QUẢN LÝ TIẾN TRÌNH

- Các khái niệm về tiến trình
- Điều phối các tiến trình
- Liên lạc giữa các tiến trình
- Đồng bộ các tiến trình
- Tính trạng tắc nghẽn (deadlock)

Tiến trình (Process)

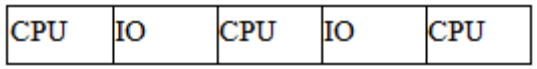
- Tiến trình là một chương trình đang xử lý
- Mỗi tiến trình có một không gian địa chỉ, một con trỏ lệnh, một tập các thanh ghi và stack riêng.
- Tiến trình có thể cần đến một số tài nguyên như CPU, bộ nhớ chính, các tập tin và thiết bị nhập/xuất.
- Hệ điều hành sử dụng bộ điều phối (scheduler) để quyết định thời điểm cần dừng hoạt động của tiến trình đang xử lý và lựa chọn tiến trình tiếp theo cần thực hiện.
- Trong hệ thống có những tiến trình của hệ điều hành và tiến trình của người dùng.

Mục đích cho nhiều tiến trình hoạt động đồng thời

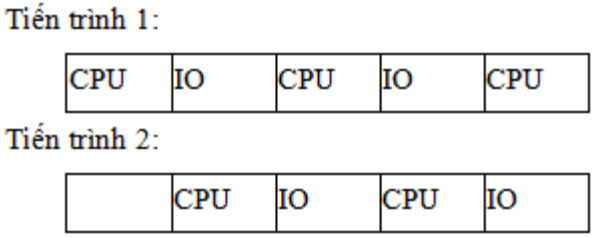
- ***Tăng hiệu suất sử dụng CPU (tăng mức độ đa chương)***
- ***Tăng mức độ đa nhiệm***
- ***Tăng tốc độ xử lý***

Tăng hiệu suất sử dụng CPU (tăng mức độ đa chương)

- Phần lớn các tiến trình khi thi hành đều trải qua nhiều chu kỳ xử lý (sử dụng CPU) và chu kỳ nhập xuất (sử dụng các thiết bị nhập xuất) xen kẽ như sau :

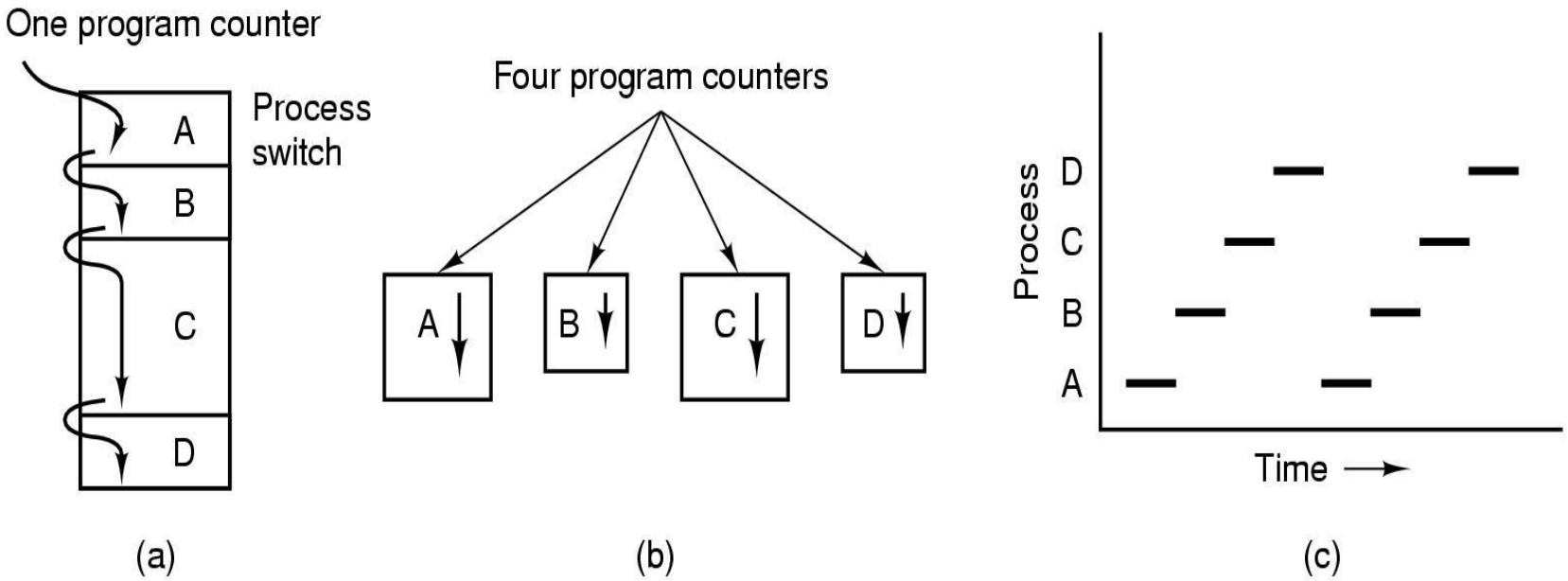


- Nếu chỉ có 1 tiến trình duy nhất trong hệ thống, thì vào các chu kỳ IO của tiến trình, CPU sẽ hoàn toàn nhàn rỗi. Ý tưởng tăng cường số lượng tiến trình trong hệ thống là để tận dụng CPU: nếu tiến trình 1 xử lý IO, thì hệ điều hành có thể sử dụng CPU để thực hiện tiến trình 2...



Tăng mức độ đa nhiệm

- Cho mỗi tiến trình thực thi luân phiên trong một thời gian rất ngắn, tạo cảm giác là hệ thống có nhiều tiến trình thực thi đồng thời.



Tăng tốc độ xử lý

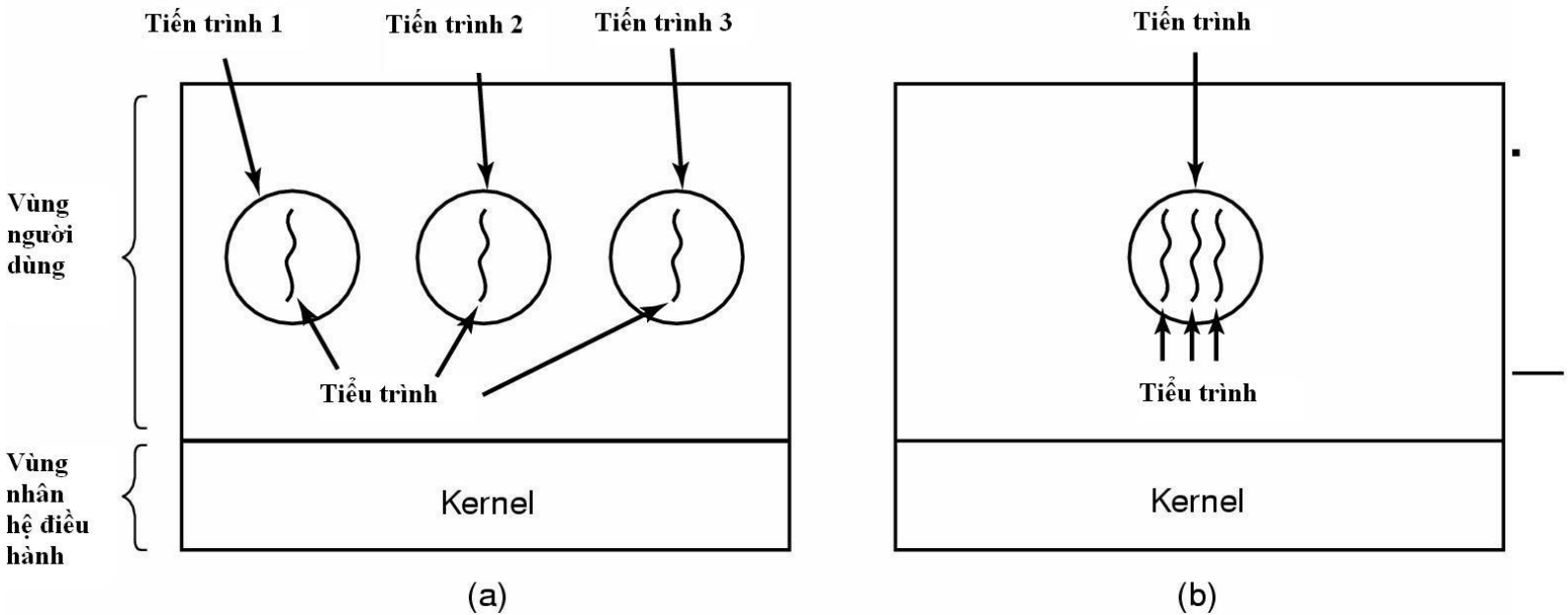
- Một số bài toán có thể xử lý song song nếu được xây dựng thành nhiều đơn thể hoạt động đồng thời thì sẽ tiết kiệm được thời gian xử lý.
- Ví dụ xét bài toán tính giá trị biểu thức kq = a*b + c*d . Nếu tiến hành tính đồng thời (a*b) và (c*d) thì thời gian xử lý sẽ ngắn hơn là thực hiện tuần tự.

Tiểu trình (thread)

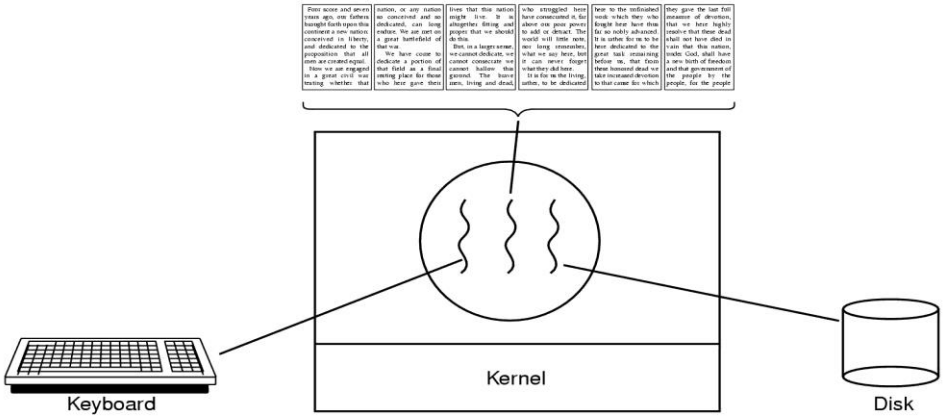
- Một tiến trình có thể tạo nhiều tiểu trình.
- Mỗi tiểu trình thực hiện một chức năng nào đó và thực thi đồng thời cũng bằng cách chia sẻ CPU.
- Các tiểu trình trong cùng một tiến trình dùng chung không gian địa chỉ tiến trình nhưng có con trỏ lệnh, tập các thanh ghi và stack riêng.
- Một tiểu trình cũng có thể tạo lập các tiểu trình con, và nhận các trạng thái khác nhau như một tiến trình.

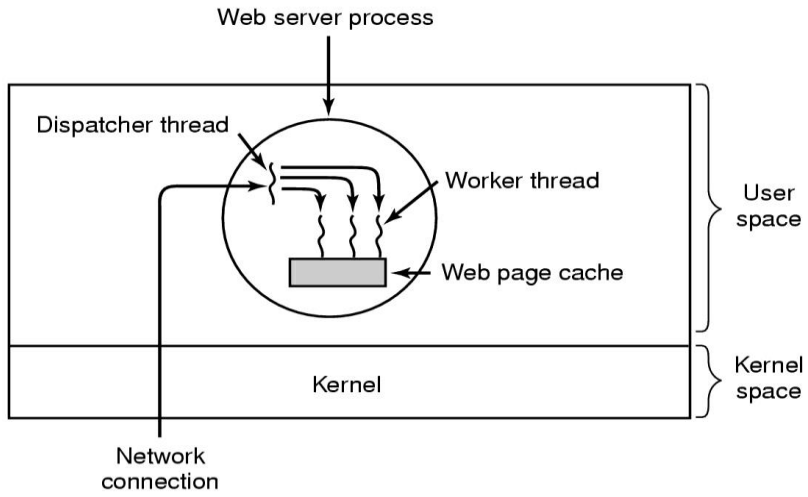
Liên lạc giữa các tiểu trình

- Các tiến trình chỉ có thể liên lạc với nhau thông qua các cơ chế do hệ điều hành cung cấp.
- Các tiểu trình liên lạc với nhau dễ dàng thông qua các biến toàn cục của tiến trình.
- Các tiểu trình có thể do hệ điều hành quản lý hoặc hệ điều hành và tiến trình cùng phối hợp quản lý



Ví dụ về tiểu trình






```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

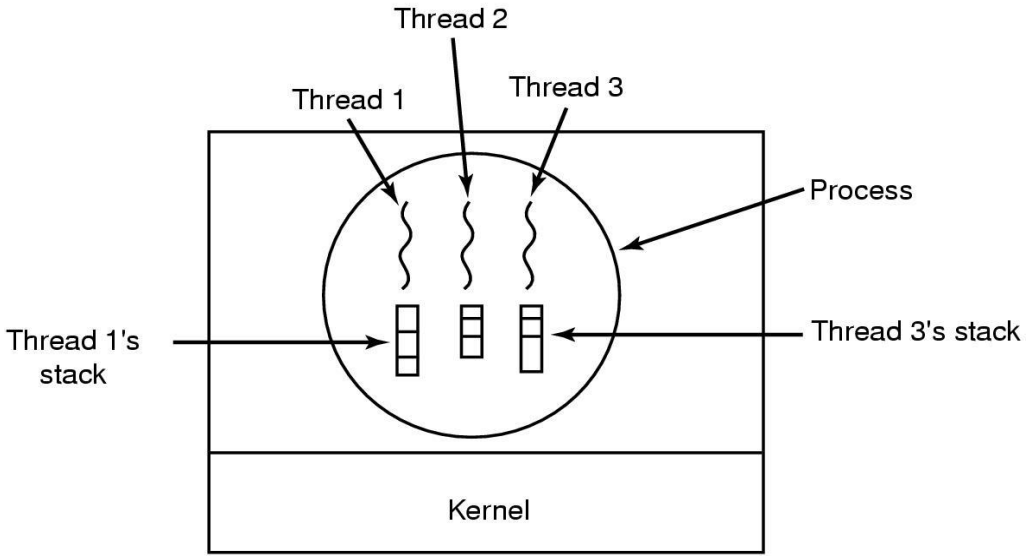
(a)

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
      read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

Ví dụ về tiểu trình

- Một process có ba thread, mỗi thread sẽ có stack riêng

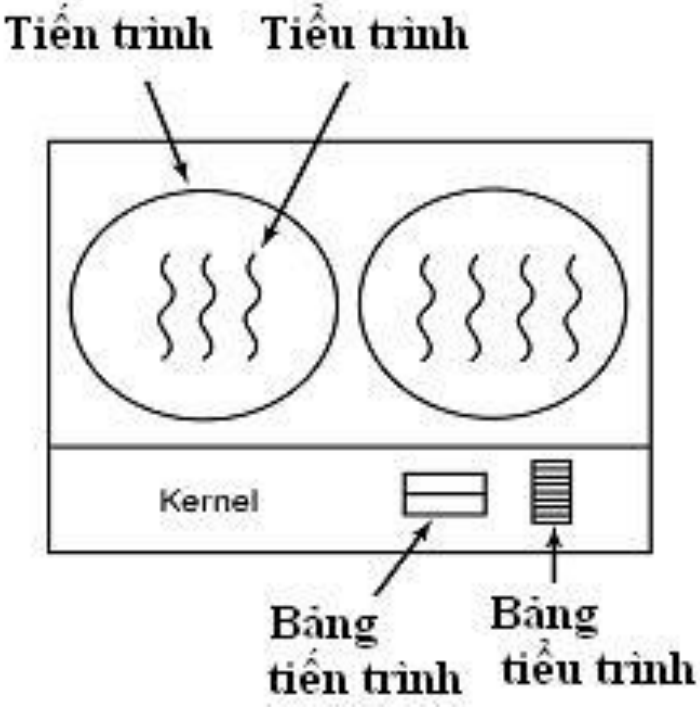


Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Cài đặt tiểu trình (Threads)

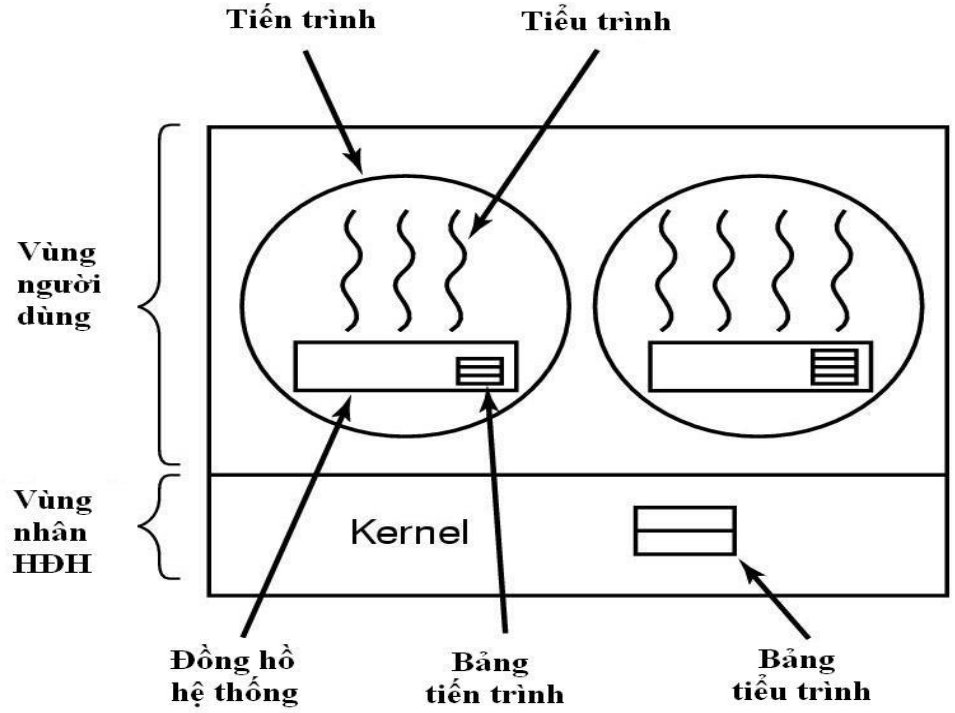
- **Cài đặt trong Kernel-space**
- **Cài đặt trong User-space**
- **Cài đặt trong Kernel-space và User-space**

Cài đặt tiểu trình trong Kernel-space



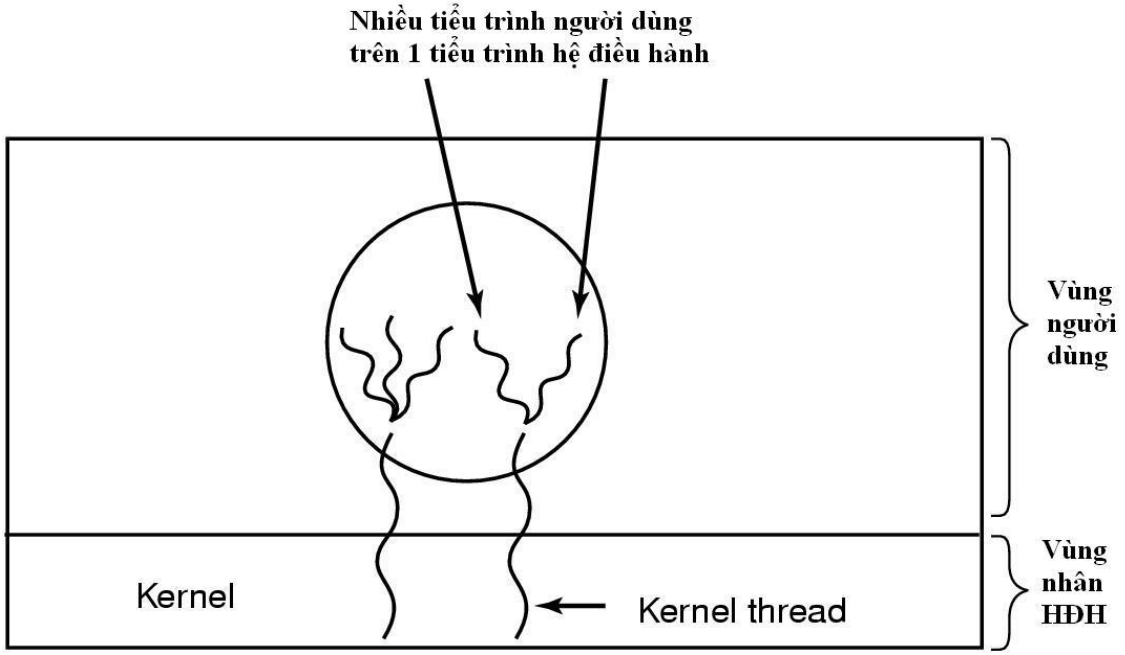
- Bảng quản lý thread lưu trữ ở phần kernel.
- Việc điều phối các thread là do hệ điều hành chịu trách nhiệm.

Cài đặt tiểu trình trong User-space



- Bảng quản lý thread lưu trữ ở phần user-space.
- Việc điều phối các thread là do tiến trình chịu trách nhiệm.

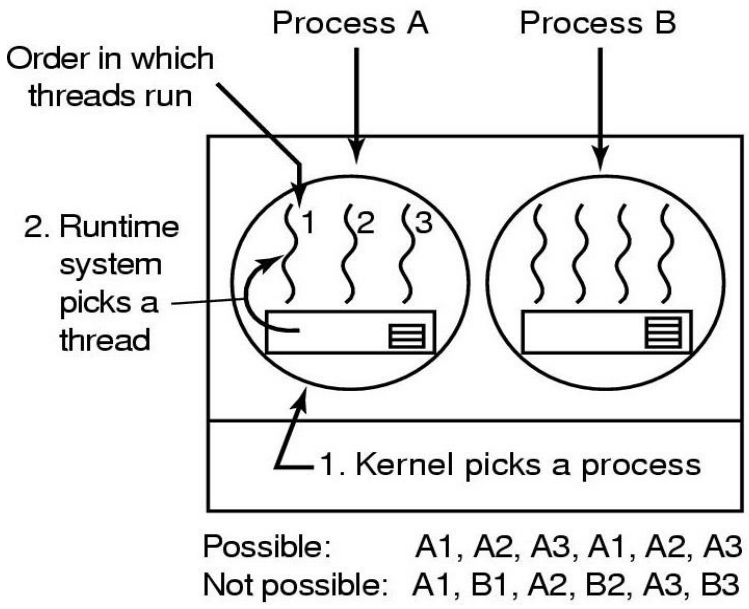
Cài đặt trong Kernel-space và User-space



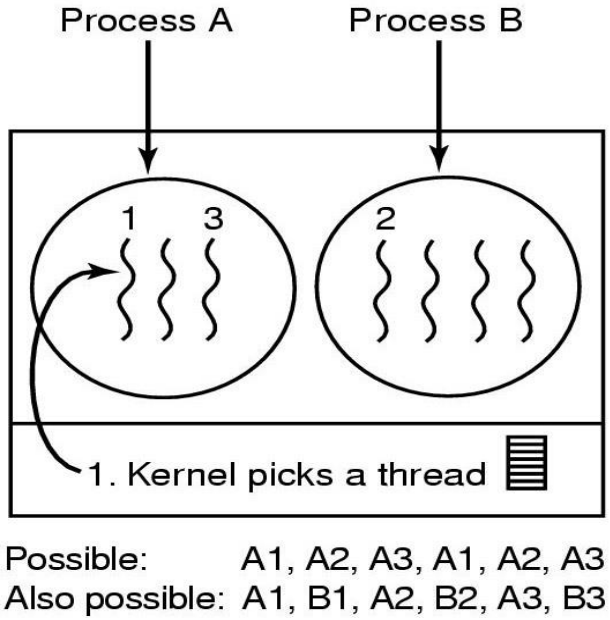
- Một thread của hệ điều hành quản lý một số thread của tiến trình

Ví dụ về điều phối tiểu trình

- quantum của process=50 msec
- quantum của thread=5 msec
- Tiến trình A có 3 thread, tiến trình B có 4 thread.



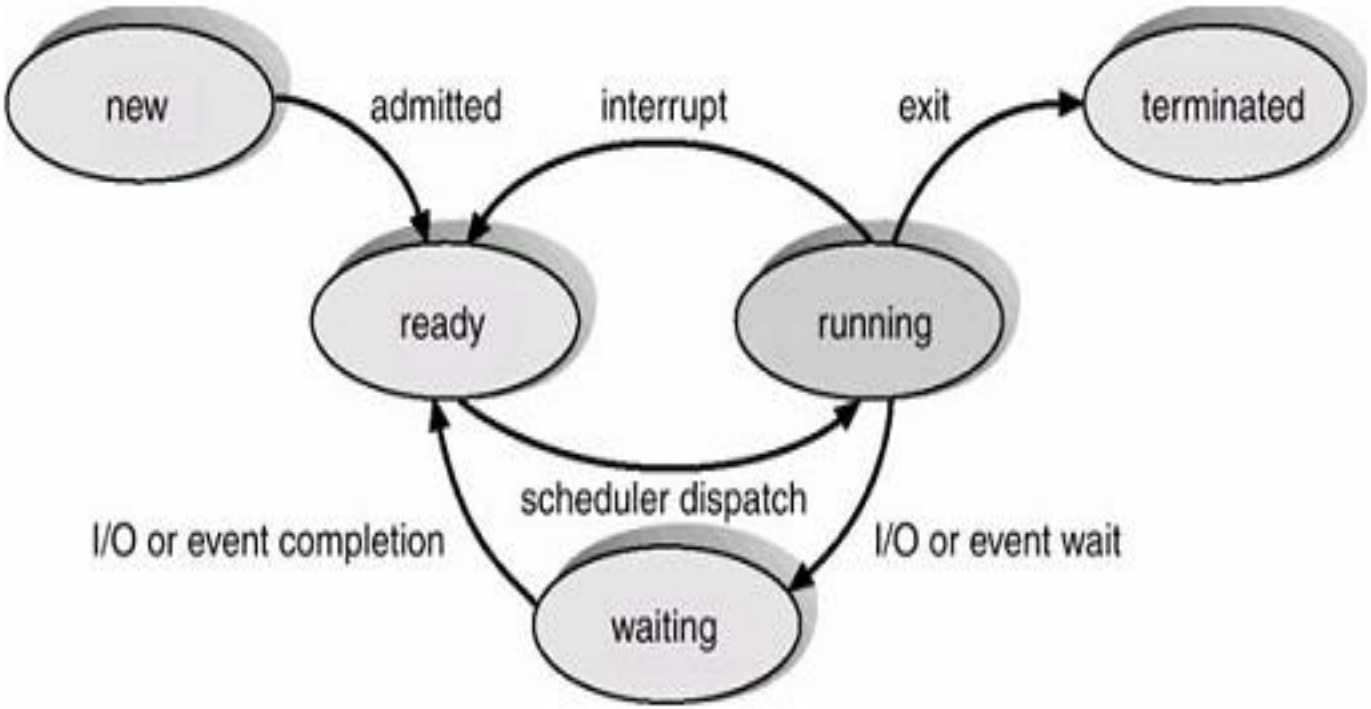
Điều phối thread được thực hiện mức user-space



Điều phối thread được thực hiện mức kernel-space

Các trạng thái của tiến trình

- **Mới tạo** : tiến trình đang được tạo lập.
- **Running** : các chỉ thị của tiến trình đang được xử lý.
- **Blocked** : tiến trình chờ được cấp phát một tài nguyên, hay chờ một sự kiện xảy ra .
- **Ready** : tiến trình chờ được cấp phát CPU để xử lý.
- **Kết thúc** : tiến trình hoàn tất xử lý.

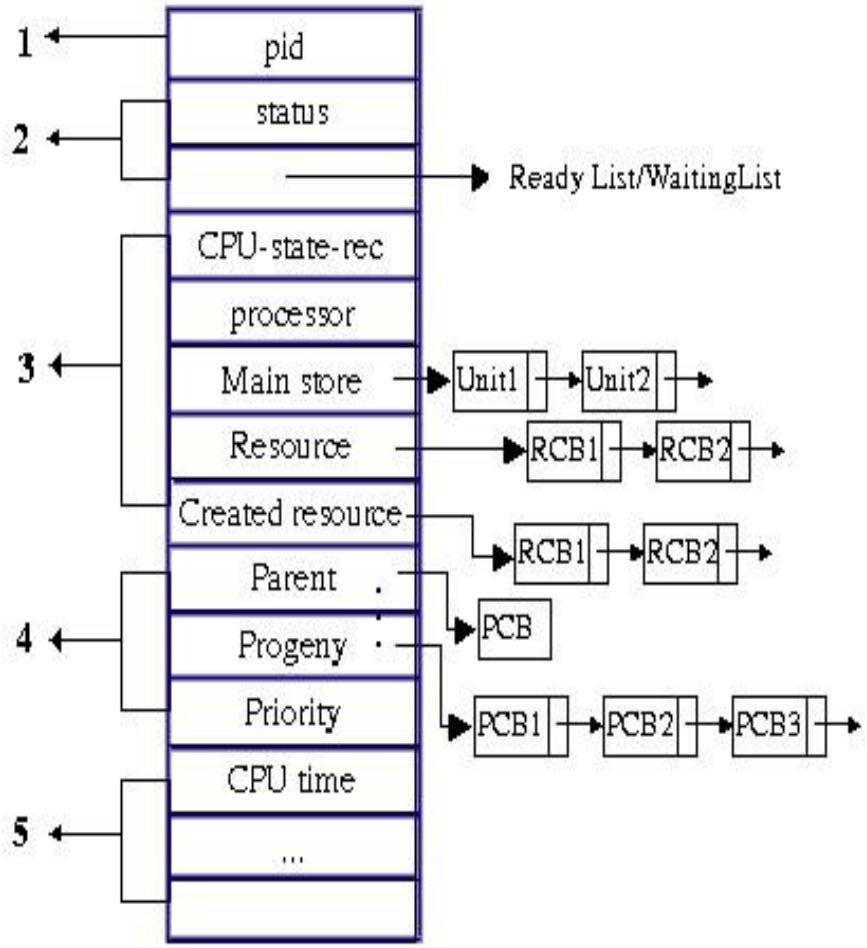


Chế độ xử lý của tiến trình

Hai chế độ xử lý

Cấu trúc dữ liệu khối quản lý tiến trình

- Khối quản lý tiến trình (PCB): là một vùng nhớ lưu trữ các thông tin mô tả cho tiến trình:
  - **Định danh của tiến trình (1)**
  - **Trạng thái tiến trình (2)**
  - **Ngữ cảnh của tiến trình (3)**
    - *Trạng thái CPU, Bộ xử lý, Bộ nhớ chính,Tài nguyên sử dụng,Tài nguyên tạo lập*
  - **Thông tin giao tiếp (4)**
    - *Tiến trình cha,Tiến trình con, Độ ưu tiên*
  - **Thông tin thống kê (5)**



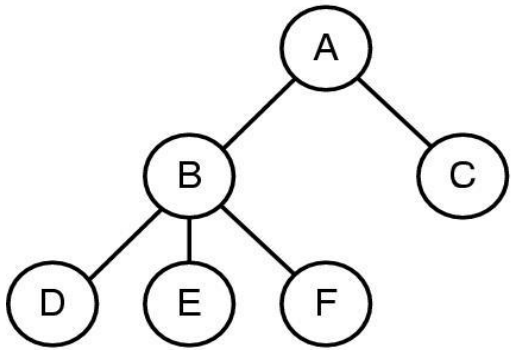
Khối mô tả tiến trình

Thao tác trên tiến trình

- tạo lập tiến trình (create)
- kết thúc tiến trình (destroy)
- tạm dừng tiến trình (suspend)
- tái kích hoạt tiến trình (resume)
- thay đổi độ ưu tiên tiến trình (change priority)

Tạo lập tiến trình

- định danh cho tiến trình mới phát sinh
- đưa tiến trình vào danh sách quản lý của hệ thống
- xác định độ ưu tiên cho tiến trình
- tạo PCB cho tiến trình
- cấp phát các tài nguyên ban đầu cho tiến trình



Kết thúc tiến trình

- thu hồi các tài nguyên hệ thống đã cấp phát cho tiến trình
- hủy tiến trình khỏi tất cả các danh sách quản lý của hệ thống
- hủy bỏ PCB của tiến trình

Cấp phát tài nguyên cho tiến trình

Khối quản lý tài nguyên

- Các mục tiêu của kỹ thuật cấp phát :
  - Bảo đảm một số lượng hợp lệ các tiến trình truy xuất đồng thời đến các tài nguyên không chia sẻ được.
  - Cấp phát tài nguyên cho tiến trình có yêu cầu trong một khoảng thời gian trì hoãn có thể chấp nhận được.
  - Tối ưu hóa sự sử dụng tài nguyên.

Điều phối tiến trình

- Hệ điều hành điều phối tiến trình thông qua bộ điều phối (scheduler) và bộ phân phối (dispatcher).
- Bộ điều phối sử dụng một giải thuật thích hợp để lựa chọn tiến trình được xử lý tiếp theo.
- Bộ phân phối chịu trách nhiệm cập nhật ngữ cảnh của tiến trình bị tạm ngưng và trao CPU cho tiến trình được chọn bởi bộ điều phối để tiến trình thực thi.
- **Mục tiêu điều phối**
  - Sự công bằng (Fairness)
  - Tính hiệu qủa (Efficiency)
  - Thời gian đáp ứng hợp lý (Response time)
  - Thời gian lưu lại trong hệ thống (TurnaroundTime)
  - Thông lượng tối đa (Throughput)

Các đặc điểm của tiến trình

- Tính hướng xuất / nhập của tiến trình (I/O-boundedness):
  - Nhiều lượt sử dụng CPU, mỗi lượt dùng thời gian ngắn.
- Tính hướng xử lý của tiến trình (CPU-boundedness):
  - Ít lượt sử dụng CPU, mỗi lượt dùng thời gian dài.
- Tiến trình tương tác *hay* xử lý theo lô
- Độ ưu tiên của tiến trình
- Thời gian đã sử dụng CPU của tiến trình
- Thời gian còn lại tiến trình cần để hoàn tất

Các nguyên lý điều phối

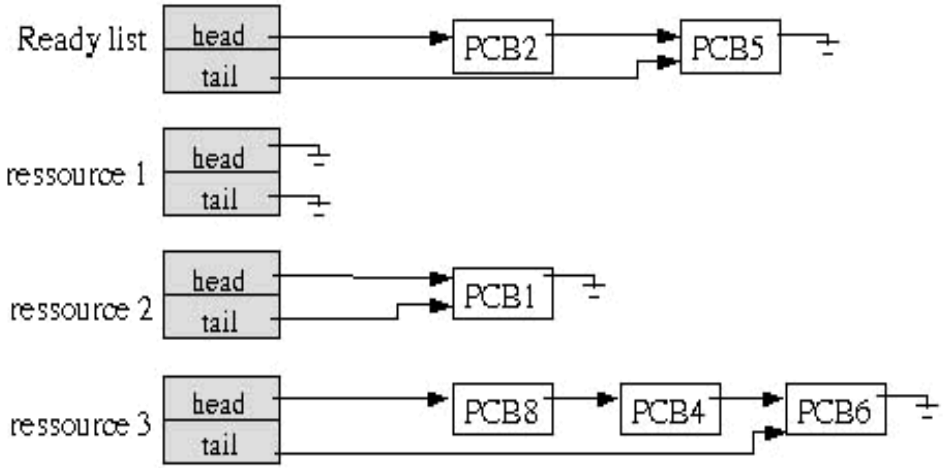
- *Điều phối độc quyền (preemptive)*
  - độc chiếm CPU
  - không thích hợp với các hệ thống nhiều người dùng
- *Điều phối không độc quyền (nopreemptive)*
  - tránh được tình trạng một tiến trình độc chiếm CPU
  - có thể dẫn đến các mâu thuẫn trong truy xuất-> cần phương pháp đồng bộ hóa thích hợp để giải quyết.
  - phức tạp trong việc phân định độ ưu tiên
  - phát sinh thêm chi phí khi chuyển đổi CPU qua lại giữa các tiến trình

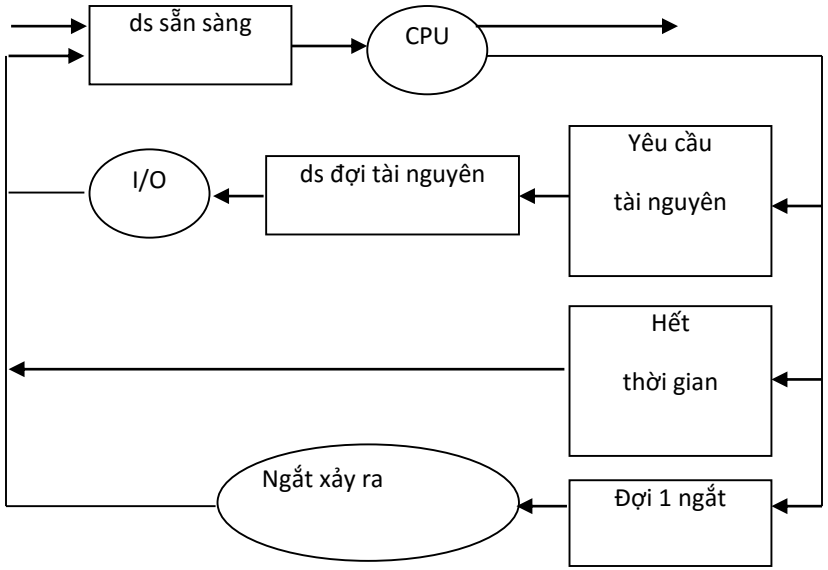
Thời điểm thực hiện điều phối

- running -> blocked
  - ví dụ chờ một thao tác nhập xuất hay chờ một tiến trình con kết thúc…
- running -> ready
  - ví dụ xảy ra một ngắt.
- blocked -> ready
  - ví dụ một thao tác nhập/xuất hoàn tất.
- Tiến trình kết thúc.
- Tiến trình có độ ưu tiên cao hơn xuất hiện
  - chỉ áp dụng đối với điều phối không độc quyền

Tổ chức điều phối - Các danh sách điều phối

- danh sách tác vụ (job list)
- danh sách sẵn sàng (ready list)
- danh sách chờ đợi (waiting list)





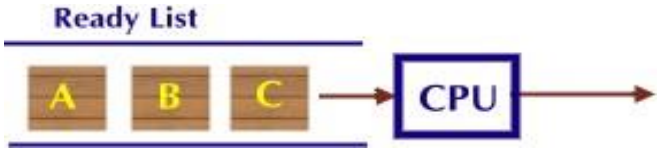
Các loại điều phối

- ***Điều phối tác vụ (job scheduling)***
  - chọn tác vụ nào được đưa vào bộ nhớ chính để thực hiện
  - quyết định mức độ đa chương
  - tần suất hoạt động thấp
- ***Điều phối tiến trình (process scheduling)***
  - Chọn một tiến trình ở trạng thái sẵn sàng (đã được nạp vào bộ nhớ chính, và có đủ tài nguyên để hoạt động) để cấp phát CPU cho tiến trình đó thực hiện
  - có tần suất hoạt động cao(1 lần/100 ms).
  - sử dụng các thuật toán tốt nhất

Các thuật toán điều phối

- ***Thuật toán FIFO***
- ***Thuật toán phân phối xoay vòng (Round Robin)***
- ***Thuật toán độ ưu tiên***
- ***Thuật toán công việc ngắn nhất (Shortest-job-first SJF)***
- ***Thuật toán nhiều mức độ ưu tiên***
- ***Chiến lược điều phối xổ số (Lottery)***

Thuật toán FIFO



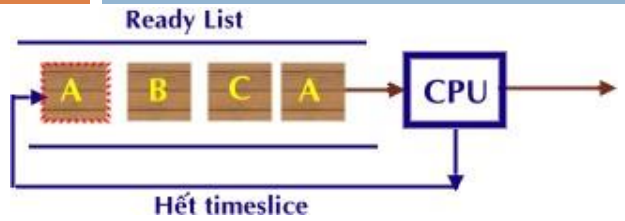
Điều phối theo nguyên tắc độc quyền

Tiến trình	Thời điểm vào RL	Thời gian xử lý
P1	0	24
P2	1	3
P3	2	3

P1	P2	P3
0	24	27
		30

- Thời gian chờ đợi được xử lý là 0 đối với P1, (24 - 1) với P2 và (27-2) với P3. Thời gian chờ trung bình là (0+23+25)/3 = 16 miliseconds.

Thuật toán phân phối xoay vòng (Round Robin)



Tiến trình	Thời điểm vào RL	Thời gian xử lý
P1	0	24
P2	1	3
P3	2	3

P1	P2	P3	P1	P1	P1	P1	P1	
0	4	7	10	14	18	22	26	30

quantum là 4 miliseconds,

- Thời gian chờ đợi trung bình sẽ là (6+3+5)/3 = 4.66 milisecondes.

Thuật toán độ ưu tiên

- Độ ưu tiên P2 > P3> P1

Tiến trình	Thời điểm vào RL	Độ ưu tiên	Thời gian xử lý
P1	0	3	24
P2	1	1	3
P3	2	2	3

Thuật giải độ ưu tiên độc quyền

P1	P2	P3
0	24	27 30

Thời gian chờ đợi trung bình sẽ là (0+23+25)/3 =16 milisecondes

Thuật giải độ ưu tiên không độc quyền

P1	P2	P3	P1
0	1	4	7 30

Thời gian chờ đợi trung bình sẽ là (6+0+2)/3 = 2.7 milisecondes

Thuật toán công việc ngắn nhất (Shortest-job-first SJF)

- t: thời gian xử lý mà tiến trình còn yêu cầu
- Độ ưu tiên p = 1/t

Tiến trình	Thời điểm vào RL	Thời gian xử lý
P1	0	6
P2	1	8
P3	2	4
P4	3	2

Thuật giải SJF độc quyền

P1	P4	P3	P2
0	6	8	12 20

Thời gian chờ đợi trung bình sẽ là (0+11+6+3)/4 = 5 milisecondes

Thuật giải SJF không độc quyền

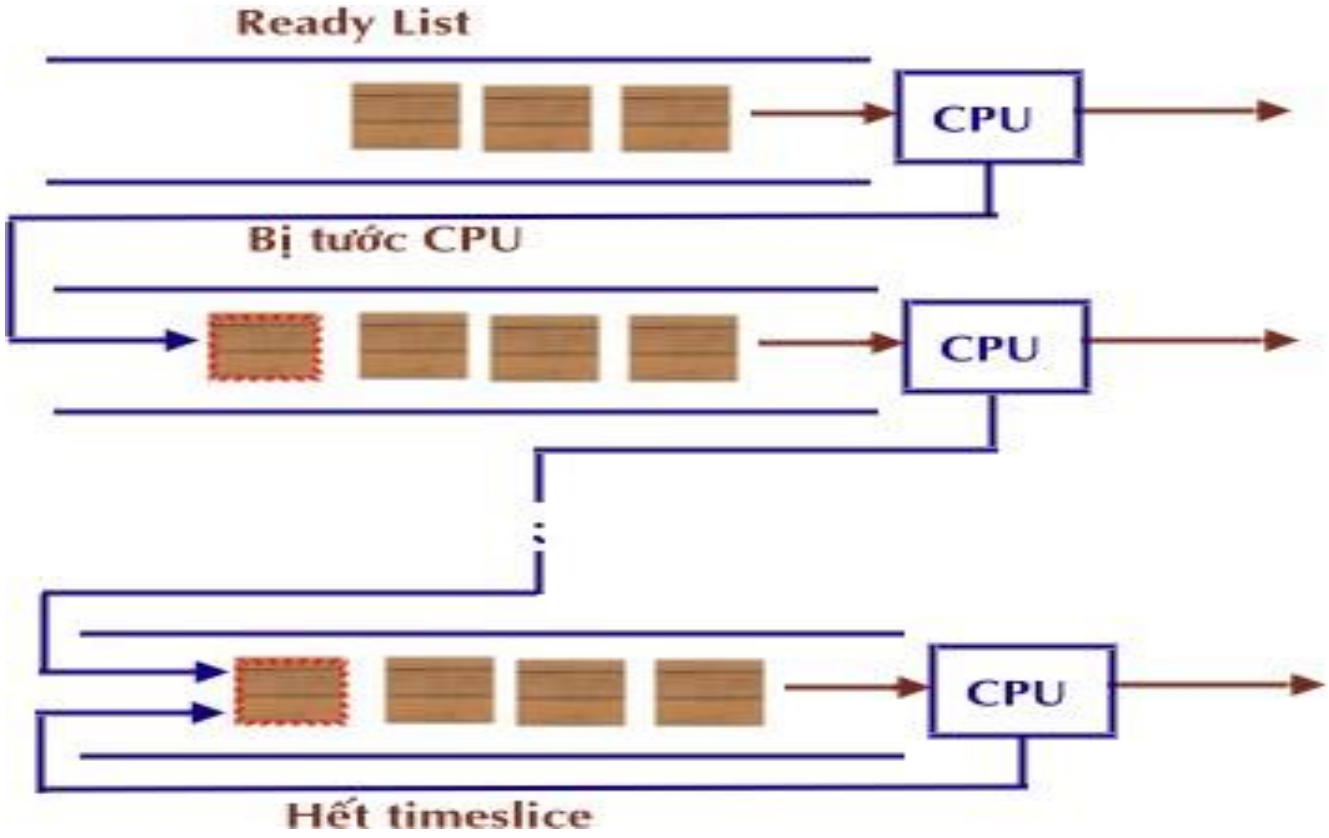
P1	P4	P1	P3	P2
0	3	5	8	12 20

Thời gian chờ đợi trung bình sẽ là (2+11+6+0)/4 = 4.75 milisecondes

Thuật toán nhiều mức độ ưu tiên

Danh sách sẵn sàng được chia thành nhiều danh sách.
Mỗi danh sách gồm các tiến trình có cùng độ ưu tiên và được áp dụng một giải thuật điều phối riêng

Điều phối theo nhiều mức ưu tiên xoay vòng (Multilevel Feedback)



Chiến lược điều phối xổ số (Lottery)

- Mỗi tiến trình được cấp một "vé số".
- HĐH chọn 1 vé "trúng giải", tiến trình nào sỡ hữu vé này sẽ được nhận CPU.
- Là giải thuật độc quyền.
- Đơn giản, chi phí thấp, bảo đảm tính công bằng cho các 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

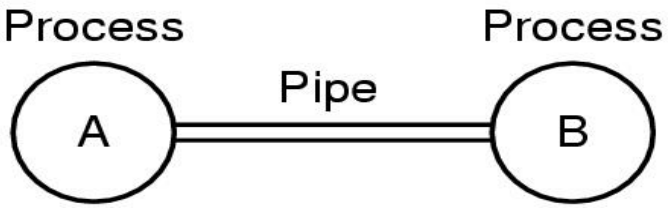
Liên lạc bằng tín hiệu (Signal)

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.

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.

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

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 một thông điệp tới port A
    - Receive(A, message): nhận một thông điệp từ port A
  - ***Liên lạc trực tiếp (direct communication)***
    - Send(P, message) : gởi một thông điệp đến tiến trình P
    - Receive(Q,message) : nhận một thông điệp từ tiến trình Q

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);
    }
}
```

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"
  - ***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 CÁC TIẾN TRÌNH

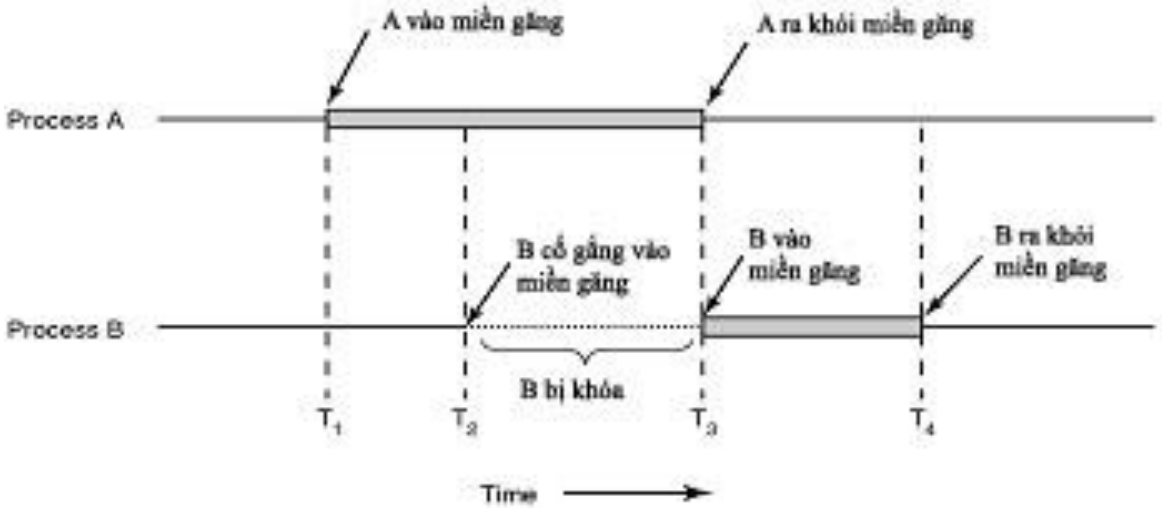
- bảo đảm các tiến trình xử lý song song không tác động sai lệch đến nhau.
- ***Yêu cầu độc quyền truy xuất (Mutual exclusion):***
  - tại một thời điểm, chỉ có một tiến trình được quyền truy xuất một tài nguyên không thể chia sẻ.
- ***Yêu cầu phối hợp (Synchronization):***
  - các tiến trình cần hợp 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".

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 ! “;



Các điều kiện cấn 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

Các nhóm giải pháp đồng bộ

- Busy Waiting
- Sleep And Wakeup
  - Semaphore
  - Monitor
- Message.

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)***

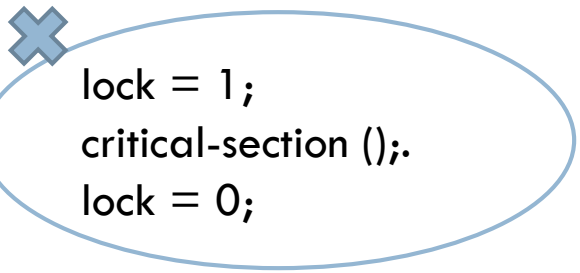
Thuật toán sử dụng biến cờ hiệu *(dùng cho nhiều tiến trình)*

- 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();
}
```



Vi phạm :"Hai tiến trình có thể cùng ở trong miền găng tại một thời điểm".

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.

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) -> Pi muốn vào miền găng và turn=i là đến lượt Pi.
- Để có thể vào được miền găng:
  - Pi đặt giá trị flag [i]=TRUE để thông báo nó muốn vào miền găng.
  - đặt turn=j để thử đề nghị tiến trình Pj vào miền găng.
- Nếu tiến trình Pj không quan tâm đến việc vào miền găng (flag [j] = FALSE), thì Pi có thể vào miền găng
  - nếu flag [j]=TRUE thì Pi phải chờ đến khi flag [j]=FALSE.
- Khi tiến trình Pi rời khỏi miền găng, nó đặt lại giá trị cho flag [i] là FALSE.

Thuật toán Peterson (*đoạn code*)

```
// tiến trình P0 (i=0)
while (TRUE)
{
        flag [0]= TRUE;//P0 thông báo là P0 muon vao mg
        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 ngoai mg
        noncritical_section ();
}
// tiến trình P1 (i=1)
while (TRUE)
{
        flag [1]= TRUE; //P1 thông báo là P1 muon vao mg
        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 ngoai mg
        Noncritical_section ();
}
```

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.

Sử dụng lệnh TSL (Test and Set Lock)

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

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

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

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

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

- *Sử dụng lệnh SLEEP VÀ WAKEUP*
- *Sử dụng cấu trúc Semaphore*
- *Sử dụng cấu trúc Monitors*
- *Sử dụng thông điệp*

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

Sử dụng lệnh SLEEP VÀ WAKEUP

```
int busy=FALSE;   // TRUE là có tiến trình trong miền găng, FALSE là không có tiến trình trong miền găng.
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 ();
}
```

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).

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; //chuyển P sang trạng thái bị khoá (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 của hệ thống
     }
}
```

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.

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 ();
}
```

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();
}
```

Nếu đặt Down và Up sai vị trí hoặc thiếu thì có thể bị sai.

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 ();
}
```

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

Sử dụng cấu trúc Monitors

- 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):

Sử dụng cấu trúc Monitors

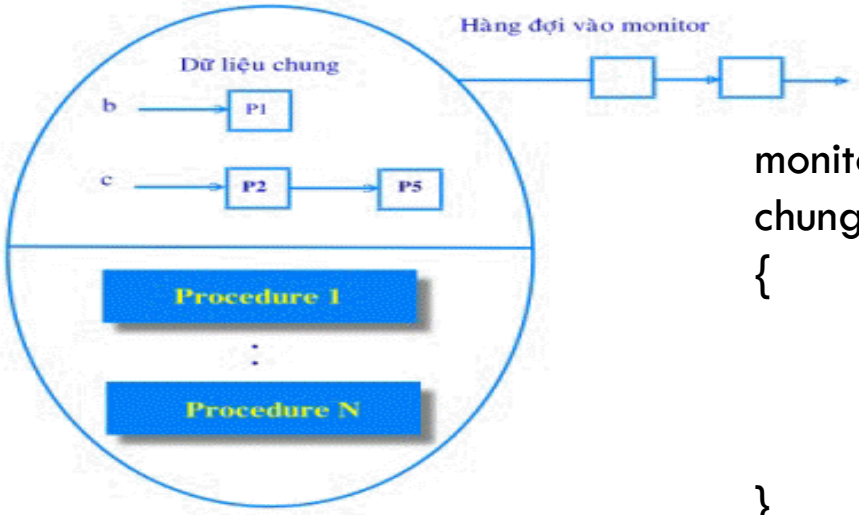
```
Wait(c)
{   status(P)= blocked;        //chuyển P sang trạng thái chờ
    enter(P,f(c));             //đặt P vào hàng đợi f(c) của biến điều kiện c
}

Signal(c)
{
    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.
    }
}
```

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

Signal(c): nếu có một tiến trình đang chờ trong hàng đợi của c, tái kích hoạt tiến trình đó và tiến trình gọi sẽ rời khỏi monitor. Nếu không có tiến trình nào đang chờ trong hàng đợi của c thì lệnh signal(c) bị bỏ qua.

Sử dụng cấu trúc Monitors



critical-section

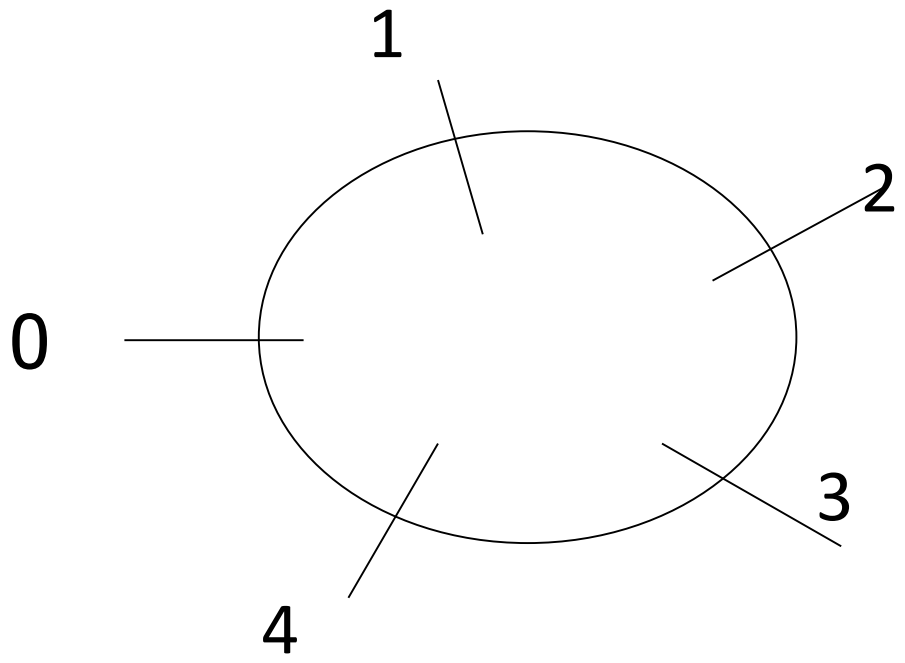
```
monitor <tên monitor > //khai báo monitor dùng chung cho các tiến trình
{
        <cac bien dung chung>;
        <các biến điều kiện>;
        <cac phuong thuc độc quyền>;
}
//tiến trình Pi:
while (1) //cấu trúc tiến trình thứ i
{
        Noncritical-section ();
        <ten monitor>.Phươngthức_i;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

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.

Giải pháp "busy and waiting" không phải thực hiện việc chuyển đổi ngữ cảnh trong khi giải pháp "sleep and wakeup" sẽ tốn thời gian cho việc này.

Monitors và Bài toán 5 triết gia ăn tối



Monitors và Bài toán 5 triết gia ăn tối

```
monitor philosopher
{    enum {thinking, hungry, eating} state[5];// cac bien dung chung cho các triết gia
     condition self[5];//cac bien dieu kien dung de dong bo viec an toi

     //cac pt doc quyen (cac mien gang), viec doc quyen do nnlt ho tro.
     void init();//phương thức khoi tao
     void test(int i); //phương thức kiểm tra điều kiện trước khi cho triết gia thứ i ăn
     void pickup(int i);              //phương thức lấy đũa
     void putdown(int i); //phương thức trả đũa
}
```

Monitors và Bài toán 5 triết gia ăn tối

```
void philosopher()//phương thức khởi tạo (constructor)
{   //gán trạng thái ban đầu cho các triết gia là "đang suy nghĩ"
    for (int i = 0; i < 5; i++) state[i] = thinking;
}
void test(int i)
{   //nếu tg_i đói và các tg bên trái, bên phải không đang ăn thì cho tg_i ăn
    if ( (state[i] == hungry) && (state[(i + 4) % 5] != eating) &&(state[(i + 1)
    % 5] != eating))
    {
            self[i].signal();//đánh thức tg_i, nếu tg_i đang chờ
            state[i] = eating; //ghi nhận tg_i đang ăn
    }
}
```

Monitors và Bài toán 5 triết gia ăn tối

```
void pickup(int i)
{
    state[i] = hungry; //ghi nhận tg_i đói
    test(i); //kiểm tra đk trước khi cho tg_i ăn
    if (state[i] != eating)  self[i].wait(); //doi tai nguyen
}
void putdown(int i)
{
    state[i] = thinking; //ghi nhận tg_i đang suy nghĩ
    test((i+4) % 5); //kt tg bên phải, nếu hợp lệ thì cho tg này ăn
    test((i+1) % 5);//kt tg bên  trái nếu hợp lệ thì cho tg này ăn
}
```

Monitors và Bài toán 5 triết gia ăn tối

```
// Cấu trúc tiến trình Pi  thực hiện viec ăn của triết gia thứ  i

philosopher pp; //bien monitor dung chung
Pi:
while (1)
{
    Noncritical-section ();
    pp.pickup(i); //pickup là miền găng và được truy xuất độc quyền
    eat(); //triet gia an
    pp.putdown(i);//putdown là miền găng và được truy xuất độc quyền
    Noncritical-section ();
}
```

Sử dụng thông điệp

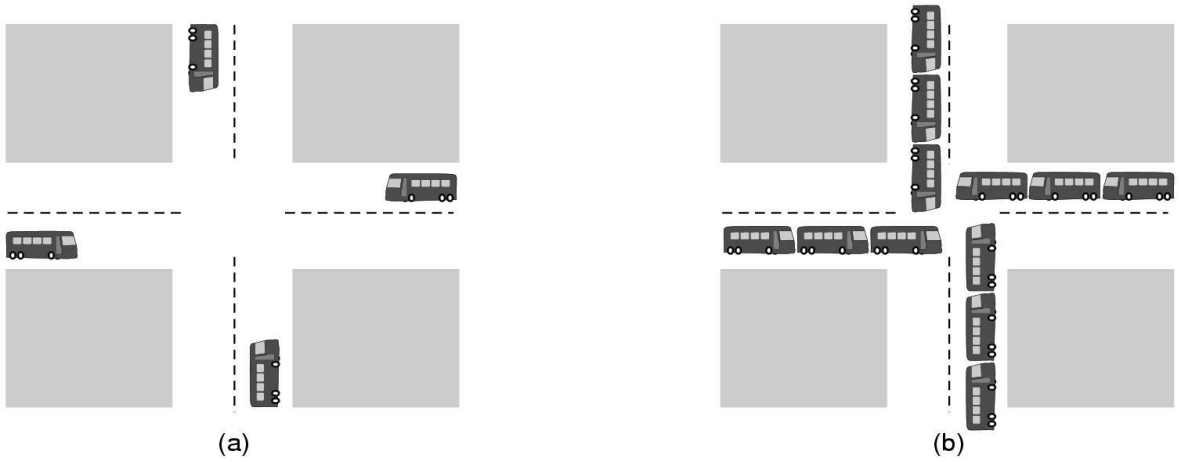
- Một tiến trình kiểm soát việc sử dụng tài nguyên và nhiều tiến trình khác yêu cầu tài nguyên.

```
while (1)
{
  Send(process controler, request message); //goi td yc tn va chuyen sang blocked
  Receive(process controler, accept message); //nhan td chap nhan su dung tn
  critical-section (); //doc quyen su dung tai nguyen dung chung
  Send(process controler, end message); //goi td ket thuc su dung tn.
  Noncritical-section ();
}
```

Trong những hệ thống phân tán, cơ chế trao đổi thông điệp sẽ đơn giản hơn và được dùng để giải quyết bài toán đồng bộ hóa.

TÌNH TRẠNG TẮC NGHẼN (DEADLOCKS)

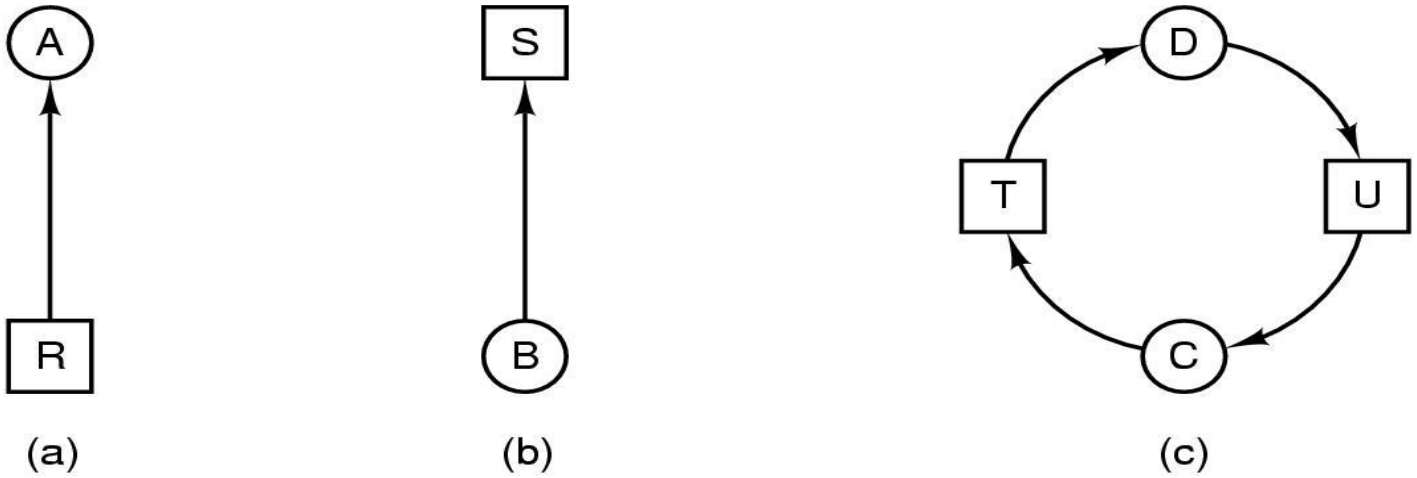
- Một tập hợp các tiến trình gọi là ở tình trạng tắc nghẽn nếu mỗi tiến trình trong tập hợp đều chờ đợi tài nguyên mà tiến trình khác trong tập hợp đang chiếm giữ.



Điều kiện xuất hiện tắc nghẽn

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

Đồ thị cấp phát tài nguyên



Ví dụ về tắc nghẽn

A

Request R
Request S
Release R
Release S

(a)

B

Request S
Request T
Release S
Release T

(b)

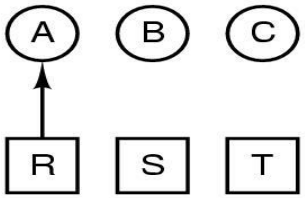
C

Request T
Request R
Release T
Release R

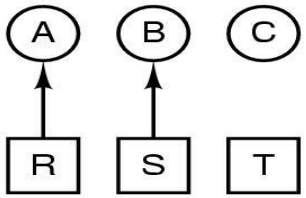
(c)

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

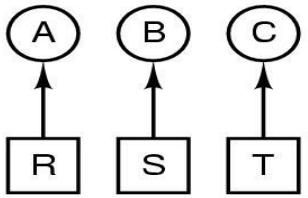
(d)



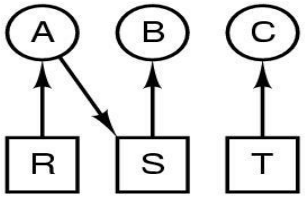
(e)



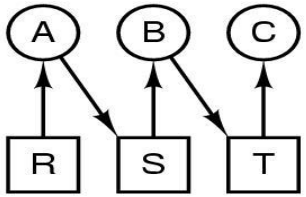
(f)



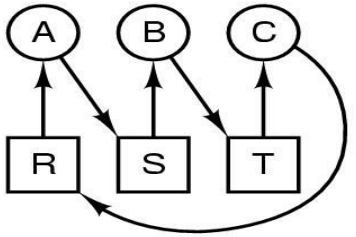
(g)



(h)



(i)



(j)

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.

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 sẽ cũ được cấp phát trở lại cùng lần 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 $F(Rj) > F(Ri)$.

Giải thuật cấp phát tài nguyên tránh tắc nghẽn

- Giải thuật xác định trạng thái an toàn
- Giải thuật Banker

Giải thuật xác định trạng thái an toàn

- int NumResources; //số tài nguyên
- int NumProcs; //số tiến trình trong hệ thống
- int Available[NumResources] //số lượng tài nguyên còn tự do

- int Max[NumProcs, NumResources];

  //Max[p,r]= nhu cầu tối đa của tiến trình p về tài nguyên r

- int Allocation[NumProcs, NumResources];

  //Allocation[p,r] = số tài nguyên r đã cấp phát cho tiến trình p

- int Need[NumProcs, NumResources];

  //Need[p,r] = Max[p,r] - Allocation[p,r]= số tài nguyên r mà tiến trình p còn cần sử dụng

- int Finish[NumProcs] = false;

  //Finish[p]=true là tiến trình p đã thực thi xong;

Giải thuật xác định trạng thái an toàn

- B1.
  - If ∃i:
    - Finish[i] = false //tiến trình i chưa thực thi xong
    - Need[i,j] <= Available[j], ∀j // Mọi nhu cầu về tn của tt i đều có thể đáp ứng
  - Do B2 else goto B3

- B2. Cấp phát mọi tài nguyên mà tiến trình i cần

  Allocation[i,j]= Allocation[i,j]+Need[i,j]; ∀j
  need[i,j]=0 ; ∀j
  Available[j]= Available[j] - Need[i,j];

  Cấp phát đủ tài nguyên cho tiến trình i

  Finish[i] = true;

  Đánh dấu tiến trình i thực hiện xong

  Available[j]= Available[j] + Allocation[i,j];

  Cập nhật lại số tài nguyên j khả dụng

  - goto B1

- B3. ∀i If Finish[i] = true -> « hệ thống ở trạng thái an toàn », else « không an toàn »

Giải thuật Banker

Pi yêu cầu kr thể hiện của tài nguyên r

- B1. If kr <= Need[i,r] ∀r, goto B2, else « lỗi »
- B2. If kr <= Available[r] ∀r, goto B3;
  else Pi « wait»
- B3. ∀r:

  Available[r]=Available[r]-kr;
  Allocation[i,r] =Allocation[i,r]+ kr;
  Need[i,r] = Need[i,r] – kr;
- B4: Kiểm tra trạng thái an toàn của hệ thống.

Ví dụ - cấp phát tài nguyên tránh tắc nghẽn

	Max			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	2	2	1	0	0	4	1	2
P2	6	1	3	2	1	1			
P3	3	1	4	2	1	1			
P4	4	2	2	0	0	2			

- Nếu tiến trình P2 yêu cầu 4 R1, 1 R3. hãy cho biết yêu cầu này có thể đáp ứng mà không xảy ra deadlock hay không?

Ví dụ - cấp phát tài nguyên tránh tắc nghẽn

- B0: Tính Need là nhu cầu còn lại về mỗi tài nguyên j của mỗi tiến trình i:
- Need[i,j]=Max[i,j]-Allocation[i,j]

	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	2	2	2	1	0	0	4	1	2
P2	4	0	2	2	1	1			
P3	1	0	3	2	1	1			
P4	4	2	0	0	0	2			

Ví dụ - cấp phát tài nguyên tránh tắc nghẽn

- B1+B2: yêu cầu tài nguyên của P2 thoả đk ở B1, B2.
- B3: Thử cấp phát cho P2, cập nhật tình trạng hệ thống

	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	2	2	2	1	0	0	0	1	1
P2	0	0	1	6	1	2			
P3	1	0	3	2	1	1			
P4	4	2	0	0	0	2			

Ví dụ - cấp phát tài nguyên tránh tắc nghẽn

- B4: Kiểm tra trạng thái an toàn của hệ thống.
  - Lần lượt chọn tiến trình để thử cấp phát:
    - Chọn P2, thử cấp phát, g/s P2 thực thi xong thu hồi
    - Available[j]= Available[j] + Allocation[i,j];

	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	2	2	2	1	0	0	6	2	3
P2	0	0	0	0	0	0			
P3	1	0	3	2	1	1			
P4	4	2	0	0	0	2			

Ví dụ - cấp phát tài nguyên tránh tắc nghẽn

+ Chọn P1									
	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	0	0	0	0	0	7	2	3
P2	0	0	0	0	0	0			
P3	1	0	3	2	1	1			
P4	4	2	0	0	0	2			

Ví dụ - cấp phát tài nguyên tránh tắc nghẽn

+ Chọn P3:									
	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	0	0	0	0	0	9	3	4
P2	0	0	0	0	0	0			
P3	0	0	0	0	0	0			
P4	4	2	0	0	0	2			

Ví dụ - cấp phát tài nguyên tránh tắc nghẽn

+ Chọn P4:

	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	0	0	0	0	0	9	3	6
P2	0	0	0	0	0	0			
P3	0	0	0	0	0	0			
P4	0	0	0	0	0	0			

Mọi tiến trình đã được cấp phát tài nguyên với yêu cầu cao nhất, nên trạng thái của hệ thống là an toàn, do đó có thể cấp phát các tài nguyên theo yêu cầu của P2.

Giải thuật phát hiện tắc nghẽn

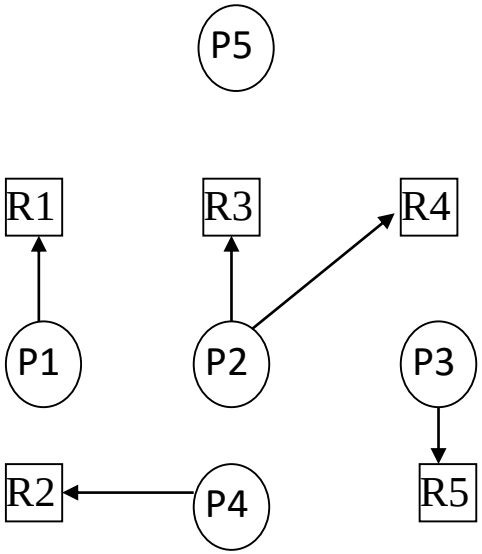
- Đối với ***Tài nguyên chỉ có một thể hiện***
- Đối với ***Tài nguyên có nhiều thể hiện***

Đối với *Tài nguyên chỉ có một thể hiện*

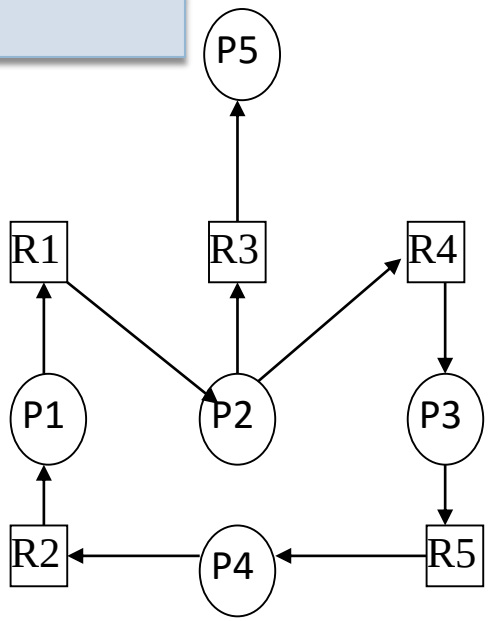
- Dùng đồ thị đợi tài nguyên (wait-for graph)
  - được xây dựng từ đồ thị cấp phát tài nguyên (resource-allocation graph) bằng cách bỏ những đỉnh biểu diễn loại tài nguyên.
  - cạnh từ Pi tới Pj : Pj đang đợi Pi giải phóng một tài nguyên mà Pj cần.
- Hệ thống bị tắc nghẽn nếu và chỉ nếu đồ thị đợi tài nguyên có chu trình.

Ví dụ

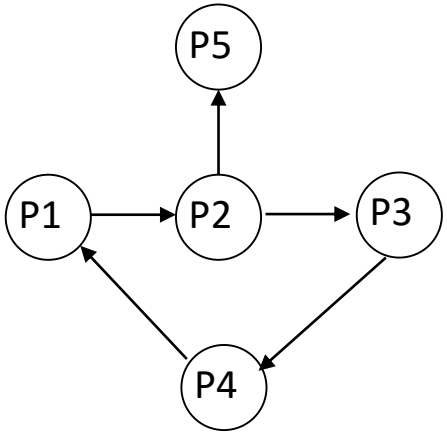
Tiến trình	Yêu cầu	Chiếm giữ
P1	R1	R2
P2	R3, R4	R1
P3	R5	R4
P4	R2	R5
P5		R3



đồ thị cấp phát tài nguyên



đồ thị thử cấp phát tài nguyên theo yêu cầu

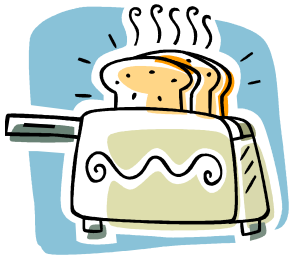


đồ thị đợi tài nguyên

Đối với *Tài nguyên có nhiều thể hiện*

- **Bước 1**: Chọn Pi đầu tiên sao cho có yêu cầu tài nguyên *có thể* được đáp ứng,
  - nếu không có thì hệ thống bị tắc nghẽn.
- **Bước 2**: *Thử cấp phát* tài nguyên cho Pi và kiểm tra trạng thái hệ thống,
  - nếu hệ thống an toàn thì tới Bước 3,
  - ngược lại thì quay lên Bước 1 tìm Pi kế tiếp.
- **Bước 3**: Cấp phát tài nguyên cho Pi. Nếu tất cả Pi được đáp ứng thì hệ thống không bị tắc nghẽn, ngược lại quay lại Bước 1.

Hiệu chỉnh tắc nghẽn



- **Hủy tiến trình trong tình trạng tắc nghẽn**
  - Hủy cho đến khi không còn chu trình gây tắc nghẽn
  - Dựa vào các yếu tố như độ ưu tiên, thời gian đã xử lý, số lượng tài nguyên đang chiếm giữ , số lượng tài nguyên còn yêu cầu thêm...
- **Thu hồi tài nguyên**
  - **Chọn lựa một nạn nhân**: tiến trình nào sẽ bị thu hồi tài nguyên? và thu hồi những tài nguyên nào?
  - **Trở lại trạng thái trước tắc nghẽn (rollback)**: khi thu hồi tài nguyên của một tiến trình, cần phải phục hồi trạng thái của tiến trình trở lại trạng thái gần nhất trước đó mà chưa bị tắc nghẽn.
  - **Tình trạng « đói tài nguyên »**: làm sao bảo đảm rằng không có một tiến trình nào luôn luôn bị thu hồi tài nguyên?