Chương 4: Process Schronization – Đồng bộ giữa các tiến trình

Hiểu biết vấn đề đồng bộ giữa các tiến trình, các giải pháp đồng bộ.

Làm bài tập, viết chương trình mô phỏng các giải pháp đồng bộ.

Nội dung

- Nhu cầu đồng bộ hóa (synchronisation)
- Vấn đề đồng bộ
- Giải pháp « **busy waiting** »
- Các giải pháp « **SLEEP and WAKEUP** »
  - Semaphore
  - Monitors
  - Trao đổi thông điệp

1. Nhu cầu đồng bộ hóa

- Trong hệ thống, nhiều tiến trình liên lạc với nhau
- HĐH luôn cần cung cấp những cơ chế đồng bộ hóa để bảo đảm hoạt động đồng thời của các tiến trình không tác động sai lệch đến nhau
- Việc tác động sai lệch do:
  - Yêu cầu độc quyền truy xuất
  - Yêu cầu phối hợp

1.1. Yêu cầu độc quyền truy xuất

- Tài nguyên trong hệ thống phân 2 loại:
  - Tài nguyên chia sẻ: cho phép nhiều tiến trình đồng thời truy xuất
  - Tài nguyên không thể chia xẻ: tại một thời điểm chỉ có một tiến trình sử dụng
- Không thể chia sẻ do:
  - Đặc điểm phần cứng
  - Nhiều tiến trình đồng thời sử dụng tài nguyên này sẽ gây ra kết quả không dự đoán trước được
- Giải pháp:
  - HĐH cần đảm bảo vấn đề độc quyền truy xuất tài nguyên: tại một thời điểm chỉ cho phép một tiến trình sử dụng tài nguyên

1.2. Yêu cầu phối hợp đồng bộ

- Các tiến trình trong hệ thống hoạt động độc lập, thường không đồng bộ
- Khi có nhiều tiến trình phối hợp hoàn thành một tác vụ có thể dẫn đến yêu cầu đồng bộ:
  - Tiến trình này sử dụng kết quả của tiến trình kia
  - Cần hoàn thiện các tiến trình con mới có thể hoàn thiện tiến trình cha

2. Vấn đề đồng bộ(1)

Bài toán 1:

Tiến trình tăng x lên 1

- *x:=x+1;*

Tiến trình giảm x đi 1

- *x:=x-1*

•x khởi tạo == 1
•x được chia sẻ giữa 2 tiến trình
•**Giá trị của X là bao nhiêu sau khi cả 2 tiến trình hoàn thành?**

2. Vấn đề đồng bộ(2)

P1

x:=x+1; dịch thành

1. Load RegAx, x
2. tăng RegAx
3. Save x, RegAx

P2

x:=x-1; compiles to

1. Load RegAx, x
2. giảm RegAx
3. Save x, RegAx

Nếu giá trị ban đầu của x là 1, sau khi thực hiện các bước trên, **x sẽ có giá trị 0** (chỉ = 1 khi 2 tiến trình lần lượt thực hiện: P1 xong, P2 mới thực hiện)

2. Vấn đề đồng bộ(3)

- Bài toán 2:
  - Khách hàng có tài khoản 800K, cần thực hiện yêu cầu rút 400K. Việc thực hiện yêu cầu thông qua tiến trình P1
  - Ở một vị trí khác, Hacker có được mật khẩu của khách hàng, truy nhập vào tài khoản khách hàng yêu cầu rút 700K. Việc thực hiện yêu cầu hacker thông qua tiến trình P2
  - Cả P1, P2 đều truy nhập vào biến dùng chung là **account** của khách hàng; mỗi tiến trình rút tiền có biến **require**(số tiền cần rút)
  - Cả 2 tiến trình P1, P2 đều có đoạn rút tiền và cập nhật tài khoản:

```
if (account >= require)
  account -= require;
else
  printf("Error");
```


2. Vấn đề đồng bộ(4)

- Bài toán 2(tiếp)
  - Tình huống nảy sinh:
    - P1 kiểm tra thấy **account > require** nên thực hiện đoạn code trên để rút tiền nhưng chưa cập nhật tài khoản(chưa thực hiện lệnh **account -= require**) do hết thời gian sử dụng CPU được phân phối cho nó
    - P2 kiểm tra điều kiện vẫn thỏa mãn(**account** vẫn = 800K) nên nó thực hiện việc rút tiền. Giả sử P2 được phân phối đủ thời gian sử dụng CPU, cập nhật **account** = 100K và kết thúc
    - P1 quay lại thực hiện(vì đã kiểm tra điều kiện từ lần trước) nốt công việc và cập nhật **account** = -300 => tình huống lỗi
  - Giải pháp:
    - Áp dụng cơ chế ***truy xuất độc quyền*** trên tài nguyên đó (**account**): khi một tiến trình đang sử dụng tài nguyên thì các tiến trình khác không được sử dụng

2. Vấn đề đồng bộ(5): đoạn găng

- Critical session(- *critical region* đoạn găng, miền găng)
- Trong ví dụ trên, tiến trình P1, P2 đều bao gồm chuỗi các lệnh riêng và các lệnh thực hiện rút tiền:

…//các lệnh kết nối, lệnh kiểm tra

```
if (account >= require)
  account -= require;
  print('drawed money');
else
  printf("Error");
```

// các lệnh kết thúc tiến trình…

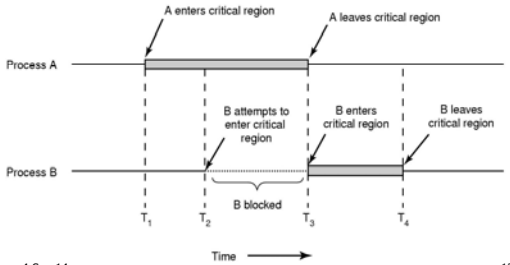
Đoạn lệnh thao tác tài nguyên Chung, có thể xảy ra mâu thuẫn => **Đoạn găng**

2. Vấn đề đồng bộ(6): đoạn găng

- Khái niệm **đoạn găng(miền găng):**
  - là đoạn mã lệnh có khả năng xảy ra mâu thuẫn khi truy xuất tài nguyên chung
- HĐH cần cài đặt các giải pháp đồng bộ để giải quyết vấn đề độc quyền truy xuất
  - Quyết định tiến trình nào sẽ được vào miền găng(thực hiện các lệnh trong miền này)
  - Khi một tiến trình đang ở miền găng thì các tiến trình khác không thể vào miền găng

2. Vấn đề đồng bộ(7): đoạn găng

- Minh họa dòng thời gian của đoạn găng:



3. Giải pháp « **busy waiting** »

- Giải pháp phần mềm:
  - Sử dụng các biến cờ hiệu(simaphore)
  - Sử dụng việc kiểm tra luân phiên
  - Giải pháp của Peterson
- Giải pháp có sự hỗ trợ phần cứng:
  - Cấm ngắt
  - Test & Set

3.1. Sử dụng các biến cờ hiệu(Simaphore)(1)

- Ý tưởng:
  - Các tiến trình **chia sẻ một biến chung đóng vai trò « chốt cửa » (lock)** , biến này được **khởi động là 0**.
  - Một tiến trình muốn vào miền găng, trước tiên phải kiểm tra giá trị của biến **lock**. Nếu **lock = 0**, tiến trình đặt lại giá trị cho **lock = 1** và đi vào miền găng. Nếu **lock đang nhận giá trị** 1, tiến trình phải chờ đến khi **lock có giá trị 0**.
- Như vậy giá trị 0 của lock mang ý nghĩa là không có tiến trình nào đang ở trong miền găng, và lock=1 khi có một tiến trình đang ở trong miền găng.

```
while (TRUE) {
    while (lock == 1); { // đợi lock = 0}
    //trường hợp lock == 0
    lock = 1; //đặt lock = 1 để cấm các tiến trình khác
    critical-section (); //thực hiện đoạn găng
    lock = 0; //kết thúc đoạn găng phải đặt lock = 0 để giải phóng tài nguyên
    Noncritical-section (); // thực hiện các lệnh bên ngoài đoạn găng
}
```


3.1. Sử dụng các biến cờ hiệu(Simaphore)(2)

- Nhận xét:
  - Giải pháp này vẫn xảy ra trường hợp 2 tiến trình cùng trong đoạn găng khi:
    - Lock==0, Tiến trình P1 vào đoạn găng nhưng chưa kịp đặt lock=1 vì hết thời gian sử dụng CPU
    - P2 kiểm tra thấy lock==0, đặt lock=1 và đang thực hiện các lệnh trong đoạn găng nhưng chưa xong vì hết thời gian CPU
    - P1 được phân phối CPU và thực hiện các lệnh trong đoạn găng
    - ⇒ P1, P2 cùng trong đoạn găng

3.2. Sử dụng việc kiểm tra luân phiên(1)

- Ý tưởng:
  - Giải pháp **đề nghị cho hai tiến trình**.
  - Hai tiến trình **sử dụng chung biến *turn***
    - Khởi động với giá trị 0.
    - Nếu *turn = 0*, tiến trình P1 được vào đoạn găng, *turn*= 1 P2 được vào đoạn găng.
    - Nếu *turn = 1*, tiến trình P1 đi vào một vòng lặp chờ đến khi *turn* nhận giá trị *0*.
    - Khi tiến trình P1 rời khỏi đoạn găng, nó đặt giá trị *turn* về 1 để cho phép tiến trình P2 đi vào đoạn găng.

3.2. Sử dụng việc kiểm tra luân phiên(2)

- Cấu trúc tiến trình P1

```
while (TRUE) {
    while (turn != 0); {// wait}
    critical-section ();//thực hiện đoạn găng xong mới đặt turn=1
    turn = 1;
    Noncritical-section ();
}
```

- Cấu trúc tiến trình P2

```
while (TRUE) {
    while (turn != 1); {// wait}
    critical-section ();
    turn = 0;
    Noncritical-section ();
}
```


3.2. Sử dụng việc kiểm tra luân phiên(3)

- Nhận xét:
  - Ngăn được trường hợp 2 tiến trình đồng thời trong đoạn găng
  - Xảy ra tình huống một tiến trình bị ngăn vào đoạn găng bởi một tiến trình bên ngoài đoạn găng khi:
    - Turn=0, P1 vào đoạn găng xong, đặt **turn=1** rồi ra và muốn nhanh chóng quay lại đoạn găng lần nữa
    - Nhưng P2 vẫn thực hiện các lệnh bên ngoài đoạn găng với lượng rất lớn nên thời gian thực hiện rất lâu không thể vào đoạn găng ngay được. Do đó turn vẫn = 1 và P1 không thể vào
  - Số lần vào đoạn găng của P1, P2 là cân bằng(luân phiên nhau) do đó gặp vấn đề khi P1 cần vào đoạn găng liên tục còn P2 không cần thiết lắm.

3.3. Giải pháp Peterson(1)

- Do Peterson đề nghị
- Ý tưởng: kết hợp 2 giải pháp trên
  - P0, P1 sử dụng 2 biến chung *turn, interesse[2]*
    - *Turn* = 0 đến phiên P0, turn=1 đến phiên P1
    - *Interesse[i]*=TRUE, Pi muốn vào đoạn găng
  - Khởi tạo:
    - Interesse[0]=Interesse[1]=false; turn = 0 hoặc 1
  - Để có thể vào được miền găng:
    - Pi đặt giá trị *interesse[i]*=TRUE
    - Sau đó đặt turn=*j* (đề nghị thử tiến trình khác vào miền găng).
    - Nếu tiến trình Pj không quan tâm đến việc vào miền găng (*interesse[j]=FALSE*), thì Pi có thể vào miền găng, nếu không, Pi phải chờ đến khi *interesse[j]=FALSE*.
    - Khi tiến trình Pi rời khỏi miền găng, nó đặt lại giá trị cho *interesse[i]* = FALSE.

3.3. Giải pháp Peterson()

- Cấu trúc tiến trình Pi//gia su: p0 muon vao, p1 ko

```
//ban dau: interese[1]=interese[0]=false; turn=0
while (TRUE) {
      interesse[i]= TRUE;//interesse[0]=true
      int j = 1-i; // j là tiến trình còn lại
         turn = j;
      while (turn == j && interesse[j]==TRUE);{//wait}
      critical-section ();
      interesse[i] = FALSE;
      Noncritical-section ();
  }
```


3.3. Giải pháp Peterson()

- 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 vậy chỉ có một tiến trình được vào miền găng

3.4. Cấm ngắt

- Ý tưởng:
  - *Cho phép tiến trình cấm tất cả các ngắt trước khi vào miền găng, và phục hồi ngắt khi ra khỏi miền găng.*
  - Khi đó, ngắt đồng hồ cũng không xảy ra, do vậy hệ thống không thể tạm dừng hoạt động của tiến trình đang xử lý để cấp phát CPU cho tiến trình khác, nhờ đó tiến trình hiện hành yên tâm thao tác trên miền găng mà không sợ bị tiến trình nào khác tranh chấp.
- Nhận xét:
  - Không được ưa chuộng vì rất thiếu thận trọng khi cho phép tiến trình người dùng được phép thực hiện lệnh cấm ngắt.
  - Hệ thống có nhiều CPU, lệnh cấm ngắt chỉ có tác dụng trên CPU đang xử lý tiến trình hiện tại, các tiến trình hoạt động trên các CPU khác vẫn có thể truy xuất đến miền găng

3.5. Test & Set(1)

- Giải pháp:
  - Tập lệnh máy có thêm một **chỉ thị** đặc biệt cho phép **kiểm tra** và **cập nhật** nội dung một vùng nhớ trong một **thao tác đơn vị**, gọi là chỉ thị ***Test-and-Set Lock* (TSL)**
  - Định nghĩa:

```
Test-and-Setlock(boolean target)
{ boolean temp = target;
   target = TRUE;//thiết lập giá trị mới = True để khóa
   return temp;//lấy giá trị cũ để kiểm tra
  }
```


3.5. Test & Set(2)

- Nếu có hai chỉ thị TSL xử lý đồng thời (trên hai CPU khác nhau), chúng sẽ được xử lý tuần tự.
- Có thể cài đặt giải pháp truy xuất độc quyền với TSL bằng cách sử dụng thêm một biến chung lock, được khởi tạo là FALSE. Tiến trình phải kiểm tra giá trị của biến lock trước khi vào miền găng, nếu lock = FALSE, tiến trình có thể vào miền găng.
- Cấu trúc một ctr sử dụng giải pháp TSL:

```
while (TRUE) {
    while (Test-and-Setlock(lock)){//wait}
    critical-section ();
    lock = FALSE;//ra khỏi đoạn găng, lock=False(không khóa)
    Noncritical-section ();
  }
```

3.5. Test & Set(3)

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

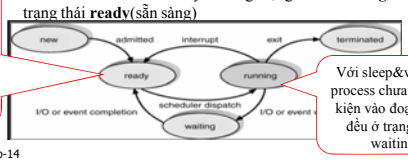
3.6. Kết luận về giải pháp **Busy waiting**

- Hoạt động chung:
  - Tất cả các giải pháp trên đều phải **thực hiện một vòng lặp để kiểm tra** xem có được phép vào đoạn găng.
  - Nếu điều kiện chưa cho phép, tiến trình phải chờ tiếp tục trong vòng lặp kiểm tra này.
  - Các giải pháp buộc tiến trình phải liên tục kiểm tra điều kiện để phát hiện thời điểm thích hợp được vào đoạn găng như thể được gọi các giải pháp « *busy waiting* »-"*Bận **vì** chờ*".
- Hạn chế:
  - Việc kiểm tra như thế tiêu thụ rất nhiều thời gian sử dụng CPU, do vậy tiến trình đang chờ vẫn chiếm dụng CPU(để kiểm tra điều kiện).
- Xu hướng giải quyết vấn đề đồng bộ hoá là nên tránh các giải pháp « ***busy waiting*** »

4. Các giải pháp « **SLEEP and WAKEUP** »(1)

- Khắc phục nhược điểm của các giải pháp **busy waiting** bằng cách cho một tiến trình chưa đủ điều kiện vào đoạn găng chuyển sang trạng thái **waiting**
  - Tạm khóa tiến trình không cho sử dụng CPU ngay vì tiến trình chỉ sử dụng CPU khi ở trạng thái **running**
  - Tiến trình chỉ có thể chuyển sang trạng thái **running** khi đang ở trạng thái **ready**(sẵn sàng)

Với busy waiting, process chưa đủ điều kiện vào đoạn găng đều ở trạng thái ready



Với sleep&wakeup process chưa đủ điều kiện vào đoạn găng đều ở trạng thái waiting

4. Các giải pháp « **SLEEP and WAKEUP** »(2)

- Giải pháp:
  - Hệ điều hành sử dụng 2 thủ tục sleep và wakeup
    - *SLEEP* là một lời gọi hệ thống có tác dụng tạm dừng hoạt động của tiến trình (chuyển sang trạng thái **waiting**) gọi nó và chờ đến khi được một tiến trình khác « đánh thức ».
    - Lời gọi hệ thống *WAKEUP* nhận *một tham số duy nhất : tiến trình sẽ được tái kích hoạt* (đặt về trạng thái **ready**).
  - *Ý tưởng*:
    - Khi một tiến trình chưa đủ điều kiện vào đoạn găng, nó gọi *SLEEP* để tự khóa đến khi có một tiến trình 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

4. Các giải pháp « **SLEEP and WAKEUP** »(3)

- Cấu trúc chương trình trong giải pháp SLEEP and WAKEUP

```
int busy;   // 1 nếu miền găng đang bị chiếm, nếu không là 0
int blocked;   // đếm số lượng tiến trình đang bị tạm khóa
while (TRUE) { if (busy){
                    ++blocked;// = blocked + 1;
                    sleep();
             }
             else busy = 1;
                    critical-section ();
             busy = 0;
             if(blocked){
                    wakeup(process);//truyền vào process cần đánh thức
                    blocked = blocked - 1;
             }
             Noncritical-section ();
}
```


4. Các giải pháp « **SLEEP and WAKEUP** »(4)

- Các giải pháp phổ biến:
  - Simaphore(Dijktra đề xuất)
  - Monitors
  - Trao đổi thông điệp

4.1. Simaphore – Dijkstra(1)

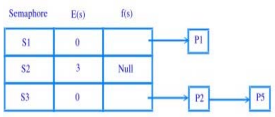
- Một semaphore ***s*** là một *biến(cấu trúc)* có các thuộc tính sau:
  - Một giá trị nguyên *e(s)*
  - Một hàng đợi *f(s)* lưu danh sách các tiến trình đang có trạng thái **waiting**(chờ) trên semaphore *s*
- Hai thao tác được định nghĩa trên semaphore:
  - **Down(s):** giảm giá trị của semaphore e(*s)* đi 1 đơn vị. Nếu semaphore có trị e(s) >= 0 thì tiếp tục xử lý. Ngược lại, tiến trình phải chờ.
  - **Up(s)**: tăng giá trị của semaphore *s* lên 1 đơn vị. Nếu có một hoặc nhiều tiến trình đang chờ trên semaphore *s*, bị khóa bởi thao tác **Down**, thì hệ thống sẽ chọn một trong các tiến trình này để kết thúc thao tác **Down** và cho tiếp tục xử lý

4.1. Simaphore – Dijkstra(2)

- Cài đặt: tiến trình P thực hiện **down** và **up** trên simaphore s

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

Minh họa simaphore



4.1.1. Tổ chức truy xuất độc quyền với Semaphores

- *Semaphore* bảo đảm nhiều tiến trình cùng truy xuất mà không có sự mâu thuẫn.
- Giả sử *n* tiến trình cùng sử dụng một semaphore s, e(s) được khởi gán là 1.
- Để thực hiện đồng bộ hóa, tất cả các tiến trình cần phải áp dụng cùng cấu trúc chương trình:

```
while (TRUE) {
    Down(s)
    critical-section ();
    Up(s)
    Noncritical-section ();
}
```


4.1.2. Tổ chức đồng bộ hóa với Semaphores

- Sử dụng *semaphore* có thể đồng bộ hóa hoạt động của hai tiến trình trong tình huống một tiến trình phải đợi một tiến trình khác hoàn tất thao tác nào đó mới có thể bắt đầu hay tiếp tục xử lý.
- Hai tiến trình chia sẻ một semaphore s, khởi gán e(s) là 0. Cả hai tiến trình có cấu trúc như sau:

```
P1:
while (TRUE) {
    job1();
    Up(s); //đánh thức P2
}
P2:
while (TRUE) {
    Down(s); // chờ P1
    job2();
}
```


4.2. Monitors(1)

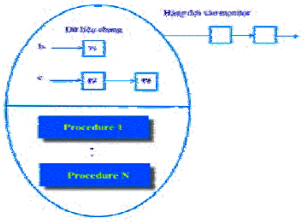
- Do Hoare(1974) Brinch & Hansen đề nghị
- Monitors là cơ chế cao hơn được cung cấp bởi ngôn ngữ lập trình
- Monitors là một cấu trúc dữ liệu(bao gồm các biến và các thủ tục) có đặc điểm:
  - Các biến và cấu trúc dữ liệu bên trong monitor chỉ có thể được thao tác bởi các thủ tục định nghĩa bên trong monitor đó
  - Tại một thời điểm, chỉ có một tiến trình duy nhất được hoạt động bên trong một monitor (*mutual exclusive*).

4.2. Monitors(2)

- Trong một monitor, có thể định nghĩa các *biến điều kiện* và hai thao tác kèm theo là **Wait** và **Signal** như sau : gọi ***c*** là biến điều kiện được định nghĩa trong monitor:
  - **Wait(c):** chuyển trạng thái tiến trình gọi sang trạng thái **waiting**, và đặt tiến trình này vào *hàng đợi trên biến điều kiện* ***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

4.2. Monitors(3)

- Minh họa Monitor:



4.2. Monitors(4)

- Cài đặt : Mỗi monitor có một hàng đợi toàn cục lưu các tiến trình đang chờ được vào monitor, ngoài ra, mỗi biến điều kiện ***c*** cũng gắn với một hàng đợi ***f(c)*** và hai thao tác trên đó được định nghĩa như sau:

Wait(c) :

```
status(P)= waiting;
enter(P,f(c));
```

Signal(c) :

```
if (f(c) != NULL){
exit(Q,f(c)); //chọn 1 tiến trình chơ Q trên c
statusQ) = ready;
enter(Q,ready-list);
}
```


4.2. Monitors(5)

- Sử dụng:
  - Với mỗi nhóm tài nguyên cần chia sẻ, có thể định nghĩa một monitor trong đó đặc tả tất cả các thao tác trên tài nguyên này với một số điều kiện

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

4.2. Monitors(6)

- Sử dụng(tiếp)
  - Các tiến trình muốn sử dụng tài nguyên chung này chỉ có thể thao tác thông qua các thủ tục bên trong monitor được gắn kết với tài nguyên:
  - Cấu trúc tiến trình Pi trong giải pháp monitor

```
while (TRUE) {
Noncritical-section ();
<monitor>.Action1(Pi);//critical-section();
Noncritical-section ();
}
```


4.3. Trao đổi thông điệp(1)

- Giải pháp: dựa trên cơ sở trao đổi thông điệp với hai primitive(thao tác nguyên tử) 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 thừ một tiến trình hay từ bất kỳ một tiến trình nào, tiến trình gọi sẽ chờ nếu không có thông điệp nào để nhận.

4.3. Trao đổi thông điệp(2)

- 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.
  - 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

4.3. Trao đổi thông điệp(3)

- Cấu trúc tiến trình yêu cầu tài nguyên trong giải pháp message

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


Q & A

- List Câu hỏi