

CHƯƠNG 1. Giới thiệu.....	5
1.1. Hệ điều hành làm gì?.....	5
1.1.1. Quan điểm người dùng .....	5
1.1.2. Quan điểm hệ thống .....	6
1.1.3. Định nghĩa hệ điều hành .....	6
1.2. Tổ chức hệ thống máy tính .....	7
1.2.1. Ngắn .....	8
1.2.2. Cấu trúc lưu trữ .....	11
1.2.3. Bộ xử lý.....	13
1.3. Hoạt động của hệ điều hành .....	17
1.3.1. Đa chương trình và đa tác vụ .....	17
1.3.2. Chế độ kép và đa chế độ .....	18
1.3.3. Timer .....	20
1.4. Quản lí tài nguyên .....	20
1.4.1. Quản lí tiến trình .....	20
1.4.2. Quản lí bộ nhớ.....	21
1.4.3. Quản lí hệ thống file .....	22
1.4.4. Quản lí bộ nhớ chung (mass store) .....	23
1.4.5. Quản lí bộ nhớ đệm (cache).....	23
1.4.6. Quản lí hệ thống vào/ra (I/O).....	25
1.5. Tóm tắt .....	25
CHƯƠNG 2. Cấu trúc hệ điều hành.....	29
2.1. Các dịch vụ hệ điều hành .....	29
2.2. Lời gọi hệ thống.....	31
2.2.1. Ví dụ .....	31
2.2.2. Giao diện lập trình ứng dụng (API).....	33
2.2.3. Các kiểu gọi hệ thống .....	34
2.2.4. Dịch vụ hệ thống .....	35
2.2.5. Tại sao các ứng dụng lại dành riêng cho hệ điều hành .....	36
2.3. Thiết kế và cài đặt hệ điều hành .....	38
2.3.1. Mục tiêu thiết kế .....	38
2.3.2. Cơ chế và chính sách .....	38
2.3.3. Cài đặt .....	39
2.4. Cấu trúc hệ điều hành .....	39
2.4.1. Cấu trúc nguyên khối.....	39
2.4.2. Cấu trúc phân lớp.....	41

2.4.3. Cấu trúc vi nhân.....	42
2.4.4. Mô đun .....	43
2.4.5. Hệ thống hỗn hợp.....	44
2.5. Tóm tắt .....	44
CHƯƠNG 3. tiến trình và luồng .....	47
3.1. Khái niệm tiến trình .....	47
3.1.1. Tiến trình (process).....	47
3.1.2. Trạng thái của tiến trình .....	49
3.1.3. Khối điều khiển tiến trình (PCB) .....	49
3.2. Lập lịch tiến trình.....	50
3.2.1. Hàng đợi lập lịch .....	51
3.2.2. Lập lịch CPU .....	52
3.2.3. Chuyển đổi ngữ cảnh.....	53
3.3. Thao tác trên tiến trình.....	54
3.3.1. Tạo tiến trình .....	54
3.3.2. Kết thúc tiến trình.....	56
3.4. Giao tiếp giữa tiến trình .....	57
3.4.1. Cơ chế chia sẻ bộ nhớ.....	58
3.4.2. Cơ chế gởi thông điệp .....	60
3.5. Luồng .....	60
3.5.1. Giới thiệu .....	60
3.5.2. Lập trình đa lõi.....	63
3.6. Tóm tắt .....	66
CHƯƠNG 4. Lập lịch tiến trình.....	69
4.1. Các khái niệm cơ bản.....	69
4.1.1. Chu kỳ CPU và chu kỳ I/O .....	69
4.1.2. Bộ lập lịch CPU (scheduler).....	69
4.1.3. Lập lịch ưu tiên và không ưu tiên .....	70
4.1.4. Bộ điều phối (Dispatcher).....	71
4.2. Tiêu chuẩn lập lịch .....	72
4.3. Thuật toán lập lịch .....	73
4.3.1. First-In-First-Out (FIFO) Scheduling .....	73
4.3.2. Shortest-Jobs-First (SJF).....	74
4.3.3. Thuật toán Round-Robin (RR).....	76
4.3.4. Lập lịch với độ ưu tiên .....	79
4.3.5. Thuật toán Highest-Response-Ratio-Next (HRRN) .....	81

4.3.6. Lập lịch hàng đợi đa cấp.....	81
4.3.7. Lập lịch hàng đợi phản hồi đa cấp.....	83
4.4. Lập lịch đa bộ xử lý.....	85
4.4.1. Phương pháp lập lịch đa xử lý .....	85
4.4.2. Bộ xử lý đa nhân.....	86
4.4.3. Cân bằng tải.....	90
4.4.4. Liên kết bộ xử lý .....	90
4.4.5. Đa xử lý không đồng nhất .....	91
4.5. Lập lịch thời gian thực.....	92
4.5.1. Cực tiểu hóa độ trễ.....	92
4.5.2. Lập lịch dựa trên mức độ ưu tiên.....	94
4.5.3. Lập lịch đơn điệu tỉ lệ .....	95
4.5.4. Lập lịch thời hạn ngắn nhất (EDF) .....	97
4.5.5. Lập lịch theo tỉ lệ cổ phiếu .....	98
4.6. Cơ chế lập lịch trong Window .....	98
4.7. Tóm tắt .....	100
4.8. Câu hỏi ôn tập .....	101
CHƯƠNG 5. Đồng bộ hóa tiến trình.....	109
5.1. Giới thiệu.....	109
5.2. Bài toán miền găng (Critical-Section) .....	110
5.3. Thuật toán Dekker.....	112
5.3.1. Phiên bản 1 (đồng bộ hóa bước khóa và chờ khi bận) .....	113
5.3.2. Phiên bản 2 (vi phạm loại trừ lẫn nhau).....	114
5.3.3. Phiên bản 3 (tắt nghẽn).....	115
5.3.4. Phiên bản 4 (hoãn vô thời hạn).....	116
5.3.5. Thuật toán Dekker.....	118
5.4. Thuật toán Peterson.....	119
5.5. Thuật toán Lamport .....	122
5.6. Sử dụng phần cứng để đồng bộ hóa .....	125
5.6.1. Rào cản bộ nhớ.....	125
5.6.2. Các lệnh phần cứng .....	126
5.6.3. Biến nguyên thủy.....	128
5.7. Khóa Mutex .....	128
5.8. Semaphore .....	129
5.8.1. Cách sử dụng Semaphore.....	130
5.8.2. Cài đặt semaphore .....	131

5.9. Tóm tắt .....	132
5.10. Câu hỏi ôn tập.....	133
CHƯƠNG 6. Tắc nghẽn.....	135
6.1. Mô hình .....	135
6.2. Các ví dụ tắc nghẽn.....	136
6.2.1. Tắc nghẽn giao thông .....	136
6.2.2. Bài toán các nhà triết học ăn tối.....	136
6.3. Đặc tính của tắc nghẽn .....	137
6.3.1. Điều kiện xuất hiện tắc nghẽn .....	137
6.3.2. Đồ thị cấp phát tài nguyên .....	138
6.3.3. Phương pháp xử lí tắc nghẽn .....	141
6.4. Ngăn ngừa tắc nghẽn.....	141
6.4.1. Loại trừ lẫn nhau.....	141
6.4.2. Giữ và đợi .....	141
6.4.3. Không ưu tiên (độc quyền).....	142
6.4.4. Chờ vòng tròn .....	142
6.5. Tránh tắc nghẽn.....	143
6.5.1. Trạng thái an toàn .....	144
6.5.2. Thuật toán đồ thị cấp phát tài nguyên .....	145
6.5.3. Thuật toán Banker .....	147
6.6. Phát hiện tắc nghẽn.....	150
6.6.1. Kiểu một thể hiện cho mỗi loại tài nguyên.....	150
6.6.2. Kiểu nhiều thể hiện cho một loại tài nguyên.....	151
6.6.3. Cách sử dụng thuật toán phát hiện tắc nghẽn .....	153
6.7. Phục hồi trạng thái không tắc nghẽn.....	154
6.7.1. Kết thúc tiến trình và luồng .....	154
6.7.2. Nhường tài nguyên .....	155
6.8. Tóm tắt .....	155
6.9. Câu hỏi ôn tập.....	156

## CHƯƠNG 1. GIỚI THIỆU

### 1.1. Hệ điều hành làm gì?

Chúng ta bắt đầu xem xét vai trò của hệ điều hành trong hệ thống máy tính tổng thể. Một hệ thống máy tính có thể được chia thành bốn thành phần: phần cứng, hệ điều hành, chương trình ứng dụng và người dùng (Hình 1.1). Phần cứng – bộ xử lý trung tâm (CPU), bộ nhớ và thiết bị nhập/xuất (I/O) - cung cấp các tài nguyên máy tính cơ bản cho hệ thống. Các chương trình ứng dụng - chẳng hạn như trình xử lý văn bản, bảng tính, trình biên dịch và trình duyệt web - xác định cách thức sử dụng các tài nguyên này để giải quyết các vấn đề tính toán của người dùng. Hệ điều hành kiểm soát phần cứng và điều phối việc sử dụng nó giữa các chương trình ứng dụng khác nhau cho nhiều người dùng khác nhau.

Chúng ta cũng có thể xem một hệ thống máy tính bao gồm phần cứng, phần mềm và dữ liệu. Hệ điều hành cung cấp các phương tiện để sử dụng hợp lý các tài nguyên này. Hệ điều hành cung cấp một môi trường trong đó các chương trình khác có thể thực hiện công việc. Để hiểu đầy đủ hơn về vai trò của hệ điều hành, chúng ta sẽ khám phá hệ điều hành từ hai quan điểm: quan điểm của người dùng và quan điểm của hệ thống.

#### 1.1.1. Quan điểm người dùng

Quan điểm người dùng máy tính thay đổi tùy theo giao diện đang được sử dụng. Nhiều người dùng máy tính ngồi với máy tính xách tay hoặc trước PC bao gồm màn hình, bàn phím và chuột. Một hệ thống như vậy được thiết kế để một người dùng độc quyền tài nguyên của nó. Mục đích là tối đa hóa công việc (hoặc giải trí) mà người dùng đang thực hiện. Trong trường hợp này, hệ điều hành được thiết kế chủ yếu để dễ sử dụng, với một số chú ý đến hiệu suất và bảo mật và không quan tâm đến việc sử dụng tài nguyên - cách các tài nguyên phần cứng và phần mềm khác nhau được chia sẻ.

Càng ngày, nhiều người dùng càng tương tác với các thiết bị di động như điện thoại thông minh và máy tính bảng - những thiết bị đang thay thế hệ thống máy tính để bàn và máy tính xách tay đối với một số người dùng. Các thiết bị này thường được kết nối với mạng thông qua công nghệ di động hoặc không dây khác. Giao diện người dùng dành cho máy tính di động thường có màn hình cảm ứng, nơi người dùng tương tác với

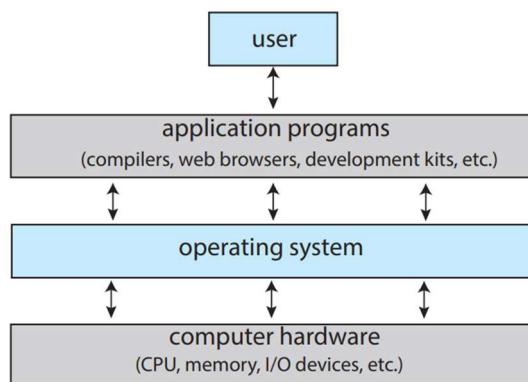
hệ thống bằng cách nhấn và vuốt ngón tay trên màn hình thay vì sử dụng bàn phím và chuột vật lý. Nhiều thiết bị di động cũng cho phép người dùng tương tác thông qua giao diện nhận dạng giọng nói, chẳng hạn như Apple's Siri.

Một số máy tính có rất ít hoặc không có chế độ xem người dùng. Ví dụ: máy tính nhúng trong thiết bị gia đình và ô tô có thể có bàn phím số và có thể bật hoặc tắt đèn báo để hiển thị trạng thái, nhưng chúng cũng như các hệ điều hành và ứng dụng của chúng được thiết kế chủ yếu để chạy mà không có sự can thiệp của người dùng.

### 1.1.2. Quan điểm hệ thống

Theo quan điểm của máy tính, hệ điều hành là chương trình liên quan mật thiết nhất đến phần cứng. Trong tình huống này, chúng ta có thể xem hệ điều hành như một bộ phân bổ tài nguyên. Hệ thống máy tính có nhiều tài nguyên có thể được yêu cầu để giải quyết một vấn đề: thời gian CPU, không gian bộ nhớ, không gian lưu trữ, thiết bị I/O, v.v. Hệ điều hành đóng vai trò là người quản lý các tài nguyên này. Đôi mặt với nhiều yêu cầu và có thể xung đột về tài nguyên, hệ điều hành phải quyết định cách phân bổ chúng cho các chương trình và người dùng cụ thể để có thể vận hành hệ thống máy tính một cách hiệu quả và công bằng.

Một quan điểm hơi khác về hệ điều hành nhấn mạnh sự cần thiết phải kiểm soát các thiết bị I/O và các chương trình người dùng khác nhau. Hệ điều hành là một chương trình điều khiển. Chương trình điều khiển quản lý việc thực thi các chương trình của người dùng để ngăn ngừa lỗi và việc sử dụng máy tính không đúng cách. Nó đặc biệt quan tâm đến hoạt động và điều khiển của các thiết bị I/O.



**Hình 1.1. Các thành phần chính của hệ thống máy tính.**

### 1.1.3. Định nghĩa hệ điều hành

Thuật ngữ hệ điều hành bao gồm nhiều vai trò và chức năng. Máy tính có mặt trong lò nướng bánh mì, ô tô, tàu thủy, tàu vũ trụ, gia đình và doanh nghiệp. Chúng là cơ sở cho máy trò chơi, bộ thu sóng truyền hình cáp và hệ thống điều khiển công nghiệp.

Để giải thích sự đa dạng này, chúng ta có thể tìm hiểu lịch sử của máy tính. Máy tính bắt đầu như một thử nghiệm để xác định những gì có thể được thực hiện và nhanh chóng được chuyển sang các hệ thống có mục đích cố định cho mục đích quân sự, chẳng hạn như phá mã và vẽ biểu đồ quỹ đạo, cũng như các mục đích sử dụng của chính phủ, chẳng hạn như tính toán điều tra dân số. Những máy tính ban đầu đó đã phát triển thành các máy tính lớn, đa năng, và đó là khi hệ điều hành ra đời. Vào những năm 1960, định luật Moore đã dự đoán rằng số lượng bóng bán dẫn trên một mạch tích hợp sẽ tăng gấp đôi sau mỗi 18 tháng và dự đoán đó đã đúng. Máy tính được nâng cấp về chức năng và thu nhỏ về kích thước, dẫn đến việc sử dụng rất nhiều và một số lượng lớn và đa dạng các hệ điều hành.

Nói chung, chúng ta không có định nghĩa hoàn toàn đầy đủ về hệ điều hành. Hệ điều hành tồn tại bởi vì chúng đưa ra một cách hợp lý để giải quyết vấn đề tạo ra một hệ thống máy tính có thể sử dụng được. Mục tiêu cơ bản của hệ thống máy tính là thực thi các chương trình và giúp giải quyết các vấn đề của người dùng dễ dàng hơn. Phần cứng máy tính được xây dựng để hướng tới mục tiêu này. Vì phần cứng không dễ sử dụng, nên các chương trình ứng dụng được phát triển. Các chương trình này yêu cầu một số hoạt động phổ biến, chẳng hạn như các hoạt động điều khiển thiết bị I/O. Các chức năng chung của việc kiểm soát và phân bổ tài nguyên sau đó được tập hợp lại thành một phần mềm: hệ điều hành.

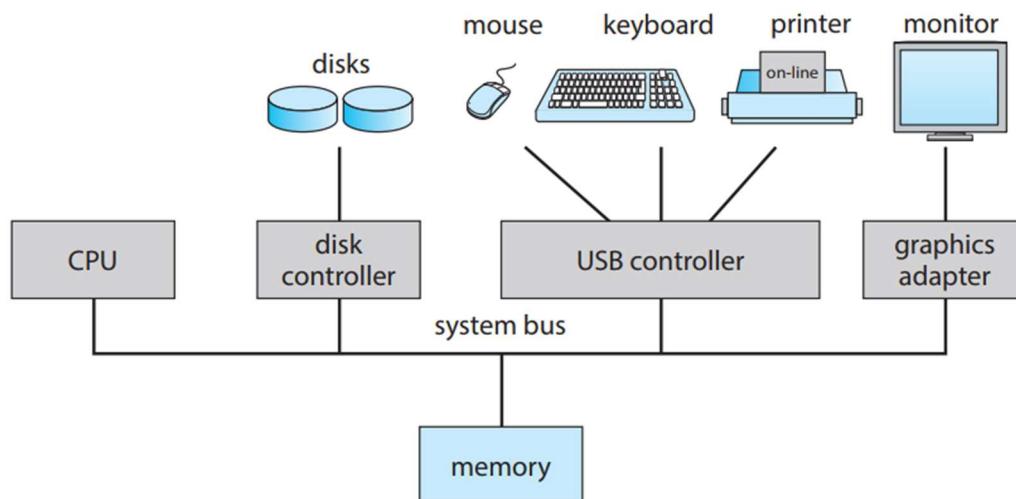
Một định nghĩa phổ biến hơn, đó là hệ điều hành là một chương trình chạy mọi lúc trên máy tính - thường được gọi là nhân. Cùng với nhân, có hai loại chương trình khác: chương trình hệ thống, được liên kết với hệ điều hành nhưng không nhất thiết là một phần của hạt nhân và chương trình ứng dụng, bao gồm tất cả các chương trình không liên quan đến hoạt động của hệ thống.

Tóm lại, hệ điều hành bao gồm nhân luôn chạy, các phần mềm trung gian giúp dễ dàng phát triển ứng dụng, các chương trình hỗ trợ quản lý hệ thống khi nó đang chạy. Phần lớn nội dung này liên quan đến nhân của các hệ điều hành đa năng, nhưng các thành phần khác sẽ được thảo luận khi cần thiết để giải thích đầy đủ về thiết kế và hoạt động của hệ điều hành.

## 1.2. Tổ chức hệ thống máy tính

Một hệ thống máy tính đa năng hiện đại bao gồm một hoặc nhiều CPU và một số bộ điều khiển (Controller) thiết bị được kết nối thông qua một bus chung cung cấp quyền truy cập giữa các thành phần và bộ nhớ dùng chung (Hình 1.2). Mỗi bộ điều khiển thiết bị phụ trách một loại thiết bị cụ thể (ví dụ: ổ đĩa, thiết bị âm thanh hoặc màn hình đồ

hoa). Tùy thuộc vào bộ điều khiển, nhiều thiết bị có thể được gắn vào. Ví dụ, một cổng USB của hệ thống có thể kết nối với một bộ chia USB, một số thiết bị có thể kết nối với nhau. Bộ điều khiển thiết bị duy trì một số lưu trữ bộ đệm cục bộ và một tập hợp các thanh ghi có mục đích đặc biệt. Bộ điều khiển thiết bị chịu trách nhiệm di chuyển dữ liệu giữa các thiết bị ngoại vi mà nó điều khiển và bộ nhớ đệm cục bộ của nó.



**Hình 1.2. Kiến trúc điển hình của một hệ thống máy tính.**

Thông thường, các hệ điều hành có trình điều khiển thiết bị cho mỗi thiết bị. Trình điều khiển thiết bị này hiểu thiết bị và cung cấp cho hệ điều hành một giao diện thống nhất cho các thiết bị. CPU và bộ điều khiển thiết bị có thể thực thi song song, cạnh tranh nhau về bộ nhớ. Để đảm bảo truy cập có trật tự vào bộ nhớ dùng chung, bộ điều khiển bộ nhớ sẽ đồng bộ hóa quyền truy cập vào bộ nhớ.

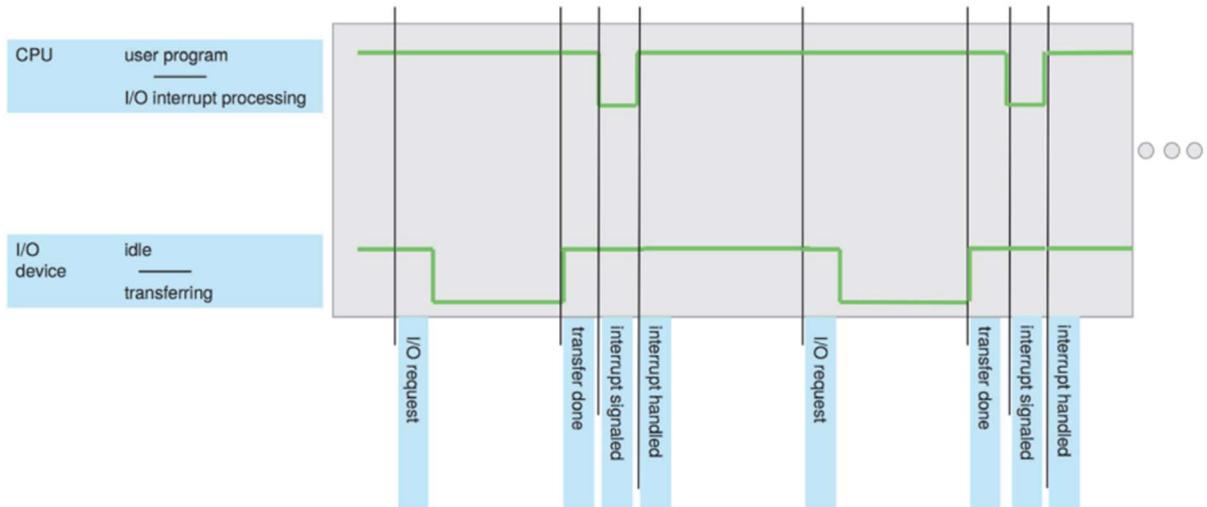
### 1.2.1. Ngắt

Hãy xem xét một hoạt động máy tính điển hình: một chương trình thực hiện I/O. Để bắt đầu thao tác I/O, driver tải các thanh ghi thích hợp trong controller. Đến lượt mình, controller sẽ kiểm tra nội dung của các thanh ghi này để xác định hành động cần thực hiện (chẳng hạn như “đọc một ký tự từ bàn phím”). Controller bắt đầu chuyển dữ liệu từ thiết bị đến bộ đệm cục bộ của nó. Khi quá trình chuyển dữ liệu hoàn tất, controller sẽ thông báo cho driver rằng nó đã kết thúc hoạt động. Nhưng làm thế nào để controller thông báo cho driver rằng nó đã kết thúc hoạt động? Điều này được thực hiện thông qua một ngắt.

#### a. Tổng quan về ngắt

Phần cứng có thể kích hoạt ngắt bất kỳ lúc nào bằng cách gửi tín hiệu đến CPU, thường là bằng đường bus hệ thống. Ngắt cũng được sử dụng cho nhiều mục đích khác và là một phần quan trọng trong cách hệ điều hành và phần cứng tương tác.

Khi CPU bị ngắt, nó sẽ dừng những gì nó đang làm và ngay lập tức chuyển việc thực thi đến một vị trí cố định. Vị trí cố định thường chứa địa chỉ bắt đầu nơi đặt tiến trình dịch vụ cho ngắt. Tiến trình dịch vụ ngắt thực thi; khi hoàn thành, CPU tiếp tục thực hiện công việc trước ngắt. Lịch trình của hoạt động này được thể hiện ở Hình 1.3.

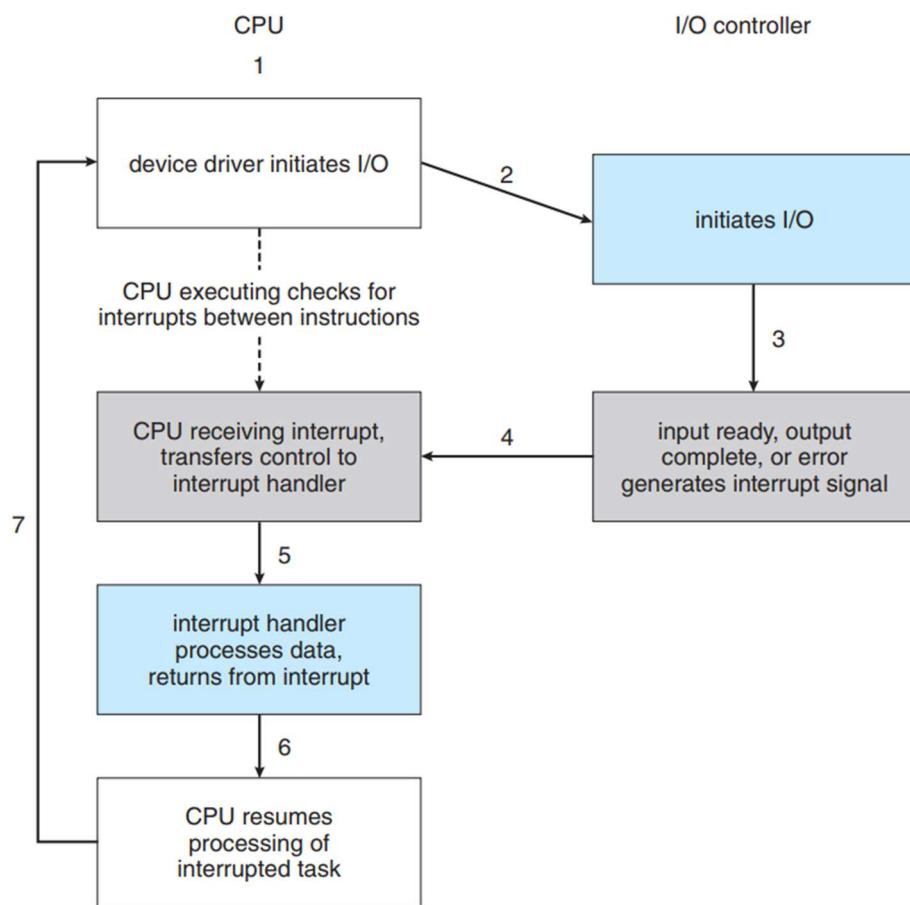


**Hình 1.3.** Mô hình thời gian ngắt cho một chương trình duy nhất đang thực hiện đầu ra.

Ngắt là một phần quan trọng của kiến trúc máy tính. Mỗi thiết kế máy tính có cơ chế ngắt riêng, nhưng một số chức năng là chung. Ngắt phải chuyển quyền điều khiển sang tiến trình phục vụ ngắt thích hợp. Phương pháp đơn giản để quản lý việc chuyển giao này là gọi một tiến trình chung để kiểm tra thông tin ngắt và gọi trình xử lý ngắt cụ thể. Tuy nhiên, các ngắt phải được xử lý nhanh, vì chúng xảy ra rất thường xuyên. Thay vào đó, sử dụng một bảng con trả trỏ đến các tiến trình để tăng tốc. Tiến trình ngắt gọi gián tiếp thông qua bảng này, không cần tiến trình trung gian. Các hệ điều hành khác nhau như Windows và UNIX điều phối ngắt theo cách này.

### b. Cài đặt

Cơ chế ngắt cơ bản hoạt động như sau. Phần cứng CPU có một dây gọi là dòng yêu cầu ngắt mà CPU nhận được. Khi CPU phát hiện thấy Controller phát tín hiệu trên dòng yêu cầu ngắt, nó sẽ đọc số ngắt và chuyển đến tiến trình xử lý ngắt bằng cách sử dụng số ngắt đó làm chỉ số vào vectơ ngắt. Sau đó, nó bắt đầu thực thi tại địa chỉ được liên kết với chỉ mục đó. Trình xử lý ngắt lưu bất kỳ trạng thái nào mà nó sẽ thay đổi trong quá trình hoạt động, xác định nguyên nhân của ngắt, thực hiện xử lý cần thiết, thực hiện khôi phục trạng thái và quay trở lại từ lệnh ngắt để đưa CPU về trạng thái thực thi trước khi ngắt. Hình 1.4 tóm tắt chu kì I/O điều khiển ngắt.

**Hình 1.4.** Chu kì điều khiển ngắt.

Cơ chế ngắt cơ bản vừa mô tả cho phép CPU phản hồi với sự kiện không đồng bộ, như khi bộ điều khiển thiết bị sẵn sàng hoạt động. Tuy nhiên, trong một hệ điều hành hiện đại, chúng ta cần các tính năng xử lý ngắt phức tạp hơn.

Hầu hết các CPU đều có hai dòng yêu cầu ngắt. Một là ngắt không che được, dành riêng cho các sự kiện như lỗi bộ nhớ không thể khôi phục. Dòng ngắt thứ hai có thể che được: nó có thể được CPU tắt trước khi thực hiện các chuỗi lệnh quan trọng mà không được ngắt. Ngắt có thể che được được sử dụng bởi controllers để yêu cầu dịch vụ.

Mục đích của vecto ngắt là giảm thời gian một trình xử lý ngắt tìm kiếm tất cả các nguồn ngắt để xác định cái nào cần thực hiện. Tuy nhiên, trong thực tế, máy tính có nhiều thiết bị (và do đó, trình xử lý ngắt) hơn là chúng có các phần tử trong vecto ngắt. Một cách phổ biến để giải quyết vấn đề này là sử dụng chuỗi ngắt, trong đó mỗi phần tử trong vecto ngắt trỏ đến phần đầu của danh sách các trình xử lý ngắt. Khi một ngắt được đưa ra, các trình xử lý trong danh sách tương ứng được gọi lần lượt cho đến khi tìm thấy một ngắt có thể phục vụ yêu cầu. Cấu trúc này là sự cân bằng giữa chi phí của một bảng ngắt khổng lồ và sự kém hiệu quả của việc điều phối đến một trình xử lý ngắt. Bảng 1.1 minh họa thiết kế của vector ngắt cho bộ xử lý Intel. Các sự kiện từ 0 đến 31, không thể

che, được sử dụng để báo hiệu các điều kiện lỗi khác nhau. Các sự kiện từ 32 đến 255, có thể che được, được sử dụng cho các mục đích như ngắt do thiết bị tạo.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

**Bảng 1.1.** Bảng vecto ngắt của Intel.

Tóm lại, ngắt được sử dụng trong khắp các hệ điều hành hiện đại để xử lý các sự kiện không đồng bộ (và cho các mục đích khác). Controller và lỗi phần cứng gây ra ngắt, để cho phép thực hiện công việc khẩn cấp. Các máy tính hiện đại sử dụng một hệ thống ưu tiên ngắt. Bởi vì ngắt được sử dụng rất nhiều cho quá trình xử lý nhẹ về thời gian, cần phải xử lý ngắt hiệu quả để hệ thống hoạt động tốt.

### 1.2.2. Cấu trúc lưu trữ

CPU chỉ có thể thực hiện các lệnh từ bộ nhớ, vì vậy bất kỳ chương trình nào trước tiên phải được tải vào bộ nhớ để chạy. Máy tính đa năng chạy hầu hết các chương trình của chúng từ bộ nhớ có thể ghi lại, được gọi là bộ nhớ chính (còn được gọi là bộ nhớ truy cập ngẫu nhiên, hoặc RAM). Bộ nhớ chính thường được sản xuất bằng công nghệ bán dẫn được gọi là bộ nhớ truy cập ngẫu nhiên động (DRAM).

Máy tính cũng sử dụng các dạng bộ nhớ khác. Ví dụ: chương trình đầu tiên chạy trên máy tính khi bật nguồn là chương trình bootstrap, chương trình này sau đó sẽ tải hệ

điều hành. Vì RAM dễ “bay hơi-volatile” - nội dung của nó bị mất khi tắt nguồn - chúng ta không thể lưu chương trình bootstrap. Thay vào đó, máy tính sử dụng bộ nhớ chỉ đọc có thể lập trình và có thể xóa bằng điện (electrically erasable programmable read-only memory - EEPROM). EEPROM có thể được thay đổi nhưng không thể thay đổi thường xuyên. Ngoài ra, nó có tốc độ thấp nên chủ yếu chứa các chương trình tĩnh và dữ liệu không được sử dụng thường xuyên. Ví dụ, iPhone sử dụng EEPROM để lưu trữ số sê-ri và thông tin phần cứng về thiết bị.

Tất cả các dạng bộ nhớ đều cung cấp một mảng byte. Mỗi byte có địa chỉ riêng của nó. Thao tác chủ yếu là các load hoặc store đến các địa chỉ bộ nhớ cụ thể. Lệnh load di chuyển một byte từ bộ nhớ chính sang một thanh ghi bên trong CPU, trong khi lệnh store di chuyển nội dung của một thanh ghi vào bộ nhớ chính. Bên cạnh việc load và store, CPU sẽ tự động tải các lệnh từ bộ nhớ chính để thực thi từ vị trí được lưu trữ trong bộ đệm chương trình.

Lý tưởng nhất là chúng ta muốn các chương trình và dữ liệu thường trú trong bộ nhớ chính. Điều này thường không thể thực hiện được trên hầu hết các hệ thống vì hai lý do:

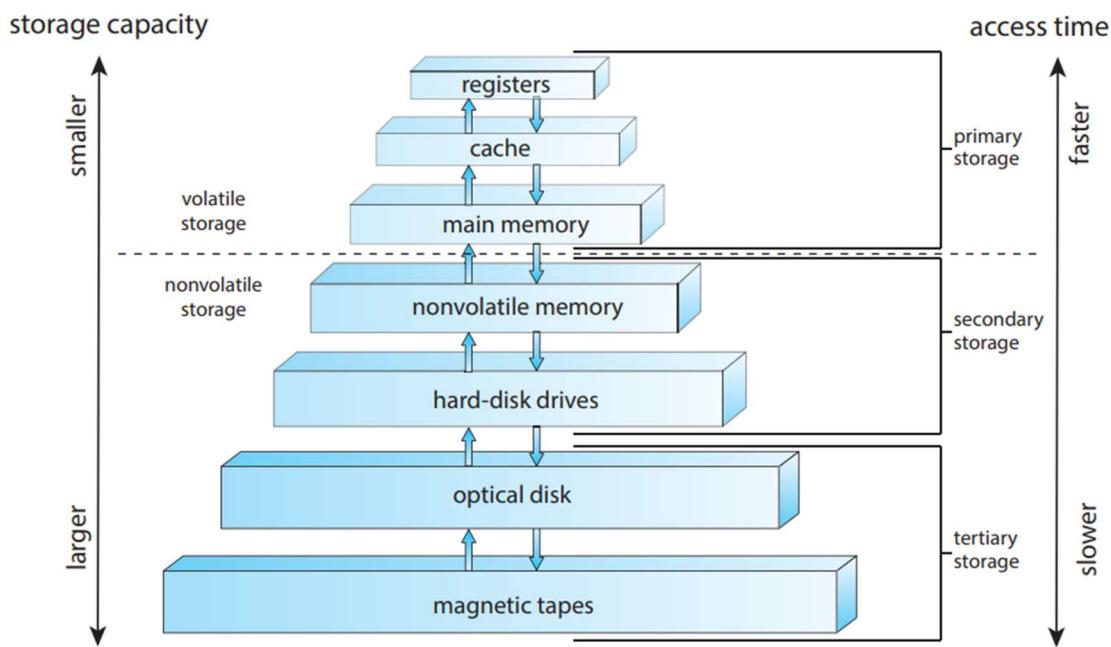
1. Bộ nhớ chính thường quá nhỏ để lưu trữ vĩnh viễn tất cả các chương trình và dữ liệu.
2. Bộ nhớ chính rất dễ bay hơi - nó mất nội dung khi tắt nguồn.

Vì vậy, hầu hết các hệ thống máy tính cung cấp bộ nhớ phụ (còn gọi là bộ nhớ thứ cấp) như một phần mở rộng của bộ nhớ chính. Yêu cầu chính đối với bộ nhớ thứ cấp là nó có thể lưu trữ số lượng lớn dữ liệu vĩnh viễn.

Các thiết bị lưu trữ thứ cấp phổ biến nhất là ổ đĩa cứng (HDD) và thiết bị bộ nhớ không bay hơi (Nonvolatile), cung cấp khả năng lưu trữ cho cả chương trình và dữ liệu. Hầu hết các chương trình (hệ thống và ứng dụng) được lưu trữ trong bộ nhớ thứ cấp cho đến khi chúng được tải vào bộ nhớ chính. Bộ nhớ phụ cũng chậm hơn nhiều so với bộ nhớ chính. Do đó, việc quản lý bộ nhớ thứ cấp cũng có tầm quan trọng đối với hệ thống máy tính.

Các thanh ghi, bộ nhớ chính và bộ nhớ phụ chỉ là một trong nhiều kiểu lưu trữ. Các kiểu khác có thể bao gồm bộ nhớ đệm, CD-ROM hoặc blu-ray, băng từ, v.v. Những thiết bị này chậm nhưng đủ lớn để sử dụng cho các mục đích khác (lưu trữ các bản sao dự phòng của tài liệu). Mỗi kiểu lưu trữ cung cấp các chức năng cơ bản để lưu trữ dữ liệu. Sự khác biệt chính giữa các kiểu lưu trữ là tốc độ, kích thước và tính “bay hơi”.

Nhiều kiểu lưu trữ có thể được tổ chức theo thứ bậc (Hình 1.5) tùy theo dung lượng lưu trữ và thời gian truy cập. Theo nguyên tắc chung, có sự cân bằng giữa kích thước và tốc độ, với bộ nhớ nhỏ hơn và nhanh hơn gần CPU hơn. Như thể hiện trong hình, ngoài việc khác nhau về tốc độ và dung lượng, các hệ kiểu lưu trữ khác nhau “bay hơi” hoặc “không bay hơi”.



**Hình 1.5. Các kiểu thiết bị lưu trữ.**

### 1.2.3. Bộ xử lý

Trước khi vào chi tiết ta tìm hiểu vài định nghĩa liên quan:

- CPU: Phần cứng thực thi các lệnh.
- Processor: Một chip vật lý có chứa một hoặc nhiều CPU.
- Core: Đơn vị tính toán cơ bản của CPU.
- Multicore: Bao gồm nhiều lõi điện toán trên cùng một CPU.
- Multiprocessor: Bao gồm nhiều bộ xử lý.

#### a. Hệ thống đơn bộ xử lý

Nhiều năm trước, hầu hết các hệ thống máy tính đều sử dụng một bộ xử lý chứa một CPU với một lõi xử lý. Phần lõi là thành phần thực thi các lệnh và các thanh ghi lưu trữ dữ liệu cục bộ. Một CPU chính với lõi của nó có khả năng thực hiện một tập lệnh có mục đích chung, kể cả các lệnh từ các tiến trình. Các hệ thống này cũng có các bộ xử lý có mục đích riêng khác. Chúng có thể ở dạng bộ xử lý dành riêng cho thiết bị, chẳng hạn như đĩa, bàn phím và bộ điều khiển đồ họa.

Tất cả các bộ xử lý có mục đích riêng này đều chạy một tập lệnh hạn chế và không chạy các tiến trình. Đôi khi, chúng được quản lý bởi hệ điều hành, trong đó hệ điều hành gửi cho chúng thông tin về tác vụ tiếp theo và giám sát trạng thái của chúng. Ví dụ, một bộ vi xử lý điều khiển nhận một chuỗi các yêu cầu từ lõi CPU chính và bổ sung vào hàng đợi của nó và thực hiện thuật toán lập. Sự bố trí này giải phóng CPU chính khỏi lập lịch đĩa. PC có chứa một bộ vi xử lý trong bàn phím để chuyển các lần gõ phím thành mã để gửi đến CPU. Trong các hệ thống hoặc trường hợp khác, bộ xử lý có mục đích riêng là các thành phần cấp thấp được tích hợp trong phân cứng. Hệ điều hành không thể giao tiếp với các bộ xử lý này; nó làm công việc một cách tự chủ. Việc sử dụng các bộ vi xử lý có mục riêng là phổ biến và không biến một hệ thống đơn bộ xử lý thành đa bộ xử lý. Nếu chỉ có một CPU đa năng với một lõi xử lý duy nhất, thì hệ thống là hệ thống đơn bộ xử lý. Tuy nhiên, theo định nghĩa này, rất ít hệ thống máy tính hiện nay là hệ thống đơn bộ xử lý.

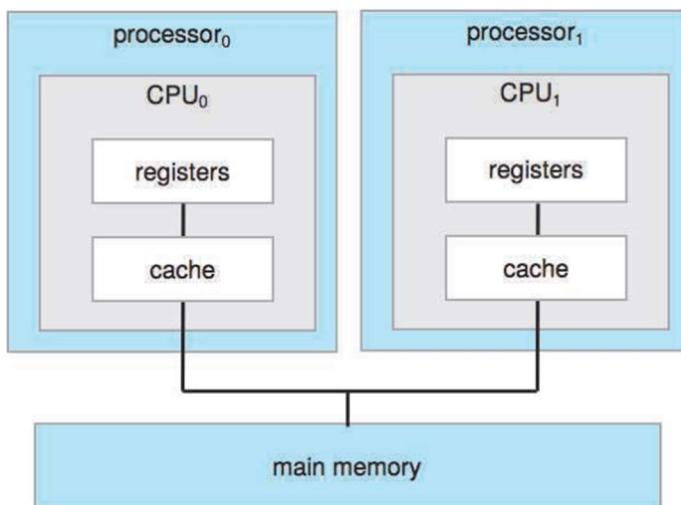
#### b. Hệ thống đa bộ xử lý

Trên các máy tính hiện đại, từ thiết bị di động đến máy chủ, thường là các hệ thống đa bộ xử lý. Các hệ thống như vậy có hai (hoặc nhiều) bộ xử lý, mỗi bộ có một CPU lõi đơn. Các bộ xử lý chia sẻ bus máy tính và đôi khi là đồng hồ, bộ nhớ và các thiết bị ngoại vi. **Ưu điểm chính** của hệ thống đa bộ xử lý là tăng thông lượng. Tức là, bằng cách tăng số lượng bộ xử lý, chúng ta hy vọng sẽ hoàn thành nhiều công việc hơn trong thời gian ngắn hơn. Tuy nhiên, tỷ lệ tăng tốc với  $N$  bộ xử lý không phải là  $N$ ; nhỏ hơn  $N$ . Khi nhiều bộ xử lý hợp tác trong một nhiệm vụ, một lượng chi phí nhất định sẽ phát sinh để giữ cho tất cả các bộ phận hoạt động chính xác. Chi phí này, cộng với sự tranh giành tài nguyên được chia sẻ, làm giảm lợi ích mong đợi từ các bộ xử lý bổ sung.

Các hệ thống đa bộ xử lý phổ biến nhất sử dụng đa xử lý đối xứng (Symmetric Multiprocessing-SMP), trong đó mỗi bộ xử lý CPU ngang hàng thực hiện tất cả các tác vụ, bao gồm các chức năng của hệ điều hành và các tiến trình người dùng. Hình 1.6 minh họa một kiến trúc SMP điển hình với hai bộ xử lý, mỗi bộ có một CPU riêng. Lưu ý rằng mỗi bộ xử lý CPU có một bộ thanh ghi riêng, một bộ nhớ đệm riêng (cục bộ). Tuy nhiên, tất cả các bộ xử lý đều chia sẻ bộ nhớ vật lý qua bus hệ thống.

Lợi ích của mô hình này là nhiều tiến trình có thể chạy đồng thời -  $N$  tiến trình có thể chạy nếu có  $N$  CPU - mà không làm giảm hiệu suất. Tuy nhiên, vì các CPU riêng biệt nên một CPU có thể không hoạt động trong khi một CPU khác bị quá tải, dẫn đến hoạt động kém hiệu quả. Việc kém hiệu quả này có thể tránh được nếu các bộ xử lý chia sẻ một số cấu trúc dữ liệu nhất định. Một hệ thống đa xử lý dạng này sẽ cho phép các tiến trình và tài nguyên - chẳng hạn như bộ nhớ - được chia sẻ động giữa các bộ xử lý

khác nhau và có thể giám bót sự chênh lệch khói lượng công việc giữa các bộ xử lý.



**Hình 1.6.** Kiến trúc đa xử lý đối xứng.

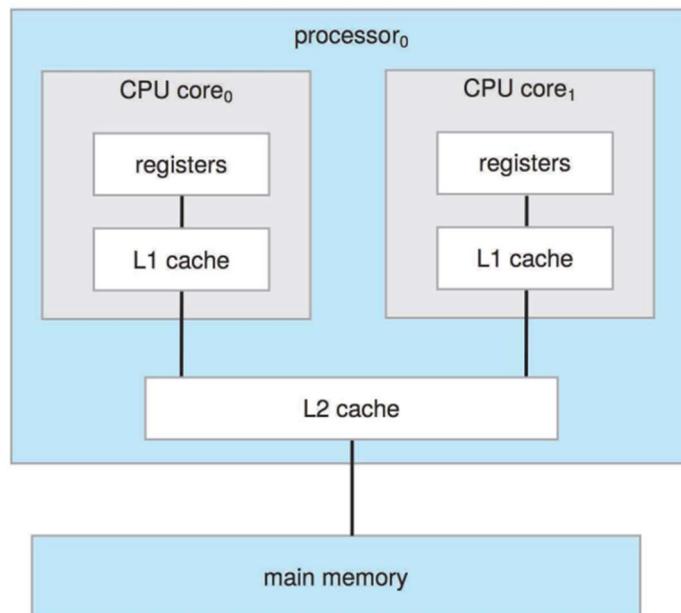
Khái niệm đa bộ xử lý đã bao gồm các hệ thống đa lõi, trong đó nhiều lõi tính toán nằm trên một chip duy nhất. Hệ thống đa lõi có thể hiệu quả hơn đa chip với lõi đơn vì giao tiếp trên cùng chip nhanh hơn giao tiếp giữa các chip. Ngoài ra, một chip với nhiều lõi sử dụng ít năng lượng hơn so với nhiều chip trên lõi đơn, một vấn đề quan trọng đối với các thiết bị di động cũng như máy tính xách tay.

Trong Hình 1.7, chúng ta thấy một thiết kế lõi kép với hai lõi trên cùng một chip xử lý. Trong thiết kế này, mỗi lõi có bộ thanh ghi riêng, bộ đệm cục bộ riêng, thường được gọi là bộ đệm cấp 1 (Cache L1). Cũng lưu ý rằng bộ nhớ đệm cấp 2 (Cache L2) là cục bộ của chip nhưng được chia sẻ bởi hai lõi. Hầu hết các kiến trúc áp dụng cách tiếp cận này, kết hợp bộ nhớ đệm cục bộ và bộ nhớ đệm chia sẻ, trong đó bộ nhớ đệm cục bộ, bộ nhớ đệm cấp thấp hơn thường nhỏ hơn và nhanh hơn bộ nhớ đệm chia sẻ cấp cao hơn. Bên cạnh những cân nhắc về kiến trúc, chẳng hạn như bộ nhớ đệm, bộ nhớ chính và tranh chấp bus, một bộ xử lý đa lõi với N lõi xuất hiện trong hệ điều hành dưới dạng N CPU tiêu chuẩn. Đặc điểm này gây áp lực lên các nhà thiết kế hệ điều hành - và các nhà lập trình ứng dụng - để sử dụng hiệu quả các lõi này. Hầu như tất cả các hệ điều hành hiện đại - bao gồm Windows, macOS và Linux, cũng như Android và iOS - hỗ trợ hệ thống SMP đa lõi.

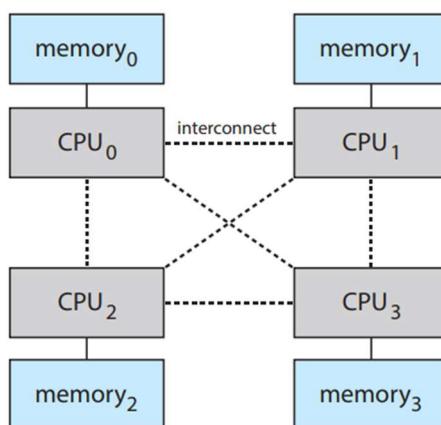
Việc bổ sung CPU vào hệ thống đa xử lý sẽ tăng sức mạnh tính toán; tuy nhiên, khi chúng ta thêm quá nhiều CPU, sự tranh giành bus hệ thống sẽ trở thành một nút thắt cổ chai và hiệu suất bắt đầu suy giảm. Một cách tiếp cận khác là cung cấp cho mỗi CPU (hoặc nhóm CPU) bộ nhớ cục bộ riêng được truy cập thông qua một bus cục bộ nhỏ, nhanh. Các CPU được kết nối bằng hệ thống kết nối dùng chung, do đó tất cả các CPU

đều chia sẻ một không gian địa chỉ vật lý. Cách tiếp cận này - được gọi là truy cập bộ nhớ không đồng nhất (non-uniform memory access - NUMA) - được minh họa trong Hình 1.8. Ưu điểm là, khi một CPU truy cập bộ nhớ cục bộ của nó, nó không chỉ nhanh mà còn không có tranh chấp về kết nối hệ thống. Do đó, các hệ thống NUMA có thể mở rộng quy mô hiệu quả hơn khi nhiều bộ xử lý được thêm vào.

Hạn chế của hệ thống NUMA là độ trễ tăng lên khi CPU phải truy cập bộ nhớ từ xa qua kết nối hệ thống, giảm hiệu suất. Nói cách khác, ví dụ, CPU<sub>0</sub> không thể truy cập bộ nhớ cục bộ của CPU<sub>3</sub> nhanh như nó có thể truy cập bộ nhớ cục bộ của chính nó, làm chậm hiệu suất. Hệ điều hành có thể giảm thiểu hạn chế của NUMA thông qua việc lập lịch CPU và quản lý bộ nhớ. Bởi vì các hệ thống NUMA có thể mở rộng quy mô để chứa một số lượng lớn bộ xử lý, chúng ngày càng trở nên phổ biến trên các máy chủ cũng như các hệ thống tính toán hiệu suất cao.



**Hình 1.7.** CPU lõi kép trên cùng một chip.



**Hình 1.8.** Kiến trúc đa xử lý NUMA.

### 1.3. Hoạt động của hệ điều hành

Để một máy tính bắt đầu chạy - ví dụ, khi nó được khởi động hoặc khởi động lại - nó cần phải có một chương trình ban đầu để chạy. Chương trình ban đầu này (bootstrap) thì đơn giản. Thông thường, nó được lưu trữ trong phần cứng máy tính dưới dạng firmware. Nó khởi tạo tất cả phần của hệ điều hành, từ thanh ghi CPU đến bộ điều khiển thiết bị đến nội dung bộ nhớ. Chương trình bootstrap phải biết cách tải hệ điều hành và bắt đầu thực thi hệ thống đó. Để thực hiện điều này, chương trình bootstrap phải định vị nhân của hệ điều hành (kernel) và tải nó vào bộ nhớ.

Khi nhân được tải và thực thi, nó có thể bắt đầu cung cấp các dịch vụ cho hệ thống và người dùng. Một số dịch vụ được bên ngoài nhân cũng được tải vào bộ nhớ tại thời điểm khởi động để trở thành trình nền hệ thống, chạy trong toàn bộ thời gian nhân đang chạy. Trên Linux, chương trình hệ thống đầu tiên là “systemd” và nó khởi động nhiều daemon khác. Khi giai đoạn này hoàn tất, hệ thống được khởi động hoàn toàn và sẽ đợi một số sự kiện xảy ra.

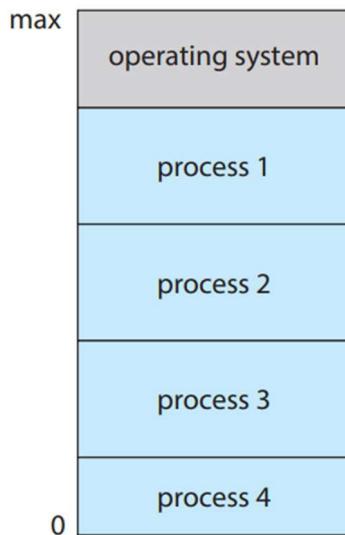
Nếu không có tiến trình nào để thực thi, không có thiết bị I/O nào để phục vụ và không có người dùng nào để phản hồi, hệ điều hành sẽ ngồi yên lặng, chờ đợi sự kiện gì đó xảy ra. Các sự kiện hầu như luôn được báo hiệu bằng sự xuất hiện của một ngắt. Một dạng khác của ngắt là một bẫy (hoặc một ngoại lệ), là một ngắt do phần mềm tạo ra do lỗi (ví dụ: chia cho 0 hoặc truy cập bộ nhớ không hợp lệ) hoặc bởi một yêu cầu cụ thể từ một chương trình người dùng mà một dịch vụ hệ điều hành được thực hiện bằng cách thực hiện một thao tác đặc biệt gọi là lệnh gọi hệ thống.

#### 1.3.1. Đa chương trình và đa tác vụ

Một trong những khía cạnh quan trọng nhất của hệ điều hành là khả năng chạy nhiều chương trình, vì một chương trình đơn lẻ nói chung không thể giữ cho CPU hoặc các thiết bị I/O luôn bận rộn. Hơn nữa, người dùng thường muốn chạy nhiều chương trình cùng một lúc. Đa chương trình làm tăng khả năng sử dụng CPU, cũng như giữ cho người dùng hài lòng, bằng cách tổ chức các chương trình sao cho CPU luôn có một chương trình để thực thi. Trong một hệ thống đa chương trình, một chương trình đang được thực thi được gọi là một tiến trình (process).

Ý tưởng như sau: Hệ điều hành giữ một số tiến trình đồng thời trong bộ nhớ (Hình 1.9). Hệ điều hành chọn và bắt đầu thực hiện một trong các tiến trình này. Khi tiến trình đợi một số tác vụ, chẳng hạn như hoạt động I/O, để hoàn thành. Trong một hệ thống không đa chương trình, CPU sẽ không hoạt động. Trong một hệ thống đa chương trình, hệ điều hành chỉ cần chuyển sang và thực thi một tiến trình khác. Khi tiến trình đó cần

chờ, CPU sẽ chuyển sang một tiến trình khác, v.v. Cuối cùng, tiến trình đầu tiên kết thúc và chờ CPU trở lại. Miễn là ít nhất một tiến trình cần thực thi, CPU sẽ không bao giờ rảnh.



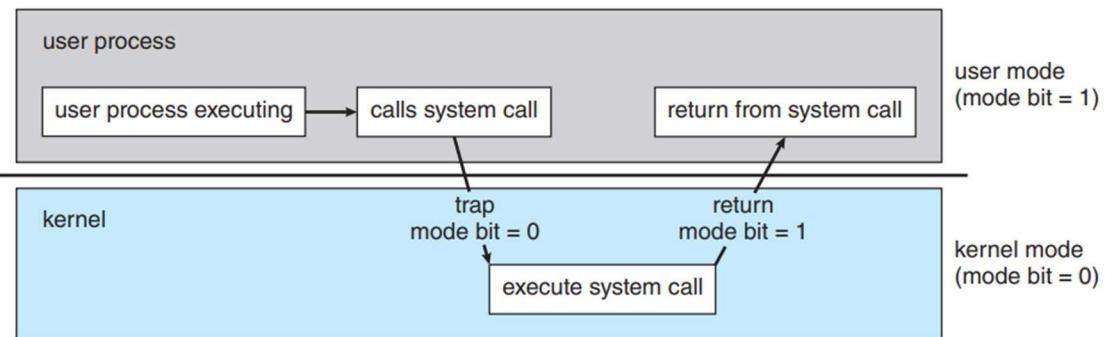
**Hình 1.9.** Mô phỏng bộ nhớ của hệ thống đa chương trình.

Đa nhiệm là khái niệm phần mở rộng của đa chương trình. Trong các hệ thống đa nhiệm, CPU thực hiện nhiều tiến trình bằng cách chuyển đổi giữa chúng, nhưng việc chuyển đổi xảy ra thường xuyên, cung cấp cho người dùng thời gian phản hồi nhanh. Hãy xét khi một tiến trình thực thi, nó thường chỉ thực thi trong một thời gian ngắn trước khi kết thúc hoặc cần thực hiện I/O. I/O có thể tương tác; nghĩa là, đầu ra được chuyển đến màn hình cho người dùng và đầu vào đến từ bàn phím, chuột hoặc màn hình cảm ứng của người dùng. Vì I/O tương tác thường chạy ở tốc độ chậm, nên có thể mất nhiều thời gian để hoàn thành. Ví dụ: đầu vào có thể bị giới hạn bởi tốc độ nhập của người dùng; bảy ký tự mỗi giây là nhanh đối với con người nhưng lại cực kỳ chậm đối với máy tính. Thay vì để CPU không hoạt động khi quá trình nhập tương tác này diễn ra, hệ điều hành sẽ nhanh chóng chuyển CPU sang một tiến trình khác.

### 1.3.2. Chế độ kép và đa chế độ

Vì hệ điều hành và người dùng chia sẻ tài nguyên phần cứng và phần mềm của hệ thống máy tính, một hệ điều hành được thiết kế phải đảm bảo rằng một chương trình không hợp lệ (hoặc độc hại) không thể khiến các chương trình khác (hoặc chính hệ điều hành) thực thi không chính xác. Để đảm bảo hệ thống thực thi đúng, chúng ta phải có khả năng phân biệt giữa việc thực thi mã hệ điều hành và mã do người dùng định nghĩa. Cách tiếp cận được thực hiện bởi hầu hết các hệ thống máy tính là cung cấp hỗ trợ phần cứng cho phép phân biệt giữa các phương thức thực thi khác nhau.

Ít nhất, chúng ta cần có hai chế độ hoạt động riêng biệt: chế độ người dùng và chế độ nhân (còn được gọi là chế độ giám sát, chế độ hệ thống, hoặc chế độ đặc quyền). Một bit, được gọi là bit chế độ, được thêm vào phần cứng của máy tính để chỉ ra chế độ hiện tại: nhân (0) hoặc người dùng (1). Với bit mode, chúng ta có thể phân biệt giữa tác vụ được thực thi của hệ điều hành và tác vụ được thực của người dùng. Khi hệ thống máy tính đang thực thi chương trình người dùng, hệ thống đang ở chế độ người dùng. Tuy nhiên, khi ứng dụng người dùng yêu cầu một dịch vụ từ hệ điều hành (qua lệnh gọi hệ thống), hệ thống phải chuyển từ chế độ người dùng sang chế độ nhân để thực hiện yêu cầu. Điều này được thể hiện trong Hình 1.10. Như chúng ta sẽ thấy, cải tiến kiến trúc này cũng hữu ích cho nhiều khía cạnh khác của hoạt động hệ thống.



**Hình 1.10.** Chuyển đổi chế độ người dùng sang chế độ nhân.

Tại thời điểm khởi động hệ điều hành, phần cứng khởi động ở chế độ nhân. Hệ điều hành sau đó được tải và khởi động các ứng dụng người dùng ở chế độ người dùng. Bất cứ khi nào bẫy hoặc ngắt xảy ra, phần cứng sẽ chuyển từ chế độ người dùng sang chế độ nhân (nghĩa là thay đổi trạng thái của bit chế độ thành 0). Do đó, bất cứ khi nào hệ điều hành giành được quyền kiểm soát máy tính, nó sẽ ở chế độ nhân. Hệ thống luôn chuyển sang chế độ người dùng (bằng cách đặt bit chế độ thành 1) trước khi chuyển quyền điều khiển cho chương trình người dùng.

Chế độ hoạt động kép cung cấp cho chúng ta các phương tiện để bảo vệ hệ điều hành khỏi những người dùng sai lầm. Chúng ta thực hiện biện pháp bảo vệ này bằng cách chỉ định một số lệnh máy có thể gây hại làm lệnh đặc quyền. Phần cứng chỉ cho phép các lệnh đặc quyền được thực thi trong chế độ nhân. Nếu cố gắng thực hiện một lệnh đặc quyền trong chế độ người dùng, phần cứng sẽ không thực hiện lệnh đó mà coi nó là bất hợp pháp và chuyển nó vào hệ điều hành.

Khái niệm về chế độ có thể được mở rộng ra ngoài hai chế độ. Ví dụ, bộ xử lý Intel có bốn vòng bảo vệ riêng biệt, trong đó vòng 0 là chế độ hạt nhân và vòng 3 là chế độ người dùng. (Mặc dù các vòng 1 và 2 có thể được sử dụng cho các dịch vụ hệ điều hành

khác nhau, nhưng trong thực tế, chúng hiếm khi được sử dụng.) Hệ thống ARMv8 có bảy chế độ. Các CPU hỗ trợ ảo hóa thường có một chế độ riêng để cho biết khi nào trình quản lý máy ảo (VMM) kiểm soát hệ thống. Trong chế độ này, VMM có nhiều đặc quyền hơn các tiến trình của người dùng nhưng ít hơn hạt nhân. Nó cần mức đặc quyền đó để nó có thể tạo và quản lý các máy ảo, thay đổi trạng thái CPU để làm như vậy.

### 1.3.3. Timer

Chúng ta phải đảm bảo rằng hệ điều hành duy trì quyền kiểm soát đối với CPU. Chúng ta không cho phép một chương trình người dùng bị kẹt trong một vòng lặp vô hạn hoặc không gọi được các dịch vụ hệ thống và không bao giờ trả lại quyền kiểm soát cho hệ điều hành. Để thực hiện mục tiêu này, chúng ta có thể sử dụng bộ đếm thời gian (Timer). Bộ hẹn giờ có thể được đặt để ngắt máy tính sau một khoảng thời gian nhất định. Khoảng thời gian có thể cố định (ví dụ: 1/60 giây) hoặc biến đổi (ví dụ: từ 1 mili giây đến 1 giây). Một bộ đếm thời gian thay đổi thường được thực hiện bởi một đồng hồ tốc độ cố định và một bộ đếm. Hệ điều hành đặt bộ đếm. Mỗi khi đồng hồ chạy, bộ đếm được giảm dần. Khi bộ đếm về 0, ngắt xảy ra. Ví dụ: bộ đếm 10 bit với đồng hồ 1 mili giây cho phép ngắt ở khoảng thời gian từ 1 mili giây đến 1.024 mili giây, trong các bước 1 mili giây.

## 1.4. Quản lý tài nguyên

### 1.4.1. Quản lý tiến trình

Một chương trình không thể làm gì trừ khi các lệnh của nó được CPU thực thi. Một chương trình đang thực thi là một tiến trình. Một chương trình như trình biên dịch là một tiến trình và một chương trình xử lý văn bản đang chạy là một tiến trình. Tương tự, một ứng dụng mạng xã hội trên thiết bị di động là một tiến trình. Ta có thể coi một tiến trình là một thể hiện của một chương trình đang thực thi.

Một tiến trình cần một số tài nguyên nhất định - bao gồm thời gian CPU, bộ nhớ, tệp và thiết bị I/O - để hoàn thành nhiệm vụ của nó. Các tài nguyên này thường được cấp phát cho tiến trình khi nó đang chạy. Ngoài các tài nguyên vật lý và logic mà một tiến trình nhận được khi nó được tạo, nhiều dữ liệu khởi tạo (đầu vào) khác có thể được chuyển đến tiến trình. Ví dụ: một tiến trình duyệt web có chức năng là hiển thị nội dung của trang web trên màn hình. Tiến trình sẽ được cung cấp URL làm đầu vào và sẽ thực hiện các lệnh thích hợp và lệnh gọi hệ thống để lấy và hiển thị thông tin mong muốn trên màn hình. Khi tiến trình kết thúc, hệ điều hành sẽ lấy lại mọi tài nguyên có thể tái sử dụng.

Chúng ta nhấn mạnh rằng bản thân một chương trình không phải là một tiến trình.

Một chương trình là một thực thể thụ động, giống như nội dung của một tệp được lưu trữ trên đĩa, trong khi một tiến trình là một thực thể hoạt động. Một tiến trình đơn luồng có một bộ đếm chương trình chỉ định lệnh tiếp theo để thực thi. Việc thực hiện một tiến trình như vậy phải theo trình tự. CPU thực hiện hết lệnh của tiến trình này đến lệnh khác cho đến khi tiến trình hoàn tất. Hơn nữa, tại bất kỳ thời điểm nào, chỉ có một lệnh của một tiến trình được thực hiện. Do đó, mặc dù hai tiến trình có thể được liên kết với cùng một chương trình, nhưng chúng vẫn được coi là hai trình tự thực thi riêng biệt. Một tiến trình đa luồng có nhiều bộ đếm chương trình, mỗi bộ đếm chỉ đến lệnh tiếp theo để thực thi cho một luồng nhất định.

Tiến trình là đơn vị công việc trong một hệ thống. Một hệ thống bao gồm tập hợp các tiến trình, một số trong số đó là các tiến trình của hệ điều hành (những tiến trình thực thi mã hệ thống) và phần còn lại là tiến trình của người dùng (những tiến trình thực thi mã người dùng). Tất cả các tiến trình này có khả năng thực thi đồng thời - trên một lõi CPU duy nhất, hoặc song song trên nhiều lõi CPU.

Hệ điều hành chịu trách nhiệm về các hoạt động sau liên quan đến quản lý tiến trình:

- Tạo và xóa cả tiến trình của người dùng và hệ thống
- Lập lịch các tiến trình và luồng trên CPU
- Tạm dừng và tiếp tục các tiến trình
- Cung cấp các cơ chế để đồng bộ hóa tiến trình
- Cung cấp cơ chế giao tiếp tiến trình

#### 1.4.2. Quản lý bộ nhớ

Bộ nhớ chính (main memory) là trung tâm của hoạt động của một hệ thống máy tính hiện đại. Bộ nhớ chính là một mảng lớn các byte, có kích thước từ hàng trăm nghìn đến hàng tỷ. Mỗi byte có địa chỉ riêng của nó.

Bộ nhớ chính là một kho lưu trữ dữ liệu có thể truy cập nhanh chóng được chia sẻ bởi CPU và các thiết bị I/O. CPU đọc lệnh từ bộ nhớ chính, và cả đọc/ghi dữ liệu từ bộ nhớ chính. Bộ nhớ chính thường là thiết bị lưu trữ lớn mà CPU có thể xử lý và truy cập trực tiếp. Ví dụ, để CPU xử lý dữ liệu từ đĩa, những dữ liệu đó trước tiên phải được chuyển vào bộ nhớ chính bằng các lệnh gọi I/O do CPU tạo ra. Theo cách tương tự, các lệnh phải nằm trong bộ nhớ để CPU thực thi chúng.

Để cải thiện cả việc sử dụng CPU và tốc độ phản hồi của máy tính với người dùng,

các máy tính đa năng phải lưu giữ một số chương trình trong bộ nhớ, tạo ra nhu cầu quản lý bộ nhớ. Nhiều chương trình quản lý bộ nhớ khác nhau được sử dụng. Các lược đồ này phản ánh nhiều cách tiếp cận khác nhau và hiệu quả của thuật toán phụ thuộc vào tình huống. Khi lựa chọn sơ đồ quản lý bộ nhớ cho một hệ thống cụ thể, chúng ta phải tính đến nhiều yếu tố - đặc biệt là thiết kế phần cứng của hệ thống. Mỗi thuật toán yêu cầu hỗ trợ phần cứng riêng.

Hệ điều hành chịu trách nhiệm về các hoạt động sau liên quan đến quản lý bộ nhớ:

- Theo dõi phần nào của bộ nhớ hiện đang được sử dụng và tiến trình nào đang sử dụng chúng.
- Phân bổ và thu hồi không gian bộ nhớ khi cần thiết.
- Quyết định tiến trình (hoặc các phần của tiến trình) và dữ liệu sẽ di chuyển vào và ra khỏi bộ nhớ.

#### 1.4.3. Quản lý hệ thống file

Quản lý tệp là một trong những thành phần của hệ điều hành. Máy tính có thể lưu trữ thông tin trên một số loại phương tiện vật lý khác nhau. Lưu trữ thực hiện trong bộ nhớ thứ cấp (Secondary storage), lưu trữ cấp ba (tertiary storage). Mỗi phương tiện này có đặc điểm và tổ chức vật lý riêng. Hầu hết được điều khiển bởi một thiết bị, chẳng hạn như ổ đĩa, cũng có những đặc điểm riêng biệt. Các thuộc tính này bao gồm tốc độ truy cập, dung lượng, tốc độ truyền dữ liệu và phương thức truy cập (tuần tự hoặc ngẫu nhiên).

Tệp là một tập hợp các thông tin liên quan do người dùng định nghĩa. Thông thường, tệp đại diện cho chương trình (cả dạng nguồn và dạng đối tượng) và dữ liệu. Tệp dữ liệu có thể là số, chữ cái, chữ và số hoặc nhị phân. Các tệp có thể ở dạng tự do (ví dụ: tệp văn bản) hoặc chúng có thể được định dạng cứng nhắc (ví dụ: các trường cố định như tệp nhạc mp3). Khái niệm tệp là một khái niệm cực kỳ chung chung.

Hệ điều hành chịu trách nhiệm về các hoạt động sau liên quan đến quản lý tệp:

- Tạo và xóa tệp
- Tạo và xóa các thư mục để tổ chức lưu trữ các tệp
- Hỗ trợ các nguyên tắc cơ bản để thao tác các tệp và thư mục
- Ánh xạ các tệp vào bộ nhớ chung
- Sao lưu các tệp trên phương tiện lưu trữ ổn định (không bay hơi)

#### 1.4.4. Quản lý bộ nhớ chung (mass store)

Hệ thống máy tính phải cung cấp bộ nhớ phụ để sao lưu bộ nhớ chính. Hầu hết các hệ thống máy tính hiện đại sử dụng ổ cứng HDD và thiết bị NVM làm phương tiện lưu trữ chính cho cả chương trình và dữ liệu.

Hầu hết các chương trình - bao gồm trình biên dịch, trình duyệt web, trình xử lý văn bản và trò chơi - được lưu trữ trên các thiết bị này cho đến khi được tải vào bộ nhớ. Sau đó, các chương trình sử dụng các thiết bị vừa là nguồn vừa là đích xử lý của chúng.

Do đó, việc quản lý thích hợp lưu trữ thứ cấp có tầm quan trọng đối với một hệ thống máy tính. Hệ điều hành chịu trách nhiệm về các hoạt động sau liên quan đến quản lý bộ nhớ thứ cấp:

- Gắn (mounting) và tháo (unmounting)
- Quản lý không gian trống
- Phân bổ lưu trữ
- Lập lịch đĩa
- Phân vùng
- Bảo vệ

Bởi vì bộ nhớ thứ cấp được sử dụng thường xuyên và rộng rãi, nó phải được sử dụng một cách hiệu quả. Toàn bộ tốc độ hoạt động của máy tính có thể phụ thuộc vào tốc độ của hệ thống con lưu trữ thứ cấp và các thuật toán điều khiển hệ thống con đó.

Đồng thời, có nhiều cách sử dụng để lưu trữ chậm hơn và chi phí thấp hơn (và đôi khi là dung lượng cao hơn) so với lưu trữ thứ cấp. Sao lưu dữ liệu đĩa, lưu trữ dữ liệu ít sử dụng và lưu trữ lâu dài là một số ví dụ. Ổ đĩa từ và băng của chúng, ổ đĩa CD DVD và Blu-ray, đĩa cứng là những thiết bị lưu trữ cấp ba điển hình.

Lưu trữ cấp ba không quan trọng đối với hiệu suất của hệ thống, nhưng nó vẫn phải được quản lý. Một số hệ điều hành đảm nhận nhiệm vụ này, trong khi những hệ điều hành khác để lại việc quản lý lưu trữ cấp ba cho các chương trình ứng dụng. Một số chức năng mà hệ điều hành có thể cung cấp bao gồm gắn và ngắt kết nối phương tiện trong thiết bị, cấp phát và giải phóng thiết bị để các tiến trình sử dụng riêng và di chuyển dữ liệu từ bộ nhớ thứ cấp sang bộ nhớ thứ ba.

#### 1.4.5. Quản lý bộ nhớ đệm (cache)

Bộ nhớ đệm là một nguyên tắc quan trọng của hệ thống máy tính. Đây là cách nó hoạt động: Thông tin thường được lưu giữ trong một số hệ thống lưu trữ (chẳng hạn như

bộ nhớ chính). Khi được sử dụng, nó được sao chép tạm thời vào một hệ thống lưu trữ nhanh hơn - bộ nhớ cache. Khi cần một thông tin cụ thể, trước tiên chúng ta kiểm tra xem nó có nằm trong bộ nhớ cache hay không. Nếu có, chúng ta sử dụng thông tin trực tiếp từ bộ nhớ cache. Nếu không, chúng ta đưa một bản sao vào bộ nhớ đệm.

Ngoài ra, các thanh ghi bên trong có thể lập trình cung cấp một bộ đệm tốc độ cao cho bộ nhớ chính. Lập trình viên (hoặc trình biên dịch) thực hiện các thuật toán cấp phát thanh ghi và thay thế thanh ghi để quyết định thông tin nào cần giữ trong các thanh ghi và thông tin nào sẽ giữ trong bộ nhớ chính.

Các bộ nhớ đệm khác được thực hiện hoàn toàn trong phần cứng. Ví dụ, hầu hết các hệ thống đều có bộ đệm để lưu giữ các lệnh dự kiến sẽ được thực thi tiếp theo. Nếu không có bộ nhớ đệm này, CPU sẽ phải đợi vài chu kỳ trong khi một lệnh được tìm nạp từ bộ nhớ chính. Vì những lý do tương tự, hầu hết các hệ thống có một hoặc nhiều bộ đệm dữ liệu tốc độ cao trong hệ thống phân cấp bộ nhớ.

Bởi vì bộ nhớ đệm có kích thước hạn chế, quản lý bộ nhớ cache là một vấn đề thiết kế quan trọng. Việc lựa chọn cẩn thận kích thước bộ nhớ cache và chính sách thay thế có thể tăng hiệu suất lên đáng kể, như bạn có thể thấy bằng cách xem Hình 1.11.

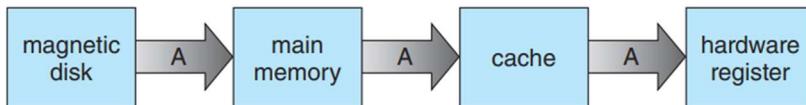
Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

**Hình 1.11.** Một số đặc điểm của các kiểu lưu trữ.

Sự di chuyển thông tin giữa các cấp của hệ thống phân cấp lưu trữ có thể rõ ràng hoặc ngầm định, tùy thuộc vào thiết kế phần cứng và phần mềm hệ điều hành kiểm soát. Ví dụ, truyền dữ liệu từ bộ nhớ cache đến CPU và thanh ghi thường là một chức năng phần cứng, không có sự can thiệp của hệ điều hành. Ngược lại, việc chuyển dữ liệu từ đĩa sang bộ nhớ thường do hệ điều hành kiểm soát.

Trong cấu trúc lưu trữ phân cấp, dữ liệu giống nhau có thể xuất hiện ở các cấp khác nhau của hệ thống lưu trữ. Ví dụ, giả sử rằng một số nguyên A được tăng thêm 1 nằm trong tệp B và tệp B nằm trên đĩa cứng. Phép toán tăng tiến hành bằng cách thực hiện

một thao tác I/O trước tiên để sao chép khôi đĩa mà A nằm trên đó vào bộ nhớ chính. Sau đó sao chép A vào bộ nhớ cache và vào một thanh ghi bên trong. Do đó, bản sao của A xuất hiện ở một số nơi: trên đĩa cứng, trong bộ nhớ chính, trong bộ đệm và trong một thanh ghi bên trong (Hình 1.12). Khi phép toán tăng diễn ra trong thanh ghi, giá trị của A sẽ khác trong các hệ thống lưu trữ khác nhau. Giá trị của A chỉ trở nên giống nhau sau khi giá trị mới của A được ghi từ thanh ghi bên trong trở lại đĩa cứng.



**Hình 1.12.** Di chuyển số A từ đĩa sang thanh ghi.

Trong môi trường máy tính chỉ có một tiến trình thực thi tại một thời điểm, cách sắp xếp này không gây khó khăn gì, vì quyền truy cập vào số nguyên A sẽ luôn là bản sao ở cấp cao nhất của hệ thống phân cấp. Tuy nhiên, trong môi trường đa nhiệm, nơi CPU được chuyển đổi qua lại giữa các tiến trình khác nhau, cần hết sức thận trọng để đảm bảo rằng, nếu một số tiến trình muốn truy cập A, thì mỗi tiến trình này sẽ nhận được giá trị được cập nhật gần nhất của A.

Tình hình trở nên phức tạp hơn trong môi trường đa xử lý, ngoài việc duy trì các thanh ghi bên trong, mỗi CPU còn chứa một bộ nhớ đệm cục bộ (xem lại Hình 1.7). Trong một môi trường như vậy, một bản sao của A có thể tồn tại đồng thời trong một số bộ nhớ đệm. Vì tất cả các CPU khác nhau đều có thể thực thi song song, chúng ta phải đảm bảo rằng cập nhật giá trị của A trong một bộ nhớ cache được phản ánh ngay lập tức trong tất cả các bộ nhớ cache khác nơi A cư trú. Tình huống này được gọi là đồng biến bộ nhớ cache và nó thường là sự cố phần cứng (được xử lý dưới cấp hệ điều hành).

#### 1.4.6. Quản lý hệ thống vào/ra (I/O)

Một trong những mục đích của hệ điều hành là che giấu những đặc trưng của các thiết bị phần cứng cụ thể khỏi người dùng. Ví dụ, trong UNIX, các đặc thù của thiết bị I/O được hệ thống con I/O ẩn khỏi phần lớn hệ điều hành. Hệ thống con I/O bao gồm một số thành phần:

- Thành phần quản lý bộ nhớ bao gồm bộ đệm, bộ nhớ đệm và spooling
- Giao diện trình điều khiển thiết bị chung
- Trình điều khiển cho các thiết bị phần cứng cụ thể.

### 1.5. Tóm tắt

- Hệ điều hành là phần mềm quản lý phần cứng máy tính, cũng như cung cấp môi

trường cho các chương trình ứng dụng chạy.

- Ngắt là một cách quan trọng trong đó phần cứng tương tác với hệ điều hành. Thiết bị phần cứng kích hoạt ngắt bằng cách gửi tín hiệu đến CPU để cảnh báo CPU rằng một số sự kiện cần được chú ý. Ngắt được quản lý bởi trình xử lý ngắt.

- Để một máy tính thực hiện công việc thực thi các chương trình, các chương trình phải nằm trong bộ nhớ chính, đây là vùng lưu trữ lớn duy nhất mà bộ xử lý có thể truy cập trực tiếp.

- Bộ nhớ chính thường là một thiết bị lưu trữ dễ bay hơi, mất nội dung khi tắt hoặc mất nguồn.

- Lưu trữ không bay hơi là một phần mở rộng của bộ nhớ chính và có khả năng lưu trữ một lượng lớn dữ liệu vĩnh viễn.

- Thiết bị lưu trữ không bay hơi phổ biến nhất là đĩa cứng, có thể cung cấp khả năng lưu trữ cả chương trình và dữ liệu.

- Sự đa dạng của các hệ thống lưu trữ trong một hệ thống máy tính có thể được tổ chức theo thứ bậc tùy theo tốc độ và chi phí. Các cấp độ cao hơn là đắt tiền, nhưng chúng nhanh. Khi di chuyển xuống cấp thấp, chi phí trên mỗi bit thường giảm, trong khi thời gian truy cập thường tăng lên.

- Kiến trúc máy tính hiện đại là các hệ thống đa xử lý, trong đó mỗi CPU chứa một số lõi tính toán.

- Để sử dụng CPU tốt nhất, các hệ điều hành hiện đại sử dụng đa chương trình, cho phép một số công việc được đưa vào bộ nhớ cùng một lúc, do đó đảm bảo rằng CPU luôn có công việc để thực thi.

- Đa nhiệm là một phần mở rộng của đa chương trình, trong đó các thuật toán lập lịch trình CPU chuyển đổi nhanh chóng giữa các tiến trình, cung cấp cho người dùng thời gian phản hồi nhanh.

- Để ngăn các chương trình người dùng can thiệp vào hoạt động bình thường của hệ thống, phần cứng hệ thống có hai chế độ: chế độ người dùng và chế độ hạt nhân.

- Các lệnh đặc quyền khác nhau chỉ có thể được thực thi trong chế độ hạt nhân. Các ví dụ bao gồm lệnh chuyển sang chế độ hạt nhân, điều khiển I/O, quản lý bộ định thời và quản lý ngắt.

- Tiến trình là đơn vị cơ bản của công việc trong hệ điều hành. Quản lý tiến trình bao gồm việc tạo và xóa các tiến trình và cung cấp cơ chế để các tiến trình giao tiếp và

đồng bộ hóa với nhau.

- Hệ điều hành quản lý bộ nhớ bằng cách theo dõi những phần nào của bộ nhớ đang được sử dụng và bởi ai. Nó cũng chịu trách nhiệm cấp phát động và giải phóng không gian bộ nhớ.
- Không gian lưu trữ được quản lý bởi hệ điều hành; điều này bao gồm việc cung cấp hệ thống tệp để biểu diễn cho tệp và thư mục và quản lý dung lượng trên các thiết bị lưu trữ lớn.



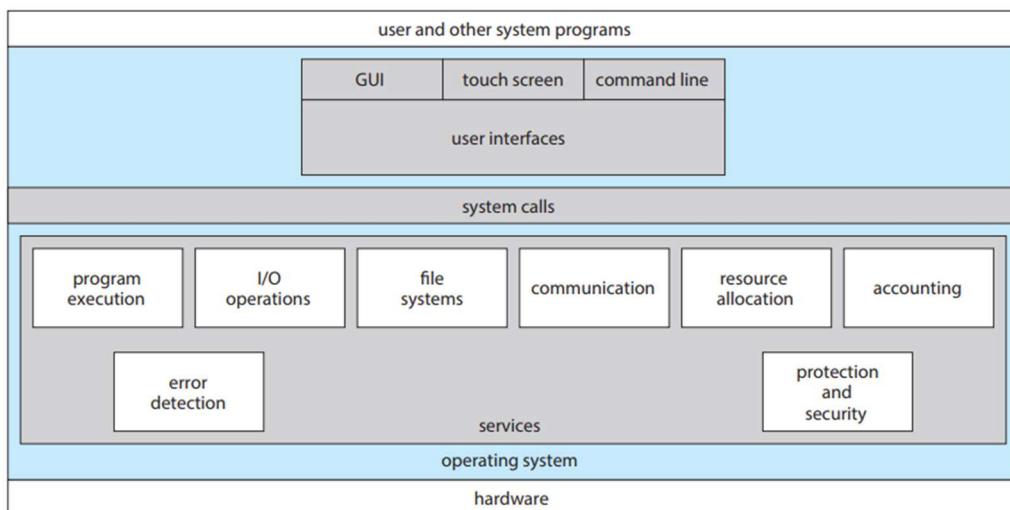
## CHƯƠNG 2. CẤU TRÚC HỆ ĐIỀU HÀNH

Hệ điều hành cung cấp môi trường mà các chương trình được thực thi. Các hệ điều hành khác nhau rất nhiều về cấu trúc của chúng, vì chúng được tổ chức theo nhiều cách khác nhau. Việc thiết kế một hệ điều hành mới là một nhiệm vụ chính. Điều quan trọng là các mục tiêu của hệ thống phải được xác định rõ ràng trước khi bắt đầu thiết kế. Những mục tiêu này tạo cơ sở cho các lựa chọn giữa các thuật toán và chiến lược khác nhau.

Chúng ta có thể xem một hệ điều hành từ một số quan điểm. Một quan điểm tập trung vào các dịch vụ mà hệ thống cung cấp; quan điểm dựa trên giao diện mà nó cung cấp cho người dùng và lập trình viên; quan điểm khác về các thành phần của nó và các kết nối của chúng. Trong chương này, chúng ta tìm hiểu cả ba khía cạnh của hệ điều hành, thể hiện quan điểm của người dùng, người lập trình và người thiết kế hệ điều hành.

### 2.1. Các dịch vụ hệ điều hành

Hệ điều hành cung cấp một môi trường để thực thi các chương trình. Nó cung cấp một số dịch vụ nhất định cho các chương trình và cho người sử dụng các chương trình đó. Tất nhiên, các dịch vụ cụ thể được cung cấp khác nhau từ hệ điều hành khác nhau, nhưng chúng ta có thể xác định các lớp chung. Hình 2.1 cho thấy cách nhìn về các dịch vụ hệ điều hành khác nhau và cách chúng tương tác với nhau.



**Hình 2.1.** Các dịch vụ cơ bản của hệ điều hành.

Một tập hợp các dịch vụ hệ điều hành cung cấp các chức năng hữu ích cho người dùng.

- Giao diện người dùng. Hầu hết tất cả các hệ điều hành đều có giao diện người dùng (UI). Giao diện này có thể có nhiều dạng. Thông thường nhất, giao diện người dùng đồ họa (GUI). Giao diện này là hệ thống cửa sổ với con chuột đóng vai trò là thiết bị trỏ để điều hướng vào/ra trực tiếp, chọn từ các menu, chọn và bàn phím để nhập văn bản. Các hệ thống di động như điện thoại và máy tính bảng cung cấp giao diện màn hình cảm ứng. Một tùy chọn khác là giao diện dòng lệnh (CLI, sử dụng các lệnh văn bản). Một số hệ thống cung cấp hai hoặc cả ba biến thể này.

- Thực hiện chương trình. Hệ thống phải có khả năng tải một chương trình vào bộ nhớ và chạy chương trình đó. Chương trình phải có thể kết thúc quá trình thực thi của nó, bình thường hoặc bất thường (chỉ ra lỗi).

- Hoạt động I/O. Một chương trình đang chạy có thể yêu cầu I/O, liên quan đến tệp hoặc thiết bị I/O. Đối với các thiết bị cụ thể, có các chức năng cụ thể. Để đạt hiệu suất và bảo vệ, người dùng thường không thể điều khiển các thiết bị I/O trực tiếp. Do đó, hệ điều hành phải cung cấp một phương tiện để thực hiện I/O.

- Thao tác hệ thống tập tin. Các chương trình cần đọc và ghi các tệp và thư mục. Nó cũng cần tạo và xóa chúng theo tên, tìm kiếm một tệp và liệt kê thông tin tệp. Một số hệ điều hành có quyền quản lý để cho phép hoặc từ chối quyền truy cập vào tệp hoặc thư mục dựa trên quyền sở hữu. Nhiều hệ điều hành cung cấp nhiều hệ thống tệp khác nhau.

- Truyền thông. Có nhiều trường hợp trong đó một tiến trình cần trao đổi thông tin với một tiến trình khác. Giao tiếp như vậy có thể xảy ra giữa các tiến trình đang thực thi trên cùng một máy tính hoặc giữa các tiến trình đang thực thi trên các hệ thống máy tính khác nhau được liên kết với nhau bởi một mạng. Truyền thông có thể được thực hiện thông qua bộ nhớ dùng chung, trong đó hai hoặc nhiều tiến trình đọc và ghi vào một phần bộ nhớ dùng chung, hoặc truyền thông điệp, trong đó các gói thông tin ở các định dạng xác định trước được hệ điều hành di chuyển giữa các tiến trình.

- Phát hiện lỗi. Hệ điều hành cần được phát hiện và sửa lỗi liên tục. Các lỗi có thể xảy ra trong phần cứng CPU và bộ nhớ (chẳng hạn như lỗi bộ nhớ hoặc lỗi nguồn), trong các thiết bị I/O (chẳng hạn như lỗi chấn lě trên đĩa, lỗi kết nối trên mạng hoặc thiếu giấy trong máy in), và trong chương trình người dùng (chẳng hạn như tràn số học hoặc cố gắng truy cập vị trí bộ nhớ bất hợp pháp). Đối với mỗi loại lỗi, hệ điều hành phải thực hiện hành động thích hợp để đảm bảo tính toán chính xác và nhất quán. Đôi khi, nó

không có lựa chọn nào khác ngoài việc tạm dừng hệ thống. Hoặc nó có thể chấm dứt một tiến trình gây lỗi hoặc trả lại mã lỗi cho một tiến trình để tiến trình phát hiện và có thể sửa chữa.

Một số chức năng khác của hệ điều hành không phải để giúp người dùng mà là để đảm bảo hoạt động hiệu quả của chính hệ thống. Các hệ thống có nhiều tiến trình có thể đạt được hiệu quả bằng cách chia sẻ tài nguyên máy tính giữa các tiến trình khác nhau.

- Phân bổ tài nguyên. Khi có nhiều tiến trình chạy cùng một lúc, tài nguyên phải được phân bổ cho từng tiến trình đó. Hệ điều hành quản lý nhiều loại tài nguyên khác nhau. Ví dụ: để xác định cách tốt nhất sử dụng CPU, hệ điều hành có các tiến trình lập lịch trình CPU. Cũng có thể có các tiến trình để phân bổ máy in, ổ lưu trữ USB và các thiết bị ngoại vi khác.

- Ghi nhật ký (logging). Chúng ta muốn theo dõi những chương trình nào sử dụng bao nhiêu và những loại tài nguyên máy tính nào. Việc lưu trữ hồ sơ này có thể được sử dụng để tích lũy số liệu thống kê sử dụng. Số liệu thống kê sử dụng có thể là một công cụ có giá trị cho các quản trị viên hệ thống muốn cấu hình lại hệ thống để cải thiện các dịch vụ máy tính.

- Bảo vệ và bảo mật. Chủ sở hữu thông tin được lưu trữ trong hệ thống máy tính đa người dùng hoặc nối mạng có thể muốn kiểm soát việc sử dụng thông tin đó. Khi một số tiến trình riêng biệt thực thi đồng thời, không thể để một tiến trình can thiệp vào các tiến trình khác hoặc với chính hệ điều hành. Bảo vệ liên quan đến việc đảm bảo rằng tất cả các truy cập vào tài nguyên hệ thống đều được kiểm soát. Bảo mật hệ thống khỏi người ngoài cũng rất quan trọng. Bảo mật như vậy bắt đầu bằng việc yêu cầu mỗi người dùng xác thực chính mình vào hệ thống, thường là bằng mật khẩu, để có quyền truy cập vào tài nguyên hệ thống.

## 2.2. Lời gọi hệ thống

Lệnh gọi hệ thống cung cấp giao diện cho các dịch vụ được cung cấp bởi hệ điều hành. Các lệnh gọi này thường có sẵn dưới dạng các hàm được viết bằng C và C++, một số tác vụ cấp thấp (ví dụ: các tác vụ mà phần cứng phải được truy cập trực tiếp) có thể phải được viết bằng lệnh hợp ngữ.

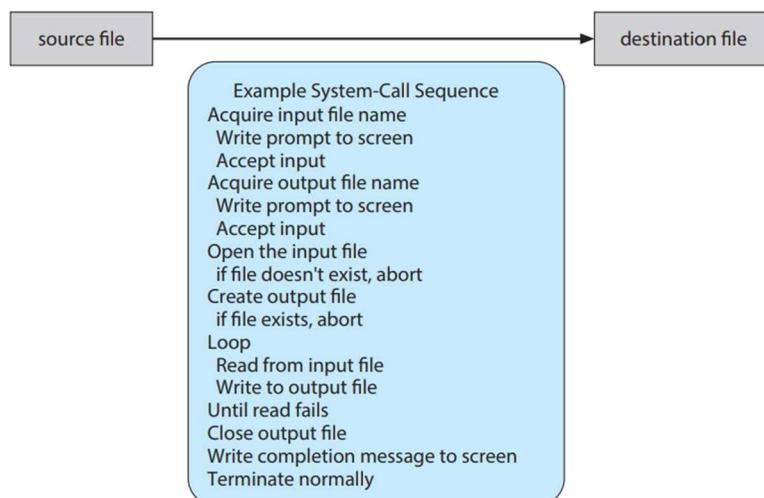
### 2.2.1. Ví dụ

Trước tiên, chúng ta hãy sử dụng một ví dụ để minh họa cách sử dụng lệnh gọi hệ thống: viết một chương trình đơn giản để đọc dữ liệu từ một tệp và sao chép chúng sang một tệp khác. Đầu vào đầu tiên mà chương trình sẽ cần là tên của hai tệp: tệp đầu vào

và tệp đầu ra. Những tên này có thể được chỉ định theo nhiều cách, tùy thuộc vào thiết kế hệ điều hành. Một cách thực hiện là chương trình hỏi người dùng tên file. Trong một hệ thống tương tác, cách tiếp cận này sẽ yêu cầu một chuỗi các lệnh gọi hệ thống, đầu tiên là viết một thông báo nhắc nhở trên màn hình và sau đó đọc từ bàn phím các ký tự xác định tên hai tệp. Chuỗi này yêu cầu nhiều lệnh gọi hệ thống I/O. Khi đã có được hai tên tệp, chương trình phải mở tệp đầu vào và mở tệp đầu ra. Mỗi hoạt động này yêu cầu một lệnh gọi hệ thống khác. Các điều kiện lỗi có thể xảy ra đối với mỗi lệnh gọi hệ thống phải được xử lý. Ví dụ: khi chương trình cố gắng mở tệp đầu vào, nó có thể thấy rằng không có tệp nào có tên đó hoặc tệp đó được bảo vệ chống lại quyền truy cập. Trong những trường hợp này, chương trình sẽ xuất ra thông báo lỗi (một chuỗi lệnh gọi hệ thống khác) và sau đó kết thúc bất thường (lệnh gọi hệ thống khác). Nếu tệp đầu vào tồn tại, thì chúng ta phải tạo một tệp đầu ra mới. Chúng tôi có thể thấy rằng đã có một tệp đầu ra có cùng tên. Tình huống này có thể khiến chương trình bị hủy bỏ (lệnh gọi hệ thống) hoặc chúng tôi có thể xóa tệp hiện có (lệnh gọi hệ thống khác) và tạo tệp mới (lệnh gọi hệ thống khác).

Khi cả hai tệp được thiết lập, chúng ta vào vòng lặp đọc từ tệp đầu vào (một lệnh gọi hệ thống) và ghi vào tệp đầu ra (một lệnh gọi hệ thống khác). Mỗi lần đọc và ghi phải trả về thông tin trạng thái liên quan đến các điều kiện lỗi khác nhau có thể xảy ra. Đối với đầu vào, lỗi có thể đến cuối tệp hoặc có lỗi phần cứng trong quá trình đọc (chẳng hạn như hư đĩa). Thao tác ghi có thể gặp nhiều lỗi khác nhau, tùy thuộc vào thiết bị đầu ra (ví dụ: không còn dung lượng đĩa trống).

Cuối cùng, sau khi toàn bộ tệp được sao chép, chương trình đóng cả hai tệp (hai lệnh gọi hệ thống), viết thông báo vào console và cuối cùng kết thúc bình thường (lệnh gọi hệ thống cuối cùng). Chuỗi cuộc gọi hệ thống này được thể hiện trong Hình 2.2.



**Hình 2.2.** Một ví dụ cách lệnh gọi hệ thống được sử dụng.

### 2.2.2. Giao diện lập trình ứng dụng (API)

Như ta thấy, ngay cả những chương trình đơn giản cũng có thể sử dụng nhiều lệnh trong hệ điều hành. Hệ thống có thể thực hiện hàng nghìn lệnh gọi hệ thống mỗi giây. Tuy nhiên, hầu hết các lập trình viên không bao giờ nhìn thấy mức độ chi tiết này. Thông thường, các nhà phát triển ứng dụng thiết kế chương trình theo một giao diện lập trình ứng dụng (API). API chỉ định một tập hợp các hàm có sẵn cho lập trình viên, bao gồm các tham số được truyền cho mỗi hàm và các giá trị trả về. Ba trong số các API phổ biến nhất có sẵn cho các lập trình viên là Windows API cho các hệ thống Windows, API POSIX cho các hệ thống dựa trên POSIX (bao gồm hầu như tất cả các phiên bản của UNIX, Linux và macOS) và API Java cho các chương trình chạy trên máy ảo Java. Lập trình viên truy cập API thông qua thư viện do hệ điều hành cung cấp.

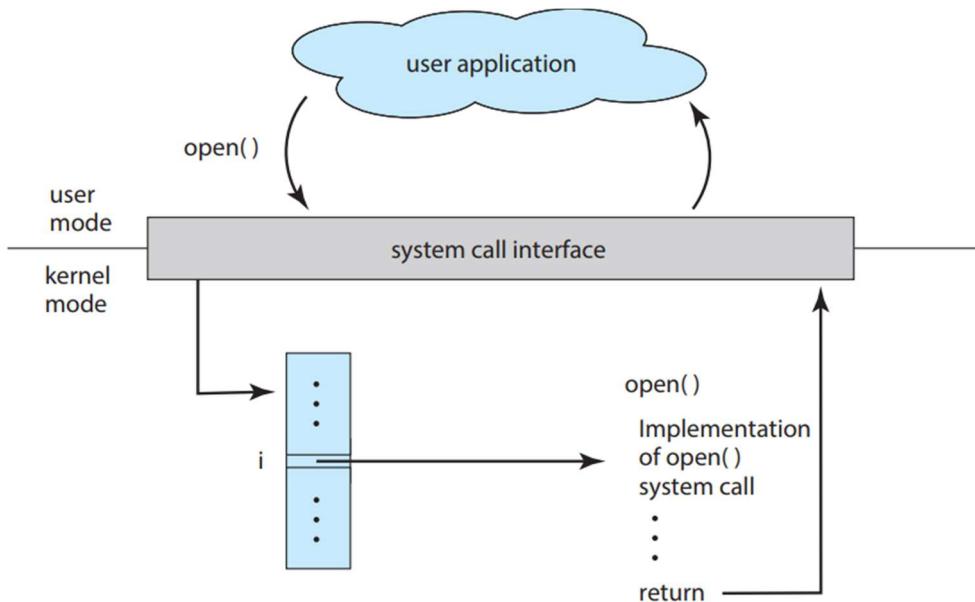
Bên trong, các hàm API thường gọi “giúp” các lệnh gọi hệ thống cho lập trình viên. Ví dụ, hàm Windows CreateProcess() (được sử dụng để tạo một tiến trình mới) thực sự gọi lệnh gọi hệ thống NTCREATEPROCESS() trong nhân Windows.

Tại sao một lập trình viên ứng dụng lại thích lập trình theo một API hơn là gọi các lệnh gọi hệ thống thực tế? Có một số lý do như sau. Thứ nhất, liên quan đến tính di động của chương trình. Một lập trình viên ứng dụng viết chương trình sử dụng API có thể mong chương trình của mình biên dịch và chạy trên bất kỳ hệ thống nào hỗ trợ cùng một API. Thứ hai, các lệnh gọi hệ thống thực tế thường có thể chi tiết và khó làm việc hơn so với API có sẵn cho một lập trình viên ứng dụng.

Một yếu tố quan trọng khác trong việc xử lý các lệnh gọi hệ thống là môi trường thời gian chạy (RTE) - bộ phần mềm đầy đủ cần thiết để thực thi các ứng dụng được viết bằng một ngôn ngữ lập trình nhất định, bao gồm trình biên dịch hoặc trình thông dịch của nó cũng như các phần mềm khác, chẳng hạn như thư viện và trình tải. RTE cung cấp giao diện cuộc gọi hệ thống để liên kết đến các cuộc gọi hệ thống do hệ điều hành cung cấp. Giao diện lệnh gọi hệ thống chặn các lệnh gọi hàm trong API và gọi các lệnh gọi hệ thống cần thiết trong hệ điều hành. Thông thường, một số được liên kết với mỗi lệnh gọi hệ thống và giao diện lệnh gọi hệ thống duy trì một bảng được lập chỉ mục theo các số này. Sau đó, giao diện cuộc gọi hệ thống sẽ gọi lệnh gọi hệ thống dự kiến trong nhân hệ điều hành và trả về trạng thái của lệnh gọi hệ thống.

Người gọi không cần biết gì về cách thực hiện lệnh gọi hệ thống hoặc nó làm gì trong quá trình thực thi. Thay vào đó, người gọi chỉ cần tuân theo API và hiểu hệ điều hành sẽ làm gì khi thực hiện lệnh gọi hệ thống đó. Do đó, hầu hết các chi tiết của giao diện hệ điều hành được ẩn với lập trình viên bởi API và được quản lý bởi RTE. Mọi

quan hệ giữa một API, giao diện lệnh gọi hệ thống và hệ điều hành được thể hiện trong Hình 2.3, minh họa cách hệ điều hành xử lý một ứng dụng người dùng gọi lệnh gọi hệ thống open().



**Hình 2.3. Xử lý ứng dụng người dùng dùng hàm hệ thống `open()`.**

### 2.2.3. Các kiểu gọi hệ thống

Các cuộc gọi hệ thống có thể được nhóm gần như thành sáu loại chính: kiểm soát tiến trình, quản lý file, quản lý thiết bị, bảo trì thông tin, truyền thông và bảo vệ. Một số lệnh gọi hệ thống có thể được cung cấp bởi một hệ điều hành.

#### a. Process control

- Tạo tiến trình, kết thúc tiến trình
- Tải tiến trình vào bộ nhớ, thực thi tiến trình
- Lấy/thiết lập thuộc tính tiến trình
- Đợi sự kiện, phát sinh sự kiện
- Cấp phát và giải phóng bộ nhớ

#### b. File management

- Tạo file, xóa file
- Mở, đóng file
- Đọc, viết, di chuyển con trỏ file
- Lấy/thiết lập thuộc tính file

*c. Quản lý thiết bị*

- Yêu cầu/giải phóng thiết bị
- Đọc, ghi, định vị thiết bị
- Lấy/thiết lập thuộc tính thiết bị
- Kết nối/gỡ kết nối (logic) thiết bị

*d. Bảo trì thông tin*

- Lấy/thiết lập ngày/giờ
- Lấy/thiết lập dữ liệu hệ thống
- Lấy thuộc tính của tiến trình, file hoặc thiết bị
- Thiết lập thuộc tính của tiến trình, file hoặc thiết bị

*e. Truyền thông*

- Tạo, xóa kết nối
- Gởi, nhận thông điệp
- Truyền thông tin trạng thái
- Kết nối, gỡ kết nối các thiết bị từ xa

*f. Bảo vệ*

- Lấy quyền truy cập file
- Thiết lập quyền truy cập file

**2.2.4. Dịch vụ hệ thống**

Dịch vụ hệ thống, còn được gọi là tiện ích hệ thống, cung cấp một môi trường thuận tiện cho việc phát triển và thực thi chương trình. Chúng có thể được chia thành các loại sau:

- Quản lý tập tin. Các chương trình này tạo, xóa, sao chép, đổi tên, in, liệt kê, và nói chung là truy cập và thao tác các tệp và thư mục.
- Thông tin trạng thái. Một số chương trình chỉ cần hỏi hệ thống về ngày, giờ, dung lượng bộ nhớ có sẵn hoặc dung lượng đĩa, số lượng người dùng hoặc thông tin trạng thái tương tự. Những thứ khác phức tạp hơn, cung cấp thông tin chi tiết về hiệu suất, ghi nhật ký và gỡ lỗi. Thông thường, các chương trình này định dạng và in đầu ra tới thiết bị đầu cuối hoặc các thiết bị hoặc tệp đầu ra khác hoặc hiển thị nó trong một cửa sổ của GUI.

Một số hệ thống cũng hỗ trợ sổ đăng ký, được sử dụng để lưu trữ và truy xuất thông tin cấu hình.

- Điều chỉnh tệp. Một số trình soạn thảo văn bản có thể có sẵn để tạo và sửa đổi nội dung của các tệp được lưu trữ trên đĩa hoặc các thiết bị lưu trữ khác. Cũng có thể có các lệnh đặc biệt để tìm kiếm nội dung của tệp hoặc thực hiện các phép biến đổi văn bản.
- Hỗ trợ ngôn ngữ lập trình. Trình biên dịch, trình assembler, trình gỡ lỗi và trình thông dịch cho các ngôn ngữ lập trình phổ biến (chẳng hạn như C, C++, Java và Python) thường được cung cấp cùng với hệ điều hành hoặc có sẵn dưới dạng bản tải xuống riêng biệt.
- Tải và thực thi chương trình. Khi một chương trình được biên dịch, nó phải được tải vào bộ nhớ để được thực thi. Hệ thống có thể cung cấp bộ tải tuyệt đối, bộ tải có thể thay đổi vị trí, bộ chỉnh sửa liên kết và bộ tải lớp phủ. Hệ thống gỡ lỗi cho cả ngôn ngữ cấp cao hơn hoặc ngôn ngữ máy cũng cần thiết.
- Thông tin liên lạc. Các chương trình này cung cấp cơ chế tạo kết nối ảo giữa các tiến trình, người dùng và hệ thống máy tính. Chúng cho phép người dùng gửi tin nhắn đến màn hình của nhau, duyệt các trang web, gửi tin nhắn e-mail, đăng nhập từ xa hoặc chuyển tệp từ máy này sang máy khác.
- Dịch vụ nền. Tất cả các hệ thống đều có các phương pháp khởi chạy các tiến trình hệ thống nhất định tại thời điểm khởi động. Một số tiến trình này kết thúc sau khi hoàn thành nhiệm vụ của chúng, trong khi những tiến trình khác tiếp tục chạy cho đến khi hệ thống tạm dừng. Các tiến trình hệ thống liên tục chạy được gọi là các dịch vụ, hệ thống con hoặc daemon. Một ví dụ bao gồm bộ lập lịch tiến trình khởi động các tiến trình theo một lịch trình cụ thể, dịch vụ giám sát lỗi hệ thống và máy chủ in. Các hệ thống điển hình có hàng chục daemon. Ngoài ra, hệ điều hành chạy các hoạt động quan trọng trong ngữ cảnh người dùng chứ không phải trong ngữ cảnh hạt nhân có thể sử dụng daemon để chạy các hoạt động này.

Cùng với các chương trình hệ thống, hầu hết các hệ điều hành đều được cung cấp các chương trình hữu ích trong việc giải quyết các vấn đề chung hoặc thực hiện các hoạt động thông thường. Các chương trình ứng dụng này bao gồm trình duyệt web, trình xử lý văn bản và định dạng văn bản, bảng tính, hệ thống cơ sở dữ liệu, trình biên dịch, các gói phân tích thống kê và vẽ biểu đồ, cũng như trò chơi.

### 2.2.5. Tại sao các ứng dụng lại dành riêng cho hệ điều hành

Về cơ bản, các ứng dụng được biên dịch trên một hệ điều hành không thể thực thi trên các hệ điều hành khác. Mỗi hệ điều hành cung cấp một nhóm lệnh gọi hệ thống. Lệnh gọi hệ thống là một phần của các dịch vụ được cung cấp bởi hệ điều hành để các ứng dụng sử dụng. Một ứng dụng có thể được chạy trên nhiều hệ điều hành theo một trong ba cách:

1. Ứng dụng có thể được viết bằng ngôn ngữ thông dịch (chẳng hạn như Python hoặc Ruby) có trình thông dịch có sẵn cho nhiều hệ điều hành. Trình thông dịch đọc từng dòng của chương trình nguồn, thực hiện các lệnh tương đương trên tập lệnh gốc và gọi các lệnh gọi hệ điều hành gốc. Hiệu suất bị chậm hơn so với các ứng dụng gốc và trình thông dịch chỉ cung cấp một tập hợp con các tính năng của mỗi hệ điều hành, có thể giới hạn các bộ tính năng của các ứng dụng được liên kết.

2. Ứng dụng có thể được viết bằng ngôn ngữ bao gồm một máy ảo chứa ứng dụng đang chạy. Máy ảo là một phần của RTE đầy đủ của ngôn ngữ. Một ví dụ của phương pháp này là Java. Java có RTE bao gồm trình tải, trình kiểm tra byte-code và các thành phần khác tải ứng dụng Java vào máy ảo Java. RTE này đã được chuyển hoặc phát triển cho nhiều hệ điều hành, từ máy tính lớn đến điện thoại thông minh và về lý thuyết, bất kỳ ứng dụng Java nào cũng có thể chạy trong RTE ở bất kỳ nơi nào có sẵn. Các hệ thống kiểu này có những nhược điểm tương tự như hệ thống thông dịch, đã thảo luận ở trên.

3. Nhà phát triển ứng dụng có thể sử dụng một ngôn ngữ hoặc API tiêu chuẩn trong đó trình biên dịch tạo ra các tệp nhị phân trong một ngôn ngữ cụ thể của máy và hệ điều hành. Ứng dụng phải được dịch theo từng hệ điều hành mà nó sẽ chạy. Việc dịch này tốn thời gian và phải được thực hiện cho mỗi phiên bản mới của ứng dụng.

Về lý thuyết, ba cách tiếp cận này dường như cung cấp các giải pháp đơn giản để phát triển các ứng dụng có thể chạy trên các hệ điều hành khác nhau. Tuy nhiên, phát triển ứng dụng chạy trên các hệ điều hành khác nhau gặp một số trở ngại sau:

- Mỗi hệ điều hành có một định dạng nhị phân khác nhau cho các ứng dụng: quy định cách bố trí của tiêu đề, lệnh và biến. Các thành phần đó cần phải ở một số vị trí nhất định trong cấu trúc được chỉ định trong tệp thực thi để hệ điều hành có thể mở tệp và tải ứng dụng để thực thi đúng.
- CPU có các bộ lệnh khác nhau và chỉ các ứng dụng có chứa các lệnh thích hợp mới có thể thực thi chính xác.
- Hệ điều hành cung cấp các lệnh gọi hệ thống cho phép các ứng dụng yêu cầu các hoạt động khác nhau, chẳng hạn như tạo tệp và mở kết nối mạng. Các lệnh gọi hệ thống đó khác nhau giữa các hệ điều hành theo nhiều khía cạnh, bao gồm các toán hạng cụ thể

và thứ tự toán hạng được sử dụng, cách một ứng dụng gọi các lệnh gọi hệ thống, cách đánh số, ý nghĩa của chúng và trả về kết quả của chúng.

### 2.3. Thiết kế và cài đặt hệ điều hành

#### 2.3.1. Mục tiêu thiết kế

Vấn đề đầu tiên trong việc thiết kế một hệ thống là xác định mục tiêu và thông số kỹ thuật. Thiết kế của hệ thống sẽ bị ảnh hưởng bởi sự lựa chọn phần cứng và loại hệ thống: máy tính để bàn/máy tính xách tay, di động, phân tán hoặc thời gian thực. Ngoài ra, các yêu cầu có thể khó xác định hơn nhiều. Tuy nhiên, các yêu cầu có thể được chia thành hai nhóm cơ bản: mục tiêu người dùng và mục tiêu hệ thống.

Đối với người dùng, hệ điều hành phải thuận tiện để sử dụng, dễ học, đáng tin cậy, an toàn và nhanh chóng.

Đối với người phát triển hệ điều hành, tạo, duy trì và vận hành. Hệ điều hành phải dễ thiết kế, triển khai và bảo trì; và nó phải linh hoạt, đáng tin cậy, không có lỗi và hiệu quả.

Tóm lại, không có giải pháp duy nhất nào cho vấn đề xác định các yêu cầu một hệ điều hành. Một loạt các hệ điều hành đang tồn tại cho thấy rằng các yêu cầu khác nhau có thể dẫn đến nhiều giải pháp cho các môi trường khác nhau. Đặc tả và thiết kế một hệ điều hành là một công việc mang tính sáng tạo cao. Mặc dù không có sách giáo khoa nào có thể cho bạn biết cách thực hiện, nhưng các nguyên tắc chung đã được phát triển trong lĩnh vực kỹ thuật phần mềm.

#### 2.3.2. Cơ chế và chính sách

Một nguyên tắc quan trọng là tách chính sách (policy) khỏi cơ chế (mechanism). Cơ chế xác định cách thực hiện một điều gì đó; chính sách xác định những gì sẽ được thực hiện. Ví dụ, cấu trúc Timer là một cơ chế để đảm bảo bảo vệ CPU, nhưng việc quyết định thời gian đặt bộ hẹn giờ cho một người dùng cụ thể là một chính sách.

Sự tách biệt giữa chính sách và cơ chế là rất quan trọng để tạo ra sự linh hoạt. Các chính sách có thể thay đổi theo địa điểm hoặc theo thời gian. Trong trường hợp xấu nhất, mỗi thay đổi trong chính sách sẽ đòi hỏi một sự thay đổi trong cơ chế cơ bản. Ưu tiên một cơ chế chung đủ linh hoạt để thực hiện nhiều chính sách. Khi đó, một thay đổi trong chính sách sẽ chỉ yêu cầu xác định lại một số tham số nhất định của hệ thống. Ví dụ, hãy xem xét cơ chế ưu tiên cho một số loại chương trình hơn những loại chương trình khác. Nếu cơ chế được tách biệt phù hợp với chính sách, nó có thể được sử dụng để hỗ trợ quyết định chính sách mà các chương trình sử dụng nhiều I/O nên được ưu tiên hơn các

chương trình sử dụng nhiều CPU hoặc để hỗ trợ chính sách ngược lại.

### 2.3.3. Cài đặt

Một khi hệ điều hành được thiết kế, nó phải được cài đặt. Bởi vì hệ điều hành là tập hợp của nhiều chương trình, được viết bởi nhiều người trong một khoảng thời gian dài, rất khó để đưa ra những tuyên bố chung về cách chúng được thực hiện.

Hệ điều hành ban đầu được viết bằng hợp ngữ. Nay giờ, hầu hết được viết bằng các ngôn ngữ cấp cao hơn như C hoặc C++. Trên thực tế, nhiều hơn một ngôn ngữ cấp cao hơn thường được sử dụng. Các cấp thấp nhất của nhân có thể được viết bằng hợp ngữ và C. Các tiến trình cấp cao hơn có thể được viết bằng C và C++, và các thư viện hệ thống có thể được viết bằng C++ hoặc thậm chí các ngôn ngữ cấp cao hơn. Android cung cấp một ví dụ: nhân của nó được viết chủ yếu bằng C với một số ít bằng hợp ngữ. Hầu hết các thư viện hệ thống Android được viết bằng C hoặc C++ và các framework - cung cấp giao diện nhà phát triển cho hệ thống - được viết chủ yếu bằng Java.

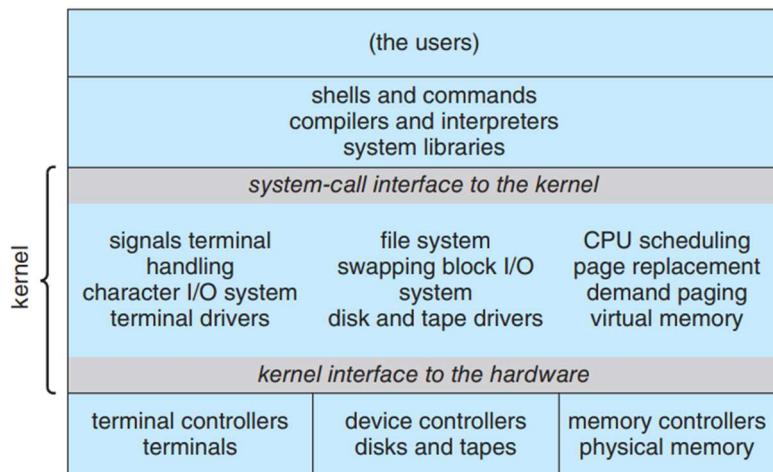
## 2.4. Cấu trúc hệ điều hành

Một hệ thống lớn và phức tạp như một hệ điều hành hiện đại phải được thiết kế cẩn thận nếu nó hoạt động tốt và dễ dàng sửa đổi. Một cách tiếp cận phổ biến là phân chia nhiệm vụ thành các thành phần nhỏ hoặc mô-đun, thay vì có một hệ thống duy nhất. Mỗi mô-đun này là một phần rõ ràng của hệ thống với các giao diện và chức năng được xác định cẩn thận. Ta có thể sử dụng một cách tiếp cận tương tự khi cấu trúc chương trình của mình: thay vì đặt tất cả mã của bạn trong hàm main(), ta tách thành một số hàm, trình bày rõ ràng các tham số và trả về giá trị, sau đó gọi các hàm từ hàm main().

### 2.4.1. Cấu trúc nguyên khôi

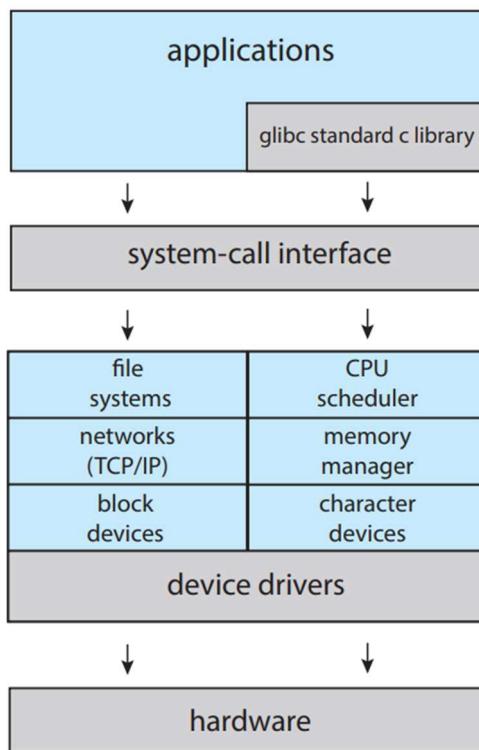
Cấu trúc đơn giản nhất để tổ chức một hệ điều hành là không có cấu trúc nào cả. Đó là, đặt tất cả các chức năng của nhân vào một tệp nhị phân tĩnh, duy nhất chạy trong một không gian địa chỉ duy nhất. Cách tiếp cận này - được gọi là cấu trúc nguyên khôi - là một kỹ thuật phổ biến để thiết kế hệ điều hành.

Một ví dụ về cấu trúc hạn chế như vậy là hệ điều hành UNIX, bao gồm hai phần tách biệt: nhân và các chương trình hệ thống. Nhân được tách ra thành một loạt các giao diện và trình điều khiển thiết bị, đã được thêm vào và mở rộng qua nhiều năm (Hình 2.4). Mọi thứ bên dưới giao diện gọi hệ thống và bên trên phần cứng vật lý là nhân. Kernel cung cấp hệ thống tệp, lập lịch CPU, quản lý bộ nhớ và các chức năng khác của hệ điều hành thông qua các lệnh gọi hệ thống. Tóm lại, đó là một lượng lớn chức năng được kết hợp vào một không gian địa chỉ duy nhất.



**Hình 2.4.** Cấu trúc hệ điều hành UNIX.

Hệ điều hành Linux dựa trên UNIX và có cấu trúc tương tự như trong Hình 2.5. Các ứng dụng thường sử dụng thư viện C chuẩn glibc khi giao tiếp với giao diện lời gọi hệ thống với nhân. Nhân Linux là nguyên khói ở chỗ nó chạy hoàn toàn ở chế độ nhân trong một không gian địa chỉ duy nhất, nhưng nó có thiết kế mở-đun cho phép nhân được sửa đổi trong thời gian chạy.



**Hình 2.5.** Cấu trúc hệ điều hành Linux.

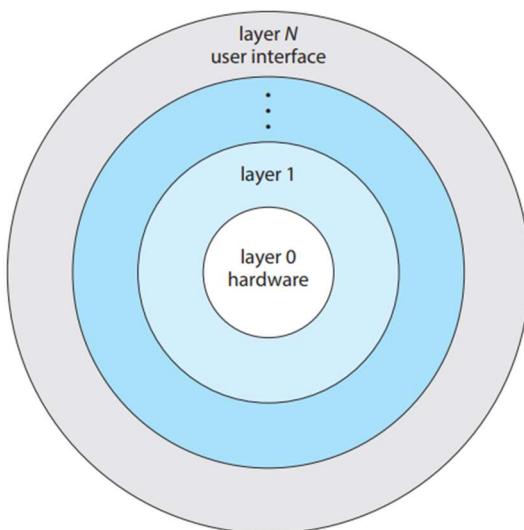
Mặc dù cấu trúc nhân nguyên khói là sự đơn giản, nhưng chúng rất khó cài đặt và mở rộng. Tuy nhiên, nhân nguyên khói có một lợi thế hiệu suất: có rất ít chi phí trong giao diện cuộc gọi hệ thống và giao tiếp bên trong hạt nhân rất nhanh. Do đó, dù có

những hạn chế, nhưng tốc độ và hiệu quả của chúng nên điều hành UNIX, Linux và Windows vẫn còn sử dụng cấu trúc này.

#### 2.4.2. Cấu trúc phân lớp

Cách tiếp cận nguyên khôi là một hệ thống liên kết chặt chẽ bởi vì những thay đổi đối với một phần của hệ thống có thể có ảnh hưởng đến với các phần khác. Chúng ta có thể thiết kế một hệ thống liên kết lỏng lẻo hơn, tức là chia thành các thành phần riêng biệt, nhỏ hơn có chức năng cụ thể và hạn chế. Tất cả các thành phần này cùng nhau tạo nên nhau. Ưu điểm của cách tiếp cận mô-đun này là những thay đổi trong một thành phần chỉ ảnh hưởng đến thành phần đó và không ảnh hưởng đến thành phần khác, cho phép người cài đặt hệ thống tự do hơn trong việc tạo và thay đổi hoạt động bên trong của hệ thống.

Một hệ thống có thể được tạo thành mô-đun theo nhiều cách. Một phương pháp là phân lớp, trong đó hệ điều hành được chia thành một số lớp (cấp độ). Lớp dưới cùng (lớp 0) là phần cứng; cao nhất (lớp N) là giao diện người dùng. Cấu trúc phân lớp này được mô tả trong Hình 2.6.



**Hình 2.6. Cấu trúc hệ điều hành phân lớp.**

Một lớp trong hệ điều hành là một cài đặt của một đối tượng trừu tượng được tạo thành từ dữ liệu và các thao tác với dữ liệu đó. Một lớp hệ điều hành điển hình - ví dụ, lớp M - bao gồm các cấu trúc dữ liệu và một tập hợp các hàm có thể được gọi bởi các lớp cấp cao hơn. Đến lượt mình, lớp M có thể gọi các hàm ở các lớp cấp thấp hơn.

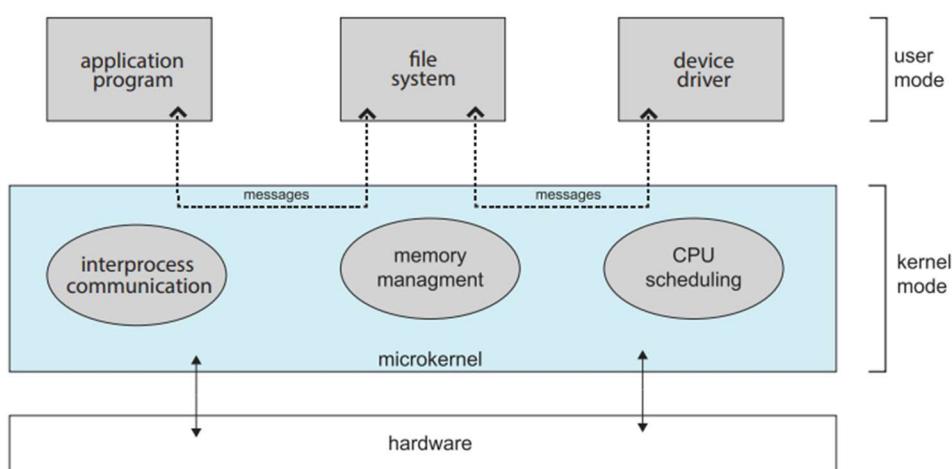
Ưu điểm chính của cách tiếp cận phân lớp là sự đơn giản của việc cài đặt và gỡ rối. Các lớp được chọn để mỗi lớp chỉ sử dụng các hàm (thao tác) và dịch vụ của các lớp cấp thấp hơn. Cách tiếp cận này đơn giản hóa việc gỡ rối. Lớp đầu tiên có thể được

gỡ rối mà không cần quan tâm phần còn lại của hệ thống, bởi vì, nó chỉ sử dụng phần cứng cơ bản để thực hiện các chức năng của nó. Khi lớp đầu tiên được gỡ rối, chức năng đã chính xác và có thể sử dụng để gỡ rối lớp thứ hai, v.v. Nếu lỗi được tìm thấy trong quá trình gỡ rối một lớp cụ thể, thì lỗi đó phải nằm trên lớp đó, vì các lớp bên dưới nó đã được gỡ lỗi. Do đó, việc thiết kế và triển khai hệ thống được đơn giản hóa.

Mỗi lớp chỉ được cài đặt với các thao tác được cung cấp bởi các lớp thấp hơn. Một lớp không cần biết các thao tác này được thực hiện như thế nào; nó chỉ cần biết những thao tác này làm gì. Do đó, mỗi lớp ẩn cấu trúc dữ liệu, thao tác và phần cứng từ các lớp cấp cao hơn.

#### 2.4.3. Cấu trúc vi nhân

Chúng ta thấy hệ thống UNIX ban đầu có cấu trúc nguyên khôi. Khi UNIX mở rộng, nhân trở nên lớn và khó quản lý. Vào giữa những năm 1980, các nhà nghiên cứu tại Đại học Carnegie Mellon đã phát triển một hệ điều hành gọi là Mach có thể mô-đun hóa nhân bằng cách sử dụng phương pháp vi nhân (microkernel). Phương pháp này xây dựng hệ điều hành bằng cách loại bỏ tất cả các thành phần không cần thiết khỏi nhân và cài đặt chúng dưới dạng các chương trình cấp người dùng nằm trong các không gian địa chỉ riêng biệt. Kết quả là một nhân nhỏ hơn. Có rất ít sự đồng thuận liên quan đến việc dịch vụ nào nên duy trì trong nhân và dịch vụ nào nên được cài đặt trong không gian người dùng. Tuy nhiên, thông thường, các vi nhân cung cấp tối thiểu khả năng quản lý tiến trình và bộ nhớ. Hình 2.7 minh họa kiến trúc của một vi nhân điển hình.



**Hình 2.7. Cấu trúc hệ điều hành vi nhân.**

Chức năng chính của vi nhân là cung cấp truyền thông giữa chương trình và các dịch vụ đang chạy trong không gian người dùng. Giao tiếp được cung cấp thông qua việc truyền thông điệp. Ví dụ, nếu chương trình muốn truy cập một tệp, nó phải tương tác với máy chủ tệp. Chương trình khách và dịch vụ không bao giờ tương tác trực tiếp.

Thay vào đó, họ giao tiếp gián tiếp bằng cách trao đổi thông điệp thông qua vi nhân.

Một lợi ích của phương pháp vi nhân là nó giúp mở rộng hệ điều hành dễ dàng hơn. Tất cả các dịch vụ mới được thêm vào không gian người dùng và do đó không yêu cầu sửa đổi nhân. Khi nhân phải sửa đổi, các thay đổi có xu hướng ít. Kết quả là hệ điều hành dễ dàng chuyển từ phần cứng này sang phần cứng khác. Vi nhân cũng cung cấp tính bảo mật và độ tin cậy cao hơn, vì hầu hết các dịch vụ đang chạy với tư cách tiến trình người dùng. Nếu một dịch vụ lỗi, phần còn lại của hệ điều hành vẫn không bị ảnh hưởng.

Minh họa về hệ điều hành microkernel là Darwin, thành phần nhân của hệ điều hành macOS và iOS. Thực tế, Darwin gồm hai nhân, một trong số đó là vi nhân Mach.

Một ví dụ khác là QNX, một hệ điều hành thời gian thực cho các hệ thống nhúng. Vi nhân QNX Neutrino cung cấp các dịch vụ truyền thông điệp và lập lịch. Nó cũng xử lý giao tiếp mạng mức độ thấp và các ngắt phần cứng. Tất cả các dịch vụ khác trong QNX được cung cấp bởi các tiến trình chuẩn chạy bên ngoài nhân ở chế độ người dùng.

Hiệu suất của các vi nhân có thể bị ảnh hưởng do chi phí chức năng hệ thống tăng lên. Khi hai dịch vụ mức người dùng phải giao tiếp, các thông điệp phải được sao chép giữa các dịch vụ, các thông điệp này nằm trong các không gian địa chỉ riêng biệt. Ngoài ra, hệ điều hành có thể phải chuyển từ tiến trình này sang tiến trình tiếp theo để trao đổi các thông điệp. Chi phí liên quan đến việc sao chép thông điệp và chuyển đổi giữa các tiến trình là trở ngại lớn nhất đối với sự phát triển của các hệ điều hành dựa trên vi nhân. Xét lịch sử của Windows NT: Bản phát hành đầu tiên tổ chức vi nhân. Hiệu suất của phiên bản này thấp so với Windows 95. Windows NT 4.0 đã khắc phục một phần sự cố hiệu suất bằng cách chuyển các lớp từ không gian người dùng sang không gian nhân và tích hợp chúng chặt chẽ hơn. Vào thời điểm Windows XP được thiết kế, kiến trúc Windows đã trở nên nguyên khôi.

#### 2.4.4. Mô đun

Có lẽ phương pháp tốt nhất hiện tại để thiết kế hệ điều hành là sử dụng các mô-đun nhân có thể tải được (LKM). Ở đây, nhân có một tập hợp các thành phần lõi và có thể liên kết với dịch vụ bổ sung thông qua các mô-đun, tại thời điểm khởi động hoặc trong thời gian chạy. Kiểu thiết kế này phổ biến trong các hệ điều hành hiện nay, chẳng hạn như Linux, macOS và Solaris, Windows.

Ý tưởng để nhân cung cấp các dịch vụ cốt lõi, trong khi các dịch vụ khác được thực hiện động, khi nhân đang chạy. Liên kết động các dịch vụ thích hợp hơn là thêm các tính năng mới trực tiếp vào nhân (điều này sẽ yêu cầu biên dịch lại hạt nhân mỗi khi

có thay đổi). Vì vậy, chẳng hạn, chúng ta có thể xây dựng các thuật toán lập lịch CPU và quản lý bộ nhớ trực tiếp vào nhân và sau đó thêm hỗ trợ cho các hệ thống tệp khác nhau bằng cách các mô-đun có thể tải được.

Kết quả tổng thể giống như một hệ thống phân lớp trong đó mỗi phần nhân có các giao diện được xác định, được bảo vệ; nhưng nó linh hoạt hơn hệ thống phân lớp, bởi vì bất kỳ mô-đun nào cũng có thể gọi bất kỳ mô-đun khác. Cách tiếp cận này cũng tương tự như cách tiếp cận vi nhân ở chỗ mô-đun chính chỉ có các chức năng lõi và biết cách tải và giao tiếp với các mô-đun khác; nhưng nó hiệu quả hơn, bởi vì các mô-đun không cần phải truyền thông điệp để giao tiếp.

Linux sử dụng các mô-đun nhân có thể tải được, chủ yếu để hỗ trợ trình điều khiển thiết bị và hệ thống tệp. LKM có thể được “chèn” vào hạt nhân khi hệ thống được khởi động hoặc trong thời gian chạy, chẳng hạn như khi thiết bị USB được cắm vào máy đang chạy. Nếu nhân Linux không có trình điều khiển cần thiết, nó có thể được tải động. LKM cũng có thể bị xóa khỏi hạt nhân trong thời gian chạy.

#### 2.4.5. Hệ thống hỗn hợp

Trong thực tế, rất ít hệ điều hành áp dụng một cấu trúc duy nhất. Thay vào đó, chúng kết hợp các cấu trúc khác nhau, dẫn đến các hệ thống kết hợp để giải quyết các vấn đề về hiệu suất, bảo mật và khả năng sử dụng. Ví dụ, Linux là nguyên khôi, bởi vì việc đặt hệ điều hành trong một không gian địa chỉ duy nhất mang lại hiệu suất cao. Tuy nhiên, nó cũng theo mô-đun, do đó chức năng mới có thể được thêm động vào nhân. Windows cũng phần lớn là nguyên khôi (lý do hiệu suất), nhưng nó vẫn giữ một số hành vi điển hình của các hệ thống vi nhân, bao gồm việc cung cấp hỗ trợ cho các hệ thống con riêng biệt chạy dưới dạng các tiến trình chế độ người dùng. Hệ điều hành Windows cũng cung cấp sự hỗ trợ cho các mô-đun nhân có thể tải động.

### 2.5. Tóm tắt

- Hệ điều hành cung cấp môi trường để thực thi các chương trình bằng cách cung cấp dịch vụ cho người dùng và chương trình.
- Ba cách tiếp cận chính để tương tác với hệ điều hành là (1) trình thông dịch lệnh, (2) giao diện người dùng đồ họa và (3) giao diện màn hình cảm ứng.
- Các cuộc gọi hệ thống cung cấp giao diện cho các dịch vụ được cung cấp bởi hệ điều hành. Người lập trình sử dụng giao diện lập trình ứng dụng (API) của lệnh gọi hệ thống để truy cập các dịch vụ lệnh gọi hệ thống.
- Các cuộc gọi hệ thống có thể được chia thành sáu loại chính: (1) kiểm soát tiến

trình, (2) quản lý tệp, (3) quản lý thiết bị, (4) bảo trì thông tin, (5) truyền thông và (6) bảo vệ.

- Thư viện C chuẩn cung cấp giao diện gọi hệ thống cho các hệ thống UNIX và Linux.

- Hệ điều hành cũng bao gồm một tập hợp các chương trình hệ thống cung cấp các tiện ích cho người dùng.

- Một trình liên kết kết hợp một số mô-đun đối tượng có thể định vị lại thành một tệp thực thi nhị phân duy nhất. Bộ tải nạp tệp thực thi vào bộ nhớ, nơi nó đủ điều kiện để chạy trên một CPU có sẵn.

- Có một số lý do tại sao các ứng dụng dành riêng cho hệ điều hành. Chúng bao gồm các định dạng nhị phân khác nhau cho các tệp thực thi chương trình, các tập lệnh khác nhau cho các CPU khác nhau và các lệnh gọi hệ thống thay đổi từ hệ điều hành này sang hệ điều hành khác.

- Hệ điều hành được thiết kế với các mục tiêu cụ thể. Những mục tiêu này cuối cùng xác định các chính sách của hệ điều hành. Hệ điều hành thực hiện các chính sách này thông qua các cơ chế cụ thể.

- Một hệ điều hành nguyên khôi không có cấu trúc; tất cả các chức năng được cung cấp trong một tệp nhị phân tĩnh, duy nhất chạy trong một không gian địa chỉ duy nhất. Mặc dù những hệ thống như vậy rất khó sửa đổi, nhưng lợi ích chính của chúng là hiệu quả.

- Hệ điều hành phân lớp được chia thành một số lớp rời rạc, trong đó lớp dưới cùng là giao diện phần cứng và lớp cao nhất là giao diện người dùng. Mặc dù các hệ thống phần mềm phân lớp đã đạt được một số thành công, nhưng cách tiếp cận này thường không lý tưởng để thiết kế hệ điều hành do các vấn đề về hiệu suất.

- Phương pháp microkernel để thiết kế hệ điều hành sử dụng một nhân tối thiểu; hầu hết các dịch vụ chạy dưới dạng ứng dụng cấp người dùng. Giao tiếp diễn ra thông qua việc chuyển tin nhắn.

- Cách tiếp cận mô-đun để thiết kế hệ điều hành cung cấp các dịch vụ hệ điều hành thông qua các mô-đun có thể được tải và gỡ bỏ trong thời gian chạy. Nhiều hệ điều hành hiện đại được xây dựng như một hệ thống lai sử dụng sự kết hợp của một nhân nguyên khôi và các mô-đun.

- Bộ nạp khởi động tải hệ điều hành vào bộ nhớ, thực hiện khởi tạo và bắt đầu thực thi hệ thống.



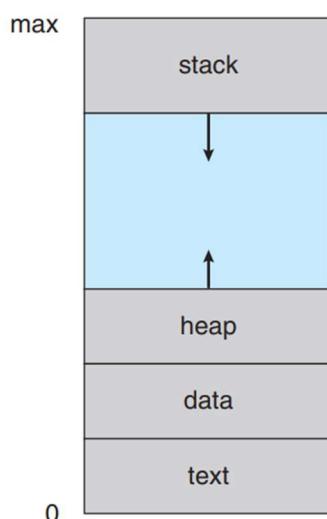
## CHƯƠNG 3. TIẾN TRÌNH VÀ LUỒNG

### 3.1. Khái niệm tiến trình

#### 3.1.1. Tiến trình (process)

Thuật ngữ "tiến trình" trong ngữ cảnh của hệ điều hành lần đầu tiên được sử dụng bởi các nhà thiết kế của hệ thống Multics vào những năm 1960. Kể từ thời điểm đó, thuật ngữ tiến trình, được sử dụng thay thế cho nhau với thuật ngữ nhiệm vụ, đã được đưa ra nhiều định nghĩa, chẳng hạn như: một chương trình đang thực thi.

Bộ nhớ của một tiến trình thường được chia thành nhiều phần và được thể hiện trong Hình 3.1. Các phần này bao gồm:

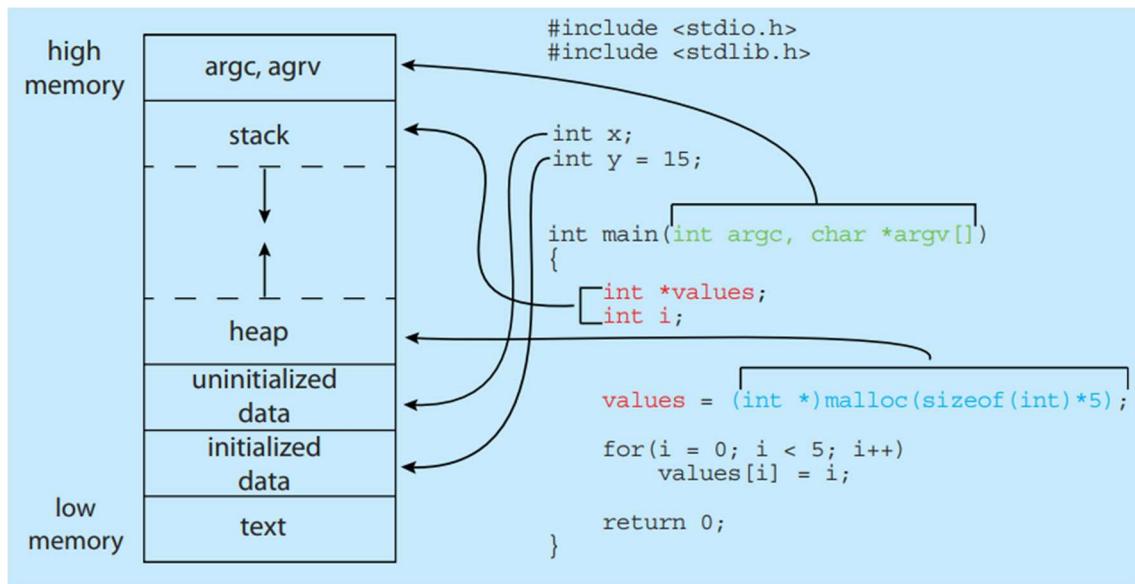


**Hình 3.1.** Bộ nhớ của tiến trình.

- Phần Text: mã thực thi
- Phần Data: các biến toàn cục
- Phần Heap: bộ nhớ được cấp phát động trong thời gian chạy chương trình
- Phần Stack: lưu trữ dữ liệu tạm thời khi gọi hàm (chẳng hạn như tham số hàm, địa chỉ trả về và biến cục bộ).

Lưu ý kích thước của phần text và data là cố định, tức là kích thước của chúng không thay đổi trong thời gian chạy chương trình. Tuy nhiên, các phần stack và heap có

thể thu nhỏ và mở rộng trong lúc thực thi chương trình. Mỗi khi hàm được gọi, một bản ghi chứa các tham số hàm, biến cục bộ và địa chỉ trả về được đẩy lên ngăn xếp; khi quyền điều khiển được trả về từ hàm, bản ghi sẽ lấy từ ngăn xếp. Tương tự, heap sẽ phát triển khi bộ nhớ được cấp phát động và sẽ thu hẹp lại khi bộ nhớ được trả về hệ thống. Mặc dù phần stack và heap mở rộng về phía nhau, nhưng hệ điều hành đảm bảo chúng không chồng lên nhau.



**Hình 3.2.** Một chương trình C trong bộ nhớ.

Chúng ta nhớ rằng bản thân một chương trình không phải là một tiến trình. Chương trình là một thực thể thụ động, chẳng hạn như một tệp chứa danh sách các lệnh được lưu trữ trên đĩa (thường được gọi là tệp thực thi). Ngược lại, tiến trình là một thực thể hoạt động, với một bộ đếm chương trình chỉ định lệnh tiếp theo để thực thi và tập hợp các tài nguyên liên quan. Một chương trình trở thành tiến trình khi một tệp thực thi được tải vào bộ nhớ. Hai kỹ thuật phổ biến để tải tệp thực thi là nhấp đúp vào biểu tượng đại diện cho tệp thực thi và nhập tên của tệp thực thi trên dòng lệnh.

Mặc dù một chương trình có thể có hai tiến trình, tuy nhiên chúng được coi là hai luồng thực thi riêng biệt. Ví dụ: người dùng có thể chạy các bản sao khác nhau của chương trình mail hoặc có thể gọi nhiều bản sao của chương trình trình duyệt web. Mỗi tiến trình này là một tiến trình riêng biệt; và các phần text là tương đương nhau, các phần data, heap và stack khác nhau. Tiến trình cũng có thể sinh ra tiến trình khác khi nó chạy.

Lưu ý rằng bản thân một tiến trình có thể là một môi trường thực thi cho tiến trình khác. Mỗi trường lập trình Java cung cấp một ví dụ điển hình. Một chương trình Java được thực thi trong máy ảo Java (JVM). JVM như một tiến trình thông dịch mã Java đã

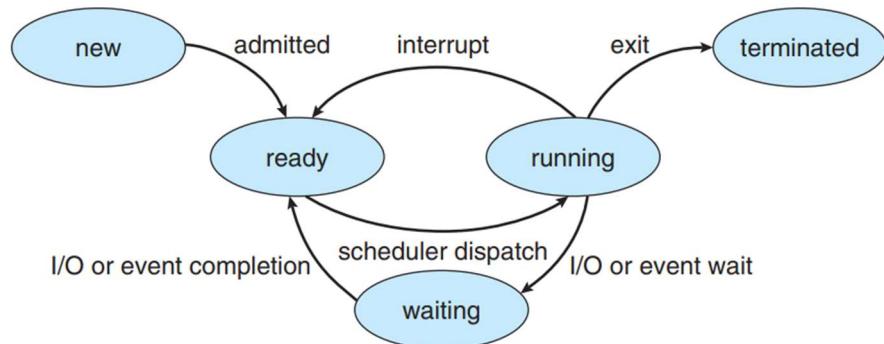
được tải và thực hiện các hành động (qua các lệnh gốc của máy).

### 3.1.2. Trạng thái của tiến trình

Khi một tiến trình thực thi, nó sẽ thay đổi trạng thái. Tiến trình có thể ở một trong các trạng thái sau:

- New: tiến trình này đang được tạo.
- Running: lệnh đang được thực hiện.
- Waiting: tiến trình đang chờ một số sự kiện xảy ra (chẳng hạn như hoàn thành I/O hoặc nhận được tín hiệu).
- Ready: tiến trình này đang chờ được gán cho một bộ xử lý.
- Terminated: tiến trình đã kết thúc thực hiện.

Những trạng thái này có tên là tùy ý và có thể khác nhau giữa các hệ điều hành. Tuy nhiên, bảng chất các trạng thái có trên tất cả các hệ thống. Một số hệ điều hành cũng phân định rõ ràng hơn các trạng thái tiến trình. Điều quan trọng là phải nhận ra rằng chỉ có một tiến trình có thể chạy trên bất kỳ lõi bộ xử lý nào tại bất kỳ thời điểm nào. Tuy nhiên, nhiều tiến trình có thể đã sẵn sàng và đang chờ đợi. Biểu đồ trạng thái tương ứng với các trạng thái này được trình bày trong Hình 3.3.



**Hình 3.3. Sơ đồ trạng thái tiến trình.**

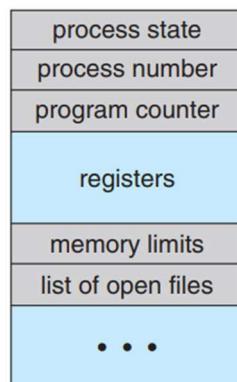
### 3.1.3. Khối điều khiển tiến trình (PCB)

Mỗi tiến trình được biểu diễn trong hệ điều hành bởi một khối điều khiển tiến trình (Process Control Block-PCB) - cũng được gọi là khối điều khiển tác vụ. Một PCB được thể hiện trong Hình 3.4. Nó chứa nhiều thông tin liên quan đến một tiến trình cụ thể, bao gồm:

- Trạng thái tiến trình. Trạng thái có thể là new, ready, running, waiting, v.v.
- Bộ đếm chương trình. Bộ đếm chỉ ra địa chỉ của lệnh tiếp theo sẽ được thực hiện cho tiến trình này.

- Các thanh ghi CPU. Các thanh ghi khác nhau về số lượng và kiểu, tùy thuộc vào kiến trúc máy tính. Chúng bao gồm bộ tích lũy, thanh ghi chỉ mục, con trỏ ngăn xếp và thanh ghi mục đích chung. Cùng với bộ đếm chương trình, thông tin trạng thái này phải được lưu khi xảy ra ngắt, để cho phép tiến trình được tiếp tục một cách chính xác sau đó khi nó được lập lịch chạy lại.
- Thông tin lập lịch CPU. Thông tin này bao gồm mức độ ưu tiên của tiến trình, con trỏ đến hàng đợi lập lịch và các tham số lập lịch khác.
- Thông tin quản lý bộ nhớ. Thông tin này có thể bao gồm các mục như giá trị của thanh ghi cơ sở và giới hạn và bảng trang hoặc bảng phân đoạn, tùy thuộc vào hệ thống bộ nhớ được sử dụng bởi hệ điều hành
- Thông tin kế toán. Thông tin này bao gồm lượng CPU và thời gian thực được sử dụng, giới hạn thời gian, số tài khoản, số công việc hoặc tiến trình, v.v.
- Thông tin trạng thái I/O. Thông tin này bao gồm danh sách các thiết bị I/O được phân bổ cho tiến trình, danh sách các tệp đang mở, v.v.

Tóm lại, PCB chỉ đơn giản đóng vai trò là kho lưu trữ tất cả dữ liệu cần thiết để bắt đầu hoặc khởi động lại một tiến trình, cùng với một số dữ liệu kế toán.



**Hình 3.4.** Khối điều khiển tiến trình.

### 3.2. Lập lịch tiến trình

Mục tiêu đa chương trình là luôn có một số tiến trình chạy để tối đa hóa việc sử dụng CPU. Mục tiêu chia sẻ thời gian là chuyển đổi lõi CPU giữa các tiến trình thường xuyên để người dùng có thể tương tác với từng chương trình khi chương trình đang chạy. Để đáp ứng hai mục tiêu này, bộ lập lịch chọn một tiến trình có sẵn (có thể từ tập hợp một số tiến trình có sẵn) để thực hiện trên một lõi. Mỗi lõi CPU có thể chạy một tiến trình tại một thời điểm.

Đối với một hệ thống có một lõi CPU, sẽ không bao giờ có nhiều hơn một tiến

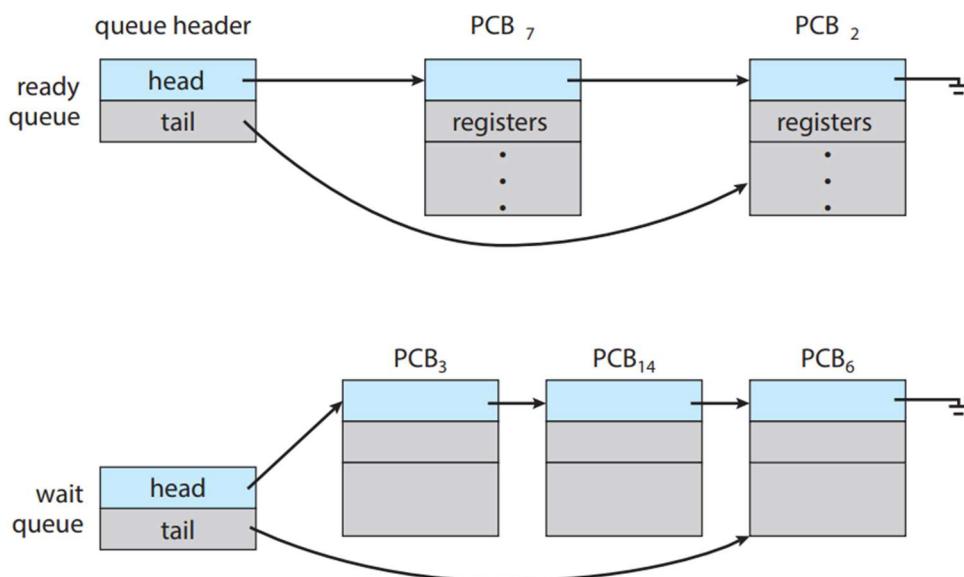
trình chạy cùng một lúc, trong khi một hệ thống đa lõi có thể chạy nhiều tiến trình cùng một lúc. Nếu có nhiều tiến trình hơn lõi, các tiến trình thừa sẽ phải đợi cho đến khi một lõi trống và có thể được lập lịch lại. Số lượng tiến trình hiện có trong bộ nhớ được gọi là mức độ đa chương trình.

Việc cân bằng các mục tiêu đa chương trình và chia sẻ thời gian cũng đòi hỏi phải tính đến hành vi chung của một tiến trình. Nói chung, hầu hết các tiến trình có thể được mô tả là liên kết I/O hoặc liên kết CPU. Tiến trình liên kết I/O (bound-I/O) là tiến trình dành nhiều thời gian để thực hiện I/O hơn là thời gian tính toán. Ngược lại, một tiến trình liên kết CPU (bound-CPU) tạo ra các yêu cầu I/O không thường xuyên, sử dụng nhiều thời gian hơn để thực hiện các phép tính.

### 3.2.1. Hàng đợi lập lịch

Khi các tiến trình vào hệ thống, chúng được đưa vào hàng đợi ready, nơi chúng sẵn sàng và chờ thực thi trên lõi của CPU. Hàng đợi này thường được lưu trữ dưới dạng danh sách liên kết; đầu hàng đợi ready chứa con trỏ đến PCB đầu tiên trong danh sách và mỗi PCB bao gồm một trường con trỏ trỏ đến PCB tiếp theo trong hàng đợi ready.

Hệ thống cũng bao gồm các hàng đợi khác. Khi một tiến trình được cấp phát một lõi CPU, nó sẽ thực thi một lúc và cuối cùng kết thúc, bị gián đoạn hoặc đợi sự xuất hiện của một sự kiện (chẳng hạn như việc hoàn thành yêu cầu I/O). Giả sử tiến trình thực hiện yêu cầu I/O tới một thiết bị bị chia hạn như đĩa. Vì thiết bị này chạy chậm hơn so với bộ xử lý, tiến trình sẽ phải đợi I/O khả dụng. Các tiến trình đang chờ một sự kiện xảy ra - chia hạn như hoàn thành I/O - được đặt trong hàng đợi wait (Hình 3.5).

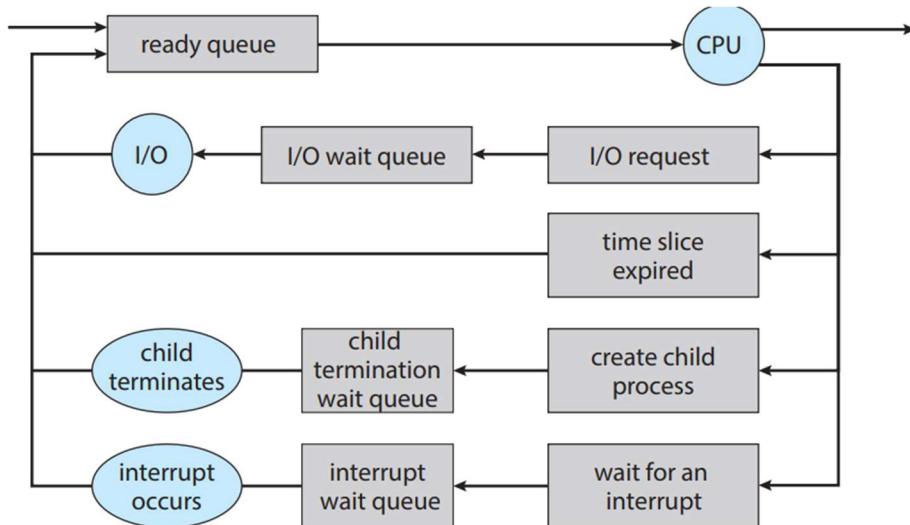


**Hình 3.5. Hàng đợi ready và hàng đợi wait**

Một mô tả phổ biến của việc lập lịch là một sơ đồ hàng đợi, chẳng hạn như trong Hình 3.6. Có hai loại hàng đợi: hàng đợi ready và hàng đợi wait. Các hình tròn đại diện cho các tài nguyên phục vụ hàng đợi và các mũi tên biểu thị di chuyển tiến trình trong hệ thống.

Một tiến trình mới được đưa vào hàng đợi ready. Nó đợi ở đó cho đến khi nó được chọn để thực thi hoặc được gửi đi. Khi tiến trình được cấp phát lõi CPU và thực thi, có thể xảy ra một số các sự kiện:

- Tiến trình có thể đưa ra yêu cầu I/O và sau đó được đưa vào hàng đợi wait I/O.
- Tiến trình có thể tạo một tiến trình con mới và sau đó được đặt trong hàng đợi wait trong khi chờ đợi tiến trình con chấm dứt.
- Tiến trình có thể bị loại bỏ cưỡng bức khỏi lõi, do ngắt hoặc hết thời gian và được đưa trở lại hàng đợi ready.



**Hình 3.6.** Sơ đồ hàng đợi cho các tiến trình.

Trong hai trường hợp đầu tiên, tiến trình cuối cùng chuyển từ trạng thái waiting sang trạng thái ready và sau đó được đưa trở lại hàng đợi ready. Một tiến trình tiếp tục chu kỳ này cho đến khi nó kết thúc, tại thời điểm đó nó được xóa khỏi tất cả các hàng đợi, PCB và tài nguyên được phân bổ.

### 3.2.2. Lập lịch CPU

Một tiến trình di chuyển giữa hàng đợi ready và hàng đợi wait khác nhau trong suốt thời gian tồn tại của nó. Vai trò của bộ lập lịch CPU là chọn trong số các tiến trình nằm trong hàng đợi ready và phân bổ lõi CPU cho một trong số chúng. Bộ lập lịch CPU

phải thường xuyên chọn một tiến trình mới cho CPU. Tiến liên kết I/O chỉ có thể thực thi trong vài mili giây trước khi đợi yêu cầu I/O. Mặc dù tiến trình liên kết CPU sẽ yêu cầu lõi CPU dài hơn, nhưng bộ lập lịch không có khả năng cấp lõi cho tiến trình trong một thời gian dài. Thay vào đó, nó được thiết kế để buộc loại bỏ tiến trình ra khỏi CPU và lập lịch cho một tiến trình khác chạy. Do đó, bộ lập lịch CPU thực thi ít nhất một lần sau mỗi 100 mili giây.

Một số hệ điều hành có dạng lập lịch trung gian, được gọi là swapping, ý tưởng chính của nó là xóa một tiến trình khỏi bộ nhớ (và khôi phục trạng thái của nó) và đưa nó vào bộ nhớ và thực thi tiếp tục ở điểm nó đã dừng lại. Cơ chế này được gọi là hoán đổi vì một tiến trình có thể được “hoán đổi” từ bộ nhớ sang đĩa và sau đó được “hoán đổi” từ đĩa trở lại bộ nhớ. Việc hoán đổi thường chỉ cần thiết khi bộ nhớ đã được cấp quá nhiều và phải được giải phóng.

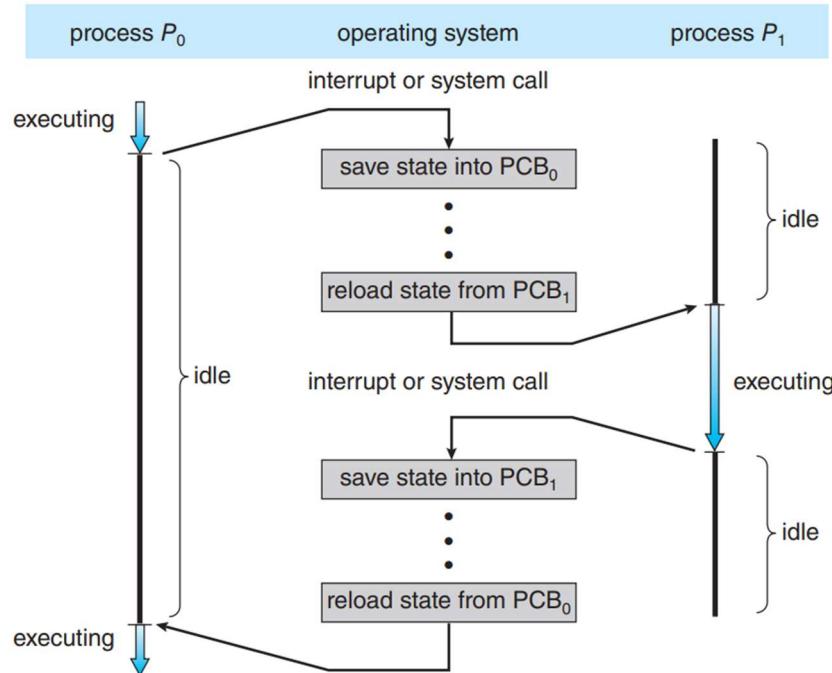
### 3.2.3. Chuyển đổi ngữ cảnh

Như đã đề cập, ngắt khiến hệ điều hành thay đổi lõi CPU từ tiến trình hiện tại và chạy một tiến trình nhân. Các hoạt động như vậy xảy ra thường xuyên trên các hệ thống. Khi xảy ra ngắt, hệ thống cần lưu ngữ cảnh hiện tại của tiến trình đang chạy trên lõi CPU để có thể khôi phục ngữ cảnh đó khi tiến trình được thực hiện lại, về cơ bản là tạm dừng tiến trình và sau đó tiếp tục lại. Ngữ cảnh được thể hiện trong PCB của tiến trình. Nó bao gồm giá trị của các thanh ghi CPU, trạng thái tiến trình (xem Hình 3.3) và thông tin quản lý bộ nhớ. Nói chung, ta cần thực hiện lưu trạng thái hiện tại của lõi CPU, có thể là ở chế độ nhân hoặc chế độ người dùng, sau đó khôi phục trạng thái để tiếp tục hoạt động.

Việc chuyển lõi CPU sang một tiến trình khác yêu cầu thực hiện lưu trạng thái của tiến trình hiện tại và khôi phục trạng thái của một tiến trình khác. Tác vụ này được biết đến như chuyển ngữ cảnh và được minh họa trong Hình 3.7. Khi một chuyển ngữ cảnh xảy ra, nhân sẽ lưu ngữ cảnh của tiến trình cũ trong PCB của nó và tải ngữ cảnh đã lưu của tiến trình mới được lập lịch chạy. Thời gian chuyển đổi liên tục là lãng phí. Tốc độ chuyển đổi khác nhau giữa các máy, tùy thuộc vào tốc độ bộ nhớ, số lượng thanh ghi phải được sao chép và sự tồn tại của các lệnh đặc biệt (chẳng hạn như một lệnh duy nhất để tải hoặc lưu trữ tất cả các thanh ghi). Tốc độ điển hình là vài micro giây.

Thời gian chuyển ngữ cảnh phụ thuộc nhiều vào phần cứng. Ví dụ, một số bộ xử lý cung cấp nhiều bộ thanh ghi. Chuyển ngữ cảnh ở đây chỉ đơn giản là yêu cầu thay đổi con trỏ đến tập thanh ghi hiện tại. Nếu có nhiều tiến trình hoạt động hơn tập thanh ghi,

hệ thống sẽ sao chép dữ liệu thanh ghi vào/từ bộ nhớ. Ngoài ra, hệ điều hành càng phức tạp thì khối lượng công việc phải thực hiện trong tiến trình chuyển ngữ cảnh càng lớn.



Hình 3.7. Sơ đồ chuyển đổi ngữ cảnh.

### 3.3. Thao tác trên tiến trình

Các tiến trình trong hầu hết các hệ thống có thể thực thi đồng thời và có thể được tạo và xóa động. Do đó, các hệ thống phải cung cấp một cơ chế để tạo và kết thúc tiến trình. Trong phần này, chúng ta tìm hiểu các cơ chế liên quan đến việc tạo tiến trình và minh họa việc tạo tiến trình trên hệ điều hành Windows.

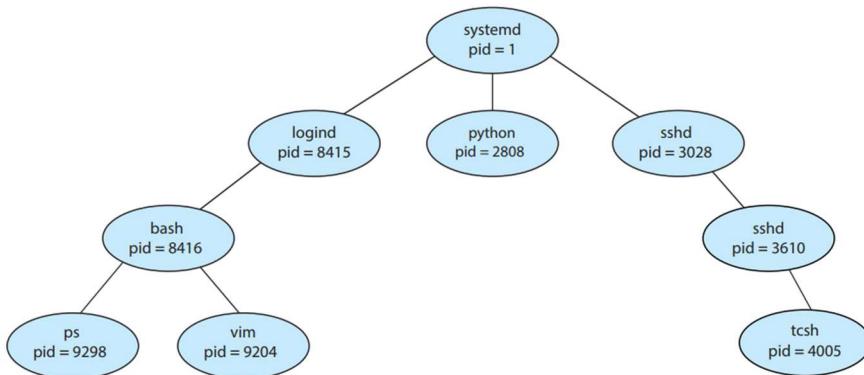
#### 3.3.1. Tạo tiến trình

Trong quá trình thực thi, một tiến trình có thể tạo ra một số tiến trình mới. Tiến trình tạo được gọi là tiến trình cha/mẹ, và các tiến trình mới được gọi là tiến trình con của tiến trình đó. Mỗi tiến trình mới này có thể lần lượt tạo ra các tiến trình khác, hình thành một cây tiến trình.

Hầu hết các hệ điều hành (bao gồm UNIX, Linux và Windows) xác định các tiến trình theo một mã nhận dạng tiến trình duy nhất (hoặc pid), thường là một số nguyên. Pid cung cấp giá trị duy nhất cho mỗi tiến trình trong hệ thống và nó có thể được sử dụng như một chỉ mục để truy cập các thuộc tính khác nhau của một tiến trình trong nhân.

Hình 3.8 minh họa một cây tiến trình điển hình của hệ điều hành Linux, cho thấy tên của mỗi tiến trình và pid của nó. Tiến trình systemd (luôn có pid là 1) đóng vai trò

là tiến trình gốc cho tất cả các tiến trình của người dùng và là tiến trình người dùng đầu tiên được tạo khi hệ thống khởi động. Khi hệ thống đã khởi động, tiến trình systemd tạo ra các tiến trình cung cấp các dịch vụ bổ sung như web, máy in, máy chủ ssh, v.v. Trong Hình 3.8, ta thấy hai con của systemd - logind và sshd. Tiến trình logind chịu trách nhiệm quản lý các máy khách đăng nhập trực tiếp vào hệ thống. Trong ví dụ này, một khách đã đăng nhập và đang sử dụng bash shell, đã được gán pid 8416. Sử dụng giao diện dòng lệnh bash, người dùng này đã tạo tiến trình ps cũng như trình soạn thảo vim. Tiến trình sshd chịu trách nhiệm quản lý các máy khách kết nối với hệ thống bằng cách sử dụng ssh (viết tắt của secure shell).



**Hình 3.8.** Cây tiến trình trong hệ điều hành Linux.

Nói chung, khi một tiến trình tạo ra một tiến trình con, tiến trình con sẽ cần một số tài nguyên nhất định (thời gian CPU, bộ nhớ, tệp, thiết bị I/O) để hoàn thành nhiệm vụ của nó. Tiến trình con có thể lấy tài nguyên trực tiếp từ hệ điều hành, hoặc nó có thể bị hạn chế trong một tập hợp con các tài nguyên của tiến trình mẹ. Tiến trình mẹ có thể phải phân vùng tài nguyên của nó giữa các tiến trình con, hoặc nó có thể chia sẻ một số tài nguyên (như bộ nhớ hoặc tệp) giữa một số tiến trình con. Việc hạn chế một tiến trình con đối với một tập hợp con tài nguyên của cha mẹ sẽ ngăn không cho bất kỳ tiến trình nào làm hệ thống quá tải bằng do tạo ra quá nhiều tiến trình con.

Ngoài việc cung cấp các tài nguyên vật lý và logic khác nhau, tiến trình mẹ có thể truyền dữ liệu khởi tạo (đầu vào) cho tiến trình con.

Khi một tiến trình tạo ra một tiến trình mới, tồn tại hai khả năng thực thi:

1. Tiến trình mẹ tiếp tục thực hiện đồng thời với các con của nó.
2. Tiến trình mẹ đợi cho đến khi một số hoặc tất cả các con của nó họ kết thúc.

Ngoài ra còn có hai khả năng không gian bộ nhớ cho tiến trình mới:

1. Tiến trình con là trùng với tiến trình mẹ (nó có cùng vùng text và data với tiến

trình mẹ).

2. Tiền trình con có một địa chỉ mới.

### 3.3.2. Kết thúc tiền trình

Một tiền trình kết thúc khi nó thực hiện câu lệnh cuối cùng và yêu cầu hệ điều hành xóa nó bằng cách sử dụng lệnh gọi hệ thống exit(). Tại thời điểm đó, tiền trình có thể trả về một giá trị trạng thái (thường là một số nguyên) cho tiền trình mẹ của nó ( thông qua lệnh gọi hệ thống wait()). Tất cả các tài nguyên của tiền trình - bao gồm bộ nhớ vật lý và ảo, các tệp đang mở và bộ đệm I/O - sẽ bị thu hồi.

Việc kết thúc tiền trình cũng có thể xảy ra trong trường hợp khác. Tiền trình có thể kết thúc tiền trình khác thông qua lệnh gọi hệ thống thích hợp (ví dụ: TerminaProcess() trong Windows). Thông thường, lệnh gọi hệ thống như vậy chỉ có thể được gọi bởi tiền trình mẹ. Nếu không, người dùng - hoặc một ứng dụng hoạt động sai - có thể tự ý kết thúc tiền trình của người dùng khác. Lưu ý rằng tiền trình mẹ cần biết định danh của những tiền trình con để kết thúc chúng. Do đó, khi một tiền trình tạo ra một tiền trình mới, định danh của tiền trình mới được tạo sẽ được chuyển cho tiền trình mẹ.

Tiền trình mẹ có thể kết thúc tiền trình con của mình vì nhiều lý do khác nhau, chẳng hạn như sau:

- Tiền trình con đã sử dụng quá mức một số tài nguyên mà nó đã được cấp phát. (Để xác định xem điều này có xảy ra hay không, tiền trình mẹ phải có cơ chế kiểm tra tình trạng của con cái mình.)
- Nhiệm vụ được giao tiền trình con không còn cần thiết.
- Tiền trình mẹ đang thoát và hệ điều hành không cho phép tiền trình con tiếp tục nếu tiền trình mẹ kết thúc.

Một số hệ thống không cho phép tiền trình con tồn tại nếu tiền trình mẹ của nó đã kết thúc. Trong các hệ thống như vậy, nếu một tiền trình kết thúc (bình thường hoặc bất thường), thì tất cả các tiền trình con phải kết thúc. Hiện tượng này, được gọi là kết thúc theo tầng, thường do hệ điều hành thực hiện.

Khi một tiền trình kết thúc, tài nguyên của nó sẽ được thu hồi bởi hệ điều hành. Tuy nhiên, điểm vào (entry) của nó trong bảng tiền trình phải duy trì cho đến khi tiền trình mẹ gọi hàm wait(). Một tiền trình đã kết thúc, nhưng tiền trình mẹ của nó vẫn chưa gọi wait(), được gọi là một tiền trình “ma” (zombie). Tất cả các tiền trình chuyển sang trạng thái exit khi chúng kết thúc, nhưng nhìn chung chúng chỉ tồn tại dưới dạng “ma” trong thời gian ngắn. Sau khi gọi wait(), mã định danh tiền trình của tiền trình ma và

mục nhập của nó trong bảng tiến trình sẽ được giải phóng.

### 3.4. Giao tiếp giữa tiến trình

Các tiến trình thực thi đồng thời trong hệ điều hành có thể là các tiến trình độc lập hoặc các tiến trình hợp tác. Một tiến trình là độc lập nếu nó không chia sẻ dữ liệu với bất kỳ tiến trình nào khác đang thực thi trong hệ thống. Một tiến trình là hợp tác nếu nó có thể tác động hoặc bị ảnh hưởng bởi các tiến trình khác đang thực thi trong hệ thống. Rõ ràng, bất kỳ tiến trình nào chia sẻ dữ liệu với các tiến trình khác đều là tiến trình hợp tác.

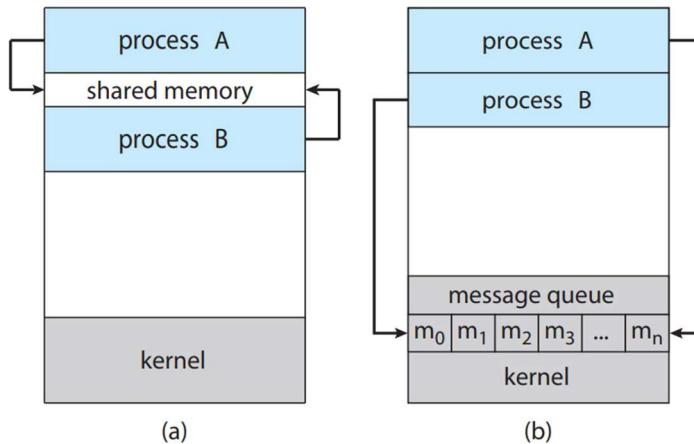
Có một số lý do để cung cấp một môi trường cho phép hợp tác tiến trình:

- Chia sẻ thông tin. Vì một số ứng dụng có thể quan tâm đến cùng một phần thông tin (ví dụ: sao chép và dán), chúng ta phải cung cấp một môi trường để cho phép truy cập đồng thời vào thông tin đó.
- Tăng tốc độ tính toán. Nếu chúng ta muốn một nhiệm vụ cụ thể chạy nhanh hơn, ta phải chia nó thành các nhiệm vụ con, mỗi nhiệm vụ sẽ được thực hiện song song với các nhiệm vụ khác. Lưu ý rằng tốc độ tăng như vậy chỉ có thể đạt được nếu máy tính có nhiều lõi xử lý.
- Tính mô đun. Chúng ta có thể muốn xây dựng hệ thống theo kiểu mô-đun, chia các chức năng của hệ thống thành các tiến trình hoặc luồng riêng biệt.

Các tiến trình hợp tác yêu cầu cơ chế giao tiếp giữa các tiến trình (Interprocess Communication - IPC) cho phép chúng trao đổi dữ liệu - nghĩa là gửi dữ liệu đến và nhận dữ liệu từ nhau. Có hai mô hình cơ bản của giao tiếp giữa các tiến trình: bộ nhớ chia sẻ và truyền thông điệp. Trong mô hình bộ nhớ chia sẻ, một vùng bộ nhớ được chia sẻ bởi các tiến trình hợp tác được thiết lập. Các tiến trình sau đó có thể trao đổi thông tin bằng cách đọc và ghi dữ liệu vào vùng được chia sẻ. Trong mô hình truyền thông điệp, giao tiếp diễn ra bằng cách thông điệp được trao đổi giữa các tiến trình hợp tác. Hai mô hình truyền thông tương phản trong Hình 3.9.

Cả hai mô hình vừa đề cập đều phổ biến trong các hệ điều hành và nhiều hệ thống cài đặt cả hai. Truyền thông điệp rất hữu ích để trao đổi lượng dữ liệu nhỏ, vì không cần cơ chế tránh xung đột. Truyền thông điệp cũng dễ dàng cài đặt hơn trong hệ thống phân tán so với chia sẻ bộ nhớ. Chia sẻ bộ nhớ có thể nhanh hơn truyền thông điệp, vì hệ thống truyền thông điệp thường được cài đặt bằng cách sử dụng các lệnh gọi hệ thống và do đó yêu cầu tác vụ can thiệp ở nhân tố nhiều thời gian hơn. Trong các hệ thống chia sẻ bộ nhớ, các lệnh gọi hệ thống chỉ được yêu cầu để thiết lập các vùng bộ nhớ

dùng chung. Khi bộ nhớ chia sẻ được thiết lập, tất cả các truy cập được coi là truy cập bộ nhớ thông thường và không cần sự hỗ trợ từ nhân.



**Hình 3.9.** Mô hình giao tiếp. (a) Chia sẻ bộ nhớ. (b) truyền thông điệp.

### 3.4.1. Cơ chế chia sẻ bộ nhớ

Giao tiếp giữa các tiến trình sử dụng bộ nhớ dùng chung yêu cầu các tiến trình giao tiếp thiết lập một vùng bộ nhớ dùng chung. Thông thường, vùng bộ nhớ dùng chung nằm trong không gian địa chỉ của tiến trình tạo bộ nhớ dùng chung. Các tiến trình khác muốn giao tiếp bằng cách sử dụng bộ nhớ dùng chung này phải gắn nó vào không gian địa chỉ của chúng. Nhớ lại rằng, thông thường, hệ điều hành có gắng ngăn một tiến trình truy cập vào bộ nhớ của tiến trình khác. Bộ nhớ dùng chung yêu cầu hai hoặc nhiều tiến trình đồng ý để loại bỏ hạn chế này. Sau đó, nó có thể trao đổi thông tin bằng cách đọc và ghi dữ liệu trong các khu vực được chia sẻ. Hình thức của dữ liệu và vị trí được xác định bởi các tiến trình này và không nằm trong sự kiểm soát của hệ điều hành. Các tiến trình cũng chịu trách nhiệm đảm bảo rằng chúng không được ghi đồng thời vào cùng một vị trí.

Để minh họa khái niệm về các tiến trình hợp tác, chúng ta hãy xem xét bài toán người sản xuất - người tiêu dùng, đây là một mô hình phổ biến cho các tiến trình hợp tác. Một tiến trình của người sản xuất tạo ra thông tin được tiêu thụ bởi một tiến trình của người tiêu dùng. Ví dụ, một trình biên dịch có thể tạo ra mã assembly được sử dụng bởi một trình hợp ngữ. Đến lượt mình, trình hợp ngữ có thể tạo ra các mô-đun đối tượng được sử dụng bởi trình nạp. Vẫn đề nhà sản xuất - người tiêu dùng cũng giống mô hình client - server. Chúng ta coi server một nhà sản xuất và một client là một người tiêu dùng. Ví dụ: một máy chủ web sản xuất (nghĩa là cung cấp) nội dung web như tệp HTML và hình ảnh, được sử dụng (nghĩa là đọc) bởi trình duyệt web của khách hàng yêu cầu tài nguyên.

Một giải pháp cho bài toán nhà sản xuất - người tiêu dùng sử dụng bộ nhớ chung. Để cho phép các tiến trình của nhà sản xuất và người tiêu dùng chạy đồng thời, chúng ta phải có sẵn một bộ đệm gồm các mặt hàng mà người sản xuất có thể đưa vào và người tiêu dùng lấy ra. Bộ đệm này sẽ nằm trong một vùng bộ nhớ được chia sẻ bởi các tiến trình của nhà sản xuất và người tiêu dùng. Nhà sản xuất có thể sản xuất một mặt hàng trong khi người tiêu dùng đang tiêu dùng một mặt hàng khác. Nhà sản xuất và người tiêu dùng phải được đồng bộ, để người tiêu dùng không cố gắng tiêu thụ một mặt hàng chưa được sản xuất.

Có thể sử dụng hai loại đệm. Bộ đệm không giới hạn không giới hạn kích thước. Người tiêu dùng có thể phải đợi các mặt hàng mới, nhưng người sản xuất luôn có thể sản xuất các mặt hàng mới. Bộ đệm giới hạn kích thước. Trong trường hợp này, người tiêu dùng phải đợi nếu bộ đệm trống và nhà sản xuất phải đợi nếu bộ đệm đầy.

Chúng ta hãy xem xét kỹ hơn cách bộ đệm giới hạn minh họa giao tiếp giữa các tiến trình sử dụng bộ nhớ dùng chung. Các biến sau đây nằm trong vùng bộ nhớ được chia sẻ bởi các tiến trình của nhà sản xuất và người tiêu dùng:

```
#define BUFFER_SIZE 10
typedef struct {
    ...
}item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Bộ đệm chia sẻ được cài đặt dưới dạng một mảng tròn với hai con trỏ logic: in và out. Biến in trỏ đến vị trí trống tiếp theo trong bộ đệm; out trỏ đến vị trí đầy đầu tiên trong bộ đệm. Bộ đệm trống khi `in == out`; bộ đệm đầy khi `((in + 1) % BUFFER_SIZE) == out`.

Mã cho tiến trình sản xuất được hiển thị trong Hình 3.10 và mã cho tiến trình tiêu dùng được hiển thị trong Hình 3.11. Tiến trình sản xuất có một biến cục bộ `next_produced` lưu trữ mục mới sẽ được sản xuất. Tiến trình tiêu dùng có một biến cục bộ `next_consumed` lưu trữ mặt hàng sẽ được tiêu thụ.

```
item next_produced;
while (true){
    /* produce an item in next_produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
```

```
    in = (in + 1) % BUFFER_SIZE;  
}
```

**Hình 3.10.** Tiến trình của nhà sản xuất.

```
item next_consumed;  
while (true){  
    while (in == out)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
  
    /* consume the item in next consumed */  
}
```

**Hình 3.11.** Tiến trình người tiêu dùng.

### 3.4.2. Cơ chế gởi thông điệp

Truyền thông điệp cung cấp một cơ chế cho phép các tiến trình giao tiếp và đồng bộ hóa các hành động của chúng mà không cần chia sẻ cùng một không gian địa chỉ. Nó đặc biệt hữu ích trong môi trường phân tán, nơi các tiến trình giao tiếp có thể nằm trên các máy tính khác nhau được kết nối bởi mạng. Ví dụ, một chương trình trò chuyện trên Internet có thể được thiết kế để những người tham gia trò chuyện giao tiếp với nhau bằng cách trao đổi tin nhắn.

Một cơ chế truyền thông báo cung cấp ít nhất hai hoạt động:

```
send(message)  
receive(message)
```

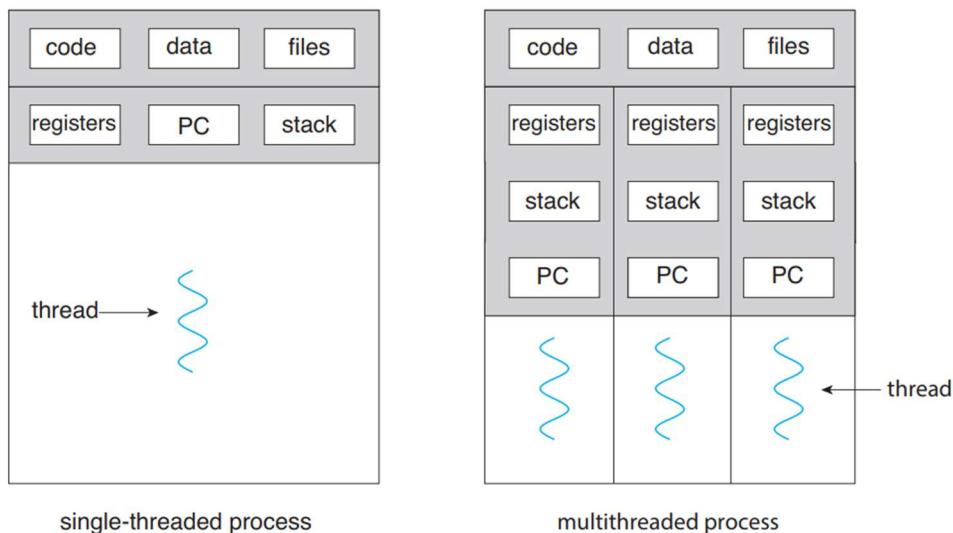
Thông điệp được gửi bởi một tiến trình có thể có kích thước cố định hoặc thay đổi. Nếu thông điệp có kích thước cố định, thì việc cài đặt cấp hệ thống đơn giản. Tuy nhiên, hạn chế này làm cho nhiệm vụ lập trình trở nên khó khăn hơn. Ngược lại, các thông điệp có kích thước thay đổi yêu cầu cài đặt cấp hệ thống phức tạp hơn, nhưng nhiệm vụ lập trình trở nên đơn giản hơn. Đây là một kiểu cân bằng phổ biến được thấy trong quá trình thiết kế hệ điều hành.

## 3.5. Luồng

### 3.5.1. Giới thiệu

Một luồng là một đơn vị cơ bản của việc sử dụng CPU; nó bao gồm một ID luồng, một bộ đếm chương trình (PC), một bộ thanh ghi và một ngăn xếp. Nó chia sẻ với các luồng khác thuộc cùng một tiến trình, phần mã, phần dữ liệu và các tài nguyên hệ điều

hành khác, chẳng hạn như các tệp và tín hiệu đang mở. Một tiến trình truyền thống có một luồng điều khiển duy nhất. Nếu một tiến trình có nhiều luồng điều khiển, thì nó có thể thực hiện nhiều tác vụ cùng một lúc. Hình 3.12 minh họa sự khác biệt giữa tiến trình đơn luồng truyền thống và tiến trình đa luồng.



**Hình 3.12. Đơn luồng và đa luồng.**

#### a. *Động lực thúc đẩy*

Hầu hết các ứng dụng phần mềm chạy trên máy tính và thiết bị di động hiện đại là đa luồng. Một ứng dụng thường được cài đặt như một tiến trình riêng biệt với một số luồng điều khiển. Dưới đây, chúng ta nêu một số ví dụ về các ứng dụng đa luồng:

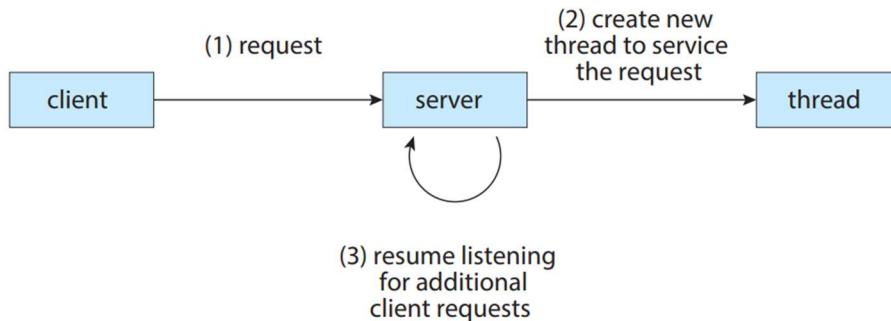
- Ứng dụng tạo hình thu nhỏ ảnh từ một bộ sưu tập hình ảnh có thể sử dụng một luồng riêng để tạo hình thu nhỏ từ mỗi hình ảnh riêng biệt.
- Trình duyệt web có thể có một luồng hiển thị hình ảnh hoặc văn bản trong khi một luồng khác truy xuất dữ liệu từ mạng.
- Bộ xử lý văn bản có thể có một luồng để hiển thị đồ họa, một luồng khác để phản hồi các thao tác gõ phím từ người dùng và một luồng thứ ba để thực hiện kiểm tra chính tả và ngữ pháp.

Các ứng dụng cũng có thể được thiết kế để tận dụng khả năng xử lý trên các hệ thống đa lõi. Các ứng dụng như vậy có thể thực hiện song song một số tác vụ đòi hỏi nhiều CPU trên nhiều lõi máy tính.

Trong một số tình huống nhất định, một ứng dụng có thể được yêu cầu thực hiện một số tác vụ tương tự nhau. Ví dụ, một máy chủ web nhận các yêu cầu của khách hàng đối với các trang web, hình ảnh, âm thanh, v.v. Một máy chủ web bạn có thể có một số

(có thể hàng nghìn) máy khách truy cập đồng thời. Nếu máy chủ web chạy như một tiến trình đơn luồng truyền thống, nó sẽ chỉ có thể phục vụ một máy khách tại một thời điểm và một máy khách có thể phải đợi rất lâu để yêu cầu của nó được phục vụ.

Một giải pháp là để máy chủ chạy như một tiến trình duy nhất nhận các yêu cầu. Khi máy chủ nhận được một yêu cầu, nó sẽ tạo ra một tiến trình riêng để phục vụ yêu cầu đó. Trên thực tế, phương pháp tạo tiến trình này đã được sử dụng phổ biến trước khi các luồng trở nên phổ biến. Tuy nhiên, việc tạo tiến trình tốn nhiều thời gian và tài nguyên. Nói chung sẽ hiệu quả hơn nếu sử dụng một tiến trình chứa nhiều luồng. Nếu tiến trình máy chủ web là đa luồng, máy chủ sẽ tạo một luồng riêng để lắng nghe các yêu cầu của máy khách. Khi một yêu cầu được thực hiện, thay vì tạo một tiến trình khác, máy chủ sẽ tạo một luồng mới để phục vụ yêu cầu và tiếp tục lắng nghe các yêu cầu bổ sung. Điều này được minh họa trong Hình 3.13.



**Hình 3.13. Kiến trúc máy chủ đa luồng.**

Nhiều ứng dụng cũng có thể tận dụng nhiều luồng, bao gồm các thuật toán sắp xếp, cây và đồ thị. Ngoài ra, các lập trình viên phải giải quyết các vấn đề về CPU song song trong khai thác dữ liệu, đồ họa và trí tuệ nhân tạo có thể tận dụng sức mạnh của các hệ thống đa lõi hiện đại bằng cách thiết kế các giải pháp chạy song song.

#### b. Lợi ích

Các lợi ích của lập trình đa luồng có thể được chia thành bốn loại chính:

1. Khả năng đáp ứng. Đa luồng một ứng dụng tương tác có thể cho phép một chương trình tiếp tục chạy ngay cả khi một phần của nó bị chặn hoặc đang thực hiện một thao tác kéo dài, do đó tăng khả năng phản hồi cho người dùng. Điều này đặc biệt hữu ích trong việc thiết kế giao diện người dùng. Ví dụ: xét tình huống người dùng nhấp vào một nút lệnh thực hiện một thao tác tốn nhiều thời gian. Một ứng dụng đơn luồng sẽ không phản hồi với người dùng cho đến khi hoạt động hoàn tất. Ngược lại, nếu thao tác tốn thời gian được thực hiện trong một luồng riêng biệt, không đồng bộ, ứng dụng vẫn phản hồi cho người dùng.

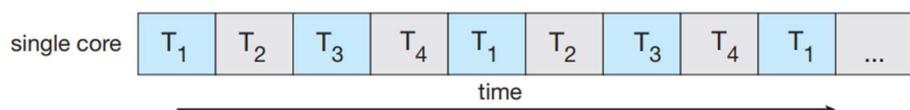
2. Chia sẻ tài nguyên. Các tiến trình chỉ có thể chia sẻ tài nguyên thông qua các kỹ thuật như bộ nhớ dùng chung và truyền thông điệp. Các kỹ thuật như vậy phải được người lập trình sắp xếp một cách rõ ràng. Tuy nhiên, các luồng chia sẻ bộ nhớ và tài nguyên của tiến trình mà chúng thuộc về theo mặc định. Lợi ích của việc chia sẻ mã và dữ liệu là nó cho phép một ứng dụng có nhiều luồng hoạt động khác nhau trong cùng một không gian địa chỉ.

3. Kinh tế. Việc phân bổ bộ nhớ và tài nguyên để tạo tiến trình rất tốn kém. Vì các luồng chia sẻ tài nguyên của tiến trình mà chúng thuộc về, nên việc tạo và các luồng chuyển đổi ngữ cảnh sẽ tiết kiệm hơn. Theo kinh nghiệm đánh giá sự khác biệt về chi phí có thể khó khăn, nhưng nói chung việc tạo luồng tiêu tốn ít thời gian và bộ nhớ hơn so với tạo tiến trình. Ngoài ra, chuyển đổi ngữ cảnh thường nhanh hơn giữa các luồng so với giữa các tiến trình.

4. Khả năng mở rộng. Lợi ích của đa luồng có thể còn lớn hơn trong kiến trúc đa xử lý, nơi các luồng có thể chạy song song trên các lõi xử lý khác nhau. Tiến trình đơn luồng chỉ có thể chạy trên một bộ xử lý, bất kể có bao nhiêu bộ xử lý.

### 3.5.2. Lập trình đa lõi

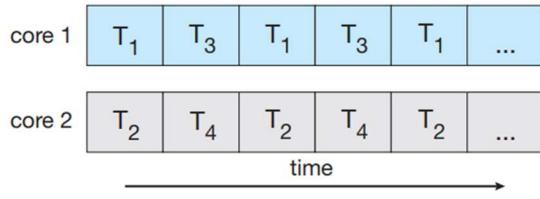
Trước đó trong lịch sử thiết kế máy tính, để đáp ứng nhu cầu về hiệu suất tính toán cao hơn, các hệ thống đơn CPU đã phát triển thành hệ thống đa CPU. Một xu hướng sau này, nhưng tương tự, trong thiết kế hệ thống là đặt nhiều lõi tính toán trên một chip xử lý duy nhất, nơi mỗi lõi xuất hiện như một CPU riêng biệt cho hệ điều hành. Hãy xem xét một ứng dụng có bốn luồng. Trên hệ thống có một lõi tính toán duy nhất, thực hiện đồng thời có nghĩa là việc thực thi các luồng sẽ được xen kẽ theo thời gian (Hình 3.14), vì lõi xử lý chỉ có thể thực thi một luồng tại một thời điểm. Tuy nhiên, trên một hệ thống có nhiều lõi, đồng thời có nghĩa là một số luồng có thể chạy song song, vì hệ thống có thể gán một luồng riêng cho mỗi lõi (Hình 3.15).



**Hình 3.14.** Thực hiện đồng thời trên hệ thống đơn nhân.

Lưu ý sự phân biệt giữa đồng thời và song song. Một hệ thống đồng thời hỗ trợ nhiều hơn một nhiệm vụ bằng cách cho phép tất cả các nhiệm vụ được thực hiện. Ngược lại, một hệ thống song song có thể thực hiện nhiều hơn một nhiệm vụ cùng thời điểm. Như vậy, có thể tồn tại đồng thời mà không có song song. Trước khi kiến trúc đa xử lý và đa lõi ra đời, hầu hết các hệ thống máy tính chỉ có một bộ xử lý duy nhất và các bộ

lập lịch CPU được thiết kế để cung cấp ảo tưởng về sự song song bằng cách chuyển đổi nhanh chóng giữa các tiến trình. Các tiến trình như vậy đã chạy đồng thời, nhưng không song song.



**Hình 3.15.** Thực hiện song song trên hệ thống đa nhân.

#### b. Thách thức lập trình

Xu hướng đối với các hệ thống đa lõi tiếp tục gây áp lực lên các nhà thiết kế hệ thống và lập trình ứng dụng để tận dụng tốt hơn nhiều lõi tính toán. Người thiết kế hệ điều hành phải viết các thuật toán lập lịch sử dụng nhiều lõi xử lý để cho phép thực hiện song song như trong Hình 3.15. Đối với các lập trình viên ứng dụng, thách thức là sửa đổi các chương trình hiện có cũng như thiết kế các chương trình mới đa luồng. Nhìn chung, năm thách thức trong lập trình cho các hệ thống đa lõi:

1. Xác định nhiệm vụ. Điều này liên quan đến việc kiểm tra các ứng dụng để tìm các lĩnh vực có thể được chia thành các nhiệm vụ riêng biệt, đồng thời. Lý tưởng nhất là các tác vụ độc lập với nhau và do đó có thể chạy song song trên các lõi riêng lẻ.

2. Cân bằng. Trong khi xác định các tác vụ có thể chạy song song, người lập trình cũng phải đảm bảo rằng các tác vụ thực hiện công việc bình đẳng có giá trị như nhau. Trong một số trường hợp, một nhiệm vụ nhất định có thể không đóng góp nhiều giá trị cho quá trình tổng thể như các nhiệm vụ khác. Sử dụng một lõi thực thi riêng biệt để chạy tác vụ đó có thể không đáng giá.

3. Tách dữ liệu. Cũng giống như các ứng dụng được chia thành các tác vụ riêng biệt, dữ liệu được các tác vụ truy cập và thao tác phải được phân chia để chạy trên các lõi riêng biệt.

4. Sự phụ thuộc dữ liệu. Dữ liệu được truy cập bởi các tác vụ phải được kiểm tra sự phụ thuộc giữa hai hoặc nhiều tác vụ. Khi một nhiệm vụ phụ thuộc vào dữ liệu từ một tác vụ khác, các lập trình viên phải đảm bảo rằng việc thực thi các tác vụ được đồng bộ hóa để phù hợp với sự phụ thuộc dữ liệu.

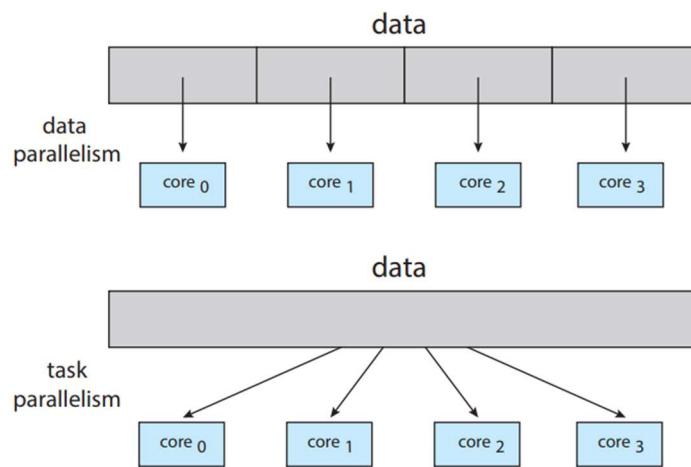
5. Kiểm tra và gỡ lỗi. Khi một chương trình đang chạy song song trên nhiều lõi, có thể có nhiều đường dẫn thực thi khác nhau. Việc kiểm tra và gỡ lỗi các chương trình đồng thời như vậy vốn đã khó hơn kiểm tra và gỡ lỗi các ứng dụng đơn luồng.

Vì những thách thức này, nhiều nhà phát triển phần mềm cho rằng sự ra đời của các hệ thống đa lõi sẽ đòi hỏi một cách tiếp cận hoàn toàn mới để thiết kế hệ thống phần mềm trong tương lai. (Tương tự, nhiều nhà giáo dục khoa học máy tính tin rằng phát triển phần mềm phải được giảng dạy với sự chú trọng nhiều hơn vào lập trình song song.)

### c. Các kiểu song song

Nói chung, có hai loại song song: song song dữ liệu và song song nhiệm vụ. Song song dữ liệu là việc phân phối các tập con của cùng một dữ liệu trên nhiều lõi tính toán và thực hiện cùng một hoạt động trên mỗi lõi. Ví dụ, hãy xem xét tính tổng một mảng có kích thước  $N$ . Trên hệ thống lõi đơn, một luồng chỉ đơn giản là tính tổng các phần tử  $[0] \dots [N-1]$ . Tuy nhiên, trên hệ thống lõi kép, luồng A, chạy trên lõi 0, có thể tính tổng các phần tử  $[0] \dots [N/2-1]$  trong khi luồng B, chạy trên lõi 1, có thể tính tổng các phần tử  $[N/2] \dots [N-1]$ . Hai luồng sẽ chạy song song trên các lõi máy tính riêng biệt.

Song song nhiệm vụ bao gồm việc phân phối không phải dữ liệu mà là các tác vụ (luồng) trên nhiều lõi máy tính. Mỗi luồng đang thực hiện một hoạt động duy nhất. Các luồng khác nhau có thể hoạt động trên cùng một dữ liệu hoặc chúng có thể hoạt động trên các dữ liệu khác nhau. Hãy xem xét lại ví dụ của chúng ta ở trên. Ngược lại với tình huống đó, một ví dụ về tính song song nhiệm vụ có thể liên quan đến hai luồng, mỗi luồng thực hiện một phép toán thống kê duy nhất trên mảng các phần tử. Các luồng lại đang hoạt động song song trên các lõi tính toán riêng biệt, nhưng mỗi lõi thực hiện một hoạt động duy nhất.



**Hình 3.16.** Song song dữ liệu và song song nhiệm vụ.

Về cơ bản, song song dữ liệu liên quan đến việc phân phối dữ liệu trên nhiều lõi và song song nhiệm vụ liên quan đến việc phân phối các tác vụ trên nhiều lõi, như thể hiện trong Hình 3.16. Tuy nhiên, tính song song của dữ liệu và nhiệm vụ không loại trừ

lẫn nhau và trên thực tế, một ứng dụng có thể sử dụng kết hợp hai chiến lược này.

### 3.6. Tóm tắt

- Tiến trình là một chương trình đang được thực thi, và trạng thái của hoạt động hiện tại của một tiến trình được biểu thị bằng bộ đếm chương trình, cũng như các thanh ghi khác.

- Bộ nhớ của một tiến trình trong bộ nhớ được thể hiện bằng bốn phần khác nhau: (1) văn bản, (2) dữ liệu, (3) heap và (4) stack.

- Khi một tiến trình thực thi, nó sẽ thay đổi trạng thái. Có bốn trạng thái chung của một tiến trình: (1) ready, (2) running, (3) waiting và (4) terminating.

- Khối điều khiển tiến trình (PCB) là cấu trúc dữ liệu nhân biểu diễn cho một tiến trình trong hệ điều hành.

- Vai trò của bộ lập lịch tiến trình là chọn một tiến trình có sẵn để chạy trên CPU.

- Hệ điều hành thực hiện chuyển đổi ngữ cảnh khi nó chuyển việc chạy một tiến trình này sang chạy một tiến trình khác.

- Lệnh gọi hệ thống fork() và CreateProcess() được sử dụng để tạo các tiến trình trên hệ thống UNIX và Windows tương ứng.

- Khi bộ nhớ dùng chung được sử dụng để liên lạc giữa các tiến trình, hai (hoặc nhiều) tiến trình chia sẻ cùng một vùng bộ nhớ.

- Hai tiến trình có thể giao tiếp bằng cách trao đổi thông điệp với nhau. Hệ điều hành Mach sử dụng truyền thông điệp giao tiếp giữa các tiến trình chính. Windows cũng cung cấp một dạng truyền thông báo.

- Một luồng (tiểu trình) đại diện cho một đơn vị cơ bản của việc sử dụng CPU và các luồng thuộc cùng một tiến trình chia sẻ nhiều tài nguyên của tiến trình, bao gồm mã và dữ liệu.

- Có bốn lợi ích chính đối với các ứng dụng đa luồng: (1) khả năng đáp ứng, (2) chia sẻ tài nguyên, (3) tính kinh tế và (4) khả năng mở rộng.

- Thuật ngữ đồng thời là nhiều luồng đang thực hiện, thuật ngữ song song là khi nhiều luồng đang thực hiện cùng thời điểm. Trên một hệ thống có một CPU, chỉ có thể sử dụng đồng thời; song song xuất hiện trong hệ thống đa lõi cung cấp nhiều CPU.

- Có một số thách thức trong việc thiết kế các ứng dụng đa luồng. Chúng bao gồm phân chia và cân bằng công việc, phân chia dữ liệu giữa các luồng khác nhau và xác định bất kỳ phụ thuộc dữ liệu nào. Cuối cùng, các chương trình đa luồng đặc biệt khó

kiểm tra và gỡ lỗi.

- Song song dữ liệu phân phối các tập con của cùng một dữ liệu trên các lõi tính toán khác nhau và thực hiện cùng một hoạt động trên mỗi lõi. Song song nhiệm vụ phân phối không phải dữ liệu mà là các tác vụ trên nhiều lõi. Mỗi nhiệm vụ đang chạy một hoạt động duy nhất.



## CHƯƠNG 4. LẬP LỊCH TIẾN TRÌNH

### 4.1. Các khái niệm cơ bản

Trong một hệ thống có một lõi CPU, chỉ một tiến trình có thể chạy tại một thời điểm. Những tiến trình khác phải đợi cho đến khi lõi của CPU trống và có thể được lập lịch lại. Mục tiêu của đa chương trình là luôn có một số tiến trình chạy, để tối đa hóa việc sử dụng CPU. Ý tưởng là tương đối đơn giản. Một tiến trình được thực thi cho đến khi nó phải đợi, thường là để hoàn thành một số yêu cầu I/O. Trong một hệ thống máy tính đơn giản, CPU chỉ ở chế độ chờ. Tất cả thời gian chờ đợi này bị lãng phí; không có công việc hữu ích nào được thực hiện. Với đa chương trình, chúng ta cố gắng sử dụng thời gian này một cách hiệu quả. Một số tiến trình được lưu trong bộ nhớ cùng một lúc. Khi một tiến trình phải chờ, hệ điều hành sẽ đưa CPU ra khỏi tiến trình đó và đưa CPU cho một tiến trình khác. Trên hệ thống đa lõi, khái niệm giữ CPU bận rộn này được mở rộng cho tất cả các lõi xử lý trên hệ thống.

Lập lịch là một chức năng cơ bản của hệ điều hành. Hầu hết tất cả các tài nguyên máy tính đều được lập lịch trước khi sử dụng. Tất nhiên, CPU là một trong những tài nguyên máy tính chính. Do đó, lập lịch CPU là trọng tâm của thiết kế hệ điều hành.

#### 4.1.1. Chu kỳ CPU và chu kỳ I/O

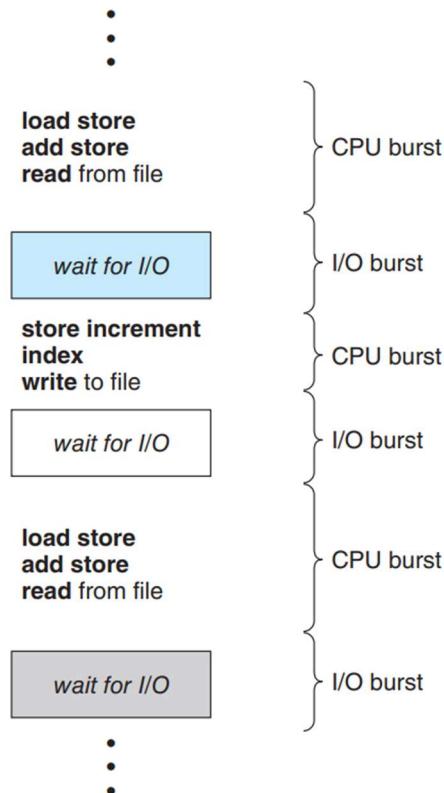
Lập lịch CPU phụ thuộc vào tính chất sau của tiến trình: thực thi tiến trình bao gồm một chu kỳ thực thi CPU và chờ I/O. Các tiến trình luân phiên giữa hai trạng thái này. Thực thi tiến trình bắt đầu bằng một chu kỳ CPU. Sau đó là một loạt I/O, sau đó là một loạt CPU khác, sau đó là một loạt I/O khác, v.v. Cuối cùng, chu kỳ CPU cuối cùng kết thúc với yêu cầu hệ thống chấm dứt thực thi (Hình 4.1).

#### 4.1.2. Bộ lập lịch CPU (scheduler)

Bất cứ khi nào CPU rỗi, hệ điều hành phải chọn một trong các tiến trình trong hàng đợi sẵn sàng để được thực thi. Quá trình lựa chọn được thực hiện bởi bộ lập lịch CPU, bộ lập lịch này sẽ chọn một tiến trình từ các tiến trình trong bộ nhớ sẵn sàng thực thi và cấp phát CPU cho tiến trình đó.

Lưu ý rằng hàng đợi sẵn sàng không nhất thiết phải là hàng đợi nhập trước FIFO,

hàng đợi sẵn sàng có thể được cài đặt dưới dạng hàng đợi FIFO, hàng đợi ưu tiên, cây hoặc đơn giản là danh sách liên kết không có thứ tự. Tuy nhiên, về mặt khái niệm, tất cả các tiến trình trong hàng đợi sẵn sàng được xếp hàng chờ cơ hội chạy trên CPU. Các bản ghi trong hàng đợi thường là các khối điều khiển tiến trình (PCB) của các tiến trình.



**Hình 4.1.** Chuỗi thực thi CPU và chờ I/O.

#### 4.1.3. Lập lịch ưu tiên và không ưu tiên

Các quy tắc lập lịch là ưu tiên hoặc không ưu tiên. Lập lịch là không ưu tiên (độc quyền - Nonpreemptive) nếu, một khi hệ thống đã chỉ định một bộ xử lý cho một tiến trình, hệ thống không thể xóa bộ xử lý đó khỏi tiến trình đó. Lập lịch ưu tiên (hoặc không độc quyền - Preemptive) nếu hệ thống có thể xóa bộ xử lý khỏi tiến trình nó đang chạy. Theo chế độ lập lịch không ưu tiên, mỗi tiến trình, khi được cung cấp cho một bộ xử lý, sẽ chạy đến khi hoàn thành hoặc cho đến khi nó tự nguyện từ bỏ bộ xử lý của mình. Theo chế độ lập lịch ưu tiên, bộ xử lý có thể thực thi một phần mã của tiến trình và sau đó thực hiện chuyển đổi ngữ cảnh.

Lập lịch ưu tiên rất hữu ích trong các hệ thống trong đó các tiến trình có mức độ ưu tiên cao yêu cầu phản hồi nhanh chóng. Ví dụ, trong các hệ thống thời gian thực không đáp ứng với một ngắt có thể có hậu quả lớn. Trong các hệ thống chia sẻ thời gian tương tác, lập lịch ưu tiên giúp đảm bảo thời gian phản hồi của người dùng có thể chấp

nhận được. Lập lịch ưu tiên chi phí chuyển đổi ngữ cảnh. Để thực hiện quyền ưu tiên hiệu quả, hệ thống phải duy trì nhiều tiến trình trong bộ nhớ chính, để tiến trình tiếp theo sẵn sàng khi có bộ xử lý.

Trong lập lịch không ưu tiên, các tiến trình ngắn có thể chờ đợi lâu trong khi các tiến trình dài hơn hoàn thành, nhưng thời gian quay vòng dễ dự đoán hơn, bởi vì các tiến trình có mức độ ưu tiên cao không thể thay thế các tiến trình đang chờ. Bởi vì một hệ thống không có tính ưu tiên không thể xóa một tiến trình khỏi bộ xử lý cho đến khi nó hoàn thành, các chương trình sai sót không bao giờ hoàn thành (ví dụ: bằng cách nhập một vòng lặp vô hạn) có thể không bao giờ từ bỏ quyền kiểm soát hệ thống. Ngoài ra, trong lập lịch không ưu tiên, việc thực thi các tiến trình không quan trọng có thể khiến các tiến trình quan trọng phải chờ đợi.

Để ngăn người dùng độc quyền hệ thống (dù là do cố ý hoặc vô tình), lập lịch ưu tiên tạm dừng một tiến trình. Điều này thường được thực hiện bằng cách đặt đồng hồ ngắt để tạo ra một ngắt định kỳ. Khi một bộ xử lý được gán cho một tiến trình, tiến trình sẽ thực thi cho đến khi nó tự nguyện giải phóng bộ xử lý, hoặc cho đến khi ngắt đồng hồ hoặc một số ngắt khác xảy ra. Sau đó, hệ điều hành có thể quyết định xem tiến trình đang chạy có nên tiếp tục hay một số tiến trình "tiếp theo" khác sẽ thực thi.

Ngắt đồng hồ giúp đảm bảo thời gian phản hồi hợp lý cho người dùng, ngăn hệ thống bị treo đói với một vòng lặp vô hạn, và cho phép các tiến trình phản hồi các sự kiện phụ thuộc vào thời gian. Các tiến trình cần chạy định kỳ phụ thuộc vào đồng hồ ngắt.

Hầu như tất cả các hệ điều hành hiện đại như Windows, macOS, Linux và UNIX đều sử dụng các thuật toán lập lịch ưu tiên.

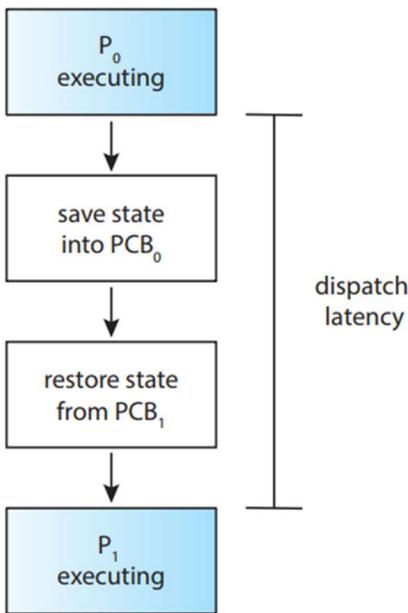
#### 4.1.4. Bộ điều phối (Dispatcher)

Một thành phần khác tham gia vào chức năng lập lịch CPU là bộ điều phối. Bộ điều phối là mô-đun cung cấp quyền kiểm soát lõi CPU đối với tiến trình do bộ lập lịch CPU chọn. Chức năng này bao gồm những thao tác sau:

- Chuyển ngữ cảnh từ tiến trình này sang tiến trình khác.
- Chuyển sang chế độ người dùng.
- Chuyển đến vị trí thích hợp trong chương trình người dùng để tiếp tục chương trình đó.

Bộ điều phối phải nhanh nhất có thể, vì nó được gọi trong mọi lần chuyển đổi ngữ cảnh. Thời gian để bộ điều phối dừng một tiến trình và bắt đầu một tiến trình chạy khác

được gọi là độ trễ điều phối và được minh họa trong Hình 4.2.



**Hình 4.2.** Vai trò của bộ điều phối.

## 4.2. Tiêu chuẩn lập lịch

Các thuật toán lập lịch trình CPU khác nhau có các thuộc tính khác nhau và việc chọn một thuật toán cụ thể phụ thuộc vào nhóm tiến trình cụ thể. Nhiều tiêu chí đã được đề xuất để so sánh các thuật toán lập lịch CPU. Các tiêu chí bao gồm những điều sau:

- Sử dụng CPU. Chúng ta giữ cho CPU bận rộn nhất có thể. Về mặt khái niệm, việc sử dụng CPU có thể nằm trong khoảng từ 0 đến 100%. Trong một hệ thống thực, nó sẽ nằm trong khoảng từ 40% (đối với hệ thống được tải nhẹ) đến 90% (đối với hệ thống được tải nặng).
- Thông lượng. Là số lượng tiến trình được hoàn thành trên một đơn vị thời gian.
- Thời gian quay vòng. Khoảng thời gian từ khi gửi tiến trình đến khi hoàn thành là thời gian quay vòng. Thời gian quay vòng là tổng khoảng thời gian dành cho việc chờ đợi trong hàng đợi sẵn sàng, thực thi trên CPU và thực hiện I/O.
- Thời gian chờ. Thuật toán lập lịch trình CPU không tác động đến lượng thời gian tiến trình thực thi hoặc thực hiện I/O. Nó chỉ tác động đến lượng thời gian mà một tiến trình chờ trong hàng đợi sẵn sàng.
- Thời gian đáp ứng. Là thời gian từ khi gửi yêu cầu cho đến khi phản hồi đầu tiên được đưa ra.

### 4.3. Thuật toán lập lịch

#### 4.3.1. First-In-First-Out (FIFO) Scheduling

Có lẽ thuật toán lập lịch đơn giản nhất là FIFO (có tên khác là FCFS), đó là tiến trình nào vào trước được phục vụ trước. Các tiến trình lần lượt được gửi đến hàng đợi sẵn sàng. FIFO là cơ chế lập lịch độc quyền - khi một tiến trình có bộ xử lý, thì tiến trình đó sẽ chạy cho đến lúc hoàn thành. FIFO công bằng ở chỗ, nó lập lịch các tiến trình theo thời gian đến của chúng, vì vậy tất cả các tiến trình đều được đối xử bình đẳng, nhưng hơi không công bằng vì các tiến trình dài khiến các tiến trình ngắn phải chờ, và các tiến trình không quan trọng khiến các tiến trình quan trọng phải chờ. FIFO không hữu ích trong việc lập lịch các tiến trình tương tác vì nó không thể đảm bảo thời gian phản hồi ngắn.

Thuật toán này có nhược điểm là thời gian chờ đợi trung bình thường khá lâu. Xét tập hợp các tiến trình sau đây đến tại thời điểm 0, với thời gian sử dụng CPU tính bằng mili giây:

Tiến trình	Thời gian
P <sub>1</sub>	24
P <sub>2</sub>	3
P <sub>3</sub>	3

Nếu các tiến trình này đến theo trật tự P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub> và được phục vụ theo cơ chế FIFO. Ta có kết quả thể hiện trong biểu đồ Gantt:



Thời gian đợi của tiến trình P<sub>1</sub> là 0, tiến trình P<sub>2</sub> là 24, tiến trình P<sub>3</sub> là 27 và thời gian đợi trung bình của các tiến trình là  $(0 + 24 + 27)/3 = 17$ .

Nếu tiến trình đến theo trật tự P<sub>2</sub>, P<sub>3</sub>, P<sub>1</sub> kết quả thể hiện ở biểu đồ Gantt như sau:



Thời gian chờ trung bình là  $(6+0+3)/3 = 3$ . Mức giảm này là đáng kể. Do đó, thời gian chờ trung bình theo chính sách FCFS nói chung không phải là tối thiểu và có thể thay đổi đáng kể nếu thời gian các tiến trình ràng buộc CPU khác nhau nhiều.

Ngoài ra, hãy xem xét hiệu suất của lập lịch FCFS trong một tình huống động. Giả

sử chúng ta có một tiến trình liên kết CPU và nhiều tiến trình liên kết I/O. Khi các tiến trình chạy trên hệ thống, trường hợp sau có thể xảy ra. Tiến trình ràng buộc CPU sẽ lấy và giữ CPU. Trong thời gian này, tất cả các tiến trình khác sẽ kết thúc I/O của chúng và sẽ chuyển vào hàng đợi sẵn sàng, chờ CPU. Trong khi các tiến trình chờ đợi trong hàng đợi sẵn sàng, các thiết bị I/O không hoạt động. Cuối cùng, tiến trình ràng buộc CPU kết thúc quá trình sử dụng CPU của nó và chuyển đến thiết bị I/O. Tất cả các tiến trình liên kết I/O, có CPU ngắn, thực thi nhanh chóng và quay trở lại hàng đợi I/O. Tại thời điểm này, CPU không hoạt động. Tiến trình ràng buộc CPU sau đó sẽ chuyển trở lại hàng đợi sẵn sàng và được cấp phát CPU. Một lần nữa, tất cả các tiến trình I/O kết thúc chờ đợi trong hàng đợi sẵn sàng cho đến khi tiến trình ràng buộc CPU được thực hiện. Có một tình huống khi tất cả các tiến trình khác chờ đợi một tiến trình lớn để thoát khỏi CPU. Tình huống này dẫn đến việc sử dụng CPU và thiết bị thấp hơn mức có thể nếu các tiến trình ngắn hơn được phép thực hiện trước.

Cũng lưu ý rằng thuật toán lập lịch FCFS không mang tính ưu tiên. Khi CPU đã được cấp phát cho một tiến trình, tiến trình đó sẽ giữ CPU cho đến khi nó giải phóng CPU, bằng cách kết thúc hoặc bằng cách yêu cầu I/O. Do đó, thuật toán FCFS đặc biệt rắc rối đối với các hệ thống tương tác, trong đó điều quan trọng là mỗi tiến trình phải nhận được một phần của CPU trong các khoảng thời gian đều đặn. Sẽ là tai hại nếu cho phép một tiến trình giữ CPU trong một thời gian dài.

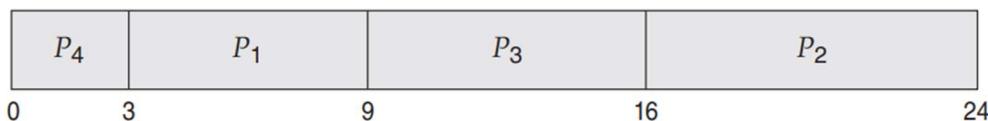
#### 4.3.2. Shortest-Jobs-First (SJF)

Một cách tiếp cận khác để lập lịch CPU là thuật toán lập lịch short-job-first (SJF). Thuật toán này kết hợp mỗi tiến trình với khoảng thời gian sử dụng CPU. Khi CPU khả dụng, nó được gán cho tiến trình có khoảng thời gian nhỏ nhất. Nếu có hai tiến trình cùng thời gian, lập lịch FCFS được sử dụng để điều phối.

Ví dụ về lập lịch SJF, hãy xem xét tập hợp các tiến trình sau, với thời lượng CPU được tính bằng mili giây:

Tiến trình	Thời gian
P <sub>1</sub>	6
P <sub>2</sub>	8
P <sub>3</sub>	7
P <sub>4</sub>	3

Sử dụng lập lịch SJF, ta lập lịch các tiến trình này theo biểu đồ Gantt sau:



Thời gian chờ là 3 mili giây đối với tiến trình  $P_1$ , 16 mili giây đối với tiến trình  $P_2$ , 9 mili giây đối với tiến trình  $P_3$  và 0 mili giây đối với tiến trình  $P_4$ . Như vậy, thời gian chờ trung bình là  $(3 + 16 + 9 + 0) / 4 = 7$  mili giây. Để so sánh, nếu ta sử dụng lập lịch FCFS, thời gian chờ trung bình sẽ là 10,25 mili giây. Thuật toán lập lịch SJF là tối ưu thời gian chờ trung bình.

Mặc dù thuật toán SJF là tối ưu, nhưng nó không thể cài đặt lập lịch mức thấp, vì không có cách nào để biết độ dài của lần dùng CPU tiếp theo. Một giải pháp cho vấn đề này là lập lịch SJF gần đúng. Chúng ta không biết độ dài của lần dùng CPU tiếp theo, nhưng có thể dự đoán giá trị của nó. Hy vọng rằng đợt dùng CPU tiếp theo sẽ có độ dài tương tự như những đợt trước. Bằng cách ước lượng độ dài của lần dùng CPU tiếp theo, ta có thể chọn tiến trình với thời gian dùng CPU dự đoán ngắn nhất.

Lần dùng CPU tiếp theo thường được dự đoán là trung bình theo cấp số nhân của độ dài đo được của các lần dùng CPU trước đó. Ta có thể xác định giá trị trung bình theo cấp số nhân với công thức sau. Gọi  $t_n$  là độ dài của lần dùng CPU thứ  $n$  và đặt  $\tau_{n+1}$  là giá trị dự đoán cho lần dùng CPU tiếp theo. Khi đó, với  $\alpha$ ,  $0 \leq \alpha \leq 1$ , định nghĩa:

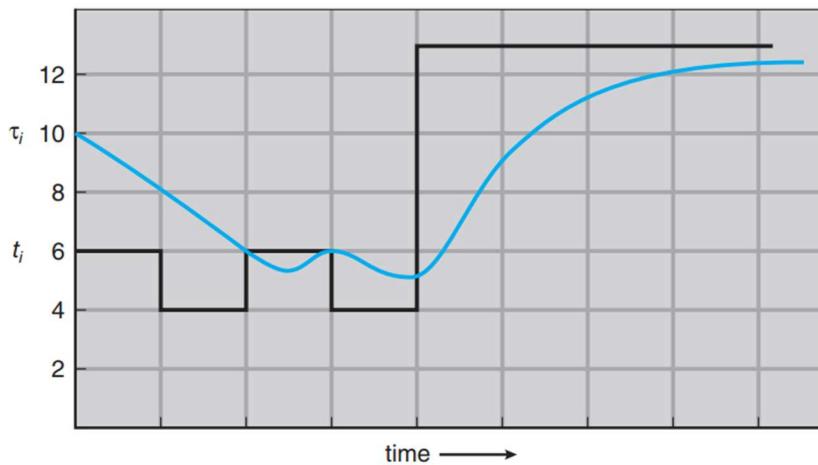
$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

Giá trị của  $t_n$  chưa thông tin gần đây nhất, trong khi  $\tau_n$  giá trị dự đoán ở lần trước. Tham số  $\alpha$  điều khiển trọng số tương đối của giá trị gần đây và dự đoán. Thông thường hơn,  $\alpha = 1/2$ , vì vậy lịch sử gần đây và lịch sử quá khứ có trọng số như nhau.  $t_0$  ban đầu có thể được định nghĩa là một hằng số hoặc là một giá trị trung bình của toàn hệ thống.

Để hiểu công thức này, ta có thể mở rộng công thức cho  $\tau_{n+1}$  bằng cách thay thế cho  $\tau_n$  để tìm:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0.$$

Thuật toán SJF có thể là ưu tiên hoặc không ưu tiên. Sự lựa chọn phát sinh khi một tiến trình mới đến hàng đợi sẵn sàng trong khi một tiến trình trước đó vẫn đang thực thi. Thời gian sử dụng CPU tiếp theo của tiến trình mới đến có thể ngắn hơn những tiến trình còn lại trong hàng đợi sẵn sàng. Thuật toán SJF ưu tiên có thể chặn tiến trình đang thực thi, trong khi thuật toán SJF không ưu tiên sẽ cho phép tiến trình hiện đang chạy kết thúc khoảng thời gian dùng CPU. Lập lịch SJF ưu tiên đôi khi được gọi là lập lịch thời gian còn lại ngắn nhất (Shortest Remaining Time - SRT).

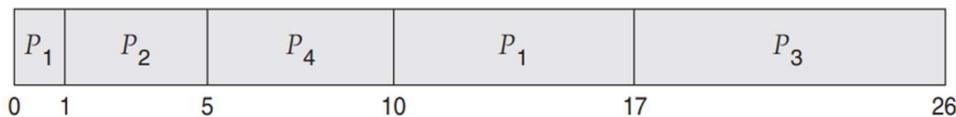


CPU burst ( $t_i$ )	6	4	6	4	13	13	13	...	
"guess" ( $\tau_i$ )	10	8	6	6	5	9	11	12	...

Ví dụ, hãy xem xét bốn tiến trình sau, với thời lượng sử dụng CPU được tính bằng mili giây:

Tiến trình	Thời gian đến	Thời gian dùng CPU
P <sub>1</sub>	0	8
P <sub>2</sub>	1	4
P <sub>3</sub>	2	9
P <sub>4</sub>	3	5

Kết quả lập lịch SJF ưu tiên thể hiện ở biểu đồ Gantt như sau:



Tiến trình P<sub>1</sub> được bắt đầu tại thời điểm 0, vì nó là tiến trình duy nhất trong hàng đợi. Tiến trình P<sub>2</sub> đến tại thời điểm 1. Thời gian còn lại cho tiến trình P<sub>1</sub> (7 mili giây) lớn hơn thời gian yêu cầu của tiến trình P<sub>2</sub> (4 mili giây), vì vậy tiến trình P<sub>2</sub> được ưu tiên trước và tiến trình P<sub>1</sub> chờ. Thời gian chờ trung bình cho ví dụ này là  $[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)] / 4 = 26/4 = 6,5$  mili giây. Lập lịch không ưu tiên SJF có thời gian chờ trung bình là 7,75 mili giây.

#### 4.3.3. Thuật toán Round-Robin (RR)

Thuật toán lập lịch round-robin (RR) tương tự như lập lịch FCFS, nhưng bổ sung cơ chế không độc quyền để cho phép hệ thống chuyển đổi giữa các tiến trình. Định nghĩa một đơn vị thời gian nhỏ, được gọi là lượng tử (quantum). Một lượng tử thường

có độ dài từ 10 đến 100 mili giây. Hàng đợi sẵn sàng được coi là hàng đợi vòng tròn. Bộ lập lịch CPU đi xung quanh hàng đợi sẵn sàng, phân bổ CPU cho mỗi tiến trình trong khoảng thời gian tối đa 1 lần lượng tử.

Để cài đặt lập lịch RR, chúng ta lại coi hàng đợi sẵn sàng như một hàng đợi FIFO. Các tiến trình mới được thêm vào đuôi của hàng đợi sẵn sàng.

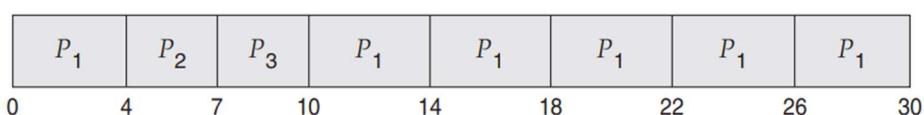
Bộ lập lịch CPU chọn tiến trình đầu tiên từ hàng đợi sẵn sàng, đặt bộ đếm thời gian để ngắt sau lượng tử thời gian và điều phối tiến trình.

Một trong hai điều sau đó sẽ xảy ra. Tiến trình này có thể có một thời gian CPU ít hơn 1 lần lượng tử. Trong trường hợp này, chính tiến trình sẽ tự giải phóng CPU. Bộ lập lịch sau đó sẽ thực hiện tiến trình tiếp theo trong hàng đợi đã sẵn sàng. Nếu thời gian dùng CPU của tiến trình hiện đang chạy lâu hơn 1 lần lượng tử, bộ đếm thời gian sẽ tắt và phát sinh ra ngắt cho hệ điều hành. Một chuyển đổi ngữ cảnh sẽ được thực thi và tiến trình này sẽ được đặt ở phần cuối của hàng đợi sẵn sàng. Bộ lập lịch CPU sau đó sẽ chọn tiến trình tiếp theo trong hàng đợi sẵn sàng.

Thời gian chờ đợi trung bình theo thuật toán RR thường dài. Hãy xét tập hợp các tiến trình sau đây đến tại thời điểm 0, với thời lượng sử dụng CPU được tính bằng mili giây:

Tiến trình	Thời gian dùng CPU
P <sub>1</sub>	24
P <sub>2</sub>	3
P <sub>3</sub>	3

Nếu ta sử dụng lượng tử thời gian là 4 mili giây, thì tiến trình P<sub>1</sub> nhận được 4 mili giây đầu tiên. Vì nó yêu cầu thêm 20 mili giây nữa, nên nó sẽ được ưu tiên sau lượng tử thời gian đầu tiên và CPU được cấp cho tiến trình tiếp theo trong hàng đợi, tiến trình P<sub>2</sub>. Tiến trình P<sub>2</sub> không cần 4 mili giây, vì vậy nó sẽ thoát trước khi lượng tử thời gian của nó hết hạn. Sau đó, CPU được đưa cho tiến trình tiếp theo, tiến trình P<sub>3</sub>. Khi mỗi tiến trình đã nhận được 1 lượng tử thời gian, CPU sẽ được quay trở lại để xử lý P<sub>1</sub> cho một lượng tử thời gian bổ sung. Lịch trình RR kết quả biểu đồ Gantt như sau:

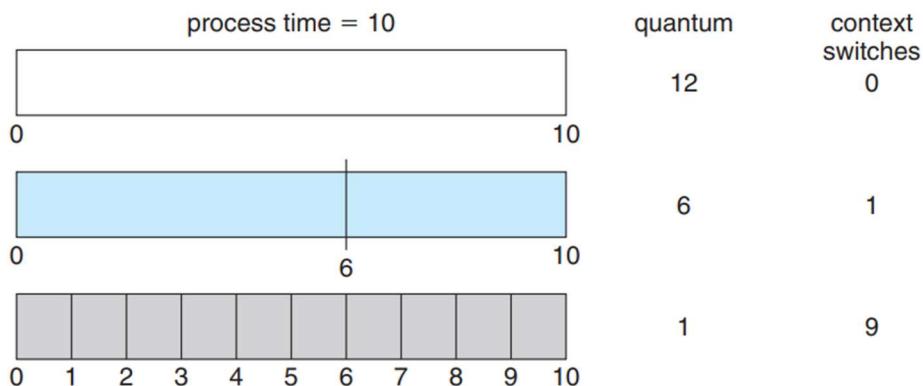


Hãy tính thời gian chờ trung bình thuật toán lập lịch này. P<sub>1</sub> đợi 6 mili giây (10 - 4), P<sub>2</sub> đợi 4 mili giây và P<sub>3</sub> đợi 7 mili giây. Như vậy, thời gian chờ trung bình là 17/3 =

5,66 mili giây.

Trong thuật toán lập lịch RR, không có tiến trình nào được cấp phát CPU nhiều hơn 1 lần lượng tử liên tiếp (trừ khi đó là tiến trình duy nhất có thể chạy được). Nếu thời gian sử dụng CPU của một tiến trình vượt quá 1 lần lượng tử, thì tiến trình đó sẽ được ưu tiên trước và đưa trở lại hàng đợi sẵn sàng.

Hiệu suất của thuật toán RR phụ thuộc vào kích thước của lượng tử thời gian. Nếu lượng tử thời gian là cực lớn, thì thuật toán RR giống như FCFS. Ngược lại, nếu lượng tử thời gian cực kỳ nhỏ (ví dụ, 1 mili giây), thì phương pháp RR có thể dẫn đến một số lượng lớn các chuyển ngữ cảnh. Ví dụ, giả sử rằng chúng ta chỉ có một tiến trình gồm 10 đơn vị thời gian. Nếu lượng tử là 12 đơn vị thời gian, tiến trình kết thúc với ít hơn 1 lượng tử thời gian, không có chuyển ngữ cảnh. Tuy nhiên, nếu lượng tử là 6 đơn vị thời gian, tiến trình yêu cầu 2 lần lượng tử, dẫn đến chuyển đổi ngữ cảnh. Nếu lượng tử là 1 đơn vị thời gian, thì xảy ra 9 chuyển ngữ cảnh, làm chậm tiến trình thực hiện tương ứng (Hình 4.3).

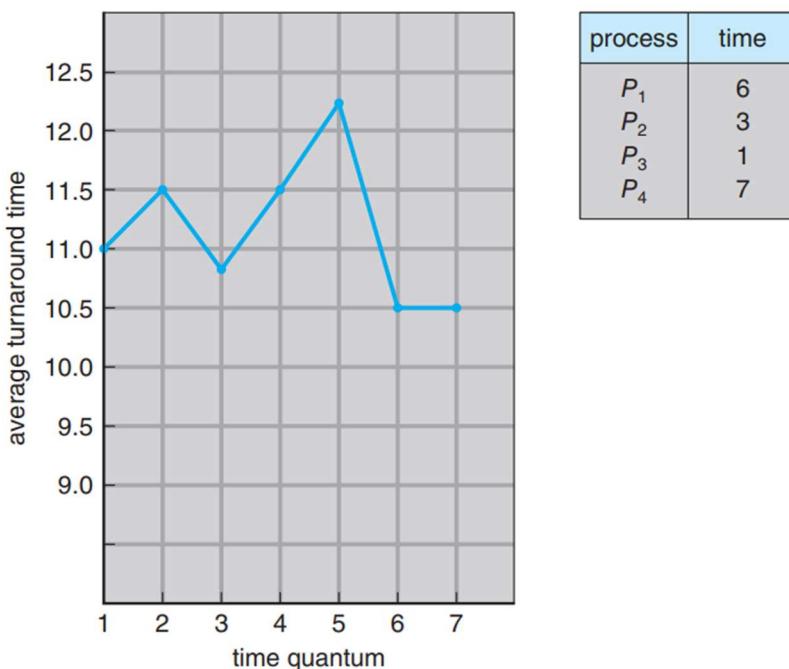


**Hình 4.3.** Lượng tử thời gian nhỏ tăng chuyển ngữ cảnh.

Do đó, chúng ta muốn lượng tử thời gian lớn hơn so với thời gian chuyển ngữ cảnh. Nếu thời gian chuyển ngữ cảnh xấp xỉ 10% lượng tử thời gian, thì khoảng 10% thời gian CPU sẽ được sử dụng trong chuyển ngữ cảnh. Trong thực tế, hầu hết các hệ thống hiện đại có lượng tử thời gian nằm trong khoảng từ 10 đến 100 mili giây. Thời gian cần thiết cho một chuyển ngữ cảnh thường ít hơn 10 micro giây; do đó, thời gian chuyển đổi ngữ cảnh là một phần nhỏ của lượng tử thời gian.

Thời gian quay vòng cũng phụ thuộc vào kích thước của lượng tử thời gian. Như chúng ta có thể thấy từ Hình 4.4, thời gian quay vòng trung bình của một tập các tiến trình không cải thiện khi kích thước lượng tử thời gian tăng lên. Nói chung, thời gian quay vòng trung bình có thể được cải thiện nếu hầu hết các tiến trình kết thúc đợt sử dụng CPU tiếp theo của chúng trong một lượng tử thời gian duy nhất. Ví dụ: với ba tiến

trình, mỗi tiến trình gồm 10 đơn vị thời gian và lượng tử là 1 đơn vị thời gian, thì thời gian quay vòng trung bình là 29. Tuy nhiên, nếu lượng tử thời gian là 10, thì thời gian quay vòng trung bình giảm xuống còn 20. Nếu thêm thời gian chuyển đổi ngũ cảnh. trong đó, thời gian quay vòng trung bình thậm chí còn tăng nhiều hơn đối với lượng tử thời gian nhỏ hơn, vì cần nhiều chuyển ngũ cảnh hơn.



**Hình 4.4.** Thời gian quay vòng sẽ thay đổi khi lượng tử thay đổi.

#### 4.3.4. Lập lịch với độ ưu tiên

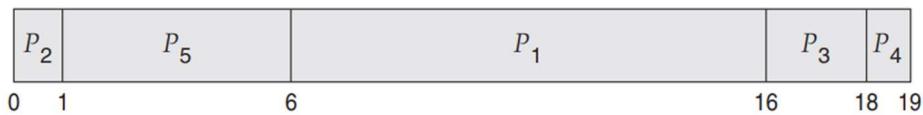
Thuật toán SJF là một trường hợp đặc biệt của thuật toán lập lịch ưu tiên. Mỗi tiến trình liên kết một độ ưu tiên và CPU được phân bổ cho tiến trình có độ ưu tiên cao nhất. Các tiến trình ưu tiên bằng nhau được lập lịch theo thứ tự FCFS. Thuật toán SJF là trường hợp đơn giản thuật toán lập lịch với độ ưu tiên trong đó độ ưu tiên ( $p$ ) là nghịch đảo của thời gian CPU tiếp theo (dự đoán). Thời gian CPU càng lớn thì mức độ ưu tiên càng thấp và ngược lại.

Ví dụ, hãy xem xét tập hợp các tiến trình sau, giả sử đến vào thời điểm 0 theo thứ tự  $P_1, P_2, \dots, P_5$ , với thời gian CPU được tính bằng mili giây:

Tiến trình	Thời gian	Ưu tiên
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4

P <sub>4</sub>	1	5
P <sub>5</sub>	5	2

Sử dụng lập lịch với độ ưu tiên, ta lập lịch các tiến trình này theo biểu đồ Gantt sau:



Thời gian chờ trung bình là 8.2 mili giây.

Lập lịch với độ ưu tiên (Priority) có thể là không độc quyền (Preemptive) hoặc độc quyền (Nonpreemptive). Khi một tiến trình đến hàng đợi sẵn sàng, mức độ ưu tiên của nó được so sánh với mức độ ưu tiên của tiến trình hiện đang chạy. Thuật toán lập lịch ưu tiên, không độc quyền sẽ nhường CPU nếu độ ưu tiên của tiến trình mới đến cao hơn độ ưu tiên của tiến trình hiện đang chạy. Một thuật toán lập lịch ưu tiên, độc quyền chỉ đơn giản là đặt tiến trình mới ở đầu hàng đợi sẵn sàng.

Một vấn đề lớn với các thuật toán lập lịch với độ ưu tiên là có tiến trình bị chặn vô hạn hoặc chết đói. Tiến trình đã sẵn sàng để chạy nhưng đang chờ CPU được coi là bị chặn. Một thuật toán lập lịch ưu tiên có thể khiến một số tiến trình có ưu tiên thấp phải chờ vô hạn.

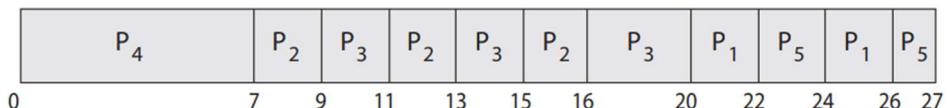
Một giải pháp cho vấn đề chặn vô hạn của các tiến trình ưu tiên thấp là lão hóa (aging). Lão hóa là tăng dần mức độ ưu tiên của các tiến trình chờ trong hệ thống trong một thời gian dài. Ví dụ: nếu độ ưu tiên nằm trong khoảng từ 127 (thấp) đến 0 (cao), ta có thể định kỳ (giả sử mỗi giây) tăng mức độ ưu tiên của một tiến trình chờ lên 1. Cuối cùng, ngay cả một tiến trình có mức độ ưu tiên ban đầu là 127 sẽ đạt ưu tiên cao nhất trong hệ thống và sẽ được thực thi. Trên thực tế, sẽ mất hơn 2 phút để tiến trình ưu tiên 127 chuyển thành tiến trình ưu tiên 0.

Một tùy chọn khác là kết hợp lập lịch RR và ưu tiên theo cách hệ thống thực thi tiến trình có mức ưu tiên cao nhất và chạy các tiến trình có cùng mức ưu tiên bằng lập lịch RR. Ví dụ tập hợp các tiến trình sau, với thời gian sử dụng CPU tính bằng mili giây:

Tiến trình	Thời gian	Ưu tiên
P <sub>1</sub>	4	3
P <sub>2</sub>	5	2
P <sub>3</sub>	8	2

P <sub>4</sub>	7	1
P <sub>5</sub>	3	3

Sử dụng lập lịch ưu tiên kết hợp với RR cho các tiến trình có mức độ ưu tiên ngang nhau, ta sẽ lập lịch các tiến trình này theo biểu đồ Gantt sau đây sử dụng lượng tử thời gian là 2 mili giây:



Trong ví dụ này, tiến trình P<sub>4</sub> có mức độ ưu tiên cao nhất, vì vậy nó sẽ chạy đến khi hoàn thành. Các tiến trình P<sub>2</sub> và P<sub>3</sub> có mức độ ưu tiên cao nhất tiếp theo và chúng sẽ thực hiện theo kiểu vòng tròn. Lưu ý rằng khi tiến trình P<sub>2</sub> kết thúc ở thời điểm 16, tiến trình P<sub>3</sub> là tiến trình có mức độ ưu tiên cao nhất, vì vậy nó sẽ chạy cho đến khi hoàn thành việc thực thi. Nay giờ, chỉ còn lại các tiến trình P<sub>1</sub> và P<sub>5</sub> và vì chúng có mức độ ưu tiên ngang nhau, chúng sẽ thực thi theo thứ tự vòng lại cho đến khi hoàn thành.

#### 4.3.5. Thuật toán Highest-Response-Ratio-Next (HRRN)

Brinch Hansen đã phát triển thuật toán tỷ lệ phản hồi cao nhất tiếp theo (HRRN) để khắc phục một số điểm yếu trong SJF, đặc biệt là sự “thành kiến” quá mức đối với các tiến trình dài và “thiên vị” quá mức đối với các tiến trình ngắn. HRRN là một cơ chế lập lịch độc quyền, trong đó mức độ ưu tiên của mỗi tiến trình là một hàm với tham số thời gian phục vụ và thời gian chờ đợi của nó. Khi một tiến trình có CPU, nó chạy đến lúc hoàn thành. HRRN tính toán mức độ ưu tiên động theo công thức:

$$\text{priority} = \frac{\text{waiting time} + \text{service time}}{\text{service time}}$$

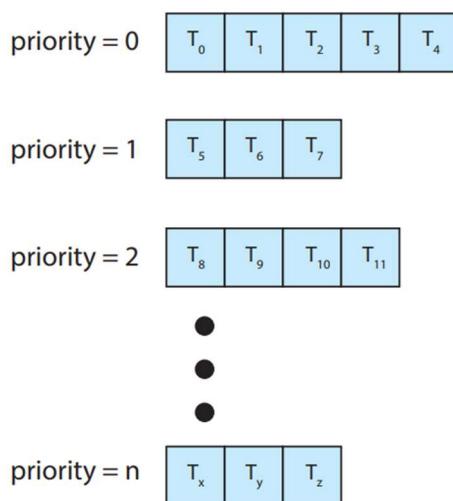
Vì thời gian phục vụ xuất hiện ở mẫu số, các tiến trình ngắn hơn nhận được ưu tiên. Tuy nhiên, vì thời gian chờ xuất hiện trong tử số, các tiến trình lâu hơn đã chờ đợi cũng sẽ được xử lý thuận lợi. Kỹ thuật này tương tự như quá trình lão hóa và ngăn chặn bộ lập lịch trì hoãn vô thời hạn các tiến trình.

#### 4.3.6. Lập lịch hàng đợi đa cấp

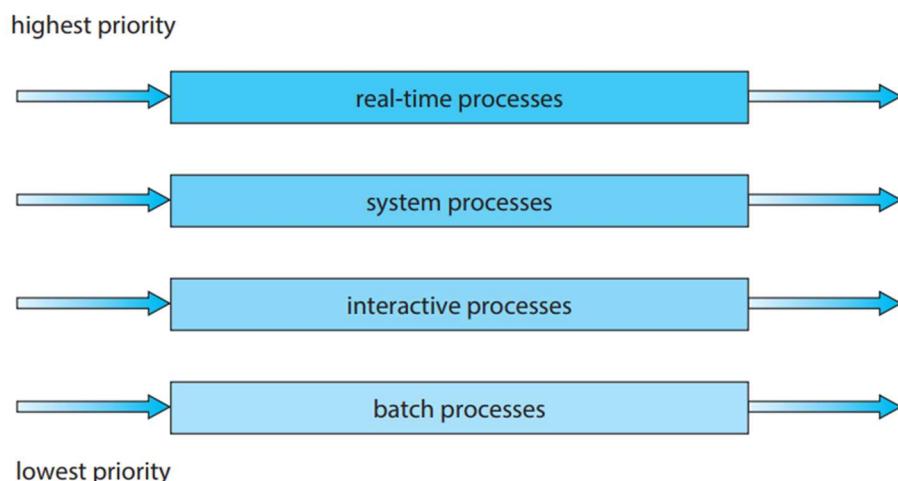
Với lập lịch ưu tiên và lập lịch RR, tất cả các tiến trình có thể được đặt trong một hàng đợi duy nhất và sau đó bộ lập lịch sẽ chọn tiến trình có mức độ ưu tiên cao nhất để chạy. Tùy thuộc vào cách quản lý các hàng đợi, tìm kiếm với độ phức tạp O(n) xác định tiến trình có mức độ ưu tiên cao nhất. Trong thực tế, việc có các hàng đợi riêng biệt cho từng mức độ ưu tiên riêng biệt thường dễ dàng hơn và việc lập lịch với độ ưu tiên chỉ đơn giản là lập lịch tiến trình trong hàng đợi có mức độ ưu tiên cao nhất. Điều này được

minh họa trong Hình 4.5. Cách tiếp cận này - được gọi là hàng đợi đa cấp - cũng hoạt động tốt khi lập lịch ưu tiên được kết hợp với RR: nếu có nhiều tiến trình trong hàng đợi có mức ưu tiên cao nhất, chúng sẽ được thực thi theo RR. Ở dạng tổng quát nhất của phương pháp này, một độ ưu tiên được chỉ định tĩnh cho mỗi tiến trình và một tiến trình vẫn nằm trong cùng một hàng đợi trong suốt thời gian chạy của nó.

Một thuật toán lập lịch hàng đợi đa cấp cũng có thể được sử dụng để chia các tiến trình thành một số hàng đợi riêng biệt dựa trên loại tiến trình (Hình 4.6). Ví dụ, phân chia giữa các tiến trình tương tác và các tiến trình hàng loạt. Hai loại tiến trình này có các yêu cầu về thời gian phản hồi khác nhau và do đó có thể có các nhu cầu lập lịch khác nhau. Các hàng đợi riêng biệt có thể được sử dụng cho các tiến trình nền tương tác và hàng loạt và mỗi hàng đợi có thể có thuật toán lập lịch riêng. Ví dụ: hàng đợi tiến trình tương tác có thể được lập lịch bởi thuật toán RR, trong khi hàng đợi tiến trình hàng loạt lập lịch bằng thuật toán FCFS.



**Hình 4.5.** Hàng đợi riêng cho mỗi mức ưu tiên.



**Hình 4.6.** Lập lịch hàng đợi đa cấp.

Ngoài ra, phải có lập lịch giữa các hàng đợi, thường được cài đặt dưới dạng lập lịch ưu tiên với độ ưu tiên cố định. Ví dụ: hàng đợi thời gian thực có thể có mức độ ưu tiên tuyệt đối cao hơn so với hàng đợi tương tác.

Hãy xem ví dụ về thuật toán lập lịch hàng đợi đa cấp với bốn hàng đợi, được liệt kê bên dưới theo thứ tự ưu tiên:

1. Tiến trình thời gian thực
2. Tiến trình hệ thống
3. Tiến trình tương tác
4. Tiến trình hàng loạt

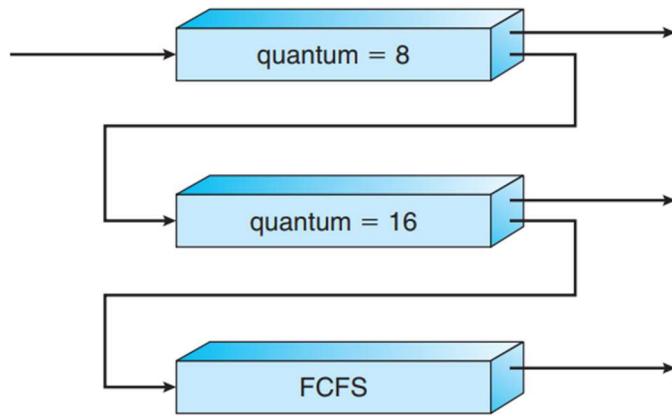
Một vấn đề khác là phân chia thời gian giữa các hàng đợi. Ở đây, mỗi hàng đợi nhận được một phần thời gian nhất định của CPU, sau đó nó có thể lập lịch giữa các tiến trình khác nhau của nó. Chẳng hạn hàng đợi tương tác có thể được cấp 80 phần trăm thời gian CPU cho việc lập lịch RR giữa các tiến trình của nó, trong khi hàng đợi hàng loạt (batch) nhận 20 phần trăm CPU cho các tiến trình của nó để lập lịch FCFS.

#### 4.3.7. Lập lịch hàng đợi phản hồi đa cấp

Thông thường, khi thuật toán lập lịch hàng đợi đa cấp được sử dụng, các tiến trình được gán vĩnh viễn vào một hàng đợi khi chúng vào hệ thống. Ví dụ: nếu có các hàng đợi riêng biệt cho các tiến trình tương tác và hàng loạt, thì các tiến trình không di chuyển từ hàng đợi này sang hàng đợi khác, vì các tiến trình không thay đổi bản chất. Thiết lập này có ưu điểm là chi phí lập lịch thấp, nhưng nó không linh hoạt.

Ngược lại, thuật toán lập lịch hàng đợi phản hồi đa cấp cho phép một tiến trình di chuyển giữa các hàng đợi. Ý tưởng là tách các tiến trình theo đặc điểm thời gian sử dụng CPU của chúng. Nếu một tiến trình sử dụng quá nhiều thời gian CPU, nó sẽ được chuyển đến hàng đợi có mức độ ưu tiên thấp hơn. Cơ chế này dùng cho các tiến trình tương tác và tiến trình I/O - thường được đặc trưng thời gian CPU ngắn - trong các hàng đợi có mức độ ưu tiên thấp hơn. Ngoài ra, tiến trình chờ đợi quá lâu trong hàng đợi có mức độ ưu tiên thấp hơn có thể được chuyển sang hàng hàng có mức độ ưu tiên cao hơn.

Ví dụ, xét lập lịch hàng đợi phản hồi đa cấp với ba hàng đợi, được đánh số từ 0 đến 2 (Hình 4.7). Đầu tiên bộ lập lịch thực thi tất cả các tiến trình trong hàng đợi 0. Chỉ khi hàng đợi 0 trống thì nó mới thực thi các tiến trình trong hàng đợi 1. Tương tự, các tiến trình trong hàng đợi 2 sẽ chỉ được thực thi nếu hàng 0 và 1 trống. Một tiến trình đến hàng đợi 1 sẽ ưu tiên trước một tiến trình ở hàng đợi 2.

**Hình 4.7.** Hàng đợi phản hồi đa cấp.

Một tiến trình mới vào được đưa vào hàng đợi 0. Một tiến trình ở hàng 0 được cho một lượng tử thời gian là 8 mili giây. Nếu nó không kết thúc trong thời gian này, nó sẽ được chuyển đến phần cuối của hàng đợi 1. Nếu hàng 0 trống, tiến trình ở đầu hàng đợi 1 được cung cấp một lượng tử là 16 mili giây. Nếu nó không hoàn thành, nó sẽ được đưa vào cuối hàng đợi 2. Các tiến trình trong hàng đợi 2 được chạy trên cơ sở FCFS nhưng chỉ được chạy khi hàng đợi 0 và 1 trống. Để ngăn chặn tình trạng chết đói, một tiến trình chờ đợi quá lâu trong hàng đợi có mức độ ưu tiên thấp hơn có thể dần dần được chuyển sang hàng đợi có mức độ ưu tiên cao hơn.

Thuật toán lập lịch này dành ưu tiên cao nhất cho bất kỳ tiến trình nào có thời gian CPU từ 8 mili giây trở xuống. Tiến trình như vậy sẽ nhanh chóng được CPU, kết thúc đợt thời gian CPU và bắt đầu với đợt I/O tiếp theo của nó. Các tiến trình cần nhiều hơn 8 nhưng dưới 24 mili giây cũng được phục vụ nhanh chóng, mặc dù có mức độ ưu tiên thấp hơn so với các tiến trình ngắn hơn. Các tiến trình dài tự động chìm xuống hàng đợi 2 và được phục vụ theo thứ tự FCFS với bất kỳ chu kỳ thời gian CPU nào còn sót lại từ hàng đợi 0 và 1.

Nói chung, bộ lập lịch hàng đợi phản hồi đa cấp được xác định bởi các tham số sau:

- Số lượng hàng đợi
- Thuật toán lập lịch cho mỗi hàng đợi
- Phương pháp được sử dụng để xác định thời điểm nâng cấp tiến trình lên hàng đợi ưu tiên cao hơn
- Phương pháp được sử dụng để xác định thời điểm hạ cấp một tiến trình xuống hàng đợi có mức độ ưu tiên thấp hơn

- Phương pháp được sử dụng để xác định tiến trình sẽ vào hàng đợi nào khi tiến trình đó cần dịch vụ.

Lập lịch hàng đợi phản hồi đa cấp là thuật toán lập lịch CPU chung nhất. Nó có thể được cấu hình để phù hợp với một hệ thống cụ thể đang được thiết kế. Thật không may, nó cũng là thuật toán phức tạp nhất, vì việc xác định bộ lập lịch tốt nhất đòi hỏi một số phương tiện để chọn giá trị cho tất cả các tham số.

#### 4.4. Lập lịch đa bộ xử lý

Các nội dung trước tập trung vào các vấn đề lập lịch trình cho CPU trong hệ thống có một lõi xử lý duy nhất. Nếu có nhiều CPU, khả năng chia sẻ tải, trong đó nhiều luồng có thể chạy song song, trở nên khả thi, tuy nhiên, các vấn đề lập lịch trở nên phức tạp hơn. Nhiều khả năng đã được thử; và như chúng ta đã thấy với lập lịch CPU đơn nhân, không có giải pháp nào tốt nhất.

Thuật ngữ đa bộ xử lý dùng để chỉ các hệ thống cung cấp nhiều bộ xử lý vật lý, trong đó mỗi bộ xử lý chứa một CPU lõi đơn. Tuy nhiên, định nghĩa về đa xử lý đã phát triển đáng kể và trên các hệ thống máy tính hiện đại, đa xử lý hiện áp dụng cho các kiến trúc hệ thống sau:

- CPU đa lõi
- Lõi đa luồng
- Hệ thống NUMA
- Đa xử lý không đồng nhất

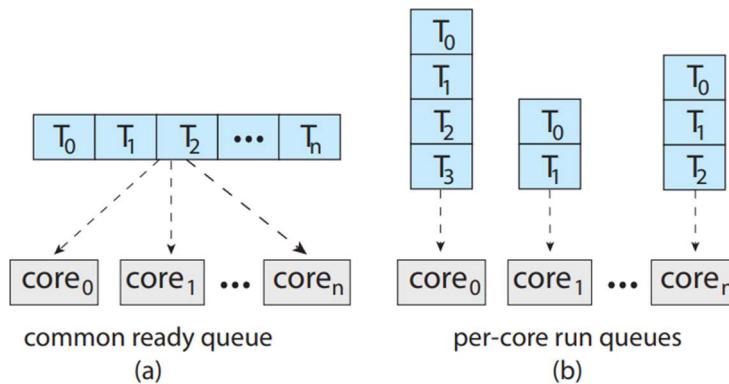
Ở đây, ta thảo luận lập lịch đa xử lý trong ngữ cảnh các kiến trúc khác nhau. Trong ba ví dụ đầu tiên, ta tập trung vào hệ thống trong đó các bộ xử lý giống hệt nhau - đồng nhất - về chức năng của chúng. Sau đó, ta có thể sử dụng bất kỳ CPU nào có sẵn để chạy bất kỳ tiến trình nào trong hàng đợi. Trong ví dụ cuối, ta khám phá một hệ thống mà các bộ xử lý không giống nhau về khả năng của chúng.

##### 4.4.1. Phương pháp lập lịch đa xử lý

Một cách tiếp cận để lập lịch CPU trong hệ thống đa xử lý có tất cả các quyết định lập lịch, xử lý I/O và các hoạt động hệ thống khác được xử lý bởi một bộ xử lý duy nhất - máy chủ chính. Các bộ xử lý khác chỉ thực thi mã người dùng. Đa xử lý bất đối xứng này rất đơn giản vì chỉ có một lõi truy cập vào cấu trúc dữ liệu hệ thống, giảm nhu cầu chia sẻ dữ liệu. Hạn chế lớn của cách tiếp cận này là máy chủ chính sẽ trở thành một nút cỗ chai tiềm ẩn nơi hiệu suất tổng thể của hệ thống có thể bị giảm.

Cách tiếp cận tiêu chuẩn để hỗ trợ đa bộ xử lý là đa xử lý đối xứng (Symmetric Multiprocessing - SMP), trong đó mỗi bộ xử lý tự lập lịch. Việc lập lịch tiến hành bằng cách yêu cầu bộ lập lịch cho mỗi bộ xử lý kiểm tra hàng đợi sẵn sàng và chọn một luồng để chạy. Lưu ý rằng điều này cung cấp hai chiến lược khả thi để tổ chức các luồng đủ điều kiện được lên lịch:

1. Tất cả các luồng có thể nằm trong một hàng đợi sẵn sàng chung.
2. Mỗi bộ xử lý có thể có hàng đợi luồng riêng của nó.



**Hình 4.8. Cách tổ chức hàng đợi sẵn sàng**

Hai chiến lược này được đổi chiều trong Hình 4.8. Nếu ta chọn chiến lược đầu tiên, có thể có tình huống cạnh tranh trên hàng đợi sẵn sàng và do đó phải đảm bảo rằng hai bộ xử lý riêng biệt không chọn lập lịch cho cùng một luồng và các luồng không bị mất khỏi hàng đợi. Chiến lược thứ hai cho phép mỗi bộ xử lý lập lịch các luồng từ hàng đợi riêng của nó và do đó không bị các vấn đề về hiệu suất liên quan đến hàng đợi chung. Do đó, đây là cách tiếp cận phổ biến nhất trên các hệ thống hỗ trợ SMP. Ngoài ra trên thực tế, việc có hàng đợi riêng, chạy trên mỗi bộ xử lý có thể dẫn đến việc sử dụng bộ nhớ đậm hiệu quả hơn. Vấn đề với hàng đợi chạy trên mỗi bộ xử lý là khói lượng công việc có kích thước khác nhau. Tuy nhiên, như chúng ta sẽ thấy, các thuật toán cân bằng có thể được sử dụng để cân bằng khói lượng công việc giữa tất cả các bộ xử lý.

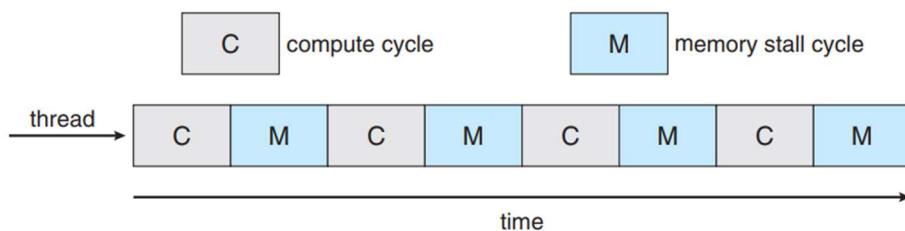
Hầu như tất cả các hệ điều hành hiện đại đều hỗ trợ SMP, bao gồm Windows, Linux và macOS cũng như các hệ thống di động bao gồm Android và iOS.

#### 4.4.2. Bộ xử lý đa nhân

Các hệ thống SMP cho phép một số tiến trình chạy song song bằng cách cung cấp nhiều bộ xử lý vật lý. Tuy nhiên, hầu hết phần cứng máy tính hiện nay đều đặt nhiều lõi trên cùng một chip vật lý, dẫn đến một bộ xử lý đa lõi. Mỗi lõi duy trì trạng thái kiến trúc của nó và do đó, đối với hệ điều hành dường như là một CPU logic riêng biệt. Các hệ thống SMP sử dụng bộ xử lý đa lõi nhanh hơn và tiêu thụ ít năng lượng hơn các hệ

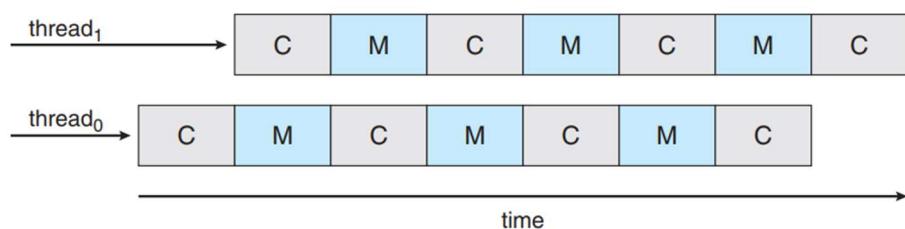
thông trong đó mỗi CPU có một chip vật lý riêng.

Bộ xử lý nhiều lõi có thể làm phức tạp các vấn đề lập lịch. Các nhà nghiên cứu đã phát hiện ra rằng khi một bộ xử lý truy cập vào bộ nhớ, nó sẽ dành một khoảng thời gian đáng kể để chờ dữ liệu có sẵn. Tình trạng này, được gọi là đinh trệ bộ nhớ, xảy ra chủ yếu do các bộ vi xử lý hiện đại hoạt động với tốc độ nhanh hơn nhiều so với bộ nhớ. Tuy nhiên, đinh trệ bộ nhớ cũng có thể xảy ra do lỗi bộ nhớ đệm (truy cập dữ liệu không có trong bộ nhớ đệm). Hình 4.9 minh họa đinh trệ bộ nhớ. Trong trường hợp này, bộ xử lý có thể dành tới 50% thời gian để chờ dữ liệu có sẵn từ bộ nhớ.



**Hình 4.9. Bộ nhớ bị đinh trệ.**

Để khắc phục tình trạng này, nhiều thiết kế phần cứng gần đây đã cài đặt các lõi xử lý đa luồng trong đó hai (hoặc nhiều) luồng phần cứng được gán cho mỗi lõi. Bằng cách đó, nếu một luồng phần cứng dừng lại trong khi chờ bộ nhớ, lõi có thể chuyển sang một luồng khác. Hình 4.10 minh họa một lõi xử lý luồng kép trên đó xen kẽ việc thực thi luồng 0 và luồng 1.



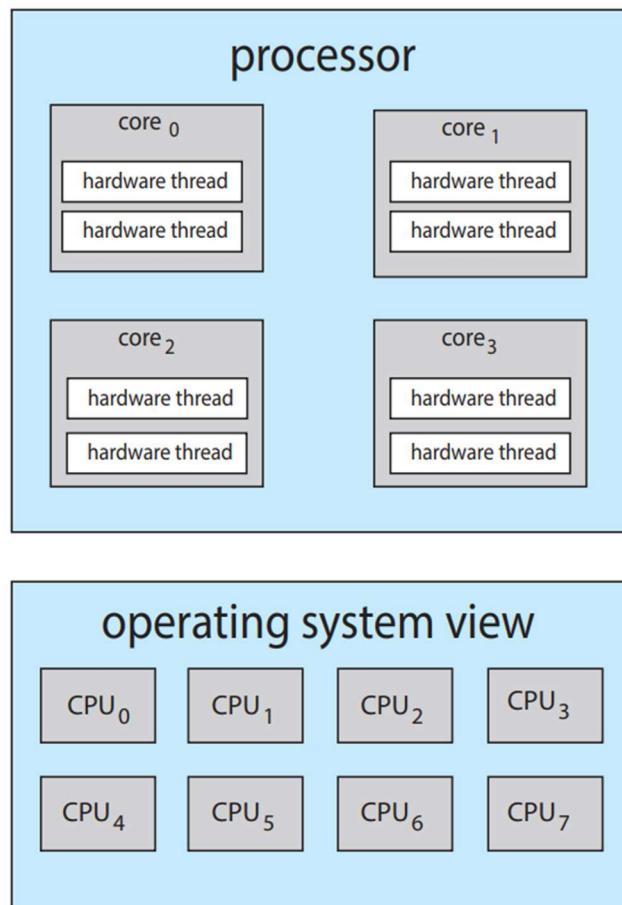
**Hình 4.10. Hệ thống đa nhân, đa luồng.**

Từ góc độ hệ điều hành, mỗi luồng phần cứng duy trì trạng thái kiến trúc của nó, chẳng hạn như con trỏ lệnh và bộ thanh ghi, và do đó xuất hiện như một CPU logic sẵn có để chạy một luồng phần mềm. Kỹ thuật này là chip đa luồng (Chip MultiThreading - CMT) - được minh họa trong Hình 4.11. Ở đây, bộ xử lý chứa bốn lõi tính toán, với mỗi lõi chứa hai luồng phần cứng. Từ quan điểm của hệ điều hành, có tám CPU logic.

Bộ xử lý Intel sử dụng thuật ngữ siêu phân luồng (hyper-threading), còn được gọi là đa luồng đồng thời (simultaneous multithreading) hoặc SMT, để mô tả việc gán nhiều luồng phần cứng cho một lõi xử lý duy nhất. Các bộ xử lý Intel đương đại - chẳng hạn như i7 - hỗ trợ hai luồng cho mỗi lõi, trong khi bộ xử lý Oracle Sparc M7 hỗ trợ tám

luồng cho mỗi lõi, với tám lõi cho mỗi bộ xử lý, do đó cung cấp cho hệ điều hành 64 CPU logic.

Nói chung, có hai cách để đa luồng một lõi xử lý: đa luồng thô và đa luồng mịn. Với đa luồng thô, một luồng thực thi trên một lõi cho đến khi xảy ra sự kiện có độ trễ dài như đình trệ bộ nhớ. Do sự chậm trễ gây ra bởi sự kiện độ trễ dài, lõi phải chuyển sang một luồng khác để bắt đầu thực thi. Tuy nhiên, chi phí chuyển đổi giữa các luồng là cao, vì các lệnh phải được xóa trước khi luồng khác có thể bắt đầu thực thi trên lõi bộ xử lý. Khi luồng mới này bắt đầu thực thi, các lệnh phải tải vào bộ nhớ. Đa luồng mịn (hoặc xen kẽ) chuyển đổi giữa các luồng ở mức độ chi tiết tốt hơn nhiều — thường là ở ranh giới của một chu kỳ lệnh. Tuy nhiên, thiết kế kiến trúc của các hệ thống đa luồng mịn bao gồm chuyển luồng. Do đó, chi phí chuyển đổi giữa các luồng là nhỏ.

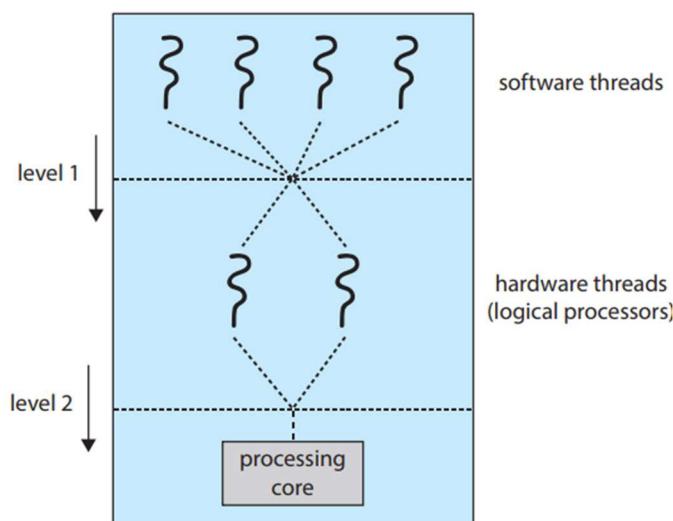


**Hình 4.11.** Chip đa luồng.

Lưu ý quan trọng là tài nguyên của lõi vật lý (chẳng hạn như bộ nhớ đệm cache) phải được chia sẻ giữa các luồng phần cứng của nó và do đó lõi xử lý chỉ có thể thực thi một luồng phần cứng tại một thời điểm. Do đó, một bộ xử lý đa luồng, đa lõi thực sự yêu cầu hai mức lập lịch khác nhau, như trong Hình 4.12, minh họa một lõi xử lý hai luồng.

Ở cấp độ một là các quyết định lập lịch phải được thực hiện bởi hệ điều hành khi nó chọn luồng phần mềm nào sẽ chạy trên mỗi luồng phần cứng (CPU logic).

Mức lập lịch thứ hai chỉ định cách mỗi lõi quyết định luồng phần cứng nào sẽ chạy. Có một số chiến lược để áp dụng trong tình huống này. Một cách tiếp cận là sử dụng một thuật toán RR đơn giản để lên lịch một luồng phần cứng đến lõi xử lý. Đây là cách tiếp cận được UltraSPARC T3 áp dụng. Một cách tiếp cận khác được sử dụng bởi Intel Itanium, một bộ xử lý lõi kép với hai luồng do phần cứng quản lý trên mỗi lõi. Gán cho mỗi luồng phần cứng là một giá trị khẩn cấp dao động từ 0 đến 7, với 0 là mức độ khẩn cấp thấp nhất và 7 là cao nhất. Itanium xác định năm sự kiện khác nhau có thể kích hoạt chuyển đổi luồng. Khi một trong những sự kiện này xảy ra, chuyển mạch luồng sẽ so sánh mức độ khẩn cấp của hai luồng và chọn luồng có giá trị mức độ khẩn cấp cao nhất để thực thi trên lõi bộ xử lý.



**Hình 4.12. Hai mức lập lịch.**

Lưu ý rằng hai mức lập lịch khác nhau được thể hiện trong Hình 4.12 không nhất thiết phải loại trừ lẫn nhau. Trên thực tế, nếu bộ lập lịch của hệ điều hành (mức đầu tiên) đoán trước được việc chia sẻ tài nguyên bộ xử lý, nó có thể đưa ra các quyết định lập lịch hiệu quả hơn. Ví dụ, giả sử rằng một CPU có hai lõi xử lý và mỗi lõi có hai luồng phần cứng. Nếu hai luồng phần mềm đang chạy trên hệ thống này, chúng có thể chạy trên cùng một lõi hoặc trên các lõi riêng biệt. Nếu cả hai đều được lập lịch chạy trên cùng một lõi, chúng phải chia sẻ bộ xử lý và do đó có khả năng tiến hành chậm hơn so với khi chúng được lập lịch trên các lõi riêng biệt. Nếu hệ điều hành đoán trước được mức độ chia sẻ tài nguyên của bộ xử lý, nó có thể lập lịch các luồng phần mềm lên các bộ xử lý logic không chia sẻ tài nguyên.

#### 4.4.3. Cân bằng tải

Trên các hệ thống SMP, điều quan trọng là phải giữ cân bằng khói lượng công việc giữa tất cả các bộ xử lý để tận dụng đầy đủ các lợi ích của việc có nhiều hơn một bộ xử lý. Nếu không, một hoặc nhiều bộ xử lý có thể không hoạt động trong khi các bộ xử lý khác có khói lượng công việc cao, cùng với hàng đợi luồng sẵn sàng đang chờ CPU. Cân bằng tải có cố gắng giữ cho khói lượng công việc được phân bổ đồng đều trên tất cả các bộ xử lý trong hệ thống SMP. Lưu ý quan trọng là cân bằng tải thường chỉ cần thiết trên các hệ thống mà mỗi bộ xử lý có hàng đợi ready riêng của các luồng đủ điều kiện để thực thi. Trên các hệ thống có hàng đợi chạy chung, việc cân bằng tải là không cần thiết, vì một khi bộ xử lý không hoạt động, nó ngay lập tức trích xuất một luồng chạy được từ hàng đợi sẵn sàng chung.

Có hai cách tiếp cận chung để cân bằng tải: di chuyển đẩy và di chuyển kéo. Với di chuyển đẩy, một tác vụ cụ thể định kỳ kiểm tra tải trên mỗi bộ xử lý và — nếu phát hiện thấy sự mất cân bằng — phân bổ đều tải bằng cách đẩy các luồng từ bộ xử lý quá tải sang bộ xử lý nhàn rỗi hoặc ít bận hơn. Di chuyển kéo xảy ra khi một bộ xử lý nhàn rỗi kéo một tác vụ đang chờ từ một bộ xử lý bận. Di chuyển đẩy và kéo không cần phải loại trừ lẫn nhau và trên thực tế, thường được thực hiện song song trên các hệ thống cân bằng tải.

Khái niệm “cân bằng tải” có thể có các ý nghĩa khác nhau. Một cách nhìn của cân bằng tải có thể yêu cầu đơn giản rằng tất cả các hàng đợi có số lượng luồng xấp xỉ như nhau. Ngoài ra, cách nhìn khác sự cân bằng tải có thể yêu cầu sự phân bổ đồng đều các ưu tiên của luồng trên tất cả các hàng đợi.

#### 4.4.4. Liên kết bộ xử lý

Xét tình huống xảy ra với bộ nhớ đệm khi một luồng đang chạy trên một bộ xử lý cụ thể. Dữ liệu được luồng truy cập gần đây nhất sẽ điền vào bộ nhớ đệm cho bộ xử lý. Do đó, các lần truy cập bộ nhớ liên tiếp của luồng thường được đáp ứng trong bộ nhớ đệm (được gọi là “bộ nhớ đệm ấm”). Nếu luồng di chuyển sang một bộ xử lý khác - chẳng hạn như do cân bằng tải. Nội dung của bộ nhớ đệm phải bị vô hiệu đối với bộ xử lý thứ nhất và tạo lại bộ nhớ đệm cho bộ xử lý thứ hai. Do chi phí cao cho việc làm mát hiệu lực và lưu trữ lại bộ nhớ đệm, hầu hết các hệ điều hành có hỗ trợ SMP cố gắng tránh di chuyển một luồng từ bộ xử lý này sang bộ xử lý khác và thay vào đó cố gắng giữ cho một luồng chạy trên cùng một bộ xử lý và tận dụng bộ nhớ đệm ấm. Điều này được gọi là liên kết bộ xử lý - nghĩa là, một tiến trình có một mối quan hệ với bộ xử lý mà nó hiện đang chạy.

Liên kết bộ xử lý có nhiều dạng. Khi một hệ điều hành có chính sách cố gắng giữ một tiến trình chạy trên cùng một bộ xử lý - nhưng không đảm bảo rằng nó sẽ làm như vậy - chúng ta gặp một tình huống được gọi là liên kết mềm. Ở đây, hệ điều hành sẽ cố gắng giữ một tiến trình trên một bộ xử lý duy nhất, nhưng một tiến trình có thể di chuyển giữa các bộ xử lý trong quá trình cân bằng tải. Ngược lại, một số hệ thống cung cấp lệnh gọi hệ thống hỗ trợ liên kết cứng, cho phép một tiến trình chỉ định một tập hợp con bộ xử lý mà nó có thể chạy. Nhiều hệ thống cung cấp cả liên kết mềm và cứng. Ví dụ: Linux cài đặt liên kết mềm, nhưng nó cũng cung cấp lời gọi hệ thống `Sched_setaffinity()`, hỗ trợ liên kết cứng bằng cách cho phép một luồng chỉ định tập hợp các CPU mà nó đủ điều kiện để chạy.

#### 4.4.5. Đa xử lý không đồng nhất

Trong các ví dụ mà chúng ta đã thảo luận cho đến nay, tất cả các bộ xử lý đều giống nhau về khả năng của chúng, do đó cho phép mọi luồng chạy trên bất kỳ lõi xử lý nào. Sự khác biệt duy nhất là thời gian truy cập bộ nhớ có thể thay đổi dựa trên cân bằng tải và chính sách chung của bộ xử lý.

Mặc dù các hệ thống di động hiện nay bao gồm các kiến trúc đa lõi, một số hệ thống hiện được thiết kế bằng cách sử dụng các lõi chạy cùng một tập lệnh, nhưng khác nhau về tốc độ và năng lượng. Các hệ thống như vậy được gọi là đa xử lý không đồng nhất (heterogeneous multiprocessing - HMP). Lưu ý rằng đây không phải là một dạng đa xử lý không đối xứng như được mô tả trong Phần 4.4.1 vì cả tác vụ của hệ thống và người dùng đều có thể chạy trên bất kỳ lõi nào. Thay vào đó, HMP là quản lý tốt hơn mức tiêu thụ điện năng bằng cách giao nhiệm vụ cho một số lõi nhất định dựa trên nhu cầu cụ thể của nhiệm vụ.

Đối với các bộ xử lý ARM hỗ trợ nó, kiểu kiến trúc này được gọi là big.LITTLE trong đó các lõi lớn có hiệu suất cao hơn được kết hợp với các lõi nhỏ tiết kiệm năng lượng. Các lõi lớn tiêu thụ năng lượng lớn hơn và do đó chỉ nên được sử dụng trong thời gian ngắn. Tương tự như vậy, các lõi nhỏ sử dụng ít năng lượng hơn và do đó có thể được sử dụng trong thời gian dài hơn.

Có một số lợi thế cho cách tiếp cận này. Bằng cách kết hợp một số lõi chậm hơn với lõi nhanh hơn, bộ lập lịch CPU có thể chỉ định các tác vụ không yêu cầu hiệu suất cao nhưng có thể cần chạy trong thời gian dài hơn, (chẳng hạn như các tác vụ nền) cho các lõi nhỏ, do đó giúp tiết kiệm pin. Tương tự, các ứng dụng tương tác đòi hỏi nhiều sức mạnh xử lý hơn, nhưng có thể chạy trong thời gian ngắn hơn, có thể được gán cho các lõi lớn. Ngoài ra, nếu thiết bị di động đang ở chế độ tiết kiệm năng lượng, các lõi

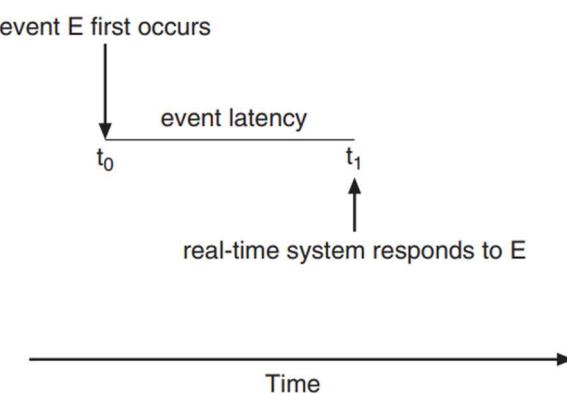
lớn sử dụng nhiều năng lượng có thể bị vô hiệu hóa và hệ thống chỉ có thể dựa vào các lõi nhỏ tiết kiệm năng lượng. Windows 10 hỗ trợ lập lịch HMP bằng cách cho phép một luồng chọn chính sách lập lịch hỗ trợ tốt nhất cho nhu cầu quản lý năng lượng của nó.

## 4.5. Lập lịch thời gian thực

Lập lịch CPU cho hệ điều hành thời gian thực liên quan đến các vấn đề đặc biệt. Nói chung, chúng ta có thể phân biệt giữa hệ thống thời gian thực mềm và hệ thống thời gian thực cứng. Hệ thống thời gian thực mềm không đảm bảo khi nào một tiến trình thời gian thực quan trọng sẽ được lên lịch. Nó chỉ đảm bảo rằng tiến trình sẽ được ưu tiên hơn các tiến trình không quan trọng. Các hệ thống thời gian thực cứng có yêu cầu khắt khe hơn. Một nhiệm vụ phải được hoàn thành trước thời hạn của nó. Trong phần này, chúng ta tìm hiểu một số vấn đề liên quan đến lập lịch tiến trình trong cả hệ điều hành thời gian thực mềm và cứng.

### 4.5.1. Cực tiểu hóa độ trễ

Xét bản chất hướng sự kiện của hệ thống thời gian thực. Hệ thống thường chờ một sự kiện thời gian thực xảy ra. Các sự kiện có thể phát sinh trong phần mềm (như khi bộ hẹn giờ hết hạn), hoặc phần cứng (như khi một phương tiện được điều khiển từ xa phát hiện ra rằng nó đang đến gần chướng ngại vật). Khi một sự kiện xảy ra, hệ thống phải phản hồi và phục vụ nó càng nhanh càng tốt. Chúng ta đề cập đến độ trễ của sự kiện là khoảng thời gian trôi qua từ khi sự kiện xảy ra đến khi nó được phục vụ (Hình 4.13).



**Hình 4.13. Độ trễ sự kiện.**

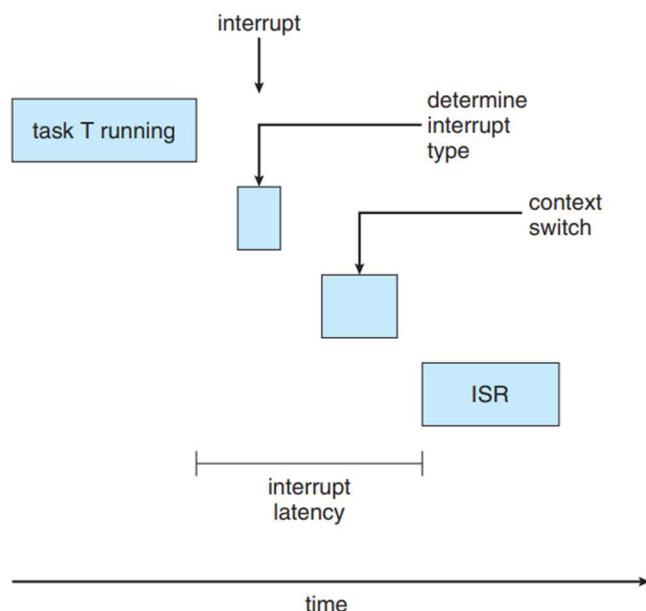
Thông thường, các sự kiện khác nhau có các yêu cầu về độ trễ khác nhau. Ví dụ: yêu cầu về độ trễ đối với hệ thống chống bó cứng phanh có thể là 3 đến 5 mili giây. Tức là, kể từ thời điểm một bánh xe đầu tiên phát hiện ra rằng nó đang trượt, hệ thống điều khiển phanh chống bó cứng có 3 đến 5 phần nghìn giây để phản hồi và kiểm soát tình huống. Bất kỳ phản hồi nào mất nhiều thời gian hơn có thể dẫn đến việc ô tô mất kiểm soát. Ngược lại, một hệ thống nhúng điều khiển radar trong máy bay có thể chịu được

khoảng thời gian trễ vài giây.

Hai loại độ trễ ảnh hưởng đến hiệu suất của hệ thống thời gian thực:

1. Độ trễ ngắt
2. Độ trễ điều phối

Độ trễ ngắt đề cập đến khoảng thời gian từ khi xuất hiện một ngắt ở CPU đến khi bắt đầu tiến trình phục vụ ngắt. Khi một ngắt xảy ra, trước tiên hệ điều hành phải hoàn thành lệnh mà nó đang thực thi và xác định loại ngắt đã xảy ra. Sau đó, nó phải lưu trạng thái của tiến trình hiện tại trước khi phục vụ ngắt bằng cách sử dụng tiến trình dịch vụ ngắt cụ thể (ISR). Tổng thời gian cần thiết để thực hiện các tác vụ này là độ trễ ngắt (Hình 4.14).



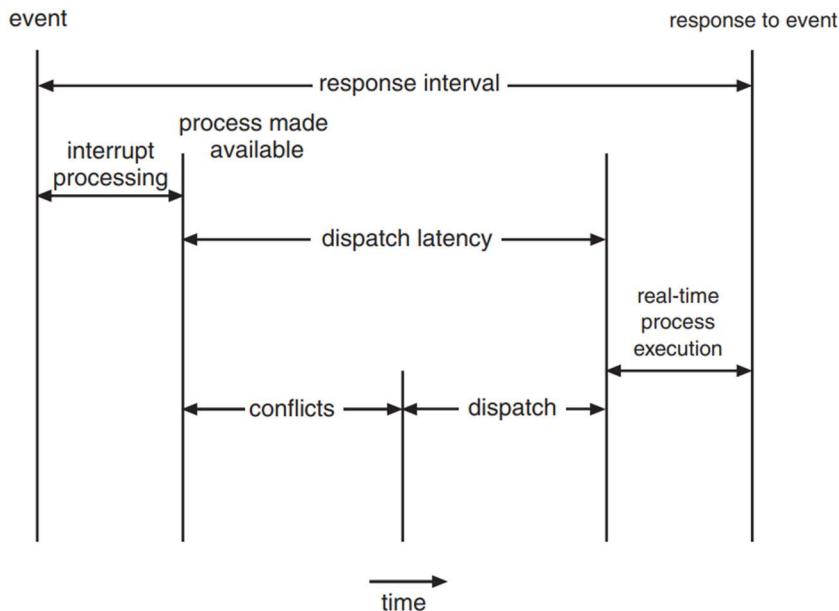
**Hình 4.14. Độ trễ ngắt.**

Rõ ràng, điều quan trọng đối với các hệ điều hành thời gian thực là giảm thiểu độ trễ ngắt để đảm bảo rằng các tác vụ thời gian thực nhận được sự chú ý ngay lập tức. Đối với các hệ thống thời gian thực cứng, độ trễ ngắt không chỉ đơn giản được giảm thiểu, mà nó phải được giới hạn để đáp ứng các yêu cầu nghiêm ngặt của các hệ thống này.

Một yếu tố quan trọng góp phần vào độ trễ ngắt là lượng thời gian ngắt có thể bị vô hiệu hóa trong khi cấu trúc dữ liệu hạt nhân đang được cập nhật. Hệ điều hành thời gian thực yêu cầu ngắt chỉ được vô hiệu hóa trong khoảng thời gian rất ngắn.

Khoảng thời gian cần thiết để điều phối lập lịch dừng một tiến trình và bắt đầu một tiến trình khác được gọi là độ trễ điều phối. Việc cung cấp các tác vụ thời gian thực với quyền truy cập ngay lập tức vào CPU có nghĩa là hệ điều hành thời gian thực cũng giảm

thiểu độ trễ này. Kỹ thuật hiệu quả nhất để giữ cho độ trễ điều phối thấp là cung cấp các nhân ưu tiên. Đối với các hệ thống thời gian thực cứng, độ trễ điều phối thường được đo bằng vài micro giây.



**Hình 4.15. Độ trễ điều phối.**

Trong Hình 4.15, chúng ta vẽ sơ đồ cấu tạo của độ trễ điều phối. Giai đoạn xung đột về độ trễ điều phối có hai thành phần:

1. Dự đoán bất kỳ tiến trình nào đang chạy trong nhân

2. Giải phóng tài nguyên của tiến trình ưu tiên thấp mà cần thiết của tiến trình ưu tiên cao.

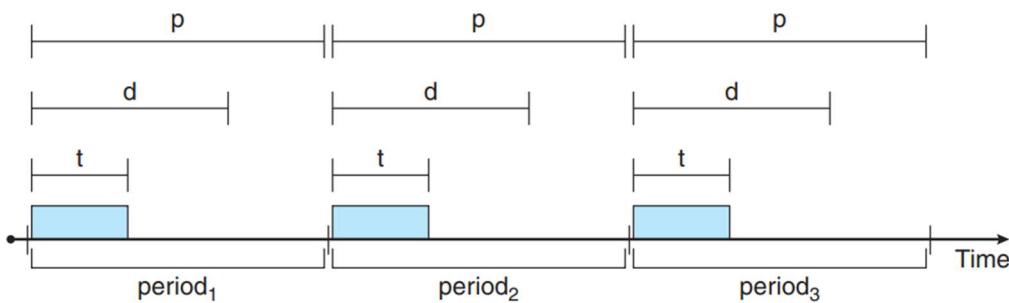
Tiếp theo giai đoạn xung đột, giai đoạn điều phối lập lịch cho tiến trình ưu tiên cao trên cho một CPU có sẵn.

#### 4.5.2. Lập lịch dựa trên mức độ ưu tiên

Tính năng quan trọng nhất của hệ điều hành thời gian thực là đáp ứng ngay lập tức một tiến trình thời gian thực ngay khi tiến trình đó yêu cầu CPU. Do đó, bộ lập lịch cho hệ điều hành thời gian thực phải hỗ trợ thuật toán dựa trên mức độ ưu tiên với cơ chế không độc quyền. Nhớ lại rằng các thuật toán lập lịch dựa trên mức độ ưu tiên gán mỗi tiến trình một mức độ ưu tiên dựa trên tầm quan trọng của nó; những nhiệm vụ quan trọng hơn được gán mức độ ưu tiên cao hơn những nhiệm vụ ít quan trọng hơn. Nếu bộ lập lịch cũng hỗ trợ quyền ưu tiên, một tiến trình hiện đang chạy trên CPU sẽ được nhường cho tiến trình có mức độ ưu tiên cao hơn để chạy.

Tuy nhiên, trước khi tìm hiểu chi tiết về lập lịch, ta phải xác định các đặc điểm của

các tiến trình sẽ được lập lịch. Đầu tiên, đặc điểm chu kỳ tiến trình. Tiến trình yêu cầu CPU ở những khoảng thời gian không đổi (chu kỳ). Khi một tiến trình yêu cầu CPU, nó có thời gian xử lý cố định  $t$ , thời hạn  $d$  mà nó phải được CPU phục vụ và chu kỳ  $p$ . Mối quan hệ của thời gian xử lý, thời hạn và chu kỳ có thể được biểu thị bằng công thức  $0 \leq t \leq d \leq p$ . Chu kỳ là  $1/p$ . Hình 4.16 minh họa việc thực hiện một tiến trình có chu kỳ. Bộ lập lịch trinh có thể tận dụng những đặc điểm này và chỉ định chu kỳ theo thời hạn.



**Hình 4.16. Các nhiệm vụ chu kỳ.**

#### 4.5.3. Lập lịch đơn điệu tỉ lệ

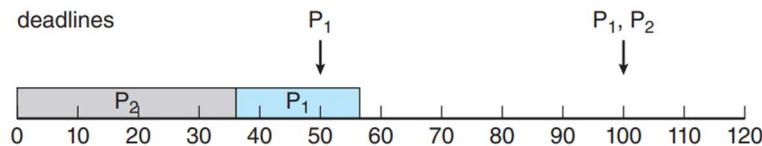
Thuật toán lập lịch đơn điệu tỉ lệ lập lịch các nhiệm vụ chu kỳ bằng cách sử dụng chính sách ưu tiên tĩnh với cơ chế không độc quyền. Nếu một tiến trình có mức độ ưu tiên thấp hơn đang chạy và một tiến trình có mức độ ưu tiên cao hơn có sẵn để chạy, nó sẽ tạm dừng. Khi vào hệ thống, mỗi nhiệm vụ chu kỳ được chỉ định một mức độ ưu tiên nghịch đảo của chu kỳ. Khoảng thời gian càng ngắn, mức độ ưu tiên càng cao; thời gian càng dài, mức độ ưu tiên càng thấp.

Xét ví dụ, có hai tiến trình,  $P_1$  và  $P_2$ . Chu kỳ cho  $P_1$  và  $P_2$  tương ứng là 50 và 100 - nghĩa là,  $p_1 = 50$  và  $p_2 = 100$ . Thời gian xử lý là  $t_1 = 20$  cho  $P_1$  và  $t_2 = 35$  cho  $P_2$ . Thời hạn cho mỗi tiến trình yêu cầu nó phải hoàn thành thời gian CPU trước khi bắt đầu chu kỳ tiếp theo, tức là  $d=p$ .

Trước tiên, chúng ta phải xác định có thể lập lịch những công việc này để mỗi công việc đáp ứng đúng thời hạn của nó hay không. Nếu ta đo mức sử dụng CPU của một tiến trình  $P_i$  là tỷ số giữa thời gian dùng với chu kỳ của nó -  $t_i/p_i$  - thì mức sử dụng CPU của  $P_1$  là  $20/50 = 0,40$  và của  $P_2$  là  $35/100 = 0,35$ , và tổng sử dụng CPU là 75%. Do đó, có thể lập lịch các nhiệm vụ này theo cách vừa đáp ứng thời hạn của chúng mà vẫn để CPU có các chu kỳ khả dụng.

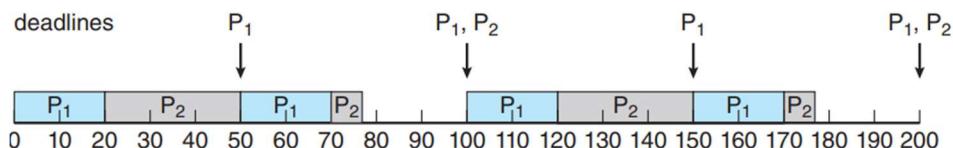
Giả sử chúng ta gán  $P_2$  có mức độ ưu tiên cao hơn  $P_1$ . Việc thực thi  $P_1$  và  $P_2$  trong tình huống này được thể hiện trong Hình 4.17. Như ta thấy,  $P_2$  bắt đầu thực hiện trước và hoàn thành tại thời điểm 35. Tại thời điểm này,  $P_1$  bắt đầu; nó hoàn thành đợt sử dụng

CPU tại thời điểm 55. Tuy nhiên, thời hạn cho  $P_1$  là tại thời điểm 50, vì vậy bộ lập lịch đã khiến  $P_1$  trễ hạn.



**Hình 4.17.** Lập lịch khi  $P_2$  có mức ưu tiên cao hơn  $P_1$ .

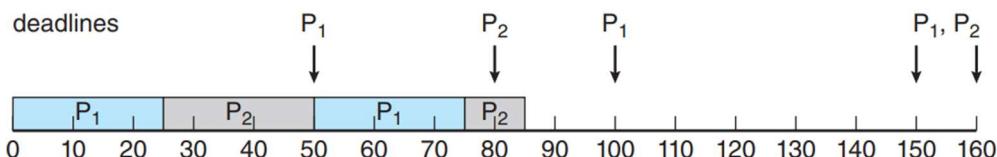
Bây giờ, giả sử ta sử dụng lập lịch đơn điệu tỷ lệ, trong đó chỉ định  $P_1$  mức độ ưu tiên cao hơn  $P_2$  vì chu kỳ của  $P_1$  ngắn hơn  $P_2$ . Việc thực hiện các tiến trình trong tình huống này được thể hiện trong Hình 4.18.  $P_1$  chạy đầu tiên và hoàn thành sử dụng CPU của nó ở thời điểm 20, do đó đáp ứng thời hạn.  $P_2$  bắt đầu chạy tại thời điểm này và chạy cho đến thời điểm 50. Tại thời điểm này, nó ưu tiên cho  $P_1$ , mặc dù nó vẫn còn 5 mili giây sử dụng CPU.  $P_1$  hoàn tất thời gian dùng CPU của nó tại thời điểm 70, tại thời điểm đó bộ lập lịch tiếp tục  $P_2$ .  $P_2$  hoàn thành sử dụng CPU của nó ở thời điểm 75, cũng là thời hạn đầu tiên của nó. Hệ thống không hoạt động cho đến thời điểm 100, khi  $P_1$  được lên lịch lại.



**Hình 4.18.** Lập lịch đơn điệu tỉ lệ.

Lập lịch đơn điệu tỷ lệ được coi là tối ưu ở chỗ nếu thuật toán này không thể lập lịch một tập các tiến trình, thì nó không thể được lập lịch bằng bất kỳ thuật toán nào khác chỉ định các ưu tiên tĩnh. Tiếp theo, chúng ta hãy kiểm tra một tập hợp các tiến trình không thể lập lịch sử dụng thuật toán đơn điệu tỷ lệ.

Giả sử rằng tiến trình  $P_1$  có chu kỳ là  $p_1 = 50$  và thời gian dùng CPU là  $t_1 = 25$ . Đối với  $P_2$ , các giá trị tương ứng là  $p_2 = 80$  và  $t_2 = 35$ . Lập lịch đơn điệu tỷ lệ sẽ gán cho tiến trình  $P_1$  một mức độ ưu tiên cao hơn, vì nó có thời gian ngắn hơn. Tổng mức sử dụng CPU của hai tiến trình là  $(25/50) + (35/80) = 0,94$ , và do đó, có vẻ hợp lý khi hai tiến trình có thể được lên lịch và vẫn để CPU có 6% thời gian khả dụng. Hình 4.19 thể hiện lập lịch các tiến trình  $P_1$  và  $P_2$ . Ban đầu,  $P_1$  chạy cho đến khi nó hoàn thành thời gian CPU tại thời điểm 25. Sau đó, tiến trình  $P_2$  bắt đầu chạy và chạy cho đến thời điểm 50. Tại thời điểm này,  $P_2$  vẫn còn 10 mili giây trong thời gian dùng CPU của nó. Tiến trình  $P_1$  chạy cho đến thời điểm 75; do đó,  $P_2$  kết thúc sử dụng CPU ở thời điểm 85, sau thời hạn hoàn thành của nó ở thời điểm 80.



**Hình 4.19.** Ví dụ không thể lập lịch đơn điệu tỉ lệ.

Mặc dù là tối ưu, nhưng lập lịch đơn điệu tỉ lệ có một hạn chế: việc sử dụng CPU bị giới hạn và không phải lúc nào cũng có thể tối đa hóa tài nguyên CPU.

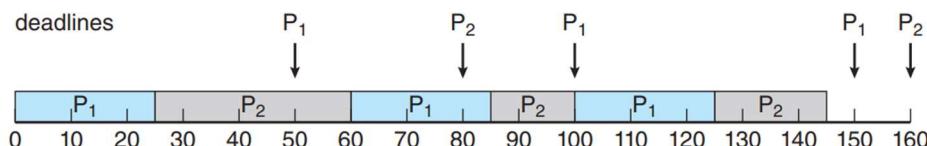
Với một tiến trình trong hệ thống, việc sử dụng CPU là 100%, nhưng nó giảm xuống còn xấp xỉ 69% khi số lượng tiến trình tiến đến vô cùng. Với hai tiến trình, việc sử dụng CPU bị giới hạn ở khoảng 83%. Sử dụng CPU kết hợp cho hai tiến trình được lập lịch trong Hình 4.17 và Hình 4.18 là 75%; do đó, thuật toán lập lịch đơn điệu tỉ lệ được đảm bảo lập lịch để nó có thể đúng thời hạn. Đối với hai tiến trình được lập lịch trong Hình 4.19, mức sử dụng CPU kết hợp là khoảng 94%; do đó, lập lịch đơn điệu tỉ lệ không thể đảm bảo rằng chúng có thể được lập lịch để đúng thời hạn.

#### 4.5.4. Lập lịch thời hạn ngắn nhất (EDF)

Lập lịch "Earliest-deadline-first" (EDF) sẽ chỉ định các mức độ ưu tiên một cách linh hoạt theo thời hạn. Thời hạn càng sớm, ưu tiên càng cao; thời hạn càng muộn, mức độ ưu tiên càng giảm. Theo chính sách EDF, khi một tiến trình có thể chạy được, nó phải thông báo các yêu cầu về thời hạn của nó cho hệ thống. Các mức ưu tiên có thể phải được điều chỉnh để phản ánh thời hạn của tiến trình có thể chạy được.

Để minh họa cho việc lập lịch EDF, ta lập lịch lại cho các tiến trình thể hiện trong Hình 4.19, tiến trình này không đáp ứng được các yêu cầu về thời hạn theo lập lịch đơn điệu tỉ lệ. Nhớ lại rằng P<sub>1</sub> có các giá trị  $p_1 = 50$  và  $t_1 = 25$  và P<sub>2</sub> có các giá trị  $p_2 = 80$  và  $t_2 = 35$ . Lập lịch EDF cho các tiến trình này được thể hiện trong Hình 4.20. Tiến trình P<sub>1</sub> có thời hạn sớm nhất, vì vậy mức độ ưu tiên ban đầu của nó cao hơn tiến trình P<sub>2</sub>. Sau khi tiến trình P<sub>1</sub> kết thúc thời gian CPU tiến trình P<sub>2</sub> bắt đầu chạy. Tuy nhiên, trong khi lập lịch đơn điệu tỉ lệ cho phép P<sub>1</sub> trước P<sub>2</sub> vào đầu chu kỳ tiếp theo của nó tại thời điểm 50, lập lịch EDF cho phép tiến trình P<sub>2</sub> tiếp tục chạy. P<sub>2</sub> hiện có mức độ ưu tiên cao hơn P<sub>1</sub> vì thời hạn tiếp theo của nó (tại thời điểm 80) sớm hơn so với P<sub>1</sub> (tại thời điểm 100). Do đó, cả P<sub>1</sub> và P<sub>2</sub> đều đáp ứng thời hạn đầu tiên của chúng. Tiến trình P<sub>1</sub> lại bắt đầu chạy ở thời điểm 60 và hoàn thành thời gian CPU thứ hai của nó ở thời điểm 85, cũng đáp ứng thời hạn thứ hai tại thời điểm 100. P<sub>2</sub> bắt đầu chạy tại thời điểm này, và phải nhường P<sub>1</sub> khi bắt đầu chu kỳ tiếp theo tại thời điểm 100. P<sub>2</sub> phải nhường P<sub>1</sub> vì P<sub>1</sub> có thời hạn sớm hơn (thời gian 150) so với P<sub>2</sub> (thời gian 160). Tại thời điểm 125, P<sub>1</sub> hoàn thành thời gian CPU và P<sub>2</sub> tiếp tục thực thi, kết thúc ở thời điểm 145 và cũng đáp

ứng thời hạn của nó. Hệ thống không hoạt động cho đến thời điểm 150, khi P<sub>1</sub> được lên lịch chạy lại.



**Hình 4.20.** Lập lịch thời hạn ngắn nhất trước.

Không giống như thuật toán đơn điệu tỷ lệ, lập lịch EDF không yêu cầu các tiến trình phải tuần hoàn, cũng không phải một tiến trình yêu cầu lượng thời gian CPU không đổi trên mỗi cụm. Yêu cầu duy nhất là một tiến trình thông báo thời hạn của nó cho bộ lập lịch khi nó có thể chạy được. Sự hấp dẫn của lập lịch EDF là nó tối ưu về mặt lý thuyết - về mặt lý thuyết, nó có thể lập lịch để mỗi tiến trình có thể đáp ứng các yêu cầu về thời hạn của nó và việc sử dụng CPU sẽ là 100%. Tuy nhiên, trong thực tế, không thể đạt được mức sử dụng CPU này do chi phí chuyển đổi ngữ cảnh giữa các tiến trình và xử lý ngắn.

#### 4.5.5. Lập lịch theo tỉ lệ cổ phiếu

Lập lịch theo tỉ lệ cổ phiếu hoạt động bằng cách phân bổ T cổ phần giữa tất cả các ứng dụng. Một ứng dụng có thể nhận N cổ phiếu thời gian, do đó đảm bảo rằng ứng dụng sẽ có N/T trong tổng thời gian xử lý. Ví dụ, giả sử rằng tổng số T = 100 cổ phiếu được chia cho ba tiến trình, A, B và C. A được nhận 50 cổ phiếu, B được nhận 15 cổ phiếu và C được nhận 20 cổ phiếu. Điều này nghĩa rằng A sẽ có 50% tổng thời gian xử lý, B sẽ có 15% và C sẽ có 20%.

Lập lịch theo tỉ lệ cổ phiếu phải làm việc kết hợp với cơ chế kiểm soát tiếp nhận để đảm bảo rằng ứng dụng nhận được cổ phiếu. Chính sách kiểm soát tiếp nhận sẽ chỉ chấp nhận một tiến trình yêu cầu một số lượng cổ phiếu vừa đủ. Trong ví dụ hiện tại, ta đã phân bổ  $50 + 15 + 20 = 85$  cổ phiếu trong tổng số 100 cổ phiếu. Nếu một tiến trình mới D yêu cầu 30 cổ phiếu, bộ kiểm soát tiếp nhận sẽ từ chối cho D vào hệ thống.

#### 4.6. Cơ chế lập lịch trong Window

Windows lập lịch các tiến trình bằng cách sử dụng cơ chế lập lịch không độc quyền kết hợp với độ ưu tiên. Bộ lập lịch của Windows đảm bảo rằng tiến trình có mức độ ưu tiên cao nhất sẽ luôn chạy. Phần nhân Windows xử lý việc lập lịch được gọi là bộ điều phối (Dispatcher). Một tiến trình được chọn để chạy bởi bộ điều phối sẽ chạy cho đến khi: nó gặp một tiến trình có mức độ ưu tiên cao hơn, cho đến khi nó kết thúc, cho đến khi hết lượng tử thời gian hoặc cho đến khi nó thực hiện một lệnh gọi hệ thống chặn, chẳng hạn như đối với I/O. Nếu tiến trình gian thực có độ ưu tiên cao hơn sẵn sàng trong

khi tiến trình có độ ưu tiên thấp hơn đang chạy, thì luồng có mức độ ưu tiên thấp hơn sẽ nhường CPU.

Bộ điều phối sử dụng lược đồ 32 mức ưu tiên để xác định thứ tự thực hiện tiến trình. Các ưu tiên được chia thành hai lớp. Lớp biến đổi chứa các tiến trình có mức độ ưu tiên từ 1 đến 15 và lớp thời gian thực chứa các tiến trình có mức độ ưu tiên từ 16 đến 31. (Cũng có một luồng đang chạy ở mức ưu tiên 0 được sử dụng để quản lý bộ nhớ.) Bộ điều phối sử dụng một hàng đợi cho mỗi mức độ ưu tiên lập lịch và duyệt qua tập hợp các hàng đợi từ cao nhất đến thấp nhất cho đến khi nó tìm thấy một tiến trình sẵn sàng chạy. Nếu không tìm thấy tiến trình sẵn sàng, bộ điều phối sẽ thực thi một luồng đặc biệt được gọi là luồng rỗi.

Có một mối quan hệ giữa các ưu tiên của lõi Windows và API Windows. API Windows xác định một tiến trình có thể nhận sáu lớp ưu tiên sau:

- IDLE\_PRIORITY\_CLASS
- BELOW\_NORMAL\_PRIORITY\_CLASS
- NORMAL\_PRIORITY\_CLASS
- ABOVE\_NORMAL\_PRIORITY\_CLASS
- HIGH\_PRIORITY\_CLASS
- REALTIME\_PRIORITY\_CLASS

Các tiến trình thường là thành viên của lớp NORMAL\_PRIORITY\_CLASS. Một tiến trình thuộc về lớp này trừ khi tiến trình cha của nó là thành viên của lớp IDLE\_PRIORITY\_CLASS hoặc trừ khi một lớp khác được chỉ định khi tiến trình được tạo. Ngoài ra, quyền ưu tiên của lớp các tiến trình có thể được thay đổi bằng hàm SetPosystemClass() trong Windows API. Mức độ ưu tiên trong tất cả các lớp ngoại trừ REALTIME\_PRIORITY\_CLASS có thể thay đổi.

Một tiến trình trong một lớp ưu tiên nhất định cũng có mức ưu tiên tương đối. Các giá trị cho các ưu tiên tương đối bao gồm:

- IDLE
- LOWEST
- BELOW\_NORMAL
- NORMAL
- ABOVE\_NORMAL

- HIGHEST
- TIME\_CRITICAL

Mức độ ưu tiên của mỗi tiến trình dựa trên cả lớp ưu tiên mà nó thuộc về và mức độ ưu tiên tương đối của nó trong lớp đó. Mỗi quan hệ này được thể hiện trong Bảng 4.1. Giá trị của các lớp ưu tiên xuất hiện ở hàng trên cùng. Cột bên trái chứa các giá trị cho các mức độ ưu tiên tương đối. Ví dụ: nếu mức độ ưu tiên tương đối của một luồng trong ABOVE\_NORMAL\_PRIORITY\_CLASS, thì mức độ ưu tiên của luồng đó là 10.

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Bảng 4.1. Các mức ưu tiên trong luồng của Window.

#### 4.7. Tóm tắt

- Lập lịch cho CPU là nhiệm vụ chọn một tiến trình chờ từ hàng đợi sẵn sàng và phân bổ CPU cho nó. CPU được phân bổ cho tiến trình đã chọn bởi bộ điều phối.
- Các thuật toán lập lịch có thể là ưu tiên (không độc quyền - trong đó CPU có thể bị lấy đi khỏi một tiến trình) hoặc không ưu tiên (độc quyền - trong đó một tiến trình phải tự nguyện từ bỏ quyền kiểm soát CPU). Hầu hết tất cả các hệ điều hành hiện đại đều dùng cơ chế ưu tiên.
- Các thuật toán lập lịch có thể được đánh giá theo năm tiêu chí sau: (1) Sử dụng CPU, (2) thông lượng, (3) thời gian quay vòng, (4) thời gian chờ và (5) thời gian phản hồi.
- Lập lịch đến trước, được phục vụ trước (FCFS) là thuật toán lập lịch đơn giản nhất, nhưng nó có thể khiến các tiến trình ngắn phải chờ các tiến trình rất dài.
- Lập lịch việc ngắn nhất thực hiện trước (SJF) là tối ưu nhất vì cho thời gian chờ trung bình ngắn nhất. Tuy nhiên, việc thực hiện lập lịch SJF là rất khó vì việc dự đoán độ dài của lần sử dụng CPU tiếp theo là rất khó.
- Lập lịch round-robin (RR) phân bổ CPU cho mỗi tiến trình cho một lượng tử thời gian. Khi tiến trình hết thời gian cho phép, thì tiến trình này sẽ nhường cho một tiến

trình khác được lên lịch chạy với lượng tử thời gian.

- Lập lịch ưu tiên chỉ định mỗi tiến trình một mức độ ưu tiên và CPU được phân bổ cho tiến trình có mức độ ưu tiên cao nhất. Các tiến trình có cùng mức độ ưu tiên có thể được lập lịch theo thứ tự FCFS hoặc sử dụng lập lịch RR.

- Lập lịch hàng đợi đa cấp phân các tiến trình thành một số hàng đợi riêng biệt được sắp xếp theo mức độ ưu tiên, và bộ lập lịch thực thi các tiến trình trong hàng đợi có mức độ ưu tiên cao nhất. Các thuật toán lập lịch khác nhau có thể được sử dụng trong mỗi hàng đợi.

- Hàng đợi phản hồi đa cấp tương tự như hàng đợi đa cấp, ngoại trừ việc một tiến trình có thể di chuyển giữa các hàng đợi khác nhau.

- Bộ xử lý đa lõi đặt một hoặc nhiều CPU trên cùng một chip vật lý và mỗi CPU có thể có nhiều luồng phần cứng. Từ quan điểm của hệ điều hành, mỗi luồng phần cứng dường như là một CPU logic.

- Cân bằng tải trên các hệ thống đa lõi là cân bằng tải giữa các lõi CPU, mặc dù việc di chuyển luồng giữa các lõi để cân bằng tải có thể làm mất hiệu lực nội dung bộ đệm và do đó có thể tăng thời gian truy cập bộ nhớ.

- Lập lịch thời gian thực mềm ưu tiên cho các tác vụ thời gian thực hơn các tác vụ không theo thời gian thực. Lập lịch thời gian cứng cung cấp đảm bảo về thời gian cho các tác vụ thời gian thực,

- Lập lịch thời gian thực đơn điệu theo tỷ lệ lập lịch các nhiệm vụ định kỳ bằng cách sử dụng chính sách ưu tiên tĩnh với cơ chế không độc quyền.

- Lập lịch sớm Earliest-deadline-first (EDF) phân công các ưu tiên theo thời hạn. Thời hạn càng sớm, ưu tiên càng cao; thời hạn càng muộn, mức độ ưu tiên càng giảm.

- Lập lịch chia sẻ theo tỷ lệ phân bổ T chia sẻ giữa tất cả các ứng dụng. Nếu một ứng dụng được phân bổ N phần thời gian, nó được đảm bảo có  $N/T$  trong tổng thời gian xử lý.

- Lập lịch của Windows sử dụng lược đồ ưu tiên 32 cấp trước, để xác định thứ tự lập lịch luồng.

#### 4.8. Câu hỏi ôn tập

**Câu 4.1.** Thuật toán lập lịch CPU xác định thứ tự thực hiện các tiến trình đã lên lịch của nó. Cho n tiến trình được lập lịch trên một bộ xử lý, có bao nhiêu lịch trình khác nhau? Đưa ra công thức về n.

**Câu 4.2.** Giải thích sự khác biệt giữa lập lịch không độc quyền (preemptive) và độc quyền (Nonpreemptive).

**Câu 4.3.** Giả sử rằng các tiến trình sau đến hàng đợi sẵn sàng vào thời gian được chỉ định. Mỗi tiến trình sẽ chạy trong khoảng thời gian được liệt kê:

Tiến trình	Thời gian đến (ms)	Thời cần CPU (ms)
$P_1$	0.0	8
$P_2$	0.4	4
$P_3$	1.0	1

Sử dụng cơ chế lập lịch độc quyền:

- Tính thời gian quay vòng trung bình của các tiến trình này với thuật toán FCFS.
- Tính thời gian quay vòng trung bình của các tiến trình này với thuật toán SJF.

**Câu 4.4.** Cho các tiến trình sau với thời gian sử dụng CPU và độ ưu tiên tương ứng:

Tiến trình	Thời gian (ms)	Mức ưu tiên
$P_1$	2	2
$P_2$	1	1
$P_3$	8	4
$P_4$	4	2
$P_5$	5	3

Các tiến trình được giả định là đã đến hàng đợi theo thứ tự  $P_1, P_2, P_3, P_4, P_5$ , tất cả tại thời điểm 0.

- Vẽ bốn biểu đồ Gantt minh họa việc thực hiện các tiến trình này bằng cách sử dụng các thuật toán lập lịch sau: FCFS, SJF, độc quyền với độ ưu tiên (số ưu tiên lớn hơn nghĩa mức ưu tiên cao hơn) và RR (lượng tử = 2).
- Thời gian quay vòng của mỗi tiến trình đối với mỗi thuật toán lập lịch trong phần a là bao nhiêu?
- Thời gian chờ của mỗi tiến trình đối với mỗi thuật toán lập lịch này là bao nhiêu?
- Thuật toán nào dẫn đến thời gian chờ trung bình nhỏ nhất (tất cả các tiến trình)?

**Câu 4.5.** Cho các tiến trình sau với thời gian sử dụng CPU và độ ưu tiên tương ứng:

Tiến trình	Thời gian sử dụng CPU	Mức ưu tiên
$P_1$	5	4
$P_2$	3	1
$P_3$	1	2
$P_4$	7	2
$P_5$	4	3

Các tiến trình được giả định là đã đến theo thứ tự  $P_1, P_2, P_3, P_4, P_5$ , tất cả tại thời điểm 0.

- Vẽ bốn biểu đồ Gantt minh họa việc thực hiện các tiến trình này bằng cách sử dụng các thuật toán lập lịch sau: FCFS, SJF, độc quyền với mức ưu tiên (số ưu tiên lớn hơn ngụ ý mức ưu tiên cao hơn) và RR (lượng tử = 2).
- Thời gian quay vòng của mỗi tiến trình đối với mỗi thuật toán lập lịch trong phần a là bao nhiêu?
- Thời gian chờ của mỗi tiến trình đối với mỗi thuật toán lập lịch này là bao nhiêu?
- Thuật toán nào có thời gian chờ trung bình tối thiểu (trên tất cả các tiến trình)?

**Câu 4.6.** Cho các tiến trình sau được lập lịch bằng thuật toán RR với cơ chế không độc quyền:

Tiến trình	Độ ưu tiên	Thời sử dụng CPU	Thời gian đến
$P_1$	40	20	0
$P_2$	30	25	25
$P_3$	30	25	30
$P_4$	35	15	60
$P_5$	5	10	100
$P_6$	10	10	105

Mỗi tiến trình được chỉ định một độ ưu tiên bằng số, với số cao hơn cho biết độ ưu tiên cao hơn. Ngoài các tiến trình được liệt kê ở trên, hệ thống còn có một tác vụ nhàn rỗi (không sử dụng tài nguyên CPU và được gọi là  $P_{idle}$ ). Tác vụ này có mức ưu tiên 0 và được lên lịch bắt cứ khi nào hệ thống không có tiến trình nào khác để chạy. Độ dài của lượng tử thời gian là 10 đơn vị. Nếu một tiến trình nào bị nhường CPU bởi một tiến trình có độ ưu tiên cao hơn, thì tiến trình đó sẽ được đặt ở cuối hàng đợi.

- Cho biết thứ tự lập lịch của các tiến trình bằng biểu đồ Gantt.
- Thời gian quay vòng của mỗi tiến trình là bao nhiêu?
- Thời gian chờ cho mỗi tiến trình là bao nhiêu?
- Tỷ lệ sử dụng CPU là bao nhiêu?

**Câu 4.7.** Các tiến trình sau đang được lập lịch sử dụng thuật toán lập lịch RR không độc quyền, dựa trên mức ưu tiên.

Tiến trình	Độ ưu tiên	Thời gian sử dụng CPU	Thời gian đến
$P_1$	8	15	0
$P_2$	3	20	0
$P_3$	4	20	20

$P_4$	4	20	25
$P_5$	5	5	45
$P_6$	5	15	55

Mỗi tiến trình được chỉ định một mức ưu tiên, với số cao hơn cho biết mức độ ưu tiên cao hơn. Bộ lập lịch sẽ thực hiện tiến trình có mức ưu tiên cao nhất. Đối với các tiến trình có cùng mức độ ưu tiên, sử dụng bộ lập lịch RR với lượng tử thời gian là 10 đơn vị. Nếu một tiến trình nhường CPU cho một tiến trình khác có mức ưu tiên cao hơn, thì tiến trình được ưu tiên trước sẽ được đặt ở cuối hàng đợi.

- a. Cho biết thứ tự lập lịch của các tiến trình bằng biểu đồ Gantt.
- b. Thời gian quay vòng của mỗi tiến trình là bao nhiêu?
- c. Thời gian chờ cho mỗi tiến trình là bao nhiêu?

**Câu 4.8.** Nếu những điểm thuận lợi khi sử dụng lượng tử khác nhau cho những hàng đợi khác nhau trong lập lịch hàng đợi đa mức.

**Câu 4.9.** Nhiều thuật toán lập lịch trình CPU được tham số hóa. Ví dụ, thuật toán RR yêu cầu một tham số để chỉ ra lượng tử thời gian. Hàng đợi phản hồi đa cấp yêu cầu các tham số về số lượng hàng đợi, các thuật toán lập lịch cho mỗi hàng đợi, các tiêu chí được sử dụng để di chuyển các tiến trình giữa các hàng đợi, v.v.

Do đó, những thuật toán này thực sự là tập hợp các thuật toán (ví dụ, tập hợp các thuật toán RR cho mọi lượng cắt thời gian, v.v.). Một tập các thuật toán có thể chứa những thuật toán khác (ví dụ, thuật toán FCFS là thuật toán RR với lượng tử thời gian hữu hạn). Mỗi quan hệ nào (nếu có) giữ giữa các cặp tập hợp thuật toán sau đây?

- a. Priority và SJF
- b. Multilevel feedback queues và FCFS
- c. Priority and FCFS
- d. RR và SJF

**Câu 4.10.** Giả sử rằng một thuật toán lập lịch CPU “ủng hộ” những tiến trình đã sử dụng ít thời gian xử lý nhất trong quá khứ gần đây. Tại sao thuật toán này lại ưu tiên các tiến trình ràng buộc I/O nhưng lại không “bỏ sói” vĩnh viễn các tiến trình ràng buộc CPU?

**Câu 4.11.** Trong số hai loại chương trình này:

- a. I/O-ràng buộc (sử dụng nhiều thao tác vào/ra hơn sử dụng CPU)
- b. ràng buộc CPU (sử dụng nhiều CPU hơn các thao tác vào ra).

Cái nào có nhiều khả năng tự nguyện chuyển ngữ cảnh hơn và cái nào có nhiều khả năng không tự nguyện chuyển ngữ cảnh hơn? Giải thích câu trả lời của bạn.

**Câu 4.12.** Thảo luận về cách các cặp tiêu chí lập lịch sau đây xung đột trong các cài đặt nhất định.

- a) Thời gian phản hồi và sử dụng CPU.
- b) Thời gian quay vòng trung bình và thời gian chờ tối đa.
- c) Sử dụng thiết bị I/O và sử dụng CPU.

**Câu 4.13.** Một kỹ thuật để thực hiện lập lịch xô số hoạt động bằng cách ấn định các tiến trình vé số, được sử dụng để phân bổ thời gian CPU. Bất cứ khi nào phải đưa ra quyết định về lịch trình, một vé số được chọn ngẫu nhiên và quá trình giữ vé đó sẽ lấy CPU. Hệ điều hành BTV thực hiện lập lịch xô số bằng cách tổ chức xô số 50 lần mỗi giây, với mỗi người trúng số nhận được 20 mili giây thời gian CPU ( $20 \text{ mili giây} \times 50 = 1 \text{ giây}$ ). Mô tả cách bộ lập lịch BTV có thể đảm bảo rằng các luồng có mức độ ưu tiên cao hơn nhận được nhiều sự chú ý từ CPU hơn các luồng có mức độ ưu tiên thấp hơn.

**Câu 4.14.** Hầu hết các thuật toán lập lịch duy trì một hàng đợi run, hàng đợi này liệt kê các tiến trình đủ điều kiện để chạy trên một bộ xử lý. Trên các hệ thống đa lõi, có hai tùy chọn chung: (1) mỗi lõi xử lý có hàng đợi run riêng của nó, hoặc (2) một hàng đợi chạy run được chia sẻ bởi tất cả các lõi xử lý. Ưu điểm và nhược điểm của từng cách tiếp cận này là gì?

**Câu 4.15.** Một biến thể của công cụ lập lịch RR là lập lịch RR hồi quy. Bộ lập lịch này chỉ định cho mỗi tiến trình một lượng tử thời gian và một mức ưu tiên. Giá trị ban đầu của lượng tử thời gian là 50 mili giây. Tuy nhiên, mỗi khi một tiến trình đã được cấp phát CPU và sử dụng toàn bộ lượng tử thời gian của nó (không chặn I/O), 10 mili giây được thêm vào lượng tử thời gian của nó và mức ưu tiên của nó được tăng lên. (Lượng tử thời gian cho một tiến trình có thể tăng lên tối đa 100 mili giây.) Khi một tiến trình bị chặn trước khi sử dụng toàn bộ lượng tử thời gian của nó, lượng tử thời gian của nó sẽ giảm đi 5 mili giây, nhưng mức độ ưu tiên của nó vẫn như cũ. Loại tiến trình nào (liên kết hạn CPU hoặc liên kết I/O) mà bộ lập lịch RR hồi quy ưu tiên? Giải thích.

**Câu 4.16.** Thuật toán lập lịch nào sau đây có thể dẫn đến tình trạng chờ vô hạn hoặc chết đói?

- a. First-come, first-served
- b. Shortest job first

- c. Round robin
- d. Ưu tiên

**Câu 4.17.** Hãy xem xét một biến thể của thuật toán lập lịch RR trong đó các mục nhập trong hàng đợi sẵn sàng là con trỏ đến PCB.

- a. Hiệu quả của việc đặt hai con trỏ đến cùng một tiến trình trong hàng đợi sẵn sàng là gì?
- b. Hai ưu điểm chính và hai nhược điểm của chương trình này là gì?
- c. Bạn sẽ sửa đổi thuật toán RR cơ bản như thế nào để đạt được hiệu quả tương tự mà không có các con trỏ trùng lặp?

**Câu 4.18.** Hãy xem xét một hệ thống đang chạy mười nhiệm vụ liên kết I/O và một nhiệm vụ liên kết CPU. Giả sử rằng các nhiệm vụ liên kết I/O thực hiện thao tác I/O một lần cho mỗi mili giây tính toán của CPU và mỗi thao tác I/O mất 10 mili giây để hoàn thành. Cũng giả sử rằng chi phí chuyển ngữ cảnh là 0,1 mili giây và tất cả các tiến trình đều là các tác vụ chạy lâu dài. Mô tả việc sử dụng CPU cho bộ lập lịch RR khi:

- a. Lượng tử thời gian là 1 mili giây.
- b. Lượng tử thời gian là 10 mili giây.

**Câu 4.19.** Giải thích cách các thuật toán lập lịch sau “quan tâm” hay “không quan tâm” các tiến trình ngắn:

- a. FCFS
- b. RR
- c. Multilevel Feedback Queues.

**Câu 4.20.** Mô tả lý do tại sao hàng đợi sẵn sàng được chia sẻ cho các CPU có thể gấp sự cố về hiệu suất trong môi trường SMP.

**Câu 4.21.** Hãy xem xét một thuật toán cân bằng tải đảm bảo rằng mỗi hàng đợi có số luồng xấp xỉ bằng nhau, không phụ thuộc vào mức độ ưu tiên. Thuật toán lập lịch dựa trên mức độ ưu tiên sẽ xử lý tình huống này như thế nào cho hiệu quả nếu một hàng đợi đang chạy có tất cả các luồng có mức độ ưu tiên cao và hàng đợi thứ hai có tất cả các luồng có mức độ ưu tiên thấp?

**Câu 4.22.** Cung cấp một tình huống cụ thể minh họa nơi lập lịch đơn điệu tỷ lệ kém hơn so với lập lịch thời hạn sớm nhất đầu tiên trong việc đáp ứng thời hạn tiến trình thời gian thực?

**Câu 4.23.** Xét hai tiến trình,  $P_1$  và  $P_2$ , trong đó  $p_1 = 50$ ,  $t_1 = 25$ ,  $p_2 = 75$  và  $t_2 = 30$ .

a. Hai tiến trình này có thể được lập lịch bằng cách sử dụng lập lịch đơn điệu tỉ lệ không? Minh họa câu trả lời bằng biểu đồ Gantt.

b. Minh họa việc lập lịch hai tiến trình này bằng cách sử dụng lập lịch thời hạn sớm nhất trước (EDF).

**Câu 4.24.** Giải thích tại sao thời gian trễ ngắn và điều phối phải được giới hạn trong một hệ thống thời gian thực cứng.

**Câu 4.25.** Mô tả những ưu điểm của việc sử dụng đa xử lý không đồng nhất trong hệ thống di động.



## CHƯƠNG 5. ĐỒNG BỘ HÓA TIẾN TRÌNH

Tiến trình hợp tác là một tiến trình có thể ảnh hưởng hoặc bị ảnh hưởng bởi các tiến trình khác đang thực thi trong hệ thống. Các tiến trình hợp tác có thể chia sẻ trực tiếp không gian địa chỉ logic (nghĩa là cả mã và dữ liệu) hoặc chỉ được phép chia sẻ dữ liệu thông qua bộ nhớ dùng chung hoặc truyền thông điệp. Tuy nhiên, việc truy cập đồng thời vào dữ liệu được chia sẻ có thể dẫn đến sự không nhất quán của dữ liệu. Trong chương này, chúng ta thảo luận về các cơ chế khác nhau để đảm bảo các tiến trình hợp tác thực hiện có trật tự trên các không gian địa chỉ, để duy trì tính nhất quán của dữ liệu.

### 5.1. Giới thiệu

Chúng ta đã thấy rằng các tiến trình có thể thực thi đồng thời hoặc song song. Phần trước đã giới thiệu vai trò của lập lịch tiến trình và mô tả cách bộ lập lịch CPU chuyển đổi nhanh chóng giữa các tiến trình để cung cấp khả năng thực thi đồng thời. Điều này có nghĩa là một tiến trình có thể chỉ hoàn thành một phần việc trước khi tiến trình khác được lên lịch. Trên thực tế, một tiến trình có thể bị gián đoạn tại bất kỳ điểm nào trong luồng lệnh của nó và lỗi xử lý có thể được chỉ định để thực thi các lệnh của một tiến trình khác. Ngoài ra, trong thực thi song song, trong đó hai luồng (đại diện cho các tiến trình khác nhau) thực thi đồng thời trên các lỗi xử lý riêng biệt.

Xem xét một mail-Server xử lý e-mail cho một tổ chức. Giả sử chúng ta muốn hệ thống liên tục theo dõi tổng số e-mail đã được gửi kể từ ngày bắt đầu. Và giả sử rằng việc nhận e-mail được xử lý bởi một trong nhiều tiến trình đồng thời. Mỗi khi một trong các tiến trình này nhận được e-mail từ người dùng, tiến trình sẽ tăng biến chia sẻ `mailCount` lên 1. Xem điều gì sẽ xảy ra nếu hai tiến trình cố gắng tăng biến `mailCount` đồng thời. Đầu tiên, giả sử rằng mỗi tiến trình chạy mã hợp ngữ:

```
LOAD mailCount
ADD 1
STORE mailCount
```

Giả sử rằng lệnh LOAD sao chép `mailCount` từ bộ nhớ vào một thanh ghi, lệnh ADD thêm 1 ngay lập tức từ bộ nhớ vào giá trị trong thanh ghi và lệnh STORE sao chép giá trị trong thanh ghi vào bộ nhớ biến `mailCount`.

Giả sử `mailCount` hiện là 21687. Bây giờ, giả sử tiến trình đầu tiên thực hiện các lệnh `LOAD` và `ADD`, do đó để lại 21688 trong thanh ghi (nhưng chưa cập nhật giá trị `mailCount` trong bộ nhớ, vẫn là 21687). Sau đó, do hết lượng tử thời gian, tiến trình đầu tiên mất bộ xử lý và ngũ cảnh hệ thống chuyển sang tiến trình thứ hai. Tiến trình thứ hai hiện thực thi tất cả ba lệnh, do đó đặt `mailCount` thành 21688. Tiến trình này mất bộ xử lý và ngũ cảnh hệ thống chuyển trở lại tiến trình đầu tiên, sau đó tiếp tục bằng cách thực hiện lệnh `STORE` - cũng đặt 21688 vào `mailCount`.

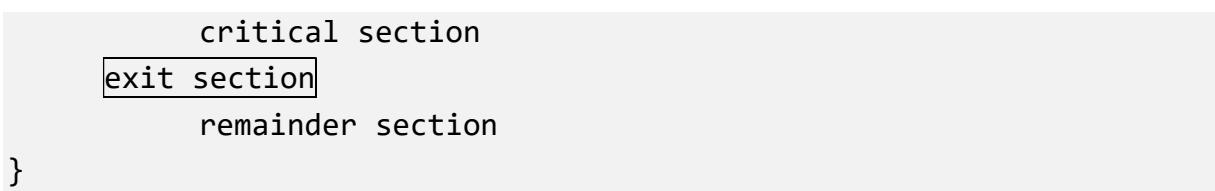
Do quyền truy cập không kiểm soát vào biến `mailCount`, hệ thống đã mất dấu vết của một trong các email - `mailCount` phải là 21689. Ta nhận được tình huống không chính xác này vì đã cho phép cả hai tiến trình thao tác đồng thời với biến `mailCount`. Một tình huống tương tự, trong đó một số tiến trình truy cập và thao tác đồng thời cùng một dữ liệu và kết quả của việc thực thi phụ thuộc vào thứ tự cụ thể mà việc truy cập diễn ra, được gọi là điều kiện tương tranh (race condition). Để tránh điều kiện tương tranh ở trên, chúng ta cần đảm bảo rằng mỗi lần chỉ có một tiến trình có thể thao tác với biến `mailCount`. Việc đảm bảo như vậy gọi là đồng bộ hóa tiến trình.

Các tình huống như vừa mô tả thường xuyên xảy ra trong hệ điều hành khi các phần khác nhau của hệ thống thao tác tài nguyên. Hơn nữa, như chúng ta đã thấy sự phát triển của các hệ thống đa lõi đã làm tăng sự chú trọng vào việc phát triển các ứng dụng đa luồng. Trong các ứng dụng đa luồng, một số luồng - có thể chia sẻ dữ liệu - đang chạy song song trên các lõi xử lý khác nhau. Rõ ràng, chúng ta muốn bất kỳ thay đổi nào phát sinh từ các thao tác đó không ảnh hưởng đến nhau.

## 5.2. Bài toán miền găng (Critical-Section)

Xét một hệ thống gồm  $n$  tiến trình  $\{P_0, P_1, \dots, P_{n-1}\}$ . Mỗi tiến trình có một đoạn mã, được gọi là miền găng, trong đó tiến trình có thể đang cập nhật dữ liệu được chia sẻ với ít nhất một tiến trình khác. Đặc điểm quan trọng là, khi một tiến trình đang thực thi trong miền găng của nó, thì không một tiến trình nào khác được phép thực thi miền găng của mình. Có nghĩa là, không có hai tiến trình nào đang thực thi trong miền găng của chúng cùng một lúc. Bài toán miền găng là thiết kế một giao thức mà các tiến trình có thể sử dụng để đồng bộ hóa hoạt động đồng thời của chúng để chia sẻ dữ liệu. Mỗi tiến trình phải yêu cầu quyền để vào miền găng của nó. Cấu trúc chung của một tiến trình điển hình được thể hiện trong Hình 5.1. Phần vào và phần ra được đặt trong các hộp để làm nổi bật các đoạn mã quan trọng này.

```
while (true){  
    entry section  
}
```

**Hình 5.1.** Cấu trúc mã của bài toán miền găng.

Một giải pháp cho bài toán miền găng phải đáp ứng ba yêu cầu sau:

1. Loại trừ lẫn nhau. Nếu tiến trình  $P_i$  đang thực thi trong miền găng của nó, thì không tiến trình nào khác có thể đang thực thi trong miền găng của chúng.
2. Tiến độ. Nếu không có tiến trình nào đang thực thi trong miền găng của nó và một số tiến trình muốn đi vào các miền găng của chúng, thì chỉ những tiến trình không thực thi trong các phần còn lại (remainder section) của chúng mới có thể tham gia vào việc quyết định cái nào sẽ vào miền găng tiếp theo và lựa chọn này không thể bị hoãn vô thời hạn.
3. Chờ đợi có giới hạn. Tồn tại một giới hạn về số lần các tiến trình khác được phép vào các miền găng của chúng sau khi một tiến trình đã đưa ra yêu cầu vào miền găng của nó và trước khi yêu cầu đó được chấp nhận.

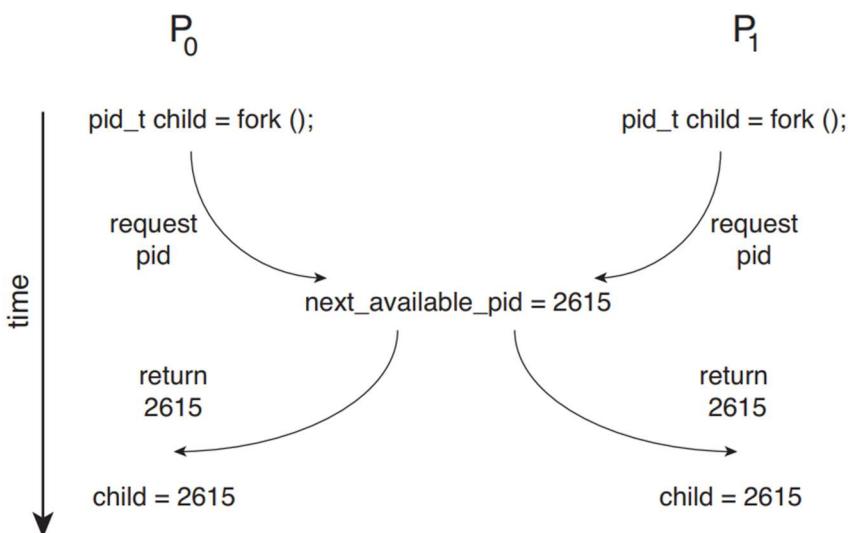
Chúng ta giả định rằng mỗi tiến trình đang thực thi với tốc độ khác nhau. Tại một thời điểm nhất định, nhiều tiến trình chế độ nhân có thể đang hoạt động trong hệ điều hành. Do đó, mã cài đặt hệ điều hành (mã nhân) phải tuân theo một số điều kiện tương tranh có thể xảy ra. Xét tình huống thực tế, cấu trúc dữ liệu nhân duy trì danh sách tất cả các tệp đang mở trong hệ thống. Danh sách này phải được sửa đổi khi một tệp mới được mở hoặc đóng (thêm tệp vào danh sách hoặc xóa tệp khỏi danh sách). Nếu hai tiến trình mở tệp đồng thời, các bản cập nhật riêng biệt cho danh sách này có thể dẫn đến điều kiện tương tranh.

Một ví dụ khác được minh họa trong Hình 5.2. Trong tình huống này, hai tiến trình,  $P_0$  và  $P_1$ , đang tạo các tiến trình con bằng cách sử dụng lệnh gọi hệ thống `fork()` (Linux). Hàm này trả về mã định danh của tiến trình mới được tạo cho tiến trình mẹ. Trong ví dụ này, có một sự tương tranh trên biến nhân `next_available_pid` đại diện cho giá trị của mã định danh tiến trình có sẵn tiếp theo. Khi đó có thể cùng một số nhân dạng tiến trình có thể được gán cho hai tiến trình riêng biệt.

Bài toán miền găng có thể được giải quyết đơn giản trong môi trường đơn lõi nếu chúng ta có thể chặn ngắt xảy ra trong khi một biến được chia sẻ đang được sửa đổi. Bằng cách này, chúng ta có thể chắc chắn rằng chuỗi lệnh hiện tại sẽ được phép thực thi

theo thứ tự mà không có quyền ưu tiên. Không có lệnh khác được chạy, vì vậy không thể thực hiện sửa đổi bất ngờ nào đối với biến được chia sẻ.

Thật không may, giải pháp này không khả thi trong môi trường đa bộ xử lý. Việc tắt ngắt trên bộ đa xử lý có thể tồn tại nhiều thời gian, vì thông báo được chuyển đến tất cả các bộ xử lý. Truyền thông báo này gây chậm trễ khi vào miền găng và hiệu quả của hệ thống giảm. Ngoài ra, hãy xem xét ảnh hưởng đến đồng hồ của hệ thống nếu đồng hồ được cập nhật bằng cách lần ngắt



**Hình 5.2.** Tình trạng tương tranh khi gán pid.

Hai cách tiếp cận chung được sử dụng để xử lý miền găng trong hệ điều hành: nhân không độc quyền và nhân độc quyền. Nhân không độc quyền cho phép một tiến trình được nhường CPU trong khi nó đang chạy ở chế độ nhân. Một nhân độc quyền không cho phép một tiến trình chạy cho đến khi nó thoát khỏi chế độ nhân, chặn hoặc tự nguyện nhường quyền kiểm soát CPU.

Rõ ràng, một nhân độc quyền về cơ bản không có các điều kiện tương tranh trên cấu trúc dữ liệu của nhân, vì chỉ có một tiến trình hoạt động trong nhân tại một thời điểm. Tuy nhiên đối với nhân không độc quyền thì khác, vì vậy chúng phải được thiết kế cẩn thận để đảm bảo rằng dữ liệu nhân chia sẻ không bị ảnh hưởng bởi các điều kiện tương tranh. Các nhân không độc quyền đặc biệt khó thiết kế cho các kiến trúc SMP, vì trong những môi trường này, hai tiến trình chế độ nhân có thể chạy đồng thời trên các lõi CPU khác nhau.

### 5.3. Thuật toán Dekker

Trong phần này, chúng ta xem xét một số nỗ lực để thực hiện loại trừ lẫn nhau. Mỗi lần cài đặt đều chứa một vấn đề mà cài đặt tiếp theo sẽ khắc phục được. Cuối cùng,

trình bày một cách cài đặt phần mềm chính xác để loại trừ lẫn nhau mà không bị tắc nghẽn và trì hoãn vô thời hạn.

### 5.3.1. Phiên bản 1 (đồng bộ hóa bước khóa và chờ khi bận)

Hình 5.3 cho thấy phiên bản đầu tiên trong việc mô tả mã thực thi loại trừ lẫn nhau giữa hai tiến trình. Mã giả được trình bày bằng cú pháp giống C. Các lệnh của mỗi tiến trình có thể được chia thành ba phần: lệnh không ở miền găng (tức là lệnh không sửa đổi dữ liệu được chia sẻ), lệnh trong miền găng (tức là lệnh sửa đổi dữ liệu được chia sẻ) và lệnh đảm bảo loại trừ lẫn nhau (tức là lệnh cài đặt vào miền găng và thoát miền găng). Mỗi tiến trình liên tục vào/ra miền găng của nó cho đến khi hoàn thành.

```

/* Shared data */
int processNumber = 0;

/* P0 */
while (true){
    while (processNumber == 1); /* entry section */

    /* critical section */

    processNumber = 1; /* exit section */

    /* remainder section */
}

/* P1 */
while (true){
    while (processNumber == 0); /* entry section */

    /* critical section code */

    processNumber = 0; /* exit section */

    /* Remainder section */
}

```

**Hình 5.3. Phiên bản 1.**

Theo phiên bản loại trừ lẫn nhau này, hệ thống sử dụng một biến được gọi là `processNumber`, mà cả hai tiến trình đều có quyền truy cập. Trước khi hệ thống bắt đầu thực thi các tiến trình, `processNumber` được đặt thành 0. Sau đó, hệ thống khởi động cả

hai tiến trình. Vòng lặp while kiểm tra nếu giá trị biến processNumber bằng số hiệu của tiến trình nào thì tiến trình đó được vào miền găng. Khi thoát khỏi miền găng, biến processNumber được gán số hiệu của tiến trình còn lại để cho phép tiến trình còn lại đó vào miền găng.

Mặc dù việc cài đặt này đảm bảo loại trừ lẫn nhau, nhưng nó có những hạn chế đáng kể. Trong lệnh vào miền găng (vòng lặp while kiểm tra biến processNumber), bộ xử lí không thực hiện lệnh có ích nào. Một tiến trình như vậy được cho là đang bận chờ đợi. Việc chờ bận có thể là một kỹ thuật không hiệu quả để thực hiện loại trừ lẫn nhau trên các hệ thống đơn xử lý. Hãy nhớ lại rằng một mục tiêu của đa chương trình là tăng khả năng sử dụng bộ xử lý.

Một hạn chế nữa nghiêm trọng hơn đó là, nếu P<sub>0</sub> không có nhu cầu vào miền găng mà P<sub>1</sub> cần vào miền găng, rõ ràng trong tình huống này P<sub>1</sub> không thể vào miền găng. Vì ban đầu biến processNumber=0 (tức là cho phép P<sub>0</sub> vào miền găng) và biến này chỉ được thay đổi giá trị là 1 khi tiến trình P<sub>0</sub> thực thi.

### 5.3.2. Phiên bản 2 (vi phạm loại trừ lẫn nhau)

Hình 5.4 thể hiện một giải pháp có găng loại bỏ sự đồng bộ hóa bước khóa của cài đặt trước. Trong giải pháp đầu tiên, hệ thống chỉ duy trì một biến toàn cục duy nhất - điều này buộc phải đồng bộ hóa bước khóa. Phiên bản 2 duy trì hai biến - p0\_Inside đúng nếu P<sub>0</sub> ở bên trong miền găng và p1\_Inside đúng nếu P<sub>1</sub> ở bên trong miền găng.

```
/* Shared data */
boolean p0_Inside = false;
boolean p1_Inside = false;

/* P0 */
while (true){
    /* entry section */
    while (p1_Inside) ;
    p0_Inside = true;

    /* critical section */

    p0_Inside = false; /* exit section */

    /* remainder section */
}

/* P1 */
```

```

while (true){
    /* entry section */
    while (p0_Inside) ;
    p1_Inside = true;

    /* critical section */

    p1_Inside = false; /* exit section */

    /* remainder section */
}

```

**Hình 5.4.** Phiên bản 2.

Bằng cách sử dụng hai biến để điều chỉnh quyền truy cập vào các miền găng (một cho mỗi tiến trình), chúng ta có thể loại bỏ đồng bộ hóa bước khóa trong cài đặt phiên bản hai. P<sub>0</sub> sẽ có thể liên tục đi vào miền găng của nó nhiều lần nếu cần trong khi p<sub>1</sub>\_Inside=false. Ngoài ra, nếu P<sub>0</sub> vào miền găng và đặt p<sub>0</sub>\_Inside=true, thì P<sub>1</sub> sẽ bận chờ đợi. Cuối cùng, P<sub>0</sub> sẽ hoàn thành miền găng của nó và thực hiện mã thoát miền găng, đặt p<sub>0</sub>\_Inside=false.

Mặc dù giải pháp này loại bỏ sự có đồng bộ hóa bước khóa, nhưng nó không đảm bảo loại trừ lẫn nhau. Hãy xem xét tình huống sau đây. Các biến p<sub>0</sub>\_Inside và p<sub>1</sub>\_Inside là false và cả hai tiến trình đều cố gắng vào miền găng của chúng cùng một lúc. P<sub>0</sub> kiểm tra giá trị của p<sub>1</sub>\_Inside, xác định giá trị là false và vào miền găng. Bây giờ, giả sử rằng hệ thống tạm dừng P<sub>0</sub> và thực thi P<sub>1</sub>. P<sub>1</sub> thực thi, xác định p<sub>0</sub>\_Inside là false và do đó P<sub>1</sub> vào miền găng của nó. Cả hai tiến trình đang thực hiện đồng thời miền găng của chúng, vi phạm loại trừ lẫn nhau.

### 5.3.3. Phiên bản 3 (tắt nghẽn)

Để khắc nhược điểm của phiên bản hai, khi một luồng đang kiểm tra while, nó phải được đảm bảo rằng tiến trình khác không thể tiếp tục quá trình kiểm tra while của chính nó. Phiên bản 3 (Hình 5.5) có găng giải quyết vấn đề này bằng cách yêu cầu mỗi luồng đặt cờ riêng trước khi vào vòng lặp while. Do đó, P<sub>0</sub> cho biết nó muốn vào miền găng bằng cách đặt p<sub>0</sub>\_WantsToEnter thành true. Nếu p<sub>1</sub>\_WantsToEnter là false, thì P<sub>0</sub> đi vào miền găng của nó và ngăn P<sub>1</sub> đi vào miền găng. Vì vậy, loại trừ lẫn nhau được đảm bảo, và có vẻ như là một giải pháp chính xác. Chúng ta đảm bảo rằng, khi một tiến trình báo hiệu ý định vào miền găng, thì một tiến trình khác không thể vào miền găng.

```
/* Shared data */
boolean p0_WantsToEnter = false;
boolean p1_WantsToEnter = false;

/* P0 */
while (true){

    /* entry section */
    P0_WantsToEnter = true;
    while (p1_WantsToEnter);

    /* critical section */

    p0_WantsToEnter = false; /* exit section */
    /* remainder section */
}

/* P1 */
while (true){
    /* entry section */
    p1_WantsToEnter = true;
    while (p0_WantsToEnter);

    /* critical section */

    p1_WantsToEnter = false; /* exit section */
    /* remainder section */
}
```

**Hình 5.5. Phiên bản 3.**

Ván đề phiên bản hai đã được giải quyết, nhưng một ván đề khác đã được đưa ra. Nếu mỗi tiến trình thiết lập cờ của nó trước khi tiếp tục kiểm tra while, thì mỗi tiến trình sẽ tìm thấy cờ của tiến trình kia và sẽ lặp lại mãi mãi trong thời gian đó. Đây là một ví dụ về tắc nghẽn hai tiến trình.

#### 5.3.4. Phiên bản 4 (hoãn vô thời hạn)

Để tạo ra một cài đặt loại trừ lẫn nhau hiệu quả, chúng ta cần một cách để "thoát ra" khỏi các vòng lặp vô hạn mà đã gặp trong phiên bản trước. Phiên bản 4 (Hình 5.6) hoàn thành điều này bằng cách buộc mỗi luồng lặp lại đặt cờ của nó thành false

trong các khoảng thời gian ngắn. Điều này cho phép luồng khác tiếp tục thông qua vòng lặp `while` của nó với cờ của chính nó được đặt thành `true`.

```

/* Shared data */
boolean p0_WantsToEnter = false;
boolean p1_WantsToEnter = false;

/* P0 */
while (true){
    /* entry section */
    p0_WantsToEnter = true;
    while (p1_WantsToEnter){
        p0_WantsToEnter = false;
        /* wait for small, random amount of time */
        p0_WantsToEnter = true;
    }

    /* critical section */

    p0_WantsToEnter = false; /* exit section */

    /* remainder section */
}

/* P1 */
while (true){
    /* entry section */
    p1_WantsToEnter = true;
    while (p0_WantsToEnter){
        p1_WantsToEnter = false;
        /* wait for small, random amount of time */
        p1_WantsToEnter = true;
    }

    /* critical section */

    p1_WantsToEnter = false; /* exit section */

    /* remainder section */
}

```

**Hình 5.6.** Phiên bản 4.

Phiên bản 4 đảm bảo loại trừ lẫn nhau và ngăn chặn sự tắc nghẽn, nhưng đưa ra tình huống khác, đó là trì hoãn vô thời hạn. Bởi vì chúng ta không thể biết trước về tốc độ tương đối của các tiến trình đồng thời, chúng ta phải xem xét tất cả các trình tự thực thi có thể có. Ví dụ, các tiến trình có thể tiến hành song song với nhau. Mỗi tiến trình có thể đặt cờ của nó thành `true`, sau đó thực hiện kiểm tra `while`, vào trong vòng lặp `while`, đặt cờ của nó thành `false`, đợi một khoảng thời gian ngẫu nhiên, đặt cờ của nó thành `true`, tiếp tục lặp lại trình tự bắt đầu với kiểm tra `while`. Khi các tiến trình thực hiện điều này, các điều kiện đã kiểm tra sẽ vẫn đúng. Tuy nhiên, một chuỗi thực thi như vậy sẽ xảy ra với xác suất thấp - nhưng tuy nhiên, nó có thể xảy ra. Nếu một hệ thống sử dụng kiểu loại trừ lẫn nhau này đang điều khiển một chuyến bay vũ trụ, máy điều hòa nhịp tim hoặc hệ thống kiểm soát không lưu, thì khả năng bị hoãn vô thời hạn và lỗi hệ thống sau đó có thể khiến tính mạng con người gặp rủi ro. Do đó, phiên bản bốn, cũng là một giải pháp không thể chấp nhận được cho vấn đề loại trừ lẫn nhau.

### 5.3.5. Thuật toán Dekker

Hình 5.7 minh họa Thuật toán Dekker - một giải pháp loại trừ lẫn nhau, hai tiến trình được cài đặt hoàn toàn bằng phần mềm mà không có lệnh phần cứng. Thuật toán Dekker vẫn sử dụng cờ để chỉ ra tiến trình muốn vào miền găng của nó, nhưng nó cũng kết hợp khái niệm "tiến trình được ưu tiên" sẽ được lựa chọn để vào miền găng trong trường hợp hòa (tức là khi mỗi tiến trình đồng thời muốn vào miền găng của nó).

```
/* shared data */
int favoredProcess = 0;
boolean p0_WantsToEnter = false;
boolean p1_WantsToEnter = false;

/* P0 */
while (true){
    /* entry section */
    p0_WantsToEnter = true;
    while (p1_WantsToEnter){
        if (favoredProcess == 1){
            p0_WantsToEnter = false;
            while (favoredProcess == 1); // busy wait
            p0_WantsToEnter = true;
        }
    }
}
```

```

/* critical section */

/* exit section */
favoredProcess = 1;
p0_WantsToEnter = false;

/* remainder section */
}

/* P1 */
while (true){
    /* entry section */
    p1_WantsToEnter = true;
    while (p0_WantsToEnter){
        if (favoredprocess == 0){
            p1_WantsToEnter = false;
            while (favoredThread == 0); // busy wait
            p1_WantsToEnter = true;
        }
    }

    /* critical section */

    /* exit section */
    favoredThread = 0;
    p1_WantsToEnter = false;

    /* remainder section */
}

```

### **Hình 5.7. Thuật toán Dekker.**

Thuật toán Dekker đảm bảo loại trừ lẫn nhau trong khi ngăn chặn tắc nghẽn và trì hoãn vô thời hạn.

#### 5.4. Thuật toán Peterson

Peterson là một giải pháp cổ điển dựa trên phần mềm cho bài toán miền găng. Do cách các kiến trúc máy tính hiện đại thực hiện các lệnh ngôn ngữ máy cơ bản, chẳng hạn như tải và lưu trữ, không có gì đảm bảo rằng giải pháp của Peterson sẽ hoạt động chính xác trên các kiến trúc đó. Tuy nhiên, chúng ta trình bày giải pháp vì nó cung cấp một mô tả thuật toán tốt về việc giải quyết bài toán miền găng và minh họa sự phức tạp liên

quan đến việc thiết kế phần mềm giải quyết các yêu cầu loại trừ lẫn nhau, tiến độ và chờ đợi có giới hạn.

Thuật toán của Peterson bị giới hạn ở hai tiến trình thực hiện luân phiên giữa các miền găng và các phần còn lại của chúng. Các tiến trình được đánh số P0 và P1.

Thuật toán Peterson yêu cầu hai tiến trình chia sẻ hai dữ liệu chung: **turn** và **flag**

```
int turn;
boolean flag[2];
//process P0
while (true){
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1)
        ;//do nothing
    /* critical section */
    flag[0] = false;
    /*remainder section */
}

//process P1
while (true){
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0)
        ;//do nothing
    /* critical section */
    flag[1] = false;
    /*remainder section */
}
```

### **Hình 5.8. Cấu trúc tiến trình P0 và P1.**

Trong đó biến **turn** cho biết ai sẽ đến lượt vào miền găng của mình. Tức là, nếu **turn=0**, thì tiến trình P0 được phép thực thi trong miền găng của nó, tương tự P1. Mảng **flag** được sử dụng để cho biết một tiến trình đã sẵn sàng để đi vào miền găng của nó hay chưa. Ví dụ: nếu **flag[0]** là **true**, P0 đã sẵn sàng tham gia miền găng của nó.

Để vào miền găng, đầu tiên tiến trình P0 đặt **flag[0]=true** và sau đó đặt **turn=1**. Nếu cả hai tiến trình cố gắng vào cùng một lúc, **turn** sẽ được gán cả 0 và 1 gần như cùng một lúc. Tuy nhiên **turn** chỉ nhận 1 giá trị; giá trị khác sẽ được gán nhưng sẽ bị ghi đè ngay lập tức. Giá trị cuối cùng của **turn** xác định tiến trình nào trong hai tiến

trình được phép vào miền găng. Để chứng minh giải pháp này hợp lệ, ta chứng minh 3 điều sau:

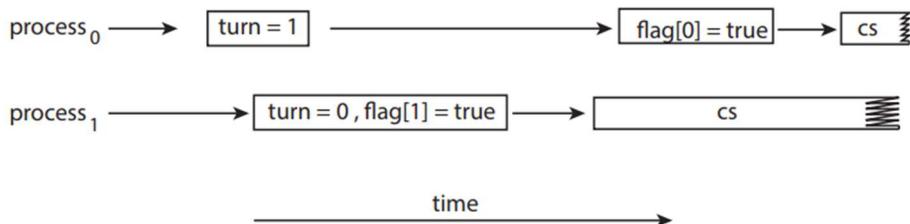
1. Loại trừ lẫn nhau được bảo toàn.
2. Yêu cầu về tiến độ được thỏa mãn.
3. Yêu cầu chờ đợi có giới hạn được đáp ứng.

Để chứng minh tính chất 1, chúng ta lưu ý rằng: (1) P0 chỉ đi vào miền găng khi `flag[1]=false` hoặc `turn=0`, tương tự P1 vào miền găng khi `flag[0]=false` hoặc `turn=1`. (2) Nếu cả hai tiến trình có thể thực thi đồng thời trong miền găng của nó, thì `flag[0]=flag[1]=true`. Từ (1) và (2) ta thấy rằng P0 và P1 không thể thực hiện thành công các câu lệnh `while` của chúng cùng một lúc, vì giá trị của lần lượt có thể là 0 hoặc 1 nhưng không thể là cả hai. Do đó, một trong các tiến trình - ví dụ P1 - phải thực hiện thành công câu lệnh `while`, trong khi P0 phải thực thi ít nhất một câu lệnh `turn=1`. Tuy nhiên, tại thời điểm đó, `flag[1]=true` và `turn=1`, và điều kiện này sẽ tồn tại miễn là P1 nằm trong miền găng của nó; kết quả là, sự loại trừ lẫn nhau được bảo toàn.

Để chứng minh các tính chất 2 và 3, chúng ta lưu ý rằng một tiến trình P0 có thể bị chặn vào miền găng chỉ khi nó bị kẹt trong vòng lặp `while` với điều kiện `flag[1]=true` và `turn=1`; vòng lặp này chỉ đúng cho 1 tiến trình. Nếu P1 chưa sẵn sàng vào miền găng, thì `flag[1]=false` và P0 có thể vào miền găng. Nếu P1 đã đặt `flag[1]=true` và cũng đang thực thi trong câu lệnh `while` của nó, thì `turn=0` hoặc `turn=1`. Nếu `turn=0`, thì P0 sẽ đi vào miền găng. Nếu `turn=1`, thì P1 sẽ vào miền găng. Tuy nhiên, khi P1 thoát khỏi miền găng, nó sẽ đặt lại `flag[1]=false`, cho phép P0 đi vào miền găng. Nếu P1 đặt lại `flag[1]=true`, nó cũng phải đặt `turn=0`. Thực vậy, vì P0 không thay đổi giá trị của biến `turn` trong khi thực hiện câu lệnh `while`, P0 sẽ vào miền găng (đang trong tiến độ) sau nhiều nhất một lần P1 đang chờ.

Giải pháp của Peterson không đảm bảo sẽ hoạt động hợp lệ trên các kiến trúc máy tính hiện đại. Vì lý do chính là để cải thiện hiệu suất hệ thống, bộ xử lý và/hoặc trình biên dịch có thể sắp xếp lại các thao tác đọc và ghi những lệnh không có phụ thuộc nhau. Đối với một ứng dụng đơn luồng, việc sắp xếp lại thứ tự này là không quan trọng khi có liên quan đến tính đúng đắn của chương trình, vì các giá trị cuối cùng phù hợp với những gì được mong đợi. (Điều này tương tự như việc cân bằng sổ kê toán - thứ tự thực tế trong tính toán ghi có và ghi nợ được thực hiện là không quan trọng, bởi vì số dư cuối cùng sẽ vẫn giống nhau.) Nhưng đối với một ứng dụng đa luồng với dữ liệu được chia sẻ, việc sắp xếp lại các lệnh có thể dẫn đến kết quả không nhất quán hoặc bất ngờ.

Điều này ảnh hưởng đến giải pháp của Peterson như thế nào? Hãy xét điều gì sẽ xảy ra nếu các phép gán trong Hình 5.8 được sắp xếp lại, tức là lệnh `turn=1` thực hiện trước lệnh `flag[0]=true`. Có thể cả hai luồng vào miền găng cùng lúc như mô tả trong Hình 5.9.



**Hình 5.9.** Tính không hợp lệ của giải pháp Peterson khi thay đổi trật tự.

## 5.5. Thuật toán Lamport

Nhiều giải pháp ban đầu cho vấn đề loại trừ lẫn nhau của  $n$  tiến trình rất khó hiểu vì chúng yêu cầu một số lượng lớn các biến dùng chung và các vòng lặp phức tạp để xác định xem một luồng có thể đi vào miền găng của nó hay không. Thuật toán của Lamport cung cấp một giải pháp đơn giản hơn mượn từ một kịch bản phổ biến trong thế giới thực - chờ đợi phục vụ tại một tiệm bánh. Hơn nữa, thuật toán của Lamport không yêu cầu bất kỳ hoạt động nào phải xảy ra nguyên tử.

Thuật toán của Lamport được mô phỏng trên một tiệm bánh trong đó một nhân viên phục vụ các yêu cầu của khách hàng về các món nướng; nhân viên này có thể phục vụ chính xác một khách hàng tại một thời điểm. Nếu chỉ có một khách hàng, giao dịch rất đơn giản: khách hàng yêu cầu đồ nướng, nhân viên lấy đồ, khách hàng thanh toán đồ ăn và ra khỏi tiệm bánh. Tuy nhiên, khi có nhiều khách hàng đồng thời yêu cầu dịch vụ thì nhân viên phải xác định phục vụ khách hàng theo thứ tự nào. Nhiều tiệm bánh phục vụ khách hàng theo thứ tự ai đến trước được phục vụ trước bằng cách yêu cầu họ lấy một phiếu đánh số từ máy phát vé khi họ bước vào tiệm bánh. Các vé được phân phát theo thứ tự tăng dần (tức là nếu vé hiện tại chứa giá trị  $n$  thì vé tiếp theo chứa giá trị  $n+1$ ). Sau mỗi giao dịch, nhân viên phục vụ khách hàng sở hữu vé có giá trị thấp nhất, điều này đảm bảo rằng khách hàng được phục vụ theo thứ tự ai đến trước được phục vụ trước. Hình 5.10 trình bày cài đặt thuật toán của Lamport cho  $n$  tiến trình. Trong thuật toán của Lamport, mỗi tiến trình cho một khách hàng phải "lấy vé" để xác định thời điểm tiến trình có thể vào miền găng của nó. Khi một tiến trình sở hữu vé có giá trị thấp nhất, nó có thể vào miền găng của nó. Loại trừ lẫn nhau được thực thi bằng cách đặt lại giá trị vé của tiến trình khi nó thoát khỏi miền găng của nó. Không giống như một bộ phân phối vé trong thế giới thực, thuật toán của Lamport cho phép nhiều tiến trình có cùng một giá trị vé.

```

/* shared data */
/* array that records which process are taking a ticket */
boolean choosing[n];

/* value of the ticket for each process initialized to 0 */
int ticket[n];

/* process Px */
while (true){

    /* take a ticket */
    choosing[x] = true; /* begin ticket selection process */
    ticket[x] = maxValue(ticket) + 1;
    choosing[x] = false; /* end ticket selection process */

    /* wait for number to be called by comparing current ticket
    value to other process's ticket value */
    for (int i = 0 ; i < n ; i++){
        if (i == x){ continue; }

        /* busy wait while process i is choosing */
        while (choosing[i] != false);

        /* busy wait until current ticket value is lowest */
        while (ticket[i] != 0 && ticket[i] < ticket[x]);

        /* tie-breaker code favors smaller thread number */
        if (ticket[i] == ticket[x] && i < x)
            /* loop until
            process i leaves it's critical section */
            while (ticket[i] != 0); /* busy wait */
    }

    /* critical section */

    ticket[x] = 0; /* exit section */

    /* remainder section */
}

```

**Hình 5.10.** Thuật toán Lamport.

Tiến trình Px gọi phương thức `maxValue()`, phương thức này trả về giá trị lớn nhất trong mảng số nguyên `ticket`. Sau đó, tiến trình sẽ thêm một vào giá trị đó và lưu trữ nó dưới dạng giá trị vé, `ticket[x]`. Sau khi tiến trình đã gán giá trị vé của nó cho `ticket[x]`, tiến trình đặt `choosing[x]` thành `false`, cho biết rằng nó không còn chọn giá trị vé nữa. Lưu ý rằng khi một tiến trình thoát khỏi miền găng của nó, giá trị vé được đặt thành 0, có nghĩa là giá trị vé của một tiến trình chỉ khác không nếu nó muốn vào miền găng của nó. Trước khi vào miền găng, tiến trình Px phải thực hiện vòng lặp `for` xác định trạng thái của tất cả các tiến trình trong hệ thống. Nếu tiến trình Pi khác Px ( $i \neq x$ ), tiến trình Px xác định xem Pi có đang chọn giá trị vé hay không. Nếu Px không đợi cho đến khi Pi chọn xong vé trước khi vào miền găng, loại trừ lẫn nhau có thể bị vi phạm.

Vòng lặp `while` thứ 3 buộc tiến trình Px phải đợi tiến trình Pi nếu `ticket[x] < ticket[i]`. Điều kiện này tương tự như tiệm bánh trong thế giới thực - mỗi tiến trình phải đợi cho đến khi nó có giá trị vé không thấp nhất. Tuy nhiên, không giống như một tiệm bánh trong thế giới thực, hai hoặc nhiều tiến trình trong hệ thống có thể nhận được cùng một giá trị vé. Do đó, trong trường hợp hòa, tiến trình có chỉ số thấp sẽ tiến hành trước (vòng lặp `while` thứ 4).

Chúng ta hãy quay trở lại vòng lặp `while` thứ 2 để xét việc loại trừ lẫn nhau bị vi phạm như thế nào nếu tiến trình Px không đợi khi tiến trình Pi đang chọn giá trị vé. Ví dụ: hãy xem xét tình huống nếu tiến trình Px tạm dừng sau khi trả về từ phương thức `maxValue()` nhưng trước khi thêm một vào giá trị vé. Giả sử cho ví dụ này rằng `maxValue()` trả về giá trị 215. Sau khi Px tạm dừng, một số luồng khác chẳng hạn Py, thực hiện và hoàn thành chọn vé, khi đó tiến trình Py với giá trị vé 216 (lưu ý rằng giá trị vé của Px hiện là 0). Py có thể vào miền găng và thoát khỏi phần miền găng trước khi Px được thực thi lại. Giả sử Py đang ở trong miền găng và bị tạm dừng, Px tiếp tục thực hiện. Ta thấy Px nhận giá trị vé là 216 và giả sử chỉ số x nhỏ hơn chỉ số y, khi đó `ticket[x]=ticket[y]` và  $x < y$  nên Px vào miền găng. Lúc này cả Px và Py vào miền găng, vi phạm cơ chế loại trừ lẫn nhau.

Ngoài việc là một trong những thuật toán loại trừ lẫn nhau n-luồng đơn giản nhất. Thuật toán làm bánh của Lamport thể hiện một số đặc tính thú vị. Ví dụ, nó không yêu cầu các lệnh của nó phải được thực thi nguyên tử. Điều này là cần thiết vì cả thuật toán của Dekker và Peterson đều yêu cầu nhiều tiến trình sửa đổi một biến được chia sẻ để kiểm soát quyền truy cập vào miền găng. Nếu mỗi tiến trình có thể đọc và sửa đổi biến này đồng thời trên các bộ xử lý khác nhau, các tiến trình có thể đọc các giá trị không nhất quán của các biến được chia sẻ của chúng. Điều này có thể cho phép cả hai tiến

trình vào miền găng của chúng đồng thời, vi phạm loại trừ lẫn nhau.

Thuật toán Lamport cung cấp một giải pháp thanh lịch để loại trừ lẫn nhau trên các hệ thống đa xử lý, bởi vì mỗi tiến trình được gán một bộ biến riêng để kiểm soát quyền truy cập vào miền găng của nó. Mặc dù tất cả các tiến trình trong hệ thống chia sẻ các mảng `choosing` và `ticket`, nhưng tiến trình Px là tiến trình duy nhất có thể sửa đổi các giá trị trong `choosing[x]` và `ticket[x]`. Điều này ngăn các tiến trình đọc dữ liệu không nhất quán, bởi vì các biến mà một tiến trình kiểm tra trong khi thực hiện mã loại trừ lẫn nhau của nó không thể được sửa đổi đồng thời bởi một tiến trình khác.

Một tính chất thú vị khác của thuật toán Lamport là các tiến trình đang chờ để vào miền găng của chúng sẽ được chấp nhận theo thứ tự ai đến trước được phục vụ trước (FCFS) trừ khi nhiều tiến trình có cùng giá trị vé. Cuối cùng, thuật toán của Lamport có thể tiếp tục thực thi loại trừ lẫn nhau ngay cả khi một hoặc nhiều tiến trình không thành công, miễn là hệ thống đặt giá trị của mỗi tiến trình bị lỗi trong mảng `choosing` thành `false` và giá trị của mỗi tiến trình trong mảng `ticket` là 0. Đưa ra điều khoản cuối cùng này, Thuật toán Bakery của Lamport không thể bị tắc nghẽn hoặc trì hoãn vô thời hạn, một thuộc tính đặc biệt quan trọng trong các hệ thống đa xử lý và phân tán, trong đó sự cố của một thiết bị phần cứng như bộ xử lý không nhất thiết dẫn đến lỗi hệ thống.

## 5.6. Sử dụng phần cứng để đồng bộ hóa

Chúng ta vừa mô tả một giải pháp dựa trên phần mềm cho bài toán miền găng. (Chúng ta gọi là một giải pháp dựa trên phần mềm vì thuật toán không liên quan đến sự hỗ trợ từ hệ điều hành hoặc lệnh phần cứng để đảm bảo loại trừ lẫn nhau.) Tuy nhiên, các giải pháp dựa trên phần mềm không được đảm bảo hoạt động trên các kiến trúc máy tính hiện đại. Trong phần này, chúng ta giới thiệu ba lệnh phần cứng cung cấp để giải quyết bài toán miền găng. Các thao tác cơ bản này có thể sử dụng trực tiếp làm công cụ đồng bộ hóa, hoặc có thể sử dụng để tạo nền tảng cho các cơ chế đồng bộ hóa khác.

### 5.6.1. Rào cản bộ nhớ

Chúng ta thấy hệ thống có thể sắp xếp lại các lệnh, và từ đó có thể dẫn đến trạng thái dữ liệu không đáng tin cậy. Mô hình bộ nhớ là cách trúc máy tính xác định bộ nhớ nào đảm bảo trật tự các lệnh sẽ cung cấp cho chương trình ứng dụng. Nói chung, một mô hình bộ nhớ thuộc một trong hai loại:

1. Có trật tự mạnh, trong đó tất cả các bộ xử lý khác có thể nhìn thấy ngay lập tức việc sửa đổi bộ nhớ trên một bộ xử lý.
2. Có thứ tự yếu, trong đó các sửa đổi đối với bộ nhớ trên một bộ xử lý có thể

không hiển thị ngay lập tức đối với các bộ xử lý khác.

Để giải quyết vấn đề này, kiến trúc máy tính cung cấp các lệnh có thể buộc bất kỳ thay đổi nào trong bộ nhớ được truyền cho tất cả các bộ xử lý khác, do đó đảm bảo rằng các sửa đổi bộ nhớ có thể hiển thị đối với các luồng chạy trên các bộ xử lý khác. Những lệnh như vậy được gọi là hàng rào bộ nhớ.

Đối với giải pháp của Peterson, chúng ta có thể đặt lệnh `memory_barrier()` giữa hai câu lệnh gán đầu tiên trong phần vào miền găng để tránh việc thay đổi trật tự các thao tác như trong Hình 5.9. Lưu ý rằng `memory_barrier()` được coi là các hoạt động cấp rất thấp và thường chỉ được các nhà phát triển nhân sử dụng khi viết mã chuyên biệt để đảm bảo loại trừ lẫn nhau.

### 5.6.2. Các lệnh phân cứng

Nhiều hệ thống máy tính hiện đại cung cấp các lệnh phân cứng đặc biệt cho phép chúng ta kiểm tra và sửa đổi nội dung của một từ hoặc hoán đổi nội dung của hai từ một cách nguyên tử (atomically) - nghĩa là, như một đơn vị liên tục. Chúng ta có thể sử dụng các lệnh đặc biệt này để giải quyết bài toán miền găng một cách tương đối đơn giản. Thay vì thảo luận về một lệnh cụ thể cho một máy cụ thể, chúng ta tóm tắt các khái niệm chính các loại lệnh này bằng cách mô tả các lệnh `test_and_set()` và `compare_and_swap()`.

Lệnh `test_and_set()` có thể được định nghĩa như trong Hình 5.11. Đặc điểm quan trọng của lệnh này là nó được thực thi nguyên tử (atomically). Do đó, nếu hai lệnh `test_and_set()` được thực thi đồng thời (mỗi lệnh trên một lõi khác nhau), chúng sẽ được thực thi tuần tự theo một trật tự tùy ý. Nếu máy tính hỗ trợ lệnh `test_and_set()`, thì chúng ta có thể thực hiện loại trừ lẫn nhau bằng cách báo một biến logic `lock`, được khởi tạo thành `false`. Cấu trúc của tiến trình `Pi` ( $i=0, 1$ ) được thể hiện trong Hình 5.12.

```
boolean test_and_set(boolean *target){  
    boolean rv = *target;  
    *target = true;  
    return rv;  
}
```

**Hình 5.11. Định nghĩa lệnh `test_and_set()`.**

```
do {  
    while (test_and_set(&lock))
```

```

; /* do nothing */

/* critical section */

lock = false;

/* remainder section */
} while (true);

```

**Hình 5.12.** Cài đặt cơ chế loại trừ lẫn nhau bằng hàm `test_and_set()`.

Lệnh `compare_and_swap()` (CAS), giống như lệnh `test_and_set()`, hoạt động trên hai biến nguyên tử, nhưng sử dụng một cơ chế khác dựa trên việc hoán đổi nội dung của hai biến.

Lệnh CAS hoạt động trên ba toán hạng và được định nghĩa trong Hình 5.13. Toán hạng `value` chỉ được đặt thành `new_value` nếu biểu thức (`*value==expected`) là đúng. Bất kể, CAS luôn trả về giá trị ban đầu của biến `value`. Đặc điểm quan trọng của lệnh này là nó được thực thi nguyên tử. Do đó, nếu hai lệnh CAS được thực thi đồng thời (mỗi lệnh trên một lõi khác nhau), chúng sẽ được thực hiện tuần tự theo một số thứ tự tùy ý.

```

int compare_and_swap(int *value, int expected, int new_value){
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}

```

**Hình 5.13.** Định nghĩa lệnh `compare_and_swap()`.

Loại trừ lẫn nhau sử dụng CAS có thể được thực hiện như sau: Một biến toàn cục (`lock`) được khởi tạo thành 0. Tiến trình đầu tiên gọi `compare_and_swap()` sẽ đặt `lock` thành 1. Sau đó, nó sẽ vào miền găng, bởi vì giá trị ban đầu của `lock` bằng giá trị `expected` là 0. Các lệnh gọi `compare_and_swap()` tiếp theo sẽ không thành công, bởi vì `lock` bây giờ không bằng giá trị `expected` là 0. Khi một tiến trình thoát khỏi miền găng, nó đặt `lock` về 0, điều này cho phép một tiến trình khác vào miền găng. Cấu trúc của tiến trình `Pi` ( $i=0, 1$ ) được thể hiện trong Hình 5.14.

```

while (true){
    while (compare_and_swap(&lock, 0, 1) != 0)

```

```

; /* do nothing */

/* critical section */

lock = 0;

/* remainder section */
}

```

**Hình 5.14.** Cài đặt loại trừ lẫn nhau bằng lệnh compare\_and\_swap().

### 5.6.3. Biến nguyên thủy

Thông thường, lệnh compare\_and\_swap() không được sử dụng trực tiếp để loại trừ lẫn nhau. Thay vào đó, nó được sử dụng như một khái niệm cơ bản để xây dựng các công cụ khác giải quyết bài toán miền găng. Một công cụ như vậy là một biến nguyên thủy, cung cấp các phép toán nguyên thủy trên các kiểu dữ liệu cơ bản như số nguyên và boolean. Chúng ta biết từ Phần 5.1 rằng việc tăng hoặc giảm một giá trị số nguyên có thể tạo ra một điều kiện tương tranh. Các biến nguyên thủy có thể được sử dụng để đảm bảo loại trừ lẫn nhau trong các tình huống có thể có tương tranh dữ liệu trên một biến duy nhất trong khi nó đang được cập nhật, khi bộ đếm được tăng lên.

## 5.7. Khóa Mutex

Các giải pháp dựa trên phần cứng cho bài toán miền găng rất phức tạp và thường không thể tiếp cận được đối với các lập trình viên ứng dụng. Thay vào đó, các nhà thiết kế hệ điều hành xây dựng các công cụ phần mềm cấp cao hơn để giải quyết bài toán miền găng. Công cụ đơn giản nhất trong số những công cụ này là khóa mutex. (thuật ngữ mutex là viết tắt của mutual exclusion – loại trừ lẫn nhau). Chúng ta sử dụng khóa mutex để bảo vệ các miền găng và do đó ngăn chặn các điều kiện tương tranh. Đó là, một tiến trình phải có được khóa trước khi bước vào miền găng; nó thả khóa khi thoát khỏi miền găng. Hàm acquire() nhận được khóa và hàm release() giải phóng khóa, như minh họa trong Hình 5.15.

```

while (true){
    acquire lock
        critical section
    release lock
        remainder section
}

```

**Hình 5.15.** Giải pháp cho bài toán miền găng dùng khóa mutex.

Hàm `acquire()` định nghĩa như sau:

```
acquire(){
    while (!available)
        ; /* busy wait */
    available = false;
}
```

Định nghĩa hàm `release()` như sau:

```
release(){
    available = true;
}
```

Các lệnh gọi đến `acquire()` hoặc `release()` phải được thực hiện nguyên tử. Do đó, khóa `mutex` có thể được thực hiện bằng cách sử dụng thao tác CAS được mô tả trong mục 5.6.

Nhược điểm chính của việc cài đặt này là nó đòi hỏi phải chờ đợi khi bận. Trong khi một tiến trình nằm trong miền găng, bất kỳ tiến trình nào khác có găng đi vào miền găng của nó phải lặp lại liên tục lệnh gọi tới `acquire()`. Vòng lặp liên tục này rõ ràng là một vấn đề trong một hệ thống đa chương trình, nơi một lõi CPU duy nhất được chia sẻ giữa nhiều tiến trình. Chờ đợi khi bận làm lãng phí thời gian CPU mà một số tiến trình khác có thể sử dụng.

Kiểu khóa `mutex` mà chúng ta đã mô tả còn được gọi là khóa quay (spinlock) vì tiến trình "quay" trong khi chờ khóa khả dụng. (Chúng ta thấy vấn đề tương tự với các ví dụ minh họa lệnh `compare_and_swap()`.) Tuy nhiên, Spinlock có một lợi thế trong đó không yêu cầu chuyển đổi ngữ cảnh khi một tiến trình phải chờ khóa và chuyển đổi ngữ cảnh có thể mất nhiều thời gian. Trong một số trường hợp nhất định trên các hệ thống đa lõi, spinlock trên thực tế là lựa chọn thích hợp hơn để khóa. Nếu khóa được giữ trong thời gian ngắn, một luồng có thể "quay" trên một lõi xử lý trong khi một luồng khác thực hiện miền găng của nó trên lõi khác. Trên các hệ thống đa lõi hiện đại, spinlock được sử dụng rộng rãi trong nhiều hệ điều hành.

## 5.8. Semaphore

Khóa `mutex`, như chúng ta đã đề cập trước đó, thường được coi là công cụ đồng bộ hóa đơn giản nhất. Trong phần này, chúng ta xem xét một công cụ mạnh mẽ hơn có thể hoạt động tương tự như khóa `mutex` nhưng cũng có thể cung cấp các cách phức tạp hơn để các tiến trình đồng bộ hóa các hoạt động của chúng.

`Semaphore S` là một biến số nguyên, ngoài việc khởi tạo, nó chỉ được truy cập

through two primitive operations: `wait()` and `signal()`. The definition of `wait()` is as follows:

```
wait(S){  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

The definition of `signal()` is as follows:

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

All changes to the semaphore value must be atomic. This means that when one thread changes the semaphore value, no other thread can change it at the same time. In addition, when checking if `S` is less than or equal to zero in `wait(S)`, the check must be performed atomically, so that if another thread increments the value before the check, the check will fail.

### 5.8.1. Cách sử dụng Semaphore

The operating system provides two types of semaphores: binary and general. The value of a general semaphore can exceed the number of available resources. The value of a binary semaphore is either 0 or 1. Therefore, a binary semaphore acts like a mutex lock. In practice, on systems without mutex locks, semaphores are used to implement mutual exclusion between processes.

A general semaphore can be used to control access to shared resources. A semaphore is created with a specific initial value. Any process that wants to use a resource must call `wait()` on the semaphore (which decreases its value). When a process releases a resource, it calls `signal()` on the semaphore (which increases its value). When the value of the semaphore reaches zero, all processes waiting on it will be blocked until the value increases again.

We can also use semaphores to solve synchronization problems between processes. For example, consider two processes P1 and P2. P1 has a command S1 and P2 has a command S2. We want P1 to execute S1 before P2 executes S2. We can achieve this by creating a shared semaphore with an initial value of 0. Both P1 and P2 increment the semaphore before executing their respective commands. When P1 increments the semaphore, it becomes 1, allowing P2 to execute S2. When P2 increments the semaphore, it becomes 2, allowing P1 to execute S1. This ensures that S1 is executed before S2.

chúng ta chèn các câu lệnh:

```
S1;
signal(synch);
```

Trong tiến trình P2, chúng ta chèn các câu lệnh:

```
wait(synch);
S2;
```

Vì biến `synch` được khởi tạo bằng 0, P2 sẽ chỉ thực thi S2 sau khi P1 đã gọi `signal(synch)`, tức là sau khi câu lệnh S1 đã được thực thi.

### 5.8.2. Cài đặt semaphore

Các định nghĩa thao tác `semaphore` `wait()` và `signal()` vừa được mô tả gặp trường hợp chờ khi bận. Để khắc phục vấn đề này, chúng ta có thể sửa đổi `wait()` và `signal()` như sau: Khi một tiến trình thực hiện thao tác `wait()` và nhận thấy rằng giá trị `semaphore` không dương, nó phải đợi. Tuy nhiên, thay vì tham gia vào việc chờ đợi bận rộn, tiến trình có thể tự tạm ngưng. Thao tác tạm ngưng đặt một tiến trình vào hàng đợi liên kết với `semaphore`, và trạng thái của tiến trình được chuyển sang trạng thái waiting. Sau đó, quyền điều khiển được chuyển đến bộ lập lịch CPU, bộ này sẽ chọn một tiến trình khác để thực thi.

Tiến trình bị tạm ngưng, đang chờ trên `semaphore` S, sẽ được khởi động lại khi một số tiến trình khác thực hiện thao tác `signal()`. Tiến trình được khởi động lại bằng thao tác `wakeup()`, thao tác này sẽ thay đổi tiến trình từ trạng thái waiting sang trạng thái ready. Tiến trình sau đó được đặt trong hàng đợi sẵn sàng.

Để cài đặt `semaphore` theo định nghĩa này, chúng ta định nghĩa một cấu trúc `semaphore` như sau:

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

Mỗi `semaphore` có một số nguyên `value` và một danh sách các tiến trình `list`. Khi một tiến trình phải đợi trên `semaphore`, nó sẽ được thêm vào danh sách các tiến trình. Thao tác `signal()` xóa một tiến trình khỏi danh sách các tiến trình đang chờ và đánh thức tiến trình đó. Thao tác `wait()` cài đặt như sau:

```
wait(semaphore *S){
    S->value--;
    if (S->value < 0) {
```

```
    add this process to S->list;
    sleep();
}
}
```

Thao tác signal() cài đặt như sau:

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

Thao tác sleep() tạm dừng tiến trình gọi nó. Thao tác wakeup(P) tiếp tục thực hiện tiến trình bị tạm ngưng P. Hai thao tác này được hệ điều hành cung cấp dưới dạng các lệnh gọi hệ thống cơ bản.

## 5.9. Tóm tắt

- Điều kiện tương tranh xảy ra khi các tiến trình có quyền truy cập đồng thời vào dữ liệu được chia sẻ và kết quả cuối cùng phụ thuộc vào thứ tự cụ thể mà các truy cập xảy ra đồng thời. Điều kiện tương tranh có thể dẫn đến các giá trị dữ liệu được chia sẻ bị hỏng.

- Miền găng là phần mã nơi dữ liệu được chia sẻ có thể bị thao tác và có thể xảy ra tình trạng tương tranh. Bài toán miền găng là thiết kế một giao thức theo đó các tiến trình có thể đồng bộ hóa hoạt động của chúng để chia sẻ dữ liệu một cách hợp tác.

- Một giải pháp cho bài toán miền găng phải thỏa mãn ba yêu cầu sau: (1) loại trừ lẫn nhau, (2) tiến độ và (3) chờ đợi có giới hạn. Loại trừ lẫn nhau đảm bảo rằng chỉ có một tiến trình tại một thời điểm đang hoạt động trong miền găng của nó. Tiến độ đảm bảo rằng các chương trình sẽ hợp tác xác định tiến trình tiếp theo sẽ đi vào miền găng của nó. Chờ có giới hạn là hạn chế thời gian một chương trình sẽ đợi trước khi nó có thể vào miền găng của nó.

- Các giải pháp phần mềm cho bài toán miền găng, chẳng hạn như giải pháp của Dekker, Peterson, không hoạt động tốt trên các kiến trúc máy tính hiện đại.

- Hỗ trợ phần cứng cho bài toán miền găng bao gồm các rào cản bộ nhớ; lệnh phân cứng, chẳng hạn như lệnh compare-and-swap; và các biến nguyên tử.

- Khóa mutex cung cấp khả năng loại trừ lẫn nhau bằng cách yêu cầu một tiến trình

phải có khóa trước khi vào miền găng và nhả khóa khi thoát khỏi miền găng.

• **Semaphore**, như khóa **mutex**, có thể được sử dụng để loại trừ lẫn nhau. Tuy nhiên khóa **mutex** có giá trị nhị phân cho biết khóa có sẵn hay không, thì **semaphore** có giá trị nguyên và do đó có thể được sử dụng để giải quyết nhiều vấn đề đồng bộ hóa.

### 5.10. Câu hỏi ôn tập

**Câu 5.1.** Nghĩa của thuật ngữ đợi khi bạn là gì? Có những loại chờ đợi nào khác trong một hệ điều hành? Có thể tránh được hoàn toàn việc chờ đợi bận rộn không? Giải thích câu trả lời của bạn.

**Câu 5.2.** Giải thích tại sao spinlock không thích hợp cho các hệ thống đơn xử lý nhưng lại thường được sử dụng trong các hệ thống đa xử lý.

**Câu 5.3.** Chỉ ra rằng, nếu các thao tác của semaphore như `wait()` và `signal()` không được thực thi nguyên tử, thì việc loại trừ lẫn nhau có thể bị vi phạm.

**Câu 5.4.** Minh họa cách sử dụng một semaphore nhị phân để thực hiện loại trừ lẫn nhau giữa n tiến trình.

**Câu 5.5.** Điều kiện tương tranh có thể xảy ra trong nhiều hệ thống máy tính. Hãy xét một hệ thống ngân hàng duy trì số dư tài khoản với hai chức năng: gửi - `deposit(số tiền)` và rút - `withdraw(số tiền)`. Hai chức năng này là gửi và rút số tiền sẽ trong ngân hàng. Giả sử rằng vợ và chồng dùng chung một tài khoản ngân hàng. Đồng thời, người chồng gọi hàm `withdraw()` và người vợ gọi `deposit()`. Mô tả điều kiện tương tranh có thể xảy ra và cách có thể thực hiện để ngăn chặn tình trạng tương tranh.



## CHƯƠNG 6. TẮT NGHẼN

### 6.1. Mô hình

Một hệ thống bao gồm một số lượng tài nguyên hữu hạn được phân phối cho một số luồng cạnh tranh. Các tài nguyên có thể được phân chia thành một số loại, mỗi loại bao gồm một số cá thể (instance) giống hệt nhau. Chẳng hạn như CPU, tệp và thiết bị I/O. Nếu một hệ thống có bốn CPU, thì loại tài nguyên CPU có bốn cá thể. Nếu một luồng yêu cầu một cá thể của kiểu tài nguyên, thì hệ thống cấp phát một cá thể của kiểu tài nguyên đó.

Các công cụ đồng bộ hóa khác nhau chẳng hạn như khóa mutex và semaphores, cũng là tài nguyên hệ thống; và trên các hệ thống máy tính hiện đại, chúng là những nguyên nhân phổ biến nhất gây ra tắc nghẽn. Một khóa thường được liên kết với một cấu trúc dữ liệu cụ thể - nghĩa là, một khóa có thể được sử dụng để bảo vệ quyền truy cập vào hàng đợi, khóa khác để bảo vệ quyền truy cập vào danh sách được liên kết, v.v. Vì lý do đó, mỗi phiên bản của một khóa thường được gán cho lớp tài nguyên riêng của nó.

Một luồng phải yêu cầu một tài nguyên trước khi sử dụng nó và phải giải phóng tài nguyên sau khi sử dụng nó. Một luồng có thể yêu cầu nhiều tài nguyên để thực hiện nhiệm vụ của nó. Rõ ràng, số lượng tài nguyên được yêu cầu không được vượt quá tổng số tài nguyên hiện có trong hệ thống.

Trong chế độ hoạt động bình thường, một luồng chỉ có thể sử dụng tài nguyên theo trình tự sau:

1. Yêu cầu. Luồng yêu cầu tài nguyên. Nếu yêu cầu không thể được cấp ngay lập tức (ví dụ: nếu một khóa mutex hiện đang được giữ bởi một luồng khác), thì luồng yêu cầu phải đợi cho đến khi nó có thể nhận được tài nguyên.
2. Sử dụng. Luồng có thể hoạt động trên tài nguyên (ví dụ: nếu tài nguyên là khóa mutex, luồng có thể truy cập vào miền găng).
3. Giải phóng. Luồng giải phóng tài nguyên.

Khi một luồng sử dụng tài nguyên do nhân quản lý, hệ điều hành sẽ kiểm tra để

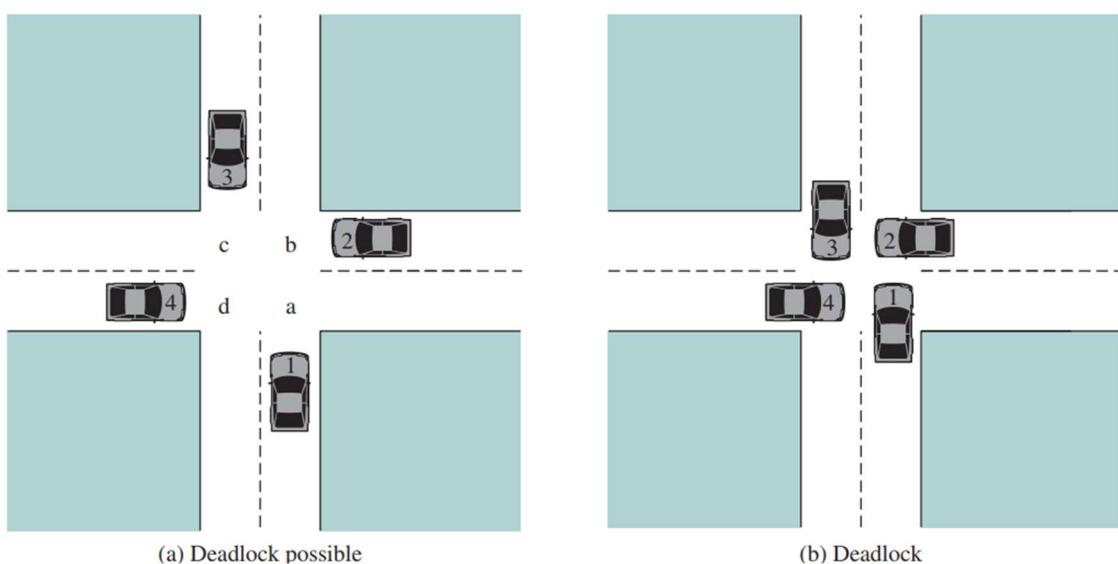
đảm bảo rằng luồng đã yêu cầu và đã được cấp phát tài nguyên. Một bảng hệ thống ghi lại xem mỗi tài nguyên là tự do hay đã cấp phát. Đôi với mỗi tài nguyên được cấp phát, bảng cũng ghi lại luồng mà nó được cấp phát. Nếu một luồng yêu cầu một tài nguyên hiện được cấp phát cho một luồng khác, nó có thể được thêm vào một hàng đợi các luồng đang chờ tài nguyên này.

Một tập các luồng ở trạng thái tắc nghẽn khi mọi luồng trong tập đó đang chờ một sự kiện mà gây ra bởi luồng khác trong tập hợp. Các sự kiện này chủ yếu là nhận và giải phóng tài nguyên. Các tài nguyên thường logic (không phải vật lí), chẳng hạn như khóa mutex, semaphores và tệp.

## 6.2. Các ví dụ tắc nghẽn

### 6.2.1. Tắc nghẽn giao thông

Một loại tắc nghẽn ta hay gặp trong thực tế đó là tắc nghẽn giao thông trong các thành phố lớn. Ví dụ như Hình 6.1.



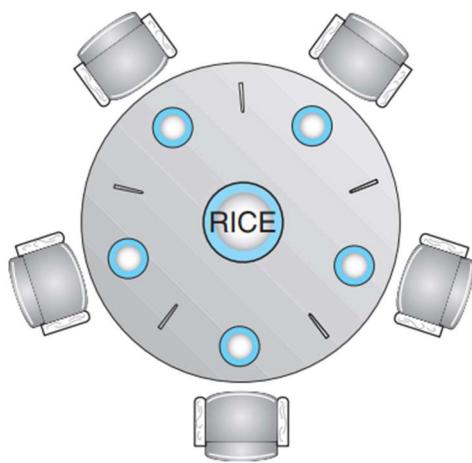
**Hình 6.1.** Tắc nghẽn giao thông; (a) có thể tắc nghẽn, (b) tắc nghẽn.

Trong hình trên ta thấy xe 1 chờ xe 2, xe 2 chờ xe 3, xe 3 chờ xe 4 và xe 4 lại chờ xe 1.

### 6.2.2. Bài toán các nhà triết học ăn tối

Hãy xét năm triết gia dành cả đời để suy nghĩ và ăn uống. Các nhà triết học chia sẻ một chiếc bàn tròn được bao quanh bởi năm chiếc ghế, mỗi chiếc thuộc về một triết gia. Chính giữa bàn là một bát cơm, trên bàn được bày năm chiếc đũa đơn (Hình 6.2). Khi một triết gia suy nghĩ, ông ấy không tương tác với các đồng nghiệp của mình. Đôi khi, một nhà triết học cảm thấy đói và cố gắng nhặt hai chiếc đũa gần nhất (chiếc đũa

nằm giữa ông ấy và những người hàng xóm bên trái và bên phải). Một triết gia có thể chỉ nhặt một chiếc đũa mỗi lần. Rõ ràng là ông ta không thể nhặt một chiếc đũa đã có trong tay của một người hàng xóm. Khi một triết gia đó có cả hai chiếc đũa cùng một lúc, ông ta ăn mà không cần nhả đũa ra. Khi ăn xong, ông ta đặt cả hai đũa xuống và bắt đầu suy nghĩ lại. Bài toán nhà triết học ăn uống được coi là một bài toán đồng bộ cổ điển không phải vì tầm quan trọng thực tế của nó mà vì nó là một ví dụ của một nhóm các bài toán điều khiển đồng thời. Nó là một đại diện đơn giản về sự cần thiết phải phân bổ một số tài nguyên giữa một số tiến trình theo cách không có tắc nghẽn và không bị chết đói.



**Hình 6.2.** Mô tả tình huống 5 nhà triết học ăn tối.

### 6.3. Đặc tính của tắc nghẽn

#### 6.3.1. Điều kiện xuất hiện tắc nghẽn

Một tình huống tắc nghẽn có thể phát sinh nếu bốn điều kiện sau đây đồng thời tồn tại trong một hệ thống:

1. Loại trừ lẫn nhau. Ít nhất một tài nguyên phải được giữ ở chế độ không thể chia sẻ; nghĩa là mỗi lần chỉ có một luồng có thể sử dụng tài nguyên. Nếu một luồng khác yêu cầu tài nguyên đó, thì luồng yêu cầu phải chờ cho đến khi tài nguyên đó đã được giải phóng.

2. Giữ và chờ đợi. Một luồng phải đang nắm giữ ít nhất một tài nguyên và chờ đợi để có được các tài nguyên bổ sung hiện đang được các luồng khác nắm giữ.

3. Độc quyền. Các tài nguyên độc quyền; nghĩa là, một tài nguyên chỉ có thể được giải phóng một cách tự nguyện bởi luồng đang giữ nó, sau khi luồng đó đã hoàn thành nhiệm vụ của nó.

4. Vòng chờ. Tập hợp các luồng  $\{T_0, T_1, \dots, T_n\}$  chờ.  $T_0$  đang chờ tài nguyên do  $T_1$

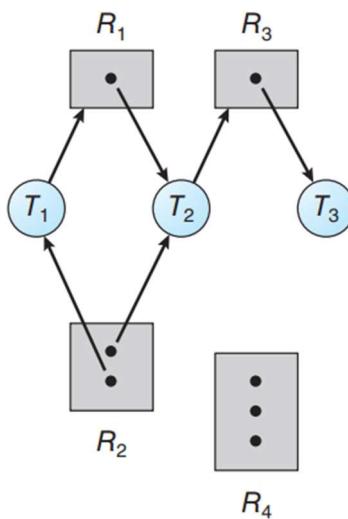
giữ,  $T_1$  đang chờ tài nguyên do  $T_2$  giữ, ...,  $T_{n-1}$  đang chờ tài nguyên do  $T_n$  giữ và  $T_n$  đang chờ tài nguyên do  $T_0$  giữ.

### 6.3.2. Đồ thị cấp phát tài nguyên

Tắc nghẽn có thể được mô tả chính xác hơn dưới dạng đồ thị có hướng được gọi là đồ thị phân bổ tài nguyên hệ thống. Đồ thị này bao gồm một tập các đỉnh  $V$  và một tập các cạnh  $E$ . Tập các đỉnh  $V$  được chia thành hai loại nút khác nhau:  $T = \{T_1, T_2, \dots, T_n\}$ , tập hợp bao gồm tất cả các luồng hoạt động trong hệ thống và  $R = \{R_1, R_2, \dots, R_m\}$ , tập hợp bao gồm tất cả các loại tài nguyên trong hệ thống.

Một cạnh có hướng từ luồng  $T_i$  đến kiểu tài nguyên  $R_j$  được ký hiệu là  $T_i \rightarrow R_j$ ; nó biểu thị rằng luồng  $T_i$  đã yêu cầu một thể hiện của loại tài nguyên  $R_j$  và hiện đang đợi tài nguyên đó. Một cạnh có hướng từ kiểu tài nguyên  $R_j$  đến luồng  $T_i$  được ký hiệu là  $R_j \rightarrow T_i$ ; nó biểu thị rằng một thể hiện của kiểu tài nguyên  $R_j$  đã được cấp phát cho luồng  $T_i$ . Một cạnh có hướng  $T_i \rightarrow R_j$  được gọi là một cạnh yêu cầu; một cạnh có hướng  $R_j \rightarrow T_i$  được gọi là một cạnh gán.

Về cơ bản, chúng ta biểu diễn mỗi luồng  $T_i$  dưới dạng hình tròn và mỗi loại tài nguyên  $R_j$  dưới dạng hình chữ nhật. Ví dụ đơn giản, đồ thị phân bổ tài nguyên trong Hình 6.3 minh họa tình huống deadlock. Vì kiểu tài nguyên  $R_j$  có thể có nhiều hơn một thể hiện, chúng ta biểu diễn mỗi thể hiện đó như một dấu chấm trong hình chữ nhật. Lưu ý rằng cạnh yêu cầu trỏ đến hình chữ nhật  $R_j$ , trong khi cạnh gán cũng phải trỏ đến một trong các dấu chấm trong hình chữ nhật.



**Hình 6.3.** Đồ thị cấp phát tài nguyên.

Khi luồng  $T_i$  yêu cầu một thể hiện của kiểu tài nguyên  $R_j$ , một cạnh yêu cầu sẽ được chèn vào đồ thị phân bổ tài nguyên. Khi yêu cầu này có thể được thực hiện, cạnh

yêu cầu được chuyển đổi ngay lập tức thành một cạnh gán. Khi luồng không cần truy cập tài nguyên nữa, nó sẽ giải phóng tài nguyên. Kết quả là, cạnh gán bị xóa.

Biểu đồ phân bổ tài nguyên trong Hình 6.3 mô tả tình huống sau:

- Tập hợp T, R và E:

- $T = \{T_1, T_2, T_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$

- Các thể hiện tài nguyên:

- Một thể hiện của loại tài nguyên  $R_1$ .
- Hai thể hiện của loại tài nguyên  $R_2$ .
- Một thể hiện của loại tài nguyên  $R_3$ .
- Ba thể hiện của loại tài nguyên  $R_4$ .

- Trạng thái luồng:

- Luồng  $T_1$  đang giữ một thể hiện của loại tài nguyên  $R_2$  và đang chờ một thể hiện của loại tài nguyên  $R_1$ .
  - Luồng  $T_2$  đang giữ một thể hiện của  $R_1$  và một thể hiện của  $R_2$  và đang đợi một thể hiện của  $R_3$ .
  - Luồng  $T_3$  đang giữ một thể hiện của  $R_3$ .

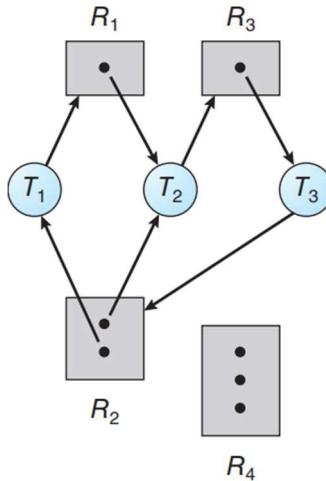
Đưa ra định nghĩa về đồ thị phân bổ tài nguyên, có thể chỉ ra rằng, nếu đồ thị không chứa chu trình, thì không có luồng nào trong hệ thống bị tắc nghẽn. Nếu đồ thị có chứa một chu trình, thì có thể tồn tại một tắt nghẽn.

Nếu mỗi loại tài nguyên có chính xác một thể hiện, thì một chu trình ngụ ý rằng một tắc nghẽn đã xảy ra. Nếu chu trình chỉ liên quan đến một tập hợp các loại tài nguyên, mỗi loại chỉ có một thể hiện duy nhất, thì một tắc nghẽn đã xảy ra. Mỗi luồng tham gia vào chu trình bị khóa. Trong trường hợp này, một chu trình trong đồ thị vừa là điều kiện cần vừa là điều kiện đủ để tồn tại deadlock.

Nếu mỗi loại tài nguyên có một số thể hiện, thì một chu kỳ không nhất thiết ngụ ý rằng một tắc nghẽn đã xảy ra. Trong trường hợp này, một chu trình trong đồ thị là điều kiện cần nhưng không phải là điều kiện đủ để tồn tại tắc nghẽn.

Để minh họa khái niệm này, chúng ta quay lại đồ thị phân bổ tài nguyên được mô

tả trong Hình 6.3. Giả sử rằng luồng  $T_3$  yêu cầu một thẻ hiện của loại tài nguyên  $R_2$ . Vì hiện tại không có thẻ hiện tài nguyên nào, chúng ta thêm một cạnh yêu cầu  $T_3 \rightarrow R_2$  vào đồ thị (Hình 6.4).



**Hình 6.4.** Đồ thị cấp phát tài nguyên có tắc nghẽn.

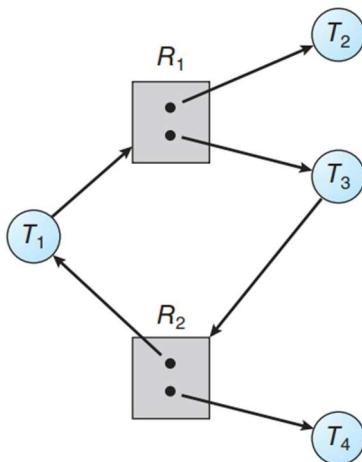
Tại thời điểm này, tồn tại hai chu trình trong hệ thống:

$$T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1.$$

$$T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2.$$

Các luồng  $T_1$ ,  $T_2$  và  $T_3$  bị khóa. Luồng  $T_2$  đang đợi tài nguyên  $R_3$ , được giữ bởi luồng  $T_3$ . Luồng  $T_3$  đang đợi luồng  $T_1$  hoặc luồng  $T_2$  giải phóng tài nguyên  $R_2$ . Ngoài ra, luồng  $T_1$  đang chờ luồng  $T_2$  giải phóng tài nguyên  $R_1$ .

Xét ví dụ biểu đồ phân bổ tài nguyên trong Hình 6.5.



**Hình 6.5.** Đồ thị cấp phát tài nguyên có chu trình nhưng không tắc nghẽn.

Trong ví dụ này, chúng ta cũng có một chu trình:  $T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$ . Tuy nhiên, không có tắc nghẽn. Ta thấy luồng  $T_4$  có thẻ giải phóng thẻ hiện của loại tài

nguyên  $R_2$ . Tài nguyên này sau đó có thể được phân bổ cho  $T_3$ , phá vỡ chu trình.

Tóm lại, nếu đồ thị phân bổ tài nguyên không có chu trình thì hệ thống không ở trạng thái tắc nghẽn. Nếu có một chu trình, thì hệ thống có thể ở trạng thái tắc nghẽn hoặc không. Quan sát này rất quan trọng khi chúng ta giải quyết vấn đề tắc nghẽn.

### 6.3.3. Phương pháp xử lý tắc nghẽn

## 6.4. Ngăn ngừa tắc nghẽn

Như chúng ta đã lưu ý trong Phần 6.3.1, để xảy ra tắc nghẽn, cả bốn điều kiện cần thiết phải xuất hiện. Bằng cách đảm bảo rằng ít nhất một trong những điều kiện này không thể xảy ra, chúng ta có thể ngăn chặn sự xuất hiện của tắc nghẽn.

### 6.4.1. Loại trừ lẫn nhau

Điều kiện để xuất hiện tình huống loại trừ lẫn nhau, đó là, ít nhất một tài nguyên phải không chia sẻ được. Các tài nguyên có thể chia sẻ không yêu cầu quyền truy cập loại trừ lẫn nhau và do đó không thể liên quan đến tắc nghẽn. Tệp chỉ đọc là một ví dụ điển hình về tài nguyên có thể chia sẻ. Nếu một số luồng có gắng mở tệp chỉ đọc cùng một lúc, chúng có thể được cấp quyền truy cập đồng thời vào tệp. Một luồng không cần phải đợi một tài nguyên có thể chia sẻ được. Tuy nhiên, nói chung, chúng ta không thể ngăn chặn tắc nghẽn bằng cách loại bỏ điều kiện loại trừ lẫn nhau, bởi vì bản chất một số tài nguyên không thể chia sẻ được. Ví dụ, một khóa mutex không thể được chia sẻ đồng thời bởi một số luồng.

### 6.4.2. Giữ và đợi

Để đảm bảo rằng điều kiện giữ và chờ không bao giờ xảy ra trong hệ thống, chúng ta phải đảm bảo rằng, bất cứ khi nào một luồng yêu cầu một tài nguyên, nó không giữ bất kỳ tài nguyên nào khác. Một giao thức mà chúng ta có thể sử dụng bắt buộc mỗi luồng yêu cầu và được cấp phát tất cả tài nguyên của nó trước khi nó bắt đầu thực thi. Tuy nhiên, điều này là không thực tế đối với hầu hết các ứng dụng do tính chất động của việc yêu cầu tài nguyên.

Một giao thức thay thế cho phép một luồng chỉ yêu cầu tài nguyên khi nó không có. Một luồng có thể yêu cầu một số tài nguyên và sử dụng chúng. Trước khi nó có thể yêu cầu bất kỳ tài nguyên bổ sung nào, nó phải giải phóng tất cả các tài nguyên mà nó hiện được cấp phát. Cả hai giao thức này đều có hai nhược điểm chính. Thứ nhất, việc sử dụng tài nguyên có thể thấp, vì tài nguyên có thể được phân bổ nhưng không được sử dụng trong một thời gian dài.

Ví dụ: một luồng có thể được cấp phát một khóa mutex cho toàn bộ quá trình thực

thi của nó, nhưng chỉ yêu cầu nó trong một thời gian ngắn. Thứ hai, có thể xảy ra chết đói. Một luồng cần một số tài nguyên phổ biến có thể phải chờ vô thời hạn, bởi vì ít nhất một trong các tài nguyên mà nó cần luôn được phân bổ cho một số luồng khác.

#### 6.4.3. Không ưu tiên (độc quyền)

Điều kiện cần thiết thứ ba đối với tắc nghẽn là độc quyền các tài nguyên đã được cấp phát. Để đảm bảo rằng điều kiện này không tồn tại, chúng ta có thể sử dụng giao thức sau. Nếu một luồng đang giữ một số tài nguyên và yêu cầu một tài nguyên khác mà không thể cấp phát ngay cho nó (tức là luồng phải đợi), thì tất cả các tài nguyên mà luồng hiện đang nắm giữ sẽ nhường lại (không độc quyền chiếm giữ). Nói cách khác, các tài nguyên này được giải phóng một cách ngầm định. Các tài nguyên được không độc quyền được thêm vào danh sách các tài nguyên mà luồng đang đợi. Luồng sẽ chỉ được khởi động lại khi nó có thể lấy lại các tài nguyên cũ, cũng như các tài nguyên mới mà nó đang yêu cầu.

Ngoài ra, nếu một luồng yêu cầu một số tài nguyên, trước tiên chúng ta kiểm tra xem chúng có sẵn hay không. Nếu có thì cấp phát nó. Nếu không, chúng ta kiểm tra xem chúng có được cấp phát cho một số luồng khác đang chờ tài nguyên bổ sung hay không. Nếu có, chúng ta giành các tài nguyên mong muốn từ luồng chờ và phân bổ chúng cho luồng yêu cầu. Nếu các tài nguyên không có sẵn hoặc không được giữ bởi một luồng đang chờ, thì luồng yêu cầu phải đợi. Trong khi chờ đợi, một số tài nguyên của nó có thể được nhường lại cho những luồng khác yêu cầu chúng. Một luồng chỉ có thể được khởi động lại khi nó được cấp phát tài nguyên mới mà nó đang yêu cầu và khôi phục bất kỳ tài nguyên nào đã được nhường trong khi chờ đợi.

Giao thức này thường được áp dụng cho các tài nguyên mà trạng thái của chúng có thể dễ dàng lưu và khôi phục sau này, chẳng hạn như thanh ghi CPU và các giao dịch cơ sở dữ liệu. Nó thường không thể áp dụng cho các tài nguyên như khóa mutex và semaphores (chính xác là loại tài nguyên mà tắc nghẽn thường xảy ra nhất).

#### 6.4.4. Chờ vòng tròn

Ba lựa chọn được trình bày cho đến nay để ngăn chặn tắc nghẽn nói chung là không thực tế trong hầu hết các tình huống. Tuy nhiên, điều kiện thứ tư và cũng là điều kiện cuối cùng cho tắc nghẽn - điều kiện chờ vòng tròn - tạo cơ hội cho một giải pháp thực tế bằng cách làm mất hiệu lực của một trong những điều kiện cần thiết. Một cách để đảm bảo điều kiện này không bao giờ đúng là áp đặt thứ tự tổng thể của tất cả các loại tài nguyên và yêu cầu mỗi luồng yêu cầu tài nguyên theo thứ tự liệt kê ngày càng tăng.

Để minh họa, ta đặt  $R = \{R_1, R_2, \dots, R_m\}$  là tập các loại tài nguyên. Chúng ta gán

cho mỗi loại tài nguyên một số nguyên duy nhất, cho phép so sánh hai tài nguyên và xác định xem tài nguyên này đứng trước tài nguyên khác trong thứ tự của chúng. Về mặt hình thức, chúng ta định nghĩa hàm  $F: R \rightarrow N$ , trong đó  $N$  là tập các số tự nhiên.

Bây giờ xét giao thức sau để ngăn chặn tắc nghẽn: Mỗi luồng chỉ có thể yêu cầu tài nguyên theo thứ tự liệt kê ngày càng tăng. Có nghĩa là, ban đầu một luồng có thể yêu cầu một phiên bản của tài nguyên - chẳng hạn như  $R_i$ . Sau đó, luồng có thể yêu cầu một thể hiện của tài nguyên  $R_j$  khi và chỉ khi  $F(R_j) > F(R_i)$ . Ví dụ: bằng cách sử dụng hàm được định nghĩa ở trên, một luồng muốn sử dụng cả mutex thứ nhất và mutex thứ hai cùng một lúc, trước tiên phải yêu cầu mutex thứ nhất và sau đó là mutex thứ hai. Ngoài ra, chúng ta có thể bắt buộc một luồng yêu cầu một thể hiện của tài nguyên  $R_j$  phải giải phóng bất kỳ tài nguyên nào  $R_i$  mà sao cho  $F(R_i) \geq F(R_j)$ . Cũng lưu ý rằng nếu cần một số thể hiện của cùng một loại tài nguyên, thì chỉ đưa ra một yêu cầu duy nhất cho tất cả chúng phải được đưa ra.

Nếu hai giao thức này được sử dụng, thì điều kiện chờ vòng tròn không thể xảy ra. Có thể chứng minh thực tế này bằng cách giả định rằng tồn tại một chờ vòng tròn (chứng minh phản chứng). Đặt tập hợp các luồng tham gia vào vòng chờ là  $\{T_0, T_1, \dots, T_n\}$ , trong đó  $T_i$  đang đợi một tài nguyên  $R_i$ , được giữ bởi luồng  $T_{i+1}$ .  $T_n$  đang đợi tài nguyên  $R_n$  do  $T_0$  nắm giữ. Sau đó, vì luồng  $T_{i+1}$  đang giữ tài nguyên  $R_i$  trong khi yêu cầu tài nguyên  $R_{i+1}$ , chúng ta phải có  $F(R_i) < F(R_{i+1}) \forall i$ . Nhưng điều kiện này có nghĩa là  $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$ . Dẫn đến,  $F(R_0) < F(R_0)$ , điều này là không thể. Do đó, không thể có sự chờ đợi vòng tròn.

## 6.5. Tránh tắc nghẽn

Các thuật toán ngăn chặn tắc nghẽn, như đã thảo luận trong Phần 6.4, ngăn chặn các tắc nghẽn bằng cách đảm bảo rằng ít nhất một trong các điều kiện cần thiết để không thể xảy ra tắc nghẽn. Tuy nhiên, các nhược điểm của việc ngăn chặn tắc nghẽn bằng phương pháp này là hiệu suất sử dụng thiết bị thấp và giảm thông lượng hệ thống.

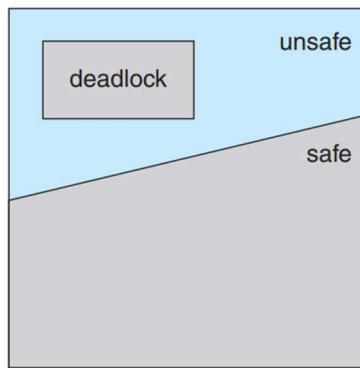
Một phương pháp thay thế để tránh tắc nghẽn là yêu cầu thông tin bổ sung về cách yêu cầu tài nguyên. Ví dụ, trong một hệ thống có tài nguyên  $R_1$  và  $R_2$ , hệ thống có thể cần biết rằng luồng  $P$  sẽ yêu cầu  $R_1$  đầu tiên và sau đó là  $R_2$  trước khi giải phóng cả hai tài nguyên, trong khi luồng  $Q$  sẽ yêu cầu  $R_2$  và sau đó là  $R_1$ . Với thông tin về chuỗi hoàn chỉnh các yêu cầu và giải phóng tài nguyên cho mỗi luồng, hệ thống có thể quyết định liệu luồng đó có nên đợi hay không để tránh tình trạng tắc nghẽn có thể xảy ra trong tương lai. Mỗi yêu cầu tài nguyên, hệ thống phải xem xét các tài nguyên hiện có, tài nguyên hiện được phân bổ cho mỗi luồng cũng như các yêu cầu và giải phóng trong

tương lai của mỗi luồng.

Các thuật toán khác nhau sử dụng cách tiếp cận này khác nhau về số lượng và loại thông tin được yêu cầu. Mô hình đơn giản và hữu ích nhất yêu cầu mỗi luồng khai báo số lượng tài nguyên tối đa của mỗi loại mà nó có thể cần. Với thông tin tiên nghiệm này, có thể xây dựng một thuật toán đảm bảo rằng hệ thống sẽ không bao giờ đi vào trạng thái tắc nghẽn. Một thuật toán tránh tắc nghẽn tự động kiểm tra trạng thái cấp phát tài nguyên để đảm bảo rằng điều kiện chờ vòng tròn không bao giờ có thể tồn tại. Trạng thái phân bổ tài nguyên được xác định bởi số lượng tài nguyên có sẵn và được phân bổ cũng như nhu cầu tối đa của các luồng. Trong các phần sau, chúng ta khám phá hai thuật toán tránh tắc nghẽn.

#### 6.5.1. Trạng thái an toàn

Trạng thái là an toàn nếu hệ thống có thể phân bổ tài nguyên cho mỗi luồng (tối đa) theo một số thứ tự mà vẫn tránh được tắc nghẽn. Định nghĩa hình thức, một hệ thống ở trạng thái an toàn nếu tồn tại một chuỗi an toàn. Một chuỗi các luồng  $\langle T_1, T_2, \dots, T_n \rangle$  ở trạng thái an toàn nếu đối với mỗi  $T_i$ , các yêu cầu tài nguyên mà  $T_i$  vẫn có thể được đáp ứng bởi các tài nguyên hiện có cộng với các tài nguyên được giữ bởi tất cả  $T_j$ , với  $j < i$ . Trong tình huống này, nếu tài nguyên mà  $T_i$  cần không có sẵn ngay lập tức, thì  $T_i$  có thể đợi cho đến khi tất cả các  $T_j$  đã hoàn thành. Khi chúng hoàn thành,  $T_i$  có thể lấy tất cả các tài nguyên cần thiết, hoàn thành nhiệm vụ được chỉ định, trả lại các tài nguyên đã được phân bổ và kết thúc. Khi  $T_i$  kết thúc,  $T_{i+1}$  có thể nhận được các tài nguyên cần thiết, v.v. Nếu không có trình tự nào như vậy tồn tại, thì trạng thái hệ thống được cho là không an toàn.



**Hình 6.6.** Không gian trạng thái an toàn, không an toàn, tắc nghẽn.

Trạng thái an toàn không phải là trạng thái tắc nghẽn. Ngược lại, trạng thái tắc nghẽn là trạng thái không an toàn. Tuy nhiên, không phải tất cả các trạng thái không an toàn đều là tắc nghẽn (Hình 6.6). Trạng thái không an toàn có thể dẫn đến tắc nghẽn.

Miễn là trạng thái an toàn, hệ điều hành có thể tránh được trạng thái không an toàn (và tắc nghẽn).

Để minh họa, hãy xem xét một hệ thống có 12 tài nguyên và ba luồng:  $T_0$ ,  $T_1$  và  $T_2$ . Luồng  $T_0$  yêu cầu 10 tài nguyên, luồng  $T_1$  có thể cần đến 4 tài nguyên và luồng  $T_2$  có thể cần đến 9 tài nguyên. Giả sử rằng, tại thời điểm  $t_0$ , luồng  $T_0$  đang giữ 5 tài nguyên, luồng  $T_1$  đang giữ 2 tài nguyên và luồng  $T_2$  đang giữ 2 tài nguyên. (Do đó, có 3 tài nguyên đang rối).

	Nhu cầu tối đa	Hiện tại đang giữ
$T_0$	10	5
$T_1$	4	2
$T_2$	9	2

Tại thời điểm  $t_0$ , hệ thống ở trạng thái an toàn. Chuỗi  $\langle T_1, T_0, T_2 \rangle$  thỏa mãn điều kiện an toàn. Luồng  $T_1$  ngay lập tức có thể được cấp phát tất cả các tài nguyên của nó và sau đó trả lại chúng (khi đó hệ thống sẽ có 5 tài nguyên khả dụng); thì luồng  $T_0$  có thể lấy tất cả các tài nguyên của nó và trả lại chúng (khi đó hệ thống sẽ có sẵn 10 tài nguyên); và cuối cùng luồng  $T_2$  có thể lấy tất cả các tài nguyên của nó và trả lại chúng (khi đó hệ thống sẽ có sẵn tất cả 12 tài nguyên).

Một hệ thống có thể đi từ trạng thái an toàn sang trạng thái không an toàn. Giả sử rằng, tại thời điểm  $t_1$ , luồng  $T_2$  yêu cầu và được cấp phát thêm 1 tài nguyên. Hệ thống không còn ở trạng thái an toàn. Tại thời điểm này, chỉ luồng  $T_1$  có thể được cấp phát tất cả các tài nguyên của nó. Khi nó trả về chúng, hệ thống sẽ chỉ có 4 tài nguyên khả dụng. Vì luồng  $T_0$  đã được cấp phát 5 tài nguyên nhưng có tối đa 10 tài nguyên, nó có thể yêu cầu thêm 5 tài nguyên. Nếu như vậy, nó sẽ phải đợi, vì tài nguyên không có sẵn. Tương tự, luồng  $T_2$  có thể yêu cầu 6 tài nguyên bổ sung và phải chờ, dẫn đến tắc nghẽn. Sai lầm của chúng ta là cấp cho luồng  $T_2$  1 tài nguyên khác. Nếu chúng ta đã bắt  $T_2$  đợi cho đến khi một trong các luồng khác kết thúc và giải phóng tài nguyên của nó, thì chúng ta có thể tránh được tắc nghẽn.

Với khái niệm về trạng thái an toàn, chúng ta có thể xác định các thuật toán tránh để đảm bảo rằng hệ thống sẽ không bao giờ bị tắc nghẽn. Ý tưởng chỉ đơn giản là đảm bảo rằng hệ thống sẽ luôn duy trì ở trạng thái an toàn. Ban đầu, hệ thống ở trạng thái an toàn. Bất cứ khi nào một luồng yêu cầu một tài nguyên hiện đang có sẵn, hệ thống phải quyết định xem liệu tài nguyên đó có thể được cấp phát ngay lập tức hay luồng phải đợi. Yêu cầu chỉ được cấp nếu việc phân bổ khiến hệ thống ở trạng thái an toàn.

### 6.5.2. Thuật toán đồ thị cấp phát tài nguyên

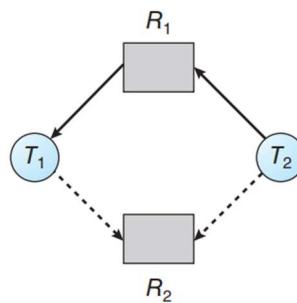
Nếu có một hệ thống cấp phát chỉ có một thể hiện của mỗi loại tài nguyên, ta có thể sử dụng một biến thể của đồ thị cấp phát tài nguyên được xác định trong Phần 6.3.2 để tránh tắc nghẽn. Ngoài các cạnh yêu cầu và cấp phát đã được mô tả, ta định nghĩa một loại cạnh mới, được gọi là cạnh đăng ký trước. Một cạnh đăng ký trước  $T_i \rightarrow R_j$  chỉ ra rằng luồng  $T_i$  có thể yêu cầu tài nguyên  $R_j$  vào một thời điểm nào đó trong tương lai. Cạnh này giống cạnh yêu cầu về hướng nhưng được biểu diễn trong biểu đồ bằng một đường nét đứt. Khi luồng  $T_i$  yêu cầu tài nguyên  $R_j$ , cạnh đăng ký trước  $T_i \rightarrow R_j$  được chuyển đổi thành cạnh yêu cầu. Tương tự, khi một tài nguyên  $R_j$  được giải phóng bởi  $T_i$ , cạnh cấp phát  $R_j \rightarrow T_i$  được chuyển đổi lại thành một đăng ký trước  $T_i \rightarrow R_j$ .

Lưu ý rằng các tài nguyên phải được xác nhận quyền sở hữu trước trong hệ thống. Nghĩa là, trước khi luồng  $T_i$  bắt đầu thực thi, tất cả các cạnh đăng ký trước của nó phải xuất hiện trong đồ thị phân bổ tài nguyên. Chúng ta có thể nói lòng điều kiện này bằng cách chỉ cho phép thêm một cạnh đăng ký trước  $T_i \rightarrow R_j$  vào đồ thị nếu tất cả các cạnh được liên kết với luồng  $T_i$  là các cạnh đăng ký trước.

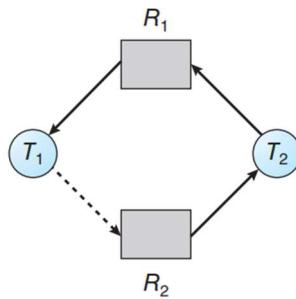
Bây giờ, giả sử rằng luồng  $T_i$  yêu cầu tài nguyên  $R_j$ . Yêu cầu chỉ có thể được đáp ứng nếu việc chuyển đổi cạnh yêu cầu  $T_i \rightarrow R_j$  thành cạnh cấp phát  $R_j \rightarrow T_i$  không dẫn đến việc hình thành một chu trình trong đồ thị phân bổ tài nguyên. Ta kiểm tra trạng thái an toàn bằng cách sử dụng thuật toán phát hiện chu trình. Thuật toán để phát hiện một chu trình trong đồ thị có độ phức tạp là  $O(n^2)$ , trong đó  $n$  là số luồng trong hệ thống.

Nếu không có chu trình nào tồn tại, thì việc phân bổ tài nguyên sẽ làm hệ thống ở trạng thái an toàn. Nếu có một chu trình, thì việc phân bổ sẽ đưa hệ thống vào trạng thái không an toàn. Khi đó, luồng  $T_i$  sẽ phải đợi các yêu cầu của nó được đáp ứng.

Để minh họa thuật toán này, xét đồ thị phân bổ tài nguyên ở Hình 6.7. Giả sử rằng  $T_2$  yêu cầu  $R_2$ . Mặc dù  $R_2$  hiện đang rỗi, nhưng không thể cấp phát nó cho  $T_2$ , vì hành động này sẽ tạo ra một chu trình trong đồ thị (Hình 6.8) và hệ thống sẽ ở trạng thái không an toàn. Nếu  $T_1$  yêu cầu  $R_2$  và  $T_2$  yêu cầu  $R_1$ , thì xảy ra tắc nghẽn.



**Hình 6.7.** Đồ thị cấp phát tài nguyên để tránh tắc nghẽn.



**Hình 6.8.** Trạng thái không an toàn trong đồ thị cấp phát.

### 6.5.3. Thuật toán Banker

Thuật toán đồ thị cấp phát tài nguyên không áp dụng được cho hệ thống cấp phát tài nguyên có nhiều thể hiện của mỗi loại tài nguyên. Thuật toán tránh tắc nghẽn sau đây có thể áp dụng cho một hệ thống như vậy nhưng kém hiệu quả hơn so với đồ thị phân bổ tài nguyên. Thuật toán này thường được gọi là thuật toán của chủ ngân hàng (Banker). Tên này được chọn vì thuật toán có thể được sử dụng trong hệ thống ngân hàng để đảm bảo rằng ngân hàng không bao giờ phân bổ tiền mặt sẵn có của mình theo cách mà nó không còn có thể đáp ứng nhu cầu của tất cả khách hàng.

Khi một luồng mới vào hệ thống, nó phải khai báo số thể hiện tối đa của mỗi loại tài nguyên mà nó cần. Con số này không được vượt quá tổng số tài nguyên trong hệ thống. Khi người dùng yêu cầu một tập hợp tài nguyên, hệ thống phải xác định xem việc phân bổ các tài nguyên này có khiến hệ thống ở trạng thái an toàn hay không? Nếu an toàn, các tài nguyên sẽ được cấp phát; nếu không, luồng phải đợi cho đến khi một số luồng khác giải phóng đủ tài nguyên.

Một số cấu trúc dữ liệu được duy trì để cài đặt thuật toán Banker. Các cấu trúc dữ liệu này mã hóa trạng thái của hệ thống cấp phát tài nguyên. Ta cần các cấu trúc dữ liệu sau, trong đó  $n$  là số luồng trong hệ thống và  $m$  là số loại tài nguyên:

- **Available:** Một vectơ có độ dài  $m$  cho biết số lượng tài nguyên hiện có của mỗi loại. Nếu  $\text{Available}[j] = k$ , thì  $k$  thể hiện của loại tài nguyên  $R_j$  có sẵn.
- **Max:** Một ma trận  $n \times m$  xác định nhu cầu tối đa của mỗi luồng. Nếu  $\text{Max}[i][j] = k$ , thì luồng  $T_i$  có thể yêu cầu tối đa  $k$  trường hợp của loại tài nguyên  $R_j$ .
- **Allocation:** Một ma trận  $n \times m$  xác định số lượng tài nguyên của mỗi loại hiện được phân bổ cho mỗi luồng. Nếu  $\text{Allocation}[i][j] = k$ , thì luồng  $T_i$  hiện được cấp phát  $k$  thể hiện của loại tài nguyên  $R_j$ .
- **Need:** Một ma trận  $n \times m$  cho biết nhu cầu tài nguyên còn lại của mỗi luồng. Nếu  $\text{Need}[i][j] = k$ , thì luồng  $T_i$  có thể cần thêm  $k$  thể hiện của loại tài nguyên  $R_j$  để hoàn

thành nhiệm vụ của nó. Lưu ý rằng  $\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$ .

Các cấu trúc dữ liệu này thay đổi theo thời gian cả về kích thước và giá trị.

Để đơn giản hóa việc trình bày thuật toán Banker, tiếp theo ta định nghĩa một số ký hiệu. Gọi  $X$  và  $Y$  là các vectơ có độ dài  $n$ . Ta có  $X \leq Y$  khi và chỉ khi  $X[i] \leq Y[i] \forall i = 1, 2, \dots, n$ . Ví dụ, nếu  $X = (1, 7, 3, 2)$  và  $Y = (0, 3, 2, 1)$ , thì  $Y \leq X$ . Ngoài ra,  $Y < X$  nếu  $Y \leq X$  và  $Y \neq X$ .

Ta có thể coi mỗi hàng trong ma trận Allocation và Need là vectơ và gọi chúng là  $\text{Allocation}_i$  và  $\text{Need}_i$ . Vector  $\text{Allocation}_i$  chỉ rõ các tài nguyên hiện được cấp cho luồng  $T_i$ ; vectơ  $\text{Need}_i$  chỉ rõ các tài nguyên bổ sung mà luồng  $T_i$  vẫn có thể yêu cầu để hoàn thành nhiệm vụ của nó.

a. *Thuật toán kiểm tra trạng thái an toàn*

Đây là thuật toán kiểm tra liệu hệ thống có ở trạng thái an toàn hay không. Thuật toán này mô tả như sau:

1. Gọi  $\text{Work}$  và  $\text{Finish}$  lần lượt là các vectơ có độ dài  $m$  và  $n$ . Khởi tạo  $\text{Work} = \text{Available}$  và  $\text{Finish}[i] = \text{false } \forall i = 0..n-1$ .
2. Tìm một chỉ số  $i$  sao cho cả hai:
  - a.  $\text{Finish}[i] == \text{false}$ , và
  - b.  $\text{Need}_i \leq \text{Work}$ .
- Nếu không có  $i$  như vậy tồn tại, hãy chuyển sang bước 4.
3.  $\text{Work} = \text{Work} + \text{Allocation}_i$   
 $\text{Finish}[i] = \text{true}$   
Chuyển sang bước 2.
4. Nếu  $\text{Finish}[i] == \text{true } \forall i$ , thì hệ thống đang ở trạng thái an toàn.

Thuật toán có độ phức tạp  $O(mn^2)$ .

b. *Thuật toán yêu cầu tài nguyên*

Tiếp theo là thuật toán xác định liệu các yêu cầu có thể được cấp phát một cách an toàn hay không? Gọi  $\text{Request}_i$  là vectơ yêu cầu cho luồng  $T_i$ . Nếu  $\text{Request}_i[j] = k$ , nghĩa là luồng  $T_i$  muốn  $k$  đơn vị của loại tài nguyên  $R_j$ . Khi luồng  $T_i$  yêu cầu tài nguyên, các hành động sau được thực hiện:

1. Nếu  $\text{Request}_i \leq \text{Need}_i$ , chuyển sang bước 2. Nếu không, phát sinh lỗi vì luồng yêu cầu vượt quá nhu cầu tối đa của nó.
2. Nếu  $\text{Request}_i \leq \text{Available}$ , chuyển sang bước 3. Nếu không,  $T_i$  phải đợi, vì tài nguyên không có sẵn.

3. Hệ thống cấp phát thử tài nguyên được yêu cầu đến luồng  $T_i$  bằng cách sửa đổi trạng thái như sau:

$$\text{Available} = \text{Available} - \text{Request}_{T_i}$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_{T_i}$$

Nếu kết quả trạng thái phân bổ tài nguyên này là an toàn, giao dịch đã hoàn tất và luồng  $T_i$  được cấp phát tài nguyên của nó. Tuy nhiên, nếu trạng thái mới không an toàn, thì  $T_i$  phải đợi  $\text{Request}_i$  và khôi phục trạng thái cấp phát tài nguyên cũ.

### c. Ví dụ minh họa

Để minh họa việc sử dụng thuật toán Banker, xét một hệ thống có 5 luồng từ  $T_0$  đến  $T_4$  và 3 loại tài nguyên A, B và C. Loại tài nguyên A có 10 thể hiện, tài nguyên B có 5 thể hiện và tài nguyên C có 7 bảy thể hiện. Giả sử rằng bảng sau đây biểu diễn trạng thái hiện tại của hệ thống:

Luồng	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
$T_0$	0	1	0	7	5	3	7	4	3			
$T_1$	2	0	0	3	2	2	1	2	2			
$T_2$	3	0	2	9	0	2	6	0	0	3	3	2
$T_3$	2	1	1	2	2	2	0	1	1			
$T_4$	0	0	2	4	3	3	4	3	1			

Nội dung cột Need được định nghĩa là Max – Allocation.

Ta thấy hệ thống hiện đang ở trạng thái an toàn. Thật vậy, dãy  $\langle T_1, T_3, T_4, T_2, T_0 \rangle$  thỏa mãn các tiêu chí an toàn. Giả sử bây giờ luồng  $T_1$  yêu cầu 1 thể hiện bổ sung của loại tài nguyên A và 2 thể hiện của loại tài nguyên C, vì vậy  $\text{Request}_1 = (1, 0, 2)$ . Để quyết định liệu yêu cầu này có thể được cấp ngay lập tức hay không, trước tiên ta kiểm tra xem  $\text{Request}_1 \leq \text{Available}$ , tức là  $(1, 0, 2) \leq (3, 3, 2)$ , là đúng. Sau đó, ta thử đáp ứng yêu cầu này và dẫn đến trạng thái mới sau:

Luồng	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
$T_0$	0	1	0	7	4	3	2	3	0
$T_1$	3	0	2	0	2	0			
$T_2$	3	0	2	6	0	0			
$T_3$	2	1	1	0	1	1			
$T_4$	0	0	2	4	3	1			

Ta phải xác định xem trạng thái hệ thống mới này có an toàn hay không, bằng cách thực hiện thuật toán an toàn và nhận thấy rằng chuỗi  $\langle T_1, T_3, T_4, T_0, T_2 \rangle$  thỏa mãn yêu

cầu an toàn. Do đó có thể đáp ứng ngay lập tức cấp yêu cầu của luồng  $T_1$ .

Tuy nhiên khi hệ thống ở trạng thái này,  $T_4$  yêu cầu  $(3, 3, 0)$  không thể được cấp phát, vì tài nguyên không có sẵn. Hơn nữa, yêu cầu  $(0, 2, 0)$  của  $T_0$  không thể được chấp nhận, ngay cả khi các tài nguyên có sẵn, vì trạng thái kết quả là không an toàn.

## 6.6. Phát hiện tắc nghẽn

Nếu một hệ thống không sử dụng thuật toán ngăn chặn tắc nghẽn hoặc tránh tắc nghẽn, thì tình huống tắc nghẽn có thể xảy ra. Trong môi trường này, hệ thống có thể cung cấp:

- Một thuật toán kiểm tra trạng thái của hệ thống để xác định xem có xảy ra tắc nghẽn hay không?
- Một thuật toán để khôi phục từ tắc nghẽn.

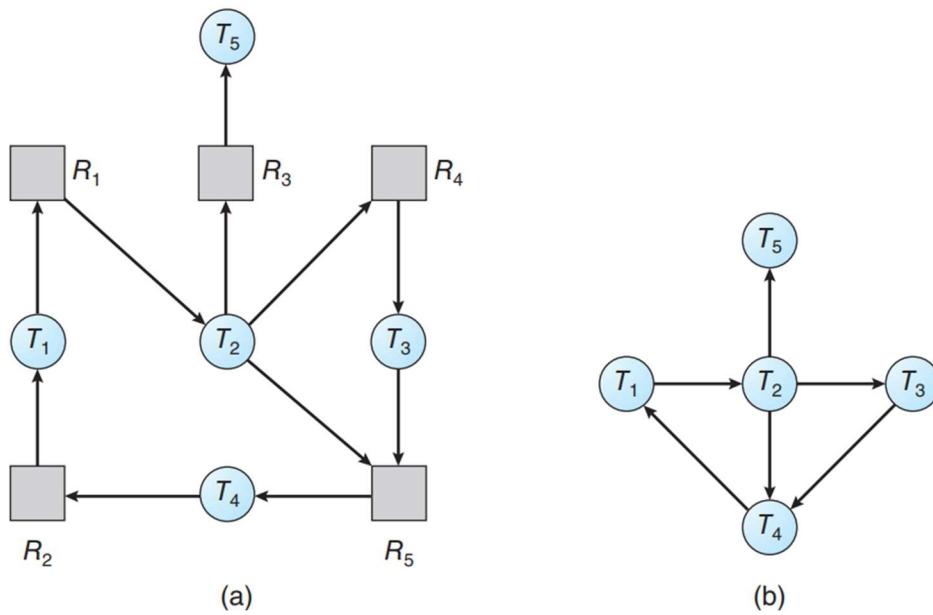
Tiếp theo, ta thảo luận về hai yêu cầu này vì chúng liên quan đến các hệ thống chỉ có một thể hiện duy nhất của mỗi loại tài nguyên, cũng như các hệ thống có nhiều thể hiện của mỗi loại tài nguyên.

### 6.6.1. Kiểu một thể hiện cho mỗi loại tài nguyên

Nếu tất cả các tài nguyên chỉ có một thể hiện duy nhất, thì có thể xác định một thuật toán phát hiện tắc nghẽn sử dụng biến thể của đồ thị phân bổ tài nguyên, được gọi là đồ thị chờ đợi. Đồ thị này nhận được từ đồ thị phân bổ tài nguyên bằng cách loại bỏ các nút tài nguyên và thu gọn các cạnh thích hợp.

Chính xác hơn, một cạnh từ  $T_i$  đến  $T_j$  trong đồ thị chờ đợi ngụ ý rằng luồng  $T_i$  đang chờ luồng  $T_j$  giải phóng một tài nguyên mà  $T_i$  cần. Một cạnh  $T_i \rightarrow T_j$  tồn tại trong đồ thị chờ khi và chỉ khi đồ thị phân bổ tài nguyên tương ứng chứa hai cạnh  $T_i \rightarrow R_q$  và  $R_q \rightarrow T_j$  đối với tài nguyên  $R_q$ . Trong Hình 6.9 trình bày đồ thị phân bổ tài nguyên và đồ thị chờ tương ứng.

Một tắc nghẽn tồn tại trong hệ thống khi và chỉ khi đồ thị chờ chứa một chu trình. Để phát hiện các tắc nghẽn, hệ thống cần duy trì đồ thị chờ và định kỳ gọi thuật toán tìm kiếm một chu trình trong đồ thị. Thuật toán phát hiện một chu trình trong đồ thị có độ phức tạp  $O(n^2)$ , trong đó  $n$  là số đỉnh trong đồ thị.



**Hình 6.9.** (a) Đồ thị cấp phát tài nguyên; (b) Đồ thị chờ tương ứng.

### 6.6.2. Kiểu nhiều thể hiện cho một loại tài nguyên

Đồ thị chờ không áp dụng được cho hệ thống phân bổ tài nguyên có nhiều thể hiện của mỗi loại tài nguyên. Nay ta chuyển sang thuật toán phát hiện tắc nghẽn có thể áp dụng cho một hệ thống như vậy. Thuật toán sử dụng một số cấu trúc dữ liệu thay đổi theo thời gian tương tự như cấu trúc được sử dụng trong thuật toán Banker:

- **Available:** Một vectơ có độ dài  $m$  cho biết số lượng tài nguyên hiện có của mỗi loại.
- **Allocation:** ma trận  $n \times m$  xác định số lượng tài nguyên của mỗi loại hiện được phân bổ cho mỗi luồng.
- **Request:** ma trận  $n \times m$  cho biết yêu cầu hiện tại của mỗi luồng. Nếu  $\text{Request}[i][j]=k$ , thì luồng  $T_i$  đang yêu cầu thêm  $k$  phiên bản của loại tài nguyên  $R_j$ .

Quan hệ  $\leq$  giữa hai vectơ được định nghĩa như trong mục 6.5.3. Để đơn giản hóa ký hiệu, ta coi các hàng trong ma trận Allocation và Request là vectơ; và gọi chúng là  $\text{Allocation}_i$  và  $\text{Request}_i$ . Thuật toán phát hiện được mô tả ở đây chỉ đơn giản là kiểm tra mọi trình tự phân bổ có thể có cho các luồng để hoàn thành công việc. So sánh thuật toán này với thuật toán Banker trong mục 6.5.3.

1. Đặt Work và Finish là vector có độ dài  $m$  và  $n$  tương ứng.

```

Work = Available
For i = 0 to n-1 do
    if Allocationi ≠ 0 then
        Finish[i] = false
    
```

```

        else
            Finish[i] = true.
2. Tìm chỉ số i thỏa hai điều kiện:
    a. Finish[i] == false
    b. Requesti ≤ Work
    Nếu không tồn tại số i, chuyển đến bước 4.
3. Work = Work + Allocationi
    Finish[i] = true
    Chuyển đến bước 2.
4. Nếu tồn tại i mà Finish[i] = false, thì hệ thống ở trạng thái tắc nghẽn. Hơn nữa nếu Finish[i] = false thì luồng Ti bị khóa.

```

Thuật toán này có độ phức tạp là  $O(m.n^2)$ .

Chú ý: ta lấy lại tài nguyên của luồng T<sub>i</sub> (ở bước 3) ngay sau khi xác định Request<sub>i</sub> ≤ Work (ở bước 2b). Vì T<sub>i</sub> hiện không tham gia vào tắc nghẽn (vì Request<sub>i</sub> ≤ Work). Vì vậy, giả định rằng T<sub>i</sub> sẽ không cần thêm tài nguyên để hoàn thành nhiệm vụ của nó; do đó nó sẽ sớm trả lại tất cả các tài nguyên hiện có. Nếu giả định này là không chính xác, có thể xảy ra tắc nghẽn sau này. Tắc nghẽn đó sẽ được phát hiện vào lần tiếp theo khi gọi thuật toán phát hiện tắc nghẽn.

Để minh họa thuật toán này, xét một hệ thống có 5 luồng từ T<sub>0</sub> đến T<sub>4</sub> và 3 loại tài nguyên A, B và C. Loại tài nguyên A có 7 thể hiện, tài nguyên B có 2 thể hiện và tài nguyên C có 6 thể hiện. Bảng sau biểu diễn cho trạng thái hiện tại của hệ thống:

Luồng	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
T <sub>0</sub>	0	1	0	0	0	0	0	0	0
T <sub>1</sub>	2	0	0	2	0	2			
T <sub>2</sub>	3	0	3	0	0	0			
T <sub>3</sub>	2	1	1	1	0	0			
T <sub>4</sub>	0	0	2	0	0	2			

Ta khẳng định hệ thống không ở trạng thái tắc nghẽn. Thật vậy, nếu thực hiện thuật toán kiểm tra trạng thái tắc nghẽn, ta nhận được chuỗi  $\langle T_0, T_2, T_3, T_1, T_4 \rangle$  cho kết quả là  $Finish[i]=true \forall i$ .

Giả sử bây giờ luồng T<sub>2</sub> thực hiện một yêu cầu bổ sung 1 thể hiện kiểu tài nguyên C. Ma trận Request được sửa đổi như sau:

Luồng	Request		
	A	B	C
T <sub>0</sub>	0	0	0

T <sub>1</sub>	2	0	2
T <sub>2</sub>	0	0	1
T <sub>3</sub>	1	0	0
T <sub>4</sub>	0	0	2

Ta khẳng định hệ thống hiện đã bị tắc nghẽn. Mặc dù có thể lấy lại các tài nguyên do luồng T<sub>0</sub> đang giữ, nhưng số lượng tài nguyên có sẵn không đủ để thực hiện các yêu cầu của các luồng khác. Do đó, tồn tại một tắc nghẽn, bao gồm các luồng T<sub>1</sub>, T<sub>2</sub>, T<sub>3</sub>, T<sub>4</sub>.

### 6.6.3. Cách sử dụng thuật toán phát hiện tắc nghẽn

Khi nào ta nên gọi thuật toán phát hiện tắc nghẽn? Câu trả lời phụ thuộc vào hai yếu tố sau:

1. Bao lâu thì một tắc nghẽn có khả năng xảy ra?
2. Có bao nhiêu luồng sẽ bị ảnh hưởng bởi tắc nghẽn khi nó xảy ra?

Nếu tắc nghẽn xảy ra thường xuyên, thì thuật toán phát hiện nên được gọi thường xuyên. Tài nguyên được phân bổ cho các luồng bị khóa sẽ không hoạt động cho đến khi tắc nghẽn bị phá vỡ. Hơn nữa, số lượng luồng tham gia vào chu trình tắc nghẽn có thể tăng lên.

Các tắc nghẽn chỉ xảy ra khi một số luồng đưa một yêu cầu không thể được cấp phát ngay lập tức. Yêu cầu này có thể là yêu cầu cuối cùng hoàn thành một chuỗi các luồng đang chờ. Do đó, ta có thể sử dụng thuật toán phát hiện tắc nghẽn mỗi khi một yêu cầu cấp phát không thể được đáp ứng ngay lập tức. Trong trường hợp này, có thể xác định không chỉ tập hợp các luồng bị khóa mà còn cả các luồng cụ thể đã "gây ra" tắc nghẽn đó. (Trong thực tế, mỗi luồng bị khóa là một liên kết chu trình trong biểu đồ tài nguyên, vì vậy tất cả chúng, cùng gây ra tắc nghẽn.) Nếu có nhiều loại tài nguyên khác nhau, một yêu cầu có thể tạo ra nhiều chu trình trong biểu đồ tài nguyên, mỗi chu trình được hoàn thành bởi yêu cầu gần đây nhất và "gây ra" bởi một luồng có thể nhận dạng.

Tất nhiên, việc sử dụng thuật toán phát hiện tắc nghẽn cho mọi yêu cầu tài nguyên sẽ phát sinh chi phí đáng kể thời gian tính toán. Một giải pháp thay thế ít tốn kém hơn là gọi thuật toán vào những khoảng thời gian xác định - ví dụ: một lần mỗi giờ hoặc bất cứ khi nào mức sử dụng CPU giảm xuống dưới 40 phần trăm. (Tắc nghẽn cuối cùng làm tê liệt thông lượng của hệ thống và khiến hiệu suất sử dụng CPU giảm xuống.) Nếu thuật toán phát hiện được gọi vào các thời điểm tùy ý, đồ thị tài nguyên có thể chứa nhiều chu trình. Trong trường hợp này thường không thể biết được luồng nào trong số nhiều luồng bị khóa đã "gây ra" sự tắc nghẽn.

## 6.7. Phục hồi trạng thái không tắc nghẽn

Khi một thuật toán phát hiện xác định có một tắc nghẽn tồn tại, một số giải pháp để khắc phục. Một khả năng là thông báo cho người vận hành biết một tắc nghẽn đã xảy ra và để người vận hành xử lý tắc nghẽn theo cách thủ công. Một khả năng khác là để hệ thống tự động khôi phục từ trạng thái tắc nghẽn. Có hai lựa chọn để phá vỡ tắc nghẽn. Một, đơn giản là hủy bỏ một hoặc nhiều luồng để phá vỡ chờ vòng tròn. Cách khác, là lấy một số tài nguyên từ một hoặc nhiều luồng bị khóa.

### 6.7.1. Kết thúc tiến trình và luồng

Để loại bỏ các tắc nghẽn bằng cách hủy bỏ một tiến trình hoặc một luồng, ta sử dụng một trong hai phương pháp. Trong cả hai phương pháp, hệ thống thu hồi tất cả các tài nguyên được phân bổ cho các tiến trình đã kết thúc.

- Hủy bỏ tất cả các tiến trình bị khóa. Phương pháp này rõ ràng sẽ phá vỡ chu trình tắc nghẽn, nhưng phải trả giá rất đắt. Các tiến trình bị khóa có thể đã được tính toán trong một thời gian dài, và kết quả của các tính toán từng phần này phải bị loại bỏ và có thể sẽ phải tính toán lại sau này.
- Hủy bỏ một tiến trình tại một thời điểm cho đến khi chu trình tắc nghẽn được loại bỏ. Phương pháp này phát sinh chi phí đáng kể, vì sau khi mỗi tiến trình bị hủy bỏ, một thuật toán phát hiện tắc nghẽn phải được sử dụng để xác định xem có bất kỳ tiến trình nào vẫn bị khóa không.

Việc hủy bỏ một tiến trình có thể không dễ dàng. Nếu tiến trình đang trong quá trình cập nhật một tệp, việc chấm dứt nó có thể khiến tệp đó ở trạng thái không chính xác. Tương tự, nếu tiến trình đang trong quá trình cập nhật dữ liệu được chia sẻ trong khi giữ khóa mutex, hệ thống phải khôi phục trạng thái của khóa như khả dụng, mặc dù không có đảm bảo nào về tính toàn vẹn của dữ liệu được chia sẻ.

Nếu phương pháp chấm dứt từng phần được sử dụng, thì chúng ta phải xác định tiến trình (hoặc các tiến trình) bị khóa nào nên được chấm dứt. Quyết định này là một quyết định chính sách, tương tự như các quyết định lập lịch CPU. Câu hỏi cơ bản là một câu hỏi kinh tế; chúng ta nên hủy bỏ những tiến trình mà việc chấm dứt sẽ phải chịu chi phí tối thiểu. Thật không may, thuật ngữ chi phí tối thiểu không phải là một thuật ngữ chính xác. Nhiều yếu tố có thể ảnh hưởng đến tiến trình được chọn, bao gồm:

1. Mức độ ưu tiên của tiến trình là gì.
2. Tiến trình đã tính toán trong bao lâu và tiến trình sẽ tính toán bao lâu trước khi hoàn thành nhiệm vụ được chỉ định.

3. Tiến trình đã sử dụng bao nhiêu và loại tài nguyên nào (ví dụ: liệu các tài nguyên đó có đơn giản không để nhường quyền ưu tiên).

4. Tiến trình cần thêm bao nhiêu tài nguyên để hoàn thành.

5. Có bao nhiêu tiến trình sẽ cần được kết thúc.

### 6.7.2. Nhường tài nguyên

Để loại bỏ các tắc nghẽn bằng cách sử dụng quyền ưu tiên tài nguyên, ta liên tiếp loại bỏ một số tài nguyên từ các tiến trình và cung cấp các tài nguyên này cho các tiến trình khác cho đến khi chu trình tắc nghẽn bị phá vỡ.

Nếu cần phải có quyền ưu tiên tài nguyên để giải quyết các tắc nghẽn, thì cần giải quyết ba vấn đề:

1. Lựa chọn nạn nhân. Những tài nguyên nào và những tiến trình nào cần nhường ưu tiên? Như trong kết thúc tiến trình, ta phải xác định thứ tự ưu tiên để giảm thiểu chi phí. Các yếu tố chi phí có thể bao gồm các thông số như số lượng tài nguyên mà một tiến trình bị tắc nghẽn đang nắm giữ và lượng thời gian mà tiến trình đã tiêu tốn cho đến nay.

2. Hoàn vốn. Nếu ta loại trừ một tài nguyên khỏi một tiến trình, thì nên làm gì với tiến trình đó? Rõ ràng, nó không thể tiếp tục với quá trình thực hiện bình thường của nó; nó đang thiếu một số tài nguyên cần thiết. Ta phải khôi phục tiến trình về trạng thái an toàn nào đó và khởi động lại từ trạng thái đó. Nói chung, rất khó để xác định trạng thái an toàn là gì, giải pháp đơn giản nhất là khôi phục hoàn toàn: hủy bỏ tiến trình và sau đó khởi động lại nó. Mặc dù sẽ hiệu quả hơn nếu chỉ quay lại tiến trình trong chừng mực cần thiết để phá vỡ tắc nghẽn, nhưng phương pháp này yêu cầu hệ thống phải giữ thêm thông tin về trạng thái của tất cả các tiến trình đang chạy.

3. Bỏ đói. Làm thế nào để ta đảm bảo rằng bỏ đói sẽ không xảy ra? Đó là, làm thế nào ta có thể đảm bảo rằng các tài nguyên sẽ không phải lúc nào cũng được nhường từ tiến trình đó? Trong một hệ thống mà việc lựa chọn nạn nhân chủ yếu dựa trên các yếu tố chi phí, có thể xảy ra tiến trình tương tự luôn được chọn làm nạn nhân. Kết quả là, tiến trình này không bao giờ hoàn thành nhiệm vụ được chỉ định của nó, một tình huống chết đói mà bất kỳ hệ thống thực tế nào cũng phải giải quyết. Rõ ràng, chúng ta phải đảm bảo rằng một tiến trình có thể được chọn làm nạn nhân chỉ một (nhỏ) số lần hữu hạn. Giải pháp phổ biến nhất là bao gồm số lần hoàn vốn trong yếu tố chi phí.

## 6.8. Tóm tắt

- Tắc nghẽn xảy ra trong một tập hợp các tiến trình khi mọi tiến trình trong tập hợp

đang chờ một sự kiện chỉ có thể do một tiến trình khác trong tập hợp gây ra.

- Có bốn điều kiện cần thiết cho tắc nghẽn: (1) loại trừ lẫn nhau, (2) giữ và chờ, (3) không có quyền ưu tiên (độc quyền), và (4) chờ vòng tròn. Chốt lại chỉ có thể xảy ra khi có đủ cả 4 điều kiện.

- Các tắc nghẽn có thể được mô hình hóa bằng các đồ thị phân bổ tài nguyên, trong đó một chu trình biểu thị tắc nghẽn.

- Có thể ngăn chặn tắc nghẽn bằng cách đảm bảo rằng một trong bốn điều kiện cần thiết để tắc nghẽn không thể xảy ra. Trong bốn điều kiện cần, loại bỏ sự chờ vòng tròn là cách tiếp cận thực tế duy nhất.

- Có thể tránh được tắc nghẽn bằng cách sử dụng thuật toán Banker, thuật toán này không cấp tài nguyên nếu như việc cấp như vậy sẽ dẫn hệ thống vào trạng thái không an toàn, nơi có thể xảy ra tắc nghẽn.

- Một thuật toán phát hiện tắc nghẽn có thể đánh giá các tiến trình và tài nguyên trên một hệ thống đang chạy để xác định xem một tập các tiến trình có ở trạng thái bị khóa hay không.

- Nếu tắc nghẽn xảy ra, hệ thống có thể cố gắng khôi phục từ tắc nghẽn bằng cách hủy bỏ một trong các tiến trình trong chờ vòng tròn hoặc lấy lại các tài nguyên đã được gán cho một tiến trình đã khóa.

## 6.9. Câu hỏi ôn tập

**Câu 6.1.** Liệt kê ba ví dụ về các trường hợp tắc nghẽn không liên quan đến môi trường hệ thống máy tính.

**Câu 6.2.** Giả sử rằng một hệ thống đang ở trạng thái không an toàn. Cho thấy rằng các luồng có thể hoàn thành quá trình thực thi của chúng mà không đi vào trạng thái bị khóa.

**Câu 6.3.** Xét một trạng thái sau của hệ thống:

Luồng	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
T <sub>0</sub>	0	0	1	2	0	0	1	2				
T <sub>1</sub>	1	0	0	0	1	7	5	0				
T <sub>2</sub>	1	3	5	4	2	3	5	6				
T <sub>3</sub>	0	6	3	2	0	6	5	2				
T <sub>4</sub>	0	0	1	4	0	6	5	6				

a. Nội dung của ma trận Need là gì ?

b. Hệ thống có ở trạng thái an toàn hay không?

- c. Giả sử luồng  $T_1$  yêu cầu  $(0, 4, 2, 0)$ , yêu cầu này có thể được đáp ứng ngay lập tức hay không?