# Race condition

# 

Happen when many processes simultaneously work with shared data

# 1. IPC

In some situations, processes need to communicate with each other:

* To send/receive data (web browser – web server)
* To control the other process
* To synchronize with each other

IPC is implemented differently among OSes Linux: message queue, semaphore, shared segment, …

IPC can be divided into 2 categories

* IPC among processes within the same system
  + Linux: pipe, named pipe, file mapping, …
* IPC among processes in different systems
  + Remote Procedure Call (RPC), Socket, Remote Method Invocation (RMI), ...

# 2. Synchronization (đồng bộ tiến trình)

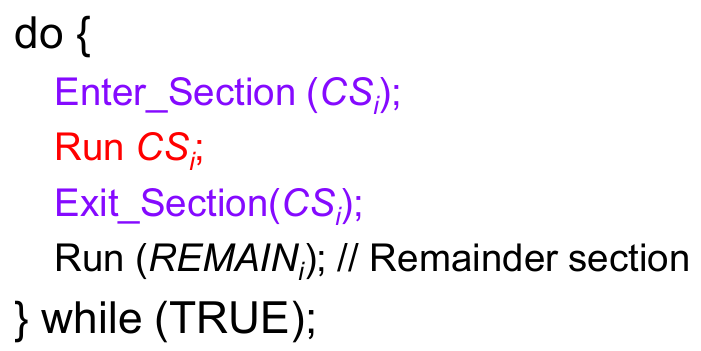
**Process synchronization** refers to the idea that multiple processes are to join up or **handshake at a certain point**, in order to reach an agreement or c**ommit to a certain sequence of actions.**

# 3. Critical section

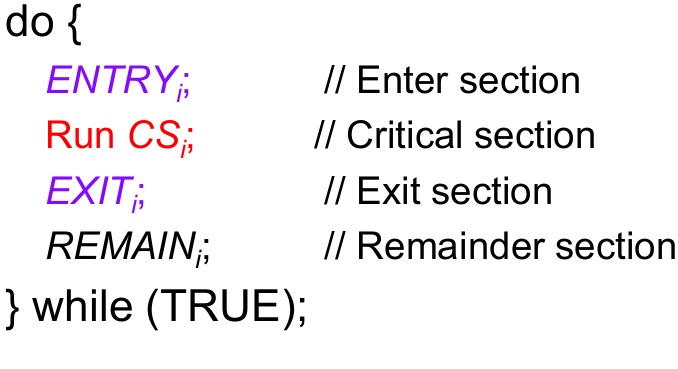
In concurrent programming a critical section is a piece of code that **accesses** **a shared resource** (data structure or device) that must not be concurrently accessed by **more than one thread of execution**. A critical section will usually terminate in fixed time, and a thread, task or

process will have to wait a fixed time to enter it (aka bounded waiting). Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore.

**Common structure**



**Short description**

****

## **CS** must satisfy 3 **conditions:**

1.**Mutual Exclusion** (loại trừ lẫn nhau)

* If a process is in its critical section, then no other processes can be in their critical sections

2. **Progress** (liên tiếp)

* If no process is in its **CSi** other processes waiting to enter their critical section, then the selection of the process to enter the critical section cannot be postponed(hoãn) indefinitely (vô hạn)

3. **Bounded Waiting**

* No process has to wait indefinitely(vô hạn) to enter its critical

section

* Loại trừ lẫn nhau (mutual exclusion): Nếu P i đang thực hiện CS i thì P j không thể thực hiện CS j ∀j≠i.
* Tiến triển (progress): Nếu không có tiến trình P i nào thực hiện CS i và có m tiến trình P j1 , P j2 , ..., P jm muốn thực hiện CS j1 , CS j2 , ..., CS jm thì chỉ có các tiến trình không thực hiện REMAIN jk (k=1,...,m) mới được xem xét thực hiện CS jk .
* Chờ có giới hạn (bounded waiting): sau khi một tiến trình P i có yêu cầu vào CS i và trước khi yêu cầu đó được chấp nhận, số lần các tiến trình P j (với j≠i) được phép thực hiện CS j phải bị giới hạn.

Which is the purpose of the first condition of critical section CS ?

* It supports the priority of process
* It ensures the correct use of the shared resource
* It tries to utilize the shared resource effectively
* It makes the implementation of OS simplera

## Giải pháp 1: Peterson’s Solution

* Solution for two processes
* The two processes share two variables:
  + int turn; // with the value of 0 or 1
  + Boolean flag[2]
* The variable turn indicates whose turn it is to enter the critical section
  + If turn==i then P i is in turn to run its CS i
* The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P i is ready!

Program P i :

do {

flag[i] = TRUE;

turn = j;

while (flag[j] && turn == j) ;

**CS i ;**

flag[i] = FALSE;

REMAIN i ;

} while (1);

# 

# SEMAFORE

Semaphore is an integer, can be only access through two atomic operators wait (or P) and signal (or V).

* P: proberen – check (in Dutch)
* V: verhogen – increase (in Dutch)

Processes can share a semaphore

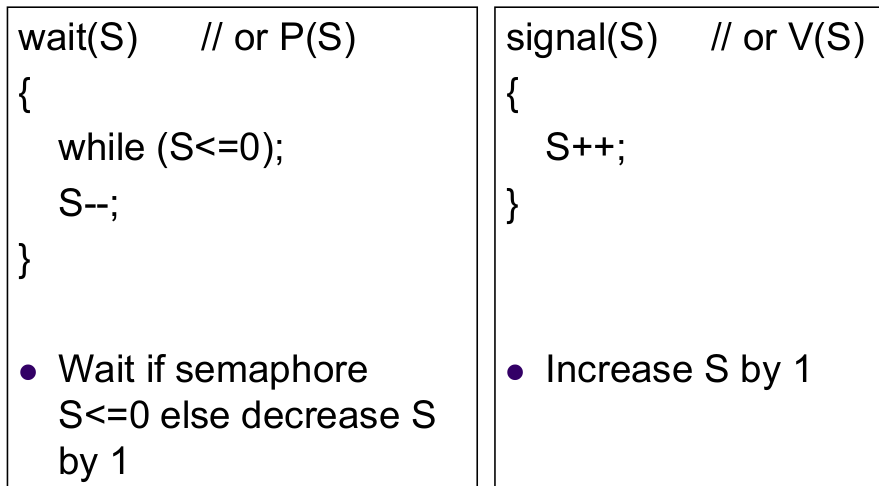
Atomic operators guarantee the consistency

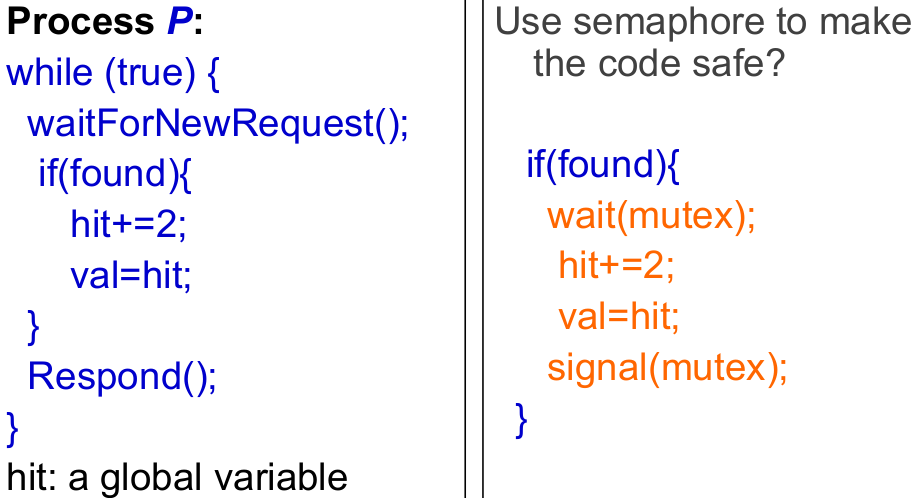
Semaphore là một biến nguyên, nếu không tính đến toán tử khởi tạo, chỉ có thể truy cập thông qua hai toán tử nguyên tố là wait (hoặc P) và

signal (hoặc V).

Các tiến trình có thể sử dụng chung semaphore

Các toán tử là nguyên tố để đảm bảo không xảy ra trường hợp như ví dụ đồng bộ hóa đã nêu



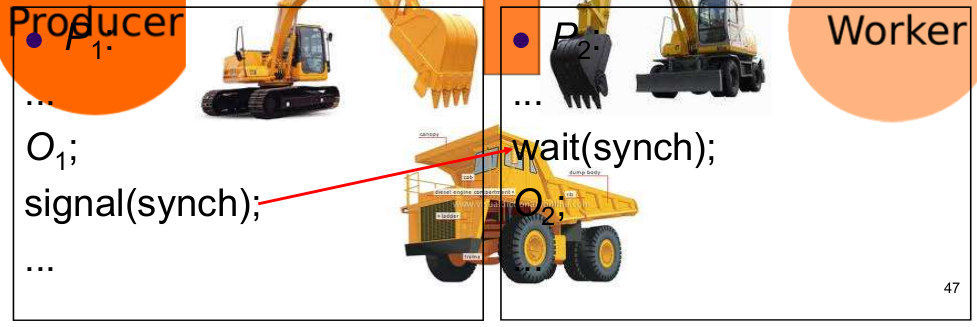


**Using semaphore (cont’d)**

P 1 needs to do O 1 ; P 2 need to do O 2 ; O 2 can

only be done after O 1

Solution: use a **semaphore synch = 0**



TH chạy sen vào, hai bên chạy song song:

* chưa chạy xong O1 và signal => synch = 0
* nếu vào wait(synch) thì sẽ đơị mãi đến khi signal tăng ở P1.

## Semaphore implementation

* In the above semaphore implementation
  + Use busy waiting (while loop)
  + Resource wasting
* Atomic operators
  + When a process called wait(), it will be blocked if the semaphore is not free
* This type of semaphore is called **spinlock**
* Other wait() implementation just returns true/false and does not block the calling process

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

## Semaphore V2:

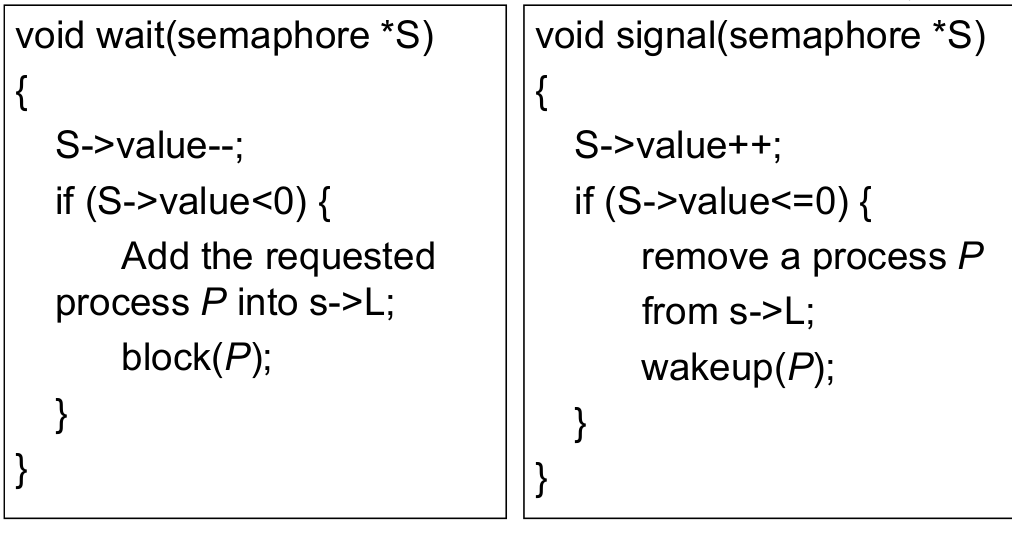
* Remove the busy waiting loop by using block
* To restored a blocked process, use wakeup
* Semaphore data structure

typedef struct {

int value; // value of semaphore

struct process \*L; //waiting process

} semaphore;



## Binary semaphore

Semaphore only has the value of 0 or 1

Other semaphore type is counting semaphore

## Bài toán 1: Bounded-Buffer Problem (Bài toán bộ đệm hữu hạn)

* N buffers, each can hold one item
* Semaphore mutex initialized to the value 1
* Semaphore full initialized to the value 0
* Semaphore empty initialized to the value N.

P : tạo và ghi DL và buffer

C : lấy DL từ buffer xử lý.

Ràng buộc:

- C lấy DL khi có data mới

- P ghi tối đa N phần tử

|  |  |
| --- | --- |
| Write process P:  do {  wait(empty);  wait(mutex);  Write (item);  signal(mutex);  signal(full);  } while (TRUE); | Read process Q:  do {  wait(full);  wait(mutex);  Read(item);  signal(mutex);  signal(empty);  } while (TRUE); |

## Bài toán 2: Readers-writers problem

Problem:

* Allow multiple readers to read at the same time
* Only one writer can access the shared data at the a time

Init:

* Semaphore *wrt* = 1
  + Used to manage write access
* Integer readcount = 0 to count the
  + number of readers that are reading
* Semaphore mutex initialized = 1
  + Used to manage readcount access

|  |  |
| --- | --- |
| Process writer P w :  do {  wait(wrt);  write(data\_set);  signal(wrt);  }while (TRUE); | Process reader P r :  do {  wait(mutex);  readcount++;  if (readcount ==1) wait(wrt);  signal(mutex);  read(data\_set);  wait(mutex);  readcount--;  if (readcount ==0) signal(wrt);  signal(mutex);  } while (TRUE); |

## Bài toán 3- Dining-Philosophers Problem ( bữa ăn của các triết gia)

* Use semaphore to handle chopstick access
* semaphore chopstick[5];
* Solution is provided as in the text box

Code of philosopher i:

do {

wait(chopstick[i]);

wait(chopstick[(i+1)%5];

Eat(i);

signal(chopstick[i]);

signal(chopstick[(i+1)%5];

Think(i);

} while (TRUE);

What value chopstick[i] is initialized?

* 1
* 2
* 0
* 5

Is there any problem with the solution?

* No problem
* Only one philosopher can eat at a time
* Only three philosophers can eat at a time
* No philosopher could eat in case each takes a chopstick and waits for the second one

Which of the following is incorrect about the solution to the above problem?

* No solution available
* Create an order of philosophers to eat
* Create an order of philosophers to think
* Allow at most 4 philosophers to request to eat at a time

## LIMIT of semaphore

1. Compare the two code snippets:

|  |  |
| --- | --- |
| Snippet 1  ...  wait(mutex);  //Critical section  signal(mutex);  ... | Snippet 2  ...  signal(mutex);  //Critical section  wait(mutex);  ... |

What is the problem of the two code snippets?

* Snippet 1 has problem
* Snippet 2 has problem
* Both snippets have problem
* No problem at all

2. Semaphores need correct calls to wait and signal

3. Incorrect use of semaphore may lead to deadlock

4. Even correct use of semaphores may lead to deadlock, in some cases

GIải pháp LIMIT

### What is monitor?

* Monitor means to supervise
* It is a type of construct in a high level programming language for synchronization purpose
* Monitor was studied and developed to overcome the limitations of semaphores

### 

### **A monitor usually has:**

1. Member variables as shared resources
2. A set of procedures which operate on the shared resources
3. Exclusive lock
4. Constraints to manage race condition

This description of monitor is like a class

### A sample monitor type

monitor monitor\_name {

//Shared resources

procedure P1(...) { ...

}

procedure P2(...) { ...

}

...

procedure Pn(...) { ...

}

initialization\_code (..) { ...

}

}

* Monitor must be implemented so that
  + only one process can enter the monitor at a time (mutual exclusive)
  + programmer do not need to write code for this
* Other monitor implementation
  + have more synchronization mechanism
  + add condition variable

### Kiểu condition

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

### Đặc điểm của x.signal()

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

### Signal wait/continue

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

## 

## Bài toán Ăn tối.. với monitor

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

monitor dp {

enum {thinking, hungry, eating} state[5];

condition self[5];

void pickup(int i) {

state[i] = hungry;

test(i);

if (state[i] != eating) self[i].wait();

}

}

void putdown(int i) {

state[i] = thinking;

test((i+4)%5);

test((i+1)%5);

}

initialization\_code() {

for (int i=0;i<5;i++) state[i] = thinking;

}

void test(int i) {

if ((state[(i+4)%5] != eating) &&

(state[i] == hungry) &&

(state[(i+1)%5] != eating)) {

state[i] = eating;

self[i].signal();

}

}

**Each philosopher invokes the operations pickup() and putdown()** in the following sequence

dp.pickup (i)

EAT

dp.putdown (i)

**Monitor Implementation Using Semaphores**

Variables

semaphore mutex; // (initially = 1)

semaphore next; // (initially = 0)

int next-count = 0;

Each procedure F will be replaced by

wait(mutex);

...

body of F;

...

if (next-count > 0)

signal(next)

else

signal(mutex);

Mutual exclusion within a monitor is ensured.

**Monitor Implementation**

For each condition variable x, we have:

semaphore x-sem; // (initially = 0)

int x-count = 0;

The operation x.wait can be implemented as:

x-count++;

if (next-count > 0)

signal(next);

else

signal(mutex);

wait(x-sem);

x-count--;

The operation x.signal can be implemented as:

if (x-count > 0) {

next-count++;

signal(x-sem);

wait(next);

next-count--;

}

## Linux Synchronization

Linux:

* disables interrupts to implement short critical sections

Linux provides:

* semaphores
* spin locks

## Pthreads Synchronization

* pthreads API is OS-independent
* It provides:
  + mutex locks
  + condition variables
* Non-portable extensions include:
  + read-write locks
  + spin locks

------------------------------------------------------------------------------------------------------ Chương 3:

# DEADLOCK:

* A set of blocked processes each
  + holding a resource and
  + waiting to acquire a resource held by another
* process in the set
* There must be a circular wait in this set

## Bế tắc xuất hiện nếu 4 điều kiện sau xuất hiện đồng thời (điều kiện cần):

C1: Loại trừ lẫn nhau (mutual exclusion)

* Một tài nguyên bị chiếm bởi một tiến trình, và không tiến trình nào khác có thể sử dụng tài nguyên này

C2: Giữ và chờ (hold and wait)

* Một tiến trình giữ ít nhất một tài nguyên và chờ một số tài nguyên khác rỗi để sử dụng. Các tài nguyên này đang bị một tiến trình khác chiếm giữ

C3: Không có đặc quyền (preemption)

* Tài nguyên bị chiếm giữ chỉ có thể rỗi khi tiến trình “tự nguyện” giải phóng tài nguyên sau khi đã sử dụng xong.

C4: Chờ vòng (circular wait)

* Một tập tiến trình {P 0 , P 1 , ..., P n } có xuất hiện điều kiện “chờ vòng” nếu P 0 chờ một tài nguyên do P 1 chiếm giữ, P 1 chờ một tài nguyên khác do P 2 chiếm giữ, ..., P n-1 chờ tài nguyên do P n chiếm giữ và P n chờ tài nguyên do P 0 chiếm giữ

## **Qui trình sử dụng tài nguyên (Each process utilizes a resource:)**

Một tiến trình thường sử dụng tài nguyên

theo các bước tuần tự sau:

* Xin phép sử dụng (request)
* Sử dụng tài nguyên (use)
* Giải phóng tài nguyên sau khi sử dụng (release)

## Các phương pháp xử lý bế tắc

Một cách tổng quát, có 3 phương pháp:

* Sử dụng một giao thức để hệ thống không bao giờ rơi vào trạng thái bế tắc: Deadlock prevention (ngăn chặn bế tắc) hoặc Deadlock avoidance (tránh bế tắc)
* Có thể cho phép hệ thống bị bế tắc, phát hiện bế tắc và khắc phục nó
* Bỏ qua bế tắc, xem như bế tắc không bao giờ xuất hiện trong hệ thống (Giải pháp này dùng trong nhiều hệ thống, ví dụ Unix, Windows!!)

Ngăn chặn bế tắc:

* Ngăn chặn bế tắc (deadlock prevention) là phương pháp xử lý bế tắc, không cho nó xảy ra bằng cách làm cho ít nhất một điều kiện cần của bế tắc là C1, C2, C3 hoặc C4 không được thỏa mãn (không xảy ra)
* Ngăn chặn bế tắc theo phương pháp này có tính chất tĩnh (statically)

### 1. Ngăn chặn “loại trừ lẫn nhau”

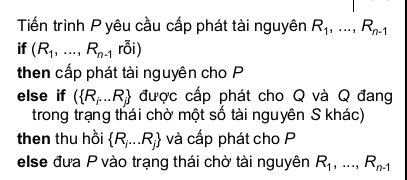
* C1 (Loại trừ lẫn nhau): là điều kiện bắt buộc cho các tài nguyên không sử dụng chung được → Khó làm cho C1 không xảy ra vì các hệ thống luôn có các tài nguyên không thể sửdụng chung được

### 2. Ngăn chặn “giữ và chờ”

* C2 (Giữ và chờ): Có thể làm cho C2 không xảy ra bằng cách đảm bảo:
  + Một tiến trình luôn yêu cầu cấ phát tài nguyên chỉ khi nó không chiếm giữ bất kỳ một tài nguyên nào, hoặc
  + Một tiến trình chỉ thực hiện khi nó được cấp phát toàn bộ các tài nguyên cần thiết

### 3. Ngăn chặn “không có đặc quyền”

Để ngăn chặn không cho điều kiện này xảy ra, có thể sử dụng giao thức sau:

* 

### 3. Ngăn chặn “chờ vòng”

* Một giải pháp ngăn chặn chờ vòng là đánh số thứ tự các tài nguyên và bắt buộc các tiến trình yêu cầu cấp phát tài nguyên theo số thứ tự tăng dần
* Giao thức ngăn chặn chờ vòng:
  + Khi tiến trình P không chiếm giữ tài nguyên nào, nó có thể yêu cầu cấp phát nhiều thể hiện của một tài nguyên R i bất kỳ
  + Sau đó P chỉ có thể yêu cầu các thể hiện của tài nguyên R j nếu và chỉ nếu f(R j ) > f(R i ). Một cách khác, nếu P muốn yêu cầu cấp phát tài nguyên R j , nó đã giải phóng tất cả các tài nguyên R i thỏa mãn f(R i )≥f(R j )
  + Nếu P cần được cấp phát nhiều loại tài nguyên, P phải lần lượt yêu cầu các thể hiện của từng tài nguyên đó

## Ưu nhược điểm của ngăn chặn giải pháp bế tắc

* Nhược điểm:
  + Giảm khả năng tận dụng tài nguyên và giảm thông lượng của hệ thống
  + Không mềm dẻo

## Tránh bế tắc (Deadlock avoidance)

* Tránh bế tắc là phương pháp sử dụng thêm các thông tin về phương thức yêu cầu cấp phát tài nguyên để ra quyết định cấp phát tài nguyên sao cho bế tắc không xảy ra.
* Có nhiều thuật toán theo hướng này
  + Thuật toán đơn giản nhất và hiệu quả nhất là:
  + Mỗi tiến trình P đăng ký số thể hiện của mỗi loại tài nguyên mà P sẽ sử dụng. Khi đó hệ thống sẽ có đủ thông tin để xây dựng thuật toán cấp phát không gây ra bế tắc
  + Các thuật toán như vậy kiểm tra trạng thái cấp phát tài nguyên một cách “động” để đảm bảo điều kiện chờ vòng không xảy ra
  + Trạng thái cấp phát tài nguyên được xác định bởi số lượng tài nguyên rỗi, số lượng tài nguyên đã cấp phát và số lượng lớn nhất các yêu cầu cấp phát tài nguyên của các tiến trình
* Hai thuật toán sẽ nghiên cứu: Thuật toán đồ thị cấp phát tài nguyên và thuật toán banker

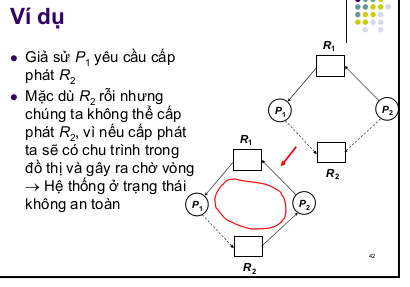
## Trạng thái an toàn (safe-state)

* Một trạng thái (cấp phát tài nguyên) được gọi là an toàn nếu hệ thống có thể cấp phát tài nguyên cho các tiến trình theo một thứ tự nào đó mà vẫn tránh được bế tắc, hay
* Hệ thống ở trong trạng thái an toàn nếu và chỉ nếu tồn tại một thứ tự an toàn (safe-sequence)

Thứ tự an toàn

# Thuật toán cấp phát tài nguyên

* Các tài nguyên 1 thể hiện (Cạnh Claim), k tạo chu trình



# Thuật toán Banker

Tài nguyên nhiều thể hiện, kém hiệu quả hơn thuật toán cấp phát tài nguyên