**MÔN HỌC: HỆ ĐIỀU HÀNH**

**CÂU HỎI VÀ BÀI TẬP CHƯƠNG 5**

1. **Khi nào thì xảy ra tranh chấp?**

* **Đọc/Ghi dữ liệu chia sẻ:** Khi nhiều tiến trình cùng thực hiện đọc và ghi trên cùng một vùng dữ liệu chia sẻ mà không có biện pháp đồng bộ hóa.
* **Thực hiện các phép toán không nguyên tác (non-atomic operations):** Các phép toán mà không được đảm bảo là nguyên tác có thể tạo ra race condition. Ví dụ, nếu bạn đọc một giá trị, thực hiện một phép toán và sau đó ghi lại giá trị, thì có thể có race condition nếu giá trị bị thay đổi bởi một tiến trình khác trong quá trình bạn đang thực hiện phép toán.
* **Sự kiện không đồng bộ:** Khi các tiến trình chạy không đồng bộ với nhau và không có cơ chế đồng bộ hóa để đảm bảo thứ tự thực hiện chính xác.
* **Chia sẻ tài nguyên không an toàn:** Sử dụng tài nguyên chia sẻ mà không có cơ chế đảm bảo an toàn, như không sử dụng mutex hoặc semaphores để kiểm soát truy cập đồng thời.

1. **Vấn đề vùng tranh chấp (critical section) là gì?**

Xem xét một hệ thống có n tiến trình

Mỗi tiến trình có một vùng tranh chấp là một đoạn code:

• Thực hiện việc thay đổi giá trị của dữ liệu được chia sẻ (có thể là các biến, bảng dữ liệu, file, ...)

• Khi một tiến trình đang thực hiện vùng tranh chấp của mình thì các tiến trình khác **KHÔNG** được thực hiện vùng tranh chấp của chúng.

Vấn đề vùng tranh chấp chính là thiết kế cách thức xử lý các vấn đề trên.

1. **Có những yêu cầu nào dành cho lời giải của bài toán vùng tranh chấp?**

Lời giải phải thỏa ba tính chất:

(1) **Loại trừ tương hỗ (Mutual exclusion):** Khi một process P đang thực thi trong vùng tranh chấp (CS) của nó thì không có process Q nào khác đang thực thi trong CS của Q.

(2) **Progress:** Một tiến trình tạm dừng bên ngoài vùng tranh chấp không được ngăn cản các tiến trình khác vào vùng tranh chấp.

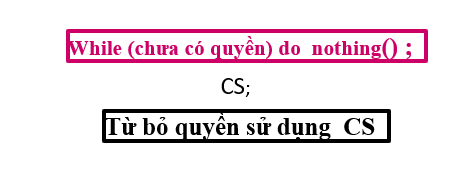
(3) **Chờ đợi giới hạn (Bounded waiting):** Mỗi process chỉ phải chờ để được vào vùng tranh chấp trong một khoảng thời gian có hạn định nào đó. Không xảy ra tình trạng đói tài nguyên (starvation).

1. **Có mấy loại giải pháp đồng bộ? Kể tên và trình bày đặc điểm của các loại giải pháp đó?**

Có 2 nhóm giải pháp chính:

**Nhóm giải pháp Busy Waiting**

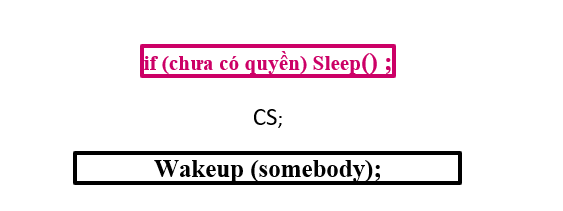
* Tính chất:
  + Tiếp tục tiêu thụ CPU trong khi chờ đợi vào miền găng (thông qua việc kiểm tra điều kiện vào CS liên tục).
  + Không đòi hỏi sự trợ giúp của hệ điều hành.
* Cơ chế chung:



* Bao gồm một vài loại:
  + Sử dụng các biến cờ hiệu.
  + Sử dụng việc kiểm tra luân phiên.
  + Giải pháp của Peterson.
  + Cấm ngắt (giải pháp phần cứng – hardware).
  + Chỉ thị TSL (giải pháp phần cứng – hardware).

**Nhóm giải pháp Sleep & Wakeup**

* Tính chất:
  + Từ bỏ CPU khi chưa được vào CS.
  + Cần sự hỗ trợ từ hệ điều hành (để đánh thức process và đưa process vào trạng thái blocked).
* Cơ chế chung:



* Bao gồm một vài loại:
  + Semaphore.
  + Monitor.
  + Message.

1. **Phân tích và đánh giá ưu, nhược điểm của các giải pháp đồng bộ busy waiting (cả phần cứng và phần mềm)?**

**Ưu điểm:**

* Phần Cứng (Hardware-based) Busy Waiting

*Đơn giản và hiệu quả:* Busy waiting thường đơn giản và dễ triển khai trên phần cứng. Nó không đòi hỏi nhiều tài nguyên và có thể được thực hiện một cách hiệu quả.

*Không có chuyển đổi ngữ cảnh (Context Switch):* Busy waiting không đòi hỏi việc chuyển đổi ngữ cảnh giữa các tiến trình. Trong môi trường đòi hỏi hiệu suất cao, điều này có thể là một ưu điểm.

* Phần Mềm (Software-based) Busy Waiting:

*Tích hợp trực tiếp vào mã nguồn:* Giải pháp busy waiting dễ tích hợp trực tiếp vào mã nguồn của chương trình mà không cần sự hỗ trợ từ phần cứng.

*Phù hợp cho môi trường ít chuyển đổi ngữ cảnh:* Trong môi trường mà chuyển đổi ngữ cảnh có chi phí cao, busy waiting có thể là một lựa chọn tốt.

**Nhược Điểm:**

* Phần Cứng (Hardware-based) Busy Waiting:

*Lãng phí CPU:* Busy waiting tiêu tốn nguồn lực CPU mà không làm gì cả khi điều kiện đồng bộ không đạt được. Điều này dẫn đến lãng phí tài nguyên.

*Không phù hợp cho môi trường đa nhiệm:* Trong môi trường đa nhiệm, busy waiting có thể làm giảm hiệu suất của hệ thống vì nó chiếm tài nguyên CPU mà không thực hiện công việc hữu ích.

* Phần Mềm (Software-based) Busy Waiting:

*Lãng phí tài nguyên:* Busy waiting trên phần mềm có thể dẫn đến lãng phí tài nguyên CPU và năng lượng, đặc biệt là khi không có sự kiện đồng bộ xảy ra trong thời gian dài.

*Không phù hợp cho các tác vụ chờ lâu:* Trong các trường hợp nơi thời gian chờ lâu, busy waiting có thể làm tăng chi phí của việc giữ lại tài nguyên CPU mà không có nhiệm vụ thực sự được thực hiện.

*Nguy cơ bị đóng băng (Deadlock):* Trong môi trường đa nhiệm, có nguy cơ xảy ra deadlock nếu một tiến trình giữ lock và không thể hoàn thành công việc để giải phóng lock.

1. **Semaphore là gì? Đặc điểm của semaphore? Cách thức hiện thực semaphore? Có mấy loại semaphore? Khi sử dụng semaphore cần lưu ý những vấn đề gì?**

Semaphore là một cơ chế đồng bộ được sử dụng để kiểm soát truy cập vào tài nguyên chia sẻ. Semaphore là một biến được bảo vệ và có thể được tăng hoặc giảm giá trị. Khi giá trị của semaphore là 0, thì tài nguyên đang được sử dụng và các tiến trình khác phải chờ đợi cho đến khi giá trị của semaphore tăng lên.

**Đặc điểm của semaphore:**

* Semaphore là một biến được bảo vệ, tức là chỉ có các tiến trình được phép truy cập vào semaphore.
* Semaphore có thể được tăng hoặc giảm giá trị.
* Giá trị của semaphore được sử dụng để kiểm soát truy cập vào tài nguyên chia sẻ

**Cách thức hiện thực semaphore:**

* Phần cứng: Semaphore được hiện thực bằng phần cứng trong các vi xử lý hiện đại. Các vi xử lý thường cung cấp các thanh ghi bit để hỗ trợ semaphore.
* Phần mềm: Semaphore được hiện thực bằng phần mềm trong các hệ điều hành. Hệ điều hành sẽ cung cấp các API để tạo và sử dụng semaphore.

**Có hai loại semaphore:**

* Semaphore nhị phân (binary semaphore): Semaphore nhị phân chỉ có thể có hai giá trị là 0 hoặc 1. Semaphore nhị phân thường được sử dụng để kiểm soát truy cập vào tài nguyên chia sẻ chỉ có thể được sử dụng bởi một tiến trình tại một thời điểm.
* Semaphore đếm (counting semaphore): Semaphore đếm có thể có nhiều giá trị khác nhau. Semaphore đếm thường được sử dụng để kiểm soát truy cập vào tài nguyên chia sẻ có thể được sử dụng bởi nhiều tiến trình cùng một lúc.

**Khi sử dụng semaphore cần lưu ý những vấn đề:**

* Cần đảm bảo rằng chỉ các tiến trình được phép truy cập vào semaphore.
* Cần đảm bảo rằng giá trị của semaphore luôn được cập nhật một cách chính xác.
* Cần lưu ý đến các vấn đề về deadlock và starvation khi sử dụng semaphore.

1. **Monitor và Critical Region là gì?**

* Monitor là một kiểu dữ liệu trừu tượng cung cấp các cơ chế đồng bộ hóa cho các tiến trình chia sẻ dữ liệu. Một monitor bao gồm một tập hợp các biến được bảo vệ và một tập hợp các hàm. Các biến được bảo vệ chỉ có thể được truy cập bởi các hàm được định nghĩa trong monitor. Các hàm trong monitor được sử dụng để kiểm soát truy cập vào các biến được bảo vệ.
* Critical Region là một đoạn mã được bảo vệ, trong đó chỉ có một tiến trình có thể thực thi tại một thời điểm. Critical region thường được sử dụng để bảo vệ truy cập vào dữ liệu chia sẻ.

1. **Đặc điểm và yêu cầu đồng bộ của các bài toán đồng bộ kinh điển?**

**Bài toán Tiếp sức (Producer-Consumer):**

* Đặc điểm: Bài toán này liên quan đến việc sản xuất và tiêu thụ dữ liệu trong một kho lưu trữ có giới hạn.
* Yêu cầu đồng bộ:

Đảm bảo rằng tiến trình sản xuất không thêm dữ liệu mới vào kho khi nó đầy.

Đảm bảo rằng tiến trình tiêu thụ không xóa dữ liệu khỏi kho khi nó trống.

**Bài toán Đọc-Ghi (Readers-Writers):**

* Đặc điểm: Bài toán này thường liên quan đến việc quản lý quyền truy cập vào một tài nguyên có thể được đọc và ghi bởi nhiều tiến trình.
* Yêu cầu đồng bộ:

Nếu có một tiến trình đang đọc, các tiến trình khác cũng có thể đọc.

Nếu có một tiến trình đang ghi, không có tiến trình nào khác được phép đọc hoặc ghi.

**Bài toán Ăn trưa 5 người tri thức (Dining Philosophers):**

* Đặc điểm: Bài toán này mô phỏng tình huống 5 tri thức ngồi quanh một bàn, giữ một đĩa spaghetti giữa mỗi đôi nắm đũa.
* Yêu cầu đồng bộ:

Tránh deadlock: Không có tri thức nào được phép giữ một nắm đũa mà không giữ được cặp đĩa còn lại.

Tránh starvation: Mỗi tri thức phải có cơ hội để ăn.

**Yêu cầu Đồng Bộ chung cho các bài toán:**

* Ngăn chặn Race Condition: Đảm bảo rằng không có hai tiến trình nào cùng thực hiện một thao tác thay đổi trên dữ liệu chia sẻ cùng một lúc.
* Bảo vệ Dữ Liệu Chia Sẻ: Đảm bảo tính nhất quán và an toàn của dữ liệu chia sẻ giữa các tiến trình hoặc luồng.
* Không Deadlock và Starvation: Tránh tình trạng mắc kẹt (deadlock) và không đợi đến mãi mãi (starvation) trong quá trình đồng bộ hóa.
* Hiệu suất: Cần đảm bảo rằng cơ chế đồng bộ hóa không tạo ra chi phí quá lớn và không làm giảm hiệu suất của hệ thống.

1. **(Bài tập mẫu) Xét giải pháp phần mềm do Dekker đề nghi để tổ chức truy xuất độc quyền cho 2 tiến trình. Hai tiến trình P0 và P1 chia sẻ các biến sau:**

|  |
| --- |
| **boolean flag[2]; /\* initially false \*/**  **int turn;** |

**Cấu trúc một tiến trình Pi (với i = 0 hay 1 và j là tiến trình còn lại) như sau:**

|  |
| --- |
| **while (true) {**  **flag[i] = true;**  **while (flag[j]) {**  **if (turn == j) {**  **flag[i] = false;**  **while (turn == j)**  **; /\* do nothing \*/**  **flag[i] = true;**  **}**  **}**  **/\* critical section \*/**  **turn = j;**  **flag[i] = false;**  **/\* remainder section \*/**  **}** |

**Giải pháp này có thỏa 3 yêu cầu trong viêc giải quyết tranh chấp không?**

Trả lời:  
Giải pháp này thỏa 3 yêu cầu trong giải quyết tranh chấp vì:

- Loại trừ tương hỗ: Tiến trình Pi chỉ có thể vào vùng tranh chấp khi flag[j] = false. Giả sử P0 đang ở trong vùng tranh chấp, tức là flag[0] = true và flag[1] = false. Khi đó P1 không thể vào vùng tranh chấp (do bị chặn bởi lệnh while (flag[j])). Tương tự cho tình huống P1 vào vùng tranh chấp trước.

- Progress: Giá trị của biến turn chỉ có thể thay đổi ở cuối vùng tranh chấp. Giả sử chỉ có 1 tiến trình Pi muốn vào vùng tranh chấp. Lúc này, flag[j] = false và tiến trình Pi sẽ được vào vùng tranh chấp ngay lập tức. Xét trường hợp cả 2 tiến trình đều muốn vào vùng tranh chấp và giá trị của turn đang là 0. Cả flag[0] và flag[1] đều bằng true. Khi đó, P0 sẽ được vào vùng tranh chấp, bởi tiến trình P1 sẽ thay đổi flag[1] = false (lệnh kiểm tra điều kiện if (turn == j) chỉ đúng với P1). Tương tự cho trường hợp turn = 1.

- Chờ đợi giới hạn: Pi chờ đợi lâu nhất là sau 1 lần Pj vào vùng tranh chấp (flag[j] = false sau khi Pj ra khỏi vùng tranh chấp). Tương tự cho trường hợp Pj chờ Pi.

1. **Xét giải pháp đồng bộ hóa sau:**

|  |
| --- |
| **while (TRUE) {**  **int j = 1-i;**  **flag[i]= TRUE;**  **turn = i;**  **while (turn == j && flag[j]==TRUE);**  **critical-section ();**  **flag[i] = FALSE;**  **Noncritical-section ();**  **}** |

**Giải pháp này có thỏa yêu cầu độc quyền truy xuất không?**

Ta thấy code này có đoạn j = 1 – i => đề chỉ đang nói tới xét 2 tiến trình P1 và P0 vì khi i = 0 thì j = 1 và ngược lại.

- Giải thuật này không thoả mãn:

Xét tình huống khi flag[0] = 1; turn = 0; lúc này P0 vào CS, Nếu lúc đó flag[1] = 1, P1 có thể gán turn = 1 và vào luôn CS (2 tiến trình cùng vào CS một lúc).

1. **Giả sử một máy tính không có chỉ thị TSL, nhưng có chỉ thị Swap có khả năng hoán đổi nội dung của hai từ nhớ chỉ bằng một thao tác không thể phân chia:**

|  |
| --- |
| **procedure Swap(var a,b: boolean){**  **var temp : boolean;**  **begin**  **temp := a;**  **a:= b;**  **b:= temp;**  **end;**  **}** |

**Sử dụng chỉ thị này có thể tổ chức truy xuất độc quyền không? Nếu có, xây dựng cấu trúc chương trình tương ứng.**

Câu trả lời là có, có thể sử dụng chỉ thị Swap để tổ chức truy xuất độc quyền.

Sử dụng hai biến toàn cục là turn và flag. Biến turn được sử dụng để xác định tiến trình nào đang có quyền truy cập vào vùng truy xuất độc quyền. Biến flag được sử dụng để đánh dấu tiến trình nào đang muốn truy cập vào vùng truy xuất độc quyền.

Khi một tiến trình i muốn vào vùng truy xuất độc quyền, nó sẽ thực hiện các bước sau:

1. Đặt flag[i] thành TRUE.
2. Swap turn và i.
3. Kiểm tra xem turn có bằng i hay không. Nếu không đúng, thì tiến trình i sẽ chờ đợi cho đến khi turn bằng i.
4. Tiến trình i thực hiện các thao tác trong vùng truy xuất độc quyền.
5. Đặt flag[i] thành FALSE.

|  |
| --- |
| void Swap(boolean \*a, boolean \*b)  {  boolean temp = \*a;  \*a = \*b;  \*b = temp;  } |
| * Biến chia sẻ (khởi tạo là false)   bool lock;   bool key;   * Process *Pi*     do {  key = true;  while (key == true)  Swap(&lock, &key);  *critical section*  lock = false;  *remainder section*  } while (1)  Không thỏa mãn bounded waiting |

1. **Xét hai tiến trình sau:**

**process A {while (TRUE) na = na +1;}**

**process B {while (TRUE) nb = nb +1;}**

**a. Đồng bộ hóa xử lý của 2 tiến trình trên, sử dụng 2 semaphore tổng quát, sao cho tại bất kỳ thời điểm nào cũng có nb <= na <= nb +10.**

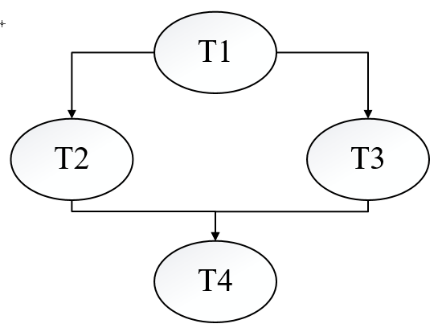
**b. Nếu giảm điều kiện chỉ còn là na <= nb +10, cần sửa chữa giải pháp trên như thế nào?**

**c. Giải pháp trên còn đúng nếu có nhiều tiến trình loại A và B cùng thực hiện?**

|  |
| --- |
| **a.**  Semaphore\_1 = 0;  Semaphore\_2 = 10;  Process A:      While(1)      {          wait(Semaphore\_2);          na = na + 1;          signal(Semaphore\_1);      }  Process B:      While(1)      {          wait(Semaphore\_1);          nb = nb + 1;          signal(Semaphore\_2);      } |
| **b.**  Semaphore\_2 = 10;  Process A:      While(1)      {          wait(Semaphore\_2);          na = na + 1;      }  Process B:      While(1)      {          nb = nb + 1;          signal(Semaphore\_2);      } |

**c.** Đúng, vì có thể có nhiều tiến trình loại A hoặc loại B cùng thực hiện nhưng chỉ có 2 biến Semaphore toàn cục mà chúng sẽ thao tác (đối với câu b là 1 Semaphore).

1. **(Bài tập mẫu) Xét một hệ thống có 4 tiểu trình T1, T2, T3, T4. Quan hệ giữa các tiểu trình này được biểu diễn như sơ đồ bên dưới, với mũi tên từ tiểu trình (Tx) sang tiểu trình (Ty) có nghĩa là tiểu trình Tx phải kết thúc quá trình hoạt động của nó trước khi tiểu trình Ty bắt đầu thực thi. Giả sử tất cả các tiểu trình đã được khởi tạo và sẵn sàng để thực thi. Hãy sử dụng semaphore để đồng bộ hoạt động của các tiểu trình sao cho đúng với sơ đồ đã cho.**



Trả lời:

Khai báo và khởi tạo các semaphore:

init(sem1,0); //khởi tạo semaphore sem1 có giá trị bằng 0

init(sem2,0); //khởi tạo semaphore sem2 có giá trị bằng 0

void T4(void)   
{

wait(sem2)

wait(sem2)

//T4 thực thi

}

void T3(void)   
{

wait(sem1)

//T3 thực thi

signal(sem2)

}

void T2(void)   
{

wait(sem1)

//T2 thực thi

signal(sem2)

}

void T1(void)   
{

//T1 thực thi

signal(sem1)

signal(sem1)

}

1. **Một biến X được chia sẻ bởi 2 tiến trình cùng thưc hiện đoạn code sau:**

|  |
| --- |
| **do**  **X = X + 1;**  **if (X == 20) X = 0;**  **while (TRUE);** |

**Bắt đầu với giá trị X = 0, chứng tỏ rằng giá tri X có thể vượt quá 20. Cần sửa chữa đoạn chương trình trên như thế nào để đảm bảo X không vượt quá 20?**

Có thể dẫn đến tình trạng đồng thời (concurrency) khi hai tiến trình cùng lúc thực hiện trên biến X. Điều này có thể khiến giá trị của X vượt quá 20 trước khi được đặt lại về 0.

|  |
| --- |
| do {  mutex.lock();  X = X + 1;  if (X == 20) X = 0;  mutex.unlock();  } while (TRUE); |

1. **Xét 2 tiến trình xử lý đoạn chương trình sau:**

**process P1 {A1 ; A2 }**

**process P2 {B1 ; B2 }**

**Đồng bộ hóa hoạt động của 2 tiến trình này sao cho cả A1 và B1 đều hoàn tất trươc khi A2 và B2 bắt đầu.**

|  |
| --- |
| Semaphore s1 = 0, s2 = 0;  process P1 {  A1;  signal(s1);  wait(s2);  A2;  }  process P2 {  B1;  signal(s2);  wait(s1);  B2;  } |

1. **Tổng quát hóa bài tập 14 cho các tiến trình có đoạn chương trình sau:**

**process P1 { for ( i = 1; i <= 100; i ++) Ai }**

**process P2 { for ( j = 1; j <= 100; j ++) Bj }**

**Đồng bộ hóa hoạt động của 2 tiến trình này sao cho với k bất kỳ (2<=k<=100), Ak chỉ có thể bắt đầu khi B(k-1) đã kết thúc và Bk chỉ có thể bắt đầu khi A(k-1) đã kết thúc.**

|  |
| --- |
| Semaphore s1 = 1, s2 = 0;  process P1 {  for (i = 1; i <= 100; i++) {  wait(s1);  Ai;  signal(s2);  }  }  process P2 {  for (j = 1; j <= 100; j++) {  wait(s2);  Bj;  signal(s1);  }  } |

1. **Sử dụng semaphore để viết lại chương trình sau theo mô hình xử lý đồng hành:**

**w := x1 \* x2**

**v := x3 \* x4**

**y := v \* x5**

**z := v \* x6**

**x := w \* y**

**z := w \* z**

**ans := y + z**

|  |
| --- |
| Semaphore s1 = 0, s2 = 0, s3 = 0, s4 = 0, s5 = 0, s6 = 0;  process P1 {  w := x1 \* x2;  signal(s1);  }  process P2 {  v := x3 \* x4;  signal(s2);  }  process P3 {  wait(s2);  y := v \* x5;  signal(s3);  }  process P4 {  wait(s2);  z := v \* x6;  signal(s4);  }  process P5 {  wait(s1);  wait(s3);  x := w \* y;  signal(s5);  }  process P6 {  wait(s1);  wait(s4);  z := w \* z;  signal(s6);  }  process P7 {  wait(s5);  wait(s6);  ans := y + z;  } |