bắt đầu phục vụ các yêu cầu đi xuống.

Mô tả chi tiết giải pháp (Monitor): Chúng ta sẽ thiết kế một `ElevatorMonitor` để quản lý toàn bộ logic.

- **Trạng thái (bên trong Monitor):** Toàn bộ các biến trạng thái (các mảng ‘requests[]‘, ‘current_floor‘, ‘direction‘, ‘state‘).
- **Logic điều độ (bên trong Monitor):** Thuật toán **SCAN** sẽ là logic cốt lõi mà tiến trình `ElevatorThread` thực thi.
- **Biến điều kiện (bên trong Monitor):**
  - `cv_elevator_idle`: Thang máy chờ trên biến này khi nó rảnh.
  - `cv_floor_arrival[M]`: Một mảng các CV (mỗi tầng 1 cái). Hành khách (cả bên trong và bên ngoài) sẽ chờ trên CV của tầng tương ứng.

Thuật toán điều độ

Cấu trúc Monitor: Chúng ta định nghĩa một đối tượng `ElevatorMonitor`. Tất cả các hàm `Procedure` dưới đây đều là hàm thành viên của Monitor và **tự động được khóa** khi gọi.

Algorithm 24 Định nghĩa Monitor ElevatorMonitor (Phần 1: Trạng thái & Giao diện cho Hành khách)

```
1: Monitor ElevatorMonitor {
2: // Biến Trạng thái (Nội bộ, được bảo vệ)
3: current_floor (int), direction (UP, DOWN, IDLE), state (MOVING,
   STOPPED)
4: requests_up[M], requests_down[M], requests_panel[M] (bool[])

5: // Biến Điều kiện (Nội bộ, được bảo vệ)
6: cv_elevator_idle (Condition)
7: cv_floor_arrival[M] (Condition[])

8: // Giao diện (API) cho Hành khách (Client)
9: procedure CallOutside(from_floor, to_floor)
10:    if to_floor > from_floor then
11:       requests_up[from_floor] ← true
12:    else
13:       requests_down[from_floor] ← true
14:    cv_elevator_idle.signal()          ▷ Đánh thức thang máy nếu nó rảnh

15: procedure SelectInside(to_floor)
16:    requests_panel[to_floor] ← true
17:    cv_elevator_idle.signal()

18: procedure WaitForFloor(floor)
19:    while current_floor ≠ floor OR state ≠ STOPPED do
20:       cv_floor_arrival[floor].wait()          ▷ Tự động nhả khóa  ngủ
21: // (Còn tiếp phần điều khiển thang máy ở bảng sau...)
```

Algorithm 25 Định nghĩa Monitor ElevatorMonitor (Phần 2: Logic Thang máy)

```
// (...Tiếp theo phần 1)
// Giao diện (API) cho Thang máy
function DecideAndMove returns bool (should_stop)
    while AllRequestsAreClear() do
        direction ← IDLE
        cv_elevator_idle.wait()                          ▷ Tự động nhả khóa  ngủ
    if direction == IDLE then
        direction ← UP
    if direction == UP AND NoRequestsAbove(current_floor) then
        direction ← DOWN
    else if direction == DOWN AND NoRequestsBelow(current_floor) then
        direction ← UP
    state ← MOVING
    if direction == UP then
        current_floor ← current_floor + 1
    else
        current_floor ← current_floor - 1
    if ShouldStopAt(current_floor, direction) then return true
    elsereturn false

procedure OpenDoorAndSignal
    state ← STOPPED
    requests_panel[current_floor] ← false
    if direction == UP then
        requests_up[current_floor] ← false
    else
        requests_down[current_floor] ← false
    cv_floor_arrival[current_floor].broadcast()          ▷ Đánh thức mọi người
}                                                        ▷ Kết thúc Monitor
```

Mã giả cho các tiến trình: Các tiến trình bên ngoài (Hành khách và Thang máy) giờ chỉ cần gọi các hàm của Monitor.

Algorithm 26 Thuật toán cho tiến trình Hành khách (Person)

`MyMonitor` (một thể hiện của `ElevatorMonitor`)

```
1: procedure PersonThread(from_floor, to_floor)
2:     // 1. Gọi thang máy
3:     MyMonitor.CallOutside(from_floor, to_floor)
4:
5:     // 2. Chờ thang máy đến
6:     MyMonitor.WaitForFloor(from_floor)
7:
8:     // 3. Lên tàu và bấm nút
9:     MyMonitor.SelectInside(to_floor)
10:
11:    // 4. Chờ đến tầng đích
12:    MyMonitor.WaitForFloor(to_floor)
13:
14:    // Hành khách rời đi
```

Algorithm 27 Thuật toán cho tiến trình Thang máy (Elevator)

`MyMonitor` (một thể hiện của `ElevatorMonitor`)

```
1: procedure ElevatorThread
2:     while true do
3:         should_stop ← MyMonitor.DecideAndMove()
4:         sleep(MOVE_ONE_FLOOR_DURATION)        ▷ Mô phỏng (BÊN NGOÀI MONITOR)
5:         if should_stop then
6:             MyMonitor.OpenDoorAndSignal()
7:             sleep(DOOR_OPEN_DURATION)          ▷ Mô phỏng (BÊN NGOÀi MONITOR)
```

Chứng minh tính hợp lý của điều độ

Yêu cầu 1 (Loại trừ lẫn nhau)**.** *Không được có hai tiến trình cùng lúc sửa đổi trạng thái chung.*

Chứng minh. Giải pháp này **thỏa mãn** yêu cầu. Toàn bộ giải pháp được thiết kế theo mô hình Monitor. Tất cả các hàm (ví dụ: `CallOutside`, `DecideAndMove`) đều là một phần của Monitor. Theo định nghĩa, Monitor đảm bảo chỉ một luồng có thể thực thi mã `bên trong` Monitor tại một thời điểm. Mọi hoạt động đọc/ghi tài nguyên găng đều xảy ra bên trong các hàm này, do đó chúng được bảo vệ hoàn toàn khỏi race condition. □

Yêu cầu 2 (Tiến triển)**.** *Nếu thang máy đang ở trạng thái nghỉ (IDLE) và có một yêu cầu mới được đưa ra, thang máy phải được kích hoạt để bắt đầu phục vụ yêu cầu đó.*

Chứng minh. Giải pháp này **thỏa mãn** yêu cầu. Khi không có yêu cầu nào, tiến trình Thang máy (`ElevatorThread`) sẽ gọi hàm `DecideAndMove`. Bên trong hàm này, điều kiện

`AllRequestsAreClear()` là đúng, khiến thang máy đi vào trạng thái chờ trên biến điều kiện `cv_elevator_idle.wait()` .

Khi một Hành khách đưa ra yêu cầu mới, họ sẽ gọi `CallOutside` hoặc `SelectInside`. Cả hai hàm này đều thực hiện lệnh `cv_elevator_idle.signal()`.

Tín hiệu này sẽ đánh thức tiến trình Thang máy đang chờ, giúp nó thoát khỏi vòng lặp `while` (vì `AllRequestsAreClear()` giờ là `false`) và bắt đầu di chuyển để phục vụ. Điều này đảm bảo hệ thống luôn tiến triển khi có yêu cầu. □

Yêu cầu 3 (Chờ đợi hữu hạn)**.** *Một hành khách ở một tầng không bị chờ đợi vô hạn.*

Chứng minh. Giải pháp này **thỏa mãn** yêu cầu. Thuật toán SCAN trong mã giả đảm bảo tính công bằng này. Vì thang máy di chuyển theo một chu trình quét đầy đủ (quét hết lên, rồi quét hết xuống), nó được đảm bảo sẽ ghé qua mọi tầng. Một hành khách ở tầng 2 gọi lên (gọi `CallOutside`) sẽ được phục vụ. Khi thang máy đến tầng 2 và dừng, hàm `OpenDoorAndSignal` sẽ được gọi, thực thi `cv_floor_arrival[2].broadcast()`, đánh thức hành khách đang chờ trong hàm `WaitForFloor`. Không ai bị chờ đợi vô hạn. □

References

[1] Phạm Đăng Hải, *Nguyên lí hệ điều hành.*

[2] A. S. Tanenbaum, H. Bos, *Modern Operating Systems.*

[3] A. Silberschatz, P. B. Galvin, G. G. Gagne, *Operating System Concepts.*

[4] William Stallings, *Operating Systems: Internals and Design Principles.*

[5] Leslie Lamport, A New Solution of Dijkstra's Concurrent Programming Problem.

[6] *P.J. Courtois, F. Heymans, D.L. Parnas, Concurrent control with "readers" and "writers".*

[7] Max Hailperin, *Operating Systems and Middleware.*