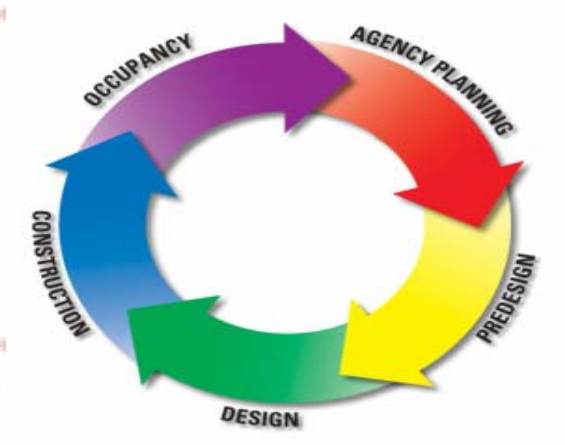
BÀI 2: QUẢN LÝ TIẾN TRÌNH



Nội dung

- Khái niệm tiến trình.
- Đồng bộ và giải pháp giải quyết tranh chấp.
- Tắc nghẽn.

Mục tiêu

- Trình bày được khái niệm về tiến trình, các trạng thái của tiến trình và quá trình biến đổi trạng thái đó.
- Trình bày được các khái niệm về luồng, đồng bộ và giải quyết tranh chấp.
- Trình bày được những vấn đề liên quan đến Deadlock.

Thời lượng học

- 9 tiết.

TÌNH HUỐNG DẪN NHẬP

Tình huống

Task Manager là một tiện ích giúp chúng ta quản lý các tiến trình trong máy tính. Vậy các tiến trình đó hoạt động như thế nào?

Câu hỏi

Đôi lúc ta gặp tình huống máy bị treo. Tại sao lại có hiện tượng đó? Có thể ngăn chặn và khôi phục nếu hiện tượng đó xảy ra không?

2.1. Khái niệm tiến trình (Process)

2.1.1. Định nghĩa

Trong bài này chúng ta sẽ xem xét khái niệm tiến trình, một khái niệm quan trọng nhất để hình dung về công việc của máy tính ngày nay.

Chúng ta sẽ tìm hiểu khái niệm về các trạng thái (rời rạc) của tiến trình và cũng như cách mà tiến trình chuyển từ trạng thái này sang trạng thái khác cùng với các thao tác cơ bản trên tiến trình.

Khái niệm tiến trình lần đầu tiên được các kỹ sư thiết kế hệ thống MULTICS vào những năm 60. Trong thời kỳ đầu tiên, tiến trình được hiểu trong nhiều trường hợp đồng nghĩa như là chương trình, bài toán (Task) hay là đối tượng được bộ xử lý phục vụ,...

Người ta thường dùng định nghĩa tiến trình như là chương trình trong lúc chạy.

2.1.2. Khối điều khiển tiến trình (Process Control Block – PCB)

Đại diện cho một tiến trình trong hệ điều hành là khối điều khiển tiến trình (PCB). PCB là một cấu trúc dữ liệu chứa những thông tin quan trọng về tiến trình và có thể khác nhau trong các hệ thống khác nhau, trong đó thường có:

- Trạng thái hiện tại của tiến trình.
- ID (Identifier) duy nhất cho tiến trình.
- Độ ưu tiên (Priority) của tiến trình.
- Thông tin về bộ nhớ.
- Thông tin về các tài nguyên tiến trình đang sử dụng.
- Vùng để cho các thanh ghi.

PCB là đối tượng quan trọng, nhờ nó hệ điều hành có thể có được toàn bộ thông tin cơ bản nhất về một tiến trình. Khi hệ điều hành chuyển (Switch) bộ xử lý từ đang phục vụ tiến trình này sang phục vụ tiến trình khác, nó dùng vùng cho các thanh ghi trong PCB lưu thông tin giá trị các thanh ghi của hệ thống để có thể tiếp tục thực hiện tiến trình mỗi khi tiến trình đến lượt được sử dụng bộ xử lý.

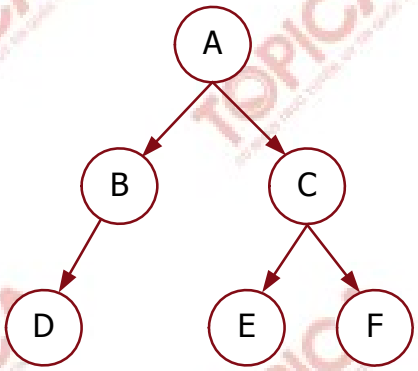
Tóm lại, PCB là đối tượng chính đại diện cho tiến trình đối với hệ điều hành. Vì hệ điều hành phải có khả năng thực hiện các thao tác với các PCB khác nhau một cách nhanh chóng, trong nhiều hệ thống có những thanh ghi đặc biệt luôn chỉ tới PCB của tiến trình đang chạy (Running Process) và cũng có những lệnh cài đặt ngay trong phần cứng để đảm bảo nhanh chóng ghi thông tin trạng thái vào PCB và tiếp theo là nhanh chóng đọc các thông tin đó.

Các thao tác với tiến trình:

Hệ thống điều khiển tiến trình cần có khả năng thực hiện các thao tác với tiến trình, trong đó có:

- Tạo tiến trình (Create).
- Huỷ tiến trình (Free, Destroy).
- Thay đổi độ ưu tiên (Priority).
- Dừng (Block) tiến trình.
- Kích hoạt (Waikup) tiến trình.
- Thực hiện (Dispatch) tiến trình.

Tiến trình tạo một tiến trình gồm nhiều thao tác nhỏ:



- Gán tên cho tiến trình.
- Đưa tên tiến trình vào danh sách các tiến trình của hệ thống.
- Xác định mức ưu tiên (Priority) ban đầu cho tiến trình.
- Tạo, nạp thông tin PCB.
- Phân chia tài nguyên khởi đầu cho tiến trình.

Tạo mới tiến trình:

Một tiến trình có thể tạo ra tiến trình mới. Tiến trình đầu tiên là tiến trình cha (Parent Process) còn tiến trình mới được tạo ra là tiến trình con (Child Process). Để tạo tiến trình mới chỉ cần một tiến trình đã có. Tức là mỗi tiến trình con chỉ có một tiến trình cha còn một tiến trình cha có thể có nhiều tiến trình con. Các quan hệ đó tạo ra kiến trúc tiến trình.

Xoá tiến trình:

Xoá một tiến trình là loại bỏ nó khỏi hệ thống. Khi đó các tài nguyên được phân chia cho tiến trình sẽ được giải phóng, trả lại cho hệ điều hành, tên của tiến trình được xoá khỏi tất cả các danh sách của hệ thống, còn PCB cũng được giải phóng.

Dừng/ hoãn tiến trình:

Một tiến trình bị hoãn–dừng (Suspended Process) là tiến trình không tiếp tục được thực hiện đến khi có một tiến trình khác kích hoạt nó. Tạm dừng (Suspending) là một thao tác quan trọng được sử dụng trong nhiều hệ thống với các cách cài đặt, thực hiện khác nhau. Tạm dừng thường chỉ diễn ra trong khoảng thời gian ngắn.

Ví dụ: hệ điều hành phải tạm dừng một số tiến trình (không phải luôn là tất cả) trong thời gian ngắn khi hệ thống quá tải,…

Trong trường hợp tiến trình bị dừng trong thời gian dài hơn thì các tài nguyên của nó phải được giải phóng trả lại cho hệ điều hành. Việc một loại tài nguyên có cần giải phóng hay không còn phụ thuộc vào kiểu của nó.

Ví dụ: bộ nhớ cần được giải phóng ngay, còn thiết bị vào ra có thể vẫn thuộc quyền sử dụng tiến trình trong trường hợp tiến trình bị hoãn–dừng trong thời gian ngắn còn sẽ được giải phóng khi thời gian Suspend dài hay không xác định.

Tiến trình kích hoạt (Activate) là thao tác chuẩn bị để tiến trình có thể tiếp tục thực hiện từ đúng trạng thái mà nó bị dừng trước đó.

Quá trình huỷ bỏ một tiến trình sẽ khá phức tạp nếu nó là tiến trình cha. Trong một số hệ thống thì các tiến trình con sẽ tự động bị huỷ bỏ theo, còn trong một số hệ thống khác thì tiến trình con vẫn tồn tại (độc lập với tiến trình cha).

Sự thay đổi độ ưu tiên của một tiến trình thường đơn giản là thay đổi giá trị ưu tiên trong PCB bởi hệ điều hành.

2.1.3. Trạng thái của tiến trình

Trong thời gian tồn tại của mình, tiến trình tồn tại trong các trang thái tách biệt (rời rạc). Sự đổi từ trạng thái này sang trạng thái khác có thể xảy ra bởi các sự kiện khác nhau.

Ta nói rằng tiến trình ở trạng thái hoạt động (Running State) nếu nó đang được bộ xử lý phục vụ. Còn nếu tiến trình đã sẵn sàng để được bộ xử lý phục vụ nhưng đang chờ đến lượt thì tiến trình ở trạng thái sẵn sàng (Ready State). Nói rằng tiến trình ở trạng thái bị cản, chặn (Blocked State) nếu như nó đang chờ một sự kiện nào đó (ví dụ kết thúc tác vụ vào/ra) để có thể tiếp tục hoạt động. Ngoài 3 trạng thái nói trên còn một số trạng thái khác nhưng tạm thời chúng ta chỉ xem xét quan hệ giữa 3 trạng thái trên.

Để đơn giản chúng ta xem xét trường hợp máy tính chỉ có một bộ xử lý. Trong hệ thống một bộ xử lý, tại một thời điểm chỉ có thể có một tiến trình được thực hiện, còn một số tiến trình nằm trong trạng thái sẵn sàng (Ready) và một số khác trong trạng thái bị chặn (Blocked). Do đó chúng ta có thể lập một danh sách chứa các tiến trình ở trạng thái Ready và một danh sách các tiến trình bị khóa. Mỗi tiến trình sẵn sàng nằm trong danh sách thứ nhất sẽ có mức độ ưu tiên riêng (Priority) của mình– tức là các tiến trình đó được sắp xếp theo thứ tự và tiến trình nằm ở đầu danh sách sẽ là tiến trình có độ ưu tiên cao nhất và sẽ được bộ xử lý thực hiện tiếp theo (có nhiều tiêu chuẩn để gán độ ưu tiên và thay đổi độ ưu tiên). Còn danh sách các Blocked Process nói chung không có thứ tự vì Blocked Process sẽ được giải phóng (Unblock) bởi các sự kiện mà nó đang chờ.

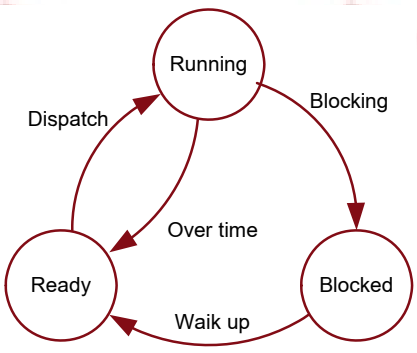
2.1.4. Biến đổi trạng thái

Khi có một chương trình (Task) bắt đầu được thực hiện, hệ thống sinh ra một tiến trình tương ứng và tiến trình đó được đưa vào danh sách các tiến trình sẵn sàng, đơn giản nhất là đưa vào cuối danh sách – tức là có mức ưu tiên thấp nhất. Tiến trình này sẽ dịch chuyển dần lên phía đầu danh sách bởi vì các tiến trình trước nó dần dần được bộ xử lý phục vụ. Khi tiến trình nằm ở đầu danh sách và bộ xử lý được giải phóng thì tiến trình này được bộ xử lý phục vụ và lúc đó xảy ra sự thay đổi trạng thái của tiến trình – chuyển từ trạng thái Ready sang Running. Việc trao quyền sử dụng bộ xử lý cho tiến trình đầu tiên trong danh sách các tiến trình sẵn sàng gọi là tiến trình dispatching, điều đó được thực hiện bởi module chương trình nằm trong OS gọi là Dispatcher.

tiến trình đổi trạng thái đó có thể biểu diễn bằng ký hiệu:

```
dispatch(Process name): Ready ⇒ Running
```

Tiến trình đang sử dụng bộ xử lý được gọi là tiến trình đang được thực hiện.



Để ngăn chặn trường hợp vô tình hoặc cố ý độc quyền chiếm tài nguyên hệ thống của tiến trình, hệ điều hành sinh ra một ngắt cứng đặc biệt ngắt thời gian (Timer Interrupt) nhằm xác định khoảng thời gian lớn nhất mà một tiến trình được sử dụng bộ xử lý liên tục. Nếu như sau khoảng thời gian đó, tiến trình không tự giải phóng bộ xử lý thì hệ thống sẽ sinh ngắt, theo đó quyền điều khiển được chuyển lại cho hệ điều hành. Lúc đó hệ điều hành sẽ chuyển tiến trình đang được thực hiện từ trạng thái Running về trạng thái Ready, đưa nó vào danh sách các tiến trình sẵn sàng, sau đó đưa tiến trình đầu tiên trong danh sách (tiến trình có mức ưu tiên cao nhất) vào thực hiện (Running State). Các sự biến đổi này có thể biểu diễn bằng hai thao tác:

```
interval gone (Process name): Running ⇒ Ready
dispatch (Process name): Ready ⇒ Running
```

Nếu như một tiến trình đang sử dụng bộ xử lý (Running State) trong quá trình hoạt động của mình thực hiện tác vụ vào/ra (I/O) thì nó sẽ tự mình giải phóng bộ xử lý (tự mình chuyển vào trạng thái Blocked để chờ tác vụ vào/ra kết thúc). Sự chuyển trạng thái này có thể biểu diễn:

```
blocking (Process name): Running ⇒ Blocked
```

Còn một tiến trình thay đổi trạng thái cuối cùng, đó là khi kết thúc tác vụ vào/ra (hay nói chung xảy ra một sự kiện mà tiến trình bị khóa đang chờ) lúc đó tiến trình chuyển từ trạng thái Blocked sang trạng thái Ready – sẵn sàng để thực hiện tiếp. Tiến trình này có thể biểu diễn:

```
waikup(Process name): Blocked ⇒ Ready
```

Với 3 trạng thái cơ bản trên, chúng ta có 4 khả năng chuyển trạng thái của một tiến trình đó là:

```
Dispatch (Process name): Ready ⇒ Running
interval gone(Process name): Running ⇒ Ready
blocking (Process name): Running ⇒ Blocked
waikup (Process name): Blocked ⇒ Ready
```

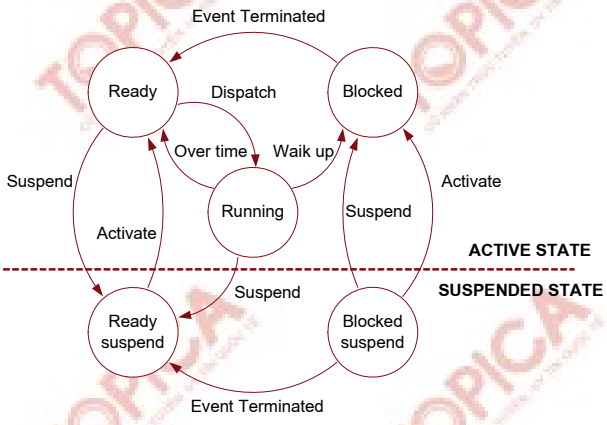
Chú ý:

Trong 4 khả năng trên, chỉ có khả năng thứ 3 là có thể sinh ra bởi chính chương trình người sử dụng, còn lại các khả năng khác đều do các đối tượng khác ở bên ngoài tiến trình gây ra.

Tạm dừng và kích hoạt (Suspending and Activating):

Chúng ta đã biết các khái niệm tạm dừng và kích hoạt. Các thao tác này khá quan trọng do các lý do:

- Nếu hệ thống hoạt động không ổn định có dấu hiệu trục trặc thì các tiến trình đang diễn ra cần tạm dừng để lại được kích hoạt sau khi sửa lỗi.
- Người sử dụng (lập trình viên) có thể cần tạm dừng (không phải huỷ bỏ) tiến trình để kiểm tra kết quả trung gian xem chương trình có hoạt động đúng hay không.



- Một số tiến trình có thể bị tạm dừng trong khoảng thời gian ngắn khi hệ thống quá tải và sau đó lại được kích hoạt khi có đủ tài nguyên (hệ thống trở về trạng thái bình thường).

So với mục trước, ta có thêm hai trạng thái ứng với các thao tác tạm dừng và kích hoạt.

Tác nhân dừng có thể là chính bản thân tiến trình hay là tiến trình khác.

Trong hệ có một bộ xử lý thì tiến trình chỉ có thể dừng chính bản thân nó vì không có tiến trình khác nào đang chạy đồng thời với nó. Còn trong hệ điều hành có nhiều bộ xử lý thì một tiến trình có thể bị dừng bởi tiến trình khác đang chạy trên bộ xử lý khác.

Một tiến trình ở trạng thái Ready chỉ có thể bị dừng bởi tiến trình khác, lúc đó xảy ra sự chuyển trạng thái:

```
Suspend (Process name): Ready ⇒ Suspended-Ready
```

Tiến trình đang ở trạng thái Suspended–Ready có thể chuyển về trạng thái Ready bởi tiến trình khác; tiến trình chuyển trạng thái đó có thể biểu diễn bởi:

```
Activate (Process name): Suspend-Ready ⇒ Ready
```

Tiến trình đang ở trạng thái Blocked có thể chuyển sang trạng thái Suspend bởi một tiến trình khác, khi đó diễn ra sự đổi trạng thái:

```
Suspend (Process name): Blocked ⇒ Suspend-Blocked
```

Và ngược lại, tiến trình ở trạng thái Suspended Blocked có thể được kích hoạt bởi một tiến trình khác.

```
Activate (Process name): Suspended-Blocked ⇒ Blocked
```

Chúng ta có thể đặt vấn đề tại sao không thay vì tạm dừng một tiến trình ở trạng thái Blocked, ta vẫn chờ đến khi có sự kiện (kết thúc I/O) mà tiến trình đợi xảy ra để tiến trình chuyển về trạng thái Ready. Tuy nhiên tác vụ I/O hay sự kiện tiến trình chờ có thể không xảy ra hay không biết khi nào mới xảy ra.

Như thế, các nhà thiết kế cần phải chọn lựa: tạm dừng một Blocked Process (đưa về trạng thái Suspended–Blocked) hoặc phải sinh ra cơ chế cho phép đưa tiến trình từ trạng thái Blocked sang trạng thái Ready và sau đó chuyển thành trạng thái Suspended–Ready khi kết thúc I/O hay diễn ra sự kiện tiến trình đang chờ.

Mặt khác thao tác tạm dừng thường có mức ưu tiên cao và cần thực hiện ngay, do đó phần lớn các hệ thống sử dụng cách thứ nhất. Khi sự kiện tiến trình đang chờ xảy ra (nếu như nó xảy ra), trạng thái của tiến trình sẽ chuyển từ Suspended – Blocked sang trạng thái Suspended – Ready:

```
Incommingevent (Process name): Suspended-Blocked ⇒ Suspended-Ready
```

2.1.5. Khái niệm luồng

Khái niệm:

Luồng là một dòng điều khiển trong phạm vi một tiến trình. Tiến trình đa luồng gồm nhiều dòng điều khiển khác nhau trong cùng không gian địa chỉ.

Một luồng thường được gọi là tiến trình nhẹ (Lightweight Process–LWP), là một đơn vị cơ bản của việc sử dụng CPU; nó hình thành gồm:

- Một định danh luồng (Thread ID).
- Một bộ đếm chương trình.
- Tập thanh ghi.
- Ngăn xếp.

Một luồng chia sẻ với các luồng khác thuộc cùng một tiến trình phần mã, phần dữ liệu, và tài nguyên hệ điều hành như các tập tin đang mở và các tín hiệu.

Một tiến trình truyền thống (hay tiến trình nặng) có một luồng điều khiển đơn.

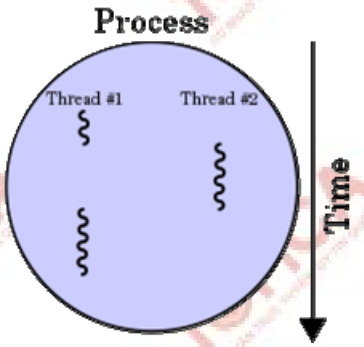
Nếu tiến trình có nhiều luồng điều khiển, nó có thể thực hiện nhiều hơn một tác vụ tại một thời điểm.

Tại sao đa luồng?

Trong nhiều trường hợp, một ứng dụng có thể được yêu cầu thực hiện nhiều tác vụ.

Ví dụ: Một máy chủ Web thể phải phục vụ hàng trăm trình duyệt truy xuất đồng thời. Nếu máy chủ Web chạy như một tiến trình đơn luồng truyền thống thì nó sẽ có thể chỉ phục vụ một trình khách tại cùng thời điểm. Dẫn đến thời gian mà trình khách phải chờ yêu cầu của nó được phục vụ là rất lớn.

Giải pháp thứ nhất là sử dụng đa tiến trình, có một tiến trình phục vụ chạy như một tiến trình đơn tiếp nhận các yêu cầu. Khi trình phục vụ nhận một yêu cầu, nó sẽ tạo một tiến trình mới để phục vụ yêu cầu đó. Phương pháp đa tiến trình này là cách sử dụng thông thường trước khi luồng trở nên phổ biến. Tuy nhiên, việc tạo ra tiến trình chiếm nhiều tài nguyên và thời gian của hệ thống.



Hình vẽ: ***Tiến trình đa luồng với hệ thống một bộ xử lý***

Nếu tiến trình mới sẽ thực hiện cùng tác vụ như tiến trình đã có thì tại sao lại phải chịu tất cả chi phí đó? Thường sẽ hiệu quả hơn cho một tiến trình gồm nhiều luồng phục vụ cùng một mục đích. Tiếp cận này sẽ đa luồng hóa tiến trình máy chủ Web. Tiến trình sẽ tạo một luồng riêng lắng nghe các yêu cầu; khi yêu cầu được thực hiện nó không tạo ra tiến trình khác mà sẽ tạo một luồng khác xử lý yêu cầu.

Ưu điểm của đa luồng:

Khả năng đáp ứng: đa luồng một ứng dụng cho phép một chương trình tiếp tục chạy, thậm chí một phần của nó bị khóa hay đang thực hiện một thao tác dài. Do đó cải thiện sự đáp ứng đối với người dùng. Ví dụ, máy chủ Web vẫn có thể đáp ứng yêu cầu của người dùng mới bằng một luồng trong khi yêu cầu của người dùng cũ đang được xử lý.

Chia sẻ tài nguyên: mặc định, các luồng chia sẻ bộ nhớ và các tài nguyên của các tiến trình mà chúng thuộc về. Thuận lợi của việc chia sẻ mã là nó cho phép một ứng dụng có nhiều hoạt động của các luồng khác nhau nằm trong cùng không gian địa chỉ.

Kinh tế: cấp phát bộ nhớ và các tài nguyên cho việc tạo các tiến trình là rất đắt. Vì các luồng chia sẻ tài nguyên của tiến trình mà chúng thuộc về nên nó kinh tế hơn để tạo và chuyển ngữ cảnh giữa các luồng. Khó để đánh giá theo kinh nghiệm sự khác biệt chi phí cho việc tạo và duy trì một tiến trình hơn một luồng, nhưng thường nó sẽ mất nhiều thời gian để tạo và quản lý một tiến trình hơn một luồng. Ví dụ, trong hệ điều hành Sun Solaris 2, tạo một tiến trình chậm hơn khoảng 30 lần tạo một luồng và chuyển đổi ngữ cảnh chậm hơn 5 lần.

Khai thác kiến trúc đa xử lý: các lợi điểm của đa luồng có thể phát huy trong kiến trúc đa xử lý, ở đó mỗi luồng thực thi song song trên một bộ xử lý khác nhau. Một tiến trình đơn luồng chỉ có thể chạy trên một CPU. Đa luồng trên một máy nhiều CPU gia tăng tính đồng hành. Trong kiến trúc đơn xử lý, CPU thường chuyển đổi qua lại giữa mỗi luồng quá nhanh để tạo ra hình ảnh của sự song song nhưng trong thực tế chỉ một luồng đang chạy tại một thời điểm.

Luồng người dùng và luồng nhân:

Chúng ta vừa mới thảo luận là xem xét luồng như một chiều hướng chung. Tuy nhiên, hỗ trợ luồng được cung cấp hoặc ở cấp người dùng, cho các luồng người dùng hoặc ở cấp nhân, cho các luồng nhân.

Luồng người dùng: được hỗ trợ dưới nhân và được cài đặt bởi thư viện luồng tại cấp người dùng. Thư viện cung cấp hỗ trợ cho việc tạo luồng, lập thời biểu, và quản lý

mà không có sự hỗ trợ từ nhân. Vì nhân không biết các luồng cấp người dùng, tất cả việc tạo luồng và lập thời biểu được thực hiện trong không gian người dùng mà không cần sự can thiệp của nhân. Do đó, các luồng cấp người dùng thường tạo và quản lý nhanh, tuy nhiên chúng cũng có những trở ngại. Ví dụ, nếu nhân là đơn luồng thì bất cứ luồng cấp người dùng thực hiện một lời gọi hệ thống nghẽn sẽ làm cho toàn bộ tiến trình bị nghẽn.

Luồng nhân: được hỗ trợ trực tiếp bởi hệ điều hành. Nhân thực hiện việc tạo luồng, lập thời biểu, và quản lý không gian nhân. Vì quản lý luồng được thực hiện bởi hệ điều hành, luồng nhân thường tạo và quản lý chậm hơn luồng người dùng. Tuy nhiên, vì nhân được quản lý các luồng nếu một luồng thực hiện lời gọi hệ thống nghẽn, nhân có thể lập thời biểu một luồng khác trong ứng dụng thực thi. Trong môi trường đa xử lý, nhân có thể lập thời biểu luồng trên một bộ xử lý khác. Hầu hết các hệ điều hành hiện nay như Windows NT, Windows 2000, Solaris 2, BeOS và Tru64 UNIX (trước Digital UNIX) – hỗ trợ các luồng nhân.

2.1.6. Xử lý ngắt

Trong thực tế có nhiều trường hợp tương tự ngắt trong máy tính.

Khái niệm:

Trong kỹ thuật máy tính, ngắt (Interrupt) là sự kiện làm thay đổi trình tự thực hiện lệnh bình thường của bộ xử lý.

Tín hiệu ngắt được phần cứng xử lý. Khi xảy ra ngắt, trình tự thực hiện như sau:

Bước 1: Điều khiển chuyển cho hệ điều hành.

Bước 2: Hệ điều hành lưu lại trạng thái của tiến trình bị ngắt. Trong nhiều hệ thống thì thông tin đó được lưu trong PCB của tiến trình bị ngắt.

Bước 3: Hệ điều hành phân tích loại ngắt và chuyển điều khiển cho chương trình xử lý ngắt tương ứng.

Bước 4: Tác nhân gây ra ngắt có thể là chính bản thân tiến trình đang chạy, hay là một sự kiện có thể liên quan hoặc không liên quan đến tiến trình đó.

2.1.6.1. Các dạng ngắt

Chúng ta xem xét các dạng ngắt trong các hệ thống máy lớn của IBM:

Dạng 1. SVC– Interrupt

Ngắt này do tiến trình đang chạy sinh ra. SVC do chương trình ứng dụng sinh ra để yêu cầu một dịch vụ nào đó của hệ thống, ví dụ thực hiện tác vụ vào/ra, cấp phát bộ nhớ... Cơ chế SVC giúp bảo vệ hệ điều hành, người sử dụng không được tự do xâm nhập OS mà anh ta phải yêu cầu dịch vụ thông qua lệnh SVC. Do đó hệ điều hành luôn kiểm soát được các thao tác vượt quá giới hạn ứng dụng và hoàn toàn có thể từ chối yêu cầu.

Dạng 2. Ngắt vào/ra

Do các thiết bị vào/ra sinh ra. Các ngắt này thông báo cho bộ xử lý về sự thay đổi trạng thái nào đó ví dụ kết thúc tác vụ in, máy in hết giấy,...

Dạng 3. External Interrupt

Ngắt này có thể do nhiều nguyên nhân sinh ra, trong đó có ngắt thời gian overtime, ngắt bàn phím, ngắt từ các bộ xử lý khác trong hệ thống đa bộ xử lý, ...

Dạng 4. Restart Interrupt

Sinh ra khi người điều kiển cần khởi động lại hệ thống, hay lệnh Restart SIGP của một bộ xử lý khác trong hệ thống đa bộ xử lý.

Dạng 5. Program check Interrupt

Ngắt sinh ra do lỗi hoạt động của chương trình ví dụ lệnh chia cho 0, ...

Dạng 6. Machine check Interrupt

Sinh ra do lỗi phần cứng trong hệ thống.

2.1.6.2. Chuyển ngữ cảnh – (Context Switching)

Để xử lý các loại ngắt, trong hệ điều hành có chương trình chuyên biệt gọi là Interrupt Handler. Như trên đã nêu, trong hệ thống có 6 loại ngắt, như thế trong hệ điều hành có 6 IH (Interrupt Handler) để xử lý 6 loại ngắt khác nhau. Khi có ngắt thì hệ điều hành ghi lại trạng thái của tiến trình bị ngắt và chuyển điều khiển cho chương trình xử lý ngắt tương ứng. Điều đó được thực hiện bởi phương pháp gọi là "chuyển ngữ cảnh" (Context Switching).

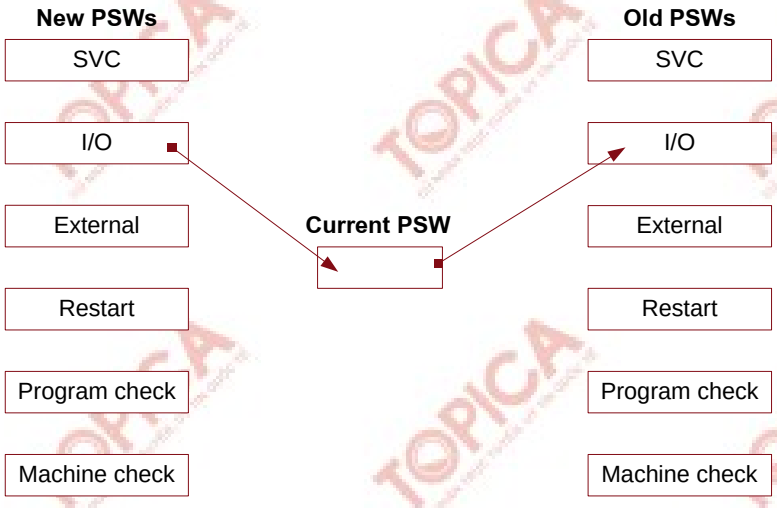
Trong phương pháp này sử dụng các thanh ghi trạng thái chương trình PSW (Program status word), trong đó chứa thứ tự thực hiện lệnh và các thông tin khác nhau liên quan đến trạng thái của tiến trình. Có 3 loại PSW:

- PSW hiện thời (Current).
- PSW mới (New).
- PSW cũ (Old).

Địa chỉ của lệnh tiếp theo (sẽ được thực hiện) được chứa trong Current PSW, trong Current PSW cũng chứa thông tin về những loại Interrupt nào hiện đang bị cấm (Disable) hay được phép (Enable). Bộ xử lý chỉ phản ứng với những loại ngắt được phép, còn các ngắt đang bị cấm sẽ được xử lý sau hoặc bỏ qua. Có một số Interrupt không bao giờ bị cấm: SVC, Restart,...

Trong hệ có một bộ xử lý thì chỉ có một Current PSW, nhưng có 6 New PSW (tương ứng cho mỗi loại ngắt) và 6 Old PSW tương ứng. New PSW của một loại ngắt chứa địa chỉ của chương trình xử lý ngắt (Interrupt Handler) loại đó.

Khi xảy ra ngắt (nếu loại ngắt đó không bị cấm), lúc đó sẽ tự động (do phần cứng thực hiện) xảy ra tiến trình chuyển đổi PSW như sau:



- Current PSW trở thành Old PSW của loại ngắt tương ứng.
- New PSW của loại ngắt đó trở thành Current PSW.

Như thế, sau khi chuyển đổi thì Current PSW chứa địa chỉ của chương trình xử lý ngắt và sau đó chương trình xử lý ngắt sẽ được thực hiện.

Khi kết thúc chương trình xử lý ngắt, bộ xử lý lại hoạt động bình thường, bộ xử lý sẽ tiếp tục phục vụ tiến trình bị ngắt hoặc có thể một tiến trình khác trong danh sách các tiến trình sẵn sàng.

Trong trường hợp tiến trình không cho phép giải phóng (nhường) quyền sử dụng bộ xử lý thì nó sẽ tiếp tục được bộ xử lý phục vụ, còn nếu nó cho phép thì nó tiếp tục được sử dụng bộ xử lý khi không có tiến trình sẵn sàng nào.

Trong các hệ thống, có nhiều mô hình xử lý ngắt khác nhau không hoàn toàn như mô hình trên.

2.1.7. Thiết kế phân lớp

Kiến trúc phân cấp của hệ thống

Trong việc thiết kế hệ điều hành, phương pháp xây dựng hệ điều hành phân cấp thành nhiều khối có nhiều ưu điểm. Tầng thấp nhất của kiến trúc thường là phần cứng, ở các lớp tiếp theo thường là các module với các chức năng khác nhau của hệ điều hành. Tổng hợp các module của Kernel ta có máy tính mở rộng (Extended Machine), nhờ đó hệ thống cung cấp nhiều dịch vụ khác nhau cho người dùng. Các chức năng mở rộng đó (do Kernel cung cấp) được gọi là các Primitive.

Phía trên Kernel là các tiến trình hệ thống (System Process) của hệ điều hành để phục vụ cho các tiến trình của người dùng. Còn trên cùng là các tiến trình của người sử dụng (User Process).

Kinh nghiệm cho thấy kiến trúc lớp làm cho công việc thiết kế, sửa đổi, Test dễ dàng hơn. Trong hệ thống mà Kernel gồm nhiều lớp, thì cần xem xét cẩn thận chức năng nào nằm ở lớp nào. Trong các hệ đó thường người ta hạn chế cho phép truy xuất từ trên xuống tức là tại mỗi lớp chỉ có thể thâm nhập đến lớp dưới kế tiếp mà thôi.

Hạt nhân của OS

Khái niệm: Tất cả các thao tác liên quan đến tiến trình, thực hiện bởi một phần hệ điều hành gọi là hạt nhân – Kernel.

Kernel chỉ là một phần không lớn (về kích thước Code) của hệ điều hành nhưng nó là một trong số những thành phần được sử dụng nhiều nhất trong hệ điều hành. Do đó Kernel thường luôn được nạp vào bộ nhớ, trong khi các thành phần khác có thể nằm ở bộ nhớ ngoài và chỉ được nạp vào khi cần.

Chức năng: Một trong những chức năng quan trọng nhất trong Kernel là xử lý ngắt. Trong các hệ lớn nhiều thành phần (Component) thường xuyên có dòng lớn (nhiều) ngắt. Do đó xử lý ngắt nhanh đóng vai trò quan trọng trên quan điểm sử dụng tài nguyên hệ thống và đảm bảo thời gian phản ứng với các yêu cầu của người dùng một cách nhanh chóng.

Khi Kernel xử lý ngắt, nó cấm các ngắt khác và chỉ cho phép tiếp tục xử lý ngắt sau khi xử lý xong ngắt hiện thời.

Trong trường hợp có dòng liên tục các ngắt thì có thể xuất hiện tình huống các ngắt bị chặn trong thời gian tương đối lớn tức là hệ thống không phản ứng kịp thời với các sự

kiện. Do đó Kernel thường được thiết kế sao cho nó chỉ thực hiện việc tiền xử lý tối thiểu và chuyển việc xử lý tiếp theo cho tiến trình hệ thống (System Process) tương ứng và có thể cho phép xử lý các ngắt tiếp theo. Theo đó các ngắt bị cấm trong khoảng thời gian nhỏ hơn do đó tốc độ phản ứng của hệ thống tăng đáng kể.

Các chức năng chính của Kernel

Kernel thường gồm các chương trình thực hiện các chức năng sau:

- Xử lý ngắt.
- Tạo và xoá các tiến trình.
- Đổi trạng thái của tiến trình.
- Dispatching.
- Suspend and Activate tiến trình.
- Đồng bộ (Synchronize) các tiến trình.
- Xử lý, tổ chức mối quan hệ giữa các tiến trình.
- Điều khiển PCBs.
- Quản lý bộ nhớ.
- Hỗ trợ làm việc hệ thống File.
- Cho phép (Enable) và cấm (Diasable) ngắt.

Xâm nhập Kernel thường được thực hiện thông qua ngắt, khi Kernel phản ứng với ngắt nào đó thì nó cấm các ngắt khác. Sau khi phân tích nó chuyển việc xử lý cho một tiến trình hệ thống chuyên làm việc với loại ngắt đó.

Trong một số hệ thống mỗi ngắt đều được xử lý bởi cả hệ điều hành cồng kềnh, do đó các ngắt thường bị cấm trong phần lớn thời gian nhưng về nguyên tắc hệ điều hành lại đơn giản hơn. Cách này thường áp dụng cho các máy nhỏ, làm việc với ít tiến trình. Còn với các hệ thống phức tạp, thường có một phần hệ điều hành chuyên xử lý ngắt cho phép nâng cao các chỉ số của cả hệ thống.

Thực hiện Kernel với MicroCode

Xu hướng: thiết kế nhiều chức năng với MicroCode đó là cách hiệu quả bảo vệ Kernel ngoài ra viết MicroProgram tốt có thể nâng cao tốc độ của cả hệ thống. Tất nhiên đổi lại việc thiết kế phức tạp hơn rất nhiều.

2.2. Đồng bộ và giải quyết tranh chấp (Asynchronous Concurent Process)

2.2.1. Tiến trình đồng thời

Các tiến trình gọi là đồng thời/tương tranh nếu các tiến trình đó tồn tại đồng thời. Các tiến trình tương tranh (Concurent Process) có thể hoạt động hoàn toàn độc lập với nhau hoặc song song không đồng bộ (Asynchronous), tức là theo chu kỳ chúng cần đồng bộ và tương tác với nhau. Song song không đồng bộ (Asynchronism) là một vấn đề phức tạp.

Chúng ta sẽ xem xét một số vấn đề liên quan đến điều khiển các tiến trình tương tranh không đồng bộ (Asynchronous Concurent Process). Các ví dụ được đưa ra với ngôn ngữ giả Pascal. Một số ví dụ về ngôn ngữ cho phép lập trình song song là ngôn ngữ Modula (Nicolar Witt), ngôn ngữ Ada.

2.2.2. Xử lý song song

Theo sự phát triển của máy tính chúng ta có thấy sự phổ biến của các hệ thống đa xử lý (MultiProcessor) và cùng với nó là sự phổ biến của xử lý song song. Nếu như một tiến trình Logic có thể xử lý song song Logic thì các hệ thống mới có thể xử lý chúng song song thực sự và có thể công việc được phân chia giữa các bộ xử lý khác nhau.

Xử lý song song là vấn đề được quan tâm và có nhiều khó khăn do một loạt nguyên nhân khác nhau. Con người theo tự nhiên có xu hướng chỉ chú ý đến một công việc tại mỗi thời điểm hơn là nghĩ đến nhiều việc khác nhau cùng một lúc. Thông thường khó mà xác định những thao tác nào có thể thực hiện song song và theo dõi một chương trình song song khó hơn nhiều so với chương trình xử lý tuần tự.

Các tiến trình song song không đồng bộ cần tương tác qua lại lẫn nhau theo chu kỳ thời gian và tương tác này có thể khá phức tạp. Cuối cùng, việc chứng tỏ sự đúng đắn cho các chương trình song song khó hơn nhiều so với trường hợp chương trình tuần tự và chúng ta cũng quan tâm đến phương pháp hiệu quả để chứng minh tính đúng đắn của chương trình, có như thế chúng ta mới có thể xây dựng các hệ thống có tính ổn định cao.

Các lệnh chỉ thị xử lý song song: parbegin và parend.

Trong nhiều ngôn ngữ lập trình đã có các chỉ thị yêu cầu xử lý song song (như trong Ada, Modula,...) các chỉ thị này thường đi theo cặp:

Chỉ thị đầu tiên chỉ ra rằng bắt đầu từ sau lệnh đó, chương trình được tách thành một số dòng điều khiển (Thread Control) thực hiện song song.

Chỉ thị thứ hai chỉ ra rằng từ đó chương trình lại được xử lý tuần tự.

Có nhiều tên khác nhau nhưng người ta thường dùng cặp parbegin/parend (Dijktra – Cooperating Sequenical Process). Nói chung đoạn mã chương trình được thực hiện song song có dạng:

```
parbegin
    operator 1
    operator 2
    ......
    operator n
parend
```

Việc thực hiện đoạn chương trình song song có thể hình dung như sau. Chương trình được thực hiện theo một luồng điều khiển tuần tự, đến khi gặp lệnh parbegin, luồng xử lý sẽ được chia thành n tiến trình xử lý độc lập, mỗi tiến trình sẽ xử lý một thao tác tương ứng từ operator 1, ... đến operator n. Thao tác này có thể là các lệnh đơn, lời gọi hàm, khối các lệnh tuần tự nằm giữa begin/end hay là tổ hợp các thao tác đó.

Các tiến trình xử lý sẽ dần thực hiện đến lệnh parend lúc đó luồng điều khiển lại hợp nhất thành một luồng xử lý các lệnh tiếp theo một cách tuần tự.

Ví dụ:

Xét biểu thức: x:= (–b + (b2 – 4*a*c) * 5) / (2*a)

Giải: Nếu tiến trình xử lý là hoàn toàn tuần tự chúng ta có thể làm theo các bước sau:

```
b2
4*a
(4*a)*c
```

```
b2-(4*a*c)
(b2-(4*a*c))*5
- b
- b+((b2-(4*a*c))*5)
2*a
(- b+(b2-(4*a*c))*5))/(2*a)
```

Các bước xử lý trên theo đúng trình tự quy tắc thực hiện phép toán.

Với hệ thống hỗ trợ xử lý song song chúng ta có thể làm như sau:

```
parbegin
temp1:= -b
temp2:= b2
temp3:= 4*a
temp4:= 2*a
parend
temp5:= temp3*c
temp5:= temp2-temp5
temp5:= temp5*5
temp5:= temp1+temp5
x:= temp5/temp4
```

Ta thấy nếu thực hiện xử lý song song thì thời gian tính toán giảm đi đáng kể so với khi tính tuần tự.

2.2.3. Vùng tranh chấp

Loại trừ lẫn nhau chỉ cần thiết trong trường hợp khi các tiến trình cùng truy nhập đến dữ liệu chung, hay khi chúng thực hiện các thao tác có thể dẫn tới tranh chấp (conflic) dữ liệu nói chung, còn khi chúng thực hiện các thao tác operation không dẫn tới tranh chấp thì hoàn toàn có thể thực hiện song song đồng thời. Khi tiến trình truy nhập đến dữ liệu chung thì người ta nói rằng lúc đó tiến trình nằm trong khoảng tới hạn (Critical Region).

Rõ ràng là để giải quyết tranh chấp thì khi có một tiến trình nằm trong khoảng tới hạn, cần phải không cho phép tiến trình khác (ít nhất là các tiến trình truy nhập đến cùng một dữ liệu) được vào khoảng tới hạn (tất nhiên chúng vẫn có thể thực hiện các thao tác khác ngoài khoảng tới hạn). Còn khi tiến trình ra khỏi khoảng tới hạn thì một trong số các tiến trình đang chờ vào khoảng tới hạn phải được tiếp tục vào khoảng tới hạn. Đảm bảo 'loại trừ lẫn nhau' là một trong những vấn đề mấu chốt của lập trình xử lý song song. Có nhiều phương pháp được đề xuất từ thực hiện hoàn toàn bằng phần mềm đến thực hiện bằng phần cứng, từ mức thấp đến mức cao, có những phương pháp cho phép loại trừ lẫn nhau trong điều kiện tương đối thoải mái, còn có những phương pháp thì bắt buộc phải có thêm các điều kiện chặt chẽ.

Khi một tiến trình nằm trong khoảng tới hạn thì có nhiều vấn đề cần quan tâm. Trong chế độ này nó có toàn quyền truy nhập dữ liệu chung còn các tiến trình khác phải chờ. Do đó các tiến trình cần ra khỏi chế độ này càng nhanh càng tốt, trong chế độ này nó không được chuyển sang trạng thái Blocked, do đó các khoảng tới hạn cần thiết kế, kiểm tra cẩn thận (ví dụ không cho phép xảy ra vòng lặp chờ trong khoảng tới hạn...).

Khi tiến trình kết thúc (ra khỏi) khoảng tới hạn (bình thường hoặc ngay cả khi có lỗi) thì hệ điều hành phải kiểm soát được để huỷ bỏ chế độ tới hạn, nhờ thế các tiến trình khác có thể đến lượt vào khoảng tới hạn.

Mutual Exclusion Primitive

Chương trình song song dưới đây đảm bảo bài toán trong mục 2.2.2 hoạt động đúng. Từ đây trở đi chúng ta xét trường hợp chỉ có hai tiến trình song song, cần phải nói rằng trường hợp có n tiến trình (n > 2) thì bài toán phức tạp hơn rất nhiều.

Trong chương trình 1 chúng ta dùng hai chỉ thị: Enter Mutual Exclusion và Exit Mutual Exclusion trong mỗi tiến trình ở đoạn Code truy nhập đến dữ liệu chung (biến toàn cục Total Line), hai chỉ thị này bao hai đầu khoảng tới hạn. Đôi khi các chỉ thị này được gọi là Mutual Exclusion Primitive

Chương trình 1

```
Program MutualExclusionSample
var
    totalLine:integer;
procedure Process1
begin
    while True do begin
    get line { readln}
    EnterMutualExclusion;
            totalLine:= totalLine +1;
    ExitMutualExclusion;
    Processing line ...
 end;
end;
procedure Process2
begin
 while True do begin
    get line { readln}
    EnterMutualExclusion;
            TotalLine:= totalLine +1;
    ExitMutualExclusion;
    Processing line ...
 end;
end;
begin
    totalLine:= 0;
    parbegin
            Process1;
            Process2;
    parend;
end.
```

Trong trường hợp có hai tiến trình, các lệnh Primitive làm việc như sau:

khi tiến trình 1 thực hiện chỉ thị Enter Mutual Exclusion và lúc đó tiến trình 2 ở ngoài khoảng tới hạn thì nó được vào khoảng tới hạn, thực hiện các lệnh trong khoảng tới hạn và đến khi thực hiện lệnh Exit Mutual Exclusion là lúc báo hiệu nó ra khỏi khoảng tới hạn.

Còn nếu khi tiến trình 1 muốn vào khoảng tới hạn, trong lúc đó tiến trình 2 đã ở trong khoảng tới hạn thì tiến trình 1 nó phải chờ đến khi tiến trình 2 ra khỏi khoảng tới hạn để có thể tiếp tục vào khoảng tới hạn.

Nếu cả hai tiến trình thực hiện Enter Mutual Exclusion cùng một lúc thì một trong hai tiến trình sẽ được phép vào khoảng tới hạn còn tiến trình kia sẽ phải chờ, có thể sự lựa chọn là ngẫu nhiên.

2.2.4. Tính loại trừ tương hỗ

Xét trường hợp hệ thống phục vụ trong chế độ phân chia thời gian cho nhiều thiết bị đầu cuối – Terminal. Giả sử khi người sử dụng đánh hết một dòng và gõ Enter, cần tính số dòng của tất cả các người sử dụng đã gõ từ tất cả các terminal. Để thực hiện điều đó, mỗi khi một người sử dụng nào đó gõ Enter thì tiến trình của người dùng đó sẽ tăng thêm 1 đơn vị cho một biến toàn cục (Global) Total Line. Vì hệ thống là đa xử lý và đa người dùng, do đó hoàn toàn có khả năng là hai người dùng khác nhau gõ Enter gần như đồng thời, khi đó 2 tiến trình điều khiển ứng với 2 người dùng đó sẽ đồng thời muốn truy nhập đến biến toàn cục Total Line. Để tăng biến đó giả sử mỗi tiến trình ứng dụng đều dùng các lệnh sau:

```
1- load totalline
2- totalline:= totalline + 1
3- store totalline
```

Giả sử tại một thời điểm, totalline có giá trị 62829.

Bây giờ nếu mới tiến trình 1 thực hiện được 2 lệnh đầu tiên:

load totalline (đọc giá trị hiện thời của biến) và tăng 1 cho giá trị biến totalline := totalline + 1 khi đó trong một thanh ghi (ví dụ Ax) chứa giá trị mới 62830 của biến totalline.

Sau đó tiến trình 1 không được quyền sử dụng bộ xử lý nữa (ví dụ do ngắt thời gian) và đến lượt tiến trình 2 được thực hiện. Giả sử tiến trình 2 kịp thực hiện cả 3 lệnh trên, khi đó giá trị của biến totalline sau khi thực hiện xong sẽ có giá trị 62830. Sau đó điều khiển được trả lại cho hệ điều hành và đến lượt tiến trình 1 được tiếp tục, nó thực hiện nốt lệnh thứ 3 tức là ghi lại giá trị 62830 vào biến totalline. Chúng ta thấy do sự điều khiển truy xuất không đúng mà chương trình hoạt động không đúng.

Ta thấy rằng vấn đề này có thể khắc phục nếu mỗi tiến trình có quyền truy nhập duy nhất đến biến totalline, tức là khi một tiến trình đang truy nhập đến biến totalline thì các tiến trình khác phải đợi đến khi tiến trình đầu tiên kết thúc truy nhập.

Như thế, khi một tiến trình truy nhập đến dữ liệu chung thì cần cấm tất cả các tiến trình khác truy nhập đến cùng dữ liệu vào thời điểm đó. Điều đó gọi là loại trừ lẫn nhau (Mutual Exclusion).

2.2.5. Phương pháp phần mềm

Chúng ta sẽ tìm cách thực hiện các Primitive Enter và Exit với các hạn chế sau:

- Vấn đề giải quyết hoàn toàn bằng chương trình (phần mềm) trên hệ thống không có lệnh chuyên cho loại trừ lẫn nhau. Mỗi lệnh được thực hiện trọn vẹn không bị ngắt giữa chừng. Khi có nhiều tiến trình cùng muốn truy nhập đến dữ liệu chung thì tranh chấp được giải quyết bằng phần cứng, một cách tình cờ sẽ có một tiến trình được chọn.

- Không có bất cứ giả sử gì về tốc độ tương đối của các Asynchronous parallel Process.
- Các tiến trình nằm ngoài khoảng tới hạn không thể cấm các tiến trình khác vào khoảng tới hạn.
- Không được để tiến trình chờ vô hạn để vào khoảng tới hạn.
- Cơ chế thực hiện loại trừ lẫn nhau bằng chương trình được nhà toán học Hà lan Dekker đề ra đầu tiên. Chúng ta sẽ xem xét các Version của thuật toán Dekker do Dijkstra thực hiện.

Thuật toán Dekker

Version 1:

Đầu tiên, chúng ta xem xét Version đầu tiên.

Có thể coi mỗi tiến trình như là một vòng lặp vô tận với nhiều lần lặp vào chế độ Mutual Exclusion. Trong Version này Primitive Enter Mutual Exclusion được thực hiện nhịp vòng lặp While chờ đến khi biến Processno bằng số của tiến trình, còn Primitive Exit Mutual Exclusion thực hiện như một lệnh đặt biến Processno bằng số của tiến trình khác.

Chương trình 1

```
Program Version1
var
    ProcessNo:integer;
procedure Process1
begin
while True do begin
    ....
    while ProcessNo = 2 do ;
    {Critical Region}
    ProcessNo:= 2;
    ....
end;
end;
procedure Process2
begin
while True do begin
    ....
    while ProcessNo = 1 do ;
    {Critical Region}
    ProcessNo:= 1;
    ....
end;
end;
begin
    ProcessNo:= 1;
    parbegin
            Process1;
            Process2;
    parend;
end.
```

Hoạt động: Cả hai tiến trình cùng thực hiện.

Vì đầu tiên biến ProcessNo gán bằng 1 do đó chỉ tiến trình 1 được vào Critical Region.

Tiến trình 2 khi muốn vào Critical Region kiểm tra thấy biến Processno có giá trị 1 do đó nó phải chờ bằng vòng lặp rỗng, nó chờ đến khi ProcessNo :=2 tức là khi tiến trình 1 ra khỏi Critical Region và đặt ProcessNo :=2.

Vậy chương trình đã đảm bảo loại trừ lẫn nhau (Mutual Exclusion).

Nhược điểm: Version1 có nhiều nhược điểm, tiến trình 1 phải vào Critical Region trước tiên (tức là dù tiến trình 2 sẵn sàng trước thì nó vẫn phải chờ) và các tiến trình lần lượt vào Critical Region theo thứ tự cố định (do đó nếu một tiến trình thực hiện các lệnh trong Critical Region thường xuyên hơn thì nó phải làm việc với tốc độ chậm hơn nhiều).

Chương trình này đảm bảo không rơi vào tình trạng Deadlock vì khi cả hai tiến trình cùng muốn vào Critical Region thì có ít nhất một tiến trình tiếp tục. Còn khi một tiến trình kết thúc thì sau đó tiến trình còn lại cũng kết thúc.

Version 2:

Trong Version 1 việc thực hiện Mutual Exclusion chỉ bằng một biến do đó có vấn đề các tiến trình vào khoảng tới hạn theo thứ tự cố định. Để cải tiến, trong Version 2 sử dụng hai biến Logic: flag1 và flag2, chúng nhận giá trị True khi tiến trình tương ứng nằm trong Critical Region.

Trong Version này tiến trình 1 chủ động chờ (Active wait) trong khi biến flag2 vẫn là True. Khi tiến trình 2 ra khỏi khoảng tới hạn, nó đặt lại flag2 := False và do đó tiến trình 1 ra khỏi vòng chờ while, đặt biến flag1 := True và vào khoảng tới hạn. Khi flag1 còn là True thì tiến trình 2 không thể vào khoảng tới hạn.

Chương trình 2

```
Program Version2
var
    flag1, flag2:boonlean;
procedure Process1
begin
while True do begin
    ....
    while flag2 = True do ;
    flag1:= True;
    {Critical Region}
    flag1:= False;
    ....
end;
end;
procedure Process2
begin
while True do begin
    ....
    while flag1 = True do ;
    flag2:= True;
```

```
        {Critical Region}
        flag2:= False;
        ....
    end;
    end;
    begin
        flag1:= False;
        flag2:= False;
        parbegin
                Process1;
                Process2;
        parend;
    end.
```

Nhưng lại xuất hiện một số vấn đề liên quan đến lập trình song song. Vì tiến trình 1 và tiến trình 2 là song song do đó chúng có thể đồng thời thử vào Critical Region. Đầu tiên các biến flag1 và flag2 có giá trị False. Tiến trình 1 kiểm tra biến flag2 thấy flag2 := False và thoát khỏi vòng lặp chờ, trước khi nó kịp đặt flag1 thành True bằng lệnh flag1 := True thì đến lượt tiến trình 2 được chiếm bộ xử lý, tiến trình 2 cũng có thể kiểm tra thấy flag1 là False (vì lúc đó tiến trình 1 chưa kịp đặt) lúc đó tiến trình 2 đặt flag2 := True và cũng vào Critical Region. Như thế cả hai tiến trình cùng vào chế độ Critical, do đó Version 2 không đảm bảo loại trừ lẫn nhau.

Nhược điểm: Điểm yếu của Version 2 là giữa thời điểm khi tiến trình nằm trong vòng chờ xác định thấy nó có thể đi tiếp và thời điểm nó đặt cờ (biến flag) nói rằng nó đã vào Critical Region. Do đó cần thiết để vào lúc tiến trình kết thúc vòng lặp chờ, tiến trình khác không thể ra khỏi vòng lặp chờ.

Version 3:

Trong chương trình Version3 (4,5) để giải quyết vấn đề này người ta đặt cờ cho mỗi tiến trình trước khi thực hiện vòng lặp chờ.

Chương trình 3

```
Program Version3
var
    flag1,flag2:boonlean;
procedure Process1
begin
while True do begin
    ....
    flag1:= True;
    while flag2 = True do ;
    {Critical Region}
    flag1:= False;
    ....
end;
end;
procedure Process2
begin
while True do begin
    ....
```

```
        flag2:= True
        while flag1 = True do ;
        {Critical Region}
        flag2:= False;
        ....
end;
end;
begin
        flag1:= False;
        flag2:= False;
        parbegin
                Process1;
                Process2;
        parend;
end.
```

Version 3 giải quyết được một vấn đề nhưng lại nảy sinh vấn đề khác. Nếu như mỗi tiến trình trước khi vào vòng chờ đặt cờ của mình thì mỗi tiến trình có thể kiểm tra thấy cờ của tiến trình khác đã đặt và cả hai tiến trình đều chờ ở vòng lặp vô tận. Chương trình này là một ví dụ của khái niệm Deadlock – tắc nghẽn.

Nhược điểm: Điểm yếu của Version 3 là ở chỗ mỗi tiến trình đều có thể bị Block ở vùng chờ. Chúng ta cần có biện pháp thoát khỏi vòng chờ này.

Version 4:

Trong Version 4 để làm điều đó, ta đặt lại cờ (biến flag) trong khoảng thời gian ngắn về giá trị False để tiến trình kia có cơ hội thoát khỏi vòng chờ.

Chương trình 4

```
Program Version4
var
        flag1,flag2:boonlean;
procedure Process1
begin
while True do begin
        ....
        flag1:= True;
        while flag2 = True do begin
                flag1:= False;
                Delay(random);
                flag1:= True;
        end;
        {Critical Region}
        flag1go:= False;
        ....
end;
end;
procedure Process2
begin
while True do begin
        ....
        flag2:= True
        while flag1 = True do begin
```

```
                flag2:= False;
                Delay(random);
                flag2:= True;
        end;
        {Critical Region}
        flag2:= False;
        ....
end;
end;
begin
        flag1:= False;
        flag2:= False;
        parbegin
                Process1;
                Process2;
        parend;
end.
```

Nhược điểm: Thoạt xem, Version4 đảm bảo sự loại trừ lẫn nhau và không bị Deadlock, nhưng lại xuất hiện vấn đề khác cũng rất quan trọng là vòng chờ vô tận. Chúng ta xét xem tại sao điều đó xảy ra. Vì chúng ta không có giả sử gì về tốc độ tương đối giữa các tiến trình do đó có thể có trường hợp các tiến trình lần lượt thực hiện dãy thao tác như sau trong một interval:

1. Đặt cờ flag giá trị True, vào vòng lặp chờ.

2. Đặt lại cờ flag giá trị False, chờ random.

3. Lại đặt cờ giá trị True và lặp lại tiến trình trong vòng chờ.

Khi đó các tiến trình thực hiện xong tất cả các thao tác đó, điều kiện kiểm tra luôn đúng và không thể thoát khỏi vòng chờ. Mặc dù trường hợp đó rất hiếm nhưng về nguyên tắc là có thể, do đó Version 4 cũng không thể áp dụng ví dụ như trong hệ thống điều khiển chuyến bay vũ trụ, ... dù xác suất nhỏ.

Thuật toán Dekker:

Như thế chúng ta cần khắc phục sự chờ vô tận. Thuật toán Dekker loại trừ tình trạng chờ vô tận trong Version 4. Với một số lượng không lớn Code, chương trình theo thuật toán của Dekker cho phép giải quyết triệt để vấn đề loại trừ lẫn nhau cho hai tiến trình, không đòi hỏi các lệnh đặc biệt nào.

Chương trình 5

```
Program Dekker
var
        flag1, flag2go: boonlean;
        selection: byte; {set of 1,2}
procedure Process1
begin
while True do begin
        ....
        flag1:= True;
        while flag2 = True do begin
                if selection = 2 then begin
        flag1:= False;
```

```
        while selection = 2 do ;
        flag1:= True;
                end;
        end;
        { Critical Region}
        selection:= 2;
flag1:= False;
        ....
end;
end;
procedure Process2
begin
while True do begin
        ....
        flag2:= True
while flag1 = True do begin
if selection = 1 then begin
        flag2:= False;
        while selection = 1 do ;
        flag2:= True;
                end;
end;
        { Critical Region}
        selection:= 1;
flag2:= False;
        ....
end;
end;
begin
        flag1:= False;
        flag2:= False;
        selection:= 1;
        parbegin
                Process1;
                Process2;
        parend;
end.
```

Nguyên lý hoạt động:

- Tiến trình 1 thông báo rằng muốn vào khoảng tới hạn bằng cách đặt cờ của mình (Process1go := True).
- Sau đó nó vào vòng lặp kiểm tra xem tiến trình 2 có muốn vào khoảng tới hạn hay không.
- Nếu như cờ của tiến trình 2 không đặt (Process2go := True) thì nó ra khỏi vòng lặp và vào khoảng tới hạn. Giả sử trong vòng lặp kiểm tra nói trên nó thấy cờ của tiến trình 2 đã đặt thì nó phải vào vòng lặp kiểm tra biến selectProcess – biến dùng để giải quyết tranh chấp khi cả hai tiến trình đồng thời muốn vào khoảng tới hạn.

- Nếu select Process := 1 thì nó ra khỏi vòng lặp của lệnh if và lặp lại kiểm tra cờ của tiến trình 2 – cho đến khi tiến trình 2 bỏ cờ của mình (chúng ta sẽ thấy tiến trình 2 nhất định sẽ bỏ cờ của mình).
- Nếu tiến trình 1 xác định thấy quyền ưu tiên thuộc về tiến trình 2 (biến select Process := 2) thì nó vào lệnh if và bỏ cờ của mình, sau đó lặp trong vòng chờ khi select Process vẫn là 2.
- Với việc bỏ cờ của mình tiến trình 1 cho phép tiến trình 2 ra khỏi vòng kiểm tra và đi vào khoảng tới hạn. Cùng với thời gian, tiến trình 2 ra khỏi khoảng tới hạn và thực hiện chỉ thị Exit Critical Region bằng 2 lệnh đặt quyền ưu tiên cho tiến trình 1 (select Process := 1) và bỏ cờ của mình (Process2go := False).
- Khi đó tiến trình 1 có cơ hội ra khỏi vòng chờ và đi vào khoảng tới hạn nếu như Process2go := False.
- Nếu tiến trình 2 ngay lập tức lại muốn vào khoảng tới hạn tức là nó lại đặt Process2go := True thì tiến trình 1 chưa thoát khỏi vòng lặp chờ ở ngoài (while Process2go := True do), khi đó tiến trình 2 vào vòng chờ, vào lệnh kiểm tra if và vì selectProcess := 1 – quyền ưu tiên thuộc về tiến trình 1 nên tiến trình 2 bỏ cờ của nó, do đó tiến trình 1 ra khỏi vòng lặp ngoài để vào khoảng tới hạn còn tiến trình 2 chờ đến lượt mình.
- Còn một khả năng mà chúng ta cần xem xét:
  - Khi tiến trình 1 ra khỏi vòng lặp ở trong, nó chưa kịp đặt cờ của nó thì bị mất quyền sử dụng bộ xử lý, đến lượt tiến trình 2 cũng muốn vào khoảng tới hạn; Nó đặt cờ của mình, kiểm tra thấy cờ của tiến trình 1 chưa dựng và lại đi vào khoảng tới hạn.
  - Khi tiến trình 1 lại có bộ xử lý, nó đặt cờ của mình và bị tiếp tục chờ ở vòng lặp ngoài. Vì select Process sẽ có giá trị 1 nên khi tiến trình 2 ra khỏi khoảng tới hạn tiến trình 2 sẽ không thể vào khoảng tới hạn một lần nữa và phải chờ ở vòng lặp trong, do đó tiến trình 1 có cơ hội vào khoảng tới hạn.

2.2.6. Phương pháp phần cứng

Thuật toán Decker là giải pháp phần mềm cho vấn đề loại trừ lẫn nhau. Bây giờ chúng ta sẽ xem xét giải pháp phần cứng. Điều quan trọng để đảm bảo giải quyết vấn đề là có một lệnh phần cứng: ***lệnh này đọc biến, ghi giá trị của biến vào vùng lưu trữ và đặt giá trị cụ thể cho biến đó. Lệnh này thường gọi là Test and set.*** Tất cả các thao tác trên được gói gọn trong một lệnh và được thực hiện trọn vẹn không bị ngắt.

Lệnh Test and set (a,b) đọc giá trị của biến logic b, copy giá trị vào biến a và sau đó đặt giá trị True cho b.

Ví dụ: sử dụng lệnh Testandset để thực hiện Mutual Exclusion được thể hiện trong chương trình 1.

Biến Active kiểu boolean có giá trị True khi một trong các tiến trình nằm trong khoảng tới hạn và giá trị False trong trường hợp ngược lại. Tiến trình 1 có được vào khoảng tới hạn hay không phụ thuộc vào giá trị biến local kiểu boolean Disable1. Nó đặt Disable1 := True, sau đó vào vòng lặp kiểm tra thực hiện lệnh Testandset đối với biến toàn cục active. Nếu lúc đó tiến trình 2 không nằm trong khoảng tới hạn, biến

active có giá trị False. Lệnh Test and set sẽ ghi giá trị đó vào biến Disable1 và đặt giá trị active := True. Do đó tiến trình 1 ra khỏi vòng lặp chờ kiểm tra và vào khoảng tới hạn. Vì biến active có giá trị True nên tiến trình 2 không thể vào khoảng tới hạn.

Chương trình 1

```
Program TestAndSet
var
    Disable1,Disable2: boonlean;
    active: boolean;
procedure Process1
var
    Disable1: boonlean;
begin
while True do begin
    ....
    Disable1:= True;
    while Disable1 = True do
            TestAndSet(Disable1, active);
    {Critical Region}
    active:= False;
    ....
end;
end;
procedure Process2
var
    Disable2: boonlean;
begin
while True do begin
    ....
    Disable2:= True;
    while Disable2 = True do
            TestAndSet(Disable2, active);
    {Critical Region}
    active:= False;
    ....
end;
end;
begin
    active:= False;
    parbegin
            Process1;
            Process2;
    parend;
end.
```

Giả sử rằng tiến trình 2 đã nằm trong khoảng tới hạn, khi đó:

tiến trình 1 muốn vào khoảng tới hạn nó đặt biến Disable1 := True và vào vòng kiểm tra. Vì giá trị của active là True (do tiến trình 2 đang trong khoảng tới hạn) nên lệnh

Test and set copy giá trị đó (True) vào biến Disable1 và như vậy điều kiện vòng lặp vẫn đúng và tiến trình 1 phải thực hiện vòng lặp kiểm tra.

Khi tiến trình 2 ra khỏi khoảng tới hạn, nó đặt lại biến active := False, lúc đó lệnh Testandset (trong vòng lặp kiểm tra của tiến trình 1) thấy biến active = False, copy giá trị này vào biến Disable1 và đặt active := True (làm cho tiến trình 2 không thể tiếp tục vào khoảng tới hạn) và vì Disable1 có giá trị False nên tiến trình 1 có thể đi vào khoảng tới hạn.

Nhược điểm: Nhưng phương pháp này không loại trừ được tình huống chờ vô tận dù xác suất xảy ra thấp, đặc biệt khi hệ thống có lớn hơn hai bộ xử lý. Khi một tiến trình ra khỏi khoảng tới hạn, đặt biến active := False, lệnh Test and set của tiến trình khác có thể 'chiếm' biến active (đặt lại giá trị bằng True) trước khi tiến trình đầu kịp thực hiện xong thao tác.

Loại trừ lẫn nhau cho N tiến trình

Giải pháp thực hiện bằng chương trình cho vấn đề Mutual Exclusion Primitive đối với N tiến trình lần đầu tiên được Dijkstra đề xuất. Sau đó Knuth hoàn thiện thêm phương pháp của Dijkstra, loại trừ được tình trạng chờ vô tận, nhưng vẫn còn nhược điểm là một số tiến trình vẫn có thể bị chờ lâu. Do đó nhiều nhà nghiên cứu tìm tòi những thuật toán cho phép rút ngắn thời gian trễ (chờ). Eisenberg và McGuite đã đề ra lời giải đảm bảo rằng tiến trình bất kỳ sẽ được vào khoảng tới hạn sau không quá N–1 lần thử. Lamport đã thiết kế thuật toán được áp dụng riêng cho các hệ cơ sở dữ liệu phân tán.

2.2.7. Semaphore

Các khái niệm trên liên quan đến loại trừ nhau, đã được Dijkstra tổng kết qua khái niệm Semaphore.

Semaphore là biến được bảo vệ, giá trị của nó chỉ có thể đọc, ghi bằng các lệnh đặc biệt P, V và lệnh khởi tạo. Semaphore nhị phân (Binary Semaphore) chỉ có thể nhận 2 giá trị: 0 và 1. Semaphore đếm (Counting Semaphore) chỉ có thể nhận giá trị tự nhiên.

Lệnh P trên Semaphore S, ký hiệu P(S) thực hiện như sau:

```
if S > 0 then S:= S - 1
else waiting for S
```

Lệnh V trên Semaphore S, ký hiệu V(S) thực hiện như sau:

```
if(exist Process waiting for S)then
allow one of them (waiting Process)continue
else S:= S + 1;
```

Chúng ta sẽ giả sử rằng hàng đợi các tiến trình đối với một Semaphore S sẽ được phục vụ theo nguyên tắc FIFO.

Cũng giống như lệnh Tets and Set, các lệnh P và V trên Semaphore S là một đơn vị, không chia cắt. Đoạn chương trình loại trừ nhau đối với semahore S của tiến trình sẽ được bao bởi các lệnh P(S) và V(S). Nếu đồng thời có nhiều tiến trình cùng muốn thực hiện lệnh P(S) thì chỉ có 1 tiến trình được phép, còn các tiến trình khác sẽ phải chờ.

Semaphore và các lệnh trên S có thể được xây dựng–cài đặt (implement) bằng phần mềm cũng như bằng phần cứng. Nói chung chúng được cài đặt trong nhân của hệ điều hành, là nơi thực hiện việc đổi trạng thái của tiến trình.

Chương trình 1.1 và 1.2 sau cho ta ví dụ đảm bảo loại trừ nhau bằng Semaphore. Trong đó, lệnh P có vai trò như giả lệnh EnterExclusion còn lệnh V báo về sự kết thúc.

Chương trình 1.1

```
Program Semaphore1
var active: Semaphore
procedure Process1;
begin
    while True do begin
            P(active);
            Critical Region;
            V(active);
    end;
end;
```

Chương trình 1.2

```
procedure Process2;
begin
    while True do begin
            P(active);
            Critical Region;
            V(active);
    end;
end;
begin
    initilizeSemaphore(active,1);
    parbegin
    Process1;
    Process2;
    parend
end.
```

Đồng bộ tiến trình dùng Semaphore

Khi tiến trình yêu cầu thực hiện I/O, nó tự block và chuyển sang trạng thái chờ sự kiện kết thúc thao tác I/O (Blocked Process). Tiến trình ở trạng thái Blocked cần phải được kích hoạt bởi 1 tiến trình nào đó. Sự kích hoạt đó có thể được thực hiện ví dụ bằng hàm nào đó liên quan đến giao thức khoá/kích hoạt.

Chúng ta xem xét trường hợp tổng quát hơn, khi 1 tiến trình cần phải được thông báo về một sự kiện nào đó xảy ra. Chúng ta giả sử rằng có 1 tiến trình khác nhận biết được sự kiện đó. Ví dụ sau đây là ví dụ chúng ta có thể dùng Semaphore để thực hiện đồng bộ giữa 2 tiến trình.

Ví dụ:

```
Program block_Activate
var event: Semaphore;
procedure Process1;
begin
    P(event);
end;
procedure Process2;
```

```
begin
    V(event);
end;
begin
    initializeSemaphore(event,0)
    parbegin
    Process1;
    Process2;
    parend
end.
```

Trong chương trình này, tiến trình 1 thực hiện bình thường, sau khi thực hiện yêu cầu I/O thì thực hiện lệnh P(event), bởi vì ban đầu, event = 0 nên tiến trình 1 sẽ phải chờ event.

Theo thời gian, khi sự kiện xảy ra, tiến trình 2 nhận biết nó và thực hiện V(event), bởi vì lúc đó có tiến trình 1 đang chờ event nên lệnh V rẽ nhánh True và cho phép tiến trình 1 tiếp tục.

Chú ý: chương trình này vẫn thực thi đúng nhiệm vụ của nó ngay cả khi sự kiện xảy ra trước khi tiến trình 1 thực hiện lệnh P(event).

Khi có sự kiện, tiến trình 2 thực hiện V(event) theo đó event := event + 1 = 1 vì không có tiến trình nào chờ. Sau đó, khi tiến trình 1 thực hiện đến lệnh P(event) thì vì event = 1 nên nó rẽ nhánh đặt lại event := event – 1 = 0 và tiếp tục mà không chờ.

Cặp "Cung – cầu":

Trong chương trình hoạt động tuần tự, khi một thủ tục (Procedure) gọi thủ tục thứ 2 và truyền thông tin cho nó, hai thủ tục này là thành phần của chương trình và không hoạt động song song. Nhưng khi 1 tiến trình truyền thông tin cho tiến trình khác, có thể xuất hiện vấn đề. Sự truyền thông tin đó có thể như là một ví dụ về loại trừ nhau.

```
Program provider1
var
    enable:semaphore;
    bufferReady:semaphore;
    buffer: integer;
procedure process1;
var
    nextResult:integer;
begin
    while true do
begin
    {xử lý, tính toán kết quả}
    P(enable);
    buffer:= nextResult;
    V(enable);
    V(bufferReady);
end;
end;
procedure process2;
var
```

```
        nextResult:integer;
begin
        while true do
begin
        P(bufferReady);
        P(enable);
        nextResult:= buffer;
        V(enable);
        {tiếp nhận thông tin nextResult }
end;
end;
Begin
        Init(enable,1);
        Init(bufferReady,0);
parbegin
        process1;
        process2;
        parend
End.
```

Chúng ta xem xét hoạt động của cặp tiến trình. Giả sử 1 tiến trình (tiến trình cho) sinh thông tin được tiến trình thứ 2 (tiến trình nhận) sử dụng. Giả sử chúng tương tác với nhau thông qua biến số nguyên mang thông tin có tên buffer. Tiến trình cho tạo thông tin và lưu kết quả vào biến buffer, sau đó tiến trình nhận đọc thông tin từ biến buffer đó để có thông tin.

Các tiến trình có thể hoạt động với tốc độ như nhau hoặc khác nhau nhiều. Nếu mỗi khi tiến trình 1 có kết quả, lưu vào buffer, tiến trình 2 lập tức đọc thông tin và in ra, sau đó đến tiến trình 1,.. thì đầu ra là dãy kết quả đúng như dãy do tiến trình 1 tính được.

Bây giờ giả sử 2 tiến trình hoạt động không đồng bộ. Nếu tiến trình 2 hoạt động nhanh hơn thì sẽ dẫn đến tình huống 1 kết quả tính toán (của tiến trình 1) được in 2 hay nhiều lần. Còn nếu ngược lại tiến trình 1 hoạt động nhanh hơn thì sẽ có những kết quả tính toán bị bỏ qua không được in ra.

Tất nhiên chúng ta muốn trong bất kỳ trường hợp nào, các tiến trình phải tương tác sao cho kết quả tính toán được in ra không nhiều và không lặp, nghĩa là ta muốn chúng đồng bộ.

Chương trình trên là một ví dụ thực hiện đồng bộ 2 tiến trình bằng Semaphore. Trong đó Semaphore Enable đảm bảo biến buffer chung không bị truy cập đồng thời, còn Semaphore bufferReady đảm bảo sự đồng bộ.

Counting Semaphore

Counting Semaphore hữu ích khi có tài nguyên được lấy từ nhóm (Pool) các tài nguyên cùng loại. Khi khởi tạo Semaphore, giá trị của nó là chỉ số lượng tài nguyên của nhóm. Mỗi lệnh P sẽ giảm giá trị của Semaphore đi 1, tương ứng với việc có 1 tài nguyên đã được lấy từ nhóm. Mỗi lệnh V sẽ tăng giá trị semaphare 1 tương ứng với việc tiến trình trả lại 1 tài nguyên cho nhóm và tài nguyên đó có thể được cấp cho tiến trình khác. Trong trường hợp thực hiện lệnh P mà giá trị của Semaphore là 0, nghĩa là không còn tài nguyên trống, thì tiến trình sẽ phải chờ đến khi nào giá trị Semaphore khác 0 nghĩa là có lệnh V được thực hiện (ứng với sự kiện có tài nguyên giải phóng).

2.3. Tắc nghẽn

2.3.1. Định nghĩa

Trong hệ thống đa chương trình, một tiến trình nằm trong trạng thái Deadlock hay treo, nếu như nó chờ sự kiện (Event) nào đó không bao giờ xảy ra. Tình huống treo hệ thống là tình huống có một hay nhiều tiến trình nằm trong trạng thái treo.

Trong các hệ thống đa chương trình một trong những chức năng quan trọng của hệ điều hành là quản lý, phân chia tài nguyên. Khi tài nguyên được chia sẻ giữa các user, mỗi người có toàn quyền điều khiển, sử dụng tài nguyên đã được phân chia cho anh ta, do đó hoàn toàn có thể xảy ra Deadlock và tiến trình của người dùng có thể chẳng bao giờ kết thúc.

Trong chương này chúng ta xem xét vấn đề Deadlock và một số kết quả nghiên cứu về vấn đề ngăn ngừa, phòng tránh và phát hiện tình trạng Deadlock cũng như khôi phục sau đó. Chúng ta cũng xem xét vấn đề liên quan là chờ vô tận (Indefinite Postponement – hoãn không xác định) khi tiến trình chờ một sự kiện nào đó có thể chẳng bao giờ xảy ra do lý do chủ quan nào đó trong hệ thống điều khiển tài nguyên.

Các khả năng, giải pháp cân bằng giữa giá phải trả cho các phương tiện ngăn chặn Deadlock với lợi ích nó mang lại cũng được xem xét. Trong nhiều trường hợp, giá phải trả cho việc loại trừ tình trạng Deadlock quá cao. Còn trong các trường hợp khác (ví dụ các hệ thống điều khiển thời gian thực) thì giá đó là không tránh khỏi vì Deadlock có thể gây ra những hậu quả không lường trước.

Ví dụ tình trạng tắc nghẽn

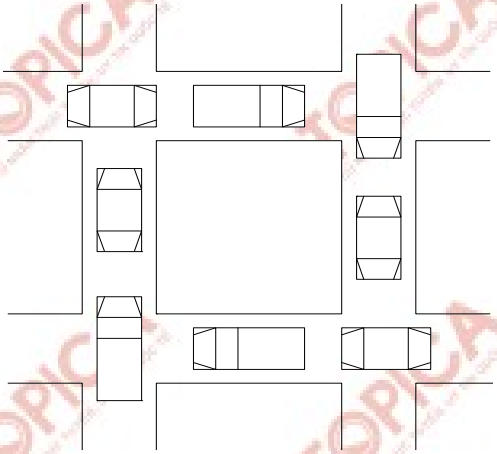
Có lẽ cách đơn giản nhất để tạo Deadlock là chương trình mà Holt đưa ra bằng ngôn ngữ PL/1, chạy dưới OS 360:

```
Revenge: procedure options (main,Task);
wait (event)
end revenge;
```

Tiến trình gắn với chương trình này sẽ luôn chờ sự kiện (event) nhưng nó lại không xem xét dấu hiệu xuất hiện sự kiện. Hệ thống bắt buộc nhận thấy tiến trình đó bị treo và sau đó phải loại bỏ tiến trình để thoát khỏi tình trạng Deadlock.

Deadlock do lỗi chương trình hay thuật toán thường khó phát hiện.

Ví dụ 1: Tắc nghẽn giao thông trong thực tế



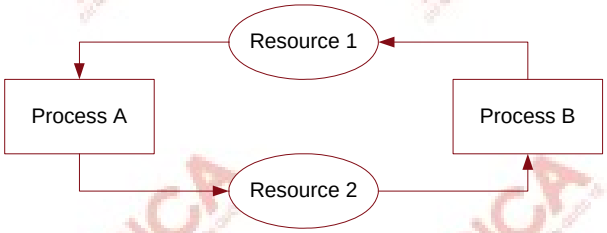
Hình 2.3.1–1

Ví dụ 2: Deadlock trong phân chia tài nguyên

Trong hệ điều hành, Deadlock phần lớn xuất hiện do hậu quả của sự tranh chấp sử dụng (chiếm) các tài nguyên mà tại mỗi thời điểm chỉ cấp cho một user, do đó đôi khi được gọi là tài nguyên sử dụng tuần tự. Trên hình 2.3.1–2 đưa ra một ví dụ đơn giản Deadlock dạng này. Trên sơ đồ phân chia tài nguyên có hai tiến trình và hai tài nguyên. Mũi tên đi ra từ tài nguyên vào tiến trình chỉ ra rằng tài nguyên đó đang thuộc quyền sử dụng của tiến trình đó. Còn mũi tên đi ra từ tiến trình vào tài nguyên chỉ ra rằng tiến trình đang yêu cầu sử dụng tài nguyên nhưng chưa được cấp phát tài nguyên tương ứng.

Sơ đồ này biểu diễn hệ thống đang ở trong tình trạng Deadlock: tiến trình A đang chiếm resource1 và để tiếp tục hoạt động nó cần resource2. Trong khi đó tiến trình B đang chiếm resource2 lại cần resource1 để tiếp tục.

Mỗi tiến trình đợi để tiến trình kia giải phóng resource mà nó đang cần, mặt khác mỗi tiến trình không giải phóng resource khi mà tiến trình kia chưa giải phóng tài nguyên của mình. Tình huống đợi vòng quanh này đưa hệ thống vào tình trạng Deadlock.



Hình 2.3.1–2

Ví dụ 3: Deadlock trong hệ thống dùng Spooling

Hệ thống Spooling thường xảy ra Deadlock. Chế độ Spooling (vào/ra với buffer) được áp dụng để nâng cao hiệu suất của hệ thống bằng cách phân cách chương trình khỏi các liên lạc trực tiếp với thiết bị ngoại vi tốc độ thấp như máy in, ...

Nếu như chương trình đưa một dòng văn bản ra máy in mà phải đợi đến khi in xong dòng đó mới tiếp tục in dòng tiếp theo thì chương trình hoạt động chậm đi nhiều do hạn chế tốc độ của máy in.

Để tăng tốc độ thực hiện chương trình, đầu tiên các dòng dữ liệu được ghi ra các thiết bị có tốc độ cao hơn như đĩa cứng và được lưu tạm thời ở đó trước khi được đưa ra máy in. Trong một số hệ thống Spooling, chương trình phải định dạng (Format) toàn bộ thông tin ra, chỉ sau đó mới bắt đầu thực sự in. Do đó khi một vài tiến trình đưa dữ liệu vào (Spooling File) để in, hệ thống có thể rơi vào tình trạng Deadlock, nếu như buffer định trước bị đầy khi chưa hoàn tất công việc.

Để khôi phục, hay thoát khỏi tình trạng đó có thể phải Restart System – dẫn đến mất toàn bộ kết quả công việc đã tiến hành, hoặc phải loại bỏ một số chương trình để các chương trình còn lại có thể hoạt động tiếp tục.

Khi người điều hành bắt đầu công việc anh ta thiết lập kích thước cho Spooling File. Một trong những cách làm giảm khả năng xuất hiện Deadlock khi Spooling là thiết lập kích thước ban đầu lớn hơn so với dự tính. Nhưng cách này không phải luôn thực hiện được (khả thi) khi bộ nhớ thiếu,... Giải pháp thông dụng hơn đối với tiến trình thiết lập ngưỡng để Spooling không tiếp nhận thêm công việc từ các chương trình khác khi Spooling File đã sử dụng ví dụ tới 75% không gian. Giải pháp này có thể dẫn tới giảm hiệu suất của hệ thống nhưng đó là giá phải trả để giảm xác suất xảy ra Deadlock.

Các hệ thống ngày nay hoàn thiện hơn, nó có thể cho phép bắt đầu in trước khi kết tất cả dữ liệu được định dạng nhờ đó một phần hay toàn bộ Spooling File được giải phóng (xoá) ngay trong tiến trình thực hiện công việc. Trong nhiều hệ thống có khả

năng cấp phát bộ đệm (Buffer) động để khi Spooling File sắp đầy thì nó được tăng kích thước.

Dù sao đi nữa ưu thế của Spooling vẫn lớn hơn rất nhiều những vấn đề Deadlock có thể nảy sinh.

Vấn đề chờ vô tận–hoãn không xác định (Indefinite Postponement)

Trong hệ thống, khi các tiến trình phải chờ ví dụ khi nó chờ được cấp phát tài nguyên hay lập lịch trình, có thể xuất hiện tình huống mà (quyền) được sử dụng bộ xử lý bị hoãn không xác định. Tình huống này gọi là hoãn vô thời hạn (không xác định) có thể dẫn tới tình huống không chấp nhận được cũng như tình trạng Deadlock.

Tình trạng hoãn vô thời hạn có thể xảy ra do cách điều khiển tài nguyên của hệ thống. Khi tài nguyên được phân bố theo nguyên tắc ưu tiên thì có thể xảy ra trường hợp một tiến trình sẽ chờ được cấp tài nguyên lâu vô hạn, bởi vì luôn có các tiến trình khác với độ ưu tiên cao hơn.

Khi thiết kế hệ điều hành cần xem xét các chiến lược điều khiển các tiến trình nằm trong trạng thái chờ. Trong nhiều hệ thống tình trạng hoãn vô hạn được ngăn chặn do độ ưu tiên của tiến trình tăng dần cùng với thời gian nó chờ được cấp tài nguyên. Do đó cuối cùng thì tiến trình đó có độ ưu tiên cao nhất và nó sẽ được phục vụ.

Khái niệm tài nguyên

Một chức năng quan trọng của hệ điều hành là quản lý và điều khiển các tài nguyên của hệ thống. Nó có trách nhiệm phân phối các tài nguyên thuộc các loại khác nhau của hệ thống.

Chúng ta xem xét các tài nguyên được coi là Preemtible (được xử dụng nhiều nhất) như là bộ xử lý hay bộ nhớ; Chương trình đang nằm trong bộ nhớ có thể bị đưa ra ngoài để cấp vùng nhớ đó cho các chương trình khác cần bộ nhớ.

Ví dụ 1: Khi chương trình thực hiện yêu cầu vào/ra nói chung nó không sử dụng bộ nhớ trong suốt khoảng thời gian thực hiện thao tác vào/ra. Nói chung, trong hệ thống, tài nguyên được sử dụng nhiều nhất (năng động nhất) là bộ xử lý, bộ xử lý luôn phải phục vụ các tiến trình song song, chuyển từ tiến trình này sang tiến trình khác để tất cả tiến trình đều được tiếp tục với tốc độ chấp nhận được.

Nếu một tiến trình không sử dụng hợp lý bộ xử lý ví dụ khi đang thực hiện thao tác I/O, quyền sử dụng bộ xử lý của tiến trình này cần tạn thời ngăn cấm, trao cho các tiến trình khác. Do đó, việc tổ chức phân chia tài nguyên động là chỉ tiêu rất quan trọng để đảm bảo hoạt động hiệu quả của các hệ thống đa chương trình.

Các dạng tài nguyên xác định là "không phân chia" với ý nghĩa là chúng không thể tuỳ ý lấy lại từ tiến trình mà chúng được cấp.

Ví dụ 2: Băng từ, thông thường được cấp cho một tiến trình trong khoảng thời gian dài. Thiết bị kiểu này không thể đơn giản lấy lại từ tiến trình đó để cấp cho tiến trình khác.

Có những loại tài nguyên khác cho phép chia sẻ giữa một vài tiến trình, còn có những loại chỉ do một tiến trình độc quyền sử dụng.

Ví dụ 3: Ổ đĩa thường chứa các File thuộc nhiều tiến trình nhưng đôi khi do một tiến trình chiếm giữ. Bộ nhớ và bộ xử lý được sử dụng chia sẻ bởi nhiều tiến trình.

Dữ liệu và chương trình cũng là các tài nguyên và chúng cũng cần các cơ cấu điều khiển, cấp phát tương ứng.

Ví dụ 4: Trong một hệ, nhiều user có thể cùng cần chạy một chương trình. Bởi vì nếu mỗi user có một bản copy của chương trình trong bộ nhớ thì không tiết kiệm, do đó có thể chỉ nạp một bản copy của chương trình vào bộ nhớ còn mỗi user sẽ có một vùng dữ liệu riêng.

Code của chương trình mà không được thay đổi trong thời gian chạy gọi là Reentrant (hay có thể chạy nhiều lần đồng thời) còn Code của chương trình có thể thay đổi trong thời gian chạy gọi là Code sử dụng nối tiếp (Serial Reusable). Với Reentrant Code – có thể có một số tiến trình cùng làm việc còn với Code nối tiếp chỉ một tiến trình làm việc với nó tại một thời điểm. Khi nói về một tài nguyên nào đó, chúng ta cần hình dung xem chúng có thể được sử dụng bởi nhiều tiến trình đồng thời hay bởi nhiều tiến trình nhưng chỉ một tiến trình tại mỗi thời điểm. Chính loại tài nguyên thứ hai thường gây ra Deadlock.

2.3.2. Các điều kiện cần để có thể xảy ra tắc nghẽn

Coffman, Elphick và Sheshani đã phát biểu 4 điều kiện xuất hiện Deadlock:

- Các tiến trình yêu cầu quyền độc quyền sử dụng tài nguyên sẽ cấp phát cho nó (điều kiện loại trừ nhau).
- Tiến trình giữ cho mình các tài nguyên đã được cấp và đồng thời yêu cầu tài nguyên bổ sung (điều kiện chờ tài nguyên).
- Tài nguyên không được lấy lại từ tiến trình khi các tài nguyên đó chưa được sử dụng để kết thúc công việc (điều kiện không phân chia).
- Tồn tại vòng kín các tiến trình, trong đó mỗi tiến trình giữ tài nguyên mà tiến trình kế tiếp đang đòi hỏi (điều kiện chờ vòng).

2.3.3. Phương pháp giải quyết

Vấn đề Deadlock là một trong các vấn đề được nghiên cứu nhiều trong lĩnh vực công nghệ thông tin, các thành tựu trong lĩnh vực đó cho phép đề ra các thuật toán giải quyết nhiều bài toán. Các nghiên cứu có thể chia ra làm 4 hướng chính sau:

- Ngăn chặn Deadlock.
- Tránh Deadlock.
- Phát hiện Deadlock.
- Khôi phục sau Deadlock.

Hướng thứ 1: Ngăn chặn Deadlock: có mục đích tìm những điều kiện để loại trừ khả năng xuất hiện tình trạng Deadlock. Hướng này là giải pháp trực diện đối với vấn đề Deadlock, nhưng nó thường dẫn tới việc sử dụng tài nguyên không hiệu quả. Nhưng dù sao các phương pháp ngăn chặn Deadlock được áp dụng khá phổ biến.

Hướng thứ 2: Tránh Deadlock: Mục đích của các biện pháp tránh Deadlock là ở chỗ cho phép những điều kiện ít nghiêm ngặt hơn so với các biện pháp ngăn chặn Deadlock và cũng đảm bảo sử dụng tài nguyên tốt hơn. Các biện pháp tránh Deadlock không đòi hỏi loại bỏ hoàn toàn để không xảy ra tình trạng deadlock trong hệ thống. Ngược lại nó chú ý các khả năng xuất hiện Deadlock, trong trường hợp xác suất xuất hiện Deadlock tăng lên thì áp dụng các biện pháp tránh xảy ra Deadlock.

Hướng thứ 3: Phát hiện Deadlock: Các phương pháp phát hiện Deadlock áp dụng trong các hệ thống có khả năng xuất hiện Deadlock do hậu quả của các thao tác vô ý

hay cố ý. Mục đích của các biện pháp phát hiện là xác định sự tồn tại tình trạng Deadlock trong hệ thống, xác định các tiến trình và tài nguyên nằm trong tình trạng Deadlock. Sau đó có thể áp dụng các biện pháp thích hợp để khắc phục.

Hướng thứ 4: Khôi phục sau Deadlock: Các biện pháp khôi phục sau Deadlock áp dụng khi loại bỏ tình trạng Deadlock để hệ thống có thể tiếp tục hoạt động, còn các tiến trình rơi vào tình trạng Deadlock có thể phải kết thúc và các tài nguyên của nó được giải phóng. Sau đó các tiến trình đó thường được nạp và bắt đầu từ đầu (các công việc đã thực hiện đến lúc xảy ra Deadlock sẽ bị mất).

2.3.4. Ngăn chặn

Đến nay các nhà thiết kế khi giải quyết các vấn đề Deadlock thường đi theo hướng loại trừ những tình huống Deadlock hay xảy ra nhất. Chúng ta sẽ xem xét các phương pháp ngăn chặn Deadlock và đánh giá hậu quả của nó đối với user cũng như với hệ thống đặc biệt là về mặt hiệu quả công việc và tính năng sử dụng.

Havender đã chỉ ra rằng khả năng xuất hiện Deadlock không thể có ngay cả khi tồn tại một trong bốn điều kiện xuất hiện Deadlock. Havender đề ra các chiến lược sau:

- Mỗi tiến trình phải yêu cầu tất cả các tài nguyên nó cần ngay một lần và không thể bắt đầu cho đến khi chưa có đủ các tài nguyên đó.
- Nếu tiến trình đang có các tài nguyên nào đó, bị từ chối cấp phát tài nguyên mới (bổ sung) thì nó phải giải phóng các tài nguyên ban đầu (đã có) và trong trường hợp cần thiết phải yêu cầu cấp lại cùng với các tài nguyên bổ sung.
- Đưa vào các tài nguyên tuyến tính đối với tất cả tiến trình, tức là nếu tiến trình được cấp tài nguyên loại nào đó (phân loại theo thứ tự) thì sau đó nó chỉ có thể yêu cầu các tài nguyên loại có bậc lớn hơn.
- Chúng ta thấy rằng chỉ có ba chiến lược (chứ không phải bốn).
- Mỗi nguyên tắc có mục đích loại trừ (phá vỡ) một trong các điều kiện tồn tại của Deadlock. Điều kiện đầu tiên (điều kiện loại trừ nhau), theo đó tiến trình có độc quyền điều khiển tài nguyên được cấp cho nó, chúng ta không muốn loại trừ bởi vì chúng ta cần cho phép khả năng làm việc với tài nguyên đơn.

Nguyên tắc 1: Loại bỏ điều kiện "chờ tài nguyên bổ sung"

Nguyên tắc thứ nhất của Havender đòi hỏi tiến trình phải yêu cầu tất cả (toàn bộ) tài nguyên mà nó cần ngay từ đầu. Hệ thống phải cấp các tài nguyên này theo nguyên tắc "có tất cả hoặc không có gì". Nếu tập hợp các tài nguyên có đủ thì hệ thống có thể cấp tất cả để cho tiến trình có thể tiếp tục công việc. Nếu lúc đó không có đủ tài nguyên thì tiến trình phải chờ đến khi các tài nguyên có đủ (nhờ được các tiến trình khác giải phóng). Bởi vì tiến trình nằm trong trạng thái chờ không được giữ tài nguyên nào, do đó ngăn chặn được sự xuất hiện điều kiện "chờ tài nguyên bổ sung" và tình huống Deadlock không thể xảy ra.

Kết quả đó được trả giá bởi việc sử dụng tài nguyên không hiệu quả. Ví dụ, chương trình vào lúc nào đó cần 10 thiết bị băng từ, bắt buộc phải yêu cầu và có được đủ cả 10 thiết bị trước khi có thể bắt đầu. Nếu như 10 thiết bị đó cần suốt trong thời gian hoạt động thì không có vấn đề gì về hiệu suất sử dụng. Nhưng nói chung thì không phải tất cả chúng đều được sử dụng trong cả tiến trình tồn tại của tiến trình mà chúng chỉ cần trong những khoảng thời gian nào đó, như thế các thiết bị được sử dụng với hiệu suất rất thấp, không thể chấp nhận.

Một trong những cách để nâng cao hiệu suất là phân chia chương trình thành một vài giai đoạn tương đối độc lập với nhau. Do đó có thể thực hiện việc cấp phát tài nguyên cho từng giai đoạn chương trình thay vì cấp một lần cho cả chương trình. Điều này cho phép giảm việc lãng phí tài nguyên nhưng lại tăng chi phí khi thiết kế và cả khi thực hiện chương trình.

Chiến lược này làm tăng khả năng tiến trình phải chờ vô hạn (không xác định) bởi vì không phải tất cả tài nguyên cần thiết đều có vào lúc tiến trình cần. Như thế hệ thống phải thực hiện (đến khi kết thúc) số lượng khá lớn bài toán khác và giải phóng tài nguyên cấp cho chúng để bài toán đang chờ có thể hoạt động. Vào khoảng thời gian các tài nguyên cần có được thu thập thì chúng không thể cấp cho các Task khác, tức là lúc đó chúng không hoạt động.

Từ khía cạnh kinh tế thì các chi phí đó cần phải tính. Có nhiều ý kiến khác nhau về việc ai phải trả chi phí đó. Có người cho rằng vì khi đó các tài nguyên được thu thập cho người sử dụng nên người dùng phải trả tiền cả cho thời gian chúng dừng. Còn những người khác cho rằng điều đó không hợp lý vì lúc đó người dùng không thực sự sử dụng tài nguyên. Nếu người dùng muốn thực hiện chương trình vào giờ cao điểm thì anh ta sẽ phải trả nhiều hơn đáng kể so với trường hợp thực hiện vào thời gian khác.

Nguyên tắc 2: Loại bỏ điều kiện “không phân bố lại”

Tiêu chuẩn thứ hai của Havender (xem xét độc lập), ngăn chặn việc xuất hiện điều kiện "không phân bố lại". Giả sử rằng hệ thống cho phép các tiến trình yêu cầu thêm các tài nguyên bổ sung và vẫn giữ những tài nguyên đã được cấp. Nếu như hệ thống có đủ tài nguyên trống để phân phối thì không có tình trạng Deadlock. Nhưng nếu yêu cầu không được giải quyết thì có thể có tình huống một tiến trình chiếm giữ tài nguyên mà tiến trình khác yêu cầu để tiếp tục hoạt động, còn tiến trình thứ hai đó lại chiếm giữ tài nguyên mà tiến trình đầu cần tức là xuất hiện Deadlock.

Nguyên tắc Havender thứ hai đòi hỏi rằng nếu một tiến trình không được cấp phát tài nguyên bổ sung khi nó yêu cầu thì nó phải giải phóng tất cả tài nguyên nó đang chiếm giữ và trong trường hợp cần thiết phải yêu cầu lại tất cả cùng với tài nguyên bổ sung.

Nguyên tắc này thật sự loại trừ được yếu tố "không phân bố lại" và có thể lấy được tài nguyên từ các tiến trình đang chiếm chúng trước khi các tiến trình hoàn thành.

Nguyên tắc này cũng có nhiều nhược điểm. Nếu như tiến trình trong tiến trình làm việc sử dụng các tài nguyên mà sau đó giải phóng chúng thì nó có thể mất các kết quả làm việc đến lúc đó. Giá đó có vẻ quá lớn, tuy nhiên nó còn phụ thuộc vào tần số (xác suất) xảy ra. Nếu tình huống đó ít xảy ra thì có thể nói rằng tiêu chuẩn này không phải là quá đắt. Còn nếu như thường xuyên xảy ra thì giá của nó là quá đắt, đặc biệt với các tiến trình có mức ưu tiên cao hay khẩn cấp.

Một trong những nhược điểm lớn của biện pháp này là xác suất xảy ra 'chặn vô hạn' (Indefinite Posponement). Sự thực hiện tiến trình mà nhiều lần phải yêu cầu rồi giải phóng cùng một tài nguyên có thể bị dừng một thời hạn không xác định. Trong trường hợp đó hệ thống có thể phải loại tiến trình đó để các tiến trình khác có thể tiếp tục. Cũng không thể không tính khả năng khi tiến trình bị chặn vô hạn nhưng hệ thống không nhận ra, tình huống này làm lãng phí tài nguyên và giảm hiệu suất của cả hệ thống.

Nguyên tắc 3: Loại trừ điều kiện “chờ vòng quanh”

Nguyên tắc thứ ba của Havender loại trừ sự xuất hiện tình trạng chờ vòng. Vì tất cả các tài nguyên được gán một số duy nhất và các tiến trình phải yêu cầu các tài nguyên với

số tăng dần do đó không thể xuất hiện tình trạng 'chờ vòng quanh'. Nguyên tắc này được thực hiện trong nhiều hệ điều hành nhưng nó cũng gây ra các khó khăn nhất định.

Vì các tài nguyên được yêu cầu (cấp phát) theo thứ tự tăng dần và các số gán cho tài nguyên được gán từ đầu do đó trong trường hợp đưa thêm vào hệ thống các tài nguyên loại mới có thể gây ra tình huống phải thiết kế lại chương trình và cả hệ thống.

Rõ ràng rằng việc gán số cho tài nguyên cần thể hiện thứ tự thông thường mà phần lớn các Task sử dụng tài nguyên. Đối với các Task thực hiện theo thứ tự đó thì các tài nguyên có thể được sử dụng có hiệu quả. Còn nếu như Task lại yêu cầu tài nguyên theo thứ tự ngược lại thì nó sẽ chiếm giữ tài nguyên lâu hơn cần thiết (tính đến lúc tài nguyên thực sự được dùng).

Bởi vì chức năng quan trọng của hệ điều hành là cho phép người dùng sử dụng thuận tiện nhất. Người dùng phải có khả năng thiết kế chương trình với ít thứ không cần thiết nhất do sự hạn chế từ phía máy tính và hệ điều hành. Nguyên tắc thứ ba của Havender ngăn chặn được tình trạng 'chờ vòng' nhưng lại ảnh hưởng xấu đến công việc của user trong tiến trình làm việc (lập trình).

Ngăn chặn Deadlock và thuật toán Banker

Ngay cả khi tồn tại các điều kiện xuất hiện Deadlock thì vẫn có thể tránh tình trạng đó nếu như tài nguyên được cấp phát theo những qui tắc nhất định. Có lẽ thuật toán thông dụng nhất để tránh tình trạng Deadlock là thuật toán Banker do Dijstra đề xuất.

Thuật toán Banker của Dijstra:

- Khi xem xét thuật toán Banker chúng ta sẽ xem xét các tài nguyên và giới hạn xét các tài nguyên cùng một loại, nhưng thuật toán này có thể dễ dàng thay đổi để áp dụng cho các tài nguyên nhiều loại khác nhau.
- Chúng ta xem trường hợp ví dụ như phân chia t thiết bị lưu trữ băng từ.
- Hệ điều hành phải đảm bảo phân chia một số thiết bị t cho một số cố định người sử dụng. Mỗi người dùng sẽ phải báo trước số thiết bị lớn nhất mà anh ta sẽ cần khi thực hiện bài toán. Hệ điều hành sẽ chấp nhận yêu cầu của người dùng nếu yêu cầu cao nhất đó không vượt quá số thiết bị t.
- Người dùng có thể chiếm hay giải phóng từng thiết bị một. Có thể rằng anh ta sẽ phải chờ được cấp tài nguyên nhưng hệ điều hành đảm bảo rằng sự chờ đợi đó không phải là vô hạn. Số thiết bị cấp cho người dùng tại một thời điểm không bao giờ vượt quá số thiết bị nhiều nhất anh ta cần đến. Nếu hệ điều hành có đủ số thiết bị thoả mãn yêu cầu lớn nhất của người dùng thì người sử dụng đảm bảo rằng các thiết bị đó sẽ được sử dụng và trả lại cho hệ điều hành sau khoảng thời gian hữu hạn nào đó.
- Trạng thái hiện thời của máy tính gọi là ổn định nếu hệ điều hành có thể đảm bảo tất cả các chương trình ứng dụng hiện thời trong hệ thống có thể hoàn thành (kết thúc bình thường) sau một khoảng thời gian hữu hạn nào đó. Còn trong trường hợp ngược lại thì trạng thái là không ổn định.
- Giả sử rằng có n người sử dụng.
- Giả sử l(i) là số thiết bị cấp cho người sử dụng thứ i. Ví dụ người dùng thứ 5 đang dùng 4 thiết bị thì l(5) = 4.
- Giả sử m(i) là số thiết bị lớn nhất mà người dùng thứ i có thể cần ví dụ người dùng thứ 5 cần nhiều nhất 6 thiết bị thì m(5) = 6. Tại một thời điểm, c(i) là yêu cầu lớn

nhất hiện thời của người dùng i– bằng hiệu giữa số thiết bị nhiều nhất có thể yêu cầu và số thiết bị hiện có, tức là c(i) = m(i) – l(i) ví dụ ở trên ta có c(5) = m(5) – l(5) = 6 – 4 = 2.

- Trong hệ thống với t thiết bị thì số thiết bị còn rỗi tại một thời điểm là a sẽ bằng t trừ tổng các thiết bị được cấp phát: a = t – Σl(i).
- Thuật toán Banker của Dijkstra yêu cầu rằng các thiết bị chỉ được cấp phát cho người dùng theo yêu cầu trong trường hợp sau khi cấp phát thì hệ thống vẫn ở trạng thái ổn định. Trạng thái ổn định là trạng thái mà với các tài nguyên đang có, tất cả ứng dụng của người dùng có khả năng kết thúc (bình thường) công việc của mình. Còn trạng thái không ổn định là trạng thái có thể dẫn tới Deadlock.

Ví dụ trạng thái ổn định:

Giả sử hệ thống có 12 thiết bị và chúng được phân chia giữa 3 người dùng với trạng thái status1 được biểu diễn trong bảng sau:

Trạng thái status1

	Số thiết bị đang được cấp	Số thiết bị lớn nhất có thể cần
Người dùng 1	1	4
Người dùng 2	4	6
Người dùng 3	5	8
Dự trữ còn lại		2

Trạng thái đó là ổn định vì cả 3 người dùng có khả năng kết thúc công việc.

Ví dụ trạng thái không ổn định:

Giả sử hệ thống cũng có 12 thiết bị và chúng được phân chia giữa 3 người dùng với trạng thái status2 được biểu diễn trong bảng sau:

Trạng thái status2

	Số thiết bị đang được cấp	Số thiết bị lớn nhất có thể cần
Người dùng 1	8	10
Người dùng 2	2	5
Người dùng 3	1	3
Dự trữ còn lại		1

Trong trường hợp này 11 thiết bị trong trạng thái được cấp phát và chỉ còn 1 thiết bị dự trữ. Trạng thái này không ổn định vì bất cứ người dùng nào yêu cầu thêm một thiết bị và hệ thống đáp ứng thì chúng ta không thể đảm bảo các chương trình đều kết thúc bình thường.

Chúng ta cần chú ý rằng thuật ngữ 'trạng thái không ổn định' không có nghĩa là vào lúc đó hoặc thời điểm nào đó 'nhất định' sẽ xuất hiện tình trạng Deadlock, mà nó chỉ nói rằng trong trường hợp xấu, hệ thống 'có thể' rơi vào tình trạng Deadlock.

Ví dụ chuyển từ trạng thái ổn định sang không ổn định:

Nếu như trạng thái hiện thời của hệ thống là ổn định thì điều đó không có nghĩa là tất cả các trạng thái sau đó đều là ổn định. Cơ chế cấp phát cần phải phân tích các yêu cầu trước khi cấp phát tài nguyên.

Giả sử hệ thống đang ở trạng thái 3, rõ ràng trạng thái đó là ổn định. Người dùng thứ 3 yêu cầu tài nguyên bổ sung. Nếu như thoả mãn yêu cầu đó thì hệ thống chuyển sang trạng thái 4. Dễ thấy trạng thái 4 là trạng thái không ổn định.

Trạng thái 3

	Số thiết bị đang được cấp	Số thiết bị lớn nhất có thể cần
Người dùng 1	1	4
Người dùng 2	4	6
Người dùng 3	5	8
Dự trữ còn lại		2

Trạng thái 4

	Số thiết bị đang được cấp	Số thiết bị lớn nhất có thể cần
Người dùng 1	1	4
Người dùng 2	4	6
Người dùng 3	6	8
Dự trữ còn lại		1

Tất nhiên trạng thái 4 không nhất thiết dẫn tới tình trạng Deadlock. Tuy nhiên hệ thống đã chuyển từ trạng thái 3 (ổn định) sang trạng thái 4 (không ổn định).

Phân phối tài nguyên theo thuật toán Banker:

- Chúng ta sẽ xem xét sự phân phối tài nguyên theo Dijkstra được thực hiện thế nào. Các điều kiện 'loại trừ nhau', 'chờ tài nguyên bổ sung' và 'không phân chia lại' có thể xảy ra. Các tiến trình có thể được độc quyền sử dụng các tài nguyên cấp cho nó. Các tiến trình được quyền yêu cầu và chờ tài nguyên bổ sung trong khi vẫn giữ các tài nguyên đã được cấp, ngoài ra tài nguyên không bị lấy khỏi tiến trình. Người dùng không đặt ra cho hệ thống bài toán quá phức tạp khi tại mỗi thời điểm chỉ yêu cầu một tài nguyên. Hệ thống hoặc thoả mãn hoặc từ chối yêu cầu. Nếu yêu cầu bị từ chối thì user vẫn giữ các tài nguyên đã được cấp và chờ trong một khoảng thời gian (hữu hạn) nào đó đến khi được cấp.
- Hệ thống chỉ thoả mãn các yêu cầu mà sau đó trạng thái của hệ thống vẫn ổn định. Còn các yêu cầu có thể dẫn hệ thống tới trạng thái không ổn định thì bị hoãn một khoảng thời gian nào đó và cuối cùng vẫn sẽ được thoả mãn. Bởi thế, hệ thống luôn ở trạng thái ổn định, do đó tất cả các yêu cầu sẽ được thoả mãn và tất cả user có thể kết thúc công việc.

Những nhược điểm của thuật toán Banker:

Thuật toán Banker có nhiều ưu điểm bởi vì nó cho phép cấp phát tài nguyên, tránh tình trạng Deadlock. Nó cho phép tiếp tục thực hiện các tiến trình mà trong trường hợp dùng các biện pháp ngăn chặn thì chúng đã bị dừng. Nhưng thuật toán Banker cũng vẫn có những nhược điểm mà do đó các nhà thiết kế hệ thống có thể phải lựa chọn các cách khác nhau để giải quyết vấn đề Deadlock:

- Thuật toán Banker xuất phát từ giả thiết số tài nguyên là cố định. Nhưng bởi vì các tài nguyên không thể làm việc mãi (ví dụ dừng lại để bảo dưỡng) do đó chúng ta không thể cho rằng số lượng tài nguyên là cố định.

- Thuật toán đòi hỏi rằng số người dùng là không đổi. Yêu cầu đó cũng không thực tế, vì trong các hệ đa chương trình, số lượng người dùng luôn thay đổi
- Thuật toán đòi hỏi bộ phận phân phối tài nguyên phải đảm bảo thoả mãn tất cả các yêu cầu sau khoảng thời gian hữu hạn nào đó. Tuy nhiên trong thực tế người ta cần những con số cụ thể hơn nhiều.
- Cũng như thế, thuật toán đòi hỏi người dùng phải trả lại các tài nguyên được cấp, sau một khoảng thời gian nào đó– và trong thực tế cũng cần các chỉ số cụ thể.
- Thuật toán yêu cầu người dùng phải báo trước số lượng lớn nhất tài nguyên anh ta cần. Nhưng sự phân phối tài nguyên ngày càng phải linh động và do đó càng khó đánh giá yêu cầu lớn nhất. Vì máy tính ngày càng thân thiện với người dùng nên sẽ ngày càng nhiều người dùng không có hình dung chính xác về số tài nguyên lớn nhất mà anh ta sẽ cần, thay vào đó khi nào cần tài nguyên người dùng mới yêu cầu.

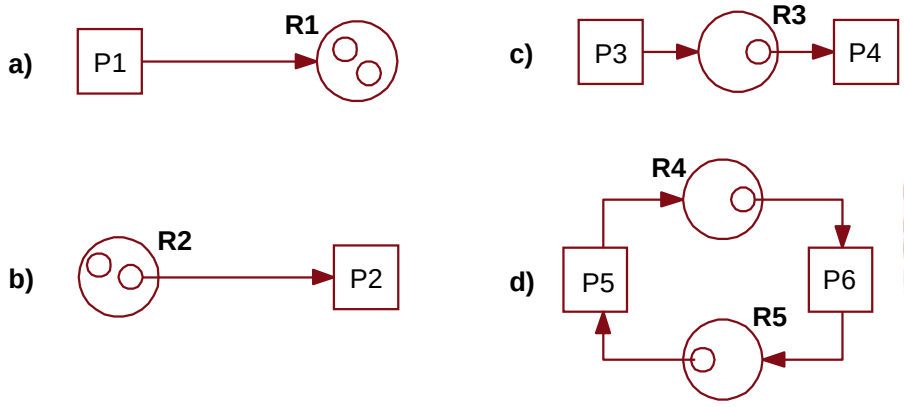
2.3.5. Phát hiện tắc nghẽn

Phát hiện Deadlock – là xác định sự kiện xuất hiện trạng thái Deadlock, xác định các tiến trình và tài nguyên nằm trong tình trạng Deadlock. Các thuật toán xác định Deadlock thường được áp dụng trong các hệ thống có xuất hiện ba điều kiện đầu tiên trong số các điều kiện làm xuất hiện Deadlock và sau đó mới xác định xem có tồn tại trạng thái 'chờ vòng' hay không.

Tất nhiên sử dụng các thuật toán phát hiện Deadlock cũng phải trả giá, đó là chi phí về thời gian máy và chúng ta lại gặp vấn đề phải xác định giải pháp trung hoà: các chi phí thực hiện thuật toán phát hiện Deadlock có tiết kiệm hơn hẳn so với khi dùng các biện pháp cô lập, loại bỏ Deadlock hay không.

Graph phân bố tài nguyên:

Trong các thuật toán phát hiện Deadlock, thường áp dụng cách biểu diễn phân bố tài nguyên dưới dạng đồ thị có hướng. Các hình vuông biểu diễn tiến trình, còn các hình tròn lớn biểu diễn các lớp tài nguyên. Các vòng tròn nhỏ biểu diễn số lượng tài nguyên của lớp đó.



Hình 2.3.5–1: ***các quan hệ trên đồ thị giữa các tiến trình và tài nguyên***

Ví dụ trong vòng tròn lớn R1 có 2 vòng nhỏ tức là hệ thống có 2 tài nguyên thuộc lớp R1

Trong hình 2.3.5–1a tiến trình P1 đang yêu cầu tài nguyên lớp R1. Mũi tên từ P1 đến vòng tròn lớn có nghĩa rằng hiện thời yêu cầu của tiến trình đang được xem xét. Hình 2.3.5–1b thể hiện rằng tiến trình P2 được cấp 1 tài nguyên lớp R2, ở đây mũi tên đi từ vòng tròn nhỏ nằm trong vòng tròn lớn R2 đến tiến trình P2.

Trên hình.2.3.5–1c biểu diễn tình huống ở một ý nghĩa nào đó gần với tình trạng Deadlock: tiến trình P3 yêu cầu tài nguyên lớp R3 đang được cấp cho tiến trình P4.

Hình 2.3.5–1d biểu diễn tình trạng Deadlock: tiến trình P5 yêu cầu tài nguyên lớp R4, trong khi tất cả tài nguyên lớp đó được cấp cho tiến trình P6 và ngược lại tiến trình P6 yêu cầu tài nguyên lớp R5 – đang được cấp cho P5 – tình trạng chờ vòng này là tình trạng Deadlock hay gặp.

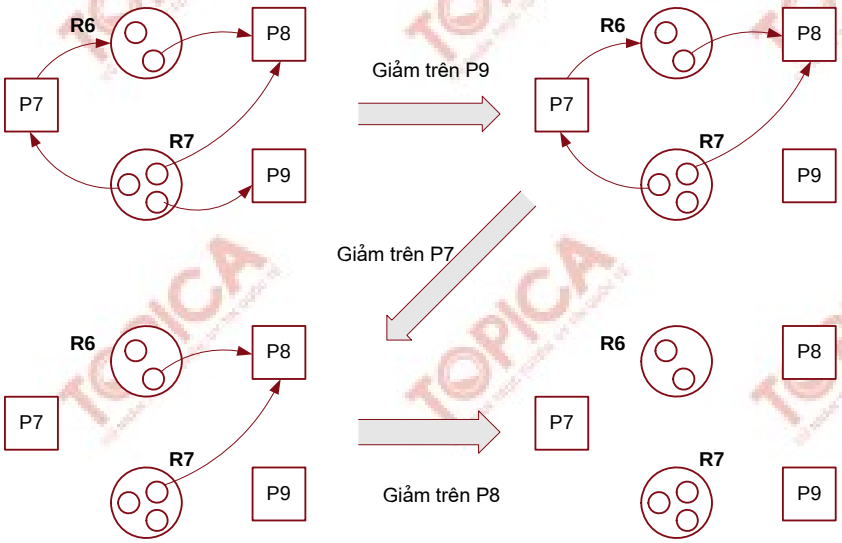
Đồ thị phân bố và yêu cầu cấp phát tài nguyên luôn thay đổi do tiến trình các tiến trình yêu cầu tài nguyên, nhận tài nguyên và sau đó trả lại cho hệ điều hành.

Sử dụng đồ thị phân bố tài nguyên

Cơ chế phát hiện Deadlock – xác định xem có xuất hiện tình trạng Deadlock hay không. Một trong những phương pháp phát hiện là rút gọn đồ thị phân bố tài nguyên (Reduction of a Resource Allocation Graph), nó cho phép xác định các tiến trình có thể kết thúc và các tiến trình có mặt trong tình huống Deadlock.

Nếu như yêu cầu tài nguyên của một tiến trình nào đó có thể thoả mãn thì chúng ta nói rằng graph có thể rút gọn đối với tiến trình đó. Sự rút gọn đó tương đương với sự biểu diễn graph dưới dạng mà nó có khi tiến trình kết thúc công việc và trả lại các tài nguyên cho hệ thống. Sự rút gọn graph đối với một tiến trình nào đó được thể hiện bằng việc ngắt bỏ các cung đi tới tiến trình đó từ các tài nguyên (tức là tài nguyên đã được cấp cho tiến trình) và các cung từ tiến trình đến tài nguyên (tức là các yêu cầu cấp phát hiện thời của tiến trình). Nếu như graph có thể rút gọn thành tất cả tiến trình cô lập có nghĩa là không có tình trạng Deadlock, còn nếu không thể làm được thì các tiến trình không rút gọn được là các tiến trình nằm trong trạng thái Deadlock.

Trên hình 2.3.5–2 là dãy thao tác rút gọn graph, kết quả cho ta thấy đối với trường hợp này thì không xuất hiện trạng thái Deadlock. Chúng ta cần để ý rằng thứ tự thực hiện thao tác là không quan trọng. Dù ta rút gọn theo thứ tự nào thì kết quả cuối cùng vẫn là một.



Hình 2.3.5–2

Khôi phục sau Deadlock

Hệ thống nằm trong trạng thái Deadlock cần thoát ra khỏi tình trạng đó bằng cách loại bỏ các điều kiện tồn tại Deadlock. Thông thường thì một số tiến trình sẽ mất một phần

hay toàn bộ kết quả đã thực hiện được. Nhưng giá của nó nhỏ hơn nhiều khi hệ thống bị treo hoàn toàn. Mức độ phức tạp của việc khôi phục hệ thống phụ thuộc nhiều yếu tố.

Ngay từ đầu việc xác định tình trạng Deadlock không phải luôn rõ ràng.

Trong nhiều hệ thống không có các công cụ đủ hiệu quả để dừng tiến trình một thời hạn không xác định, đưa nó ra khỏi hệ thống và lại khởi động nó sau đó. Trong thực tế thì có một số tiến trình phải làm việc liên tục không cho phép dừng lại và khởi động.

Ngay cả khi hệ thống có các công cụ hữu hiệu để dừng/khởi động tiến trình thì việc sử dụng chúng cũng đòi hỏi chi phí đáng kể về thời gian máy,...

Khôi phục tình trạng Deadlock ở quy mô nhỏ (có ít tiến trình nằm trong tình trạng đó) thường đòi hỏi ít công sức nhưng khi có nhiều tiến trình nằm trong tình trạng Deadlock thì có thể đòi hỏi một chi phí lớn.

Trong các hệ thống hiện nay việc khôi phục thường được thực hiện bằng cách loại một số tiến trình để có thể sử dụng các tài nguyên của chúng. Kết quả của các tiến trình bị huỷ thường bị mất nhưng nhờ đó các tiến trình còn lại có thể tiếp tục, kết thúc công việc của mình. Thuật ngữ 'khôi phục' cũng không chính xác lắm, ý nghĩa của nó là khôi phục khả năng làm việc bình thường của hệ thống.

Việc lựa chọn tiến trình để loại ra khỏi hệ thống cần phải theo một chỉ tiêu nào đó, ví dụ mức độ ưu tiên của tiến trình. Nhưng cũng có một số khó khăn.

Tiến trình nằm trong tình trạng Deadlock có thể không còn giá trị ưu tiên.

Độ ưu tiên của tiến trình có thể không đúng do nguyên nhân nào đó ví dụ độ ưu tiên của tiến trình có thể tăng lên theo thời gian nó phải chờ.

Có lẽ phương pháp khôi phục tốt nhất là cơ chế tạm dừng/khởi động tiến trình (Suspend/Activate). Cơ chế này cho phép chúng ta chuyển tiến trình vào trạng thái chờ, sau đó nó sẽ được khởi động, các kết quả làm việc vẫn giữ nguyên không bị mất.

Deadlock có thể dẫn tới những hậu quả lớn trong các hệ thống thời gian thực. Các hệ thống đó phải hoạt động liên tục do đó không cho phép xảy ra tình trạng Deadlock. Nhưng không thể đảm bảo tuyệt đối là nó không xảy ra – đó vẫn là một vấn đề cần giải quyết.

TÓM LƯỢC CUỐI BÀI

- Các trạng thái của tiến trình, quá trình biến đổi các trạng thái đó:
  - Dispatch (Process Name): Ready ⇒ Running.
  - Interval Gone (Process Name): Running ⇒ Ready.
  - Blocking (Process Name): Running ⇒ Blocked.
  - Waitup (Process Name): Blocked ⇒ Ready.
- Chuyển đổi ngữ cảnh, tiến trình đồng thời.
- Deadlock:
  - Phát hiện deadlock.
  - Điều kiện loại trừ nhau.
  - Điều kiện chờ tài nguyên.
  - Điều kiện không phân chia.
  - Điều kiện chờ vòng.
- Ngăn chặn deadlock.
- Khôi phục sau deadlock,….

CÂU HỎI TỰ LUẬN

Câu 1. Trình bày rõ về quá trình biến đổi trạng thái?

Câu 2. Nêu các dạng ngắt trong các hệ thống máy lớn của IBM?

Câu 3. Nêu thuật toán Dekker Version 4? Ưu điểm của thuật toán này so với các thuật toán trước đó?

Câu 4. Thuật toán Banker ngăn chặn Deadlock?

Câu 5. Nêu ví dụ về trạng thái ổn định, quá trình chuyển từ trạng thái ổn định sang không ổn định?

BÀI TẬP TRẮC NGHIỆM

1. Tiến trình là gì?

a) Một chương trình lưu trên đĩa.

b) Một chương trình được nạp vào bộ nhớ.

c) Một chương trình nạp vào bộ nhớ và được CPU thực thi.

d) Tất cả các đáp án đều sai.

2. Việc chuyển trạnh thái nào của tiến trình không thể thực hiện?

a) Blocked =>Running. b) Ready =>Running. c) Blocked =>Ready. d) Running =>Blocked.

3. Độ ưu tiên của các tiến trình cho biết:

a) Tiến trình sử dụng CPU nhiều hay ít.

b) Tiến trình chiếm nhiều hay ít vùng nhớ.

c) Tầm quan trọng của tiến trình.

d) Tất cả các đáp án đều đúng.

4. …… không phải là trạng thái của tiến trình?

a) Blocked. b) Privileged. c) Ready. d) Running.

5. Điều kiện để có thể xảy ra Deadlock?

a) Các tiến trình yêu cầu quyền độc quyền sử dụng tài nguyên sẽ cấp phát cho nó.

b) Tiến trình giữ cho mình các tài nguyên đã được cấp và đồng thời yêu cầu tài nguyên bổ sung.

c) Tồn tại vòng kín các tiến trình, trong đó mỗi tiến trình giữ tài nguyên mà tiến trình kế tiếp đang đòi hỏi.

d) Tất cả các đáp án đều đúng.

6. Trạng thái BLOCKED của một tiến trình là do:

a) Đang chờ nhập xuất.

b) Đang chờ một sự kiện nào đó chưa xảy ra.

c) Cả 2 đáp án trên đều đúng.

d) Không có đáp án nào đúng.

7. Để cài đặt loại trừ sử dụng thuật toán

a) Dekker. b) Dijkstra. c) Banker. d) Havender.

8. Hàng đợi dành cho các tiến trình xếp hàng chờ bộ xử lý được gọi là?

a) Busy–Waitting buffer. b) Ready queue. c) Waitting queue. d) Suspended queue.

9. Khi một tiến trình chuẩn bị vào hay ra khỏi một vùng Critical Section thì phải?

a) Xin phép hệ điều hành.

b) Phất cờ hiệu khi vào và trả khi ra.

c) Cả hai đáp án trên là đúng.

d) Không có đáp án nào đúng.