MÔN HỆ ĐIỀU HÀNH

Chương 3

TƯƠNG TRANH GIỮA CÁC PROCESS

3.1 Giới thiệu về tương tranh
3.2 Loại trừ tương hỗ giữa các đoạn code CS
3.3 Các phương pháp dừng chờ chủ động (busy waiting)
3.4 Đồng bộ các process : Bài toán Sản xuất-Tiêu dùng
3.5 Các phương pháp dừng chờ thụ động (sleep-wakeup)
3.6 Các bài toán IPC kinh điển và giải quyết

Tài liệu tham khảo : chương 2, sách "Modern Operating Systems", Andrew S. Tanenbaum: , 2nd ed, Prentice Hall

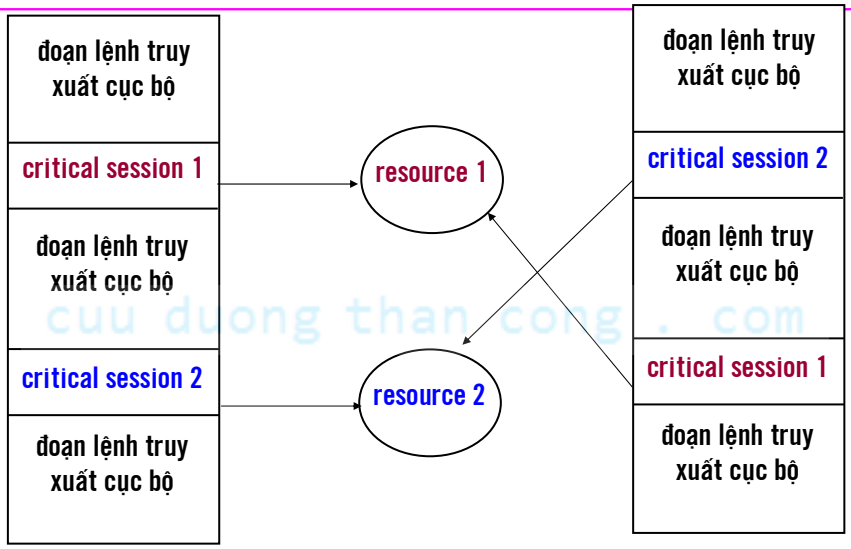
3.1 Giới thiệu về tương tranh

- Trong hệ đa chương, thường có nhiều process chạy song hành, nhưng mỗi process có không gian làm việc độc lập, không ai có thể truy xuất trực tiếp không gian làm việc của process khác => rất tốt cho việc bảo vệ chúng lẫn nhau nhất là khi các process này là những chương trình độc lập.
- Nếu 2 hay nhiều process cần giao tiếp nhau để đồng bộ hay để trao đổi dữ liệu, ta cần cung cấp cơ chế cho chúng. Có 2 cơ chế giao tiếp chính giữa các process : truy xuất bộ nhớ dùng chung và gởi/nhận thông báo.
- Truy xuất bộ nhớ chung là 1 trong nhiều hoạt động tương tranh giữa các process. Vấn đề tương tranh trên 1 tài nguyên dùng chung là vấn đề lớn cần phải giải quyết triệt để vì nếu nhiều process truy xuất đồng thời vào 1 tài nguyên dùng chung mà không có sự kiểm soát thì dễ xảy ra lỗi làm hư hỏng tài nguyên (điều kiện Race).

Giới thiệu về tương tranh

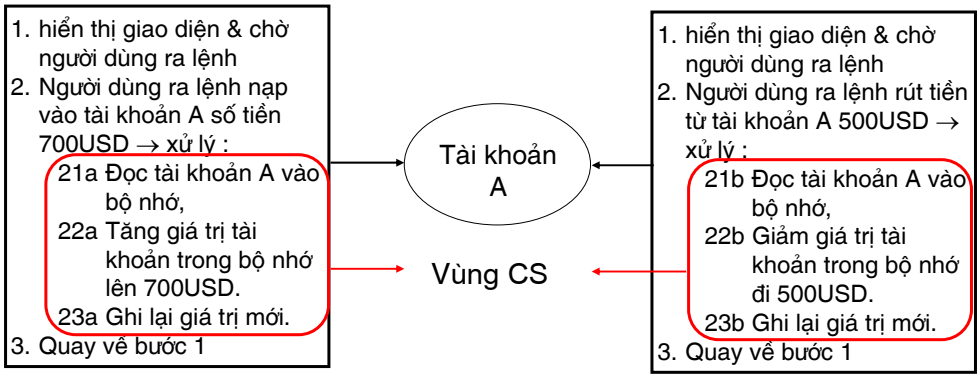
- Phân tích kỹ code của chương trình, ta nhận thấy chúng là danh sách liên tiếp của 2 loại đoạn code :
- đoạn code truy xuất các biến cục bộ của chương trình. Đoạn code này thường dài và xuất hiện nhiều. May mắn là chúng ta không cần quan tâm và kiểm soát đoạn code này.
- Đoạn code truy xuất tài nguyên dùng chung và có thể tranh chấp với process khác. Đây là đoạn code, mặc dù ít xuất hiện và thường rất ngắn, nhưng dễ gây lỗi trên tài nguyên nên ta gọi nó là 'critical session' (viết tắt là CS), chúng ta cần kiểm soát cẩn thận đoạn code CS này.

Giới thiệu về tương tranh



Giới thiệu về tương tranh

Thí dụ 2 ứng dụng truy xuất tài khoản A đồng thời :



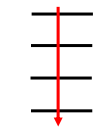
Nếu tài khoản A là 1000USD và HĐH điều khiển chạy 2 process P1 và P2 theo thứ tự 21a→22a→21b→22b→23b→23a thì kết quả tài khoản A sẽ là 1700USD (giá trị đúng là 1200USD).

3.2 Loại trừ tương hỗ giữa các đoạn CS

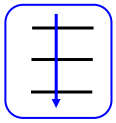
- Để kiểm soát việc truy xuất tài nguyên đồng thời giữa nhiều process, ta không cho phép hơn 1 vùng CS của các process đó cùng truy xuất tài nguyên xác định tại từng thời điểm. Phương pháp này được gọi là loại trừ tương hỗ giữa các đoạn CS (Mutual Exclusion).
- Có nhiều phương pháp loại trừ tương hỗ cụ thể khác nhau và được chia làm 2 nhóm chính : nhóm các phương pháp dừng chờ chủ động (busy waiting) và nhóm các phương pháp dừng chờ thụ động (sleep/wakeup).
- Tinh thần của nhóm phương pháp dừng chờ chủ động là khi 1 process cần thực hiện đoạn code CS tương tranh với process khác thì nó phải dừng chờ đến khi process khác chạy xong đoạn code CS này, trong quá trình dừng chờ, nó vẫn chiếm CPU liên tục để dò điều kiện chạy tiếp liên tục (nhưng không thành công).
- Tinh thần của nhóm phương pháp sleep/wakeup là khi phải dừng chờ process khác, nó xin ngủ (trả CPU) và nằm bất động. Khi process khác thực hiện xong vùng CS, hệ thống sẽ đánh thức process ngủ để bắt đầu thực hiện đoạn lệnh CS…

Loại trừ tương hỗ giữa các đoạn CS

Mỗi lần muốn vào vùng CS, ta phải gọi hàm In_Control() để kiểm soát việc thi hành vùng CS, khi hoàn thành vùng CS, ta phải gọi hàm Out_Control() để thông báo cho các process khác đang chờ để chúng kiểm tra lại việc đi vào.

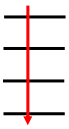


In_Control();



Vùng CS truy xuất tài nguyên dùng chung

Out_Control();



3.3 Các phương pháp chờ chủ động

1. Phương pháp dựa trên Interrupt

- Tính chất cơ bản của CPU là sau khi thi hành 1 lệnh máy, nó sẽ tự động thi hành lệnh máy kế tiếp,… mà không để ý bất kỳ thứ gì bên ngoài.
- Tuy nhiên, nếu chỉ vậy, CPU sẽ không bao giờ đáp ứng kịp thời 1 sự kiện nào đó xảy ra trong quá trình thi hành chương trình của CPU. Do đó, người ta phải tạo thêm 1 chân nhập có tên là IRQ. Một thiết bị nào đó nếu muốn yêu cầu CPU xử lý dùm công việc nào thì hãy tạo tín hiệu (ngắt) gởi về CPU trên chân IRQ. Bình thường, mỗi khi CPU thấy có tín hiệu trên chân IRQ, nó sẽ dừng tạm thời chương trình đang chạy hiện hành, chạy đoạn lệnh qui định trước (trình phục vụ ngắt) để xử lý công việc dùm cho thiết bị yêu cầu. Sau khi hoàn thành trình phục vụ ngắt, CPU sẽ quay về tiếp tục thi hành chương trình từ vị trí ngừng trước đây.

3.3 Các phương pháp chờ chủ động

1. Phương pháp dựa trên Interrupt

Tuy nhiên, nếu CPU đang thi hành chương trình real-time hay nhạy cảm với thời gian, nó cần có khả năng phớt lờ yêu cầu ngắt. Để giúp chương trình quyết định lúc nào cho phép CPU đáp ứng ngắt, lúc nào phớt lờ, người ta cung cấp 2 lệnh máy sau :

Cli : clear Interrupt : cấm CPU được phục vụ ngắt

Sti : set interrupt : cho phép CPU được đáp ứng ngắt.

Như vậy, nếu ta viết :

In_Control() ≡ cli

Out_Control() ≡ sti

thì tại từng thời điểm chỉ có tối đa 1 process chạy được vùng CS, vì không có process nào khác (kể cả trình lập lịch) có thể ngắt nó tạm thời.

3.3 Các phương pháp chờ chủ động

Phương pháp dựa trên ngắt quảng rất đơn giản, hiệu quả nhưng nếu lập trình sai thì nguy hiểm khôn lường. Thí dụ nếu ta quên viết lệnh sti sau khi hoàn thành vùng CS thì process tương ứng sẽ tiếp tục chạy đến hết chương trình rồi treo máy vì không process nào khác, kể cả trình lập lịch, có thể chiếm được CPU để chạy.

Hiện nay, chỉ có HĐH mới được phép dùng phương pháp này, chương trình ứng dụng không được phép dùng.

Hơn nữa, phương pháp dựa trên ngắt chỉ tác động trên 1 CPU, nếu máy có nhiều CPU thì các CPU khác không bị ảnh hưởng bởi lệnh cấm ngắt trên 1 CPU nào đó.

3.3 Các phương pháp chờ chủ động

2. Phương pháp dùng biến khóa

Mỗi vùng CS được bảo vệ bởi 1 biến khóa, biến này lúc đầu = 0 để xác định rằng chưa process nào thi hành vùng CS.

Mỗi lần muốn thi hành vùng CS, process sẽ kiểm tra biến khóa, nếu nó = 0 thì set lên 1 và tiếp tục thi hành vùng CS đến khi hoàn thành sẽ set lại biến khóa = 0. Trường hợp biến khóa = 1 thì phải chờ process khác thi hành xong vùng CS.

```
Bool process_in_CS = 0;
In_Control() {
    for (;;) if (process_in_CS ==0) break;
    process_in_CS = 1;
}
```

```
Out_Control() {
    process_in_CS = 0;
}
```


3.3 Các phương pháp chờ chủ động

2. Phương pháp dùng biến khóa

Về ý tưởng thì phương pháp dùng biến khóa giải quyết tốt vấn đề tranh chấp tài nguyên dùng chung, nhưng nếu hiện thực bằng đoạn lệnh C như slide trước thì có thể thất bại trong 1 số tình huống.

Giả sử process P1 muốn thi hành vùng CS, nó kiểm tra biến process_in_CS và thấy đang mở (=0). Ngay lúc này, process hết khe thời gian, trình lập lịch dừng nó và chọn process P2 chạy tiếp, nếu P2 cũng muốn thi hành vùng CS, nó kiểm tra biến khóa, lúc này biến khóa vẫn = 0 nên P2 sẽ set lên 1 rồi thi hành vùng CS. Trong lúc thi hành CS, P2 hết khe thời gian và CPU được giao lại P1. P1 chạy tiếp và cũng set biến khóa lên 1 rồi vào vùng CS. Như vậy lúc này 2 process P1 và P2 đang tranh chấp tài nguyên dùng chung!

3.3 Các phương pháp chờ chủ động

3. Phương pháp dùng lệnh TSL

Phân tích lỗi của phương pháp dùng biến khóa, ta nhận thấy nếu 2 lệnh kiểm tra biến khóa và set nó lên 1 được đảm bảo thi hành theo cơ chế nguyên tử, không chia cắt (hoặc thực hiện cả hai, hoặc không thi hành lệnh nào) thì lỗi của slide trước không thể xảy ra, nghĩa là phương pháp dùng biến khóa sẽ chạy đúng.

Để đảm bảo được ý tưởng trên, về mặt phần cứng, hãng chế tạo CPU sẽ cung cấp thêm 1 lệnh máy đặc biệt có tên là TSL (Test-and-Set) có dạng sau :

TSL al, process_in_CS

≡

al ← process_in_CS

process_in_CS ← 1

3.3 Các phương pháp chờ chủ động

3. Phương pháp dùng lệnh TSL

Nhờ đặc tính của lệnh máy TSL, ta viết lại 2 hàm In_Control() và Out_Control() của phương pháp dùng biến khóa như sau :

```
Bool process_in_CS = 0;
Out_Control() {
    process_in_CS = 0;
}
```

```
In_Control() {
    for (;;) {
        TSL al, process_in_CS;
        if (al ==0) break;
    }
}
```


Phân tích đoạn code trên, ta thấy process nào thực hiện được lệnh TSL đầu tiên thì nó đã set biến khóa lên 1 nên nếu ngay sau đó có process khác chạy In_Control() thì phải chờ process ban đầu, chứ không thể vào vùng CS trước được.

3.3 Các phương pháp chờ chủ động

4. Phương pháp luân phiên

Ý tưởng của phương pháp này là cho các process luân phiên thi hành vùng CS, từng thời điểm chỉ 1 process được thi hành CS.

Giả sử có N process cần thi hành CS được đánh số từ 0 đến N-1.

Tạo 1 biến "turn" chứa chỉ số process được phép thi hành CS tại từng thời điểm. Lúc đầu turn được set = 0.

```
In_control (int idproc) {
   while (turn != idproc) ;     // chờ đến lượt mình
}
Out_control (int idproc) {
        turn = (turn +1)%N;  // cho process đi ngay sau mình vào
}
```


3.3 Các phương pháp chờ chủ động

4. Phương pháp luân phiên

Về lý thuyết, phương pháp luân phiên tạo được sự bình đẳng tuyệt đối giữa các process về việc thi hành vùng CS, process nào cũng được thi hành CS với cơ hội ngang nhau, không process nào có thể thi hành CS nhiều lần hơn các process khác.

Tuy nhiên trong thực tế, các process thường có độ phức tạp khác nhau, có nhu cầu chạy vùng CS rất khác nhau, process này cần chạy CS nhiều lần, process khác chỉ cần chạy CS ít lần. Như vậy, process cần chạy CS ít lần hơn sẽ ngăn cản process cần chạy CS nhiều lần hơn.

3.3 Các phương pháp chờ chủ động

5. Phương pháp Peterson

Để khắc phục vấn đề của phương pháp luân phiên, Peterson đã nâng cấp thuật giải In_Control() và Out_Control() như sau :

```
#define FALSE       0
#define TRUE        1
#define N           2
int turn = 0;           // chỉ số process được phép thi hành vùng CS
int interested[N];      // dãy ghi nhận ý muốn các process, ban đầu = 0
void In_control (int idproc) {
int other = 1 - idproc;
   interested[idproc] = TRUE;          // khai báo ý muốn vào CS
   turn = idproc;                      // khẳng định ý muốn vào CS
   while (interested[other]==TRUE && idproc!=turn) ; //chờ
}
void Out_control(int idproc) {
        interested[idproc] = FALSE;
}
```


3.3 Các phương pháp chờ chủ động

5. Phương pháp Peterson

Phân tích hàm In_Control(), ta thấy nếu 2 process cùng muốn vào vùng CS 1 lượt thì process nào xin vào trước sẽ thắng còn process xin vào sau sẽ phải chờ. Hơn nữa, mỗi process có quyền vào vùng CS mà không cần chờ đến lượt mình như trước đây.

Lưu ý đoạn code ở slide trước chỉ đúng cho trường hợp 2 process. Nếu có nhiều process cần truy xuất vùng CS, ta phải hiệu chỉnh lại đoạn code phức tạp hơn nhiều mới giải quyết được vấn đề.

3.4 Đồng bộ các process : Bài toán Sản xuất-Tiêu dùng

Trong hệ thống có 2 loại phần tử :

- Sản xuất chuyên tạo sản phẩm mới và để vào kho chứa.
- Tiêu dùng chuyên lấy sản phẩm từ kho chứa ra để sử dụng.

Bài toán Sản xuất -Tiêu dùng trên có 2 vấn đề cần giải quyết :

- làm sao để phần tử Sản xuất và Tiêu dùng không được tranh chấp nhau khi truy xuất kho chứa sản phẩm.
- làm sao để đồng bộ tốc độ thi hành của 2 phần tử để chúng có thể hoạt động tốt theo thời gian, không gây khủng hoảng thừa hay khủng hoảng thiếu.

Ý tưởng giải quyết Bài toán Sản xuất -Tiêu dùng

```
#define N   100                     /* number of slots in the buffer */
int count = 0;                      /* number of items in the buffer */

void producer(void)
{
  while (TRUE) {                    /* repeat forever */
       produce_item();              /* generate next item */
       if (count == N) sleep();     /* if buffer is full, go to sleep */
       enter_item();                /* put item in buffer */
       count = count + 1;           /* increment count of items in buffer */
       if (count == 1) wakeup(consumer);/* was buffer empty? */
  }
}
```

Ý tưởng giải quyết Bài toán Sản xuất -Tiêu dùng

```
void consumer(void)
{
  while (TRUE) {                    /* repeat forever */
        if (count == 0) sleep();    /* if buffer is empty, got to sleep */
        remove_item();              /* take item out of buffer */
        count = count – 1;          /* decrement count of items in buffer */
        if (count == N–1) wakeup(producer);/* was buffer full? */
        consume_item();             /* print item */
  }
}
```


Ý tưởng giải quyết Bài toán Sản xuất -Tiêu dùng

Ý tưởng của đoạn code trong 2 slide trước là :

- kiểm tra điều kiện chạy tiếp cho process Sản xuất và Tiêu dùng. Nếu không thể chạy tiếp (kho đầy/rỗng) thì gọi hàm sleep() để dừng chạy đến lúc được process khác đánh thức.
- Khi cần thiết, process sẽ gọi hàm wakeup() để đánh thức dùm process khác dậy để nó tiếp tục chạy từ lúc ngủ trước đây.

Mặc dù ý tưởng trên giải quyết được vấn đề đồng bộ giữa các process, nhưng nếu lập trình bằng ngôn ngữ C như slide trước thì có thể gây lỗi.

Ý tưởng giải quyết Bài toán Sản xuất -Tiêu dùng

Cụ thể nếu Producer kiểm tra kho chứa đầy, tính gọi hàm sleep() để ngủ thì hết khe thời gian chạy. Trình lập lịch sẽ dừng tạm Producer, chọn Consumer chạy. Consumer kiểm tra kho chứa, lấy được sản phẩm và đánh thức Producer dậy. Tuy nhiên việc đánh thức này không có giá trị vì Producer chứa ngủ.

Sau đó producer chạy tiếp, nó sẽ gọi sleep() để ngủ và sẽ không bao giờ thức dậy vì consumer không bao giờ đánh thức nó nữa.

Riêng Consumer chạy tiếp và theo thời gian, nó sẽ lấy hết sản phẩm trong kho chứa, khi đó nó sẽ gọi sleep() để ngủ chờ Producer đánh thức nhưng sẽ không bao giờ thức dậy được vì Producer cũng đã và đang ngủ như Consumer. Hiện tượng này được gọi là Deadlock và ta sẽ giới thiệu các phương pháp khác nhau để giải quyết trong chương 4.

3.5 Các phương pháp sleep/wakeup

1. Phương pháp dùng Semaphore

Semaphore là đối tượng được cung cấp sẵn bởi hệ thống, đối tượng này gồm :

- 1 thuộc tính chứa giá trị nguyên dương, ta gọi là biến semaphore s.
- hàm down(s) có chức năng giảm s 1 đơn vị, nếu giảm không được thì phải chờ đến khi có điều kiện giảm được thì làm lại. Thời gian thực hiện hàm down có thể rất dài, nhưng các process khác không thể thấy được trạng thái trung gian của hàm down này. Nói cách khác việc thi hành hàm down có tính nguyên tử, không chia cắt được.
- hàm up(s) có chức năng tăng s 1 đơn vị, nếu sau khi tăng mà s = 1 thì phải đánh thức các process đang ngủ vì đã thực hiện down(s) trước đây mà chưa được. Thời gian thực hiện hàm up rất nhanh, việc thi hành hàm up cũng có tính nguyên tử, không chia cắt được.

3.5 Các phương pháp sleep/wakeup

Ta có thể dùng Semaphore để giải quyết tương tranh giữa nhiều process như sau :

- kết hợp 1 semaphore nhị phân với vùng CS tương ứng. Semaphore này sẽ được gán trị đầu là 1 và sau này nó chỉ có thể chứa 2 trị : hoặc 0 hoặc 1. Ta gọi semaphore này là semaphore nhị phân.
- hàm In_Control() để kiểm soát vào vùng CS của process sẽ là lời gọi hàm down (s).
- hàm Out_Control() để kiểm soát ra vùng CS của process sẽ là lời gọi hàm up (s).

Như vậy, tại 1 thời điểm chỉ có 1 process down(s) được và vào vùng CS, các process khác nếu down(s) đều bị thất bại và phải ngủ chờ. Khi process đầu tiên thực hiện xong CS, nó thực hiện up(s) và sẽ đánh thức các process ngủ dậy. Trong các process dậy này, chỉ có 1 process thực hiện thành công lệnh down(s) để vào vùng CS...

3.5 Các phương pháp sleep/wakeup

//dùng Semaphore giải quyết bài toán Sản xuất – Tiêu dùng

```
#define N 100                       /* number of slots in the buffer */
typedef int semaphore;              /* semaphores are a special kind of int */
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N;                /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */

void producer(void)
{
  int item;

  while (TRUE) {                    /* TRUE is the constant 1 */
        produce_item(&item);        /* generate something to put in buffer */
        down(&empty);               /* decrement empty count */
        down(&mutex);               /* enter critical region */
        enter_item(item);           /* put new item in buffer */
        up(&mutex);                 /* leave critical region */
        up(&full);                  /* increment count of full slots */
  }
}
```

3.5 Các phương pháp sleep/wakeup

//dùng Semaphore giải quyết bài toán Sản xuất – Tiêu dùng

```
void consumer(void)
{
  int item;

  while (TRUE) {                  /* infinite loop */
        down(&full);              /* decrement full count */
        down(&mutex);             /* enter critical region */
        remove_item(&item);       /* take item from buffer */
        up(&mutex);               /* leave critical region */
        up(&empty);               /* increment count of empty slots */
        consume_item(item);       /* do something with the item */
  }
}
```


3.5 Các phương pháp sleep/wakeup

Phân tích code dùng semaphore để giải quyết đồng thời 2 vấn đề tương tranh và đồng bộ process trong bài toán Sản xuất – Tiêu dùng trong slide trước, ta thấy việc dùng semaphore khá gọn nhẹ và hiệu quả. Tuy nhiên khuyết điểm của việc dùng semaphore là độ an toàn không cao. Cụ thể nếu ta hoán vị 2 lệnh down(&full) và down(&mutex) trong process Consumer thì dễ dẫn đến deadlock vì khi hết sản phẩm trong kho chứa, Consumer tới sẽ khóa kho trước rồi mới kiểm tra kho chứa, lúc này kho chứa rỗng nên Consumer sẽ dừng chờ Producer đánh thức, tuy nhiên Producer sẽ phải ngủ chờ Consumer mở kho chứa → Producer và Consumer đã ngủ chờ nhau và cả 2 sẽ ngủ mãi, đây là hiện tượng deadlock mà ta sẽ trình bày trong chương 4.

3.5 Các phương pháp sleep/wakeup

2. Phương pháp dùng Monitor

Monitor là đối tượng được cung cấp sẵn bởi hệ thống. Về cấu trúc, monitor cũng gồm nhiều thuộc tính dữ liệu và nhiều tác vụ chức năng :

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .
        .
        .
        end;
end monitor;
```


3.5 Các phương pháp sleep/wakeup

Sự khác biệt giữa Monitor và 1 đối tượng (module phần mềm) bình thường được thể hiện bởi 2 tính chất sau :

1. trong monitor, ta có thể định nghĩa nhiều biến điều kiện. Biến điều kiện chỉ có tên, không có kiểu và không có giá trị nên ta không dùng nó như biến bình thường. Chỉ có 2 tác vụ định sẵn có thể truy xuất biến điều kiện :
   - hàm wait(cond) : bắt process ngủ chờ trên biến điều kiện cond.
   - hàm signal(cond) : đánh thức các process đang ngủ chờ trên biến cond.
2. Về mặt điều khiển, Monitor chỉ cho phép tối đa 1 process được vào thi hành 1 tác vụ nào đó của Monitor tại từng thời điểm. Nhờ tính chất này, ta dễ dàng giải quyết việc loại trừ tương hỗ giữa các process khi chúng truy xuất đồng thời 1 tài nguyên dùng chung nào đó bằng cách đặt mỗi đoạn CS vào 1 tác vụ riêng và đặt tất cả các tác vụ này trong một Monitor nào đó.

3.5 Các phương pháp sleep/wakeup

//dùng Monitor giải quyết bài toán Sản xuất – Tiêu dùng

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```


3.5 Các phương pháp sleep/wakeup

Code Java dùng Monitor hiện thực bài toán Sản xuất-Tiêu dùng

```
//đặc tả class ứng dụng demo bài toán Sản xuất — Tiêu dùng
public class ProducerConsumer {
   static final int N = 100;                    //kích thước kho chứa
   static producer p = new producer();          //đối tượng thread producer
   static producer c = new consumer();          //đối tượng thread consumer
   static ourMonitor mon = new ourMonitor();    //đối tượng monitor
   //điểm nhập ứng dụng demo
   public static void main(String args[]) {
     p.start();                                 //khởi động thread Producer
     c.start();                                 //khởi động thread Consumer
   }
   //(xem tiếp slide kế)
```

3.5 Các phương pháp sleep/wakeup

Code Java dùng Monitor hiện thực bài toán Sản xuất-Tiêu dùng

```
//class đặc tả thread Producer
static class producer extends Thread {
   public void run() {
   int item;
     while (true) {
       item = produce_item();
       mon.insert (item);
     }
   }
   priavte int produce_item () {...}
}
```


3.5 Các phương pháp sleep/wakeup

Code Java dùng Monitor hiện thực bài toán Sản xuất-Tiêu dùng

```
//class đặc tả thread Consumer
static class consumer extends Thread {
   public void run() {
   int item;
      while (true) {
          item = mon.remove();
          consume_item(item);
      }
   }
   priavte void consume_item (int item) {...}
}
```

3.5 Các phương pháp sleep/wakeup

Code Java dùng Monitor hiện thực bài toán Sản xuất-Tiêu dùng

```
//class đặc tả Monitor
static class our_monitor {
   private int buffer[] = new int[N];          // kho chứa
   private int count = 0, lo = 0, hi = 0;      //các biến quản lý kho chứa

   //hàm thêm sản phẩm vào kho chứa
   public synchronized vois insert (int val) {
      if (count == N) goto_sleep(); //nếu kho đầy thì ngủ
      buffer[hi] = val;                        // để sản phẩm vào vị trí
      hi = (hi+1) % N;                         //tăng vị trí để sản phẩm
      count = count +1;                        //tăng counter sản phẩm
      if (count == 1) notify();                //đánh thức consumer nếu cần
   }
```


3.5 Các phương pháp sleep/wakeup

Code Java dùng Monitor hiện thực bài toán Sản xuất-Tiêu dùng

```
   //hàm lấy sản phẩm từ kho chứa
   public synchronized int remove () {
   int val;
      if (count == 0) goto_sleep();          //nếu kho hết thì ngủ
      val = buffer[lo];                      // lấy sản phẩm từ vị trí
      lo = (lo+1) % N;                       //tăng vị trí lấy sản phẩm
      count = count - 1;                     //giảm counter sản phẩm
      if (count == N-1) notify();            //đánh thức producer nếu cần
      return val;
   }
   private void goto_sleep () {
    try { wait();
    } catch (InterruptedException exc) {}
  }
}
```

3.5 Các phương pháp sleep/wakeup

3. Phương pháp gởi/nhận thông báo

Hệ thống cung cấp 2 hàm chức năng :

- send (proc_id, message) cho phép gởi 1 chuỗi byte tới prcoess proc_id.
- receive (proc_id, message) cho phép chờ nhận chuỗi byte từ process proc_id gởi tới.

Đoạn code C sau sẽ giải quyết bài toán Sản xuất – Tiêu dùng dùng phương pháp gởi/nhận thông báo.

3.5 Các phương pháp sleep/wakeup

3. Phương pháp gởi/nhận thông báo

//Code C hiện thực bài toán Sản xuất-Tiêu dùng

```
#define N 100
void Producer(void) {
int item;
message m;
    while (TRUE) {
        item = produce_item();              //tạo sản phẩm mới
        receive(consumer, &m);              //chờ nhận yêu cầu từ consumer
        build_message(&m, tiem);            //xây dựng thông báo chứa sản phẩm
        send(consumer, &m);                 //gởi đến consumer
    }
}
```

3.5 Các phương pháp sleep/wakeup

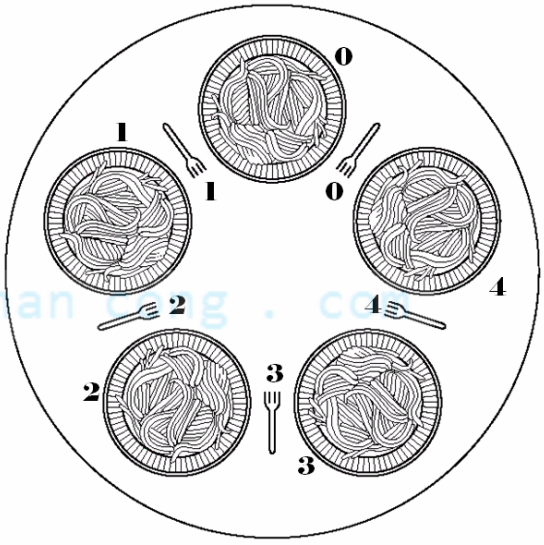
3. Phương pháp gởi/nhận thông báo

```
//Code C hiện thực bài toán Sản xuất-Tiêu dùng
#define N 100
void Consumer(void) {
int item;
message m;
   //gởi n yêu cầu sản xuất tới Producer
   for (i = 0; i <N; i++)
      send (producer, &m);
   while (TRUE) {
      receive(producer, &m);          //chờ nhận sản phẩm từ consumer
      item = extract_item(&m);        //rút trích sản phẩm mới từ thông báo
      send(producer, &m);             //gởi yêu cầu mới đến producer
      consume_item(&tiem);            //tiêu dùng sản phẩm
   }
}
```


3.6 Các bài toán IPC kinh điển

1. Bài toán 5 SV ăn cơm :

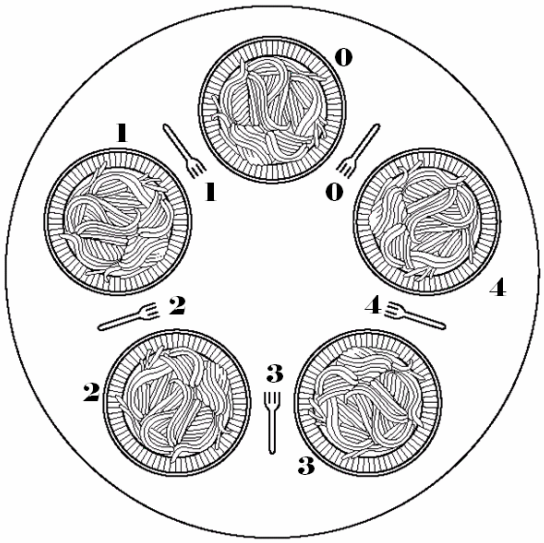
1. bàn ăn có 5 chén cơm dành cho 5 SV, đánh số thứ tự từ 0 - 4.
2. do thiếu đủa, chỉ có 5 chiếc được để xen kẻ giữa các chén cơm, đánh số thứ tự từ 0 -4.
3. mỗi SV ăn cơm cần 1 đôi đủa. Vì lịch sự, mỗi SV chỉ được phép lấy 2 chiếc kề chén cơm của mình (chiếc bên trái và chiếc bên phải).



3.6 Các bài toán IPC kinh điển

1. Bài toán 5 SV ăn cơm :

4. hãy tìm qui trình ăn cơm cho mỗi SV sao cho các điều kiện sau được thỏa :
   - các SV ăn cơm được càng đồng thời càng tốt (hiệu quả nhất).
   - các SV không được tranh chấp nhau trong việc lấy đôi đủa.
   - các SV không bị deadlock trong quá trình chờ lấy đủa.



3.6 Các bài toán IPC kinh điển

1. Bài toán 5 SV ăn cơm :

Phân tích hoạt động ăn cơm của SV :

- quá trình ăn cơm không đòi hỏi phải sở hữu liên tục đôi đủa.
- quá trình ăn cơm là hoạt động lặp, mỗi chu kỳ gồm 3 công đoạn :
  1. đang nhai cơm hay nói chuyện. (THINKING)
  2. cố gắng chiếm hữu đôi đủa. (HUNGRY)
  3. và cơm và gắp thức ăn rồi để đủa xuống bàn. (EATING)
- trong đó công đoạn 1 tốn nhiều thời gian nhất nhưng may mắn là trong công đoạn này, SV không cần đủa (tài nguyên). Còn bước 2 và 3 thì rất ngắn và cần loại trừ tương hỗ giữa các SV.

3.6 Các bài toán IPC kinh điển

1. Bài toán 5 SV ăn cơm :

Phát họa qui trình ăn cơm cho mỗi SV :

```
#define N  5
void Sinhvien (int i) {
   while (còn cơm) {
      think();                    //công đoạn 1
      take_fork(i);               //công đoạn 2, có thể bị kẹt 1 thời gian
      take_fork((i+1)%N);
      eat();
      put_fork(i);                //công đoạn 3 thường rất nhanh
      put_fork((i+1)%N);
   }
}
```

Qui trình ăn ở trên dễ bị deadlock, thí dụ như 5 SV đều thực hiện lệnh take_fork(i); đồng thời và đều thành công. Bây giờ họ thực hiện tiếp lệnh kế thì tất cả đều bị dừng chờ mãi mãi vì không còn đủa trên bàn.

3.6 Các bài toán IPC kinh điển

1. Bài toán 5 SV ăn cơm :

//thuật giải cải tiến để tránh deadlock :

```
#define N  5
void Sinhvien (int i) {
   while (còn cơm) {
      think();                                  //công đoạn 1
L1:   take_fork(i);
      if (!asyn_take_fork((i+1)%5)) {           //thất bại
         put_fork(i); sleep(random()); goto L1; //công đoạn 2
      }
      eat();                                    //công đoạn 3
      ...
   }
}
```

Qui trình ăn ở trên tránh được deadlock, nhưng còn dựa vào yếu tố ngẫu nhiên nên chưa hiệu quả triệt để.

3.6 Các bài toán IPC kinh điển

```
//thuật giải dùng semaphore để giải quyết tranh chấp và đồng bộ :
#define N                5
#define LEFT             (i-1)%N
#define RIGHT            (i+1)%N
#define THINKING         0
#define HUNGRY           1
#define EATING           2
typedef unsigned int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];
void Sinhvien (int i) {
   while (còn cơm) {
      think();           //công đoạn 1
      take_forks(i);     //công đoạn 2, có thể bị kẹt 1 thời gian
      eat(); put_forks(i); //công đoạn 3
    }
}
```


3.6 Các bài toán IPC kinh điển

```
//thuật giải dùng semaphore để giải quyết tranh chấp và đồng bộ :
void take_forks(int i) {
   down(&mutex);          //đảm bảo 0 tranh chấp khi lấy đủa
   state[i] = HUNGRY;     //ghi nhận trạng thái đang cố gắng lấy đủa
   test(i);               //kiểm tra lấy được không ? nếu được s[i] → 1
   up(&mutex);            //đánh thức các SV khác để họ truy xuất đủa
   down(&s[i]);           //cố gắng và cơm & gắp thức ăn, có thể bị kẹt
}

void test (int i) {
   if (state[i] == HUNGRY              //SV i muốn lấy đủa
      && state[LEFT] != EATING         //SV bên trái chưa lấy
      && state[RIGHT] != EATING) {     //SV bên phải chưa lấy
      state[i] = EATING;               //ghi nhận SV i đã lấy được đủa
      up(&s[i]);                       //tăng s[i] → 1
   }
}
```

3.6 Các bài toán IPC kinh điển

```
//thuật giải dùng semaphore để giải quyết tranh chấp và đồng bộ :
void put_forks (int i) {
    down(&mutex);          //đảm bảo 0 tranh chấp khi lấy đủa
    state[i] = THINKING;   //ghi nhận trạng thái không dùng đủa
    test(LEFT);            //kiểm tra lấy được không ? nếu được đánh thức
                           //SV LEFT dậy để và cơm và gắp thức ăn
    test(RIGHT);           //kiểm tra lấy được không ? nếu được đánh thức
                           //SV RIGHT dậy để và cơm và gắp thức ăn
    up(&mutex);            //đánh thức các SV khác để họ truy xuất đủa
}
```


3.6 Các bài toán IPC kinh điển

2. Bài toán đọc/ghi database :

1. có nhiều process cần đọc/ghi database.
2. làm sao cho việc đọc/ghi database của các process thỏa các điều kiện sau :
   - không tranh chấp nhau và không làm hỏng dữ liệu.
   - việc truy xuất database hiệu quả cao nhất.

Cách giải quyết đơn giản nhất :

1. xem toàn bộ database như 1 tài nguyên dùng chung, dùng 1 semaphore nhị phân db để bảo vệ nó.
2. bất kỳ process đọc/ghi database nào cũng phải down(db) trước khi truy xuất database và up(db) sau khi hoàn tất việc truy xuất database.

3.6 Các bài toán IPC kinh điển

2. Bài toán đọc/ghi database :

```
semaphore db = 1;
void Reader (void) {
    while (cần đọc datbase) {
        down(&db);              //đảm bảo 0 tranh chấp
        read_database();        //đọc database
        up(&db);                //đánh thức các process khác dậy
        use_database(();        //dùng kết quả đọc được
    }
}
void Writer (void) {
    while (cần ghi database) {
        prepare_data();         //chuẩn bị dữ liệu để ghi
        down(&db);              //đảm bảo 0 tranh chấp
        write_database();       //ghi lên database
        up(&db);                //đánh thức các process khác dậy
    }
}
```


3.6 Các bài toán IPC kinh điển

2. Bài toán đọc/ghi database :

Thuật giải cho Reader ở slide trước còn kém hiệu quả, vì các process Reader phải loại trừ tương hỗ trong việc đọc database. Thực tế, việc đọc đồng thời database không làm hư database nên cần cho phép các Reader cùng đọc. Giải thuật Reader có thể hiệu chỉnh thành :

```
semaphore db = 1, mutex = 1; int rc = 0;
void Reader (void) {
    while (cần đọc datbase) {
        down(&mutex);               //đảm bảo 0 tranh chấp
        if (++rc == 1) down(&db);   //nếu là reader đầu tiên thì loại trừ với WR
        up(&mutex);                 //cho phép các process khác truy xuất rc
        read_database();            //đọc database
        down(&mutex);               //đảm bảo 0 tranh chấp
        if (--rc == 0) up(&db);     //nếu là reader cuối thì cho phép writer
        up(&mutex);                 //cho phép các process khác truy xuất rc
        use_database();             //dùng kết quả đọc được
    }
}
```

3.6 Các bài toán IPC kinh điển

3. Bài toán ở tiệm hớt tóc :

1. tiệm chỉ có 1 ghế hớt và 1 thợ cắt.
2. tiệm có hàng ghế chờ N ghế. Tìm qui trình làm việc của thợ và khách trong tiệm sao cho các điều kiện sau đây được thỏa :
   - không tranh chấp nhau về việc ngồi ghế chờ và ghế cắt.
   - ít tốn năng lượng (ít thao tác thừa) nhất.

Phân tích :

1. Tiệm có lúc vắng, lúc nhiều khách. Thợ nên ngủ khi không có khách. Nếu được đánh thức, thợ sẽ mời từng khách đến ghế cắt và cắt tóc cho khách cho đến khi hết khách, thợ lại tiếp tục ngủ.
2. khi đến tiệm, nếu thấy còn ghế chờ, khách sẽ ngồi vào 1 ghế, đánh thức thợ rồi ngủ ngay. Khi nào thợ đánh thức và mời đến ghế cắt thì khách thức dậy và đến ngồi vào ghế cắt rồi ngủ chờ thợ cắt xong, trả tiền và ra về.

3.6 Các bài toán IPC kinh điển

3. Bài toán ở tiệm hớt tóc :

```
//thuật giải cho thợ dùng semaphore
  semaphore customers = 0;
  semaphore barbers = 0;
  semaphore mutex = 1; int waiting = 0;
  void Barber (void) {
     while (còn làm việc) {
        down(&customers);    //nếu 0  có khách, ngủ chờ
        down(&mutex);        //đảm bảo 0 tranh chấp
        waiting--;           //giảm số lượng khách chờ 1 đơn vị
        up(&barbers);        //đánh thức 1 khách và mời họ cắt tóc
        up(&mutex);          //cho phép các process khác truy xuất waiting
        cut_hair();          //cắt tóc cho khách
     }
  }
```

Các bài toán IPC kinh điển

3. Bài toán ở tiệm hớt tóc :

```
//thuật giải cho khách dùng semaphore
  semaphore customers = 0;
  semaphore barbers = 0;
  semaphore mutex = 1; int waiting = 0;
  void Customer (void) {
     down(&mutex);              //đảm bảo 0 tranh chấp
     if (waiting<CHAIRS) {      //nếu còn ghế chờ
           waiting++;           //tăng waiting 1 đơn vị
           up(&customers);      //đánh thức thợ dậy
           up(&mutex);          //cho phép các process khác truy xuất waiting
           down(&barbers);      //ngủ chờ thợ
           get_haircut();       //đến ghế cắt cho thợ cắt tóc
     } else up(&mutex);
  }
```


cuu duong than cong . com