**Inter Process Communication (IPC)**

**I. Inter Process Compunication (IPC) introduction.**

Nói ngắn gọn thì có 3 vấn đề cần quan tâm, giải quyết:

* Các tiến trình truyền thông tin cho nhau như thế nào?
* Bảo đảm rằng không có 2 hoặc nhiều hơn tiến trình đi vào đường truyền của nhau (... do not get each other's way).
* Nếu các tiến trình có phụ thuộc thì thứ tự phải phù hợp. Ví dụ nếu A tạo ra dữ liệu và B in chúng thì B phải chờ cho đến khi A tạo xong phần dữ liệu cần thiết cho việc in.
  1. **Race Conditons - Các tình trạng tương tranh**

Tình trạng tương tranh (Race conditions) là một tình huống xảy ra khi nhiều tiến trình hay luồng (threads) cùng truy cập vào tài nguyên chung.

Hai tiến trình có thể sử dụng tài nguyên chung,ví dụ một phần bộ nhớ hoặc tệp.Một ví dụ vùng đệm máy in (a print spooler). Khi tiến trình muốn in thì ghi tên tệp vào thư mục đệm (spooler directory). Một tiến trình khác làm nhiệm vụ in (printer daemon), theo chu kỳ nó kiểm tra nếu có tệp cần in thì in tệp đó và loại khỏi thư mục. Nếu có 2 P cùng muốn ghi tên tệp muốn in vào 1 khe của thư mục đệm thì tương tranh sẽ xảy ra. Ta cần giải quyết như thế nào?

**Critical Regions** or Critical section (The part of the program which accesses the shared memory or file)

Critical Region là đoạn chương trình (của một tiến trình) truy cập tài nguyên chung. Nếu ta loại trừ được trường hợp 2 tiến trình thực hiện CR cùng lúc thì sẽ loại bỏ được tương tranh. Ta có 4 điều kiện để đảm bảo cho một lời giải tốt:

1. Không có 2 tiến trình cùng lúc ở trong CR.

2. Không có điều kiện ràng buộc gì về tốc độ và số CPU.

3. Không có tiến trình nào đang ở ngoài CR có thể chặn (block) một tiến trình khác.

4. Không có tiến trình nào phải chờ đợi vô thời hạn để vào CR của nó.

**II. Các phương pháp Mutual exclusion with Busy waiting**

**2.1 Disabling Interrupts**

Vô hiệu hóa ngắt chỉ dùng được trên máy tính chỉ có 1 CPU. Ngay trường hợp này cũng có vấn đề. Nếu tiến trình trước khi ra khỏi CR quên không enable ngắt thì sao? Nếu máy có nhiều CPU thì chỉ có 1 CPU có hiệu lực và tiến trình ngoài CPU này vẫn có thể sử dụng tài nguyên.

**2.2 Lock Variables**

Tài nguyên có khóa (spin lock), nếu giá trị khóa bằng 0: tài nguyên đang free, 1 tài nguyên đang được sử dụng. Vậy nếu tiến trình thấy L=0 thì đặt L=1 và chạy CR, khi kết thúc đạt lại L=0, Nếu L=1 thì phải chờ.

Tình huống sai: A thấy L=0 nhưng chưa kịp đặt L=1 thì hết thời gian CPU, đến lượt B thấy L=0, đặt L=1 và vào CR. Khi A trở lại đặt L=1 và cũng vào CR, như thế cả A và B cùng ở trong CR: có tương tranh.

Trường hợp khác: A và B vào CR cùng lúc nên đều thấy L=0, cùng đặt L=1 và vào CR.

**2.3 The TSL (Test-and-Set Lock) Instruction**

Giống Lock, nhưng nhờ có phần cứng hỗ trợ và thực hiện thay đổi giá trị Lock in an atomic way, nghĩa là bảo đảm rằng trong quá trình đọc và thay đổi L không có tiến trình nào được truy cập L. Tuy nhiên vẫn có nguy cơ tiến trình không trả lại giá trị khóa = 0.

**2.4 Turn Variable or Strict Alternation Approach**

Phương pháp này chỉ dùng cho trường hợp có 2 tiến trình chia sẻ tài nguyên. Hai tiến trình sử dụng một biết chung có tên là turn, biến này nhận 2 giá trị tương ứng với PID của hai tiến trình. Ví dụ giả sử tiến trình A có PID=i, tiến trình B có PID=j. Khi đó cách hoạt động như sau: Khi chuẩn bị bước vào CR, A kiểm tra, nếu turn=i thì thực hiện CR,sau đó đặt lại giá trị turn=j. Nếu không thì phải chờ cho đến khi turn turn=i. Vấn đề: Nếu A đến lượt nhưng và dặt turn=i không vào CR hoặc sau khi thực hiện xong CR mà không đặt lại turn=j thì B sẽ chờ mãi mãi. Chú ý: phương pháp này không cần sự hỗ trợ của hệ điều hành.

**2.5 Peterson’s Solution (1981)**

So với phương pháp Turn variable thì phương pháp này dùng thêm biến flag. Nếu A muốn vào CR thì đặt flag=1. Nếu flag của A bằng 1, flag của B = 1 và giá trị turn = pid của B thì A được vào CR trước. Nếu chỉ có một flag=1 thì tiến trình tương ứng được vào trước.

**Tổng kết các phương pháp Busy Waiting**:

Phương pháp Peterson khá hoàn hảo nhưng một tt vẫn phải chờ và dùng thời gian CPU.

**III. Các phương pháp Synchronous solution**

**3.1 Sleep & Wakeup**

Phương pháp Peterson khá hoàn hảo nhưng một tt vẫn phải chờ và dùng thời gian CPU.

Làm sao để giải phóng CPU trong khi có tt phải chờ?

Giải pháp là dùng cặp Sleep và Wakeup

Sleep là lời gọi hệ thống đưa một tt vào trạng thái block, tức là suspended cho đến khi được tt khác đánh thức.

**The Producer-Consumer Problem (also known as the bounded-buffer problem)**

Vấn đề: Producer (P) lần lượt đặt item mới vào Buffer (B), còn Consumer thì lần lượt lấy các item từ B.

Nếu P muốn đặt item vào B nhưng B bị đầy (full) thì P đi ngủ (sleep), khi C lấy item từ B thì B sẽ có chỗ trống, vì thế C sau khi lấy đi thì đánh thức P dậy (wakeup) để đưa item vào B.

Cũng tương tự, nếu V muốn lấy item từ B nhưng B bị rỗng (empty) thì C đi ngủ (sleep), khi P đưa item vào B thì B có item, vì thế P sau đưa item vào B thì đánh thức C dậy (wakeup) để lấy item từ B.

Cách làm này đã giải quyết vấn đề Busy waiting mà không đòi hỏi CPU phải tham gia khi một tt đang sleep.

Vấn đề: C định lấy item từ B nhưng thấy B rỗng nên định đi ngủ. Tuy nhiên C chuwakijp ngủ thì scheduler dừng C. Khi P đưa i tem mới vào B và đánh thức C thì không thành công do C chưa ngủ, sau đó P đi ngủ. Khi C được scheduler cho tiếp tục thì C đọc count đã có từ lần trước, thấy giá trị này = 0 nên đi ngủ. Vây là cả P và C đều ở trạng thái ngủ mà không có ai đánh thức và ngủ mãi.

**3.2 Semaphore**