TRƯỜNG ĐẠI HỌC CÔNG NGHỆ ĐÔNG Á
EAST ASIA UNIVERSITY OF TECHNOLOGY

Chương 4

4.2 Đồng bộ hóa tiến trình

Nội dung

- Cơ sở
- Vấn đề đoạn găng
- Giải pháp của Peterson
- Phần cứng đồng bộ hóa
- Kỹ thuật cờ báo (Semaphores)

4.2.1. Cơ sở

- Sự truy nhập đồng thời đến dữ liệu chia sẻ có thể gây ra sự mâu thuẫn.
- Để duy trì tính nhất quán dữ liệu cần có cơ chế đảm bảo thực hiện các tiến trình hợp tác theo thứ tự.
- Giả sử rằng chúng ta muốn đưa ra một giải pháp cho vấn đề tiến trình sản xuất - tiến trình tiêu thụ mà đều điền vào buffer. Chúng ta có thể làm được bằng cách có một biến nguyên count để theo dõi số phần tử trong buffer.
  - Khởi tạo count=0.
  - Nó được tăng bởi tiến trình sản xuất khi thêm vào buffer 1 phần tử.
  - Nó bị giảm bởi tiến trình tiêu thụ khi lấy khỏi buffer 1 phần tử

Producer

```
while (true) {
        /* produce an item and put in nextProduced */
        while (count == BUFFER_SIZE)
                ; // do nothing
            buffer [in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;
            count++;
}
```

Consumer

```
while (true) {
        while (count == 0)
                ; // do nothing
                nextConsumed =  buffer[out];
                 out = (out + 1) % BUFFER_SIZE;
                  count--;

                    /*  consume the item in nextConsumed
}
```

Trạng thái tranh đua (Race condition)

- count++ có thể được thực thi như sau:

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- có thể được thực thi như sau:

```
register2 = count
register2 = register2 - 1
count = register2
```

- Xét sự thực hiện đan xen với ban đầu "count = 5":

S0: producer execute register1 = count	{register1 = 5}
S1: producer execute register1 = register1 + 1	{register1 = 6}
S2: consumer execute register2 = count	{register2 = 5}
S3: consumer execute register2 = register2 - 1	{register2 = 4}
S4: producer execute count = register1	{count = 6}
S5: consumer execute count = register2	{count = 4}

4.2.2. Vấn đề đoạn găng (Critical-Section)

- Xét hệ thống gồm *n* tiến trình $\{P_0, P_1, \ldots, P_{n-1}\}$.
- Mỗi tiến trình có một đoạn mã, gọi là **đoạn găng**, mà tại đó tiến trình có thể thay đổi các biến chung, cập nhật bảng, ghi tệp…
- Đặc điểm quan trọng của hệ thống là tại mỗi thời điểm chỉ có 1 tiến trình thực hiện trong đoạn găng của nó.

⇔ sự thực hiện các đoạn găng là *loại trừ lẫn nhau* theo thời gian.

- Vấn đề đoạn găng là thiết kế một giao thức mà các tiến trình sử dụng để hợp tác. Mỗi tiến trình phải yêu cầu sự cho phép để bước vào đoạn găng của nó. Đoạn mã thực hiện yêu cầu này được gọi là **đoạn vào**. Sau đoạn găng có thể có **đoạn ra**.

Cấu trúc tổng quát của tiến trình P_i

```
do {
        đoạn vào

          đoạn găng

        đoạn ra

          đoạn còn lại
} while (TRUE) ;
```

Giải pháp cho vấn đề đoạn găng

Một giải pháp cho vấn đề đoạn găng phải thỏa mãn 3 yêu cầu:

1. **Loại trừ lẫn nhau**: nếu tiến trình P_i đang thực hiện trong đoạn găng của nó thì các tiến trình khác không được thực hiện trong đoạn găng của chúng.
2. **Chọn tiến trình tiếp theo được vào đoạn găng**: nếu không có tiến trình nào đang trong đoạn găng của nó và một số tiến trình muốn vào đoạn găng của chúng thì chỉ những tiến trình đang không trong đoạn còn lại mới là ứng cử viên.
3. **Chờ đợi có hạn**: tồn tại giới hạn số lần các tiến trình khác được phép vào đoạn găng của chúng sau khi một tiến trình yêu cầu vào đoạn găng đến trước khi yêu cầu đó được đáp ứng.

Các phương pháp xử lý đoạn găng

- **kernel không ưu tiên trước**: không cho phép một tiến trình bị ưu tiên trước khi nó đang chạy trong kernel mode; tiến trình đó sẽ chạy cho đến khi nó thoát khỏi kernel mode.
  - Không gây tình trạng đua tranh trong cấu trúc
  - Windows 2000/XP, UNIX cũ, Linux trước phiên bản 2.6
- **kernel có ưu tiên trước**: cho phép một tiến trình bị ưu tiên trước khi nó đang chạy trong kernel mode.
  - Cần thiết kế cẩn thận để tránh tình trạng đua tranh, nhất là với kiến trúc đa xử lý đối xứng (SMP). Vì sao?
  - Thích hợp hơn với lập trình thời gian thực, vì nó sẽ cho phép 1 tiến trình thời gian thực ưu tiên trước 1 tiến trình khác đang chạy trong kernel.
  - Linux 2.6, một số phiên bản thương mại của UNIX (Solaris, IRIX)

4.2.3. Giải pháp của Peterson

- Giải pháp cho 2 tiến trình P_0, P_1
- Giả sử các lệnh LOAD và STORE là nguyên tử (atomic); nghĩa là không thể bị ngắt.
- Hai tiến trình chia sẻ 2 biến:
  - int turn;
  - boolean flag[2]
- Biến turn bằng 0/1. `turn==i` thì P_i được phép vào đoạn găng.
- `flag[i]=true` cho biết tiến trình P_i sẵn sàng vào đoạn găng.

Thuật toán cho tiến trình P_i

```
while (true) {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);

            ĐOẠN_GĂNG

      flag[i] = FALSE;

            ĐOẠN_CÒN_LẠI

}
```

Chứng minh thuật toán trên thỏa mãn cả 3 điều kiện của giải pháp?

4.2.4. Phần cứng đồng bộ hóa

- Nhiều HĐH cung cấp sự hỗ trợ phần cứng cho mã đoạn găng
- Đơn bộ xử lý – có thể vô hiệu các ngắt
  - Đoạn mã đang chạy thực hiện mà không bị giành ưu tiên
  - Nói chung rất không hiệu quả với các hệ thống đa bộ xử lý
    - Việc chuyển thông điệp đến tất cả các bộ xử lý tốn rất nhiều thời gian, làm trễ sự vào đoạn găng của các tiến trình
- Nhiều HĐH hiện đại cung cấp các lệnh phần cứng nguyên tử
  - Nguyên tử = không thể bị ngắt
  - Hoặc là **test** từ nhớ (memoryword) và **set** giá trị
  - Hoặc là hoán đổi (**swap**) nội dung của 2 từ nhớ

Lệnh TestAndSet

- Định nghĩa:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

Giải pháp dùng TestAndSet

- Biến boolean chia sẻ là lock, được khởi tạo là false.
- Giải pháp cho mỗi tiến trình:

```
while (true) {
        while (TestAndSet (&lock))
                ;   /* do nothing

                //    đoạn găng

        lock = FALSE;

                //      đoạn còn lại

}
```

Lệnh Swap

- Định nghĩa:

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp;
}
```

Giải pháp dùng Swap

- Biến boolean chia sẻ là lock, được khởi tạo là false; Mỗi tiến trình có một biến boolean cục bộ là key.
- Giải pháp cho mỗi tiến trình:

```
while (true) {
        key = TRUE;
        while (key == TRUE)
              Swap (&lock, &key);

                //    đoạn găng

         lock = FALSE;

                //    đoạn còn lại

}
```

4.2.5. Kỹ thuật dùng cờ báo (Semaphore)

- Công cụ đồng bộ hóa dễ dùng hơn với người lập trình ứng dụng.
- Semaphore *S* – biến integer
- Hai hoạt động nguyên tử chuẩn có thể thay đổi S:
  - wait() và signal(), còn được gọi là P() và V()

```
wait (S) {
     while S <= 0
          ; // no-op
     S--;
 }

signal (S) {
   S++;
}
```

Semaphore – Công cụ đồng bộ hóa tổng quát

- Counting semaphore – giá trị S có thể không bị giới hạn
- Binary semaphore – giá trị S chỉ có thể bằng 0 hoặc 1; dễ thực hiện hơn
  - Còn được gọi là khóa loại trừ (mutexlocks)
- Có thể thực thi counting semaphore S như binary semaphore
- Cung cấp sự loại trừ lẫn nhau
  - Semaphore S; // khởi tạo bằng 1
  - wait (S);

```
        Đoạn găng
    signal (S);
```

Thực thi Semaphore

- Phải đảm bảo rằng không thể có 2 tiến trình có thể thực hiện wait () và signal () trên cùng semaphore tại cùng thời điểm.
- Do đó, sự thực thi trở thành vấn đề đoạn găng: mã của wait và signal được đặt trong đoạn găng.
- Khi 1 tiến trình trong đoạn găng, các tiến trình khác cố gắng vào đoạn găng phải lặp liên tục trong mã đoạn vào, làm lãng phí các chu kỳ CPU – gọi là busy waiting.

Thực thi Semaphore không có Busy waiting

- Với mỗi semaphore có một waiting queue. Mỗi phần tử trong waiting queue có 2 trường dữ liệu:
  - value (kiểu integer)
  - pointer, con trỏ tới bản ghi kế tiếp trong list
- Hai hoạt động:
  - block – đặt tiến trình gọi vào waiting queue thích hợp
    - Khi tiến trình thực hiện wait(), nếu giá trị S không dương thì thay vì đợi busywaiting, tiến trình có thể gọi block()
    - Trạng thái tiến trình được chuyển thành *waiting*
  - wakeup – loại 1 tiến trình khỏi waiting queue và đặt nó vào ready queue.
    - Khi 1 tiến trình khác gọi signal (), tiến trình được khởi động lại bởi wakeup()

Thực thi Semaphore không có Busy waiting (tiếp)

- Sự thực thi của wait:

```
wait (S){
    value--;
    if (value < 0) {
        thêm tiến trình này vào waiting queue
        block(); }
}
```

- Sự thực thi của signal:

```
signal (S){
    value++;
    if (value <= 0) {
        loại tiến trình P khỏi waiting queue
        wakeup(P); }
}
```

Deadlock và Starvation

- Deadlock (bế tắc) – hai hoặc nhiều tiến trình đang đợi vô hạn một sự kiện chỉ có thể được gây ra bởi một trong những tiến trình đợi đó.
- Gọi **S** và **Q** là hai semaphore được khởi tạo bằng 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- Starvation – khóa vô hạn. Một tiến trình có thể không bao giờ được đưa ra khỏi waiting queue tương ứng của semaphore.
  - Có thể xuất hiện khi waiting queue tổ chức dạng LIFO

BÀI TẬP