Nhắc lại Chương 4: Định thời CPU

- Tại sao phải định thời? Các tiêu chuẩn định thời CPU?
- Có bao nhiêu giải thuật định thời? Kể tên và mô tả và nêu ưu điểm, nhược điểm của từng giải thuật định thời? FCFS, SJF, SRTF, RR, Priority Scheduling, ...
- Tính các giá trị thời gian đợi, thời gian đáp ứng và thời gian hoàn thành trung bình và vẽ giản đồ Gaint

Thread	Priority	Burst	Arrival
P_1	20	20	0
P_2	30	25	25
P_3	15	25	20

Chương 5: Đồng bộ hóa tiến trình

Mục tiêu của chương

- Giới thiệu các giải pháp cụ thể để xử lý bài toán đồng bộ hoá.
  - Giải pháp « **busy waiting** »
  - Giải pháp « **sleep and wakeup** »

Chương 5: Đồng bộ hóa tiến trình

Kiến thức sinh viên phải nắm được sau chương này

- Nhiệm vụ của việc đồng bộ hóa tiến trình.
- Hiểu và áp dụng được các giải pháp đồng bộ, đặc biệt với các giải pháp **"sleep and wakeup".**

Chương 5: Đồng bộ hóa tiến trình

Đưa ra bài toán

- Tại sao cần đồng bộ hóa tiến trình?
- Đối tượng tác động?
- Thuật ngữ miền găng?

Chương 5: Đồng bộ hóa tiến trình

Tại sao cần đồng bộ hóa tiến trình?

- Khảo sát các process/thread thực thi đồng thời và chia sẻ dữ liệu (qua shared memory, file): nếu không có sự kiểm soát khi truy cập các dữ liệu chia sẻ thì có thể đưa đến ra trường hợp không nhất quán dữ liệu (data inconsistency).
- Để duy trì sự nhất quán dữ liệu, hệ thống cần có cơ chế bảo đảm sự thực thi có trật tự của các process đồng thời (độc quyền truy xuất, cơ chế phối hợp,..).

Ví dụ

Hai tiến trình P_1 và P_2 cùng truy xuất dữ liệu chung là một tài khoản ngân hàng:

```
if (So_du > Tien_rut)
        So_du = So_du – Tien_rut
else
        Access denied!
```

Chuyển khoản vs Rút tiền ATM

- P1

```
if (So_du > Tien_rut)
```

```
So_du = So_du – Tien_rut
else
   Access denied!
```

- P2

```
if (So_du > Tien_rut)
   So_du = So_du –
   Tien_rut
```

```
else
   Access denied!
```

Tranh đoạt điều khiển – Race condition

```
i=0;
```

Thread a:

```
while(i < 10)
    i = i +1;
print "A won!";
```

Thread b:

```
while(i > -10)
    i = i - 1;
print "B won!";
```

Miền găng – Critical Section

- Race condition (tương tranh): nhiều tiến trình cùng thực thi mà kết quả phụ thuộc vào thứ tự thực thi của các tiến trình.
- Miền găng (critical section): đoạn chương trình có khả năng gây ra lỗi truy xuất đối với tài nguyên chung.

Chương 5: Đồng bộ hóa tiến trình

Vấn đề Critical Section (CS – Miền găng):

- Giả sử có n process truy xuất đồng thời dữ liệu chia sẻ
- Cấu trúc của mỗi process Pi có đoạn code như sau:

```
Do {
        entry section      /* vào critical section */
                critical section      truy xuất dữ liệu chia xẻ */
            exit  section      /* rời critical section */
                remainder section  /* làm những việc khác */
} While (1)
```

- Trong mỗi process có những đoạn code có chứa các thao tác lên dữ liệu chia sẻ. Đoạn code này được gọi là vùng tranh chấp (critical section, CS).

Vấn đề Critical Section: phải bảo đảm sự loại trừ tương hỗ (mutual exclusion, mutex), tức là khi một process đang thực thi trong vùng tranh chấp, không có process nào khác đồng thời thực thi các lệnh trong vùng tranh chấp.

Yêu cầu của lời giải cho CS Problem

- Lời giải phải thỏa ba tính chất:
  - (1) Loại trừ tương hỗ (Mutual exclusion): Khi một process P đang thực thi trong vùng tranh chấp (CS) của nó thì không có process Q nào khác đang thực thi trong CS của Q.
  - (2) Progress: Một tiến trình tạm dừng bên ngoài vùng tranh chấp không được ngăn cản các tiến trình khác vào vùng tranh chấp.
  - (3) Chờ đợi giới hạn (Bounded waiting): Mỗi process chỉ phải chờ để được vào vùng tranh chấp trong một khoảng thời gian có hạn định nào đó. Không xảy ra tình trạng đói tài nguyên (starvation).
  - Không có giả thiết nào đặt ra cho sự liên hệ về tốc độ của các tiến trình, cũng như về số lượng bộ xử lý trong hệ thống.

Chương 5: Đồng bộ hóa tiến trình

- **Nhóm Giải pháp « busy and waiting »**
  - *Các giải pháp phần mềm*
  - *Các giải pháp phần cứng*
- **Nhóm giải pháp « SLEEP and WAKEUP »**
  - *Semaphore*
  - *Monitors*
  - *Trao đổi thông điệp*

Chương 5: Đồng bộ hóa tiến trình

Nhóm Giải pháp « busy and waiting »

- Tiếp tục tiêu thụ CPU trong khi chờ đợi vào vùng tranh chấp
- Không đòi hỏi sự trợ giúp của Hệ điều hành

While (chưa có quyền) do_nothing() ;

CS;

Từ bỏ quyền sử dụng CS

Chương 5: Đồng bộ hóa tiến trình

Nhóm giải pháp « SLEEP and WAKEUP »

- Từ bỏ CPU khi chưa được vào vùng tranh chấp
- Cần Hệ điều hành hỗ trợ

```
if (chưa có quyền) Sleep() ;

CS;

Wakeup (somebody);
```

Chương 5: Đồng bộ hóa tiến trình

Giải pháp « busy waiting »

- Các giải pháp phần mềm
  - Sử dụng các biến cờ hiệu
  - Sử dụng việc kiểm tra luân phiên
  - Giải pháp của Peterson
- Các giải pháp phần cứng
  - Cấm ngắt
  - Chỉ thị TSL

Chương 5: Đồng bộ hóa tiến trình

Các giải pháp phần mềm

a) Sử dụng các biến cờ hiệu:

- **Tiếp cận:** Các tiến trình chia sẻ một biến chung đóng vai trò lock , được khởi động=0.
- Một tiến trình muốn vào miền găng trước tiên phải kiểm tra giá trị của biến lock. Nếu lock = 0, tiến trình đặt lock = 1 và đi vào miền găng.
- Nếu lock đang nhận giá trị 1, tiến trình phải chờ bên ngoài miền găng cho đến khi lock có giá trị 0.

Chương 5: Đồng bộ hóa tiến trình

Các giải pháp phần mềm

a) Sử dụng các biến cờ hiệu:

- **Cấu trúc của 1 tiến trình**

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

Chương 5: Đồng bộ hóa tiến trình

Các giải pháp phần mềm

Thảo luận về biện pháp sử dụng biến "Lock"

- Có thể vi phạm điều kiện 1: hai tiến trình cùng ở trong miền găng tại một thời điểm.

- **P1**

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

- **P2**

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

Chương 5: Đồng bộ hóa tiến trình

Các giải pháp phần mềm

b) Sử dụng việc kiểm tra luân phiên

- **Tiếp cận :** Đây là một giải pháp đề nghị cho hai tiến trình.
- Hai tiến trình này sử dụng chung biến *turn* (phản ánh phiên tiến trình nào được vào miền găng), được khởi động với giá trị 0. Nếu *turn = 0*, chỉ có tiến trình A được vào miền găng. Nếu *turn = 1*, chỉ có tiến trình B được đi vào miền găng.

Chương 5: Đồng bộ hóa tiến trình

Các giải pháp phần mềm

b) Sử dụng việc kiểm tra luân phiên

```
while (TRUE) {
while (turn != 0); // wait
critical-section ();
turn = 1;
Noncritical-section ();
}// Cấu trúc tiến trình A
```

```
while (TRUE) {
while (turn != 1); // wait
critical-section ();
turn = 0;
Noncritical-section ();
}// Cấu trúc tiến trình B
```

Điều gì xảy ra nếu P0 có RS (remainder section) rất lớn còn P1 có RS nhỏ?

Chương 5: Đồng bộ hóa tiến trình

Thảo luận về biện pháp kiểm tra luân phiên

- Ngăn chặn được tình trạng hai tiến trình cùng vào miền găng. (?)
- Có thể vi phạm điều kiện 3: một tiến trình có thể bị ngăn chặn vào miền găng bởi một tiến trình khác không ở trong miền găng. (?)

Chương 5: Đồng bộ hóa tiến trình

Các giải pháp phần mềm

c) Giải pháp của Peterson

- **Tiếp cận** : Petson đưa ra một giải pháp kết hợp ý tưởng của cả hai giải pháp kể trên. Các tiến trình chia sẻ hai biến chung :
- int turn; // đến phiên ai
- int interesse[2]; // khởi động là FALSE

Chương 5: Đồng bộ hóa tiến trình

Các giải pháp phần mềm

c) Giải pháp của Peterson

```
while (TRUE) {
    int j = 1-i; // nếu I là tiến trình 1 thì  j là tiến trình 0
    interesse[i]= TRUE;
    turn = j;
    while (turn == j && interesse[j]==TRUE);
    critical-section ();
    interesse[i] = FALSE;
    Noncritical-section ();
}
```

Cấu trúc tiến trình Pi trong giải pháp Peterson

Giải thuật 3 (Peterson) cho 2 tiến trình

Process P_0

```
do {
   /*  0 wants in  */
   flag[0] = true;
   /*  0 gives a chance to 1  */
   turn = 1;
   while (flag[1] &&turn == 1);
      critical section
   /*  0 no longer wants in  */
   flag[0] = false;
      remainder section
} while(1);
```

Process P_1

```
do {
   /*  1 wants in  */
   flag[1] = true;
   /*  1 gives a chance to 0  */
   turn = 0;
   while (flag[0] && turn == 0);
      critical section
   /*  1 no longer wants in  */
   flag[1] = false;
      remainder section
} while(1);
```

Chương 5: Đồng bộ hóa tiến trình

Thảo luận về giải pháp của Peterson

- Giải thuật 3 thỏa mutual exclusion, progress, và bounded waiting
- Mutual exclusion được đảm bảo bởi vì
  - P0 và P1 đều ở trong CS nếu và chỉ nếu flag[0] = flag[1] = true và turn = i cho mỗi Pi (không thể xảy ra)
- Chứng minh thỏa yêu cầu về progress và bounded waiting
  - Pi không thể vào CS nếu và chỉ nếu bị kẹt tại vòng lặp while() với điều kiện flag[j] = true và turn = j
  - Nếu Pj không muốn vào CS thì flag[j] = false và do đó Pi có thể vào CS

Chương 5: Đồng bộ hóa tiến trình

Thảo luận về giải pháp của Peterson

- Nếu Pj đã bật flag[j] = true và đang chờ tại while() thì có chỉ hai trường hợp là turn = i hoặc turn = j
  - Nếu turn = i thì Pi vào CS. Nếu turn = j thì Pj vào CS nhưng sẽ bật flag[j] = false khi thoát ra -> cho phép Pi và CS
- Nhưng nếu Pj có đủ thời gian bật flag[j] = true thì Pj cũng phải gán turn = i
- Vì Pi không thay đổi trị của biến turn khi đang kẹt trong vòng lặp while(), Pi sẽ chờ để vào CS nhiều nhất là sau một lần Pj vào CS (bounded waiting)

Chương 5: Đồng bộ hóa tiến trình

Từ software đến hardware

- Khuyết điểm của các giải pháp software:
  - Các process khi yêu cầu được vào vùng tranh chấp đều phải liên tục kiểm tra điều kiện (busy waiting), tốn nhiều thời gian xử lý của CPU
  - Nếu thời gian xử lý trong vùng tranh chấp lớn, một giải pháp hiệu quả nên có cơ chế block các process cần đợi.
- Các giải pháp phần cứng:
  - Cấm ngắt (disable interrupts)
  - Dùng các lệnh đặc biệt

Chương 5: Đồng bộ hóa tiến trình

Các giải pháp phần cứng

a) Cấm ngắt

- **Tiếp cận:** 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.

Chương 5: Đồng bộ hóa tiến trình

Thảo luận về phương pháp cấm ngắt

- Giải pháp này không được ưa chuộng vì rất thiếu thận trọng khi cho phép tiến trình người dùng được phép thực hiện lệnh cấm ngắt.
- Nếu hệ thống có nhiều bộ xử lý, lệnh cấm ngắt chỉ có tác dụng trên bộ xử lý đang xử lý tiến trình, còn các tiến trình hoạt động trên các bộ xử lý khác vẫn có thể truy xuất đến miền găng !
- Có thể một tiến trình sẽ phải chờ vô hạn.

Chương 5: Đồng bộ hóa tiến trình

Các giải pháp phần cứng

b) Chỉ thị TSL (Test-and-Set)

- **Tiếp cận:** Nhiều máy tính cung cấp một chỉ thị đặc biệt cho phép kiểm tra và cập nhật nội dung một vùng nhớ trong một thao tác không thể phân chia, gọi là chỉ thị *Test-and-Set Lock* (TSL) và được định nghĩa như sau:

Chương 7: Đồng bộ hóa tiến trình

b) Chỉ thị TSL (Test-and-Set)

- Đọc và ghi một biến trong một thao tác atomic (không chia cắt được)

```
boolean TestAndSet( boolean *target){
   boolean rv = *target;
   *target = true;
   return rv;
}
```

- Shared data:
  boolean lock = false;

- Process P_i :

```
do {
    while (TestAndSet(&lock));
        critical section
    lock = false;
        remainder section
} while (1);
```

Chương 5: Đồng bộ hóa tiến trình

Thảo luận về TSL (Test-and-Set)

- Mutual exclusion được bảo đảm: nếu Pi vào CS, các process Pj khác đều đang busy waiting
- Khi Pi ra khỏi CS, quá trình chọn lựa process Pj vào CS kế tiếp là tùy ý ⇒ không bảo đảm điều kiện bounded waiting. Do đó có thể xảy ra starvation (bị bỏ đói)
- Các processor (ví dụ Pentium) thông thường cung cấp một lệnh đơn là Swap(a, b) có tác dụng hoán chuyển nội dung của a và b.
  - Swap(a, b) cũng có ưu nhược điểm như TestAndSet

Chương 5: Đồng bộ hóa tiến trình

Giải thuật dùng TestAndSet thoả mãn 3 yêu cầu

```
do {
        waiting[ i ] = true;
        key = true;
        while (waiting[ i ] && key)
            key = TestAndSet(lock);
        waiting[ i ] = false;
            critical section
        j = (i + 1) % n;
        while ( (j != i)  &&  !waiting[ j ] )
            j = (j + 1) % n;
        if (j == i)
            lock = false;
        else
            waiting[ j ] = false;
            remainder section
} while (1)
```

Chương 5: Đồng bộ hóa tiến trình

Thảo luận về các giải pháp "busy waiting"

- Tất cả các giải pháp busy waiting buộc tiến trình phải liên tục kiểm tra điều kiện để phát hiện thời điểm thích hợp được vào miền găng.
- Việc kiểm tra như thế tiêu thụ rất nhiều thời gian sử dụng CPU, do vậy tiến trình đang chờ vẫn chiếm dụng CPU.
- Xu hướng giải quyết vấn đề đồng bộ hoá là nên tránh các giải pháp « ***busy waiting*** ».

Chương 5: Đồng bộ hóa tiến trình

Các giải pháp « SLEEP and WAKEUP »

- Semaphore
- Monitors
- Trao đổi thông điệp

Chương 7: Đồng bộ hóa tiến trình

Tiếp cận

- Phải loại bỏ được bất tiện của giải pháp « busy waiting ».
- Theo hướng cho một tiến trình chưa đủ điều kiện vào miền găng chuyển sang trạng thái blocked, từ bỏ quyền sử dụng CPU.

Chương 5: Đồng bộ hóa tiến trình

Tiếp cận

- Ý tưởng sử dụng SLEEP và WAKEUP: khi một tiến trình chưa đủ điều kiện vào miền 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 gang
- SLEEP và WAKEUP là các thao tác đơn (không thể bị ngắt ở giữa)

Chương 5: Đồng bộ hóa tiến trình

Giải pháp "sleep and wakeup"

```
int busy; // 1 nếu miền găng đang bị chiếm, ngược lại là 0
int blocked; // đếm số lượng tiến trình đang bị 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 ();
}
```

Tình huống:

-A vào miền găng

-B vào sau nên phải "ngủ"

-B chưa "ngủ" thì CPU chuyển cho A.

-A ra khỏi miền găng và "đánh thức" B

-B đang "thức" nên không nhận tín hiệu đánh thức.

-B không bao giờ được vào miền găng!

Chương 5: Đồng bộ hóa tiến trình

Giải pháp "sleep and wakeup"

- Tồn tại: Tiến trình vẫn có thể bị chặn không cho vào miền găng do:
  - Thao tác kiểm tra điều kiện và thao tác sleep có thể bị ngắt.
  - Tín hiệu wakeup có thể bị "thất lạc"
- Giải pháp:
  - Dùng semaphore
  - Dùng monitor
  - Dùng message

Chương 7: Đồng bộ hóa tiến trình

3.2.1. Semaphore

- **Tiếp cận:** Được **Dijkstra** đề xuất vào 1965, một semaphore ***s*** là một *biến* có các thuộc tính sau:

  Một giá trị nguyên dương ***e(s)***

  Một hàng đợi ***f(s)*** lưu danh sách các tiến trình đang bị khóa (chờ) trên semaphore ***s***

Chương 5: Đồng bộ hóa tiến trình

3.2.1. Semaphore

- Chỉ có hai thao tác được định nghĩa trên semaphore

Down(s):

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

Up(s):

```
e(s) = e(s) + 1;
if e(s) < 0 {
exit(Q,f(s));
status (Q) = ready;
enter(Q,ready-list);}
```

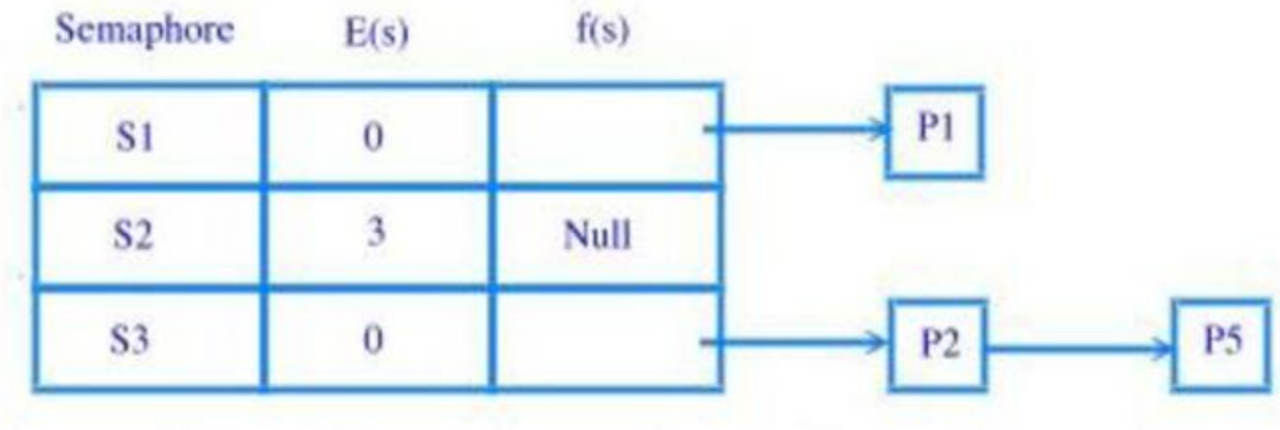
Down(s): *giảm giá trị e đi 1. Nếu e ≥ 0 thì tiếp tục xử lý. Ngược lại, nếu e < 0, tiến trình phải chờ.*

Up(s): *tăng giá trị của e lên 1. Nếu có tiến trình đang chờ thì chọn một tiến trình để đánh thức.*

Chương : Đồng bộ hóa tiến trình

Semaphore

- Trị tuyệt đối của semaphore |e(s)| cho biết số tiến trình đang chờ trên semaphore.
- Để semaphore hoạt động được các thao tác down(s), up(s) cần thực hiện một cách không bị phân chia.



Chương 5: Đồng bộ hóa tiến trình

Semaphore

- Ứng dụng 1:
  - Giải quyết điều kiện 1 của miền găng:

    Có n tiến trình dùng chung một semaphore để đồng bộ, semaphore được khởi tạo = 1.

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

Tiến trình đầu tiên vào được miền găng (được truy xuất tài nguyên).

Các tiến trình sau phải chờ vì e(s)<0.

- Giải pháp trên đồng bộ hóa 2 hay nhiều tiền trình cùng truy xuất vào tài nguyên S

Chương 5: Đồng bộ hóa tiến trình

Semaphore

- Ứng dụng 2:

■ Đồng bộ tiến trình

Dùng chung 1 semaphore với giátrị khởi tạo =0

P1:

```
while (TRUE) {
   job1();
   Up(s); //đánh thức P2
}
```

P2:

```
while (TRUE) {
   Down(s); // chờ P1
   job2();
}
```

Ngữ cảnh đồng bộ: có hai tiến trình tương tranh, và tiến trình này phải chờ tiến trình kia kết thúc thì mới xử lý được.

Chương 5: Đồng bộ hóa tiến trình

Thảo luận về Semaphore

- Đã giải quyết hoàn toàn lỗi truy xuất
- Không chiếm dụng CPU khi tiến trình bị blocked
- Down và Up phải được thực hiện một cách không bị phân chia.
- Phức tạp trong lập trình, không được đặt nhầm lẫn vị trí giữa Down và Up

Người lập trình quên gọi Up(s), hậu quả là từ đó về sau không có tiến trình nào vào được miền găng!

Chương 5: Đồng bộ hóa tiến trình

Monitors

- **Tiếp cận:** Để có thể dễ viết đúng các chương trình đồng bộ hóa hơn, Hoare(1974) và Brinch & Hansen (1975) đã đề nghị một cơ chế cao hơn được cung cấp bởi ngôn ngữ lập trình , là *monitor*.

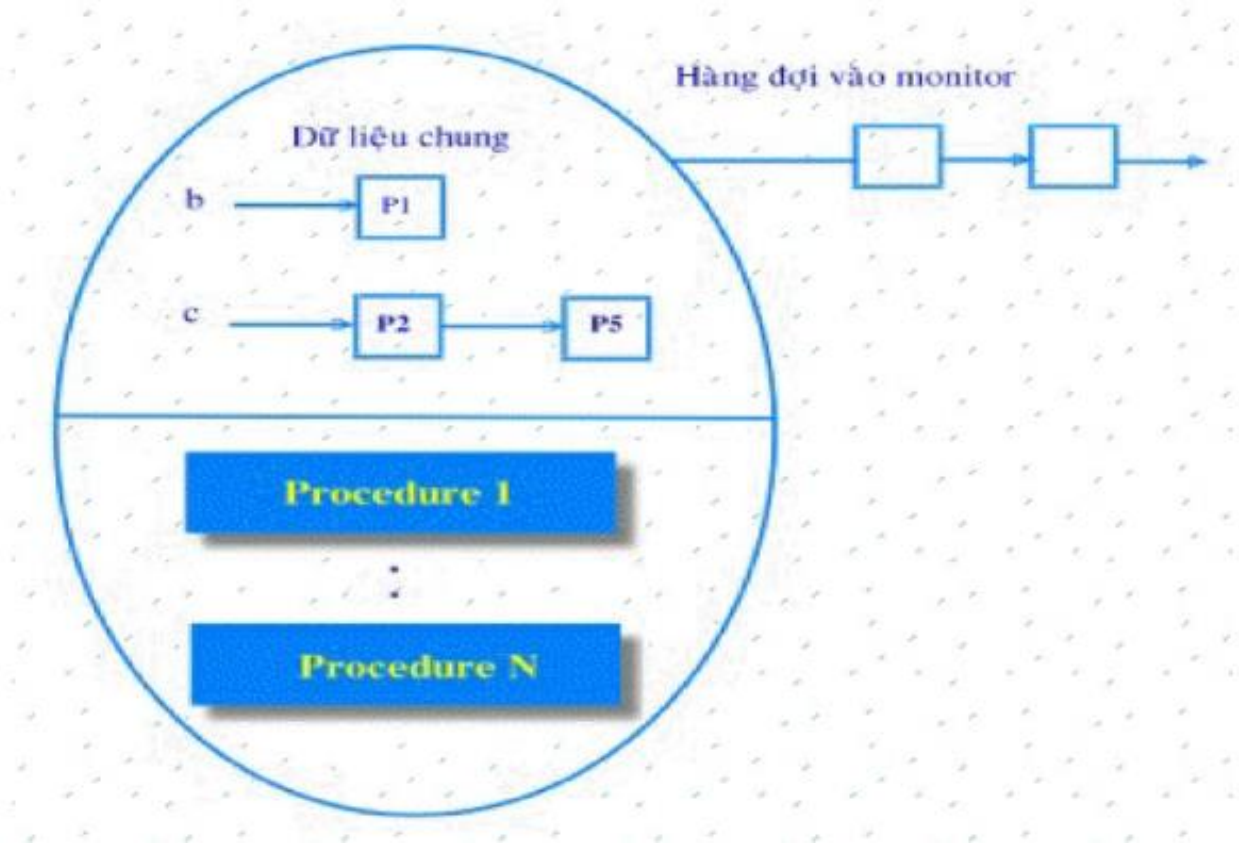
Chương 5: Đồng bộ hóa tiến trình

Monitors

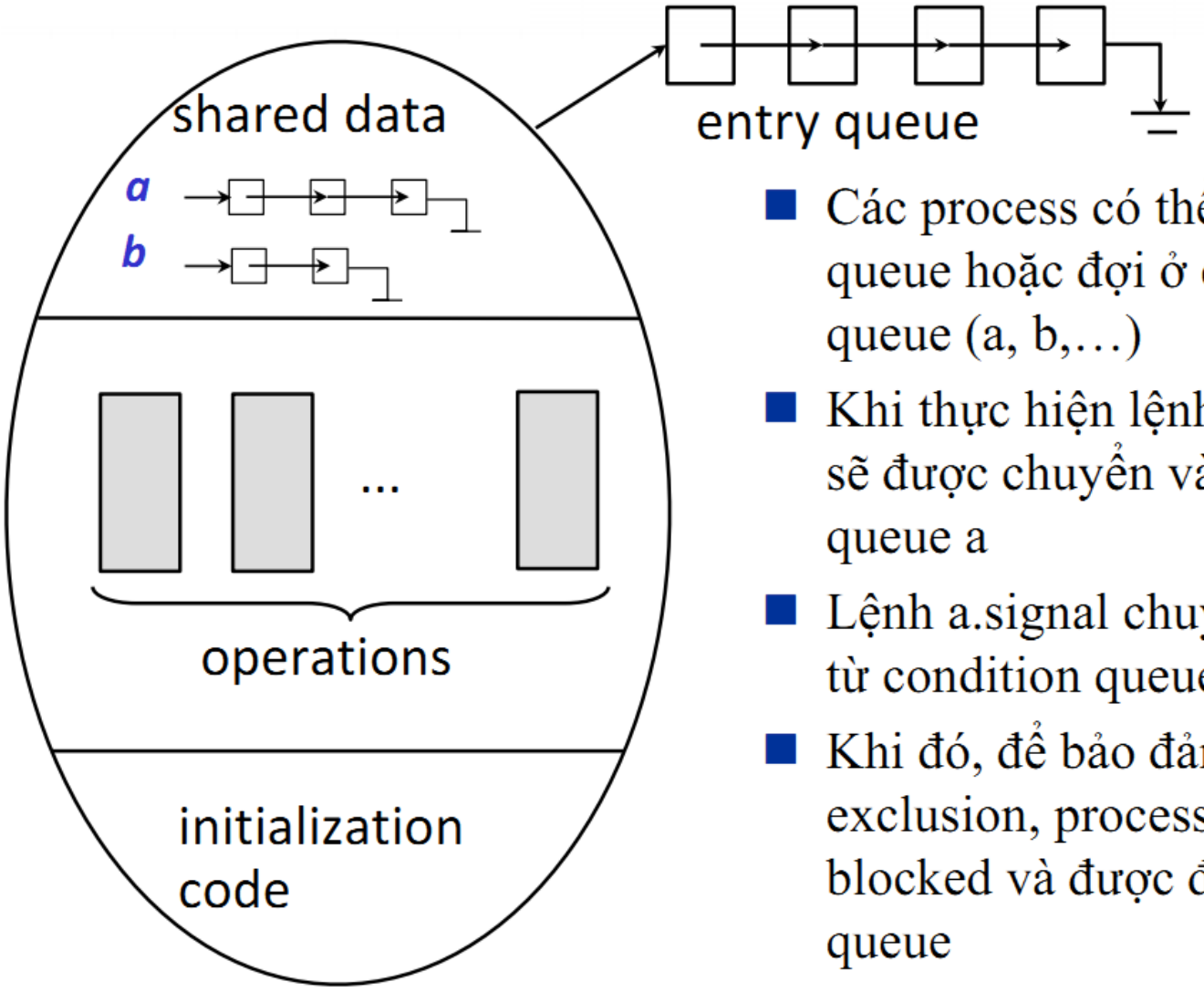
- Monitor là một cấu trúc đặc biệt bao gồm các thủ tục, các biến và cấu trúc dữ liệu có các thuộc tính sau :
  - Các biến điều kiện (c) và cấu trúc dữ liệu hàng đợi chứa các tiến trình bị khóa f(c) bên trong monitor chỉ có thể được thao tác bởi các thủ tục bên trong nó, và 2 thao tác:
    - Wait(c): 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.
  - Tại một thời điểm, chỉ có một tiến trình duy nhất được hoạt động bên trong một monitor (*mutual exclusive*).

Chương 5: Đồng bộ hóa tiến trình

Monitor



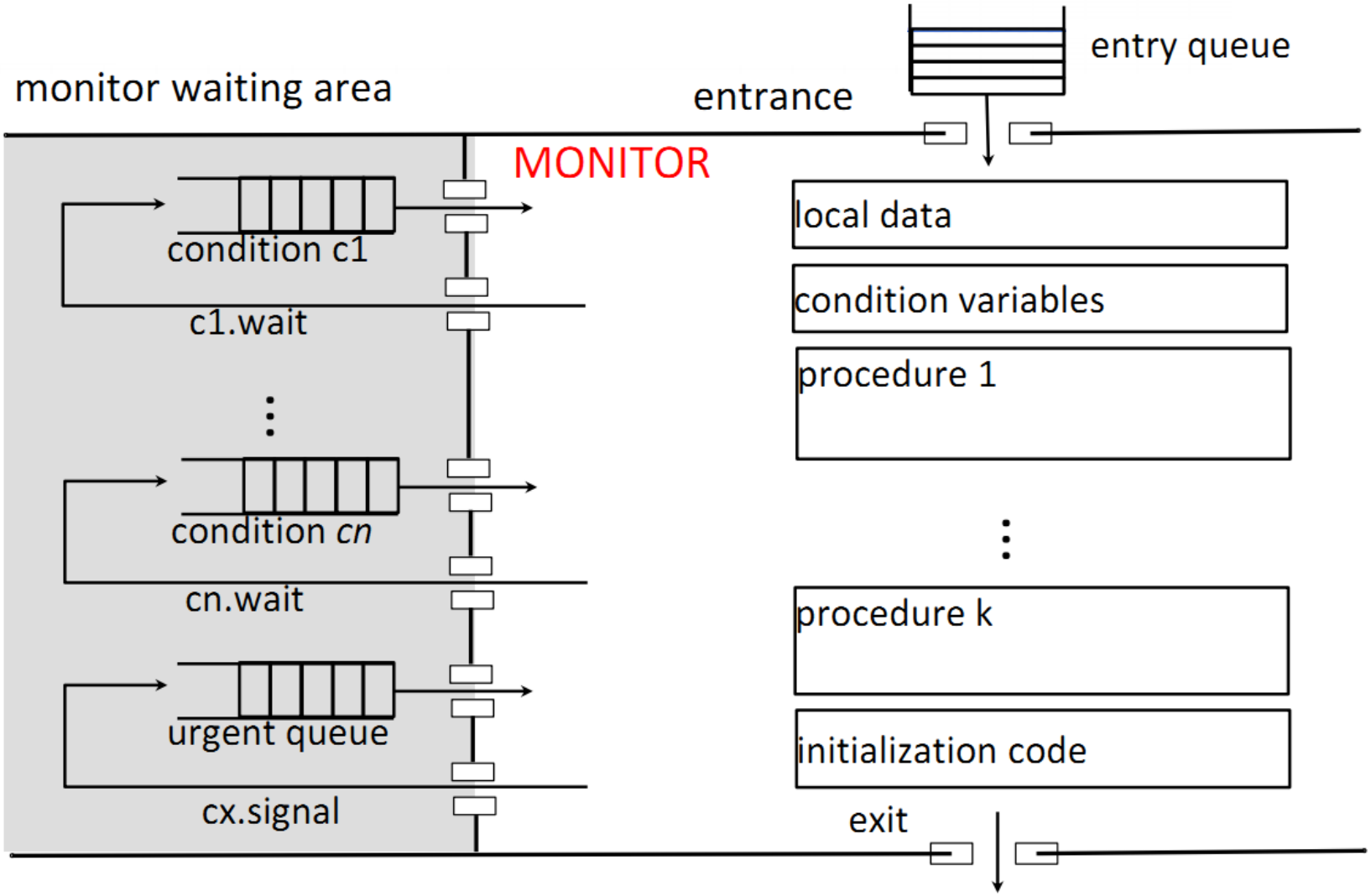
Chương 5: Đồng bộ hóa tiến trình



- Các process có thể đợi ở entry queue hoặc đợi ở các condition queue (a, b,…)
- Khi thực hiện lệnh a.wait, process sẽ được chuyển vào condition queue a
- Lệnh a.signal chuyển một process từ condition queue a vào monitor
- Khi đó, để bảo đảm mutual exclusion, process gọi a.signal sẽ bị blocked và được đưa vào urgent queue

Chương 5: Đồng bộ hóa tiến trình

Monitor có condition variable (tt)



Chương 5: Đồng bộ hóa tiến trình

- **Cài đặt :** trình biên dịch chịu trách nhiệm thực hiện việc truy xuất độc quyền đến dữ liệu trong monitor. Để thực hiện điều này, một semaphore nhị phân thường được sử dụng.

Thao tác Wait và Signal

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

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

Chương 5: Đồng bộ hóa tiến trình

Monitors

- **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 nào đó.:

```
monitor <tên monitor >
        condition <danh sách các biến điều kiện>;
        <variables>;
        procedure Action1();  {  }
        procedure Actionn();  {  }
end monitor; // Cấu trúc một monitor
```

Monitor dùng để điều khiển truy xuất độc quyền đối với tài nguyên. Mỗi tài nguyên có một monitor riêng và tiến trình truy xuất thông qua monitor đó

Chương 7: Đồng bộ hóa tiến trình

3.2.2. Monitors

- 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:

```
while (TRUE) {
        Noncritical-section ();
        <monitor>.Action_i; //critical-section();
        Noncritical-section ();
}// Cấu trúc tiến trình Pi trong giải pháp monitor
```

Nhận xét: -Thao tác trên monitor do trình biên dịch thực hiện -> giảm sai sót.

-Trình biên dịch phải hỗ trợ monitor

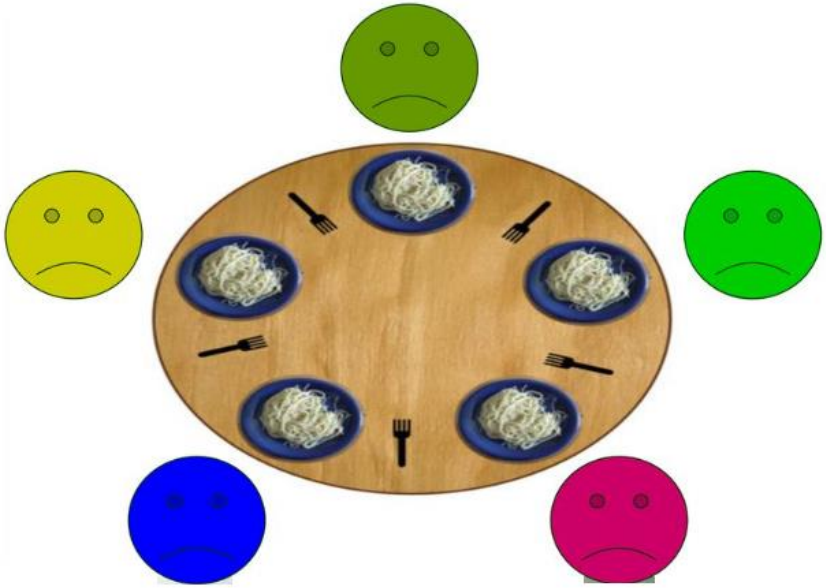
Chương 5: Đồng bộ hóa tiến trình

Thảo luận về Monitors

- Với monitor, việc truy xuất độc quyền được bảo đảm bởi trình biên dịch mà không do lập trình viên, do vậy nguy cơ thực hiện đồng bộ hóa sai giảm rất nhiều.
- Tuy nhiên giải pháp monitor đòi hỏi phải có một ngôn ngữ lập trình định nghĩa khái niệm monitor, và các ngôn ngữ như thế chưa có nhiều.

Chương 5: Đồng bộ hóa tiến trình

Bài tóan triết gia ăn tối



- 5 triết gia ngồi ăn và suy nghĩ
- Mỗi người cần 2 chiếc đũa (chopstick) để ăn
- Trên bàn chỉ có 5 đũa
- Bài toán này minh họa sự khó khăn trong việc phân phối tài nguyên giữa các process sao cho không xảy ra deadlock và starvation

Chương 5: Đồng bộ hóa tiến trình

- Bài toán Triết gia ăn tối

Xây dựng monitor

```
monitor philosopher{
    enum (thinking, hungry, eating} state[5];
    condition self[5];
    void init();
    void test(int i);
    void pickup( int i);
    void putdown (int i);
}
```

Chương 5: Đồng bộ hóa tiến trình

- Bài toán Triết gia ăn tối

Khởi tạo monitor

- Tất cả các triết gia đang suy nghĩ…

```
void init (){
for(int i=0; i < 5; i++)
    state[i] = thinking;
}
```

Chương 5: Đồng bộ hóa tiến trình

- Bài toán Triết gia ăn tối

Kiểm tra điều kiện trước khi ăn

- Nếu TGi đang đói và cả hai TG bên cạnh đều không ăn thì TGi ăn.

```
void test (int i) {
    if ((state[i] == hungry) && state[(i + 4)%5] !=
    eating) && state[(i + 1)%5] != eating)
    self[i].signal(); //Đánh thức TGi
    state[i] = eating;
}
```

Chương 5: Đồng bộ hóa tiến trình

- Bài toán Triết gia ăn tối

Lấy một chiếc nĩa

```
void pickup(int i) {
   state[i] = hungry;
   test(i);
   if (state[i] != eating)
        self[i].wait();   //TGi chờ đến lượt mình
}
```

Trả một chiếc nĩa

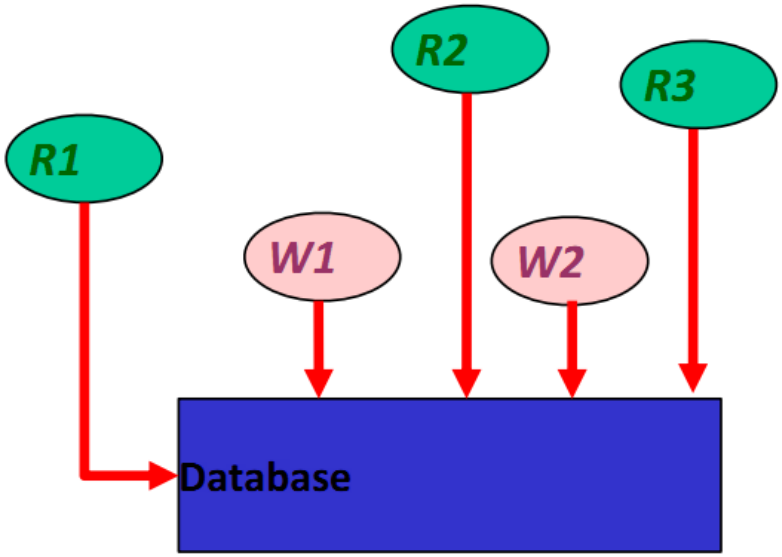
```
void putdown(int i){
        state[i] = thinking;
        test((i + 4) % 5);
        test((i + 1) % 5);
}
```

Cài đặt tiến trình

```
Philosophers pp;
Tiến trình i:
while(1) {
   Noncritical-section();
   pp.pickup(i);
   eat();
   pp.putdown(i);
   Noncritical-section();
}
```

Chương 5: Đồng bộ hóa tiến trình

- Bài toán Reader-Writers
  - Writer không được cập nhật dữ liệu khi có một Reader đang truy xuất CSDL
  - Tại một thời điểm, chỉ cho phép một Writer được sửa đổi nội dung CSDL



Chương 5: Đồng bộ hóa tiến trình

- Bài toán Reader-Writers
  - Bộ đọc trước bộ ghi (first reader-writer)
  - Dữ liệu chia sẻ

```
semaphore  mutex = 1;
semaphore  wrt   = 1;
int        readcount  = 0;
```

  - Writer process

```
wait(wrt);
...
writing is performed
...
signal(wrt);
```

Reader process

```
wait(mutex);
readcount++;
if (readcount == 1)
        wait(wrt);
signal(mutex);
...
reading is performed
...
wait(mutex);
readcount--;
if (readcount == 0)
        signal(wrt);
signal(mutex);
```

Chương 5: Đồng bộ hóa tiến trình

- Bài toán Reader-Writers
  - mutex: "bảo vệ" biến readcount
  - wrt
    - Bảo đảm mutual exclusion đối với các writer
    - Được sử dụng bởi reader đầu tiên hoặc cuối cùng vào hay ra khỏi vùng tranh chấp.
  - Nếu một writer đang ở trong CS và có n reader đang đợi thì một reader được xếp trong hàng đợi của wrt và n − 1 reader kia trong hàng đợi của mutex
  - Khi writer thực thi signal(wrt), hệ thống có thể phục hồi thực thi của một trong các reader đang đợi hoặc writer đang đợi.

Chương : Đồng bộ hóa tiến trình

Các vấn đề đồng bộ hóa

- Vấn đề Người sản xuất – Người tiêu thụ (Producer-Consumer)

Chương : Đồng bộ hóa tiến trình

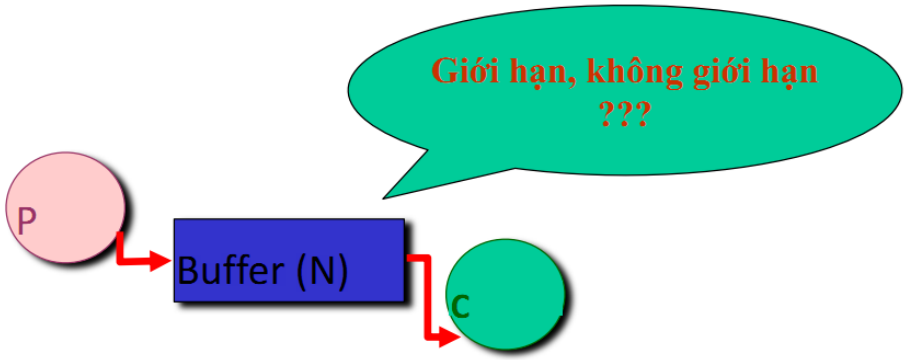
Vấn đề Producer-Consumer

- **Nêu vấn đề**: Hai tiến trình cùng chia sẻ một bộ đệm có kích thước giới hạn.
  - Một tiến trình đóng vai trò người sản xuất – tạo ra dữ liệu và đặt dữ liệu vào bộ đệm.
  - Một tiến trình đóng vai trò người tiêu thụ – lấy dữ liệu từ bộ đệm ra để xử lý.

Chương 7: Đồng bộ hóa tiến trình

Vấn đề Producer-Consumer

- **Nêu vấn đề**: Để đồng bộ hóa hoạt động của hai tiến trình sản xuất, tiêu thụ cần tuân thủ các quy định sau :
  - Producer P không được ghi dữ liệu vào bộ đệm đã đầy.
  - Consumer C không được đọc dữ liệu từ bộ đệm đang trống.
  - Hai tiến trình sản xuất P và tiêu thụ C không được thao tác trên bộ đệm (Buffer[N]) cùng lúc.



Chương 7: Đồng bộ hóa tiến trình

Vấn đề Producer-Consumer

- Quá trình Producer

```
item nextProduce;
while(1){
  while(count == BUFFER_SIZE); /*ko lam gi*/
  buffer[in] = nextProducer;
  count++;
  in = (in+1)%BUFFER_SIZE;
```

biến count được chia sẻ giữa producer và consumer

- Quá trình Consumer

```
item nextConsumer;
while(1){
  while(count == 0); /*ko lam gi*/
  nextConsumer = buffer[out];
  count--;
  out = (out+1)%BUFFER_SIZE;
```

Chương 7: Đồng bộ hóa tiến trình

Giải pháp với Semaphore

- Sử dụng ba semaphore :
  - ***full***, đếm số chỗ đã có dữ liệu trong bộ đệm
  - ***empty***, đếm số chỗ còn trống trong bộ đệm
  - ***mutex***, kiểm tra việc Producer và Consumer không truy xuất đồng thời đến bộ đệm

Chương 7: Đồng bộ hóa tiến trình

Giải pháp với Semaphore

BufferSize = 3; **semaphore** mutex = 1; **semaphore** empty = BufferSize; **semaphore** full = 0;

Producer(){	Consumer(){
int item;	**int** item;

while(TRUE){
produce_item(&item);
down(&empty);
down(&mutex);
enter_item(item);
up(&mutex);
up(&full); }}
	while(TRUE){
	down(&full);
	down(&mutex);
	remove_item(&item);
	up(&mutex);
	up(&empty);
	consume_item(item);}}

Chương 7: Đồng bộ hóa tiến trình

Giải pháp với Semaphore

- Kinh nghiệm gì thu được khi sử dụng semaphore?

Giải pháp với Monitor

- Định nghĩa một monitor ***ProducerConsumer*** với hai thủ tục ***enter*** và ***remove*** thao tác trên bộ đệm. Xử lý của các thủ tục này phụ thuộc vào các biến điều kiện ***full*** và ***empty***.

Chương 7: Đồng bộ hóa tiến trình

Giải pháp với Monitor

- **monitor** ProducerConsumer

  condition full, empty;

  int count;

  procedure enter(); {

    if (count == N) wait(full);
    enter_item(item);

    count ++;

    if (count == 1)

    signal(empty);}

Chương 7: Đồng bộ hóa tiến trình

Giải pháp với Monitor

```
procedure remove(); {
        if (count == 0)
        wait(empty)
        remove_item(&item);
        count --;
        if (count == N-1) signal(full);}
count = 0;
end monitor;
```

Chương 7: Đồng bộ hóa tiến trình

Giải pháp với Monitor

Producer(); {
while(TRUE)
{
produce_item(&item);
ProducerConsumer.enter;
} }
	Consumer();{
	while(TRUE)
	{
	ProducerConsumer.remove;
	consume_item(item);
	} }

Thảo luận

Bài tập 1

- Xét giải pháp phần mềm do Dekker đề nghị để tổ chức truy xuất độc quyền cho 2 tiến trình. Hai tiến trình P0 và P1 chia sẻ các biến sau:
  - Var flag : array [0..1] of Boolean; (khởi động là false)
  - Turn : 0..1;
- Cấu trúc một tiến trình Pi (i=0 hay 1, và j là tiến trình còn lại như sau:

```
repeat
flag[i] := true;
while flag[j] do
if turn = j then
        begin
                                  flag[i]:= false;
                                  while turn = j do ;
                                  flag[i]:= true;
        end;
     critical_section();
turn:= j;
flag[i]:= false;
non_critical_section();
until false;
```

Giải pháp này có thỏa 3 yêu cầu trong việc giải quyết tranh chấp không?

Bài tập 2

- Xét giải pháp đồng bộ hóa sau:

```
while (TRUE) {
int j = 1-i;
flag[i]= TRUE;
turn = i;
while (turn == j && flag[j]==TRUE);
critical-section ();
flag[i] = FALSE;
   Noncritical-section ();
}
```

Giải pháp này có thỏa yêu cầu độc quyền truy xuất không?

Bài tập 3

- Giả sử một máy tính không có chỉ thị TSL, nhưng có chỉ thị Swap có khả năng hoán đổi nội dung của hai từ nhớ chỉ bằng một thao tác không thể phân chia:

```
procedure Swap() var a,b: boolean);
var temp : boolean;
begin
   temp := a;
   a:= b;
   b:= temp;
end;
```

Sử dụng chỉ thị này có thể tổ chức truy xuất độc quyền không? Nếu có, xây dựng cấu trúc chương trình tương ứng.

Bài tập 4

- Xets hai tiến trình sau:

```
process A  {while (TRUE)   na = na +1;       }
process B  {while (TRUE)   nb = nb +1;       }
```

a. Đồng bộ hóa xử lý của 2 tiến trình trên, sử dụng 2 semaphore tổng quát, sao cho tại bất kỳ thời điểm nào cũng có nb <= na <= nb +10

b. Nếu giảm điều kiện chỉ có là na <= nb +10, giải pháp của bạn sẽ được sửa chữa như thế nào?

c. Giải pháp của bạn có còn đúng nếu có nhiều tiến trình loại A và B cùng thực hiện?

Bài tập 5

- Một biến X được chia sẻ bởi 2 tiến trình cùng thực hiện đoạn code sau :

```
do
   X = X +1;
   if ( X == 20) X = 0;
while ( TRUE );
```

- Bắt đầu với giá trị X = 0, chứng tỏ rằng giá trị X có thể vượt quá 20. Cần sửa chữa đoạn chương trình trên như thế nào để đảm bảo X không vượt quá 20?

Bài tập 6

- Xét 2 tiến trình xử lý đoạn chương trình sau:

  process P1 { A1 ; A2 }

  process P2 { B1 ; B2 }

Đồng bộ hóa hoạt động của 2 tiến trình này sao cho cả A1 và B1 đều hoàn tất trước khi A2 và B2 bắt đầu

- Tổng quát hóa câu hỏi 6 cho các tiến trình có đoạn chương trình sau:

  process P1 { for (i = 1; i <= 100; i ++) A_i }

  process P2 { for (j = 1; j <= 100; j ++) B_j }

Đồng bộ hóa hoạt động của 2 tiến trình này sao cho với *k* bất kỳ (2<=k<=100), A_k chỉ có thể bắt đầu khi $B_{(k-1)}$ đã kết thúc và B_k chỉ có thể bắt đầu khi $A_{(k-1)}$ đã kết thúc.