CHƯƠNG 2. TIẾN TRÌNH

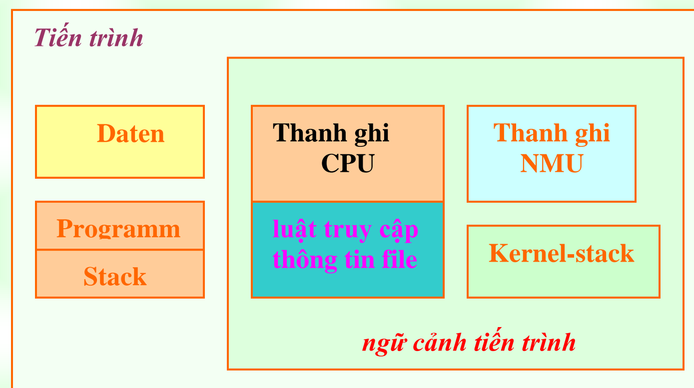
2.0. Quan niệm về tiến trình

Trước đây tuỳ từng thời điểm, máy tính được xác định một nhiệm vụ chính; tất cả các chương trình được bó lại thành gói (paket) và được gởi đi liên tục. Điều đó được gọi là xử lý đóng gói (*pile processing*) hay quản lý lô (*batch manager*). Ngày nay, không chỉ có một chương trình chạy trên máy tính, mà nhiều chương trình cùng thực hiện (*multi-tasking*). Cũng như thế, không chỉ có một người sử dụng làm việc, mà nhiều người sử dụng cùng làm việc (*multi- user*). Để hạn chế sự tranh chấp giữa chúng ở việc dùng máy tính, do đó sự phân bổ các phương tiện điều hành phải được điều chỉnh trên chương trình.

Ngoài ra, điều đó còn tiết kiệm thời gian chạy máy và giảm đáng kể thời gian thao tác. Thí dụ, người ta có thể điều chỉnh sự phân chia bộ vi xử lý chính (Central Processing Unit- CPU) cho việc biểu thị Text song song với việc xử lý Text, điều đó cho thấy rằng, CPU đã trợ giúp việc xử lý Text trong thời gian máy in in ký tự. Nếu điều đó hoàn thiện thì bộ vi xử lý đẩy một ký tự mới cho máy in và tiếp tục việc xử lý Text.

Thêm vào đó, chương trình phải được lưu trữ khi cần thiết sử dụng phương tiện điều hành nào: không gian nhớ, thế hệ CPU, dùng lượng CPU… Từ đó, ta hiểu, ***tiến trình là thông tin trạng thái của các phương tiện điều hành đối với một chương trình*** (thường gọi là **một Job**).

Hình 2.1 minh họa điều trên đây:



Hình 2.1. Sự cấu thành các dữ liệu tiến trình

Một tiến trình này có thể sinh ra một tiến trình khác, khi đó người ta gọi tiến trình đầu là tiến trình cha, còn tiến trình được sinh ra là tiến trình con.

Một hệ thống đa chương trình (*multi-programming system*) cho phép thực hiện đồng thời nhiều chương trình và nhiều tiến trình. Một chương trình (gọi là một job) cũng có thể tự phát sinh ra nhiều tiến trình.

Thí dụ về hệ điều hành UNIX:

Các chương trình hệ thống của Unix được gọi là nền tảng, nó tổng hợp các giải pháp đồng bộ và thích ứng thuận tiện. Sự độc lập của các tiến trình và kể cả các chương trình của hệ điều hành Unix cho phép khởi động đồng thời nhiều công việc.Thí dụ, chương trình **pr** hình thành **Text1,** chương trình **lpr** biểu diễn **Text2** thì người ta có thể kết nối thành chương trình **cat** bằng dòng lệnh sau:

cat Text1 Text2 | pr | lpr

Ở đây, bộ thông dịch, mà người ta sẽ chuyển lệnh cho nó, khởi động ba chương trình với tư cách là ba tiến trình riêng lẻ, mà ở đây ký tự "|" tạo ra một sự thay đổi cho việc xuất ra một chương trình thành việc nhập vào một chương trình khác. Nếu trong hệ thống có nhiều bộ vi xử lý, do đó, mỗi bộ vi xử lý có thể được sắp xếp theo một tiến trình, và quả vậy, sự điều hành được tiến hành song song. Ngoài ra, cũng có khi một bộ vi xử lý chỉ thực hiện một phần tiến trình và dẫn tới bộ tiếp theo.

Ở hệ thống đơn vi xử lý thì luôn chỉ có 1 tiến trình thực hiện, những tiến trình khác được giữ lại và chờ đợi. Điều này sẽ được khảo sát ở các phần dưới.

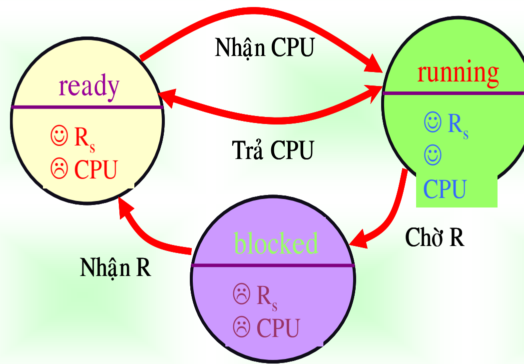
2.1 Các trạng thái tiến trình

Kế tiếp trạng thái hoạt động (*running*) đối với một tiến trình đang diễn ra, chúng ta phải xem xét những tiến trình khác chờ đợi ở đâu. Đối với một trong nhiều khả năng biến cố, nó có một hàng đợi riêng, mà trong đó các tiến trình được phân loại.

Một tiến trình bị hãm phải chờ đợi, để:

+ đón nhận một bộ vi xử lý hoạt động, lúc đó ta có trạng thái sẵn sang (ready),
+ đón nhận một thông tin (massage) của một tiến trình khác,
+ đón nhận tín hiệu của một bộ chỉ thị thời gian (timer),
+ đón nhận những dữ liệu của một thiết bị xuất nhập.

Thực ra, trạng thái sẵn sang rất đặc biệt: tất cả các tiến trình nhận được các thay đổi và được giải hãm, tiếp đến, đầu tiên chúng được chuyển dịch vào trong danh sách sẵn sàng và sau đó, chúng đón nhận bộ vi xử lý ở trong dãy tuần tự.Các trạng thái và sự quá độ của chúng được sơ đồ hoá trên hình 2.2



Hình 2.2.Các trạng thái tiến trình

Ở đây, chúng ta còn quan tâm tới điều, rằng các chương trình và các tiến trình thì không tồn tại vĩnh viễn, mà chúng có thể được sinh ra và kết thúc bất kỳ khi nào. Do đó, từ các lý do bảo vệ, các tiến trình không tự quản lý được, mà chúng được thuyên chuyển từ một chức năng đặc biệt của một hệ điều hành cho bộ định giờ, hay thuyên chuyển từ một trạng thái này thành một trạng thái liền kề. Việc chuyển đổi của các tín hiệu, việc lưu trữ các dữ liệu tiến trình và việc sắp xếp thành các hàng đợi được một chức năng trung tâm hoàn thiện, các chức năng này người sử dụng không trực tiếp điều khiển. Bởi vậy, qua việc gọi hệ điều hành thì những mong muốn của các tiến trình được khai báo, mà những cái đó trong khuôn khổ của việc quản lý các phương tiện điều hành của bộ định thời phù hợp với sự quan tâm đối với người sử dụng khác.

Tất cả các trạng thái chứa đựng một hay nhiều danh sách. Các tiến trình ứng với một trạng thái thì được đưa vào danh sách đó. Điều đã rõ, rằng một tiến trình có thể được luôn luôn chứa đựng chỉ trong một danh sách.

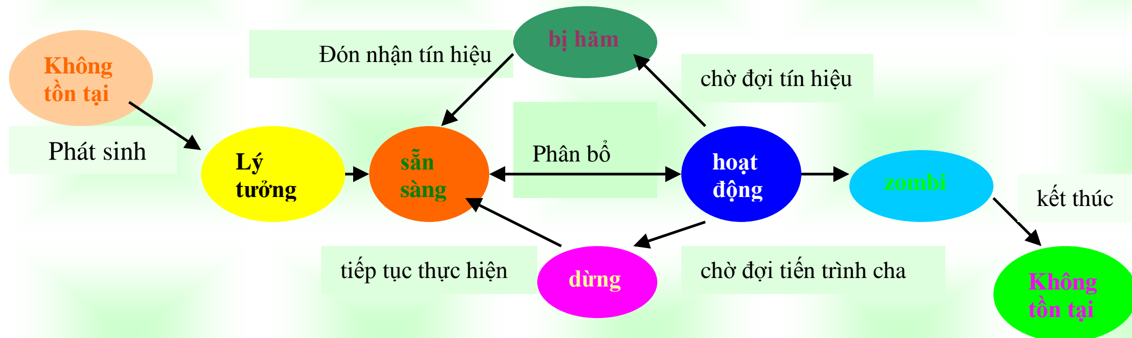
Trong sự khác nhau với mã máy, những dữ liệu trạng thái của phần cứng (CP, FPU, MNU), mà với các tiến trình làm việc, chúng được biểu thị là văn cảnh tiến trình (stask context), xem hình 2.1. Ở một tiến trình hãm, phần dữ liệu chứa đựng trạng thái sau cùng của CPU thì nó như một bản sao của CPU có thể được biểu thị là nột vi xử lý ảo và phải được nạp mới nhờ sự chuyển đổi tới một tiến trình khác cũng như chuyển đổi văn cảnh (context switch).

Những hệ điều hành khác nhau sẽ thu hẹp chỉ số các biến cố và thu hẹp số lượng cũng như kiểu hàng đợi. Điều đó cũng được phân biệt, rằng những giao thức nào chúng dự định cho việc bắt đầu và kết thúc của bộ vi xử lý cũng như việc phân chia và sắp xếp danh sách chờ. Ở đây, người ta còn phân biệt giữa việc đặt kế hoạch phân bổ các phương tiện điều hành (scheduling) và việc phân bổ trên thực tế (dispatching).

2.1.1. Thí dụ về Unix

Trong hệ điều hành Unix có sáu trạng thái khác nhau. Có ba trạng thái đã nhắc tới ở trên. Đó là trạng thái *running*(SRUN), trạng thái *blocked* (SSLEEP) và trạng

thái *ready* (SWAIT). Trạng thái tiếp theo là trạng thái *stopped* (SSTOP), mà một cái gì đó phù hợp với sự chờ đợi của các tiến trình cha ở việc tìm lỗi (*tracing anh debugging*).



++++++++++++++++++++

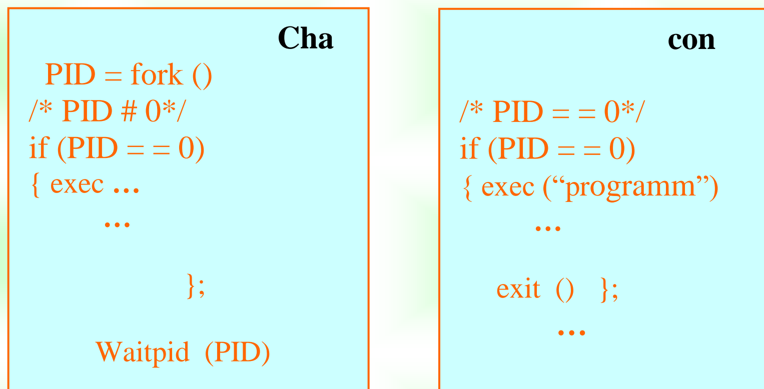
Hình 2.3.Những trạng thái tiến trình và sự quá độ ở Unix.

Ngoài ra còn tồn tại những trạng thái trung gian phụ như trạng thái *idle* (SIDL) và trạng thái *zombie* (SZOMB), mà chúng xuất hiện bởi việc sinh ra và kết thúc của một tiến trình. Sự quá độ trạng thái có những hình thái như trong hình vẽ 2.3 ở trên.

Sự quá độ của một trạng thái tới một trạng thái kế tiếp đạt được qua sự thăm dò gọi hệ thống. Thí dụ, nếu có một tiến trình gọi hàm *fork()*, do đó một bản sao một tiến trình được lôi ra và đem treo vào danh sách sẵn sàng. Với điều đó, bây giờ ta có hai tiến trình đồng nhất, mà cả hai trở lại từ việc gọi hàm fork(). Sự khác nhau giữa hai tiến trình là ở chỗ giá trị trả lại của hàm: tiến trình cha đón nhận chỉ số tiến trình (PID) của con; khi PID = 0 thì nó nhận ra rằng, đó là tiến trình con và nó thể hiện sự diễn biến tiếp tục của chương trình qua lần gọi hệ thống khác. Đối với các chương trình thực thi (execute) có thể nhận thấy rằng, chương trình chạy sẽ quá tải bởi mã chương trình. Tất cả các thiết bị hiển thị và các biến số được kích thích (thí dụ sử dụng bộ đếm địa chỉ gọi hệ thống của chương trình) và tiến trình hoàn tất được treo vào danh sách sẵn sang. Ở hiệu quả cuối cùng của tiến trình cha thì một chương trình hoàn toàn mới được khởi động.

Tiến trình cha có khả năng chờ đợi hàm gọi hệ thống exit() và chờ đợi sự kết thúc của tiến trình con với hàm waitpid(PID). Trong hình 2.4 chỉ ra quá trình phát sinh một tiến trình như vậy.

Người ta quan tâm rằng, tiến trình con đạt được hàm gọi hệ thống exit() như nói ở trên chỉ khi, nếu một lỗi xuất hiện tại hàm exec(). Điều đó có nghĩa, nếu tệp tin *programm* không tồn tại , thì nó không thể đọc được. Ngoài ra, lệnh của chương trình kế tiếp theo hàm exec() ở trạng thái người sử dụng thì giống hệt với lệnh đầu tiên của chương trình ‘*programm*’.



Hình 2.4.Sự phát sinh và loại trừ một tiến trình ở hệ điều hành Unix

Tiến trình con kết thúc chỉ khi, nếu như trong '*programm*' một hàm gọi exit() tự đạt tới.

Với suy nghĩ này, thí dụ sau đây sẽ làm sáng tỏ một tiến trình đối với sự thỉnh cầu của người sử dụng ở thiết bị đầu cuối. Tuy nhiên, mã (nói ở trên) chỉ là cơ sở cho việc thỉnh cầu đó ở trong Unix để mỗi người sử dụng khởi động shell.

Thí dụ shell của Unix:

```
LOOP
   Write(prompt);                        (*thí dụ có dạng :>*)
   ReadLine(command, params);    (*đọc chuỗi, phân cách qua ý tự trống *)
   pid := fork();                        (*tái bản của tiến trình này*)
   IF  (pid=0)
      THEN execve(command, params,0)            (*con chờ tải Programm*)
      ELSE waitpid(-1, status, 0)          (*cha chờ sự kết thúc của con*)
  END;
END;
```

Tất cả các tiến trình trong Unix thích hợp với tiến trình khởi đầu (PID =1). Nếu ở sự chấm dứt của một tiến trình con mà không có một tiến trình cha nào tồn tại nữa, khi đó tiến trình khởi đầu nói trên được thông báo. Trong khoảng thời gian gọi hệ thống với hàm exit() và sự tiếp nhận các thông tin tại tiến trình cha, thì tiến trình con đạt được một trạng thái đặc biệt gọi là "*zombi*" (xem hình 2.1).

Văn cảnh tiến trình nội bộ (*intern process context*) được phân thành hai phần: Phần thứ nhất là phần mang tiến trình ở trong một bảng nhớ trú ngụ, nó thì rất quan trọng đối với việc điều khiển tiến trình và do đó nó luôn luôn tồn tại. Phần thứ hai gọi là phần cấu trúc người sử dụng (user structure), nó chỉ quan trọng, nếu nó là tiến trình hoạt động và nếu nó có thể được xuất ra trên bộ nhớ quảng đại với mã còn lại và các dữ liệu.

Thực chất hai phần kể trên là:

Các khối điều khiển tiến trình của bảng tiến trình (process control bock- PCB)

+ Thông số định giờ

+ Những tham chiếu nhớ: địa chỉ mã, địa chỉ dữ liệu, địa chỉ ngăn xếp ở bộ nhớ chính cũng như bộ nhớ quảng đại.

+ Các dữ liệu tín hiệu: mặt nạ, trạng thái.

+ Những điều khác: trạng thái tiến trình, biến cố chờ đợi, trạng thái định thời, PID, PID cha, người sử dụng.

- Văn cảnh người sử dụng (user context):

+ Trạng thái bộ vi xử lý: thanh ghi, thanh ghi FPU…

+ Gọi hệ thống: thông số…

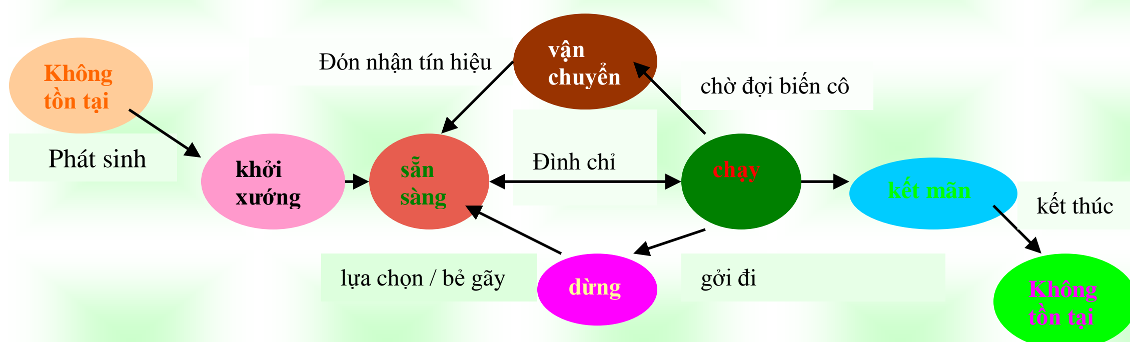
+ Bảng thông tin file

+ Ngăn xếp nhân: không gian ngăn xếp đối với gọi hệ thống của tiến trình.

Khác biệt với PCB là tiến trình có thể thay đổi và kiểm tra chỉ gián tiếp qua gọi hệ thống, cho phép gọi hệ thống Unix để kiểm tra trực tiếp cấu trúc người sử dụng và để thay đổi các phần

2.1.2. Thí dụ về Windows NT

Vì trong Windows NT phải được được các loại tiến trình khác nhau trợ giúp, mà những tiến trình đó không hạn chế sự phát sinh đa dạng, cho nên chỉ đối với một loại riêng lẻ của các tiến trình (đối tượng xâu: *thread object*) thì một hệ thống tiến trình được tạo nên. Việc phát sinh các đối tượng (như OS/2, POSIX,*Windows32*) thì được liên hợp lại thành các đối tượng và ở sự thay đổi trạng thái của chúng không đóng vai trò gì cả. Sơ đồ đơn giản hoá các quá độ trạng thái được chỉ ra trong hình 2.5.



Hình 2.5.Các trạng thái tiến trình của Windows NT

Việc sản sinh tiến trình ở Windows NT thì phức tạp hơn trong Unix, vì để có sự chuyển giao thì nhiều trạng thái tiến trình phải được thực hiện. Do đó, những sự phát sinh đặc biệt được liên kết trong những hệ thống con.

Để sản sinh ra các tiến trình thì chỉ có duy nhất một hàm gọi hệ thống *NtCreateProcess()*, ở đấy, bên cạnh sự kích thích nhờ các mà thì còn có tiến trình cha có thể được thông báo. Trên cơ sở đó, tất cả các biến gọi hệ thống con khác được thiết lập, mà cái đó sẽ được người sử dụng quan tâm và cần tới.

Thật vậy, cái đó đã tạo ra cơ cấu của hàm gọi POSIX-*fork()*. Thí dụ, chương trình POSIX (hay tiến trình POSIX) gọi lệnh với hàm *fork()* qua giao diện người lập trình ứng dụng (*Application Programming Interface*). Cái đó sẽ được chuyển đổi thành một thông tin và được gởi tới một hệ thống con POSIX qua nhân hệ thống (xem hình 1.7). Cái đó trở lại gọi hàm *NtCreateProcess()* và thông báo chương trình POSIX cho PID cha. Chìa khoá đối tượng (*object handle*) được trao trở lại hệ thống con POSIX quản lý; tất cả gọi hệ thống của tiến trình POSIX, mà nó đưa ra thông tin tới hệ thống con POSIX, thì được hoàn thiện ở đó với sự trợ giúp của gọi hệ thống của Windows NT và đưa kết quả có dạng POSIX trở lại tiến trình gọi. Tương tự, điều đó cũng dẫn tới gọi tiến trình của các hệ thống con khác.

2.1.3. Các tiến trình trọng lượng nhẹ.

Nhu cầu lưu trữ của một tiến trình thì rất toàn diện. Nó chứa đựng không chỉ vài con số, như số tiến trình và các dữ liệu,mà cả những thông báo về các files thông thường như các mã chương trình và các dữ liệu của chúng. Điều đó có hầu hết ở các tiến trình, khi nó thích ứng ở trong bộ nhớ chính. Cho nên, tiến trình chiếm rất ít không gian trên bộ nhớ quảng đại (chẳng hạn *harddisk*). Vì có sự chuyển đổi tiến trình, bộ nhớ hiện tại bị tiêu tốn (chiếm chỗ), còn bộ nhớ trước đó của đĩa cứng được phục hồi trở lại, do đó một sự thay đổi tiến trình đều làm cho tải hệ thống nặng nề và thời gian thực hiện tương đối dài.

Ở nhiều ứng dụng thì không có tiến trình mới được sử dụng, mà chỉ có những đoạn mã độc lập (*threads*) được sử dụng. Những đoạn mã độc lập này được mô tả bằng văn cảnh tiến trình (thí dụ các thủ tục của một chương trình). Trường hợp này người ta gọi là đồng lập thức (*coroutine*).

Việc ứng dụng các đoạn mã theards có điều kiện để tạo trong một khoảng tiến trình bởi một hệ thống tiến trình tiếp theo mà người ta gọi là các tiến trình trọng lượng nhẹ (*light weight process*: LWP). Với hình dạng đơn giản nhất thì những tiến trình này tự chuyển đổi sự điều khiển một cách dứt khoát, mà người gọi là bản phác thảo đồng lập thức (*coroutine concept*). Có lý do để nói rằng, những tiến trình mới này cũng là những tiến trình gọi hệ thống. Nếu mỗi tiến trình mà càng sinh ra nhiều tiến trình khác, thì điều đó càng khó khăn hơn. Từ lý do đó, người ta có thể dẫn ra đây một bộ định thời, mà bộ định thời này luôn luôn chứa đựng sự điều khiển và sự điều khiển này được chuyển tiếp tục tới một tiến trình kế tiếp trong danh sách sẵn sang của nó. Nếu điều đó không được lập trình bởi người sử dụng, thì nó đã được chứa đựng trong hệ điều hành qua việc gọi hệ thống. Do đó,

qua thời gian chuyển đổi của gọi hệ thống thì các tiến trình threads sẽ là tiến trình trọng lượng nặng (*heavy weight process*: HWP).

Mỗi tiến trình đều phải thâu giữ các dữ liệu riêng của nó một cách độc lập với các tiến trình khác. Điều đó thì cũng thuận với tiến trình trọng lượng nhẹ: Nếu chúng phân bổ các files đồng đều (nói chính xác là vùng địa chỉ ảo đồng đều, xem chương 3) với các tiến trình trọng lượng nhẹ khác. Do vậy, hầu hết các ngăn xếp của nó được sử dụng, mà ngăn xếp này được dữ trữ không gian để phát sinh cho mỗi tiến trình. Trong sự khác biệt với các tiến trình xác thực, thì do đó, các tiến trình trọng lượng nhẹ sử dụng chỉ ít các dữ liệu văn cảnh (context data), mà các dữ liệu này phải được thay đổi khi chuyển đổi. Từ đó, trạng thái vi xử lý (processor-status: PS) và con trỏ ngăn xếp (stack-pointer:SP) là những thứ quan trọng nhất. Còn, tự bản thân bộ đếm chương trình (programm-counter) có thể được tách khỏi ngăn xếp, do đó, nó không phải chuyển giao một cách rõ ràng. Bằng ngôn ngữ Assemble, việc chuyển đổi được thực thi một cách hiệu nghiệm và làm cho việc gọi hệ thống của các tiến trình này xảy ra rất nhanh.

2.1.4. Trạng thái tiến trình ở Unix

Ở hệ điều hành Unix, các tiến trình trọng lượng nhẹ được thực thi bởi thư viện của người sử dụng và bằng ngôn ngữ C hay C++ (xem phần Unix ở chương 3). Tuỳ theo sự thực thi, mà hoặc là có một hệ thống đơn giản với việc chuyển giao điều khiển một cách trực tiếp, hoặc là có một hệ thống phức tạp hơn với bộ định thời đặc biệt (xem mục 2.2).

Lợi thế của việc thực thi bằng thư viện là tồn tại một sự chuyển đổi rất nhanh, vì các cơ cấu gọi hệ điều hành và các cơ cấu giải mã của chúng sẽ không có điều kiện thực hiện theo số dịch vụ và theo các thông số. Còn nhược điểm của nó là tiến trình thread phải chờ đợi một biến cố (thí dụ biến cố vào/ra) và nó chặn tiến trình tổng thể lại.

Có những thí nghiệm để tiêu chuẩn hóa các tiến trình threads và để giảm nhẹ sự thực thi chương trình (xem chuẩn IEEE năm 1922)

Ở các phiên bản mới nhất của Unix, chúng chứa đựng loại 64bit –Unix, còn gọi là Unix-98.

2.1.5. Trạng thái tiến trình ở Windows NT

Khác với Unix, trong hệ điều hành Windows NT, các tiến trình trọng lượng nhẹ LWP được thực thi với chức năng gọi hệ điều hành. Tuy nhiên, sự chuyền đổi chậm chạp hơn, nên được gọi là tiến trình trọng lượng nặng (*heavy weight thread*), nhưng nó vẫn có ưu điểm. Đó là, người lập trình hệ thống có một giao diện kết nối chắc chắn. Nó làm giảm nhẹ sự thực thi chương trình, vì chúng được sử dụng các tiến trình LWP và nó cũng tránh được việc thực nghiệm để phát triển những hệ thống lệch lạc riêng lẻ như đối với Unix. Một điều khác nữa là nhân của hệ điều hành cũng được điều khiển qua các tiến trình LWP.Ở đây, điều cần phải lưu ý là,

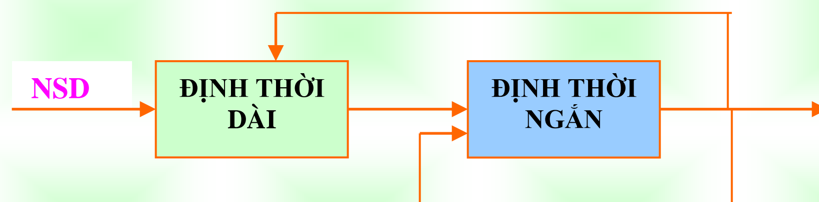
các tiến trình LWP được thực hiện song song ở trong hệ thống đa vi xử lý và đối với biến cố I/O thì chỉ có tiến trình thread ngăn hãm chỉ một tiến trình.

Vì một tiến trình thread trọng lượng nặng dẫn tới việc thu hẹp không cần thiết những cái đang cần thiết sử dụng, do đó, trong Windows NT với version 4.0 được dẫn vào trạng thái các files. Đó là những thủ tục được tiến hành song song, mà những thủ tục đó được hoạt động theo bản phác thảo đồng lập thức: Sự chuyển đổi của một tiến trình fiber (thớ) tới một tiến trình thread khác được thực hiện một cách tự do. Nếu tiến trình thread bị ngăn hãm, do đó tất cả các tiến trình fiber cũng bị ngăn hãm tương tự. Điều đó cũng giảm nhẹ việc thực thi các chương trình như trên hệ thống Unix.

2.2 Định thời tiến trình

Nếu ở một hệ điều hành có nhiều nhu cầu về phương tiện điều hành, khi đó, việc truy cập phải được phối hợp. Thật vậy, đóng vai trò quan trọng là bộ định thời đã nói ở trên và các giao thức của nó ở việc sắp xếp các tiến trình theo hàng chờ. Nếu chúng ta khảo sát hệ thống đơn vi xử lý, thì sẽ thấy trên đó các tiến trình độc lập làm việc một cách tuần tự (*sequemtiell*).

Trong hệ thống tính toán thông thường, chúng ta có thể phân biệt ra hai loại nhiệm vụ định thời: định thời dự định việc thực hiện Job (còn gọi là định thời dài cho Job) và dự định việc phân bổ bộ vi xử lý hoạt động (còn gọi là định thời ngắn). Ở việc định thời dài, người ta phải lưu ý:(1). Khi mà có nhiều người sử dụng được phép đi vào hệ thống (*login*) với công việc của họ, khi ra (*logout*) người sử dụng phải báo như thế nào đó; (2). Nếu trong hệ thống có người sử dụng quá nhiều, thì việc dẫn vào phải được chặn lại cho đến khi tải hệ thống chất đầy.



Hình 2.6. Định thời dài và định thời ngắn

Tuy nhiên ở việc định thời ngắn, công việc chính là phải dẫn ra giao thức để điều phối bộ vi xử lý ở các tiến trình. Sau đây, chúng ta sẽ khảo sát một giao thức thông dụng nhất.

2.1.1 Tranh chấp mục đích

Tất cả các giao thức định thời là để thực hiện những mục đích nào đó. Người ta thấy có những mục đích thông dụng sau đây:

- *Khả năng chịu tải của CPU*:

Nếu CPU là phương tiện điều hành, thì ít nhất, chúng ta muốn thể hiện sự sử dụng hiệu nghiệm nhất. Mục đích là CPU tải 100%, thông thường chỉ tải khoảng 40-90%.

- *Lưu lượng* (*througput*):

Số công việc trên một đơn vị thời gian được gọi là lưu lượng, nó chính là mức độ chịu tải của hệ thống.

- *Cách điều khiển thật:*

Không có công việc nào ưu tiên hơn việc khác, khi chưa được thoả thuận đích xác. Điều đó có ý nghĩa rằng, mỗi một người sử dụng nhận được các phương tiện một cách đồng đều trong thời gian truy cập CPU.

- *Thời gian thực hiện:*

Thời gian thực hiện (*turnround time*) là khoảng thời gian từ khi bắt đầu Job cho tới khi kết thúc Job, nó chứa đựng tất cả thời gian trong các hàng đợi, thời gian thực hiện và thời gian xuất nhập. Tất nhiên chúng phải là tối thiểu.

- *Thời gian chờ đợi:*

Trong khoảng thời gian tổng cộng, bộ định thời chỉ ảnh hưởng tới thời gian chờ ở trong danh sách ready (sẵn sàng). Đối với giao thức định thời, người ta có thể giới hạn mục đích để làm giảm thời gian chờ.

- *Thời gian trả lời:*

Ở sự hoạt động bên trong của hệ thống, người sử dụng cảm thấy đặc biệt không dễ chịu, vì sau một sự truy nhập nào đó, người ta phải chờ đợi lâu phản ứng của máy tính. Một cách độc lập với thời gian tổng cộng thực hiện Job, thời gian giữa việc nhập vào và việc chuyển giao dữ liệu trả lời thì được gọi là thời gian trả lời.

Danh sách của việc chuyển giao mục đích không những phải đầy đủ mà còn phải chặt chẽ. Thí dụ, mỗi một sự chuyển đổi tiến trình thì cần có một sự thay đổi văn cảnh tiến trình (*context switch*). Những tiến trình ngắn thì được ưa chuộng hơn, bởi vì thời gian trả lời được rút ngắn- đó là thời gian giữa hai lần truy nhập, nhờ vậy năng suất được gia tăng. Ngược lại, các tiến trình chậm thì không được ưa chuộng. Mặc khác, nếu khả năng chịu tải được nâng cao, thì do diễn biến bên trong của Job, thời gian trả lời sẽ kéo dài.

Tương tự, trong đời sống thường nhật, người ta có thể nhìn thấy điều đó: Thí dụ ở việc cho thuê ô tô, những khách hàng xác định sẽ được dịch vụ thuận tiện, mặc dụ chật chội, còn những khách hàng khác phải chờ đợi lâu hơn. Nếu muốn thuê một chiếc ô tô chạy tốt, thì một khách hàng mới tới phải đợi cho đến khi anh ta nhận được chiếc ô tô thích muốn đó. Đối với một thời gian phản ứng ngắn, thì khi có nhiều ô tô cùng được đưa vào sử dụng.

Vì đối mỗi một nhóm người sử dụng thì sự nhượng bộ mục đích có thể được thay đổi, nếu không có thuật toán định thời lý tưởng đối với mỗi tình huống. Trên cơ sở này, có rất nhiều phương hướng để tách chia cơ cấu định thời thành các

giao thức định thời riêng lẻ và thành các thông số của chúng. Thí dụ, một tiến trình của ngân hàng dữ liệu phát sinh một vài tiến trình trợ giúp, thì nó sẽ nhận biết đặc trưng của sự trợ giúp đó và vì thế, tạo ra khả năng để ảnh hưởng tới giao thức định thời của các tiến trình con qua các tiến trình cha. Những bộ phận của nhân hệ điều hành, các cơ cấu định thời bên trong và cơ cấu điều phối cần thiết được dùng nhờ giao diện đã được chuẩn hoá (tức là nhờ gọi hệ thống). Giao thức định thời sẽ chỉ có thể có được do người sử dụng lập trình.

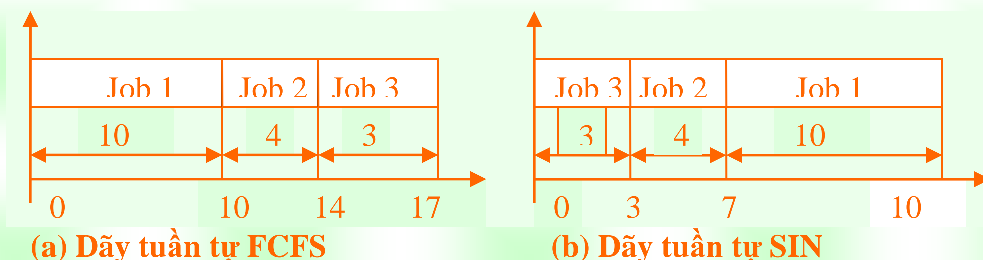
2.2.2. Định thời không có ưu tiên trước.

Trong trường hợp đơn giản, các tiến trình có thể chạy thật lâu cho đến khi rời khỏi trạng thái hoạt động và chờ đợi một biến cố (I/O hoặc một thông tin) hoặc trao việc điều khiển cho tiến trình khác, rồi tự kết thúc: nghĩa là chúng không được ngắt khỏi quá sớm. Trường hợp này được gọi là định thời không có ưu tiên trước. Loại định thời này rất có lợi đối với tất cả các hệ thống, mà ở đây người ta phải hiểu chính xác là những tiến trình nào tồn tại và chúng có những đặc trưng nào. Thí dụ, có một chương trình ngân hàng dữ liệu, người ta phải hiểu chính xác: một cách thông thường, một sự dàn xếp nào để chương trình thực thi thôi qua trong bao lâu (?). Trong trường hợp này, người ta có thể sử dụng một hệ thống tiến trình trọng lượng nhẹ để thực hiện. Đối với loại định thời này, những chiến lược sau đây thường được sử dụng nhất:

- *Chiến lược đến trước dịch vụ trước (First Com First Serve: FCFS):*

Một chiến lược đơn giản loại này thì bao gồm các tiến trình được sắp xếp theo thứ tự xuất hiện ở trong hàng đợi. Tất cả các tác vụ xảy ra theo tuần tự, mà không cần biết, chúng cần bao nhiêu thời gian. Cho nên việc thực thi giao thức này với hàng đợi FCFS thì rất đơn giản.

Tuy nhiên, hiệu quả của thuật toán này thì rất giới hạn. Chúng ta giả định, chúng ta có 3 Job với chiều dài 10, 4 và 3. Các Job được sắp xếp và làm việc theo giao thức FCFS. Hình 2.7 mô tả điều đó.



(a) Dãy tuần tự FCFS **(b) Dãy tuần tự SIN**

Hình 2.7. Dãy tuần tự các Job

Thời gian thực hiện của Job1 là 10, của Job 2 là 14 và của Job3 là 17, vậy thời gian thực hiện trung bình là (10+14+17): 3= 13,67. Tuy nhiên, chúng ta có thể sắp

xếp lại các Job này theo kiểu: Job có chiều dài ngắn nhất làm việc đầu tiên, xem hình (b). Khi đó, ta có thời gian thực hiện trung bình ngắn hơn (3+7+17):3=9.

- *Chiến lược đầu tiên Job ngắn nhất (Shortest Job First: SJF):*

Tiến trình có thời gian dịch vụ ngắn nhất được chuộng hơn các tiến trình khác. Nghĩa là chiến lược loại này tránh được các nhược điểm nói trên. Thật vậy, những tiến trình hoạt động nội bộ thì cần thời gian CPU ít và hầu như việc chờ đợi sự kết thúc của các hoạt động diễn ra song song cùng các kênh xuất nhập. Do đó, thời gian trả lời trung bình được giảm đáng kể.

Người ta có thể chỉ ra rằng, giao thức SJF đã giảm thiểu đáng kể thời gian chờ đợi trung bình của từng Job trong dãy các Job. Vì theo nguyên tắc ưu tiên Job ngắn, thì thời gian chờ đợi của nó giảm đi rất mạnh, trong khi đó thời gian chờ của Job dài tăng lên.

Ở loại giao thức này vẫn còn tồn tại một vấn đề: Tại dòng vào lớn của các tiến trình ngắn và với rất nhiều yêu cầu của CPU, tuy rằng một tiến trình không bị hãm chặn, nhưng nó vẫn không đón nhận CPU.. Điều này được gọi là sự làm đói (*starvation*). Đó là một vấn đề quen thuộc, mà nó cũng hay xuất hiện ở nhiều hoàn cảnh khác nhau nữa.

- *Chiến lược tỷ lệ kế cận đáp ứng cao nhất (Hightest Response Ratio Next: HRN):*

Ở đây, các Job được làm việc theo một tỷ lệ mong muốn, mà trong đó, những nhận xét và phân tích về thời gian đáp ứng và thời gian dịch vụ được sử dụng để làm cơ sở cho việc đánh giá đo đạc trước đó. Chiến lược này chỉ chú ý các Job có thời gian dịch vụ ngắn, nhưng mà nó giới hạn thời gian chờ của các Job có thời gian dịch vụ dài, vì ở một sự thiệt hại đáng kể, thời gian đáp ứng của chúng bị kéo dài.

- *Chiến lược định thời có ưu tiên trước (Priority Scheduling: PS):*

Mỗi một tiến trình sẽ chiếm dụng một sự ưu tiên. Nếu một tiến trình mới đi vào hàng đợi, do đó nó sẽ được sắp xếp, rằng những tiến trình có sự ưu tiên cao nhất sẽ đứng đầu hàng chờ; những tiến trình có ít sự ưu tiên đứng cuối. Nếu có nhiều tiến trình có sự ưu tiên như nhau, thì dãy tuần tự trong các tiến trình này phải được quyết định theo một chiến lược khác thí dụ chiến lược FCFS.

Người ta cũng lưu ý rằng, những Job bị làm tổn thất thì có thể tiếp tục làm đói. Ở việc định thời có ưu tiên trước, vấn đề này cần phải được nhìn bao quát, rằng việc ưu tiên là không cố định, mà nó là một quá trình động. Nếu một tiến trình nhận được sự ưu tiên trong một dãy hợp lý, do đó nó sẽ có sự ưu tiên cao nhất bất kỳ khi nào và cũng như thế, nó nhận được CPU.

Sự giả định của các chiến lược SJF và HRN được thiết đặt bằng các câu hỏi viện cớ, rằng thời gian thực hiện của các Job thì không thống nhất và thường hay thay đổi. Do đó, lợi thế của các giao thức ở các hệ thống khác nhau bị hạn chế. Điều đó thì khác với trường hợp của các Job thường hay xuất hiện, các Job có thể

được nhìn bao quát và các Job quen thuộc, tức là những Job tồn tại trong ngân hàng dữ liệu (datenbank) hay các hệ thống tiến trình (chỉ đối với hệ thống thời gian thực). Ở đây, điều có lợi là để nhận xét các tham số quen thuộc một cách thường xuyên mới mẻ và để tối ưu việc định thời.

Ở việc làm thích hợp thường xuyên các tham số (như thời gian thực hiện và thời gian dịch vụ) và ở việc thực thi, người ta có thể đạt được với các thuật toán khác nhau. Một trong các thuật toán nổi tiếng, đó là: Với tham số a của một tiến trình tại một thời điểm t, thì từ giá trị tức thời b_t và giá trị trước đó a(t), người ta xác định giá trị trung bình theo biểu thức sau:

$$a(t+1) = (1-\alpha)\, a(t) + \alpha b_t$$

Ở đây, nó sẽ được diễn giải như sau:

$$a(0) = b_0$$
$$a(1) = (1-\alpha)\, b_0 + \alpha b_1$$
$$a(2) = (1-\alpha)^2 b_0 + (1-\alpha)\alpha b_1 + \alpha b_2$$
$$a(3) = (1-\alpha)^3 b_0 + (1-\alpha)^2 \alpha b_1 + (1-\alpha)\alpha b_2 + b_3$$
$$\ldots$$
$$a(n) = (1-\alpha)^n b_0 + (1-\alpha)^{n-1}\alpha b_1 + \ldots + (1-\alpha)^{n-i}\alpha b_i + \ldots + \alpha b_n$$

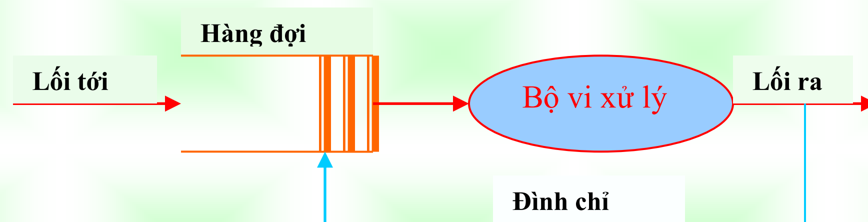
Người ta thấy rằng, với a<1, ảnh hưởng của việc đo đạc sớm sẽ giảm đi theo hàm số mũ. Nguyên tắc này cũng được tìm thấy ở nhiều phương pháp thích ứng khác. Ý nghĩa hạn hẹp của việc đo sớm sẽ tạo nên sự thay đổi bản chất của tiến trình và điều này cũng được nhìn thấy ở trong sự phân tích các tham số tức thời. Khi đó người ta nhận thấy dãy các giá trị tham số như là những biến cố của một biến ngẫu nhiên. Điều đó có ý nghĩa rằng, sự phân chia của biến này không phải là hằng số, mà thực ra, nó thay đổi theo thời gian. Một thuật toán thích ứng như thế không dễ xác định giá trị trung bình của đo đạc (giá trị tham số chờ đợi) mà nó chỉ đánh giá trạng thái tức thời của chức năng phân bổ phụ thuộc thời gian. Đối với trường hợp $\alpha = 1/2$ thì thuật toán được thực thi đặc biệt nhanh. Ở đây, phép chia 2 thì phù hợp với cấp bậc một phép xê dịch con số sang phải một vị trí.

Sự đánh giá thích ứng các tham số của một tiến trình đối với một thuật toán định thời (tức là việc phân bộ vi xử lý thích ứng) phải được thực hiện cho mỗi tiến trình một cách đích thực. Trong thí dụ ở trên, tham số a phải nhận hai chỉ số: một chỉ số cho số tham số trên một tiến trình và một chỉ số cho số tiến trình. Phương pháp đánh giá các tham số thì độc lập với thuật toán, vì thuật toán chỉ được dùng cho việc định thời. Ngoài ra, thuật toán này không chỉ được dùng cho việc định thời không có ưu tiên trước, nó còn là phương pháp để nghiên cứu việc định thời có ưu tiên trước.

2.2.3. Định thời có chặn trước (*preemptive scheduling*)

Ở hệ thống có nhiều người sử dụng sẽ có nhiều Job của nhiều người sử dụng cùng khởi động, khi đó sẽ có điều không vừa ý, nếu có một Job hãm chặn các Job khác. Do đó, đòi hỏi phải có một kiểu định thời khác, để ở đó mỗi Job có thể được ngắt hãm sớm.

Một trong các chiến lược quan trọng là chiến lược nói về việc phân chia khoảng thời gian sử dụng các phương tiện điều hành (chẳng hạn CPU) thành các khoảng thời gian riêng lẻ và bằng nhau. Nếu tiến trình đó là tiến trình sẵn sang thì nó sẽ được sắp xếp một vị trí thích hợp trong một hàng đợi theo một chiến lược. Ở việc khởi đầu một khoảng thời gian, bộ điều phối sẽ cho một ngắt thời gian được gọi, cho đến khi tiến trình được thực hiện, thì nó bị chặn lại và một tiến trình ready mới sẽ được xếp vào hàng đợi. Sau đó, tiến trình đầu tiên của hàng đợi được chuyển vào trạng thái hoạt động. Điều đó được trình bày ở trong hình 2.8 dưới đây.



Hình 2.8. Định thời có chặn trước

Ở đây, đường thẳng góc đậm tượng trưng cho các tiến trình, mà nó được dịch chuyển vào từ trái sang trong ống hàng đợi. Bộ phận công tác - ở đây là bộ vi xử lý- được biểu thị tượng trưng hình ê-líp. Sau một sự ngắt đoạn, tiến trình được xếp một vị trí giữa các tiến trình khác trong hàng đợi. Dưới đây sẽ khảo sát các chiến lược định thời khác nhau.

- *Chiến lược quay tròn Robin (Round Robin: RR):*

Chiến lược đơn giản nhất ở phương pháp lát cắt thời gian là chiến lược FCFS và hàng đợi FIFO (vào trước ra trước). Sự kết hợp giữa chiến lược là phương pháp lát cắt thời gian được gọi là thuật toán quay vòng Robin. Việc phân tích này chỉ ra rằng, ở đây, các thời gian đáp ứng thì tỷ lệ với thời gian dịch vụ, nó độc lập với sự phân bổ thời gian dịch vụ và chỉ phụ thuộc vào thời gian dịch vụ trung bình.

Điều đã rõ, hiệu suất của chiến lược RR thì phụ thuộc mạnh vào lát cắt thời gian. Nếu người ta chọn lát cắt thời gian không kết thúc lâu, thì do đó chỉ còn giao thức đơn giản FCFS được thực hiện. Ngược lại, nếu người ta chọn lát cắt thời gian rất nhỏ (thí dụ đúng bằng một lệnh), do đó, tất cả n Job đón nhận mỗi lần chừng 1/n hiệu suất bộ vi xử lý; bộ vi xử lý thì phân thành n bộ vi xử lý ảo. Tuy nhiên, điều đó chỉ xảy ra khi, nếu bộ vi xử lý chạy rất nhanh so với các thiết bị ngoại vi

(thí dụ bộ nhớ) và việc chuyển đổi tiến trình nhờ cơ cấu phần cứng được thực hiện rất nhanh. Đối với các hệ thống chuẩn thì điều đó không còn đúng nữa. Ở đây, sự chuyển đổi văn cảnh tiến trình xảy ra qua cơ cấu phần mềm và sử dụng một khoảng thời gian để sắp xếp thô áng chừng 10 đến 100 µs. Nếu chúng ta chọn lát cắt thời gian quá ngắn, do đó sẽ có tỷ số quan hệ giữa thời gian làm việc và thời gian chuyển đổi rất nhỏ. Bởi vậy, năng suất giảm và thời gian chờ đợi gia tăng. Trong trường hợp tại thời điểm cực trị, bộ vi xử lý chỉ chuyển đổi, nhưng không thực thi Job.

Đối với một sự tương quan hợp lý giữa giao thức FCFS và chu kỳ chuyển đổi, thì sự hiểu biết các tham số khác nhau là rất cần thiết. Ở đây, quy tắc số 1 được chỉ ra: lát cắt thời gian phải lớn hơn nhu cầu trung bình của CPU giữa hai lần truy cập I/O (CPU - burst) khoảng 80% của Job, tức là nó phù hợp với một giá trị khoảng 100ms.

- *Chiến lược quay vòng Robin có ưu tiên động: (Dynamic Priority Round Robin:DPRR*)

Định thời kiểu RR đối với một Job được làm đầy đủ thêm nhờ tầng đầu tiên của hàng đợi có ưu tiên. Sự ưu tiên của tiến trình trong tầng đầu tiên được thay đổi sau mỗi lát cắt thời gian, kéo dài cho tới khi có đạt sự ưu tiên bung ra theo phương pháp RR riêng lẻ và rồi nó được sắp xếp vào hàng đợi chính. Do đó, một sự xử lý khác nhau của Job sẽ đạt được theo ưu tiên hệ thống, mà vẫn không làm thay đổi trực tiếp phương pháp RR.

- *Chiến lược thời gian còn lại ngắn nhất ở trước (Shortest Remaining Time First*):

Ở đây, chiến lược SJF để sắp xếp hàng đợi có ý nghĩa rằng, Job phải được phân tích để biểu thị Job có thời gian dịch vụ còn lại nhỏ nhất. (xem giao thức SJF được trình bày ở phía trước).

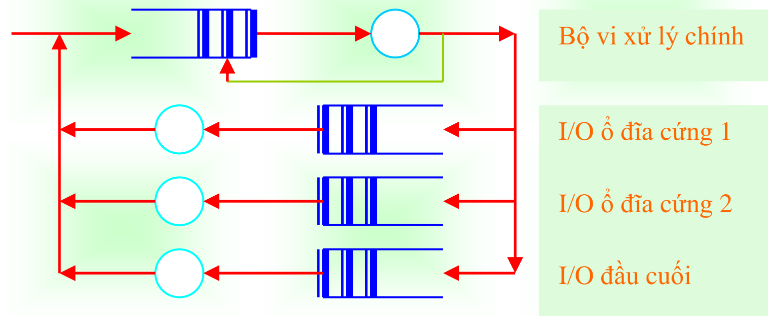
Tóm lại, sự định thời có ưu tiên chỉ có ý nghĩa khi: Tiến trình đang diễn biến có thể được thay thế bởi một tiến trình mới tới (từ hàng đợi I/O) có ưu tiên cao hơn hay được sắp xếp trở lại trong danh sách sẵn sàng. Trong thực tế, một liên hiệp hai phương pháp thường hay được sử dụng. Chẳng hạn, giao thức FCFS được thuyên chuyển cho hàng đợi quay vòng Robin (RR) ở sự định thời có ưu tiên.

2.2.4 Đa hàng đợi và đa bộ định thời

Ở một hệ thống vi xử lý hiện đại vẫn chỉ có một bộ vi xử lý chính, còn hầu hết các thiết bị vào ra thì nhanh hơn nhờ sử dụng một bộ điều khiển, mà bộ điều khiển này thì độc lập với bộ vi xử lý chính, và các dữ liệu có thể được tạo ra từ bộ nhớ chính đến bộ nhớ quảng đại và ngược lại (Direct Memory Acess:DMA). Bộ điều khiển DMA này có tác dụng như là những bộ vi xử lý chuyên dụng và chúng được xem như là phương tiện điều hành độc lập. Mục đích là, để tạo ra một hàng đợi cho mỗi cách xuất nhập mà nó được dịch vụ bởi bộ điều khiển DMA. Sự điều phối

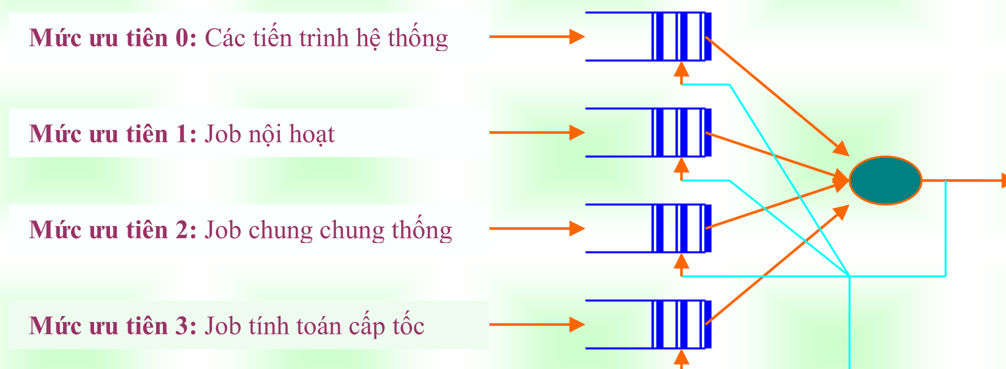
chung thì được phủ lên toàn bộ Job từ hàng đợi này tới hàng đợi kế tiếp, do những phản ứng ngắn của CPU (CPU bursts) nằm trong khoảng đó.

Một biến cố nữa cho thấy, chúng ta không chỉ có một loại Job, thực ra có rất nhiều loại Job vì do có sự ưu tiên khác nhau. Vì vậy, đối với mỗi loại Job thì một hàng đợi được thông báo. Khi đó, ta có định thời đa mức (*Multi – level - Scheduling*).



Hình 2.9. Định thời với đa hàng đợi

Với sự ưu tiên khác nhau, các hàng đợi được sắp xếp theo một tuần tự xác định, tức là theo một thứ tự làm việc xác định: hàng đợi có ưu tiên cao nhất làm việc trước tiên, tiếp đến hàng đợi thứ hai. Vì để có một Job mới luôn luôn đi tới, do đó dãy tuần tự làm việc cũng luôn luôn thay đổi. Điều đó được mô hình hoá thành bốn bình diện, thể hiện trong hình 2.10 ở dưới đây.



Hình 2.10. Định thời đa mức

Khi thời gian chờ đợi lâu hơn ở trong hàng chờ, Job có thể chuyển đến vị trí cao hơn. Lúc đó, người ta nói định thời ăn sau đa mức (*multi-level-feedback-Scheduling*).

Ở những thuật toán định thời được trình bày ở trên, cho đến nay, ta đã bỏ qua một tính huống, rằng tất cả các tiến trình ở trong bộ nhớ chính không thể cùng đồng thời được sử dụng. Để có thể định thời các tiến trình, người ta phải dựa vào các dữ liệu quan trọng của tiến trình ở trong bộ nhớ chính, mà cái đó được mô phỏng đầy đủ trong khối điều khiển tiến trình (Process Controll Block: PCB); tất

cả các dữ liệu khác thì được di chuyển trên bộ nhớ quảng đại. Nếu một tiến trình được hoạt động, thì đầu tiên nó phải nhận được sự sao chép từ bộ nhớ quảng đại vào bộ nhớ chính và sau đó, nó thực hiện. Cái đó yêu cầu thời gian bổ sung đáng kể thay đổi văn cảnh tiến trình và nâng cao thời hạn làm việc. Tuy nhiên, tốt nhất là phải có tiến trình sẵn sang đúng ở trong bộ nhớ chính. Nghĩa là, sự quá độ của tiến trình cần thiết phải được điều chỉnh từ bộ nhớ quảng đại tới bộ nhớ chính.

Cách giải quyết vấn đề này là dẫn vào một bộ định thời thứ hai để nó chỉ có nhiệm vụ gộp hay tachs ra các tiến trình. Bộ định thời thứ nhất điều hành việc sắp xếp các tiến trình tới các phương tiện điều hành (như bộ vi xử lý) và nó làm việc ngắn hạn. Còn bộ định thời thứ hai là bộ định thời trung bình hay dài hạn (giống trong hình 2.6) và nó điều chỉnh sự sắp xếp các tiến trình tự phương tiện điều hành (bộ nhớ chính), mà trong đó nó điều chỉnh độ lớn của phạm vi bộ nhớ chính. Trong những khoảng thời gian lớn hơn, loại định thời thứ hai cũng được gọi là loại định thời ngắn hạn. Cả hai loại đều sử dụng những hàng đợi riêng lẻ mà điều chỉnh sự sắp xếp và cả sự dẫn vào của tiến trình. Chiến lược cho định thời như kiểu đã nói này sẽ được trình bày trong chương 3.

2.2.5. Định thời ở trong hệ điều hành thời gian thực.

Có một loạt các hệ thống máy tính mà chúng được gọi là hệ thống thời gian thực (real time system). Với sự biểu thị này, người ta sẽ hiểu được điều gì? Một quan điểm trực giác về điều đó cho rằng: Đó là những hệ thống phải tác dụng nhanh, những hệ thống này cũng còn được gọi là những hệ thống thời gian thực. Với khái niệm “nhanh”, điều đó làm cho chúng ta có thể hiểu một cách chính xác hơn: Một hệ thống đang thực hiện một Job, thì Job đó phải tuân theo những quy định về thời gian đã được đề ra. Nhưng điều đó cũng chưa đủ đúng, vì có thể những quy định đó chưa thể là những quy định cứng được, ví dụ: một người soạn thảo không cần thiết phải sử dụng lâu hơn 2 giây để đưa một ký tự lên màn hình; một ngân hàng cần thiết phải thực hiện một việc chuyển tiền trong khoảng một tuần để tránh một sự nhầm lẫn đáng tiếc…Lúc đó, ta gọi nó là hệ thống thời gian thực mềm. Hệ thống thời gian thực mềm có đặc điểm: tại đó, các ngăn xếp các ngăn xếp thời gian là mềm và không được chuyên môn hoá. Lẽ tất nhiên, không có sự thoả mãn nào để dẫn tới sự phán quyết nặng cân. Trái ngược với hệ thống thời gian thực mềm là hệ thống thời gian thực cứng. Throng cog nigh may tin, *he thing this gain theca conga cons được gọi tắt là hệ thống thời gian thực*; nó cũng thường được dùng trong điều khiển các nhà máy điện nguyên tử, điều khiển máy bay, điều khiển giao thông…Vậy một hệ thống thời gian thực phải thừa nhận giới hạn thời gian đầu cuối rõ ràng đối với các tiến trình, để loại trừ được những quyết định sai phạm nghiêm trọng làm cho hệ thống tổn thất nặng nề.

Những thuật toán định thời phải được hướng tới kiểu dạng của các tiến trình. Kiểu dạng hệ thống thời gian thực là tình huống, mà các tiến trình luôn luôn quay trở lại khoảng thời gian đã được xác định chính xác và các tiến trình này thì có thể nhìn thấy trước đó ở trong sự thường xuyên xuất hiện của chúng cũng như ở trong

chu kỳ làm việc và ở trong sự xác định các phương tiện điều hành…Cho nên, điều đó thì có lợi để kiến tạo một sự định thời cố định.

Thí dụ về tác vụ định kỳ (*Priodic task*): Một máy bay(thí dụ loại Airbus A-340) được điều khiển bằng máy tính. Để điều khiển, máy tính cần sử dụng những số liệu bay khác nhau, mà nó phải được xác định và xử lý thành những quảng khác nhau: giá trị gia tốc theo hướng x,y,z khoảng 5ms, ba giá trị của các chuyển động quay khoảng 40 giây, nhiệt độ khoảng 1 giây và vị trí tuyệt đối để điều khiển khoảng 10giây. Trên màn hình cho thấy sự diễn biến trong từng giây. Những chiến lược định thời quan trọng theo chuẩn IEEE năm 1993 có những loại sau đây:

- *Chiến lược vòng được xén (Polled Loop):*

Bộ vi xử lý thực hiện một chu trình tính,mà ở đó, nó luôn luôn kiểm tra trở lại thiết bị, xem những số liệu mới có tồn tại không. Nếu tồn tại, thì do đó, nó sẽ xử lý ngay. Chiến lược này thích hợp với những thiết bị riêng lẻ, mà không thích hợp, nếu có một biến cố khác xuất hiện trong khi xử lý và do đó, các số liệu cũng không được sờ tới.

- *Chiến lược điều khiển ngắt các hệ thống:*

Bộ vi xử lý thực hiện một chu trình chờ.Nếu những số liệu mới xuất hiện, do đó, mỗi một ngắt của thiết bị được gọi để xử lý các số liệu mới này. Phương pháp điều khiển ngắt hệ thống này được gọi là lập thức dịch vụ ngắt (*Interrupt Service Routine*: ISR).

Nếu các ưu tiên được sắp xếp cho lập thức ISR, thì do đó, sự định thời có ưu tiên sẽ xẩy ra một cách tự động nhờ ngắt logic của điều khiển ngắt. Vấn đề còn lại của chiến lược này là, nếu các biến cố bị chất đống, thì khi đó, các ngắt có ưu tiên thấp không bị bẻ gãy và có thể được đẩy lên đầu.

- *Chiến lược đường tử ít nhất- trước nhất (Minimal Deadline First: MDF):*

Đâu tiên tiến trình được chỉnh lý: nó sẽ chiếm trước ngăn xếp thời gian nhỏ nhất (deadline time T_d: thời gian chết), rồi đến ngăn xếp tiếp theo. Giao thức này cũng thường hay được sử dụng (thí dụ để triển khai những dự án phần mềm), nhưng mà nó cũng có một vài nhược điểm. Thí dụ, chúng không có lợi, nếu tất cả các tiến trình chiếm các ngăn xếp thời gian như nhau.

- *Chiến lược thời gian xử lý ít nhất-trước nhất (Minimal Processing Time First: MPTF)*

Một tiến trình được chọn làm tiến trình điều khiển khi tiến trình này chiếm phần thời gian dịch vụ nhỏ nhất (control time T_c: thời gian điều khiển). Điều đó thì phù hợp với chiến lược SJF và nó có ý nghĩa rằng, Job ngắn với ưu tiên thấp thì được ưa chuộng hơn Job dài có ưu tiên cao.

- *Chiến lược định thời đơn điệu tỷ suất (Rate Monotonic Scheduling : RMS):*

Nếu chúng ta có một hệ thống ưu tiên cố định với các tỷ suất thực hiện cố định của các tiến trình tham gia (xem thí dụ định thời điều khiển máy bay ở trên), thì do đó, một cách tối ưu là, nếu chúng ta sắp xếp những ưu tiên cao nhất cho tỷ suất thực hiện cao và những ưu tiên thấp cho tỷ suất thực hiện thấp (gọi là định thời đơn điệu tỷ suất). Nếu trường hợp không có sự định thời đơn điệu tỷ suất đối với một tiến trình được tìm thấy, thì điều đó được chứng minh rằng, sau đó vẫn không có một sự định thời khác tồn tại, do đó sự định thời nói trên đạt yêu cầu. Thật vậy, nếu CPU có một khả năng tải nhỏ hơn 70%, thì với chiến lược RMS, tất cả các ngăn xếp thời gian được giữ đúng một cách bảo đảm. Tuy nhiên, điều cần thiết là, sự ưu tiên thấp của các tiến trình quan trọng với tần số thực hiện hạn chế phải được nâng lên. Cái đó được gọi là đảo ngược ưu tiên.

- *Chiến lược định thời hậu cảnh- tiền cảnh (Foreground Background Scheduling):*

Trong các hệ thống thời gian thực có một số tiến trình có ích, nhưng mà cũng không cần thiết lắm. Những tiến trình đó có thể được thu hẹp ở hậu cảnh, ngay khi mà bộ vi xử lý được giải phóng và nó không được dùng việc gì khác nữa. Mỗi một tiến trình có thể làm cho các hệ thống gián đoạn. Minh hoạ cho điều đó có vài ví dụ sau đây:

- *Tự thử nghiệm* để khám phá ra những khuyết tật.
- *Lắp thêm RAM* để đọc và viết lại nội dùng của RAM. Với những hệ thống tiện dụng thì, chúng ta có bus dữ liệu để sửa lỗi bit ở trong RAM.
- *Nâng cao khả năng tải của màn hình* để phát hiện sớm các lỗi. Thí dụ nhờ việc cảnh giới quá thời gian (*watch dog time*) mà tránh được một sự báo động khẩn cấp.

Một hệ điều hành thời gian thực bây giờ không chỉ có những tiến trình giới hạn, mà các ngăn xếp thời gian (time stack) của chúng nhất thiết phải được giữ cố định; nó còn có các tiến trình tới hạn cần thiết và các tiến trình không có giới hạn. Tất cả đều được sửa lỗi tương tự nếu còn thời gian. Chúng ta thấy rằng, những tiến trình tới hạn cần thiết được tháo gở theo một chiến lược *RMS* cố định. Loại tiến trình không có tới hạn được định thời theo chiến lược *hậu cảnh*. Còn loại tiến trình tới hạn quan trọng được định thời chiến lược *điều khiển ngắt hệ thống.*

Những nhà thiết kế hệ thống đã phát triển thêm nhiều chiến lược phụ cho loại các hệ thống vừa nêu. Họ đã tách chia để phân biệt các biến: biến về sự quan trọng, biến về thời gian ngăn xếp (với T_d là thời gian đình chỉ hay thời gian chết) và biến về phương tiện điều hành cần thiết (với T_c là thời gian dịch vụ hay thời gian điều khiển)…Đồng thời, họ cũng liên hiệp các biến này tới những chiến lược mới:

- Chiến lược tình trạng biến động nhỏ đầu tiên (*Minimum Laxity First*): Tiến trình được chọn cho kiểu định thời này phải có thời gian tự do nhỏ nhất (minimum free time), tức là biểu thức $[T_d - (T_s + T_c)]$ đạt nhỏ nhất.
- Chiến lược liên hiệp tiêu chuẩn 1: Tiến trình được chọn cho kiểu

định thời này có thời gian tự do biểu diễn trong biểu thức [T_d +T_c] đạt nhỏ nhất.

- Chiến lược liên hiệp tiêu chuẩn 2: Tiến trình được chọn cho kiểu định thời này có thời gian tự do ở dạng [T_d +T_s] đạt nhỏ nhất.

Những sự mô hình hoá cho thấy rằng, tất cả sự định thời, mà nó chỉ dùng một mình thời gian T_d, đều mang tới những kết quả tồi cho hệ thống đơn cũng như đa vi xử lý. Ngược lại, các tiêu chuẩn liên hiệp đã loại trừ cái đó một cách tốt đẹp, đặc biệt, liên hiệp tiêu chuẩn 2 đã đưa tới kiểu định thời tốt nhất, vì nó đã quan tâm tới các phương tiện điều hành.

2.2.6. Định thời ở hệ thống đa vi xử lý

Nói chung, đối với mỗi một phương tiện điều hành vẫn còn tồn tại những vấn đề, và do đó, đối với mỗi bộ vi xử lý, mỗi hàng đợi riêng lẻ hay mỗi kiểu định thời riêng lẻ cũng vậy. Tuy nhiên, sự quá độ giữa các hàng đợi là không thể tuỳ tiện được, đặc biệt, ở nhiều tiến trình xẩy ra song song hay cùng đồng thời làm việc, thì phải xem xét vấn đề trên cùng hệ trục toạ độ. Điều cho thấy rằng, giữa các tiến trình riêng lẻ tồn tại nhiều sự phụ thuộc ở dãy tuần tự làm việc.

Nếu chúng ta biểu thị các phương tiện điều hành bằng các chữ cái A,B,C và các yêu cầu đối với chúng là A_i, B_j, C_k , do đó mỗi yếu tố gây được ấn tượng qua ký tự “>”. Chẳng hạn $A_i > B_j$ có ý nói: đầu tiên A_i và sau đó (bất kỳ khi nào) B_j phải thực hiện. Nếu A_i là hành động trực tiếp ngay trước B_j là hành động kế gần, thì quan hệ giữa chúng được viết bằng dấu “>>”.

Sau đây dẫn ra một vài ví dụ:

$A_1 >> B_1 >> C_1 >> A_5 >> B_3 >> A_6$

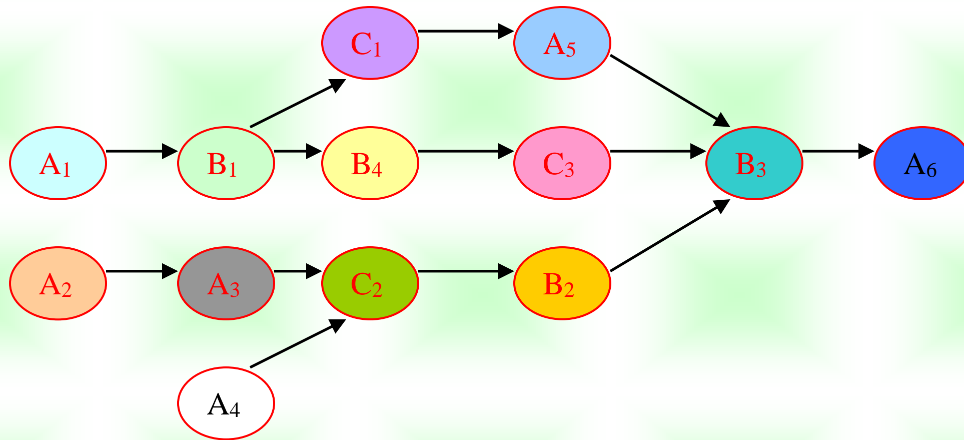
$B_1 >> B_4 >> C_3 >> B_3$

$A_2 >> A_3 >> B_4$

$A_3 >> C_2 >> B_2 >> B_3$

$A_4 >> C_2$

Những tương quan trong 5 hàng ví dụ trên sẽ được mô hình hoá qua một sơ đồ: một nút chỉ một yêu cầu của hệ điều hành, còn tương quan (>>) giữa chúng được biểu thị bằng mũi tên (với gốc là nguồn, ngọn là đích). Chu kỳ làm việc t_i của các yêu cầu phương tiện điều hành thì do đó một con số (gọi là trọng số) viết cạnh nút (để biểu thị một yêu cầu nào đó). Hình 2.11 chỉ ra một thí dụ. Một dãy tuần tự được gọi là đúng nhưng chưa cần thiết, nếu nó bao gồm những tiến trình độc lập, mà những tiến trình này chẳng có dữ liệu để mà thay đổi. Nhưng chúng cũng cần tới phương tiện điều hành, do đó dẫn tới tranh chấp tiến trình.



Hình 2.11.Sơ đồ tương quan điển hình

Một sơ đồ định thời có khả năng thực thi được mô hình hoá bằng đồ thị cột, mà ở đó, mỗi phương tiện điều hành được xếp vào một hang ngang.

Chúng ta chờ đợi một cái gì đó ở giao thức định thời tốt với tổng thời gian thực hiện T ? Điều đã rõ: với n tiến trình độc lập có các trọng số $t_1 \ldots t_n$ (chính là thời gian thực thi của mỗi tiến trình riêng lẻ) và m bộ vi xử lý, thì T_{opt} là tổng thời gian thực hiện tối ưu đối với một sự định thời có ưu tiên được phân bổ trên mỗi vi xử lý và được xác định bằng biểu thức sau:

$$T_{opt} = \max\{ 1/m \sum_{i=1}^{n} ti, \text{ với } \max t_i \} \quad 1<=i<=n$$

Đó là trường hợp thuận tiện nhất mà chúng ta chờ đợi. Đối với ví dụ ở trong hình 2.11, ta có n=13 tiến trình và m=3 bộ vi xử lý. Trong trường hợp thuận lợi, thời gian thực hiện tối ưu sẽ là T_{opt}=1/3(43)<=15. Trong khi đó cũng ví dụ này kiểu định thời trong hình 2.12 thì thời gian thực hiện là T=30. Sở dĩ có sự khác biệt đó là vì người ta đã dẫn ra lý do sau đây để giải thích cho ví dụ ở hình 2.12: Nếu tiến trình kiểu A_i chỉ thực hiện trên bộ vi xử lý A, tiến trình B_j chỉ thực hiện trên bộ vi xử lý B, còn tiến trình $C_k$chỉ thực hiện trên bộ vi xử lý C, thì do đó, thời gian thực hiện sẽ kéo dài.

Với các phương tiện trợ giúp ở trên, bây giờ, chúng ta muốn khảo sát một số giao thức định thời để các tiến trình thực hiện song song trên hệ đa vi xử lý gần kề nhau.

2.2.6.1. Định thời song song trên hệ đa vi xử lý

Chúng ta nhận thấy rằng, sự yêu cầu đối với một phương tiện điều hành (ta hiểu đó là thời gian thực hiện một tiến trình tại một nút) là phép tích luỹ của một khoảng thời gian cố định Δt. Các khoảng thời gian cố định này thì tỷ lệ với nhau.

Số lượng các nút trên sơ đồ được phân chia thành các cụm nhỏ, sao cho, tất cả các nút của một cụm thì độc lập với nhau và cũng không có quan hệ đặc biệt giữa chúng.

Thí dụ: Số lượng các nút trên hình 2.11 được phân chia thành những cụm độc lập là $\{A_1,A_2\}$,$\{B_1,A_3,A_4\}$, $\{C_1,B_4,C_2\}$,$\{A_5,C_3,B_2\}$,$\{B_3\}$, $\{A_6\}$. Tuy nhiên, B_4 thì phụ thuộc vào cụm A_3 và B_1, còn cụm $\{C_1,B_4,C_2\}$ thì phụ thuộc vào cụm $\{B_1,A_3,A_4\}$, nhưng nó không phải là các cumj nhỏ vừa chia.

Một sự tách chia như vậy sẽ có thể được nhận từ các phương thức khác nhau. Nghĩa là một sự phân chia, phân đoạn rõ ràng.

Nếu một hệ thống có N cụm, thì người ta gọi hệ thống đó có N bậc (mức): bậc 1 cho cụm có nút vào, bậc 2 cho cụm phụ thuộc vào cụm bậc 1… và bậc N cho cụm có nút cuối cùng. Điều đó được gọi là sự phân bổ tương quan.

Thí dụ: Ở sự tách chia nói trên, hệ có N=6 bậc, cụm $\{A_1,A_2\}$ là cụm bậc 1 và bậc 6 cho cụm có nút cuối cùng A_6.

Tuy nhiên, để tìm thấy một sự định thời tối ưu, vấn đề cơ bản là: Đối với một tiến trình nào đó thì sẽ có một yêu cầu tương ứng và do đó cũng có thuật toán tương ứng. Nhờ nó, mà người ta thiết kế được một chiến lược định thời khả thi. Thật vậy, hai ông R. Muntxz và E. Coffman đã đưa ra (1961) ba giao thức kinh điển sau đây để thiết kế định thời cho các tiến trình có ưu tiên.

- *Chiến lược định thời kiểu tai nghe (Earlisten Scheduling):*

Một tiến trình được xử lý thì một bộ vi xử lý sắp được tự do. Dựa theo ý kiến đó, chúng ta bắt đầu với bậc thứ nhất và sử dụng các bộ vi xử lý tự do được dụng cho các tiến trình của bậc thứ hai (ngay tức khắc nếu có điều kiện)…Tiếp đến, cũng theo cách tương tự, các bộ vi xử lý được tự do lại được dùng cho bậc kế tiếp cho đến khi tất cả các bậc đều làm việc.

Thí dụ: Sơ đồ kiểu khung của sự phân bổ đặc biệt của thí dụ ở hình 2.12 đã giải thích sự định thời cho các bộ vi xử lý P_1,P_2,P_3 và theo chiến lược định thời kiểu tai nghe, sơ đồ này có dạng như sau:

Hình 2.13------------------------

Đồ thị ở trên cho thấy, bằng sự định thời kiểu song song này, chúng ta đạt được thời gian dịch vụ T=20.

- *Chiến lược định thời kiểu muộn nhất (Latest Scheduling):*

Một tiến trình được thực hiện đến thời điểm muộn nhất, thì tại đó nó vẫn còn làm việc. Thêm vào đó, chúng ta thay đổi dãy tuần tự tên gọi các bậc thành sự trao đổi quan hệ: bậc N cuối cùng thì bây giờ thành bậc thứ nhất, còn bậc đầu tiên thì

tới bậc cuối cùng N. Bây giờ, chúng ta biểu thị trở lại như trước: đầu tiên bố trí bộ vi xử lý cho tiến trình của bậc 1, sau đó cho tiến trình của bậc 2…

Thí dụ, với giao thức định thời kiểu muộn nhất, thí dụ ở hình 2.11 được dẫn ra sơ đồ khung như trong hình 2.14 sau đây:

Hình 2.14-----------

Với kiểu định thời như hình 2.14, chúng ta đạt được thời gian diễn biến dài T=22.

- *Chiến lược định thời kiểu danh sách (List Scheduling):*

Tất cả các tiến trình có ưu tiên được dẫn vào một danh sách trung tâm. Nếu một bộ vi xử lý được tự do, khi đó, nó nhận được một tiến trình dẫn tới từ danh sách các tiến trình và thực hiện tiến trình được dẫn tới này, vì nó có ưu tiên cao nhất. Nếu ta lưu ý sự ưu tiên đã được sắp xếp trong danh sách sẵn sàng và nếu các tiến trình đã được sắp xếp thành hàng đợi, thì do đó, chúng ta nhận được một hàng đợi các tiến trình đa vi xử lý. Ở mục 2.3.5 thì sự điều khiển hàng đợi ở các máy tính siêu hạng NYU sẽ mô tả kỷ càng.

Người ta thấy rằng, với nhiều thuật toán, tuy người ta không tìm thấy sự định thời tối ưu, nhưng nhờ giao thức này, người ta có thể tìm thấy một sự định thời năng động.

Với m =2 bộ vi xử lý và các tiến trình có độ dài $t_i = t_k$, chúng ta có thể đạt được một sự định thời tối ưu có ưu tiên của toàn bộ các tiến trình. Không những thế, đối với từng cụm bậc các tiến trình, người ta cũng tìm thấy một sự định thời tối ưu có ưu tiên. Muntz và Coffman đã mở rộng suy nghĩ này (1969) cho trường hợp t_i # t_k : Ở đây, mỗi một tiến trình có thời gian dịch vụ s.Δt được tồn tại một cách ảo từ một kết quả của s đơn vị tiến trình. Chúng ta nhận được một sơ đồ quan hệ mới, mà nó chỉ chứa đựng những tiến trình cùng độ dài thời gian thực hiện và với m=2 bộ vi xử lý thì nó cho phép có một sự định thời tối ưu có ưu tiên. Tuy nhiên, ở một số lượng không đúng mức các tiến trình thì cần thiết, tại ba tiến trình C_1,C_2,C_3 của một trong các cụm bậc phải tách ra một tiến trình (thí dụ C_2) và phải phân các vi xử lý ra làm 2 (xem hình 2.15). Nếu sự định thời của cụm được tối ưu, thì do đó, sự định thời của cả hệ cũng tối ưu.

Hình 2.15------------------

Những sự khảo sát này có thể cũng được mở rộng ta cho những đồ thị tương quan với nhiều chuỗi Job song song đơn giản, hoặc cũng như các đồ thị với một nút khởi đầu và một nút cuối, và chỉ nhiều nhất một nút kế tiếp một nút khác.

2.2.6.2 Thời gian thực hiện nhỏ nhất ở sự định thời đa vi xử lý

Chúng ta sử dụng dãy hàng đợi trung tâm được định hướng điển hình và có ưu tiên. Vậy thời gian thực hiện nhỏ nhất (T_{prio}) là gì, mà chúng ta chờ đợi nó ở trong danh sách các Job, để không có sự nhầm lẫn về đặc điểm ưu tiên và điển hình ?

Để trả lời câu hỏi này, chúng ta thực hiện một sự tính toán, mà hai nhà hệ thống J.W.S Lliu và C.L Liu đã đề xướng (1978). Chúng ta nhận thấy, nếu T_{prio} là thời gian thực hiện đối với một sự định thời có ưu tiên, mà sự định thời này thông báo cho một bộ vi xử lý (đã được giải phóng) có một Job: Job này có thể được thực hiện với sự ưu tiên cao từ các danh sách liệt kê. Sau đó T_{prio} được phân định cho mỗi bộ vi xử lý P_j thực hiện một thời gian t_j và một thời gian trống Φ_j. Chúng ta có thể đặt chỉ số j cho tất cả các bộ vi xử lý làm việc, còn không, các phần việc của Job được đặt bởi hai chỉ số: chỉ số i cho số Job ở một bộ vi xử lý và chỉ số k để chỉ vi xử lý đó. Với các ký hiệu vừa nêu, ta có biểu thức:

$$\mathbf{t_{prio} = t_j + \Phi_j = t_{ik} + \Phi_{ik}}$$

Thời gian thực thi của các Job diễn ra song song ở tất cả m bộ vi xử lý với r kiểu bộ vi xử lý khác nhau thì được biểu diễn như sau:

$$\mathbf{T_{*prio} = (1/m) \sum_{j=1}^{m} T_{prio} = (1/m) \sum_{j=1}^{m} [t_j + \Phi_j]}$$

Hay ta có

$$\mathbf{T_{*prio} = (1/m) \sum_{k=1}^{r} \sum_{i=1}^{rk} [t_{ik} + \Phi_{ik}]}$$

Chúng ta gộp r_k lần các thời gian thực hiện, thời gian trống ở các bộ vi xử lý cùng kiểu k, ta nhận được:

$$\mathbf{T_k = \sum_{i=1}^{rk} t_{ik} \ ; \ \Phi_k = \sum_{i=1}^{rk} \Phi_{ik} \ ; \ \Phi = \sum_{k=1}^{r} \Phi_k}$$

Do đó dẫn tới biểu thức chung:

$$\mathbf{T_{*prio} = (1/m) \sum_{k=1}^{r} [t_k + \Phi_k] = (1/m) (\Phi + \sum_{k=1}^{r} t_k) \quad (2.1)}$$

Một ngăn xếp tốt thì cho thời gian trống Φ là bao nhiêu ? Việc trả lời cho câu hỏi này được quyết định qua chất lượng của việc định thời.

Ta ký hiệu H là tổng các thời gian trống của từng bộ vi xử lý, ở đây, tối thiểu một vi xử lý thực hiện một lần trống và K_j là tổng thời gian trống của các việc khác. Ta có biểu thức biểu diễn điều kiện:

$$\Phi <= H + \sum_{j=1}^{r} K_j \qquad (2.2)$$

Việc làm của Job trong khoảng thời gian trống H được đặc trưng rằng, tại đó, những điều kiện cưỡng bức phải được thực hiện để ngăn ngừa sự làm việc song song của các kiểu Job giống nhau. Nghĩa là, các Job chỉ có thể được làm việc một cách tuần tự. Không thể có định thời tối ưu, nếu thời gian thực hiện tổng cộng ở tối thiểu một bộ vi xử lý (đã được giải phóng) thì nhỏ hơn thời gian trống trên mỗi bộ vi xử lý. Do đó, ở việc định thời tối ưu, thời gian thực hiện tối thiểu T_0 phải có giá trị lớn hơn:

$$T_0 >= \frac{H}{m-1} \quad \text{hay} \quad T_0 >= \frac{1}{m}\sum_{k=1}^{r} T_k \qquad (2.3)$$

Đối với thời gian trống K_j với tối thiểu một bộ vi xử lý thứ j làm việc sẽ dẫn tới các ký hiệu sau: S_j là thành phần thời gian thực hiện của các Job ở bộ vi xử lý thứ j (tức là tại đó, tối thiểu có bộ vi xử lý thứ j trống), còn ký hiệu $T_k - S_j$ là thời gian mà tại đó không có bộ vi xử lý thứ j trống: nghĩa là, tất cả đều làm việc. Khoảng thời gian này ở một chu trình định thời là $(T_k + S_j)/m_j$. Còn khoảng thời gian mà ở đó các bộ vi xử lý khác $(m-m_j)$ thì trống và còn lại các bộ vi xử lý …từ bộ thứ j làm việc, thì nó phải nhỏ hơn hay bằng khoảng thời gian làm việc của tất cả m_j bộ vi xử lý:

$$K_j /(m=m_j) <= (T_j - S_j) /m_j$$

Nếu tat hay biểu thức này vào trong phương trình (2.2), do đó, chúng ta nhận được sự đánh giá cho khoảng thời gian trống:

$$\Phi <= H + \sum_{j=1}^{r} ([m-m_j]/m_j)/ (T_j - S_j) = H\sum_{j=1}^{r} T_j(m-m_j)/m_j - \sum_{j=1}^{r} S_j(m-m_j)/m_j$$

Hay

$$\Phi <= H + m\sum_{j=1}^{r} (1/m_j) T_j - \sum_{j=1}^{r} T_j (1/m_j)S_j + \sum_{j=1}^{r} S_j$$

Biểu thức tổng cuối cùng có thể được thu nhỏ thêm, ở đây chúng ta sẽ sử dụng hệ số nhỏ nhất và đưa vào biểu thức trên với tư cách là thừa số chung:

$$\Phi <= H + m\sum_{j=1}^{r} (1/m_j) T_j - \sum_{j=1}^{r} T_j - m(\min [1/m_j]) \sum_{j=1}^{r} S_j + \sum_{j=1}^{r} S_j \qquad (2.4)$$

Bây giờ, chúng ta phải đánh giá tổng các thời gian trống S_j, Nếu luôn luôn có một bộ vi xử lý trống, do đó, H là tổng tất cả những khoảng thời gian như thế ở (m-1) bộ vi xử lý sẽ là:

$$H = (m-1)\sum_{j} S_j$$

Trường hợp chung, nếu các thời gian trống đơn lẻ S_j gộp lại thành tổng, thì tổng này phải lớn hơn H/(m-1), nghĩa là:

$$\sum_{j} S_j >= H/(m-1) \qquad (2.5)$$

Nếu chúng ta xem xét các quan hệ trong (2.5)với (2.2) và (2.3) thì dẫn tới:

$$\sum_{j=1}^{r} (1/m_j)\, T_j - \sum_{j=1}^{r} T_j <= mT_0(r-1)$$

Thay biểu thức này vào trong (2.4), chúng ta nhận được sự đánh giá cho tổng thời gian trống trong biểu thức sau:

$$\Phi <= H + mT_0(r-1) - (\min[m/m_j]-1)\sum_{j=1}^{r} T_j$$

Và với quan hệ trong biểu thức (2.5), biểu thức (2.6) được biến đổi thành:

$$\Phi <= H + mT_0(r-1) - (\min[m/m_j]-1)H/(m-1)$$

Hay $$\Phi <= H + mT_0(r-1) - (m - \min[m/m_j])\, H/(m-1)$$

Vì phần trong dấu ngoặc là một số dương, do đó, ta sử dụng biểu thức (2.3) và biến đổi biểu thức trên trở thành:

$$\Phi <= H + mT_0(r-1) - T_0(m - \min[m/m_j]) = mT_0 r - mT_0(\min[1/m_j])$$

Hay $$\Phi <= H + mT_0(r - \min[1/m_j]) \qquad (2.7)$$

Bây giờ chúng ta có thể trực tiếp dẫn ra sự tương quan của thời gian thực hiện theo định thời có ưu tiên P_{prio} với thời gian thực hiện theo định thời tối ưu bất kỳ T_0 ta thay biểu thức (2.7) vào biểu thức (2.1) và với sự quan tâm tới quan hệ thứ 2 trong biểu thức (2.3), ta nhận được bất phương trình sau đây:

$$T_{prio} = 1/m\left(\Phi + \sum_{k=1}^{r} t_k\right) <= T_0(r - \min[1/m_j]) + T_0$$

Hay $$T_{prio}/T_0 <= 1 + r - \min[1/m_j] \qquad (2.8)$$

Đối với các bộ vi xử lý cùng kiểu (r=1) và có m bộ vi xử lý, công thức (2.8) trở thành:

$$T_{prio} / T_0 <= 2 - 1/m \qquad (2.9)$$

Từ biểu thức (2.9), người ta thấy, với các bộ vi xử lý cùng loại,thì trong trường hợp nhanh nhất, một sự định thời có ưu tiên sẽ diễn ra lâu gấp đôi so với định thời tối ưu; trường hợp tốt nhất là bằng nhau. Khi chỉ có một bộ vi xử lý thì cả hai trường hợp hoàn toàn giống nhau, vì một bộ vi xử lý luôn luôn đảm nhận toàn bộ công việc, dù dãy tuần tự thế nào.

Nếu người ta dẫn vào nhiều bộ vi xử lý, tức là đưa vào nhiều phương tiện điều hành, thì tổng thời gian thực hiện hầu như cũng được thu nhỏ nhờ những khả năng làm việc song song của các vi xử lý. Vì với những phương tiện điều hành bổ trợ lẫn nhau này, làm cho sự quá tải giảm, do đó, thời gian trống trung bình của phương tiện điều hành tăng cao hơn.

Sự giảm nhẹ các hạn chế đã dẫn ra các phương pháp thích hợp sau đây:

- Nâng cao một vài điều kiện đặc biệt;
- Thu hẹp một ít thời gian thực hiện ($t_i = t_j$);
- Gia tăng số lượng bộ vi xử lý (m=m).

Điều đó không thể dẫn tới rút ngắn hơn thời gian thực hiện một cách tự động, mà nó chỉ có thể dẫn tới sự định thời bất lợi, người ta gọi hiện tượng này là sự khác thường đa vi xử lý. Trong mối quan hệ này thì công trình nghiên cứu của R.L.Graham (1972) đem lại nhiều thú vị. Ông nhận thấy rằng, trong một hệ thống với m bộ vi xử lý giống hệt nhau, mà trong hệ thống này các tiến trình được sắp xếp cho các vi xử lý một cách hoàn toàn theo ý muốn, thì tổng thời gian thực hiện T không nhiều hơn 2 lần so với sự định thời tối ưu: Gọi T là tổng thời gian thực hiện của một sự định thời được thay đổi với các hạn chế được thu hẹp, dẫn tới quan hệ:

$$(T/T) <= 1 + (m-1)/m \qquad (2.10)$$

Do đó, khi m=m thì giới hạn sẽ dẫn tới (2-1/m).

Để thiết kế việc định thời một cách thành đạt, một phương pháp tốt tồn tại trong sự tất yếu của các tiến trình (hầu hết được sinh ra do một tiến trình cha) tới một nhóm tiến trình, mà nhóm tiến trình này cũng được thực hiện trên một hệ thống đa vi xử lý, lúc đó, người ta gọi là định thời nhóm (*group Scheduling*). Vì hầu như có các sự phụ thuộc về hình dạng vào cơ cấu trao đổi thông tin dưới các tiến trình của nhóm, mà các tiến trình này có thể hoạt động được nhờ bộ nhớ chia xẻ (bộ nhớ chung), do đó, người ta loại trừ được sự đưa vào hay xuất ra của các phía bộ nhớ và của văn cảnh tiến trình tham gia.

Một giả thiết quan trọng đối với việc định thời có ưu tiên là một sự tiêu phí cho trao đổi thông tin thấp hơn để có thể bắt đầu và tiếp tục một tiến trình với văn cảnh của nó trên một bộ vi xử lý. Điều đó thì không xảy ra ở hệ thống phân bổ, mà hầu

như chỉ xẩy ra ở các hệ thống đa vi xử lý được liên kết chặt chẽ hay ở việc sử dụng các nhóm tiến trình. Nếu người ta sử dụng định thời không ưu tiên, thì do đó, sự bất lợi của nó được loại bỏ; tuy nhiên, việc định thời này cũng có khó khăn hơn. Thật vậy, các giao thức định thời được trình bày ở trên cũng có giá trị đối với việc định thời không có ưu tiên của các tiến trình dài; dù vậy, sự định thời không có ưu tiên nói chung phải được lưu ý một cách đặc biệt.

Các chiến lược định thời đặc biệt tiếp theo được M. Gonzales (1977) tìm thấy.

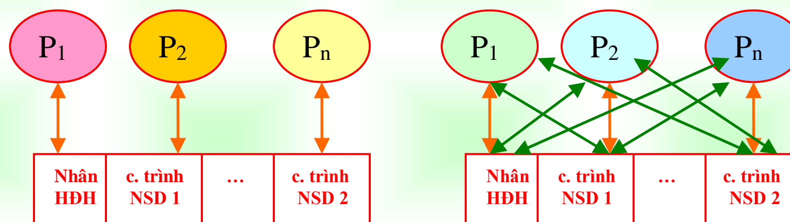
2.2.6.3. Sự phân bổ tải ở hệ thống đa vi xử lý

Gia tốc có thể đạt được cao nhất (speedup là sự tương quan giữa thời gian cũ và thời gian mới) của việc thực hiện chương trình nhờ các bộ vi xử lý phân bổ song song nhau hoặc nhờ các kênh I/O...thì rất là hạn chế. Một cách có lợi nhất, chúng ta có thể phân bổ tải trên m phương tiện điều hành theo nguyên tắc: thời gian mới thì bằng thời gian củ chia cho m, tức là, một cái gì đó gây nên gia tốc tuyến tính của m (bộ vi xử lý). Trong thực tế, một sự chỉ dẫn khác có tính quyết định hơn, đó là thành phần của mã tuần tự không thể xếp song song. Người ta ký hiệu thời gian thực hiện tuần tự của mã tuần tự là T_{par} , do đó, khi m ->∞ bộ vi xử lý và T_{par} ->∞, thì gia tốc thay đổi giữa thời gian mới xảy ra mạnh mẽ. Cuối cùng, tương quan giữa mã tuần tự và mã song song được xác định

$$\textbf{speedup} = t_{cũ} / t_{mới} = (T_{seq} + T_{par}) / T_{seq} = 1 + (T_{par} / T_{seq}) \qquad \textbf{(2.11)}$$

Thí dụ: Chúng ta có một chương trình, mà nó có thể nhận được khoảng 90% thời gian theo mã xếp song song, chỉ có 10% thời gian thực hiện theo mã tuần tự. Vậy, với giao thức định thời tốt nhất và với phần cứng song hành nhanh nhất, chúng ta có thể chỉ tăng tốc mã với 90% thời gian, còn phần kia 10% phải thực hiện tiếp theo, và cho phép chúng ta chỉ nhận được một gia tốc lớn nhất khoảng (90% +10%)/10%= 10 lần.

Vì hầu hết các chương trình có việc tính toán cấp bách ít hơn nhưng lại chiếm phần I/O cao khoảng 20- 80%. Điều ấy có nghĩa là, các phần mềm của hệ điều hành cần thiết được đánh giá ở trong việc định thời song song, để làm giảm một cách rõ ràng thời gian thực hiện. Đối với điều đó, hình 2.16 chỉ ra hai khả năng.



Hình 2.16. Đa xử lý đối xứng và không đối xứng

Hình phía trái là tình huống của hình 2.11 được trình bày một cách đơn giản: Mỗi một bộ vi xử lý được làm việc một cách nghiêm ngặt để tách chia một trong các tiến trình tồn tại song song, mà ở đó, hệ điều hành được xếp đặt như là các mã tuần tự cho một bộ vi xử lý. Cấu hình này được biểu thị là *đa vi xử lý không đối xứng*.

Ngược lại với cái đó, người ta có thể phân tách hệ điều hành cũng như phân tách các chương trình người sử dụng thành những đoạn mã thực hiện song song, mà những đoạn mã này được mỗi bộ vi xử lý thực hiện. Do đó, các thành phần của hệ điều hành có thể được thực hiện một cách song song, vậy năng suất đã được nâng cao một cách rõ ràng. Cấu hình này người ta biểu thị là *đa xử lý đối xứng*.

Đối với việc định thời các hệ thống đa vi xử lý thì một hàng đợi diện rộng (*global-wait-range*) được sử dụng, mà ở đó tất cả tiến trình sẵn sàng (*ready-process*) được đưa vào. Mỗi bộ vi xử lý khi hoàn tất nhiệm vụ (Job) của nó, thì nó đón nhận một Job mới kế nhau; nó còn chỉ ra tại các lỗi phương tiện điều hành rằng, cho đến khi Job được gặp thì các Job khác phải thực hiện tiếp tục và việc điều hành tính toán cũng diễn ra tiếp tục. Những lỗi phương tiện điều hành này thì không thể hiện trong các hàm lỗi riêng lẻ, mà nó thể hiện trong sự tổn thất chung của bộ vi xử lý.

Vấn đề cơ bản cho đến nay chưa được giải quyết là việc phân bổ n nhiệm vụ cho m bộ vi xử lý. Về vấn đề này, nhiều chiến lược khác nhau đã được ghi nhận một cách mạnh mẽ nhờ các giả định về các tính chất của các nhiệm vụ. Vì sự phụ thuộc của các nhiệm vụ với nhau cũng như sự phụ thuộc của các phương tiện điều hành trong khi làm việc tại các Job không đều đặn chỉ có thể là những giả định, cho nên, các thuật toán định thời được nêu ra hoặc là ít hơn hoặc là nhiều hơn các phép tính gần đúng.

2.2.7 Mô hình định thời ngẫu nhiên

Trên cơ sở này, có hai giả định cơ bản để thiết kế những thuật toán định thời tốt dựa trên những quan sát một cách ngẫu nhiên các tính chất của nhiệm vụ: chúng không được tách biệt, mà chúng được điền thêm vào.

Đối với giả định thứ nhất, người ta có thể xây dựng một mô hình toán học với một sự tiếp nhận hợp lý về các Job và công việc của chúng. Đó là nhiệm vụ của việc phân tích giả định, mà nó đã được P.B Hausen (1973) sử dụng để phát triển lý thuyết hàng đợi.

Thí dụ: Gọi P_j là xác xuất mà một Job mới tại một hàng đợi xuất hiện trong khoảng thời gian Δt thì tỷ lệ với khoảng thời gian đó và độc lập với lai lịch trước đó (gọi là sự tiếp nhận phân bổ Poisson):

$$P_j \sim \Delta t \quad \text{hay:} \quad P_j = \lambda \Delta t \qquad (2.12)$$

λ gọi là hệ số tỷ lệ.

Gọi P_N là xác suất mà không có Job nào đến trong khoảng thời gian Δt, do vậy ta có:

$$P_N(\Delta t) = 1 - P_j = 1 - \lambda \Delta t \qquad (2.13)$$

Do đó, xác suất mà không có Job nào đến trong khoảng thời gian t +Δt được xác định:

$P_N(t + \Delta t) = P$ (không có Job trong khoảng t ∩ không có Job trong khoảng Δt)

$$= P_N(t)\, P_N(\Delta t)$$

$$= P_N(t)\,(1 - \lambda\Delta t)$$

Bởi vậy, ta rút ra:

$$[P_N(t + \Delta t) - P_N(t)] / \Delta t = -\lambda P_N(t)$$

Ở giới hạn quá độ, khi Δt -> 0, thì ta có:

$$dP_N(t) / d(t) = -\lambda P_N(t) \qquad (2.14)$$

với $P_N(0) = 1$ thì: $P_N(t) = e^{-\lambda t}$ (2.15)

Từ đó ta nhận được biểu thức xác suất phân bổ theo hàm số mũ:

$$P_j(t) = 1 - P_N(t) = 1 - e^{-\lambda t} \qquad (2.16)$$

Với biểu thức này, ta đã nhận được một sự giả định thống kê quan trọng, nó không chỉ đối với thời gian đi tới mà cả thời gian làm việc của Job. Gọi P(t) là mật độ xác suất khi đến cũng như khi làm việc của Job, lấy đạo hàm 2 vế của biểu thức (2.16), ta có:

$$dP(t)/dt = \lambda e^{-\lambda t}$$

Hay: $dP(t) = \lambda e^{-\lambda t} dt$

Lấy tích phân 2 vế của biểu thức, ta nhận được:

$$P(t) = \int_0^t \lambda\, e^{-\lambda t} dt = \lambda e^{-\lambda t} \qquad (2.17)$$

Gọi T_j là thời gian đến trung bình hay thời gian đến chờ đợi, bằng sự biến đổi toán học thông thường, ta có:

$$T_j = \int_0^\infty P(t)dt = \int_0^\infty t\, e^{-\lambda t} dt \qquad (2.18)$$

Nghịch đảo của T_j là tần số hay là tỷ lệ thời gian tới λ. Tương tự, chúng ta nhận được tỷ lệ thời gian điều khiển là μ. Nếu $\lambda < \mu$, thì hàng đợi ở trong trạng thái trọng lượng nhẹ. Đối với trường hợp này ta nhận thấy: chiều dài trung bình L của hàng đợi thì tỷ lệ với thời gian chờ trung bình T_w của Job, tức là:

$$L = \lambda T_w \qquad (2.19)$$

Một giả định đối với định thời đa vi xử lý ngẫu nhiên được J.T.Robinson tìm thấy (1979). Đáng tiếc, những giả thiết này đã phủ định hoàn toàn mô hình này, mà nó đã tạo điều kiện cho những công trình nghiên cứu bằng toán học. Đáng lẽ những điều này phải được các giả định tổng hợp thực hiện, mà những giả định này gây khó khăn cho việc nghiên cứu bằng toán học. Do đó, những kết quả của một sự mô hình hoá toán học gần như bị giới hạn một cách có thể dùng được; chúng chỉ ở trong ý tưởng và trình bày một tình huống đặc biệt.

Trên cơ sở này, điều rất có lý là, với lập luận thứ 2, sự mô hình hoá các hệ thống nghiên cứu được thực hiện. Mấu chốt của cái đó là những gói thông tin ready (phần mềm) khác nhau, nhờ nó mà công việc nghiên cứu thuận lợi. Ở việc mô hình hoá, tất cả các phương tiện điều hành song song (như các bộ vi xử lý, các kênh I/O...) với mỗi hàng đợi, tỷ lệ xuất hiện, loại điều khiển được mô hình hoá. Hệ thống máy tính thực hiện được kết nối thành mạng từ các trạm điều khiển, do đó, người ta có thể làm thích hợp những tham số riêng lẻ thực hiện tốt. Những vướng mắc và những nan giải có thể được giải quyết bằng mô hình hoá, được khắc phục và được thông dịch một cách thích hợp.

2.2.8 Định thời trong Unix

Trong Unix, phương pháp quay vòng Robinson được áp dụng, để mô hình hoá một cách thích hợp cho các tiến trình có ưu tiên. Mỗi một tác vụ (tiến trình) đón nhận một sự ưu tiên nào đó mà nó được phân bổ. Qua đó, các tiến trình hệ thống được phân bổ trước đó, mà từ một phía tới phía khác được đảm bảo rằng, cả những tiến trình chờ lâu thì tối thiểu một lần được đưa tới hàng đợi.

Ngoài ra, trong Unix, mỗi sự ưu tiên được ký hiệu một chữ số từ -127 đến +127, ở đây, những con số nhỏ thì có ưu tiên cao. Trong các tiến trình hệ thống, những con số âm có ý nghĩa rằng, nó không thể bị bẻ gãy một cách bình thường. Nó chỉ cho phép người sử dụng dùng lệnh **nice** , để người sử dụng tiếp tục dùng các tiêu chuẩn ưu tiên riêng lẻ, vì vậy tại các Job chưa tới hạn thì vẫn có thể còn dùng được cho người sử dụng khác. Một cách hình thức thì điều đó không nên dùng.

Vì ở những con số đầy đủ tồn tại nhiều Job với sự ưu tiên như nhau, do đó, đối với tất cả các Job có ưu tiên giống nhau sẽ nhận thấy một hàng đợi theo định thời FCGS. Nếu tất cả các Job của một hàng đợi đã thực hiện xong, do đó những hàng đợi sau đó với ưu tiên thấp được làm việc. Ngoại lệ, những Job có ưu tiên mà phải chờ lâu, thì cũng được treo vào hàng đợi khác (theo kiểu định thời đa mức bước lùi).

Trong các phiên bản mới của Unix thì sự ưu tiên được đánh số từ 0 đến 255 và ngoài ra, nó còn được chia nhỏ thêm. Hình 2.17 chỉ ra dãy hàng đợi có ưu tiên của kiểu định thời đa mức này.

Các hàng đợi tạo thành một danh sách chuỗi, mà các hiển thị nội dùng của nó được chỉ ra trên khối điều khiển tiến trình (PCB) của bảng tiến trình. Tất cả các Job của ưu tiên hệ thống (128 đến 177) và tất cả các Job của người sử dụng (188 đến 255) được phân chia theo phương pháp lát cắt thời gian, các Job được xử lý đặc biệt, nếu nó được khởi động bởi gọi hệ thống đặc biệt với hàm gọi *rtprio()*. Sự ưu tiên (0 đến 127) dành cho những ưu tiên cao hơn và không được thay đổi nhiều.

Những tính chất không tồn tại trong các phiên bản khác nhau của Unix như việc định thời có ưu tiên các tiến trình, mà nó thuộc kiểu định thời trạng thái nhân thì không thể ngắt một cách bình thường (nó chỉ tồn tại một ngăn xếp nhân riêng lẻ, do đó không thể vượt qua được), việc phân bổ bộ nhớ chính được dự trữ cứng để phòng tránh việc đổi tráo và cũng như các files với các khối hợp lý cũng tạo nên một đặc trưng thời gian thực ở trong Unix.

2.2.9 Định thời trong Windows NT

Ngay cả ở trong Windows NT cũng có một sự định thời đa mức, mà nó đã thu nạp những Job có thời gian thực. Những sự ưu tiên đi từ mức ưu tiên 0, đó là mức ưu tiên thu hẹp nhất (đối với tiến trình trống hệ thống) tới ưu tiên mức cao nhất 31 (đối với các tiến trình thời gian thực). Trong hình 2.18 dẫn tới một cách nhìn tổng quát. Tuy nhiên, không có Job nào được quản lý, ngay cả các tiến trình trọng lượng nhẹ (threads). Sự định thời cũng được tách chia bởi sự điều phối. Thật vậy, bộ điều phối đã cung cấp một cơ sở dữ liệu riêng, mà nó đã giữ chặt trạng thái của tiến trình trọng lượng nhẹ và của CPU, đồng thời, nó đã tạo nên những quyết định nguyên tắc cho bộ định thời. Windows NT trợ giúp cho ta xử lý cân đối. Nếu quá trình trọng lượng nhẹ không kết thúc, thì do đó, các CPU sẽ thực hiện một thread đặc biệt gọi là idle thread.

Những tiến trình threads được phân bổ và làm việc theo phương pháp lát cắt thời gian ưu tiên. Sau mỗi lát cắt thời gian (tức là bộ ngắt thời gian), thì ưu tiên của tiến trình thread (ưu tiên mức 1 đến 15) bị thu hẹp một ít cho đến khi đạt nhỏ nhất tới ưu tiên cơ sở (ưu tiên 2 đến 6). Sau đó, cần được quyết định rằng, những tiến trình thread nào ở trong hàng đợi có ưu tiên cao nhất và liệu tiến trình thread có được thực hiện trên bộ vi xử lý ? Cho cái đó, có một tinh chất phụ có thể thay đổi được, đó là tính chất ảnh xạ tương đồng của các bộ vi xử lý.

Nếu có một tiến trình thread được treo vào một hàng đợi ở danh sách sẵn sàng, do đó, nó nhận được sự ưu tiên bổ sung, mà sự ưu tiên này thì phụ thuộc vào loại danh sách chờ: Đầu cuối I/O nhận được nhiều ưu tiên bổ sung như là các bản I/O.

Nếu tiến trình thread tới sẵn sàng, thì tiến trình này có ưu tiên thời gian thực (real time) cao hơn (mức 16 đến 31) như là một trong các tiến trình threads được thực hiện ngay trên một trong các bộ vi xử lý và do đó, tiến trình thread được dẫn vào hàng đợi, đồng thời một bộ vi xử lý đón nhận một ngắt. Do vậy, nó đẩy tiến trình trở lại vào hàng đợi và thay đổi cho đến khi đạt ưu tiên mức cao hơn.

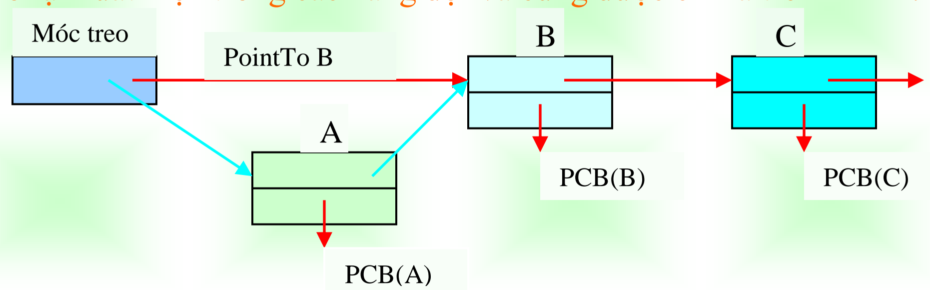
Những mức ưu tiên được thiết lập sẽ được phân bổ theo loại của Job: Những Job nội hoạt thì quan trọng hơn các Job với I/O; các Job I/O thì quan trọng hơn các Job thuần tính toán.

2.3 Đồng bộ tiến trình.

Việc trình bày các tiến trình như là những đơn vị chương trình có thể thực hiện độc lập một cách giả song song mang lại nhiều đặc điểm mềm dẻo và hiệu quả trong việc tổ chức tính toán; điều đó đã cung cấp cho chúng ta phương pháp về sự định thời các tiến trình và cả các vấn đề trợ giúp. Bởi vậy, đầu tiên chúng ta nghiên cứu những tình huống sau đây.

2.3.1 Các điều kiện chạy đua và các giai đoạn tới hạn

Người ta nhận thấy rằng, ở một trong nhiều hàng đợi của hệ thống tiến trình, thì nhiều tiến trình được treo vào và chờ đợi sự điều khiển. Tình trạng này luôn luôn trở lại xuất hiện trong các hàng đợi và cũng được chỉ ra trên hình 2.19.



Hình 2.19.Các điều kiện chạy đua ở một hàng đợi

Về điều đó, chưa có thể phân biệt các trạng thái rằng, hoặc là một tiến trình mới được treo vào (ở đây là tiến trình A) hoặc là một tiến trình đang tồn tại được treo vào (ở đây là tiến trình B).

Để treo vào hay bứt ra, tiến hành những bước sau đây:

Treo vào:	*Bứt ra*:
(1) Lấy móc treo: PointToB	(1) Lấy móc treo: PointToB
(2) Lập dấu chỉ dẫn kế cạnh : = PointToB	(2) Lập dấu chỉ dẫn kế cạnh := PointToB
(3) Đặt móc treo:= PointToA	(3)Đặt móc treo:= PointToC

Sau khi bứt ra một tiến trình thì không gian nhớ của việc ghi chép danh sách được trả lại tự do và một sự chuyển đổi tiến trình được đón nhận với dấu hiệu chỉ dẫn tới khối điều khiển tiến trình.

Mỗi một trong các tác vụ tiến hành tốt, cho đến khi, nó được thực hiện cho chính nó (trong một khoảng thời gian ngắn). Nếu chúng ta có một hệ thống các tiến trình, rằng một sự chuyển đổi bất kỳ sau một lệnh trong khoảng một tác vụ của một tiến trình, thì với cái đó, tạo điều kiện cho một sự chuyển đổi kịp thời của

một tác vụ khác. Bây giờ, chúng ta khảo sát bước đầu tiên của tác vụ "bứt ra" tiến trình. Chúng ta nhận thấy: tiến trình B cần thiết phải được bứt ra, tức là bước (1) và (2) được thực hiện để bứt tiến trình ra. Bấy giờ xuất hiện một ngắt thời gian, lát cắt thời gian của B là cuối cùng. Tiến trình A xuất hiện và treo vào danh sách trong khoảng tác vụ "treo vào"; nó còn tạo thêm một việc gì đó hay nằm yên. Nếu bây giờ B lại nhận sự điều khiển, do đó B tiếp tục làm việc một cách khó nhọc với các dữ liệu cũ. Trong bước thứ (3), móc treo được đặt vào vị trí C và với cái đó, thì tiến trình A không còn ở trong danh sách nữa; nếu đó là danh sách sẵn sàng, thì khi đó không nhận sự điều khiển nữa và cũng không làm việc.

Sự xảo trá của lỗi này là ở chỗ: nó không xuất hiện luôn luôn, mà chỉ xuất hiện một cách phiếm định, một cái gì đó không rõ ràng, một cách không thể tái định và nó chỉ xuất hiện do những điều kiện phụ được kết nối. Vì có cái đó xảy ra, cho nên nếu, *một tiến trình chạy vượt qua một tiến trình khác, thì người ta chỉ cái đó là điều kiện chạy đua*; đó là một đoạn mã mà trong đó lỗi tồn tại; và tất nhiên, các đoạn mã này không được phép bẻ gãy, chúng là những đoạn mã tới hạn (*critical section*).

Vấn đề này là đặc trưng cho mọi hệ thống, mà ở đó, nhiều phương tiện điều hành độc lập làm việc trên một dải dữ liệu. Đặc biệt tại những hệ thống đa vi xử lý, thì cái đó là một vấn đề quan trọng được các nhà thiết kế hệ thống nhận biết và quan tâm.

Vấn đề cơ bản là ở chỗ phải đảm bảo an toàn đối với mỗi khoảng tới hạn ở trong chương trình người sử dụng hay ở trong hệ điều hành: tức là tại đó luôn luôn chỉ có một tiến trình hay một vi xử lý. Việc dẫm lên hay loại bỏ mã phải được xác định đồng bộ giữa các tiến trình và phải đảm bảo một sự loại bỏ tương tác (*mutial exclusion*) ở trong khoảng tới hạn.

Vấn đề này đã được nhận biết trong những năm 60 và đã được xử lý với nhiều thuật toán khác nhau. Ở phương pháp giải quyết của E.W. Dijkstra (1965) đã đưa ra những yêu cầu sau đây:

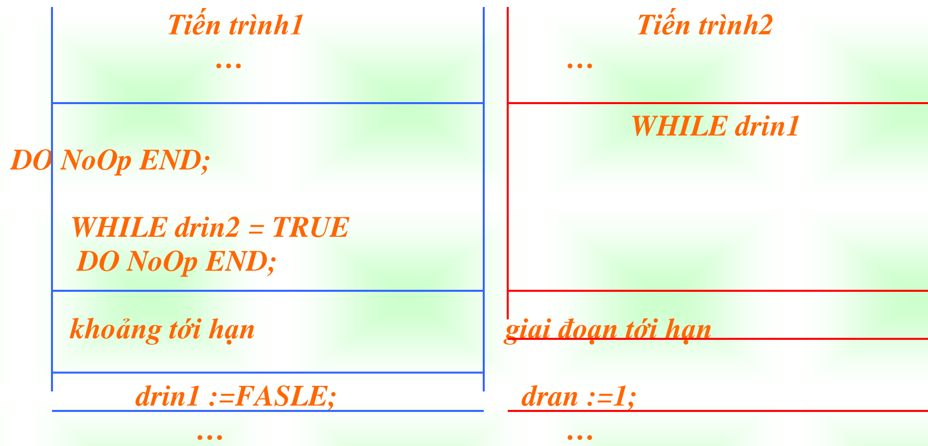
+ Hai tiến trình không được đồng thời ở trong khoảng tới hạn của chúng;

+ Không được phép hãm một tiến trình ngoài khoảng tới hạn của một tiến trình khác;

+ Mỗi tiến trình đang chờ ở lối vào của một khoảng tới hạn, thì nó phải được phép dẫm lên khoảng tới hạn: Một sự chờ đợi vĩnh viễn phải được chấm dứt (*fairness condition*).

2.3.2. Tín hiệu, cờ hiệu và các hoạt động nhân tử.

Ý tưởng đơn giản để tạo lập một sự loại bỏ tương tác là ở chỗ: một tiến trình phải chờ đợi tại lối vào ở khoảng tới hạn quá lâu, cho tới khi, khoảng tới hạn này trở lại trống. Đối với hai tiến trình diễn biến song song, thì mã của chúng được chỉ trong hình 2.20



Hình 2.20.Thí nghiệm đầu tiên để đồng bộ tiến trình

Nếu chúng ta thiết lập biến chung *dran* thí dụ với 1, do đó, mã nói ở trên đạt tới sự loại bỏ tương tác, nhưng cả hai tiến trình có thể thực hiện một cách thay đổi khoảng tới hạn. Điều đó tương ứng với yêu cầu (3) ở trên, vì một tiến trình tự cản trở để dẫm lên lần thứ hai khoảng tới hạn, mà không phải ở trong khoảng tới hạn.

Nếu chúng ta giả sử, để làm đồng bộ cả hai tiến trình trong dãy tuần tự bất kỳ, do đó, chúng ta lưu ý, rằng điều đó thì không đơn giản. Thí dụ, cấu trúc trong hình 2.21 có lợi, rằng một tiến trình chờ không lâu nữa ở một sự phân bổ đặc biệt, ngoài ra còn phải lưu ý thêm: một tiến trình khác thì không ở trong khoảng tới hạn.

Hình 2.21----------------

Tuy nhiên, ở đây có một lỗi khác xuất hiện: Người ta thấy rằng, các biến đồng bộ được thiết lập với drin2 := drin1 := FALSE. Tiến trình P_1 tìm thấy drin2 := FALSE và tiến trình P_2 tìm thấy drin1 := FALSE. Do đó, các tiến trình được đặt là drin1 := TRUE cũng như drin2 := TRUUE và cả hai đồng thời ở trong khoảng tới hạn, tức chúng mâu thuẩn với yêu cầu (1) theo sự loại bỏ tương tác (*mutical exclusion*).

Cũng vậy, một sự trao đổi của hai hàng (khoảng máu xám thứ nhất) không mang lại sự trợ giúp nào. Đầu tiên là đề nghị của T. Dekker nhà toán học Hà Lan đã chỉ ra phương hướng để giải quyết vấn đề đồng bộ của hai tiến trình (1965). Dựa vào đó, G. Peterson đã đưa ra (1981) một phương pháp đơn giản đồng bộ hai tiến trình, nó được chỉ ra trên hình 2.22 dưới đây.

Hình 2.22---------------

Ta có: Interesse1, Interesse2 và dran là các biến toàn cục. Cả 2 biến Interesse1, Interesse2 được thiết lập(mặc định) với FALSE. Chúng ta nhận thấy, một tiến trình làm việc với các dãy lệnh ở vùng màu xám trước khoảng tới hạn. Vì tiến trình khác đã không chỉ ra biến Interesse, cho nên, điều kiện của vòng lặp WHILE không được thực hiện và tiến trình dẫm lên khoảng tới hạn. Nếu một tiến trình khác xuất hiện hơi muộn, thì do đó, nó phải chờ đợi quá lâu, cho đến khi tiến trình khác bỏ qua khoảng tới hạn và không biểu lộ biến Interesse nữa.

Trong trường hợp cả hai tiến trình đồng thời dẫm lên vùng màu xám, thì đối với trường hợp này, sự làm việc của chúng được quyết định bởi một lệnh, mà với lệnh này, biến toàn cục dran được đưa vào. Tiến trình đưa vào cuối cùng thì làm giải thoát tiến trình khác và phải chờ đợi; còn tiến trình khác có thể dẫm lên khoảng tới hạn.

Người ta có thể trình bày bản phác thảo này một cách chắc chắn hơn với một kết cấu ngôn ngữ nhất định. Một ý tưởng cho cái đó là việc qui định một tín hiệu đối với một khoảng tới hạn. Qua đó, những tiến trình này tự động bộ và tín hiệu được quy định vừa nói tới được kiểm tra trước khi dẫm lên khoảng tới hạn. Nó được đưa ra với giá trị khởi xướng có thể dùng được. Tiến trình gọi sẽ bị hãm lại bởi lệnh: waitFor(Signal), trường hợp này không có tín hiệu nào được đặt vào, mà nó đặt tín hiệu trở lại và dẫm lên khoảng này. Nếu tiến trình gọi hoàn thành, thì nó tác động lên một trong các tiến trình với lệnh: send(Signal). Tiến trình nào được tác động, đó là công việc của một chiến lược dự định bổ sung.

Cấu trúc được E.W. Dijkstra nêu ra (1965), đã tạo điều kiện tới một sự kết thúc không hoàn hảo đối với một khoảng tới hạn. Ông gọi những tín hiệu này là cờ cầm tay (*Semaphore*). Những cờ tín hiệu này xuất hiện được hiểu như là một đèn hiệu giao thông. Chúng được quản lý thành 2 kiểu tác vụ sau đây:

- *Tác vụ vượt qua P(s)*:

Khi đi vào khoảng tới hạn thì P(s) được gọi với cờ hiệu s. Tiến trình gọi được thuyên chuyển vào trạng thái chờ đợi, cho tới khi một tiến trình khác ở trong khoảng đổi cờ tín hiệu.

- *Tác vụ rời khỏi* V(s):

Khi rời khỏi khoảng tới hạn, tiến trình V(s) gọi đến và gây nên cho cái đó một tiến trình chờ đợi được kích hoạt và được phép dẫm lên khoảng tới hạn.

Điều đó thì chưa thực chất, liệu là, đối với một tín hiệu hay một cờ hiệu chỉ tồn tại một khoảng tới hạn, mà khoảng này sẽ được tất cả các vi xử lý ở trong bộ nhớ chính thực hiện (các hệ thống đa vi xử lý); liệu là, có nhiều bản sao ở trong các tiến trình (thí dụ hệ thống nhiều máy tính), hay liệu, chúng là những khoảng mã khác nhau, mà chúng trao đổi chức năng lẫn nhau và chúng phải được đảm bảo với một tín hiệu toàn cục như ở thí dụ nói trên, đó là việc treo vào hay bứt ra ở trong một hàng đợi.

Thí dụ về sử dụng các tác vụ P(s) và V(s):

Nhiều tiến trình cần phải gia tăng một con số và giá trị của chúng phải được xuất ra, do vậy, một cách bình thường "1,2,3…" được đếm tăng lên. Khi đó, khoảng tới hạn được dẫn ra ở đây là:

```
…
z := z +1;
WriteInt(z,3);
…
```

Với cờ tín hiệu này thì điều được dẫn tới là:

```
...
P(s);   {* Kiểm tra: khoảng tới hạn đã bị chiếm giữ ?*}
z := z +1;  {* khoảng tới hạn*}
WriteInt(z,3);
WriteInt(z,3);
V(s);   {* Tiến trình chờ đợi được gọi tới*}
...
```

Người ta lưu ý rằng, các khoảng tới hạn không thể tự bẽ gãy, chúng vừa biểu thị các hàm send(Signal) và waitFor(signal) vừa diễn tả các hàm P(s) và V(s). Tuy nhiên, chúng chỉ rất ngắn, và cho nên chúng được thi hành một cách nhẹ nhàng hơn với tư cách là những hoạt động nhân tử. Một hoạt động nhân tử thì hoặc là được thực hiện hoàn toàn (tất cả các tác vụ của khoảng này) hoặc là chẳng được thực hiện gì cả (không có tác vụ riêng lẻ). Những bộ đệm chuyên dụng (*special buffer*) tạo điều kiện để nhớ trạng thái ban đầu. Nếu sự hoạt động nhân tử phải được bẻ gãy, thì do đó, trạng thái khởi điểm được tái sinh.

Thí dụ về các hoạt động nhân tử:

Đối với việc chuyển tiền, bạn muốn nhập 2 triệu đồng vào tài khoản của bạn. Đáng tiếc, trong khoảnh khắc này, một khiếm khuyết xuất hiện trong hệ thống các máy tính, mà trước khi số tiền có thể được nhập vào tài khoản; hệ thống máy tính phải dừng lại. Sau khi khởi động mới hệ thống, bạn chỉ có thể lựa chọn: hoặc để đón nhận lần nhập lại mới 2 triệu đồng và để chuyển số tiền cho tài khoản đích, hoặc không làm gì cả. Vì số tiền này đã không tới người nhận, điều ấy có nghĩa, đối với trường hợp của bạn thì cũng giống như trong trường hợp tiền bị đánh rơi. Với tư cách là khách hàng, bạn sẽ nhận biết một cách không chắc chắn sự thông báo về tình hình vừa qua của máy tính.

Trường hợp đón nhận sự cố này, nếu việc chuyển tiền được dự đoán là hoạt động nhân tử. Vì hoạt động đã không được kết thúc, do đó, tính chất phá huỷ đảm bảo cho bạn rằng, việc nhập sổ lần đầu không thực hiện và nó chỉ tồn tại với tư cách là sự thay đổi nhịp điệu: Bạn không bị thất lạc 2 triệu đồng. Việc nhập sổ ngay tức khắc được thực hiện lại ngay sau khi khởi động máy tính.

Việc đặt trở lại thì không có vấn đề gì trong các ngân hàng dữ liệu được phân bổ. Do đó, kiểu ngân hàng này được ưa chuộng đối với hầu hết các ngân hàng dữ liệu trung tâm và các ngân hàng dữ liệu được quản lý bởi các máy tính lớn.

Ý tưởng hiện đại để đồng bộ hoá tiến trình cho phép dẫn tới một vấn đề lớn: Mỗi một tiến trình phải chờ đợi một cách tích cực ở một chuỗi lệnh (*busy waiting*) cho đến khi đủ điều kiện để nó có thể dẫm lên khoảng tới hạn. Loại tác vụ chờ đợi này được gọi là spin locks (các khoá vòng quay). Sự chờ đợi này không chỉ chất tải lên các vi xử lý một cách không cần thiết, mà còn có thể gây tổn hại điều kiện cân bằng (*fairness condition*). Nếu chúng ta lưu ý tới điều kiện mà một tiến trình ưu tiên cao đón nhận ngay sự điều khiển sau đó; còn nếu là một tiến trình ưu tiên thấp thì phải ở trong khoảng tới hạn của nó. Trong trường hợp này thì một tiến trình ở lại lâu tại tác vụ *spin lock*, cho tới khi một tiến trình khác rời bỏ khoảng tới hạn. Vì tiến trình có ưu tiên thấp, cho nên, không có trường hợp nào tiến trình sẽ không phụ thuộc các ưu tiên thống kê và điều kiện cân bằng bị phá vỡ.

Việc chờ đợi chuỗi lệnh (*busy waiting*) đã tránh được một tiến trình chờ đợi phải ngủ lịm, mà sự điều khiển được chuyển giao bằng các câu lệnh waitFor(Signal) hay P(s).

Phần mềm thực thi:

Một trong các kiểu thực thi của các cờ hiệu được mô hình hoá: một cờ hiệu như là một bộ đệm, mà bộ đệm này được làm giảm lại sự đi tới của một tiến trình. Các tác vụ P(s) và V(s) được gọi là các hoạt động nhân tử. Sau đây, một thực thi kiểu *busy wait* được chỉ ra, chúng có thể được sử dụng như thế nào ở trong nhân hệ điều hành để chắn đường trong khoảnh khắc. Ở đây, đầu tiên lấy s =1.

```
PROCEDURE  P(VAR s: INTEGER)
BEGIN
    WHILE s<=0 DO NoOp END;
     s := s-1;
END;

PROCEDURE  V(VAR s: INTEGER)
BEGIN
    s := s+1;
END V;
```

Một giải pháp tiến bộ hơn đối với tiến trình của người sử dụng và đối với thời gian chặn lại lâu thì được thực hiện bởi lệnh *sleep* và *wakeup()* và cờ hiệu được lưu ý như là các biến được cấu thành, mà chúng đón nhận một danh sách các tiến trình chờ. Việc thực thi danh sách các tiến trình thì không được chỉ ra ở đây. Còn s được lấy giá trị ban đầu s =1.

```
TYPE Semaphor = RECORD
                      value : INTEGER;
```

```
                        list : ProcessList;
                       END;
PROCEDURE  P(VAR s: Semaphor)
BEGIN
        s.value := s.value -1;
        IF s.value <0 THEN
            Einhangen (MyID, s.list); sleep;
        END
END P;

PROCEDURE  V(VAR s: Semaphor)
VAR  PID: ProcessId;
BEGIN
  IF s.value <0 THEN
     PID := aushangen(s.list); wakeup(PID);
  END;
s.value := s.value +1
END V;
```

Phần cứng thực thi:

Đối với việc lập trình hệ thống và ngay cả trong phạm vi đa vi xử lý, một sự trợ giúp quan trọng chỉ ra việc thực thi của các hoạt động nhân tử bằng phương tiện phần cứng. Đến nay, có nhiều phiên bản khác nhau, có thể giới thiệu vắn tắt như sau:

- *Loại bỏ ngắt*:

Một trong các phương pháp đơn giản cho các hoạt động nhân tử ở trong các hệ thống đơn vi xử lý là loại bỏ tất cả sự làm gián đoạn bằng việc sắp đặt các bit trạng thái ở trong CPU cũng như ở bộ điều khiển ngắt. Điều đó có thể mang lại hiệu quả phụ, ví dụ, nếu có ngắt hẹn thời bị loại bỏ và việc đếm thời gian đến tuần tự thì tại thời điểm ách tắt dòng, ngắt power failure sẽ bị từ chối và các thanh ghi không thể được cứu thoát nữa…Tốt nhất là, không cần cản lại các ngắt và với lý do này, cần mượn vay ưu tiên có khả năng cao nhất cho tiến trình ở trong khoảng tới hạn.

- *Dãy lệnh nhân tử*:

Ở các thiết bị đa vi xử lý, sự ngăn cản các ngắt cho thấy không có phương tiện thích ứng. Ở đây, những hoạt động nhân tử (*atomic operation, atomic action*) đã giữ được hình dáng các lệnh máy không thể bị bẻ gãy, các lệnh này thực hiện một lệnh máy tổng hợp ở trong chu kỳ nhớ riêng lẻ. Thídụ về các cái đó là:

+ Lệnh Test And Set (*thử nghiệm và thiết đặt*):

Lệnh *Test And Set* sẽ lựa chọn nội dùng của một phần tử nhớ và thay thế nó vào khoảng chu kỳ nhớ như vậy qua một giá trị khác. Điều đó được chuyên môn hoá bởi một hàm:

```
PROCEDURE  TestAndSet(VAR target: BOOLEAR): BOOLEAR
VAR tmp: BOOLEAR;
BEGIN
tmp :=  target;  target := TRUE; RETURN  tmp;
END Te stAndSet;
```

Việc đặt lui FALSE chỉ được thực hiện bởi một tiến trình và có thể xẩy một cách bình thường mà không có hoạt động *atomic action.*

+ Lệnh swap (tráo đổi)

Người ta có thể khái quát tác vụ ở trên và chuyên môn hoá giá trị viết lui tới. Khi đó hàm swap tráo đổi trị số các biến nguồn và đích với nhau:

```
PROCEDURE  swap(VAR source, target: BOOLEAR)
VAR tmp: BOOLEAR;
BEGIN
   tmp := target; target := source;  source:= tmp;
END swap;
```

+ Lệnh *Fetch And Add* (*lấy về và cộng vào*)

Lệnh này lựa chọn nội dung của một phần tử nhớ và viết nó vào trong chu kỳ nhớ tương tự một giá trị mới hay một giá trị được chọn làm tăng lên một con số.

```
PROCEDURE  fetchAndAdd(VAR a, value:INTEGER): INTEGER;
VAR tmp :INTEGER;
BEGIN
     tmp := a;  a:=tmp+value;  RETURE tmp;
END fetchAndAdd;
```

Điều đã rõ, với các lệnh nhân tử của phần cứng, lệnh chờ đợi *busy wait* của một khóa vòng (*spin lock*) có thể được thực thi một cách dễ dàng đối với một sự loại bỏ không đáng kể. Nếu chúng ta sử dụng điều này để thực thi các tác vụ nhân tử P() và V() của cờ hiệu, điều ấy có nghĩa rằng, những tác vụ cờ hiệu đã được lưu ý ở phần cứng, mà không cần thực hiện các lệnh máy phức tạp đối với các thủ tục P() và V().

Thí dụ về đồng bộ đa vi xử lý:

Trong các thiết bị đa vi xử lý với bộ nhớ chung, một sự đồng bộ phần cứng chỉ được thực hiện khi có một lệnh nhân tử, vì những ngắt chung không tồn tại. Những lệnh nhân tử của các vi xử lý riêng lẻ này được thực hiện song song với

nhau và chúng không thể tách chia nhau, vì chúng xảy ra trong sự kết nối chặt chẽ với hệ thống tiếp nhận bộ nhớ, mà hệ thống này đối với một chỗ nhớ chỉ tồn tại một lần và nó đảm bảo cho cái đó trong một thời gian ngắn (một chu kỳ nhớ) một sự tiếp nhận thông thường trên các biến toàn cục.

Việc thực thi các tác vụ cờ hiệu ở hệ thống đa vi xử lý với một lệnh máy nhân tử có thể nhìn nhận như sau:

```
VAR s: INTEGER; (*lấy giá trị đầu s=1*)
PROCEDURE  P(S);
BEGIN
  LOOP
     WHILE s>1 DO NoOp  END;                          (a)
      IF (FetchAndAdd(S,-1)>=1) THEN RETURN END;  (b)
      FetchAndAdd(s,1);                               (c)
   END;
END P;

PROCEDURE  V(s);
BEGIN
FetchAndAdd(s,1);
END V;
```

Những điều này được A. Gittlieb thực hiện lần đầu (1981) và đã chỉ ra, bên cạnh lệnh LOOP không chỉ có một vòng đợi ở trong P(), còn có một hàng đợi ở trong dòng lệnh (a) với vòng lặp WHILE.

Nhờ sự bao quát của hệ thống đa vi xử lý, điều ấy thì có lý, nếu chúng ta thoát khỏi vòng lặp WHILE, do đó, phép thử với FetchAndAdd(FAA) ở trong dòng lệnh (b) sẽ không đạt nữa. Thật vậy, nếu chúng ta lưu ý trường hợp, mà tại đó ba vi xử lý P(s) cũng đồng thời thực hiện một thủ tục (Procedure), khi đó người ta lấy giá trị ban đầu s=1. Cả ba vi xử lý cần thiết nhận sự trả lại đối với s các giá trị 1, 0 và -1, và cuối cùng đạt được s=-2. Bộ vi xử lý đầu tiên dẫm lên, rồi rời khỏi khoảng tới hạn, còn hai vi xử lý kia chờ đợi vòng lặp cho đến khi mọi chuyện xong xuôi.

Chúng ta nhận thấy rằng, vi xử lý thứ hai bấy giờ đã hạ thấp xuống lệnh FAA thứ nhất ở dòng lệnh (b) tại ngay cờ hiệu với s= - 1, khi vi xử lý thứ nhất thực hiện tác vụ V() của nó và gia tăng cờ hiệu với s= 0. Nếu bây giờ, bộ vi xử lý thứ hai thực hiện lệnh FAA thứ hai ở dòng lệnh (c) và gia tăng cờ hiệu với s = 1, do đó, sau khi bộ vi xử lý này thực hiện lệnh FAA ở dòng lệnh (b) như bộ vi xử lý thứ nhất, thì nó có thể rời bỏ tác vụ P() một cách bình thường. Sự việc này có thể được phòng ngừa nhờ bộ vi xử lý thứ ba, lúc đó, nó thực hiện lệnh FAA ở dòng lện (b) mà trước đó bộ vi xử lý thứ hai đã thực hiện dòng lệnh (c) và do vậy, cờ hiệu đã được giảm xuống trở lại với s = -1; tiếp theo, cờ hiệu được tăng lên bởi vi xử lý thứ hai với s = 0 (đáng lẽ s = 1).

Nhờ dãy điều chỉnh (*timing*) [P_3(b),P_2(c),P_2(b),P_3(c)], … thì một dãy cờ hiệu với s = 0,-1,0, -1… là có thể thực hiện được, ở trong đó, các cờ hiệu không có giá trị đúng phải được loại bỏ khỏi một trong các bộ vi xử lý từ vòng đợi: cả hai bộ vi xử lý được hãm lại, dụ cho khoảng tới hạn còn trống.

Bấy giờ, nhờ vòng lặp WHILE mà điều đó được phòng ngừa, vì ở đây, s chỉ được đọc, chứ không được thay đổi, do đó, tất cả các tiến trình bị loại bỏ…một cách đơn giản: tiếp tục chờ đợi, chứ không có phương cách nào hơn.

2.3.3 Cờ hiệu trong Unix

Ở một vài phiên bản của hệ điều hành Unix, có sự tiếp nhận cờ hiệu như là việc gọi hệ thống. Sau đây chỉ ra các gọi hệ thống ở trong HP-UX:

lockf	Tác vụ cờ hiệu để truy xuất files
msem_init	Việc khởi xướng một cờ hiệu để trích nạp bộ nhớ
msem_lock	Việc ngăn hãm một cờ hiệu
msem_unlock	Giải phóng một cờ hiệu
msem_remove	Sự loại bỏ cờ hiệu để truy cập bộ nhớ
semct1	Tác vụ điều khiển cờ hiệu nói chung
semget	Đón cờ hiệu
semop	Tác vụ cờ hiệu

Cú pháp và ngữ nghĩa của các tác vụ cờ hiệu được phân biệt khác nhau từ phiên bản này tới phiên bản khác của hệ điều hành Unix. Các tác vụ nhân tử ẩn thì quan trọng hơn các cờ hiệu ẩn, những tác vụ này được đưa ra một chuẩn để thực thi các cờ hiệu.

Thí dụ việc tạo ra một file (nói chính xác hơn: tạo ra một thông tin quản lý file kèm theo file) chính là việc tạo ra một tác vụ nhân tử: hoặc là tồn tại một file mới với các tên file được tạo lập theo gọi hệ thống, hoặc là một thủ tục đưa ra một thông báo lỗi, thí dụ: tên file tồn tại hai lần. Điều cần phải loại trừ là hai tiến trình đồng thời sinh ra hai file khác nhau với các tên file tương tự. Cho nên, người ta sử dụng điều đó để tạo khả năng cùng phô diễn giữa việc tạo lập dữ liệu và sự biểu lộ tiến trình (đó là kiểu của người sử dụng và nhà sản xuất) đối với việc phô bày các file (*printer demon*). Những file sinh ra cho cái đó cũng như cờ hiệu được biểu thị là người thợ khoá hay file khoá (*lock file*).

Ở hệ điều hành Unix, các hoạt động nhân tử được yêu cầu rằng, một tiến trình (của một loại nhân tử) thì có thể không bị bẻ gãy. Thật vậy, nếu quá lâu mà nó vẫn chưa nhận được sự điều khiển, cho đến khi nó được đặt yên tĩnh trong thủ tục nhân (*kernel procedure*) hay trở lại trạng thái người sử dụng.

Ở các hệ thống đa vi xử lý của Unix, các cờ hiệu ở trong nhân được đặt tới toạ độ của các vi xử lý. Ở phiên bản HP-UX (*Hewlett- Packard Unix version*), tất cả các khoảng tới hạn đối với các dữ liệu của nhân quan trọng và các bảng được bảo vệ nhờ cờ hiệu **busy wait**.

2.3.4 Cờ hiệu ở trong Windows NT

Có nhiều kiểu kết cấu đồng bộ trong hệ điều hành Windows NT. Đối với sự đồng bộ cổ điển, các tiến trình và các xâu (*threads*) có các lệnh gọi hệ thống CreatSemaphore() và OpenSemaphore() mà nó sinh ra các cờ hiệu ở trong không gain biến toàn cục (giống như một file) cũng như bắt đầu việc truy cập về cái đó. Các tác vụ P() và V() thì phù hợp với các thủ tục WaitForSingleObject(Sema, TimeOutValue) và ReleaseSemaphore().

Về cái đó, người ta chọn các cờ hiệu với tư cách là những bộ đếm với một giá trị kết thúc lớn nhất hay với tư cách là những biến nhị phân cục bộ. Qua việc truy cập hệ thống ở trong các biến toàn cục thì cấu trúc cờ hiệu này với một tiêu phí gia tăng là có thể đáp ứng được. Do đó, đối với toạ độ của các vòng quay kín (*spin locks*) ở trong khoảng một tiến trình thì cờ hiệu trọng lượng nhẹ phụ sẽ được tạo lập, mà những cờ hiệu này chỉ quen thuộc ở trong khoảng một tiến trình hay trong khoảng một chương trình. Cờ hiệu này được gọi là khoảng tới hạn và được bắt đầu với lệnh InitialezeCriticalSection(s). Ở đây, các tác vụ P() và V() cũng có ý nghĩa là EnterCriticalSection(s) và LeaveCriticalSection(s).

Vì hệ điều hành Windows NT thì rất lý tưởng đối với các hệ đa vi xử lý kết nối chặt chẽ với bộ nhớ quảng đại, cho nên, nhân của hệ điều hành Windows NT được sử dụng thuận tiện cho việc đồng bộ đa vi xử lý. Ở đây, vòng quay kín (*spin lock*) được sử dụng như là cờ hiệu, do đó, một xâu (*thread*) với một vòng quay kín sẽ hãm giữ bộ vi xử lý thật lâu, cho tới khi nó lại rời bỏ vòng quay kín này. Các tác vụ vòng quay kín này chỉ được sử dụng để dịch vụ hệ điều hành Windows NT Executive. Tuy nhiên, chúng vẫn có một vài hạn chế: thí dụ ở khoảng tới hạn thì không có tham chiếu nào tới khoảng nhớ được thực hiện, mà chúng đã được nạp trên bộ nhớ quảng đại, người ta có thể không cần gọi các thủ tục bên ngoài hay có thể không cần loại bỏ ngắt hoặc một ngoại tệ nào.

Đối với các dịch vụ cao hơn thì cần tới các đối tượng nhân có những tính chất đồng bộ khác nhau và các đối tượng nhân này được quy tụ trong hệ thống định thời bình thường. Ví dụ về các đối tượng đó là các cờ hiệu, các biến cố (*event*), các cặp biến cố, các bộ hẹn thời, các quay vòng kín, các đối tượng loại trừ lẫn nhau…

Các đối tượng có thể tồn tại ở hai trạng thái: báo trước và không báo trước. Thí dụ, một đối tượng cờ hiệu đi qua trạng thái báo trước, nếu như bộ đếm chỉ số không (0) và rồi tất cả trở thành vòng chờ trống. Ở trong hệ điều hành Windows NT, một vòng chờ có thể chờ đợi trên những vòng chờ trống khác, chờ đợi các biến cố hay chờ đợi cờ hiệu và nhờ thế, nó có thể được làm đồng bộ. Win32 API thì sử dụng các thủ tục WaitForSingleObject() và WaitForMultipleObject(). Thí dụ, một vòng chờ ở trong một chương trình tờ trãi (hay còn gọi là chương trình bảng tính) có thể gọi một vòng chờ khác. Nếu người sử dụng kết thúc chương trình, do đó, tiến trình chính với lệnh WaitForSingleObject() chờ đợi sự kết thúc của tiến trình cưỡng bức, trước khi chương trình được kết thúc hoàn toàn và tất cả các dữ liệu bên trong bị loại bỏ.

2.3.5 Các ứng dụng

Cờ hiệu được sử dụng rộng rãi cho những vấn đề làm đồng bộ khác nhau. Ở đây, một vài ví dụ quan trọng và phổ dụng được giới thiệu như sau đây.

Đồng bộ các tiến trình:

Tác vụ P() đối với cờ hiệu để ngăn hãm một tiến trình thật lâu, cho tới khi nó được tự do nhờ tác vụ V(). Điều đó chúng ta có thể sử dụng để đem lại sự tính toán cho sự phụ thuộc tương hỗ trong một hệ thống tiến trình và để khẳng định một cách chính xác dãy tuần tự của tác động tiến trình.
Về cái đó, chúng ta định nghĩa cho mỗi một sự phụ thuộc với một cờ hiệu đặc biệt để bắt đầu hay kết thúc sự chờ đợi của tiến trình.

Thí dụ về sự đồng bộ bằng cờ hiệu:

Dưới đây chỉ ra sơ đồ tương quan của năm tác cụ A, B, C, D và E:

Hình ------------

Bây giờ, các tiến trình được diễn giải một cách tách biệt như sau:

```
Tiến trình A: BEGIN                TaskBodyA;  V(b);  V(c);  END A;
Tiến trình B: BEGIN   P(b)         TaskBodyB;  V(d1);  V(e);  END B;
Tiến trình C: BEGIN   P(c)         TaskBodyC;  V(d2);          END C;
Tiến trình D: BEGIN    P(d1);      P(d2)    TaskBodyD;         END  D;
Tiến trình E: BEGIN      P(e )      TaskBodyE;                 END E;
```

Các cờ hiệu b,c,d1,d2, e phải được định nghĩa là các biến toàn cục và giá trị đầu của chúng đều lấy bằng không (0). Qua đó, tất cả các tiến trình đều chờ đợi cờ hiệu cho sự hoạt động của chúng, nếu chúng không dẫn tới những khoảng thời gian khác nhau. Điều cần quan tâm là sự tác động của chúng chưa được mở, tức là, tại đây một tiến trình đến quá muộn: Tín hiệu đánh thức đã được nạp trong cờ hiệu, do đó, nó không thể mất đi được.

Quan hệ sản sinh và tiêu dùng:

Các nhiệm vụ như thiết lập dữ liệu, quản lý dữ liệu, lọc dữ liệu đều được mô tả bởi một hệ thống của hai hay nhiều tiến trình và nhờ các tiến trình này mà một trong các dữ liệu vừa nêu được thiết lập, còn một trong các dữ liệu khác bị loại bỏ. Thật vậy, xin xem ví dụ về hệ điều hành Unix ở đầu chương 2 này.
Nếu chúng ta biểu thị việc chế tạo (sản sinh) là nhà sản xuất, còn việc sử dụng là người tiêu dùng, do đó, quan hệ nhà sản xuất và người tiêu dùng cũng giống

như việc thiết lập và loại bỏ các dữ liệu với các tỷ phần khác nhau (số dữ liệu trên một đơn vị thời gian). Nói chung, người ta sẽ đưa một bộ đệm (buffer) vào giữa các tiến trình, mà do đó, bộ đệm này sẽ được làm đầy bởi việc tạo lập và được làm trống bởi việc tiêu dùng. Vì vậy, bộ đệm chỉ kết thúc khi mà dòng dữ liệu giữa việc tạo lập được nghỉ ngơi khi bộ đệm đã được điền đầy và nó được người tiêu dùng gọi làm việc trở lại khi bộ đệm đã vơi cạn. Tương tự như vậy, đối vời người sử dụng cũng được sử lý ngược lại cho phù hợp. Hình 2.23 chỉ ra sự thiết đặt đơn giản, mà ở đó, dung lượng của bộ đệm với N và biến toàn cục used được lấy giá trị đầu bằng không (0).

Hình 2.23----------------

Giải pháp này có một vấn đề cần lưu ý: Liệu nó có dễ dàng dẫn tới một điều kiện chạy đua(?). Để xem thử thế nào, chúng ta khảo sát trường hợp sau đây. Khi bộ đệm vơi cạn, thì sự tiêu dùng cho biết ngay used = 0, và đồng thời sự sinh sản được chuyển đổi. Khi sự sinh sản tạo ra được một lượng nào đó (*item*), thì nó đẩy bộ đệm và làm gia tăng biến used lên bằng 1, cho nên, bộ đệm muốn đánh thức người tiêu dùng. Nhưng mà, nó vẫn chưa bị đánh thức: đầu tiên, ở sự phân bổ vi xử lý gần đó, nó vẫn ở trạng thái ngủ, vì lúc đó biến used vẫn bằng không (0). Sau khi việc sinh sản đã làm đầy bộ đệm, thì tương tự, nó ở trạng thái ngủ. Sau đó cả hai tiếp tục ngủ mãi mãi.

Sở dĩ có tình trạng trên, vì tín hiệu đánh thức (*wakeup-signal*) đến rất sớm và trôi qua mau. Để loại bỏ tồn tại này thì giải pháp khắc phục là: bit đánh thức (*wakeup-bit*) được lưu trữ, để nó trợ giúp sau những thời gian ngắn, tại các tiến trình tiếp theo, các bit này được dẫn vào. Nếu chúng ta có một số lượng không rõ các tiến trình, điều đó thì cũng giống như việc các bit đánh thức này tồn tại ngoài các hệ thống, và do đó, giải pháp chưa giải quyết trọn vẹn vấn đề.

Bởi vậy việc dẫn tới các cờ hiệu cho một giải pháp được chỉ ra. Vì cờ hiệu được gia tăng rồi lại giảm đi, do đó, tất cả các tín hiệu đánh thức được lưu trữ hay được giải thoát khỏi sự cần thiết của các bit trạng thái riêng biệt. Hình 2.24 chỉ ra một giải pháp với các cờ hiệu được lấy các giá trị: belegt:= 0, free := N, mutex :=1.

Hình 2.24-----------------

Phương pháp thực thi tương tác để dẫn tới bộ đệm được đảm bảo nhờ cờ hiệu *mutex*. Với cờ hiệu *belegt* thì con số của không gian bộ đệm được đếm, còn cờ hiệu *free* là con số của không gian trống của bộ đệm được đếm. Do việc đọc chọn của bộ đếm cũng như các cờ gọi *wakeup* và *sleep*, cho nên các tác vụ cờ hiệu được gọi, mà tại đó, một cờ hiệu riêng lẻ được dùng với các điều kiện biên (ghi ngủ) used = N và used = 0. Các tiến trình sản sinh chỉ giảm con số của phần không gian trống, nếu chúng bằng không, thì tiến trình bị hãm lại, trước khi nó có thể xếp vào vị trí kế tiếp. Nếu nó được xếp vào vị trí, thì do đó nó tăng con số vị trí được đặt

vào và đồng thời đánh thức tiến trình tiêu dùng, mà tiến trình này được đặt vào vị trí ngủ (used =0).

Người ta lưu ý rằng, giải pháp này nhờ sự khởi đầu của cờ hiệu *mutex* thì tác dụng lên một và chỉ một sự loại trừ tương hỗ ở khoảng tới hạn của việc quản lý bộ đệm. Ở việc sản sinh và việc lấy đi (tiêu dùng), tồn tại nhiều tiến trình sản sinh và nhiều tiến trinh tiêu dùng với cùng loại mã và do đó có thể làm đầy hay làm trống bộ đệm, mà không hề có một sự xáo trộn nào.

Vấn đề đọc-viết:

Để xác định việc đọc và viết nhiều tiến trình trên phạm vi bộ nhớ chung, vấn để cơ bản của việc đồng bộ là xác định lại việc đọc và viết các dữ liệu. Yêu cầu cơ bản là phải làm tách biệt việc đọc và viết đồng thời trên phạm vi dữ liệu chung và phải tránh các dữ liệu không xác định. Cái đó được điều chỉnh nhờ một cờ hiệu riêng lẻ. Nếu chúng ta thiết lập những yêu cầu phụ thêm, thì do đó, các giải pháp trở nên phức tạp. Để xem lại một giải pháp hoàn thiện thì cần lưu ý những điểm sau đây:

+ Một người cần thiết phải chờ đợi, khi một người viết đã nhận được luật truy cập của việc ghi chép (gọi là vấn đề thứ nhất của việc đọc-viết). Điều đó có nghĩa rằng, không có người đọc nào phải chờ đợi một người đọc khác, mà chỉ phải chờ đợi một người viết.

+ Nếu một người viết đã sẵn sàng, anh ta sẽ thực hiện việc viết nhanh như anh ta có thể (gọi là vấn đề thứ hai của việc đọc -viết). Đồng thời, nếu một người viết đã sẵn sàng rồi, để nhận được luật truy cập, thì không một người đọc mới nào có thể bắt đầu đọc.

Người ta lưu ý rằng, giải pháp cho vấn đề thứ nhất là cần phải thực hiện: người viết phải chờ đợi mãi mãi; còn giải pháp cho vấn đề thứ hai là phải thực hiện: người đọc phải chờ đợi mãi mãi. Trong cả hai trường hợp,các tiến trình bị làm đói: vì việc chờ đợi chỉ phụ thuộc vào dòng đọc hay chỉ phụ thuộc vào dòng viết và không bị tiêu biến đi bởi sự phong toả nào.

Một giải pháp có thể đạt được đối với vấn đề thứ nhất là thực hiện một bộ đếm chung để ghi số đọc ở khoảng tới hạn. Để đọc được ở trong khoảng tới hạn, thì người viết bị hãm lại, ngoài ra không có gì hơn. Cho cái đó có hai cờ hiệu *ReadSem* và *RWSem*: một cờ hiệu để bảo vệ bộ đếm khi đọc và một cờ hiệu để điều chỉnh lối dẫn vào đọc- viết. Mã giả để đọc (Reading) được viết như sau:

```
P(ReadSem);
    readcount:= readcount +1;
    IF  readcount = 1 THEN P(RWSem) END;
V(ReadSem);
…
Reading_Data();
…
P(ReadSem);
  readcount:= readcount - 1;
```

```
    IF  readcount = 0 THEN V(RWSem) END;
V(ReadSem);
```

Mã giả cho viết (*Writing*) như sau:

```
P(RWSem);
…
Writing_Data();
…
V(RWSem);
```

Trường hợp người viết ở trong khoảng tới hạn, do đó, người đọc thứ nhất được hãm trên cờ hiệu *RWSem*; còn tất cả những người đọc khác cũng chờ đợi cờ hiệu *RWSem*, do vậy việc định thời cần quyết định: cờ hiệu nào được đánh thức tại tác vụ V(RWSem) của tiến trình viết thứ hai.

Quản lý hàng đợi đa vi xử lý ở máy tính NYU-Ultra:

Một trong các hệ thống vi xử lý song song được quan tâm của những năm 80 là chiếc máy tính NYU-Ultra của trường đại học New York(1983). Nó có một cấu trúc đa vi xử lý có sơ đồ nguyên lý được chỉ ra ở hình 1.10. Đối với việc định thời thì một thuật toán đặc biệt đã được thực thi. Thuật toán này đã sử dụng tác vụ *fetch and add* để làm đồng bộ các vi xử lý trên bộ nhớ chia xẻ và ở đây, nó được mô tả một cách ngắn gọn.

Ở việc truy cập song song, hàng đợi của các tiến trình sẵn sàng (*ready queue*) được hoạt động nhờ hệ thống đa vi xử lý, mà hệ thống này đã tạo ra các tính chất kết nối FIFO: Trường hợp các tác vụ kết nối làm kết thúc Job *p* trước khi bắt đầu Job *q*; sau đó, cái đó hết điều kiện và rồi tác vụ lấy đi đối với Job *q* được kết thúc trước tác vụ lấy đi đối với Job *p*. Thuật toán sau đây dẫn tới một bộ đệm hình xuyến có chiều dài N được chỉ ra trên hình 2.25:

Hình 2.25---------------

Các biến toàn cục *InSlot* và *OutSlot* được biểu thị bởi các rãnh (*Slot*) ở trong bộ đệm hình xuyến, mà ở đó, một Job mới được treo vào, cũng như thế, Job cũ có thể được lấy đi. Bộ đệm có ba vùng: vùng điền vào, vùng lấy đi và vùng cố định (*fix*), trong vùng cố định không có sự hoạt động nào.

Đầu tiên, bộ đệm trống, khi các biến có giá trị *InSlot = OutSlot=0.* Tiếp theo, khoảng cố định dịch chuyển một cách chậm chạp theo hướng kim đồng hộ ở trong bộ đệm hình xuyến, mà ở đó, các biến *InSlot*, *OutSlot* được bảo đảm ở giữa các biến sơ khai *add* và *fetch*. Vì vấn đề giải phóng bộ đệm cần phải được quan tâm đến biến Full:= TRUE cũng như khi bộ đệm trống cần lưu ý biến Empty :=TRUE, do đó, vấn đề quản lý hàng đợi trở nên khó khăn hơn. Vì tại thời điểm đầu tiên là việc treo tiến trình vào hàng đợi thì biến *InSlot* được gia tăng, còn tại thời điểm

thứ hai là việc bứt tiến trình ra khỏi hàng đợi thì biến *OutSlot* được gia tăng, nhưng đáng tiếc, điều đó không còn đúng nữa: Thật vậy, ta nhận được Full=(*InSlot = OutSlot=N*) hay Empty=(*InSlot = OutSlot=0*), vì khi đó, nó có thể đem lại một dãy các khe vừa được điền đầy nhưng chưa hoàn hảo. Từ lý do này, đối với trường hợp Full (làm đầy) thì điều kiện đặt ra InUse = N được dò kiểm tra, còn đối với trường hợp Empty (làm vơi) thì điều kiện Fix=0 được dò kiểm tra. Số lượng các bộ vi xử lý đang làm việc trong hàng đợi được phân phối nhờ các biến InUse và Fix.

Hình 2.26------------------

Mã ở trong hình 2.26 có độ tinh vi khác nhau. Trong thực tế, không có cờ hiệu nào phù hợp với việc thực thi các tác vụ cờ hiệu như đã chỉ ra ở ví dụ với tác vụ fetch and add; nó được dùng để bảo vệ khoảng tới hạn, bảo vệ việc treo vào hay bứt ra các dữ liệu của bộ đệm tổng cộng cũng như để đạt được lối dẫn vào đã được điều khiển. Trước khi bước kiểm tra lần thứ hai được quay trở lại với thông báo lỗi cho các trường hợp Full và Empty, cái đó cũng rất cần thiết đối với các điều căn bản được nhắc tới trong ví dụ này. Điều có ý nghĩa là, tiến trình gọi phải quyết định: cái gì phải làm trong các trường hợp miễn giảm này.

Hai cờ hiệu vừa nói ở trên có thể cùng được thực thi nhờ các tác vụ fetch and add và nó là bản thảo đầu tiên (1983) của tác giả A. Gottlieb. Ngoài ra, sự tồn tại của chúng còn có ý nghĩa khác. Thật vậy, nếu cả hai tác vụ đầu tiên được làm đầy trong danh sách còn trống, thì do đó, bộ vi xử lý thứ hai hoàn thành công việc trước. Bây giờ, tác vụ bứt ra tiếp theo chuyển ngay mức 1, mà không cần phải ghi nhớ rằng, cái đó không tồn tại một cách hiện hành và các dữ liệu bị báo lỗi sẽ được sử dụng tiếp tục. Do đó, không chỉ số lượng các sản sinh (tiến trình được treo vào hàng đợi) sẽ được điều chỉnh, mà với cái đó, vấn đề bộ đệm nhảy (*bounded buffer*) cũng được điều chỉnh. Chẳng những thế, mỗi một rãnh riêng lẻ của bộ đệm hình xuyến được bố trí cho một cờ hiệu. Do đó, nó dẫn tới kết quả: hoặc là treo vào, hoặc là bứt ra được thực hiện trên rảnh, nhưng cả hai không được đồng thời. Việc sử dụng hai cờ hiệu này phải đảm bảo các điều kiện ban đầu InSem[1]:=1 và OutSem :=0, để tại mỗi rảnh trước tiên xảy ra sự điền đầy, sau đó xảy ra sự bứt ra, nghĩa là có sự chuyển đổi liên tục.

2.3.6 Khoảng tới hạn và việc dò kiểm tra

Việc đưa vào các cờ hiệu đã làm giảm nhẹ một cách đáng kể việc đồng bộ các tiến trình. Cho nên, cách sử dụng các cờ hiệu này cũng không có vấn đề gì. Thật vậy, nếu chúng ta thay đổi dãy các tác vụ đi tới V(mutex);..critical duration..; P(mutex); thì do đó, kết quả sẽ đảo ngược ngay tới điều được mong muốn: Tất cả thì được phép ở trong khoảng tới hạn. Còn lỗi của các tác vụ V(mutex);..critical duration..; hay sự loại bỏ của một trong hai tác vụ đều dẫn tới những vấn đề lớn. Về cái đó, một chương trình con(subtile) sẽ đạt được sự thay đổi không đáng kể

trong các chương trình lớn, để sinh ra các lỗi chẳng hề mong muốn., mà nó chỉ xuất hiện một cách phiến diện. Việc trao đổi các tác vụ ở giải pháp của vấn đề sản sinh và tiêu dùng được biểu thị bằng các hàm dưới đây:

+ cho sản sinh: P(free), P(mutex), putInBuffer(item), V(mutex), V(belegt)

+ cho tiêu dùng: P(mutex), P(free), putInBuffer(item), V(mutex), V(belegt)

Do đó, người ta đã đề nghị rằng (1972), việc sản sinh và sắp xếp các tác vụ cờ hiệu *đúng* đã phó mặc cho việc biên dịch , và vì vậy, cấu trúc một ngôn ngữ mới được thực hiện cho khoảng tới hạn. Cú pháp ngôn ngữ đối với một biến nào đó (thí dụ cờ hiệu) của một kiểu bất kỳ là:

region s do <tuyên bố>

và kết quả thích hợp cho một cái gì đó sẽ là:

P(s); <statement>; V(s);

Nếu có nhiều tiến trình thì chúng sẽ dẫm lên khoảng tới hạn với biến s, do đó, các cơ cấu đồng bộ (được sự biên dịch sinh ra) đảm bảo rằng, luôn luôn chỉ có một tiến trình.

Nếu có hai tiến trình xảy ra song song:

region s do s1 và region s do s2

thì chúng sẽ sinh ra dãy tuần tự có dạng s1, s2…hay s2, s1…, do đó, dãy tuần tự các lệnh s1 và s2 là bất kỳ, đồng thời việc thực thi các lệnh này không được pha trộn.

Nếu người ta còn dẫn thêm một điều kiện B, thì nó sẽ điều chỉnh lối dẫn vào một khoảng tới hạn như vậy, có dạng:

region s when B do <statement>

do đó, chúng ta nhận được một khoảng tới hạn có điều kiện (*conditional critical region*). Sơ đồ của khoảng tới hạn có điều kiện của cấu trúc này được dự đoán rằng, một tiến trình sau khi xác nhận biến s thì có thể dẫm lên khoảng tới hạn, khi điều kiện B cũng được làm đầy. Nếu điều đó không xảy ra, thì biến s được gởi đi trở lại và do đó tiến trình lại nằm ngủ cho đến khi điều kiện được làm đầy. Sau đó, tiến trình tiếp tục được thử nghiệm để xác nhận s.

Bản phác thảo còn được mở rộng trong phương pháp hướng đối tượng để bảo vệ các dữ liệu thụ động. Do đó, B.Hansen đã đề xướng kiểu dữ liệu trừu tượng mới (1973), mà nó đã đảm nhận nhiệm vụ: bảo vệ tất cả các thủ tục có thể nhạy bén từ bên ngoài (gọi là phương pháp kiểu dữ liệu trừu tượng), bảo vệ một cách tự động các dữ liệu và các biến cục bộ (được sử dụng) nhờ sự đồng bộ sơ khai và đảm bảo sự loại trừ lẫn nhau của các tiến trình. Các kiểu này được gọi là dò kiểm tra. Cấu trúc bộ dò kiểm tra (*monitor*) được dẫn ra với sơ đồ cú pháp như sau:

```
MONITOR   <name>
                    (*các dữ liệu địa phương và các thủ tục*)
   PUBLIC
                    (*các thủ tục của giao diện theo bên ngoài *)
   BEGIN            (*lấy các giá trị ban đầu*)
    …
```

```
ENDMONITOR
```

Việc gọi các thủ tuc được bảo vệ và các thủ tục nhạy cảm xảy ra nhờ việc thử trước các tên thủ tục.

Thí dụ về sản sinh và tiêu dùng:

```
MONITOR    Buffer;
TYPE              Item = ARRAY [1..M] of BYTE;
VAR               RingBuf: ARRAY [1..N] of Item;
                  InSlot, (*vị trí trống thứ 1*)
                  OutSlot, (*vị trí điền đầy thứ 1*)
                  used: INTEGER;
PUBLIC PROCEDURE  putInBuffer(item:Item);
                BEGIN  … END  putInBuffer;
                PROCEDURE getFromBuffer(VAR item: Item);
                BEGIN…END getFromBuffer;
BEGIN
      used :=0; InSlot:=1; OutSlot :=1;
ENDMONITOR;
```

Nhu cầu của bộ dò kiểm tra được trình bày như trên hình 2.27

Hình 2.27---------------

Tuy nhiên, việc thảo mã vẫn chưa hoàn tất. Trong ví dụ ở hình 2.24, chúng ta nhận thấy rằng, điều cần thiết là phải dự đoán những điều kiện chờ đợi phụ (ở đây: để điều khiển dòng chảy) ở trong khoảng tới hạn. Ở bộ dò kiểm tra, điều đó đã được thấy trước nhờ các biến chuyên dụng có dạng CONDITION. Những tác vụ riêng lẻ đã tiếp nhận các biến này, thì chúng cũng thuộc hai tác vụ sau đây:

+ signal(s)

Nếu một tín hiệu s được gởi đi trong khoảng dò kiểm tra tới tất cả, thì các tác vụ nói trên phải chờ đợi một cách độc lập.

+ wait(s)

Nếu như chờ đợi quá lâu, cho đến khi một tín hiệu s được gởi đi, rồi sau đó, thực hiện các lệnh kế tiếp.

Do đó, chúng ta mở rộng việc dò kiểm tra tới các biến điều kiện:

VAR free, belegt: CONDITION

Những biến điều kiện này rất cần thiết cho việc điều khiển dòng chảy. Cả hai tác vụ truy cập được bộ dò kiểm tra bảo vệ có dạng như sau:

```
PROCEDURE putInBuffer(item:Item);
BEGIN
    IF used = N THEN wait(free) END;
    RingBuf[InSlot]:= item;
```

```
        used := used +1;
        InSlot := (InSlot +1) MOD   N;
        signal (belegt);
    END putInBuffer;

    PROCEDURE    getFromBuffer(VAR ITEM: Item);
    BEGIN
         IF used =  0 THEN wait(belegt) END;
         item:= RingBuf[OutSlot]
        used := used -1;
        OutSlot := (OutSlot +1) MOD   N;
        signal (free);
    END getFromBuffer;
```

Việc thực thi này giúp chúng ta hồi tưởng lại đoạn chương trình nguyên sơ được trình bày ở trong hình 2.23. Trong sự khác biệt với đoạn chương trình này, bây giờ chúng ta đã phòng tránh được sự đọc chọn bằng việc điều khiển dòng chảy với sự chuyển đổi tương hỗ, do đó, không có các điều kiện chạy đua (*race condition*) có thể được thiết lập.

Việc sử dụng cấu trúc dò kiểm tra ở trong hệ điều hành thì chẳng có vấn đề gì. Những điểm phê phán thực chất đó là sự thu xếp không cần thiết và không mong đợi của các tiến trình hệ điều hành. Với các cấu trúc cơ bản như các tiến trình và các tín hiệu, người ta có thể tránh được những vấn đề đã nêu ở lập trình thích ứng.

Một vấn đề thực tiễn tiếp theo đó là: Các tác vụ cờ hiệu có thể được mô tả như những đơn thể bổ sung của thư viện người sử dụng (thí dụ với các lệnh máy đặc biệt) và có thể được sử dụng một cách dễ dàng trong hầu hết các ngôn ngữ lập trình. Đối lập với cái đó, việc biên dịch phải được mở rộng đối với khoảng tới hạn và đối với việc dò kiểm tra.

Thí dụ về ngôn ngữ lập trình JAVA:

Một ấn bản quan trọng của ngôn ngữ lập trình C và C++ đó là ngôn ngữ lập trình JAVA. Những đơn thể ứng dụng (*applets*) của chúng thì rất tiện lợi: chúng có thể tác dụng như những đơn thể phần mềm của một hệ thống nhỏ, cũng như chúng có thể thực hiện các chức năng chuyên dụng như những bộ phận của trang WEB (World- Wide-Web). Thuộc về những cái đó, mã nguồn (bao gồm mã ký tự, mã biên dịch) được nạp trên mạng máy tính, và ở đó, nó được thông dịch bởi trình thông dịch JAVA: nó có thể là một bộ phận của trình đọc lướt (*browser*) hay của hệ điều hành (như OS/2).

Mã JAVA đem lại những kết quả giống nhau trên các hệ máy tính khác nhau, nhờ thế, các tiến trình threads có thể được làm việc một cách song song. Sự phối hợp của nhiều tiến trình trọng lượng nhẹ đối với việc truy cập trên một đối tượng

chung thì đạt được với sự trợ giúp của từ khoá synchronized. Để làm đồng bộ tại một khoảng tới hạn, cú pháp được viết:

Synchronized (<expression>) <statement>

Một xâu tiến trình chỉ có thể thực hiện lệnh (hay dãy lệnh) biểu thị ở <statement>, nếu trước đó, đối tượng (*object*) hay trường (*fild*) biểu thị ở <expression> đã được ghi vào danh sách và đã được bảo vệ nhờ cờ hiệu. Nếu đối tượng đã được ghi vào danh sách rồi, thì do đó nó sẽ bị hãm, cho tới khi xâu tiến trình khác lại rời khỏi khoảng tới hạn. Trong khoảng động bộ, với mọi phương pháp, người ta có thể chờ đợi thêm tín hiệu lệnh wait(), mà nó được gởi đi bởi lệnh notify().

2.3.6 Những vướng mắc

Ngay cả trong một hệ điều hành đã làm đồng bộ hoàn mỹ, thì vẫn có thể xuất hiện các tình trạng mà ở trong đó, các tiến trình bị ngăn cản hay bị hãm, do đó, các chương trình có thể không thực hiện. Tại sao vậy?

Chúng ta quan sát tình trạng sau đây: Tiến trình 1 thu lượm các dữ liệu, sắp xếp chúng thành một file và chép chúng lên một máy in. Bây giờ tiến trình thứ 2 bắt đầu, nó cần phải phô bày file đã được thu lượm đó. Hai phương tiện điều hành là máy in và file phải được bảo vệ bởi sự chuyển đổi tương hỗ, do đó, đã tránh được một sự mất trật tự khi viết và đọc. Bấy giờ sự việc xảy ra như sau: Sau khi thu lượm dữ liệu, tiến trình 1 ngăn file lại, thể hiện một khoảng đánh dấu và nó muốn in khoảng đánh dấu thứ 1 này trước khi thể hiện khoảng đánh dấu thứ 2. Trong khi đó, tiến trình thứ 2 đã ngăn cản máy in, nó đã in nhan đề và thỉnh cầu file. Vì file này sẽ được sử dụng ngay, cho nên tiến trình thứ 2 sẽ nằm ngủ khi có cờ hiệu. Nhưng tiến trình thứ 1 cũng nằm ngủ ở thời điểm thử nghiệm để nhằm nhận được máy in. Bấy giờ, cả hai tiến trình cũng nằm ngủ mãi mãi: trong hệ thống các tiến trình, người ta nói tại đó đã xuất hiện một khoá tử (*deadlock*).

Chúng ta cũng có một tình huống khác: Một tiến trình che phủ một phương tiện điều hành A (chẳng hạn một file) và muốn có một phương tiện điều hành B khác (thí dụ một máy in), mà về phía nó, phương tiện điều hành này bị che phủ bởi một tiến trình khác. Ngược lại, tiến trình khác này cũng muốn có phương tiện điều hành A.

Từ thí dụ này, một vài tính chất được E.G.Coffman phỏng đoán (1971): đó là những điều kiện cần thiết sau đây:

(1). Điều kiện ngăn chặn lẫn nhau (*mutual exclusion*): Mỗi một phương tiện điều hành được chứa đựng thì hoặc là che phủ lẫn nhau hoặc là tự do.

(2). Điều kiện ngăn chặn và chờ đợi(*hold and wait*): Các tiến trình đã sẵn sàng sẽ ngăn chặn phương tiện điều hành, muốn chặn thêm các phương tiện điều hành tiếp theo và chờ đợi cho đến khi chúng được tự do. Và, các phương tiện điều hành cần thiết được yêu cầu không phải chỉ một lần.

(3). Điều kiện không có ưu tiên trước(*no preemption*): Những phương tiện điều hành có thể không đơn giản được các tiến trình chiếm trở lại, chúng cũng giống

như việc định thời có ưu tiên trước của CPU, các phương tiện điều hành này phải được các tiến trình tự đưa trả lại một cách rõ ràng.

(4). Điều kiện chờ đợi luân chuyển (*circulare wait*): phải tồn tại một chuỗi khép kín gồm hai hay nhiều tiến trình, mà tại đó, mỗi một phương tiện điều hành sẽ có tiếp theo, thì đã bị một tiến trình che phủ, và do đó các tiến trình chưa được tự do.

Một tình huống như vậy không chỉ xảy ra tại các phương tiện điều hành như máy in, máy quét và các thiết bị ngoại vi, mà cả tại các khối logic với các thứ như các cờ hiệu…Nếu chúng ta khảo sát thí dụ sản sinh và tiêu dùng ở trong hình 2.24: người ta thiết lập biến belegt =0 khi thực hiện tác vụ và biến free =0 khi thoát khỏi việc thực hiện tác vụ.

Bây giờ chúng ta có tình huống:

Sản sinh	Tiêu dùng
…	…
wait (free)	wait(belegt)
…	…
send (belegt)	send(free)

Cả hai đều chờ đợi sự trả lại tự do của cờ hiệu: nó chỉ được trả lại tự do khi tiến trình khác đã chiếm chỗ và khi mong muốn của chúng được thoả mãn.

Đối với cách mô tả khoá tử thì có 4 chiến lược hiệu nghiệm sau đây:

+ Chiến lược bỏ qua khoá tử
+ Chiến lược nhận dạng và loại bỏ khoá tử
+ Chiến lược phòng tránh khoá tử
+ Chiến lược làm mất khả năng khoá tử

Sau đây chúng ta sẽ lần lượt trình bày 4 chiến lược kể trên.

Chiến lược bỏ qua khoá tử (*bypass deaddlock*):

Đầu tiên, chiến lược này chỉ ra: việc bỏ qua khoá tử thì cũng giống như việc tháo gỡ khoá tử, nghĩa là tuyệt đối không tiếp nhận. Điều đó có ý nghĩa đối với nhà toán học. Nếu chúng ta quan tâm đến một sự kiện, tại những hệ thống lớn bao giờ cũng xảy ra nhiều khả năng biến động, do đó chúng ta thấy, những biến động ấy bao giờ cũng ảnh hưởng đến sự sống và làm cho sự sống khó khăn hơn. Chúng ta bắt đầu bởi các vấn đề về phần cứng (thí dụ các chỗ tiếp xúc xấu, các tổn thất của chip IC, các dây cáp bị ọp ẹp, độ ổn định của lưới điện kém …) cho tới các vấn đề của phần mềm (thí dụ các lỗi ở trong hệ điều hành, các phiên bản biên dịch quá mới, không rõ cấu hình của các phần mềm ứng dụng, việc nhập sai dữ liệu…), do đó chúng có thể dẫn tới các trường hợp: hai tiến trình bị mắc kẹt, trong đời sống thường nhật có thể đó là sự phân cấp thiếu ý nghĩa, cũng giống như việc chất chứa đầy tràn của bảng tiến trình hay việc vượt qua giới hạn lớn nhất của việc các file cùng mở đồng thời. Nhà vật lý phải khảo sát cái đó trước khi người ta loại bỏ những hiệu quả xấu, kéo dài cho tới khi nó không còn rối loạn nhiều nữa. Những người quản lý việc phát triển máy tính (EDV) nhìn thấy cần phòng ngừa các khoá

tử trước khi làm chệch hướng những nhiệm vụ quan trọng. Do đó, ở trong hệ điều hành Unix, không có những biện pháp đặc biệt đối với các khoá tử.

Điều đó thì độc lập với sự thường xuyên xuất hiện của các khoá tử. Tuy nhiên, tốt hơn, các khoá tử ở trong hệ điều hành phải được chạy chữa một cách tự động, để việc phòng tránh làm cho những tai biến giảm xuống mức thấp nhất.

Chiến lược nhận dạng và loại bỏ khoá tử:

Một chiến lược để chạy chữa các khoá tử thì bao gồm việc kiểm tra các chức năng của hệ điều hành và việc tiến hành một thuật toán cho sự xuất hiện những triệu chứng nghi ngờ. Thuật toán này nghiên cứu và phát hiện một cách hệ thống tình trạng tồn tại của một khoá tử. Những triệu chừng nghi ngờ có thể là:

+ Khi rất nhiều tiến trình chờ đợi và bộ vi xử lý không làm việc (idle task). Khái niệm "rất nhiều" được hiểu chính xác đó là giới hạn trên, mà người ta cũng gọi là giá trị ngưỡng cửa.

+ Khi tối thiểu có hai tiến trình phải chờ đợi quá lâu các phương tiện điều hành, ở đây khái niệm "quá lâu" cũng được hiểu là giới hạn trên.

Tình hình diễn biến của các tiến trình và các phương tiện điều hành được minh hoạ bằng một sơ đồ như trong hình 2.28 ở dưới đây.

Hình 2.28--------------------

Ở đây, mỗi tiến trình được chỉ bằng một vòng tròn (P_1,P_2…), các phương tiện điều hành hay cá tài nguyên được chỉ bằng các hình vuông (B_1,B_2…), còn sự phân bổ được chỉ bằng mũi tên. Hướng mũi tên chỉ tử P-> B có ý nghĩa rằng một tiến trình muốn có một phương tiện điều hành. Hướng mũi tên chỉ từ P<- B có ý nói rằng một tiến trình có chắc chắn một phương tiện điều hành.

Bốn điều kiện của khoá tử nói ở trên có ý nghĩa đối với sơ đồ cấp phát phương tiện điều hành. Trong thí dụ ở trên, chúng ta nhận thấy có một số khoá tử đã tạo thành một vòng tuần hoàn, cụ thể là các nút P_4,B_3,P_5 và B_4. Điều đó có thể được kiểm tra lại ở các sơ đồ lớn hơn nhờ các thuật toán đồ thị thích hợp. R. C. Holt đã giới thiệu một thuật toán (1872) cho trường hợp có nhiều phương tiện điều hành như loại vừa lấy thí dụ ở trên.

A.V. Hebermann cũng đưa ra một thuật toán khác (1969) đơn giản hơn, được dùng để kiểm tra lại như đã nói. Nếu gọi A_s là số lượng đạt được của các phương tiện điều hành ứng với kiểu biến s để tạo điều kiện cho các dòng chảy khoá tử tự do. Chúng ta phải xét xem, liệu một tiến trình có tồn tại, khi nó đi ra khỏi phương tiện điều hành. Nếu một tiến trình như thế được tìm thấy, thì nó được nhận ra rằng, nó phải trả lại phương tiện điều hành đã chiếm sau khi nó kết thúc và số lượng A_s của các phương tiện điều hành gia tăng. Khi đó, tiến trình này được đánh dấu và được giữ chặt (tức là không có khoá tử nào tồn tại), hoặc là, còn tồn tại nhiều tiến trình chưa được đánh dấu (tức là chúng bị hãm một cách tương hỗ với các yêu cầu của chúng). Tất nhiên, chúng ta sẽ xuất phát từ đó: Tại các phương tiện điều hành

chưa được che phủ hoàn toàn, mỗi tiến trình đã có đủ phương tiện điều hành để sử dụng, và do đó, chúng có thể hoàn toàn làm việc một mình. Chúng ta có thể mô phỏng thuật toán một cách chính xác bởi các ký hiệu sau:

E_s: số lượng các phương tiện điều hành tồn tại ở kiểu s; thí dụ $E_2 = 5$ tức là có 5 phương tiện điều hành tồn tại ở kiểu s=2 (chẳng hạn kiểu máy in);

B_{ks}: số lượng các phương tiện điều hành tồn tại ở kiểu s mà tiến trình thứ k đã chiếm;

C_{ks}: số lượng các phương tiện điều hành tồn tại ở kiểu s mà tiến trình thứ k yêu cầu thêm.

Ở đây, chúng ta nhận được tổng của các ma trận hàng B_{ks} chính là tổng của các phương tiện điều hành đã bị che phủ $\sum B_{ks}$. Do đó, số lượng A_s của các phương tiện điều hành còn tự do của kiểu s là:

$$A_s = E_s - \sum_k B_{ks} \qquad (2.19)$$

Các bước riêng lẻ của thuật toán diễn ra như sau:

(0) Xem xét tất cả các tiến trình chưa được đánh dấu;

(1) Tìm một tiến trình k chưa được đánh dấu, mà tại đó, đối với tất cả các phương tiện điều hành s phải thoả mãn điều kiện: $A_s >= C_{ks}$;

(2) Trường hợp một tiến trình như (1) tồn tại, người ta đánh dấu nó và thiết lập:

$$A_s := A_s + B_{ks}$$

(3) Nếu có một tiến trình như (2) tồn tại, thì STOP. Nếu không người ta thực hiện mới bước (1) và (2).

Ở mã giả, thì điều đó được thực hiện như sau:

```
FOR k:=1 TO N unmark Process(k) END; d:= 0;               (*bước 0*)
REPEAT
    k:= satifiedProcess();                                 (*bước 1*)
    IF k # 0 THEN d:=d+1; markProcess(k);                  (*bước 2*)
       FOR s :=1 TO M DO As := A[s] + B[k,s] END;
    END;
UNTIL k=0;                                                 (*bước 3*)
IF d < N THEN Error('Deadlock exists')  END;
```

Nếu thuật toán dẫn tới sự kết thúc mà không có sự thông báo lỗi, do đó, điều cần được khẳng định rằng, các tình trạng không có lối thoát không thể tồn tại được; thật vậy, như đã nói, sau khi thử trắc nghiệm, các tình trạng này được phát triển. Ngược lại, khi chúng ta nhận được thông báo lỗi, thì do đó, điều được chỉ ra rằng, một khoá tử được tồn tại trên thực tế cũng giống như việc chúng ta đã đánh dấu một tiến trình ở trong một dãy tuần tự kết hợp nào đó. Cở sở của sự độc lập thực tế đối với dãy tuần tự đánh dấu là sự gia tăng liên tục của các phương tiện điều hành.

Thí dụ về các khoá tử:

Hình 2.29 ----------

Ở đây, B_{ij} và C_{ij} là hai ma trận như đã nói ở trên. Chúng có các biến cột là A: dãi băng, B: các thiết bị vẽ, C: các máy in, D: các ổ CD ROM và các biến hàng là các tiến trình P_1, $P_2 \ldots P_5$. Khi ta có E= (6 3 4 2) thì hai ma trận B_{ij} và C_{ij} được xác định như hình 2.29 ở trên.

Nếu sự phân bổ tổng cộng là (5 3 2 2) thì do đó còn lại A = (1 0 2 0). Trong tất cả các tiến trình chỉ có P_4 được phủ thêm (0 0 1 0), công việc của nó cũng được thực hiện hoàn tất và tổng các phương tiện điều hành giảm xuống còn (1 1 1 1). Với điều đó, chúng ta nhận được A = (2 1 1 1). Bây giờ, các tiến trình P_1 và P_5 có thể kết thúc công việc, khi đó, chúng ta nhận được A = (5 1 3 2) và cuối cùng, với P_2 và P_3 thì A = (6 3 4 2). Người ta lưu ý rằng, dãy tuần tự P_4, P_1 , P_5 ,P_2 , P_3 thì không bị di chuyển cưỡng bức, nó cũng có thể theo tuần tự P_4, P_5 , P_1 ,P_2 , P_3.

Bây giờ, chúng ta loại bỏ khoá tử xuất hiện như thế nào? Từ thuật toán nói ở trên, chúng ta nhìn thấy dễ dàng những phương tiện điều hành và tiến trình nào sẽ bị kẹp chặt. Sau đây chúng ta nêu ra một cách thực hiện việc loại bỏ khoá tử như sau:

- *Bẻ gãy các tiến trình*:

Đây là một phương pháp đơn giản để mong muốn hay để bẻ gãy một tiến trình bị kẹp chặt và sau đó chúng tự sắp xếp trở lại. Những tiến trình nào có thể được bẻ gãy một cách có mục đích thì phải được người xác định. Thuật toán nói trên cũng dẫn tới một sự trợ giúp, nó cung cấp một kết quả định thời cho các tiến trình được đánh dấu.

- *Đặt tiến trình trở lại*:

Ở trong các hệ thống ngân hàng dữ liệu, từ lý do bảo vệ, có hầu hết những thời điểm đều đặn, ở tại đó, trạng thái chung của hệ thống được lưu trữ. Bây giờ nếu xuất hiện một khoá tử, thì do đó, người ta có thể đặt một trong nhiều tiến trình trở lại trên một trạng thái, mà đáng lẽ chúng bị bẽ gãy một cách không thương tiếc. Khi đó, chúng chưa che phủ các phương tiện điều hành và khi đó việc thực thi tiến trình cũng bị trì hoãn thật lâu cho tới khi các phương tiện điều hành được giải phóng khỏi các tiến trình bị khoá tử.

- *Chiếm lấy các phương tiện điều hành:*

Trong những trường hợp như vậy, thì có thể, một tiến trình muốn chiếm lấy một phương tiện điều hành, trong khi phương tiện điều hành đã bị các tiến trình khác sử dụng. Cho đến khi nó sắp được trả tự do, thì tiến trình này có thể được thỉnh cầu trở lại và được thiết lập trở lại ở vị trí chương trình bị bẻ gãy.

Chiến lược này thì không phải luôn luôn có khả năng, thí dụ như ở việc diễn tả một file. Hay nói một cách khác, việc sử dụng giao thức này để giải quyết vấn đề sẽ đem lại một tiêu phí lớn.

Yếu điểm của các phương pháp vừa trình bày thì đã rõ: Nếu chúng ta bẻ gãy tiến trình một cách đơn giản, thì do đó dẫn tới cả các công việc tiếp theo bị tuyệt vọng. Ngoài ra, có rất nhiều vấn đề cần được bổ khuyết: tiến trình bị bẻ gãy cũng

như tiến trình được đặt trở lại có thể đã được viết các dữ liệu hay đã gởi đi các thông tin, mà chúng không được đón nhận trở lại và dưới các tình trạng này có thể dẫn tới các files bất kiên định. Việc lựa chọn tiến trình phù hợp cần thiết được uốn nắn theo tiêu chuẩn của việc tính toán các hư hỏng được thu hẹp nhất.

Chiến lược phòng tránh khoá tử:

Một trong các chiến lược đơn giản để tránh khỏi khoá tử được trình bày như sau. Khi xuất hiện một lỗi: phương tiện điều hành che phủ tiến trình ready(sẵn sàng), thì lỗi này sẽ chỉ cho người sử dụng và do đó yêu cầu người sử dụng tự có phản ứng đúng về lỗi đó. Ở các máy tính đơn (personalcomputer), với hầu hết mọi trường hợp, người sử dụng có thể nhìn được bao quát, anh ta chiếm toàn bộ quyền truy cập và do đó, anh ta thực hiện tránh khỏi các khoá tử một cách dễ dàng.

Ở hệ thống đa người sử dụng (như mạng LAN, mạng WAN…), thì chiến lược này không thể sử dụng được. Khi đó, các kỷ thuật viên giàu kinh nghiệm sẽ dùng nó với một ý tưởng: cần phân biệt các trạng thái bị đe doạ bởi khoá tử và các trạng thái bảo vệ, và phải luôn luôn chỉ thu nạp những tiến trình che phủ mà hệ thống tạo ra nó thành các trạng thái bảo vệ. Trạng thái bảo vệ là cái gì? Trạng thái bảo vệ là trạng thái mà tại đó luôn luôn chỉ có một con đường (tức là chỉ có một chiến lược định thời) cho các Job được tồn tại và kết thúc một cách suôn sẻ. Ngược lại, trạng thái không bảo vệ là trạng thái mà nó không bị cưỡng bức dẫn tới nhưng có thể dẫn tới các khoá tử. Đến đây, chúng ta đã nghiên cứu một thuật toán để kiểm tra các trạng thái, nó được gọi là thuật toán nhận biết các khoá tử. Về điều đó, chúng ta khảo sát những yêu cầu về phương tiện điều hành của các tiến trình ở thuật toán này với tư cách là những yêu cầu tối đa mà tất cả chúng có thể xuất hiện một lần, nhưng không bắt buộc.

Sự phỏng đoán các khoá tử của thuật toán trắc nghiệm này có ý nghĩa rằng, một khoá tử có thể xuất hiện nhưng không phải bắt buộc phải xuất hiện. Nếu có xuất hiện, thì các phương tiện điều hành còn lại sẽ được các tiến trình khác yêu cầu một cách riêng lẻ kế tiếp nhau và được đưa trả lại. Trong sự khác biệt với việc che phủ ở giai đoạn trước đó, sự che phủ sẽ được đưa trở lại và được thực hiện một lần, khi đó không có khoá tử xuất hiện. Tiếp theo thì đã rõ: Trạng thái bị khoá tử đe doạ, và tất nhiên, khi đó phương tiện điều hành từ chối tiến trình. Trong thí dụ ở hình 2.29, tiến trình P_2 có thể nhận lại máy in, mà in hệ thống không bị khoá tử đe doạ và hệ thống cũng không từ bỏ trạng thái bảo vệ. Nếu chúng ta trao cho các tiến trình P_5 máy in cuối cùng, thì do đó, tiến trình P_4 không thể làm việc một cách đầy đủ được nữa: Hệ thống bị khoá tử đe doạ và quá độ chuyển thành trạng thái không bảo vệ. Do đó, khoá tử cuối cùng (được mô tả ở trên) đã được tránh khỏi. Bây giờ, hệ thống không bị khoá chặt, mà chỉ bị đe doạ: Thí dụ, trước khi kết thúc, tiến trình P_1 trả lại máy in của nó (đã chiếm), thì do đó, không có mối nguy hiểm nào đe doạ nữa.

Thuật toán trắc nghiệm được E.W. Dijkstra đưa ra đầu tiên (1965) và được gọi là *thuật toán chủ nhà băng*, vì nó phản ảnh quan điểm của chủ nhà băng: anh ta

thử nghiệm với nguồn vốn (tức phương tiện điều hành) hạn hẹp, mà vẫn thoả mãn sự mong muốn vay tín dụng của khách hàng. Chiến lược của chủ nhà băng là: kiểm tra mỗi một mong muốn vay của khách hàng, rồi quan tâm đến những mong muốn của khách hàng khác trong khuôn khổ cho vay tín dụng đầy đủ và nhiều nhất, do đó, với việc giao trả kết nối, chủ nhà băng đã đáp ứng và thoả mãn được mong muốn tiếp nối của khách hàng.

Việc sử dụng thuật toán này để tránh khỏi khoá tử, cần thiết lưu ý những vấn đề sau đây:

+ Ở các ứng dụng, số lượng các phương tiện điều hành cần thiết lớn nhất là chưa rõ.

+ Số lượng các tiến trình ở trong hệ thống là năng động và luôn thay đổi.

+ Số lượng các phương tiện điều hành thì cũng thay đổi tương tự.

+ Thuật toán là lưu trữ cấp tốc và diễn biến cấp tốc.

Chiến lược làm mất khả năng khoá tử:

Ý tưởng cơ bản của giao thức chạy chữa khoá tử này là ở chỗ: chỉ một trong bốn điều kiện nói ở phần đầu mục 2.3.7 thì chưa đủ, mà cả bốn mới là điều kiện đủ và cần thiết đối với một khoá tử. Ở mỗi điều kiện thì việc làm mất khả năng các khoá tử sẽ thực hiện khác nhau.

(1). *Đối với điều kiện ngăn hãm lẫn nhau (matual exclusion*):

Cuộc tranh giành đối với một phương tiện điều hành được tạo ra, để người ta chuyển giao nó cho một tiến trình có quyền chiếm dụng và để người ta biến đổi các cuộc thăm dò đối với phương tiện điều hành này, cũng như biến đổi việc dịch vụ tới một tiến trình.

Lấy máy in làm thí dụ: Một tiến trình đặc biệt (gọi là *printer demon*) nhận được máy in một cách chắc chắn; còn các tiến trình khác đang ở trong một hàng đợi để chuyển giao các dữ liệu cần in cho tiến trình này. Tiến trình *printer demon* đại diện cho các tiến trình đang được thăm dò cuộn các hàng đợi lại (*spooling*), thật lâu cho tới khi công việc in xong xuôi và sau đó nằm ngủ khi hàng đợi trống. Do đó, đầu tiên khoá tử được tránh khỏi.

Chiến lược này thì không có vấn đề gì: Đối với một tiến trình này thì không phải tất cả các phương tiện điều hành đều có điều kiện, còn đối với một tiến trình khác, thì nó được chiếm dụng trước (vì đối với gọi tương hỗ, thì các cờ hiệu luôn luôn sẵn sàng). Ở trong thí dụ này, sự chuyển đổi tương hỗ của máy in được chuyển biến thông qua bộ đệm máy in (*printer buffer*) ở trong bộ nhớ chính, mà nó bày ra một không gian hạn hẹp tương tự như một phương tiện điều hành. Nếu chúng ta có hai tiến trình, mà nhu cầu bộ đệm của chúng vượt quá một nữa không gian có thể được sử dụng, thì do đó, ở đây, vấn đề tương tự lại đặt ra như trước đây. Mỗi tiến trình viết dữ liệu của nó vào bộ đệm cho tới khi hết không gian. Vì công việc in chỉ sau khi được thông báo đầy đủ mới được treo vào hàng đợi và nếu hàng đợi nhiệm vụ in trống thì cả hai tiến trình đều phải chờ đợi, cho tới khi không

gian bộ đệm được sử dụng trở lại để kết thúc sự chuyển giao, tức là chúng bị khoá lại. Với lý do này, các hệ thống cuộn hàng đợi(spoolingsystem) cũng thường được lập trình: chỉ file gốc được sử dụng để in và trước đó không cần làm bản sao.

Tóm lại, sự chuyển đổi của việc sắp xếp trực tiếp trên hệ thống client- server là có điều kiện sử dụng được.

(2). Đối với điều kiện giữ lại và chờ đợi (*holt and wait*):

Nếu các tiến trình đều yêu cầu các tài nguyên bổ sung, thì do đó, chúng ta phải ngăn ngừa các khoá tử. Một khả năng xảy ra được điều đó là phải thực hiện: một tiến trình chỉ được cung cấp một phương tiện điều hành đơn lẻ. Nhưng, điều không thể chấp nhận: tức là một tiến trình sẽ bị cấm, nếu tiến trình này vừa đọc các dữ liệu của một file và vừa in chúng.

Một khả năng khác cũng được nói tới: các phương tiện điều hành mà cần thiết cho một Job, thì khi bắt đầu Job, nó được một lần yêu cầu tới. Tuy nhiên, điều này đòi hỏi một tiêu phí cho việc lập trình bổ sung thêm, tức là tiêu phí này không phải dành cho Job cũng chẳng phải dành cho phương tiện điều hành, mà đặc biệt chỉ cho người lập trình. Nhược điểm của phương pháp này là ở chỗ: tất cả các phương tiện điều hành được sử dụng để che phủ suốt thời gian diễn biến Job, và hoàn toàn không được có một Job khác chiếm dụng. Thí dụ, một chương trình được tính toán cho 5 tiếng đồng hồ và sau đó thực hiện một ấn tượng ngắn, chương trình này để ngăn ngừa kéo dại 5 tiếng đồng hồ cho mỗi ấn tượng, mà mỗi ấn tượng là một tính chất không thể chấp nhận ở trong phạm vi của nhiều người sử dụng (trong mạng máy tính) và nó cũng là một khả năng quá tải của phương tiện điều hành.

Một phương án thú vị khác cũng được chỉ ra: ở việc che phủ các phương tiện điều hành bổ sung thì tất cả các phương tiện điều hành ở trạng thái sẵn sàng được đưa trở lại một lần và sau đó, tại một sự nghi vấn thì tất cả các phương tiện điều hành được yêu cầu để sử dụng cho giai đoạn kế tiếp. Nhờ tiếp, nó nâng cao được tiêu chí che phủ và ngăn chặn được một khoá tử.

(3).*Đối với điều kiện không có ưu tiên trước* (*no preemption*):

Một sự chiếm sớm một phương tiện điều hành thì không thể đơn giản, vì một sự thay đổi nào đó có thể sẵn sàng được thực hiện, thí dụ: khi viết các dữ liệu hay khi in vào giấy. Một sự chiếm sớm các phương tiện điệu hành như vậy nhờ việc cảnh giới theo thời gian khi chương trình đang thực thi, và do đó, chương trình này có thể xảy ra một cách tự động và thông suốt.

(4). *Đối với điều kiện chờ đợi luân chuyển (circular wait*):

Một khả năng bẻ gãy vòng chờ đợi luẩn quẩn là ở chỗ: yêu cầu ngăn chặn ngay bước đầu tiên của vòng đợi này, do đó, cần phải định nghĩa khái niệm bậc tương đối của yêu cầu về phương tiện điều hành, tức là phải dẫn tới việc tuyến tính hoá các yêu cầu về phương tiện điều hành:

+ Tất cả các phương tiện điều hành phải được đánh số và chỉ cho phép một tiến trình chiếm dụng một phương tiện điều hành có số hiệu cao hơn cả. Nhờ đó, khi có một tiến trình nào đó chờ đợi một phương tiện điều hành có số hiệu nhỏ hơn, mà phương tiện điều hành này đã bị một tiến trình khác chiếm được sớm hơn: một cách nguyên tắc, điều đó không thể xảy ra.

Bằng sự chấp nhận này, bây giờ chúng ta khảo sát một vòng chờ đợi luẩn quẩn. Nếu như trong sơ đồ các phương tiện điều hành, chúng ta mô tả mỗi số hiệu ứng với một ô chữ nhật biểu thị một phương tiện điều hành, chúng ta mô tả mỗi số hiệu ứng với một ô chữ nhật biểu thị một phương tiện điều hành, mà nó diễn biến trong vòng chờ. Ở giới hạn trên thì nó không thể đạt được từ một phương tiện điều hành được giữ chặt này tới một phương tiện được yêu cầu khác, nó chỉ đạt được khi có số hiệu cao. Bất kỳ khi nào, vòng chờ này cũng được khoá lại và sau đó, những ký hiệu số hiệu nhỏ nhất được tiếp diễn như là một yêu cầu tới ký hiệu số hiệu lớn hơn. Nhưng điều đó thì không phù hợp với giả thiết ban đầu. Nhờ vậy, không thể có một vòng chờ luẩn quẩn nào được tạo thành.

Yếu điểm của phương pháp tuyến tính hoá này là ở chỗ: Không có một bậc nào được tìm thấy một cách dễ dàng cho các ứng dụng. Nếu một dãy tuần tự các yêu cầu mong muốn được xác định, thì không thể có một tiến trình phải yêu cầu các nguồn tài nguyên một lần nữa, điều đó cho thấy khả năng chịu tải của tài nguyên đã bị suy giảm.

+ Một khả năng tiếp theo để đem lại một sự phân bổ các phương tiện điều hành, mà với sự phân bổ này, các phương tiện điều hành được sắp xếp theo cấu trúc bậc và bị một trong các tiến trình chiếm dụng. Các tiến trình này thoả mãn các yêu cầu của người sử dụng và lưu ý tới sự tự do của các khoá tử tại những chỗ mà nhiệm vụ của người sử dụng được giới hạn theo thời gian.

Tổng các phương tiện điều hành (thí dụ như tất cả các Files) có thể được phân phối và có thể được chuyển giao việc quản lý chúng cho các tiến trình con. Vì điều đó cũng có giá trị đối với các tiến trình con, do đó xuất hiện một cấu trúc bậc hình cây để biểu diễn sự che phủ.

Thí dụ về quản lý files:

Hình 2.30--------------------

Trên hình 2.30, mỗi record được sở hữu bởi một tiến trình riêng lẻ. Các nhiệm vụ (như viết- đọc) được dẫn rộng ra từ rễ cây tới các tiến trình con. Cấu trúc cây đảm bảo rằng, dãy tuần tự để gia tăng các nhiệm vụ con thì bằng nhau khắp mọi nơi. Nhờ thế, các trạng thái của các record tồn tại một cách bền vững.

2.4. Trao đổi thông tin giữa tiến trình (*process- communication*)

Một phương tiện quan trọng để kết hợp các hoạt động của nhiều tiến trình trong một hệ điều hành và để đạt được một mục đích của công việc chung, đó là

sự trao đổi thông tin. Ở các hệ điều hành cổ điển thì điều đó bị lảng quên. Ở trong hệ điều hành Unix thì khác, có những chương trình được coi là hòn đá tảng, mà với sự tương tác kết hợp của chúng có thể đem lại một chương trình mới. Ở các ấn bản cũ của Unix, các phương tiện cho cái đó bị giới hạn. Trước hết sự cấp thiết đã dẫn tới: cần kết hợp các công việc trong mạng máy tính (tức là bỏ qua giới hạn một máy đơn) và cần tạo điều kiện trao đổi với các tiến trình bên ngoài như việc dịch vụ hệ điều hành ở trên từng máy tính riêng lẻ.

Do đó trong phần này, những suy nghĩ cơ bản được giới thiệu, ví dụ, chúng kết nối như thế nào giữa mạng máy tính với hệ thống đơn vi xử lý cũng như với hệ thống đa vi xử lý. Những quan điểm về hệ điều hành có tác động mạnh mẽ đối với mạng máy tính sẽ được trình bày một cách kỹ càng trong chương 6.

Việc trao đổi thông tin giữa các tiến trình đã dẫn tới những suy nghĩ chắc chắn sẽ được trình bày lần lượt sau đây.

2.4.1. Trao đổi thông tin

Ở sự trao đổi thông tin, người phân biệt ba kiểu kết nối khác nhau:

+ Kiểu kết nối *unicast*: là kiểu kết nối từ một tiến trình riêng lẻ này tới một tiến trình riêng lẻ khác (tức là diểm nối điểm);

+ Kiểu kết nối *multicast*: là kiểu kết nối từ một tiến trình này tới một nhóm các tiến trình khác;

+ Kiểu kết nối *broadcast*: là kiểu kết nối từ một tiến trình tới nhiều tiến trình riêng lẻ.

Các kiểu kết nối này được mô tả như ở hình 2.31 dưới đây, với các ký hiệu P_1,P_2,…, P_k và P_n là các tiến trình riêng lẻ.

Hình 2.31------------------

Ở đây, mỗi kiểu kết nối được thực thi nhờ kiểu kết nối khác. Thí dụ, người ta có thể thực hiện kiểu kết nối *broadcast* hay kiểu *multicast* nhờ kiểu kết nối *unicast* hay nhờ một kiểu kết nối mà ở đó, một danh sách các người nhận được gởi đi theo địa chỉ xác định và danh sách này được người nhận sử dụng để tạo lập sự kết nối tiếp theo. Ngược lại, người ta có thể gởi một thông tin tới tất cả các người nhận, mà ở đây chỉ cần một người nhận được trao thông tin, còn những người nhận khác thì nhìn theo hay lờ đi.

Vì thế, người ta có thể trao đổi thông tin một cách khác nhau. Theo cách truyền thống, sự trao đổi thông tin được đặc trưng giống như sự kết nối điện thoại, vì thế nó được gọi là sự trao đổi thông tin được đặc trưng giống như sự kết nối điện thoại, vì thế nó được gọi là sự trao đổi thông tin kết nối định hướng và được chỉ trên hình 2.32 sau đây.

Hình 2.32------------

Sự tiêu phí cho việc tạo lập và cho việc giữ gìn một sự kết nối (*canal*) thì thực ra rất lớn. Từ lý do này, một loại trao đổi thông tin khác hiện đại hơn: trao đổi thông tin không có kết nối. Trong sự khác biệt với việc tạo lập kết nối đồng bộ, ở đây, người gởi và người nhận phải biểu lộ sự sẵn sàng của họ đối với sự kết nối, do đó, thông tin được gởi đi một cách không đồng bộ:

send (Adresse, Message) / receive (Adresse, Message)

Vì thế, bấy giờ nó thì không còn dám chắc rằng, thông tin có thật đã tới chưa (vì sự truyền đạt và sự tiếp nhận bị quấy nhiễu), do đó, mỗi thông tin phải được xác nhận một cách rõ ràng. Nếu trong một khoảng thời gian nào đó không có thông tin truyền tới xuất hiện, thì do đó, người gởi phải gởi phục hồi ngay thông tin của anh ta, cho tới khi anh ta nhận được biên lai đã nhận được. Sau đó, anh ta tiếp tục gởi thông tin khác. Nếu thông tin biên nhận bị đánh mất, thì do đó, người nhận sẽ nhận được bản sao của thông tin tương tự. Để có thể phân biệt thông tin mới với thông tin cũ, trong trường hợp này, các thông tin được thực hiện bởi các số hiệu liên tục. Một thông tin có thể được thiết lập từ cấu trúc cơ bản sau đây:

```
TYPE  tMessage = RECORD
                    ReceiveAdresse: STRING;
                    SendAdresse: STRING;
                    Information Typ: tMsgTyp;
                    SequentialNummer: INTEGER;
                    Length: CARDINAL;
                    Data : POINTER TO  tBlock;
                 END;
```

Điều đó được người ta biểu thị là gói thông tin (*message header*) và nó phù hợp với việc chuyển tải, giống như việc chuyển một bức thư. Ở đây, một số lượng tối thiểu các thông tin chuyển tới chỉ cho ta thấy: thực ra, các thông tin thực giữa các tiến trình được phóng đi rất nhiều lần. Bởi vậy, cần phải có sự đồng bộ giữa người gởi và người nhận, cũng giống như việc thông dịch các trường thông tin. Đồng thời người ta cũng lưu ý tới các vấn đề quan trọng khác: kiểu dữ liệu (thí dụ kiểu tMsgTyp), dãy tuần tự của các trường thông tin và sự thông dịch các số liệu (nó thường dẫn tới kiểu INTEGER với đầu tiên là nhưng Byte có giá trị cao nhất hay những Byte có giá trị thấp nhất). Với lý do này, việc trao đổi thông tin giữa các máy tính (khác kiểu nhau), thì đầu tiên mô tả dữ liệu và sau đó là các dữ liệu được gởi đi.

Việc quản lý bên ngoài các thông tin có chiều dài khác nhau thì không đơn giản. Các gói thông tin luôn luôn có chiều dài giống nhau, nhưng chiều dài của các dữ liệu khi truyền đạt trong trường Data thì cho phép không cần cố định. Tại bên người nhận, đầu tiên gói thông tin được đọc và tiếp đến phải cấp phát không gian lưu trữ cho các khối dữ liệu vừa được trao đổi.

Việc gởi đi cũng như việc tiếp nhận được thực hiện bằng phương pháp khác nhau. Về phía gởi đi, thì hoặc là, người gởi bị làm trì hoãn thật lâu, cho đến khi một thông báo trả lời được xảy ra (gọi là gởi đồng bộ hay gởi có kìm hãm), hoặc

là, hệ thống thực hiện đệm lại thông tin (giống như bản sao hay bản trực tiếp), hay có thể hệ thống quan tâm đến sự giao chuyển hợp cảnh và cuối cùng hệ thống để cho người gởi tiếp tục thực hiện việc chuyển gởi của mình (tức là gởi không có hãm hay gởi không đồng bộ). Tuy nhiên, ở việc chuyển gởi không có hãm, thì một thông tin báo lỗi được chuyển trở lại, nếu bộ đệm hệ thống ngoài bị tràn, hoặc trong trường hợp này người gởi phải được làm chậm lại.

Đối với việc nhận hay việc đọc các thông tin (cũng vậy) cũng có ấn bản có hãm và không có hãm: hoặc là người nhận bị làm trễ cho tới khi một thông tin bày ra, hoặc là người nhận đón nhận một thông báo trở lại thích hợp với việc đọc không có hãm.

Định vị (*addressing*):

Việc xác định địa chỉ người nhận thì rất khác nhau. Ở kiểu kết nối *unicast*, địa chỉ người nhận bao gồm số hiệu của tiến trình. Nếu một tiến trình chờ đợi một máy tính khác, do đó, số hiệu hay tên của máy tính khác sẽ thêm vào, có thể đó là máy chủ của công ty, của địa phương hay của quốc gia…Do đó địa chỉ của người nhận được biểu diễn bằng một chuỗi ký tự của các tên, mà sự tách chia giữa chúng được biểu thị bằng một dấu chấm:

Address = Process-ID**.**ComputerName**.**Company**.**City**.**Country

Thí dụ: 4743CNTT**.**Dng-Uni**.**Vnn**.**Vn

Đối với kết nối kiểu *multicast*, người sử dụng chỉ một danh sách các tiến trình cũng như các máy tính, mà danh sách này có thể được gởi tới cùng với thông tin.

Tuy nhiên, một kiểu định vị địa chỉ như vậy thì chưa thuận tiện lắm, khi đôi bạn trao đổi thông tin chưa hề quen biết nhau trực tiếp và chưa hề biết tên danh định (process-ID) của nhau. Thí dụ, một chương trình muốn gởi các dữ liệu cho một máy in để in lên trang giấy. Trường hợp không biết tên danh định của máy in, nhưng có biết chắc máy in đang tồn tại, thì cách tốt hơn: định nghĩa tên logic của người nhận là máy in ở trong hệ thống, mà một sự sắp xếp như thế được thay đổi tới một tên danh định hiện hành ID theo mỗi trạng thái của hệ thống và được cố định ở trong một bảng sắp xếp của nhân hệ thống ở phía người nhận. Tất cả việc gọi hệ thống tới *máy in* được dẫn tới địa chỉ xác định theo yêu cầu của *tiến trình gởi*. Người ta có thể mở rộng nguyên tắc của việc định vị *tên logic và tên phi vật lý* này trong phạm vi trao đổi thông tin trên mạng máy tính. Vì thông tin đầy đủ về các tiến trình, về các máy tính và về các công sở sẽ không được cố định trên mỗi máy tính của mạng, do đó, có những máy tính đặc biệt (gọi là *name server*) chuyên dùng cho việc sắp xếp địa chỉ ở xa, gọi là địa chỉ dừng hay địa chỉ phân giải.

Những điều đã nói không chỉ có giá trị đối với việc kết nối kiểu *unicast* , mà cả đối với kiểu *multicast*, và với các kiểu kết nối này, nhiều tiến trình và cả các máy tính có thể được thu tóm lại thành các nhóm và có thể được định vị dưới tên nhóm của chúng.

Một kiểu định vị đặc biệt khác nữa, đó là việc định vị nhờ địa chỉ tên gọi. Ở đây, người ta dẫn ra một kiểu tên gọi logic, mà nó phụ thuộc vào những điều kiện logic, thí dụ, nó phụ thuộc vào kiểu máy tính hay kiểu CPU, vào bộ nhớ RAM, vào các thiết bị ngoại vi…

Người nhận sẽ định vị việc chọn đọc theo giao thức IF, nếu kết quả là WAHR, do đó, anh ta cảm nhận một cách nhạy bén, nếu kết quả là FALL, thì anh ta (người nhận) không có câu hỏi nào và khi đó thông tin bị thất lạc. Điều đó có thể làm tốt hơn, đó là phải phân bổ công việc năng động và phải sử dụng thiết bị ngoại vi còn nhàn rỗi.

Các hộp thoại (*mailboxs*):

Ở việc trao đổi thông tin không đồng bộ, thông tin được nạp trung gian nhờ một bộ đệm (*buffer*), vì người nhận chưa đọc ngay những thông tin này. Những bộ đệm thông tin này có thể bị nhầm lẫn tên, do đó, người ta sử dụng một tiến trình chưa được biết tới để chuyển các thông tin cho nó. Ngay cả việc quản lý hệ thống cũng bị thay đổi, đáng lẽ phải thực hiện việc sắp xếp tên logic các tiến trình tới các tiến trình vật lý, thì lại thực hiện việc sắp xếp tên logic các bộ đệm tới địa chỉ vật lý các bộ đệm.

Kiểu một bộ đệm thông tin như vậy có thể được cấu tạo như là một hàng đợi, mà ở đó, các thông tin được treo vào và được người nhận đọc. Nếu chúng ta thấy trước một hàng đợi tiếp theo đối với các tiến trình nhận của một nhóm tiến trình, thì chúng ta xem xét: trường hợp không có thông tin được sử dụng, ví như một hàng đợi đối với các tiến trình gởi; trường hợp dòng thông tin đã đầy, do đó, chúng ta nhận được hộp thoại với cấu trúc như sau:

```
TYPE  Mailbox = RECORD
                      SendQueue:       tList;
                      ReceiveQueue:    tList;
                      MsgQueue:        tList;
                      MsgNumb:   INTEGER;
                END;
```

Việc xâm nhập trên các hàng đợi (với các hàm Treo vào (Msg) và Bứt ra (Msg)) phải được bảo vệ nhờ các tác vụ cờ hiệu, do đó, một cờ hiệu phải được nhìn thấy trên hộp thoại. Ta thấy biến MsgNumber dịch vụ như là một bộ đếm để điều khiển dòng chảy trong khoảng tới hạn giữa hai hàm Treovào(Msg) và Bứt ra(Msg). Nếu với biến MsgNumber =N mà dung lượng của hộp thoại đạt lớn nhất, thì do đó, tiến trình gởi được treo vào hàng đợi và được nằm ngủ; ngược lại, khi biến MsgNumber =0 thì đó là trường hợp đối với người nhận.

Nếu tại MsgNumber =N thông tin được đọc, do đó, người nhậ còn phải thực hiện thêm hàm wakeup(SendQueue) để đánh thức *cờ hiệu gởi* tồn tại. Khi MsgNumber =0 thì cờ hiệu phải đánh thức người nhận đang nằm chờ.

Đối với sự kết hợp các tiến trình gởi và các tiến trình nhận phải đạt được mục đích, để tiếp nhận các thủ tục treo vào hay bứt ra ở trong tờ khai của hộp thoại và cho phép lối vào hộp thoại chỉ qua các tác vụ đã được kiểm tra. Cấu trúc dữ liệu chung là kiểu dữ liệu trừu tượng. Ở trong ngôn ngữ lập trình hướng đối tượng, tờ khai PUBLIC cho phép che dấu cấu trúc dữ liệu và che dấu hộp thoại, do vậy, chúng ta nhận được một cấu trúc dữ liệu giống như cấu trúc bộ kiểm tra.

2.4.2. Trao đổi thông tin giữa các tiến trình với các kênh ở Unix

Ở các ấn bản đầu tiên của Unix, sự trao đổi thông tin giữa các tiến trình đã sử dụng một kênh đệm chuyên dụng. Trong thí dụ ở đầu chương 2, chúng ta thấy rằng, nhiều chương trình của Unix (ở trường hợp này đó là các tiến trình) có thể tương tác với nhau qua một mệnh thức:

Programm 1| Programm 2|…| Programm N

Ở đây, dấu thẳng đứng biểu thị một cơ chế chuyển giao các dữ liệu của một chương trình tới một chương trình kế cạnh, mà cơ chế này được thực hiện nhờ một kênh. Một kênh trao đổi thông tin như vậy hoạt động theo nguyên tắc sau đây:

Với hàm gọi hệ thống pipe(), một kênh thông tin được mở, mà kênh đó, tại các tác vụ đọc và viết, dòng lệnh dưới đây có thể được truy cập một cách bình thường:

read(fileId, buffer) / write(fileId, buffer).

Đối với mỗi kiểu kết nối trao đổi thông tin, chương trình chính (ở đây là vỏ hệ thống của người sử dụng) mở một kênh (pipe) riêng lẻ và truyền tiếp kênh này với văn cảnh tiến trình tại tiến trình con; các tiến trình con của bên gởi chỉ sử dụng hàm fileId để viết, còn tiến trình bên nhận sử dụng hàm này chỉ để đọc. Vì sự nhận biết các files là những con số (số hiệu) ở trong các bảng, mà chúng được truyền cho tiến trình bởi hàm gọi hệ thống fork(), cho nên, sự trao đổi thông tin giữa tiến trình cha và tiến trình con ở trong hệ điều hành Unix được tạo thành nhờ các kênh (pipes). Ở hình 2.33 chỉ ra hai tiến trình: chúng trao đổi thông tin với nhau nhờ nặc danh pipe, tức là chúng đã truyền pipe cho tiến trình cha bằng ngữ cảnh tiến trình. Pipe được chứa đựng bằng số lượng các bước của ngữ cảnh tiến trình (các đường nét đậm).

Hình 2.33------------

Ở hệ điều hành Unix, pipe thì không định hướng; đối với việc trao đổi thông tin giữa hai tiến trình thì hai pipe được sử dụng.

Bình thường thì tiến trình gởi chỉ bị hãm, nếu pipe đầy; tiến trình đọc chỉ xảy ra, nếu pipe trống. Ở loại tiến trình không có hãm, tiến trình đọc được nhận trở lại mức không, nếu không có thông tin nào bày ra; còn không, số lượng các Byte để đọc được lưu trữ trong biến đệm.

Sự trao đổi thông tin giữa các tiến trình bất kỳ và bỏ qua giới hạn các kiểu máy tính thì khả năng đầu tiên của các ấn bản mới hơn của Unix, thì chúng được bỏ qua đối với một pipe, ví như bỏ qua cấu trúc socket.

2.4.3. Trao đổi thông tin giữa các tiến trình với các kênh ở Windows NT

Ở trong Windows NT cũng có trao đổi thông tin giữa các tiến trình với các kênh. Chúng đạt được nhờ gọi hệ thống CreadPipe(). Sau đó, người ta có thể nhận được hàm WriteFile() và ReadFile() nhờ các tác vụ đọc viết. Với một hàm gọi CloseHandle() thì chúng được kết nối.

Vì ở đây, việc chiếm dụng các kênh này (pipes) thì không tồn tại bằng một nhận biết bên ngoài, do đó, giống như thế, nó bị giới hạn bởi các nhóm tiến trình cha và tiến trình con. Tuy nhiên, theo đó, một sự trao đổi thông tin tương hỗ luôn luôn có khả năng, nếu pipe là hai hướng.

Thật vậy, để thay đổi tình trạng có hãm và không có hãm, trong Windows NT có cấu trúc *named pipes* giống cấu trúc *socket* đã nói.

2.4.4. Đồng bộ tiến trình bằng trao đổi thông tin

Ở phần trước chúng ta thấy, các cờ hiệu là kiểu động bộ nguyên thuỷ sơ đẳng và hầu như các bộ kiểm tra cũng không được sử dụng. Cả hai vấn đề này đều cần thiết trong phạm vi đơn vi xử lý cũng như đa vi xử lý, nhưng mà không ở ngay trong một mạng máy tính. Do đó, chúng ta sử dụng kiểu đồng bộ nguyên thuỷ, mà kiểu đồng bộ này cũng được dùng trong các hệ thống phân bổ. Trước hết, điều đó có thể đạt được nhờ việc trao đổi thông tin với những thông tin rất ngắn và nhờ sự chờ đợi việc gởi thông tin. Có thể nói một cách chính xác, việc nhận được một thông tin bao gồm hai phần: phần chờ đợi (tức đồng bộ) thông tin và phần đọc thông tin. Ở các thông tin dài, sự đồng bộ vẫn ở mức không. Các hàm send(Message) và receive(Message) thì giống hệt với các tác vụ send(Signal) và waitFor(Signal) đối với các thông tin như thế. Chúng ta xuất phát từ hai trường hợp này, rằng các thông tin truyền đi được lưu trữ trung gian và trước khi đọc không được biến đi mất. Một sự đồng bộ thuần khiết có thể được mở rộng một cách dễ dàng tới việc trao đổi các thông tin và ngược lại, một sự trao đổi thông tin cơ bản được dùng để làm đồng bộ thuần khiết. Tiếp theo, chúng ta khảo sát sự đồng bộ bằng tín hiệu.

Sự đồng bộ bằng tín hiệu:

Một trong các cơ sở quan trọng để nhận được thông tin là sự xuất hiện các biến cố. Đó là sự báo động các lỗi xuất hiện trong hệ thống (thí dụ lỗi dữ liệu, các địa chỉ lưu trữ không rõ ràng…), các cách sửa chữa ngoại lệ (thí dụ chia cho zero…) và các tín hiệu của các tiến trình khác. Do đó, tiến trình biên nhận có thể hoặc là chờ đợi đồng bộ trên biến cố cho đến khi có xuất hiện, hoặc là chỉ thiết lập việc xử

lý các thông báo biến cố hay bỏ lại việc xử lý thông tin xuất hiện không đồng bộ cho thủ tục được yêu cầu xử lý.

Trường hợp thứ hai thì đặc biệt hữu ích ở việc sửa chữa ngoại lệ. Bằng các tiến trình riêng biệt, người ta có thể sửa chữa được các ngoại lệ như chia cho zero, tràn ngăn xếp, tổn thương giới hạn trường…Để có số lượng nhiều các sửa chữa ngoại lệ khác nhau, một giao diện riêng lẻ thuộc hệ điều hành được định nghĩa, được khởi xướng và được biên soạn bởi các chương trình bên ngoài đối với mỗi đơn thể.

Các tín hiệu ở hệ điều hành Unix:

Ở trong hệ điều hành Unix có một hệ thống các tín hiệu, mà với nó, sự tồn tại của một biến cố có thể được thông báo cho một tiến trình. Thuộc về điều đó, những tín hiệu (POSIX) liệt kê dưới đây được định nghĩa.

SIG**ABRT**	*abort process*:	yêucầu bẻ gãy tiến trình ngay lậptức
SIG**TERM**	*terminate:*	mong muốn kết thúc tiến trình
SIG**QUIT**	*core dump:*	yêu cầu bẻ gãy tiến trình khỏi bộ nhớ
SIG**FPE**	*floating point error*	
SIG**ALRM**	*alarm-Signal:*	diễn biến đồng hồ chỉ thời gian
SIG**HUP**	*hang up:*	kết nối điện thoại được thiết đặt
SIG**KILL**	*kill- Signal:*	bẻ gãy tiến trình ở từng trường hợp
SIG**ILL**	*illegal instruction*:	lệnh máy không tồn tại
SIG**PIPE**	*pipe-data:*	không có người nhận tồn tại đối với dữ liệu pipe
SIG**SEGV**	*segmentation violation*:	địa chỉ nhớ không thể sử dụng
SIG**INT**	*interrupt-Signal:*	tín hiệu ngắt
SIG**USR1**		dùng cho những ứng dụng đặc biệt 1
SIG**USR2**		dùng cho những ứng dụng đặc biệt 2

Theo tiêu chuẩn, hệ điều hành Unix có cả thảy 16 tín hiệu POSIX, mà chúng được thể hiện qua bề rộng từ của các thanh ghi ở trong khối điều khiển tiến trình dài 16 bit. Ở Unix, dịch vụ hệ điều hành send(Signal) được xuất hiện do việc gởi tín hiệu SIFKILL để bẻ gãy tiến trình, dịch vụ này còn có tên kill(). Việc sử dụng các tín hiệu nói trên được suy cho những công việc xác định, để ghi nhận như thế nào đó phạm vi ảnh hưởng của chúng. Nhưng với các trường hợp này, một thủ tục (tự định nghĩa) có thể được kết nối trong khoảng các gọi hệ thống sigaction(), mà thủ tục này được bắt đầu chạy một khi xuất hiện tín hiệu ở tiến trình ngắt phần mềm, mà bên gởi cũng như bên nhận, các tín hiệu bất kỳ được sử dụng để trao đổi thông tin.

Với các tín hiệu này, người ta còn phân biệt thêm, liệu một tiến trình phải chờ đợi một tín hiệu xác định từ một biến cố hay chỉ chờ đợi một tín hiệu nói chung. Thuộc về điều đó, người ta thấy rằng, các hệ điều hành khác nhau có các dịch vụ

khác nhau, mà tại đó với hàm logic AND và OR, người ta thiết đặt các điều kiện cho sự hoạt động của tiến trình nhờ các biến cố hay các tín hiệu nói ở trên.

Thí dụ về chờ đợi biến cố:

Các biến cố (như nhấp mouse hai lần, ấn ký tự chuẩn ASCII, chọn menu, điều khiển cửa sổ…) đòi hỏi nhiều phản ứng khác nhau. Một cách hữu hiệu, người ta chỉ có thể trông chờ vào các chương trình tương tác, ví dụ chương trình điều hành các hệ thống cửa sổ, chương trình điều hành một trong các biến cố khi truy nhập…Vì thế, chúng được đạt tới một gọi hệ thống, gọi là chờ đợi đa biến cố (*wait-multi-event*). Chẳng hạn, nếu chúng chờ đời để kích mouse hay ấn nút ký tự ASCII, thì khi đó, một mặt nạ phù hợp được kiến lập. Thật vậy, một mặt nạ AND sẽ có điều kiện, nếu như một liên hiệp các nút bấm SHIFT và nhấp hai lần mouse được đáp ứng.

16 tín hiệu ở trong Unix có thể được sử dụng để tạo lập một sự đồng bộ tiến trình. Với sự trợ giúp của các hàm send(Signal) và wait(Signal), các tác vụ cờ hiệu được thực thi một cách dễ dàng. Ở loại máy tính MODULA-2, đoạn chương trình dưới đây cho thấy sự đồng bộ tiến trình được thiết lập trên một máy tính.

```
Type   Semaphor = POINTER to tSemaphor
         tSemaphor = RECORD
                              besetz: BOOLEAN;
                           free : SIGNAL;
                        END;
PROCEDURE    P(VAR  S: Semaphor);
BEGIN
    IF  S^.besetz  THEN  waitFor(S^.free)  END;
    S^.besetz := TRUE;
END  P;

PROCEDURE    V(VAR  S: Semaphor);
BEGIN
    S^.besetz := TRUE;
    send(S^.free)
END  V;
```

Tuy nhiên, một đơn thể cờ hiệu phải chiếm lấy một ưu tiên cao hơn các tiến trình còn lại để các tác vụ P() và V() trở thành các nhân tử.

Điều đó có thể đạt được: nếu ở máy tính MODULA-2, nó có thể thực hiện được là nhờ sự chuyển giao ưu tiên ở bản khai đơn thể; còn ở một ngôn ngữ lập trình khác thì nó thực hiện được nhờ gọi hệ thống setPrio(high) đặt trực tiếp sau từ khoá BEGIN hay nhờ gọi hệ thống setPrio(low) đặt trực tiếp trước từ khoá END.

Cờ hiệu được sinh ra và khởi xướng nhờ việc gọi thủ tục như sau:

```
PROCEDURE  createSemaphor(VAR  S: Semaphor);
BEGIN
         ALLOCATE(S, TSIZE(tSemaphor));
         S^.besetz:=FALSE
         initignal(S^.free);
END createSemaphor.
```

Hệ thống cờ hiệu được tạo ra như vừa nói vẫn có nhược điểm: nó chỉ hoạt động giữa các tiến trình trên các bộ vi xử lý giống nhau, mà ở đó, sự gia tăng ưu tiên trước có thể loại bỏ một ngắt. Như vậy, ở tại hệ thống đa vi xử lý cũng như tại các hệ thống nhiều máy tính, có những cơ cấu khác nhau được sử dụng để đạt một sự đồng bộ.

Đồng bộ nhờ kiểu kết nối broadcast:

Để có một bản phác thảo quan trọng hơn cho việc đồng bộ các thông tin trao đổi, câu hỏi đặt ra là: liệu một thông tin có đến người nhận hay không. Nếu một vài người nhận được, còn một số khác không nhận được, thì điều đó thật khó khăn cho người gởi để quản lý sự trao đổi thông tin được đúng đắn và để đảm bảo một cơ sở dữ liệu phù hợp thống nhất cho các thông tin của mình. Để thu hẹp các chi phí phụ và để đảm bảo đúng mức sự kết thúc hoàn toàn khi truy cập tại ngân hàng dữ liệu, thì điều phải làm là: người ta phải mô tả sự trao đổi thông tin kiểu broadcast như một hoạt động nhân tử (*atomic action*). Theo F.Cristian (1985), một kiểu kết nối atomic broadcast được định nghĩa bởi các yêu cầu sau đây:

+ Thời gian truyền đạt thông tin là có hạn;
+ Tất cả mọi người hoặc nhận được thông tin, hoặc không nhận được;
+ Dãy tuần tự các thông tin ở tại tất cả mọi người nhận là như nhau.

Dãy tuần tự như nhau của thông tin tại tất cả mọi người nhận được sử dụng như là những cơ cấu cơ bản không có hãm của một sự thể hiện dữ liệu bền vững. Nếu dãy tuần tự và nội dung của thông tin ở khắp nơi là như nhau, do đó, các trạng thái như nhau của dữ liệu cũng như của biến toàn cục và của các files được dẫn ra bởi M. Dalcin và R.Brause (1987) nhe ỏ hình 2.35.

Hình 2.35--------------------

Người ta nhận thấy rằng, dãy tuần tự của các thời gian 1,4,3 giữ chặt trạng thái của các biến toàn cục, do vậy, điều đó độc lập với số hiệu các thông tin, tức là thông tin chỉ phụ thuộc vào việc đếm của mỗi người gởi. Nếu người gởi cũng là thành viên của nhóm, thì tất nhiên, người gởi cũng chuyển thông tin cho chính nó để lưu thông tin dữ liệu ở trong nhóm và sau khi nhận được thông tin, người gởi được phép thay đổi các biến toàn cục. Nếu trước đó, anh ta làm điều này, thì cái có thể là: tại các người nhận khác, sau một thông báo thay đổi xác định, thông tin của anh ta mới chuyển tới nơi, như vậy, hiệu lực tại chỗ anh ta thì khác với hiệu lực tại

tất cả các bộ vi xử lý khác, nghĩa là: không có sự lưu thông dữ liệu. Điều đó dẫn tới sự mong muốn che phủ của các tiến trình khác nhau đối với phương tiện điều hành. Do đó, những mong muốn của các tiến trình phải được sửa chữa mọt cách mạnh mẽ ở trong dãy tuần tự, để tạo ra khắp mọi nơi trạng thái giống nhau đối với các biến che phủ.

Phương pháp trên đây được sử dụng cho dãy tuần tự liên kết chặt chẽ các thông tin, không cần đánh số hiển thị toàn bộ các thông tin và loại trừ được nhiều vấn đề khi trao đổi thông tin giữa các nhóm thay đổi cục bộ.

Đồng bộ các chương trình:

Đối với việc gởi và nhận các tín hiệu, người ta dùng phương pháp trao đổi thông tin không cần đệm. Trong trường hợp này, chương trình bên nhận được làm chậm lại cho tới khi chương trình bên gởi chuyển thông tin: khi đó, gọi nhận đồng bộ có hãm. Tức là điều đó dẫn tới một sự đồng bộ giữa người gởi và người nhận, mà nó được mong muốn trong những hệ thống tiến trình như vậy, và nó có tên gọi nỗi tiếng Rendez-vous-Concept (bản phác thảo chỗ gặp lại). Ngôn ngữ lập trình ADA có chứa đựng một ý tưởng như vậy và nó tạo điều kiện để chạy một chương trình của các tiến trình trao đổi thông tin ở tất cả các hệ thống, mà ở đó, một chương trình biên dịch ADA được tạo lập. Hình 2.36 chỉ ra một quá trình đồng bộ như vậy.

Hình 2.36----------------------

Một ý tưởng quan trọng khác đó là việc thực hiện song song các chương trình nhờ các tiến trình trao đổi thông tin. Nhiều chương trình có thể được tạo lập một cách đơn giản hơn, có thể mở rộng được và chờ đợi nhau niềm nở hơn, nếu người ta diễn đạt một trạng thái giống như một nhiệm vụ, mà nó được hoàn tất bởi những thực thể chuyên dụng (tách biệt và nhỏ) dùng để trao đổi thông tin với nhau.

Nếu chúng ta phân tách chương trình thành những đoạn mã ngắn, thí dụ mã *threads*, do đó, những đoạn ngắn chương trình này cũng cần dùng một sự trao đổi thông tin hiệu quả để thực hiện nhiệm vụ chung của chương trình tổng thể. Kiểu trao đổi thông tin trong nội bộ một chương trình thì thật đơn giản đối với người lập trình để tránh các lỗi và để tạo ra khả năng lập trình hiệu suất.

Với mục đích này, C.A.R. More (1978) đã thiết kế một kiểu ngôn ngữ lập trình cho các tiến trình trao đổi thông tin tuần tự (*communicating sequential process*: CSP). Ở các tiến trình CSP có các cấu trúc ngôn ngữ:

receive! data1	cho thủ tục	send(receive, data1)
send?data2	cho thủ tục	receive(send, data2)

Đôi bạn trao đổi thông tin người gởi và người nhận phải chờ đợi việc trao đổi thông tin lẫn nhau. Điều này phù hợp với kiểu đồng bộ Rendez-vous-Concept (cùng nhau hẹn chờ), tức là phù hợp với kiểu trao đổi thông tin không có đệm thêm. Sau khi đồng bộ, việc sắp xếp dữ liệu data2:=data1 được thực hiện, khi đó

data1 và data2 phải cùng kiểu dữ liệu, thí dụ kiểu INTEGER hay kiểu REAL. Người gởi và người nhận có những cái tên được cắt nghĩa rõ ràng ở trong chương trình.

Việc thực hiện song song các chương trình được tác dụng với sự trợ giúp của các dấu hiệu sau đây đối với việc lựa chọn các lệnh:

$$B_1 \rightarrow S_1$$
$$[]B_2 \rightarrow S_2$$
$$\ldots$$
$$[]B_n \rightarrow S_n$$

Tất cả các điều kiện Boolean $B_1 \rightarrow B_n$ sẽ được kiểm tra: Nếu có một trong các điều kiện B_i được thoả mãn, thì do đó, một lệnh tương ứng S_i sẽ được thực hiện. Nếu có nhiều điều kiện thoả mãn, thì khi đó chỉ được phép chọn một trong số các điều kiện đó. Một dấu ngoặc vuông biểu thị dãy tuần tự tác dụng xuyên qua việc chọn lệnh cho tới khi không còn điều kiện nào thoả mãn. Sau đó, tiếp tục việc thực hiện các lệnh của chương trình tiếp theo khác.

Cấu trúc này được E.W.Dijktra phát minh để cảnh giới các lệnh, nó được đảm bảo một phần là nhờ ngôn ngữ lập trình song song OCCAM (1988). Tiến trình trọng lượng nhẹ để trao đổi thông tin có thể bao gồm vài lệnh trên một dòng. Thí dụ, kiểu *sản sinh- tác dụng* ở trong OCCAM được coi như một công việc gởi tới một tiến trình đệm, mà người sử dụng nhận lại từ đó.

Hình 2.37------------------

Bộ đệm là kiểu bộ đệm hình xuyến với 10 phần tử trong một tiến trình BufferProc được bao bọc bởi mã:

```
CHAN OF  item   producer,   consumer:
INT in, out:
SEQ
    in := 0
    out :=0
   WHILE TRUE
     ALT
          IF (in < out +10) AND  (producer? buffer(in REM 10))
              in :=in +1
           IF (out < in )AND (cosumer?more)
               out := out+1
```

Theo đó, chỉ số thích hợp của bộ đệm xuyến được chuyển cho việc vào –ra trong khoảng tác vụ REM (remmainder). Ký hiệu ALT để chỉ dãy tuần tự bất kỳ, còn ký hiệu SEQ để chỉ một dãy tuần tự xác định, mà nó xác định việc thực hiện của các dòng lệnh. Vì chỉ có việc nhập vào các điều kiện mới mới được kiểm tra, do đó, một tín hiệu bổ sung *more* của người sử dụng thì rất cần thiết để gọi thông tin tiếp từ bộ đệm.

2.4.5 Trao đổi thông tin ẩn và hiện

Trong thí dụ kiểu sản sinh- sử dụng ở mục 2.3.5, các dữ liệu của một tiến trình được chuyển tới một tiến trình khác nhờ khoảng nhớ cơ bản của bộ nhớ, tiến trình này sẽ xử lý các dữ liệu vừa chuyển tới. Sự chuyển giao thông tin này gọi là trao đổi thông tin ẩn, nếu như sử dụng putInBuffer(item); còn được gọi là trao đổi thông tin hiển thị sử dụng send(consumer, item) và receive(procedure, item).

Theo hình 2.38, người ta rút ra những nhận xét sau đây:

+ Lệnh send(consumer, item) để chuyển item tới một bộ đệm hệ thống, mà nó được làm đầy nhờ một trong hai tác vụ P() và V(). Nếu bộ đệm đầy, tiến trình bị làm chậm lại cho tới khi không gian cho item tồn tại trở lại.

+ Lệnh receive(procedure, item) để đọc một item từ bộ đệm và được bảo vệ nhờ các tác vụ P() và V(). Nếu không có item nào được sử dụng, thì tiến trình sẽ bị làm chậm cho tới khi một item xuất hiện.

Dạng trao đổi thông tin kiểu *sản sinh- sử dụng* có ưu điểm: Cơ cấu đồng bộ của kiểu này có thể được thực thi với cơ cấu đồng bộ của sự trao đổi thông tin và hoạt động bỏ qua giới hạn các kiểu máy tính.

Chúng ta đã tiết kiệm được số tác vụ cờ hiệu, mà ở đây, chúng ta giả thiết mục đích xác định thích hợp của các thủ tục trao đổi thông tin send(Msg) và receive(Msg). Cụ thể, điều này cũng được thực thi trên hệ thống đơn vi xử lý nhờ các tác vụ cờ hiệu cục bộ. Trong trường hợp này, phạm vi bộ đệm là một hộp thoại (mainbox), mà ở đó thông tin được treo vào.

Nói chung, hầu hết các cơ cấu trao đổi thông tin hoạt động giữa các tiến trình với sự trợ giúp của các vùng nhớ, chúng được trình bày tốt hơn nhờ sự trao đổi thông tin hiển. Tiêu phí sẽ giảm đi đáng kể, nếu phần mềm được tạo lập chạy được không chỉ trên hệ thống đơn vi xử lý, mà cả trong hệ phân bổ. Ở đây, người ta có thể chuyển đổi một cách dễ dàng từ sự trao đổi thông tin giữa các tiến trình này với sự trao đổi thông tin giữa các tiến trình khác, vì ở cùng một giao diện, việc thực thi các chức năng send() và receive() được sử dụng như là một thư viện.

2.5 Các bài tập của chương 2

2.5.1.Các bài tập về các trạng thái của tiến trình

Bài tập 2.1 Về các dạng điều hành (*operating-typs*)

Bạn hãy kể một vài dạng điều hành của một hệ điều hành (*operating-system*) mà bạn nhận thức được qua chương này.

Bài tập 2.2 Về các tiến trình

a) Bạn hãy giải thích sự khác nhau thực chất giữa các khái niệm chương trình, tiến trình và thread (xâu)

b) Trong hệ điều hành Unix của một máy tính, một khối điều khiển tiến trình (PCB) và các cấu trúc người sử dụng được xem xét như thế nào? Bạn hãy kiểm tra các tệp tin /include/sys/proc.h và /include/sys/user.h, đồng thời bạn hãy biểu thị các sự điền vào trong đó theo nhận thức của mình.

Bài tập 2.3 Về các tiến trình

a). Một tiến trình dẫn qua những trạng thái tiến trình nào?

b). Điều kiện chờ đợi và biến cố là gì?

c). Bạn hãy thay đổi những quá độ trạng thái ở sơ đồ trạng thái của hình 2.2 bằng một sơ đồ khác tương tự. Bạn hãy lý giải các sự thay đổi của bạn.

Bài tập 2.4. Về các tiến trình ở hệ điều hành Unix

Ở hệ điều hành Unix, với ngôn ngữ C, một gọi hệ thống ExecuteProgramm(prg) được thực hiện như thế nào? Khi đó như là một thủ tục với sự trợ giúp của các gọi hệ thống fork() và waitpid().

Bài tập 2.5. Về các tiến trình threads

Bạn hãy thực hiện hai thủ tục như là các tiến trình trọng lượng nhẹ khi chúng được chuyển giao sự điều khiển một cách đồng thời.

2.5.2. Các bài tập về định thời tiến trình

Bài tập 2.6. Về định thời

a). Sự khác nhau giữa thời gian thực hiện và thời gian của một Job là gì?

b). Bạn hãy lý giải sự khác nhau giữa giải thuật định thời kiểu phản hồi đa mức (multilevel- feedback) và giải thuật định thời kiểu tiền cảnh hậu cảnh (foreground-background).

Bài tập 2.7. Về định thời

Có 5 xấp nhiệm vụ cùng tới một máy tính gần như đồng thời; chúng có thời gian thực hiện phỏng chừng 10, 6,4,2 và 8 phút; theo đó, chúng có quyền ưu tiên theo thứ tự 3, 5, 2, 1 và 4; ở đây, số 5 là có quyền ưu tiên cao nhất và số 1 là có quyền ưu tiên thấp nhất. Bạn hãy cho biết thời gian thực thi trung bình cho mỗi giải thuật định thời dưới đây (bỏ qua tổn thất thời gian khi chuyển đổi một tiến trình.):

a). Kiểu định thời quay vòng Robin;

b). Kiểu định thời có ưu tiên;
c). Kiểu định thời đến trước dịch vụ trước;
d). Kiểu định thời Job ngắn nhất - trước nhất.

Ghi chú: Đối với kiểu (a), bạn thấy rằng, hệ thống sử dụng kiểu điều hành đa chương trình và mỗi nhiệm vụ nhận một phần xác định thời gian bộ vi xử lý. Đối với các kiểu (b), (c) và (d), bạn thấy đấy, các nhiệm vụ lần lượt kế nhau được thực hiện.

Bài tập 2.8. Về định thời song song

a). Nếu một chương trình được dẫn tới 40% mã tuần tự (không thể dẫn tới mã song song). Khi đó, độ tăng tốc (speedup) đạt được bao nhiêu?
b). Giả sử có một phương tiện điều hành A được tiếp tục sử dụng. Người ta có thể thay đổi sơ đồ Gantt trong hình 2.12 như thế nào để thời gian sử dụng là ít hơn?

2.5.1 Các bài tập về đồng bộ tiến trình

Bài tập 2.9.

Cái gì sẽ xảy ra, nếu ở mục 2.3.1, một tiến trình A nhận được sự điều khiển và sau bước 2 xảy ra sự đổi chiều? Có còn các khả năng tạo ra lỗi không?

Bài tập 2.10. Về sự ngăn hãm lẫn nhau

Bạn hãy diễn giải các thủ tục:
Entering_area(Process: INTEGER) và leaving_area(Process: INTEGER),
mà chúng chứa đựng giải pháp của Peterson đối với vấn đề ngăn hãm lẫn nhau. Cho cái đó, hạn hãy định nghĩa hai biến toàn cục cơ bản Interesse[1..2] và dran. Một sự khái quát có thể tồn tại trên n tiến trình không? Nếu có, thì như thế nào?

Bài tập 2.11. Về cờ hiệu

a). Các tác vụ cờ hiệu P và V được diễn đạt như thế nào, nếu s chứa đựng một số lượng các tiến trình chờ đợi?
b) Giải pháp mô tả quan hệ nhà sản xuất và người tiêu dùng được thay đổi như thế nào?
c). Người ta phải thay đổi s như thế nào, nếu có nhiều phương tiện điều hành tồn tại?

Bài tập 2.12. Về đồng bộ tiến trình

Việc đồng bộ các phương tiện điều hành theo hình 2.11 được dẫn ra như thế nào với sự trợ giúp của cờ hiệu?

Bài tập 2.13. Về một vài khái niệm

a). Sự khác nhau giữa các khái niệm về khoá tử, sự ngăn hãm và sự làm đói của các tiến trình là gì?
b). Sự khác nhau giữa hai khái niệm chờ đợi tích cực và chờ đợi thụ động là ở chỗ nào?
c). Bạn hãy dẫn ra một ví dụ thực tế về khái niệm khoá tử.

Bài tập 2.14. Về khoá tử và sự ngăn hãm

Một sinh viên s_1 có mượn một quyển sách A ở thư viện; trong quyển sách, anh ta tìm thấy một tài liệu hướng dẫn ở quyển sách B; do đó, anh ta muốn mượn quyển sách này. Quyển sách B hiện tại sinh viên s_2 đã mượn, anh này tìm thấy ở trong sách B một hướng dẫn nằm trong sách A; vì thế, anh ta thử đi mượn quyển sách A. Tình huống này cần phải thiết đặt một khoá tử,một sự ngăn hãm hay không có cả hai? Bạn hãy lý giải ý kiến của bạn.

Bài tập 2.15. Về vấn đề kiểm tra
a). Bạn hãy thực thi vấn đề đọc/ viết như là một giải pháp kiểm tra.
b). Ưu nhược điểm của giải pháp kiểm tra là gì?

Bài tập 2.16. Về giải thuật nhà băng(ngân hàng)

a). Dãy tuần tự nào trong hai dãy tuần tự là có khả năng trong ví dụ nêu ở hình 2.29 đối với giải thuật nhà băng của 5 tiến trình (P_1…P_5).
b). Giả sử tiến trình P_1 nhận được một ổ đĩa băng từ (một loại ổ đĩa mềm dùng trong bưu điện hay ngân hàng) bổ sung. Có phải khi đó hệ thống bị khoá tử đe dọa không?

Bài tập 2.17. Về bảo vệ hệ thống

a). Một hệ thống máy tính có 6 ổ đĩa và n tiến trình; khi đó mỗi tiến trình cần dùng hai ổ đĩa. Hỏi n phải bằng bao nhiêu để đảm bảo một hệ thống an toàn?
b). Nhằm để phân cấp trạng thái bảo vệ, nếu có m phương tiện điều hành và n tiến trình thì số lượng các tác vụ tỷ lệ với biểu thức $m^a n^b$. Hỏi a và b bằng bao nhiêu thì đạt yêu cầu?

2.5.4. Các bài tập về trao đổi thông tin

Bài tập 2.18.

a). Bạn hãy mô tả một cách chi tiết dòng lệnh sau đây tác động lên cái gì ở trong hệ điều hành Unix?

grep deb xyz | wc-1

b). Ở Unix, việc trao đổi thông tin bị bó hẹp bởi các thông tin trên nhóm tiến trình cha/ con. Tại sao việc thực thi gặp phải sự thu hẹp này?

c). Bạn hãy trình bày sự quá độ giữa các trạng thái tiến trình ở mục 2.1 với sự trợ giúp của các thông tin và các hộp thư. Ai gởi cho ai các thông tin này?

Bài tập 2.19. Về trao đổi thông tin kiểu đường kênh (còn gọi kiểu pip)

Bạn hãy quan sát một hệ thống tiến trình, hệ thống này chỉ trao đổi thông tin kiểu các đường kênh (pips) của hệ điều hành Unix, mà bộ đệm của chúng được cấp phát một không gian bộ nhớ nói chung và mỗi đường kênh có đúng một tiến trình gởi và một tiến trình nhận.

a). Dưới hoàn cảnh nào, một tiến trình được chen vào để đợi chờ?

b). Bạn hãy sơ đồ hoá một cơ chế thích hợp để khẳng định hay để phòng tránh các khoá tử (deadlocks) cho hệ thống.

c). Với hệ thống này có một quy tắc để lựa chọn một tiến trình như là vật hy sinh, khi một khoá tử còn tồn tại. Nếu đúng, bạn hãy thiết lập quy tắc này!