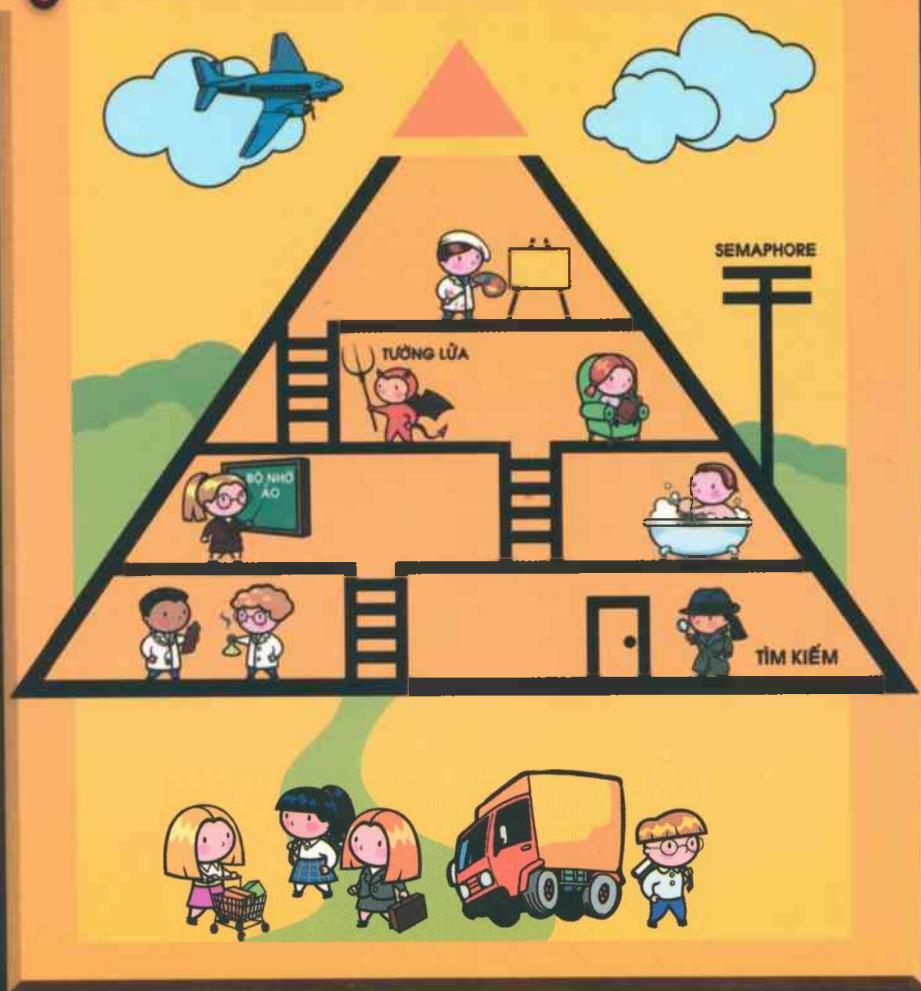


HỒ ĐẮC PHƯƠNG

GIÁO TRÌNH

NGUYÊN LÝ HỆ ĐIỀU HÀNH



NHÀ XUẤT BẢN GIÁO DỤC VIỆT NAM

HỒ ĐẮC PHƯƠNG

Giáo trình

NGUYÊN LÝ HỆ ĐIỀU HÀNH

(Tái bản lần thứ hai)

NHÀ XUẤT BẢN GIÁO DỤC VIỆT NAM

MỤC LỤC

Trang

Chương 1. GIỚI THIỆU CHUNG	7
1.1. MÁY TÍNH VÀ PHẦN MỀM.....	7
1.2. LỊCH SỬ PHÁT TRIỂN CỦA HỆ ĐIỀU HÀNH	13
CÂU HỎI ÔN TẬP	13
Chương 2. SỬ DỤNG HỆ ĐIỀU HÀNH	14
2.1. MÔ HÌNH TÍNH TOÁN TRƯỚU TƯỢNG.....	14
2.2. TÀI NGUYÊN	15
2.3. TIẾN TRÌNH	18
2.4. LUÔNG	25
2.5. ĐỘI TƯỢNG	27
2.6. NHẬN XÉT	28
CÂU HỎI ÔN TẬP	28
Chương 3. CẤU TRÚC HỆ ĐIỀU HÀNH	29
3.1. PHÂN TÍCH CÁC YÊU TỐ TÁC ĐỘNG ĐEN HỆ ĐIỀU HÀNH	29
3.2. CÁC CHỨC NĂNG CƠ BẢN	33
3.3. CÁC PHƯƠNG THỨC CÀI ĐẶT HỆ ĐIỀU HÀNH	37
3.4. NHẬN XÉT	41
CÂU HỎI ÔN TẬP	41
Chương 4. TIẾN TRÌNH	42
4.1. TIẾN TRÌNH VÀ TRANG THÁI TIẾN TRÌNH	42
4.2. THAO TÁC TRÊN TIẾN TRÌNH	45
4.3. MÔ TẢ TIẾN TRÌNH	48
4.4. LUÔNG	53
4.5. CÀI ĐẶT HỆ ĐIỀU HÀNH	57
4.6. NHẬN XÉT	59
CÂU HỎI ÔN TẬP	60
Chương 5. ĐIỀU PHÓI TIẾN TRÌNH	61
5.1. CƠ CHẾ ĐIỀU PHÓI	61
5.2. CÁC PHƯƠNG PHÁP ĐIỀU PHÓI	67
5.3. THUẬT TOÁN ĐỌC QUYỀN	71

5.4. THUẬT TOÁN KHÔNG ĐỘC QUYỀN	77
5.5. NHẬN XÉT	84
CÂU HỎI ÔN TẬP	84
Chương 6. TƯƠNG TRANH VÀ ĐÔNG BỘ.....	85
6.1. CÁC KHÁI NIỆM CƠ BẢN	85
6.2. ĐỘC QUYỀN TRUY XUẤT – GIẢI PHÁP PHẦN MỀM	88
6.3. ĐÔNG BỘ HÓA – GIẢI PHÁP PHẦN CỨNG	94
6.4. GIẢI PHÁP ĐÔNG BỘ CƠ BẢN	96
6.5. NHỮNG VẤN ĐỀ ĐÔNG BỘ KINH ĐIỀN	101
6.6. CÁC GIẢI PHÁP ĐÔNG BỘ CAO CẤP	104
6.7. CƠ CHẾ IPC	109
6.8. NHẬN XÉT	113
CÂU HỎI ÔN TẬP	114
Chương 7. BÊ TẮC	115
7.1. MÔ HÌNH HỆ THỐNG	115
7.2. ĐẶC ĐIỂM CỦA BÊ TẮC	116
7.3. NGĂN CHẶN BÊ TẮC	119
7.4. TRÁNH BÊ TẮC	121
7.5. PHÁT HIỆN BÊ TẮC	126
7.6. KHẮC PHỤC BÊ TẮC	129
7.7. NHẬN XÉT	130
CÂU HỎI ÔN TẬP	131
Chương 8. QUẢN LÝ THIẾT BỊ.....	132
8.1. NGUYÊN LÝ HOẠT ĐỘNG	132
8.2. CHIẾN LƯỢC QUẢN LÝ THIẾT BỊ	134
8.3. TRÌNH ĐIỀU KHIỂN THIẾT BỊ	150
CÂU HỎI ÔN TẬP	152
Chương 9. QUẢN LÝ BỘ NHỚ	153
9.1. CÁC LOẠI ĐỊA CHỈ	153
9.2. KHÔNG GIAN ĐỊA CHỈ	157
9.3. HOÀN CHUYỀN	158
9.4. CẤP PHÁT LIỀN TỤC	159
9.5. PHẦN TRANG	163
9.6. PHẦN ĐOẠN	173
9.7. KẾT HỢP PHẦN ĐOẠN VỚI PHẦN TRANG	178
9.8. NHẬN XÉT	180
CÂU HỎI ÔN TẬP	181

Chương 10. BỘ NHỚ ẢO	182
10.1. ĐẶT VÂN ĐÉ	182
10.2. PHÂN TRANG THEO YÊU CẦU	183
10.3. HIỆU SUẤT PHÂN TRANG THEO YÊU CẦU	186
10.4. THAY THẾ TRANG	188
10.5. THUẬT TOÁN THAY THẾ TRANG	190
10.6. CẤP PHÁT FRAME	197
10.7. PHÂN ĐOẠN THEO YÊU CẦU	200
10.8. NHẬN XÉT	201
CÂU HỎI ÔN TẬP	201
Chương 11. HỆ THỐNG FILE	202
11.1. FILE	202
11.2. CÀI ĐẶT FILE Ở MỨC THẤP	209
11.3. HỆ THỐNG THƯ MỤC	219
11.4. BẢO VỆ FILE CHIA SẺ	226
11.5. TÍNH THÔNG NHẤT CỦA NGỮ NGHĨA	229
11.6. PHỤC HỒI SAU LỖI	231
11.7. NHẬN XÉT	232
CÂU HỎI ÔN TẬP	233
Chương 12. BẢO VỆ VÀ AN NINH	234
12.1. CÁC VÂN ĐÉ CƠ BẢN	234
12.2. XÁC THỰC	238
12.3. KIÉM CHỨNG	241
12.4. CÀI ĐẶT MA TRẦN QUYỀN TRUY CẤP	249
12.5. HẬU QUẢ TỪ CHƯƠNG TRÌNH	255
12.6. GIÁM SÁT NGUY CƠ	262
12.7. MẬT MÃ VÀ ỨNG DỤNG	264
12.8. NHẬN XÉT	268
CÂU HỎI ÔN TẬP	269
TÀI LIỆU THAM KHẢO.....	270

Chương 1

GIỚI THIỆU CHUNG

Chương này khái quát về Hệ điều hành (HĐH) và lịch sử phát triển HĐH. Trước tiên, giới thiệu tổng quan khái niệm phần mềm và vị trí cụ thể của HĐH trong hệ thống phần mềm. Tiếp đến, trình bày hai động lực phát triển của HĐH hiện đại là: Trưu tượng hóa và Chia sẻ tài nguyên phần cứng.

1.1. MÁY TÍNH VÀ PHẦN MỀM

Con người sử dụng máy tính thông qua phần mềm. Phần mềm được phân loại theo mục đích sử dụng: **Phần mềm ứng dụng** giải quyết vấn đề cụ thể (MS Word, MS Excel). **Phần mềm hệ thống** thực thi những nhiệm vụ liên quan tới quá trình thực thi các chương trình ứng dụng. Bên cạnh đó, phần mềm hệ thống cung cấp những chức năng mà phần cứng không thể cung cấp, giúp lập trình viên phát triển ứng dụng,... HĐH là phần mềm hệ thống quan trọng nhất.

Mục tiêu quan trọng của HĐH là cho phép nhiều phần mềm ứng dụng cùng nhau sử dụng phần cứng máy tính một cách có trật tự. Chia sẻ làm tăng hiệu suất sử dụng hệ thống, vì các chương trình khác nhau đồng thời sử dụng những bộ phận phần cứng khác nhau. Do đó, giảm thời gian cần thiết để thực hiện một nhóm chương trình. Để chia sẻ an toàn và có hiệu quả, HĐH phải nắm sát phần cứng. Phần mềm hệ thống và tất cả các phần mềm ứng dụng gián tiếp sử dụng phần cứng thông qua HĐH.

1.1.1. Phần mềm hệ thống

Với người lập trình ứng dụng, phần mềm hệ thống đơn giản hóa môi trường lập trình và cho phép sử dụng hiệu quả phần cứng. Phần mềm hệ thống có chức năng làm môi trường thực thi cho ngôn ngữ lập trình. Trong

UNIX, chức năng này được cài đặt bằng ngôn ngữ C (bằng cách sử dụng các file.h), ví dụ:

- Thư viện vào/ra chuẩn (I/O) thực hiện các thao tác vào/ra thông qua bộ đệm trên dòng dữ liệu.
- Thư viện toán học để tính toán các hàm toán học.
- Thư viện đồ họa cung cấp hàm hiển thị hình ảnh trên màn hình đồ họa.

Một chức năng khác của phần mềm hệ thống là cung cấp hệ thống giao diện cửa sổ. WINDOWS là phần mềm hệ thống cung cấp các cửa sổ (một thiết bị đầu cuối ảo) cho chương trình ứng dụng. Lập trình viên phát triển phần mềm ứng dụng sử dụng những hàm để đọc và ghi lên cửa sổ như thẻ cửa sổ là một thiết bị đầu cuối, thậm chí cửa sổ này không gắn với bất kỳ thiết bị vật lý nào. Phần mềm hệ thống chịu trách nhiệm ánh xạ thiết bị đầu cuối ảo lên một vùng cụ thể trên màn hình. Một thiết bị đầu cuối vật lý có thể hỗ trợ nhiều thiết bị đầu cuối ảo.

HĐH cung cấp giao diện (là các hàm) để phần mềm hệ thống và phần mềm ứng dụng sử dụng khi muốn dùng tài nguyên hệ thống. HĐH là phần mềm độc lập, hỗ trợ nhiều ứng dụng trong các lĩnh vực khác nhau. Phần mềm ứng dụng sử dụng sự trừu tượng hóa tài nguyên do HĐH cung cấp khi làm việc với phần cứng. HĐH cho phép các ứng dụng khác nhau chia sẻ tài nguyên phần cứng thông qua chính sách quản lý tài nguyên. Trừu tượng hóa tài nguyên và chia sẻ là hai khía cạnh cơ bản của HĐH.

1.1.2. Trừu tượng hóa tài nguyên phần cứng

Bằng cách che dấu chi tiết hoạt động phần cứng thông qua mô hình trừu tượng hoạt động của phần cứng, phần mềm hệ thống giúp lập trình viên sử dụng phần cứng dễ dàng hơn. Mặc dù giúp đơn giản hóa cách thức điều khiển phần cứng, mô hình trừu tượng cũng giới hạn khả năng lập trình viên thao tác trực tiếp trên phần cứng vì có những thao tác phần cứng không thể trừu tượng hóa được. Có thể coi máy rút tiền tự động ATM là sự trừu tượng hóa việc rút tiền ở ngân hàng. ATM có thể cung cấp một thao tác trừu tượng cho phép người dùng rút 1 triệu hay 2 triệu đồng từ tài khoản chi thông qua án một nút duy nhất. Tuy nhiên, nếu người dùng muốn rút đúng 1,3 triệu

đồng thì phải nhấn một số nút khác nhau: đầu tiên ấn nút rút tiền, sau đó rút từ tài khoản với lượng tiền sẽ rút là 1,3 triệu đồng.

Phần cứng có thể coi là Tài nguyên hệ thống và bất kỳ tài nguyên cụ thể nào đều có giao diện riêng định nghĩa các thao tác mà lập trình viên có thể thực hiện trên tài nguyên. Tuy nhiên, phần mềm hệ thống vẫn có thể tiếp tục trừu tượng hơn nữa để đơn giản hóa giao diện tài nguyên cụ thể. Để sử dụng tài nguyên, người lập trình không nhất thiết phải biết giao diện cụ thể của tài nguyên, mà chỉ cần biết giao diện trừu tượng (bỏ qua hoạt động chi tiết của thiết bị). Do đó, lập trình viên có thể tập trung vào các vấn đề ở mức cao hơn. Các tài nguyên giống nhau có thể được trừu tượng thành một giao diện thống nhất. Ví dụ, phần mềm hệ thống có thể trừu tượng hoạt động ổ đĩa mềm và ổ đĩa cứng thành giao diện ổ đĩa trừu tượng. Người lập trình chỉ cần có kiến thức chung nhất về hoạt động của ổ đĩa trừu tượng, mà không cần biết chi tiết các thao tác vào/ra trên ổ đĩa cứng hay ổ đĩa mềm.

Giả sử phải phát triển ứng dụng phân tích xu thế đầu tư của thị trường chứng khoán. Việc thiết kế và chỉnh sửa đoạn mã thực hiện việc đọc/ghi thông tin trên ổ đĩa chiếm một phần không nhỏ trong toàn bộ đoạn mã chương trình. Kỹ năng cần thiết để viết phần mềm điều khiển ổ đĩa khác kỹ năng phân tích thị trường chứng khoán. Nếu có kiến thức chung về hoạt động của ổ đĩa, lập trình viên ứng dụng không cần quan tâm thao tác vào/ra của ổ đĩa cứng. Trừu tượng tài nguyên là cách tiếp cận tối ưu, vì người lập trình ứng dụng sử dụng mô hình trừu tượng để thực hiện việc đọc/ghi ổ đĩa. Phần mềm điều khiển ổ đĩa là ví dụ về phần mềm hệ thống. Lập trình viên có thể tập trung vào các vấn đề của ứng dụng, chứ không cần quan tâm đến những thứ không liên quan. Nói cách khác, phần mềm hệ thống "trong suốt" với người sử dụng, nhưng rất quan trọng với lập trình viên.

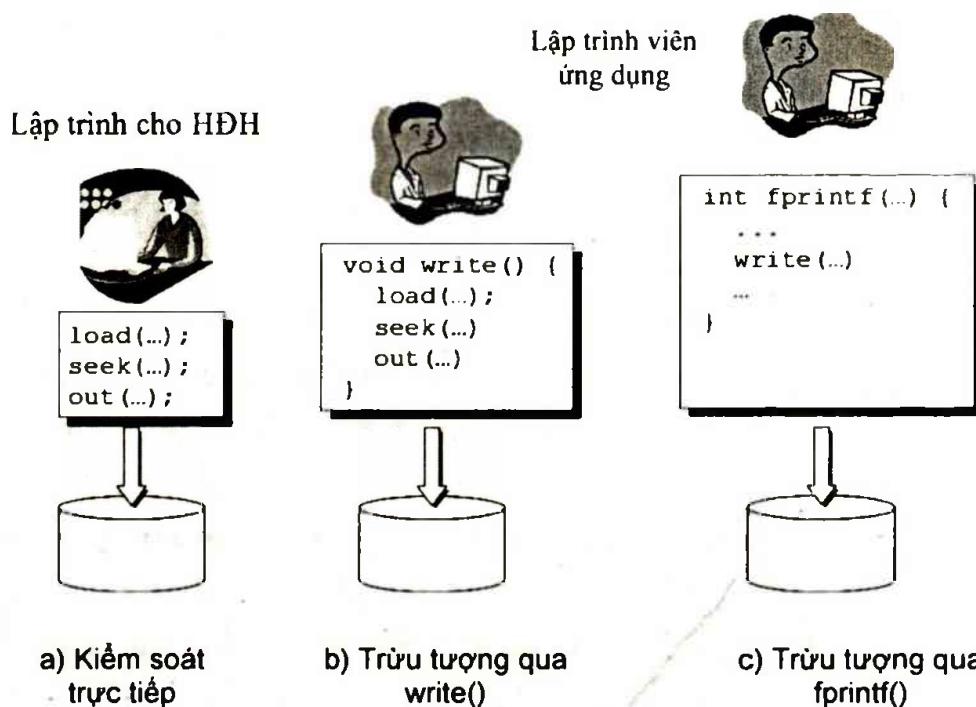
1.1.3. Ví dụ về trừu tượng hóa thiết bị ổ đĩa

Ý tưởng trừu tượng hóa tài nguyên có thể được hiểu rõ thông qua ví dụ hoạt động thiết bị ổ đĩa (Hình 1.1). Phần mềm điều khiển thiết bị sao chép một khối thông tin từ bộ nhớ chính tới bộ nhớ đệm của thiết bị bằng chỉ thị **Load(block, length, device)**. Để chuyển đầu đọc/ghi tới một vị trí cụ thể trên bề mặt đĩa sử dụng chỉ thị **Seek(device, track)**. Thao tác ghi một khối dữ liệu từ vùng đệm vào thiết bị là **out(device, sector)**.

Do đó, cần một nhóm lệnh liên tiếp để ghi khôi thông tin từ bộ nhớ chính ra ổ đĩa như Hình 1.1a. Để đơn giản công việc của người phát triển ứng dụng, hệ thống đóng gói những câu lệnh trong Hình 1.1a vào một thủ tục. Hàm **write** (Hình 1.1b), tạo thành một mức trùu tượng hóa cao hơn.

Bước trùu tượng cao hơn ở phần mềm hệ thống cho phép xem ổ đĩa là nơi lưu trữ file. Phần mềm hệ thống sử dụng định danh file (fileID) như một mức trùu tượng ổ đĩa. Khi đó, thư viện (*stdio* của C) có thể cung cấp hàm để viết biến nguyên *datum* vào thiết bị tại một vị trí nào đó trong file bằng hàm **fprintf(fileID, "%d", datum)**.

Mức trùu tượng này có thể được sử dụng cho các thao tác đọc/ghi bằng từ nếu phần mềm hệ thống triển khai sự trùu tượng đó cho thiết bị băng từ.



Hình 1.1. Trùu tượng hóa ổ đĩa cứng

Quá trình trùu tượng hóa có thể diễn ra ở nhiều cấp độ. Phần cứng được điều khiển thông qua một giao diện, phần mềm hệ thống ở mức cao có thể tiếp tục trùu tượng hóa tài nguyên này thông qua một mức giao diện cao hơn. Ví dụ trình bày ở trên minh họa rất rõ điều này.

1.1.4. Chia sẻ tài nguyên phần cứng

Tài nguyên trùu tượng và tài nguyên vật lý có thể được nhiều tiến trình đang thực thi đồng thời dùng chung (khái niệm tiến trình – sự thực thi của một chương trình được đề cập trong Chương 2). Có hai kiểu chia sẻ là theo

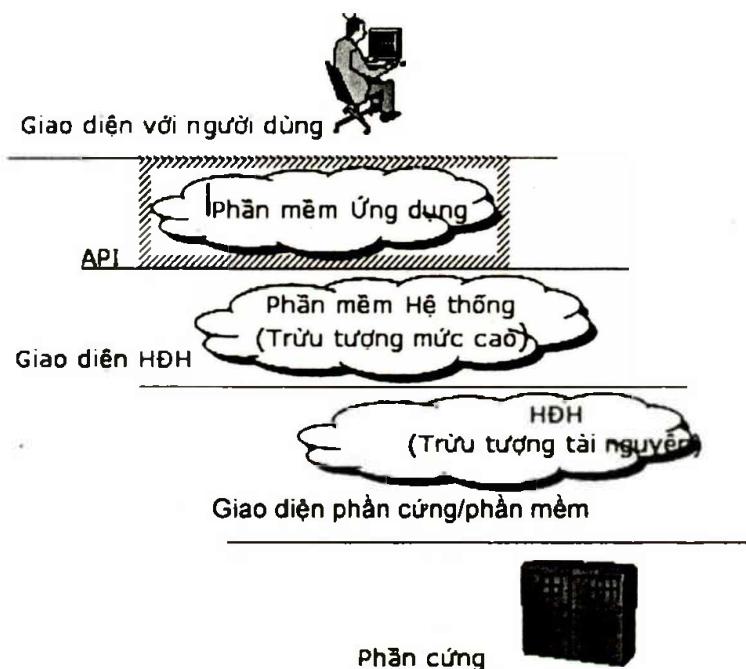
không gian và theo thời gian. Trong chia sẻ theo không gian, tài nguyên được chia ra thành nhiều đơn vị riêng biệt. Các tiến trình đồng thời được cấp phát các đơn vị tài nguyên khác nhau. Đối với các tiến trình, bộ nhớ hay ô đĩa là tài nguyên phân chia theo không gian.

Trong phương thức chia sẻ theo thời gian, tài nguyên không bị chia nhỏ mà được cấp phát trọn vẹn cho một tiến trình trong một khoảng thời gian. Sau đó, tài nguyên bị thu hồi và cấp phát cho tiến trình khác. Ví dụ, tài nguyên theo kiểu này là bộ vi xử lý. Bộ vi xử lý được điều phối giữa nhiều tiến trình đang nắm giữ các tài nguyên khác nhau. Máy tính chỉ có duy nhất một bộ vi xử lý, nhưng người dùng có cảm giác nhiều chương trình khác nhau cùng được thực hiện.

Có hai khía cạnh quan trọng khi chia sẻ tài nguyên. *Thứ nhất*, hệ thống phải có khả năng kiểm soát việc truy cập tài nguyên qua chính sách cấp phát. *Thứ hai*, hệ thống phải có khả năng cài đặt các tài nguyên chia sẻ khi cần thiết. Để ngăn cản các truy cập không hợp lệ, HĐH phải kiểm chứng tiến trình muốn sử dụng tài nguyên có hợp lệ hay không. Cài đặt tài nguyên là khả năng HĐH ngăn cản tiến trình truy cập trái phép đến tài nguyên đã được cấp phát cho tiến trình khác. Cơ chế cài đặt bộ nhớ cho phép hai chương trình được tải đồng thời vào những phần khác nhau của bộ nhớ. Cơ chế cài đặt bộ vi xử lý buộc các tiến trình chia sẻ bộ vi xử lý của hệ thống một cách tuần tự. Không tiến trình nào có thể thay đổi hay tham chiếu đến nội dung bộ nhớ được cấp phát cho tiến trình khác.

Tuy nhiên, phần mềm hệ thống phải cho phép các chương trình đang thực thi có thể chia sẻ quyền truy cập tài nguyên khi cần thiết. Giải quyết vấn đề cài đặt tài nguyên phát sinh vấn đề mới. Giả sử người lập trình có ý định cho phép hai chương trình đang chạy chia sẻ tài nguyên (bên cạnh bộ xử lý). HĐH phải đảm bảo cơ chế cài đặt, nhưng không được ngăn cản việc truy cập các tài nguyên chia sẻ. Nếu phần mềm hệ thống không cài đặt chính xác cơ chế cài đặt tài nguyên, thì không thể đảm bảo cài đặt được tài nguyên. Đến lượt mình, phần mềm hệ thống phải có bộ phận đáng tin cậy triển khai việc cài đặt tài nguyên, sao cho chương trình ứng dụng không thể vi phạm cơ chế. Bộ phận đáng tin cậy này được cài đặt trong HĐH. Thậm chí phần mềm HĐH phải phụ thuộc vào phần cứng để thực hiện các phần quan trọng nhất của cơ chế cài đặt tài nguyên. Các cơ chế chia sẻ tài nguyên trừu tượng thường được cài đặt tại phần mềm hệ thống ở tầng cao, nhưng các cơ chế này phụ thuộc vào thao tác đã được kiểm chứng là đáng tin cậy của HĐH.

Hình 1.2 minh họa sự khác biệt giữa phần mềm hệ thống và HĐH. *Thứ nhất*, phần mềm hệ thống cài đặt mô hình trừu tượng các tài nguyên mà lập trình viên có thể sử dụng, HĐH trực tiếp thực hiện trừu tượng hóa các tài nguyên vật lý. *Thứ hai*, HĐH cung cấp các thao tác cơ sở hoàn toàn đáng tin cậy để quản lý việc chia sẻ tài nguyên. Hình 1.2 minh họa một số giao diện cơ bản giữa những thành phần khác nhau trong hệ thống. Phần mềm ứng dụng sử dụng giao diện lập trình ứng dụng với phần mềm hệ thống, phần mềm hệ thống sử dụng giao diện với HĐH và HĐH sử dụng giao diện phần mềm/phần cứng để tương tác với phần cứng (hệ thống phân cấp trong hình vẽ chỉ mang tính tương đối, ví dụ chương trình ứng dụng hoàn toàn có thể thi hành trực tiếp một số chỉ thị phần cứng).



Hình 1.2. Phần mềm và HĐH

1.1.5. Các máy tính không có phần mềm hệ thống

Với các máy tính cá nhân thời kỳ đầu tiên, có thể viết phần mềm ứng dụng mà không cần trừu tượng hóa hoặc chia sẻ tài nguyên, nên cũng không cần đến phần mềm hệ thống. Các thiết bị đơn giản đến mức không cần trừu tượng hóa tài nguyên và không có nhu cầu hỗ trợ chạy đồng thời nhiều chương trình. Chương trình ứng dụng chịu trách nhiệm thực hiện tất cả các thao tác vào/ra (bàn phím, màn hình, ổ đĩa). Theo thời gian, các thiết bị vào/ra ngày càng đa dạng và phức tạp; phần mềm điều khiển những thiết bị như vậy dần trở nên quá phức tạp đối với đa phần người lập trình ứng dụng.

Các hãng chế tạo máy tính cá nhân bắt đầu đưa cơ chế trừu tượng tài nguyên vào ROM (ví dụ các thủ tục cơ sở BIOS trong máy tính cá nhân IBM).

Sức mạnh ngày càng tăng của máy tính cá nhân dẫn đến nhu cầu thực thi đồng thời nhiều tiến trình, điều này đòi hỏi cơ chế chia sẻ CPU. Kết quả là phần mềm hệ thống của máy tính cá nhân hiện nay cũng triển khai cơ chế chia sẻ tài nguyên. Với các máy tính tương thích IBM, bước đi này dẫn đến sự ra đời của nhiều HĐH như Microsoft Windows thay thế cho MS-DOS.

1.2. LỊCH SỬ PHÁT TRIỂN CỦA HỆ ĐIỀU HÀNH

Theo thống kê của Từ điển Wikipedia, hiện nay có khoảng 80 HĐH viết cho máy tính, chưa kể đến các thiết bị khác. Như đã nói, HĐH là chương trình quản lý tài nguyên phần cứng, nhằm tạo ra môi trường cho phép người sử dụng thực thi các chương trình ứng dụng. Ngày nay, một chiếc điện thoại di động cũng cần HĐH (như Symbian OS, Windows CE,...), thậm chí người ta còn viết ra những phiên bản Linux cho các thiết bị giải trí số như Xbox, Play Station,... Tuy nhiên, cách đây khoảng nửa thế kỷ, khái niệm HĐH còn chưa ra đời. Một chiếc máy tính không lồ của một viện nghiên cứu cùng với rất nhiều các chương trình tính toán phức tạp (của thời đó) được điều phối không phải bởi HĐH, mà bởi "người điều hành" (operator)! Chính sự phát triển của tốc độ xử lý, dung lượng bộ nhớ cùng yêu cầu thực thi chương trình ngày càng phức tạp, đã đặt ra nhu cầu cần có một chương trình tự động điều phối các tài nguyên máy tính phần cứng cũng như phần mềm – từ đó mà HĐH ra đời.

Chặng đường phát triển của HĐH gắn liền với sự tiến hóa của phần cứng máy tính, vì HĐH là phần mềm mức thấp, phụ thuộc rất nhiều vào kiến trúc máy tính. Có thể thấy mỗi bước tiến của công nghệ chế tạo máy tính lại tạo ra một bước đột phá mới cho HĐH.

CÂU HỎI ÔN TẬP

1. Trình bày sự khác biệt giữa phần mềm hệ thống và phần mềm ứng dụng.
2. Khái niệm trừu tượng có ưu điểm gì?
3. Tại sao phải có HĐH nằm giữa hệ thống phần mềm và phần cứng?

Chương 2

SỬ DỤNG HỆ ĐIỀU HÀNH

Chương này trình bày các thành phần cơ bản trong môi trường lập trình do HĐH cung cấp theo quan điểm của lập trình viên và nhà thiết kế hệ thống. Trước tiên, tìm hiểu mô hình máy tính được sử dụng trong HĐH hiện đại thông qua việc mô tả các tài nguyên nói chung và tài nguyên file nói riêng. Tiếp theo, tìm hiểu về quá trình hình thành tiến trình, bao gồm các ví dụ trong HĐH UNIX. Phần cuối chương giới thiệu hai mô hình luồng (thread) và đối tượng (object).

2.1. MÔ HÌNH TÍNH TOÁN TRƯU TỰNG

Người lập trình ứng dụng quan niệm máy tính là thiết bị có thể truy cập, biến đổi và lưu trữ thông tin. HĐH cung cấp môi trường giúp người lập trình định nghĩa các thao tác xử lý thông tin cơ bản nhất thông qua những khái niệm về đơn vị thực thi chương trình cũng như các thành phần cần thiết trong quá trình tính toán. Trong HĐH hiện đại, đơn vị tính toán nhỏ nhất là **Tiến trình** (*process*) và đơn vị lưu trữ thông tin nhỏ nhất là **Tập tin** (*file*). Các thành phần hệ thống khác có thể là tài nguyên được sử dụng trong quá trình tính toán. Chương trình định rõ hành vi của một hay nhiều tiến trình bằng cách định nghĩa tường minh cách đọc thông tin từ file, phương pháp biến đổi thông tin bằng cách sử dụng tài nguyên hệ thống, sau đó lưu trữ thông tin ra file. Thiết bị lưu trữ thường là nơi lưu trữ file, vì thế chương trình tương tác với thông tin thông qua giao diện file. Các tài nguyên khác có giao diện riêng, được nhà thiết kế HĐH đưa ra để định nghĩa mức độ trùu tượng tài nguyên. Ví dụ, thiết bị hiển thị ảnh nhị phân có giao diện được xây dựng xung quanh khôi bộ nhớ hiển thị. Chương trình ứng dụng sẽ sao chép thông tin lên các khôi bộ nhớ đặc biệt và hình ảnh tương ứng sẽ được hiển

thị trên màn hình. Phần mềm hệ thống có thể cung cấp thêm nhiều mức độ lưu trữ tượng hóa thông tin như file chi số, cơ sở dữ liệu. HDH cung cấp cửa sổ làm việc cho các thiết bị hiển thị đầu/cuối cũng như tiến trình là đơn vị tính toán cơ sở. Tuy nhiên, với phần lớn HDH, tiến trình và file là hai giao diện cơ bản và quan trọng nhất.

2.2. TÀI NGUYÊN

Thực thể được coi là tài nguyên nếu thỏa mãn cả hai yêu cầu sau:

- Tiến trình phải yêu cầu thực thể từ HDH.
- Tiến trình tạm thời ngừng hoạt động đến khi thực thể yêu cầu được cấp phát.

2.2.1. File

File có thể được xem là luồng byte nằm trên thiết bị lưu trữ ngoài và được xác định qua tên gọi. Thông tin được lưu trữ bằng cách mở file (tạo ra một bản mô tả file với tên xác định), sau đó ghi từng khối byte lên. Tương tự, có thể truy cập thông tin lưu trong file bằng cách mở file rồi đọc từng khối byte. HDH chịu trách nhiệm cài đặt hệ thống file trên thiết bị lưu trữ cố định như ô đĩa, bằng cách ánh xạ luồng byte vào các khối lưu trữ trên thiết bị. File được phân biệt với các tài nguyên khác vì:

1. File là hình thức lưu trữ thông tin phổ biến nhất trong máy tính.
2. HDH thường lấy file làm cơ sở để mô hình hóa các tài nguyên khác.

File có bản mô tả cho phép HDH ghi lại hiện trạng sử dụng file, tình trạng từng file, các thông tin về việc ánh xạ luồng byte trong file tới vị trí cụ thể trên thiết bị lưu trữ.

Hệ thống file theo chuẩn POSIX

File POSIX là tập hợp tuân tự các byte có định danh. BSD (Berkeley Software Distribution) UNIX có giao diện file khác. Giao diện hệ thống file POSIX chỉ cung cấp một vài thao tác file cơ bản (Bảng 2.1).

Chương trình hoàn chỉnh sau minh họa cách sử dụng file qua giao diện POSIX. Chương trình này sao chép từng ký tự từ file có tên **in_test** tới file **out_test** bằng cách mở file **in_test** để đọc từng byte và sau đó ghi vào file **out_test**.

```

#include<stdio.h>
#include<fcntl.h>
int main()
{
    int inFile outFile;
    char *inFileName = "in_test";
    char *outFileName = "out_test";
    int len;
    char c;
    inFile = open(inFileName, O_RDONLY);
    outFile = open(outFileName, O_WRONLY);
    while (len = read((inFile, &c, 1) > 0)) // Vòng đọc đọc toàn bộ file
        write(outFile, &c, 1);
    close(inFile); // Đóng file
    close(outFile);
}

```

Bảng 2.1

Lời gọi hệ thống	Miêu tả
open	Lời gọi open có tham số là tên file (kể cả đường dẫn) mà chúng ta chuẩn bị đọc hoặc ghi thông tin. Lời gọi này có thể có thêm tham số xác định chế độ truy cập file (tức là file mở theo chế độ chỉ đọc hay đọc/ghi). Khi file được mở, con trỏ file trỏ vào byte đầu tiên trong luồng byte (nếu file rỗng, vị trí này là vị trí để ghi byte đầu tiên). Nếu thành công, lời gọi trả về một giá trị nguyên không âm, được gọi là thẻ file. Sau đó người sử dụng sẽ sử dụng thẻ file khi tham chiếu tới file.
close	Lời gọi close là đóng file, sau đó giải phóng các tài nguyên hệ thống sử dụng để mô tả trạng thái file.
read	Lời gọi read có các tham số là thẻ file (giá trị được open trả về), địa chỉ và kích thước bộ đệm. Thông thường, lời gọi này khiến tiến trình gọi bị phong tỏa cho tới khi hoàn thành quá trình đọc. Tuy nhiên, ngữ nghĩa này có thể thay đổi với một hàm thích hợp trong hàm fcntl.
write	Lời gọi write tương tự như read nhưng write được sử dụng để ghi thông tin lên file.
lseek	Lời gọi lseek di chuyển con trỏ đọc/ghi trong luồng byte tới vị trí xác định. Sự di chuyển này ảnh hưởng đến các lệnh đọc và ghi tiếp sau.
fcntl	Lời gọi fcntl (viết tắt của file control) cho phép gửi một yêu cầu điều khiển bất kỳ tới HĐH. Ví dụ, thao tác đọc file thông thường phong tỏa tiến trình đang được gọi nếu nó thực hiện đọc một file trống; khi sử dụng fcntl, thao tác đọc file có thể trả lại người gọi nếu hành động cố gắng đọc file phong tỏa tiến trình đang gọi.

2.2.2. Các tài nguyên khác

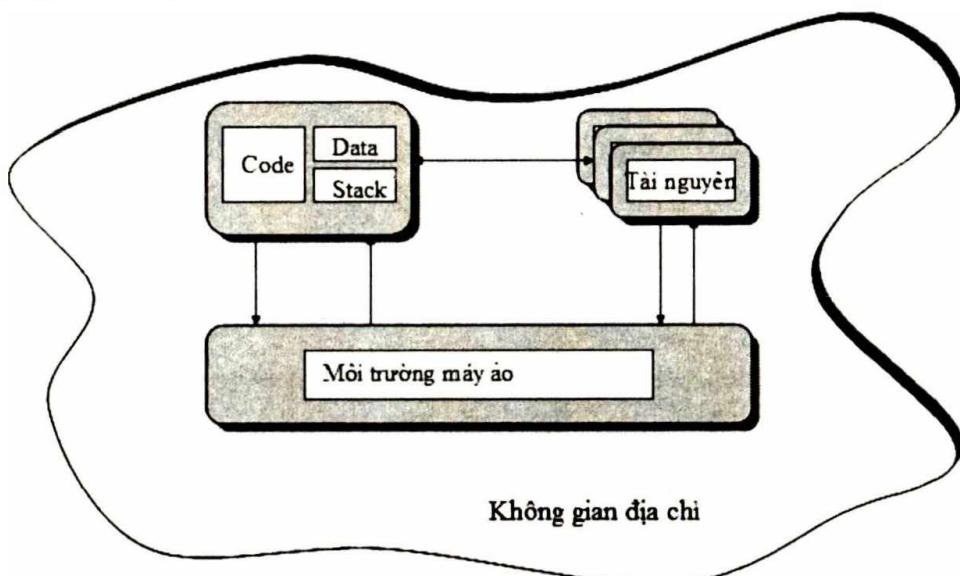
Tài nguyên phần cứng là thành phần trừu tượng bất kỳ mà chương trình cần có trước khi thực thi. Nếu yêu cầu tài nguyên mà chưa được đáp ứng thì tiến trình không thể tiếp tục thực thi mà sẽ bị phong tỏa cho đến khi có đủ tài nguyên cần thiết. CPU là tài nguyên mà bất kỳ tiến trình nào cũng cần phải có nếu muốn thực thi. Ít khi tiến trình yêu cầu cấp phát tài nguyên CPU một cách tường minh, nhưng để thực thi, nhất thiết phải có CPU. Cấp phát tài nguyên CPU cho các tiến trình là chức năng quan trọng của HĐH. Bộ nhớ cũng được xem là tài nguyên. Chương trình có thể yêu cầu tham chiếu đến khu vực bộ nhớ cần thiết trong khi thực thi hoặc đưa ra các yêu cầu cấp phát bộ nhớ động. Ổ đĩa cứng cũng là tài nguyên, vì lập trình viên muốn đọc thông tin từ ổ đĩa thì trước khi đọc dữ liệu, tiến trình phải được cấp phát quyền sử dụng ổ đĩa. HĐH có giao diện hỗ trợ chương trình truy cập một số kiểu tài nguyên khác, chẳng hạn CPU, bộ nhớ, bàn phím và màn hình hiển thị. Trong trường hợp giao diện với tất cả kiểu tài nguyên giống nhau, lập trình viên sử dụng tài nguyên dễ dàng hơn nhiều so với trường hợp các kiểu tài nguyên có giao diện khác nhau. UNIX áp dụng phương pháp này. Nói chung người thiết kế HĐH cố gắng cài đặt giao diện tài nguyên tương tự giao diện file, mặc dù điều này chưa chắc thực hiện được trong một số trường hợp.

Bộ phận quản lý bộ nhớ của HĐH UNIX cấp phát bộ nhớ cho tiến trình căn cứ trên nhu cầu bộ nhớ của chương trình. Có nhiều phương pháp quản lý bộ nhớ. Một vài phiên bản HĐH UNIX còn có cơ chế hoán chuyển: khi xuất hiện nhiều yêu cầu cấp phát CPU hoặc cấp phát bộ nhớ (do nhiều tiến trình khác nhau yêu cầu) thì HĐH sẽ thu hồi vùng nhớ của một vài tiến trình (trạng thái của tiến trình cũng như hình ảnh tiến trình trong bộ nhớ sẽ được chuyển ra lưu tạm trên thiết bị lưu trữ ngoài). Các thao tác này "trong suốt" với tiến trình, nhưng thỉnh thoảng người dùng vẫn có thể thấy khi tốc độ hệ thống suy giảm. Trong HĐH UNIX, cơ chế trừu tượng hóa cũng được áp dụng cho thiết bị. Giao diện với thiết bị cũng có các lời gọi `open`, `close`, `read`, `write`, `lseek` và `ioctl` giống như giao diện file. Các thao tác `read/write` thao tác trên luồng byte, vì thế thao tác đọc từ thiết bị cũng giống thao tác đọc file.

2.3. TIẾN TRÌNH

Tiến trình là chương trình đang được thi hành tuần tự. Tiến trình (minh họa trên Hình 2.1) bao gồm các thành phần sau:

- Đoạn mã cần thực thi.
- Dữ liệu để chương trình thực hiện các phép biến đổi.
- Tài nguyên cần thiết để thực thi chương trình.
- Trạng thái thực thi của tiến trình.



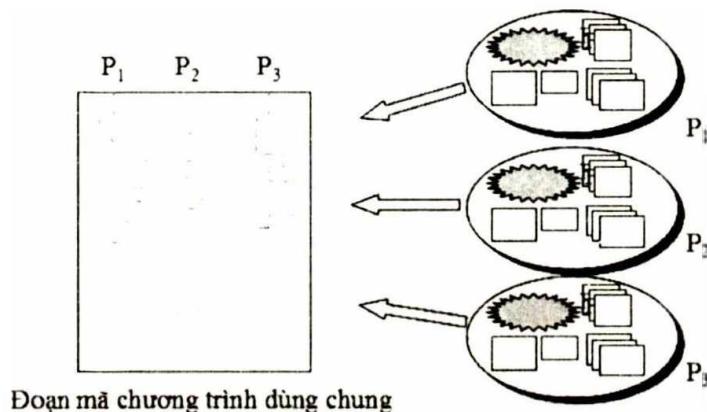
Hình 2.1. Môi trường của tiến trình

Để tiến trình thực thi cần có môi trường máy trừu tượng quản lý việc chia sẻ, cô lập tài nguyên giữa nhiều tiến trình. Trạng thái tiến trình được dùng để ánh xạ trạng thái của môi trường vào trạng thái vật lý của máy tính. Ví dụ về trạng thái tiến trình là chỉ thị nào trong đoạn mã chương trình hiện thời đang được thực hiện. Hình 2.1 minh họa môi trường máy trừu tượng quản lý các tiến trình và tài nguyên, cấp phát tài nguyên cho tiến trình khi có yêu cầu.

Cần phân biệt khái niệm *Chương trình* và *Tiến trình*. Chương trình là đối tượng tĩnh, cấu thành bởi các dòng lệnh, xác định hành vi của tiến trình khi xử lý trên một tập hợp dữ liệu nào đó. Tiến trình mang tính động, là chương trình đang được thực thi trên tập hợp dữ liệu cụ thể và sử dụng tài nguyên do hệ thống cung cấp. Nhiều tiến trình có thể cùng thực hiện một chương trình, nhưng mỗi tiến trình có dữ liệu và tài nguyên riêng như nhau.

hoa trên Hình 2.2. Cụ thể hơn, mỗi tiến trình có bản ghi trạng thái lưu trữ các thông tin như chi thị nào đang được thực hiện, hay những tài nguyên được HDH cấp phát. Tiến trình chỉ có thể được thực hiện khi đã có đủ tài nguyên cần thiết. Tiến trình có thể được mô tả qua mô hình thực thi. Cơ sở trình bày về tiến trình ở đây chỉ mang tính miêu tả và được giới hạn là chương trình chạy trên máy tính truyền thống.

HDH có cấu trúc dữ liệu riêng để mô tả tiến trình. Khi tạo mới tiến trình, HDH tạo ra bản ghi mô tả tiến trình tương ứng (bản ghi này được gọi là Khối điều khiển tiến trình). Khi tiến trình kết thúc, cấu trúc dữ liệu này cũng sẽ bị xóa bỏ. Bản mô tả tiến trình ở các HDH có thể khác nhau, tuy nhiên, thường có trường định danh tiến trình; định danh các tài nguyên đã cấp phát cho tiến trình; giá trị các thanh ghi của CPU. Tiến trình được tham chiếu thông qua bộ mô tả.



Hình 2.2. Chia sẻ đoạn mã giữa nhiều tiến trình

Tiến trình là đơn vị tính toán nhỏ nhất trong máy tính, đơn vị cơ sở này được chương trình đang thực thi sử dụng để chia sẻ CPU. Mặc dù đơn vị tính toán là tiến trình, nhưng một số HDH hiện đại có thể cài đặt hai đơn vị tính toán cơ sở khác là thread và object. Giữa thread và object không có mối quan hệ tường minh. Một số HDH sử dụng thread để cài đặt object.

2.3.1. Tạo mới tiến trình

Khi khởi động, máy tính phải bắt đầu thực hiện các chi thị nằm trong bộ nhớ. Tiến trình đầu tiên này sẽ thực hiện nhiệm vụ tải bộ nạp (mục 4.2). Bộ nạp tiếp tục tải HDH vào bộ nhớ trong. Sau đó, máy tính bắt đầu thực thi chương trình của HDH. Vậy, các tiến trình tiếp theo được khởi tạo như thế

nào? Cách thông thường để tạo mới tiến trình là thực hiện lời gọi hệ thống *spawn*. Một trường hợp riêng của *spawn* là *fork* – tạo ra một tiến trình mới để thực hiện chương trình giống như tiến trình đang thực thi. Năm 1963, Conway đưa ra ba hàm FORK, JOIN và QUIT. Những hàm cơ bản này được sử dụng để tạo mới và thực hiện một lớp tiến trình. Không giống các HĐH hiện đại, các tiến trình được tạo ra nhờ lời gọi FORK truyền thống thực hiện trong cùng không gian địa chỉ, có nghĩa là, chúng cùng chia sẻ bản sao của một chương trình và tất cả các thông tin. Khi thực hiện, tiến trình được tạo mới tham chiếu tới cùng một biến chứ không phải các biến riêng của mình. Hành vi của các câu lệnh được xác định như sau:

- **FORK(label)**: Tạo ra một tiến trình mới (xác định bởi chính chương trình đang thực thi). Tiến trình này nằm trong cùng không gian địa chỉ với tiến trình gốc và bắt đầu thực thi từ chi thị có nhãn label. Tiến trình vừa thực hiện lệnh FORK, vừa thực hiện câu lệnh tiếp theo. Ngay khi tiến trình mới được tạo ra, hai tiến trình cha và con cùng tồn tại và thực thi.
- **QUIT()**: Kết thúc tiến trình. Tiến trình bị loại bỏ và HĐH xóa bộ mô tả tiến trình.
- **JOIN(count)**: Kết hợp hai hoặc nhiều tiến trình thành một. Thực thi lệnh này tương đương với thực thi đoạn mã sau:

```
count = count - 1; // Giảm biến count đi 1  
if (count!=0) QUIT(); // Kết thúc nếu đây là tiến trình cuối cùng
```

trên biến dùng chung count. Tại thời điểm bất kỳ chỉ duy nhất một tiến trình được phép thực hiện lệnh JOIN. Ngay khi tiến trình bắt đầu thực hiện lời gọi hệ thống JOIN, không tiến trình nào được quyền sử dụng CPU cho đến khi tiến trình này thực hiện xong lời gọi hệ thống. Đoạn mã này thực hiện phương pháp độc quyền truy xuất (sẽ được trình bày chi tiết trong Chương 6) do việc thực hiện lệnh JOIN không bị ngắt giữa chừng.

FORK, JOIN và QUIT được sử dụng để mô tả các công việc tính toán cấu thành các tiến trình tuần tự, phối hợp với nhau và thực hiện đồng thời trong cùng một không gian địa chỉ duy nhất. Chú ý là các tiến trình chia sẻ cả dữ liệu lẫn chương trình.

Tuy nhiên, các lời gọi tạo mới tiến trình trong HĐH hiện đại (gọi là fork, CreatProcess,...) tạo ra tiến trình con có không gian địa chỉ riêng. Phương pháp cũ tuy cho phép chia sẻ mã chương trình và dữ liệu giữa tiến trình con với tiến trình cha cũng như giữa các tiến trình anh em, nhưng chương trình quản lý bộ nhớ không thể cô lập bộ nhớ của các tiến trình. Tiến trình con có không gian bộ nhớ riêng, cho phép tách biệt hoàn toàn các tiến trình khác nhau. Quan trọng hơn, tiến trình con phải có khả năng thực hiện chương trình khác với chương trình của tiến trình cha. Nếu không thì tất cả tiến trình con sẽ giống một tiến trình đầu tiên (vì các tiến trình khác đều được khởi tạo gián tiếp hoặc trực tiếp từ tiến trình ban đầu). Do vậy, trong HĐH hiện đại phải có kỹ thuật để tiến trình con thực thi một chương trình cụ thể.

2.3.2. Cách sử dụng FOLK, JOIN, QUIT

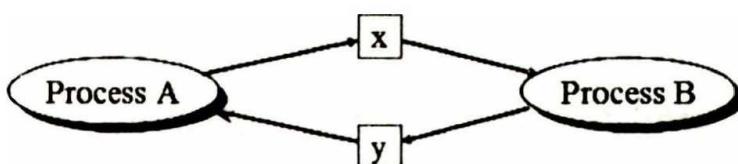
Quan sát đoạn chương trình trong Hình 2.3. Tiến trình A (procA) tính toán một vài giá trị (trong <compute section A1>), sau đó cập nhật biến dùng chung x. Trong khi đó, tiến trình B bắt đầu thực hiện procB, mặc dù nhẽ ra B chưa được phép thực hiện câu lệnh retrieve(x) cho tới khi tiến trình A cập nhật xong x. Tương tự, A không được thực hiện retrieve(y) trên biến dùng chung y cho đến khi B hoàn thành update(y). Đoạn mã này phức tạp, bởi vì hai tiến trình thực hiện vòng lặp và tiến trình này có thể thực hiện vòng lặp nhanh hơn tiến trình kia. Khi đó giá trị trước của x hoặc y có thể bị mất, vì tiến trình có tốc độ thực thi nhanh có thể thay đổi giá trị trước khi giá trị ấy được tiến trình kia đọc.

```

procA() {
    while(TRUE) {
        <compute section A1>;
        update(x);
        <compute section A2>;
        retrieve(y);
    }
}

procB() {
    while(TRUE) {
        retrieve(x);
        <compute section B1>;
        update(y);
        <compute section B2>;
    }
}

```



Hình 2.3. Phối hợp giữa hai tiến trình

Bằng các hàm tạo mới nguyên thủy, hai tiến trình khác nhau A và B có thể thực hiện đồng thời và phối hợp với nhau để ngăn chặn việc thay đổi giá trị biến dùng chung trước khi đọc. Hình 2.4 minh họa việc thiết lập trình tự thực hiện của các đoạn mã lệnh. Việc gộp A và B vào một chương trình là kết quả việc sử dụng chung không gian địa chỉ của lệnh JOIN. Vì lời gọi hệ thống FORK sử dụng nhãn để xác định vị trí khởi đầu của tiến trình mới, nên chúng ta cũng phải sử dụng nhãn trong ngôn ngữ bậc cao.

Hành vi của tiến trình trong UNIX được xác định bởi đoạn mã chương trình, đoạn dữ liệu và đoạn ngắn xếp. Đoạn mã chương trình chứa những chỉ thị đã được biên dịch ra ngôn ngữ máy, đoạn dữ liệu chứa các biến tĩnh, đoạn ngắn xếp chứa ngắn xếp trong quá trình thực thi (được sử dụng để lưu trữ biến tạm thời). Nhiều file nguồn được dịch, biên dịch và liên kết thành một file khả thi với tên mặc định a.out (tất nhiên người lập trình có thể đặt bất kỳ tên nào cho file khả thi). File khả thi xác định ba đoạn của chương trình (Hình 2.5). Trong đoạn mã chương trình, địa chỉ lệnh rẽ nhánh và địa chỉ các thủ tục nằm trong địa chỉ đoạn mã chương trình. Dữ liệu tĩnh (biến khai báo trong chương trình C) nằm trong đoạn dữ liệu và cũng được định nghĩa trong file khả thi. Hệ thống tạo ra đoạn dữ liệu, khởi tạo giá trị cho các biến cũng như không lưu trữ cho các biến trong quá trình tiến trình thực thi. Đoạn ngắn xếp dùng để cấp phát bộ nhớ cho các biến động của chương trình.

```

L0: count = 2;           L0: count = 2;
    <compute section A1>;   <compute section A1>;
    update(x);             update(x);
    FORK(L2);              FORK(L2);
    <compute section A2>;  retrieve(y);
L1: JOIN(count);         <compute section B1>
    retrieve(y);           update(y);
    goto L0;               FORK(L3)
L2: retrieve(x);         L1: JOIN(count);
    <compute section B1>;  retrieve(y);
    update(y);             goto L0;
    FORK(L3);              L2: <compute section A2>;
    goto L1;               goto L1;
L3: <compute section B2>  L3: <compute section B2>
    QUIT();                QUIT();

```

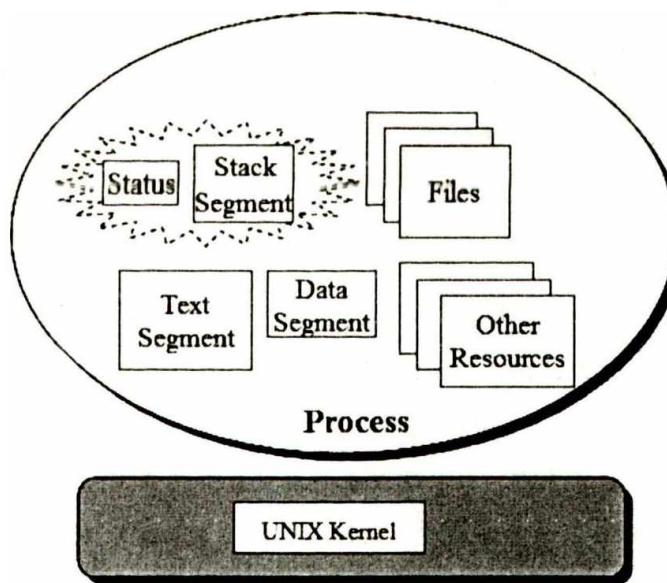
Hình 2.4. Sử dụng các hàm nguyên thủy

File khả thi được tạo ra bởi trình biên dịch, trình liên kết và bộ tải. File khả thi không định nghĩa tiến trình, mà định nghĩa đoạn mã chương trình và

khuôn dạng các thành phần dữ liệu mà tiến trình sẽ sử dụng trong quá trình thực hiện chương trình. Khi đoạn mã của tiến trình được tải vào bộ nhớ trong, hệ thống mới tạo ra đoạn dữ liệu và đoạn ngắn xếp tương ứng.

Tiến trình có duy nhất một định danh PID (process identifier), về bản chất là một con trỏ, một biến có giá trị nguyên – trỏ vào hàng nào đó trong bảng mô tả tiến trình nằm trong nhân UNIX. Mỗi hàng trong bảng này ứng với bản mô tả của tiến trình nào đó. Tiến trình sử dụng định danh tiến trình mình muốn tham chiếu tới làm tham số cho lời gọi tham chiếu. Lệnh ps của UNIX liệt kê tất cả các tiến trình trong hệ thống cùng với định danh tiến trình và định danh người sử dụng tạo ra tiến trình.

Trong UNIX, lời gọi *fork* tạo mới một tiến trình. Tiến trình cha sử dụng *fork* tạo ra tiến trình con có định danh xác định. Đoạn chương trình, đoạn dữ liệu và đoạn ngắn xếp của tiến trình con là bản sao của tiến trình cha. Tiến trình con có thể truy cập tới tất cả các file mà tiến trình cha mở. Tiến trình cha và tiến trình con thực hiện trong không gian địa chỉ riêng. Điều này có nghĩa là, dù chúng có cùng truy cập đến một biến, thì tiến trình con và tiến trình cha tham chiếu đến bản sao thông tin riêng của mình. Các tiến trình không chia sẻ không gian địa chỉ bộ nhớ, do đó tiến trình cha và tiến trình con không thể giao tiếp thông qua biến dùng chung. Trong UNIX, hai tiến trình chỉ có thể truyền thông với nhau thông qua file dữ liệu. Tạo mới một tiến trình UNIX được trình bày kỹ trong phần sau.



Hình 2.5. Tiến trình trong UNIX

Ngoài ra hệ thống UNIX còn cung cấp một số lời gọi hệ thống, chẳng hạn **execve** cho phép tiến trình có thể tải một chương trình khác vào không gian địa chỉ của mình.

Execve (char *path, char * argv[], char *envp[]);

Lời gọi hệ thống này cho phép chương trình xác định qua đường dẫn **path** thay thế chương trình hiện tại được tiến trình thực hiện. Sau khi **execve** thực hiện xong, chương trình gọi nó không còn được tải nữa. Không có khái niệm trở về sau **execve**, vì chương trình gọi không còn nằm trong bộ nhớ. Chương trình mới sử dụng danh sách tham số **argv** và các biến môi trường **envp**.

```
#include <sys/wait.h>
#define NULL 0
int main (void) {
    if (fork() == 0){ /* Đây là chương trình con */
        execve ("child", NULL, NULL);
        exit (0); /* Kết thúc*/
    }
    /* Đoạn mã của tiến trình cha*/
    printf (" Process [%d]: Parent in execution...\n", getpid ());
    sleep (2);
    if (wait (NULL) > 0) /* Tiến trình con kết thúc */
        printf (" Process [%d]: Parent detects terminating child \n",
getpid());
    printf (" Process [%d]: Parent terminating...\n", getpid())
}
```

(a) Tiến trình cha

```
int main (void) {
    /* Tiến trình con thi hành chương trình mới (thay thế chương trình của
    tiến trình cha) */
    printf (" Process [%d]: child in execution ... \n", getpid ());
    sleep (1);
    printf (" Process [%d]: child terminating ... \n", getpid ());
}
```

(b) Tiến trình con

Hình 2.6. Tiến trình cha và con

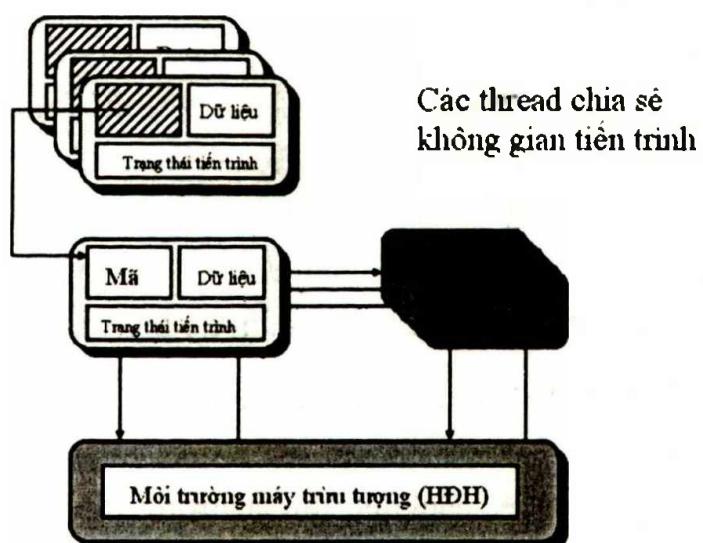
Lời gọi hệ thống **wait** (**waitpid** được sử dụng rộng rãi hơn) cho phép tiến trình cha xác định tiến trình con của mình đã hoàn thành chưa. Chi tiết trạng thái kết thúc của tiến trình con được trả về tiến trình cha thông qua giá trị tham chiếu của **wait**. Tuy nhiên, tiến trình cha có thể bỏ qua, không quan tâm đến tiến trình con. Lời gọi **wait** không phân biệt giữa các tiến trình con, trong khi các biến thể của lời gọi **waitpid** cho phép tiến trình cha có thể đợi một tiến trình con cụ thể (dựa trên PID) hoàn thành. Khi kết thúc, tài nguyên

của tiến trình bao gồm bản mô tả tiến trình trong nhân HDH được giải phóng. HDH báo hiệu cho tiến trình cha khi tiến trình con kết thúc, nhưng HDH chưa xóa bản mô tả tiến trình con cho tới khi tiến trình cha nhận được tín hiệu. Tiến trình cha thực hiện lời gọi **wait** để ghi nhận mình đã biết tiến trình con hoàn thành.

Chương trình cha trong Hình 2.6a minh họa cách sử dụng **execve** và **wait**. Trong ví dụ này, tiến trình cha tạo ra tiến trình con bằng lời gọi **fork**, sau đó thực hiện lời gọi **printf** và **sleep**. Sau khi gọi **fork**, tiến trình con thực hiện **execve** để thay thế chương trình trong Hình 2.6a bằng chương trình trong Hình 2.6b. Sau khi tiến trình con thực hiện xong **execve** trong Hình 2.6a thì câu lệnh được thực hiện tiếp theo là câu lệnh đầu tiên trong Hình 2.6b.

2.4. LUỒNG

Mô hình luồng cho phép nhiều thực thể cùng thực hiện một chương trình, đồng thời sử dụng chung các file và thiết bị. Bên cạnh tiến trình, luồng là kiểu đơn vị tính toán cơ sở mà HDH có thể điều phối. Trong mô hình này, tiến trình là thực thể mang tính trừu tượng, được cấp phát các kiểu tài nguyên khác nhau, tuy nhiên không có thành phần thực hiện chương trình. Thread (đôi khi được gọi là tiểu trình) là thực thể thực hiện chương trình và sử dụng các tài nguyên của tiến trình tương ứng. Tiến trình có thể có nhiều luồng. Các thread anh em là các thread nằm trong cùng một tiến trình, chia sẻ chương trình và tài nguyên của tiến trình. Trong mô hình thread, tiến trình theo kiểu truyền thống là tiến trình có duy nhất một thread thực hiện chương trình.



Hình 2.7. Các thread trong một process

Việc điều phối thread tương tự điều phối tiến trình trong mô hình truyền thống, thực hiện chuyển đổi quyền điều khiển CPU giữa các thread cạnh tranh. Trong một vài hệ thống, bộ điều phối thread là chương trình người dùng, tuy nhiên trong nhiều trường hợp, bộ điều phối được tích hợp bên trong HĐH. Số lượng thông tin trạng thái của thread không nhiều, do đó việc điều phối thread đơn giản hơn điều phối tiến trình. Động lực quan trọng nhất của việc sử dụng thread là giảm thiểu thời gian chuyển ngữ cảnh giữa các thread. Điều này cho phép chuyển quyền sử dụng CPU giữa các đơn vị tính toán cơ sở (thread) với chi phí quản lý phụ trội thấp nhất. Tương tự, bộ quản lý thread có bản mô tả thread ghi lại giá trị các thanh ghi và nội dung ngăn xếp cho từng thread. Vì tất cả các thread của tiến trình chia sẻ một chương trình dùng chung, chương trình này sẽ định nghĩa hành vi của tiến trình. Lập trình viên dễ dàng xây dựng chương trình có nhiều thread tương tác với nhau để quản lý tài nguyên dùng chung đã được cấp phát cho tiến trình cha.

Các hệ thống giao diện đồ họa dưới dạng cửa sổ thường sử dụng mô hình thread để quản lý nhiều cửa sổ trên cùng một màn hình. Giả sử hệ thống cửa sổ được xây dựng thông qua một tiến trình quản lý màn hình vật lý và có nhiều thread (ứng với cửa sổ). Tất cả các thread chạy cùng một đoạn mã, cùng chia sẻ màn hình vật lý, nhưng mỗi thread quản lý một cửa sổ riêng. Trong kiểu hệ thống này, thời gian đáp ứng rất quan trọng, do đó phải giảm thiểu thời gian chuyển ngữ cảnh giữa các cửa sổ.

Ngày nay, thread là cơ chế quan trọng khi lập trình song song. Tuy thực hiện trong không gian địa chỉ của tiến trình, nhưng thread lại là đơn vị tính toán độc lập và bộ điều phối có thể kiểm soát mức độ tiến triển của từng thread. Các thread cùng nhau chia sẻ tài nguyên được cấp phát cho tiến trình cha. Thread là một cách đơn giản để lọc và chia công việc trong tiến trình.

Thread trong C:

Thread có thể được hỗ trợ ở mức nhân của HĐH hoặc ở mức thư viện. Ở mức thư viện, package thread trong thư viện POSIX được sử dụng rộng rãi vì là một phần của chuẩn POSIX. Các package này có giao diện chương trình tương tự package process của UNIX, mặc dù kernel UNIX chưa chắc đã hỗ trợ thread (Sun Solaris và Linux hỗ trợ chế độ thread qua giao diện POSIX).

Đoạn mã sau minh họa cách thức chương trình tạo ra thread bằng cách sử dụng thư viện, thread con được tạo ra qua hàm **Cthread_fork**. Sau khi sinh ra, thread con dùng chung các biến tĩnh với thread cha và với các thread anh em. Các thread con có thể trì hoãn thread cha, hoặc ngược lại thông qua lời gọi **Cthread_yield**.

```
#include <cthreads.h>
...
int main (int argc, char *argv[])
{
    t_handle = cthread_fork (tChild, l);
    /* A child thread is now executing the tChild function */
    Cthread_yield (); /* Yield to another thread */
}

void tChild (int me)
{
    /* This function is executed by the child */
    ...
    Cthread_yield (); /* Yield to another thread */
    ...
}
```

2.5. ĐỐI TƯỢNG

Ý tưởng đối tượng xuất phát từ ngôn ngữ lập trình mô phỏng. Đối tượng là thực thể có tính tự trị nằm trong hệ thống. Chương trình mô phỏng có thể coi như chương trình quản lý một số lượng lớn các đơn vị tính toán riêng biệt, mỗi đơn vị tính toán thực hiện một nhiệm vụ nào đó tại một thời điểm cụ thể và các đơn vị tính toán có thể có quan hệ với nhau. Ngôn ngữ lập trình mô phỏng Simula 67 đưa ra ý tưởng về lớp (class), dùng để định nghĩa hành vi của đơn vị tính toán mô phỏng, giống như chương trình định nghĩa hành vi tiến trình. Định nghĩa lớp bao gồm các phương tiện cho phép đối tượng khai báo dữ liệu riêng của mình. Như vậy, lớp giống một kiểu dữ liệu trừu tượng có trạng thái riêng (tập hợp giá trị của các biến riêng) và được thực hiện như một đơn vị tính toán tự trị. Quá trình mô phỏng được định nghĩa bằng cách xác định tập thể hiện của các lớp, là các đối tượng và giao tiếp giữa các đối tượng thông qua việc gửi thông điệp.

Đối tượng chỉ phản ứng lại với các thông điệp. Sau khi tạo ra, đối tượng có thể nhận thông điệp từ các đối tượng khác. Đối tượng phản ứng lại bằng cách thực hiện tính toán trên dữ liệu của mình và sau đó gửi thông điệp trả lời. Do hành vi của đối tượng được xác định thông qua định nghĩa lớp, lập trình viên định nghĩa các lớp và cách thức khởi tạo đối tượng từ định nghĩa lớp. Ngày nay, hướng đối tượng được sử dụng trong tất cả các ứng dụng. Ngôn ngữ lập trình hướng đối tượng đã đưa ra mô hình lập trình ứng dụng mới. Bởi tính phổ biến của mô hình này, một số HĐH hiện nay được cài đặt bằng cách sử dụng đối tượng (HĐH Spring). Hướng đối tượng ngày càng có vai trò quan trọng trong các HĐH hiện đại.

2.6. NHẬN XÉT

Đối với lập trình viên ứng dụng, HĐH là môi trường tính toán cấu thành bởi tiến trình, file và các tài nguyên khác. Tiến trình là đơn vị tính toán nhỏ nhất mà hệ thống có thể điều phối được, nó biểu diễn sự thực thi của chương trình. File được sử dụng để lưu trữ thông tin giữa các phiên làm việc. Tất cả HĐH đều cho phép tổ chức file dưới dạng một luồng byte tuần tự, tuy nhiên, một số HĐH còn hỗ trợ nhiều loại file có cấu trúc phức tạp. Các tài nguyên khác bao gồm CPU, bộ nhớ, thiết bị và bất kỳ thành phần nào mà tiến trình phải yêu cầu HĐH cung cấp. Các tài nguyên, đặc biệt là file sẽ được hệ thống kiểm soát và tiến trình phải yêu cầu cấp phát trước khi thực thi. Tiến trình có chương trình xác định hành vi của mình, tài nguyên cần thiết để hoạt động và dữ liệu sẽ thao tác trên đó.

CÂU HỎI ÔN TẬP

1. Phân biệt tiến trình với chương trình.
2. Định nghĩa tài nguyên của hệ thống.
3. Trình bày ưu điểm của luồng so với tiến trình.

Chương 3

CẤU TRÚC HỆ ĐIỀU HÀNH

Chương 2 trình bày quan điểm của người lập trình ứng dụng về HĐH và nghiên cứu HĐH dưới góc độ người lập trình hệ thống. Chương 3 trình bày các đặc điểm chung của HĐH. Các chương sau để cập đến từng vấn đề cụ thể của HĐH. Trước tiên sẽ phân tích những nhân tố chính ảnh hưởng đến quá trình thiết kế HĐH. Sau đó sẽ trình bày các chức năng cơ bản của HĐH như quản lý thiết bị, quản lý tài nguyên, quản lý bộ nhớ, quản lý file. Phần cuối trình bày những công nghệ được HĐH sử dụng để thực hiện những chức năng trên.

3.1. PHÂN TÍCH CÁC YẾU TỐ TÁC ĐỘNG ĐẾN HỆ ĐIỀU HÀNH

HĐH trừu tượng hóa các tài nguyên máy tính, giúp lập trình viên phát triển chương trình ứng dụng, ví dụ tiến trình là sự trừu tượng hóa hoạt động của bộ vi xử lý; file là sự trừu tượng hóa của thiết bị lưu trữ. Trong quá trình tính toán, trạng thái tiến trình thay đổi liên tục khi sử dụng các tài nguyên. HĐH cung cấp các hàm để tiến trình tạo mới hoặc kết thúc tiến trình khác; yêu cầu, sử dụng hoặc giải phóng tài nguyên; phối hợp hành động với những tiến trình có liên quan. Ngoài ra, HĐH cần có khả năng quản lý và cấp phát tài nguyên theo yêu cầu; hỗ trợ chia sẻ và khi chia sẻ phải có phương thức kiểm soát sao cho không gây nên bất kỳ sự cố nào. Bên cạnh những yêu cầu quản lý cơ bản, các yếu tố sau đây cũng tác động đến quá trình thiết kế và phát triển của HĐH:

- Hiệu suất.
- Bảo vệ và an ninh.

- Tính chính xác.
- Khả năng bảo trì.
- Thương mại.
- Chuẩn và Hệ thống mở.

3.1.1. Hiệu suất

Ở mức thấp nhất, HĐH cung cấp giao diện lập trình và cơ chế quản lý việc chia sẻ tài nguyên. Hai chức năng trên đóng vai trò quản lý vì không trực tiếp giải quyết vấn đề của người sử dụng, mà chỉ tạo ra môi trường giúp chương trình ứng dụng thực thi. Chức năng quản lý tài nguyên giúp sử dụng hệ thống dễ dàng hơn, nhưng lại đòi hỏi một chi phí phụ trội nào đó. Ví dụ, giao diện trùu tượng giúp cho việc viết chương trình dễ dàng hơn, nhưng lại làm chậm tốc độ thực thi của chương trình. Ví dụ, tốc độ thực hiện thao tác lên file chậm hơn nhiều so với thao tác trực tiếp lên ổ đĩa cứng. Khi đưa chức năng mới vào HĐH, phải đánh giá xem chức năng thêm mới vào có ảnh hưởng tới hiệu suất tổng thể của hệ thống hay không. Nhiều ràng buộc về hiệu suất đã ngăn cản việc tích hợp thêm tính năng mới vào HĐH. Do phần cứng ngày càng mạnh, các nhà thiết kế đã tích hợp thêm nhiều chức năng vào HĐH và bỏ qua vấn đề suy giảm hiệu suất. Tích hợp các ngôn ngữ lập trình bậc cao, đối tượng, chức năng bộ nhớ ảo, đồ họa và kết nối mạng vào HĐH minh chứng cho xu thế này. Không có tiêu chí rõ ràng để xác định rằng, liệu một chức năng với chi phí cài đặt cao có nên đưa vào HĐH không. Vấn đề này sẽ được giải quyết dựa trên từng tình huống cụ thể và phải được phân tích chi tiết, liệu tính ưu việt của chức năng mới có đáng với hiệu suất bị suy giảm hay không.

3.1.2. Bảo vệ và an ninh

HĐH đa chương trình cho phép nhiều tiến trình thực thi tại cùng một thời điểm. Như vậy, HĐH phải có cơ chế ngăn cản không cho tiến trình tác động lên hoạt động của tiến trình khác. Tiến trình cũng không được phép sử dụng tài nguyên không hợp lệ. Do đó, HĐH phải có khả năng cấp phát riêng tài nguyên cho tiến trình, hoặc cấp phát tài nguyên cho nhiều tiến trình dùng chung. Ngoài ra, HĐH phải có cơ chế kiểm soát bằng phần mềm để vừa bảo đảm cơ chế cô lập nhưng vẫn có khả năng chia sẻ tài nguyên. Cơ chế bảo vệ

là công cụ để HĐH triển khai biện pháp an ninh được người quản trị hệ thống thiết lập. Biện pháp an ninh định nghĩa phương pháp quản lý việc truy cập tới tài nguyên. Chẳng hạn, tại một thời điểm chỉ cho phép duy nhất một tiến trình được quyền mở file để ghi, nhưng cho phép nhiều tiến trình có thể mở file để đọc. Cơ chế bảo vệ file có thể cài đặt biện pháp này thông qua cơ chế khóa đọc và khóa ghi file. Cơ chế bảo vệ thường được cài đặt trong HĐH. Tuy nhiên, có vấn đề này sinh trong quá trình thiết kế: Nếu sau khi HĐH thiết lập một biện pháp thì làm thế nào để ngăn cản phần mềm ứng dụng thay đổi nó? Đây là một vấn đề quan trọng trong các HĐH hiện đại. Bảo vệ tài nguyên là một lĩnh vực cụ thể trong nghiên cứu về HĐH (Chương 12). Tuy nhiên, cũng như vấn đề hiệu suất, tính năng này quan trọng đến mức mọi HĐH đều phải xét đến yếu tố an ninh khi đưa bất kỳ tính năng mới nào vào HĐH. Như sẽ trình bày trong mục 3.3, trên thực tế vấn đề này được giải quyết là phần cứng phải có khả năng phân biệt được giữa phần mềm HĐH và phần mềm ứng dụng.

3.1.3. Tính chính xác

Một số phần mềm có thể được coi là "tin cậy", nhưng một số phần mềm bị coi là "không tin cậy". Phương pháp bảo vệ của hệ thống phụ thuộc nhiều vào các thao tác chính xác của phần mềm HĐH "tin cậy". Mỗi chức năng phải có những yêu cầu cụ thể. Điều này cho phép nhà thiết kế có thể nói chức năng X, dưới điều kiện Y có hoạt động chính xác không. Ví dụ, không thể xác định được bộ điều phối có hoạt động chính xác không, nếu không biết bộ điều phối cần phải thực hiện công việc gì. Nói chung, rất khó đưa ra yêu cầu cụ thể cho phần mềm HĐH. Có một nhánh nghiên cứu HĐH thực hiện đánh giá thiết kế và triển khai thiết kế có đáp ứng được yêu cầu hay không. Những nhà thiết kế HĐH khác chỉ sử dụng những công cụ hình thức để chứng minh phần mềm hệ thống đáng tin cậy. Tính chính xác là yếu tố hết sức cơ bản phải được cân nhắc khi muốn tích hợp thêm chức năng vào HĐH.

3.1.4. Khả năng bảo trì

Vào những năm 1960, HĐH đã phức tạp đến mức không ai có thể hiểu mọi dòng chương trình trong mã nguồn của nó. Bên cạnh mối quan tâm về tính chính xác, một vấn đề mới này sinh là: Làm thế nào để thay đổi phần mềm HĐH nhưng vẫn đảm bảo độ tin cậy, độ chính xác của kết quả và

không phát sinh lỗi mới? Một nhóm những nhà thiết kế có xu hướng cho phép sản phẩm của mình dễ bảo trì, mà không chú ý nhiều đến yếu tố phổ dụng và hiệu suất hệ thống.

3.1.5. Thương mại

Phần lớn các HĐH thương mại hiện đại được phát triển trên nền tảng HĐH đa chương trình chia sẻ thời gian có kết hợp khả năng kết nối mạng. UNIX là HĐH chia sẻ thời gian và những phiên bản đầu tiên của BSD UNIX hay AT&T vẫn tiếp tục là các hệ thống chia sẻ thời gian. UNIX chiếm phần lớn thị phần trong môi trường đa chương trình (máy tính cá nhân và máy trạm). Mặt khác, môi trường máy tính cá nhân bị ràng buộc bởi các sản phẩm của Microsoft (trước kia là DOS và bây giờ là Microsoft Windows), có thể là do quan hệ hợp tác chặt chẽ giữa Microsoft với hãng IBM sản xuất phần cứng. Ngày nay, công nghệ phần cứng máy tính cá nhân hội tụ dần với công nghệ máy trạm và khi đó phải hỗ trợ đa chương trình trên máy tính cá nhân. Vấn đề đặt ra ở đây là, trong khi UNIX hỗ trợ đa chương trình, thì HĐH Windows lại được sử dụng rộng rãi hơn. Điều này khiến người lập trình và người sử dụng phải lựa chọn hoặc HĐH đa nhiệm tương thích với DOS, chẳng hạn Windows XP hay Windows NT, hoặc là HĐH thương mại UNIX nào đó. Có thể sau này, các HĐH thương mại sẽ hội tụ vào một giải pháp duy nhất hoặc thị trường vẫn tiếp tục hỗ trợ nhiều HĐH. Trong cả hai trường hợp, thị trường và các yếu tố thương mại chứ không phải các yếu tố công nghệ sẽ là nhân tố ảnh hưởng chính.

Sự thành công của HĐH UNIX và Microsoft Windows ảnh hưởng lớn đến quá trình phát triển của HĐH nói chung. Để được thị trường chấp nhận, HĐH mới phải có ngôn ngữ lập trình (chương trình dịch, trình kết nối và bộ tài), trình soạn thảo văn bản và thư viện runtime. Những môi trường tính toán hiện đại có nhiều công cụ và ứng dụng, tất cả đều viết trên nền các HĐH thông dụng. Do đó, bên cạnh những cài tiến, HĐH mới phải cung cấp môi trường cho phép thực thi các ứng dụng có sẵn.

3.1.6. Chuẩn và hệ thống mở

Những thay đổi về việc sử dụng máy tính trong tổ chức doanh nghiệp diễn ra vào cuối những năm 1980. Trước thời điểm này, các tổ chức thường

mua tất cả thiết bị máy tính từ cùng một nhà sản xuất. Tuy nhiên, theo quy luật kinh tế, người dùng cuối sẽ lợi hơn rất nhiều nếu có thể mua được thiết bị trong một thị trường mở và cạnh tranh. Nhu cầu sử dụng thiết bị từ nhiều nhà sản xuất khác nhau chính là động lực phát triển công nghệ Hệ thống mở. Điều này cho phép các doanh nghiệp có thể sử dụng máy tính, HĐH và ứng dụng từ nhiều nhà sản xuất khác nhau. Hệ thống mở tác động mạnh mẽ đến sự thành công của các doanh nghiệp sản xuất thiết bị công nghệ thông tin (CNTT). Mục tiêu của kiến trúc hệ thống mở là cho phép người dùng sau làm việc trên một mạng máy tính với nhiều chủng loại thiết bị khác nhau.

Cần ít nhất ba chiến lược đối với hệ thống mở:

- **Tích hợp ứng dụng:** Giao diện người dùng của tất cả các chương trình ứng dụng nên giống nhau. Các tiện ích quản lý thông tin và thiết bị cần được chuẩn hóa sao cho tạo nên giao diện nhất quán với người sử dụng.
- **Khả năng tương thích:** Các chương trình ứng dụng phải có khả năng cài đặt trên nhiều nền tảng phần cứng khác nhau.
- **Khả năng liên tác:** Các tiện ích trong môi trường mạng được chuẩn hóa sao cho đơn giản hóa việc truy cập tới các máy tính khác.

Mục tiêu của chuẩn POSIX là giải quyết phần lớn những khía cạnh của hệ thống mở. Cụ thể, POSIX.1 chuẩn hóa giao diện của chương trình ứng dụng với HĐH chứ không phải cách thức cài đặt UNIX. Chuẩn này khuyến khích các hãng sản xuất khác nhau sử dụng cùng một giao diện POSIX, khi đó các ứng dụng viết ra có khả năng chạy trên nhiều HĐH UNIX khác nhau. Phần lớn HĐH UNIX tuân theo chuẩn này.

3.2. CÁC CHỨC NĂNG CƠ BẢN

Bên cạnh nhiệm vụ trừu tượng hóa quá trình tính toán và quản lý tài nguyên hệ thống, HĐH cần quan tâm đến nhiều khía cạnh thực tế. Chẳng hạn hiệu suất, an ninh, tính chính xác, tính dễ bảo trì của hệ thống. Nói chung, không thống nhất được HĐH cần có những chức năng gì, do đó trong giáo trình này chỉ trình bày những chức năng cơ bản đã được thừa nhận rộng rãi, đó là: quản lý thiết bị; quản lý tiến trình và tài nguyên; quản lý bộ nhớ và quản lý file.

3.2.1. Quản lý thiết bị

Ngoài trừ CPU và bộ nhớ trong, phần lớn HĐH coi tất cả các thiết bị khác là giống nhau. Chương trình quản lý thiết bị quy định cách thức sử dụng một chủng loại thiết bị. Nói chung, nhiệm vụ của HĐH là cấp phát, cài đặt và chia sẻ thiết bị theo chính sách định trước. Thậm chí HĐH không hỗ trợ chế độ đa chương trình cũng phải có trình quản lý thiết bị. Trước kia, mã nguồn của HĐH được cung cấp cùng phần cứng. Nếu muốn kết nối thêm thiết bị vào máy tính, người sử dụng phải cài thêm driver của thiết bị vào HĐH. Nếu không có mã nguồn HĐH thì không thể biên dịch lại HĐH để gắn thêm driver mới. Hạn chế này thúc đẩy sự phát triển khả năng cấu hình lại driver trong HĐH hiện đại. Driver của thiết bị có thể được biên dịch và cài đặt thêm vào HĐH mà không cần dịch lại HĐH. Tuy quản lý thiết bị là một phần quan trọng, nhưng lại tương đối đơn giản trong thiết kế HĐH. Nội dung về quản lý thiết bị được trình bày trong Chương 8.

3.2.2. Quản lý tiến trình và tài nguyên

Tiến trình là đơn vị tính toán cơ sở, được người lập trình định nghĩa, còn tài nguyên là các thành phần trong môi trường tính toán mà tiến trình cần có để thực thi. Quản lý tiến trình và quản lý tài nguyên có thể nằm tách biệt, nhưng đa số HĐH kết hợp lại trong một module. Trong Chương 2 đã lấy mô hình tiến trình của HĐH UNIX minh họa cách thức định nghĩa một môi trường tính toán. HĐH UNIX cho phép tạo mới, hủy, phong tỏa và thực thi một tiến trình. Tương tự, HĐH có hỗ trợ luồng, hay hướng đối tượng (Window NT) cung cấp môi trường cho phép quản lý những đơn vị tính toán cơ sở tương ứng. Thành phần quản lý tài nguyên có trách nhiệm cấp phát tài nguyên (nếu có) cho các tiến trình có nhu cầu.

Bộ phận này cho phép nhiều người dùng (hoặc nhiều tiến trình) chia sẻ máy tính, bằng cách cấp phát CPU luân phiên giữa các tiến trình để mỗi tiến trình có thể sử dụng CPU trong khoảng thời gian phù hợp. Vấn đề chính của việc quản lý tiến trình và tài nguyên là làm thế nào để cài đặt việc truy cập tài nguyên của các tiến trình (theo chính sách định trước) và làm thế nào để các tiến trình vượt qua cơ chế cài đặt khi có chính sách chia sẻ tài nguyên giữa nhiều tiến trình. Cơ chế cấp phát tài nguyên phải kết hợp chặt chẽ với tiện ích quản lý tiến trình và tài nguyên. Cơ chế này bao gồm việc biểu diễn

tài nguyên; thực hiện cấp phát và sử dụng tài nguyên theo chính sách định trước. Quản lý tiến trình và tài nguyên được trình bày trong Chương 8 và 9.

3.2.3. Quản lý bộ nhớ

Chương trình quản lý bộ nhớ chịu trách nhiệm quản lý và cấp phát tài nguyên bộ nhớ chính. Tiến trình yêu cầu và sử dụng bộ nhớ theo định nghĩa của chương trình tương ứng. Bộ phận quản lý bộ nhớ cấp phát theo chính sách định trước. Chia sẻ khiến vấn đề thiết kế phức tạp hơn, vì chương trình quản lý bộ nhớ phải tích hợp cả cơ chế cô lập (để tiến trình không được truy cập vào không gian bộ nhớ của tiến trình khác) lẫn cơ chế cho phép các tiến trình có thể chia sẻ vùng nhớ chung.

HĐH hiện đại còn có công nghệ bộ nhớ ảo (mở rộng bộ nhớ chính lớn hơn giới hạn kích thước vật lý bằng cách sử dụng thêm thiết bị lưu trữ ngoài), cho phép tiến trình tham chiếu đến phần bộ nhớ lưu trên ổ đĩa cứng như thể đó là bộ nhớ trong. Quản lý bộ nhớ ảo phức tạp hơn nhiều so với quản lý bộ nhớ truyền thống, vì phải kết hợp chính sách quản lý bộ nhớ trong và chính sách quản lý ổ đĩa cứng. Các chương trình quản lý bộ nhớ trên HĐH hiện đại thậm chí còn cho phép tiến trình có thể truy cập và chia sẻ bộ nhớ vật lý của một máy tính khác. Xây dựng nền bộ nhớ ảo dùng chung phân tán bằng cách cho phép các tiến trình trao đổi thông điệp trên đường truyền kết nối các máy tính. Khi đó, chương trình quản lý bộ nhớ kết hợp các chức năng nguyên thủy của mình với chức năng kết nối mạng.

3.2.4. Quản lý file

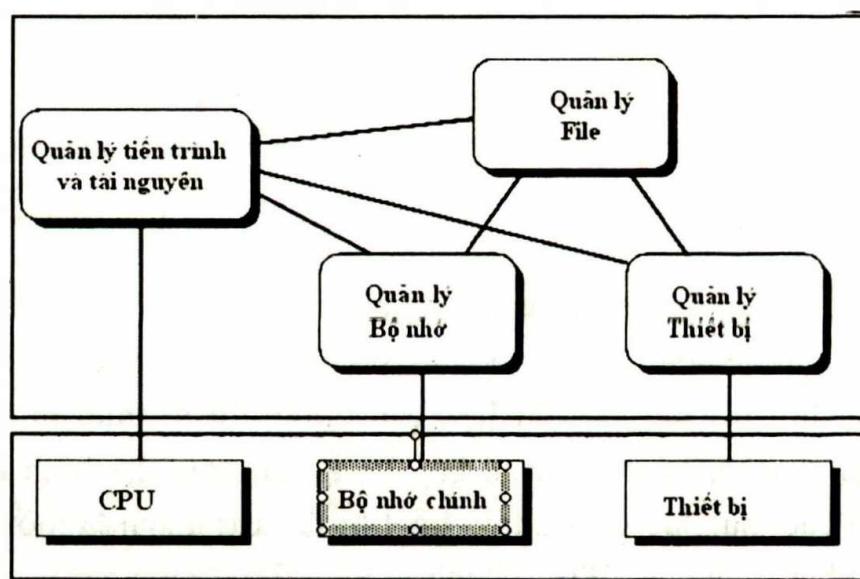
File là sự trừu tượng hóa của thiết bị lưu trữ. Thông tin nằm trên bộ nhớ chính sẽ bị ghi đè nếu khu vực bộ nhớ đó được cấp phát cho tiến trình khác. Nếu muốn lưu trữ lâu dài, dữ liệu cần được ghi ra thiết bị lưu trữ ngoài (chẳng hạn ổ đĩa). Như đã trình bày trong Chương 2, chính nhu cầu trừu tượng hóa các thao tác vào/ra trên thiết bị lưu trữ ngoài là một trong những động lực phát triển của HĐH. Chính vì vậy, file được xem là giao diện trừu tượng quan trọng nhất của HĐH.

Các bộ phận quản lý file khác nhau có các giao diện trừu tượng khác nhau, từ dạng đơn giản (như mô hình thiết bị lưu trữ dưới dạng một luồng byte) tới hết sức phức tạp (chẳng hạn mô hình cơ sở dữ liệu quan hệ). Trong

HĐH hiện đại, hệ thống file được cài đặt phân tán, cho phép tiến trình không chỉ truy cập dữ liệu trên máy tính của mình mà còn có thể truy cập tới dữ liệu trên các máy tính khác thông qua mạng máy tính. Chương 11 sẽ trình bày cách thức sử dụng tiến trình tương tác với hệ thống file thông qua giao diện lập trình ứng dụng.

3.2.5. Kết hợp các chức năng

Phần này trình bày cách thức kết hợp các chức năng cơ bản trong HĐH. Hình 3.1 minh họa quan hệ giữa các module cơ bản (kết nối giữa các module chỉ quan hệ tương tác giữa chúng). Bộ phận quản lý tiến trình và tài nguyên tạo nên tiến trình và môi trường thực thi trên nền CPU. Bộ phận này sử dụng các giao diện trừu tượng do nhiều thành phần quản lý tài nguyên khác cung cấp. Khác với các bộ phận quản lý tài nguyên khác, quản lý bộ nhớ được xếp riêng như một bộ phận độc lập của HĐH. Khi công nghệ bộ nhớ ảo thông dụng, trách nhiệm của bộ phận quản lý bộ nhớ cũng tăng lên. Quản lý file thực hiện việc trừu tượng các thao tác xuất/nhập trên thiết bị thành các thao tác đơn giản, dễ sử dụng. Bộ phận quản lý thiết bị điều khiển thao tác đọc/ghi trên thiết bị lưu trữ thứ cấp và được cài đặt dưới dạng trình điều khiển thiết bị.



Hình 3.1. Quan hệ giữa các chức năng trong tổ chức HĐH

Cấp phát tài nguyên là trách nhiệm của bộ phận quản lý tiến trình và tài nguyên. Bộ phận quản lý file sử dụng tiện ích đọc/ghi thiết bị do bộ phận quản lý thiết bị cung cấp. Các chương trình quản lý thiết bị có thể đọc/ghi

trực tiếp vào bộ nhớ chính, nên bộ phận quản lý file có quan hệ chặt chẽ với bộ phận quản lý bộ nhớ, đặc biệt trong hệ thống hỗ trợ bộ nhớ ảo. Vì thế, hệ thống tách rời bốn module để cô lập các chức năng, nhưng các module này vẫn gắn kết chặt chẽ với nhau.

3.3. CÁC PHƯƠNG THỨC CÀI ĐẶT HỆ ĐIỀU HÀMH

Các HĐH hiện đại cài đặt theo một trong ba cơ chế cơ bản sau:

- **Chế độ vi xử lý (Processor mode):** Sử dụng bit chế độ để phân biệt giữa tiến trình HĐH hay tiến trình người dùng.
- **Nhân HĐH (Kernel):** Tất cả các bộ phận chủ yếu của HĐH được đặt trong nhân. Kernel là module phần mềm cực kỳ đáng tin cậy, hỗ trợ tất cả các phần mềm khác hoạt động.
- **Phương thức yêu cầu dịch vụ hệ thống:** Vấn đề này liên quan tới cách thức tiến trình người sử dụng yêu cầu dịch vụ của HĐH, bằng cách gọi hàm hệ thống hay gửi thông điệp tới tiến trình hệ thống.

3.3.1. Chế độ của bộ vi xử lý

CPU hiện đại thường có bit chế độ để xác định khả năng thực hiện của tiến trình trên CPU. Bit này có thể được thiết lập ở chế độ supervisor (giám sát toàn bộ hệ thống) hay ở chế độ người dùng. Trong chế độ giám sát, CPU có thể thực hiện bất kỳ chỉ thị nào. Trong chế độ người dùng, CPU chỉ thực hiện được một số chỉ thị nhất định. Những chỉ thị chỉ có thể thực thi ở chế độ giám sát, được gọi là chỉ thị giám sát hay chỉ thị đặc quyền để phân biệt với chỉ thị thường.

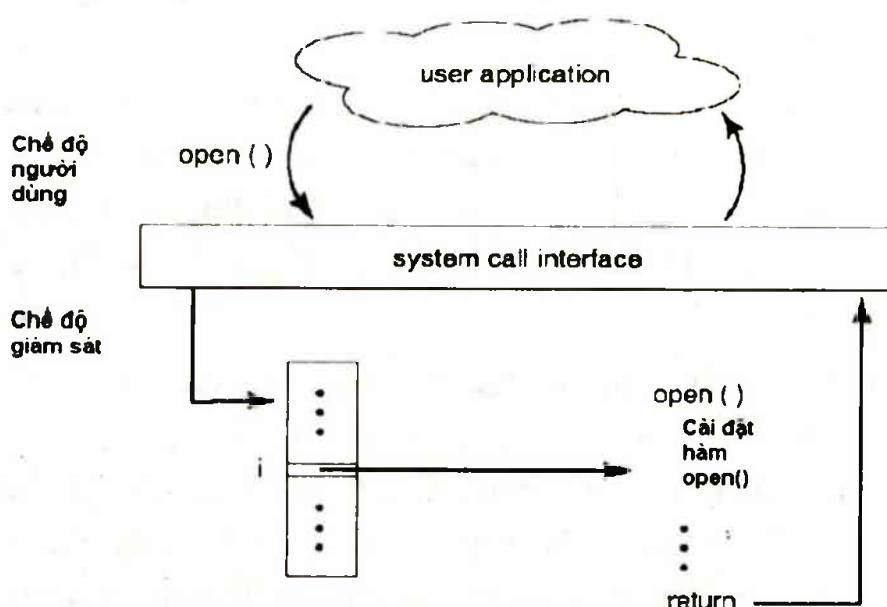
Chỉ thị vào/ra (I/O) là chỉ thị đặc quyền, vì vậy chương trình ứng dụng nếu thực thi trong chế độ người dùng không thể thực hiện thao tác vào/ra. Chương trình ứng dụng phải yêu cầu HĐH thay mặt mình thực hiện vào/ra. Khi đó, một chỉ thị phần cứng đặc biệt sẽ chuyển CPU sang chế độ giám sát và gọi driver thiết bị (driver sẽ thực hiện chỉ thị vào/ra). Bit chế độ cũng được sử dụng để giải quyết các tình huống liên quan đến an ninh hệ thống đã đề cập trong mục 3.1. Các chỉ thị có thể thay đổi trạng thái bảo vệ hiện thời của hệ thống, được gọi là chỉ thị đặc quyền. Ví dụ, HĐH sử dụng một số thanh ghi trong CPU hay những vùng nhớ đặc biệt lưu trữ trạng thái tiến

trình, con trỏ đến các nguồn tài nguyên. Khi muốn thay đổi nội dung vùng nhớ hay thanh ghi này, phải sử dụng những chỉ thị load và store đặc quyền.

Các dòng máy tính cũ như CPU Intel 8088/8086 không có bit chế độ. Do vậy, chúng không phân biệt được chỉ thị đặc quyền và chỉ thị người dùng. Kết quả là không thể cung cấp cơ chế cô lập bộ nhớ trên những máy tính này, bất kỳ chương trình người dùng nào cũng có thể tải một giá trị bất kỳ vào thanh ghi cơ sở đoạn (base segment register). Tiến trình có thể truy xuất tới bất kỳ đoạn bộ nhớ nào.

Những CPU trong họ Intel sau này đều có bit chế độ, vì vậy, chỉ có thể thay đổi giá trị thanh ghi cơ sở bằng các chỉ thị đặc quyền. Các CPU Intel đời mới tương thích "ngược" với CPU dòng 8088/8086 để phần mềm viết trên dòng CPU cũ có thể thực thi được trên hệ thống mới.

Hệ thống có thể mở rộng bit chế độ để xác định những vùng nhớ nào được sử dụng khi CPU trong chế độ giám sát và khi trong chế độ người dùng (Hình 3.2). Nếu bit chế độ được chuyển sang chế độ giám sát thì tiến trình đang chiếm dụng CPU có thể truy cập tới bất kỳ ô nhớ nào. Nếu ở trong chế độ người dùng, tiến trình chỉ có thể truy xuất tới vùng nhớ người dùng. Như vậy, vùng nhớ cũng có hai không gian là không gian người dùng và không gian bảo vệ.



Hình 3.2. Chế độ người dùng và chế độ giám sát

Nhìn chung, bit chế độ tăng cường khả năng bảo vệ của HĐH. Bit chế độ được thiết lập (đặt giá trị 1) bằng chỉ thị trap ở chế độ người dùng

(trap còn được gọi là chỉ thị yêu cầu chuyển sang chế độ giám sát). Chỉ thị thiết lập giá trị 1 cho bit chế độ và rẽ nhánh tới một vị trí xác định trong vùng nhớ hệ thống. Điều này tương tự như ngắt phần cứng. Những thủ tục của HĐH được tải vào trong vùng nhớ hệ thống và được bảo vệ vì không thể nạp mã của chương trình người dùng vào vùng nhớ đó. Vì mã hệ thống nằm trong vùng nhớ hệ thống, nên chỉ có mã chương trình hệ thống mới được gọi qua trap. Sau khi hoàn thành xong lời gọi ở chế độ giám sát, HĐH thiết lập lại bit chế độ để quay trở về chế độ người dùng.

3.3.2. Nhân hệ điều hành (Kernel)

Có thể coi bộ phận trong phần mềm hệ thống thực thi trong chế độ giám sát gọi là kernel hay nhân của HĐH. Kernel là phần mềm đáng tin cậy, có nghĩa là khi thiết kế và cài đặt, kernel sẽ triển khai cơ chế bảo vệ mà phần mềm bị coi là không đáng tin cậy (thực thi trong vùng nhớ người dùng), không thay đổi được. Tính chính xác của hệ thống không thể dựa trên những phần mở rộng của HĐH thực hiện trong chế độ người dùng. Vì vậy, khi thiết kế chức năng nào đó, vẫn đề quan trọng đặt ra là liệu có tích hợp chức năng đó trong kernel hay không. Nếu đặt trong kernel, nó sẽ thực hiện trong vùng nhớ bảo vệ và có thể truy cập tới toàn bộ kernel. Nó cũng được những bộ phận khác trong kernel coi là đáng tin cậy. Nếu thực hiện trong chế độ người dùng, chức năng không truy cập được các cấu trúc dữ liệu của kernel. Chú ý rằng, có thể dễ dàng đặt một thủ tục trong kernel, nhưng chi phí để thực hiện cơ chế trap và kiểm chứng khi gọi thủ tục là cao.

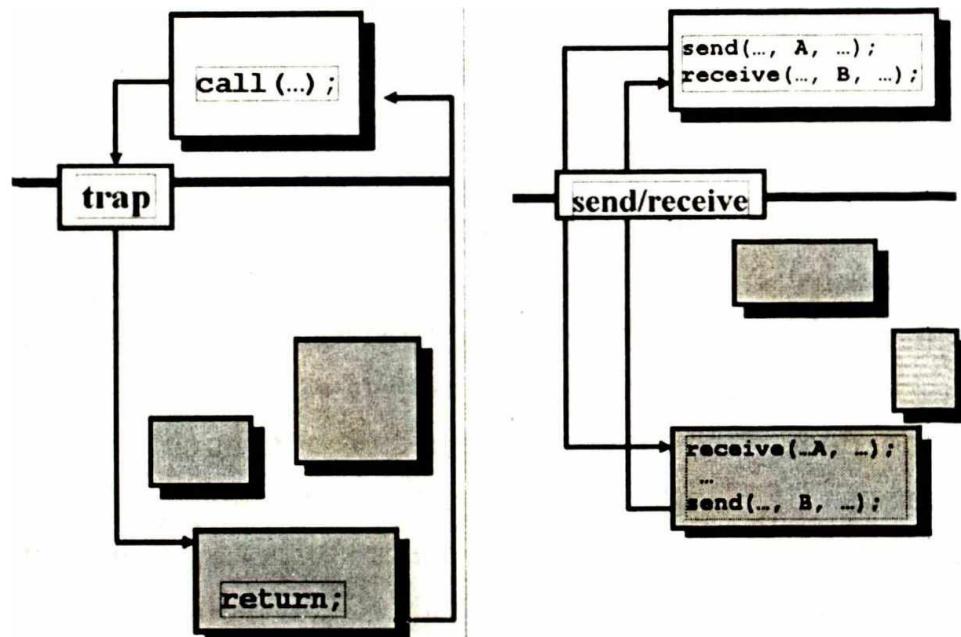
3.3.3. Yêu cầu dịch vụ từ hệ điều hành

Chương trình thực thi trong chế độ người dùng có thể yêu cầu dịch vụ của *kernel* bằng một trong hai kỹ thuật sau:

- Lời gọi hệ thống.
- Chuyển thông điệp.

Hình 3.3 minh họa sự khác biệt giữa hai kỹ thuật này. Đầu tiên, giả sử tiến trình người dùng muốn gọi một hàm hệ thống cụ thể nào đấy (biểu diễn dưới dạng hình chữ nhật tô bóng). Với kỹ thuật lời gọi hệ thống, tiến trình người dùng sử dụng chỉ thị trap (xem mục 3.3.1). Ý tưởng này là lời gọi hệ thống giống lời gọi thủ tục bình thường trong chương trình ứng dụng; HĐH

cung cấp thư viện mà mỗi hàm trong thư viện ứng với một lời gọi hệ thống cụ thể. Khi chương trình ứng dụng gọi, chỉ thị trap được thi hành để chuyển CPU sang chế độ giám sát, sau đó phân nhánh tới hàm chức năng tương ứng. Khi hàm chức năng thực thi xong, hệ thống chuyển CPU sang chế độ người dùng và trả lại quyền điều khiển cho tiến trình người dùng (tương tự việc quay trở lại sau khi kết thúc một thủ tục bình thường).



Hình 3.3. Hai kỹ thuật trong việc phát triển HĐH

Trong giải pháp chuyển thông điệp, tiến trình người dùng tạo ra thông điệp A miêu tả dịch vụ mình cần. Sau đó gửi thông điệp này cho tiến trình HĐH bằng thủ tục send. Giống trap, thủ tục send kiểm tra thông điệp, chuyển CPU sang chế độ giám sát và sau đó gửi thông điệp đến tiến trình cài đặt chức năng được yêu cầu. Trong khi đó, tiến trình người dùng chờ kết quả thực hiện dịch vụ bằng hàm đợi thông điệp receive. Khi hoàn tất dịch vụ, tiến trình HĐH gửi thông điệp trả lời B cho tiến trình người dùng.

Hiệu suất của HĐH sử dụng giao diện lời gọi hệ thống cao hơn HĐH sử dụng trao đổi thông điệp, thậm chí ngay khi lời gọi hệ thống được triển khai qua chỉ thị trap. Kỹ thuật lời gọi hệ thống có một đặc tính rất thú vị là không cần bất kỳ tiến trình HĐH nào. Thay vào đó, tiến trình hoạt động trong chế độ người dùng chuyển sang chế độ giám sát khi thực hiện đoạn mã trong kernel và khi trở về (sau lời gọi HĐH) thì quay trở lại chế độ người dùng. Nếu được thiết kế dưới dạng các tiến trình tách biệt, HĐH dễ dàng chiếm lấy quyền kiểm soát hệ thống trong những trường hợp đặc biệt hơn là trong

thiết kế mà kernel là tập hợp các hàm chức năng được tiến trình người dùng thực hiện trong chế độ giám sát.

3.4. NHẬN XÉT

HĐH tạo ra môi trường cho ứng dụng thực thi. Môi trường này quản lý việc tạo lập và thực thi tiến trình, quản lý việc sử dụng tài nguyên của tiến trình. Bên cạnh những yêu cầu cơ bản, HĐH có thể có một vài yêu cầu đặc thù khác. Mọi tính năng được cài đặt bằng phần mềm đều có một chi phí quản lý làm giảm hiệu suất của các thao tác cơ bản, do đó phải cân nhắc giữa giá trị của chức năng mới và sự suy giảm hiệu suất tổng thể. Thiết kế và cài đặt chức năng phải được tính toán kỹ lưỡng, nhằm đảm bảo hiệu suất hệ thống. HĐH phải tạo ra môi trường chia sẻ an toàn, sao cho các tiến trình không干涉 lẫn nhau. Phần mềm HĐH thường lớn và phức tạp, nhưng phải hoạt động chính xác và có thể bảo trì được. Các yếu tố thương mại, khả năng tương thích, các chuẩn mở cũng tác động mạnh mẽ lên công nghệ HĐH. HĐH hiện đại tích hợp bộ phận quản lý tiến trình và tài nguyên, ngoài ra còn có bộ phận quản lý những tài nguyên khác như bộ nhớ, file và driver. Các bộ phận này có quan hệ chặt chẽ với nhau. Các kỹ thuật triển khai HĐH hiện đại thường dựa trên một vài công nghệ cơ bản. CPU có bit chế độ xác định chế độ hoạt động là chế độ giám sát và chế độ người dùng. Nếu CPU đang ở trong chế độ người dùng và muốn thiết lập bit chế độ để chuyển sang chế độ giám sát, CPU phải thực thi chi thị trap để thiết lập bit chế độ và sau đó rẽ nhánh tới mã HĐH. Nếu đang ở trong chế độ giám sát, CPU có thể chuyển sang chế độ người dùng mà không cần hành động đặc biệt nào. Bộ phận HĐH thực thi trong chế độ giám sát được gọi là kernel. Một số HĐH hiện đại sử dụng giao diện lời gọi hệ thống để tiến trình người dùng có thể thực thi đoạn mã HĐH trong chế độ giám sát. Một số HĐH khác thiết kế kernel như những tiến trình hoạt động độc lập tương tác với tiến trình ứng dụng bằng cách trao đổi thông điệp.

CÂU HỎI ÔN TẬP

1. Trình bày các yếu tố tác động lên sự phát triển của HĐH.
2. Trình bày các chức năng chính của một HĐH.
3. Trình bày các phương thức cài đặt HĐH chính.

Chương 4

TIẾN TRÌNH

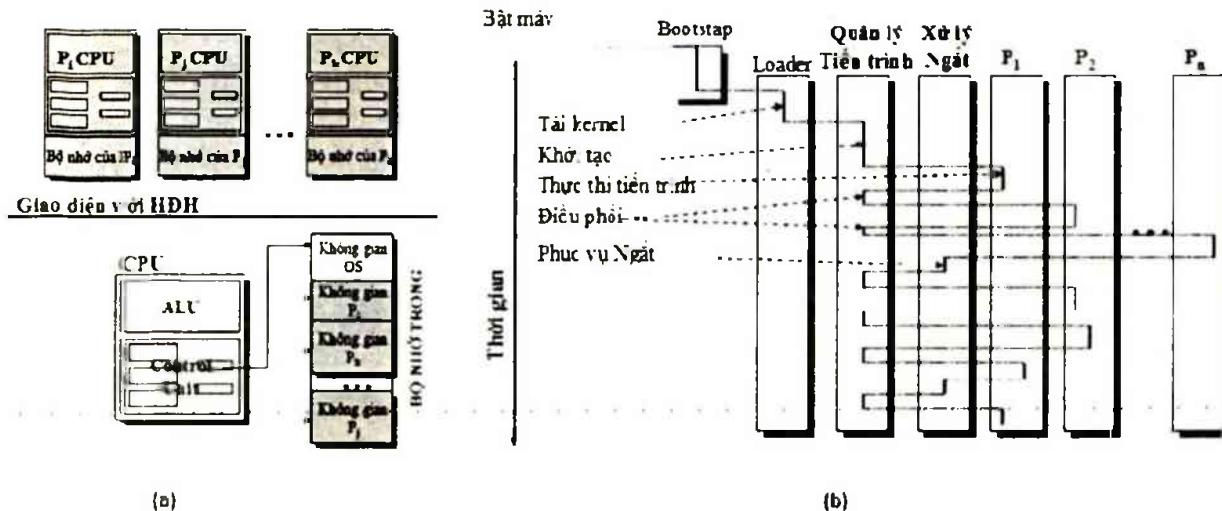
Trong những máy tính thế hệ đầu tiên, tại một thời điểm cụ thể chỉ có duy nhất một chương trình được phép chạy và sử dụng toàn bộ tài nguyên hệ thống. Các hệ thống máy tính hiện đại ngày nay, ngay cả đối với máy tính cá nhân chỉ một người sử dụng, đều cho phép tải nhiều chương trình vào bộ nhớ trong để thực thi đồng thời. Như vậy, cần có cơ chế đan xen hoạt động của các chương trình khác nhau. Nhu cầu này dẫn đến sự xuất hiện khái niệm tiến trình - là chương trình đang trong quá trình thực thi. Tiến trình là đơn vị thực thi cơ sở trong hệ thống chia sẻ thời gian thực. Ngoài ra, HĐH phải có cơ chế cấp phát tài nguyên cho tiến trình theo cơ chế định trước và cơ chế cho phép tiến trình trao đổi thông tin với nhau. Như vậy, dù công việc chính vẫn là thực thi chương trình người dùng, nhưng HĐH phải kiểm soát nhiều công việc khác của hệ thống. Nội dung của chương này trình bày về tiến trình, trạng thái của tiến trình và cách thức HĐH quản lý tiến trình.

4.1. TIẾN TRÌNH VÀ TRẠNG THÁI TIẾN TRÌNH

Thế nào là hoạt động của CPU? Hệ thống phân chia theo lô (batch system) gọi là công việc (job), trong khi hệ thống chia sẻ theo thời gian thực (time-shared system) gọi là chương trình người dùng (user programs) hay tác vụ (task). Ngay trong hệ thống đơn nhiệm như MS-DOS và Macintosh, một người dùng cũng có thể chạy đồng thời vài chương trình. Thậm chí khi ứng dụng nào đó đang chạy, thì HĐH vẫn phải thực thi chương trình bên trong của mình. Ở khía cạnh nào đó, tất cả các hoạt động trên khá giống nhau và chúng ta gọi là tiến trình.

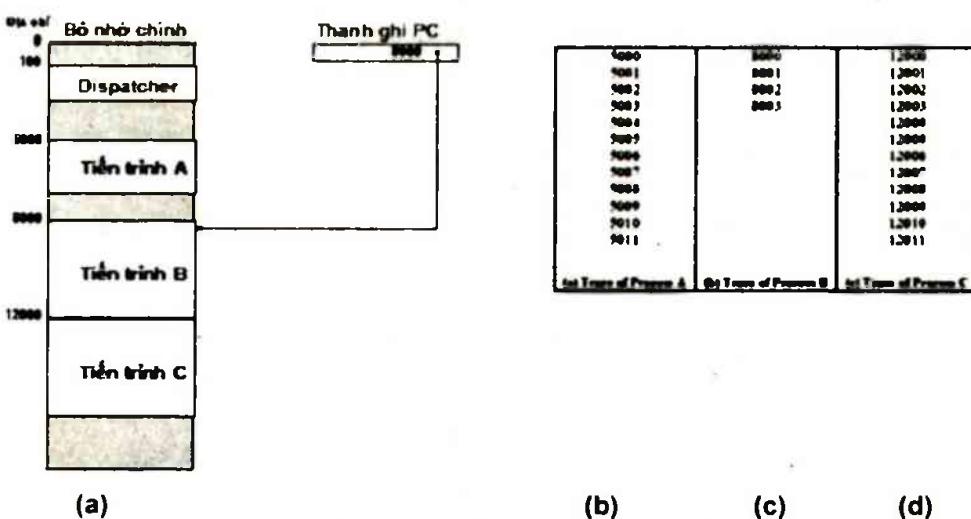
4.1.1. Tiến trình

Thông thường, tiến trình được xem là chương trình đang hoạt động. Quá trình thực thi tiến trình diễn ra tuần tự (tại một thời điểm chỉ có duy nhất một chỉ thị trong tiến trình được thực hiện).



Hình 4.1. Tiến trình

Chương trình khác với tiến trình: Chương trình là thực thể tĩnh (nội dung file exe, com ghi trên ổ đĩa cứng), trong khi tiến trình là thực thể động, với con trỏ chương trình mô tả chỉ thị sẽ được thực hiện kế tiếp và các tài nguyên được cấp phát trong quá trình thực thi. Trong Hình 4.1, chúng ta thấy có ba tiến trình là P_i , P_j và P_k . Mỗi tiến trình nằm ở vị trí khác nhau trong bộ nhớ và trạng thái tiến trình biến đổi liên tục theo thời gian. Mặc dù hai tiến trình có thể cùng là của một chương trình (chung khối mã), nhưng chúng được thực thi độc lập với nhau. Bên cạnh đó, tiến trình có thể sinh ra nhiều tiến trình con (mục 4.2).



Hình 4.2. Ba tiến trình A, B và C trong Bộ nhớ

4.1.2. Trạng thái tiến trình

Trong Hình 4.2a, ba tiến trình A, B và C lần lượt nằm ở vùng nhớ có địa chỉ 5000, 8000 và 12000. Như vậy, luồng thực thi của tiến trình A sẽ là 5000, 5001, 5002,... (Hình 4.2b). Tương tự với B và C, nhưng khi xen kẽ hoạt động, các tiến trình A, B, C và tiến trình của HĐH sẽ diễn ra như trong Hình 4.3. Để có thể thiết kế HĐH, cần mô hình hóa hành vi tiến trình.

Thời gian	Chỉ thị	Thời gian	Chỉ thị
1	5000	27	12004
2	5001	28	12005
3	5002		----- Hết giờ
4	5003	29	100
5	5004	30	101
6	5005	31	102
	----- Hết giờ	32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002		----- Hết giờ
16	8003	41	100
	----- Yêu cầu vào/ra	42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
			----- Hết giờ

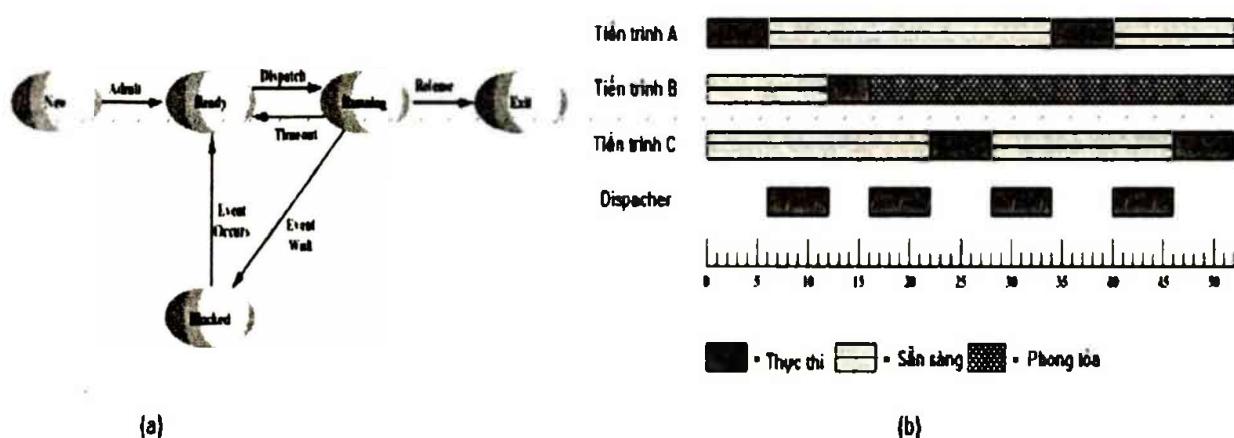
Hình 4.3. Các tiến trình hoạt động xen kẽ

Tiến trình liên tục thay đổi trạng thái trong quá trình thực thi. Tiến trình có thể ở một trong các trạng thái sau:

- **Tạo mới (New):** Tiến trình vừa được tạo ra.
- **Đang thực thi (Running):** Các chỉ thị của tiến trình lần lượt được thực thi.
- **Phong tỏa (Blocked):** Tiến trình chờ đợi sự kiện nào đó để chạy tiếp (ví dụ như hoàn tất thao tác vào/ra hoặc đợi tín hiệu).
- **Sẵn sàng (Ready):** Chờ để được sử dụng CPU. Đã có đủ tất cả các tài nguyên cần thiết.

- **Kết thúc (Exit):** Hoàn tất việc thực thi.

Chú ý, tên trạng thái trên các HĐH có thể khác nhau. Tuy nhiên, 5 trạng thái nêu trên xuất hiện trên mọi hệ thống. Tất nhiên, trên HĐH thực thì số lượng các trạng thái lớn hơn nhiều. Chú ý rằng, trên hệ thống có một CPU thì tại một thời điểm, chỉ có duy nhất một tiến trình sử dụng CPU (đang thực thi), trong khi các tiến trình còn lại ở trạng thái sẵn sàng hay chờ. Lược đồ trạng thái được minh họa trong Hình 4.4. Với các tiến trình minh họa trong Hình 4.3, quá trình biến đổi trạng thái của A, B và C được minh họa trong Hình 4.4b.



Hình 4.4. Mô hình trạng thái của tiến trình

4.2. THAO TÁC TRÊN TIẾN TRÌNH

HĐH cần có cơ chế tạo mới và chấm dứt hoạt động của tiến trình.

4.2.1. Tạo mới tiến trình

Khi tiến trình mới được đưa vào hệ thống, HĐH tạo ra cấu trúc dữ liệu (được trình bày ở phần sau) để quản lý tiến trình. Đồng thời, HĐH cấp phát một loạt tài nguyên cần thiết cho tiến trình hoạt động (chẳng hạn bộ nhớ). Có bốn nguyên nhân tạo mới tiến trình:

- Trong hệ thống lô, khi có công việc được đưa vào, hệ thống tạo một tiến trình để thực hiện công việc.
- Trong hệ thống tương tác, khi người dùng đăng nhập hệ thống.
- HĐH tạo mới tiến trình để thực hiện công việc nào đó. Ví dụ, khi người dùng in văn bản, HĐH tạo tiến trình thực hiện việc in. Điều

này cho phép tiến trình người dùng có thể ngay lập tức tiếp tục thực hiện công việc mà không cần quan tâm khi nào tiến trình in thực hiện xong.

- Tiến trình tạo ra tiến trình con.

Việc tạo mới trong 3 phương pháp đầu "trong suốt" với người dùng. Phương pháp thứ 4 cho phép người dùng tạo ra các tiến trình mới một cách linh hoạt. Ví dụ, tiến trình phục vụ máy in có thể tạo ra các tiến trình con phục vụ mỗi yêu cầu in.

Thông thường, tiến trình cần lượng tài nguyên nhất định (thời gian sử dụng CPU, bộ nhớ, file, các thiết bị vào/ra) để thực hiện công việc. Khi mới được tạo ra, tiến trình con có thể lấy tài nguyên trực tiếp từ HĐH, hoặc chỉ có thể lấy tài nguyên từ tiến trình cha. Tiến trình cha có thể phân chia tài nguyên của mình cho tiến trình con, hoặc chia sẻ các tài nguyên (như file, bộ nhớ) với các tiến trình con. Hạn chế tiến trình con chỉ được sử dụng một phần tài nguyên của tiến trình cha, ngăn cản việc một tiến trình cố tình sinh ra quá nhiều tiến trình con khiến hệ thống quá tải.

Bên cạnh việc cấp phát tài nguyên, tiến trình cha có thể truyền các tham số khởi tạo cho tiến trình con.

Khi tiến trình sinh ra tiến trình con thì:

- Tiến trình cha tiếp tục chạy cùng với tiến trình con.
- Tiến trình cha chờ cho đến khi một vài hoặc tất cả các tiến trình con kết thúc.

Xét về khía cạnh không gian địa chỉ cũng có hai trường hợp sau:

- Tiến trình con là bản sao của tiến trình cha.
- Tiến trình con tái môi chương trình khác để thực hiện.

Để mô tả sự khác biệt, xét HĐH UNIX. Trong UNIX, mỗi tiến trình được xác định qua định danh tiến trình. Tiến trình mới được tạo ra qua lời gọi hệ thống **fork()**. Không gian tiến trình con là bản sao của không gian tiến trình cha. Cơ chế này cho phép tiến trình cha có thể trao đổi dễ dàng với tiến trình con. Cả hai tiến trình cha và con tiếp tục thực thi chỉ thị đứng ngay sau **fork()**. Tuy nhiên, với tiến trình con, giá trị trả về của **fork()** là 0, còn với tiến trình cha, giá trị trả về là định danh tiến trình con. Lời gọi hệ thống **execve()**

thường được gọi ngay sau **fork()** để thay thế không gian bộ nhớ bằng một chương trình mới, tức là tải một file nhị phân mới vào bộ nhớ (xóa bỏ nội dung bộ nhớ của chương trình gọi **execve()** và sau đó thực thi chương trình mới được tải). Tiền trình cha có thể tạo nhiều tiền trình con khác, hoặc không làm gì khi các tiền trình con đang thực thi, hoặc có thể sử dụng lời gọi **wait()** để tự loại khỏi hàng đợi sẵn sàng cho đến khi tiền trình con kết thúc.

4.2.2. Kết thúc tiền trình

Có nhiều nguyên nhân kết thúc tiền trình:

1. Tiền trình kết thúc khi thực hiện xong chỉ thị cuối cùng và yêu cầu HDH loại bỏ qua lời gọi hệ thống **exit()**. Lúc này tiền trình con có thể trả dữ liệu cho tiền trình cha. Tất cả các tài nguyên của tiền trình, bao gồm tài nguyên vật lý và bộ nhớ ảo, các file đang mở, các bộ đệm vào/ra sẽ được HDH thu hồi.

2. Tiền trình có thể bị một tiền trình khác chấm dứt qua lời gọi hệ thống phù hợp (chẳng hạn **abort()**). Thường chỉ tiền trình cha mới được quyền thực hiện lời gọi này để chấm dứt hoạt động của tiền trình con (nếu không thì người dùng có thể kết thúc công việc của người khác). Chú ý, tiền trình cha cần phải biết được định danh của các tiền trình con. Chính vì thế, khi tạo ra một tiền trình mới, hệ thống trả định danh tiền trình con cho tiền trình cha.

Tiền trình cha có thể kết thúc một tiền trình con vì các lý do sau đây:

- Tiền trình con chiếm dụng nhiều tài nguyên, vượt quá số lượng được cấp phát.
- Nhiệm vụ của tiền trình con không còn cần thiết nữa.
- Tiền trình cha kết thúc và HDH không cho phép tiền trình con của nó chạy tiếp.

Trong trường hợp đầu, tiền trình cha cần có cơ chế xác định trạng thái của tiền trình con. Nhiều hệ thống, ví dụ như HDH VMS không cho phép tiền trình con tồn tại nếu tiền trình cha đã kết thúc. Trong những hệ thống này, nếu tiền trình kết thúc (một cách bình thường hay bất thường), thì tất cả các tiền trình con phải được kết thúc. Hiện tượng này được gọi là kết thúc dạng tháp và được HDH tự động thực hiện.

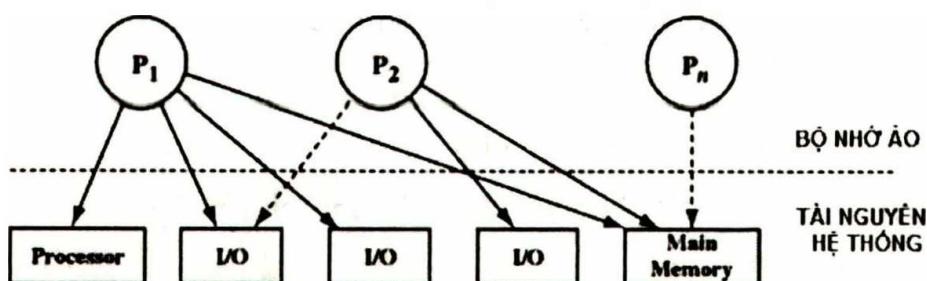
3. Tiến trình đòi hỏi lượng bộ nhớ quá lớn mà hệ thống không đáp ứng được.

4. Tiến trình truy cập tới các vùng bộ nhớ bị cấm.
5. Tiến trình sử dụng tài nguyên không hợp lệ.
6. Tiến trình bị lỗi vào/ra, chẳng hạn đọc một đĩa mềm hỏng.

Để hình dung việc bắt đầu và kết thúc của tiến trình, xét HĐH UNIX. Trong UNIX, tiến trình có thể kết thúc bằng cách gọi **exit** và tiến trình cha có thể chờ sự kiện này bằng lời gọi hệ thống **wait**. **wait()** trả về cho tiến trình cha định danh của tiến trình con đã kết thúc. Nếu tiến trình cha kết thúc thì tất cả tiến trình con cũng bị HĐH chấm dứt. Nếu không có tiến trình cha, UNIX không biết phải báo trạng thái hoạt động của tiến trình con cho tiến trình nào.

4.3. MÔ TẢ TIẾN TRÌNH

HĐH quản lý tài nguyên hệ thống (bộ nhớ, CPU, thiết bị vào/ra,...) và cấp phát khi tiến trình yêu cầu. Ví dụ, hệ thống đa chương trình trên Hình 4.5, P_1 đang thực thi và nắm quyền kiểm soát hai thiết bị vào/ra; P_2 nắm trong bộ nhớ nhưng ở chế độ phong tỏa vì đợi một thiết bị vào/ra đã được cấp phát cho P_1 ; P_n đang ở trạng thái treo vì bị chuyển ra ổ đĩa cứng. Vấn đề đặt ra ở đây là làm thế nào để HĐH quản lý được các tiến trình cũng như tài nguyên đã được cấp phát?

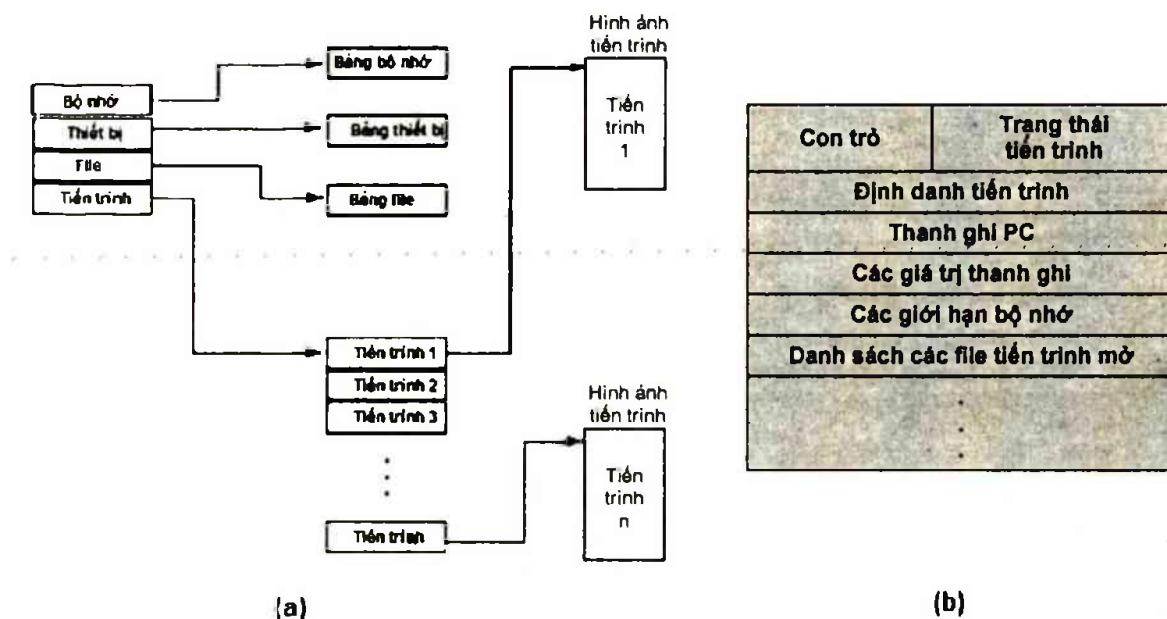


Hình 4.5. Tiến trình và tài nguyên

4.3.1. Khối điều khiển tiến trình (PCB)

Để nắm được thông tin về đối tượng mình quản lý, HĐH thường tạo ra các bảng ghi trạng thái hiện thời về từng đối tượng. Ý tưởng này được minh họa trong Hình 4.6. HĐH thường có 4 bảng để quản lý bộ nhớ, thiết bị, file

và tiến trình. Bảng quản lý bộ nhớ ghi thông tin về bộ nhớ cấp phát cho tiến trình, các thuộc tính bảo vệ của vùng bộ nhớ (trình bày trong Chương 8 và 9). Bảng vào/ra chứa thông tin về thiết bị. Chẳng hạn, thiết bị còn rỗi hay đã cấp phát cho tiến trình nào. Nếu đang thực hiện một thao tác vào/ra thì HĐH cần xác định trạng thái của thao tác. Bảng file cung cấp các thông tin về vị trí của file trên ổ đĩa cứng. Cuối cùng là bảng theo dõi tất cả các tiến trình. Bốn bảng này có quan hệ chặt chẽ với nhau.



Hình 4.6. Các bảng quản lý trong HĐH

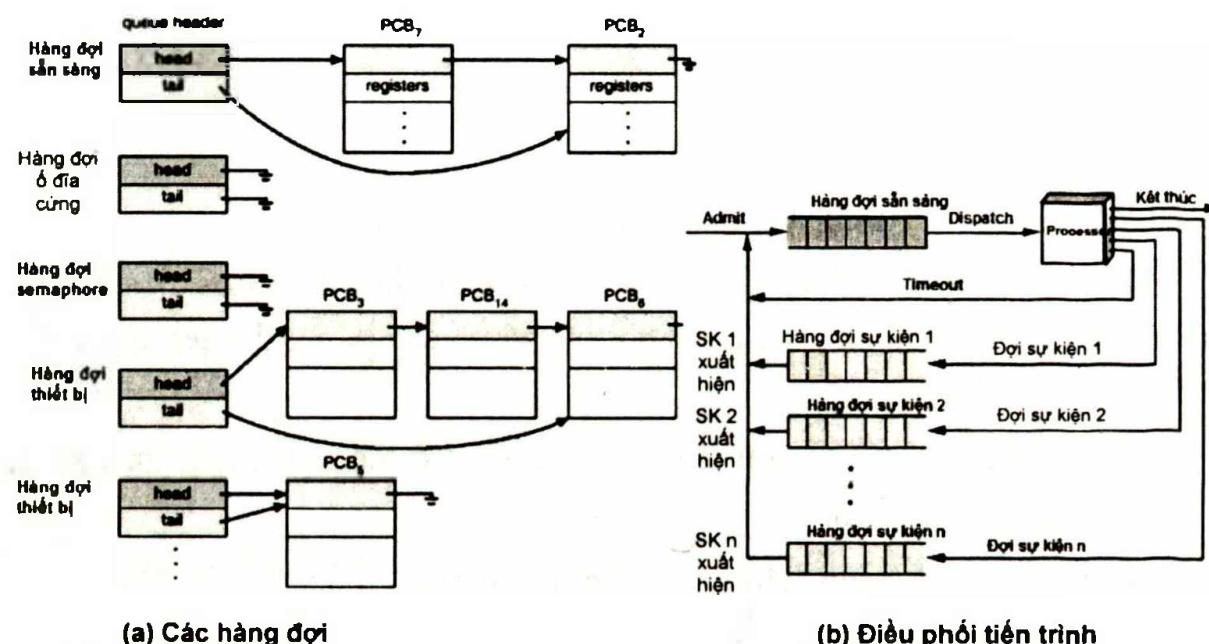
Tiến trình được HĐH kiểm soát thông qua khối điều khiển tiến trình (Process Control Block – PCB). Khối này chứa các thông tin liên quan đến tiến trình sau đây:

- **Trạng thái tiến trình (Process state):** Có thể là tạo mới, sẵn sàng, đang chạy, phong tỏa hay kết thúc.
- **Con trỏ chương trình (program counter):** Chứa địa chỉ của chỉ thị sẽ được thực thi kế tiếp.
- **Các thanh ghi CPU (CPU register):** Số lượng các thanh ghi phụ thuộc vào kiến trúc máy tính. Các thanh ghi này có thể cho phép người dùng truy cập (thanh ghi tích lũy, thanh ghi chỉ mục, thanh ghi ngăn xếp, các thanh ghi đa chức năng, thanh ghi trạng thái hay còn gọi là thanh ghi cờ) cùng các thanh ghi đặc biệt dùng cho quá trình kiểm soát của HĐH.

- Thông tin về điều phối CPU:** Quyền ưu tiên của tiến trình, con trỏ đến hàng đợi điều phối và các tham số điều phối khác.
- Thông tin quản lý bộ nhớ:** Giá trị thanh ghi cơ sở và thanh ghi giới hạn; bảng trang, bảng đoạn. Thông tin này phụ thuộc vào hệ thống quản lý bộ nhớ (xem Chương 9).
- Trạng thái vào/ra:** Danh sách các thiết bị bị vào/ra tiến trình đang sử dụng, danh sách các file đang mở.

4.3.2. Điều phối tiến trình

Mục tiêu đa nhiệm là tận dụng tối đa CPU bằng cách cho phép nhiều tiến trình chạy đồng thời. Trên hệ thống có một CPU, hai tiến trình không thể thực thi cùng lúc. Nếu có nhiều tiến trình thì các tiến trình phải đợi đến khi CPU rỗng. Như vậy, có hai vấn đề được đặt ra: Khi nào chuyển quyền sử dụng CPU của tiến trình (gọi tắt là chuyển quyền) và HĐH cần thực hiện công việc gì?



Hình 4.7. Các hàng đợi và việc điều phối

Ngay khi tạo ra, tiến trình được đặt vào hàng đợi công việc (job queue). Hàng đợi này chứa tất cả các tiến trình trong hệ thống. Những tiến trình đã nằm trong bộ nhớ và sẵn sàng sử dụng CPU được đặt trong hàng đợi sẵn sàng. Hàng đợi này thường được cài đặt dưới dạng danh sách liên kết. Header của hàng đợi chứa hai con trỏ, trỏ đến phần tử PCB đầu và PCB cuối

danh sách. PCB có trường con trỏ, trỏ đến tiến trình kế tiếp trong hàng đợi. Trong hệ thống có thể có nhiều hàng đợi. Khi chiếm dụng CPU, tiến trình thực thi trong một khoảng thời gian nào đó trước khi kết thúc hoặc tạm dừng (do rất nhiều nguyên nhân như đợi một sự kiện cụ thể, đợi yêu cầu vào/ra, cố tình truy cập vào vùng nhớ không được phép,...). Trường hợp yêu cầu vào/ra, tiến trình có thể phải đợi thao tác trên thiết bị hoàn thành. Khi hệ thống có nhiều tiến trình, thiết bị có thể trong trạng thái bận do phải phục vụ nhiều yêu cầu. Do đó, tiến trình buộc phải chờ cho đến khi thiết bị rỗi. Danh sách các tiến trình chờ đợi thiết bị vào/ra gọi là hàng đợi thiết bị. Mỗi thiết bị có hàng đợi riêng (Hình 4.7a).

Trong điều phối tiến trình, sơ đồ hàng đợi là kỹ thuật biểu diễn thông dụng (Hình 4.7b). Ở đây ta thấy có hai loại hàng đợi là hàng đợi sẵn sàng và các hàng đợi thiết bị.

Tiến trình mới sẽ được đặt vào hàng đợi sẵn sàng. Tiến trình sẽ chờ đợi cho đến khi được quyền sử dụng CPU. Khi đang thực thi, nếu phải đợi một sự kiện nào đó, tiến trình chuyển sang hàng đợi ứng với sự kiện tương ứng. Các sự kiện có thể xảy ra là:

- Tiến trình yêu cầu thực hiện vào/ra.
- Tiến trình có thể tạo ra các tiến trình con và chờ cho đến khi tiến trình con kết thúc.
- Tiến trình có thể bị chiếm đoạt quyền sử dụng CPU (chẳng hạn do ngắn). Sau đó tiến trình được đưa trở lại hàng đợi sẵn sàng.

Trong hai trường hợp đầu, tiến trình chuyển từ trạng thái đang thực thi sang trạng thái phong tỏa, sau đó lại quay trở lại trạng thái sẵn sàng. Tiến trình tiếp tục chuyển trạng thái một cách tuần hoàn như vậy cho đến khi kết thúc. Lúc này tiến trình sẽ bị xóa khỏi tất cả các hàng đợi và PCB của tiến trình cũng như mọi tài nguyên được cấp phát bị HĐH thu hồi.

4.3.3. Bộ điều phối

Tiến trình dịch chuyển giữa các hàng đợi trong suốt thời gian sử dụng của mình. HĐH có nhiệm vụ di chuyển tiến trình giữa các hàng đợi. Công việc này được thực hiện bởi bộ điều phối. Trong hệ thống lô, số lượng các tiến trình từ bên ngoài đưa vào hệ thống thường nhiều hơn số lượng các tiến trình có thể thực thi ngay lập tức. Các tiến trình này được lưu ở đâu đó

(thường là ổ đĩa) để thực thi sau. Bộ điều phối dài hạn lựa chọn các tiến trình từ nhóm này để tải vào bộ nhớ. Bộ điều phối ngắn hạn cấp phát CPU cho một tiến trình nào đó sẵn sàng thực thi đã nằm trong bộ nhớ.

Điểm khác biệt cơ bản giữa hai bộ điều phối trên là tần suất hoạt động. Bộ điều phối ngắn hạn chọn một tiến trình nắm quyền điều khiển CPU. Một tiến trình có thể chạy trong vài mili giây (ms) trước khi tạm dừng đợi yêu cầu vào/ra hoàn thành. Thường thì trong 100ms, bộ điều phối ngắn hạn chạy một lần. Vì chạy giữa hai lần thực thi, nên bộ điều phối ngắn hạn cần phải chạy cực nhanh. Nếu mỗi lần mất 10ms để quyết định xem tiến trình nào được chạy tiếp trong khoảng 100ms tiếp theo thì khi đó $10/(100 + 10) \approx 9\%$ CPU bị lãng phí cho công việc điều phối tiến trình. Tần suất chạy của bộ điều phối dài hạn thấp hơn nhiều, có thể khoảng thời gian giữa hai lần tạo mới hai tiến trình kế tiếp là một vài phút. Bộ điều phối dài hạn phải kiểm soát số lượng các tiến trình nằm trong bộ nhớ. Nếu số lượng này ổn định thì trung bình số lượng tiến trình được tạo mới xấp xỉ số lượng tiến trình rời khỏi hệ thống. Như vậy, bộ điều phối dài hạn được gọi chỉ khi có tiến trình kết thúc. Vì khoảng thời gian giữa hai lần chạy là khá dài nên bộ điều phối dài hạn có nhiều thời gian lựa chọn tiến trình nào được tải vào bộ nhớ.

Bộ điều phối dài hạn có vai trò quan trọng. Thường tiến trình có đặc điểm hướng vào/ra hoặc hướng tính toán. Tiến trình hướng vào/ra thường tốn thời gian thực hiện vào/ra hơn thời gian tính toán trên CPU. Ngược lại, tiến trình hướng tính toán có ít yêu cầu vào/ra mà phần lớn thời gian sử dụng CPU. Một điều quan trọng là bộ điều phối dài hạn phải chọn một nhóm hỗn hợp các tiến trình hướng vào/ra và hướng tính toán. Hệ thống chỉ đạt được hiệu suất cao nhất khi có sự kết hợp giữa các tiến trình hướng vào/ra và các tiến trình hướng tính toán.

Một vài hệ thống có thể không có bộ điều phối dài hạn, ví dụ, hệ thống chia sẻ thời gian thực. Sự ổn định của những hệ thống này phụ thuộc vào khả năng của phần cứng hoặc sự tự điều chỉnh của người dùng. Nếu hiệu suất hệ thống giảm đến mức quá thấp, cách giải quyết đơn giản là một vài người dùng thoát khỏi hệ thống.

Một số HĐH chia sẻ thời gian thực có thể có thêm một bộ điều phối trung gian, gọi là bộ điều phối trung hạn, có nhiệm vụ chuyển bớt các tiến trình trong bộ nhớ ra ổ đĩa cứng để giảm mức đa chương trình. Sau đó, tiến

trình có thể được đưa trở lại bộ nhớ để khôi phục việc thực thi. Phương pháp này gọi là sự hoán chuyển và sẽ được trình bày trong Chương 11.

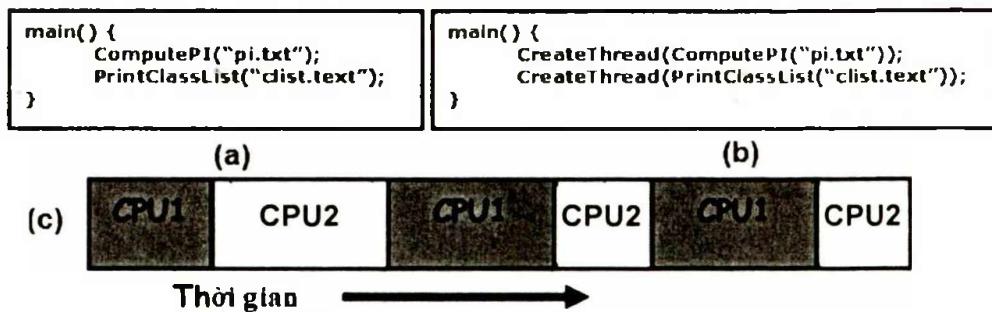
4.3.4. Chuyển ngữ cảnh

Muốn chuyển đổi quyền sử dụng CPU giữa các tiến trình thì phải lưu lại trạng thái tiến trình cũ và nạp lại trạng thái tiến trình mới. Quá trình này được gọi là chuyển ngữ cảnh (context switch). Thời gian thực hiện chuyển ngữ cảnh hoàn toàn lãng phí, vì hệ thống không làm công việc hữu ích. Tốc độ chuyển ngữ cảnh phụ thuộc vào thông số của từng máy tính như: tốc độ bộ nhớ, số lượng các thanh ghi phải lưu giữ và liệu máy tính có các chỉ thị đặc biệt hỗ trợ chuyển ngữ cảnh (chẳng hạn, có chỉ thị để nạp và lưu tất cả các thanh ghi cùng một lúc) hay không. Thường tốc độ này nằm trong khoảng từ 1 đến 1000ms.

Thời gian chuyển ngữ cảnh phụ thuộc vào sự hỗ trợ từ phần cứng. Ví dụ, một số bộ vi xử lý như DECSYSTEM-20 có nhiều tập hợp các thanh ghi. Chuyển ngữ cảnh đơn giản chỉ là chuyển con trỏ đến tập hợp các thanh ghi hiện thời. Tất nhiên, nếu số lượng các tiến trình lớn hơn số tập các thanh ghi thì hệ thống cũng phải lưu trữ giá trị của thanh ghi lên bộ nhớ giống như trên. HĐH càng phức tạp thì công việc chuyển ngữ cảnh càng nhiều. Trong Chương 9 (Quản lý bộ nhớ) ta thấy có thể phải di chuyển khá nhiều dữ liệu trong mỗi lần chuyển ngữ cảnh. Ví dụ, không gian bộ nhớ của tiến trình hiện tại cần phải được lưu giữ cũng như phải chuẩn bị không gian bộ nhớ cho tiến trình tiếp theo. Cách thức lưu giữ không gian này phụ thuộc vào phương thức quản lý bộ nhớ của HĐH. Nói chung, chuyển ngữ cảnh là "nút cổ chai", làm giảm đáng kể hiệu suất hệ thống, nên người ta hay sử dụng các luồng.

4.4. LUÔNG

Trong các HĐH trước đây, tiến trình được cấp phát lượng tài nguyên và bộ nhớ riêng. Tuy nhiên, trong nhiều trường hợp, việc sử dụng chung tài nguyên giữa các tiến trình hoạt động đồng thời lại có nhiều ưu điểm hơn, ví dụ, chương trình Web server có thể phục vụ đồng thời nhiều yêu cầu khác nhau. Điều này cũng tương tự lời gọi hệ thống `fork()` tạo ra một luồng chương trình mới, tức là có con trỏ chương trình mới, nhưng thực thi công việc trên cùng một không gian địa chỉ.



Hình 4.8. Ưu điểm của luồng

Xét chương trình được viết bằng ngôn ngữ C trong Hình 4.8a, chương trình có hai hàm, ComputePI sẽ tính giá trị số π và viết vào file pi.txt, PrintClassList in danh sách lớp từ file clist.txt. Vì π là giá trị vô hạn, nên ComputePI chạy vô tận, và như thế PrintClassList sẽ không bao giờ được dùng. Tuy nhiên, nếu cài tiến chương trình như trong Hình 4.8b, chương trình sẽ có hai thread chạy đồng thời, thread thứ nhất chạy ComputePI và thread thứ hai chạy PrintClassList. Hai hàm này luân phiên nhau sử dụng CPU như trong Hình 4.8c, do đó hàm thứ 2 không bị trì hoãn vĩnh viễn. Qua ví dụ trên, chúng ta thấy khái niệm luồng có rất nhiều ưu điểm, nên phần lớn các HĐH hiện đại đều cài đặt công nghệ luồng.

Cấu trúc luồng:

Luồng đôi khi được gọi là tiểu trình (Light Weight Process), là đơn vị cơ sở trong việc sử dụng CPU, bao gồm con trỏ chương trình, tập các thanh ghi và ngăn xếp. Luồng cùng chia sẻ đoạn mã chương trình, dữ liệu và một số tài nguyên. Các luồng hợp tác với nhau để thực hiện một tác vụ. Trong mô hình này, tiến trình theo định nghĩa trước tương đương với tác vụ được thực thi bởi một luồng duy nhất. Tác vụ không có luồng sẽ không thể hoạt động và luồng phải nằm trong duy nhất một tác vụ nào đó. Các luồng luân phiên nhau sử dụng CPU và chi phí việc tạo luồng mới thấp hơn rất nhiều so với việc tạo mới tiến trình. Mặc dù chuyển ngữ cảnh giữa các luồng vẫn yêu cầu thay đổi tập các thanh ghi, nhưng không cần gọi chức năng quản lý bộ nhớ. Một số hệ thống triển khai công nghệ luồng thông qua thư viện mức người dùng thay vì sử dụng lời gọi hệ thống, bởi vậy việc chuyển ngữ cảnh luồng không cần gọi đến HĐH hay làm gián đoạn kernel. Chuyển ngữ cảnh giữa luồng ở mức người dùng diễn ra rất nhanh và không cần sự can thiệp của HĐH, điều này giúp rút ngắn tốc độ thực hiện công việc. Do vậy, đây là giải pháp hợp lý khi phải xây dựng chương trình cần phục vụ nhiều yêu cầu

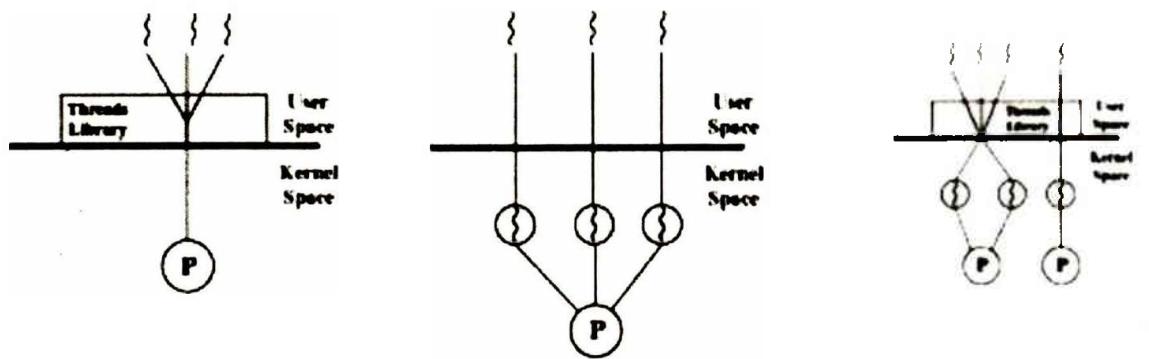
cùng lúc. Tuy nhiên, luồng ở mức người dùng có một số mặt hạn chế. Trong trường hợp phần nhân (kernel) là một luồng duy nhất, thì khi bất kỳ luồng ở mức người dùng thực hiện lời gọi hệ thống sẽ khiến toàn bộ tác vụ phải đợi đến khi lời gọi hệ thống trả lại kết quả.

Chúng ta so sánh mô hình luồng với mô hình tiến trình. Đối với đa tiến trình thì các tiến trình hoạt động độc lập với nhau, mỗi tiến trình có con trỏ chương trình, hệ thống ngăn xếp và không gian địa chỉ riêng. Kiểu tổ chức này có lợi khi công việc tiến trình thực hiện độc lập với nhau. Ví dụ, đối với hệ thống có một CPU, tiến trình file server sẽ bị dừng hoạt động (trạng thái phong tỏa) khi đang đợi đọc ô đĩa. Hiệu suất hệ thống sẽ được cải thiện nếu các tiến trình file server khác vẫn có thể làm việc trong khi tiến trình kia bị phong tỏa vì các tiến trình này có cùng chương trình.

Xét trên nhiều mặt, hoạt động của luồng giống tiến trình. Luồng có thể ở một trong các trạng thái như sẵn sàng, phong tỏa, thực thi hoặc kết thúc. Giống tiến trình, các luồng cùng nhau chia sẻ CPU và tại một thời điểm chỉ một luồng duy nhất được quyền thực thi. Luồng trong một tiến trình được thực thi tuần tự và mỗi luồng có ngăn xếp, con trỏ đếm chương trình riêng. Luồng cũng có thể tạo ra luồng con, có thể bị phong tỏa và tạm dừng do đợi thực hiện thao tác vào/ra; trong khi một luồng đang ở trạng thái phong tỏa thì các luồng khác được phép thực thi. Tuy nhiên, điểm khác biệt ở đây là các luồng có quan hệ với nhau. Vì luồng có thể truy cập đến bất kỳ địa chỉ nào trong không gian địa chỉ tiến trình, nên có thể đọc hay viết vào ngăn xếp của luồng khác. Do đó, không có cơ chế bảo vệ ngăn cách giữa các luồng và thực sự cơ chế này cũng không cần thiết.

Bây giờ quay lại ví dụ tiến trình file server bị phong tỏa trong mô hình tiến trình. Trong trường hợp này, không tiến trình phục vụ nào khác được thực thi cho đến khi tiến trình trên được giải tỏa. Ngược lại, nếu tác vụ có nhiều luồng khác nhau, thì khi một luồng phục vụ bị phong tỏa thì ngay lập tức luồng thứ hai thực hiện cùng một nhiệm vụ có thể được khởi động để thực thi. Trong ứng dụng này, việc kết hợp nhiều luồng thực hiện cùng một công việc làm tăng thông lượng cũng như hiệu suất hệ thống. Các ứng dụng khác, chẳng hạn bài toán sản xuất – tiêu thụ cài đặt bằng cách chia sẻ bộ đệm cũng có thể tận dụng ưu điểm luồng. Thành phần sản xuất hay tiêu thụ có thể là các luồng của cùng một tiến trình. Khi đó chi phí phụ trội cho việc chuyển ngữ cảnh được giảm thiểu và hiệu suất hệ thống được nâng cao.

Để minh họa ưu điểm sự thực thi đồng thời trong mô hình luồng, giả sử cần phải viết file server trong hệ thống không hỗ trợ luồng, khi đó server phải hoàn thành xong một yêu cầu nào đó rồi mới có thể phục vụ yêu cầu kế tiếp. Nếu như yêu cầu thứ nhất phải đợi để đọc ổ đĩa cứng thì CPU hoàn toàn lãng phí trong khoảng thời gian này. Vì vậy, số yêu cầu thực hiện trung bình tính trong 1 giây sẽ ít hơn rất nhiều so với trường hợp đa luồng xử lý song song. Nếu không có công nghệ luồng, người thiết kế hệ thống phải tìm cách giảm thiểu việc suy giảm hiệu suất bằng cách bắt chước cấu trúc hoạt động song song của hệ thống luồng thông qua việc sử dụng các tiến trình truyền thống. Khi đó cấu trúc của chương trình tuần tự trở nên phức tạp.



(a) Luồng ở mức người dùng

(b) Luồng ở mức nhân

(c) Giải pháp kết hợp

Hình 4.9. Luồng ở các mức khác nhau

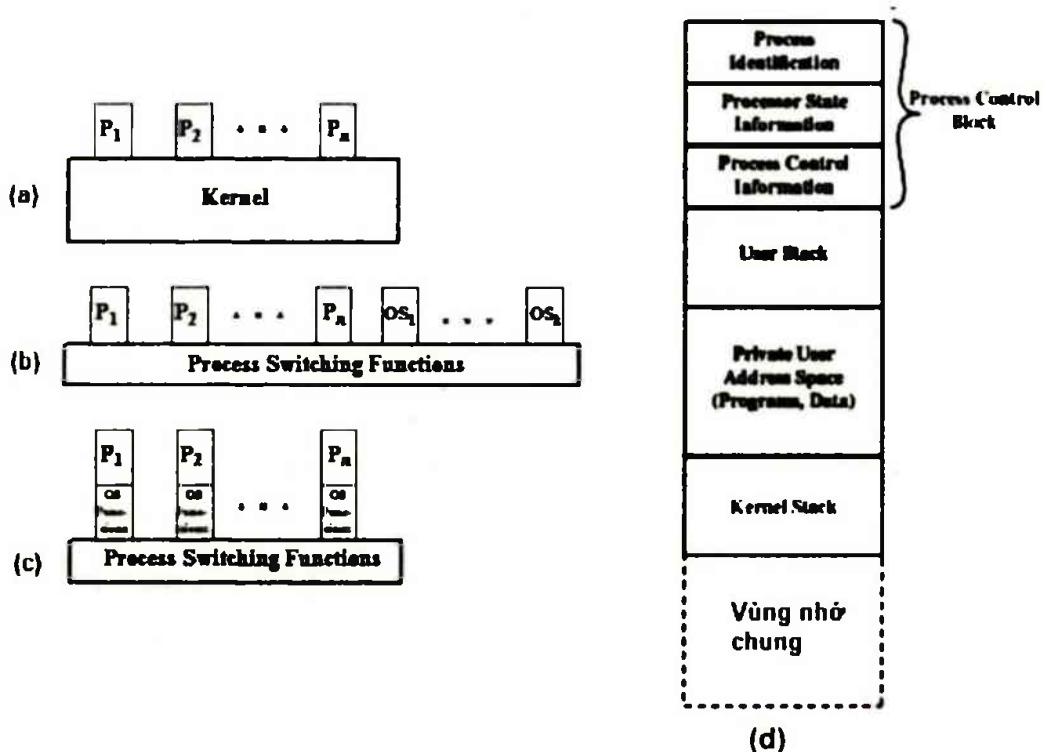
Có nhiều kiểu luồng khác nhau như luồng có thể được kernel hỗ trợ (HĐH Mach và OS/2). HĐH cung cấp các lời gọi hệ thống cho luồng tương tự lời gọi hệ thống thao tác trên tiến trình (Hình 4.9b). Một phương pháp khác là luồng hỗ trợ ở mức người dùng, thông qua lời gọi các hàm thư viện (Project Andrew của CMU) (Hình 4.9a). Luồng ở mức người dùng không liên quan đến kernel, do đó chuyển ngữ cảnh sẽ nhanh hơn nhiều so với hệ thống luồng do kernel hỗ trợ. Tuy nhiên, bất kỳ lời gọi nào tới HĐH sẽ khiến toàn bộ các luồng của cùng tiến trình phải đợi vì kernel chỉ điều phối tiến trình (kernel không biết khái niệm luồng) và tiến trình trong trạng thái đợi không được cấp phát CPU. Điều phối có thể không công bằng. Xét hai tiến trình a và b: a có 1 và b có 100 luồng. Vì mỗi tiến trình cùng nhận được một lượng tử thời gian như nhau, nên luồng của tiến trình a sẽ chạy nhanh hơn luồng của tiến trình b 100 lần. Đối với hệ thống mà kernel hỗ trợ luồng, việc chuyển ngữ cảnh giữa các luồng tốn thời gian hơn vì chính kernel phải thực hiện việc chuyển ngữ cảnh (thông qua ngắn). Tuy nhiên, các luồng

được thực hiện một cách độc lập, nên thời gian tiến trình b nhận được bằng 100 lần thời gian mà tiến trình a nhận được. Không những thế, tiến trình b có thể đồng thời gọi 100 lời gọi hệ thống, nhiều hơn rất nhiều so với trường hợp chính tiến trình này chạy trên hệ thống hỗ trợ luồng ở mức người dùng.

Trong một số hệ thống (HĐH Solaris 2), người ta cài đặt cả hai cơ chế hỗ trợ luồng để nâng cao hiệu suất (Hình 4.9c).

4.5. CÀI ĐẶT HỆ ĐIỀU HÀNH

Hoạt động của HĐH giống hoạt động của bất kỳ phần mềm nào khác, tức là một chương trình nào đó chạy trên CPU, vậy HĐH cũng phải chia sẻ quyền sử dụng CPU với các tiến trình khác. Tuy nhiên, HĐH là một tập hợp các tiến trình hay là một tiến trình? Hình 4.10 minh họa các phương pháp cài đặt HĐH.



Hình 4.10. Các phương pháp cài đặt HĐH

Kernel không là tiến trình nào cả

Kernel HĐH nằm ngoài tất cả tiến trình, có vùng nhớ và ngăn xếp riêng (Hình 4.10a). Khi tiến trình đang hoạt động bị ngắt hoặc gọi lời gọi hệ thống, kernel lưu lại trạng thái tiến trình và nắm lại quyền sử dụng CPU.

HĐH sẽ thực hiện các chức năng cần thiết, rồi sau đó khôi phục hoạt động của tiến trình bị ngắt. Khái niệm tiến trình chỉ áp dụng với chương trình người dùng. Mã HĐH được thực thi riêng biệt khi CPU trong chế độ giám sát.

☞ HĐH thực thi trong tiến trình người dùng

Một phương pháp được áp dụng phổ biến trong các HĐH trên dòng máy tính nhỏ hoặc siêu nhỏ là thực thi tất cả phần mềm của HĐH trong ngữ cảnh tiến trình người dùng. HĐH bao gồm các thường trình, được người sử dụng gọi nhằm thực hiện công việc nào đó. Thường trình này vẫn chạy trong không gian bộ nhớ của tiến trình người dùng như minh họa trong Hình 4.10b.

Hình 4.10d minh họa cấu trúc của tiến trình trong phương pháp này. Bên cạnh vùng mã và dữ liệu thông thường, tiến trình có vùng nhớ chung là nơi đặt đoạn mã thuộc HĐH. Các đoạn mã này được sử dụng chung giữa nhiều tiến trình trong hệ thống.

Khi xuất hiện ngắt hay tiến trình gọi lời gọi hệ thống, CPU chuyển sang chế độ giám sát và quyền điều khiển được chuyển cho đoạn mã của HĐH. Khi đó, ngữ cảnh của tiến trình được lưu lại và chuyển sang ngữ cảnh của chương trình HĐH. Tuy nhiên, việc thực thi vẫn xảy ra trong tiến trình người sử dụng. Tức là có chuyển ngữ cảnh, nhưng không chuyển chương trình. Nếu sau khi hoàn thành công việc, HĐH thấy cần khôi phục tiến trình vừa bị phong tỏa thì HĐH chỉ cần chuyển ngữ cảnh sang tiến trình đó. Sau khi tiến trình bị tạm thời phong tỏa để đợi một dịch vụ của HĐH, tiến trình có thể nhanh chóng được khôi phục mà không cần 2 lần chuyển ngữ cảnh. Nếu HĐH muốn cho tiến trình khác chạy, thì HĐH chuyển quyền điều khiển cho bộ phận điều phối để lựa chọn cấp phát CPU cho một tiến trình khác. Hệ thống hoạt động ổn định và chính xác vì đoạn mã thực hiện những việc trên không phải đoạn mã người dùng mà là đoạn mã HĐH. Vì có bit chế độ, nên chương trình người dùng không thể can thiệp vào chương trình của HĐH ngay cả khi chương trình HĐH thực thi trong không gian bộ nhớ người sử dụng. Điểm chú ý ở đây là phân biệt tiến trình và chương trình. Tiến trình sẽ chạy hai loại chương trình, chương trình người dùng và

chương trình HDH. Chương trình HDH thực thi trong tất cả các tiến trình đều giống nhau.

HDH dựa trên tiến trình

Phương pháp thứ 3 được minh họa trên Hình 4.10c. HDH được triển khai thông qua một nhóm các tiến trình hệ thống, các tiến trình hệ thống này cài đặt các chức năng HDH cơ sở và thực thi trong chế độ giám sát. Ưu điểm của phương pháp này chính là tính module, khiến các thành phần của HDH có giao diện tương minh với nhau.

Vi nhân (Microkernel)

Phương pháp này được áp dụng trong Windows NT. Microkernel là một hạt nhân (rất nhỏ) của HDH, chỉ cung cấp các chức năng cơ sở nhất (lời gọi hệ thống). Các dịch vụ khác (do kernel cung cấp) được những chương trình (gọi là server) thực thi trong không gian người sử dụng. Ví dụ, Microkernel cung cấp dịch vụ quản lý không gian địa chỉ, quản lý luồng, truyền thông liên tiến trình, nhưng không cung cấp các dịch vụ như kết nối mạng hay hiển thị thông tin trên màn hình. Ưu điểm của giải pháp này là khi thêm một dịch vụ mới không cần phải thay đổi kernel. Tính an ninh của hệ thống được tăng cường vì phần lớn các thao tác thực thi ở chế độ người dùng chứ không phải trong chế độ giám sát.

4.6. NHẬN XÉT

Tiến trình là chương trình trong giai đoạn thực thi. Trong quá trình thực thi, tiến trình trải qua nhiều trạng thái. Tại mỗi thời điểm, trạng thái là hoạt động cụ thể của tiến trình như tạo mới, sẵn sàng, thực thi, phong tỏa, kết thúc. HDH quản lý tiến trình qua khối điều khiển tiến trình (PCB). Tiến trình khi không thực thi được đặt trong hàng đợi nào đấy. Có hai loại hàng đợi chính trong HDH là hàng đợi vào/ra và hàng đợi sẵn sàng. Hàng đợi sẵn sàng chứa những tiến trình sẵn sàng thực thi và đang chờ đến lượt sử dụng CPU. Bộ điều phối dài hạn lựa chọn nhóm tiến trình được quyền thực thi. Bộ điều phối ngắn hạn chọn một tiến trình nằm trong hàng đợi sẵn sàng để thực thi. Các tiến trình trong hệ thống có thể được thực hiện một cách đồng thời và do đó có thể chia sẻ thông tin, tăng tốc độ tính toán, cô lập các thành phần. Để thực hiện được đồng thời, hệ thống cần có cơ chế tạo mới và xóa

bộ tiến trình. Luồng – đơn vị cơ bản được quyền sử dụng CPU dùng chung phần mã, phần dữ liệu và một số tài nguyên hệ thống khác với những luồng ngang hàng, trong cùng một tiến trình (hay tác vụ).

CÂU HỎI ÔN TẬP

1. Định nghĩa tiến trình và trạng thái tiến trình.
2. Xác định các thành phần chính của khối điều khiển tiến trình.
3. Trình bày nhiệm vụ của bộ điều phối tiến trình.

Chương 5

ĐIỀU PHỐI TIẾN TRÌNH

*Điều phối tiến trình là lập kế hoạch cấp quyền sử dụng CPU. Mục đích sử dụng máy tính quyết định chính sách điều phối. Chú ý phân biệt **cơ chế** quyết định cách thức cấp phát CPU với **chính sách** xác định thứ tự "được sử dụng CPU" của tiến trình. Chương này trình bày cơ chế điều phối và hai lớp chính sách: độc quyền sử dụng (nonpreemptive) và không độc quyền sử dụng (preemptive). Thuật toán độc quyền cho phép tiến trình chiếm dụng CPU cho tới khi hoàn thành công việc. Trong thuật toán không độc quyền, tiến trình chỉ được sử dụng CPU trong một khoảng thời gian và khi hết thời gian, bộ điều phối chuyển quyền sử dụng CPU sang cho tiến trình khác cho dù tiến trình đang chạy chưa hoàn thành công việc. Chúng ta sẽ trình bày một số cơ chế được sử dụng trong các HĐH hiện đại cũng như phương thức cài đặt chúng.*

5.1. CƠ CHẾ ĐIỀU PHỐI

Trong HĐH đa chương trình, nhiều tiến trình được tải vào bộ nhớ trong và cùng nhau chia sẻ CPU. Do vậy, cần phương thức để HĐH cùng các tiến trình ứng dụng chia sẻ CPU. Ngoài ra, thời gian thực hiện vào/ra lớn hơn nhiều thời gian tiến trình sử dụng CPU, do đó trong hệ thống đa chương trình, tiến trình thực hiện vào/ra nên nhường quyền sử dụng CPU cho tiến trình khác. Cơ chế điều phối là bộ phận của bộ quản lý tiến trình, chịu trách nhiệm loại bỏ tiến trình đang chiếm giữ CPU và sau đó lựa chọn theo chính sách định trước một tiến trình nào đó trong trạng thái sẵn sàng để thực thi.

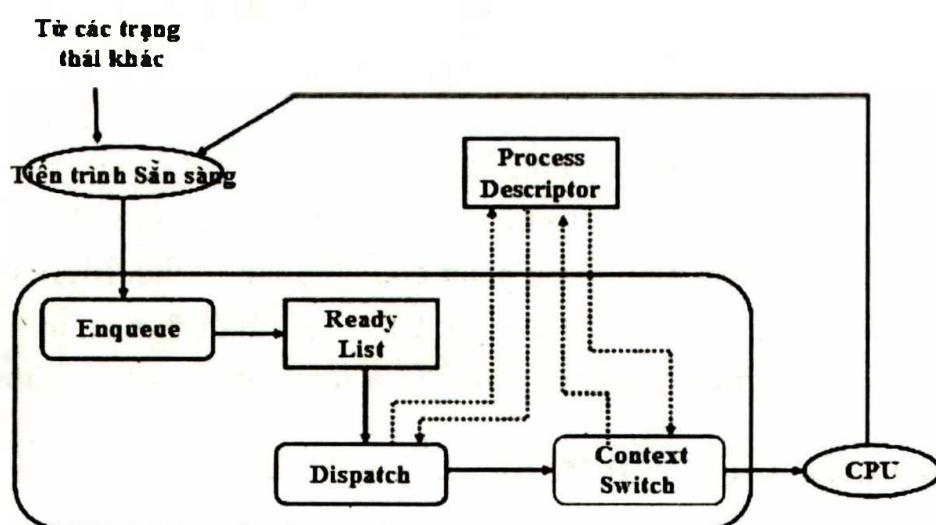
5.1.1. Bộ điều phối tiến trình

Trách nhiệm chính của bộ điều phối là luân phiên cho phép các tiến trình thực thi (tức là sử dụng CPU). Khi tiến trình đang thực thi mất quyền

sử dụng CPU (chuyển sang trạng thái sẵn sàng hoặc phong tỏa) thì một tiến trình khác ở trạng thái sẵn sàng sẽ được cấp phát CPU (tiến trình này chuyển sang trạng thái thực thi). Phương pháp điều phối xác định (1) thời điểm tiến trình không được tiếp tục sử dụng CPU và (2) tiến trình nào được cấp phát CPU. Cơ chế điều phối quyết định cách thức bộ quản lý tiến trình xác định thời điểm điều phối CPU và cách thức cấp phát/thu hồi CPU.

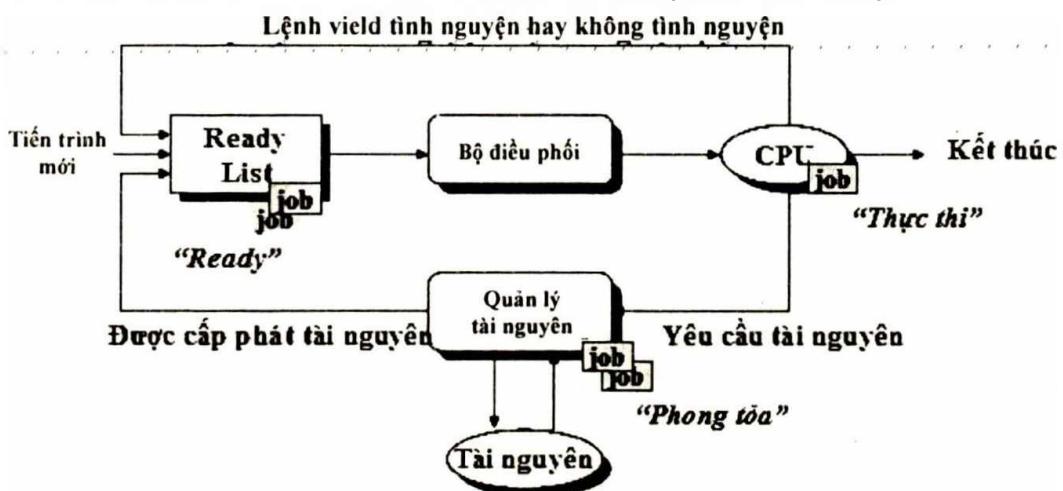
Hình 5.1 minh họa ba thành phần cơ bản của bộ điều phối là Enqueuer (bộ phận đưa tiến trình vào hàng đợi), Dispatcher (bộ điều vận) và Context Switcher (bộ chuyển ngữ cảnh). Khi tiến trình chuyển sang trạng thái sẵn sàng, khỏi PCB tương ứng được cập nhật đồng thời Enqueuer đặt con trỏ để khỏi PCB này nằm trong danh sách các tiến trình sẵn sàng sử dụng CPU. Danh sách này là cấu trúc dữ liệu của bộ quản lý tiến trình, thường được cài đặt dưới dạng hàng đợi các con trỏ, mỗi con trỏ "chỉ" tới PCB một tiến trình ở trạng thái sẵn sàng. Enqueuer tính chỉ số ưu tiên để cấp phát CPU cho tiến trình khi tiến trình được đưa vào trạng thái sẵn sàng, chỉ số ưu tiên cũng được xác định khi quyết định tiến trình nào sẽ bị loại khỏi danh sách sẵn sàng.

Khi chuyển quyền sử dụng CPU giữa hai tiến trình, bộ chuyển ngữ cảnh lưu lại giá trị của các thanh ghi trong CPU vào khỏi PCB của tiến trình cũ. Bộ phận nào gọi bộ phận chuyển ngữ cảnh sẽ xác định thời điểm chuyển quyền sử dụng CPU và ở đây có hai cơ chế là *tình nguyện* (tiến trình đang thực thi tự động giải phóng CPU bằng cách gọi bộ chuyển ngữ cảnh) và *không tình nguyện* (ngắt khiến chương trình đang thực thi phải từ bỏ quyền sử dụng CPU - trình xử lý ngắt thực hiện chuyển ngữ cảnh).



Hình 5.1. Cấu trúc bộ điều phối

Ngay sau đó, bộ điều vận được thực thi (ngữ cảnh của bộ điều vận được khôi phục) để lựa chọn và cấp phát CPU cho một tiến trình sẵn sàng nào đó (bằng cách chuyển ngữ cảnh sang tiến trình được lựa chọn). Hình 5.2 minh họa luồng đi của tiến trình trên hệ thống có một CPU. Tiến trình tự nguyện giải phóng CPU khi yêu cầu tài nguyên (chuyển quyền điều khiển cho bộ quản lý tài nguyên hoặc bộ điều phối). Tiến trình tự động giải phóng hoặc có thể bị cưỡng đoạt quyền sử dụng CPU. Tiến trình quay về trạng thái sẵn sàng khi tự nguyện (hoặc bị cưỡng ép) giải phóng CPU. Nếu như tiến trình giải phóng CPU do phải đợi tài nguyên, tiến trình sẽ không được quyền sử dụng CPU cho đến khi có đủ tài nguyên cần thiết. Sau đó, tiến trình mới chuyển sang trạng thái sẵn sàng và quay về hàng đợi sẵn sàng.



Hình 5.2. Luồng đi của các tiến trình trong hệ thống

5.1.2. Lưu lại ngữ cảnh của tiến trình

Rõ ràng mỗi lần chuyển quyền sử dụng CPU giữa hai tiến trình sẽ xuất hiện hai lần chuyển ngữ cảnh. Đầu tiên, ngữ cảnh của tiến trình đang chạy được lưu lại để khôi phục ngữ cảnh của bộ điều vận. Kế tiếp lưu lại ngữ cảnh bộ điều vận và tài ngữ cảnh của tiến trình mới. Công việc chuyển ngữ cảnh có thể ảnh hưởng lớn tới hiệu suất hệ thống, vì phải lưu lại giá trị nhiều thanh ghi bằng các chỉ thị LOAD và STORE. Do đó, nếu n là số lượng thanh ghi chung, m là số lượng thanh ghi trạng thái, b là số lượng chỉ thị cần thiết để lưu lại một thanh ghi và K là thời gian cần thiết để lưu lại một thanh ghi thì việc chuyển ngữ cảnh mất $(n + m) \times b \times K$ đơn vị thời gian.

Giá sử thời gian lưu trữ một đơn vị thông tin vào bộ nhớ trong là 50ns (10^{-9} s). Giá sử bus kết nối giữa CPU và bộ nhớ có 16 đường và độ lớn

mỗi thanh ghi là 32 bit. Như vậy, để lưu giá trị một thanh ghi cần $2 \times 50\text{ns}$. Giả sử hệ thống có 32 thanh ghi đa mục đích và 8 thanh ghi trạng thái, tổng thời gian để lưu các thanh ghi là $40 \times 2 \times 50\text{ns} = 4\text{ms}$. Cần thêm 4ms để tải các trạng thái của một tiến trình khác vào trong CPU (ở đây đã bỏ qua thời gian chạy của bộ điều vận). Như vậy, thời gian chuyển ngữ cảnh lớn hơn 8ms. Với bộ xử lý 200MHZ, một chỉ thị thực thi trong khoảng 10ns, thì trong 8ms, CPU có thể thực hiện 800 chỉ thị có ích.

Để giảm thời gian chuyển ngữ cảnh, một số kiến trúc máy tính có nhiều bộ thanh ghi, mỗi bộ thanh ghi ứng với một tiến trình. Thời gian chuyển ngữ cảnh giảm xuống vì chỉ cần lựa chọn bộ thanh ghi nào được đưa ra sử dụng (không cần lưu lại giá trị các thanh ghi).

5.1.3. Chia sẻ tự nguyện CPU

Khi nào bộ điều phối hoạt động? Cách đơn giản nhất là mỗi tiến trình định kỳ gọi bộ điều phối, tinh nguyện từ bỏ quyền sử dụng CPU. Một số phần cứng có chỉ thị **yield** cho phép tiến trình làm việc này. Chỉ thị này tương tự các lời gọi thủ tục, vì phải lưu lại địa chỉ chỉ thị thực thi kế tiếp và sau đó rẽ nhánh tới một địa chỉ khác. Điểm khác biệt ở đây là địa chỉ của chỉ thị tiếp theo không được lưu trong ngăn xếp của tiến trình mà lưu trong vùng nhớ được thiết kế sẵn. Cũng có sự tương đồng giữa hàm **yield** và trình xử lý ngắt. Chỉ thị **yield** có thể được cài đặt như sau:

```
yield(r, s)
{
    memory[r] = PC;
    PC = memory[s];
}
```

Khi tiến trình p_1 thực hiện **yield(r,s)**, thì tham số r đóng vai trò định danh của tiến trình p_1 (là địa chỉ khối PCB ứng với p_1). Có thể xác định r bằng cách xem tiến trình nào đang chiếm dụng CPU. Thường trong CPU có thanh ghi trạng thái tiến trình. Khi chuẩn bị thực thi, nội dung một số trường trong PCB ứng với tiến trình được tải vào thanh ghi trạng thái tiến trình. Giá trị r được xác định bằng cách đọc nội dung thanh ghi trạng thái này. Để đơn giản, rút gọn **yield(r,s)** thành **yield(*, s)**, trong đó * xác định tiến trình đang thực thi (là tiến trình gọi **yield**). Tham số s xác định tiến trình được thực thi kế tiếp – thực thi ngay sau khi chỉ thị **yield()** của tiến trình thứ nhất thực thi xong. Trong lúc thực thi p_1 , trước khi thực thi **yield**, **memory[r]** không xác

định. Sau khi **yield** thực thi, **memory[r]** chứa địa chỉ chỉ thị đứng ngay sau **yield** trong tiến trình p_1 (để sau này có thể khôi phục) và thanh ghi PC chứa địa chỉ của chỉ thị nào đó trong tiến trình p_2 sẽ được thực thi kế tiếp. Quyền sử dụng CPU được chuyển từ tiến trình gọi **yield** sang tiến trình mà địa chỉ của chỉ thị cuối cùng được thực thi lưu trong **memory[s]**. Giả sử **memory[s]** chứa địa chỉ của chỉ thị cuối cùng được thực thi trong p_2 , tiến trình p_1 có thể thực thi **yield(*, s)** để chuyển quyền điều khiển cho p_2 và tương tự sau này p_2 có thể dùng **yield(*, r)** để khởi động lại p_1 .

Nếu có nhiều tiến trình ở trạng thái sẵn sàng sử dụng CPU, p_2 có thể đóng vai trò bộ điều phối với định danh **scheduler**. Khi đó tiến trình có thể thực hiện **yield(*, scheduler)** khi không muốn sử dụng CPU. Sau đó bộ điều phối lựa chọn tiến trình s ở trạng thái sẵn sàng và gọi **yield(*, s)**. Dạng cộng tác đa chương trình này được phát triển trên dòng máy cá nhân của Xerox Alto. Nhiều nhân viên của Xerox đã chuyển việc sang Apple nên công nghệ cộng tác này đã được đưa vào các phiên bản đầu tiên của HĐH Macintosh.

Tuy nhiên, **yield** có nhược điểm là tiến trình có thể không tự nguyện "hợp tác" với các tiến trình khác. Nếu tiến trình không gọi **yield** thì không tiến trình nào sử dụng được CPU cho tới khi tiến trình đang sử dụng CPU thực hiện vào/ra. Điều này trở nên khó khăn nếu tiến trình đang thực thi lại nằm trong vòng lặp vô hạn và không thực hiện vào/ra. Khi đó tiến trình sẽ không bao giờ giải phóng CPU và kết quả các tiến trình khác phải đợi vô hạn. Do đó, hệ thống phải có khả năng định kỳ ngắt các tiến trình đang chạy, tức là một hình thức chia sẻ không tinh nguyen.

5.1.4. Chia sẻ CPU không tinh nguyen

Ngắt có thể định kỳ ngưng quá trình thực thi của tiến trình, nói cách khác là "ép" tiến trình thực thi chỉ thị **yield**. Điều này có thể được thực hiện nhờ bộ định thời bên trong hệ thống tạo ra ngắt theo chu kỳ. Thiết bị này được sử dụng như sau: người lập trình hệ thống thiết lập một khoảng chu kỳ thời gian cho thiết bị. Cứ mỗi lần hết một chu kỳ thời gian, thiết bị tạo ra một cảnh báo dưới dạng ngắt. Hình 5.3 minh họa cơ chế đơn giản của bộ định thời – thủ tục **IntervalTimer**. Mỗi lần đồng hồ thời gian thực (đồng hồ thạch anh trong máy tính) thực hiện xong T lần dao động của chu kỳ tinh thể thạch anh (T xung), thiết bị phần cứng định thời kết thúc một thao tác. Biến yêu cầu ngắt **InterruptRequest** nhận giá trị TRUE tương ứng với phần cứng

thiết lập cờ yêu cầu ngắt. Kết quả là cứ hết $K*T$ xung (đơn vị thời gian dao động của tinh thể thạch anh), cờ yêu cầu ngắt được thiết lập. Khoảng thời gian K (số xung) có thể được thiết lập nhờ hàm **SetInterval** minh họa trong Hình 5.3. Bộ định thời theo kiểu này được gọi là bộ định thời có thể lập trình được. Ngắt sinh ra sau mỗi K xung khiến bộ điều khiển đồng hồ gọi bộ điều khiển ngắt. Về mặt chức năng logic, chỉ thị này tương đương **yield**. Trình điều khiển thiết bị ngắt thời gian gọi bộ điều phối. Tiến trình đang thực thi không gọi **yield**, nhưng hệ thống vẫn bảo đảm bộ điều phối chạy sau mỗi K xung. Bộ điều phối sử dụng cơ chế chia sẻ CPU không tinh nguyễn còn được gọi là bộ điều phối không độc quyền (preemptive).

```

IntervalTimer() {
    InterruptCount--;
    if(InterruptCount <= 0) {
        InterruptRequest = TRUE;
        InterruptCount = K;
    }
}
SetInterval(programmableValue) {
    K = programmableValue;
    InterruptCount = K;
}

```

Hình 5.3. IntervalTimer

5.1.5. Hiệu suất

Bộ điều phối có ảnh hưởng lớn tới hiệu suất máy tính. Nếu bộ điều phối cấp phát CPU ngay cho tiến trình sẵn sàng thực thi, tiến trình sẽ tồn ít thời gian nằm chờ ở hàng đợi và được sử dụng CPU ngay khi cần thiết. Khi đó, hiệu suất chỉ phụ thuộc vào tốc độ phần cứng máy tính. Ngược lại, nếu tiến trình bị "bỏ quên" trong hàng đợi, khoảng thời gian tiến trình nằm tại hàng đợi lớn hơn rất nhiều so với khoảng thời gian sử dụng CPU. Hiệu suất cũng bị tác động bởi thời gian thực hiện chuyển ngữ cảnh. Chi phí phụ trội này lại bị ảnh hưởng bởi phần cứng. Một nhân tố khác ảnh hưởng đến hiệu suất là chính sách điều phối - xác định thời gian tiến trình đợi để được sử dụng CPU khi tiến trình đã ở trạng thái sẵn sàng. So với yếu tố thời gian chuyển ngữ cảnh, yếu tố về chính sách ảnh hưởng tới hiệu suất nhiều hơn. Ở đây chúng ta xét ảnh hưởng của điều phối tới hiệu suất tổng thể hệ thống. Thông thường, tại một thời điểm có một vài tiến trình sẵn sàng thực thi. Khi nào bộ

điều vận chọn tiến trình được thực thi kế tiếp? Tiêu chí nào được sử dụng trong quá trình lựa chọn? Tiến trình mãi mãi bị "bỏ quên" trong hàng đợi sẽ không bao giờ sử dụng được CPU để hoàn thành công việc của mình, hiện tượng này gọi là "chết đói". Nếu tiến trình được lựa chọn ngay khi sẵn sàng, khi đó thời gian thực thi thực sự của tiến trình sẽ tiệm cận với tốc độ phần cứng. Phương pháp và chính sách điều phối sẽ định ra tiêu chí lựa chọn tiến trình thực thi kế tiếp. Các cơ chế trình bày ở đây được sử dụng để cài đặt một phương pháp đã được người quản trị hoặc người thiết kế HĐH lựa chọn. Có những phương pháp tập trung vào hiệu suất hệ thống tổng thể, có phương pháp mong muốn chia sẻ công bằng CPU giữa các tiến trình, thậm chí có phương pháp cố gắng tối ưu hiệu suất của một lớp tiến trình cụ thể. Nói chung, hiệu suất sẽ quyết định lựa chọn phương pháp điều phối phù hợp.

5.2. CÁC PHƯƠNG PHÁP ĐIỀU PHỐI

Làm thế nào để bộ điều phối cấp phát CPU cho các tiến trình theo mục tiêu cụ thể? Có nên cấp phát dựa theo độ ưu tiên không? Có nên cấp phát công bằng không? Có nên gán độ ưu tiên cao cho các tiến trình có thời gian thực thi ngắn (hoặc dài) không? Những phương pháp cơ bản như vậy đã được nghiên cứu trong nhiều năm và tương tự bài toán điều phối trong ngành vận trù, chẳng hạn cách thức phục vụ khách hàng trong ngân hàng. Ví dụ trong hệ thống thời gian thực, các tiến trình phải được cấp phát CPU sớm để có thể kết thúc công việc trước mốc thời gian cụ thể. Trong hệ thống chia sẻ thời gian, mục tiêu điều phối là phân chia công bằng thời lượng sử dụng CPU giữa các tiến trình hay giữa các người dùng, hoặc giảm thiểu thời gian phản hồi cho người dùng. Phương pháp thích hợp phụ thuộc vào mục tiêu của HĐH.

Trong HĐH hiện đại, mức độ ưu tiên trong (hay đơn giản là mức độ ưu tiên) quyết định thứ tự sử dụng CPU của tiến trình. Qua đó có thể cài đặt bất cứ phương pháp điều phối nào. Ví dụ, với điều phối theo độ ưu tiên ngoài, người dùng được gán một mức ưu tiên. Độ ưu tiên (trong) của bất kỳ tiến trình nào do người dùng tạo ra bằng độ ưu tiên của người dùng. Độ ưu tiên có thể được gán động. Giả sử mục đích chia sẻ công bằng CPU (nếu n tiến trình hoạt động trong K đơn vị thời gian thì mỗi tiến trình được phép sử

dụng CPU trong K/n đơn vị thời gian). Có thể thực hiện điều này bằng cách tăng độ ưu tiên của tiến trình đang nằm trong hàng đợi sẵn sàng, nhưng giảm độ ưu tiên của tiến trình đang thực thi. Các phương thức điều chỉnh độ ưu tiên ứng với các phương pháp điều phối khác nhau.

Ngắt đơn giản hóa việc cài đặt phương pháp chia sẻ CPU không tinh nguyen. Nếu bộ quản lý tiến trình sử dụng bộ định thời (timer) điều khiển việc chuyển quyền sử dụng CPU, thì hệ thống xác định lượng tử thời gian là khoảng thời gian giữa hai ngắt liên tiếp. Trong trường hợp tiến trình kết thúc trước khi sử dụng hết lượng tử thời gian, tiến trình giải phóng CPU và bộ điều phối cấp phát CPU cho tiến trình khác. Dĩ nhiên, khi đó tiến trình mới phải được cấp phát nguyên một lượng tử thời gian, vì vậy bộ điều phối phải có khả năng thiết lập lại giá trị cho bộ định thời.

Với tập hợp cụ thể các tiến trình nằm trong hàng đợi sẵn sàng và thời lượng sử dụng CPU của mỗi tiến trình xác định, thì bộ điều phối không độc quyền hoàn toàn có thể xác định kế hoạch tối ưu cho một mục đích cụ thể (với điều kiện không đưa thêm tiến trình mới vào hàng đợi trong quá trình phục vụ các tiến trình khác). Thuật toán tối ưu xác định số lượng lượng tử thời gian sử dụng CPU của mỗi tiến trình, sau đó liệt kê tất cả các kế hoạch sử dụng CPU theo thứ tự nào đó. Phương pháp điều phối tối ưu căn cứ theo tiêu chí nào đó sẽ soát toàn bộ các kế hoạch nhằm chọn ra cái tốt nhất. Tuy nhiên, có một vài vấn đề trong phương pháp này:

- Khi có tiến trình mới đến hàng đợi trong khi hệ thống đang phục vụ các tiến trình hiện tại thì phải lập kế hoạch lại.
- Phải xác định được thời lượng sử dụng CPU thực tế của mỗi tiến trình trước khi tiến trình thực thi. Tuy nhiên, khó thực hiện được điều này.
- Các thuật toán điều phối tối ưu cho n tiến trình có độ phức tạp $O(n^2)$, nghĩa là lượng thời gian bộ điều phối sử dụng để lập kế hoạch tối ưu lớn hơn lượng thời gian thực sự dành cho việc phục vụ các tiến trình.

Giả sử: $P = \{p_i \mid 0 \leq i < n\}$ là tập các tiến trình. $S(p_i)$ là trạng thái của tiến trình p_i , trong đó $S(p_i) \in \{\text{đang thực thi}, \text{sẵn sàng}, \text{phong tỏa}\}$. Thời gian phục vụ $\tau(p_i)$ là tổng lượng thời gian sử dụng CPU cần thiết để tiến trình p_i hoàn thành nhiệm vụ. Thời gian chờ $W(p_i)$ là thời gian chờ của tiến trình, được tính từ khi tiến trình vào hàng đợi cho đến thời điểm đầu tiên

tiến trình sử dụng CPU. Thời gian lưu lại trong hệ thống $T_{TRnd}(p_i)$ là lượng thời gian từ khi tiến trình vào hàng đợi cho đến khi tiến trình hoàn tất công việc.

Mô hình tiến trình và các đại lượng thời gian trên được sử dụng khi so sánh hiệu suất các thuật toán điều phối. Trong hệ thống lô, thời gian lưu lại hệ thống là yếu tố quan trọng nhất, vì nó phản ánh lượng thời gian người dùng phải đợi để nhận được kết quả từ máy tính. Ở đây, thời gian lưu lại trong hệ thống trung bình là thời gian trung bình để hoàn thành một tiến trình (hay công việc) và nghịch đảo của đại lượng này là thông lượng hệ thống - số lượng công việc hoàn thành trong một đơn vị thời gian. Trong hệ thống lô, về mặt kỹ thuật, thời gian lưu lại hệ thống của công việc khác với thời gian lưu lại trong hệ thống của tiến trình do phải tính đến thời gian xếp hàng, cấp phát bộ nhớ và điều phối. Vì hệ thống lô quan tâm đến công việc nhiều hơn tiến trình, nên thời gian lưu lại hệ thống của công việc đóng vai trò quan trọng. Hệ thống chia sẻ thời gian thường tập trung vào thời gian thực thi riêng lẻ của từng tiến trình, chẳng hạn thời gian thi hành lệnh của người dùng. Thời gian này lại được chia thành hai giai đoạn là thời gian chờ đợi (do phải cạnh tranh quyền sử dụng CPU) và thời gian phục vụ. Với người dùng đang tương tác trực tiếp với máy tính, thì thời gian cần thiết để máy tính gửi lại một vài thông tin phản hồi nào đó (tức là thời gian chờ) có yếu tố rất quan trọng. Do vậy, trong hệ thống chia sẻ thời gian, đại lượng thời gian chờ được xem là tiêu chí đánh giá hiệu suất quan trọng nhất.



Hình 5.4. Mô hình điều phối đơn giản

Chú ý, mô hình tiến trình trên không tính đến hành vi của bộ quản lý tài nguyên (ngoại trừ khi xét đến trạng thái phong tỏa). Tiến trình có thể yêu cầu thời gian sử dụng CPU là khoảng liên tục, hoặc chia thành các khoảng thời gian sử dụng tại các thời điểm khác nhau, xen kẽ với các yêu cầu tài nguyên. Mô hình điều phối tiến trình được giản thể để bỏ qua tác động của việc tranh chấp tài nguyên (ngoại trừ tài nguyên CPU). Hình 5.4 minh họa điều này, tiến trình chỉ có thể ở một trong hai trạng thái là đang thực thi hoặc sẵn sàng. Các đại lượng đo hiệu suất trình bày ở trên, ngoại trừ thời

gian lưu lại trong hệ thống bỏ qua thời gian tiến trình bị phong tỏa do phải đợi tài nguyên. Mô hình này hoạt động như sau: Tiến trình lần đầu tiên bước vào trạng thái sẵn sàng sẽ yêu cầu thời gian sử dụng CPU xác định. Tiến trình sau khi được đáp ứng, có nghĩa là tổng thời gian thực thi trên CPU bằng thời gian phục vụ yêu cầu – sẽ kết thúc. Trên thực tế, có thể xem tiến trình được mô hình theo kiểu này khi tạo ra đã có đủ tài nguyên cần thiết (do đó sẵn sàng thực thi) và tiến trình sẽ kết thúc khi yêu cầu thêm tài nguyên mới. Các tiến trình bày trong các chương trước có thể xem như một chuỗi "nhỏ" các tiến trình không bị gián đoạn. Các thuật toán điều phối được chia làm hai lớp là lớp độc quyền (nonpreemptive) và lớp không độc quyền (preemptive). Trong thuật toán độc quyền, khi đã thực thi, tiến trình sẽ không giải phóng CPU cho đến khi hoàn thành. Các thuật toán không độc quyền thường dựa trên độ ưu tiên sau: tại bất kỳ thời điểm nào tiến trình đang sử dụng CPU luôn có độ ưu tiên cao nhất. Nếu tiến trình mới xuất hiện có độ ưu tiên cao hơn tiến trình hiện tại đang thực thi, thì tiến trình đang sử dụng CPU phải nhường quyền sử dụng CPU cho tiến trình mới.

5.2.1. Phân tiến trình thành nhiều tiến trình nhỏ

Trong quá trình hoạt động, các thao tác tính toán và vào/ra của tiến trình xen kẽ nhau. Giả sử tiến trình có k lần tính toán trên CPU và k lần thực hiện vào/ra. Như vậy, tổng thời gian phục vụ có thể là: $\tau(p_i) = \tau_1 + \tau_2 + \dots + \tau_k$. Giả sử tiến trình p_i được chia thành k tiến trình nhỏ $p_{i1}, p_{i2}, \dots, p_{ik}$, trong đó τ_{ij} là thời gian sử dụng CPU của p_{ij} . Mỗi p_{ij} sẽ được thi hành như một tiến trình độc lập và không gián đoạn. Bộ điều phối không độc quyền có thể chia τ_{ij} ra nhiều lượng tử thời gian trong quá trình điều phối cho p_{ij} . Nếu mỗi tiến trình yêu cầu k thao tác vào/ra khác nhau thì τ_i xen kẽ với các thời gian d_1, d_2, \dots , trong đó d_k là thời gian thực hiện thao tác vào/ra thứ k. Vì thế, tổng thời gian tiến trình sử dụng CPU và thời gian thực hiện vào/ra là: $\tau_1 + d_1 + \tau_2 + d_2 + \dots + \tau_k + d_k$. Với tiến trình hướng tính toán, τ_i tương đối lớn so với d_j . Với tiến trình hướng vào/ra, τ_i tương đối nhỏ so với d_j .

5.2.2. Đánh giá xấp xỉ tải hệ thống

Phụ thuộc vào các mục đích sử dụng hệ thống, các thuật toán điều phối sử dụng các tiêu chí khác nhau để lựa chọn tiến trình từ hàng đợi. Người ta có hai lựa chọn:

(1) Phân tích một thuật toán cụ thể trên một tài giả thiết và sau đó dự đoán hiệu suất của từng thuật toán.

(2) Sử dụng một tài thực và đánh giá hiệu suất từng thuật toán trên tài thực. Mục đích trong giáo trình này là đánh giá các phương pháp điều phối khác nhau chứ không tập trung nghiên cứu sâu vào việc dự đoán hiệu suất. Tuy vậy, chúng ta vẫn xét hiệu suất khi so sánh các thuật toán khác nhau. Mục đích việc nghiên cứu ở đây là giúp bạn đọc hình dung được các thuật toán thi hành như thế nào trên các tài khác nhau. Người ta dự đoán hiệu suất của từng thuật toán trên một tài ngẫu nhiên bất kỳ bằng cách tính trung bình cộng các đại lượng đo trên từng tiến trình riêng lẻ (vì xác suất tiến trình đi đến hàng đợi phân bố ngẫu nhiên).

Tài vào hệ thống có thể được mô tả bởi tốc độ tiến trình đến hàng đợi và thời gian phục vụ $\tau(p_i)$. Giả sử λ là số lượng trung bình các tiến trình đến hàng đợi (được tính là số lượng các tiến trình đến hàng đợi trong một đơn vị thời gian), $1/\mu$ sẽ là thời gian trung bình giữa hai lần đến liên tiếp. Giả sử μ biểu diễn tốc độ phục vụ trung bình ($1/\mu$ sẽ là thời gian phục vụ trung bình của tiến trình). Nếu bỏ qua thời gian chuyển ngữ cảnh và giả thiết CPU có đủ khả năng đáp ứng toàn bộ tải, thì khoảng thời gian CPU bận là

$$\rho = \lambda \times \frac{1}{\mu} = \frac{\lambda}{\mu}.$$

Nếu $\rho > 1$, CPU sẽ bị "quá tải" (lượng công việc đến lớn hơn khả năng phục vụ) và điều này không phụ thuộc vào việc sử dụng bất kỳ thuật toán điều phối nào. Đồng thời, nếu kích thước hàng đợi hữu hạn thì xuất hiện tình trạng tràn bộ đệm (overflow), vì tốc độ đến của các tiến trình lớn hơn tốc độ hệ thống có thể phục vụ. Hệ thống chỉ có thể tiến tới trạng thái ổn định khi $\lambda < \mu$ ($\rho < 1$).

Ví dụ, trong 1 phút có 10 tiến trình đến hệ thống (nghĩa là $\lambda = 10$ tiến trình/phút) và thời gian phục vụ trung bình cho mỗi tiến trình là 3 giây (nghĩa là: $1/\mu = 3$ giây = $1/20$ phút hoặc $\mu = 20$ tiến trình/phút). Tài hệ thống sẽ là: $\rho = \lambda/\mu = (10 \text{ tiến trình/phút}) / (20 \text{ tiến trình/phút}) = 0,5 = 50\%$.

5.3. THUẬT TOÁN ĐỘC QUYỀN

Thuật toán độc quyền cho phép tiến trình chiếm dụng CPU thực thi cho tới khi hoàn tất công việc. Trong hệ thống sử dụng thuật toán độc quyền,

không tiến trình nào chuyển từ trạng thái thực thi sang trạng thái sẵn sàng. Chỉ khi hoàn thành nhiệm vụ, tiến trình mới nhường quyền sử dụng CPU cho bộ điều phối. Thuật toán độc quyền "vay mượn" khá nhiều thuật toán kinh điển trong vận trù học (Nghiên cứu các phương pháp điều phối liên quan đến con người, chẳng hạn cách thức điều phối việc phục vụ khách hàng trong ngân hàng, sân bay hay siêu thị. Khi bắt đầu được phục vụ, khách hàng sẽ nhận được toàn bộ sự phục vụ cho đến khi xong việc mà không bị người khác chen ngang). Phương pháp này phù hợp với hệ thống không sử dụng ngắt thời gian để gọi bộ điều phối. Tại mỗi thời điểm có duy nhất một tiến trình sử dụng CPU và sau khi thực hiện xong sẽ nhường CPU cho tiến trình.

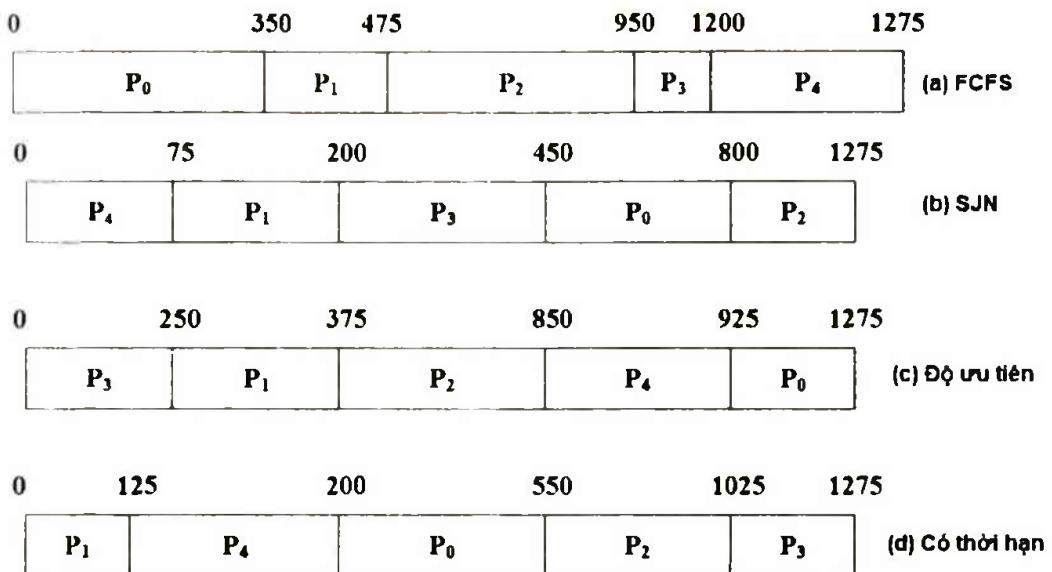
5.3.1. Đến trước phục vụ trước (First-Come-First-Served – FCFS)

Trong thuật toán FCFS, độ ưu tiên của tiến trình phụ thuộc vào thời điểm tiến trình được đưa vào hàng đợi. Bộ phận Enqueuer gắn cho tiến trình nhãn thời gian - là thời điểm tiến trình vào hàng đợi. Sau đó bộ điều vận lựa chọn tiến trình nào có nhãn thời gian bé nhất để thực thi. Hàng đợi sẵn sàng có thể được cài đặt bằng cấu trúc dữ liệu FIFO (First-In-First-Out) đơn giản (trong đó mỗi thành phần trong cấu trúc dữ liệu trả tới PCB của một tiến trình). Enqueuer đưa tiến trình mới vào đuôi hàng đợi, còn bộ điều vận chọn tiến trình ở đầu hàng đợi. Mặc dù dễ cài đặt, nhưng FCFS không quan tâm đến thời gian phục vụ và cũng bỏ qua các tiêu chí đánh giá chất lượng khác như thời gian đợi hay thời gian lưu lại hệ thống trung bình. Nói chung FCFS ít được áp dụng trên các hệ thống thực.

i	$\tau(p_i)$	Độ ưu tiên	Thời hạn chót
0	350	5	575
1	125	2	550
2	475	3	1050
3	250	1	Không xác định
4	75	4	200

Hình 5.5. Một tài giả thiết

Như minh họa trong Hình 5.5, giả sử có 5 tiến trình trong hàng đợi và thứ tự đến hàng đợi của chúng lần lượt là p_0, p_1, p_2, p_3, p_4 . Thuật toán FCFS điều phối các tiến trình như trong Hình 5.6a.



Hình 5.6. Các kế hoạch điều phối theo thuật toán khác nhau

Qua Hình 5.6a, chúng ta xác định thời gian lưu lại hệ thống của mỗi tiến trình trong thuật toán điều phối FCFS:

$$T_{TRnd}(p_0) = \tau(p_0) = 350$$

$$T_{TRnd}(p_1) = \tau(p_1) + T_{TRnd}(p_0) = 125 + 350 = 475$$

$$T_{TRnd}(p_2) = \tau(p_2) + T_{TRnd}(p_1) = 475 + 475 = 950$$

$$T_{TRnd}(p_3) = \tau(p_3) + T_{TRnd}(p_2) = 250 + 950 = 1200$$

$$T_{TRnd}(p_4) = \tau(p_4) + T_{TRnd}(p_3) = 75 + 1200 = 1275$$

Thời gian phản hồi trung bình là:

$$\bar{T}_{TRnd} = \frac{350 + 475 + 950 + 1200 + 1275}{5} = 850$$

Thời gian đợi của các tiến trình lần lượt là:

$$W(p_0) = 0$$

$$W(p_1) = T_{TRnd}(p_1) = 375$$

$$W(p_2) = T_{TRnd}(p_2) = 475$$

$$W(p_3) = T_{TRnd}(p_3) = 950$$

$$W(p_4) = T_{TRnd}(p_4) = 1200$$

Thời gian đợi trung bình là:

$$\bar{W} = \frac{0 + 350 + 475 + 950 + 1200}{5} = 595$$

5.3.2. Công việc ngắn nhất phục vụ trước (Shortest Job Next – SJN)

Thuật toán điều phối SJN lựa chọn tiến trình có thời gian phục vụ ngắn nhất. Thời gian lưu lại hệ thống của tiến trình p_i là tổng thời gian phục vụ của tất cả các tiến trình nằm trong hàng đợi có thời gian phục vụ ít hơn p_i . SJN giảm thiểu thời gian đợi trung bình vì những tiến trình có thời gian phục vụ ngắn sẽ được thực thi trước. Tuy nhiên, trong trường hợp có rất nhiều tiến trình cần phục vụ, cơ chế điều phối này có thể ngăn cản những tiến trình có thời gian phục vụ lớn được quyền thực thi. Hiện tượng tiến trình lớn có thể không được phục vụ (chết đói) là khiêm khuyết lớn trong SJN. Chúng ta vẫn giả sử danh sách hàng đợi chứa các tiến trình minh họa trong Hình 5.5. Ở đây thứ tự đến không quan trọng mà chỉ cần tất cả tiến trình nằm trong hàng đợi tại thời điểm điều phối và trong quá trình điều phối không xuất hiện thêm tiến trình mới. Thuật toán SJN tạo ra bảng kế hoạch minh họa trên Hình 5.6b.

Từ Hình 5.6b, chúng ta xác định:

$$T_{TRnd}(p_0) = \tau(p_0) + \tau(p_3) + \tau(p_1) + \tau(p_4) = 350 + 250 + 125 + 75 = 800$$

$$T_{TRnd}(p_0) = \tau(p_1) + \tau(p_4) = 125 + 75 = 200$$

$$\begin{aligned} T_{TRnd}(p_0) &= \tau(p_2) + \tau(p_0) + \tau(p_3) + \tau(p_1) + \tau(p_4) \\ &= 450 + 350 + 250 + 125 + 75 = 1275 \end{aligned}$$

$$T_{TRnd}(p_0) = \tau(p_3) + \tau(p_1) + \tau(p_4) = 250 + 125 + 75 = 450$$

$$T_{TRnd}(p_0) = \tau(p_4) = 75$$

Thời gian lưu lại hệ thống trung bình là:

$$\bar{T}_{TRnd} = \frac{800 + 200 + 1275 + 450 + 75}{5} = 560$$

Thời gian đợi của các tiến trình như sau:

$$W(p_0) = 450$$

$$W(p_1) = 75$$

$$W(p_2) = 800$$

$$W(p_3) = 200$$

$$W(p_4) = 0$$

Thời gian đợi trung bình là:

$$\overline{W} = \frac{450 + 75 + 800 + 200 + 0}{5} = 350$$

SJF được áp dụng trên hệ thống theo lô để giảm thiểu thời gian lưu lại hệ thống trung bình và người ta cũng mong muốn đạt được tính chất này trên hệ thống tương tác. Trong hệ thống tương tác, người dùng đánh lệnh, hệ thống thực hiện lệnh, gửi kết quả; người dùng đánh tiếp lệnh, hệ thống tiếp tục xử lý,... Nếu coi lệnh là "công việc", chúng ta có thể giảm thiểu thời gian phản hồi trung bình bằng cách lựa chọn tiến trình cần ít CPU nhất để thực thi. Tuy vậy, khó xác định chính xác tiến trình nào có tính chất này mà chỉ có thể ước lược dựa trên các hành vi trong quá khứ. Giả sử thời gian thực thi của một lệnh trong lần đầu là T_0 và thời gian thực thi kế tiếp của lệnh này là T_1 . Chúng ta có thể ước lược thời gian thực thi của lệnh bằng cách sử dụng thêm trọng số α : $\alpha T_0 + (1 - \alpha)T_1$. Giá trị trọng số α quyết định giá trị ước lược mới có phụ thuộc nhiều vào giá trị cũ hay không. Với $\alpha = 1/2$, ta lần lượt có giá trị các ước lược: T_0 , $T_0/2 + T_1/2$, $T_0/4 + T_1/4 + T_2/2$, $T_0/8 + T_1/8 + T_2/4 + T_3/2$,... Sau ba lần thực thi, hệ số của T_0 trong giá trị ước lược chỉ còn $1/8$.

5.3.3. Điều phối theo độ ưu tiên

Trong điều phối theo độ ưu tiên, tiến trình được cấp phát CPU căn cứ theo độ ưu tiên được gán từ bên ngoài (ở đây chúng ta coi số bé có độ ưu tiên cao). Độ ưu tiên trong được xác định căn cứ vào thao tác của tiến trình trong môi trường tính toán, chẳng hạn như độ ưu tiên được xác định theo thời gian phục vụ áp dụng trong SJN. Độ ưu tiên ngoài phản ánh tầm quan trọng của công việc, thường được người sử dụng xác định từ bên ngoài. Độ ưu tiên ngoài của tiến trình có thể bị người sử dụng thay đổi ("người quan trọng có quyền ưu tiên cao hơn"), bản chất công việc ("tiến trình phải tắt lò phản ứng hạt nhân khi nhiệt độ vượt ngưỡng nào đó"), hay bất kỳ một tiêu chuẩn ưu tiên nào khác.

Yếu tố quan trọng nhất đối với hiệu suất trong thuật toán điều phối theo độ ưu tiên là việc gán độ ưu tiên cho tiến trình. Trong cơ chế điều phối này, tiến trình có độ ưu tiên thấp vẫn có thể bị "chết đói". Tuy nhiên, có thể giải quyết vấn đề bằng cách tính độ ưu tiên từ bên trong. Hệ thống có thể dùng

khoảng thời gian đợi của tiến trình làm tham số để xác định độ ưu tiên. Thời gian đợi càng dài thì độ ưu tiên càng cao. Khi đó hiện tượng "chết đói" sẽ không xuất hiện. Chúng ta vẫn sử dụng ví dụ trong Hình 5.5. Cơ chế điều phối theo độ ưu tiên tạo ra bảng kế hoạch như Hình 5.6c.

Chúng ta tính:

$$\begin{aligned} T_{TRnd}(p_0) &= \tau(p_0) + \tau(p_4) + \tau(p_2) + \tau(p_1) + \tau(p_3) \\ &= 350 + 75 + 475 + 125 + 250 = 1275 \end{aligned}$$

$$T_{TRnd}(p_1) = \tau(p_1) + \tau(p_3) = 125 + 250 = 375$$

$$T_{TRnd}(p_2) = \tau(p_2) + \tau(p_1) + \tau(p_3) = 475 + 125 + 250 = 850$$

$$T_{TRnd}(p_3) = \tau(p_3) = 250$$

$$T_{TRnd}(p_4) = \tau(p_4) + \tau(p_2) + \tau(p_1) + \tau(p_3) = 75 + 475 + 125 + 250 = 925$$

Thời gian lưu lại hệ thống trung bình là:

$$\bar{T}_{TRnd} = \frac{1275 + 375 + 850 + 250 + 925}{5} = 735$$

Thời gian đợi của các tiến trình là:

$$W(p_0) = 925$$

$$W(p_1) = 250$$

$$W(p_2) = 375$$

$$W(p_3) = 0$$

$$W(p_4) = 850$$

Thời gian đợi trung bình là:

$$\bar{W} = \frac{925 + 25 + 375 + 0 + 850}{5} = 480$$

5.3.4. Điều phối có thời hạn (Deadline Scheduling)

Trong hệ thống thời gian thực, có những tiến trình phải được kết thúc trước một thời hạn về thời gian nhất định. Tiêu chí quan trọng để đánh giá hiệu suất là xem hệ thống có thể thực hiện xong các tiến trình trước thời hạn cuối cùng hay không. Các tiêu chí về thời gian đợi và thời gian lưu lại hệ thống không còn quan trọng. Bộ điều phối phải xác định được thời gian thực thi của tất cả các tiến trình. Tiến trình chỉ được chấp nhận vào hàng đợi nếu

bộ điều phối đảm bảo tiến trình có thể kết thúc trước thời hạn chót. Trong hệ thống truyền thông đa phương tiện, thời hạn thực hiện có thể được sử dụng để ngăn ngừa hiện tượng jitter (tốc độ đến đích của các gói tin không đều). Cột thứ 4 trong Hình 5.5 minh họa ví dụ về thời hạn cuối cho mỗi tiến trình. Hình 5.6d minh họa một kế hoạch đáp ứng được tiêu chí thời hạn cho toàn bộ tiến trình.

5.4. THUẬT TOÁN KHÔNG ĐỘC QUYỀN

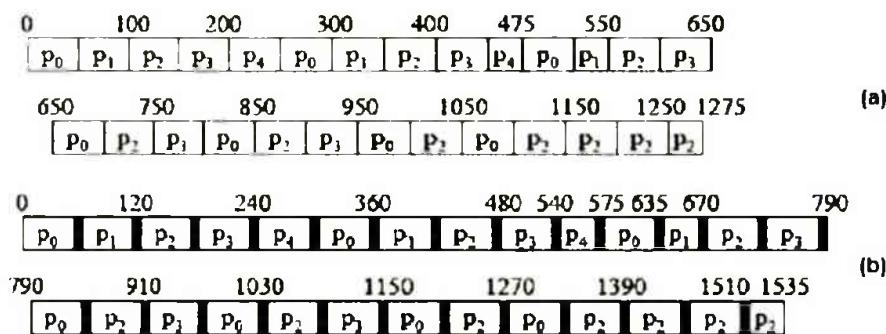
Trong thuật toán không độc quyền, tiến trình có độ ưu tiên cao nhất luôn được cấp phát CPU. Tiến trình có độ ưu tiên thấp phải nhường CPU cho tiến trình có độ ưu tiên cao. Bộ điều phối được gọi mỗi khi có tiến trình mới sẵn sàng thực thi hoặc sau khi hết mỗi lượng từ thời gian (nếu hệ thống sử dụng bộ định thời). Các thuật toán điều phối không độc quyền có thể đáp ứng nhanh chóng tiến trình có độ ưu tiên cao và bảo đảm các tiến trình chia sẻ công bằng CPU. Có thể sử dụng thuật toán SJN và thuật toán Độ ưu tiên theo thuật toán không độc quyền.

Với SJN, tiến trình có thời gian phục vụ nhỏ nhất được cấp phát CPU trước. Giả sử tiến trình p_j đến khi p_i đang thực thi, hệ thống so sánh $\tau(p_i)$ và $\tau(p_j)$. Trước khi p_j đến, p_i là tiến trình có thời gian phục vụ nhỏ nhất. Xét hệ thống có tài minh họa trong Hình 5.5 và sử dụng cơ chế điều phối SJN không độc quyền. Khi p_1 đang thực thi (sau khi p_4 thực thi xong), nếu xuất hiện tiến trình có thời gian phục vụ 35 thì hệ thống kiểm tra xem p_1 có cần nhiều hơn 35 đơn vị thời gian để thực hiện không. Nếu có, p_1 phải nhường CPU cho tiến trình mới. Tương tự, giả sử hệ thống không độc quyền sử dụng thuật toán điều phối theo độ ưu tiên với tài hệ thống minh họa trên Hình 5.5. Giả sử p_2 đang thực thi (sau p_3 và p_1) và xuất hiện tiến trình có độ ưu tiên 2 thì p_2 phải quay lại hàng đợi để nhường CPU cho tiến trình mới. Khi trình bày các thuật toán độc quyền, chúng ta bỏ qua chi phí chuyển ngữ cảnh giữa các tiến trình, vì giả định tiến trình hoàn thành công việc mà không bị gián đoạn. Trong thuật toán không độc quyền thuận túy, mỗi khi xuất hiện ngắt, tiến trình đang thực thi có thể phải nhường CPU cho tiến trình mới có độ ưu tiên cao hơn. Do đó, trong các hệ thống điều phối không độc quyền, chi phí chuyển ngữ cảnh là một yếu tố quan trọng.

5.4.1. Điều phối xoay vòng (Round Robin – RR)

Xoay vòng (RR) là thuật toán điều phối được sử dụng rộng rãi nhất, vì có thể phân chia công bằng thời gian sử dụng CPU giữa các tiến trình. Thuật toán này rất phù hợp trong hệ thống đa chương trình tương tác: Nếu trong hệ thống có n tiến trình thì mỗi tiến trình nhận được xấp xỉ $1/n$ đơn vị xử lý thời gian thực (xấp xỉ vì phải tính đến chi phí điều phối và chuyển ngữ cảnh). Nếu tiến trình kết thúc trước khi hết lượng tử thời gian, hệ thống phải khởi động bộ điều phối để cấp phát mới nguyên một lượng tử thời gian sử dụng CPU cho tiến trình khác. Tiến trình mới sẽ được đặt trong hàng đợi. Tuy nhiên, vị trí nào trong hàng đợi phụ thuộc vào phương pháp cài đặt. Nếu hàng đợi là danh sách liên kết vòng, tiến trình mới được đặt ngay sau tiến trình vừa được thực thi, như thế $n - 1$ tiến trình kia sẽ được phục vụ trước tiến trình mới. Nếu hàng đợi được cài đặt dưới dạng danh sách và bộ điều vận lựa chọn tiến trình theo thứ tự thì tiến trình mới được đặt ở cuối hàng đợi và điều này không phụ thuộc vào tiến trình nào đang thực thi khi tiến trình mới xuất hiện. Trung bình tiến trình mới phải đợi $n/2$ lượng tử thời gian trước khi được cấp phát CPU.

Xét ảnh hưởng của thời gian chuyển cảnh trong điều phối xoay vòng: Giả sử C là thời gian chuyển ngữ cảnh giữa các tiến trình người dùng (đôi khi coi C đủ nhỏ để có thể bỏ qua). Trong mỗi $n(q + C)$ đơn vị thời gian thực, mỗi tiến trình trong n tiến trình nhận được q đơn vị thời gian sử dụng CPU. Có thể dễ dàng cài đặt cơ chế điều phối xoay vòng trên hệ thống sử dụng ngắt thời gian, vì có thể thiết lập để cứ sau một lượng tử thời gian, bộ định thời sẽ tạo ra ngắt. Sau đó, trình xử lý ngắt thời gian sẽ gọi bộ điều phối. Tiến trình đang ở trạng thái phong tỏa có thể chuyển sang trạng thái sẵn sàng khi có đủ tài nguyên cần thiết (trừ tài nguyên CPU). Ngắt do bộ định thời tạo ra giúp hệ thống xác định khi nào tiến trình đang thực thi sử dụng hết lượng tử thời gian. Khi đó bộ điều phối sẽ thực hiện các việc sau: điều chỉnh hàng đợi, thiết lập lại bộ định thời, cho phép tiến trình ở đầu hàng đợi thực thi. Giả sử danh sách các tiến trình ở trạng thái sẵn sàng như trong Hình 5.5; lượng tử thời gian cho mỗi lần cấp phát là 50 và bỏ qua thời gian chuyển ngữ cảnh. Kế hoạch được minh họa trên Hình 5.7a.



Hình 5.7. Kế hoạch điều phối theo thuật toán xoay vòng

Từ biểu đồ này, chúng ta xác định được thời gian lưu lại hệ thống của tiến trình:

$$T_{TRnd}(p_0) = 1100$$

$$T_{TRnd}(p_1) = 550$$

$$T_{TRnd}(p_2) = 1275$$

$$T_{TRnd}(p_3) = 950$$

$$T_{TRnd}(p_4) = 475$$

Thời gian lưu lại hệ thống trung bình là:

$$\bar{T}_{TRnd} = \frac{1100 + 550 + 1275 + 950 + 475}{5} = \frac{4350}{5} = 870.$$

Thời gian đợi của các tiến trình:

$$W(p_0) = 0$$

$$W(p_1) = 50$$

$$W(p_2) = 100$$

$$W(p_3) = 150$$

$$W(p_4) = 200$$

Thời gian đợi trung bình là:

$$\bar{W} = \frac{0 + 50 + 100 + 150 + 200}{5} = \frac{500}{5} = 100.$$

Thời gian đợi chứng tỏ ưu điểm của điều phối xoay vòng (cũng như các thuật toán dựa trên lượng tử thời gian) khi xét theo tiêu chí sau bao lâu tiến trình bắt đầu được phục vụ. Tuy nhiên, thời gian lưu lại hệ thống trung bình không khác mấy so với các thuật toán độc quyền.

Xét ví dụ trên nhưng tính thêm thời gian chuyển ngữ cảnh là 10 (Hình 5.7b). Thời gian lưu lại hệ thống của các tiến trình là:

$$T_{TRnd}(p_0) = 1320$$

$$T_{TRnd}(p_1) = 660$$

$$T_{TRnd}(p_2) = 1535$$

$$T_{TRnd}(p_3) = 1140$$

$$T_{TRnd}(p_4) = 565$$

Thời gian lưu lại hệ thống trung bình là:

$$\bar{T}_{TRnd} = \frac{1320 + 660 + 1535 + 1140 + 565}{5} = \frac{5220}{5} = 1044.$$

Thời gian đợi của từng tiến trình là:

$$W(p_0) = 0$$

$$W(p_1) = 60$$

$$W(p_2) = 120$$

$$W(p_3) = 180$$

$$W(p_4) = 240$$

Thời gian đợi trung bình là:

$$\bar{W} = \frac{0 + 60 + 120 + 180 + 240}{5} = \frac{600}{5} = 120$$

5.4.2. Hàng đợi nhiều mức

Hàng đợi nhiều mức là dạng tổng quát của thuật toán điều phối theo độ ưu tiên: tất cả các tiến trình có cùng độ ưu tiên nằm trên cùng một hàng đợi. Có hai cấp độ điều phối CPU là giữa các hàng đợi (có độ ưu tiên khác nhau) và giữa các tiến trình nằm trong cùng hàng đợi (có cùng độ ưu tiên). Nếu sử dụng điều phối hàng đợi theo thuật toán không độc quyền thì phải phục vụ toàn bộ các tiến trình trong hàng đợi 1, rồi mới đến các tiến trình nào trong hàng đợi 2,... Trong mỗi hàng đợi k, CPU có thể được cấp phát theo bất kỳ thuật toán điều phối nào.

Có thể cải tiến thuật toán này để phân bổ quyền sử dụng CPU giữa các hàng đợi chứ không phải luôn luôn ưu tiên hàng đợi có độ ưu tiên cao. Ví dụ, hàng đợi j có thể nhận được $1/2^j$ thời gian. Trong 100s thời gian thực

(bỏ qua thời gian chuyển ngữ cảnh) các tiến trình trong hàng đợi 1 nhận được 50s; hàng đợi 2 nhận được 25s; hàng đợi 3 nhận được 12,5s... Thuật toán điều phối càng phức tạp thì thời gian chuyển ngữ cảnh càng lớn, vì thế hầu hết các thuật toán điều phối trong hệ thống chia sẻ thời gian thường là hàng đợi nhiều mức đơn giản và trong mỗi hàng đợi áp dụng thuật toán xoay vòng.

Tiến trình tiền cảnh và tiến trình hậu cảnh

Các hệ thống chia sẻ thời gian thường hỗ trợ các tiến trình tiền cảnh (foreground) và tiến trình hậu cảnh (background). Các tiến trình tiền cảnh thực hiện tương tác với người dùng có độ ưu tiên cao, trong khi tiến trình hậu cảnh có độ ưu tiên thấp chỉ được thực thi khi không thực thi tiến trình tiền cảnh nào. Tiến trình tiền cảnh luôn luôn có độ ưu tiên cao hơn tiến trình hậu cảnh.

Có nhiều thuật toán đáp ứng kiểu phân chia tiến trình tiền cảnh/hậu cảnh. Chẳng hạn, tiến trình xử lý ngắn có thể chạy với mức ưu tiên 1, tiến trình điều khiển thiết bị ở mức 2, tiến trình tương tác với người dùng ở mức 3, trình soạn thảo ở mức 4, các công việc lô thông thường ở mức 5, còn các công việc lô cần nhiều thời gian thực thi ở mức 6. Dĩ nhiên, lựa chọn như trên chỉ mang tính tương đối, vì độ ưu tiên của tiến trình có thể thay đổi trong quá trình thực thi, phụ thuộc vào đang tính toán trong giai đoạn nào. Ví dụ, nếu tiến trình soạn thảo văn bản có tương tác với người dùng sử dụng quá nhiều CPU thì độ ưu tiên của nó có thể bị giảm (do sử dụng CPU vượt mức cho phép). Ngoài ra, có thể cho phép tiến trình tăng độ ưu tiên trong giai đoạn cần thực hiện nhiều tính toán với lý do người dùng cần chiếm dụng CPU trong thời gian tương đối lâu mới có một dịch vụ với chất lượng chấp nhận được. Hệ thống cho phép các tiến trình thay đổi độ ưu tiên trong các hàng đợi con, được gọi là một hàng đợi nhiều mức có phản hồi.

Điều phối trong BSD UNIX

BSD UNIX sử dụng cơ chế điều phối hàng đợi 32 mức: tiến trình hệ thống nằm trong hàng đợi 0 đến 7, tiến trình thực thi trong không gian người dùng nằm trong hàng đợi 8 đến 31. Bộ điều vận lựa chọn tiến trình từ hàng đợi có độ ưu tiên cao trước. Trong một hàng đợi, BSD UNIX sử dụng cơ chế điều phối xoay vòng. Lượng tử thời gian phụ thuộc vào phiên bản cài đặt, nhưng nói chung bé hơn 100μs.

Ngoài ra, tiến trình còn có độ ưu tiên ngoài, gọi là *nice* – được sử dụng để xác định (nhưng còn phụ thuộc vào các yếu tố khác) tiến trình sẽ nằm trong hàng đợi nào khi ở trạng thái sẵn sàng. Giá trị *nice* biến thiên từ -20 đến 20, trong đó -20 là mức ưu tiên cao nhất. Sau mỗi một lượng tử thời gian, bộ điều phối tính lại độ ưu tiên của mỗi tiến trình, giá trị này là hàm số của *nice* và lượng thời gian tiến trình yêu cầu sử dụng CPU (yêu cầu càng nhiều thì độ ưu tiên càng giảm).

Hàm **sleep** tương tự **yield** (với việc lưu lại ngữ cảnh), khi tiến trình gọi **sleep**, bộ điều phối được gọi.

☞ Điều phối luồng trong Window NT

Bộ điều phối luồng trong Window NT có nhiều mức có phản hồi, với mục tiêu phục vụ thật nhanh các luồng cần phải đáp ứng ngay lập tức. Bộ điều phối hỗ trợ 32 cấp độ điều phối khác nhau: 16 hàng đợi có mức ưu tiên cao nhất gọi là hàng đợi mức thời gian thực (real-time queue), tiếp theo là 15 hàng đợi, các hàng đợi có mức ưu tiên thay đổi được (variable level queues) và hàng đợi có mức ưu tiên thấp nhất là hàng đợi mức hệ thống (system level). Bộ điều phối cố gắng giới hạn số lượng các luồng được đưa vào các hàng đợi thời gian thực, nhằm làm giảm thiểu sự tranh chấp trong các luồng có mức độ ưu tiên cao. Tuy nhiên, Window NT không phải hệ thống thời gian thực, nên không đảm bảo luồng ở mức ưu tiên cao có thể nhận được quyền sử dụng CPU trước một thời hạn chót nào đó. Hàng đợi mức hệ thống là một "luồng rỗng" ứng với hệ thống rơi vào trạng thái nghỉ (idle system). Nghĩa là, khi trong hệ thống không có luồng nào khả thi, hệ thống sẽ thực thi luồng rỗng (là luồng không có trang trong bộ nhớ) cho tới khi xảy ra ngắt và xuất hiện luồng khả thi. Bộ điều phối hoàn toàn không độc quyền, có nghĩa là khi chuyển sang trạng thái sẵn sàng, luồng được đặt trong trong một hàng đợi nào đầy phụ thuộc vào độ ưu tiên.

5.4.3. Điều phối có đảm bảo (Guaranteed Scheduling)

Hệ thống có thể "hứa" đảm bảo một chất lượng nào đó cho người sử dụng và sau đó cố gắng thực hiện lời hứa. Một đảm bảo dễ "hứa" và dễ thực hiện là "Nếu có n người sử dụng hệ thống (hay tiến trình) thì mỗi người (hay tiến trình) sẽ nhận được $1/n$ năng lực xử lý CPU". Để thực hiện lời hứa, hệ thống phải kiểm soát được thời gian sử dụng CPU của tiến trình. Kế tiếp xác

dịnh lượng thời gian mà hệ thống dành cho tiến trình (là khoảng thời gian tĩnh từ khi tiến trình được tạo ra) chia cho n. Tỷ lệ hai đại lượng này là tỷ lệ thời gian tiến trình thực sự sử dụng CPU. Ví dụ, tỷ lệ 0,5 nghĩa là tiến trình mới sử dụng 1/2 thời gian được cấp phát của mình. Tiến trình có tỷ lệ thấp nhất sẽ được lựa chọn trước để chạy.

5.4.4. Điều phối quay xổ số (Lottery)

Hứa với người sử dụng và cố gắng thực hiện lời hứa là một ý tưởng hay, nhưng lại khó cài đặt. Thuật toán xổ số cũng đem lại kết quả tương tự nhưng dễ cài đặt hơn. Mỗi tiến trình được phát "vé xổ số" cho tài nguyên cần thiết (thời gian sử dụng CPU cũng là tài nguyên). Khi nào cần đưa ra quyết định cấp phát, bộ điều phối "quay xổ số". Tiến trình nào có "vé số" trúng thưởng được quyền sử dụng tài nguyên. Trong trường hợp điều phối CPU, hệ thống quay số 50 lần trong 1s và tiến trình "trúng số" sử dụng CPU trong 20ms. Hệ thống có thể cho những tiến trình quan trọng nhiều vé số (làm tăng cơ hội "trúng số"). Nếu hệ thống có 100 "vé số" và cấp phát cho một tiến trình 20 "vé số", tiến trình này được sử dụng 20% CPU (tính trong thời gian dài). Các tiến trình khi hợp tác có thể trao đổi "vé số" với nhau. Ví dụ, khi gửi thông điệp yêu cầu tới tiến trình Server, tiến trình Client gửi kèm toàn bộ "vé số" của mình và sau đó tự phong tỏa. Với lượng "vé" từ Client, cơ hội thực thi của Server tăng lên. Khi thực thi xong, Server gửi trả lại toàn bộ "vé số" trong thông điệp trả lời để Client nhanh chóng được thực thi. Điều phối theo kiểu quay xổ số được sử dụng trong nhiều hệ thống không thích hợp với các kiểu điều phối khác. Chẳng hạn, trong hệ thống cài đặt video server, một vài tiến trình gửi luồng âm thanh hình ảnh tới các khách hàng, nhưng theo các tỷ lệ nén khác nhau (10, 20, 25 frame/s). Bằng cách phân phối cho mỗi tiến trình trên lần lượt 10, 20, 25 "vé số", hệ thống có vẻ phân phối thời gian sử dụng CPU theo tỷ lệ 10 : 20 : 25.

5.4.5. Điều phối công bằng

Từ trước đến giờ, chúng ta trình bày điều phối tiến trình nhưng không quan tâm đến người sử dụng (NSD) nào tạo ra tiến trình. Do đó, nếu NSD 1 tạo ra 9 tiến trình, NSD 2 tạo ra 1 tiến trình, bộ điều phối sử dụng cơ chế xoay vòng và các tiến trình có độ ưu tiên bằng nhau thì tiến trình 1 sẽ chiếm 90% CPU, trong khi NSD 2 chỉ được 10%. Để ngăn ngừa tình huống này, nhiều hệ thống đã xét tới NSD tạo ra tiến trình trước khi thực hiện điều phối.

Trong cơ chế này, mỗi NSD được cấp phát một thời lượng sử dụng CPU và quá trình điều phối phải bảo đảm điều này. Ví dụ, nếu hệ thống có 2 NSD, mỗi NSD được quyền chiếm 50% thời lượng CPU. NSD 1 có 4 tiến trình A, B, C và D; NSD 2 có 1 tiến trình E. Nếu điều phối theo kiểu xoay vòng, kết quả điều phối có thể là: **A, E, B, E, C, E, D, A, E, B, E, C, E, D, A, E, B, E, C, E, D, A, E, B, E, C, E, D**. Nếu NSD 2 được cấp phát CPU gấp đôi NSD 1 thì kết quả điều phối có thể là: **A, B, E, C, D, A, B, E, C, D**.

5.5. NHẬN XÉT

Bộ điều phối chịu trách nhiệm cấp phát CPU cho nhiều tiến trình sẵn sàng thực thi. Bộ điều phối được bộ định thời gọi định kỳ hoặc bất cứ khi tiến trình đang thực thi tự động giải phóng CPU qua chỉ thị **yield** hoặc yêu cầu tài nguyên. Từ danh sách các tiến trình ở trạng thái sẵn sàng, bộ điều phối chọn ra một tiến trình để cấp phát CPU. Thuật toán điều phối có thể được chia thành hai nhóm: độc quyền hoặc không độc quyền. Thuật toán độc quyền cho phép tiến trình chạy tới khi hoàn thành công việc, trong khi thuật toán không độc quyền sử dụng đồng hồ định khoảng thời gian và bộ điều phối sẽ định kỳ cấp phát CPU cho các tiến trình. Thuật toán FCFS và SJN có độ ưu tiên và thời hạn thuộc lớp thuật toán độc quyền, trong khi các thuật toán xoay vòng và hàng đợi nhiều mức thuộc nhóm không độc quyền. Thuật toán điều phối có thể được cài đặt theo nhiều cách khác nhau. Các bộ điều phối phức tạp trong BSD UNIX và Windows NT sử dụng cơ chế hàng đợi nhiều mức, nhưng có cải tiến. Điều phối được coi là "trái tim" của bộ quản lý tài nguyên CPU với trách nhiệm điều khiển việc chia sẻ CPU giữa nhiều tiến trình. Một khi môi trường máy tính cho phép nhiều tiến trình thực hiện đồng thời thông qua điều phối, thì bộ quản lý tiến trình phải có cơ chế cho phép các tiến trình phối hợp hoạt động với nhau. Điều này sẽ được trình bày trong chương kế tiếp.

CÂU HỎI ÔN TẬP

1. Trình bày nhiệm vụ chính của bộ điều phối tiến trình.
2. Tại sao phải lưu lại ngữ cảnh của tiến trình khi chuyển đổi?
3. Trình bày các thuật toán điều phối chính.

Chương 6

TƯƠNG TRANH VÀ ĐỒNG BỘ

Việc quản lý tiến trình và luồng trong HĐH hiện đại được chia thành ba loại: **đa chương trình** (multiprogramming) – nhiều tiến trình chạy đồng thời trên một CPU; **đa xử lý** (multiprocessing) – nhiều tiến trình hoạt động trong hệ thống có nhiều CPU và **xử lý phân tán** (distributed system) – nhiều tiến trình hoạt động trong môi trường phân tán. Trong cả 3 kiểu trên, vấn đề thiết kế cơ bản vẫn là Tương tranh – các tiến trình hoạt động đồng thời. Mặc dù lợi ích rất lớn, nhưng tương tranh cũng làm nảy sinh nhiều vấn đề, chẳng hạn như cạnh tranh tài nguyên, bế tắc, ... Truy cập đồng thời tới dữ liệu chia sẻ có thể gây ra sự không nhất quán trong dữ liệu. Chương này sẽ trình bày các kỹ thuật khác nhau nhằm đảm bảo trong cùng không gian địa chỉ logic, khi phối hợp với nhau, các tiến trình hoạt động theo một trật tự nào đó để duy trì tính thống quán của dữ liệu.

6.1. CÁC KHÁI NIỆM CƠ BẢN

Có hai cơ chế phối hợp giữa các tiến trình là chia sẻ trực tiếp với nhau qua không gian địa chỉ logic hoặc chia sẻ dữ liệu thông qua file (tiến trình viết dữ liệu vào file và tiến trình khác đọc file). Tiến trình được gọi là cộng tác (cooperating), nếu có thể ảnh hưởng tới hoặc bị ảnh hưởng bởi tiến trình khác. Tiến trình dùng chung dữ liệu với tiến trình khác được xem là tiến trình cộng tác.

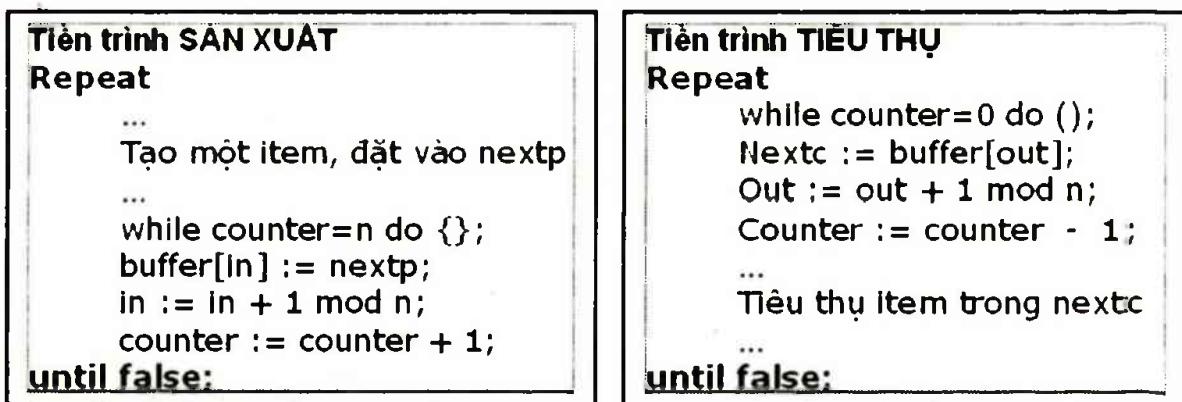
6.1.1. Hợp tác qua chia sẻ

Có nhiều lý do để tạo ra môi trường cộng tác, đó là:

- **Chia sẻ thông tin:** Khi nhiều người dùng cùng muốn sử dụng một tài nguyên thông tin nào đó (ví dụ file), thì môi trường hệ thống phải cho phép nhiều tiến trình đồng thời truy cập đến tài nguyên.

- **Tăng tốc độ tính toán:** Nếu muốn một nhiệm vụ hoàn thành trong thời gian ngắn nhất có thể, chúng ta có thể chia nhiệm vụ thành các nhiệm vụ nhỏ và các nhiệm vụ nhỏ có thể được thực thi song song. Chú ý, việc tăng tốc này chỉ có thể thực hiện khi hệ thống có nhiều đơn vị tính toán độc lập (có nhiều CPU hay các kênh vào/ra).
- **Tính module (hay tính tách biệt):** Chúng ta mong muốn chia các chức năng hệ thống thành các tiến trình riêng biệt như đã phân tích trong Chương 3.
- **Tính thuận tiện:** Người dùng có thể thực hiện song song nhiều việc như soạn thảo, in và biên dịch chương trình.

Để đảm bảo các tiến trình có thể thực thi đồng thời và cộng tác với nhau, hệ thống phải có cơ chế cho phép các tiến trình trao đổi dữ liệu cũng như đồng bộ hóa hoạt động. Bài toán sản xuất – tiêu thụ là ví dụ điển hình của vấn đề hợp tác. Tiến trình sản xuất tạo ra, còn tiến trình tiêu thụ sử dụng thông tin. Ví dụ, chương trình in tạo ra các ký tự cho driver máy in sử dụng. Để tiến trình sản xuất và tiêu thụ thực thi đồng thời, hệ thống cần bộ đệm để tiến trình sản xuất đưa thông tin vào và tiến trình tiêu thụ lấy thông tin ra. Như vậy, tiến trình sản xuất và tiến trình tiêu thụ phải được đồng bộ hóa để tiến trình tiêu thụ không được sử dụng thông tin chưa tạo ra. Khi đó, tiến trình tiêu thụ phải chờ nếu bộ đệm rỗng.

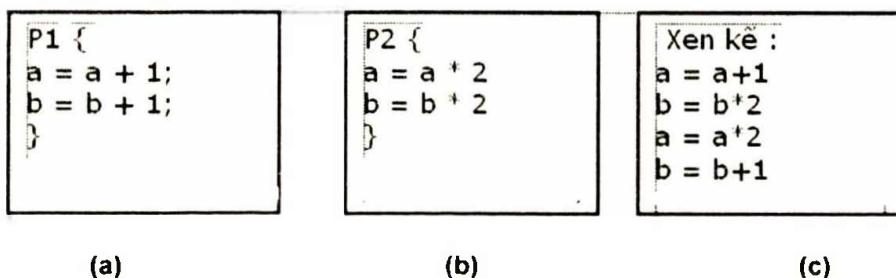


Hình 6.1. Hai tiến trình có quan hệ với nhau

Nếu độ lớn bộ đệm vô hạn, bộ phận sản xuất có thể đưa thông tin vào bất cứ lúc nào. Nhưng nếu kích thước bộ đệm hữu hạn, thì khi bộ đệm đầy thì phía sản xuất phải chờ.

Sau đây là ví dụ giải quyết bài toán bộ đệm hữu hạn bằng cách sử dụng bộ nhớ chung. Các tiến trình của cả hai phía sản xuất và tiêu thụ sử dụng chung các biến: **n**, **in**, **out**, **counter**; mảng **buffer** chứa các phần tử thuộc kiểu **item**. Giá trị khởi tạo của **in**, **out** là 0. Bộ đệm dùng chung được cài đặt thông qua mảng tuần hoàn với hai con trỏ: **in** và **out**. **in** trỏ vào vị trí trống tiếp theo trong bộ đệm, còn **out** trỏ vào vị trí đầu tiên trong bộ đệm có chứa thông tin. Bộ đệm rỗng khi **in = out**; và đầy khi $(in + 1) \bmod n = out$. Hình 6.1 minh họa cách cài đặt tiến trình sản xuất và tiêu thụ. Khối **While (điều kiện) do {}** chỉ làm nhiệm vụ kiểm tra điều kiện lặp cho đến khi điều kiện này nhận giá trị sai. Tiến trình sản xuất sử dụng biến cục bộ **nextp** để lưu thông tin mới được sinh ra và tiến trình tiêu thụ sử dụng biến cục bộ **nextc** chứa thông tin lấy ra.

Chúng ta xét tiếp ví dụ thứ 2, trong hệ thống có 2 biến **a** và **b** luôn được cập nhật thường xuyên, nhưng cần đảm bảo **a = b**. Giả sử tiến trình **P₁** và **P₂** thực hiện công việc trong Hình 6.2. Giả sử lúc đầu **a = b**, nhưng sau đó **P₁** và **P₂** chạy đồng thời. **P₁** chạy chỉ thị thứ nhất, sau đó bị phong tỏa, **P₂** được cấp phát CPU, **P₂** chạy xong thì **P₁** được khôi phục. Việc xen kẽ câu lệnh được minh họa trong Hình 6.2(c). Sau đó **a ≠ b** - hệ thống đã rơi vào trạng thái không nhất quán. Do đó, cần có cơ chế ngăn cản **P₁** và **P₂** chạy đồng thời.



Hình 6.2. Các tiến trình phối hợp

6.1.2. Hợp tác qua truyền thông

Trong hai ví dụ trên, mỗi tiến trình có ngũ cảnh hoạt động riêng, tương tác giữa chúng là gián tiếp. Hai tiến trình tuy chia sẻ biến dùng chung, nhưng không xác định nhau một cách tường minh (mặc dù vẫn phải đảm bảo tính toàn vẹn của dữ liệu). Khi hợp tác qua truyền thông, các tiến trình tham gia thực thi hướng tới một mục tiêu chung. Truyền thông giúp các tiến trình đồng bộ, kết hợp với nhau. Truyền thông nói chung được thực hiện qua

việc trao đổi thông điệp. HDH hoặc thư viện phải cung cấp các hàm cơ sở để gửi và nhận thông điệp

6.2. ĐỘC QUYỀN TRUY XUẤT – GIẢI PHÁP PHẦN MỀM

6.2.1. Nhu cầu độc quyền truy xuất

Xét ví dụ bài toán sản xuất – tiêu thụ trong mục 6.1.1, **counter** sẽ tăng mỗi khi thêm **item** mới vào bộ đếm và giảm khi xóa một **item** ra khỏi bộ đếm. Khi thực thi riêng rẽ, hai thủ tục này chạy đúng nhưng vấn đề này sinh khi thực thi đồng thời. Giả sử hiện tại **counter** = 5, **producer** và **consumer** thực thi đồng thời "**counter := counter + 1**" và "**counter := counter - 1**". Khi những câu lệnh này thực thi đồng thời, **counter** có thể nhận giá trị 4, 5 hoặc 7, mặc dù về mặt logic, kết quả chính xác phải là 5. Tại sao lại có điều này? Khi thi hành trên một dòng kiến trúc máy tính cụ thể, "**counter := counter + 1**" có thể chuyển sang ngôn ngữ máy như sau:

```
register1 := counter;  
register1 := register1 + 1;  
counter := register1;
```

với **register₁** là thanh ghi nằm trong CPU. Tương tự "**counter := counter - 1**" được chuyển thành:

```
register2 := counter;  
register2 := register2 - 1;  
counter := register2;
```

register₂ cũng là thanh ghi nằm trong CPU. Thậm chí **register₁** và **register₂** có thể là cùng một thanh ghi vật lý. Sự thực thi đồng thời "**counter := counter + 1**" và "**counter := counter - 1**" ở mức cao tương đương sự thực thi chuỗi các chỉ thị máy ở mức thấp. Giả sử các chỉ thị này xen kẽ theo thứ tự sau:

T ₀ : producer	thực thi	register ₁ := counter	{register ₁ = 5}
T ₁ : producer	thực thi	register ₁ := register ₁ + 1	{register ₁ = 6}
T ₂ : consumer	thực thi	register ₂ := counter	{register ₂ = 5}
T ₃ : consumer	thực thi	register ₂ := register ₂ - 1	{register ₂ = 4}
T ₄ : producer	thực thi	counter := register ₁	{counter = 6}
T ₅ : consumer	thực thi	counter := register ₂	{counter = 4}

Kết quả **counter = 4**. Nếu đảo ngược thứ tự T_4 và T_5 thì **counter = 6**. Nguyên nhân là cả hai tiến trình đồng thời thao tác trên biến **counter**. Các tiến trình ở trạng thái *tranh đoạt điều khiển* (race condition) khi nhiều tiến trình cùng cập nhật vào biến dùng chung và kết quả việc thực thi phụ thuộc vào thứ tự thực hiện cụ thể của các tiến trình. Do vậy, phải đảm bảo tại thời điểm cụ thể chỉ có duy nhất một tiến trình được thay đổi biến dùng chung (các tiến trình phải đồng bộ với nhau).

6.2.2. Miền găng (Critical - Section)

Xét hệ thống gồm n tiến trình $\{P_0, P_1, \dots, P_{n-1}\}$. Mỗi tiến trình có đoạn mã gọi là **miền găng** chứa các lệnh có thể thay đổi các biến dùng chung. Hệ thống phải đảm bảo tại bất kỳ thời điểm nào tối đa chỉ có một tiến trình được thi hành đoạn mã trong miền găng (gọi là bước vào miền găng). Khi đó biến dùng chung chỉ bị tác động bởi tối đa một tiến trình. Khi đó các tiến trình thay phiên nhau bước vào miền găng và hệ thống vẫn đảm bảo độc quyền truy xuất tài nguyên dùng chung. Vấn đề miền găng là thiết kế giao thức đồng bộ hóa các tiến trình. Nói chung, mỗi tiến trình phải xin phép bước vào miền găng, thực hiện cập nhật dữ liệu dùng chung rồi thông báo thoát khỏi miền găng.

Giải pháp cho miền găng phải thỏa mãn cả 3 yêu cầu sau:

1. **Độc quyền truy xuất (Mutual Exclusion)**: Nếu tiến trình P_i đang trong miền găng thì không tiến trình nào được bước vào miền găng.
2. **Tiến triển (Progress)**: Nếu không có tiến trình nào ở trong miền găng và có một số tiến trình muốn vào miền găng thì một tiến trình nào đó phải được vào miền găng.
3. **Giới hạn đợi (bounded waiting)**: Thời gian từ khi tiến trình yêu cầu cho đến khi thực sự bước vào miền găng phải bị chặn bởi giới hạn nào đó.

Ở đây chúng ta giả định mỗi tiến trình đều hoạt động, tức là có tốc độ thực thi. Tuy nhiên, không so sánh tốc độ giữa các tiến trình. Các giải pháp đưa ra ở đây không phụ thuộc vào kiến trúc phần cứng máy tính hay số lượng CPU trong hệ thống. Tuy nhiên, giả sử các chỉ thị cơ bản trong ngôn ngữ máy (chẳng hạn **load**, **store** và **test**) được thực thi đơn nhất (không thể chia nhỏ việc thực thi hơn nữa). Nếu hai chỉ thị như vậy được thực thi đồng

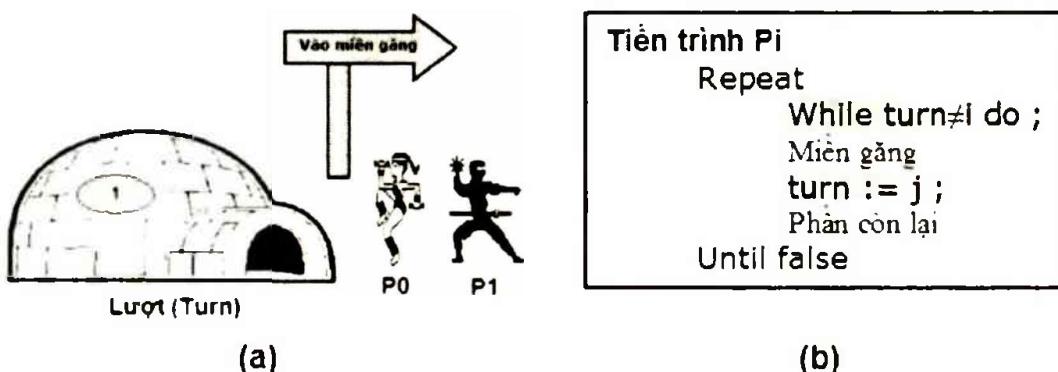
thời, thì kết quả tương đương với việc thực thi tuần tự của chúng theo trật tự nào đấy. Khi trình bày, chúng ta chỉ định nghĩa các biến phục vụ mục đích đồng bộ, còn cấu trúc chung của tiến trình P_i bất kỳ là có dạng:

```

repeat
    entry section
    Vào miền găng
    exit section
    Phần còn lại
until false;

```

Phần **entry** và **exit** được bôi đen để nhấn mạnh tầm quan trọng của chúng.



Hình 6.3. Giải pháp thứ nhất

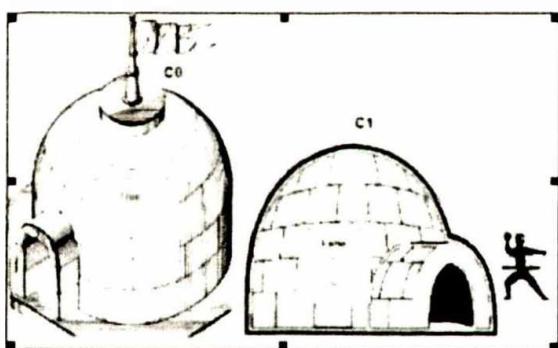
6.2.3. Giải pháp thứ nhất cho hai tiến trình

Chú ý rằng, bất kỳ giải pháp độc quyền truy xuất nào cũng đều dựa trên cơ chế độc quyền nào đó ở phần cứng. Ví dụ, tại một thời điểm, chỉ có thể xảy ra một truy cập bộ nhớ. Người ta sử dụng quy tắc "lèu Eskimo" mang tính minh họa. Lèu và cửa lèu Eskimo rất bé, chỉ có thể cho phép đúng một người đi qua (Hình 6.3a). Bảng đen trong lèu chỉ ghi được một số (0 hoặc 1). Giả sử hai người P_0 và P_1 muốn phối hợp với nhau vào miền găng. Người muốn vào miền găng đầu tiên phải vào lèu để kiểm tra bảng đen. Nếu trên bảng đen ghi tên ứng với người đó (0 với P_0 , 1 với P_1) thì được bước vào miền găng. Ngược lại, người đó phải đi ra khỏi lèu và tiếp tục đợi. Khi rời miền găng, người vừa bước vào miền găng phải quay lại lèu để ghi tên người kia trên bảng đen. Hình 6.3b là mã cho hai tiến trình P_0 và P_1 . Để

thuận tiện, khi biểu diễn P_i ta sử dụng P_j để chỉ tiến trình còn lại, tức là $j = 1 - i$. Hai tiến trình cùng chia sẻ biến **turn** (khởi tạo bằng 0 hoặc 1). Biến **turn** giống bảng đen. Nếu **turn** = i thì P_i được phép bước vào miền găng. Giải pháp này bảo đảm tại một thời điểm chỉ có duy nhất một tiến trình ở trong miền găng. Tuy nhiên, yêu cầu tiến triển không được đảm bảo vì thuật toán này luôn đòi hỏi các tiến trình phải luân phiên bước vào miền găng. Chẳng hạn, khi **turn** = 0, P_0 biến mất thì mãi mãi P_1 không thể vào được miền găng.

6.2.4. Giải pháp thứ hai cho hai tiến trình

Vấn đề của giải pháp 1 là chỉ ghi thông tin về người muốn vào miền găng trên bảng đen, trong khi cần biết trạng thái của cả hai tiến trình. Để giải quyết, mỗi người phải có dấu hiệu xác định họ có vào miền găng hay không, khi đó dù người này biến mất thì người kia vẫn có thể vào miền găng mà không bị ảnh hưởng gì. Quy tắc được sửa đổi như sau: Mỗi người có lều riêng (C_0 và C_1) và có thể xem (nhưng không được quyền sửa) bảng đen trong lều của người kia. Người muốn vào miền găng sẽ định kỳ kiểm tra bảng đen trong lều người kia cho đến khi nhìn thấy có chữ **false** (dấu hiệu người kia không muốn vào). Lúc này người đó quay về lều của mình, viết **true** lên bảng đen (thông báo mình vào miền găng) và bước vào miền găng. Sau khi rời miền găng, người đó viết **false** lên bảng đen của mình.



```

Tiến trình  $P_i$ 
var flag: array[0..1] of boolean;
Repeat
    flag[i] := true ;
    while flag[j] do;
        Miền găng
        flag[i] := false;
        Phản còn lại
    Until false;

```

(a)

(b)

Hình 6.4. Thuật toán thứ hai

Thay thế **turn** bằng mảng **flag: array[0..1] of boolean** đóng vai trò lều riêng của mỗi người (C_0 và C_1). Các phần tử của mảng được khởi tạo là **false**. Giá trị **flag[i] = true** có ý nghĩa P_i sẵn sàng bước vào miền găng. Cấu trúc P_i minh họa trong Hình 6.4b. Đầu tiên P_i đặt **flag[i] = true** để thông báo

mình sẵn sàng bước vào miền gǎng. Sau đó P_i kiểm tra xem P_j đã bước vào miền gǎng chưa. Nếu P_j đã (hoặc chuẩn bị) vào, P_i sẽ đợi cho tới khi P_j bước ra khỏi miền gǎng (tức là $\text{flag}[j] = \text{false}$). Khi đó P_i mới được vào miền gǎng. Khi rời miền gǎng, P_i đặt lại $\text{flag}[i] = \text{false}$ để cho phép P_j (nếu có nhu cầu) bước vào miền gǎng.

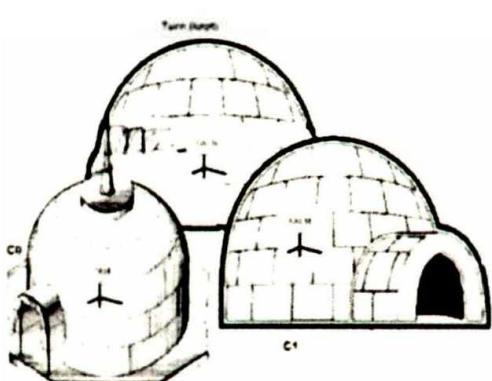
Thuật toán này không đảm bảo điều kiện tiến triển. Xét chuỗi thực thi dưới đây khi P_0 và P_1 đồng thời muốn vào miền gǎng:

$T_0: P_0$ đặt $\text{flag}[0] = \text{true}$

$T_1: P_1$ đặt $\text{flag}[1] = \text{true}$

Bây giờ P_0 và P_1 sẽ lặp mãi trong vòng lặp **while**. Thuật toán phụ thuộc về thời gian thực hiện của hai tiến trình. Trình tự thực hiện trên có thể xuất hiện do trong hệ thống có nhiều CPU hoặc ngắn (chẳng hạn, do bộ định thời gây ra) xuất hiện ngay sau T_0 và quyền điều khiển CPU được chuyển sang tiến trình khác. Thay đổi thứ tự câu lệnh, đặt $\text{flag}[i] = \text{true}$ và câu lệnh kiểm tra giá trị của $\text{flag}[j]$ cũng không giải quyết được vấn đề này, mà còn có thể cho phép cả hai tiến trình cùng bước vào miền gǎng, vi phạm yêu cầu độc quyền truy xuất.

6.2.5. Giải pháp thứ ba cho hai tiến trình



(a)

Tiến trình P_i

```
var flag: array[0..1] of boolean;
turn: 0..1;
Repeat
    flag[i] := true;
    turn := j;
    while(flag[j] and turn=j) do;
    Miền gǎng
    flag[i] := false;
    Phần còn lại
Until false;
```

(b)

Hình 6.5. Thuật toán 3

Kết hợp hai thuật toán trên, chúng ta đưa ra giải pháp hoàn chỉnh cho miền gǎng đáp ứng cả 3 yêu cầu. Bên cạnh lều riêng của mỗi người, còn có lều đóng vai trò "trọng tài", trong đó có bảng đèn ghi người nào được quyền vào miền gǎng (Hình 6.5a). Các tiến trình chia sẻ mảng **flag** và biến **turn**.

Ban đầu, $\text{flag}[0] = \text{flag}[1] = \text{false}$, giá trị khởi tạo của turn (0 hay 1) không quan trọng. Cấu trúc P_i được minh họa trong Hình 6.5b. Để vào miền găng, P_i đặt $\text{flag}[i] = \text{true}$ và sau đó kiểm tra xem P_j đã vào miền găng chưa. Nếu cả hai tiến trình cùng muốn vào miền găng, turn sẽ được gán giá trị i và j gần như đồng thời. Lệnh gán thực hiện sau sẽ ghi đè lên kết quả của lệnh gán trước. Giá trị sau cùng của turn quyết định tiến trình nào được phép vào miền găng.

Bây giờ chúng minh thuật toán này thỏa mãn cả ba yêu cầu đã nêu trong 6.2.2. Để chứng minh (1), ta thấy P_i chỉ vào miền găng khi $\text{flag}[j] = \text{false}$ hoặc $\text{turn} = i$. Giả sử khi cả hai tiến trình cùng muốn vào miền găng một lúc thì $\text{flag}[0] = \text{flag}[1] = \text{true}$. Tuy nhiên, hai tiến trình này không thể bước vào miền găng cùng một lúc vì turn chỉ có thể nhận giá trị 0 hoặc 1. Như vậy, điều kiện độc quyền truy xuất được bảo đảm. Để chứng minh điều kiện (2) và (3), chú ý tiến trình P_i bị ngăn cản vào miền găng khi và chỉ khi nó bị tắc trong vòng lặp **while do** điều kiện $\text{flag}[j] = \text{true}$ và $\text{turn} = j$; vòng lặp này chỉ được thực hiện đúng một lần. Nếu P_j không sẵn sàng vào miền găng thì $\text{flag}[j] = \text{false}$ và P_i có thể vào miền găng. Nếu P_j đặt $\text{flag}[j] = \text{true}$ và đang thực thi trong vòng lặp **while** thì $\text{turn} = i$ hoặc bằng j . Nếu $\text{turn} = i$, P_i sẽ được vào miền găng. Nếu $\text{turn} = j$, P_j sẽ vào miền găng. Tuy nhiên, khi ra khỏi miền găng, P_j sẽ đặt lại $\text{flag}[j] = \text{false}$ để cho phép P_i vào miền găng. Nếu đặt $\text{flag}[j] = \text{true}$ thì P_j cũng sẽ đặt $\text{turn} = i$. Vì P_i không thay đổi giá trị biến turn trong vòng lặp **while**, P_i sẽ vào miền găng (thỏa mãn yêu cầu tiến triển) sau khi P_j vào nhiều nhất một lần (thỏa mãn yêu cầu giới hạn đợi).

6.2.6. Giải pháp đa tiến trình

Thuật toán 3 giải quyết vấn đề miền găng cho hai tiến trình. Bây giờ sẽ trình bày thuật toán *Hiệu bánh mỳ* cho n tiến trình. Thuật toán này có tên gọi như vậy, vì điều phối việc phục vụ tại những nơi đòi hỏi phải duy trì trật tự trong tình trạng lộn xộn (các cửa hàng).

Khi vào cửa hàng, khách hàng nhận được một số thứ tự (STT). Cửa hàng sẽ phục vụ khách hàng có STT nhỏ nhất. Tuy nhiên, thuật toán không đảm bảo hai tiến trình (hai khách hàng) có STT khác nhau. Khi đó tiến trình với tên xếp trước (theo thứ tự nào đó) được phục vụ trước. Vì vậy, nếu P_i và P_j có cùng STT và $i < j$ thì P_i có độ ưu tiên cao hơn.

Cấu trúc dữ liệu chung là hai mảng **choosing**: array[0 .. n - 1] of boolean và **number**: array[0 .. n - 1] of integer. Ban đầu cấu trúc dữ liệu này được khởi tạo lần lượt là **false** và 0. Để thuận tiện, chúng ta định nghĩa $(a, b) < (c, d)$ nếu $a < c$ hoặc nếu $a = c$ và $b < d$. Cấu trúc P_i được minh họa trong Hình 6.6. Để chứng minh tính đúng đắn của thuật toán Hiệu bánh mỳ, đầu tiên ta cần chứng minh nếu P_i trong miền găng và P_k ($k \neq i$) có STT **number[k]** ≠ 0 thì **(number[i], i) < (number[k], k)**. Với kết quả này dễ dàng chứng minh được điều kiện độc quyền truy xuất. Thực vậy, coi P_i trong miền găng và P_k đang có găng vào miền găng. Khi thực thi câu lệnh **while** thứ hai trong trạng thái $j = i$, P_k thấy **number[i] ≠ 0** và **(number[i], i) < (number[k], k)**. Vòng lặp tiếp tục cho tới khi P_i ra khỏi miền găng. Để chứng minh yêu cầu tiền triển và giới hạn đợi, hãy để ý các tiền trình vào miền găng theo thứ tự đến trước phục vụ trước.

Repeat

```

choosing[I]:=true;
number[I] := max(number[0], number[1],...,number[n-1]) + 1;
choosing[I]:= false;
for j := 0 to n-1 do
    begin
        while choosing[j] do;
        while number[j] ≠ 0 and (number[j],j)< (number[i],i) do;
    end;
Vào miền găng
number[I]:= 0;
Phản còn lại
Until false

```

Hình 6.6. Thuật toán vào miền găng cho nhiều tiền trình

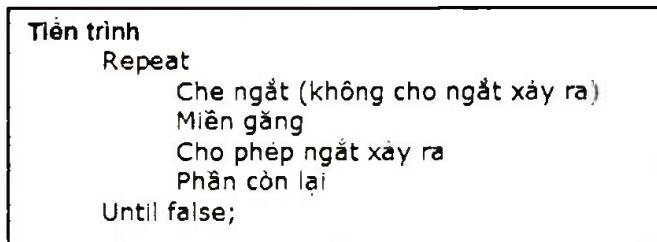
6.3. ĐÓNG BỘ HÓA – GIẢI PHÁP PHẦN CỨNG

Phần cứng có thể giúp việc lập trình đơn giản hơn và nâng cao hiệu suất hệ thống. Phần này giới thiệu một vài giải pháp bằng phần cứng có khả năng giải quyết hiệu quả vấn đề miền găng.

6.3.1. Che ngắt

Trong hệ thống có một bộ vi xử lý, có thể giải quyết vấn đề miền găng bằng cách che ngắt (không cho ngắt xảy ra), trong khi chỉnh sửa biến dùng chung. Như vậy, hệ thống đảm bảo chuỗi chỉ thị thao tác trên biến dùng

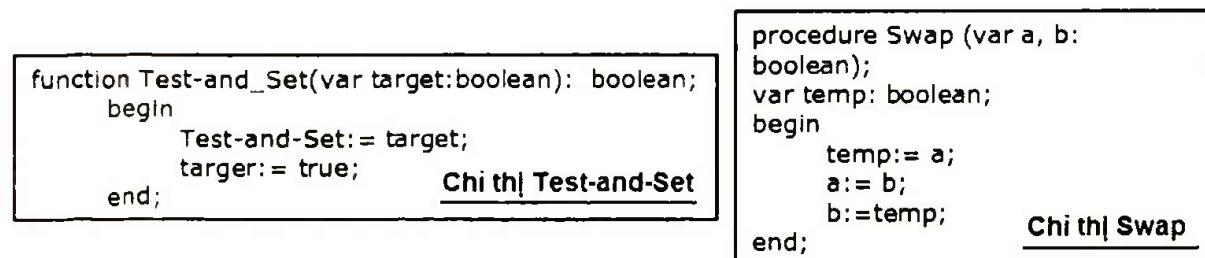
chung được thực hiện trọn vẹn mà không bị gián đoạn bởi tiến trình khác. Do các tiến trình khác không thể xen vào, nên biến dùng chung không bị thay đổi một cách bất thường. Thuật toán thực hiện như minh họa trên Hình 6.7. Tuy nhiên, che ngắt trên hệ thống nhiều CPU tốn thời gian hơn vì phải gửi thông điệp đến tất cả các CPU. Tiến trình không thể vào miền găng ngay, hiệu suất hệ thống bị suy giảm.



Hình 6.7. Miền găng bằng cách che ngắt

6.3.2. Các chỉ thị đặc biệt

Nhiều kiến trúc máy tính có chỉ thị phần cứng đặc biệt cho phép kiểm tra và chỉnh sửa nội dung một từ hoặc tráo đổi nội dung hai từ trong bộ nhớ một cách đơn nhất. Chúng có thể sử dụng giải quyết vấn đề miền găng.

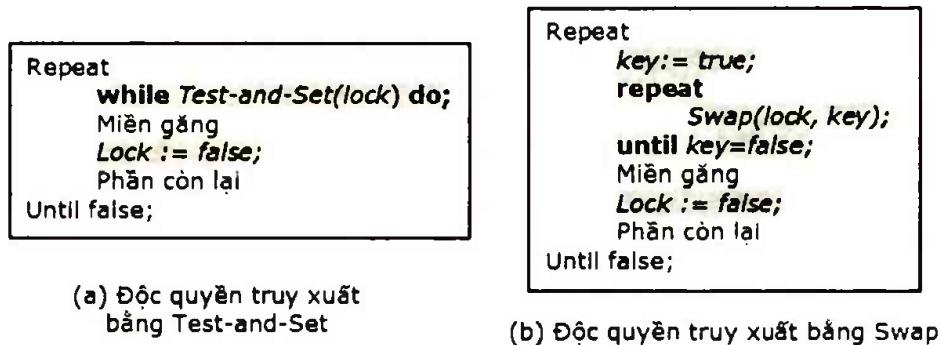


Hình 6.8. Các chỉ thị đặc biệt

Đặc điểm quan trọng của chỉ thị kiểu này là sự thực thi một cách nguyên vẹn và đơn nhất, nghĩa là quá trình thực hiện chỉ thị không bị gián đoạn. Vì vậy, nếu hai chỉ thị **Test-and-Set** được thực thi đồng thời (trên các CPU khác nhau), chúng sẽ thực thi tuần tự theo thứ tự nào đó.

Với chỉ thị **Test-and-Set**, có thể thi hành độc quyền truy xuất bằng việc khai báo biến **boolean lock** khởi tạo giá trị **false**. Cấu trúc P_i được minh họa trong Hình 6.8. Chỉ thị **swap** tráo đổi nội dung 2 từ được định nghĩa trong Hình 6.8. Giống **Test-and-Set**, **swap** được thực thi một cách đơn nhất. Hình 6.9 minh họa cách thực hiện độc quyền truy xuất. Ưu điểm ở đây là khả

năng áp dụng đơn giản trên hệ thống có một hoặc có nhiều CPU. Tuy nhiên, hiện tượng busy-waiting, khả năng chết đói và bế tắc vẫn còn có thể xảy ra.

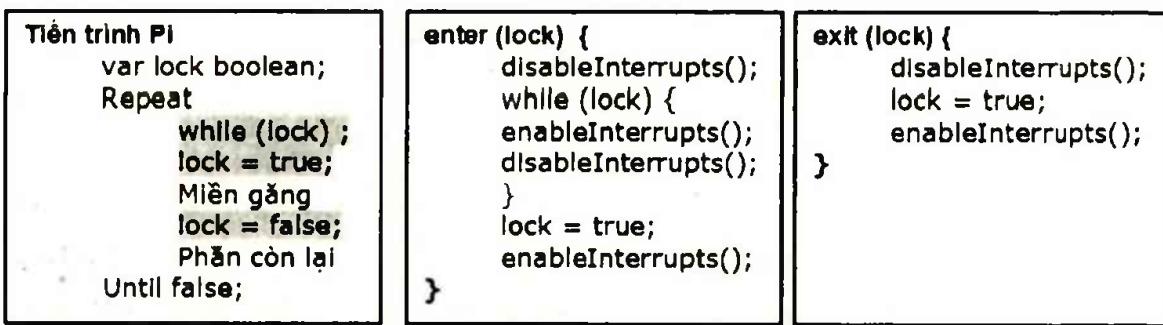


Hình 6.9. Độc quyền truy xuất bằng các chỉ thị đặc biệt

6.4. GIẢI PHÁP ĐỒNG BỘ CƠ BẢN

6.4.1. Khóa (Lock)

Với ví dụ về "lều Eskimo", chúng ta giả sử ở cửa lều có một khóa. Người muốn vào miền găng sẽ đến trước cửa lều. Nếu lều bị khóa, người đó phải đợi. Nếu lều chưa khóa, người đó khóa lều lại, đi vào miền găng (cầm theo chìa khóa). Sau khi thoát khỏi miền găng, người đó quay trở về mở khóa lều cho những người muốn vào miền găng. Hình 6.10a minh họa cách thức sử dụng khóa khi muốn vào miền găng. Tuy nhiên, cần giải quyết vấn đề hai người đồng thời muốn dùng khóa. Người ta có thể sử dụng giải pháp che ngắt khi muốn đóng và mở khóa (minh họa trên Hình 6.10b và c).



(a) Cách sử dụng khóa

(b) Đóng khóa

(c) Mở khóa

Hình 6.10. Khóa và cách cài đặt khóa

Nhược điểm chính của các giải pháp trên là tình trạng chờ bận (*Busy waiting*). Khi có tiến trình ở trong miền găng, bất kỳ tiến trình khác muốn vào miền găng sẽ thực hiện vòng lặp (trong đoạn mã xin vào miền găng) để

kiểm tra xem đã đến lượt mình vào chưa. Tình trạng *Busy waiting* lãng phí các chu kỳ CPU mà lẽ ra các tiến trình khác có thể sử dụng để thực hiện công việc hữu ích. Tình trạng này còn được gọi là khóa xoay (*Spinlock*) (vì các tiến trình quay quanh vòng lặp trong khi chờ khóa). Ưu điểm của *spinlock* trong hệ thống đa bộ xử lý là không cần phải chuyển ngữ cảnh khi một tiến trình đợi khóa (vì chuyển ngữ cảnh có thể chiếm một lượng thời gian đáng kể).

6.4.2. Semaphore

Semaphore S là biến nguyên mà sau khi khởi tạo chỉ được truy cập qua hai thao tác đơn nhất là **wait** và **signal**. Hình 6.11a là định nghĩa của **wait** và **signal**. Toán tử **wait** và **signal** thay đổi giá trị semaphore được thực hiện một cách đơn nhất. Tức là khi một tiến trình đang thay đổi giá trị semaphore thì không tiến trình nào được quyền thay đổi giá trị semaphore. Ngoài ra, trong trường hợp **wait(S)**, việc kiểm tra giá trị **S** (**S ≤ 0**) và việc thay đổi **S := S - 1** (nếu có) phải được thực thi liên tục không gián đoạn. Cách cài đặt các toán tử này được trình bày trong mục 6.4.3. Nhưng trước tiên ta trình bày cách sử dụng semaphore.

wait(S):	while S ≤ 0 do; S := S-1;
signal(S):	S := S+1;

(a) Thao tác Wait và Signal

Repeat	Wait(mutex); Miền găng Signal (mutex); Phản còn lại
Until false;	

(b) Độc quyền truy xuất bằng Semaphore

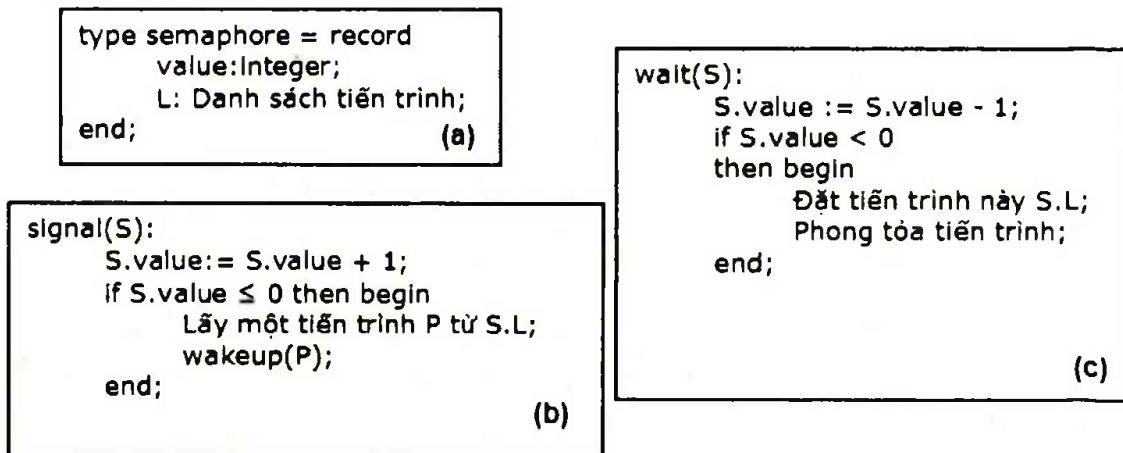
Hình 6.11. Định nghĩa và cách sử dụng Semaphore

Có thể sử dụng semaphore để giải quyết vấn đề miền găng cho n tiến trình. n tiến trình này chia sẻ semaphore **mutex** (**mutual exclusion**) khởi tạo giá trị 1. Tiến trình P_i minh họa trong Hình 6.11b. "Lèu Eskimo" trong ví dụ trước được cải tiến như sau: Bảng đèn sử dụng để ghi số lượng người trong lèu (lèu có thể bỏ trí nhiều ghế ngồi). Người muốn vào miền găng (giả sử A) bước vào lèu, tăng giá trị ghi trong bảng đèn lên 1. Nếu giá trị này là 1, A có quyền bước vào miền găng. Nếu giá trị này khác 1, A phải ngồi đợi trong lèu. Sau khi thoát khỏi miền găng, A phải quay trở về lèu, giảm giá trị ghi trên bảng đèn đi 1 và nếu giá trị này khác 0, thì người khác đang đợi trong lèu được vào miền găng.

Ngoài ra, cũng có thể sử dụng semaphore để giải quyết nhiều vấn đề đồng bộ hóa khác. Giả sử hai tiến trình chạy đồng thời: P₁ với lệnh S₁ và P₂ với lệnh S₂. Giả thiết rằng S₂ chỉ được thực thi sau khi S₁ đã thực thi xong. Có thể giải quyết vấn đề này bằng cách cho P₁ và P₂ dùng chung **semaphore synch** khởi tạo bằng 0 và chèn vào câu lệnh: {S1;signal(synch);} cho P₁ và câu lệnh {wait(synch);S2;} cho P₂. Vì **synch** khởi tạo bằng 0, P₂ sẽ thực thi S₂ chỉ sau khi P₁ thực hiện **signal(synch)** sau S₁.

6.4.3. Cài đặt semaphore

Với định nghĩa trên, semaphore vẫn bị tình trạng *busy waiting*. Để khắc phục cần định nghĩa lại toán tử **wait** và **signal** của semaphore. Tiến trình gọi **wait** và thấy giá trị semaphore không dương sẽ phải chờ. Tuy nhiên, thay vì thực hiện vòng lặp, tiến trình phong tỏa chính nó. Toán tử **block** đặt tiến trình vào hàng đợi (hang đợi này gắn với semaphore) và tiến trình chuyển sang trạng thái đợi (waiting). Sau đó điều khiển được chuyển cho bộ điều phối CPU để lựa chọn tiến trình khác thực thi. Tiến trình bị phong tỏa chờ semaphore S sẽ được khởi động lại (bằng toán tử **wakeup**) khi tiến trình nào đấy thi hành toán tử **signal**. Khi đó trạng thái tiến trình này chuyển từ đợi (waiting) sang sẵn sàng (ready).



Hình 6.12. Cài đặt semaphore

Semaphore có thể cài đặt dưới dạng bản ghi (Hình 6.12a), với hai thành phần: một giá trị nguyên và một danh sách các tiến trình đợi. Toán tử **signal** lấy một tiến trình trong danh sách đợi để kích hoạt. Hình 6.12b và c minh họa toán tử **signal** và **wait**. Khi gọi **block**, tiến trình sẽ tự treo (chuyển sang trạng thái chờ) và chuyển sang nằm trong hàng đợi. Toán tử **wakeup(P)** khôi

phục tiến trình P bị phong tỏa. Hai toán tử này được HDH cài đặt dưới dạng các lời gọi hệ thống cơ bản.

Theo định nghĩa cũ, semaphore thuộc kiểu *busy waiting* và giá trị semaphore không âm. Semaphore theo kiểu mới có thể nhận giá trị âm, khi đó độ âm xác định số lượng các tiến trình đợi trên semaphore. Thực tế này là do trong toán tử **wait**, chúng ta dào lệnh trù lén trước lệnh kiểm tra. Để dàng cài đặt danh sách các tiến trình đợi thông qua trường liên kết trong khói điều khiển tiến trình (PCB). Mỗi semaphore chứa một giá trị nguyên và một con trỏ trỏ tới danh sách PCB. Để thêm và xóa các tiến trình khỏi danh sách và để bảo đảm điều kiện giới hạn đợi, hệ thống có thể sử dụng hàng đợi FIFO (first-in, first-out). Khi đó, semaphore trỏ tới cả hai đầu của hàng đợi. Chú ý là cách sử dụng semaphore chính xác không phụ thuộc vào cách cài đặt danh sách đợi trong semaphore.

Toán tử semaphore phải được thực hiện một cách toàn vẹn và đơn nhất, nghĩa là hai tiến trình không thể cùng lúc lại có thể thực thi toán tử **wait** và **signal** trên cùng một semaphore. Đây là vấn đề miền găng và có thể giải quyết theo hai cách sau: Trong môi trường có một CPU, có thể che ngắt khi thực thi **wait** hoặc **signal**. Khi đó chỉ thị của các tiến trình khác không thể xen kẽ vào chỉ thị của toán tử. Nhưng trong môi trường nhiều CPU, khó thực hiện việc chặn ngắt. Chỉ thị của các tiến trình khác nhau (chạy trên các CPU khác nhau) có thể xen kẽ với nhau tùy ý. Nếu phần cứng không có chỉ thị đặc biệt thì có thể sử dụng giải pháp phần mềm cho vấn đề miền găng, khi đó tình trạng busy waiting vẫn xuất hiện với thủ tục **wait** và **signal**. Chúng ta chỉ loại bỏ tình trạng busy waiting khi chương trình ứng dụng bước vào miền găng. Tình trạng busy waiting bị giới hạn trong miền găng của toán tử **wait** và **signal**. Do độ lớn của miền này khá nhỏ (nếu được tối ưu sẽ không vượt quá 10 chỉ thị), nên nếu tình trạng busy waiting có xuất hiện thì cũng chỉ trong thời gian rất ngắn.

6.4.4. Bé tắc và Chết đói

Cài đặt semaphore như trên có thể dẫn đến trường hợp nhiều tiến trình đợi vô vọng một sự kiện do một tiến trình khác cũng đang trong trạng thái đợi gây ra. Sự kiện đợi ứng với việc thực thi toán tử **wait**. Khi hệ thống rơi vào tình huống này, các tiến trình liên quan sẽ ở trong tình trạng bế tắc. Ví

dụ, hệ thống có hai tiến trình P_0 và P_1 , mỗi tiến trình truy cập vào hai semaphore S và Q để đặt giá trị 1:

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
...	...
signal(S);	signal(Q);
signal(Q);	signal(S);

Giả sử P_0 thực thi **wait(S)**, sau đó P_1 thực thi **wait(Q)**. Khi thực thi **wait(Q)**, P_0 phải đợi cho đến khi P_1 thực thi xong **signal(Q)**. Tương tự, khi thực thi **wait(S)**, P_1 phải đợi cho đến khi P_0 thực thi xong **signal(S)**. Rõ ràng hai toán tử **signal** này không thể được thực thi, P_0 và P_1 rơi vào bế tắc. Ta nói một tập hợp các tiến trình trong tình trạng bế tắc nếu mỗi tiến trình trong tập đợi một sự kiện do một tiến trình khác trong tập gây ra. Phần lớn các sự kiện trình bày ở đây liên quan đến cấp phát tài nguyên. Tuy nhiên, có nhiều loại sự kiện khác cũng có thể gây ra bế tắc và sẽ được trình bày trong chương sau. Vấn đề khác liên quan đến tình trạng bế tắc là trạng thái bị phong tỏa vĩnh viễn hay chết đói (starvation), là hiện tượng khi các tiến trình chờ vô định trong semaphore. Phong tỏa vĩnh viễn có thể xảy ra nếu ta thêm và xóa các tiến trình khỏi danh sách gắn với semaphore theo thứ tự vào sau ra trước (LIFO).

6.4.5. Semaphore nhị phân

Cấu trúc semaphore nói trên gọi là *counting semaphore*, vì giá trị nguyên có thể nhận giá trị tùy ý.Semaphore nhị phân là semaphore mà giá trị nguyên chỉ là 0 hoặc 1. Cài đặt semaphore nhị phân đơn giản hơn counting semaphore do tận dụng được kiến trúc phần cứng. Bây giờ sẽ trình bày việc cài đặt counting semaphore dựa trên semaphore nhị phân. Để cài đặt counting semaphore S bằng semaphore nhị phân, ta sử dụng các cấu trúc dữ liệu sau: **S1: binary-semaphore; S2: binary-semaphore; C: integer.** Khoi đầu **S1 = 1; S2 = 0** và **C** nhận giá trị ban đầu của counting semaphore S. Toán tử **wait** và **signal** trong counting semaphore được cài đặt trong Hình 6.13.

signal():

```

wait(S1);
C := C + 1;
if (C <= 0) then signal (S2);
else signal(S1);

```

wait():

```

wait(s1);
C := C - 1;
if C < 0 then begin
    signal(S1);
    wait(S2);
end
signal(S1);

```

Hình 6.13. Cài đặt counting semaphore bằng semaphore nhị phân

6.5. NHỮNG VẤN ĐỀ ĐỒNG BỘ KINH ĐIỀN

Trong phần này, trình bày một số vấn đề đồng bộ liên quan đến điều khiển song song. Những vấn đề này thường được sử dụng như bài toán mẫu để kiểm tra các phương pháp đồng bộ hóa mới.

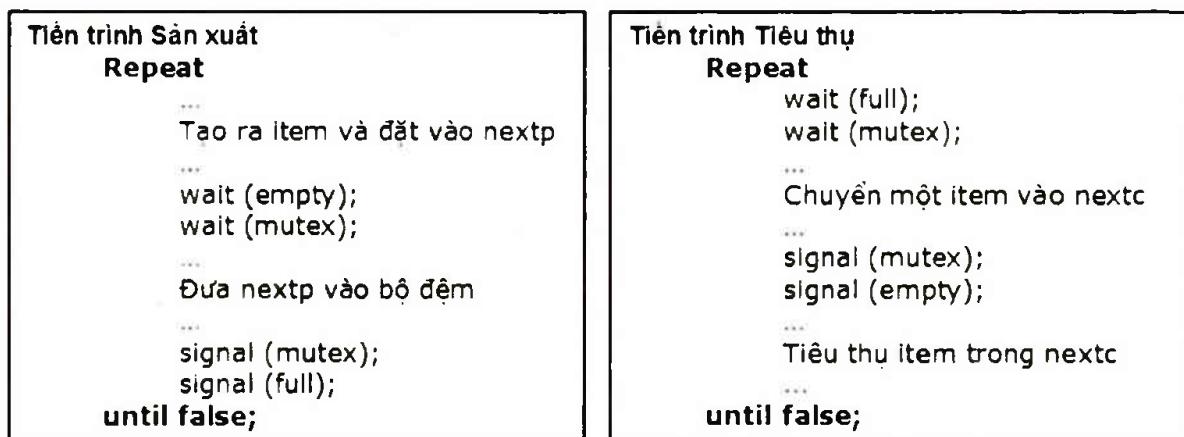
6.5.1. Vấn đề bộ đệm giới hạn (The Bounded-Buffer Problem)

Ở đây trình bày giải pháp tổng quát cho vấn đề Bộ đệm giới hạn (xem mục 6.1). Giả sử vùng đệm dùng chung có khả năng lưu giữ n item. Semaphore **Mutex** đảm bảo độc quyền truy xuất với vùng đệm và được khởi tạo giá trị 1.Semaphore **empty** và **full** đếm số bộ đệm trong vùng chưa sử dụng và số bộ đệm đã được sử dụng. Semaphore **empty** được khởi tạo giá trị n, semaphore **full** được khởi tạo giá trị 0. Mã cho tiến trình producer và consumer được minh họa trong Hình 6.14.

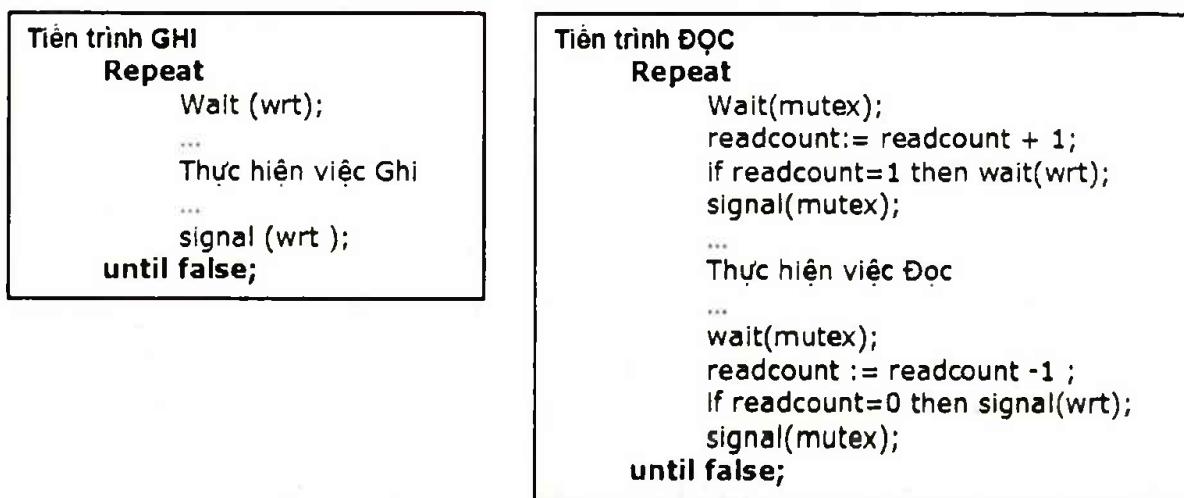
6.5.2. Vấn đề đọc – ghi (The Readers and Writers)

Nhiều tiến trình hoạt động đồng thời có thể dùng chung đối tượng dữ liệu (file hay bản ghi). Đối tượng có thể truy xuất theo chế độ chỉ đọc (reader) hoặc chế độ cập nhật (writer) vào đối tượng dùng chung. Hiển nhiên, sẽ không có chuyện gì xảy ra nếu hai tiến trình reader cùng đọc. Tuy nhiên, cần phải đồng bộ khi một tiến trình writer và một số tiến trình khác (writer hoặc reader) cùng truy cập đồng thời đến đối tượng dùng chung. Giải pháp ở đây là tiến trình writer phải được độc quyền truy cập tới đối tượng dùng chung. Đây là vấn đề đồng bộ đọc – ghi và thường được sử dụng để

kiểm tra các công cụ đồng bộ mới. Vấn đề đọc – ghi có một số biến thể liên quan đến quyền ưu tiên. Dị bản đơn giản nhất gọi là vấn đề đọc – ghi đầu tiên, yêu cầu không tiến trình reader nào phải đợi trừ khi có tiến trình writer đã được phép sử dụng đối tượng dùng chung. Nói cách khác, không tiến trình reader nào phải đợi tiến trình reader khác hoàn thành chỉ vì có một tiến trình writer đang đợi.



Hình 6.14. Bài toán sản xuất – tiêu thụ



Hình 6.15. Reader và Writer

Trong dị bản thứ hai, tiến trình writer sẵn sàng sẽ được ghi ngay khi có thể. Tức là nếu tiến trình writer đang đợi để truy cập vào đối tượng chia sẻ, không tiến trình reader mới nào được đọc. Chú ý rằng, hai phiên bản trên có thể dẫn đến tình trạng "chết đói": trường hợp đầu tiên là writer, trường hợp thứ hai là reader. Vì lý do này, nhiều biến thể khác của vấn đề đã được đưa ra. Trong phần này, chúng ta trình bày giải pháp cho vấn đề đọc – ghi thứ nhất. Sử dụng cấu trúc dữ liệu dùng chung sau đây: hai semaphore **mutex**,

wrt và số nguyên **readcount**. Giá trị khởi tạo của semaphore **mutex** và **wrt** là 1, của **readcount** là 0. Semaphore **wrt** được dùng chung giữa hai tiến trình writer và reader, semaphore **mutex** được dùng để đảm bảo độc quyền truy xuất với **readcount**. **Readcount** đếm số tiến trình đang đọc đối tượng.Semaphore **wrt** thực hiện độc quyền truy xuất giữa các writer. Ngoài ra **wrt** còn được dùng bởi tiến trình reader đầu tiên hoặc cuối cùng khi bước vào hay bước ra miền găng. Hai tiến trình này được minh họa trên Hình 6.15. Chú ý, khi một tiến trình writer đang ở trong miền găng thì với n tiến trình reader đợi, có một tiến trình reader xếp hàng trong **wrt** và $n - 1$ tiến trình còn lại xếp hàng trong **mutex**. Khi tiến trình writer thi hành **signal(wrt)**, hệ thống có thể khôi phục lại hoặc một tiến trình reader, hoặc một tiến trình writer đang chờ. Lựa chọn như thế nào phụ thuộc vào bộ điều phối.

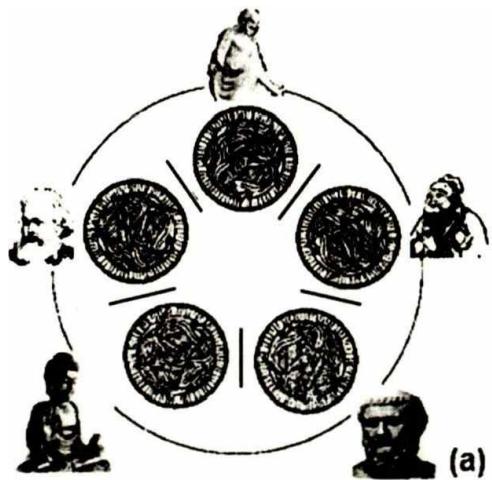
6.5.3. "Bữa ăn tối của các triết gia"

Giả sử có 5 triết gia chỉ suy nghĩ và ăn. Các triết gia ngồi quanh bàn tròn, trên bàn có một nồi cơm và 5 chiếc đũa (Hình 6.16a). Các triết gia không trao đổi với nhau. Khi đó triết gia cố gắng nhất hai chiếc đũa gần mình nhất. Trong một lần lấy, triết gia chỉ có thể nhặt lên được một chiếc đũa. Triết gia không thể "cướp" đũa ở trong tay người khác. Khi có cả đôi đũa, triết gia sẽ ăn và buông đôi đũa của mình sau khi ăn xong. Sau đó triết gia lại tiếp tục suy nghĩ. Bữa ăn của các triết gia được xem là vấn đề đồng bộ hóa kinh điển, không phải chỉ vì sự quan trọng trong thực tế mà còn là ví dụ của một lớp lớn các vấn đề kiểm soát đồng bộ. Chẳng hạn, việc cấp phát tài nguyên cho các tiến trình sao cho hệ thống không rơi vào trạng thái bế tắc hay chết dói.

Một giải pháp đơn giản là xem chiếc đũa là semaphore. Hành động Nhật đũa thực hiện qua toán tử **wait** và hành động Buông đũa thực hiện qua toán tử **signal**. Như vậy, dữ liệu được chia sẻ là: **chopstick: array [0..4] of semaphore**; ở đây tất cả các phần tử trong mảng **chopstick** được khởi tạo là 1. Cấu trúc triết gia thứ i được minh họa trên Hình 6.16b. Mặc dù đảm bảo hai triết gia ngồi cạnh nhau không thể ăn cùng lúc, giải pháp này không được sử dụng vì có thể gây bế tắc. Giả sử 5 triết gia đói cùng lúc, mỗi người lấy chiếc đũa bên trái. Khi đó tất cả phần tử của mảng **chopstick** nhận giá trị 0. Khi cố lấy chiếc đũa bên phải, tất cả các triết gia sẽ bị đợi mãi mãi. Chúng ta có thể cải tiến để giảm bớt tình trạng bế tắc, chẳng hạn:

- Chỉ cho phép triết gia nhặt lên một chiếc đũa khi đôi đũa chưa được sử dụng.
- Sử dụng giải pháp phi đối xứng. Triết gia mang số lẻ nhặt chiếc đũa trái trước chiếc đũa phải, còn triết gia mang số chẵn nhặt chiếc đũa phải trước chiếc đũa trái.

Bất cứ giải pháp nào cho vấn đề Bữa ăn của các triết gia phải ngăn ngừa trường hợp một triết gia nào đó bị chết đói. Giải quyết bế tắc chưa chắc loại trừ được tình trạng chết đói.



Thuật toán của các Triết gia:
Repeat

```
wait (chopstick[i]);  
wait (chopstick[i+1 mod 5]);
```

...

```
signal (chopstick[i]);  
signal (chopstick[i+1 mod 5]);
```

...

Suy nghĩ

...

Until false;

(b)

Hình 6.16. Bữa ăn tối của triết gia

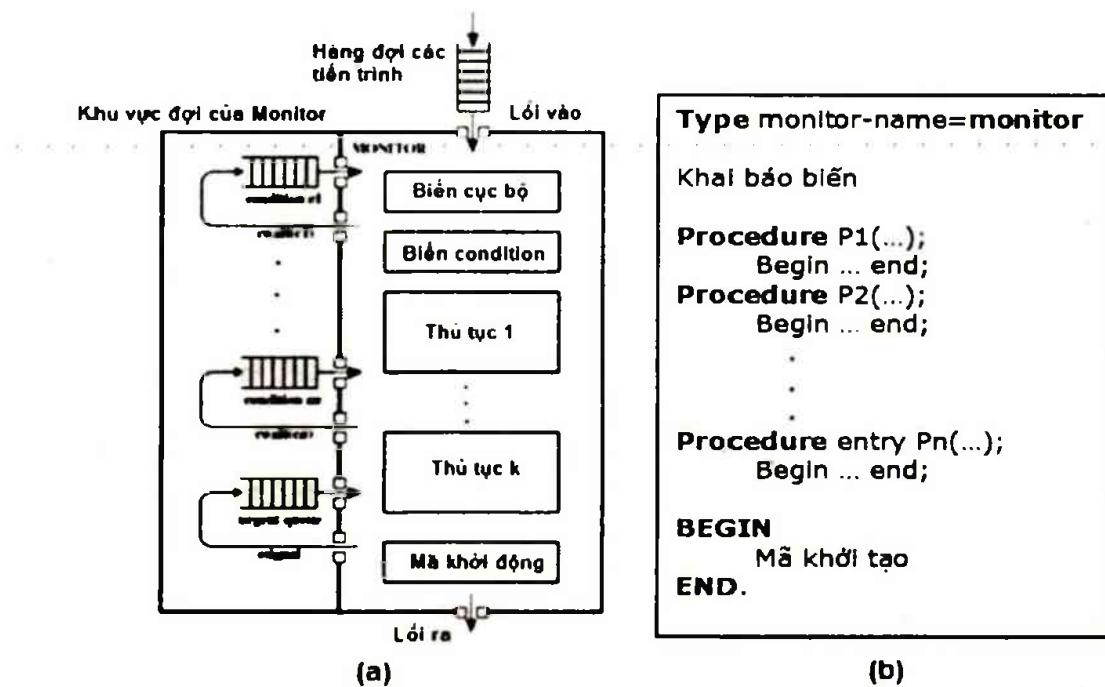
6.6. CÁC GIẢI PHÁP ĐỒNG BỘ CAO CẤP

6.6.1. Monitor và biến điều kiện Condition

Monitor - một cấu trúc đồng bộ mức cao là tập hợp các toán tử được người lập trình định nghĩa trước. Khai báo kiểu monitor gồm hai phần: phần khai báo biến cục bộ và phần thân là các thủ tục hay hàm thao tác trên kiểu đó. Cú pháp của Monitor minh họa trên Hình 6.17b. Chú ý rằng, các thủ tục/hàm định nghĩa bên trong monitor chỉ được truy cập tới những biến được khai báo cục bộ trong monitor và các tham số hình thức. Tương tự biến cục bộ trong monitor chỉ được truy cập qua những thủ tục cục bộ.

Cấu trúc monitor đảm bảo tại một thời điểm chỉ có đúng một tiến trình được thực hiện bên trong monitor. Do vậy, người lập trình không cần viết mã cho sự đồng bộ này một cách tường minh. Tuy nhiên, cấu trúc monitor như đã được định nghĩa chưa đủ mạnh để mô hình hóa một số lược đồ đồng

bộ. Vì vậy, bổ sung thêm vào monitor một cơ chế đồng bộ khác là cấu trúc **condition**. Lập trình viên cần tự viết lược đồ đồng bộ riêng của mình để có thể định nghĩa một hoặc nhiều biến thuộc kiểu **condition**. Chỉ toán tử **wait** và **signal** mới có thể được gọi trên biến kiểu **condition**. Tiến trình gọi **wait** sẽ bị treo cho đến khi tiến trình khác gọi **signal** trên biến điều kiện tương ứng. Toán tử **x.signal** khôi phục đúng một tiến trình bị treo. Nếu không có tiến trình bị treo, toán tử **signal** không có tác dụng, nghĩa là trạng thái của **x** sẽ giữ nguyên như thể **signal** chưa thi hành (Hình 6.17a). Điều này khác với toán tử **signal** của semaphore luôn luôn biến đổi trạng thái của semaphore.



Hình 6.17. Monitor

Giả sử tiến trình P gọi toán tử **x.signal()** khi tiến trình Q treo trong **x** (kiểu condition). Rõ ràng nếu Q được khôi phục lại để tiếp tục thi hành, thì sau **x.signal()**, P phải đợi vì nếu không cả P và Q sẽ cùng hoạt động đồng thời trong monitor. Chú ý cả hai tiến trình về mặt khái niệm có thể tiếp tục thực thi công việc bình thường của mình. Có thể có hai khả năng sau:

- (1) P đợi đến khi Q rời khỏi monitor, hoặc đợi điều kiện khác.
- (2) Q đợi đến khi P rời khỏi monitor, hoặc đợi điều kiện khác.

Có những lý lẽ hợp lý trong việc lựa chọn (1) hoặc (2). Khi P đang thi hành trong monitor, lựa chọn (2) hợp lý hơn. Tuy nhiên, nếu chúng ta cho phép P tiếp tục, đến thời điểm Q được khôi phục thì điều kiện logic mà Q đang đợi có thể không còn đúng nữa.

6.6.2. Bài toán "Bữa ăn tối của các triết gia"

Chúng ta minh họa những khái niệm này bằng cách trình bày giải pháp không có bế tắc cho bài toán Bữa ăn tối của các triết gia. Chú ý rằng, triết gia chỉ được phép nhặt đũa lên nếu chúng chưa được nhặt. Cần phân biệt sự khác nhau trong ba trạng thái của triết gia, ta có cấu trúc dữ liệu **state:array[0...4] of (thinking, hungry, eating)**. Triết gia thứ i có thể đặt giá trị **state[i] = eating** chỉ khi hai người ngồi cạnh không ăn (**state[i + 4 mod 5] ≠ eating and state[i + 1 mod 5] ≠ eating**), ta khai báo **self: array[0...4] of condition**. Triết gia thứ i có thể phải chờ ngay cả khi đói vì chưa có đủ đũa. Việc cấp phát đũa được kiểm soát bởi monitor **dp** – là một thể hiện của kiểu monitor **dining-philosopher** được minh họa trong Hình 6.18. Triết gia trước khi ăn phải gọi **pickup**. Khi đó, tiến trình "triết gia" này bị treo tạm thời. Sau khi ăn xong, triết gia gọi toán tử **putdown** và tiếp tục suy nghĩ. Triết gia thứ i sử dụng các toán tử **putdown** và **pickup** theo đúng trình tự: **dp.pickup(i) ⇒ Ăn ⇒ dp.putdown(i)**. Dễ dàng chứng minh giải pháp này đảm bảo không có hai người ngồi cạnh nhau đồng thời ăn và không xảy ra tình trạng bế tắc. Tuy nhiên, chú ý rằng một triết gia có thể bị chết đói.

```
Type dining-philosopher = monitor
  var state : array[0...4] of (thinking, hungry, eating );
  var self * : array[0...4] of condition;
.....
  procedure entry pickup (i:0... 4) {
    state[i]:= hungry;
    test(i);
    if state[i] ≠ eating then self[i].wait;
  }
  procedure entry putdown (i:0... 4) {
    state[i]:= thinking;
    test (i+4 mod 5);          Các hàm cục bộ
    test (i+1 mod 5);
  }
  procedure test (k:0...4) {
    if (state[k+4 mod 5] ≠ eating)      and
    (state[k]=hungry)                  and
    (state[k+1 mod 5]≠ eating) {
      state[k]:= eating;
      self[k].signal;
    };
  }
.....
BEGIN
  for i:= 0 to 4 state[i]:= thinking;    Mã khởi tạo
END
```

Hình 6.18. Monitor giải quyết bài toán Bữa ăn tối của các triết gia

6.6.3. Cài đặt monitor bằng semaphore

```
Hàm F
wait(mutex);
Thân hàm F
if next-count > 0
    signal (next)
else signal (mutex);
```

```
x.wait() {
    x-count:= x-count + 1;
    if next-count > 0
        signal (next)
    else
        signal (mutex);
    wait (x-sem);
    x-count:= x-count - 1;
}
```

```
x.signal() {
    if x-count > 0 {
        next-count := next-
        count + 1;
        signal (x-sem);
        wait (next );
        next-count := next-
        count -1;
    }
}
```

a)

b)

c)

Hình 6.19. Cài đặt monitor và condition

Mỗi monitor gắn với một semaphore **mutex** (được khởi tạo giá trị 1). Tiến trình phải thực thi **wait(mutex)** trước khi bước vào monitor và thực thi **signal(mutex)** sau khi rời khỏi monitor. Tiến trình gọi **wait** phải đợi đến khi tiến trình đang chiếm giữ semaphore hoặc rời khỏi semaphore, hoặc bị dừng lại tạm thời, nên hệ thống cài đặt thêm semaphore **next** được khởi tạo là 0. Tiến trình đang ở trong semaphore **mutex** có thể bị phong tỏa tại **next**. Biến nguyên **next-count** đếm số tiến trình bị phong tỏa tại **next**. Vì vậy, thủ tục F cài tiến được minh họa trong Hình 6.19a. Như vậy, đảm bảo được độc quyền truy xuất trong monitor.

Với **x** là kiểu condition, ta có semaphore **x-sem** và biến nguyên **x-count**, cả hai được khởi tạo là 0. Toán tử **x.wait** và **x.signal** được minh họa trên Hình 6.19b và c.

Chuyển sang vấn đề thứ tự phục hồi các tiến trình bên trong monitor, nếu có một số tiến trình bị treo trong biến **x** kiểu condition và một tiến trình nào đó thi hành toán tử **x.signal()**, khi đó phải xác định tiến trình bị treo nào sẽ được khôi phục? Một cách giải quyết đơn giản là thuật toán FCFS: tiến trình đợi lâu nhất sẽ được khôi phục trước. Tuy nhiên, thuật toán đơn giản này không phù hợp trong nhiều tình huống. Vì vậy, có thể người ta sử dụng cấu trúc conditional-wait có dạng: **x.wait(c)**; trong đó **c** là một biểu thức kiểu nguyên được tính khi thi hành toán tử **wait**. Giá trị của **c** là giá trị ưu tiên được lưu giữ cùng với tên của tiến trình bị treo. Khi toán tử **x.signal()** thi hành, tiến trình nào có giá trị ưu tiên nhỏ nhất sẽ được khôi phục.

```

R.acquire(t);
Truy cập và Sử
dụng Tài nguyên;
trong t đơn vị
thời gian

R.release;

```

Type resource-allocation = monitor

```

var      busy : boolean;
        x    : condition;

procedure acquire (time: integer)
{
    if busy then x.wait(time );
    busy:= true;
}

procedure release() {
    busy:= false;
    x.signal;
}

BEGIN
    busy:= false;
END.

```

a)

b)

Hình 6.20. Ví dụ cấp phát tài nguyên

Để minh họa cho kỹ thuật này, xét monitor điều khiển việc cấp phát tài nguyên cho các tiến trình cạnh tranh được minh họa trong Hình 6.20b. Khi yêu cầu tài nguyên, tiến trình cần chỉ rõ thời gian dự kiến sử dụng tài nguyên tối đa. Monitor sẽ cấp tài nguyên cho tiến trình có thời gian yêu cầu ngắn nhất. Tiến trình muốn truy cập tới tài nguyên phải thực hiện như trong Hình 6.20a. Ở đây, R thuộc kiểu resource-allocation. Tuy nhiên, monitor không đảm bảo trình tự truy cập trước đó sẽ được bảo tồn. Cụ thể:

- Tiến trình có thể truy cập trái phép tới tài nguyên.
- Tiến trình có thể không bao giờ giải phóng tài nguyên đã được cấp phát.
- Tiến trình có thể thử giải phóng tài nguyên mà mình chưa yêu cầu.
- Tiến trình có thể yêu cầu cùng một tài nguyên liên tiếp hai lần (mà chưa giải phóng tài nguyên trước).

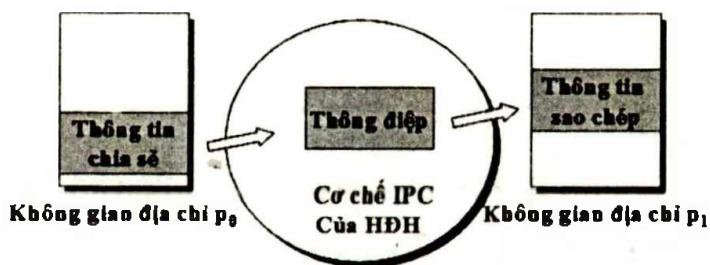
Chú ý, cấu trúc miền găng cũng có những khó khăn tương tự, và về mặt bản chất, những khó khăn này là động lực để phát triển cấu trúc vùng găng và monitor. Trước đây chúng ta phải lo lắng về cách sử dụng semaphore chính xác. Nay giờ chúng ta lo lắng về việc sử dụng chính xác những toán tử ở mức cao (mà trình biên dịch không thể giúp đỡ lập trình viên được nữa).

Một giải pháp khả dĩ cho những vấn đề trên là đặt các toán tử truy cập tài nguyên bên trong monitor *resource-allocation*. Thế nhưng, điều này dẫn đến việc điều phối sẽ tuân theo thuật toán điều phối của monitor chứ không

phai thuật toán mà chúng ta mong muốn. Để đảm bảo tất cả tiến trình thực hiện theo đúng trình tự chính xác, phải giám sát tất cả các chương trình có sử dụng monitor *resource-allocation* và các tài nguyên của nó. Có hai điều phải kiểm tra để đảm bảo tính đúng đắn của hệ thống. Thứ nhất, tiến trình người dùng phải sử dụng các toán tử của monitor theo đúng thứ tự. Thứ hai, phải chắc chắn một tiến trình "bất hợp tác" không thể vi phạm độc quyền truy xuất của monitor, tức là có thể truy cập trực tiếp tới tài nguyên dùng chung mà không sử dụng quy tắc truy cập.

6.7. CƠ CHẾ IPC

Những cơ chế đồng bộ trên đòi hỏi các tiến trình có biến dùng chung. Trong phần này, trình bày cơ chế truyền thông liên tiến trình cho phép các tiến trình trao đổi dữ liệu và đồng bộ hoạt động thông qua việc chuyển thông điệp. Thông điệp (message) là một khối thông tin theo khuôn dạng nào đó (đã được thống nhất trước) được trao đổi giữa hai tiến trình. Cơ chế bảo vệ bộ nhớ trong các HĐH hiện đại không cho các tiến trình xâm nhập dù vô tình hay cố ý vào không gian nhớ của tiến trình khác. Để thực hiện trao đổi thông tin, HĐH đóng vai trò trung chuyển thông điệp. IPC là cơ chế cho phép tiến trình sao chép nội dung thông tin muốn gửi vào thông điệp và sau đó HĐH sao chép nội dung thông điệp vào không gian địa chỉ tiến trình nhận (Hình 6.21).

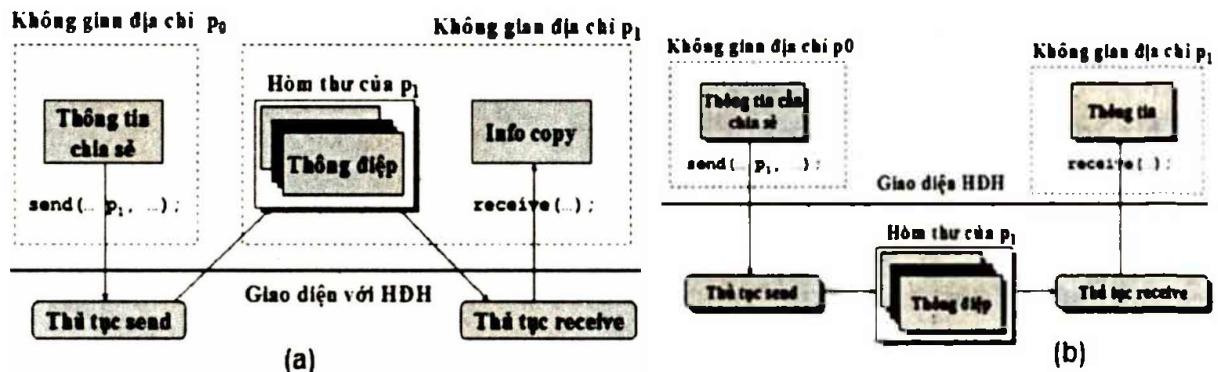


Hình 6.21. Cơ chế truyền thông liên tiến trình

6.7.1. Hộp thư

Hình 6.22 minh họa việc thông tin được sao chép trực tiếp sang không gian địa chỉ của tiến trình nhận, mà phía nhận có thể không biết sự kiện này xảy ra. Có thể tránh điều này bằng cách không chuyển thông tin trừ khi phía nhận đưa ra yêu cầu nhận tin tức minh. HĐH có thể giữ tạm thông điệp ở hộp thư trước khi sao chép thông tin sang không gian địa chỉ của tiến trình

nhận. Hộp thư có thể để ở không gian HDH (Hình 6.22b) hoặc không gian địa chỉ tiến trình nhận (Hình 6.22a). Nếu đặt ở không gian người nhận, thủ tục nhận có thể cài đặt bằng thư viện người dùng và trình biên dịch và bộ tải phải cấp phát bộ nhớ làm hộp thư. Tiến trình người dùng khi chạy có thể vô tình phá hỏng hộp thư. Nếu hộp thư đặt trong không gian HDH, phía nhận sẽ chỉ nhận được thông điệp khi tường minh gọi thủ tục nhận. Hộp thư sẽ không bị tiến trình người dùng thay đổi (do đặt trong không gian HDH), nhưng điều này lại khiến HDH phải tốn bộ nhớ làm hộp thư.



Hình 6.22. Hộp thư

6.7.2. Chuẩn thông điệp

Để hai bên có thể hiểu nhau, thông điệp phải tuân theo một quy tắc nào đấy. Thông thường, thông điệp có tiêu đề chứa các thông tin liên quan đến thông điệp (chẳng hạn định danh tiến trình gửi, nhận, kích thước thông điệp,...).

6.7.3. Thao tác gửi và nhận

Dung lượng của hộp thư là số lượng thông điệp tối đa hộp thư có thể chứa. Thuộc tính này có thể nhận 3 giá trị:

- **Dung lượng 0:** Hộp thư không thể có thông điệp đợi nhận. Tiến trình gửi phải đợi đến khi tiến trình phía bên kia nhận được thông điệp. Hai tiến trình phải đồng bộ hóa khi truyền thông điệp.
- **Dung lượng hữu hạn:** Hộp thư kích thước n có thể chứa tối đa n thông điệp. Nếu hộp thư chưa đầy, thông điệp mới gửi được đặt trong hộp thư (có thể chính thông điệp hoặc địa chỉ thông điệp được đặt vào hộp thư). Sau đó tiến trình gửi có thể tiếp tục thực hiện công

việc của mình. Khi hộp thư đầy, tiến trình gửi phải đợi cho đến khi hộp thư có chỗ trống.

- **Dung lượng vô hạn:** Mọi thông điệp có thể được đặt trong hộp thư. Tiến trình gửi không bao giờ phải ngừng.

Trường hợp dung lượng 0 được gọi là chuyển thông điệp không có bộ đệm; còn hai trường hợp kia là bộ đệm tự động. Trong trường hợp dung lượng khác 0, sau khi gửi thành công thông điệp, tiến trình không xác định được khi nào thông điệp đến đích. Nếu thông tin này quan trọng trong quá trình tính toán, tiến trình gửi phải hỏi tiến trình nhận một cách tường minh để biết khi nào thông điệp đến đích. Ví dụ, tiến trình P gửi thông điệp đến Q và có thể tiếp tục thực thi chỉ khi thông điệp đến đích. Tiến trình P thực hiện hai lệnh sau: **Send(Q, mess)** và **Receive(Q, mess)**, còn tiến trình Q thực thi hai lệnh: **Receive(P, mess)** và **Send(P, ack)**. Kiểu giao tiếp này được gọi là giao tiếp đồng bộ.

Có một trường hợp đặc biệt không nằm hoàn toàn trong ba giải pháp nêu trên:

- Sau khi gửi thông điệp, tiến trình vẫn tiếp tục thực thi. Tuy nhiên, tiến trình nhận chưa nhận thông điệp mà tiến trình gửi lại tiếp tục gửi tiếp thông điệp khác, thì thông điệp đầu tiên bị mất. Ưu điểm của phương pháp này là thông điệp lớn không bị lặp nhiều lần. Nhược điểm là làm phức tạp công việc lập trình. Các tiến trình cần đồng bộ hóa một cách tường minh để đảm bảo thông điệp đến đích và cả bên gửi lẫn bên nhận không đồng thời thao tác trên bộ đệm.
- Tiến trình sau khi gửi thông điệp sẽ đợi cho đến khi phía bên kia trả lời. Phương pháp này được sử dụng trong HĐH Thoth. Thông điệp có kích thước cố định (8 từ). Tiến trình P sau khi gửi thông điệp sẽ bị phong tỏa cho đến khi phía nhận nhận được thông điệp và gửi thông điệp trả lời **reply(P, mess)**. Thông điệp trả lời sẽ nằm đè lên vùng đệm của thông điệp gốc. Điểm khác biệt duy nhất giữa thông điệp gửi và trả lời là thông điệp gửi làm tiến trình gửi bị phong tỏa, còn thông điệp trả lời làm cả hai tiến trình tiếp tục hoạt động.

Phương thức giao tiếp đồng bộ này có thể dễ dàng mở rộng thành hệ thống gọi thủ tục từ xa (Remote Procedure Call - RPC) với đầy đủ chức

năng. Mọi hệ thống RPC đều dựa trên một ý tưởng là, trong một hệ thống có một CPU, lời gọi thủ tục hay chương trình con giống như việc chuyển thông điệp, tiến trình gửi sẽ bị treo cho đến khi tiến trình nhận gửi thông điệp phản hồi biên nhận. Sau này có thể thấy RPC có thể mở rộng cho phép các tiến trình trên các máy tính khác nhau có thể phối hợp làm việc với nhau.

6.7.4. Xử lý biệt lệ (Exception Conditions)

Ưu điểm của hệ thống chuyển thông điệp nổi bật trong môi trường phân tán vì tiến trình có thể nằm trên các máy tính khác nhau. Tuy nhiên, khả năng xuất hiện lỗi trong môi trường này rất lớn, do đó cần một cơ chế khắc phục (xử lý biệt lệ).

☞ *Tiến trình bị kết thúc*

Tiến trình gửi hoặc nhận có thể bị phong tỏa trước khi xử lý xong thông điệp. Tình huống này có thể khiến thông điệp không bao giờ nhận được hoặc tiến trình phải đợi một thông điệp sẽ không bao giờ đến. Xét 2 trường hợp sau:

1. Tiến trình P đợi thông điệp từ tiến trình Q đã kết thúc. Nếu không có cơ chế giải quyết, P sẽ bị phong tỏa vĩnh viễn. Trong trường hợp này, hệ thống có thể hoặc cho P kết thúc hoặc báo cho P biết Q đã kết thúc.
2. Tiến trình P có thể gửi một thông điệp cho tiến trình Q đã bị ngắt. Không có vấn đề gì xảy ra trong phương pháp bộ đệm tự động, P tiếp tục hoạt động. Nếu P cần xác định thông điệp của mình đã được Q xử lý chưa, P phải lập trình tường minh để đợi xác nhận từ Q. Trong trường hợp không có bộ đệm, P sẽ bị phong tỏa hoàn toàn. Giống như trong trường hợp 1, hệ thống có thể hoặc cho P kết thúc, hoặc báo cho P biết Q đã kết thúc.

☞ *Mất thông điệp*

Thông điệp P gửi cho Q có thể bị mất (do lỗi phần cứng hay lỗi đường truyền). Có ba phương thức cơ bản khắc phục điều này:

1. HĐH chịu trách nhiệm phát hiện và sau đó gửi lại thông điệp bị mất.
2. Tiến trình gửi có trách nhiệm phát hiện và truyền lại thông điệp nếu thấy cần thiết.

3. HĐH chịu trách nhiệm phát hiện mất thông điệp. Sau đó HĐH báo cho tiến trình gửi thông điệp bị mất. Tiến trình gửi tự động đưa ra phương thức giải quyết.

Không phải việc phát hiện lỗi lúc nào cũng quan trọng. Trên thực tế, bên cạnh các giao thức mạng không đảm bảo chuyển thông điệp tin cậy thì vẫn có giao thức đảm bảo tính tin cậy khi chuyển (TCP). Chính người sử dụng phải chỉ rõ (nghĩa là hoắc báo với hệ thống, hoặc tự lập trình) xem có cần đảm bảo tin cậy trong quá trình truyền không.

Làm thế nào để phát hiện mất thông điệp? Phương pháp phổ biến nhất là sử dụng cơ chế timeout (hết thời gian). Sau khi gửi thông điệp đi, tiến trình gửi sẽ đợi thông điệp biên nhận từ phía bên kia trong một khoảng thời gian định trước. Nếu hết khoảng thời gian này mà chưa nhận được phản hồi, HĐH (hoặc tiến trình) xem như thông điệp đã mất và gửi lại thông điệp. Có thể thông điệp không bị mất, mà chỉ đến muộn. Khi đó, sẽ có hai bản sao của cùng một thông điệp truyền qua mạng. Có thể sử dụng kỹ thuật đánh số thứ tự để phân biệt các thông điệp.

Thông điệp bị lỗi

Thông điệp có thể đến đích, nhưng bị lỗi trên đường truyền (ví dụ nguyên nhân do nhiễu xuất hiện trên kênh truyền). Trường hợp này cũng giống như trường hợp mất thông điệp. Thường HĐH phải gửi lại thông điệp gốc. Các kỹ thuật như tổng kiểm tra (checksum) hay bit chẵn lẻ (parity bit) được sử dụng để phát hiện lỗi trong thông điệp.

6.8. NHẬN XÉT

Khi nhiều tiến trình tuần tự kết hợp với nhau để chia sẻ dữ liệu, HĐH phải đảm bảo độc quyền truy xuất. Một cách giải quyết là đảm bảo tại một thời điểm chỉ có đúng một tiến trình (hay thread) thực thi đoạn mã trong miền găng. Nhiều thuật toán được đưa ra để giải quyết vấn đề miền găng. Nhược điểm chính trong việc sử dụng đoạn mã ở mức người dùng là tình trạng busy waiting. Semaphore khắc phục được nhược điểm này. Semaphore có thể được sử dụng để giải quyết rất nhiều vấn đề đồng bộ hóa và có thể được cài đặt hiệu quả nếu phần cứng hỗ trợ các thao tác đơn nhất. Vùng găng có thể được sử dụng để giải quyết vấn đề độc quyền truy xuất

cũng như các vấn đề đồng bộ một cách an toàn và hiệu quả. Monitor cung cấp kỹ thuật đồng bộ để chia sẻ những loại dữ liệu trừu tượng. Biến kiểu condition cung cấp phương pháp cho phép thủ tục tự phong tỏa chính nó cho đến khi nó được phép khôi phục.

CÂU HỎI ÔN TẬP

1. Tại sao phải chia sẻ thông tin và những vấn đề gặp phải khi chia sẻ thông tin?
2. Trình bày các giải pháp độc quyền truy xuất.
3. Trình bày các giải pháp đồng bộ cơ bản.

Chương 7

BẾ TẮC

Trong hệ thống, nhiều tiến trình cạnh tranh nhau quyền sử dụng lượng hữu hạn tài nguyên. Khi yêu cầu tài nguyên mà chưa được đáp ứng, tiến trình bị phong tỏa. Có thể tiến trình sẽ ở mãi trong trạng thái phong tỏa nếu tài nguyên yêu cầu bị các tiến trình khác cũng ở trạng thái phong tỏa chiếm giữ. Tình huống này được gọi là bế tắc và đã được nói qua ở phần semaphore trong Chương 6. Chương này trình bày các phương thức giải quyết bế tắc của HĐH.

7.1. MÔ HÌNH HỆ THỐNG

Trong quá trình hoạt động, các tiến trình cạnh tranh nhau quyền sử dụng tài nguyên hệ thống. Có nhiều kiểu tài nguyên, trong mỗi kiểu có thể có nhiều đối tượng giống nhau. Ví dụ, không gian bộ nhớ, CPU, file và thiết bị vào/ra (như máy in và đĩa từ) là các kiểu tài nguyên. Nếu hệ thống có hai CPU, thì kiểu tài nguyên CPU sẽ có hai đối tượng. Khi tiến trình yêu cầu một kiểu tài nguyên, hệ thống có thể cấp phát bất kỳ đối tượng nào của kiểu tài nguyên cho tiến trình. Tiến trình có thể sử dụng nhiều tài nguyên để hoàn thành công việc của mình, tuy nhiên, lượng tài nguyên tiến trình yêu cầu không thể vượt quá tổng lượng tài nguyên của hệ thống. Tiến trình sử dụng tài nguyên theo trình tự:

- Yêu cầu (Request):** Nếu không được hệ thống đáp ứng ngay thì tiến trình phải đợi cho đến có.
- Sử dụng (Use):** Tiến trình sử dụng tài nguyên.
- Giải phóng (Release):** Tiến trình trả tài nguyên cho hệ thống.

HĐH thường cung cấp cho chương trình người dùng các lời gọi hệ thống thực hiện yêu cầu và giải phóng thiết bị; yêu cầu mở và đóng file; yêu cầu cấp phát và giải phóng bộ nhớ. Yêu cầu và giải phóng các tài nguyên khác cũng có thể được thực hiện thông qua các thao tác **wait** và **signal** trên semaphore. HĐH có bảng tài nguyên hệ thống, qua đó xác định được trạng thái cấp phát tài nguyên (đã cấp phát hay chưa cấp phát, cấp phát cho ai,...). Nếu yêu cầu tài nguyên mà không được đáp ứng, tiến trình bị phong tỏa tại hàng đợi tài nguyên.

Tập hợp tiến trình ở trong trạng thái bế tắc nếu mỗi tiến trình đợi một sự kiện gây ra bởi tiến trình khác nằm trong cùng tập hợp. Ở đây quan tâm chủ yếu đến sự kiện chiếm giữ và giải phóng tài nguyên.

7.2. ĐẶC ĐIỂM CỦA BẾ TẮC

Bế tắc là tình trạng không mong muốn, vì khi xảy ra không tiến trình nào có thể kết thúc, tài nguyên hệ thống bị chiếm giữ. Trước khi trình bày các giải pháp xử lý, cần tìm hiểu một số đặc điểm của hiện tượng này.

7.2.1. Điều kiện cần

Bế tắc sẽ xảy ra nếu bốn điều kiện sau đồng thời xuất hiện:

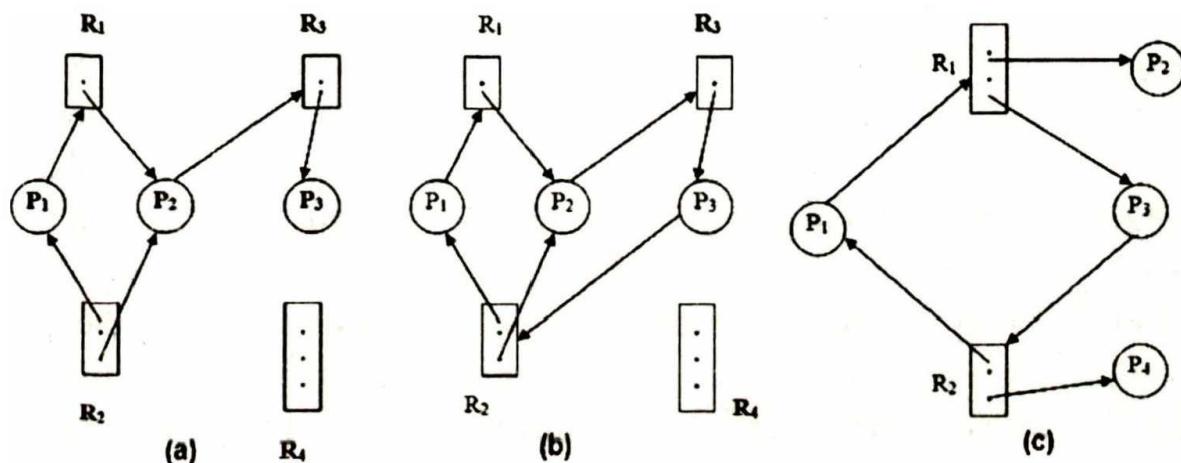
- Độc quyền truy xuất (Mutual Exclusion):** Ít nhất một tài nguyên bị nắm giữ thuộc kiểu không thể dùng chung. Nghĩa là, tại một thời điểm chỉ có tối đa một tiến trình được quyền sử dụng tài nguyên.
- Giữ và chờ (Hold and wait):** Phải có tiến trình đang nắm giữ tài nguyên đồng thời lại chờ tài nguyên bị tiến trình khác chiếm giữ.
- Không chiếm đoạt (No preemption):** Hệ thống không thể tước đoạt tài nguyên của tiến trình, trừ khi tiến trình chủ động giải phóng tài nguyên khi không cần thiết.
- Vòng đợi (Circular wait):** Tồn tại tập hợp các tiến trình $\{P_0, P_1, \dots, P_n\}$ đang trong trạng thái chờ tài nguyên. Trong đó P_0 chờ tài nguyên bị P_1 nắm giữ, P_1 chờ tài nguyên bị P_2 nắm giữ,..., P_{n-1} chờ tài nguyên bị P_n nắm giữ và P_n chờ tài nguyên bị P_0 nắm giữ.

Cả bốn điều kiện trên phải đồng thời xuất hiện thì hiện tượng bế tắc mới xảy ra. Điều kiện Vòng đợi kéo theo điều kiện Giữ và Chờ, do đó bốn điều

kiện này không hẳn hoàn toàn độc lập với nhau. Tuy nhiên, sẽ thuận tiện hơn nếu xét tách biệt các điều kiện này.

7.2.2. Đồ thị phân phối tài nguyên

Đồ thị có tập đỉnh V và tập cung E . Tập đỉnh V được phân thành hai loại: $P = \{P_1, P_2, \dots, P_n\}$ ứng với các tiến trình và $R = \{R_1, R_2, \dots, R_m\}$ ứng với kiểu tài nguyên của hệ thống. *Cung yêu cầu* $P_i \rightarrow R_j$ hướng từ tiến trình P_i tới tài nguyên R_j , có ý nghĩa P_i đã yêu cầu một đối tượng của kiểu tài nguyên R_j và hiện thời P_i đang trong trạng thái đợi. *Cung phân phối* $R_j \rightarrow P_i$ hướng từ tài nguyên R_j tới tiến trình P_i , có nghĩa là một đối tượng của kiểu tài nguyên R_j đã được cấp phát cho tiến trình P_i . Trên Hình 7.1a, tiến trình P_i được biểu diễn là cung tròn, tài nguyên R_j là hình vuông. Do kiểu tài nguyên R_j có thể có nhiều đối tượng, chúng ta biểu diễn mỗi đối tượng là một chấm nhỏ bên trong hình vuông. Lưu ý, cung yêu cầu chỉ nối tới hình vuông R_j , trong khi đó cung phân phối phải được chỉ rõ xuất phát từ chấm nào bên trong hình vuông.



Hình 7.1. Đồ thị phân phối tài nguyên

Khi tiến trình P_i yêu cầu đối tượng tài nguyên R_j , hệ thống bổ sung một cung yêu cầu vào đồ thị phân phối tài nguyên. Khi có thể đáp ứng yêu cầu, hệ thống ngay lập tức chuyển cung yêu cầu thành cung phân phối. Cung phân phối bị xóa bỏ khi tiến trình giải phóng tài nguyên.

Đồ thị phân phối tài nguyên trên Hình 7.1a mô tả tình huống sau:

$$P = \{P_1, P_2, P_3\}, R = \{R_1, R_2, R_3, R_4\},$$

$$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow R_1, R_3 \rightarrow P_3\}.$$

Số đối tượng của mỗi tài nguyên: R_1 có 1 đối tượng, R_2 có 2 đối tượng, R_3 có 1 đối tượng, R_4 có 3 đối tượng.

Trạng thái các tiến trình: P_1 đang giữ một đối tượng tài nguyên R_2 và đang đợi một đối tượng R_1 . P_2 đang giữ một đối tượng R_1 và đang đợi thêm một đối tượng R_3 . P_3 đang giữ một đối tượng R_3 .

Với đồ thị phân phối tài nguyên, có thể thấy, nếu không có chu trình trên đồ thị thì không tiến trình nào bị bế tắc. Tuy nhiên, nếu đồ thị có chứa chu trình, bế tắc chưa chắc, mà chỉ có khả năng xuất hiện. Nếu mỗi kiểu tài nguyên có đúng một đối tượng, thì chu trình dẫn đến bế tắc và mỗi tiến trình trong chu trình đều rơi vào trạng thái bế tắc. Khi đó, chu trình trong đồ thị là điều kiện cần và đủ cho trạng thái bế tắc. Nhưng, nếu kiểu tài nguyên có nhiều đối tượng, chu trình không nhất thiết kéo theo bế tắc (chu trình là điều kiện cần nhưng chưa đủ).

Xét đồ thị phân phối tài nguyên trên Hình 7.1b, giả sử P_3 yêu cầu một đối tượng R_2 . Do hệ thống không còn đối tượng tài nguyên R_2 rỗng nên cung cấp $P_3 \rightarrow R_2$ được thêm vào đồ thị. Lúc này, xuất hiện hai chu trình: $(P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1)$ và $(P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2)$. Ba tiến trình P_1 , P_2 và P_3 bế tắc: P_2 đợi R_3 – bị chiếm dụng bởi P_3 . P_3 đợi P_1 và P_2 giải phóng R_2 , còn P_1 đợi P_2 giải phóng R_1 . Đồ thị phân phối tài nguyên trên Hình 7.1c có chu trình $(P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1)$. Tuy nhiên, không có bế tắc vì P_4 có thể giải phóng một đối tượng R_2 . Tài nguyên này sau đó có thể được cấp phát cho P_3 (chu trình bị phá vỡ). Tóm lại, nếu đồ thị phân phối tài nguyên không có chu trình, thì hệ thống chắc chắn không ở trong trạng thái bế tắc. Ngược lại, nếu có chu trình, hệ thống có thể rơi vào trạng thái bế tắc. Chúng ta cần chú ý tới điểm này khi giải quyết hiện tượng bế tắc.

7.2.3. Giải quyết bế tắc

Có ba giải pháp cho hiện tượng bế tắc như sau:

- Đảm bảo hệ thống không rơi vào trạng thái bế tắc.
- Cho phép hệ thống rơi vào trạng thái bế tắc rồi sau đó khắc phục.
- Bỏ qua tất cả các vấn đề và giả định bế tắc không bao giờ xuất hiện trong hệ thống. Phương pháp này được sử dụng trong nhiều HĐH, kể cả UNIX.

Phần này chỉ mô tả ngắn gọn các giải pháp, chi tiết được trình bày trong phần sau. Để đảm bảo không xảy ra bế tắc, hệ thống có thể sử dụng phương pháp Ngăn chặn hoặc phương pháp Tránh. Ngăn chặn là phương pháp ngăn không cho ít nhất một trong bốn điều kiện cần của bế tắc xuất hiện. Ngược lại, phương pháp tránh đòi hỏi tiến trình phải báo trước cho HĐH các thông tin liên quan đến nhu cầu tài nguyên của mình. Với những thông tin biết trước này, khi tiến trình xin cấp phát tài nguyên, hệ thống có thể quyết định liệu có nên cấp phát cho tiến trình không. Với mỗi yêu cầu, hệ thống xét lượng tài nguyên chưa cấp phát, lượng tài nguyên đã cấp phát, các yêu cầu cấp phát và giải phóng tài nguyên sắp xuất hiện để quyết định xem có thỏa mãn yêu cầu hay không.

Nếu hệ thống không sử dụng phương pháp nào trong hai phương pháp trên, trạng thái bế tắc có khả năng xuất hiện. Khi đó hệ thống cần xác định bế tắc xuất hiện chưa và sau đó thì khắc phục như thế nào.

Nếu hệ thống không đảm bảo bế tắc không xuất hiện và cũng không có cơ chế kiểm tra và khắc phục bế tắc, hệ thống có thể rơi vào trạng thái bế tắc mà không nhận ra. Lúc đó, hiệu suất tổng thể hệ thống bị suy giảm vì nhiều tài nguyên bị các tiến trình trong trạng thái phong tỏa chiếm giữ, ngày càng nhiều tiến trình rơi vào bế tắc khi yêu cầu thêm tài nguyên. Cuối cùng, hệ thống ngừng hoạt động và phải khởi động lại một cách thủ công. Mặc dù giải pháp này có vẻ không ổn, nhưng trên thực tế lại được sử dụng rộng rãi nhất. Nếu bế tắc xuất hiện không thường xuyên (một lần/năm) việc cài đặt các giải pháp Ngăn chặn, Tránh hay Phát hiện và Khắc phục bế tắc quá tốn kém.

7.3. NGĂN CHẶN BẾ TẮC

Bế tắc có bốn đặc điểm được xem là điều kiện cần. Như vậy, chỉ cần ngăn không cho một trong bốn điều kiện này xuất hiện là có thể ngăn chặn bế tắc.

7.3.1. Độc quyền truy xuất (Mutual Exclusion)

Điều kiện độc quyền truy xuất luôn đúng với những tài nguyên không thể chia sẻ (ví dụ máy in). Ngược lại, kiểu tài nguyên có thể chia sẻ (ví dụ file với thuộc tính chỉ đọc) không đòi hỏi độc quyền truy xuất, do đó không

liên quan tới hiện tượng bế tắc. Do bản chất tài nguyên, nên không thể ngăn chặn bế tắc bằng cách loại bỏ điều kiện độc quyền truy xuất.

7.3.2. Giữ và chờ

Để ngăn chặn điều kiện Giữ và Chờ, phải đảm bảo chỉ khi không nắm giữ tài nguyên thì tiến trình mới được yêu cầu tài nguyên. Giải pháp thứ nhất là trước khi thực thi, tiến trình yêu cầu tất cả các tài nguyên cần thiết. Giải pháp thứ hai là, trước khi yêu cầu thêm tài nguyên, tiến trình phải giải phóng tất cả các tài nguyên mình đang chiếm giữ. Để minh họa sự khác biệt giữa hai giải pháp này, xét tiến trình sao lưu dữ liệu từ băng từ vào ổ cứng, sắp xếp các file, sau đó in kết quả ra máy in. Nếu tiến trình yêu cầu tất cả tài nguyên ngay từ đầu thì phải yêu cầu cả ba loại tài nguyên (băng từ, ổ cứng và máy in). Mặc dù chỉ sử dụng vào giai đoạn cuối, nhưng tiến trình sẽ giữ máy in trong suốt quá trình thực thi. Phương pháp thứ hai cho phép tiến trình chỉ yêu cầu băng từ và ổ đĩa lúc đầu. Tiến trình chuyển dữ liệu từ băng từ vào đĩa, rồi giải phóng cả hai tài nguyên này. Tiến trình tiếp tục yêu cầu ổ đĩa và máy in. Sau khi chuyển dữ liệu từ ổ đĩa ra máy in, tiến trình giải phóng tất cả các tài nguyên rồi kết thúc. Các giải pháp này có hai nhược điểm chính. Thứ nhất, hiệu quả sử dụng tài nguyên thấp vì nhiều loại tài nguyên bị chiếm giữ ngay từ đầu nhưng chưa chắc được sử dụng ngay. Thứ hai, có thể xảy ra hiện tượng chết đói. Tiến trình có thể phải đợi vô hạn vì có thể các tài nguyên cần thiết đã bị cấp phát cho tiến trình khác.

7.3.3. Không chiếm đoạt

Chúng ta có thể ngăn chặn bằng cách: Khi tiến trình đang giữ một số tài nguyên và yêu cầu thêm tài nguyên đã bị chiếm giữ, thì trước khi tiến trình bị phong tỏa, hệ thống sẽ thu hồi toàn bộ tài nguyên đã cấp phát cho tiến trình. Tài nguyên sau khi thu hồi được đưa vào danh sách tài nguyên chưa cấp phát. Tiến trình sẽ chỉ khởi động lại nếu được cấp phát lại tài nguyên cũ và tài nguyên mới yêu cầu. Một giải pháp linh hoạt hơn là, khi tiến trình yêu cầu tài nguyên, hệ thống kiểm tra xem có đáp ứng được không. Nếu có, hệ thống cấp phát cho tiến trình. Nếu không, hệ thống kiểm tra xem những tài nguyên này có bị chiếm giữ bởi các tiến trình đang trong trạng thái phong tỏa hay không. Nếu có, hệ thống thu hồi tài nguyên của tiến trình đang bị phong tỏa để cấp phát cho tiến trình yêu cầu. Ngược lại, tiến trình yêu cầu

sẽ phải đợi. Trong khi bị phong tỏa, một số tài nguyên của tiến trình có thể bị thu hồi cho các tiến trình khác. Tiến trình chỉ có thể khởi động lại khi được cấp phát thêm tài nguyên mới và lấy lại các tài nguyên đã bị thu hồi khi còn ở trạng thái phong tỏa. Phương pháp này thường áp dụng cho những tài nguyên mà trạng thái có thể lưu lại và khởi tạo dễ dàng (thanh ghi hay bộ nhớ trong), nhưng không thể áp dụng cho các loại tài nguyên như máy in hay băng từ.

7.3.4. Vòng đợi

Có thể ngăn chặn điều kiện "vòng đợi" bằng cách đánh số thứ tự cho tất cả các tài nguyên, và tiến trình phải yêu cầu tài nguyên theo thứ tự tăng dần. Giả sử $R = \{R_1, R_2, \dots, R_m\}$ là tập tài nguyên, ta gán cho mỗi tài nguyên một số thứ tự (STT) nguyên duy nhất. Cách đánh STT phải đảm bảo có thể so sánh được STT của hai tài nguyên, nhằm xác định tài nguyên nào đứng trước, tài nguyên nào đứng sau. Nghĩa là về mặt hình thức, ta định nghĩa hàm $F: R \rightarrow N$, trong đó N là tập các số nguyên. Ví dụ, nếu R bao gồm băng từ, ổ đĩa và máy in thì hàm F có thể định nghĩa như sau: $F(\text{băng từ}) = 1$, $F(\text{ổ đĩa}) = 5$, $F(\text{máy in}) = 12$. Xét phương pháp ngăn chặn "bé tắc" sau đây: Tiến trình phải yêu cầu tài nguyên theo STT tăng dần. Nghĩa là, lúc đầu tiến trình có thể yêu cầu bất cứ tài nguyên R_i nào. Sau đó, nếu yêu cầu thêm tài nguyên R_j thì $F(R_j) > F(R_i)$. Nếu cần nhiều đối tượng trong cùng một kiểu tài nguyên, tiến trình phải yêu cầu trong một yêu cầu duy nhất. Ví dụ, với hàm F trên, nếu muốn sử dụng băng từ và máy in tại cùng thời điểm, thì tiến trình phải yêu cầu băng từ trước khi yêu cầu máy in. Giải pháp tương đương là khi yêu cầu R_j , tiến trình phải giải phóng tất cả các tài nguyên R_i mà $F(R_i) > F(R_j)$. Nếu sử dụng hai quy tắc này, thì vòng đợi không bao giờ xảy ra (có thể chứng minh bằng phản chứng).

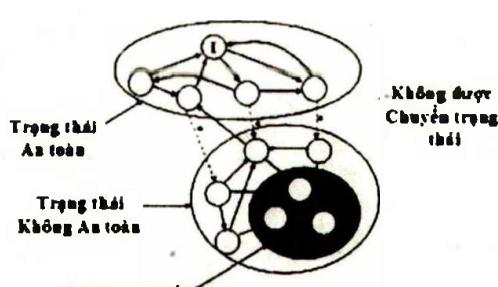
7.4. TRÁNH BÉ TẮC

Các thuật toán ngăn chặn bé tắc đặt ra quy tắc mà tiến trình phải tuân theo khi yêu cầu tài nguyên nhằm ngăn cản một trong bốn điều kiện bé tắc xuất hiện. Nhược điểm của giải pháp này là giảm hiệu suất sử dụng tài nguyên cũng như hiệu suất tổng thể hệ thống. Một thuật toán tránh bé tắc khác là, yêu cầu tiến trình cung cấp thêm thông tin về việc sử dụng tài

nguyên. Ví dụ, trong hệ thống có một băng từ và một máy in, hệ thống có thể biết tiến trình P yêu cầu băng từ trước yêu cầu máy in. Ngược lại, tiến trình Q yêu cầu máy in trước băng từ. Với những hiểu biết khi nào tiến trình sẽ yêu cầu và giải phóng tài nguyên, hệ thống có thể quyết định việc cấp phát tài nguyên. Khi có yêu cầu, hệ thống xét trạng thái cấp phát tài nguyên (các tài nguyên hiện có; các tài nguyên đã cấp phát; các yêu cầu xin và giải phóng tài nguyên trong tương lai) để quyết định xem có cấp phát tài nguyên cho tiến trình hay không. Các thuật toán khác nhau có thể đòi hỏi biết trước các kiểu thông tin khác nhau. Thông thường, tiến trình khai báo số lượng cực đại các kiểu tài nguyên mà nó cần.

7.4.1. Trạng thái an toàn

Hệ thống ở trong trạng thái an toàn nếu có thể cấp phát đầy đủ tài nguyên cho mỗi tiến trình mà không bị "bế tắc". Định nghĩa một cách hình thức, hệ thống ở trong trạng thái an toàn chỉ khi tồn tại một dãy an toàn. Dãy tiến trình $\langle P_1, P_2, \dots, P_n \rangle$ được gọi là an toàn trong trạng thái phân phối tài nguyên hiện thời, nếu yêu cầu tài nguyên của tiến trình P_i có thể được đáp ứng bằng các tài nguyên chưa cấp phát cùng các tài nguyên đang bị giữ bởi tất cả các P_j , với $j < i$. Khi đó, nếu tài nguyên tiến trình P_i đang cần không được đáp ứng ngay, P_i có thể đợi tới khi tất cả các P_j kết thúc. Sau đó, P_i có thể được cấp phát đủ tài nguyên cần thiết để thực hiện nhiệm vụ của mình. Khi P_i kết thúc, P_{i+1} có thể lấy được các tài nguyên cần thiết,... Nếu không tồn tại dãy như vậy, hệ thống được xem là không an toàn.



(a)

Tiến trình	Lượng cần tối đa	Lượng cần hiện thời
P_0	10	5
P_1	4	2
P_2	9	2

(b)

Hình 7.2. Minh họa trạng thái bế tắc

Trạng thái an toàn không phải là trạng thái bế tắc. Ngược lại, trạng thái bế tắc là một trạng thái không an toàn. Tuy nhiên, không phải trạng thái

không an toàn nào cũng là trạng thái bế tắc (Hình 7.2a). Trạng thái không an toàn có thể dẫn đến bế tắc. Chừng nào hệ thống còn ở trạng thái an toàn, HĐH có thể ngăn chặn bế tắc. Trong trạng thái không an toàn, HĐH không thể ngăn chặn các tiến trình yêu cầu tài nguyên và có thể dẫn đến bế tắc.

Dể minh họa, xét hệ thống có 12 băng từ với 3 tiến trình: P_0 , P_1 , P_2 , trong đó P_0 cần tối đa 10 băng từ, P_1 cần 4 băng từ và P_2 cần 9 băng từ. Giả sử tại thời điểm t_0 , P_0 đang giữ 5, P_1 giữ 2 và P_2 giữ 2 băng từ (3 băng từ chưa cấp phát). Ví dụ này minh họa trên Hình 7.2b. Tại thời điểm t_0 , hệ thống trong trạng thái an toàn vì dãy $\langle P_1, P_0, P_2 \rangle$ thỏa mãn điều kiện an toàn. P_1 có thể lấy 2 băng từ, thực thi, rồi giải phóng (hệ thống có 5 băng từ rỗi); kế đó P_0 lấy thêm 5 băng từ, thực thi và giải phóng (lúc này hệ thống có 10 băng từ rỗi). Cuối cùng P_2 thực thi và giải phóng các tài nguyên (cuối cùng hệ thống có 12 băng từ rỗi). Lưu ý, từ trạng thái an toàn có thể dẫn đến trạng thái không an toàn. Giả sử tại thời điểm t_1 , P_2 yêu cầu và được cấp phát thêm 1 băng từ. Hệ thống sẽ không ở trạng thái an toàn nữa. Lúc này, chỉ tiến trình P_1 có thể lấy đủ băng từ theo nhu cầu của mình và khi P_1 trả lại, hệ thống có 4 băng từ. Trong khi đó, P_0 có thể yêu cầu thêm 5 hoặc P_2 yêu cầu thêm 6 băng từ, do hệ thống không thể đáp ứng được nên P_0 và P_2 rơi vào trạng thái bế tắc. Sai lầm ở đây là cấp phát thêm một băng từ cho P_2 . Nếu bắt P_2 đợi tới khi các tiến trình khác kết thúc và giải phóng tài nguyên, hệ thống có thể tránh được trạng thái bế tắc.

Có thể đảm bảo bế tắc không xuất hiện bằng cách bắt hệ thống luôn ở trạng thái an toàn. Ban đầu, hệ thống ở trạng thái an toàn. Khi tiến trình yêu cầu tài nguyên, hệ thống phải quyết định xem liệu có nên cấp phát không. Yêu cầu chỉ được đáp ứng nếu sau khi cấp phát, hệ thống vẫn ở trạng thái an toàn. Có thể khi yêu cầu và hệ thống có đủ tài nguyên, nhưng tiến trình vẫn phải đợi. Do đó hiệu suất sử dụng tài nguyên cũng không cao.

7.4.2. Thuật toán Đò thị cấp phát tài nguyên

Nếu mỗi kiểu tài nguyên chỉ có đúng một đối tượng, thì có thể áp dụng đồ thị cấp phát tài nguyên trong mục 7.2.2 để tránh bế tắc. Ngoài cung Yêu cầu và Phân phối có thêm cung Nhu cầu. Cung nhu cầu $P_i \rightarrow R_j$ chỉ ra tiến trình P_i có thể yêu cầu tài nguyên R_j tại thời điểm nào đó trong tương lai. Cung này có hướng giống cung yêu cầu, nhưng được biểu diễn bởi một

đường đứt nét. Khi tiến trình P_i yêu cầu tài nguyên R_j , cung nhu cầu $P_i \rightarrow R_j$ biến thành cung yêu cầu. Tương tự, khi P_i giải phóng R_j , cung phân phối $R_j \rightarrow P_i$ được đổi thành cung nhu cầu $P_i \rightarrow R_j$. Giả sử tiến trình P_i yêu cầu tài nguyên R_j . Yêu cầu chỉ có thể được thỏa mãn khi việc chuyển cung yêu cầu $P_i \rightarrow R_j$ thành cung phân phối không tạo nên chu trình trong đồ thị cấp phát tài nguyên. Ở đây thuật toán xác định chu trình trong đồ thị có độ phức tạp $O(n^2)$, với n là số tiến trình trong hệ thống. Nếu không có chu trình, thì sau khi cấp phát tài nguyên, hệ thống vẫn ở trong trạng thái an toàn. Ngược lại, nếu có chu trình, việc cấp phát sẽ đặt hệ thống trong trạng thái không an toàn. Do vậy, tiến trình P_i sẽ phải đợi.

7.4.3. Thuật toán Ngân hàng

Thuật toán đồ thị cấp phát tài nguyên không áp dụng được nếu kiểu tài nguyên có nhiều đối tượng. Khi đó, phải sử dụng thuật toán Ngân hàng (được áp dụng ở bộ phận cho vay của ngân hàng). Người vay (tiến trình) sẽ được ngân hàng (bộ phận quản lý tài nguyên) cấp một hạn mức tín dụng (số tiền cực đại khách hàng được vay). Khách hàng có thể vay dưới hạn mức tín dụng của mình. Khi vay thêm, khách hàng chưa phải trả ngay lập tức khoản vay trước. Việc cho vay của ngân hàng được xác định dựa trên trạng thái của ngân hàng, tức là lượng tiền mặt có sẵn, hạn mức tín dụng và số tiền vay của từng khách hàng. Nếu ngân hàng có thể thỏa mãn ít nhất một khách hàng nào đó (kể cả khi người này vay tối đa bằng đúng hạn mức tín dụng) thì ngân hàng có thể đợi đến thời hạn thanh toán, rồi thu toàn bộ số tiền này về, sau đó tiếp tục phục vụ khách hàng khác. Nếu tất cả khách hàng đều có thể vay được theo nhu cầu của mình (tất nhiên không vượt quá hạn mức tín dụng) và sau đó trả lại tiền cho ngân hàng thì ngân hàng ở trong trạng thái an toàn. Trong thuật toán Ngân hàng, tiến trình thông báo số lượng cực đại các đối tượng của mỗi kiểu tài nguyên cần thiết. Khi tiến trình yêu cầu tài nguyên, hệ thống phải xác định liệu sau khi cấp phát thì hệ thống có còn trong trạng thái an toàn không? Nếu có, tài nguyên được cấp phát; ngược lại, tiến trình phải đợi. Sau đây là một số cấu trúc dữ liệu biểu diễn trạng thái hệ thống với n là số các tiến trình của hệ thống và m là số các kiểu tài nguyên:

Available[1..m]: Số lượng các tài nguyên trong mỗi kiểu tài nguyên. $\text{Available}[j] = k$, nghĩa là kiểu tài nguyên R_j có k đối tượng chưa cấp phát.

Max[1..n, 1..m]: Ma trận xác định yêu cầu lớn nhất của mỗi tiến trình. $\text{Max}[i, j] = k$, nghĩa là tiến trình P_i có thể yêu cầu nhiều nhất k đối tượng tài nguyên R_j .

Allocation[1..n, 1..m]: Ma trận xác định số lượng đối tượng của mỗi kiểu tài nguyên hiện thời được cấp phát cho mỗi tiến trình. $\text{Allocation}[i, j] = k$, nghĩa là P_i hiện thời được cấp phát k đối tượng R_j .

Need[1..n, 1..m]: Ma trận xác định lượng tài nguyên cần thiết của mỗi tiến trình. $\text{Need}[i, j] = k$, nghĩa là P_i có thể cần thêm k đối tượng tài nguyên R_j để hoàn thành nhiệm vụ.

Như vậy, $\text{Need}[i, j] = \text{Max}[i, j] - \text{Allocation}[i, j]$.

Giả sử X và Y là vector có độ dài n . Ta định nghĩa $X \leq Y$ khi và chỉ khi $X[i] \leq Y[i]$ với mọi $i = 1, 2, \dots, n$. Ví dụ, nếu $X = \{1, 7, 3, 2\}$ và $Y = \{0, 3, 2, 1\}$, thì $Y < X$. Định nghĩa $Y < X$ nếu $Y \leq X$ và $Y \neq X$. Có thể xem mỗi hàng trên ma trận Allocation và Need như một vector, ký hiệu là Allocation_i và Need_i . Allocation_i xác định những tài nguyên hiện thời P_i đang nắm giữ, Need_i xác định những tài nguyên mà P_i có thể yêu cầu để hoàn thành nhiệm vụ.

☞ Thuật toán xác định trạng thái an toàn

Thuật toán sau xác định trạng thái của hệ thống có an toàn hay không:

1. **Giả sử Work và Finish là vector có độ dài m và n . Khởi tạo $\text{Work} = \text{Available}$ và $\text{Finish}[i] = \text{false}$ với $i = 1, 2, \dots, n$.**
2. **Tìm i thỏa mãn cả hai điều kiện: $\text{Finish}[i] = \text{false}$ và $\text{Need}_i \leq \text{Work}$. Nếu không có i nào như thế, chuyển qua bước 4.**
3. **$\text{Work} = \text{Work} + \text{Allocation}_i$; $\text{Finish}[i] = \text{true}$. Quay lại bước 2.**
4. **Nếu $\text{Finish}[i] = \text{true}$ với tất cả i thì hệ thống ở trong trạng thái an toàn.**

Độ phức tạp của thuật toán này là $O(m \times n^2)$.

☞ Thuật toán Yêu cầu Tài nguyên

Giả sử Request_i là vector yêu cầu của tiến trình P_j . $\text{Request}[j] = k$, nghĩa là tiến trình P_j muốn k đối tượng của kiểu tài nguyên R_j . Khi tiến trình P_j yêu cầu tài nguyên, các bước sau đây được thực hiện lần lượt:

1. Nếu $\text{Request}_i \leq \text{Need}_i$, chuyển sang bước 2. Ngược lại, đưa ra một điều kiện lỗi do tiến trình vượt quá yêu cầu tối đa của nó.
 2. Nếu $\text{Request}_i \leq \text{Available}$, chuyển sang bước 3. Ngược lại, P_j phải đợi vì chưa đủ tài nguyên.
 3. Buộc hệ thống làm như thế đã cấp phát các tài nguyên P_j yêu cầu bằng cách sửa các trạng thái như sau:
 - a. $\text{Available} = \text{Available} - \text{Request}_i$
 - b. $\text{Allocation}_j = \text{Allocation}_j + \text{Request}_i$,
 - c. $\text{Need}_i = \text{Need}_i - \text{Request}_i$,
 4. Nếu hệ thống vẫn ở trong trạng thái an toàn sau khi cấp phát tài nguyên, hệ thống sẽ thực sự cấp phát tài nguyên cho P_j . Tuy nhiên, nếu trạng thái mới không an toàn, P_j phải đợi Request_i và hệ thống khôi phục lại trạng thái cấp phát tài nguyên cũ.

Ví dụ minh họa:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	R ₀ R ₁ R ₂ R ₃	R ₀ R ₁ R ₂ R ₃	R ₀ R ₁ R ₂ R ₃
P ₀	2 0 1 1	3 2 1 4	1 2 2 2
P ₁	0 1 2 1	0 2 5 2	
P ₂	4 0 0 3	5 1 0 5	
P ₃	0 2 1 4	1 5 3 0	
P ₄	1 0 3 0	3 0 3 3	

(a)

<u>Trang thái</u> <u>an toàn</u>	<u>Trang thái</u> <u>không an toàn</u>
R ₀ R ₁ R ₂ R ₃	R ₀ R ₁ R ₂ R ₃
2 0 1 1	2 0 1 1
0 1 2 1	0 1 2 1
4 0 0 3	4 0 0 3
0 2 1 4	1 2 1 0
1 0 3 0	1 0 3 0

(b)

Hình 7.3. Ví dụ về trạng thái hệ thống

Giả sử hệ thống có 4 kiểu tài nguyên R_0, R_1, R_2, R_3 với số lượng tương ứng là 8, 5, 9, 7. Yêu cầu cực đại tài nguyên cũng như lượng tài nguyên hiện thời cấp phát cho các tiến trình được minh họa trong Hình 7.3a. Lượng tài nguyên còn lại của hệ thống là (1, 2, 2, 2). Ta thấy dãy $(P_2, P_4, P_1, P_3, P_0)$ an toàn, nên hệ thống đang trong trạng thái an toàn. Hình 7.3b minh họa 2 ví dụ về hệ thống trong trạng thái an toàn và không an toàn.

7.5. PHÁT HIỆN BÉ TÁC

Nếu không ngăn chặn hoặc tránh, hệ thống có thể rơi vào trạng thái bế tắc. Khi đó, hệ thống cần xác định bế tắc xuất hiện chưa và sau đó khắc phục như thế nào? Cơ chế phát hiện và khắc phục đòi hỏi những chi phí phụ

trội không chỉ trong việc ghi nhớ những thông tin cần thiết cho thuật toán phát hiện, mà còn ở những mảnh mát chắc chắn sẽ có trong giai đoạn khắc phục.

7.5.1. Kiểu tài nguyên chỉ có một đối tượng

Trường hợp này có thể áp dụng thuật toán Đồ thị đợi - chờ được xây dựng từ đồ thị cấp phát tài nguyên, bằng cách xóa các đỉnh kiểu tài nguyên và các cung liên quan. Chính xác hơn, cung P_i tới P_j trong một đồ thị đợi - chờ có ý nghĩa là tiến trình P_j đang nắm giữ tài nguyên mà P_i cần. tồn tại cung $P_i \rightarrow P_j$ trong đồ thị đợi - chờ khi và chỉ khi đồ thị cấp phát tài nguyên tương ứng chứa hai cung $P_i \rightarrow R_q$ và $R_q \rightarrow P_j$ với tài nguyên R_q nào đó. Giống như trước kia, bế tắc xuất hiện trong hệ thống khi và chỉ khi có chu trình trong đồ thị đợi - chờ. Để xác định bế tắc, hệ thống cần xây dựng đồ thị đợi - chờ và định kỳ thực thi chương trình tìm kiếm chu trình trên đồ thị này. Độ phức tạp của thuật toán là $O(n^2)$, trong đó n là số đỉnh của đồ thị.

7.5.2. Kiểu tài nguyên có nhiều đối tượng

Nếu kiểu tài nguyên có nhiều đối tượng thì không áp dụng được kỹ thuật Đồ thị đợi - chờ, mà phải sử dụng thuật toán xác định bế tắc bằng cách kiểm tra mọi dây cấp phát có thể. Thuật toán có các cấu trúc dữ liệu **Available[1..m]**, **Allocation[1..n, 1..m]** và **Request[1..n, 1..m]**.

1. Giả sử **Work** và **Finish** là vector tương ứng có độ dài m và n . Khi tạo **Work** = **Available**. Với $i = 1, 2, \dots, n$. Nếu $\text{Allocation}_{i,i} \neq 0$ thì $\text{Finish}[i] = \text{false}$; ngược lại, $\text{Finish}[i] = \text{true}$.
2. Tìm chỉ số i thỏa mãn cả hai điều kiện sau: ($\text{Finish}[i] == \text{false}$) và ($\text{Request}_{i,i} \leq \text{Work}$). Nếu không tìm được i nào như vậy, chuyển sang bước 4.
3. $\text{Work} = \text{Work} + \text{Allocation}_{i,i}$. $\text{Finish}[i] = \text{true}$, chuyển sang bước 2.
4. Nếu với $1 \leq i \leq n$ mà $\text{Finish}[i] == \text{false}$, thì hệ thống ở trong trạng thái bế tắc. Hơn nữa, tiến trình P_i bị bế tắc.

Độ phức tạp của thuật toán là $O(m \times n^2)$. Hệ thống thu hồi các tài nguyên đã cấp phát cho P_i (trong bước 3) ngay khi xác định được $\text{Request}_{i,i} \leq \text{Work}$ vì P_i hiện thời không liên quan đến bế tắc ($\text{Request}_{i,i} \leq \text{Work}$). Do đó có thể lạc quan, giả sử P_i không yêu cầu thêm tài nguyên và sẽ sớm

trả lại hệ thống tất cả các tài nguyên đang nắm giữ. Nếu giả thiết này sai, bê tắc có thể xảy ra và sẽ được xác định trong lần thực thi thuật toán kế tiếp.

Để minh họa thuật toán, xét hệ thống có năm tiến trình từ P_0 tới P_4 , ba kiểu tài nguyên R_0, R_1, R_2 có số đối tượng tương ứng là 7, 2 và 6. Giả sử tại thời điểm T_0 , trạng thái cấp phát tài nguyên được minh họa trên Hình 7.4a. Hệ thống không ở trong tình trạng bế tắc, vì khi thuật toán thực thi, dãy $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ dẫn đến $\text{Finish}[i] = \text{true}$ với mọi i . Tuy nhiên, nếu P_2 yêu cầu thêm một đối tượng kiểu tài nguyên R_3 . Ma trận Request thay đổi như trong Hình 7.4b. Hệ thống bây giờ rơi vào trạng thái bế tắc. Mặc dù có thẻ thu hồi tài nguyên đang cấp phát cho P_0 , số lượng các tài nguyên sẵn có không đủ đáp ứng các yêu cầu của các tiến trình khác. Do đó, bế tắc xảy ra và liên quan đến các tiến trình P_1, P_2, P_3 và P_4 .

	Allocation	Request	Available	Request
	R ₀ R ₁ R ₂			
P ₀	0 1 0	0 0 0	0 0 0	0 0 0
P ₁	2 0 0	2 0 2		2 0 2
P ₂	3 0 3	0 0 0		0 0 1
P ₃	2 1 1	1 0 0		1 0 0
P ₄	0 0 2	0 0 2		0 0 2

Hình 7.4. Ví dụ về thuật toán Tránh bế tắc

7.5.3. Sử dụng thuật toán Phát hiện bể tắc

Thời điểm thực thi thuật toán phát hiện phụ thuộc vào hai yếu tố là mức độ xuất hiện và ảnh hưởng của bê tắc. Nếu bê tắc xảy ra thường xuyên, hệ thống phải liên tục thực thi thuật toán phát hiện. Khi bê tắc, tài nguyên cấp phát cho những tiến trình bê tắc bị lãng phí và theo thời gian, số lượng các tiến trình bê tắc gia tăng. Trong trường hợp cực đoan, hệ thống có thể kiểm tra bê tắc xuất hiện chưa mỗi khi có yêu cầu không được đáp ứng. Trong trường hợp này, có thể xác định không chỉ các tiến trình liên quan tới bê tắc, mà cả tiến trình gây ra bê tắc. Tất nhiên, thực hiện thường xuyên thuật toán phát hiện bê tắc gây ra chi phí phụ trội lớn. Giải pháp ít tốn kém hơn là thực hiện ở tần suất thấp. Ví dụ, một lần một giờ, hoặc khi hiệu suất sử dụng CPU giảm xuống dưới một ngưỡng nào đó (bê tắc làm giảm thông lượng hê

thống cũng như hiệu suất sử dụng CPU). Nếu thực thi thuật toán phát hiện tại các thời điểm ngẫu nhiên, sẽ có thể có nhiều chương trình bị tắc trong đồ thị tài nguyên. Khi đó, không xác định được tiến trình nào là nguyên nhân gây ra bế tắc.

7.6. KHẮC PHỤC BẾ TẮC

Làm thế nào để khắc phục bế tắc? Giải pháp đầu tiên là thông báo cho người quản trị và người quản trị có trách nhiệm xử lý. Giải pháp thứ hai là để cho hệ thống tự động phục hồi. Có hai lựa chọn để phá vỡ bế tắc:

- Thứ nhất, loại bỏ một số tiến trình để phá vỡ chu trình.
- Thứ hai, thu hồi tài nguyên của các tiến trình bị tắc.

7.6.1. Chấm dứt tiến trình

Hệ thống thu hồi tài nguyên của các tiến trình bị chấm dứt. Có hai giải pháp chấm dứt:

1. Chấm dứt tất cả các tiến trình bị tắc: Phá vỡ toàn bộ chu trình bị tắc, nhưng phí tổn khá lớn vì nhiều tiến trình bị tắc có thể đã thực thi trong thời gian dài, và một phần kết quả thực thi bị loại bỏ sau đó có thể phải thực hiện lại.
2. Lần lượt chấm dứt từng tiến trình cho tới khi phá vỡ chu trình bị tắc. Cứ sau khi kết thúc một tiến trình, hệ thống kiểm tra xem bế tắc còn hay không.

Việc chấm dứt tiến trình có thể không đơn giản. Nếu chấm dứt một tiến trình đang trong quá trình cập nhật file có thể gây lỗi cho file. Nếu sử dụng phương pháp (2), hệ thống phải xác định thứ tự chấm dứt tiến trình (hoặc nhóm tiến trình) để phá vỡ bế tắc. Quyết định này phụ thuộc vào chính sách của hệ thống, tương tự như vấn đề điều phối CPU. Nói chung, đây là vấn đề kinh tế, nên chấm dứt tiến trình với chi phí thấp nhất, do đó phải tính đến:

1. Độ ưu tiên của tiến trình.
2. Tiến trình thực thi được bao lâu, cần thêm bao lâu để kết thúc?
3. Tiến trình sử dụng bao nhiêu tài nguyên và là những kiểu tài nguyên gì (liệu có dễ dàng chiếm đoạt không)?

4. Tiến trình cần thêm bao nhiêu tài nguyên để hoàn thành?
5. Có bao nhiêu tiến trình cần bị chấm dứt?
6. Liệu tiến trình có đòi hỏi tương tác với người dùng hay là ở dạng lô?

7.6.2. Chiếm đoạt tài nguyên

Có thể chấm dứt bê tắc bằng cách lần lượt thu hồi tài nguyên của một số tiến trình và chuyển chúng cho những tiến trình khác cho tới khi chu trình bê tắc bị phá vỡ. Nếu sử dụng cách này, có 3 vấn đề cần xét:

1. **Lựa chọn nạn nhân:** Tiến trình bị thu hồi tài nguyên được gọi là nạn nhân. Trong quá trình thực hiện, hệ thống quyết định thứ tự thu hồi để giảm thiểu chi phí phụ trội đến mức thấp nhất. Các yếu tố phải tính đến như số lượng tài nguyên mà tiến trình đang nắm giữ, thời gian thực thi của tiến trình.
2. **Quay lui:** Rõ ràng tiến trình "nạn nhân" không thể tiếp tục chạy bình thường do thiếu tài nguyên. Hệ thống phải đưa tiến trình này quay lại một trạng thái an toàn, để sau này có thể khôi phục lại. Nhưng khó xác định thế nào là trạng thái an toàn? Do vậy, cách giải quyết đơn giản nhất là quay lui toàn bộ (chấm dứt tiến trình và sau đấy khởi động lại). Dĩ nhiên, nếu chỉ quay lại trạng thái vừa đủ để chấm dứt bê tắc sẽ hiệu quả hơn nhiều. Nhưng khi đó hệ thống phải duy trì nhiều thông tin trạng thái của các tiến trình đang chạy.
3. **Chết đói:** Hệ thống phải đảm bảo không xảy ra hiện tượng chết đói (có tiến trình luôn luôn bị chọn làm "nạn nhân"). Trong hệ thống đặt yếu tố kinh tế lên hàng đầu, có thể có những tiến trình luôn bị chọn làm nạn nhân và không thể thực thi. Hệ thống có thể giải quyết bằng cách giới hạn số lần bị chọn làm "nạn nhân" của một tiến trình.

7.7. NHẬN XÉT

Trạng thái bê tắc xuất hiện khi nhiều tiến trình đang chờ đợi vô định, một sự kiện được một tiến trình khác cũng đang trong trạng thái đợi gây ra. Về mặt nguyên tắc, có ba phương pháp xử lý bê tắc:

1. Buộc hệ thống không bao giờ rơi vào trạng thái bê tắc.
2. Cho phép hệ thống rơi vào trạng thái bê tắc, sau đó khắc phục.
3. Bỏ qua mọi vấn đề, giả định bê tắc không bao giờ xuất hiện.

Điều kiện cần để bê tắc xuất hiện là: Độc quyền truy xuất; giữ và chờ; không chiếm đoạt và vòng đợi. Để ngăn chặn bê tắc, hệ thống chỉ cần ngăn cản sự xuất hiện của ít nhất một trong bốn điều kiện trên. Giải pháp tránh bê tắc khác là, tiến trình báo trước về cách sử dụng tài nguyên của mình. Thuật toán Ngân hàng cần phải biết số lượng lớn nhất mỗi lớp tài nguyên tiến trình yêu cầu. Nếu hệ thống không có phương pháp nào để bảo đảm bê tắc không xuất hiện, thì cần phải phát hiện và khắc phục bê tắc. Thuật toán phát hiện được sử dụng để xác định bê tắc xuất hiện chưa, nếu có, hệ thống phải khắc phục bằng cách loại bỏ hoặc thu hồi tài nguyên của tiến trình bê tắc. Trong hệ thống chọn lựa "nạn nhân" dựa trên các yếu tố chi phí, cần chú ý đến hiện tượng "chết dói".

CÂU HỎI ÔN TẬP

1. Trình bày mô hình bê tắc.
2. Trình bày các điều kiện xảy ra bê tắc và các giải pháp tương ứng.
3. Trình bày các giải pháp ngăn chặn bê tắc.
4. Trình bày các phương pháp tránh và phát hiện bê tắc.