Slides 79 - 1:

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

Sleep là một lời gọi hệ thống có tác dụng tạm dừng hoạt động của tiến trình gọi nó và chờ đến khi được một tiến trình khác đánh thức.

Lời gọi hệ thống Wakeup nhận một tham số duy nhất: tiến trình sẽ được tái kích hoạt (đặt về trạng thái ready) .

Khi sử dụng Sleep và Wakeup, nếu không sử dụng cẩn thận thì sẽ xảy ra tình trạng mâu thuẫn truy xuất trong một vài tình huống đặc biệt

Để tránh những tình huống này, hệ điều hành cung cấp những cơ chế đồng bộ hóa dựa trên ý tưởng của chiến lược « SLEEP and WAKEUP » nhưng được xây dựng bao hàm cả phương tiện kiểm tra điều kiện vào miền găng giúp sử dụng an toàn. Đó là Semaphore (Truyền tin thị giác) và Message passing (Truyền thông điệp)

Slides 80 - 2:

Giờ ta sẽ đến với 3 giải pháp “SLEEP & WAKEUP”:

- Từ bỏ CPU khi chưa vào được miền găng

- Quá trình cần được đánh thức đến miền găng khi miền găng trống rỗng

- Và để đánh thức quá trình hay do quá trình thay đổi thì chúng sự hỗ trợ của hệ điều hành

Slides 81 - 3:

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

Miền găng được hỗ trợ bởi 2 cái cơ bản:

- Sleep(): Cuộc gọi hệ thống nhận trạng thái bị chặn

- WakeUp(P): Quy trình P nhận trạng thái sẵn sàng

Slides 82 - 4:

Và đây là cấu trúc sử dụng SLEEP & WAKWUP.

Chúng ta sẽ có 2 biến là busy và blocked. Busy = 1 khi miền găng đang bận và bằng 0 khi miền găng rảnh. Còn block là biến đếm số lượng tiến trình đang bị khóa

Ta có quy trình làm việc như sau:

Ban đầu ta sẽ kiểm tra biến busy có bằng 1 hay không? Nếu busy = 0 thì lúc này miền găng đang rảnh thì nó sẽ không thực hiện đoạn block = block + 1.

Thì khi đó biến busy sẽ bị gán = 1 và bắt đầu bay vào miền găng để xử lý. Và khi những tiến trình khác chạy tới sẽ kiểm tra biến busy và thấy nó bằng 1 chứng tỏ nó đang bận, mà khi tiến trình đang bận thì nó sẽ tăng biến block lên 1 và tiến vào trạng thái SLEEP.

Sau khi xử lý xong miền găng thì nó sẽ thoát ra khỏi miền găng. Và xét lại biến busy = 0 để báo miền găng đang rảnh, sau đó nó sẽ kiểm tra coi xem có tiến trình nào đang bị khóa hay không? Nếu có sẽ tiến hành đánh thức tiến trình đó dậy và giảm biến block đi 1. Và nó sẽ nhảy ra để thực thi ngoài miền găng

Slides 83 - 5:

Giờ ta có 1 ví dụ:

*Giả sử tiến trình A vào miền găng và trước khi nó rời khỏi miền găng thì tiến trình B được kích hoạt. Tiến trình B thử vào miền găng nhưng nó nhận thấy A đang ở trong đó, do vậy B tăng giá trị biến blocked và chuẩn bị gọi SLEEP để tự khoá. Tuy nhiên trước khi B có thể thực hiện SLEEP, tiến trình A lại được tái kích hoạt và ra khỏi miền găng. Khi ra khỏi miền găng A nhận thấy có một tiến trình đang chờ (blocked = 1) nên gọi WAKEUP và giảm giá trị của blocked. Khi đó tín hiệu WAKEUP sẽ lạc mất do tiến trình B chưa thật sự « ngủ » để nhận tín hiệu đánh thức! Khi tiến trình B được tiếp tục xử lý, nó mới gọi SLEEP và tự khóa vĩnh viễn.*

Do đó lý do hay xảy ra vấn đề là do:

- Việc kiểm tra tư cách vào miền găng và việc gọi SLEEP hay WAKEUP là những hành động tách biệt, có thể bị ngắt nửa chừng trong quá trình xử lý, do đó có khi tín hiệu WAKEUP gửi đến một tiến trình chưa bị khóa sẽ lạc mất

- Do kiểm tra tình trạng và loại bỏ CPU có thể bị hỏng

- Do biến khóa không được bảo vệ.

Slides 84 - 6:

Sau đây chúng ta sẽ tìm hiểu về Semaphore:

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

+ Một giá trị nguyên dương e(s) hay được gọi là semaphore s

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

+ Chỉ có hai thao tác được định nghĩa trên semaphore

\*Down

\* Up

+ Các thao tác trong hàm down và up cần thực hiện 1 cách không bị phân chia, không bị ngắt nửa chừng.

Slides 85 - 7:

Slides 86 - 8:

Và đây là cấu trúc của hàm Down và hàm Up:

Đối với hàm Down: thì e(s) sẽ bị giảm đi 1 và sau đó kiểm tra, nếu e(s) < 0 thì P sẽ bị khóa và đưa vào danh sách f(s)

Còn đối với hàm Up: thì e(s) sẽ tăng lên 1 và sau đó kiểm tra, nếu e(s) <= 0 thì chứng tỏ vẫn có tiến trình đang trong hàng chờ. Thì khi đó sẽ đưa 1 tiến trình trong hàng chờ ra và đưa vào trạng thái ready để đi tiếp. Và sau đó lại đưa tiến trình đó vào danh sách ready-list.

Từ đó ta thấy được 2 hàm Down và Up là 2 hàm trái ngược nhau, 1 hàm để khóa 1 tiến trình, 1 hàm để giải phóng 1 tiến trình ra khỏi hàng chờ.

Và ngoài ra giá trị tuyệt của e(s) còn cho ta biết được số tiến trình đang trong hàng chờ semaphore. Nếu trị tuyệt đối của e(s) mà bằng 1 thì có 1 tiến trình đang trong hàng chờ, còn bằng 2 thì có 2 tiến trình đang trong hàng chờ.

Slides 87 - 9:

Khi sử dụng Semaphore có thể giải quyết các vấn đề như: vấn đề truy xuất độc quyền hay tổ chức phối hợp giữa các tiến trình

Để thực hiện quá trình đồng bộ hóa, thì tất cả các chương trình cần phải áp dụng cùng cấu trúc chương trình sau đây với e(s) = 1.

Ta sẽ có 2 tiến trình A và B.

Đầu tiên A sẽ nhảy vào hàm Down và e(s) lúc này sẽ bị giảm đi 1 và bằng 0 khi này điều kiện bị sai và A sẽ nhảy luôn vào miền găng.

Sau khi A đi xong thì B với e(s) = 0 đi vào hàm Down, sau khi vào hàm Down thì lúc này e(s) bị giảm đi và bằng -1, và làm cho B bị khóa và tiến vào f(s).

Sau khi B bị khóa thì A thoát khỏi miền gawgn tiếp tục tiế vào hàm Up với e(s) = -1, sau khi đi vào hàm Up e(s) được cộng thêm 1 và bằng 0 đúng với điều kiện e(s) <= 0. Lúc này A sẽ đánh thức B và đặt B vào trạng thái ready và đưa B vào hàm ready-list để đợt sự cấp phát của CPU để thực thi tiếp. Sau khi A chạy xong rồi thì A sẽ tiến vào vùng không tranh chấp.

Sau khi B được sự cấp phát của CPU thì B tiếp tục nhảy vào miền găng để thực thi, sau khi B thực thi xong rồi thì B sẽ nhảy vào Up và tăng e(s) lên 1 và làm điều kiện này sai. Sau đó B sẽ thực thi ngoài miền găng. Xong khi xong hết e(s) lại được khởi tạo lên 1 lại từ đầu.

Đấy là đối với vấn đề truy xuất độc quyền, còn đối với vấn đề tổ chức phối hợp giữa các tiến trình thì sao?

Có tình huống sau: Với semaphore có thể đồng bộ hóa hoạt động của 2 tiến trình trong tình huống một tiến trình phải đợi một tiến trình khác hoàn tất thao tác nào đó mới có thể bắt đầu hay tiếp tục xử lý. Tức là tiến trình A phải chạy xong rồi thì tiến trình B mới chạy được thì lúc này ta có thể sử dụng tổ chức đồng bộ để giải quyết vấn đề này. Để giải quyết vấn đề này chúng ta sẽ sử dụng đồng bộ hóa Semaphore như sau:

Tiến trình 1 sẽ chạy công việc job1, tiến trình 2 sẽ chạy công việc job2.

Và trong trường hợp này e(s) sẽ được gán bằng 0.

Giờ chúng ta sẽ chạy thử job2 trước. Khi tiến trình 2 chạy trước thì sẽ gặp phải hàm down, với e(s) khởi tạo bằng 0 thì khi này nó sẽ = -1 và thỏa mãn điều kiện < 0 và job2 sẽ bị khóa và tiến vào danh sách f(s). Thì lúc này CPU sẽ cấp lại cho tiến trình 1 chạy. Cuối cùng job1 vẫn là tiến trình thực thi trước. Khi tiến trình 1 chạy sẽ gọi hàm Up, lúc này e(s) sẽ tăng lên 1 và = 0 thỏa mãn điều kiện <= 0, và tiến hành giải phóng tiến trình 2 và đưa tiến trình 2 vào ready-list. Sau khi tiến trình 2 thoát ra thì sẽ thực thi được job2.

Với cấu trúc và cách sử dụng như thế này, ta có thể khống chế được việc job1 sẽ chắc chắn luôn luôn được thực thi trước job2

Slides 88 - 10:

Để có thể dễ viết đúng các chương trình đồng bộ hóa hơn, Hoare(1974) và Brinch & Hansen (1975) đã đề nghị một cơ chế cao hơn được cung cấp bởi ngôn ngữ lập trình , là monitor.

Monitor là 1 cấu trúc đặc biệt gồm các biến, thủ tục và cấu trúc dữ liệu

Cơ chế đồng bộ được cung cấp bởi ngôn ngữ lập trình

- Hỗ trợ với các chức năng, chẳng hạn như Semaphore

- Dễ dàng sử dụng và phát hiện hơn Semaphore

• Đảm bảo tự động loại trừ lẫn nhau

• Sử dụng biến điều kiện để thực hiện đồng bộ hóa

Slides 89 - 11:

Đây là hình cấu trúc cơ bản của Monitor gồm có: biến, thủ tục và cấu trúc dữ liệu

Trong đó hàng đợi monitor, nơi diễn ra các hoạt động chia sẻ dữ liệu của các mã khởi tạo.

Slides 90 - 12:

Và đây là tất cả các thao tác trên 1 nhóm tài nguyên gồm có:

Tên monitor; phần khai báo, chia sẻ các biến; mã khởi tạo

---Slides 91 - 13:

Slides 92 - 14:

Và tiếp theo đây chúng ta sẽ tìm hiểu về Message Passing (Trao đổi thông điệp)

Giải pháp này dựa trên cơ sở trao đổi thông điệp với hai primitive Send và Receive để thực hiện sự đồng bộ hóa.

• Các quy trình phải đặt tên cho nhau một cách rõ ràng:

– send (P, message) – gửi một thông điệp đến 1 tiến trình hay gửi vào hộp thư

– receive(Q, message) – nhận 1 thông điệp từ một tiến trình hay từ bất kỳ 1 tiến trình nào, tiến trình gọi sẽ chờ nếu không có thông điệp nào để nhận.

• Thuộc tính của liên kết liên lạc

– Các liên kết được thiết lập tự động

– Một liên kết được liên kết với chính xác một cặp quy trình giao tiếp

– Giữa mỗi cặp tồn tại đúng một liên kết

– Liên kết có thể là một chiều, nhưng thường là hai hướng

Slides 93 - 15:

Các vấn đề cổ điển của đồng bộ hóa:

1. Vấn đề đầu tiên là: Vấn đề bộ đệm bị giới hạn hay còn gọi là Vấn đề nhà sản xuất-người tiêu dùng

- Vấn đề: Hai tiến trình cùng chia sẻ 1 bộ đệm có kích thước giới hạn. Một trong hai tiến trình đóng vai trò người sản xuất – tạo ra dữ liệu và đặt dữ liệu vào bộ đệm - và tiến trình kia đóng vai trò người tiêu thụ - lấy dữ liệu từ bộ đệm ra để xử lý.

- Giải pháp là Sử dụng ba semaphore : ***full***, đếm số chỗ đã có dữ liệu trong bộ đệm;***empty***, đếm số chỗ còn trống trong bộ đệm; và ***mutex***, kiểm tra việc Producer và Consumer không truy xuất đồng thời đến bộ đệm.

2. Vấn đề thứ 2 là: Vấn đề của độc giả và nhà văn

- Vấn đề: Nhiều tiến trình đồng thời sử dụng một cơ sở dữ liệu. Các tiến trình chỉ cần lấy nội dung của cơ sở dữ liệu được gọi là các tiến trình Reader, nhưng một số tiến trình khác lại có nhu cầu sửa đổi, cập nhật dữ liệu trong cơ sở dữ liệu chung này, chúng được gọi là các tiến trình Writer.

- Giải pháp là: Sử dụng 1 biến chung **rc** để ghi nhớ các tiến trình Reader muốn truy xuất cơ sở dữ liệu. Hai semaphore cũng được sử dụng: **mutex,** kiểm soát sự truy cập đến **rc,** và **db**, kiểm tra sự truy xuất độc quyền đến cơ sở dữ liệu

3. Vấn đề thứ 3 là: Vấn đề ẩm thực - triết gia

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

- Giải pháp: Có 3 hướng tiếp cận để xử lý:

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

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

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