Chương3

QUẢN LÝ TIẾN TRÌNH

Nội dung chương 3

1. Khái niệm về tiến trình (process).
2. Tiểu trình (thread).
3. Điều phối tiến trình.
4. Đồng bộ tiến trình.
5. Tình trạng tắc nghẽn (deadlock)

Đồng bộ tiến trình

- Liên lạc giữa các tiến trình
  - Mục đích:
    - ✓ Để chia sẻ thông tin như dùng chung file, bộ nhớ,…
    - ✓ Hoặc hợp tác hoàn thành công việc
  - Các cơ chế:
    - ✓ Liên lạc bằng tín hiệu (Signal)
    - ✓ Liên lạc bằng đường ống (Pipe)
    - ✓ Liên lạc qua vùng nhớ chia sẻ (shared memory)
    - ✓ Liên lạc bằng thông điệp (Message)
    - ✓ Liên lạc qua socket

Đồng bộ tiến trình

❖ Liên lạc bằng tín hiệu (Signal)

Tín hiệu	Mô tả
SIGINT	Người dùng nhấn phím Ctl-C để ngắt xử lý tiến trình
SIGILL	Tiến trình xử lý một chỉ thị bất hợp lệ
SIGKILL	Yêu cầu kết thúc một tiến trình
SIGFPT	Lỗi chia cho 0
SIGSEGV	Tiến trình truy xuất đến một địa chỉ bất hợp lệ
SIGCLD	Tiến trình con kết thúc

Tín hiệu được gởi đi bởi:

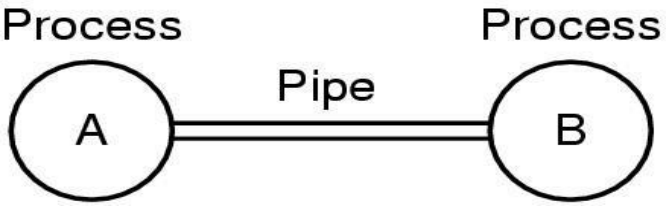
- Phần cứng
- Hệ điều hành:
- Tiến trình:
- Người sử dụng:

Khi tiến trình nhận tín hiệu:

- Gọi hàm xử lý tín hiệu.
- Xử lý theo cách riêng của tiến trình.
- Bỏ qua tín hiệu.

Đồng bộ tiến trình

- Liên lạc bằng đường ống (Pipe)



  - Dữ liệu truyền: dòng các byte (FIFO)
  - Tiến trình đọc pipe sẽ bị khóa nếu pipe trống, và đợi đến khi pipe có dữ liệu mới được truy xuất.
  - Tiến trình ghi pipe sẽ bị khóa nếu pipe đầy, và đợi đến khi pipe có chỗ trống để chứa dữ liệu.

Đồng bộ tiến trình

- Liên lạc qua vùng nhớ chia sẻ (shared memory)
  - Vùng nhớ chia sẻ độc lập với các tiến trình
  - Tiến trình phải gắn kết vùng nhớ chung vào không gian địa chỉ riêng của tiến trình

  - Vùng nhớ chia sẻ là:
    - ✓ Phương pháp nhanh nhất để trao đổi dữ liệu giữa các tiến trình.
    - ✓ Cần được bảo vệ bằng những cơ chế đồng bộ hóa.
    - ✓ Không thể áp dụng hiệu quả trong các hệ phân tán

Đồng bộ tiến trình

- Liên lạc bằng thông điệp (Message)
  - Thiết lập một mối liên kết giữa hai tiến trình
  - Sử dụng các hàm send, receive do hệ điều hành cung cấp để trao đổi thông điệp
  - Cách liên lạc bằng thông điệp:
    - ✓ Liên lạc gián tiếp (indirect communication)
      - Send(A, message): gởi thông điệp tới port A
      - Receive(A, message): nhận thông điệp từ port A
    - ✓ Liên lạc trực tiếp (direct communication)
      - Send(P, message): gởi thông điệp đến process P
      - Receive(Q,message): nhận thông điệp từ process Q

Đồng bộ tiến trình

Ví dụ: Bài toán nhà sản xuất - người tiêu thụ (producer-consumer)

```
void nsx()
{   while(1)
    {   tạo_sp();
        send(ntt,sp); //gởi sp cho ntt   }}
void ntt()
{   while(1)
    {   receive(nsx,sp); //ntt chờ nhận sp
        tiêu_thụ(sp); }}
```

Đồng bộ tiến trình

- Liên lạc qua socket
  - Mỗi tiến trình cần tạo một socket riêng
  - Mỗi socket được kết buộc với một cổng khác nhau.
  - Các thao tác đọc/ghi lên socket chính là sự trao đổi dữ liệu giữa hai tiến trình.
  - Cách liên lạc qua socket:
    - ✓ Liên lạc kiểu thư tín (socket đóng vai trò bưu cục)
      - "tiến trình gởi" ghi dữ liệu vào socket của mình, dữ liệu sẽ được chuyển cho socket của "tiến trình nhận"
      - "tiến trình nhận" sẽ nhận dữ liệu bằng cách đọc dữ liệu từ socket của "tiến trình nhận"

Đồng bộ tiến trình

- Liên lạc qua socket
  - Cách liên lạc qua socket (tt):
    - ✓ Liên lạc kiểu điện thoại (socket đóng vai trò tổng đài)
      - Hai tiến trình cần kết nối trước khi truyền/nhận dữ liệu và kết nối được duy trì suốt quá trình truyền nhận dữ liệu

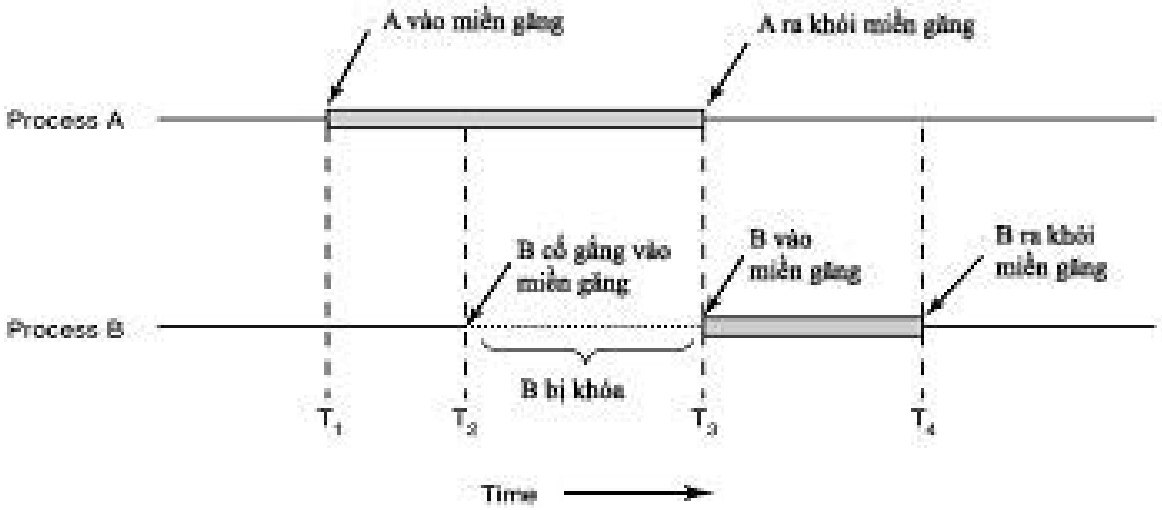
Đồng bộ tiến trình

- Bảo đảm các tiến trình xử lý song song không tác động sai lệch đến nhau.
- Yêu cầu độc quyền truy xuất (Mutual exclusion): tại một thời điểm, chỉ có một tiến trình được quyền truy xuất một tài nguyên không thể chia sẻ.
- Yêu cầu phối hợp (Synchronization): các tiến trình cần hợp tác với nhau để hoàn thành công việc.
- Hai "bài toán đồng bộ" cần giải quyết:
  bài toán "độc quyền truy xuất" ("bài toán miền găng")
  bài toán "phối hợp thực hiện".

Đồng bộ tiến trình

- Miền găng (critical section)
  - Đoạn mã của một tiến trình có khả năng xảy ra lỗi khi truy xuất tài nguyên dùng chung (biến, tập tin,…).
  - Ví dụ:

    if (taikhoan >= tienrut) taikhoan = taikhoan - tienrut;
    else Thong bao “khong the rut tien ! “;



Đồng bộ tiến trình

- Các điều kiện cần khi giải quyết bài toán miền găng
  1. Không có giả thiết về tốc độ của các tiến trình, cũng như về số lượng bộ xử lý.
  2. Không có hai tiến trình cùng ở trong miền găng cùng lúc.
  3. Một tiến trình bên ngoài miền găng không được ngăn cản các tiến trình khác vào miền găng.
  4. Không có tiến trình nào phải chờ vô hạn để được vào miền găng

Đồng bộ tiến trình

- Các nhóm giải pháp đồng bộ
  - Busy Waiting
  - Sleep And Wakeup
    - ✓ Semaphore
    - ✓ Monitor
  - Message.

Đồng bộ tiến trình

- Busy Waiting (bận thì đợi)
  - Giải pháp phần mềm
    - ✓ Thuật toán sử dụng biến cờ hiệu
    - ✓ Thuật toán sử dụng biến luân phiên
    - ✓ Thuật toán Peterson
  - Giải pháp phần cứng
    - ✓ Cấm ngắt
    - ✓ Sử dụng lệnh TSL (Test and Set Lock)

Đồng bộ tiến trình

- Thuật toán sử dụng biến cờ hiệu *(dùng cho nhiều tiến trình)*
  - ✓ lock=0 là không có tiến trình trong miền găng.
  - ✓ lock=1 là có một tiến trình trong miền găng.

```
lock=0;
while (1)
{
        while (lock == 1);
        lock = 1;
        critical-section ();
        lock = 0;
        noncritical-section();
}
```

Đồng bộ tiến trình

- Thuật toán sử dụng biến luân phiên *(dùng cho 2 tiến trình)*

  Hai tiến trình A, B sử dụng chung biến turn:

  - ✓ turn = 0, tiến trình A được vào miền găng
  - ✓ turn=1 thì B được vào miền găng

```
// tiến trình A
while (1) {
        while (turn == 1);
        critical-section ();
        turn = 1;
        Noncritical-section ();
}
```

turn=0 thi A duoc vao mien gang

```
// tiến trình B
while (1) {
        while (turn == 0);
        critical-section ();
        turn = 0;
        Noncritical-section ();
}
```

turn=1 thi B duoc vao mien gang

- Hai tiến trình chắc chắn không thể vào miền găng cùng lúc, vì tại một thời điểm turn chỉ có một gía trị.
- Vi phạm: một tiến trình có thể bị ngăn chặn vào miền găng bởi một tiến trình khác không ở trong miền găng.

Đồng bộ tiến trình

❖ Thuật toán Peterson *(dùng cho 2 tiến trình)*

- Dùng chung hai biến turn và flag[2] (kiểu int).
  - ✓ flag[0]=flag[1]=FALSE
  - ✓ turn được khởi động là 0 hay 1.
- Nếu flag[i] = TRUE (i=0,1) -> P_i muốn vào miền găng và turn=i là đến lượt P_i.
- Để có thể vào được miền găng:
  - ✓ P_i đặt trị flag[i]=TRUE để thông báo nó muốn vào miền găng.
  - ✓ Đặt turn=j để thử đề nghị tiến trình P_j vào miền găng.
- Nếu tiến trình P_j không quan tâm đến việc vào miền găng (flag[j] = FALSE), thì P_i có thể vào miền găng
  - ✓ Nếu flag [j]=TRUE thì P_i phải chờ đến khi flag [j]=FALSE.
- Khi tiến trình P_i ra khỏi miền găng, nó đặt lại trị flag[i] là FALSE.

Đồng bộ tiến trình

❖ Thuật toán Peterson *(đoạn code)*

```
// tiến trình P0 (i=0)
while (TRUE)
{
    flag [0]= TRUE; //P0 muốn vào
                    //miền găng
    turn = 1; //thu de nghi P1 vao
    while (turn==1 &&
                flag[1]==TRUE);
    //neu P1 muon vao thi P0 chờ
    critical_section();
    flag [0] = FALSE; //P0 ra ngoài mg
    noncritical_section ();
}
```

```
// tiến trình P1 (i=1)
while (TRUE)
{
    flag [1]= TRUE; //P1 muon vao
                    //miền găng
    turn = 0; //thử de nghi P0 vao
    while (turn == 0 && flag
    [0]==TRUE); //neu P0 muon vao
                //thi P1 chờ
    critical_section();
    flag [1] = FALSE;//P1 ra ngoài mg
    Noncritical_section ();
}
```

Đồng bộ tiến trình

- Cấm ngắt
  - Tiến trình cấm tất cả các ngắt trước khi vào miền găng, và phục hồi ngắt khi ra khỏi miền găng.
    - ✓ Không an toàn cho hệ thống
    - ✓ Không tác dụng trên hệ thống có nhiều bộ xử lý

Đồng bộ tiến trình

- Sử dụng lệnh TSL *(Test and Set Lock)*

```
boolean Test_And_Set_Lock (boolean lock){
        boolean temp=lock;
        lock = TRUE;
        return temp; //trả về giá trị ban đầu của
                     //biến lock
}
```

- Lệnh TSL cho phép kiểm tra và cập nhật một vùng nhớ trong một thao tác độc quyền.

```
boolean lock=FALSE; //biến dùng chung
while (TRUE) {
        while (Test_And_Set_Lock(lock));
        critical_section ();
        lock = FALSE;
        noncritical_section ();
}
```

- Hoạt động trên hệ thống có nhiều bộ xử lý.

Đồng bộ tiến trình

- Nhóm giải pháp “SLEEP and WAKEUP “ (ngủ và đánh thức)

1. Sử dụng lệnh SLEEP VÀ WAKEUP
2. Sử dụng cấu trúc Semaphore
3. Sử dụng cấu trúc Monitors
4. Sử dụng thông điệp

Đồng bộ tiến trình

- Sử dụng lệnh SLEEP VÀ WAKEUP
  - SLEEP → "danh sách sẵn sàng", lấy lại CPU cấp cho P khác.
  - WAKEUP → HĐH chọn một P trong ready list, cho thực hiện tiếp.
  - P chưa đủ điều kiện vào miền găng → gọi SLEEP để tự khóa, đến khi có P khác gọi WAKEUP để giải phóng cho nó.
  - Một tiến trình gọi WAKEUP khi ra khỏi miền găng để đánh thức một tiến trình đang chờ, tạo cơ hội cho tiến trình này vào miền găng

Đồng bộ tiến trình

❖ Sử dụng lệnh SLEEP VÀ WAKEUP

```
int busy=FALSE;   // TRUE là có tiến trình trong miền găng, FALSE là không có
int blocked=0;   // đếm số lượng tiến trình đang bị khóa
while(TRUE){
        if (busy) {
            blocked = blocked + 1;
            sleep();
        }else busy = TRUE;
        critical-section ();
        busy = FALSE;
        if (blocked>0){
            wakeup(); //đánh thức một tiến trình đang chờ
            blocked = blocked - 1;
        }
        Noncritical-section ();
}
```

Đồng bộ tiến trình

- Sử dụng cấu trúc Semaphore
  - Biến semaphore s có các thuộc tính:
    - ✓ Một giá trị nguyên dương e;
    - ✓ Một hàng đợi f : lưu danh sách các tiến trình đang chờ trên semaphore s.
  - Hai thao tác trên semaphore s:
    - ✓ Down(s): e=e-1.
      - Nếu e < 0 thì tiến trình phải chờ trong f (sleep), ngược lại tiến trình tiếp tục.
    - ✓ Up(s): e=e+1.
      - Nếu e<=0 thì chọn một tiến trình trong f cho tiếp tục thực hiện (đánh thức).

Đồng bộ tiến trình

- Sử dụng cấu trúc Semaphore

P là tiến trình thực hiện thao tác Down(s) hay Up(s)

```
Down(s) {
e = e - 1;
if(e < 0) {
status(P)= blocked; //khóa P (trạng thái chờ)
enter(P,f);    //cho P vào hàng đợi f
      }
}
Up(s){
e = e + 1;
if(e<= 0 ){
            exit(Q,f); //lấy một tt Q ra khỏi hàng đợi f
            status (Q) = ready; //chuyển Q sang trạng thái sẵn sàng
            enter(Q,ready-list); //đưa Q vào danh sách sẵn sàng
      }
}
```

Đồng bộ tiến trình

- Sử dụng cấu trúc Semaphore
  - Hệ điều hành cần cài đặt các thao tác Down, Up là độc quyền.
  - Cấu trúc của semaphore:

```
class semaphore{
    int e;
    PCB * f; //ds  riêng của semaphore
    public:
    down();
    up();
};
```

  - |e| = số tiến trình đang chờ trên f.

Đồng bộ tiến trình

- Giải quyết bài toán miền găng bằng Semaphores
  - Dùng một semaphore s, e được khởi gán là 1.
  - Tất cả các tiến trình áp dụng cùng cấu trúc chương trình sau:

```
semaphore s=1; //nghĩa là e của s=1
while (1)
{
    Down(s);
    critical-section ();
    Up(s);
    Noncritical-section ();
}
```

Đồng bộ tiến trình

- Giải quyết bài toán đồng bộ bằng Semaphores
  - Hai tiến trình đồng hành P1 và P2, P1 thực hiện công việc 1, P2 thực hiện công việc 2.
  - Cv1 làm trước rồi mới làm cv2, cho P1 và P2 dùng chung một semaphore s, khởi gán e(s)= 0:

```
semaphore s=0; //dùng chung cho hai tiến trình
P1:{
      job1();
      Up(s); //đánh thức P2
}
P2:{
      Down(s); // chờ P1 đánh thức
      job2();
}
```

Đồng bộ tiến trình

- Vấn đề khi sử dụng semaphore
  - Tiến trình quên gọi Up(s), và kết quả là khi ra khỏi miền găng nó sẽ không cho tiến trình khác vào miền găng!

```
e(s)=1;
while (1)
{
        Down(s);
        critical-section ();
        Noncritical-section ();
}
```

Đồng bộ tiến trình

- Vấn đề khi sử dụng semaphore
  - Sử dụng semaphore có thể gây ra tình trạng tắc nghẽn.

```
P1:
{
        down(s1); down(s2);
        ....
        up(s1);  up(s2);
}
P2:
{
        down(s2); down(s1);
        ....
        up(s2); up(s1);
}
```

Hai tiến trình P1, P2 sử dụng chung 2 semaphore s1=s2=1

Nếu thứ tự thực hiện như sau:
P1: down(s1), P2: down(s2) ,
P1: down(s2), P2: down(s1)
Khi đó s1=s2=-1 nên P1,P2 đều chờ mãi

Đồng bộ tiến trình

- ❖ Sử dụng cấu trúc Monitors

- ➢ Hoare(1974) và Brinch & Hansen (1975) đã đề nghị một cơ chế cao hơn gọil à monitor được cung cấp bởi 1 số ngôn ngữ lập trình
- ➢ Monitor là một cấu trúc đặc biệt (lớp)
  - ✓ các phương thức độc quyền (critical-section)
  - ✓ các biến (được dùng chung cho các tiến trình)
    - Các biến trong monitor chỉ có thể được truy xuất bởi các phương thức trong monitor
  - ✓ Tại một thời điểm, chỉ có một tiến trình duy nhất được hoạt động bên trong một monitor.
  - ✓ Biến điều kiện c
    - dùng để đồng bộ việc sử dụng các biến trong monitor.
    - Wait(c) và Signal(c):

Đồng bộ tiến trình

- Sử dụng cấu trúc Monitors

Wait(c): chuyển trạng thái tiến trình gọi sang chờ (blocked) và đặt tiến trình này vào hàng đợi trên biến điều kiện c.

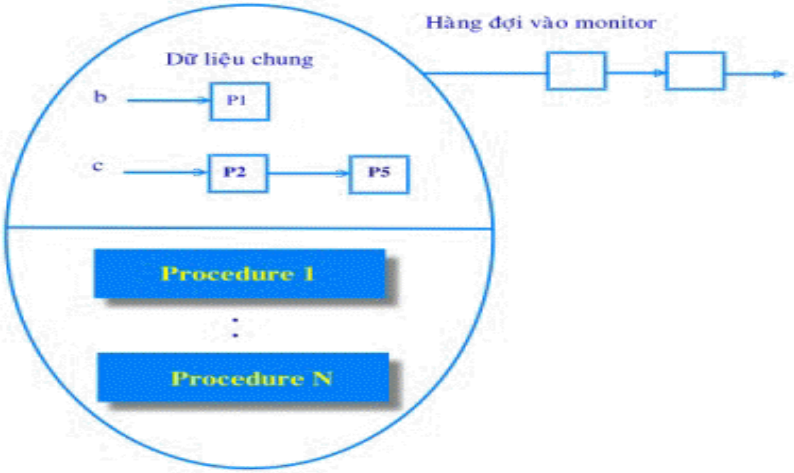
```
Wait(c)
{
    status(P)= blocked; //chuyển P sang trạng thái chờ
    enter(P,f(c)); //đặt P vào hàng đợi f(c) của biến điều  kiện c
}
```

Signal(c): khi có một tiến trình đang chờ trong hàng đợi c, kích hoạt tiến trình đó và tiến trình gọi sẽ rời khỏi monitor.

```
Signal(c)
{
    if (f(c) != NULL){
        exit(Q,f(c)); //Lấy tiến trình Q đang chờ trên c
        status(Q) = ready; //chuyển Q sang trạng thái sẵn sàng
        enter(Q,ready-list); //đưa Q vào danh sách sẵn sàng.
    }
}
```

Đồng bộ tiến trình

- Sử dụng cấu trúc Monitors




```
monitor <tên monitor > //khai báo monitor
dùng chung cho các tiến trình
{
        <cac bien dung chung>;
        <các biến điều kiện>;
        <cac phuong thuc độc quyền>;
}
//tiến trình Pi:
while (1) //cấu trúc tiến trình thứ i
{
        Noncritical-section ();
        <ten monitor>.Phươngthức_i;  //thực
                hiện công việc độc quyền thứ i
        Noncritical-section ();
}
```

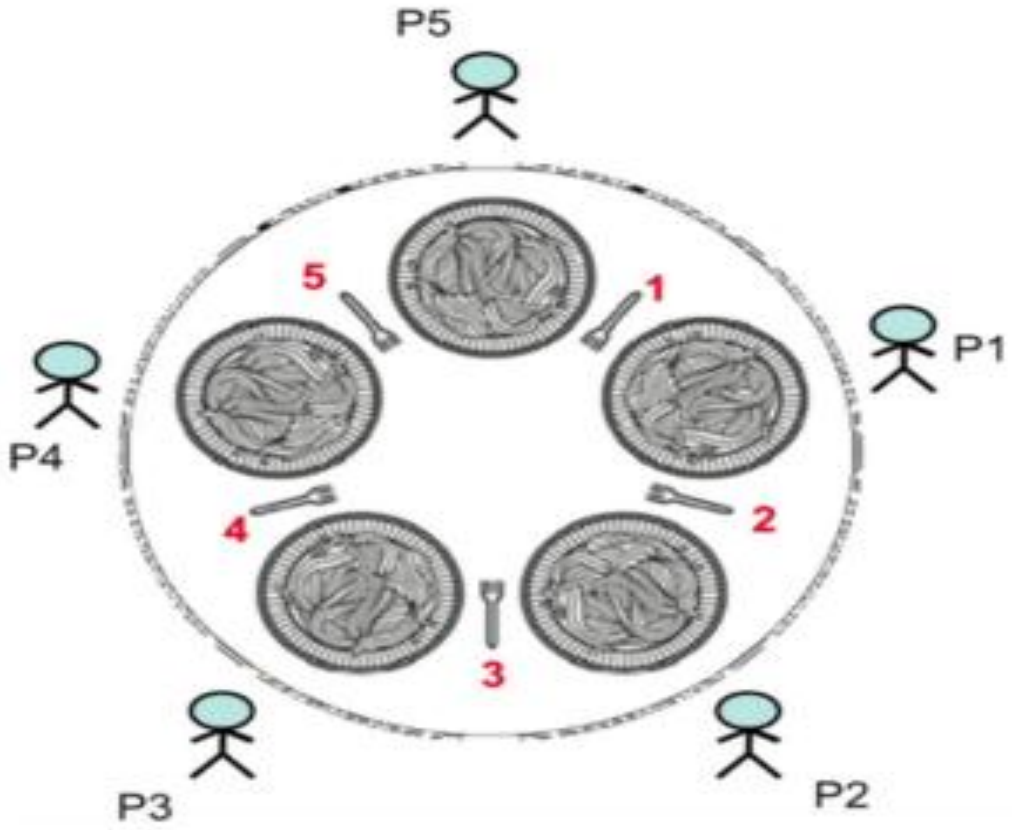
critical-section

Đồng bộ tiến trình

- Sử dụng cấu trúc Monitors
  - Nguy cơ thực hiện đồng bộ hóa sai giảm rất nhiều.
  - Ít có ngôn ngữ hỗ trợ cấu trúc monitor.

Đồng bộ tiến trình

- Monitors và Bài toán 5 triết gia ăn tối



Đồng bộ tiến trình

- Monitors và Bài toán 5 triết gia ăn tối

```
while (true) {
    // thinking
    doAction(System.nanoTime() + ": Ngồi suy nghĩ");
    synchronized (leftFork) {
        doAction(
          System.nanoTime()
            + ": Nhặt đũa bên trái");
        synchronized (rightFork) {
            // eating
            doAction(
              System.nanoTime()
                + ": Nhặt đũa bên phải - Ăn");

            doAction(
              System.nanoTime()
                + ": Đặt đũa bên phải xuống");
        }
        // Back to thinking
        doAction(
          System.nanoTime()
            + ": Đặt đũa bên trái xuống. Ngồi suy nghĩ");
    }
}
```

Đồng bộ tiến trình

- Monitors và Bài toán 5 triết gia ăn tối

```
public class TrietGia implements Runnable {

    // Khai báo forks cho triết gia
   private Object leftFork;
   private Object rightFork;

   public TrietGia(Object leftFork, Object rightFork) {
       this.leftFork = leftFork;
       this.rightFork = rightFork;
   }
   private void doAction(String action) throws InterruptedException {
       System.out.println(
         Thread.currentThread().getName() + " " + action);
       Thread.sleep(((int) (Math.random() * 100)));
   }
```

Tình trạng tắc nghẽn (deadlock)

- Điều kiện xuất hiện tắc nghẽn

1. Điều kiện 1: Có sử dụng tài nguyên không thể chia sẻ.
2. Điều kiện 2: Sự chiếm giữ và yêu cầu thêm tài nguyên không thể chia sẻ.
3. Điều kiện 3: Không thu hồi tài nguyên từ tiến trình đang giữ chúng.
4. Điều kiện 4: Tồn tại một chu trình trong đồ thị cấp phát tài nguyên.

Tình trạng tắc nghẽn (deadlock)

- Các phương pháp xử lý tắc nghẽn và ngăn chặn tắc nghẽn
  - Sử dụng một thuật toán cấp phát tài nguyên
    -> không bao giờ xảy ra tắc nghẽn.
  - Cho phép xảy ra tắc nghẽn
    -> tìm cách sữa chữa tắc nghẽn.
  - Bỏ qua việc xử lý tắc nghẽn, xem như hệ thống không bao giờ xảy ra tắc nghẽn.

Tình trạng tắc nghẽn (deadlock)

- Ngăn chặn tắc nghẽn
  - Điều kiện 1 gần như không thể tránh được.
  - Để điều kiện 2 không xảy ra:
    - ✓ Tiến trình phải yêu cầu tất cả các tài nguyên cần thiết trước khi cho bắt đầu xử lý.
    - ✓ Khi tiến trình yêu cầu một tài nguyên mới và bị từ chối
      - Giải phóng các tài nguyên đang chiếm giữ
      - Tài nguyên cũ được cấp lại cùng với tài nguyên mới.
  - Để điều kiện 3 không xảy ra:
    - ✓ Thu hồi tài nguyên từ các tiến trình bị khoá và cấp phát trở lại cho tiến trình khi nó thoát khỏi tình trạng bị khóa.
  - Để điều kiện 4 không xảy ra:
    - ✓ Khi tiến trình đang chiếm giữ tài nguyên Ri thì chỉ có thể yêu cầu các tài nguyên Rj nếu F(Rj) > F(Ri).

Tình trạng tắc nghẽn (deadlock)

- Giải thuật cấp phát tài nguyên tránh tắc nghẽn
  - Giải thuật xác định trạng thái an toàn
  - Giải thuật Banker

Giải thuật Banker

Thuật toán có thể được hiểu như sau:

1. Đặt hai biến là Work và Finish lần lượt là các vectơ có chiều dài m và n.

   Khởi tạo: Work = Available

   Finish [i] = False; for i = 1, 2,……, n.

2. Tìm giá trị của i sao cho thõa mãn hai điều kiện sau:

   a) Finish [i] = False

   b) $Need_i$ <= work

Nếu không tồn tại giá trị i thì tiếp bước 4.

3. Tính:

   Work = Work + $Allocation_i$

   Finish[i] = True

   Quay lại bước 2

4. Kiểm tra nếu Finish[i] = True cho tất các các giá trị của i thì hệ thống an toàn.

Giải thuật Banker

Ví dụ: Cho bảng giá trị như bên dưới

Process	Allocation	MAX	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

I – 0...4

Gi

Process	Allocation			MAX	
	A	B	C	A	B
P_0	0	1	0	7	5
P_1	2	0	0	3	2
P_2	3	0	2	9	0
P_3	2	1	1	2	2
P_4	0	0	2	4	3

Step 1:

R = 3, P = 5

Work = Available

Work = 3, 3, 2

Finish = False, False, False, False, False = 0,1,2,3,4.

Step 2:

For i = 0

$Need_0 = Max_0 – Allocation_0 = 7, 4, 3.$

Finish[0] = False and $Need_0$ > Work ~ 7,4,3 > 3,3,2

P_0 cần đợi vì chưa tồn tại $Need_0$ <= Work

Giải thu

Process	Allocation	
	A B C	A
P_0	0 1 0	
P_1	2 0 0	
P_2	3 0 2	
P_3	2 1 1	
P_4	0 0 2	

Step 2:

For i = 1

$Need_1 = Max_1 – Allocation_1 = 1, 2, 2.$

Finish[1] = False and $Need_1 < Work_1$ ~ 1,2,2 < 3,3,2

Nên P_1 phải được giữ theo trình tự an toàn.

Step 3:

$Work = Work + Allocation_1 = [3,3,2] + [2,0,0] = [5,3,2]$

Finish = False, True, False, False, False = 0,1,2,3,4

So True = position 1 => P1

Giải thuật Banker

Step 2:

For i = 2

$Need_2 = Max_2 – Allocation_2 = 6, 0, 0.$

Finish[2] = False and $Need_2$ > Work ~ 6,0,0 > 5,3,2

$Need_2$ > Work nên P_2 cần đợi

Step 2:

For i = 3

$Need_3 = Max_3 - Allocation_3 = 0, 1, 1.$

Finish[3] = False and $Need_3$ < Work ~ 0,1,1 < 5,3,2

Nên P_3 phải được giữ theo trình tự an toàn.

Step 3:

$Work = Work + Allocation_3 = [5,3,2] + [2,1,1] = [7,4,3]$

Finish = False, True, False, True, False = 0,1,2,3,4

So True = position 3 => P1 next P3

Process	Al
	A
P_0	0
P_1	2
P_2	3
P_3	2
P_4	0

Giải th

Process	Allocation	MA
	A B C	A B
P0	0 1 0	7 5
P1	2 0 0	3 2
P2	3 0 2	9 0
P3	2 1 1	2 2
P4	0 0 2	4 3

Step 2:

For i = 4

$Need_4 = Max_4 – Allocation_4 = 4, 3, 1.$

Finish[4] = False and $Need_4$ < Work ~ 4,3,1 < 7,4,3

Nên P_4 phải được giữ theo trình tự an toàn.

Step 3:

$Work = Work – Allocation_4 = [7,4,3] + [0,0,2] = [7,4,5]$

Finish = False, True, False, True, True = 0,1,2,3,4

So True = position 4 => P1 next P3 next P4

//Hết vòng

Process	Allocation	MAX	
	A B C	A B C	
P_0	0 1 0	7 5 3	
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Step 2:

For i = 0

$Need_0 = Max_0 – Allocation_0 = 7, 4, 3.$

Finish[0] = False and $Need_0$ < Work ~ 7,4,3 < 7,4,5

Nên P_0 phải được giữ theo trình tự an toàn.

Step 3:

$Work = Work – Allocation_0 = [7,4,5] + [0,1,0] = [7,5,5]$

Finish = True, True, False, True, True = 0,1,2,3,4

So True = position 0 => P1 next P3 next P4 next P0

Giải thuật Banker

Step 2:

For i = 2

$Need_2 = Max_2 – Allocation_2 = [9,0,2] – [3,0,2] = [6,0,0]$.

Finish[2] = False and $Need_2 < Work \sim 6,0,0 < 7,5,5$

Nên P_2 phải được giữ theo trình tự an toàn.

Step 3:

$Work = Work + Allocation_2 = [7,5,5] + [3,0,2] = [10,5,7]$

Finish = True, True, True, True, True = 0,1,2,3,4

So True = position 2 => P1 next P3 next P4 next P0 next P2

Step4: Finish[i] = True for 0 <= i <= 4

Do đó hệ thống ở trạng thái an toàn là: P1, P3, P4, P0, P2.

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

Bài tập

Sử dụng thuật toán Banker. Em hãy tính thứ tự an toàn với bảng tính đã cho như sau:

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	0	1	1	2	1	3	3	2
P1	1	0	2	2	1	3			
P2	2	1	1	5	2	2			
P3	2	0	2	5	2	2			
P4	1	1	0	3	4	1			

Sử dụng thuật toán Banker. Em hãy tính thứ tự an toàn với bảng tính đã cho như sau:

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	0	1	7	2	1	3	3	2
P1	2	1	2	2	2	3			
P2	2	1	1	5	2	2			
P3	4	3	2	5	4	2			
P4	2	2	0	3	4	1			