Nội dung

- Thread (tiểu tiến trình)
- Giao tiếp giữa các tiến trình
- Deadlocke

Khái niệm về Thread

- Trong các hệ điều hành truyền thống, mỗi tiến trình có không gian địa chỉ và một **thead** duy nhất.
- Trong thực tế, người dùng mong muốn có nhiều thread điều khiển trong cùng không gian địa chỉ chạy song song, như các tiến trình riêng biệt.
- Thread có một số tính chất của các tiến trình, nên Thread được gọi là **lightweight processes**

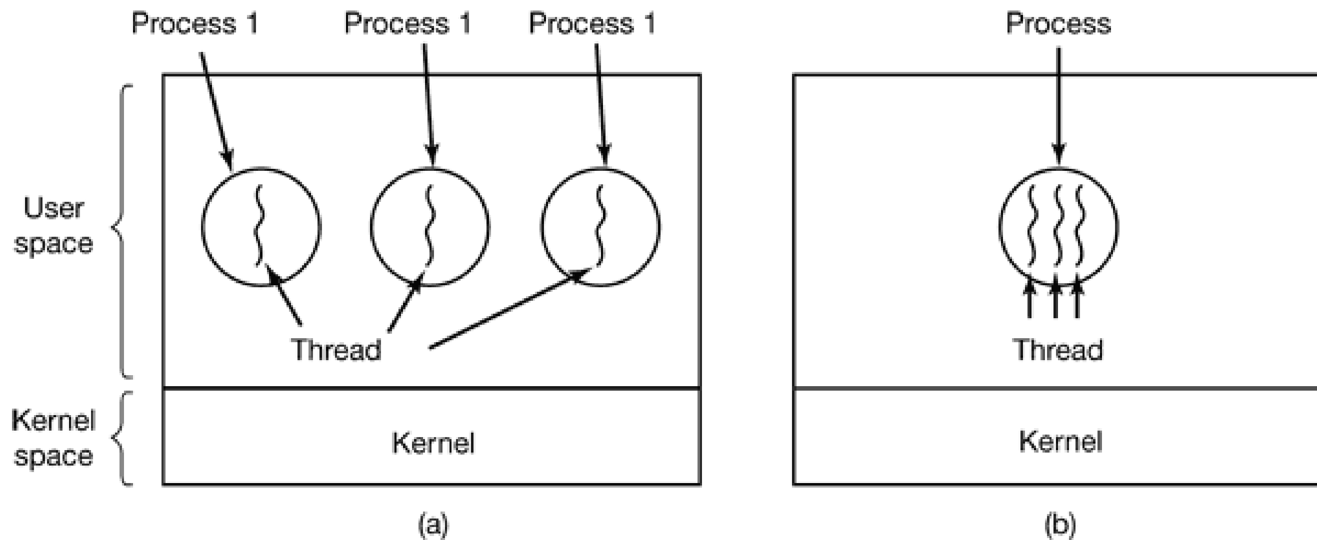
Khái niệm về Thread

Khái niệm Thread:

- Một tiến trình có thể có nhiều thread, mỗi thread thực hiện một chức năng cụ thể trong cùng không gian địa chỉ của tiến trình.
- Mỗi thread bao gồm:
  - ***Một bộ đếm chương trình (Program counter):*** theo dõi các chỉ thị thực thi tiếp theo.
  - ***Thanh ghi***: chứa các biến làm việc hiện hành
  - ***Stack***: chứa lịch sử thực hiện, với một frame cho mỗi thủ tục được gọi.

Khái niệm multithreading

- Trạng thái nhiều threads chạy song song trong cùng một tiến trình gọi là **multithreading**

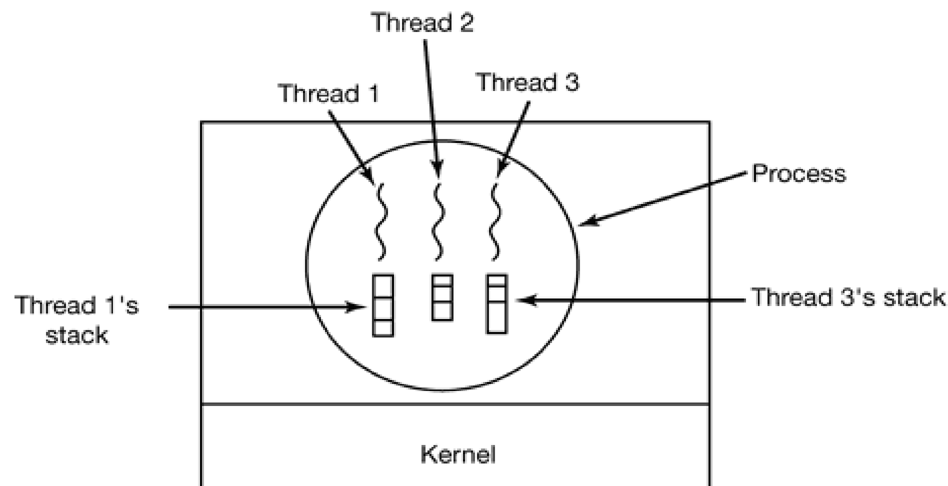


(a) 3 process , mỗi process có 1 thread.

(b) 1 process với 3 threads.

Trạng thái của thread

- Một thread có thể có các trạng thái: ***running, blocked, ready, terminated***.
- Mỗi thread có stack riêng, chứa một frame cho mỗi thủ tục được gọi, gồm các biến cục bộ và địa chỉ trả về khi thủ tục kết thúc.



Các thủ tục của thread

- *pthread_exit:* Khi thread kết thúc công việc.
- *pthread_wait:* Trong một hệ thống thread, một thread có thể chờ một thread khác.
- *pthread_yield*, cho phép một thread tự nguyện từ bỏ CPU cho một thread khác chạy.

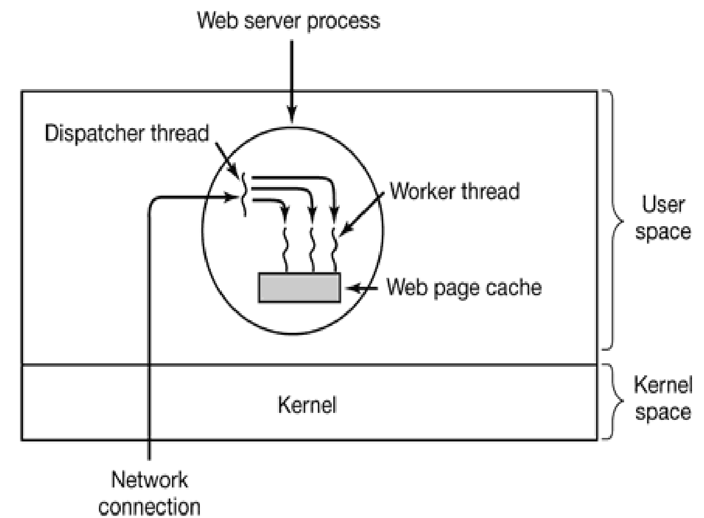
Lý do sử dụng thread

- Chia sẻ một không gian địa chỉ và tất cả các dữ liệu của thread
- Thread không có tài nguyên gắn liền với nó, do đó dễ dàng tạo ra và hủy hơn so với tiến trình.
- Tăng hiệu năng, khi có máy tính và I/O đáng kể, thread cho phép các hoạt động chồng lên nhau, làm tăng tốc độ của ứng dụng.

Ví dụ về multithreading

❑ Web server multithreading

- **Dispatcher**: đọc yêu cầu từ mạng, nó chọn một **worker thread** rỗi, đánh thức và chuyển nó từ trạng thái block sang trạng thái **ready**.
- Khi **worker thread** được đánh thức, nó kiểm tra trang web yêu cầu trong catch, nếu không có, nó bắt đầu đọc đĩa để có được những trang yêu cầu và block cho đến khi hoàn tất.



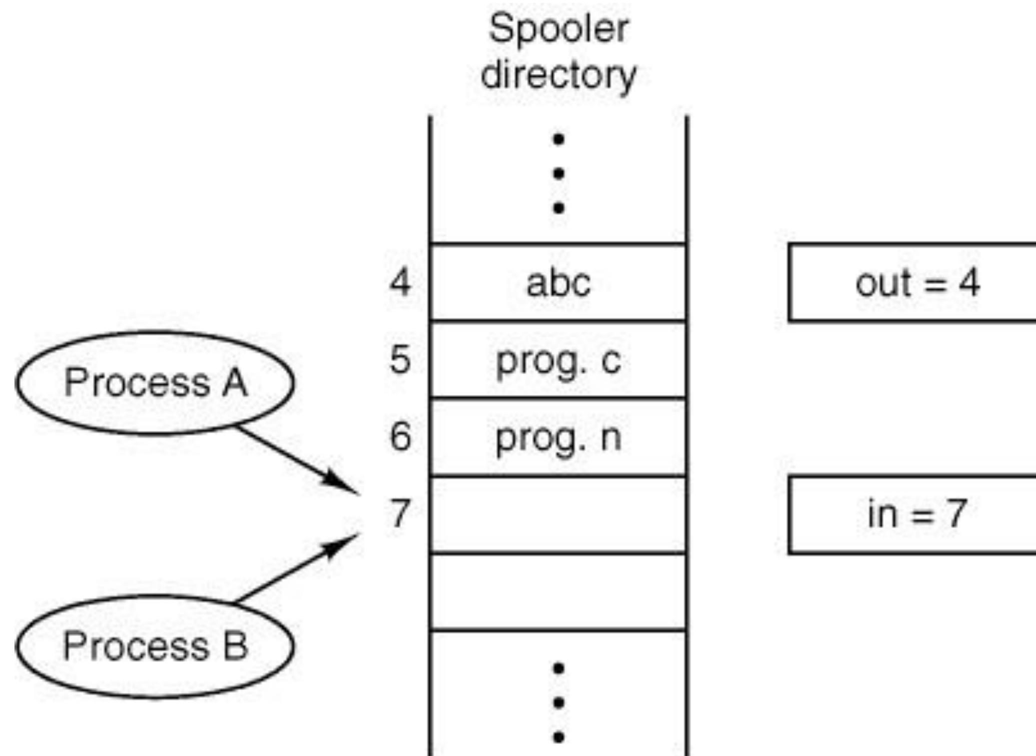
Giao tiếp giữa các tiến trình

Giao tiếp giữa các tiến trình

- Có ba lý do liên quan đến giao tiếp giữa các tiến trình:
  - Trao đổi thông tin.
  - Chia sẽ tài nguyên.
  - Trình tự hoạt động của các tiến trình

Điều kiện tương tranh
Race Conditions

- Hai tiến trình cùng truy cập vào bộ nhớ chia sẻ tại một thời điểm



Miền găng - Critical Regions

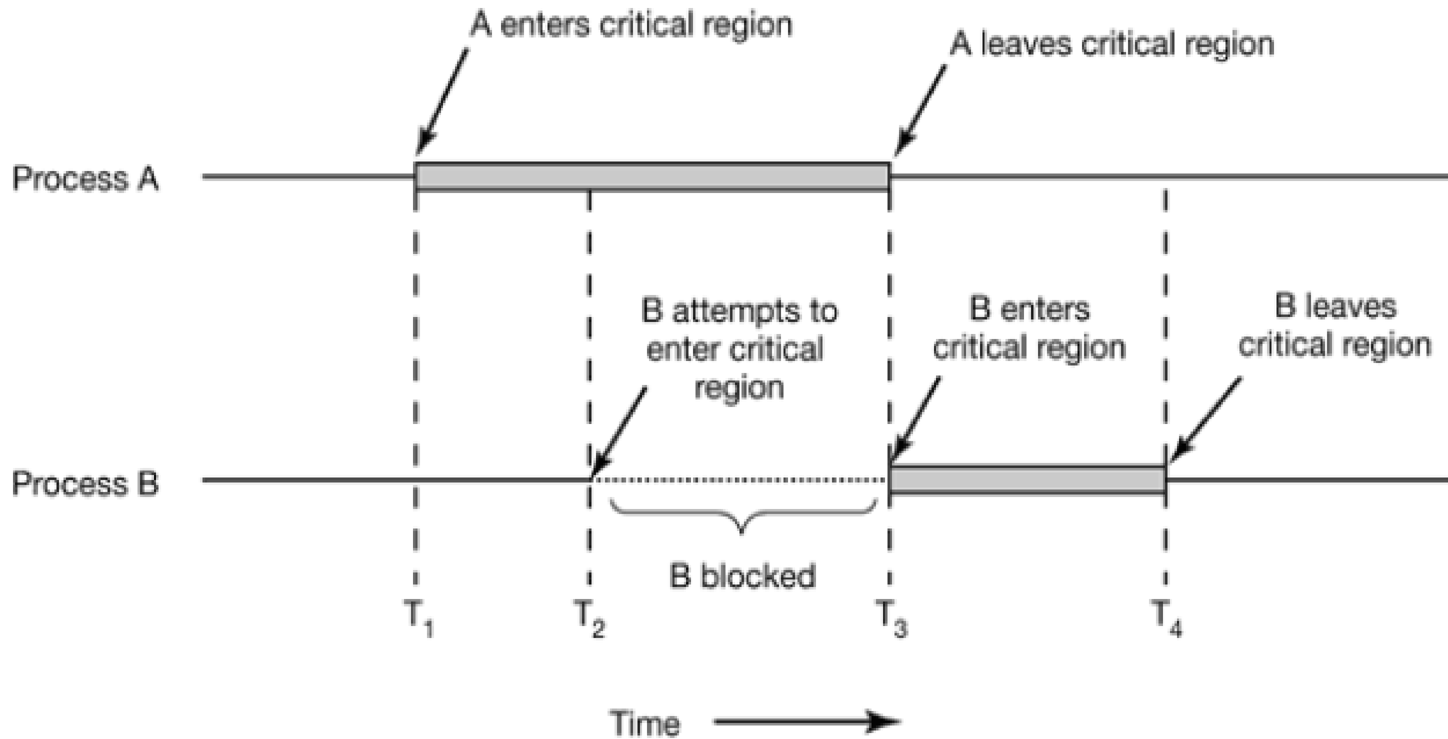
- *Một phần của chương trình mà bộ nhớ chia sẻ được truy cập được gọi là miền găng (**critical region hoặc critical section)**.*
- Nếu đảm bảo không có hai tiến trình truy cập vào miền găng tại cùng một thời điểm thì sẽ tránh được điều kiện tương tranh (race conditions).

Miền găng - Critical Regions

Tránh điều kiện tương tranh

- Không có 2 tiến trình đồng thời trong miền găng.
- Không có giả định có thể được thực hiện về tốc độ hay số CPU.
- Không có tiến trình chạy bên ngoài miền găng có thể ngăn chặn các tiến trình khác.
- Không có tiến trình cần phải chờ đợi mãi mãi để vào miền găng của nó.

Loại trừ lẫn nhau với Busy Waiting



Loại trừ lẫn nhau bằng cách sử dụng miền găng.

Loại trừ lẫn nhau với Busy Waiting

❑ Cấm ngắt – Disabling interrupts

- **Ý tưởng**
  - Mỗi tiến trình cấm tất cả các ngắt sau khi vào miền găng và kích hoạt lại trước khi ra khỏi miền găng.
  - Tiến trình cấm ngắt có thể kiểm tra và cập nhật bộ nhớ chia sẻ mà không sợ tiến trình khác can thiệp.
- **Nhận xét**
  - Rất thiếu thận trọng khi cho phép tiến trình người dùng được phép thực hiện lệnh cấm ngắt, có thể tê liệt hệ thống.

Loại trừ lẫn nhau với Busy Waiting

Sử dụng biến Lock

- Đây là một giải pháp phần mềm.
- Các tiến trình chia sẻ một biến ***lock***, giá trị khởi tạo là 0.
- Tiến trình muốn vào miền găng phải kiểm tra biến ***lock***.
  - Nếu ***lock= 0***, tiến trình thiết lập giá trị cho ***lock =1*** và đi vào miền găng.
  - Nếu ***lock= 1***, tiến trình này chờ cho đến khi ***lock =0***.

Loại trừ lẫn nhau với Busy Waiting

❑ Sử dụng biến Lock

– Đoạn code sử dụng biến lock

```
while (TRUE) {
        while (lock == 1); // wait
        lock = 1;
        critical-section(); //thực hiện đoạn găng
        lock = 0; //kết thúc đoạn găng phải đặt
                     //lock = 0 để giải phóng tài nguyên
        Noncritical-section ();
}
```

Loại trừ lẫn nhau với Busy Waiting

- **Nhận xét:**
  - Khi **lock==0**: Tiến trình thứ nhất vào miền găng nhưng chưa kịp thiết lập lock=1,
  - Một tiến trình khác chạy theo lịch trình, kiểm tra thấy lock==0, nó thiết lập lock=1, và vào miền găng.

  ➔ *Như vậy hai tiến trình có thể trong miền găng tại cùng một thời điểm*

Loại trừ lẫn nhau với Busy Waiting

Kiểm tra luân phiên - Strict Alternation

- Hai tiến trình dùng chung biến ***turn*** kiểu integer, giá trị khởi tạo là 0.
  - ***turn = 0***, tiến trình P1 được vào miền găng, khi P1 rời khỏi miền găng, *turn* =1 cho phép P2 vào miền găng.
  - ***turn = 1,*** tiến trình P2 được vào miền găng. P1 chờ đến khi *turn =0*.

Loại trừ lẫn nhau với Busy Waiting


```
while (TRUE) {
    while (turn != 0)/* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

```
while (TRUE) {
    while (turn != 1);  /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

Đoạn mã cho 2 tiến trình P1 và P2

Loại trừ lẫn nhau với Busy Waiting

- **Nhận xét:**
  - Ngăn được tình trạng 2 tiến trình đồng thời vào miền găng.
  - Số lần vào miền găng của P1, P2 là cân bằng, do đó gặp vấn đề khi P1 cần vào miền găng liên tục còn P2 không cần thiết

Loại trừ lẫn nhau với Busy Waiting

- **Giải pháp Peterson**
  - Ban đầu không có tiến trình nào trong miền găng.
  - Hai tiến trình P0 và P1 sử dụng hai biến dùng chung **turn, interesse[2]**
    - **turn = 0** đến phiên $\mathbf{P_0}$; Nếu **turn=1** đến phiên $\mathbf{P_1}$
    - Nếu **Pi** muốn vào miền găng thì thiết lập **Interesse[i]=TRUE**.

Loại trừ lẫn nhau với Busy Waiting

- **Khởi tạo**:
  - ***Interesse[0]=Interesse[1]=false*; *turn = 0 hoặc 1***
  - Giả sử Pi muốn vào miền găng nó gọi *enter_region*, và thiết lập
    - *interesse[i]=TRUE*,
    - *turn=j* (đề nghị thử Pj vào miền găng).
  - Nếu tiến trình Pj không vào miền găng thì thiết lập
    - *interesse[j]=FALSE*, thì Pi có thể vào.
    - Ngược lại, Pi phải chờ đến khi *interesse[j]=FALSE*.
  - Khi Pi hoàn tất công việc nó gọi *leave_region* rời khỏi miền găng, và thiết lập
    - *interesse[i]= FALSE*.

Loại trừ lẫn nhau với Busy Waiting


```
#define FALSE 0
#define TRUE  1
#define N     2      /* number of processes */

int turn;           /* whose turn is it? */
int interested[N];  /* all values initially 0 (FALSE) */

void enter_region(int process)      /* process is 0 or 1 */
{
    int other;                      /* number of the other process */

    other = 1 - process;            /* the opposite of process */
    interested[process] = TRUE;     /* show that you are interested */
    turn = process;                 /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}

void leave_region (int process)     /* process, who is leaving */
{
    interested[process] = FALSE;    /* indicate departure from critical region */
}
```

Loại trừ lẫn nhau với Busy Waiting

- **Nhận xét:**
  - Ngăn chặn được tình trạng mâu thuẫn truy xuất
    - Pi chỉ có thể vào miền găng khi
      - *interesse[j]=FALSE*
      - Hoặc *turn* = *i*.
    - Nếu cả hai tiến trình đều muốn vào miền găng thì
      - *interesse[i]* = *interesse[j]* =*TRUE*
      - Nhưng giá trị của *turn* chỉ có thể hoặc là 0 hoặc là 1.
  - ➔Do đó chỉ có một tiến trình được vào miền găng

Loại trừ lẫn nhau với Busy Waiting

❑ Giải pháp Test and Set Lock – TSL

– Tập lệnh của máy có thêm một chỉ thị đặc biệt để kiểm tra và cập nhật nội dung của một vùng nhớ trong một đơn vị thao tác, gọi là chỉ thị **TSL**.

TSL RX, LOCK

```
enter_region:
    TSL REGISTER,LOCK    | copy lock to register and set lock to 1
    CMP REGISTER,#0      | was lock zero?
    JNE enter_region     | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0         | store a 0 in lock
    RET | return to caller
```

Loại trừ lẫn nhau với Busy Waiting

- **Nhận xét:**
  - *Giảm nhẹ công việc lập trình nhưng việc cài đặt TSL như một lệnh máy thì phức tạp*
  - *Khi có nhiều CPU, việc điều phối thực hiện TSL cho từng CPU gặp khó khăn*

Sleep and Wakeup

- **Giải pháp:** Dùng các system calls: *sleep* và *wakeup*
  - ***Sleep***: đình chỉ một tiến trình cho đến khi tiến trình khác đánh thức nó.
  - ***Wakeup:*** tham số là tiến trình được đánh thức.
- **Ý tưởng**
  - Tiến trình chưa đủ điều kiện vào miền găng, nó gọi ***Sleep*** để tự khóa, chờ tiến trình khác gọi ***Wakeup*** để giải phóng nó.
  - Tiến trình gọi ***Wakeup*** khi ra khỏi miền găng để đánh thức một tiến trình đang chờ vào miền găng.

Sleep and Wakeup

❑ Code Sleep and Wakeup

```
int busy;   // 1 miền găng đang bị chiếm.
int blocked;//số tiến trình đang bị tạm khóa
while (1) {
    if (busy){
            blocked = blocked + 1;
            sleep();}
    else busy = 1;
    critical-section ();
    busy = 0;
    if(blocked){
            wakeup(process);
            blocked = blocked - 1;}
   Noncritical-section ();
}
```

Sleep and Wakeup

- **Ví dụ:** bài toán sản xuất - tiêu dùng.
  - Hai tiến trình chia sẻ một buffer chung, kích thước cố định.
    - P1: nhà sản xuất, đặt thông tin vào buffer,
    - P2: người tiêu dùng, lấy thông tin ra từ buffer
  - P1 muốn đặt một phần tử mới vào buffer, nhưng buffer đầy, nó ***sleep,*** nó được đánh thức khi P2 lấy 1 hoặc nhiều phần tử từ buffer.
  - **P2** muốn lấy phần tử từ buffer nhưng buffer rỗng, nó ***sleep***, P1 weakup P2 khi đặt phần tử vào buffer

Sleep and Wakeup

Hoạt động của producer

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

void producer (void)
{
    int item;

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

Sleep and Wakeup

Hoạt động của consumer

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

    while (TRUE) {                          /* repeat forever */
        if (count == 0) sleep();            /* if buffer is empty, got to sleep */
        item = 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(item);                 /* print item */
    }
}
```

Có thể xãy ra tình huống cả hai tiến trình sẽ Sleep vĩnh viễn ????

Semaphores

- **Semaphore *s* là một *biến* có 2 thuộc tính**
  - Một số nguyên dương ***e***
  - Một hàng đợi ***f*** lưu danh sách các tiến trình đang ở trạng thái **waiting**
- **Hai thao tác được định nghĩa trên semaphore**
  - ***Down(s)****: giảm giá trị **e** đi 1.*
    - *Nếu e ≥ 0 thì tiếp tục xử lý.*
    - *Ngược lại, nếu e < 0, tiến trình phải chờ.*
  - ***Up(s)***: tăng giá trị **e** lên 1 đơn vị.
    - Nếu có một hoặc nhiều tiến trình đang chờ thì hệ thống sẽ chọn một trong các tiến trình để đánh thức.

Semaphores

- Down(s)

```
e= e - 1;
if e<0
{
  status(P)=waiting;
  enter(P,f(s));
}
```

- Up(s)

```
e= e + 1;
if e=0
{
  exit(Q,f(s));
  //Q là tiến trình đang chờ trên s
  status(Q)= ready;
  enter(Q,ready-list);
}
```

Semaphores

- Sử dụng semaphore để giải quyết bài toán Nhà sản xuất và người tiêu dùng:
  - Sử dụng 3 semaphore:
    - *Full*: đếm số các khe đầy
    - *Empty*: đếm số các khe rỗng
    - *Mutex*: để đảm bảo rằng các nhà sản xuất và người tiêu dùng không truy cập vào buffer tại cùng một thời điểm.

Semaphores


```
#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 */
        item = produce_item();  /* generate something to put in buffer */
        down(&empty);           /* decrement empty count */
        down(&mutex);           /* enter critical region */
        insert_item(item);      /* put new item in buffer */
        up(&mutex);             /* leave critical region */
        up(&full);              /* increment count of full slots */
    }
}
```

Semaphores


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

    while (TRUE) {                 /* infinite loop */
        down(&full);               /* decrement full count */
        down(&mutex);              /* enter critical region */
        item a= remove_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 */
    }
}
```

Mutexes

- Một **mutex** là một biến có thể thuộc một trong hai trạng thái: **unlock** hoặc **lock**
- **Hai thủ tục được sử dụng**:
  - ***mutex_lock***: Khi một thread cần truy cập vào miền găng. Nếu mutex hiện tại đang unlock thì cuộc gọi thành công và nó đi vào miền găng
  - ***mutex_unlock***: Thread trong miền găng kết thúc sẽ gọi mutex_unlock. Nếu có nhiều thread đang bị block trên mutex, một trong số đó được chọn ngẫu nhiên và cho phép lock.

Mutexes

- Hiện thực *mutex_lock* và *mutex_unlock*

```
mutex_lock:
    TSL REGISTER,MUTEX | copy mutex to register and set mutex to 1
    CMP REGISTERS,#0   | was mutex zero?
    JZE ok             | if it was zero, mutex was unlocked, so return
    CALL thread_yield  | mutex is busy; schedule another thread
    JMP mutex_lock     | try again later
ok:   RET | return to caller; critical region entered

mutex_unlock:
    MOVE MUTEX,#0      | store a 0 in mutex
    RET | return to caller
```

Monitors

- **Monitors** (*Hoare(1974) Brinch& Hansen*) là một cấu trúc dữ liệu bao gồm các biến và thủ tục.
- **Đặc điểm:**
  - Các biến và cấu trúc dữ liệu bên trong **monitor** chỉ có thể được thao tác bởi các thủ tục định nghĩa bên trong monitor đó (*encapsulation*).
  - Tại một thời điểm, chỉ có một tiến trình duy nhất được hoạt động bên trong một monitor (*mutual exclusive*).

Monitors

- **Monitor là một cấu trúc với các thuộc tính**:
  - *Biến điều kiện (c)*
  - *Hàng đợi chứa các tiến trình bị khóa f(c)*
  - *Hai thao tác kèm theo là* Wait và Signal:
    - **Wait(c)**: chuyển trạng thái tiến trình gọi sang blocked , và đặt tiến trình này vào hàng đợi của c.
    - **Signal(c)**: nếu có một tiến trình đang bị khóa trong hàng đợi của c, tái kích hoạt tiến trình đó, và tiến trình gọi sẽ rời khỏi monitor.

Monitors

Wait

```
Wait(c)
{
  status(P)=blocked;
 enter(P,f(c));
}
```

Signal

```
if (f(c) != NULL)
{
  exit(Q,f(c));
  //chọn1 tiến trình chờ trên c
  statusQ)=ready;
  enter(Q,ready-list);
}
```

Monitors

❑ Cài đặt monitors

```
monitor <tên monitor >
{
      condition <list các biến đ/kiện>;
      <khai báo các biến>;
      procedure Action1();   {     }
      ....
      procedure Actionn();   {     }
}
```

Monitors

- Dùng monitor trong bài toán sản xuất – tiêu dùng
- Chỉ có một thủ tục monitor tại một thời điểm đang hoạt động. Buffer có N khe

```
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;
```

Monitors


```
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;
```

– **Nhận xét**
  - Việc truy suất độc quyền được đảm bảo bởi trình biện dịch
  - Đòi hỏi ngôn ngữ lập trình phải có kiểu dữ liệu monitor

Message Passing

- **Giải pháp:**
  - Dựa trên cơ sở trao đổi thông điệp với hai thao tác nguyên thủy Send và Receive để thực hiện sự đồng bộ hóa
    - **send(destination, &message):** gởi một thông điệp đến một tiến trình hay gởi vào hộp thư.
    - **receive(source, &message)**: nhận một thông điệp từ một tiến trình hay từ bất kỳ một tiến trình nào, tiến trình gọi sẽ chờ nếu không có thông điệp nào để nhận.

Message Passing

❑ Sử dụng

- Một tiến trình kiểm soát việc sử dụng tài nguyên và nhiều tiến trình khác yêu cầu tài nguyên này.
- Tiến trình có yêu cầu tài nguyên sẽ gởi một thông điệp đến tiến trình kiểm soát và sau đó chuyển sang trạng thái **waiting** cho đến khi nhận được một thông điệp chấp nhận cho truy xuất từ tiến trình kiểm soát tài nguyên.

Message Passing

- Khi sử dụng xong tài nguyên, tiến trình gởi một *thông điệp* khác đến tiến trình kiểm soát để báo kết thúc truy xuất.
- Tiến trình kiểm soát: khi nhận được thông điệp yêu cầu tài nguyên, nó sẽ chờ đến khi tài nguyên sẵn sàng để cấp phát thì gửi một thông điệp đến tiến trình đang bị khóa trên tài nguyên đó để **đánh thức** tiến trình này.

Message Passing

- Cấu trúc tiến trình yêu cầu tài nguyên trong giải pháp ***truyền thông điệp***

```
while (TRUE)
{
    Send(process controler, request message);
    Receive(process controler,accept message);
    critical-section();
    Send(process controler, end message);
    Noncritical-section ();
}
```

Message Passing

- Bài toán Sản xuất – tiêu dùng với *N* thông báo.

```
#define N 100 /* number of slots in the buffer */
void producer(void)
{
    int item;
    message m;    /* message buffer */

    while (TRUE) {
        item = produce_item( );     /* generate something to put in buffer */
        receive(consumer, &m);      /* wait for an empty to arrive */
        build_message (&m, item);   /* construct a message to send */
        send(consumer, &m);         /* send item to consumer */
    }
}
```

Message Passing

- Bài toán Sản xuất – tiêu dùng với *N* thông báo.

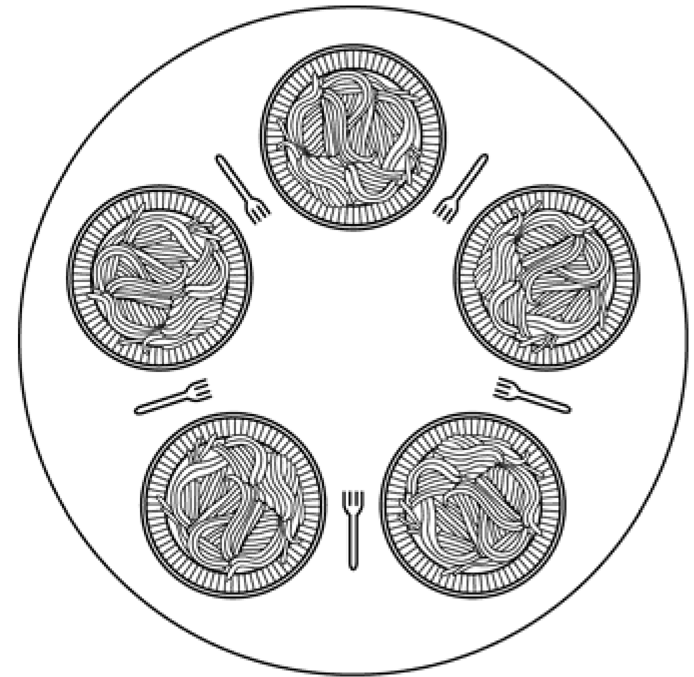
```
void consumer(void) {
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
    while (TRUE) {
        receive(producer, &m);          /* get message containing item */
        item = extract_item(&m);        /* extract item from message */
        send(producer, &m);             /* send back empty reply */
        consume_item(tem);              /* do something with the item */
    }
}
```

Message Passing

- **Nhận xét:**
  - *Semaphore* và *monitor* có thể giải quyết vấn đề truy xuất độc quyền trên các máy tính có một hoặc nhiều bộ xử lý chia sẽ bộ nhớ dùng chung.
  - *Message* dùng để giải quyết bài toán đồng bộ hóa trong những hệ thống phân tán.

Vấn đề IPC cổ điển

- Bài toán 5 triết gia ăn tối:
  - Có 5 triết gia ăn tối món mì ống, ngồi xung quanh bàn tròn, trước mặt mỗi người có một chiếc nĩa. Muốn ăn được thì cần có 2 nĩa, nếu mỗi người đều lấy chiếc nĩa trước mặt thì sẽ không có triết gia nào ăn được.
  - Hãy đồng bộ việc ăn tối của 5 triết gia sao cho tất cả đều ăn được



Vấn đề IPC cổ điển

- Hoạt động của triết gia:
  - Ăn: cầm 2 nĩa
  - Suy nghĩ: đặt 2 nĩa xuống
  - Đói: cố gắng lấy nĩa
- **Giải pháp**
  - Sử dụng một *array*, *State* để theo dõi các triết gia đang ăn, suy nghĩ, hay đói.
  - Triết gia bên cạnh được xác định bởi các macro LEFT và RIGHT.
  - Sử dụng một semaphore cho 1 triết gia

Vấn đề IPC cổ điển


```
#define N            5    /* number of philosophers */
#define LEFT         (i+N-1)%N /* number of i's left neighbor */
#define RIGHT        (i+1)%N /* number of i's right neighbor */
#define THINKING     0    /* philosopher is thinking */
#define HUNGRY       1    /* philosopher is trying to get forks */
#define EATING       2    /* philosopher is eating */
typedef int semaphore;    /* semaphores are a special kind of int */
int state[N];             /* array to keep track of everyone's state */
semaphore mutex = 1;      /* mutual exclusion for critical regions */
semaphore s[N];           /* one semaphore per philosopher */

void philosopher (int i)  /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {        /* repeat forever */
        think();          /* philosopher is thinking */
        take_forks(i);    /* acquire two forks or block */
        eat();            /* yum-yum, spaghetti */
        put_forks(i);     /* put both forks back on table */
    }
}
```

Vấn đề IPC cổ điển


```
void take_forks(int i)        /* i: philosopher number, from 0 to N-1 */
{
     down(&mutex);            /* enter critical region */
     state[i] = HUNGRY;       /* record fact that philosopher i is hungry */
     test(i);                 /* try to acquire 2 forks */
     up(&mutex);              /* exit critical region */
     down(&s[i]);             /* block if forks were not acquired */
}

void put_forks(i)             /* i: philosopher number, from 0 to N-1 */
{
     down(&mutex);            /* enter critical region */
     state[i] = THINKING;     /* philosopher has finished eating */
     test(LEFT);              /* see if left neighbor can now eat */
     test(RIGHT);             /* see if right neighbor can now eat */
     up(&mutex);              /* exit critical region */
}

void test(i)                  /* i: philosopher number, from 0 to N-1 */
{
   if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
     state[i] = EATING;
     up(&s[i]);
   }
}
```

Deadlock

Khái niệm

- Deadlock
  - Các tiến trình trong tập hợp **chờ đợi lẫn nhau**
    - do cạnh tranh trong sử dụng tài nguyên hay trong giao tiếp
  - Chờ đợi một sự kiện không bao giờ xảy ra

⇒ Tất cả các tiến trình trong tập hợp **bị khóa vĩnh viễn !**

Mô hình hệ thống

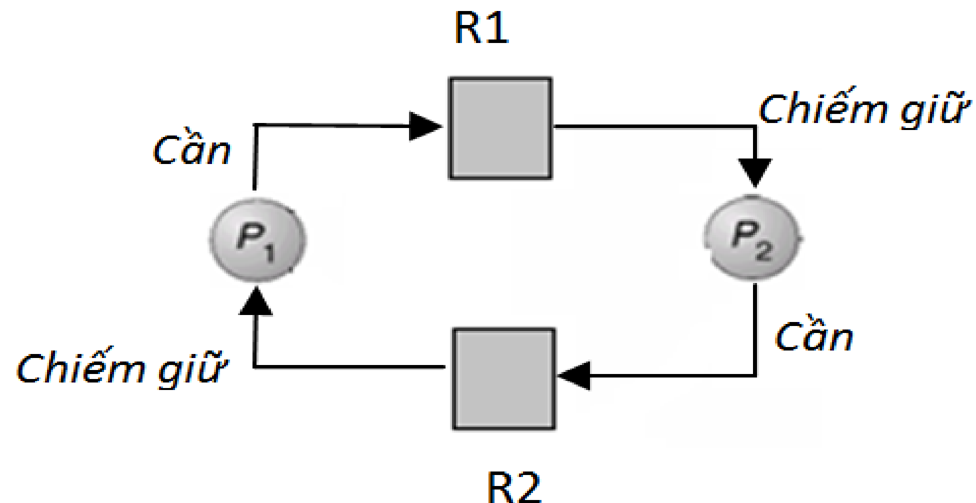
- Hệ thống bao gồm một số xác định các loại tài nguyên sẽ được chia sẻ cho các tiến trình có nhu cầu
- Mỗi loại tài nguyên có thể có nhiều thể hiện
- Mỗi tiến trình sử dụng tài nguyên theo trình tự
  - Request : yêu cầu tài nguyên, nếu yêu cầu không được thoả mãn nay, tiến trình phải đợi
  - Use : sử dụng tài nguyên được cấp phát
  - Release : giải phóng tài nguyên

Các điều kiện xảy ra Deadlock

- Coffman, Elphick và Shoshani (1971) đã đưa ra 4 điều kiện cần có thể làm xuất hiện tắc nghẽn:
  - Mutual exclusion: hệ thống có sử dụng những loại tài nguyên mang bản chất không chia sẻ được.
  - Wait for : Tiến trình tiếp tục chiếm giữ các tài nguyên đã cấp phát cho nó trong khi chờ được cấp phát thêm một số tài nguyên mới.
  - No preemption: Tài nguyên chỉ được thu hồi khi tiến trình đang chiếm giữ chúng tự nguyện trao trả.
  - Circular wait: Tồn tại một chu kỳ trong đồ thị cấp phát tài nguyên .
- Hội đủ 4 điều kiện trên đây : Deadlock có thể xảy ra

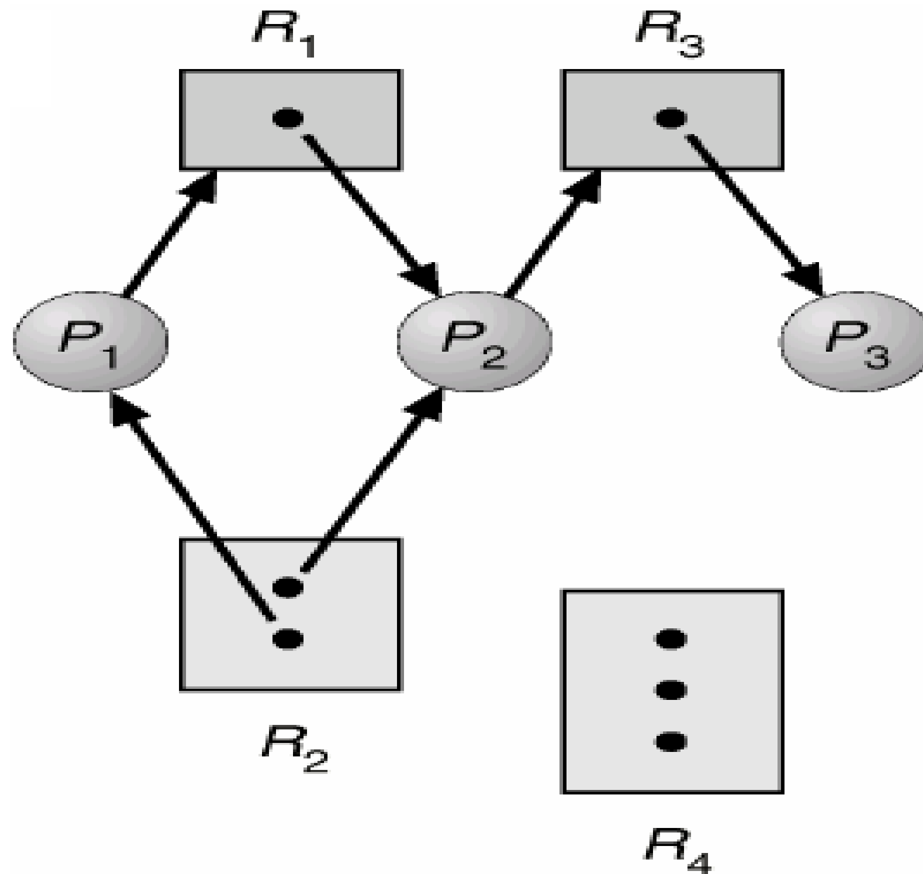
Đồ thị cấp phát tài nguyên

- Nhận xét
  - Nếu đồ thị không có chu trình ⇒ no deadlock.
  - Nếu đồ thị có 1 chu trình ⇒
    - Nếu mỗi tài nguyên chỉ có 1 thể hiện ⇒ deadlock.
    - Nếu mỗi tài nguyên có nhiều thể hiện ⇒ có thể có deadlock.



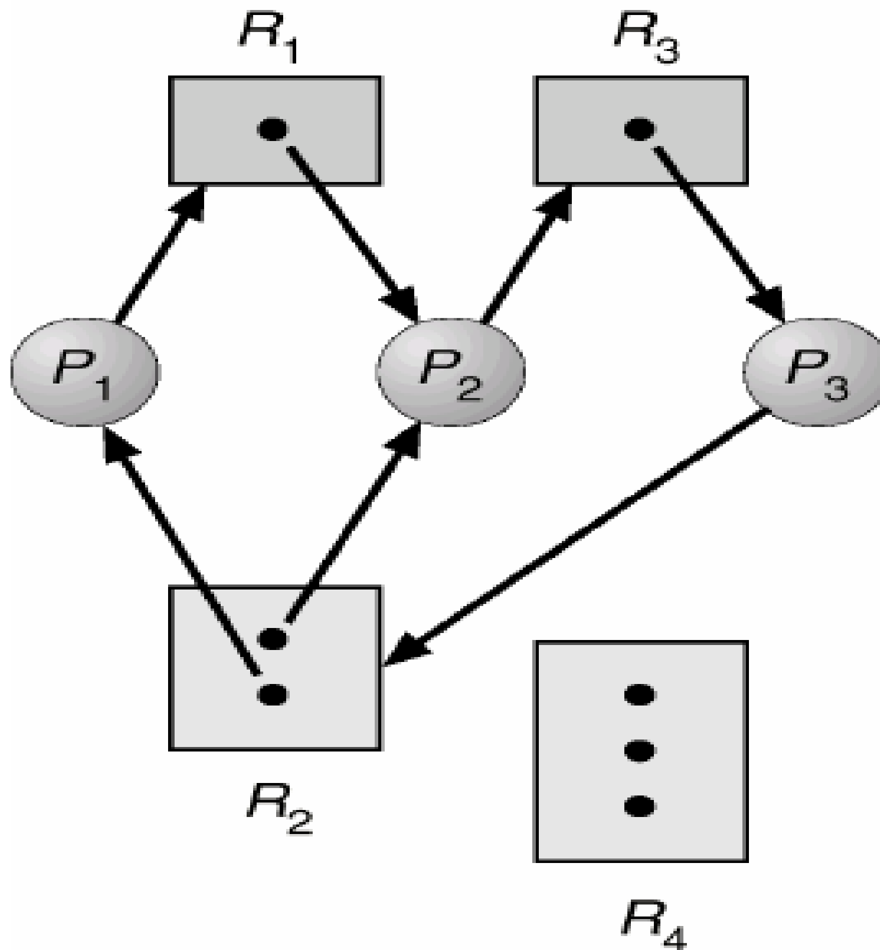
Đồ thị cấp phát tài nguyên

- Không có chu trình=> không deadlock



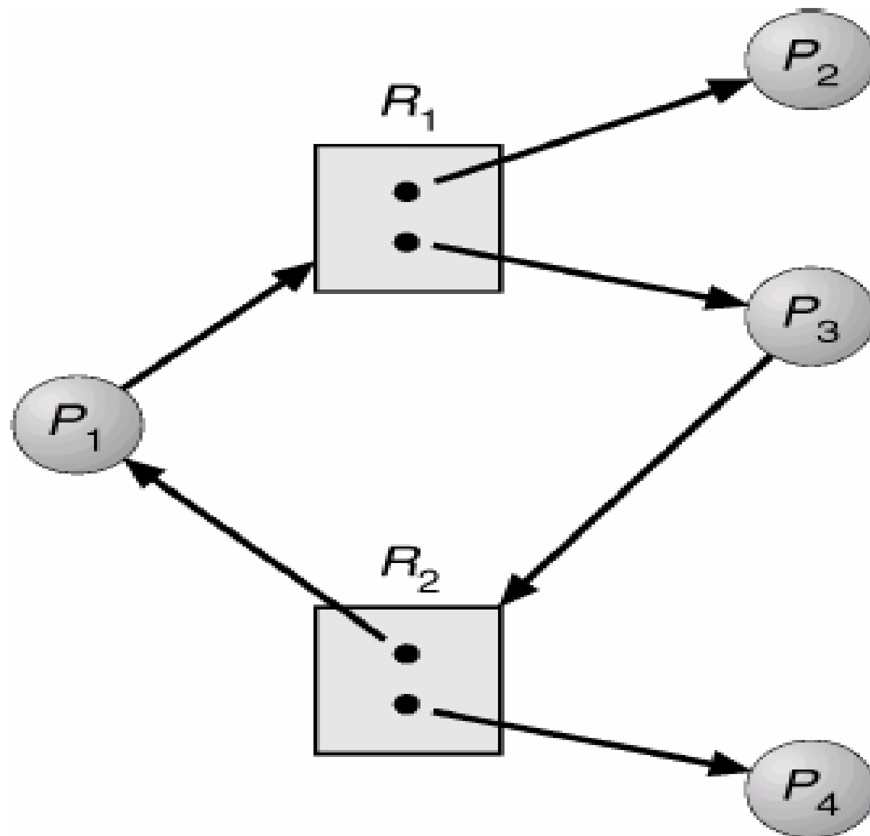
Đồ thị cấp phát tài nguyên

- Có chu trình => deadlock



Đồ thị cấp phát tài nguyên

- Có chu trình => không deadlock



Các bài toán xử lý deadlock

- Ngăn chặn tắc nghẽn
- Tránh/ Ngăn ngừa tắc nghẽn (*)
- Phát hiện và Xử lý tắc nghẽn

Bài toán ngăn ngừa deadlock

- Đảm bảo Deadlock không thể xảy ra
  - Tìm cách loại bỏ ít nhất 1 trong 4 điều kiện cần để xảy ra Deadlock
  - Nhắc lại 4 điều kiện cần
    - Mutual Exclusion
    - Hold and Wait
    - No preemption
    - Circular wait

Bài toán ngăn ngừa deadlock

- Giả sử hệ thống được mô tả với các thông tin sau :
  - int Available[NumResources];
    - Available[r]= số lượng các thể hiện còn tự do của tài nguyên r
  - 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 lượng tài nguyên r thực sự cấp phát cho p
  - int Need[NumProcs, NumResources];
    - Need[p,r] = Max[p,r] - Allocation[p,r]

Thuật toán nhà băng
Banker's Algorithm

- P_i xin k thể hiện của R_j

```
1 : if (k <= Need[i,j])  goto 2;
    else Error();
2: if (k <= Available[j]) goto 3;
    else  MakeWait(Pi);
3: /* Giả sử hệ thống đã cấp phát cho Pi các tài nguyên mà nó yêu cầu và cập nhật tình
   trạng hệ thống như sau:*/
        Available[j] = Available[j] - k;
        Allocation[i,j]= Allocation[i,j]+ k;
        Need[i,j] = Need[i,j] - k;
    Result = TestSafe();
    if (Result == Safe) Allocate(i,j,k); //cấp cho Pi k thể hiện Rj
        else  MakeWait(Pi);
```

Thuật toán nhà băng Banker's Algorithm

- TestSafe()

1. Giả sử có các mảng

   int Work[NumResources] = Available[NumResources];

   int Finish[NumProcs] = false;

2. Tìm i sao cho

   Finish[i] == false & Need[i,j] <= Work[i,j], ∀j <=NumRes

   Nếu không có i như thế, đến bước 4.

3. Work = Work + Allocation[i];

   Finish[i] = true;

   Đến bước 2

4. Nếu Finish[i] == true với mọi i, thì hệ thống ở trạng thái an toàn.

- Ý tưởng cơ bản: đảm bảo rằng chúng ta luôn có một "con đường thoát hiểm"
- Biểu đồ tài nguyên có thể rút gọn
- Điều này có thể được thực thi với thuật toán ngân hàng:
- Khi một yêu cầu được thực hiện
- Giả sử bạn đã chấp nhận nó
- Giả vờ làm tất cả các yêu cầu pháp lý khác
- Biểu đồ có thể bị giảm?
- Nếu vậy, phân bổ tài nguyên được yêu cầu
- Nếu không, hãy khóa chủ đề