Chương 4: Quản lý tiến trình

Hệ điều hành

ThS. Đinh Xuân Trường

truongdx@ptit.edu.vn

Posts and Telecommunications Institute of Technology

Faculty of Information Technology 1

CNTT1

Học viện Công nghệ Bưu chính Viễn thông

August 15, 2022

Mục lục

Nội dung chương 4

1. Các khái niệm liên quan đến tiến trình
2. Luồng - Thread
3. Điều độ tiến trình
4. Đồng bộ hóa các tiến trình đồng thời
5. Tình trạng bế tắc và đói

Đồng bộ hóa các tiến trình đồng thời

Tiến trình đồng thời hay còn được gọi *tiến trình tương tranh* là các tiến trình cùng tồn tại.
Quản lý tiến trình đồng thời bao gồm các vấn đề :

- Liên lạc giữa các tiến trình
- Cạnh tranh và chia sẻ tài nguyên
- Phối hợp và đồng bộ hóa các tiến trình
- Vấn đề bế tắc
- Đói/Thiếu tài nguyên (starvation)

Tiến trình cạnh tranh tài nguyên với nhau

- Cạnh tranh trực tiếp sử dụng chung tài nguyên:
  - Tiến trình cạnh tranh sẽ phải chờ đợi do chỉ một tiến trình được cấp tài nguyên
  - Ảnh hưởng thời gian thực hiện của tiến trình
- Ảnh hưởng gián tiếp:
  - Do cùng nhu cầu sử dụng một số tài nguyên như bộ nhớ, đĩa, thiết bị ngoại vi hỗ trợ I/O

Đối với tiến trình cạnh tranh tài nguyên với nhau cần giải quyết một số vấn đề sau:

- Đảm bảo **loại trừ tương hỗ** (mutual exclusion):
  - **Loại trừ tương hỗ** là đảm bảo tiến trình này đang truy cập tài nguyên thì tiến trình khác không được truy cập tài nguyên đó
  - Tài nguyên đó gọi là **tài nguyên nguy hiểm**. Đoạn chương trình có yêu cầu sử dụng tài nguyên nguy hiểm gọi là **đoạn nguy hiểm** (critical section)
  - Hai tiến trình không được phép thực hiện đồng thời trong đoạn nguy hiểm của mình.
- Không để xảy ra **bế tắc** (deadlock):
  - Bế tắc: tình trạng hai hoặc nhiều tiến trình không thể thực hiện tiếp do chờ đợi lẫn nhau tài nguyên mà không được cấp.

- Ví dụ: Tiến trình P1 và P2 cần cấp phát đồng thời hai tài nguyên T1, T2; hệ điều hành cấp phát T1 cho P1 và cấp phát T2 cho P2. P1 chờ P2 giải phóng T2 trong khi P2 cũng chờ P1 giải phóng T1 trước khi có thể thực hiện => P1 và P2 rơi vào bế tắc không thể thực hiện tiếp.

- Không để **đói / thiếu** tài nguyên (starvation)
  - Đói tài nguyên (starvation) là chờ đợi quá lâu mà không đến lượt sử dụng một tài nguyên nào đó.
  - Ví dụ: Ba tiến trình P1, P2, P3 cùng có yêu cầu lặp đi lặp lại một tài nguyên. P1 và P2 lần lượt được cấp tài nguyên nhiều lần trong khi P3 không bao giờ đến lượt => không thực hiện tiếp được dù không có bế tắc.

Tiến trình hợp tác với nhau qua tài nguyên chung: Trao đổi thông tin giữa các tiến trình hợp tác là chia sẻ vùng nhớ dùng chung (biến toàn thể), hay các file.

- Việc các tiến trình đồng thời truy cập dữ liệu dùng chung làm nảy sinh một số vấn đề :
  - Đòi hỏi đảm bảo loại trừ tương hỗ
  - Xuất hiện tình trạng bế tắc và đói
  - Yêu cầu đảm bảo tính nhất quán dữ liệu
- **Điều kiện chạy đua** (race condition): tình huống mà một số luồng /tiến trình đọc, ghi dữ liệu sử dụng chung và kết quả phụ thuộc vào thứ tự các thao tác đọc, ghi
  - Đặt thao tác truy cập và cập nhật dữ liệu dùng chung vào đoạn nguy hiểm
  - Sử dụng loại trừ tương hỗ để các thao tác này không bị tiến trình khác xen ngang.

Tiến trình có liên lạc nhờ gửi thông điệp:

- Các tiến trình hợp tác có thể trao đổi thông tin trực tiếp với nhau bằng cách gửi thông điệp (message passing).
- Cơ chế liên lạc được hỗ trợ bởi thư viện của ngôn ngữ lập trình hoặc HĐH.
- Không có yêu cầu loại trừ tương hỗ
- Có thể xuất hiện bế tắc và đói

Yêu cầu quan trọng khi đồng bộ hóa tến trình là giải quyết vấn đề đoạn nguy hiểm và loại trừ tương hỗ.
Giải pháp cho vấn đề đoạn nguy hiểm cần thỏa mãn yêu cầu sau:

- *Loại trừ tương hỗ* : tại mỗi thời điểm, chỉ một tiến trình được ở trong đoạn nguy hiểm
- *Tiến triển*: một tiến trình đang thực hiện ở ngoài đoạn nguy hiểm *không được phép* ngăn cản các tiến trình khác vào đoạn nguy hiểm của mình
- *Chờ đợi có giới hạn*: nếu tiến trình có nhu cầu vào đoạn nguy hiểm thì tiến trình đó phải được vào sau một khoảng thời gian hữu hạn nào đó

Giải pháp cho vấn đề đoạn nguy hiểm được xây dựng dựa trên các giả thiết sau:

- Giải pháp không phụ thuộc vào tốc độ của các tiến trình
- Không tiến trình nào được phép nằm quá lâu trong đoạn nguy hiểm
- Thao tác đọc và ghi bộ nhớ là thao tác nguyên tử (atomic) và không thể bị xen ngang giữa chừng

Các giải pháp được chia thành 3 nhóm chính:

- Nhóm giải pháp phần mềm (software solutions): người dùng tự thực hiện, HĐH cung cấp một số công cụ
- Nhóm giải pháp phần cứng (hardware solutions): Dựa trên một số lệnh máy đặc biệt (Interrupt disable, test-and-set)
- Nhóm sử dụng hỗ trợ của HDH hoặc thư viện ngôn ngữ lập trình

Giải thuật Peterson do Gary Peterson để xuất năm 1981 cho bài toán đoạn nguy hiểm là giải pháp thuộc nhóm phần mềm
Giải thuật Peterson đề xuất cho bài toán đồng bộ hai tiến trình P0 và P1:

- P0 và P1 thực hiện đồng thời với một tài nguyên chung và một đoạn nguy hiểm chung
- Mỗi tiến trình thực hiện vô hạn và xen kẽ giữa đoạn nguy hiểm với phần còn lại của tiến trình
- Yêu cầu 2 tiến trình trao đổi thông tin qua 2 biến chung:
  - *Int Turn*: xác định đến lượt tiến trình nào được vào đoạn nguy hiểm
  - Cờ cho mỗi tiến trình: flag[i]=true nếu tiến trình thứ i yêu cầu được vào đoạn nguy hiểm

Đồng bộ hóa các tiến trình đồng thời (cont.)

Giải thuật peterson

```
...
bool flag[2];
int turn;

void P0(){     //tiến trình P0
    for(;;){     //lặp vô hạn
        flag[0]=true;
        turn=1;
        while(flag[1] && turn==1);//lặp đến khi điều kiện không thỏa
        <Đoạn nguy hiểm>
        flag[0]=false;
        <Phần còn lại của tiến trình>
    }
}

void P1(){     //tiến trình P1
    for(;;){     //lặp vô hạn
        flag[1]=true;
        turn=0;
        while(flag[0] && turn==0);//lặp đến khi điều kiện không thỏa
        <Đoạn nguy hiểm>
        flag[1]=false;
        <Phần còn lại của tiến trình>
    }
}

void main(){
      flag[0]=flag[1]=0;
      turn=0;
//tắt tiến trình chính, chạy đồng thời hai tiến trình P0 và P1
      StartProcess(P0);
      StartProcess(P1);
}
```

Hình 2.9: Giải thuật Peterson cho hai tiến trình

Thỏa mãn các yêu cầu:

- Điều kiện loại trừ tương hỗ
- Điều kiện tiến triển:
  - P0 chỉ có thể bị P1 ngăn cản vào đoạn nguy hiểm nếu *flag[1] = true* và *turn =1* luôn đúng
  - Có 2 khả năng với P1 ở ngoài đoạn nguy hiểm:
    - P1 chưa sẵn sàng vào đoạn nguy hiểm => *flag[1] = false*, P0 có thể vào ngay đoạn nguy hiểm
    - P1 đã đặt *flag[1]=true* và đang trong vòng lặp while => *turn = 1* hoặc *turn = 0*:
    - Turn = 0: P0 vào đoạn nguy hiểm ngay
    - Turn = 1: P1 vào đoạn nguy hiểm, sau đó đặt *flag[1] = false* => quay lại khả năng 1
- Chờ đợi có giới hạn

Giải thuật peterson:

- Sử dụng trên thực tế tương đối phức tạp
- Đòi hỏi tiến trình đang yêu cầu vào đoạn nguy hiểm phải nằm trong trạng thái **chờ đợi tích cực**
- Chờ đợi tích cực: tiến trình vẫn phải sử dụng CPU để kiểm tra xem có thể vào đoạn nguy hiểm? gây ra lãng phí CPU

- Phần cứng máy tính có thể được thiết kế để giải quyết vấn đề loại trừ trong tương hỗ và đoạn nguy hiểm.
- Giải pháp phần cứng thường dễ sử dụng và có tốc độ tốt.
- **Cấm các ngăt**: cấm không ñể xẩy ra ngắt trong thời gian tiến trình ñang ở trong ñoạn nguy hiểm ñể truy cập tài nguyên.
- **Sử dụng lệnh máy đặc biệt**: Phần cứng được thiết kế có một số lệnh máy đặc biệt
  - Hai thao tác kiểm tra và thay đổi giá trị cho một biến (ô nhớ) *kiểm tra và xác lập* Test_and_Set, hoặc các thao tác so sánh và hoán đổi giá trị hai biến, được thực hiện trong cùng một lệnh máy
  - Đảm bảo được thực hiện cùng nhau mà không bị xen vào giữa – thao tác nguyên tử (atomic)

Loại trừ tương hỗ sử dụng lệnh máy Test_and_Set:

```
…
const int n; //n là số lượng tiến trình
bool lock;

void P(int i){     //tiến trình P(i)
    for(;;){     //lặp vô hạn
        while(Test_and_Set(lock));//lặp đến khi điều kiện không thỏa
        <Đoạn nguy hiểm>
        lock = false;
        <Phần còn lại của tiến trình>
    }
}

void main(){
      lock = false;
//tắt tiến trình chính, chạy đồng thời n tiến trình
      StartProcess(P(1));
      ...
      StartProcess(P(n));
}
```

Sử dụng lệnh máy đặc biệt :

- Ưu điểm:
  - Việc sử dụng tương đối đơn giản và trực quan
  - Có thể dùng để đồng bộ nhiều tiến trình
  - Có thể sử dụng cho trường hợp đa xử lý với nhiều CPU nhưng có bộ nhớ chung
- Nhược điểm:
  - Chờ đợi tích cực
  - Có thể gây đói tài nguyên (starvation)

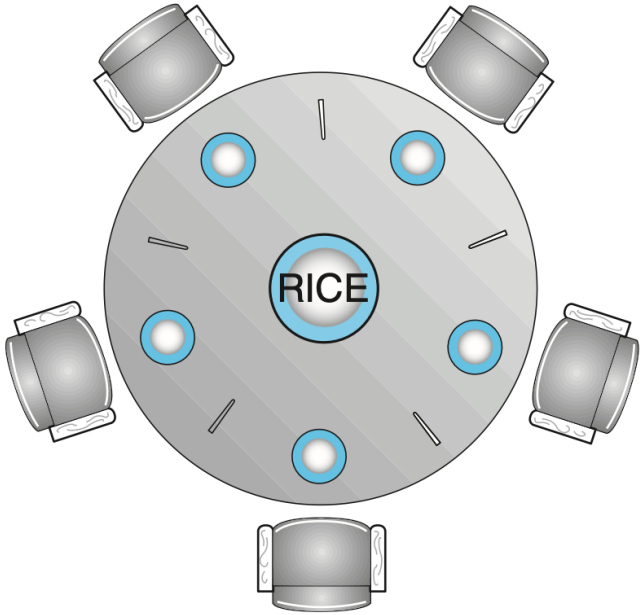
Cờ hiệu semaphore S là 1 biến nguyên được khởi tạo bằng khả năng phục vụ đồng thời của tài nguyên
Giá trị của S chỉ có thể thay đổi nhờ gọi 2 thao tác là *Wait* và *Signal*:

- Wait(S):
  - Giảm S đi 1 đơn vị
  - Nếu giá trị của S<0 thì tiến trình gọi wait(S) sẽ bị phong tỏa (blocked)
  - Nếu giá trị của S không âm, tiến trình sẽ ñược thực hiện tiếp.
- Signal(S):
  - Tăng S lên 1 đơn vị
  - Nếu giá trị của S<=0: 1 trong các tiến trình đang bị phong tỏa được giải phóng và có thể thực hiện tiếp

```
const int n; //n là số lượng tiến trình
semaphore S = 1;
void P(int i){    //tiến trình P(i)
    for(;;){    //lặp vô hạn
        Wait(S);
        <Đoạn nguy hiểm>
        Signal(S);
        <Phần còn lại của tiến trình>
    }
}
void main(){
//tắt tiến trình chính, chạy đồng thời n tiến trình
            StartProcess(P(1));
            ...
            StartProcess(P(n));
}
```

- Để tránh tình trạng chờ đợi tích cực, sử dụng 2 thao tác phong tỏa (block) và đánh thức (wakeup):
  - Nếu tiến trình thực hiện thao tác wait, giá trị cờ hiệu âm thì nó sẽ bị phong tỏa (bởi thao tác block) và thêm vào hàng đợi của cờ hiệu
  - Khi có 1 tiến trình thực hiện thao tác signal thì 1 trong các tiến trình bị khóa sẽ được chuyển sang trạng thái sẵn sàng nhờ thao tác đánh thức (wakeup) chứa trong signal
- Khi tiến trình cần truy cập tài nguyên, thực hiện thao tác Wait của cờ hiệu tương ứng
- Sau khi dùng xong tài nguyên, tiến trình thực hiện thao tác Signal trên cùng cờ hiệu: tăng giá trị cờ hiệu và cho phép một tiến trình đang phong tỏa được thực hiện tiếp

Bài toán triết gia ăn cơm - Dining-Philosophers Problem:



- 5 triết gia ngồi trên ghế quanh 1 bàn tròn
  - Trên bàn có 5 cái đũa: bên phải và bên trái mỗi người có 1 cái
  - Triết gia có thể nhặt 2 chiếc đũa theo thứ tự bất kì: phải nhặt từng chiếc một và đũa không nằm trong tay người khác

- Khi cầm cả 2 đũa triết gia bắt đầu ăn và không đặt đũa trong thời gian ăn
- Sau khi ăn xong, triết gia đặt 2 đũa xuống bàn

- 5 triết gia như 5 tiến trình đồng thời với tài nguyên nguy hiểm là đũa và đoạn nguy hiểm là đoạn dùng đũa để ăn
- Cờ hiệu cho phép giải quyết bài toán như sau:
  - Mỗi đũa được biểu diễn bằng 1 cờ hiệu
  - Nhặt đũa: wait()
  - Đặt đũa xuống: signal()

Bài toán triết gia ăn cơm - Dining-Philosophers Problem:

```
…
semaphore chopstick[5] = {1,1,1,1,1,1};

void Philosopher(int i){          //tiến trình P(i)
    for(;;){     //lặp vô hạn
        Wait(chopstick[i]);        //lấy đũa bên trái
        Wait(chopstick[(i+1)%5]); //lấy đũa bên phải
        <Ăn cơm>
        Signal(chopstick[(i+1)%5]);
        Signal(chopstick[i]);
        <Suy nghĩ>
    }
}

void main(){
// chạy đồng thời 5 tiến trình
      StartProcess(Philosopher(1));
      ...
      StartProcess(Philosopher (5));
}
```

Hình 2.15. Bài toán triết gia ăn cơm sử dụng cờ hiệu

Người sản xuất người tiêu dùng với bộ đệm hạn chế

Người sản xuất: tạo ra sản phẩm, xếp nó vào 1 chỗ gọi là bộ đệm, mỗi lần 1 sản phẩm

Người tiêu dùng: lấy sản phẩm từ bộ đệm, mỗi lần 1 sản phẩm

Dung lượng bộ đệm hạn chế, chứa tối đa N sản phẩm

- 3 yêu cầu đồng bộ:
  - Người sản xuất và tiêu dùng không được sử dụng bộ đệm cùng lúc
  - Khi bộ đệm rỗng, người tiêu dùng không nên cố lấy sản phẩm
  - Khi bộ đệm đầy, người sản xuất không được thêm sản phẩm
- Giải quyết bằng cờ hiệu:
  - Yêu cầu 1: sử dụng cờ hiệu lock khởi tạo bằng 1
  - Yêu cầu 2: cờ hiệu empty, khởi tạo bằng 0
  - Yêu cầu 3: cờ hiệu full, khởi tạo bằng N

Người sản xuất, người tiêu dùng với bộ đệm hạn chế

Const int N; // kích thước bộ đệm
Semaphore lock = 1;
	Semaphore empty = 0;
	Semaphore full = N

Void producer () {
for (; ;) {
<sản xuất>
wait (full);
wait (lock);
<thêm 1 sản phẩm vào bộ đệm>
signal (lock);
signal (empty);
}
}
	Void consumer() {
	for (; ;) {
	wait (empty);
	wait (lock);
	<lấy 1 sản phẩm từ bộ đệm>
	signal (lock);
	signal (full);
	<tiêu dùng>
	}
	}

Void main() {
startProcess(producer);
startProcess(consumer);
}	

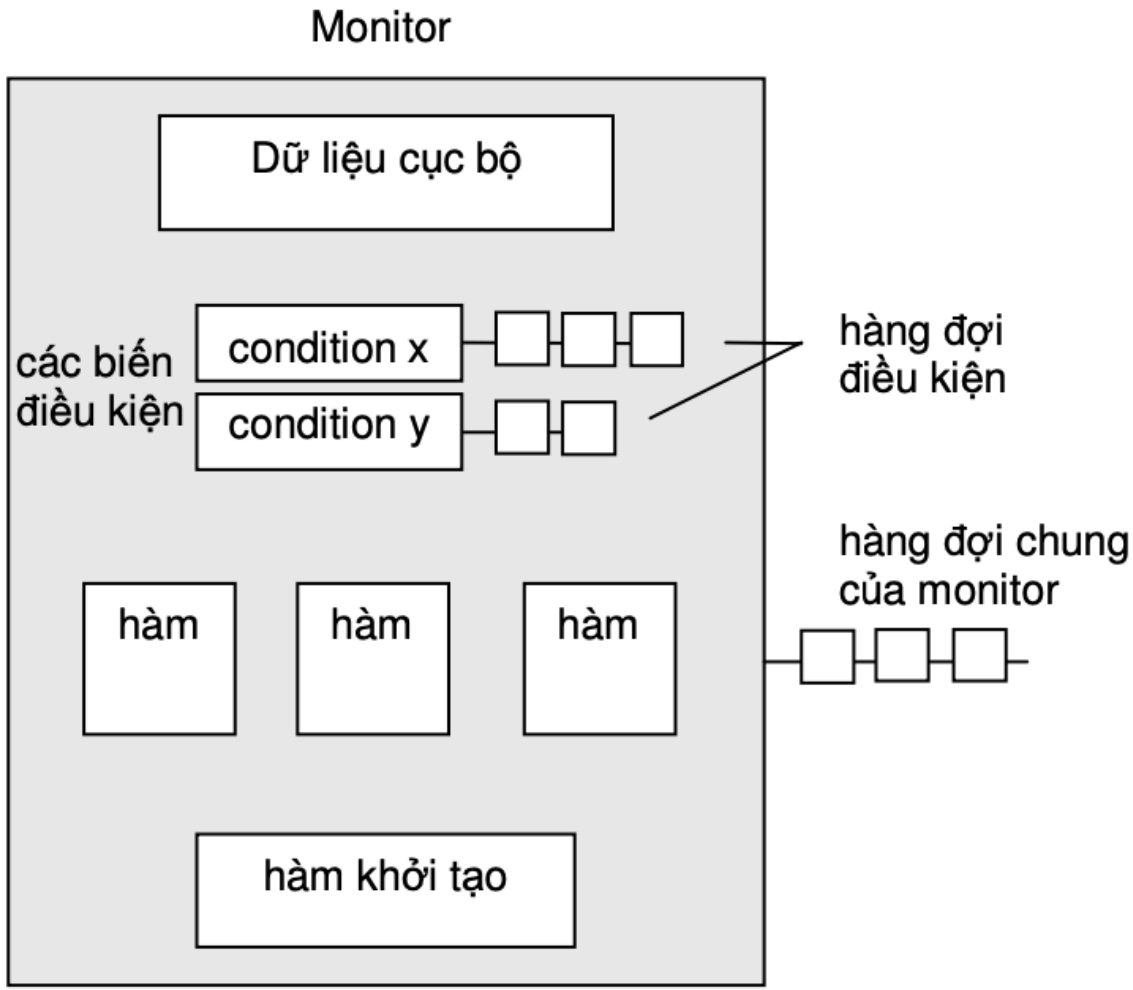
Monitor được định nghĩa dưới dạng một kiểu dữ liệu trừu tượng của ngôn ngữ lập trình bậc cao, chẳng hạn như một class của C++ hoặc Java. Mỗi monitor gồm một dữ liệu riêng, hàm khởi tạo, và một số hàm hoặc phương thức để truy cập dữ liệu với các đặc điểm sau:

- Tiến trình/dòng chỉ có thể truy cập dữ liệu của monitor thông qua các hàm hoặc phương thức của monitor
- Tại mỗi thời điểm:
  - Chỉ một tiến trình được thực hiện trong monitor
  - Tiến trình khác gọi hàm của monitor sẽ bị phong tỏa, xếp vào hàng đợi của monitor để chờ cho đến khi monitor được giải phóng

- Đảm bảo loại trừ tương hỗ đối với đoạn nguy hiểm, đặt tài nguyên nguy hiểm vào trong monitor
- Tiến trình đang thực hiện trong monitor và bị dừng lại để đợi sự kiện hay một điều kiện nào đó được thỏa mãn => trả lại monitor để tiến trình khác sử dụng.
- Tiến trình chờ đợi sẽ được khôi phục lại từ điểm dừng sau khi điều kiện đang chờ đợi được thỏa mãn => Sử dụng các biến điều kiện

- Các biến điều kiện được khai báo và sử dụng trong monitor với 2 thao tác: *cwait()* và *csignal()*:
  - *x.cwait():*
    - Tiến trình đang ở trong monitor và gọi cwait bị phong tỏa cho tới khi điều kiện x xảy ra
    - Tiến trình bị xếp vào hàng đợi của biến điều kiện x
    - Monitor được giải phóng và một tiến trình khác sẽ được vào
  - *x.csignal():*
    - Tiến trình gọi csignal để thông báo điều kiện x đã thỏa mãn
    - Nếu có tiến trình đang bị phong tỏa và nằm trong hàng đợi của x do gọi x.cwait() trước đó sẽ được giải phóng
    - Nếu không có tiến trình bị phong tỏa thì thao tác csignal sẽ không có tác dụng gì cả

Cấu trúc monitor với các biến điều kiện:



Tổng kết

Chương 4 Quản lý tiến trình

- Đồng bộ hóa các tiến trình đồng thời
- Tình trạng bế tắc và đói

Tổng kết

- Trình bày bài tập lớn
- Chữa đề thi các năm
- Các câu hỏi thắc mắc