# PHẦN 3: QUẢN LÝ TIẾN TRÌNH

Trong phần này, chúng em sẽ đi vào tìm hiểu cách thức tổ chức và điều phối các tiến trình trong máy tính, vấn đề đồng bộ hóa giữa các tiến trình cũng như các giải pháp xử lý vấn đề “tắc nghẽn”.

1. TỔ CHỨC QUẢN LÝ TIẾN TRÌNH

### Mô hình tiến trình

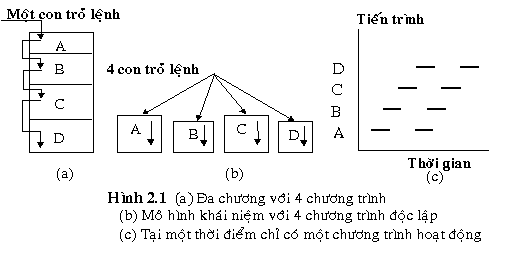


Để hỗ trợ sự đa chương, máy tính phải có khả năng thực hiện nhiều tác vụ đồng thời. Nhưng việc điều khiển nhiều hoạt động song song ở cấp độ phần cứng là rất khó khăn. Vì thế các nhà thiết kế hệ điều hành đề xuất một mô hình song song giả lập bằng cách chuyển đổi bộ xử lý qua lại giữa các chương trình để duy trì hoạt động của nhiều chương trình cùng lúc, điều này tạo cảm giác có nhiều hoạt động được thực hiện đồng thời.

Trong mô hình này, tất cả các phần mềm trong hệ thống được tổ chức thành một số những tiến trình (process). Tiến trình là một chương trình đang xử lý, sỡ hữu một con trỏ lệnh, tập các thanh ghi và các biến. Để hoàn thành tác vụ của mình, một tiến trình có thể cần đến một số tài nguyên – như CPU, bộ nhớ chính, các tập tin và thiết bị nhập/xuất.

Cần phân biệt hai khái niệm chương trình và tiến trình. Một chương trình là một thực thể thụ động, chứa đựng các chỉ thị điều khiển máy tính để tiến hành một tác vụ nào đó ; khi cho thực hiện các chỉ thị này, chương trình chuyển thành tiến trình, là một thực thể hoạt động, với con trỏ lệnh xác định chỉ thị kế tiếp sẽ thi hành, kèm theo tập các tài nguyên phục vụ cho hoạt động của tiến trình.

Về mặt ý niệm, có thể xem như mỗi tiến trình sỡ hữu một bộ xử lý ảo cho riêng nó, nhưng trong thực tế, chỉ có một bộ xử lý thật sự được chuyển đổi qua lại giữa các tiến trình. Sự chuyển đổi nhanh chóng này được gọi là sự đa chương (multiprogramming) . Hệ điều hành chịu trách nhiệm sử dụng một thuật toán điều phối để quyết định thời điểm cần dừng hoạt động của tiến trình đang xử lý để phục vụ một tiến trình khác, và lựa chọn tiến trình tiếp theo sẽ được phục vụ. Bộ phận thực hiện chức năng này của hệ điều hành được gọi là bộ điều phối (scheduler).



### Các trạng thái của tiến trình



Trạng thái của tiến trình tại một thời điểm được xác định bởi hoạt động hiện thời của tiến trình tại thời điểm đó. Trong quá trình sống, một tiến trình thay đổi trạng thái do nhiều nguyên nhân như : phải chờ một sự kiện nào đó xảy ra, hay đợi một thao tác nhập/xuất hoàn tất, buộc phải dừng hoạt động do đã hết thời gian xử lý …

Tại một thời điểm, một tiến trình có thể nhận trong một các trạng thái sau đây :

**Mới tạo** : tiến trình đang được tạo lập.



**Running** : các chỉ thị của tiến trình đang được xử lý.



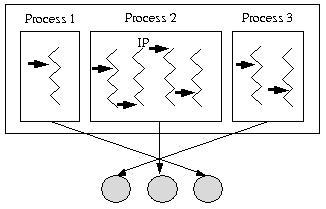
**Blocked** : tiến trình chờ được cấp phát một tài nguyên, hay chờ một  
sự kiện xảy ra .



**Ready** : tiến trình chờ được cấp phát CPU để xử lý.



**Kết thúc** : tiến trình hoàn tất xử lý.



**Hình 2.2** Sơ đồ chuyển trạng thái giữa các tiến trình

Tại một thời điểm, chỉ có một tiến trình có thể nhận trạng thái running trên một bộ xử lý bất kỳ. Trong khi đó, nhiều tiến trình có thể ở trạng thái blocked hay ready.

Các cung chuyển tiếp trong sơ đồ trạng thái biễu diễn sáu sự chuyển trạng thái có thể xảy ra trong các điều kiện sau :

Tiến trình mới tạo được đưa vào hệ thống



Bộ điều phối cấp phát cho tiến trình một khoảng thời gian sử dụng CPU



Tiến trình kết thúc



Tiến trình yêu cầu một tài nguyên nhưng chưa được đáp ứng vì tài nguyên chưa sẵn sàng để cấp phát tại thời điểm đó ; hoặc tiến trình phải chờ một sự kiện hay thao tác nhập/xuất.



Bộ điều phối chọn một tiến trình khác để cho xử lý .



Tài nguyên mà tiến trình yêu cầu trở nên sẵn sàng để cấp phát ; hay sự kiện hoặc thao tác nhập/xuất tiến trình đang đợi hoàn tất.



Chế độ xử lý của tiến trình



Để đảm bảo hệ thống hoạt động đúng đắn, hệ điều hành cần phải được bảo vệ khỏi sự xâm phạm của các tiến trình. Bản thân các tiến trình và dữ liệu cũng cần được bảo vệ để tránh các ảnh hưởng sai lạc lẫn nhau. Một cách tiếp cận để giải quyết vấn đề là phân biệt hai chế độ xử lý cho các tiến trình : chế độ không đặc quyền và chế độ đặc quyền nhờ vào sự trợ giúp của cơ chế phần cứng. Tập lệnh của CPU được phân chia thành các lệnh đặc quyền và lệnh không đặc quyền. Cơ chế phần cứng chỉ cho phép các lệnh đặc quyền được thực hiện trong chế độ đặc quyền. Thông thường chỉ có hệ điều hành hoạt động trong chế độ đặc quyền, các tiến trình của người dùng hoạt động trong chế độ không đặc quyền, không thực hiện được các lệnh đặc quyền có nguy cơ ảnh hưởng đến hệ thống. Như vậy hệ điều hành được bảo vệ. Khi một tiến trình người dùng gọi đến một lời gọi hệ thống, tiến trình của hệ điều hành xử lý lời gọi này sẽ hoạt động trong chế độ đặc quyền, sau khi hoàn tất thì trả quyền điều khiển về cho tiến trình người dùng trong chế độ không đặc quyền.



**Hình vẽ 2.3** Hai chế độ xử lý

Cấu trúc dữ liệu khối quản lý tiến trình



Hệ điều hành quản lý các tiến trình trong hệ thống thông qua khối quản lý tiến trình (process control block -PCB). PCB là một vùng nhớ lưu trữ các thông tin mô tả cho tiến trình, với các thành phần chủ yếu bao gồm :

**Định danh của tiến trình** **(1)** : giúp phân biệt các tiến trình



**Trạng thái tiến trình (2)**: xác định hoạt động hiện hành của tiến trình.



**Ngữ cảnh của tiến trình** **(3)**: mô tả các tài nguyên tiến trình đang trong quá trình, hoặc để phục vụ cho hoạt động hiện tại, hoặc để làm cơ sở phục hồi hoạt động cho tiến trình, bao gồm các thông tin về:



Trạng thái CPU: bao gồm nội dung các thanh ghi, quan trọng nhất là con trỏ lệnh IP lưu trữ địa chỉ câu lệnh kế tiếp tiến trình sẽ xử lý. Các thông tin này cần được lưu trữ khi xảy ra một ngắt, nhằm có thể cho phép phục hồi hoạt động của tiến trình đúng như trước khi bị ngắt.



Bộ xử lý: dùng cho máy có cấu hình nhiều CPU, xác định số hiệu CPU mà tiến trình đang sử dụng.



Bộ nhớ chính: danh sách các khối nhớ được cấp cho tiến trình.



Tài nguyên sử dụng: danh sách các tài mguyên hệ thống mà tiến trình đang sử dụng.



Tài nguyên tạo lập: danh sách các tài nguyên được tiến trình tạo lập.



**Thông tin giao tiếp (4)**: phản ánh các thông tin về quan hệ của tiến trình với các tiến trình khác trong hệ thống :



Tiến trình cha: tiến trình tạo lập tiến trình này .



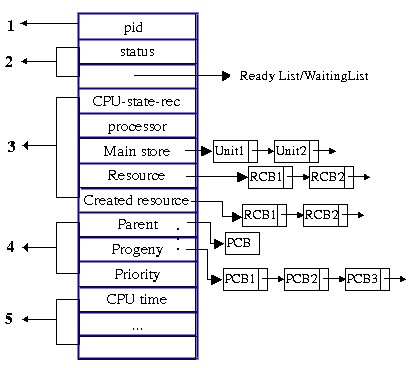
Tiến trình con: các tiến trình do tiến trình này tạo lập .



Độ ưu tiên : giúp bộ điều phối có thông tin để lựa chọn tiến trình được cấp CPU.



**Thông tin thống kê (5)**: đây là những thông tin thống kê về hoạt động của tiến trình, như thời gian đã sử dụng CPU,thời gian chờ. Các thông tin này có thể có ích cho công việc đánh giá tình hình hệ thống và dự đoán các tình huống tương lai.



**Hình vẽ 2.4** Khối mô tả tiến trình

Thao tác trên tiến trình



Hệ điều hành cung cấp các thao tác chủ yếu sau đây trên một tiến trình :

tạo lập tiến trình (create)



kết thúc tiến trình (destroy)



tạm dừng tiến trình (suspend)



tái kích hoạt tiến trình (resume)



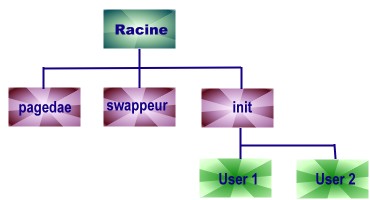
thay đổi độ ưu tiên tiến trình



**Tạo lập tiến trình**



Trong quá trình xử lý, một tiến trình có thể tạo lập nhiều tiến trình mới bằng cách sử dụng một lời gọi hệ thống tương ứng. Tiến trình gọi lời gọi hệ thống để tạo tiến trình mới sẽ được gọi là tiến trình cha, tiến trình được tạo gọi là tiến trình con. Mỗi tiến trình con đến lượt nó lại có thể tạo các tiến trình mới…quá trình này tiếp tục sẽ tạo ra một cây tiến trình.



**Hình vẽ 2.5** Một cây tiến trình trong hệ thống UNIX

Các công việc hệ điều hành cần thực hiện khi tạo lập tiến trình bao gồm :

định danh cho tiến trình mới phát sinh



đưa tiến trình vào danh sách quản lý của hệ thống



xác định độ ưu tiên cho tiến trình



tạo PCB cho tiến trình



cấp phát các tài nguyên ban đầu cho tiến trình



Khi một tiến trình tạo lập một tiến trình con, tiến trình con có thể sẽ được hệ điều hành trực tiếp cấp phát tài nguyên hoặc được tiến trình cha cho thừa hưởng một số tài nguyên ban đầu.

Khi một tiến trình tạo tiến trình mới, tiến trình ban đầu có thể xử lý theo một trong hai khả năng sau  :

Tiến trình cha tiếp tục xử lý đồng hành với tiến trình con.



Tiến trình cha chờ đến khi một tiến trình con nào đó, hoặc tất cả các tiến trình con kết thúc xử lý.



Các hệ điều hành khác nhau có thể chọn lựa các cài đặt khác nhau để thực hiện thao tác tạo lập một tiến trình.

**Kết thúc tiến trình**



Một tiến trình kết thúc xử lý khi nó hoàn tất chỉ thị cuối cùng và sử dụng một lời gọi hệ thống để yêu cầu hệ điều hành hủy bỏ nó. Đôi khi một tiến trình có thể yêu cầu hệ điều hành kết thúc xử lý của một tiến trình khác. Khi một tiến trình kết thúc, hệ điều hành thực hiện các công việc :

thu hồi các tài nguyên hệ thống đã cấp phát cho tiến trình



hủy tiến trình khỏi tất cả các danh sách quản lý của hệ thống



hủy bỏ PCB của tiến trình



Hầu hết các hệ điều hành không cho phép các tiến trình con tiếp tục tồn tại nếu tiến trình cha đã kết thúc. Trong những hệ thống như thế, hệ điều hành sẽ tự động phát sinh một loạt các thao tác kết thúc tiến trình con.

Cấp phát tài nguyên cho tiến trình



Khi có nhiều người sử dụng đồng thời làm việc trong hệ thống, hệ điều hành cần phải cấp phát các tài nguyên theo yêu cầu cho mỗi người sử dụng. Do tài nguyên hệ thống thường rất giới hạn và có khi không thể chia sẻ, nên hiếm khi tất cả các yêu cầu tài nguyên đồng thời đều được thỏa mãn. Vì thế cần phải nghiên cứu một phương pháp để chia sẻ một số tài nguyên hữu hạn giữa nhiều tiến trình người dùng đồng thời. Hệ điều hành quản lý nhiều loại tài nguyên khác nhau (CPU, bộ nhớ chính, các thiết bị ngoại vi …), với mỗi loại cần có một cơ chế cấp phát và các chiến lược cấp phát hiệu qủa. Mỗi tài nguyên được biễu diễn thông qua một cấu trúc dữ liệu, khác nhau về chi tiết cho từng loại tài nguyên, nhưng cơ bản chứa đựng các thông tin sau :

**Định danh tài nguyên**



**Trạng thái tài nguyên** : đây là các thông tin mô tả chi tiết trạng thái tài nguyên : phần nào của tài nguyên đã cấp phát cho tiến trình, phần nào còn có thể sử dụng ?



**Hàng đợi trên một tài nguyên** : danh sách các tiến trình đang chờ được cấp phát tài nguyên tương ứng.



**Bộ cấp phát**: là đoạn code đảm nhiệm việc cấp phát một tài nguyên đặc thù. Một số tài nguyên đòi hỏi các giải thuật đặc biệt (như CPU, bộ nhớ chính, hệ thống tập tin), trong khi những tài nguyên khác (như các thiết bị nhập/xuất) có thể cần các giải thuật cấp phát và giải phóng tổng quát hơn.



**Hình 2.6** Khối quản lý tài nguyên

Các mục tiêu của kỹ thuật cấp phát :

Bảo đảm một số lượng hợp lệ các tiến trình truy xuất đồng thời đến các tài nguyên không chia sẻ được.



Cấp phát tài nguyên cho tiến trình có yêu cầu trong một khoảng thời gian trì hoãn có thể chấp nhận được.



Tối ưu hóa sự sử dụng tài nguyên.



Để có thể thỏa mãn các mục tiêu kể trên, cần phải giải quyết các vấn đề nảy sinh khi có nhiều tiến trình đồng thời yêu cầu một tài nguyên không thể chia sẻ.

1. TIỂU TRÌNH (THREAD)

Trong hầu hết các hệ điều hành, mỗi tiến trình có một không gian địa chỉ và chỉ có một dòng xử lý. Tuy nhiên, có nhiều tình huống người sử dụng mong muốn có nhiều dòng xử lý cùng chia sẻ một không gian địa chỉ, và các dòng xử lý này hoạt động song song tương tự như các tiến trình phân biệt (ngoại trừ việc chia sẻ không gian địa chỉ).

**Ví dụ**: Một server quản lý tập tin thỉnh thoảng phải tự khóa để chờ các thao tác truy xuất đĩa hoàn tất.Nếu server có nhiều dòng xử lý, hệ thống có thể xử lý các yêu cầu mới trong khi một dòng xử lý bị khoá. Như vậy việc thực hiện chương trình sẽ có hiệu quả hơn. Điều này không thể đạt được bằng cách tạo hai tiến trình server riêng biệt vì cần phải chia sẻ cùng một vùng đệm, do vậy bắt buộc phải chia sẻ không gian địa chỉ.

Chính vì các tình huống tương tự, người ta cần có một cơ chế xử lý mới cho phép có nhiều dòng xử lý trong cùng một tiến trình.

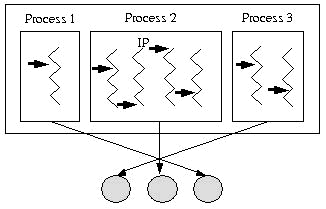
Ngày nay đã có nhiều hệ điều hành cung cấp một cơ chế như thế và gọi là tiểu trình (threads).

**Nguyên lý chung :**



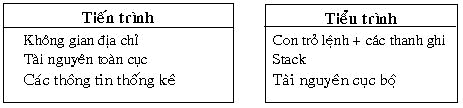
Một tiểu trình là một đơn vị xử lý cơ bản trong hệ thống . Mỗi tiểu trình xử lý tuần tự đoạn code của nó, sỡ hữu một con trỏ lệnh, tập các thanh ghi và một vùng nhớ stack riêng. Các tiểu trình chia sẻ CPU với nhau giống như cách chia sẻ giữa các tiến trình: một tiểu trình xử lý trong khi các tiểu trình khác chờ đến lượtù. Một tiểu trình cũng có thể tạo lập các tiến trình con, và nhận các trạng thái khác nhau như một tiến trình thật sự. Một tiến trình có thể sỡ hữu nhiều tiểu trình.

Các tiến trình tạo thành những thực thể độc lập. Mỗi tiến trình có một tập tài nguyên và một môi trường riêng (một con trỏ lệnh, một Stack , các thanh ghi và không gian địa chỉ ). Các tiến trình hoàn toàn độc lập với nhau, chỉ có thể liên lạc thông qua các cơ chế thông tin giữa các tiến trình mà hệ điều hành cung cấp. Ngược lại, các tiểu trình trong cùng một tiến trình lại chia sẻ một không gian địa chỉ chung , điều này có nghĩa là các tiểu trình có thể chia sẻ các biến toàn cục của tiến trình. Một tiểu trình có thể truy xuất đến cả các stack của những tiểu trình khác trong cùng tiến trình. Cấu trúc này không đề nghị một cơ chế bảo vệ nào, và điều này cũng không thật cần thiết vì các tiểu trình trong cùng một tiến trình thuộc về cùng một sỡ hữu chủ đã tạo ra chúng trong ý định cho phép chúng hợp tác với nhau.



Các tiểu trình trong cùng một tiểu trình

**Phân bổ thông tin lưu trữ**



Cấu trúc mô tả tiến trình và tiểu trình

1. ĐIỀU PHỐI TIẾN TRÌNH

Trong môi trường đa chương, có thể xảy ra tình huống nhiều tiến trình đồng thời sẵn sàng để xử lý. Mục tiêu của các hệ phân chia thời gian (time-sharing) là chuyển đổi CPU qua lại giữa các tiến trình một cách thường xuyên để nhiều người sử dụng có thể tương tác cùng lúc với từng chương trình trong quá trình xử lý.

Để thực hiện được mục tiêu này, hệ điều hành phải lựa chọn tiến trình được xử lý tiếp theo. Bộ điều phối sẽ sử dụng một giải thuật điều phối thích hợp để thực hiện nhiệm vụ này. Một thành phần khác của hệ điều hành cũng tiềm ẩn trong công tác điều phối là bộ phân phối (dispatcher). Bộ phân phối sẽ chịu trách nhiệm chuyển đổi ngữ cảnh và trao CPU cho tiến trình được chọn bởi bộ điều phối để xử lý.

Giới thiệu:



**Mục tiêu điều phối**



Bộ điều phối không cung cấp cơ chế, mà đưa ra các quyết định. Các hệ điều hành xây dựng nhiều chiến lược khác nhau để thực hiện việc điều phối, nhưng tựu chung cần đạt được các mục tiêu sau :

**a) Sự công bằng ( Fairness) :**Các tiến trình chia sẻ CPU một cách công bằng, không có tiến trình nào phải chờ đợi vô hạn để được cấp phát CPU

**b) Tính hiệu qủa (Efficiency) :**Hệ thống phải tận dụng được CPU 100% thời gian.

**c) Thời gian đáp ứng hợp lý (Response time) :**Cực tiểu hoá thời gian hồi đáp cho các tương tác của người sử dụng

**d) Thời gian lưu lại trong hệ thống ( Turnaround Time) :**Cực tiểu hóa thời gian hoàn tất các tác vụ xử lý theo lô.

**e) Thông lượng tối đa (Throughput ) :**Cực đại hóa số công việc được xử lý trong một đơn vị thời gian.

Tuy nhiên thường không thể thỏa mãn tất cả các mục tiêu kể trên vì bản thân chúng có sự mâu thuẫn với nhau mà chỉ có thể dung hòa chúng ở mức độ nào đó.

**Các đặc điểm của tiến trình**



Điều phối hoạt động của các tiến trình là một vấn đề rất phức tạp, đòi hỏi hệ điều hành khi giải quyết phải xem xét nhiều yếu tố khác nhau để có thể đạt được những mục tiêu đề ra. Một số đặc tính của tiến trình cần được quan tâm như tiêu chuẩn điều phối :

**a) Tính hướng xuất / nhập của tiến trình ( I/O-boundedness):**Khi một tiến trình nhận được CPU, chủ yếu nó chỉ sử dụng CPU đến khi phát sinh một yêu cầu nhập xuất ? Hoạt động của các tiến trình như thế thường bao gồm nhiều lượt sử dụng CPU , mỗi lượt trong một thời gian khá ngắn.

**b) Tính hướng xử lý của tiến trình ( CPU-boundedness):**Khi một tiến trình nhận được CPU, nó có khuynh hướng sử dụng CPU đến khi hết thời gian dành cho nó ? Hoạt động của các tiến trình như thế thường bao gồm một số ít lượt sử dụng CPU , nhưng mỗi lượt trong một thời gian đủ dài.

**c) Tiến trình tương tác hay xử lý theo lô :**Người sử dụng theo kiểu tương tác thường yêu cầu được hồi đáp tức thời đối với các yêu cầu của họ, trong khi các tiến trình của tác vụ được xử lý theo lô nói chung có thể trì hoãn trong một thời gian chấp nhận được.

**d) Độ ưu tiên của tiến trình :**Các tiến trình có thể được phân cấp theo một số tiêu chuẩn đánh giá nào đó, một cách hợp lý, các tiến trình quan trọng hơn ( có độ ưu tiên cao hơn) cần được ưu tiên hơn.

**e) Thời gian đã sử dụng CPU của tiến trình :**Một số quan điểm ưu tiên chọn những tiến trình đã sử dụng CPU nhiều thời gian nhất vì hy vọng chúng sẽ cần ít thời gian nhất để hoàn tất và rời khỏi hệ thống . Tuy nhiên cũng có quan điểm cho rằng các tiến trình nhận được CPU trong ít thời gian là những tiến trình đã phải chờ lâu nhất, do vậy ưu tiên chọn chúng.

**f) Thời gian còn lại tiến trình cần để hoàn tất :**Có thể giảm thiểu thời gian chờ đợi trung bình của các tiến trình bằng cách cho các tiến trình cần ít thời gian nhất để hoàn tất được thực hiện trước. Tuy nhiên đáng tiếc là rất hiếm khi biết được tiến trình cần bao nhiêu thời gian nữa để kết thúc xử lý.

**Điều phối không độc quyền và điều phối độc quyền (preemptive/nopreemptive)**



Thuật toán điều phối cần xem xét và quyết định thời điểm chuyển đổi CPU giữa các tiến trình. Hệ điều hành có thể thực hiện cơ chế điều phối theo nguyên lý độc quyền hoặc không độc quyền.

**a) Điều phối độc quyền** : Nguyên lý điều phối độc quyền cho phép một tiến trình khi nhận được CPU sẽ có quyền độc chiếm CPU đến khi hoàn tất xử lý hoặc tự nguyện giải phóng CPU. Khi đó quyết định điều phối CPU sẽ xảy ra trong các tình huống sau:

Khi tiến trình chuyển từ trạng thái đang xử lý(running) sang trạng thái bị khóa blocked ( ví dụ chờ một thao tác nhập xuất hay chờ một tiến trình con kết thúc…).



Khi tiến trình kết thúc.



Các giải thuật độc quyền thường đơn giản và dễ cài đặt. Tuy nhiên chúng thường không thích hợp với các hệ thống tổng quát nhiều người dùng, vì nếu cho phép một tiến trình có quyền xử lý bao lâu tùy ý, có nghĩa là tiến trình này có thể giữ CPU một thời gian không xác định, có thể ngăn cản những tiến trình còn lại trong hệ thống có một cơ hội để xử lý.

**b) Điều phối không độc quyền** : Ngược với nguyên lý độc quyền, điều phối theo nguyên lý không độc quyền cho phép tạm dừng hoạt động của một tiến trình đang sẵn sàng xử lý. Khi một tiến trình nhận được CPU, nó vẫn được sử dụng CPU đến khi hoàn tất hoặc tự nguyện giải phóng CPU, nhưng một tiến trình khác có độ ưu tiên có thể dành quyền sử dụng CPU của tiến trình ban đầu. Như vậy là tiến trình có thể bị tạm dừng hoạt động bất cứ lúc nào mà không được báo trước, để tiến trình khác xử lý. Các quyết định điều phối xảy ra khi :

Khi tiến trình chuyển từ trạng thái đang xử lý (running) sang trạng thái bị khóa blocked ( ví dụ chờ một thao tác nhập xuất hay chờ một tiến trình con kết thúc…).



Khi tiến trình chuyển từ trạng thái đang xử lý (running) sang trạng thái ready ( ví dụ xảy ra một ngắt).



Khi tiến trình chuyển từ trạng thái chờ (blocked) sang trạng thái ready ( ví dụ một thao tác nhập/xuất hoàn tất).



Khi tiến trình kết thúc.



Các thuật toán điều phối theo nguyên tắc không độc quyền ngăn cản được tình trạng một tiến trình độc chiếm CPU, nhưng việc tạm dừng một tiến trình có thể dẫn đến các mâu thuẫn trong truy xuất, đòi hỏi phải sử dụng một phương pháp đồng bộ hóa thích hợp để giải quyết.

Trong các hệ thống sử dụng nguyên lý điều phối độc quyền có thể xảy ra tình trạng các tác vụ cần thời gian xử lý ngắn phải chờ tác vụ xử lý với thời gian rất dài hoàn tất! Nguyên lý điều phối độc quyền thường chỉ thích hợp với các hệ xử lý theo lô.

Đối với các hệ thống tương tác(time sharing), các hệ thời gian thực (real time),cần phải sử dụng nguyên lý điều phối không độc quyền để các tiến trình quan trọng có cơ hội hồi đáp kịp thời. Tuy nhiên thực hiện điều phối theo nguyên lý không độc quyền đòi hỏi những cơ chế phức tạp trong việc phân định độ ưu tiên, và phát sinh thêm chi phí khi chuyển đổi CPU qua lại giữa các tiến trình.

Tổ chức điều phối



**Các danh sách sử dụng trong quá trình điều phối.**

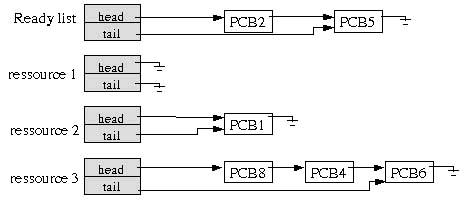


Hệ điều hành sử dụng hai loại danh sách để thực hiện điều phối các tiến trình là danh sách sẵn sàng (ready list) và danh sách chờ đợi(waiting list).

Khi một tiến trình bắt đầu đi vào hệ thống, nó được chèn vào danh sách các tác vụ (job list). Danh sách này bao gồm tất cả các tiến trình của hệ thống. Nhưng chỉ các tiến trình đang thường trú trong bộ nhớ chính và ở trạng thái sẵn sàng tiếp nhận CPU để hoạt động mới được đưa vào danh sách sẵn sàng.

Bộ điều phối sẽ chọn một tiến trình trong danh sách sẵn sàng và cấp CPU cho tiến trình đó. Tiến trình được cấp CPU sẽ thực hiện xử lý, và có thể chuyển sang trạng thái chờ khi xảy ra các sự kiện như đợi một thao tác nhập/xuất hoàn tất, yêu cầu tài nguyên chưa được thỏa mãn, được yêu cầu tạm dừng ...Khi đó tiến trình sẽ được chuyển sang một danh sách chờ đợi.

Hệ điều hành chỉ sử dụng một danh sách sẵn sàng cho toàn hệ thống, nhưng mỗi một tài nguyên ( thiết bị ngoại vi ) có một danh sách chờ đợi riêng bao gồm các tiến trình đang chờ được cấp phát tài nguyên đó.



**Hình 2.9** Các danh sách điều phối

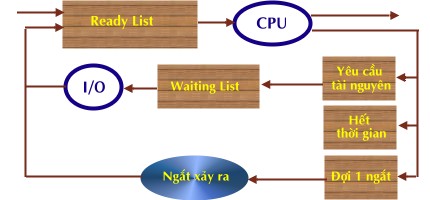
Quá trình xử lý của một tiến trình trải qua những chu kỳ chuyển đổi qua lại giữa danh sách sẵn sàng và danh sách chờ đợi. Sơ đồ dưới đây mô tả sự điều phối các tiến trình dựa trên các danh sách của hệ thống.

Thoạt đầu tiến trình mới được đặt trong danh sách các tiến trình sẵn sàng (ready list), nó sẽ đợi trong danh sách này cho đến khi được chọn để cấp phát CPU và bắt đầu xử lý. Sau đó có thể xảy ra một trong các tình huống sau  :

Tiến trình phát sinh một yêu cầu một tài nguyên mà hệ thống chưa thể đáp ứng, khi đó tiến trình sẽ được chuyển sang danh sách các tiến trình đang chờ tài nguyên tương ứng.



Tiến trình có thể bị bắt buộc tạm dừng xử lý do một ngắt xảy ra, khi đó tiến trình được đưa trở lại vào danh sách sẵn sàng để chờ được cấp CPU cho lượt tiếp theo.



**Hình 2.10** Sơ đồ chuyển đổi giữa các danh sách điều phối

Trong trường hợp đầu tiên, tiến trình cuối cùng sẽ chuyển từ trạng thái blocked sang trạng thái ready và lại được đưa trở vào danh sách sẵn sàng. Tiến trình lặp lại chu kỳ này cho đến khi hoàn tất tác vụ thì được hệ thống hủy bỏ khỏi mọi danh sách điều phối.

**Các cấp độ điều phối**



Thực ra công việc điều phối được hệ điều hành thực hiện ở hai mức độ : điều phối tác vụ (job scheduling) và điều phối tiến trình ( process scheduling).

**a) Điều phối tác vụ**

Quyết định lựa chọn tác vụ nào được đưa vào hệ thống, và nạp những tiến trình của tác vụ đó vào bộ nhớ chính để thực hiện. Chức năng điều phối tác vụ quyết định mức độ đa chương của hệ thống ( số lượng tiến trình trong bộ nhớ chính). Khi hệ thống tạo lập một tiến trình, hay có một tiến trình kết thúc xử lý thì chức năng điều phối tác vụ mới được kích hoạt. Vì mức độ đa chương tương đối ổn định nên chức năng điều phối tác vụ có tần suất hoạt động thấp .

Để hệ thống hoạt động tốt, bộ điều phối tác vụ cần biệt tính chất của tiến trình là hướng nhập xuất (I/O bounded) hay hướng xử lý ( CPU bounded). Một tiến trình được gọi là hướng nhập xuất nếu nó chủ yếu nó chỉ sử dụng CPU để thực hiện các thao tác nhập xuất. Ngược lại một tiến trình được gọi là hướng xử lý nếu nó chủ yếu nó chỉ sử dụng CPU để thực hiện các thao tác tính toán. Để cân bằng hoạt động của CPU và các thiết bị ngoại vi, bộ điều phối tác vụ nên lựa chọn các tiến trình để nạp vào bộ nhớ sao cho hệ thống là sự pha trộn hợp lý giữa các tiến trình hướng nhập xuất và các tiến trình hướng xử lý

**b) Điều phối tiến trình**

Chọn một tiến trình ở trạng thái sẵn sàng ( đã được nạp vào bộ nhớ chính, và có đủ tài nguyên để hoạt động ) và cấp phát CPU cho tiến trình đó thực hiện. Bộ điều phối tiến trình có tần suất hoạt động cao, sau mỗi lần xảy ra ngắt ( do đồng hồ báo giờ, do các thiết bị ngoại vi...), thường là 1 lần trong khoảng 100ms. Do vậy để nâng cao hiệu suất của hệ thống, cần phải tăng tốc độ xử lý của bộ điều phối tiến trình. Chức năng điều phối tiến trình là một trong chức năng cơ bản, quan trọng nhất của hệ điều hành.

Trong nhiều hệ điều hành, có thể không có bộ điều phối tác vụ hoặc tách biệt rất ít đối với bộ điều phối tiến trình. Một vài hệ điều hành lại đưa ra một cấp độ điều phối trung gian kết hợp cả hai cấp độ điều phối tác vụ và tiến trình



**Hình 2.11** Cấp độ điều phối trung gian

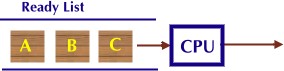
. Các chiến lược điều phối



**. Chiến lược FIFO**



**Nguyên tắc** : CPU được cấp phát cho tiến trình đầu tiên trong danh sách sẵn sàng có yêu cầu, là tiến trình được đưa vào hệ thống sớm nhất. Đây là thuật toán điều phối theo nguyên tắc độc quyền. Một khi CPU được cấp phát cho tiến trình, CPU chỉ được tiến trình tự nguyện giải phóng khi kết thúc xử lý hay khi có một yêu cầu nhập/xuất.



**Hình 2.12** Điều phối FIFO

Ví dụ :



|  |  |  |
| --- | --- | --- |
| Tiến trình | Thời điểm vào RL | Thời gian xử lý |
| P1 | 0 | 24 |
| P2 | 1 | 3 |
| P3 | 2 | 3 |

Thứ tự cấp phát CPU cho các tiến trình là :

|  |  |  |
| --- | --- | --- |
| P1 | P2 | P3 |
| 0 | 24 | 27 30 |

Thời gian chờ đợi được xử lý là 0 đối với P1, (24 -1) với P2 và (24+3-2) với P3.

Thời gian chờ trung bình là ( 0+23+25)/3 = 16 milisecondes.

Giải thuật này đặc biệt không phù hợp với các hệ phân chia thời gian, trong các hệ này, cần cho phép mỗi tiến trình được cấp phát CPU đều đặn trong từng khoảng thời gian.

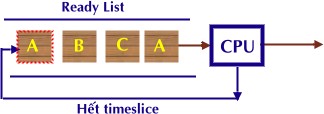
**Chiến lược phân phối xoay vòng (Round Robin)**



**Nguyên tắc** : Danh sách sẵn sàng được xử lý như một danh sách vòng, bộ điều phối lần lượt cấp phát cho từng tiến trình trong danh sách một khoảng thời gian sử dụng CPU gọi là quantum. Đây là một giải thuật điều phối không độc quyền : khi một tiến trình sử dụng CPU đến hết thời gian quantum dành cho nó, hệ điều hành thu hồi CPU và cấp cho tiến trình kế tiếp trong danh sách. Nếu tiến trình bị khóa hay kết thúc trước khi sử dụng hết thời gian quantum, hệ điều hành cũng lập tức cấp phát CPU cho tiến trình khác. Khi tiến trình tiêu thụ hết thời gian CPU dành cho nó mà chưa hoàn tất, tiến trình được đưa trở lại vào cuối danh sách sẵn sàng để đợi được cấp CPU trong lượt kế tiếp.



Ví dụ  :



**Hình 2.13** Điều phối Round Robin

|  |  |  |
| --- | --- | --- |
| Tiến trình | Thời điểm vào RL | Thời gian xử lý |
| P1 | 0 | 24 |
| P2 | 1 | 3 |
| P3 | 2 | 3 |

Nếu sử dụng quantum là 4 milisecondes, thứ tự cấp phát CPU sẽ là :

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P1 | P1 | P1 | P1 | P1 |
| 0 | ‘4 | 7 | 10 | 14 | 18 | 22 | 26 30 |

Thời gian chờ đợi trung bình sẽ là (0+6+3+5)/3 = 4.66 milisecondes.

Nếu có n tiến trìh trong danh sách sẵn sàng và sử dụng quantum q, thì mỗi tiến trình sẽ được cấp phát CPU 1/n trong từng khoảng thời gian q. Mỗi tiến trình sẽ không phải đợi quá (n-1)q đơn vị thời gian trước khi nhận được CPU cho lượt kế tiếp.

***Điều phối với độ ưu tiên***



**Nguyên tắc** : Mỗi tiến trình được gán cho một độ ưu tiên tương ứng, tiến trình có độ ưu tiên cao nhất sẽ được chọn để cấp phát CPU đầu tiên. Độ ưu tiên có thể được định nghĩa nội tại hay nhờ vào các yếu tố bên ngoài. Độ ưu tiên nội tại sử dụng các đại lượng có thể đo lường để tính toán độ ưu tiên của tiến trình, ví dụ các giới hạn thời gian, nhu cầu bộ nhớ…Độ ưu tiên cũng có thể được gán từ bên ngoài dựa vào các tiêu chuẩn do hệ điều hành như tầm quan trọng của tiến trình, loại người sử dụng sỡ hữu tiến trình…



Giải thuật điều phối với độ ưu tiên có thể theo nguyên tắc độc quyền hay không độc quyền. Khi một tiến trình được đưa vào danh sách các tiến trình sẵn sàng, độ ưu tiên của nó được so sánh với độ ưu tiên của tiến trình hiện hành đang xử lý. Giải thuật điều phối với độ ưu tiên và không độc quyền sẽ thu hồi CPU từ tiến trình hiện hành để cấp phát cho tiến trình mới nếu độ ưu tiên của tiến trình này cao hơn tiến trình hiện hành. Một giải thuật độc quyền sẽ chỉ đơn giản chèn tiến trình mới vào danh sách sẵn sàng, và tiến trình hiện hành vẫn tiếp tục xử lý hết thời gian dành cho nó.

Ví dụ : (độ ưu tiên 1 > độ ưu tiên 2> độ ưu tiên 3)



|  |  |  |  |
| --- | --- | --- | --- |
| Tiến trình | Thời điểm vào RL | Độ ưu tiên | Thời gian xử lý |
| P1 | 0 | 3 | 24 |
| P2 | 1 | 1 | 3 |
| P3 | 2 | 2 | 3 |

Sử dụng thuật giải độc quyền, thứ tự cấp phát CPU như sau :

|  |  |  |
| --- | --- | --- |
| P1 | P2 | P3 |
| 0 | ‘24 | 27 30 |

Sử dụng thuật giải không độc quyền, thứ tự cấp phát CPU như sau :

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | P2 | P3 | P1 |
| 0 | ‘1 | 4 | 7 30 |

**Chiến lược công việc ngắn nhất (Shortest-job-first SJF)**



**Nguyên tắc** : Đây là một trường hợp đặc biệt của giải thuật điều phối với độ ưu tiên. Trong giải thuật này, độ ưu tiên p được gán cho mỗi tiến trình là nghịch đảo của thời gian xử lý t mà tiến trình yêu cầu : p = 1/t. Khi CPU được tự do, nó sẽ được cấp phát cho tiến trình yêu cầu ít thời gian nhất để kết thúc- tiến trình ngắn nhất. Giải thuật này cũng có thể độc quyền hay không độc quyền. Sự chọn lựa xảy ra khi có một tiến trình mới được đưa vào danh sách sẵn sàng trong khi một tiến trình khác đang xử lý. Tiến trình mới có thể sỡ hữu một yêu cầu thời gian sử dụng CPU cho lần tiếp theo (CPU-burst) ngắn hơn thời gian còn lại mà tiến trình hiện hành cần xử lý. Giải thuật SJF không độc quyền sẽ dừng hoạt động của tiến trình hiện hành, trong khi giải thuật độc quyền sẽ cho phép tiến trình hiện hành tiếp tục xử lý.



Ví dụ :



|  |  |  |
| --- | --- | --- |
| Tiến trình | Thời điểm vào RL | Thời gian xử lý |
| P1 | 0 | 6 |
| P2 | 1 | 8 |
| P3 | 2 | 4 |
| P4 | 3 | 2 |

Sử dụng thuật giải SJF độc quyền, thứ tự cấp phát CPU như sau:

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | P4 | P3 | P2 |
| 0 | 6 | 8 | 12 20 |

Sử dụng thuật giải SJF không độc quyền, thứ tự cấp phát CPU như sau:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P1 | P4 | P1 | P3 | P2 |
| 0 | 3 | 5 | 8 | 12 20 |

**. Chiến lược điều phối với nhiều mức độ ưu tiên**



**Nguyên tắc** : Ý tưởng chính của giải thuật là phân lớp các tiến trình tùy theo độ ưu tiên của chúng để có cách thức điều phối thích hợp cho từng nhóm. Danh sách sẵn sàng được phân tách thành các danh sách riêng biệt theo cấp độ ưu tiên, mỗi danh sách bao gồm các tiến trình có cùng độ ưu tiên và được áp dụng một giải thuật điều phối thích hợp để điều phối. Ngoài ra, còn có một giải thuật điều phối giữa các nhóm, thường giải thuật này là giải thuật không độc quyền và sử dụng độ ưu tiên cố định.Một tiến trình thuộc về danh sách ở cấp ưu tiên i sẽ chỉ được cấp phát CPU khi các danh sách ở cấp ưu tiên lớn hơn i đã trống.



**Hình 2.14** Điều phối nhiều cấp ưu tiên

**Chiến lược điều phối Xổ số (Lottery)**



**Nguyên tắc** : Ý tưởng chính của giải thuật là phát hành một số vé số và phân phối cho các tiến trình trong hệ thống. Khi đến thời điểm ra quyết định điều phối, sẽ tiến hành chọn 1 vé "trúng giải", tiến trình nào sỡ hữu vé này sẽ được nhận CPU



1. LIÊN LẠC GIỮA CÁC TIẾN TRÌNH

Nhu cầu liên lạc giữa các tiến trình



Trong môi trường đa chương, một tiến trình không đơn độc trong hệ thống , mà có thể ảnh hưởng đến các tiến trình khác , hoặc bị các tiến trình khác tác động. Nói cách khác, các tiến trình là những thực thể độc lập , nhưng chúng vẫn có nhu cầu liên lạc với nhau để :

**Chia sẻ thông tin**: nhiều tiến trình có thể cùng quan tâm đến những dữ liệu nào đó, do vậy hệ điều hành cần cung cấp một môi trường cho phép sự truy cập đồng thời đến các dữ liệu chung.



**Hợp tác hoàn thành tác vụ**: đôi khi để đạt được một sự xử lý nhanh chóng, người ta phân chia một tác vụ thành các công việc nhỏ có thể tiến hành song song. Thường thì các công việc nhỏ này cần hợp tác với nhau để cùng hoàn thành tác vụ ban đầu, ví dụ dữ liệu kết xuất của tiến trình này lại là dữ liệu nhập cho tiến trình khác …Trong các trường hợp đó, hệ điều hành cần cung cấp cơ chế để các tiến trình có thể trao đổi thông tin với nhau.



### Các vấn đề nảy sinh trong việc liên lạc giữa các tiến trình



Do mỗi tiến trình sỡ hữu một không gian địa chỉ riêng biệt, nên các tiến trình không thể liên lạc trực tiếp dễ dàng mà phải nhờ vào các cơ chế do hệ điều hành cung cấp. Khi cung cấp cơ chế liên lạc cho các tiến trình, hệ điều hành thường phải tìm giải pháp cho các vấn đề chính yếu sau :

Liên kết tường minh hay tiềm ẩn (explicit naming/implicit naming) : tiến trình có cần phải biết tiến trình nào đang trao đổi hay chia sẻ thông tin với nó ? Mối liên kết được gọi là tường minh khi được thiết lập rõ ràng , trực tiếp giữa các tiến trình, và là tiềm ẩn khi các tiến trình liên lạc với nhau thông qua một qui ước ngầm nào đó.



Liên lạc theo chế độ đồng bộ hay không đồng bộ (blocking / non-blocking): khi một tiến trình trao đổi thông tin với một tiến trình khác, các tiến trình có cần phải đợi cho thao tác liên lạc hoàn tất rồi mới tiếp tục các xử lý khác ? Các tiến trình liên lạc theo cơ chế đồng bộ sẽ chờ nhau hoàn tất việc liên lạc, còn các tiến trình liên lạc theo cơ chế nonblocking thì không.



Liên lạc giữa các tiến trình trong hệ thống tập trung và hệ thống phân tán: cơ chế liên lạc giữa các tiến trình trong cùng một máy tính có sự khác biệt với việc liên lạc giữa các tiến trình giữa những máy tính khác nhau?



Hầu hết các hệ điều hành đưa ra nhiều cơ chế liên lạc khác nhau, mỗi cơ chế có những đặc tính riêng, và thích hợp trong một hoàn cảnh chuyên biệt.

CÁC CƠ CHẾ THÔNG TIN LIÊN LẠC



. Tín hiệu (Signal)



**Giới thiệu:** Tín hiệu là một cơ chế phần mềm tương tự như các ngắt cứng tác động đến các tiến trình. Một tín hiệu được sử dụng để thông báo cho tiến trình về một sự kiện nào đó xảy ra. Có nhiều tín hiệu được định nghĩa, mỗi một tín hiệu có một ý nghĩa tương ứng với một sự kiện đặc trưng.



Ví dụ : Một số tín hiệu của UNIX



|  |  |
| --- | --- |
| Tín hiệu | Mô tả |
| SIGINT | Người dùng nhấn phím DEL để ngắt xử lý tiến trình |
| SIGQUIT | Yêu cầu thoát xử lý |
| SIGILL | Tiến trình xử lý một chỉ thị bất hợp lệ |
| SIGKILL | Yêu cầu kết thúc một tiến trình |
| SIGFPT | Lỗi floating – point xảy ra ( chia cho 0) |
| SIGPIPE | Tiến trình ghi dữ liệu vào pipe mà không có reader |
| SIGSEGV | Tiến trình truy xuất đến một địa chỉ bất hợp lệ |
| SIGCLD | Tiến trình con kết thúc |
| SIGUSR1 | Tín hiệu 1 do người dùng định nghĩa |
| SIGUSR2 | Tín hiệu 2 do người dùng định nghĩa |

Mỗi tiến trình sỡ hữu một bảng biễu diễn các tín hiệu khác nhau. Với mỗi tín hiệu sẽ có tương ứng một trình xử lý tín hiệu (signal handler) qui định các xử lý của tiến trình khi nhận được tín hiệu tương ứng.

Các tín hiệu được gởi đi bởi :

Phần cứng (ví dụ lỗi do các phép tính số học)



Hạt nhân hệ điều hành gởi đến một tiến trình ( ví dụ lưu ý tiến trình khi có một thiết bị nhập/xuất tự do).



Một tiến trình gởi đến một tiến trình khác ( ví dụ tiến trình cha yêu cầu một tiến trình con kết thúc)



Người dùng ( ví dụ nhấn phím Ctl-C để ngắt xử lý của tiến trình)



Khi một tiến trình nhận một tín hiệu, nó có thể xử sự theo một trong các cách sau :

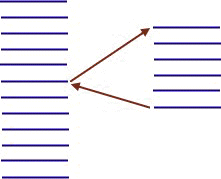
Bỏ qua tín hiệu



Xử lý tín hiệu theo kiểu mặc định



Tiếp nhận tín hiệu và xử lý theo cách đặc biệt của tiến trình.



**Hình 3.1** Liên lạc bằng tín hiệu

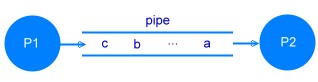
Pipe



**Giới thiệu:** Một pipe là một kênh liên lạc trực tiếp giữa hai tiến trình : dữ liệu xuất của tiến trình này được chuyển đến làm dữ liệu nhập cho tiến trình kia dưới dạng một dòng các byte.



Khi một pipe được thiết lập giữa hai tiến trình, một trong chúng sẽ ghi dữ liệu vào pipe và tiến trình kia sẽ đọc dữ liệu từ pipe. Thứ tự dữ liệu truyền qua pipe được bảo toàn theo nguyên tắc FIFO. Một pipe có kích thước giới hạn (thường là 4096 ký tự)



**Hình 3.2** Liên lạc qua pipe

Một tiến trình chỉ có thể sử dụng một pipe do nó tạo ra hay kế thừa từ tiến trình cha. Hệ điều hành cung cấp các lời gọi hệ thống read/write cho các tiến trình thực hiện thao tác đọc/ghi dữ liệu trong pipe. Hệ điều hành cũng chịu trách nhiệm đồng bộ hóa việc truy xuất pipe trong các tình huống:

Tiến trình đọc pipe sẽ bị khóa nếu pipe trống, nó sẽ phải đợi đến khi pipe có dữ liệu để truy xuất.



Tiến trình ghi pipe sẽ bị khóa nếu pipe đầy, nó sẽ phải đợi đến khi pipe có chỗ trống để chứa dữ liệu.



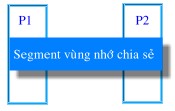
Vùng nhớ chia sẻ



**Giới thiệu:** Cách tiếp cận của cơ chế này là cho nhiều tiến trình cùng truy xuất đến một vùng nhớ chung gọi là vùng nhớ chia sẻ (shared memory).Không có bất kỳ hành vi truyền dữ liệu nào cần phải thực hiện ở đây, dữ liệu chỉ đơn giản được đặt vào một vùng nhớ mà nhiều tiến trình có thể cùng truy cập được.



Với phương thức này, các tiến trình chia sẻ một vùng nhớ vật lý thông qua trung gian không gian địa chỉ của chúng. Một vùng nhớ chia sẻ tồn tại độc lập với các tiến trình, và khi một tiến trình muốn truy xuất đến vùng nhớ này, tiến trình phải kết gắn vùng nhớ chung đó vào không gian địa chỉ riêng của từng tiến trình, và thao tác trên đó như một vùng nhớ riêng của mình.



**Hình 3.3** Liên lạc qua vùng nhớ chia sẻ

***Trao đổi thông điệp (Message)***



**Giới thiệu:** Hệ điều hành còn cung cấp một cơ chế liên lạc giữa các tiến trình không thông qua việc chia sẻ một tài nguyên chung , mà thông qua việc gởi thông điệp. Để hỗ trợ cơ chế liên lạc bằng thông điệp, hệ điều hành cung cấp các hàm IPC chuẩn (Interprocess communication), cơ bản là hai hàm:



**Send**(message)  : gởi một thông điệp



**Receive**(message)  : nhận một thông điệp



Nếu hai tiến trình P và Q muốn liên lạc với nhau, cần phải thiết lập một mối liên kết giữa hai tiến trình, sau đó P, Q sử dụng các hàm IPC thích hợp để trao đổi thông điệp, cuối cùng khi sự liên lạc chấm dứt mối liên kết giữa hai tiến trình sẽ bị hủy. Có nhiều cách thức để thực hiện sự liên kết giữa hai tiến trình và cài đặt các theo tác send /receive tương ứng : liên lạc trực tiếp hay gián tiếp, liên lạc đồng bộ hoặc không đồng bộ , kích thước thông điệp là cố định hay không … Nếu các tiến trình liên lạc theo kiểu liên kết tường minh, các hàm Send và Receive sẽ được cài đặt với tham số :

**Send**(destination, message)  : gởi một thông điệp đến destination



**Receive**(source,message)  : nhận một thông điệp từ source



**Thảo luận:** Đơn vị truyền thông tin trong cơ chế trao đổi thông điệp là một thông điệp, do đó các tiến trình có thể trao đổi dữ liệu ở dạng có cấu trúc.



***Sockets***



**Giới thiệu:** Một socket là một thiết bị truyền thông hai chiều tương tự như tập tin, chúng ta có thể đọc hay ghi lên nó, tuy nhiên mỗi socket là một thành phần trong một mối nối nào đó giữa các máy trên mạng máy tính và các thao tác đọc/ghi chính là sự trao đổi dữ liệu giữa các ứng dụng trên nhiều máy khác nhau.



Sử dụng socket có thể mô phỏng hai phương thức liên lạc trong thực tế : liên lạc thư tín (socket đóng vai trò bưu cục) và liên lạc điện thoại (socket đóng vai trò tổng đài) .

Các thuộc tính của socket:

Domaine: định nghĩa dạng thức địa chỉ và các nghi thức sử dụng. Có nhiều domaines, ví dụ UNIX, INTERNET, XEROX\_NS, ...



Type: định nghĩa các đặc điểm liên lạc:



**a) Sự tin cậy**

**b) Sự bảo toàn thứ tự dữ liệu**

**c) Lặp lại dữ liệu**

**d) Chế độ nối kết**

**e) Bảo toàn giới hạn thông điệp**

**f) Khả năng gởi thông điệp khẩn**

Để thực hiện liên lạc bằng socket, cần tiến hành các thao tác :

Tạo lập hay mở một socket



Gắn kết một socket với một địa chỉ



Liên lạc : có hai kiểu liên lạc tùy thuộc vào chế độ nối kết:



**a) Liên lạc trong chế độ không liên kết :** liên lạc theo hình thức hộp thư:

hai tiến trình liên lạc với nhau không kết nối trực tiếp



mỗi thông điệp phải kèm theo địa chỉ người nhận.



Hình thức liên lạc này có đặc điểm được :

người gởi không chắc chắn thông điệp của học được gởi đến người nhận,



một thông điệp có thể được gởi nhiều lần,



hai thông điệp được gởi theo một thứ tự nào đó có thể đến tay người nhận theo một thứ tự khác.



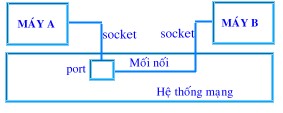
Một tiến trình sau khi đã mở một socket có thể sử dụng nó để liên lạc với nhiều tiến trình khác nhau nhờ sử hai primitive send và receive.

**b) Liên lạc trong chế độ nối kết:** Một liên kết được thành lập giữa hai tiến trình. Trước khi mối liên kết này được thiết lập, một trong hai tiến trình phải đợi có một tiến trình khác yêu cầu kết nối.Có thể sử dụng socket để liên lạc theo mô hình client-serveur.Trong mô hình này, server sử dụng lời gọi hệ thống listen và accept để nối kết với client, sau đó client và server có thể trao đổi thông tin bằng cách sử dụng các primitive send và receive.

Hủy một socket



Ví dụ : Trong nghi thức truyền thông TCP, mỗi mối nối giữa hai máy tính được xác định bởi một port, khái niệm port ở đây không phải là một cổng giao tiếp trên thiết bị vật lý mà chỉ là một khái niệm logic trong cách nhìn của người lập trình, mỗi port được tương ứng với một số nguyên dương.



**Hình 3.4** Các socket và port trong mối nối TCP.

Hình 3.4 minh họa một cách giao tiếp giữa hai máy tính trong nghi thức truyền thông TCP. Máy A tạo ra một socket và kết buộc (bind) socket nầy với một port X (tức là một số nguyên dương có ý nghĩa cục bộ trong máy A), trong khi đó máy B tạo một socket khác và móc vào (connect) port X trong máy A.

**Thảo luận:** Cơ chế socket có thể sử dụng để chuẩn hoá mối liên lạc giữa các tiến trình vốn không liên hệ với nhau, và có thể hoạt động trong những hệ thống khác nhau.



1. NHU CẦU ĐỒNG BỘ HÓA (SYNCHRONISATION)

Trong một hệ thống cho phép các tiến trình liên lạc với nhau, bao giờ hệ điều hành cũng cần cung cấp kèm theo những cơ chế đồng bộ hóa để bảo đảm hoạt động của các tiến trình đồng hành không tác động sai lệch đến nhau vì các lý do sau đây:

### Yêu cầu độc quyền truy xuất (Mutual exclusion)



Các tài nguyên trong hệ thống được phân thành hai loại: tài nguyên có thể chia sẻ cho phép nhiều tiến trình đồng thời truy xuất, và tài nguyên không thể chia sẻ chỉ chấp nhận một ( hay một số lượng hạn chế ) tiến trình sử dụng tại một thời điểm. Tính không thể chia sẻ của tài nguyên thường có nguồn gốc từ một trong hai nguyên nhân sau đây:

Đặc tính cấu tạo phần cứng của tài nguyên không cho phép chia sẻ.



Nếu nhiều tiến trình sử dụng tài nguyên đồng thời, có nguy cơ xảy ra các kết quả không dự đoán được do hoạt động của các tiến trình trên tài nguyên ảnh hưởng lẫn nhau.



Để giải quyết vấn đề, cần bảo đảm tiến trình độc quyền truy xuất tài nguyên, nghĩa là hệ thống phải kiểm soát sao cho tại một thời điểm, chỉ có một tiến trình được quyền truy xuất một tài nguyên không thể chia sẻ.

### Yêu cầu phối hợp (Synchronization)



Nhìn chung, mối tương quan về tốc độ thực hiện của hai tiến trình trong hệ thống là không thể biết trước, vì điều này phụ thuộc vào nhiều yếu tố động như tần suất xảy ra các ngắt của từng tiến trình, thời gian tiến trình được cấp phát bộ xử lý… Có thể nói rằng các tiến trình hoạt động không đồng bộ với nhau. Như ng có những tình huống các tiến trình cần hợp tác trong việc hoàn thành tác vụ, khi đó cần phải đồng bộ hóa hoạt động của các tiến trình , ví dụ một tiến trình chỉ có thể xử lý nếu một tiến trình khác đã kết thúc một công việc nào đó …

### Bài toán đồng bộ hoá



**Vấn đề tranh đoạt điều khiển (race condition)**



Giả sử có hai tiến trình P1 và P2 thực hiện công việc của các kế toán, và cùng chia sẻ một vùng nhớ chung lưu trữ biến taikhoan phản ánh thông tin về tài khoản. Mỗi tiến trình muốn rút một khoản tiền tienrut từ tài khoản:

if (taikhoan - tienrut >=0)

    taikhoan = taikhoan - tienrut;

else

    error(« khong the rut tien ! »);

Giả sử trong tài khoản hiện còn 800, P1 muốn rút 500 và P2 muốn rút 400. Nếu xảy ra tình huống như sau :

Sau khi đã kiểm tra điều kiện (taikhoan - tienrut >=0) và nhận kết quả là 300, P1 hết thời gian xử lý mà hệ thống cho phép, hệ điều hành cấp phát CPU cho P2.



P2 kiểm tra cùng điều kiện trên, nhận được kết quả là 400 (do P1 vẫn chưa rút tiền) và rút 400. Giá trị của taikhoan được cập nhật lại là 400.



Khi P1 được tái kích hoạt và tiếp tục xử lý, nó sẽ không kiểm tra lại điều kiện (taikhoan - tienrut >=0)-vì đã kiểm tra trong lượt xử lý trước- mà thực hiện rút tiền. Giá trị của taikhoan sẽ lại được cập nhật thành -100. Tình huống lỗi xảy ra !



Các tình huống tương tự như thế - có thể xảy ra khi có nhiều hơn hai tiến trình đọc và ghi dữ liệu trên cùng một vùng nhớ chung, và kết quả phụ thuộc vào sự điều phối tiến trình của hệ thống- được gọi là các tình huống tranh đoạt điều khiển (race condition) .

**Miền găng (critical section)**



Để ngăn chặn các tình huống lỗi có thể nảy sinh khi các tiến trình truy xuất đồng thời một tài nguyên không thể chia sẻ, cần phải áp đặt một sự truy xuất độc quyền trên tài nguyên đó : khi một tiến trình đang sử dụng tài nguyên, thì những tiến trình khác không được truy xuất đến tài nguyên.

Đoạn chương trình trong đó có khả năng xảy ra các mâu thuẫn truy xuất trên tài nguyên chung được gọi là miền găng (critical section). Trong ví dụ trên, đoạn mã :

if (taikhoan - tienrut >=0)

taikhoan = taikhoan - tienrut;

của mỗi tiến trình tạo thành một miền găng.

Có thể giải quyết vấn đề mâu thuẫn truy xuất nếu có thể bảo đảm tại một thời điểm chỉ có duy nhất một tiến trình được xử lý lệnh trong miền găng.

Một phương pháp giải quyết tốt bài toán miền găng cần thõa mãn 4 điều kiện sau :

Không có hai tiến trình cùng ở trong miền găng cùng lúc.



Không có giả thiết nào đặt ra cho sự liên hệ về tốc độ của các tiến trình, cũng như về số lượng bộ xử lý trong hệ thống.



Một tiến trình tạm dừng bên ngoài miền găng không được ngăn cản các tiến trình khác vào miền găng.



Không có tiến trình nào phải chờ vô hạn để được vào miền găng.



1. CÁC GIẢI PHÁP ĐỒNG BỘ HOÁ

*Có nhiều giải pháp để thực hiện việc truy xuất miền găng, các giải pháp này được phân biệt thành hai lớp tùy theo cách tiếp cận trong xử lý của tiến trình bị khóa :các giải pháp  “busy waiting” và các giải pháp “sleep and wakeup”.*

Giải pháp “Busy Waiting”



Các giải pháp phần mềm



Sử dụng các biến cờ hiệu:



Tiếp cân : các tiến trình chia sẻ một biến chung đóng vai trò « chốt cửa » (lock) , biến này được khởi động là 0. Một tiến trình muốn vào miền găng trước tiên phải kiểm tra giá trị của biến lock. Nếu lock = 0, tiến trình đặt lại giá trị cho lock = 1 và đi vào miền găng. Nếu lock đang nhận giá trị 1, tiến trình phải chờ bên ngoài miền găng cho đến khi lock có giá trị 0. Như vậy giá trị 0 của lock mang ý nghĩa là không có tiến trình nào đang ở trong miền găng, và lock=1 khi có một tiến trình đang ở trong miền găng.



while (TRUE) {

while (lock == 1); // wait  
lock = 1;  **critical-section ();**lock = 0**;**Noncritical-section ();

}

**Hình 3.5** Cấu trúc một chương trình sử dụng biến khóa để đồng bộ

Thảo luận : Giải pháp này có thể vi phạm điều kiện thứ nhất: hai tiến trình có thể cùng ở trong miền găng tại một thời điểm. Giả sử một tiến trình nhận thấy lock = 0 và chuẩn bị vào miền găng, nhưng trước khi nó có thể đặt lại giá trị cho lock là 1, nó bị tạm dừng để một tiến trình khác hoạt động. Tiến trình thứ hai này thấy lock vẫn là 0 thì vào miền găng và đặt lại lock = 1. Sau đó tiến trình thứ nhất được tái kích hoạt, nó gán lock = 1 lần nữa rồi vaò miền găng. Như vậy tại thời điểm đó cả hai tiến trình đều ở trong miền găng.



Sử dụng việc kiểm tra luân phiên :



Tiếp cận : Đây là một giải pháp đề nghị cho hai tiến trình. Hai tiến trình này sử dụng chung biến turn (phản ánh phiên tiến trình nào được vào miền găng), được khởi động với giá trị 0. Nếu turn = 0, tiến trình A được vào miền găng. Nếu turn = 1, tiến trình A đi vào một vòng lặp chờ đến khi turn nhận giá trị 0. Khi tiến trình A rời khỏi miền găng, nó đặt giá trị turn về 1 để cho phép tiến trình B đi vào miền găng.



while (TRUE) {

while (turn != 0); // wait **critical-section ();**turn = 1**;**Noncritical-section ();

}

**(a)** Cấu trúc tiến trình A

while (TRUE) {

while (turn != 1); // wait **critical-section ();**turn = 0**;**Noncritical-section ();

}

**(b)** Cấu trúc tiến trình B

**Hình 3.6** Cấu trúc các tiến trình trong giải pháp kiểm tra luân phiên

Thảo luận: Giải pháp này dựa trên việc thực hiện sự kiểm tra nghiêm nhặt đến lượt tiến trình nào được vào miền găng. Do đó nó có thể ngăn chặn được tình trạng hai tiến trình cùng vào miền găng, nhưng lại có thể vi phạm điều kiện thứ ba: một tiến trình có thể bị ngăn chặn vào miền găng bởi một tiến trình khác không ở trong miền găng. Giả sử tiến trình B ra khỏi miền găng rất nhanh chóng. Cả hai tiến trình đều ở ngoài miền găng, và turn = 0. Tiến trình A vào miền găng và ra khỏi nhanh chóng, đặt lại giá trị của turn là1, rồi lại xử lý đoạn lệnh ngoài miền găng lần nữa. Sau đó, tiến trình A lại kết thúc nhanh chóng đoạn lệnh ngoài miền găng của nó và muốn vào miền găng một lần nữa. Tuy nhiên lúc này B vẫn còn mãi xử lý đoạn lệnh ngoài miền găng của mình, và turn lại mang giá trị 1 ! Như vậy, giải pháp này không có giá trị khi có sự khác biệt lớn về tốc độ thực hiện của hai tiến trình, nó vi phạm cả điều kiện thứ hai.



Giải pháp của Peterson



Tiếp cận : Petson đưa ra một giải pháp kết hợp ý tưởng của cả hai giải pháp kể trên. Các tiến trình chia sẻ hai biến chung :



int turn; // đến phiên ai

int interesse[2]; // khởi động là FALSE

Nếu interesse[i] = TRUE có nghĩa là tiến trình Pi muốn vào miền găng. Khởi đầu, interesse[0]=interesse[1]=FALSE và giá trị của est được khởi động là 0 hay 1. Để có thể vào được miền găng, trước tiên tiến trình Pi đặt giá trị interesse[i]=TRUE ( xác định rằng tiến trình muốn vào miền găng), sau đó đặt turn=j (đề nghị thử tiến trình khác vào miền găng). Nếu tiến trình Pj không quan tâm đến việc vào miền găng (interesse[j]=FALSE), thì Pi có thể vào miền găng, nếu không, Pi phải chờ đến khi interesse[j]=FALSE. Khi tiến trình Pi rời khỏi miền găng, nó đặt lại giá trị cho interesse[i]= FALSE.

while (TRUE) {

int j = 1-i; // j là tiến trình còn lại  
interesse[i]= TRUE;  
turn = j;  
while (turn == j && interesse[j]==TRUE); **critical-section ();**interesse[i] = FALSE**;**Noncritical-section ();

}

**Hình 3.7** Cấu trúc tiến trình Pi trong giải pháp Peterson

Thảo luận: giải pháp này ngăn chặn được tình trạng mâu thuẫn truy xuất : mỗi tiến trình Pi chỉ có thể vào miền găng khi interesse[j]=FALSE hoặc turn = i. Nếu cả hai tiến trình đều muốn vào miền găng thì interesse[i] = interesse[j] =TRUE nhưng giá trị của turn chỉ có thể hoặc là 0 hoặc là 1, do vậy chỉ có một tiến trình được vào miền găng.



Các giải pháp phần cứng



Cấm ngắt:



Tiếp cân: cho phép tiến trình cấm tất cả các ngắt trước khi vào miền găng, và phục hồi ngắt khi ra khỏi miền găng. Khi đó, ngắt đồng hồ cũng không xảy ra, do vậy hệ thống không thể tạm dừng hoạt động của tiến trình đang xử lý để cấp phát CPU cho tiến trình khác, nhờ đó tiến trình hiện hành yên tâm thao tác trên miền găng mà không sợ bị tiến trình nào khác tranh chấp.



Thảo luận: giải pháp này không được ưa chuộng vì rất thiếu thận trọng khi cho phép tiến trình người dùng được phép thực hiện lệnh cấm ngắt. Hơn nữa, nếu hệ thống có nhiều bộ xử lý, lệnh cấm ngắt chỉ có tác dụng trên bộ xử lý đang xử lý tiến trình, còn các tiến trình hoạt động trên các bộ xử lý khác vẫn có thể truy xuất đến miền găng !



Chỉ thị TSL (Test-and-Set)**:**



Tiếp cận: đây là một giải pháp đòi hỏi sự trợ giúp của cơ chế phần cứng. Nhiều máy tính cung cấp một chỉ thị đặc biệt cho phép kiểm tra và cập nhật nội dung một vùng nhớ trong một thao tác không thể phân chia, gọi là chỉ thị Test-and-Set Lock (TSL) và được định nghĩa như sau:



**Test-and-Setlock**(boolean target)  
{

Test-and-Setlock = target;  
target = TRUE;

}

Nếu có hai chỉ thị TSL xử lý đồng thời (trên hai bộ xử lý khác nhau), chúng sẽ được xử lý tuần tự . Có thể cài đặt giải pháp truy xuất độc quyền với TSL bằng cách sử dụng thêm một biến lock, được khởi gán là FALSE. Tiến trình phải kiểm tra giá trị của biến lock trước khi vào miền găng, nếu lock = FALSE, tiến trình có thể vào miền găng.

while (TRUE) {

while (Test-and-Setlock(lock)); **critical-section ();**lock = FALSE**;**Noncritical-section ();

}

**Hình 3.8** Cấu trúc một chương trình trong giải pháp TSL

Thảo luận : cũng giống như các giải pháp phần cứng khác, chỉ thị TSL giảm nhẹ công việc lập trình để giải quyết vấn để, nhưng lại không dễ dàng để cài đặt chỉ thị TSL sao cho được xử lý một cách không thể phân chia, nhất là trên máy với cấu hình nhiều bộ xử lý.



Tất cả các giải pháp trên đây đều phải thực hiện một vòng lặp để kiểm tra liệu nó có được phép vào miền găng, nếu điều kiện chưa cho phép, tiến trình phải chờ tiếp tục trong vòng lặp kiểm tra này. Các giải pháp buộc tiến trình phải liên tục kiểm tra điều kiện để phát hiện thời điểm thích hợp được vào miền găng như thế được gọi các giải pháp « busy waiting ». Lưu ý rằng việc kiểm tra như thế tiêu thụ rất nhiều thời gian sử dụng CPU, do vậy tiến trình đang chờ vẫn chiếm dụng CPU. Xu hướng giải quyết vấn đề đồng bộ hoá là nên tránh các giải pháp « busy waiting ».

Các giải pháp “Sleep and Wakeup”



Để loại bỏ các bất tiện của giải pháp « busy waiting », chúng ta có thể tiếp cận theo hướng cho một tiến trình chưa đủ điều kiện vào miền găng chuyển sang trạng thái blocked, từ bỏ quyền sử dụng CPU. Để thực hiện điều này, cần phải sử dụng các thủ tục do hệ điều hành cung cấp để thay đổi trạng thái tiến trình. Hai thủ tục cơ bản SLEEP và WAKEUP thường được sử dụng để phục vụ mục đích này.

SLEEP là một lời gọi hệ thống có tác dụng tạm dừng hoạt động của tiến trình (blocked) gọi nó và chờ đến khi được một tiến trình khác « đánh thức ». Lời gọi hệ thống WAKEUP nhận một tham số duy nhất : tiến trình sẽ được tái kích hoạt (đặt về trạng thái ready).

Ý tưởng sử dụng SLEEP và WAKEUP như sau : khi một tiến trình chưa đủ điều kiện vào miền găng, nó gọi SLEEP để tự khóa đến khi có một tiến trình khác gọi WAKEUP để giải phóng cho nó. Một tiến trình gọi WAKEUP khi ra khỏi miền găng để đánh thức một tiến trình đang chờ, tạo cơ hội cho tiến trình này vào miền găng :

int busy; // 1 nếu miền găng đang bị chiếm, nếu không là 0

int blocked; // đếm số lượng tiến trình đang bị khóa

while (TRUE) {

if (busy){  
    blocked = blocked + 1;  
    sleep();  
}

else busy = 1;

**critical-section ();**busy = 0**;**if(blocked){  
    wakeup(process);  
    blocked = blocked - 1;  
}

Noncritical-section ();

}

**Hình 3.9** Cấu trúc chương trình trong giải pháp SLEEP and WAKEUP

Khi sử dụng SLEEP và WAKEUP cần hết sức cẩn thận, nếu không muốn xảy ra tình trạng mâu thuẫn truy xuất trong một vài tình huống đặc biệt như sau : giả sử tiến trình A vào miền găng, và trước khi nó rời khỏi miền găng thì tiến trình B được kích hoạt. Tiến trình B thử vào miền găng nhưng nó nhận thấy A đang ở trong đó, do vậy B tăng giá trị biến blocked và chuẩn bị gọi SLEEP để tự khoá. Tuy nhiên trước khi B có thể thực hiện SLEEP, tiến trình A lại được tái kích hoạt và ra khỏi miền găng. Khi ra khỏi miền găng A nhận thấy có một tiến trình đang chờ (blocked=1) nên gọi WAKEUP và giảm giá trị của blocked. Khi đó tín hiệu WAKEUP sẽ lạc mất do tiến trình B chưa thật sự « ngủ » để nhận tín hiệu đánh thức !Khi tiến trình B được tiếp tục xử lý, nó mới goi SLEEP và tự khóa vĩnh viễn !

Vấn đề ghi nhận được là tình trạng lỗi này xảy ra do việc kiểm tra tư cách vào miền găng và việc gọi SLEEP hay WAKEUP là những hành động tách biệ, có thể bị ngắt nửa chừng trong quá trình xử lý, do đó có khi tín hiệu WAKEUP gởi đến một tiến trình chưa bị khóa sẽ lạc mất.

Để tránh những tình huống tương tự, hệ điều hành cung cấp những cơ chế đồng bộ hóa dựa trên ý tưởng của chiến lược « SLEEP and WAKEUP » nhưng được xây dựng bao hàm cả phương tiện kiểm tra điều kiện vào miền găng giúp sử dụng an toàn.

Semaphore



Tiếp cận: Được **Dijkstra** đề xuất vào 1965, một semaphore **s** là một biến có các thuộc tính sau:



Một giá trị nguyên dương e(s)



Một hàng đợi f(s) lưu danh sách các tiến trình đang bị khóa (chờ) trên semaphore s

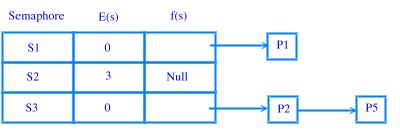


Chỉ có hai thao tác được định nghĩa trên semaphore



**Down(s):** giảm giá trị của semaphore s đi 1 đơn vị nếu semaphore có trị e(s) > 0, và tiếp tục xử lý. Ngược lại, nếu e(s)  0, tiến trình phải chờ đến khi e(s) >0.

**Up(s)**: tăng giá trị của semaphore s lên 1 đơn vị. Nếu có một hoặc nhiều tiến trình đang chờ trên semaphore s, bị khóa bởi thao tác **Down**, thì hệ thống sẽ chọn một trong các tiến trình này để kết thúc thao tác **Down** và cho tiếp tục xử lý.



**Hình 3.10** Semaphore s

Cài đặt: Gọi p là tiến trình thực hiện thao tác Down(s) hay Up(s).



**Down(s):**

e(s) = e(s) - 1;

if e(s) < 0 {

status(P)= **blocked**;

enter(P,f(s));

}

**Up(s)**:

e(s) = e(s) + 1;

if s != 0 {

exit(Q,f(s)); //Q là tiến trình đang chờ trên s

status (Q) = **ready**;

enter(Q,ready-list);

}

Lưu ý cài đặt này có thể đưa đến một giá trị âm cho semaphore, khi đó trị tuyệt đối của semaphore cho biết số tiến trình đang chờ trên semaphore.

Điều quan trọng là các thao tác này cần thực hiện một cách không bị phân chia, không bị ngắt nữa chừng, có nghĩa là không một tiến trình nào được phép truy xuất đến semaphore nếu tiến trình đang thao tác trên semaphore này chưa kết thúc xử lý hay chuyển sang trạng thái blocked.

Sử dụng: có thể dùng semaphore để giải quyết vấn đề truy xuất độc quyền hay tổ chức phối hợp giữa các tiến trình.



* **Tổ chức truy xuất độc quyền với Semaphores**: khái niệm semaphore cho phép bảo đảm nhiều tiến trình cùng truy xuất đến miền găng mà không có sự mâu thuẫn truy xuất. n tiến trình cùng sử dụng một semaphore s, e(s) được khởi gán là 1. Để thực hiện đồng bộ hóa, tất cả các tiến trình cần phải áp dụng cùng cấu trúc chương trình sau đây:

while (TRUE) {

Down(s)

**critical-section ();**

Up(s)

Noncritical-section ();

}

**Hình 3.11** Cấu trúc một chương trình trong giải pháp semaphore

* **Tổ chức đồng bộ hóa với Semaphores**: với semaphore có thể đồng bộ hóa hoạt động của hai tiến trình trong tình huống một tiến trình phải đợi một tiến trình khác hoàn tất thao tác nào đó mới có thể bắt đầu hay tiếp tục xử lý. Hai tiến trình chia sẻ một semaphore s, khởi gán e(s) là 0. Cả hai tiến trình có cấu trúc như sau:

**P1:**

while (TRUE) {

**job1();**Up(s); //đánh thức P2

}

**P2:**

while (TRUE) {

Down(s); // chờ P1 **job2();**

}

**Hình 3.12** Cấu trúc chương trình trong giải pháp semaphore

 Thảo luận : Nhờ có thực hiện một các không thể phân chia, semaphore đã giải quyết được vấn đề tín hiệu "đánh thức" bị thất lạc. Tuy nhiên, nếu lập trình viên vô tình đặt các primitive Down và Up sai vị trí, thứ tự trong chương trình, thì tiến trình có thể bị khóa vĩnh viễn.



Ví dụ :



while (TRUE) {

Down(s) **critical-section ();**Noncritical-section ();

}

tiến trình trên đây quên gọi Up(s), và kết quả là khi ra khỏi miền găng nó sẽ không cho tiến trình khác vào miền găng !

Vì thế việc sử dụng đúng cách semaphore để đồng bộ hóa phụ thuộc hoàn toàn vào lập trình viên và đòi hỏi lập trình viên phải hết sức thận trọng.

Monitors



Tiếp cận: Để có thể dễ viết đúng các chương trình đồng bộ hóa hơn, Hoare(1974) và Brinch & Hansen (1975) đã đề nghị một cơ chế cao hơn được cung cấp bởi ngôn ngữ lập trình , là monitor. Monitor là một cấu trúc đặc biệt bao gồm các thủ tục, các biến và cấu trúc dữ liệu có các thuộc tính sau :



Các biến và cấu trúc dữ liệu bên trong monitor chỉ có thể được thao tác bởi các thủ tục định nghĩa bên trong monitor đó. (encapsulation).



Tại một thời điểm, chỉ có một tiến trình duy nhất được hoạt động bên trong một monitor (mutual exclusive).



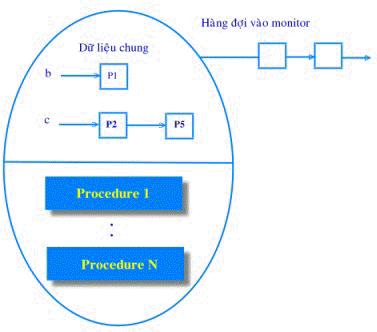
Trong một monitor, có thể định nghĩa các biến điều kiện và hai thao tác kèm theo là **Wait** và **Signal** như sau : gọi **c** là biến điều kiện được định nghĩa trong monitor:



**Wait(c):** chuyển trạng thái tiến trình gọi sang blocked , và đặt tiến trình này vào hàng đợi trên biến điều kiện **c**.



**Signal(c)**: nếu có một tiến trình đang bị khóa trong hàng đợi của **c**, tái kích hoạt tiến trình đó, và tiến trình gọi sẽ rời khỏi monitor.



**Hình 3.13** Monitor và các biến điều kiện

Cài đặt : trình biên dịch chịu trách nhiệm thực hiện việc truy xuất độc quyền đến dữ liệu trong monitor. Để thực hiện điều này, một semaphore nhị phân thường được sử dụng. Mỗi monitor có một hàng đợi toàn cục lưu các tiến trình đang chờ được vào monitor, ngoài ra, mỗi biến điều kiện **c** cũng gắn với một hàng đợi **f(c)** và hai thao tác trên đó được định nghĩa như sau:



**Wait(c) :**



status(P)= **blocked**;  
enter(P,f(c));

**Signal(c)** :

if (f(c) != NULL){

exit(Q,f(c)); //Q là tiến trình chờ trên c  
statusQ) = **ready**;  
enter(Q,ready-list);

}

Sử dụng: Với mỗi nhóm tài nguyên cần chia sẻ, có thể định nghĩa một monitor trong đó đặc tả tất cả các thao tác trên tài nguyên này với một số điều kiện nào đó.:



**monitor** <tên monitor >

**condition** <danh sách các biến điều kiện>;

**<déclaration de variables>**;

**procedure** Action1();

{

}

....

**procedure** Actionn();

{

}

**end monitor**;

**Hình 3.14** Cấu trúc một monitor

Các tiến trình muốn sử dụng tài nguyên chung này chỉ có thể thao tác thông qua các thủ tục bên trong monitor được gắn kết với tài nguyên:

while (TRUE) {

Noncritical-section (); **<monitor>.Actioni;** //critical-section();  
Noncritical-section ();

}

**Hình 3.15** Cấu trúc tiến trình Pitrong giảipháp monitor

Thảo luận: Với monitor, việc truy xuất độc quyền được bảo đảm bởi trình biên dịch mà không do lập trình viên, do vậy nguy cơ thực hiện đồng bộ hóa sai giảm rất nhiều. Tuy nhiên giải pháp monitor đòi hỏi phải có một ngôn ngữ lập trình định nghĩa khái niệm monitor, và các ngôn ngữ như thế chưa có nhiều.



Trao đổi thông điệp



Tiếp cận: giải pháp này dựa trên cơ sở trao đổi thông điệp với hai primitive Send và Receive để thực hiện sự đồng bộ hóa:



**Send(destination, message):** gởi một thông điệp đến một tiến trình hay gởi vào hộp thư.



**Receive(source,message)**: nhận một thông điệp thừ một tiến trình hay từ bất kỳ một tiến trình nào, tiến trình gọi sẽ chờ nếu không có thông điệp nào để nhận.



Sử dụng: Có nhiều cách thức để thực hiện việc truy xuất độc quyền bằng cơ chế trao đổi thông điệp. Đây là một mô hình đơn giản: một tiến trình kiểm soát việc sử dụng tài nguyên và nhiều tiến trình khác yêu cầu tài nguyên này. Tiến trình có yêu cầu tài nguyên sẽ gởi một thông điệp đến tiến trình kiểm soát và sau đó chuyển sang trạng thái blocked cho đến khi nhận được một thông điệp chấp nhận cho truy xuất từ tiến trình kiểm soát tài nguyên.Khi sử dụng xong tài nguyên , tiến trình gởi một thông điệp khác đến tiến trình kiểm soát để báo kết thúc truy xuất. Về phần tiến trình kiểm soát , khi nhận được thông điệp yêu cầu tài nguyên, nó sẽ chờ đến khi tài nguyên sẵn sàng để cấp phát thì gởi một thông điệp đến tiến trình đang bị khóa trên tài nguyên đó để đánh thức tiến trình này.



while (TRUE) {

Send(process controler, request message);  
Receive(process controler, accept message); **critical-section ();**Send(process controler, end message);  
Noncritical-section ();

}

**Hình 3.16** Cấu trúc tiến trình yêu cầu tài nguyên trong giải pháp message

Thảo luận: Các primitive semaphore và monitor có thể giải quyết được vấn đề truy xuất độc quyền trên các máy tính có một hoặc nhiều bộ xử lý chia sẻ một vùng nhớ chung. Nhưng các primitive không hữu dụng trong các hệ thống phân tán, khi mà mỗi bộ xử lý sỡ hữu một bộ nhớ riêng biệt và liên lạc thông qua mạng. Trong những hệ thống phân tán như thế, cơ chế trao đổi thông điệp tỏ ra hữu hiệu và được dùng để giải quyết bài toán đồng bộ hóa.



1. TẮC NGHẼN (DEALLOCK)

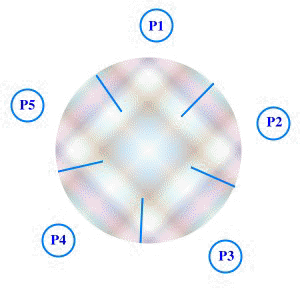
### Định nghĩa:



Một tập hợp các tiến trình được định nghĩa ở trong tình trạng tắc nghẽn khi mỗi tiến trình trong tập hợp đều chờ đợi một sự kiện mà chỉ có một tiến trình khác trong tập hợp mới có thể phát sinh được.

Nói cách khác, mỗi tiến trình trong tập hợp đều chờ được cấp phát một tài nguyên hiện đang bị một tiến trình khác cũng ở trạng thái blocked chiếm giữ. Như vậy không có tiến trình nào có thể tiếp tục xử lý , cũng như giải phóng tài nguyên cho tiến trình khác sử dụng, tất cả các tiến trình trong tập hợp đều bị khóa vĩnh viễn !

**Vấn đề Bữa ăn tối của các triết gia**: 5 nhà triết học cùng ngồi ăn tối với món spaghetti nổi tiếng. Mỗi nhà triết học cần dùng 2 cái nĩa để có thể ăn spaghetti . Nhưng trên bàn chỉ có tổng cộng 5 cái nĩa để xen kẽ với 5 cái đĩa. Mỗi nhà triết học sẽ suy ngẫm các triết lý của mình đến khi cảm thấy đói thì dự định lần lượt cầm 1 cái nĩa bên trái và 1 cái nĩa bên phải để ăn. Nếu cả 5 nhà triết học đều cầm cái nĩa bên trái cùng lúc, thì sẽ không có ai có được cái nĩa bên phải để có thể bắt đầu thưởng thức spaghetti . Đây chính là tình trạng tắc nghẽn.



**Hình 3.18** Bữa ăn tối của các triết gia

### Điều kiện xuất hiện tắc nghẽn



Coffman, Elphick và Shoshani đã đưa ra 4 điều kiện cần có thể làm xuất hiện tắc nghẽn:

Có sử dụng tài nguyên không thể chia sẻ (Mutual exclusion): Mỗi thời điểm, một tài nguyên không thể chia sẻ được hệ thống cấp phát chỉ cho một tiến trình , khi tiến trình sử dụng xong tài nguyên này, hệ thống mới thu hồi và cấp phát tài nguyên cho tiến trình khác.



Sự chiếm giữ và yêu cầu thêm tài nguyên (Wait for): Các tiến trình tiếp tục chiếm giữ các tài nguyên đã cấp phát cho nó trong khi chờ được cấp phát thêm một số tài nguyên mới.



Không thu hồi tài nguyên từ tiến trình đang giữ chúng (No preemption): Tài nguyên không thể được thu hồi từ tiến trình đang chiếm giữ chúng trước khi tiến trình này sủ dụng chúng xong.



Tồn tại một chu kỳ trong đồ thị cấp phát tài nguyên ( Circular wait): có ít nhất hai tiến trình chờ đợi lẫn nhau : tiến trình này chờ được cấp phát tài nguyên đang bị tiến trình kia chiếm giữ và ngược lại.

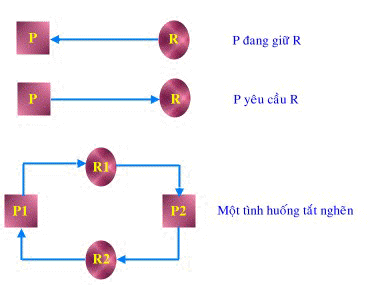


Khi có đủ 4 điều kiện này, thì tắc nghẽn xảy ra. Nếu thiếu một trong 4 điều kiện trên thì không có tắc nghẽn.

### Đồ thị cấp phát tài nguyên



Có thể sử dụng một đồ thị để mô hình hóa việc cấp phát tài nguyên. Đồ thị này có 2 loại nút : các tiến trình được biễu diễn bằng hình tròn, và mỗi tài nguyên được hiển thị bằng hình vuông



**Hình 3.19** Đồ thị cấp phát tài nguyên

### Các phương pháp xử lý tắc nghẽn



Chủ yếu có ba hương tiếp cận để xử lý tắc nghẽn :

Sử dụng một nghi thức (protocol) để bảo đảm rằng hệ thống không bao giờ xảy ra tắc nghẽn.



Cho phép xảy ra tắc nghẽn và tìm cách sữa chữa tắc nghẽn.



Hoàn toàn bỏ qua việc xử lý tắc nghẽn, xem như hệ thống không bao giờ xảy ra tắc nghẽn.



### Ngăn chặn tắc nghẽn



Để tắc nghẽn không xảy ra, cần bảo đảm tối thiểu một trong 4 điều kiện cần không xảy ra:

Tài nguyên không thể chia sẻ : nhìn chung gần như không thể tránh được điều kiện này vì bản chất tài nguyên gần như cố định. Tuy nhiên đối với một số tài nguyên về kết xuất, người ta có thể dùng các cơ chế spooling để biến đổi thành tài nguyên có thể chia sẻ.



Sự chiếm giữ và yêu cầu thêm tài nguyên: phải bảo đảm rằng mỗi khi tiến trình yêu cầu thêm một tài nguyên thì nó không chiếm giữ các tài nguyên khác. Có thể áp đặt một trong hai cơ chế truy xuất sau :



Tiến trình phải yêu cầu tất cả các tài nguyên cần thiết trước khi bắt đầu xử lý .



=> phương pháp này có khó khăn là tiến trình khó có thể ước lượng chính xác tài nguyên cần sử dụng vì có thể nhu cầu phụ thuộc vào quá trình xử lý . Ngoài ra nếu tiến trình chiếm giữ sẵn các tài nguyên chưa cần sử dụng ngay thì việc sử dụng tài nguyên sẽ kém hiệu quả.

Khi tiến trình yêu cầu một tài nguyên mới và bị từ chối, nó phải giải phóng các tài nguyên đang chiếm giữ , sau đó lại được cấp phát trở lại cùng lần với tài nguyên mới.



=> phương pháp này làm phát sinh các khó khăn trong việc bảo vệ tính toàn vẹn dữ liệu của hệ thống.

Không thu hồi tài nguyên: cho phép hệ thống được thu hồi tài nguyên từ các tiến trình bị khoá và cấp phát trở lại cho tiến trình khi nó thoát khỏi tình trạng bị khóa. Tuy nhiên với một số loại tài nguyên, việc thu hồi sẽ rất khó khăn vì vi phạm sự toàn vẹn dữ liệu .



Tồn tại một chu kỳ: tránh tạo chu kỳ trong đồ thị bằng cách cấp phát tài nguyên theo một sự phân cấp như sau :



gọi R = {R1, R2,...,Rm} là tập các loại tài nguyên.

Các loại tài nguyên được phân cấp từ 1-N.

Ví dụ : F(đĩa) = 2, F(máy in) = 12



Các tiến trình khi yêu cầu tài nguyên phải tuân thủ quy định : khi tiến trình đang chiếm giữ tài nguyên Ri thì chỉ có thể yêu cầu các tài nguyên Rj nếu F(Rj) > F(Ri).

### Tránh tắc nghẽn



Ngăn cản tắc nghẽn là một mối bận tâm lớn khi sử dụng tài nguyên. Tránh tắc nghẽn là loại bỏ tất cả các cơ hội có thể dẫn đến tắc nghẽn trong tương lai. Cần phải sử dụng những cơ chế phức tạp để thực hiện ý định này.

**Một số khái niệm cơ sở**



**Trạng thái an toàn** : trạng thái A là an toàn nếu hệ thống có thể thỏa mãn các nhu cầu tài nguyên (cho đến tối đa) của mỗi tiến trình theo một thứ tự nào đó mà vẫn ngăn chặn được tắc nghẽn.



**Một chuỗi cấp phát an toàn:** một thứ tự của các tiến trình <P1, P2,...,Pn> là an toàn đối với tình trạng cấp phát hiện hành nếu với mỗi tiến trình Pi nhu cầu tài nguyên của Pi có thể được thỏa mãn với các tài nguyên còn tự do của hệ thống, cộng với các tài nguyên đang bị chiếm giữ bởi các tiến trình Pj khác, với j<i.



Một trạng thái an toàn không thể là trạng thái tắc nghẽn. Ngược lại một trạng thái không an toàn có thể dẫn đến tình trạng tắc nghẽn.

**Chiến lược cấp phát** : chỉ thỏa mãn yêu cầu tài nguyên của tiến trình khi trạng thái kết quả là an toàn!



**Giải thuật xác định trạng thái an toàn**



Cần sử dụng các cấu trúc dữ liệu sau :

**int Available[NumResources];**

/\* Available[r]= số lượng các thể hiện còn tự do của tài nguyên r\*/

**int Max[NumProcs, NumResources];**

/\*Max[p,r]= nhu cầu tối đa của tiến trình p về tài nguyên r\*/

**int Allocation[NumProcs, NumResources];**

/\* Allocation[p,r] = số lượng tài nguyên r thực sự cấp phát cho p\*/

**int Need[NumProcs, NumResources];**

/\* Need[p,r] = Max[p,r] - Allocation[p,r]\*/

1.Giả sử có các mảng

int Work[NumProcs, NumResources] = Available;

int Finish[NumProcs] = false;

2.Tìm i sao cho

Finish[i] == false

Need[i] <= Work[i]

Nếu không có i như thế, đến bước 4.

3. Work = Work + Allocation[i];

Finish[i] = true;

Đến bước 2

4.Nếu Finish[i] == true với mọi i, thì hệ thống ở trạng thái an toàn.

Ví dụ : Giả sử tình trạng hiện hành của hệ thống được mô tả như sau :



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Max | | | Allocation | | | Available | | |
| R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 |
| P1 | 3 | 2 | 2 | 1 | 0 | 0 |  |  |
| P2 | 6 | 1 | 3 | 2 | 1 | 1 |  |  |
| P3 | 3 | 1 | 4 | 2 | 1 | 1 |  |  |
| P4 | 4 | 2 | 2 | 0 | 0 | 2 |  |  |

Nếu tiến trình P2 yêu cầu 4 cho R1, 1 cho R3. hãy cho biết yêu cầu này có thể đáp ứng mà bảo đảm không xảy ra tình trạng deadlock hay không ? Nhận thấy Available[1] =4, Available[3] =2 đủ để thõa mãn yêu cầu của P2, ta có

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Need | | | Allocation | | | Available | | |
| R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 |
| P1 | 2 | 2 | 2 | 1 | 0 | 0 |  |  |
| P2 | 0 | 0 | 1 | 6 | 1 | 2 |  |  |
| P3 | 1 | 0 | 3 | 2 | 1 | 1 |  |  |
| P4 | 4 | 2 | 0 | 0 | 0 | 2 |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Need | | | Allocation | | | Available | | |
| R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 |
| P1 | 2 | 2 | 2 | 1 | 0 | 0 |  |  |
| P2 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |
| P3 | 1 | 0 | 3 | 2 | 1 | 1 |  |  |
| P4 | 4 | 2 | 0 | 0 | 0 | 2 |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Need | | | Allocation | | | Available | | |
| R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 |
| P1 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |
| P2 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |
| P3 | 1 | 0 | 3 | 2 | 1 | 1 |  |  |
| P4 | 4 | 2 | 0 | 0 | 0 | 2 |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Need | | | Allocation | | | Available | | |
| R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 |
| P1 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |
| P2 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |
| P3 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |
| P4 | 4 | 2 | 0 | 0 | 0 | 2 |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Need | | | Allocation | | | Available | | |
| R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 |
| P1 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |
| P2 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |
| P3 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |
| P4 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |

Trạng thái kết qủa là an toàn, có thể cấp phát.

**Giải thuật yêu cầu tài nguyên**



Giả sử tiến trình Pi yêu cầu k thể hiện của tài nguyên r.

1.Nếu k <= Need[i], đến bước 2   
   Ngược lại, xảy ra tình huống lỗi

2.Nếu k <= Available[i],đến bước 3   
   Ngược lại, Pi phải chờ

3.Giả sử hệ thống đã cấp phát cho Pi các tài nguyên mà nó yêu cầu và cập nhật tình trạng hệ thống như sau:

Available[i] = Available[i] - k;   
Allocation[i]= Allocation[i]+ k;   
Need[i] = Need[i] - k;

    Nếu trạng thái kết quả là an toàn, lúc này các tài nguyên trên sẽ được cấp phát thật sự cho Pi

    Ngược lại, Pi phải chờ

### Phát hiện tắc nghẽn



Cần sử dụng các cấu trúc dữ liệu sau :

**int Available[NumResources];**

// Available[r]= số lượng các thể hiện còn tự do của tài nguyên r

**int Allocation[NumProcs, NumResources];**

// Allocation[p,r] = số lượng tài nguyên r thực sự cấp phát cho p

**int Request[NumProcs, NumResources];**

// Request[p,r] = số lượng tài nguyên r tiến trình p yêu cầu thêm

**Giải thuật phát hiện tắc nghẽn**



1. int Work[NumResources] = Available;

int Finish[NumProcs];

for (i = 0; i < NumProcs; i++)

Finish[i] = (Allocation[i] == 0);

2. Tìm i sao cho

Finish[i] == false

Request[i] <= Work

Nếu không có i như thế, đến bước 4.

3. Work = Work + Allocation[i];

Finish[i] = true;

Đến bước 2

4. Nếu Finish[i] == true với mọi i,

thì hệ thống không có tắc nghẽn

Nếu Finish[i] == false với một số giá trị i,

thì các tiến trình mà Finish[i] == false sẽ ở trong

tình trạng tắc nghẽn.

### Hiệu chỉnh tắc nghẽn



Khi đã phát hiện được tắc nghẽn, có hai lựa chọn chính để hiệu chỉnh tắc nghẽn :

**Đình chỉ hoạt động của các tiến trình liên quan**



Cách tiếp cận này dựa trên việc thu hồi lại các tài nguyên của những tiến trình bị kết thúc. Có thể sử dụng một trong hai phương pháp sau :

Đình chỉ tất cả các tiến trình trong tình trạng tắc nghẽn



Đình chỉ từng tiến trình liên quan cho đến khi không còn chu trình gây tắc nghẽn : để chọn được tiến trình thích hợp bị đình chỉ, phải dựa vào các yếu tố như độ ưu tiên, thời gian đã xử lý, số lượng tài nguyên đang chiếm giữ , số lượng tài nguyên yêu cầu...



**Thu hồi tài nguyên**



Có thể hiệu chỉnh tắc nghẽn bằng cách thu hồi một số tài nguyên từ các tiến trình và cấp phát các tài nguyên này cho những tiến trình khác cho đến khi loại bỏ được chu trình tắc nghẽn. Cần giải quyết 3 vấn đề sau:

Chọn lựa một nạn nhân: tiến trình nào sẽ bị thu hồi tài nguyên ? và thu hồi những tài nguyên nào ?



Trở lại trạng thái trước tắc nghẽn: khi thu hồi tài nguyên của một tiến trình, cần phải phục hồi trạng thái của tiến trình trở lại trạng thái gần nhất trước đó mà không xảy ra tắc nghẽn.



Tình trạng « đói tài nguyên »: làm sao bảo đảm rằng không có một tiến trình luôn luôn bị thu hồi tài nguyên ?



1. TÓM TẮT

Trong suốt chu trình sống, tiến trình chuyển đổi qua lại giữa các trạng thái ready, running và blocked.



Bộ điều phối của hệ điều hành chịu trách nhiệm áp dụng một gỉai thuật điều phối thích hợp để chọn tiến trình thích hợp được sử dụng CPU, và bộ phân phối sẽ chuyển giao CPU cho tiến trình này.



Các giải thuật điều phối thông dụng : FIFO, RoundRobin, điều phối với độ ưu tiên, SJF, Multilevel Feedback



Một số tiến trình trong hệ thống có nhu cầu trao đổi thông tin để phối hợp hoạt động, do mỗi tiến trình có một không gian địa chỉ độc lập nên viêc liên lạc chỉ có thể thực hiện thông qua các cơ chế do hệ điều hành cung cấp.



Một số cơ chế trao đổi thông tin giữa các tiến trình :



Tín hiệu : thông báo sự xảy ra của một sự kiện



Pipe : truyền dữ liệu không cấu trúc



Vùng nhớ chia sẻ : cho phép nhiều tiến trình truy cập đến cùng một vùng nhớ



Trao đổi thông điệp : truyền dữ liệu có cấu trúc, có thể vận dụng trong các



hệ phân tán

Socket : chuẩn hoán việc liên lạc giữa các hệ thống khác biệt



Khi các tiến trình trao đổi thông tin, chia sẻ tài nguyên chung, cần phải đồng bộ hoá hoạt động của chúng chủ yếu do yêu cầu độc quyền truy xuất hoặc phối hợp hoạt động.



Miền găng là đoạn lệnh trong chương trình có khả năng phát sinh mâu thuẫn truy xuất. Để không xảy ra mâu thuẫn truy xuất, cần đảm bảo tại một thời điểm chỉ có một tiến trình được vào miền găng.



Các giải pháp đồng bộ hoá do lập trình viên xây dựng không được ưa chuộng vì phải tiêu thụ CPU trong thời gian chờ vào miền găng (« busy waiting »), và khó mở rộng. Thay vào đó, lập trình viên có thể sử dụng các cơ chế đồng bộ do hệ điều hành hay trình biên dịch trợ giúp như semaphore, monitor, trao đổi thông điệp .



Tắc nghẽn là tình trạng xảy ra trong một tập các tiến trình nếu có hai hay nhiều hơn các tiến trình chờ đợi vô hạn một sự kiện chỉ có thể được phát sinh bởi một tiến trình cũng đang chờ khác trong tập các tiến trình này.



Có 3 hướng tiếp cận chính trong xử lý tắc nghẽn :



Phòng tránh tắc nghẽn : tuân thủ một vài nghi thức bảo đảm hệ thống không   
bao giờ lâm vào trạng thái tắc nghẽn.



Phát hiện tắc nghẽn : khi có tắc nghẽn xảy ra, phát hiện các tiến trình liên   
quan và tìm cách phục hồi.



*Bỏ qua tắc nghẽn* : xem như hệ thống không bao giờ lâm vào trạng thái   
tắc nghẽn.

