Nguyên lý hệ điều hành

Nguyễn Hải Châu
Khoa Công nghệ thông tin
Trường Đại học Công nghệ

Đồng bộ hóa tiến trình

Ví dụ đồng bộ hóa (1)

Tiến trình ghi *P*:

```
while (true) {
   while (counter==SIZE) ;
   buf[in] = nextItem;
   in = (in+1) % SIZE;
   counter++;
}
```

buf: Buffer
SIZE: cỡ của buffer
counter: Biến chung

Tiến trình đọc *Q*:

```
while (true) {
   while (counter==0) ;
   nextItem = buf[out];
   out = (out+1) % SIZE;
   counter--;
}
```

- Đây là *bài toán vùng đệm có giới hạn*

Ví dụ đồng bộ hóa (2)

- Các toán tử ++ và -- có thể được cài đặt như sau:

- counter++

```
register1 = counter;
register1 = register1 + 1;
counter = register1;
```

- counter--

```
register2 = counter;
register2 = register2 - 1;
counter = register2;
```

P và Q có thể nhận được các giá trị khác nhau của counter tại cùng 1 thời điểm nếu như đoạn mã xanh và đỏ thực hiện xen kẽ nhau.

Ví dụ đồng bộ hóa (3)

- Giả sử P và Q thực hiện song song với nhau và giá trị của counter là 5:

```
register1 = counter;          // register1=5
register1 = register1 + 1;    // register1=6
register2 = counter;          // register2=5
register2 = register2 - 1;    // register2=4
counter = register1;          // counter=6 !!
counter = register2;          // counter=4 !!
```


Ví dụ đồng bộ hóa (4)

- Lỗi: Cho phép *P* và *Q* **đồng thời thao tác trên biến chung counter**. Sửa lỗi:

```
register1 = counter;          // register1=5
register1 = register1 + 1;    // register1=6
counter = register1;          // counter=6
register2 = counter;          // register2=6
register2 = register2 - 1;    // register2=5
counter = register2;          // counter=5
```

Tương tranh và đồng bộ

- Tình huống xuất hiện khi nhiều tiến trình cùng thao tác trên dữ liệu chung và kết quả các thao tác đó phụ thuộc vào thứ tự thực hiện của các tiến trình trên dữ liệu chung gọi là *tình huống tương tranh* (*race condition*)
- Để tránh các tình huống tương tranh, các tiến trình cần được *đồng bộ* theo một phương thức nào đó ⇒ Vấn đề nghiên cứu: Đồng bộ hóa các tiến trình

7

Khái niệm về đoạn mã găng (1)

- Thuật ngữ: Critical section
- Thuật ngữ tiếng Việt: Đoạn mã găng, đoạn mã tới hạn.
- Xét một hệ có n tiến trình P_0, P_1, ..., P_n, mỗi tiến trình có một đoạn mã lệnh gọi là đoạn mã găng, ký hiệu là CS_i, nếu như trong đoạn mã này, các tiến trình thao tác trên các biến chung, đọc ghi file... (tổng quát: thao tác trên dữ liệu chung)

8

Khái niệm về đoạn mã găng (2)

- Đặc điểm quan trọng mà hệ n tiến trình này cần có là: Khi một tiến trình P_i thực hiện đoạn mã CS_i thì không có tiến trình P_j nào khác được phép thực hiện CS_j
- Mỗi tiến trình P_i phải "xin phép" (entry section) trước khi thực hiện CS_i và thông báo (exit section) cho các tiến trình khác sau khi thực hiện xong CS_i.

9

Khái niệm về đoạn mã găng (3)

- Cấu trúc chung của P_i để thực hiện đoạn mã găng CS_i.

```
do {
   Xin phép (ENTRY_i) thực hiện CS_i; // Entry section
   Thực hiện CS_i;
   Thông báo (EXIT_i) đã thực hiện xong CS_i; // Exit section
   Phần mã lệnh khác (REMAIN_i); // Remainder section
} while (TRUE);
```

10

Khái niệm về đoạn mã găng (4)

- Viết lại cấu trúc chung của đoạn mã găng:

```
do {
   ENTRY_i;        // Entry section
   Thực hiện CS_i; // Critical section
   EXIT_i;         // Exit section
   REMAIN_i;       // Remainder section
} while (TRUE);
```

11

Giải pháp cho đoạn mã găng

- Giải pháp cho đoạn mã găng cần thỏa mãn 3 điều kiện:
  - Loại trừ lẫn nhau (mutual exclusion): Nếu P_i đang thực hiện CS_i thì P_j không thể thực hiện CS_j $\forall j \neq i$.
  - Tiến triển (progress): Nếu không có tiến trình P_i nào thực hiện CS_i và có m tiến trình P_{j1}, P_{j2}, ..., P_{jm} muốn thực hiện CS_{j1}, CS_{j2}, ..., CS_{jm} thì chỉ có các tiến trình không thực hiện $REMAIN_{jk}$ (k=1,...,m) mới được xem xét thực hiện CS_{jk}.
  - Chờ có giới hạn (bounded waiting): *sau khi* một tiến trình P_i có yêu cầu vào CS_i và *trước khi* yêu cầu đó được chấp nhận, số lần các tiến trình P_j (với $j \neq i$) được phép thực hiện CS_j phải bị giới hạn.

12

Ví dụ: giải pháp của Peterson

- Giả sử có 2 tiến trình P_0 và P_1 với hai đoạn mã găng tương ứng CS_0 và CS_1
- Sử dụng một biến nguyên *turn* với giá trị khởi tạo 0 hoặc 1 và mảng boolean *flag*[2]
- *turn* có giá trị *i* có nghĩa là P_i được phép thực hiện CS_i (*i*=0,1)
- nếu flag[i] là TRUE thì tiến trình P_i đã sẵn sàng để thực hiện CS_i

13

Ví dụ: giải pháp của Peterson

- Mã lệnh của P_i:

```
do {
   flag[i] = TRUE;
   turn = j;
   while (flag[j] && turn == j) ;
   CSi;
   flag[i] = FALSE;
   REMAINi;
} while (1);
```

14

Chứng minh giải pháp Peterson

- Xem chứng minh giải pháp của Peterson thỏa mãn 3 điều kiện của đoạn mã găng trong giáo trình (trang 196)
- Giải pháp "kiểu Peterson":
  - Phức tạp khi số lượng tiến trình tăng lên
  - Khó kiểm soát

15

Semaphore

16

Thông tin tham khảo

- Edsger Wybe Dijkstra (người Hà Lan) phát minh ra khái niệm semaphore trong khoa học máy tính vào năm 1972
- Semaphore được sử dụng lần đầu tiên trong cuốn sách "The operating system" của ông

Edsger Wybe Dijkstra (1930-2002)

17

Định nghĩa

- Semaphore là một biến nguyên, nếu không tính đến toán tử khởi tạo, chỉ có thể truy cập thông qua hai toán tử *nguyên tố* là wait (hoặc P) và signal (hoặc V).
  - P: proberen – kiểm tra (tiếng Hà Lan)
  - V: verhogen – tăng lên (tiếng Hà Lan)
- Các tiến trình có thể *sử dụng chung* semaphore
- Các toán tử là nguyên tố để đảm bảo không xảy ra trường hợp như ví dụ đồng bộ hóa đã nêu

18

Toán tử wait và signal

```
wait(S)     // hoặc P(S)
{
   while (S<=0);
   S--;
}
```

- Toán tử wait: Chờ khi semaphore S âm và giảm S đi 1 nếu S>0

```
signal(S)     // hoặc V(S)
{
   S++;
}
```

- Toán tử signal: Tăng S lên 1

19

Sử dụng semaphore (1)

- Với bài toán đoạn mã găng:

```
do {
   wait(mutex); // mutex là semaphore khởi tạo 1
   CS_i;
   signal(mutex);
   REMAIN_i;
} while (1);
```

20

Sử dụng semaphore (2)

- Xét hai tiến trình P_1 và P_2, P_1 cần thực hiện toán tử O_1, P_2 cần thực hiện O_2 và O_2 chỉ được thực hiện sau khi O_1 đã hoàn thành
- Giải pháp: Sử dụng semaphore *synch* = 0

- P_1:

```
...
O1;
signal(synch);
...
```

- P_2:

```
...
wait(synch);
O2;
...
```

21

Cài đặt semaphore cổ điển

- Định nghĩa cổ điển của wait cho ta thấy toán tử này có *chờ bận* (busy waiting), tức là tiến trình phải chờ toán tử wait kết thúc nhưng CPU vẫn phải làm việc: Lãng phí tài nguyên
- Liên hệ cơ chế polling trong kiến trúc máy tính
- Cài đặt semaphore theo định nghĩa cổ điển:
  - Lãng phí tài nguyên CPU với các máy tính 1 CPU
  - Có lợi nếu thời gian chờ wait ít hơn thời gian thực hiện context switch
  - Các semaphore loại này gọi là *spinlock*

22

Cài đặt semaphore theo cấu trúc

- Khắc phục chờ bận: Chuyển vòng lặp chờ thành việc sử dụng toán tử block (tạm dừng)
- Để khôi phục thực hiện từ block, ta có toán tử wakeup
- Khi đó để cài đặt, ta có cấu trúc dữ liệu mới cho semaphore:

```
typedef struct {
   int value; // Giá trị của semaphore
   struct process *L; // Danh sách tiến trình chờ...
} semaphore;
```

23

Cài đặt semaphore theo cấu trúc

```
void wait(semaphore *S)
{
   S->value--;
   if (S->value<0) {
      Thêm tiến trình gọi
      toán tử vào s->L;
      block();
   }
}
```

```
void signal(semaphore *S)
{
   S->value++;
   if (S->value<=0) {
      Xóa một tiến trình P
      ra khỏi s->L;
      wakeup(P);
   }
}
```

24

Semaphore nhị phân

- Là semaphore chỉ nhận giá trị 0 hoặc 1
- Cài đặt semaphore nhị phân đơn giản hơn semaphore không nhị phân (thuật ngữ: counting semaphore)

25

Một số bài toán đồng bộ hóa cơ bản

26

Bài toán vùng đệm có giới hạn

- Đã xét ở ví dụ đầu tiên (the bounded-buffer problem)
- Ta sử dụng 3 semaphore tên là *full*, *empty* và *mutex* để giải quyết bài toán này
- Khởi tạo:
  - *full*: Số lượng phần tử buffer đã có dữ liệu (0)
  - *empty*: Số lượng phần tử buffer chưa có dữ liệu (*n*)
  - *mutex*: 1 (Chưa có tiến trình nào thực hiện đoạn mã găng)

27

Bài toán vùng đệm có giới hạn

Tiến trình ghi *P*:

```
do {
   wait(empty);
   wait(mutex);
   // Ghi một phần tử mới
   // vào buffer
   signal(mutex);
   signal(full);
} while (TRUE);
```

Tiến trình đọc *Q*:

```
do {
   wait(full);
   wait(mutex);
   // Đọc một phần tử ra
   // khỏi buffer
   signal(mutex);
   signal(empty);
} while (TRUE);
```

28

Bài toán tiến trình đọc - ghi

- Thuật ngữ: the reader-writer problem
- Tình huống: Nhiều tiến trình cùng thao tác trên một cơ sở dữ liệu trong đó
  - Một vài tiến trình chỉ đọc dữ liệu (ký hiệu: reader)
  - Một số tiến trình vừa đọc vừa ghi (ký hiệu: writer)
- Khi có đọc/ghi đồng thời của nhiều tiến trình trên cùng một cơ sở dữ liệu, có 2 bài toán:
  - Bài toán 1: reader không phải chờ, trừ khi writer đã được phép ghi vào CSDL (hay các reader không loại trừ lẫn nhau khi đọc)
  - Bài toán 2: Khi writer đã sẵn sàng ghi, nó sẽ được ghi trong thời gian sớm nhất (nói cách khác khi writer đã sẵn sàng, không cho phép các reader đọc dữ liệu)

29

Bài toán tiến trình đọc-ghi số 1

- Sử dụng các semaphore với giá trị khởi tạo: *wrt* (1), *mutex* (1)
- Sử dụng biến *rcount* (khởi tạo 0) để đếm số lượng reader đang đọc dữ liệu
- *wrt*: Đảm bảo loại trừ lẫn nhau khi writer ghi
- *mutex*: Đảm bảo loại trữ lẫn nhau khi cập nhật biến *rcount*

30

Bài toán tiến trình đọc-ghi số 1

- **Tiến trình writer P_w:**

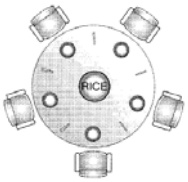
```
do {
    wait(wrt);
    // Thao tác ghi đang được
    // thực hiện
    signal(wrt);
while (TRUE);
```

- **Tiến trình reader P_r:**

```
do {
    wait(mutex);
    rcount++;
    if (rcount==1) wait(wrt);
    signal(mutex);
    // Thực hiện phép đọc
    wait(mutex);
    rcount--;
    if (rcount==0) signal(wrt);
    signal(mutex);
} while (TRUE);
```


Bữa ăn tối của các triết gia

- Thuật ngữ: the dining-philosophers problem
- Có 5 triết gia, 5 chiếc đũa, 5 bát cơm và một âu cơm bố trí như hình vẽ
- Đây là bài toán cổ điển và là ví dụ minh họa cho một lớp nhiều bài toán tương tranh (concurrency): *Nhiều tiến trình khai thác nhiều tài nguyên chung*



Bữa ăn tối của các triết gia

- Các triết gia chỉ làm 2 việc: Ăn và suy nghĩ
  - Suy nghĩ: Không ảnh hưởng đến các triết gia khác, đũa, bát và âu cơm
  - Để ăn: Mỗi triết gia phải có đủ 2 chiếc đũa gần nhất ở bên phải và bên trái mình; chỉ được lấy 1 chiếc đũa một lần và không được phép lấy đũa từ tay triết gia khác
  - Khi ăn xong: Triết gia bỏ cả hai chiếc đũa xuống bàn và tiếp tục suy nghĩ

Giải pháp cho bài toán Bữa ăn...

- Biểu diễn 5 chiếc đũa qua mảng semaphore:
  semaphore chopstick[5];
  các semaphore được khởi tạo giá trị 1
- Mã lệnh của triết gia như hình bên
- Mã lệnh này có thể gây bế tắc (deadlock) nếu cả 5 triết gia đều lấy được 1 chiếc đũa và chờ để lấy chiếc còn lại nhưng không bao giờ lấy được!!

- Mã lệnh của triết gia *i*:

```
do {
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5];
    // Ăn...
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5];
    // Suy nghĩ...
} while (TRUE);
```


Một số giải pháp tránh bế tắc

- Chỉ cho phép nhiều nhất 4 triết gia đồng thời lấy đũa, dẫn đến có ít nhất 1 triết gia lấy được 2 chiếc đũa
- Chỉ cho phép lấy đũa khi cả hai chiếc đũa bên phải và bên trái đều nằm trên bàn
- Sử dụng giải pháp bất đối xứng: Triết gia mang số lẻ lấy chiếc đũa đầu tiên ở bên trái, sau đó chiếc đũa ở bên phải; triết gia mang số chẵn lấy chiếc đũa đầu tiên ở bên phải, sau đó lấy chiếc đũa bên trái

Hạn chế của semaphore

- Mặc dù semaphore cho ta cơ chế đồng bộ hóa tiện lợi song sử dụng semaphore không đúng cách có thể dẫn đến bế tắc hoặc lỗi do trình tự thực hiện của các tiến trình
- Trong một số trường hợp: khó phát hiện bế tắc hoặc lỗi do trình tự thực hiện khi sử dụng semaphore không đúng cách
- Sử dụng không đúng cách gây ra bởi *lỗi lập trình* hoặc do *người lập trình không cộng tác*

Ví dụ hạn chế của semaphore (1)

- Xét ví dụ về đoạn mã găng:

Mã đúng:	Mã sai:

...
wait(mutex);
// Đoạn mã găng
signal(mutex);
...
	...
	signal(mutex);
	// Đoạn mã găng
	wait(mutex);
	...
	• Đoạn mã sai này gây ra vi phạm điều kiện loại trừ lẫn nhau

37

Ví dụ hạn chế của semaphore (2)

- Xét ví dụ về đoạn mã găng:

Mã đúng:	Mã sai:

...
wait(mutex);
// Đoạn mã găng
signal(mutex);
...
	...
	wait(mutex);
	// Đoạn mã găng
	wait(mutex);
	...
	• Đoạn mã sai này gây ra bế tắc

38

Ví dụ hạn chế của semaphore (3)

- Nếu người lập trình quên các toán tử wait() hoặc signal() trong trong các đoạn mã găng, hoặc cả hai thì có thể gây ra:
  - Bế tắc
  - Vi phạm điều kiện loại trừ lẫn nhau

39

Ví dụ hạn chế của semaphore (4)

Tiến trình P_1	Tiến trình P_2

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

- Hai tiến trình P_1 và P_2 đồng thời thực hiện sẽ dẫn tới bế tắc

40

Cơ chế monitor

41

Thông tin tham khảo

- Per Brinch Hansen (người Đan Mạch) là người đầu tiên đưa ra khái niệm và cài đặt monitor năm 1972
- Monitor được sử dụng lần đầu tiên trong ngôn ngữ lập trình Concurrent Pascal

Per Brinch Hansen
(1938-2007)

42

Monitor là gì?

- Thuật ngữ monitor: *giám sát*
- Định nghĩa không hình thức: Là một loại construct trong ngôn ngữ bậc cao dùng để phục vụ các thao tác đồng bộ hóa
- Monitor được nghiên cứu, phát triển để khắc phục các hạn chế của semaphore như đã nêu trên

43

Định nghĩa tổng quát

- Monitor là một cách tiếp cận để đồng bộ hóa các tác vụ trên máy tính khi phải sử dụng các tài nguyên chung. Monitor thường gồm có:
  - Tập các procedure thao tác trên tài nguyên chung
  - Khóa loại trừ lẫn nhau
  - Các biến tương ứng với các tài nguyên chung
  - Một số các giả định bất biến nhằm tránh các tình huống tương tranh
- Trong bài này: Nghiên cứu một loại cấu trúc monitor: Kiểu monitor (monitor type)

44

Monitor type

- Một kiểu (type) hoặc kiểu trừu tượng (abstract type) gồm có các dữ liệu *private* và các phương thức *public*
- Monitor type được đặc trưng bởi tập các toán tử của người sử dụng định nghĩa
- Monitor type có các biến xác định các trạng thái; mã lệnh của các procedure thao tác trên các biến này

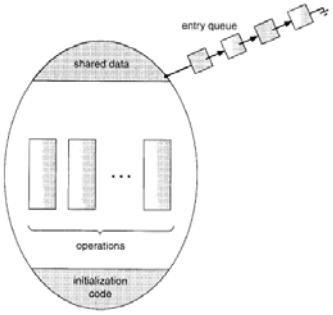
45

Cấu trúc một monitor type

```
monitor tên_monitor {
   // Khai báo các biến chung
   procedure P1(...) { ...
   }
   procedure P2(...) { ...
   }
   ...
   procedure Pn(...) { ...
   }
   initialization_code (..) { ...
   }
}
```

46

Minh họa cấu trúc monitor



47

Cách sử dụng monitor

- Monitor được cài đặt sao cho *chỉ có một tiến trình được hoạt động trong monitor* (*loại trừ lẫn nhau*). Người lập trình không cần viết mã lệnh để đảm bảo điều này
- Monitor như định nghĩa trên chưa đủ mạnh để xử lý mọi trường hợp đồng bộ hóa. Cần thêm một số cơ chế "tailor-made" về đồng bộ hóa
- Các trường hợp đồng bộ hóa "tailor-made": sử dụng kiểu *condition*.

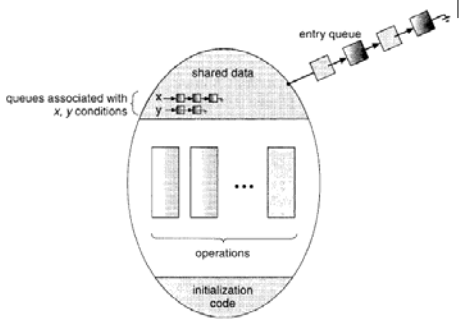
48

Kiểu condition

- Khai báo:
  condition x, y; // x, y là các biến kiểu condition
- Sử dụng kiểu condition: Chỉ có 2 toán tử là wait và signal
  - x.wait(): tiến trình gọi đến x.wait() sẽ được chuyển sang trạng thái chờ (wait hoặc suspend)
  - x.signal(): tiến trình gọi đến x.signal() sẽ khôi phục việc thực hiện (wakeup) một tiến trình đã gọi đến x.wait()

49

Monitor có kiểu condition



50

Đặc điểm của x.signal()

- x.signal() chỉ đánh thức duy nhất một tiến trình đang chờ
- Nếu không có tiến trình chờ, x.signal() không có tác dụng gì
- x.signal() khác với signal trong semaphore cổ điển: signal cổ điển luôn làm thay đổi trạng thái (giá trị) của semaphore

51

Signal wait/continue

- Giả sử có hai tiến trình P và Q:
  - *Q* gọi đến x.wait(), sau đó *P* gọi đến x.signal()
  - *Q* được phép tiếp tục thực hiện (wakeup)
- Khi đó *P* phải vào trạng thái wait vì nếu ngược lại thì *P* và *Q* cùng thực hiện trong monitor
- Khả năng xảy ra:
  - Signal-and-wait: *P* chờ đến khi *Q* rời monitor hoặc chờ một điều kiện khác (*)
  - Signal-and-continue: *Q* chờ đến khi *P* rời monitor hoặc chờ một điều kiện khác

52

Bài toán Ăn tối.. với monitor

- Giải quyết bài toán Ăn tối của các triết gia với monitor để không xảy ra bế tắc khi hai triết gia ngồi cạnh nhau cùng lấy đũa để ăn
- Trạng thái của các triết gia:
  enum {thinking, hungry, eating} state[5];
- Triết gia *i* chỉ có thể ăn nếu cả hai người ngồi cạnh ông ta không ăn:
  (state[(i+4)%5]!=eating) and (state[(i+1)%5]!=eating)
- Khi triết gia *i* không đủ điều kiện để ăn: cần có biến condition: condition self[5];

53

Monitor của bài toán Ăn tối...

```
monitor dp {
   enum {thinking, hungry, eating} state[5];
   condition self[5];
   void pickup(int i) {
        state[i] = hungry;
        test(i);
        if (state[i] != eating) self[i].wait();
   }
}
```

54

Monitor của bài toán Ăn tối...

```
void putdown(int i) {
    state[i] = thinking;
    test((i+4)%5);
    test((i+1)%5);
}
initialization_code() {
    for (int i=0;i<5;i++) state[i] = thinking;
}
```

55

Monitor của bài toán Ăn tối...

```
void test(int i) {
    if ((state[(i+4)%5] != eating) &&
      (state[i] == hungry) &&
      (state[(i+1)%5] != eating)) {
        state[i] = eating;
        self[i].signal();
    }
}
```

56

Đọc thêm ở nhà

- Khái niệm về miền găng (critical region)
- Cơ chế monitor của Java:

```
public class XYZ {
    ...
    public synchronized void safeMethod() {
      ...
    }
}
```

- Toán tử wait() và notify() trong java.util.package (tương tự toán tử wait() và signal())
- Cách cài đặt monitor bằng semaphore

57

Tóm tắt

- Khái niệm đồng bộ hóa
- Khái niệm đoạn mã găng, ba điều kiện của đoạn mã găng
- Khái niệm semaphore, semaphore nhị phân
- Hiện tượng bế tắc do sử dụng sai semaphore
- Một số bài toán cổ điển trong đồng bộ hóa
- Miền găng
- Cơ chế monitor

58

Bài tập

- Chỉ ra điều kiện nào của đoạn mã găng bị vi phạm trong đoạn mã găng sau của P_i:

```
do {
   while (turn != i) ;
   CS_i;
   turn = j;
   REMAIN_i;
} while (1);
```

59

Bài tập

- Cài đặt giải pháp cho bài toán Bữa ăn tối của các triết gia trong Java bằng cách sử dụng synchronized, wait() và notify()
- Giải pháp monitor cho bài toán Bữa ăn tối... tránh được bế tắc, nhưng có thể xảy ra trường hợp tất cả các triết gia đều không được ăn. Hãy chỉ ra trường hợp này và tìm cách giải quyết bằng cơ chế monitor
- Chú ý: Sinh viên cần làm bài tập để hiểu tốt hơn về đồng bộ hóa

60