Liên lạc giữa các tiến trình
Đồng bộ hóa

Môn học: Hệ điều hành

fit@hcmus
KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Nội dung

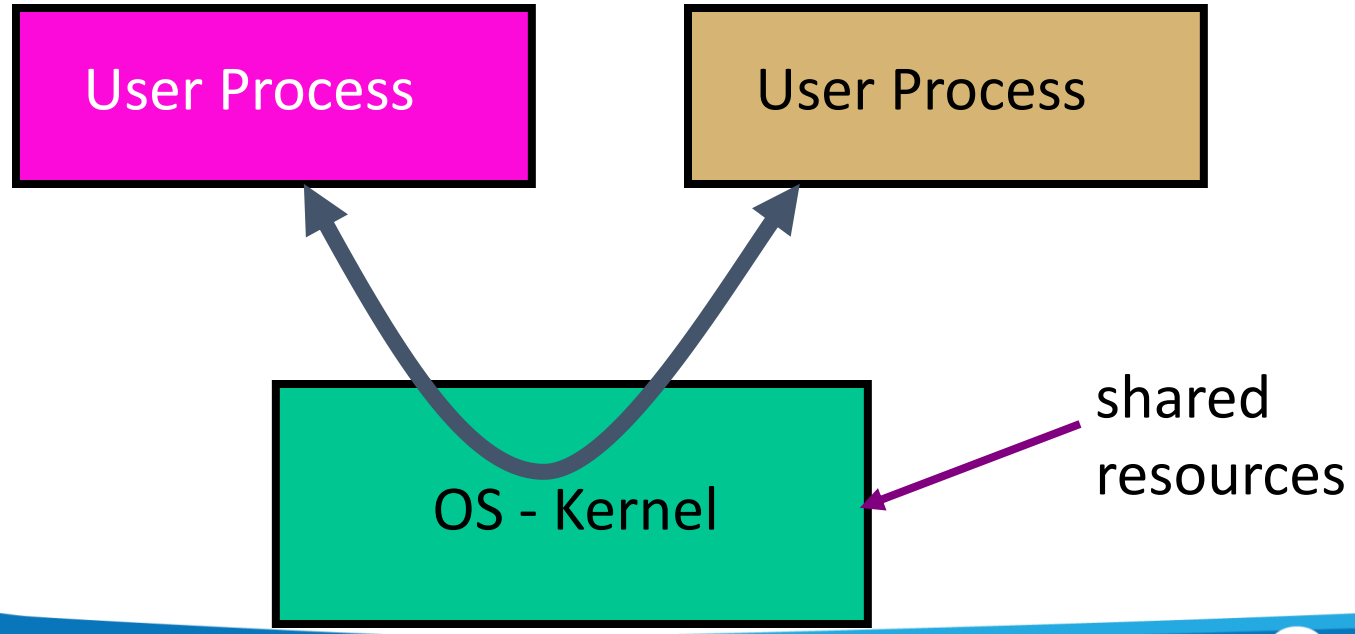
- Liên lạc giữa các tiến trình:
  - Sử dụng tài nguyên dung chung.
  - Trao đổi thông điệp.
- Xử lý đồng hành và các vấn đề:
  - Vấn đề tranh đoạt điều khiển (Race Condition) và độc quyền truy xuất (Mutual Exclusion).
  - Vấn đề phối hợp xử lý.
- Các giải pháp đồng bộ hoá:
  - Busy waiting.
  - Sleep & Wakeup.
- Các bài toán đồng bộ hoá kinh điển:
  - Producer – Consumer.
  - Readers – Writers.
  - Dinning Philosophers.

Liên lạc giữa các tiến trình (IPC – Inter-process Communication)

- Cơ chế trao đổi dữ liệu giữa các tiến trình:
  - Chia sẻ tài nguyên dung chung:
    - Signal.
    - Pipeline.
    - Shared Memory.
  - Trao đổi thông điệp:
    - Message Passing.
    - Socket.

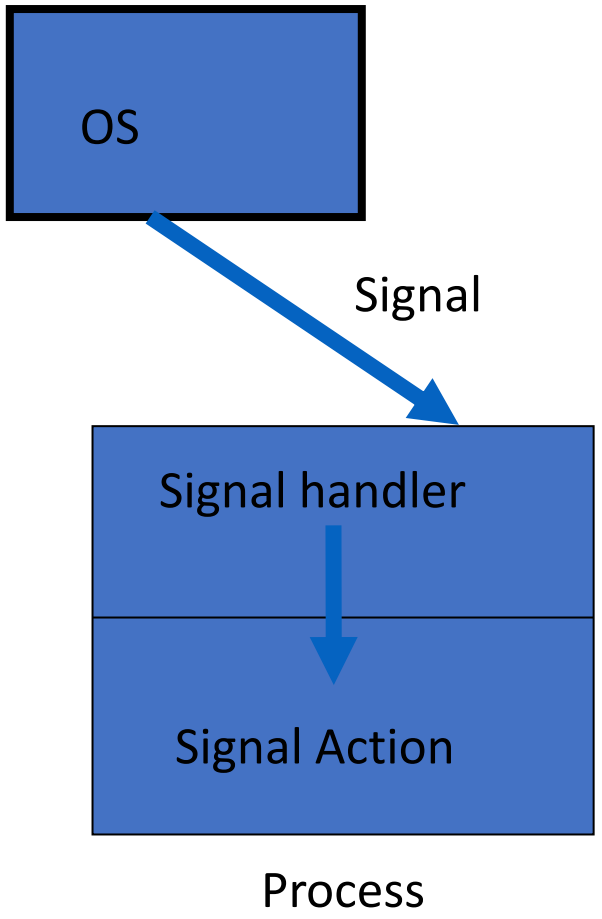
IPC theo nguyên tắc chia sẻ tài nguyên chung

- Các tiến trình chia sẻ:
  - Memory.
  - File System Space.
  - Communication Facilities, Common communication protocol.



Signal

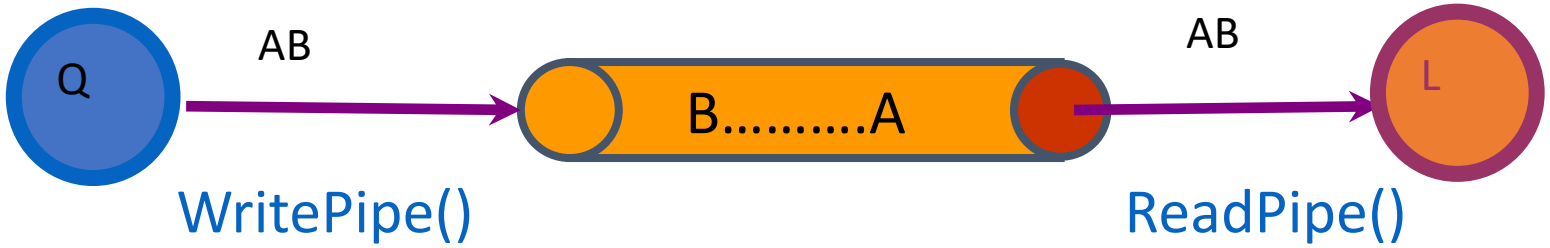
- Signal:
  - Meaning.
  - Handler.
- Định nghĩa trước khi thực hiện liên lạc:
  - SIGINT, SIGSTOP.
  - SIGUSR1, SIGUSR2.
- Hỗ trợ liên lạc:
  - Liên lạc không đồng bộ.
  - Kernel với user process:
    - Process Error.
    - Timer.
    - Child process kết thúc, …
  - User process với nhau:
    - Terminate process.
    - Suspend, Resume, …
- Không cho phép trao đổi dữ liệu.



Một số signal thường gặp

Name	Value	Function
SIGHUP	1	Terminal hangup
SIGINT	2	Interrupt by user: generated by < CTRL C >
SIGQUIT	3	Quit by user: generated by < CTRL \ >
SIGFPE	8	Floating point error such as divide by zero
SIGKILL	9	Kill the process
SIGUSR1	10	User defined signal 1
SIGSEGV	11	Segment violation: process has tried to access memory not assigned to it
SIGUSR2	12	User defined signal 2
SIGALRM	14	Timer set with alarm() function has timed out
SIGTERM	15	Termination request
SIGCHLD	17	Child process termination signal
SIGCONT	18	Continue after a SIGSTOP or SIGSTP signal
SIGSTOP	19	Stop the process
SIGTSTP	20	Terminal stop: generated by < CTRL Z >
SIGWINCH	28	Change of window size

Pipes



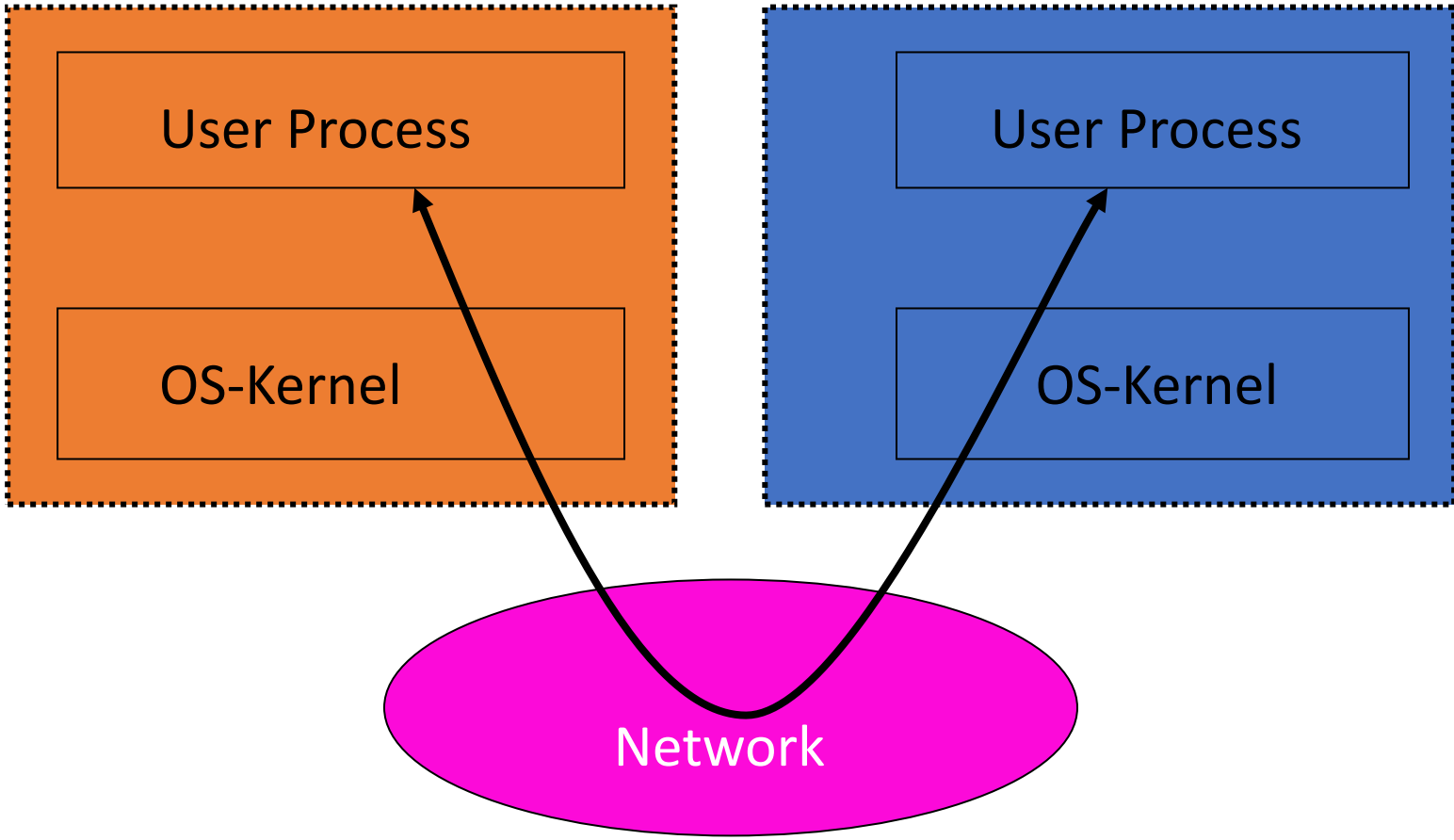
- Pipe
  - Kernel buffer (File) có kích thước giới hạn (4KB, 8KB).
  - HĐH cung cấp hàm WritePipe & ReadPipe.
    - Writepipe khi Pipe đầy ?
    - ReadPipe khi Pipe rỗng ?
    - Phải xét đến khả năng đồng bộ.
- Hỗ trợ liên lạc (UNIX original):
  - Giữa 2 tiến trình Cha – Con.
  - Một chiều.
  - Không cấu trúc (byte transfer).
- Named Pipe : Unix , Windows NT, …
  - Truyền dữ liệu có cấu trúc.
  - Liên lạc 2 chiều.

Shared Memory

- Shared Memory:
  - Là một phần không gian nhớ không thuộc sở hữu của tiến trình nào.
  - Được HĐH tạo ra.
  - Các tiến trình có thể ánh xạ địa chỉ vào không gian chia sẻ này để truy xuất dữ liệu (như đối với không gian địa chỉ nội bộ).
    - shmget(), shmat(), shmctl(), …
- Không giới hạn số lượng tiến trình, chiều trao đổi và thứ tự truy cập.
  - Mâu thuẫn truy xuất -> Nhu cầu đồng bộ.

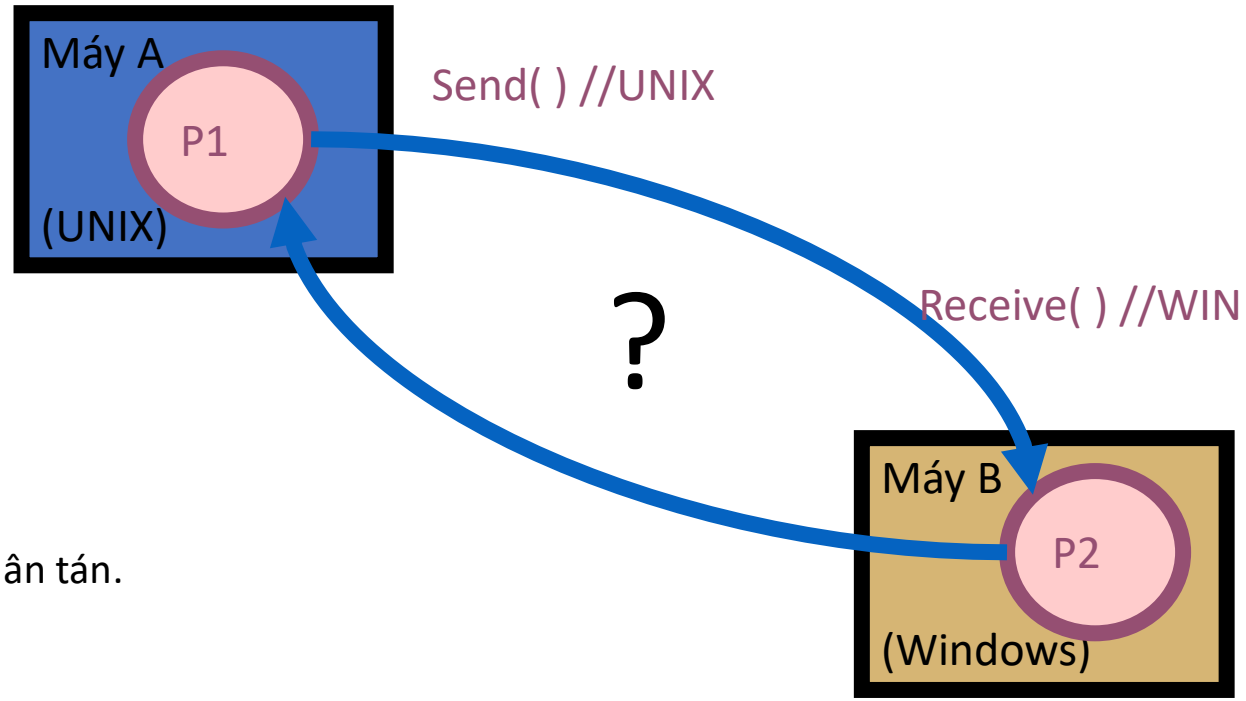
IPC theo nguyên tắc trao đổi thông điệp

- Không có bộ nhớ chung.
- Cần có đường kết nối giữa các máy tính.



Message Passing

- Message:
  - Dữ liệu có cấu trúc.
  - Cấu trúc và thông dịch msg được thỏa thuận giữa 2 tiến trình liên lạc.
- HĐH cung cấp 2 primitive chính:
  - send(destination, message).
  - receive(source, message).
- Các vấn đề quan tâm :
  - Direct or indirect addressing.
  - Blocking or non-blocking communication.
  - Reliable or unreliable communication.
  - Buffered or un-buffered communication.
- Là cơ chế IPC tổng quát.
  - Hỗ trợ liên lạc giữa các tiến trình trên cùng máy.
  - Hỗ trợ liện lạc giữa các tiến trình trong hệ thống phân tán.
- Liên lạc giữa các hệ thống không đồng nhất ?



Socket

- Kiến thức trong môn Mạng máy tính.

Các vấn đề xử lý đồng hành

- Tranh chấp (Race Condition)
  - Nhiều tiến trình truy xuất đồng thời một tài nguyên mang bản chất không chia sẻ được.
    → Xảy ra vấn đề tranh đoạt điều khiển (Race Condition).
  - Kết quả ?
    - Khó biết , thường là ...sai.
    - Luôn luôn nguy hiểm ?
      - ...Không, nhưng đủ để cân nhắc kỹ càng.
- Phối hợp
  - Các tiến trình không biết tương quan xử lý của nhau để điều chỉnh hoạt động nhịp nhàng.
    - Cần phối hợp xử lý (Rendez-vous).
  - Kết quả : khó biết, không bảo đảm ăn khớp.

Tranh chấp điều khiển – Ví dụ 1 (1/3)

- Đếm số người vào Altavista : dùng 2 tiến trình cập nhật biến đếm ***hits***.
  → P1 và P2 chia sẻ biến ***hits***.

P1

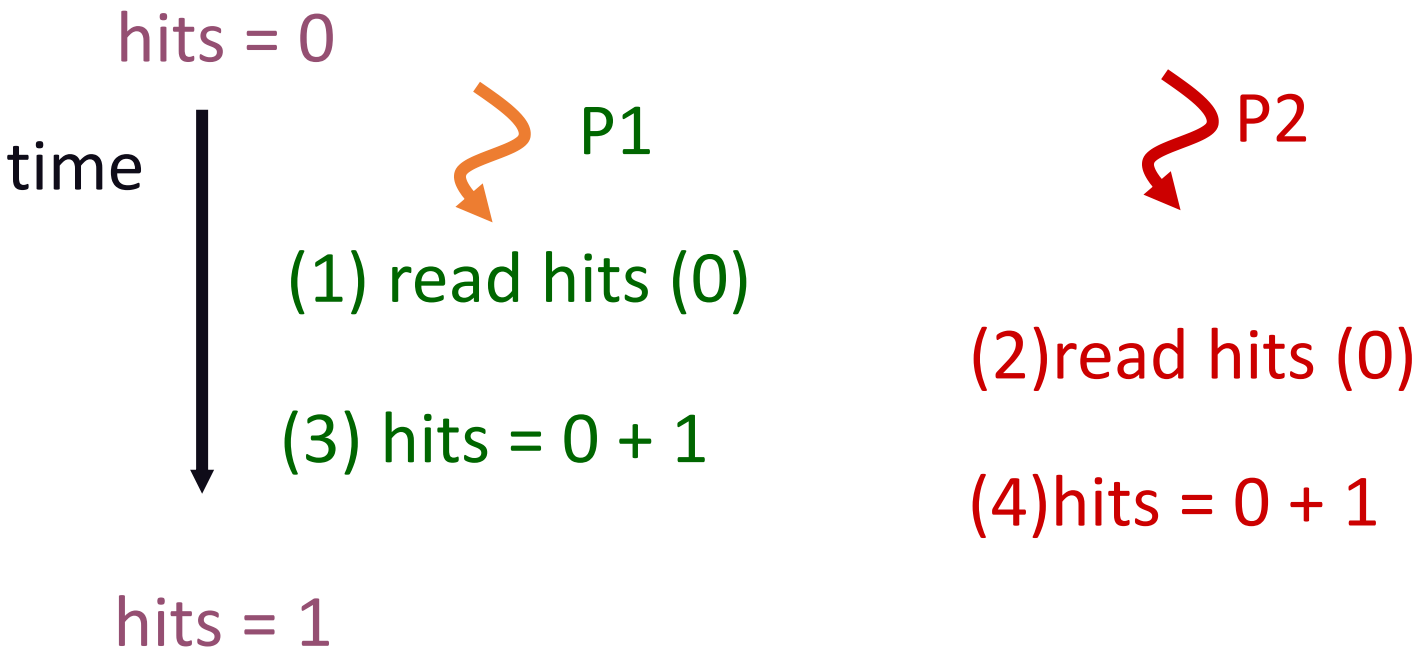
```
read hits;
hits = hits +1;
```

P2

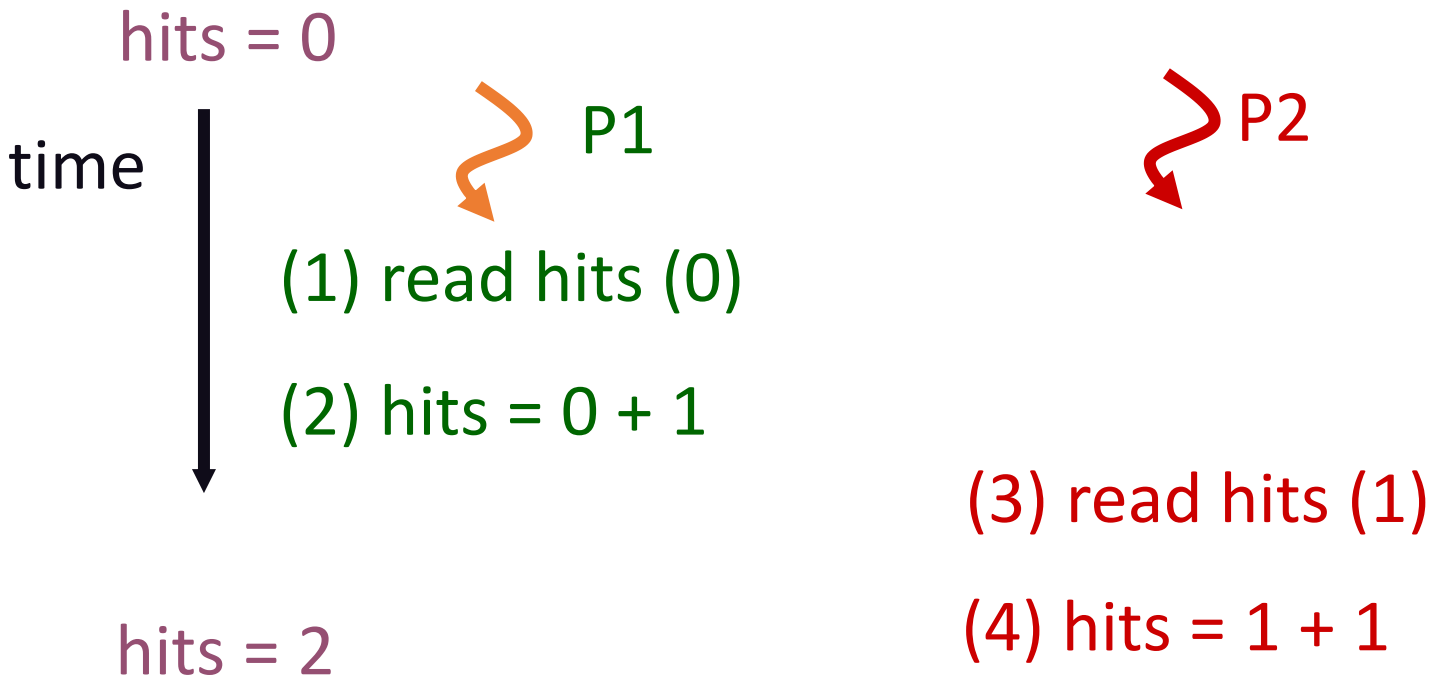
```
read hits;
hits = hits + 1;
```

- Kết quả cuối cùng là bao nhiêu ?

Tranh chấp điều khiển – Ví dụ 1 (2/3)



Tranh chấp điều khiển – Ví dụ 1 (3/3)



Tranh chấp điều khiển – Ví dụ 2

```
i=0;
```

Thread a:

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

Thread b:

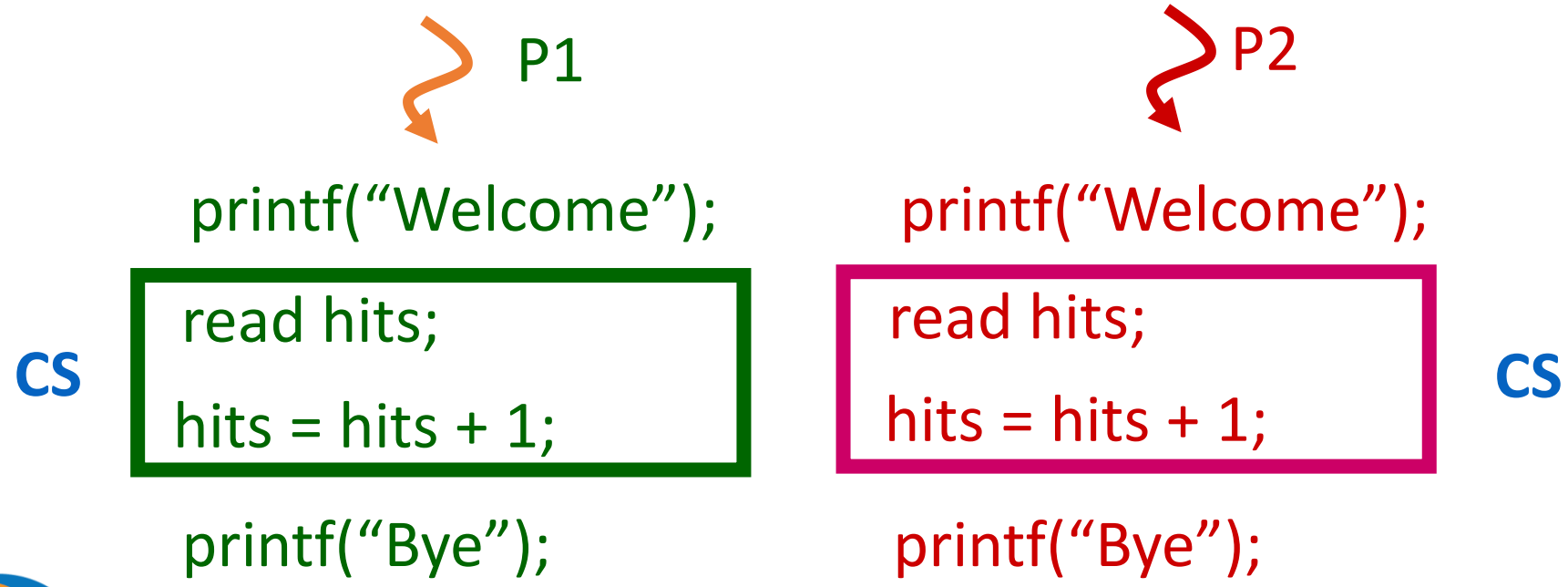
```
while(i > -10)
    i = i - 1;
print “B won!”;
```

Tranh chấp điều khiển – Nhận xét

- Kết quả thực hiện tiến trình phụ thuộc vào kết quả điều phối.
  - Cùng input, không chắc cùng output.
  - Khó debug lỗi sai trong xử lý đồng hành.
- Xử lý
  - Làm lơ .
    - Dễ , nhưng có phải là giải pháp.
  - Không chia sẻ tài nguyên chung : dùng 2 biến hits1,hits2; xây cầu 2 lane...
    - Nên dùng khi có thể, nhưng không bao giờ có thể đảm bảo đủ tài nguyên, và cũng không là giải pháp đúng cho mọi trường hợp.
  - Giải pháp tổng quát : có hay không ?
    - Lý do xảy ra Race condition ? Bad interleavings : một tiến trình "xen vào" quá trình truy xuất tài nguyên của một tiến trình khác.
    - Giải pháp : bảo đảm tính atomicity cho phép tiến trình hoàn tất trọn vẹn quá trình truy xuất tài nguyên chung trước khi có tiến trình khác can thiệp.

Tranh chấp điều khiển – Hướng giải quyết (1/2)

- Xác định miền găng (Critical Section/Region) là đoạn chương trình có khả năng gây ra hiện tượng race condition.



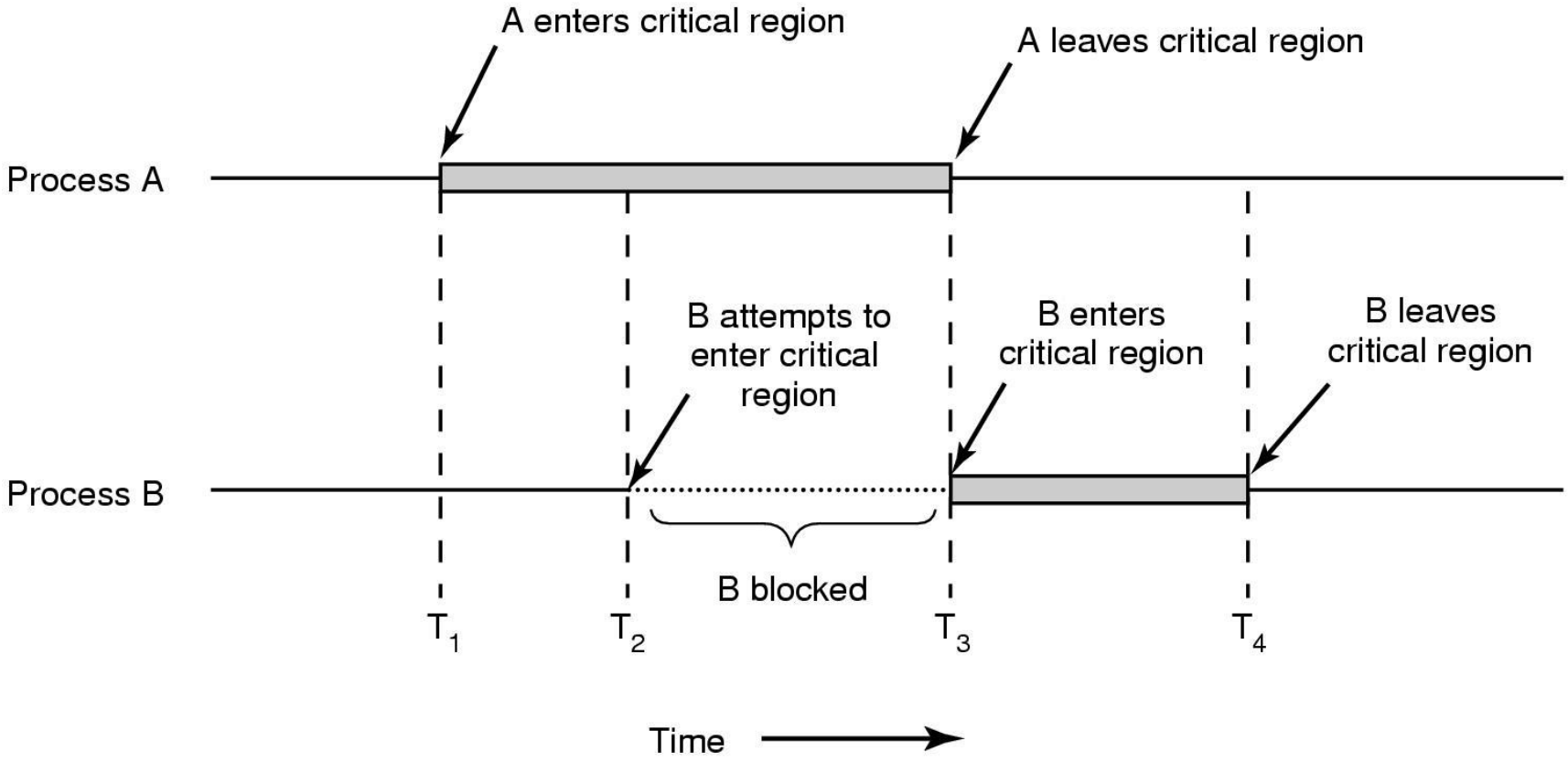
Tranh chấp điều khiển – Hướng giải quyết (2/2)

- Bảo đảm tính "độc quyền truy xuất" (Mutual Exclusion) cho miền găng.

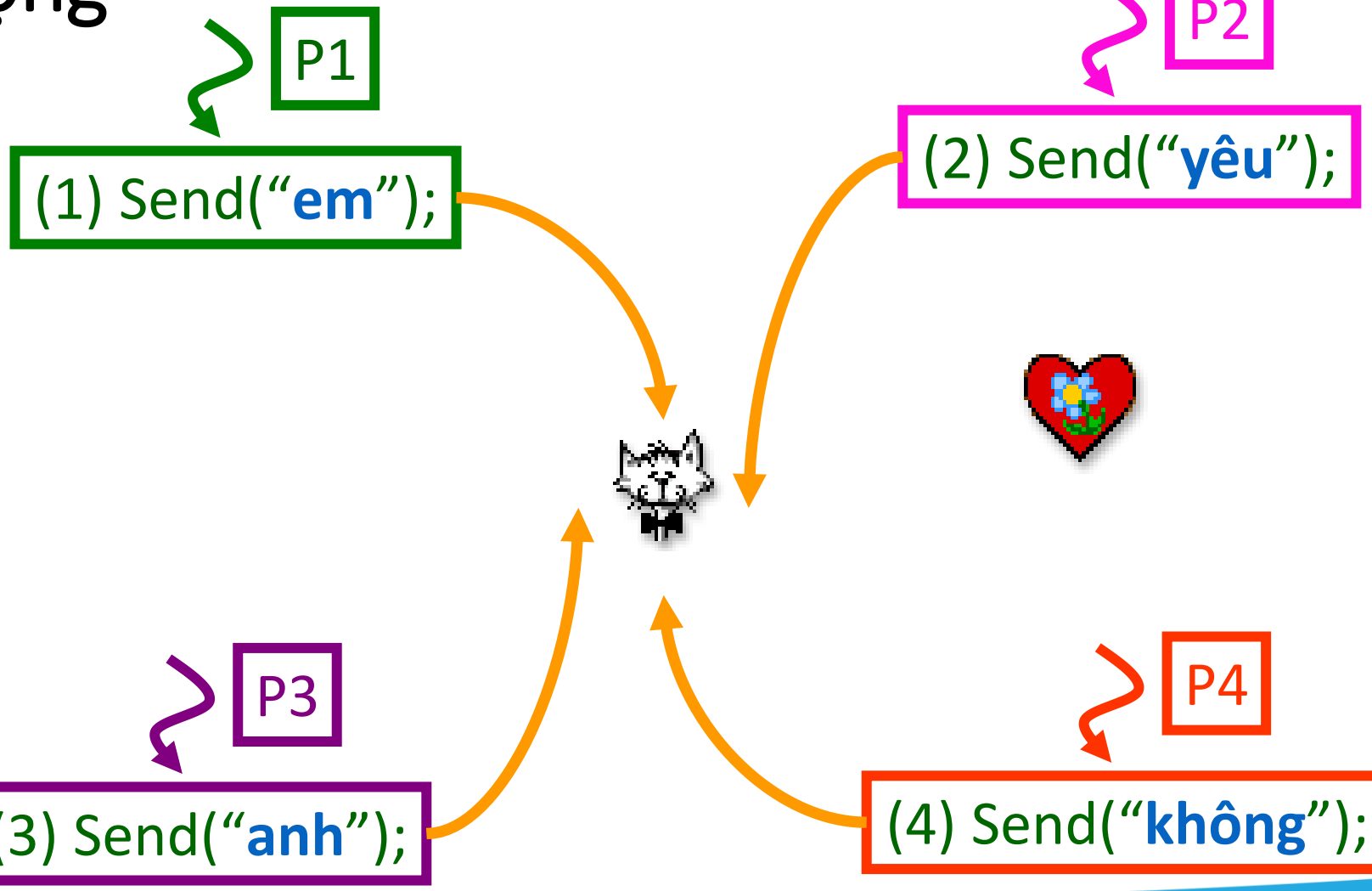
Kiểm tra và dành quyền vào CS

CS;

Từ bỏ quyền sử dụng CS

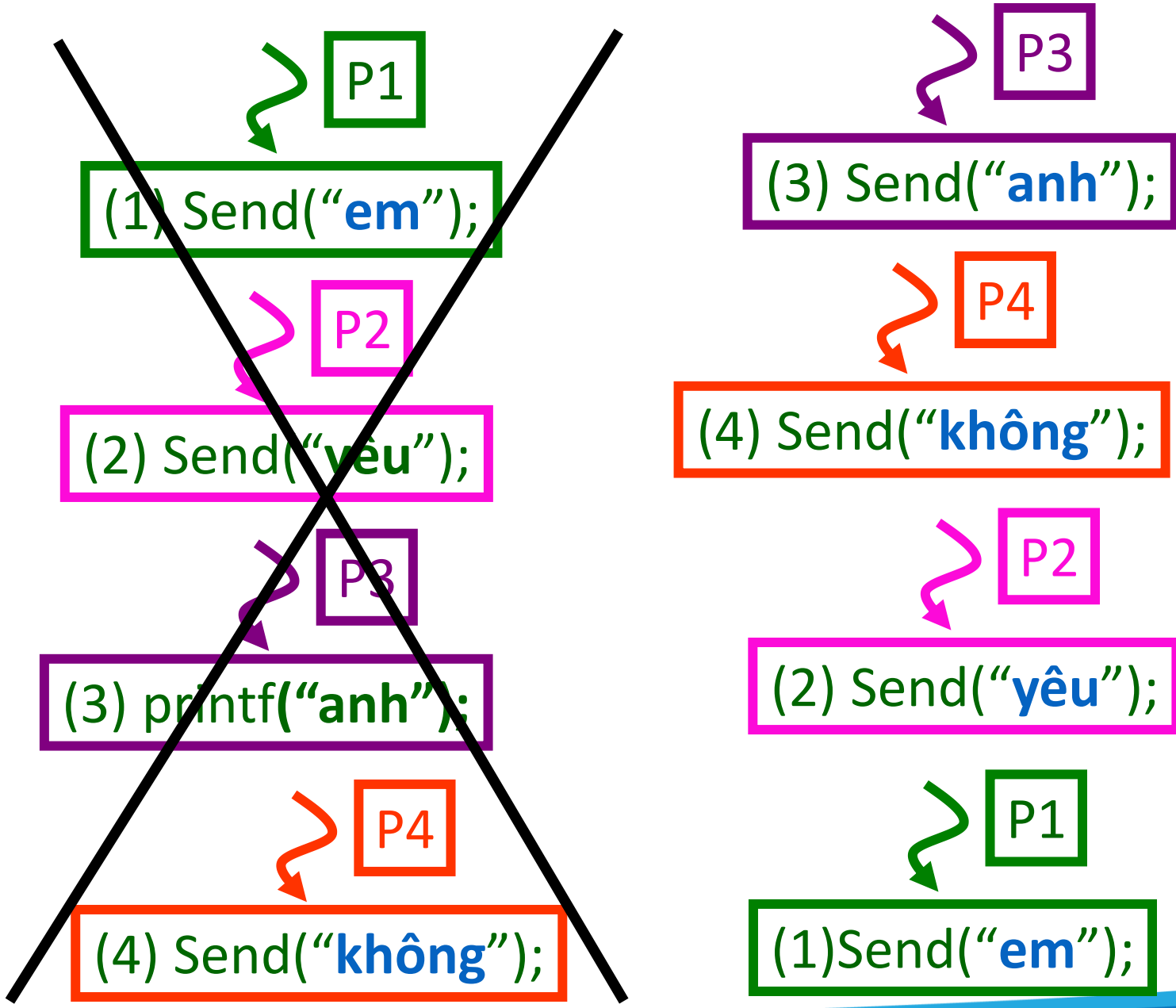


Phối hợp hoạt động – Tình huống



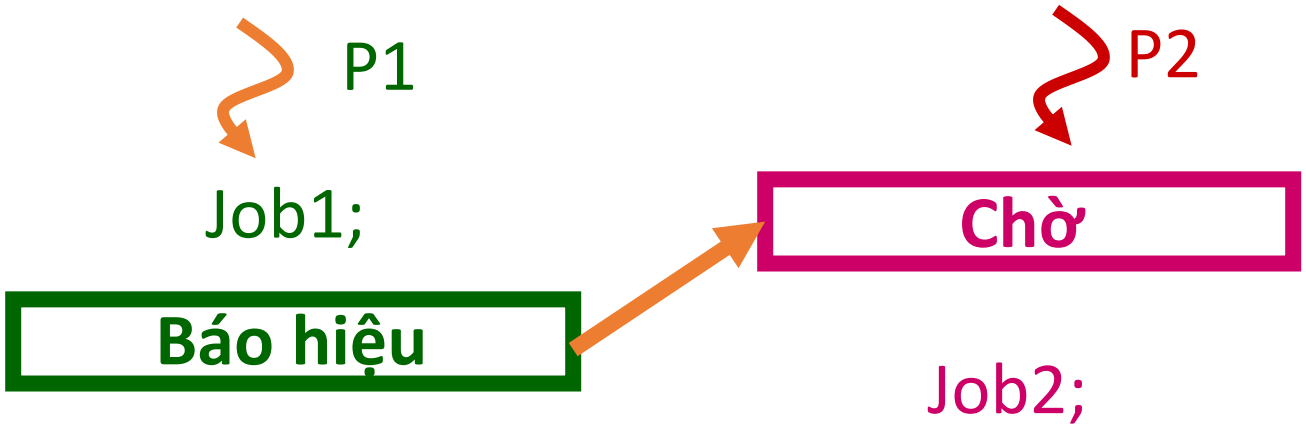
fit@hcmus

Phối hợp hoạt động – Vấn đề



Phối hợp hoạt động – Nhận xét và hướng tiếp cận

- Làm thế nào bảo đảm trình tự thực hiện Job1 - Job2 ?
  - P1 và P2 thực hiện “hẹn hò” (Rendez-vous) với nhau.
- Hỗ trợ Rendez-vous : Bảo đảm các tiến trình phối hợp với nhau theo 1 trình tự xử lý định trước.



Các giải pháp đồng bộ hóa

- Nhóm giải pháp Busy Waiting:
  - Phần mềm:
    - Sử dụng các biến cờ hiệu.
    - Sử dụng việc kiểm tra luân phiên.
    - Giải pháp của Peterson.
  - Phần cứng:
    - Cấm ngắt.
    - Chỉ thị TSL.
- Nhóm giải pháp Sleep & Wakeup:
  - Semaphore.
  - Mutex.
  - Monitor.
  - Message.

Yêu cầu giải pháp đồng bộ hóa

- Một phương pháp giải quyết tốt bài toán đồng bộ hoá cần thoả mản 3 điều kiện sau:
  - Mutual Exclusion : Không có hai tiến trình cùng ở trong miền găng cùng lúc.
  - Progess : Một tiến trình tạm dừng bên ngoài miền găng không được ngăn cản các tiến trình khác vào miền găng.
  - Bounded Waiting : Không có tiến trình nào phải chờ vô hạn để được vào miền găng.
- Ngoài ra, 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.

Các giải pháp "Busy waiting"

- Tiếp tục tiêu thụ CPU trong khi chờ đợi vào miền găng.
- Không đòi hỏi sự trợ giúp của HĐH.

While (chưa có quyền vào CS) do_nothing();

CS;

Từ bỏ quyền sử dụng CS

Giải pháp phần mềm – Biến cờ hiệu

int lock = 0;

T0

NonCS;

```
while (lock == 1); // wait
lock = 1;
```

CS;

```
lock = 0;
```

NonCS;

T1

NonCS;

```
while (lock == 1); // wait
lock = 1;
```

CS;

```
lock = 0;
```

NonCS;

Giải pháp phần mềm – Biến cờ hiệu – Tình huống

int lock = 0;

T0

NonCS;

```
while (lock == 1); // wait
lock = 1;
```

CS;

```
lock = 0;
```

NonCS;

T1

NonCS;

```
while (lock == 1); // wait
lock = 1;
```

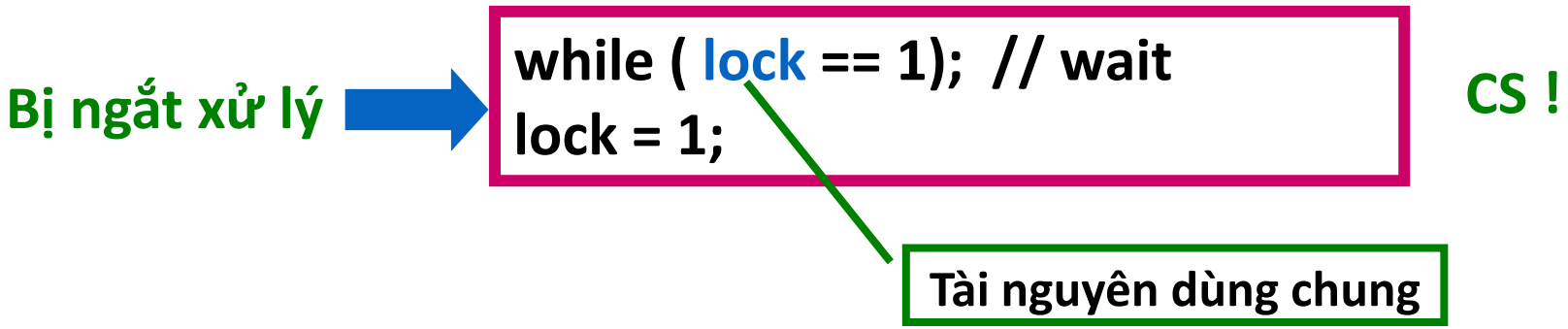
CS;

```
lock = 0;
```

NonCS;

Giải pháp phần mềm – Biến cờ hiệu – Nhận xét

- Có thể mở rộng cho N tiến trình.
- Không bảo đảm Mutual Exclusion.
  - Nguyên nhân ?



  - Bản thân đoạn code kiểm tra và dành quyền cũng là CS !

Giải pháp phần mềm – Kiểm tra luân phiên

int turn = 1;

T0

NonCS;

```
while (turn !=0); // wait
```

CS;

```
turn = 1;
```

NonCS;

T1

NonCS;

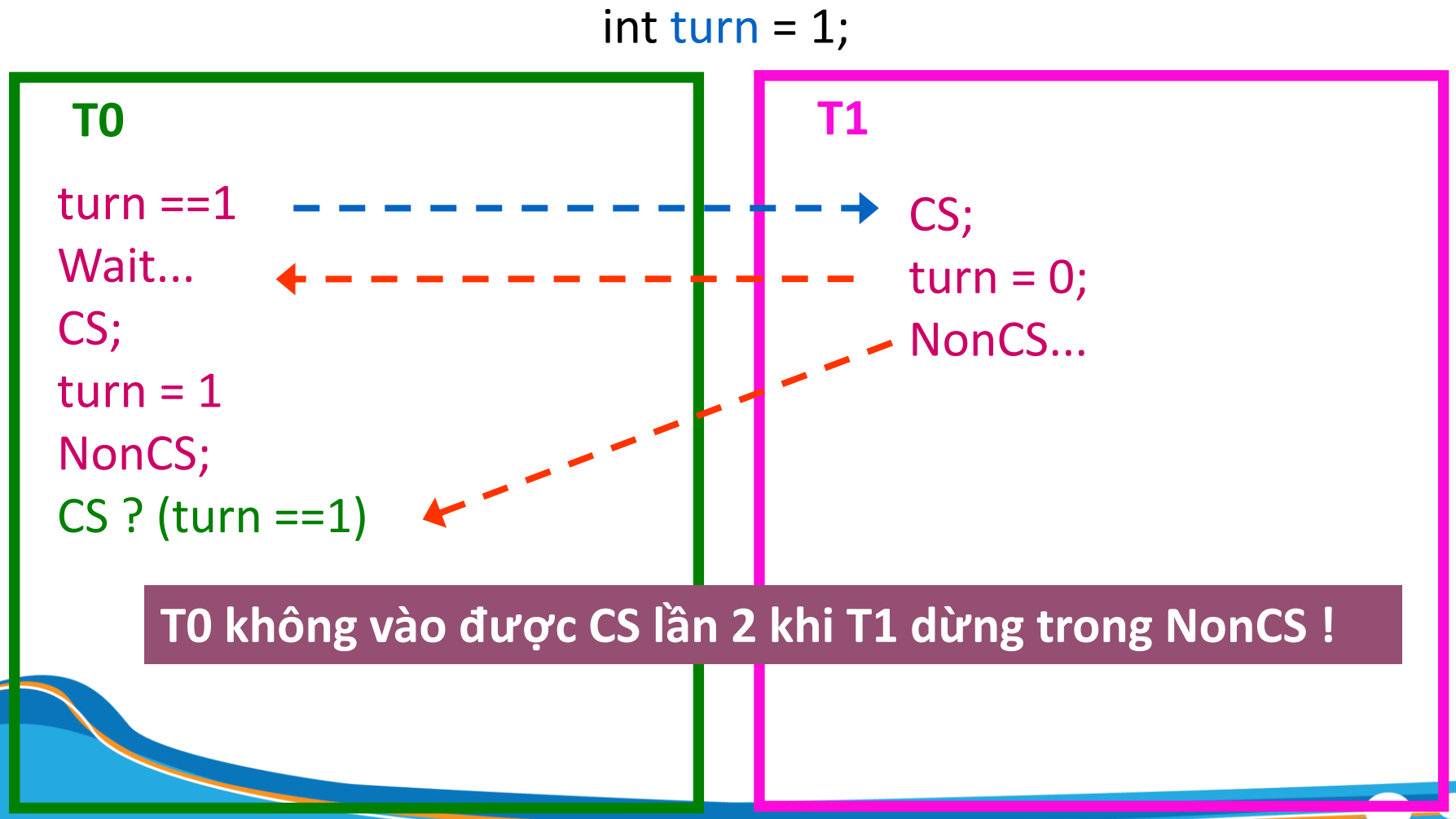
```
while (turn != 1); // wait
```

CS;

```
turn = 0;
```

NonCS;

Giải pháp phần mềm – Kiểm tra luân phiên – Tình huống



Giải pháp phần mềm – Kiểm tra luân phiên – Nhận xét

- Chỉ dành cho 2 tiến trình.
- Bảo đảm Mutual Exclusion.
  - Chỉ có 1 biến turn, tại 1 thời điểm chỉ cho 1 tiến trình turn vào CS.
- Không bảo đảm Progress
  - Nguyên nhân ?
    - “Mờ của” cho người = “Đóng cửa” chính mình !

Giải pháp phần mềm – Peterson

- Kết hợp ý tưởng của giải pháp biến cờ hiệu và kiểm tra luân phiên, các tiểu trình chia sẻ:
  - int turn; *//đến phiên ai*
  - int interest[2] = FALSE; *//interest[i] = T : Pi muốn vào CS*

Ti

NonCS;

```
j = 1 – i;
interest[i] = TRUE;
turn = j;
while (turn==j && interest[j]==TRUE);
```

CS;

```
interest[i] = FALSE;
```

NonCS;

Tj

NonCS;

```
i = 1 – j;
interest[j] = TRUE;
turn = i;
while (turn==i && interest[i]==TRUE);
```

CS;

```
interest[j] = FALSE;
```

NonCS;

Giải pháp phần mềm – Peterson – Nhận xét

- Là giải pháp phần mềm đáp ứng được cả 3 điều kiện.
  - Mutual Exclusion:
    - Pi chỉ có thể vào CS khi: *interest[j] == F* hay *turn == 1.*
    - Nếu cả 2 muốn về thì do *turn* chỉ có thể nhận giá trị 0 hay 1 nên chỉ có 1 tiến trình vào CS.
  - Progress:
    - Sử dụng 2 biến *interest[i]* riêng biệt => trạng thái đối phương không khoá mình được.
  - Bounded Wait:
    - *interest[i]* và *turn* đều có thay đổi giá trị.
- Không thể mở rộng cho N tiến trình

Nhận xét chung nhóm giải pháp phần mềm

- Không cần sự hỗ trợ của hệ thống.
- Dễ...sai, Khó mở rộng.
- Giải pháp biến cờ hiệu nếu có thể được hỗ trợ atomicity thì sẽ tốt ...
  - Nhờ đến phần cứng ?

Giải pháp phần cứng – Cấm ngắt

- Disable Interrupt : Cấm mọi ngắt, kể cả ngắt đồng hồ.
- Enable Interrupt : Cho phép ngắt.
- Thiếu thận trọng.
  - Nếu tiến trình bị khoá trong CS ?
    - System Halt.
  - Cho phép tiến trình sử dụng một lệnh đặc quyền.
    - Quá ...liều !
- Máy có nhiều CPUs hoặc multi-core ?
  - Không bảo đảm được Mutual Exclusion.

NonCS;

Disable Interrupt;

CS;

Enable Interrupt;

NonCS;

Giải pháp phần cứng – Chỉ thị TSL

- CPU hỗ trợ primitive: Test and Set Lock.
  - **TSL (boolean &target)**
    {
    TSL = target;
    target = TRUE;
    }
  - Trả về giá trị hiện hành của 1 biến, và đặt lại giá trị True cho biến.
  - Thực hiện một cách không thể phân chia (atomic).

int lock = 0

Ti

NonCS;

while (TSL(lock)); // wait

CS;

lock = 0;

NonCS;

Nhận xét chung nhóm giải pháp phần cứng

- Cần được sự hỗ trợ của cơ chế phần cứng.
  - Không dễ, nhất là trên các máy có nhiều bộ xử lý.
- Dễ mở rộng cho N tiến trình.

Nhận xét chung nhóm giải pháp "Busy-Waiting"

- Sử dụng CPU không hiệu quả.
  - Liên tục kiểm tra điều kiện khi chờ vào CS.
- Khắc phục.
  - Khóa (block) các tiến trình chưa đủ điều kiện vào CS, nhường CPU cho tiến trình khác.
    - Phải nhờ đến Scheduler.

Giải pháp "Sleep & Wake up"

- Từ bỏ CPU khi chưa được vào CS.
- Khi CS trống, sẽ được đánh thức để vào CS.
- Cần được HĐH hỗ trợ.
- Vì phải thay đổi trạng thái tiến trình.
- Hệ Điều hành hỗ trợ 2 primitive:
  - Sleep() : Tiến trình gọi sẽ nhận trạng thái Blocked.
  - WakeUp(P): Tiến trình P nhận trạng thái Ready.
- Áp dụng:
  - Sau khi kiểm tra điều kiện sẽ vào CS hay gọi Sleep() tùy vào kết quả kiểm tra.
  - Tiến trình vừa sử dụng xong CS sẽ đánh thức các tiến trình bị Blocked trước đó.

if (chưa có quyền) Sleep() ;

CS;

Wakeup(somebody);

Giải pháp "Sleep & Wake up" – Áp dụng

- int busy; // busy ==0 : CS trống
- int blocked; // đếm số tiến trình bị Blocked chờ vào CS

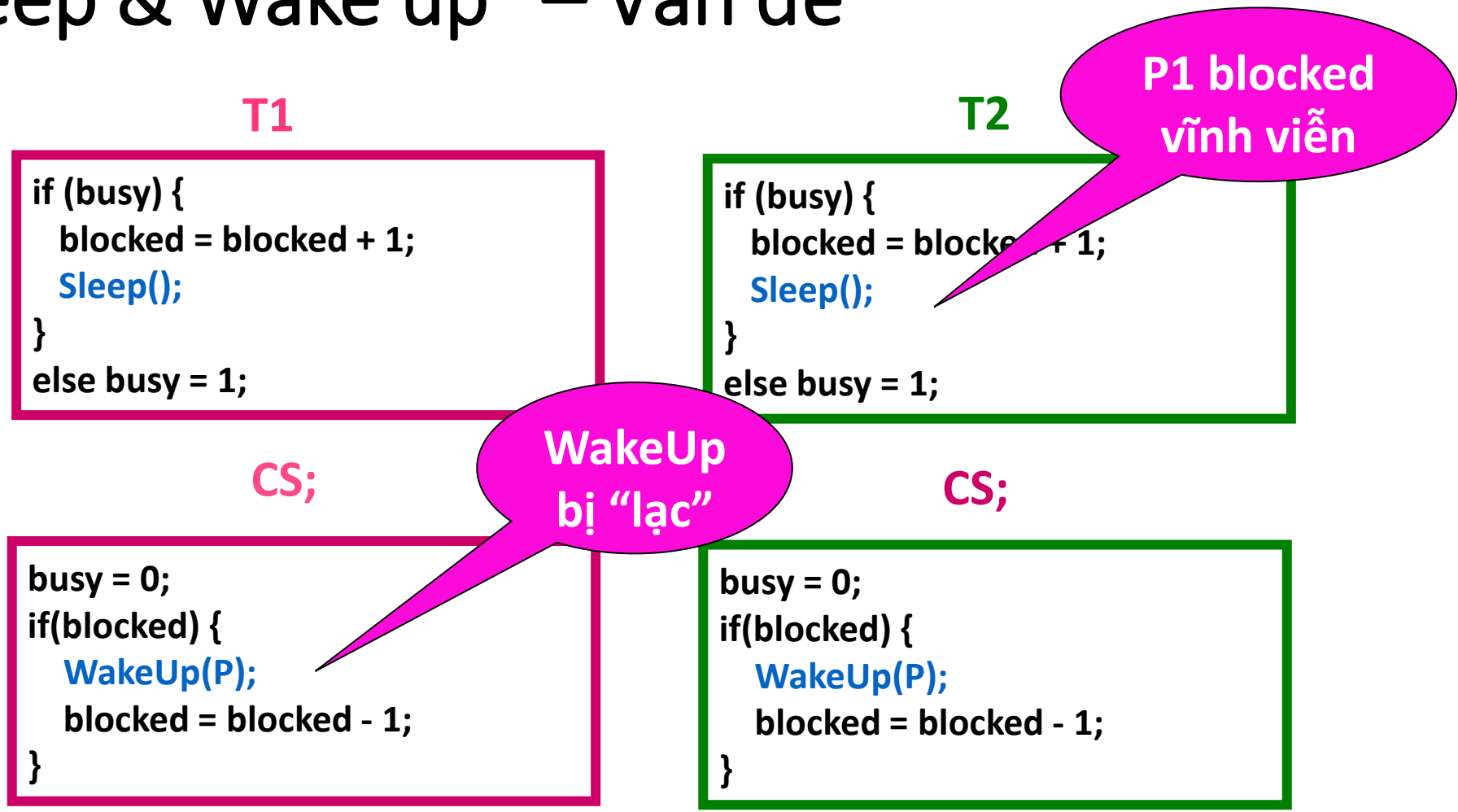
```
if (busy) {
                    blocked = blocked + 1;
                    Sleep();
            }
else busy = 1;
```

CS;

```
busy = 0;
            if(blocked) {    WakeUp(P);
                                    blocked = blocked - 1;
                             }
```

Giải pháp "Sleep & Wake up" – Vấn đề

- Nguyên nhân :
  - Việc kiểm tra điều kiện và động tác từ bỏ CPU có thể bị ngắt quãng.
  - Bản thân các biến cờ hiệu không được bảo vệ.



Giải pháp "Sleep & Wake up" – Nhận xét

- HĐH cần hỗ trợ các cơ chế cao hơn:
  - Dựa trên Sleep & WakeUp
  - Kết hợp các yếu tố kiểm tra
  - Thi hành không thể phân chia
- Nhóm giải pháp Sleep & Wakeup
  - Semaphore
  - Monitor
  - Message

Giải pháp Semapore

- Được đề nghị bởi Dijkstra năm 1965.
- Là cơ chế do HĐH cung cấp.
  - Semaphore s;
  - Có 1 giá trị.
  - Chỉ được thao tác bởi 2 primitives:
    - Down(s).
    - Up(s).
  - Các primitive Down và Up được thực hiện không thể phân chia (cấm ngắt).

Semaphore s; // s >=0

Giải pháp Semapore – Cài đặt

- Các tiến trình "yêu cầu" semaphore: gọi Down(s).
  - Nếu không hoàn tất được Down(s): chưa được cấp resource.
    - Blocked, được đưa vào S.L

```
typedef struct
{
        int value;
        struct process* L;
} Semaphore      ;
```

Giá trị bên trong của semaphore

Danh sách các tiến trình đang bị block đợi semaphore nhận giá trị dương

```
Down (S)
{
        S.value --;
        if (S.value < 0)
        {
                Add(P,S.L);
                Sleep();
        }
}
```

```
Up(S)
{
        S.value ++;
        if (S.value ≤ 0)
        {
                Remove(P,S.L);
                Wakeup(P);
        }
}
```

Giải pháp Semapore – Sử dụng

- Tổ chức "độc quyền truy xuất".

Semaphore s = ? 1

```
Pi
    Down (s);
    CS;
    Up(s);
```

- Tổ chức "phối hợp xử lý".

Semaphore s = ? 0

```
P1 :
    Job1;
    Up(s);
```

```
P2:
    Down (s);
    Job2;
```


44

Giải pháp Semaphore – Nhận xét

- Là một cơ chế tốt để thực hiện đồng bộ.
  - Dễ dùng cho N tiến trình.
- Nhưng ý nghĩa sử dụng không rõ ràng.
  - MutualExclusion : Down & Up.
  - Rendez-vous : Down & Up.
  - Chỉ phân biệt qua mô hình.
- Khó sử dụng đúng.
  - Nhầm lẫn.

Giải pháp Mutex

- Khi khả năng đếm của semaphore không cần thiết, một phiên bản đơn giản của semaphore, được gọi là mutex, đôi khi được sử dụng.
- Mutex: biến có thể ở một trong hai trạng thái bị khóa (locked – 0), không khóa (unlocked – 1 hoặc giá trị khác).
  - Dễ sử dụng.
- Phù hợp để sử dụng với tiểu trình ở không gian người dùng.
  - Tiểu trình (hoặc tiến trình) muốn truy cập CS, gọi mutex_lock.
  - Nếu mutex ở trạng thái được mở khóa (unlocked), lời gọi mutex_lock sẽ thành công. Nếu không, tiểu trình sẽ bị khóa cho đến khi mutex_unlock được gọi trong tiểu trình khác.

Giải pháp Mutex – Sử dụng pthread

- Tổ chức "độc quyền truy xuất".

```
pthread_mutex_t the mutex;
```

- Tổ chức "phối hợp xử lý".

```
pthread_mutex_t the mutex;
Pthread_cond_t cond;
```

T_i

```
pthread_mutex_lock (&the mutex);
CS;
pthread_mutex_unlock (&the mutex);
```

T_1 :

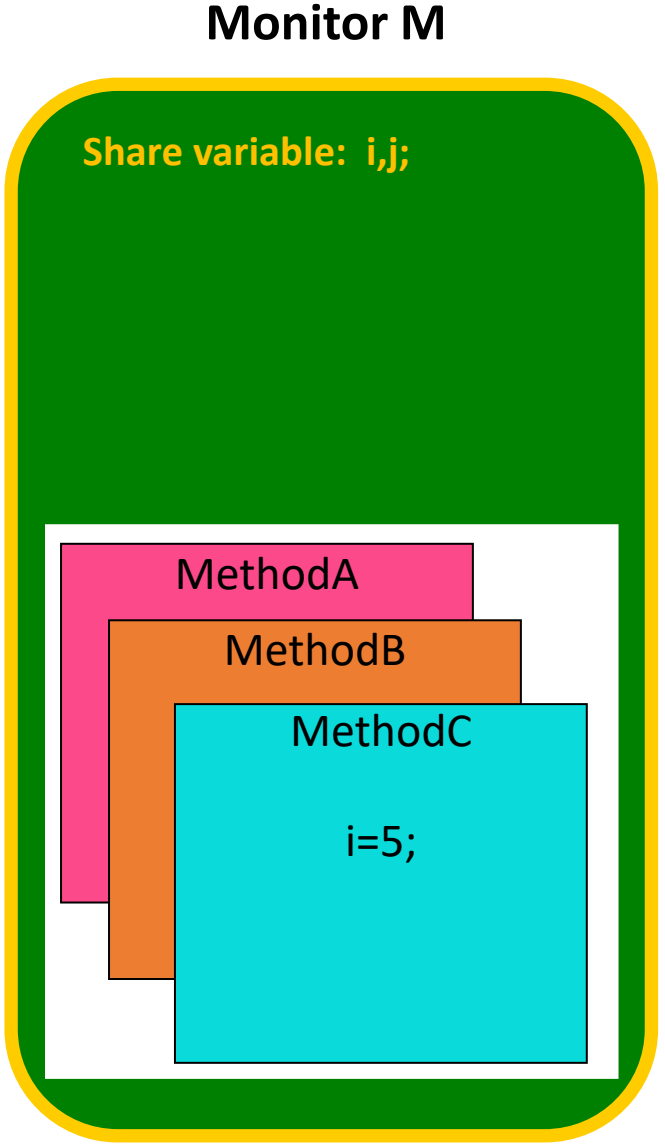
```
Job1;
pthread_mutex lock (&the mutex);
pthread cond signal (&cond);
pthread_mutex unlock (&the mutex);
```

T_2:

```
pthread_mutex_lock (&the mutex);
pthread_cond_wait (&cond, &the mutex);
pthread_mutex_unlock (&the mutex);
Job2;
```

Giải pháp Monitor

- Đề xuất bởi Hoare(1974) & Brinch (1975).
- Là cơ chế đồng bộ hoá do NNLT cung cấp.
  - Hỗ trợ cùng các chức năng như Semaphore.
  - Dễ sử dụng và kiểm soát hơn Semaphore.
    - Bảo đảm Mutual Exclusion một cách tự động.
    - Sử dụng biến điều kiện để thực hiện Synchronization.
- Là một module chương trình định nghĩa.
  - Các CTDL, đối tượng dùng chung.
  - Các phương thức xử lý các đối tượng này.
  - Bảo đảm tính encapsulation.
- Các tiến trình muốn truy xuất dữ liệu bên trong monitor phải dùng các phương thức của monitor:
  - P1 : M.C() // i=5
  - P2 : M.B() // printf(j)



Giải pháp Monitor – Sử dụng

- Tổ chức "độc quyền truy xuất".

```
Monitor       M
<resource type> RC;
Function   AccessMutual
                        CS; // access RC
```

```
Pi
M.AccessMutual(); //CS
```

- Tổ chức "phối hợp xử lý".

```
Monitor    M
Condition  c;
Function   F1()
              Job1;
              Signal(c);
Function F2()
              Wait(c);
              Job2;
```

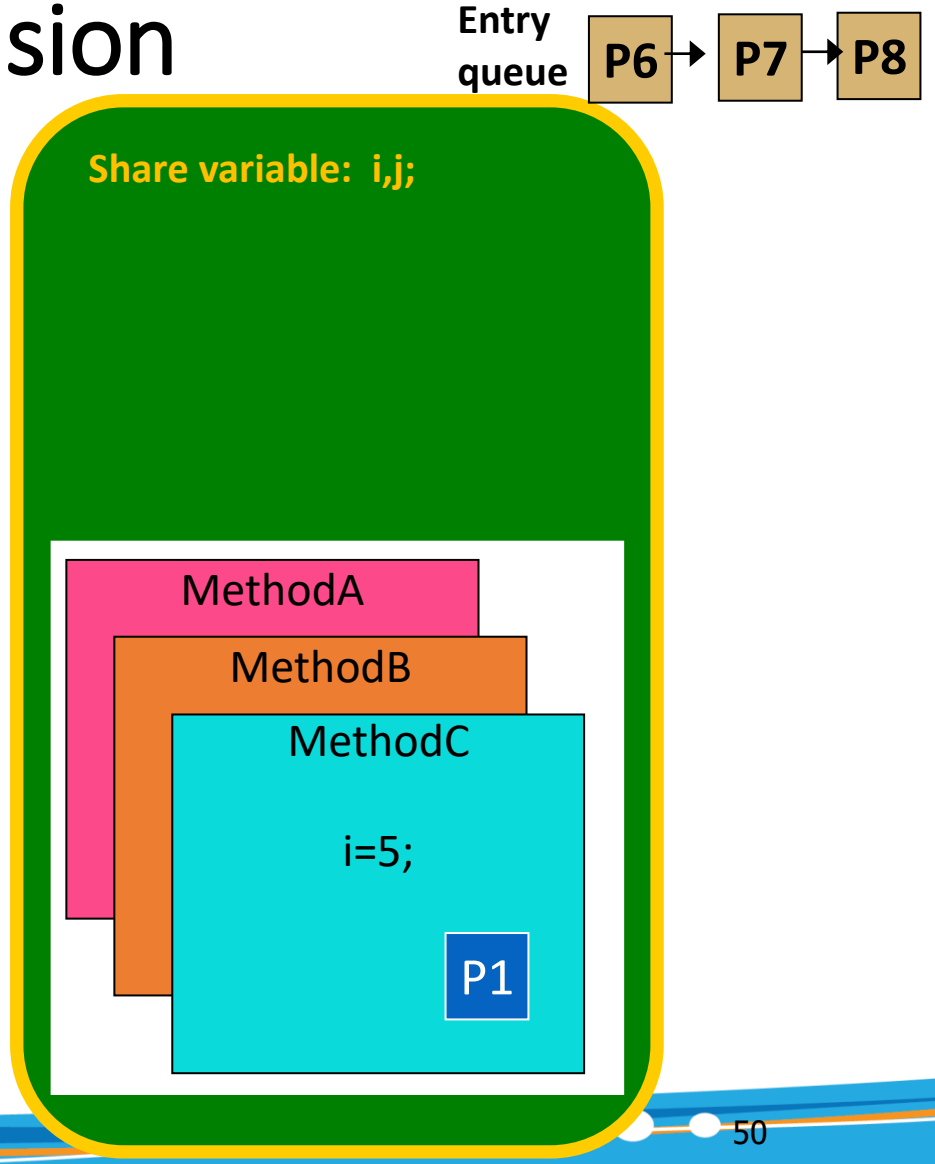
```
P1 :
M.F1();
```

→

```
P2:
 M.F2();
```

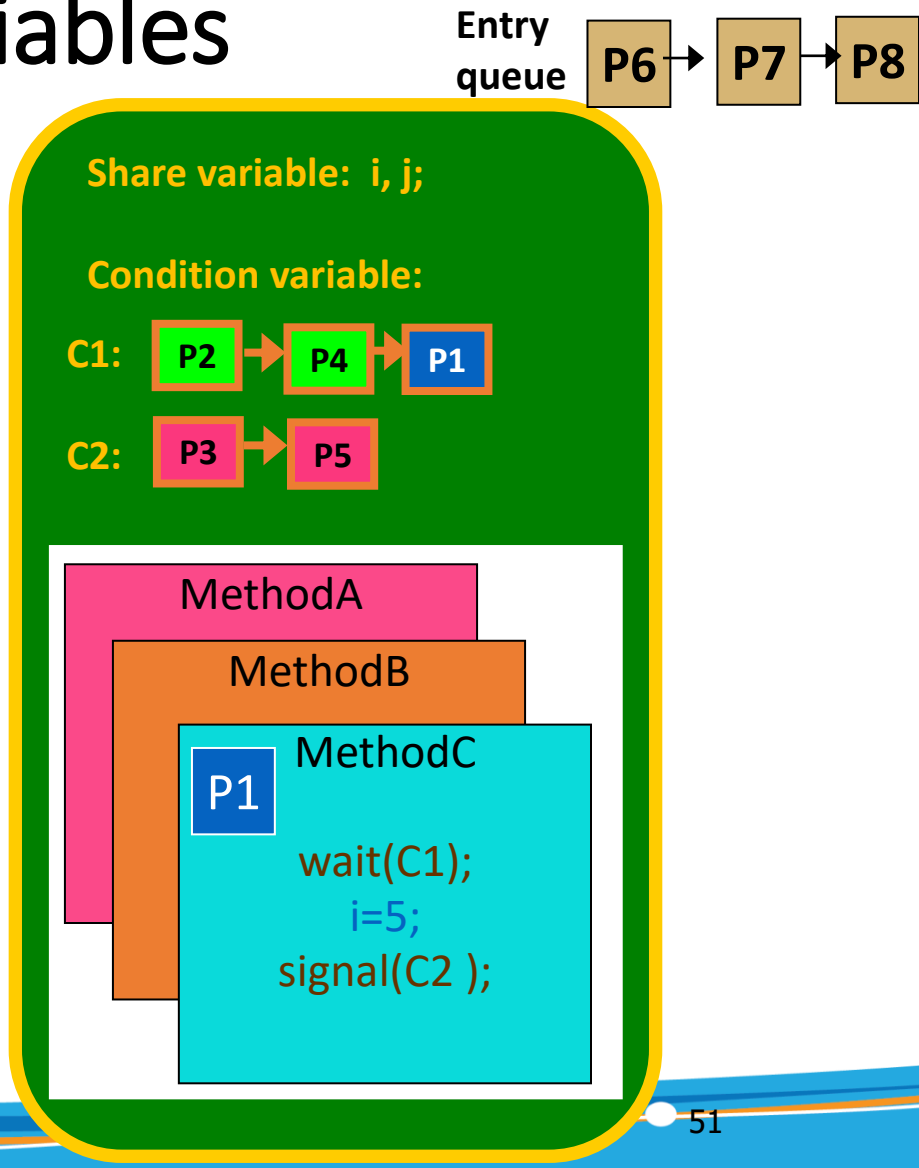
Giải pháp Monitor – Mutual Exclusion

- Tự động bảo đảm Mutual Exclusion.
  - Tại 1 thời điểm chỉ có 1 tiến trình được thực hiện các phương thức của Monitor.
  - Các tiến trình không thể vào Monitor sẽ được đưa vào Entry queue của Monitor.
- Ví dụ:
  - P1 : M.A();
  - P6 : M.B();
  - P7 : M.A();
  - P8 : M.C();



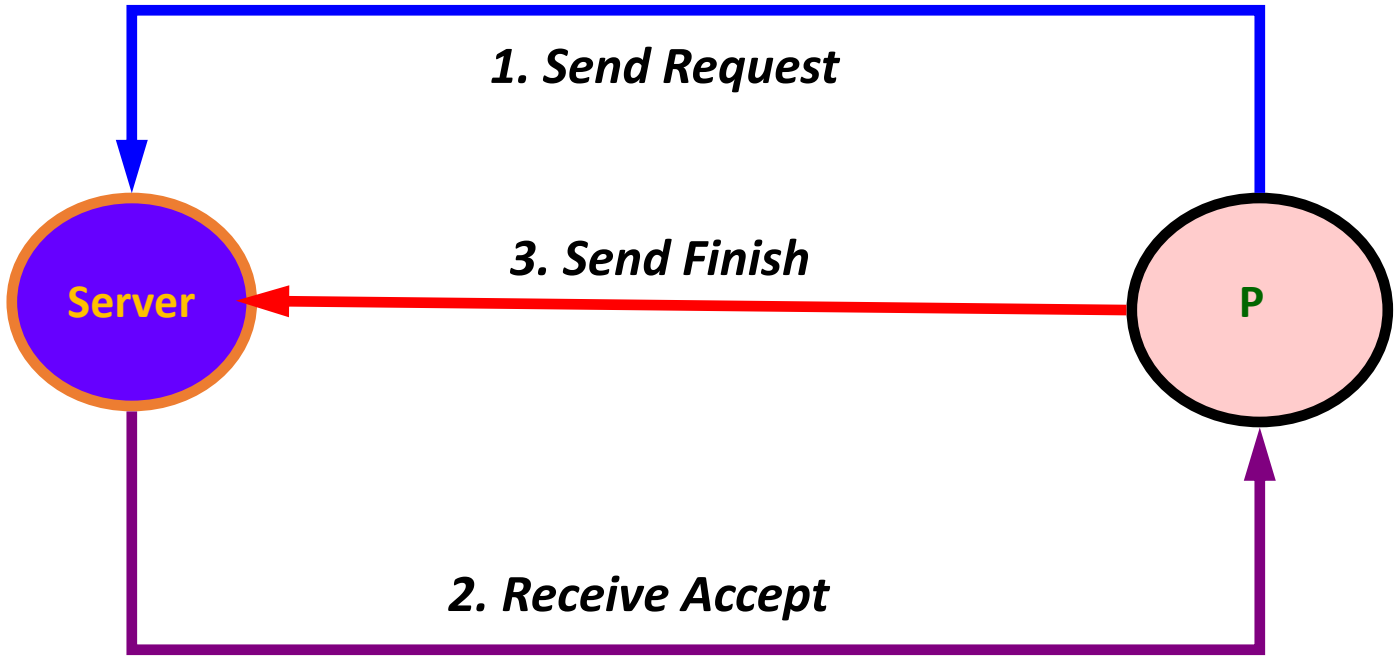
Giải pháp Monitor – Condition Variables

- Hỗ trợ Synchronization với các condition variables.
  - Wait(c) : Tiến trình gọi hàm sẽ bị blocked.
  - Signal(c): Giải phóng 1 tiến trình đang bị blocked trên biến điều kiện c.
  - C.queue : danh sách các tiến trình blocked trên c.
- Trạng thái tiến trình sau khi gọi Signal ?
  - Blocked. Nhường quyền vào monitor cho tiến trình được đánh thức.
  - Tiếp tục xử lý hết chu kỳ, rồi blocked.

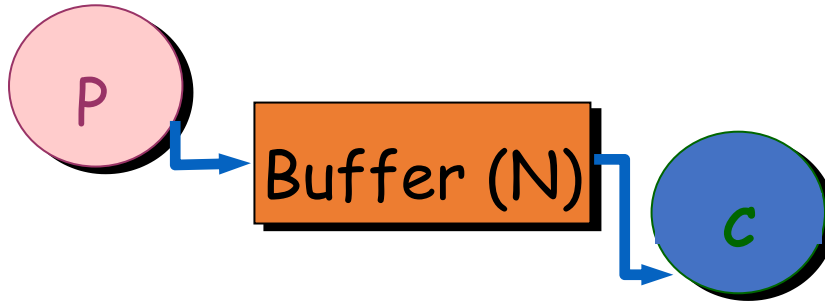


Giải pháp Message

- Được hỗ trợ bởi HĐH.
- Đồng bộ hóa trên môi trường phân tán.
- 2 primitive Send & Receive.
  - Cài đặt theo mode blocking.



Bài toán kinh điển Producer & Consumer



- Mô tả:
  - 2 tiến trình P và C hoạt động đồng hành.
  - P sản xuất hàng và đặt vào Buffer.
  - C lấy hàng từ Buffer đi tiêu thụ.
  - Buffer có kích thước giới hạn.
- Tình huống:
  - P và C đồng thời truy cập Buffer ?
  - P thêm hàng vào Buffer đầy ?
  - C lấy hàng từ Buffer trống ?
- Yêu cầu:
  - P không được ghi dữ liệu vào buffer đã đầy (Rendez-vous).
  - C không được đọc dữ liệu từ buffer đang trống (Rendez-vous).
  - P và C không được thao tác trên buffer cùng lúc (Mutual Exclusion).

```
Producer()
{
    int item;
    while (TRUE)
    {
        produce_item(&item);
        enter_item(item,Buffer);
    }
}
```

```
Consumer()
{
    int item;
    while (TRUE)
    {
        remove_item(&item,Buffer);
        consume_item(item);
    }
}
```

Producer & Consumer – Giải pháp Semaphore

- Các biến dùng chung giữa P và C:
  - BufferSize = N;
    // số chỗ trong bộ đệm
  - semaphore mutex = 1 ;
    // kiểm soát truy xuất độc quyền
  - semaphore empty = BufferSize;
    // kiểm soát số chỗ trống
  - semaphore full = 0;
    // kiểm soát số chỗ đầy
  - int Buffer[BufferSize];
    // bộ đệm dùng chung

```
Producer()
{
    int item;
    while (TRUE)
    {
        produce_item(&item);
        down(&empty);
        down(&mutex);
        enter_item(item,Buffer);
        up(&mutex);
        up(&full);
    }
}
```

```
Consumer()
{
    int item;
    while (TRUE)
    {
        down(&full);
        down(&mutex);
        remove_item(&item,Buffer);
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

Producer & Consumer – Giải pháp Semaphore – ?

```
Producer()
{
   int item;
   while (TRUE)
   {
            produce_item(&item);
             down(&mutex);
             down(&empty);
            enter_item(item,Buffer);
             up(&full);
             up(&mutex);
   }
}
```

```
Consumer()
{
   int item;
   while (TRUE)
    {
             down(&mutex);
             down(&full);
           remove_item(&item,Buffer);
             up(&empty);
             up(&mutex);
           consume_item(item);
    }
}
```

Producer & Consumer – Giải pháp Mutex

- Các biến dùng chung giữa P và C:
  - pthread_mutex_t the mutex;
  - pthread_cond_t condc, condp; /* used for signaling */
  - int buffer = 0; /* buffer used between producer and consumer */

```
Producer()
{
  for (int i= 1; i <= MAX; i++) {
    pthread_mutex_lock (&the mutex);
    /* get exclusive access to buffer */
    while (buffer != 0)
            pthread_cond_wait (&condp,
                                &the mutex);
    buffer = i; /* put item in buffer */
    pthread_cond_signal (&condc);
    /* wake up consumer */
    pthread_mutex_unlock (&the mutex);
    /* release access to buffer */
  }
  pthread exit(0);
}
```

```
Consumer()
{
  for (int i = 1; i <= MAX; i++) {
    pthread_mutex_lock(&the mutex);
    /* get exclusive access to buffer */
    while (buffer ==0 )
            pthread_cond_wait(&condc,
                                &the mutex);
    buffer = 0; /* take item out of buffer */
    pthread_cond_signal(&condp);
    /* wake up producer */
    pthread_mutex_unlock(&the mutex);
    /* release access to buffer */
  }
  pthread exit(0);
}
```

fit@hcmus

Producer & Consumer – Monitor

```
monitor ProducerConsumer
condition full, empty;
int Buffer[N], count;
procedure enter();
{
    if (count == N)
            wait(full);
    enter_item(item,Buffer);
    count ++;
   if (count == 1)
            signal(empty);
}
```

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

```
Producer()
{
    int item;
    while (TRUE)
    {
            produce_item(&item);
        ProducerConsumer.enter;
    }
}
```

```
Consumer();
{
    int item;
    while (TRUE)
    {
        ProducerConsumer.remove;
            consume_item(item);
    }
}
```

Producer & Consumer – Giải pháp Message

```
Producer()
{
    int item;
    message m;

    while (TRUE)
    {
        produce_item(&item);
        receive(consumer, Request);
        create_message(&m, item);
        send(consumer,&m);
    }
}
```

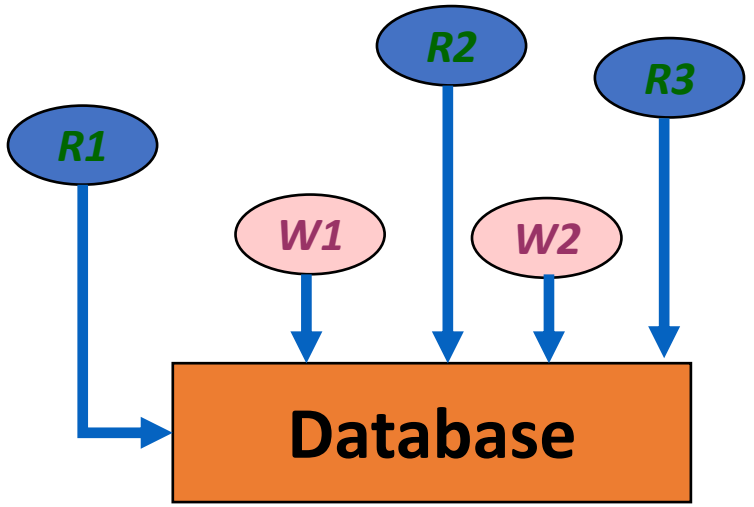
Coi chừng Deadlock

```
Consumer();
{
    int item;
    message m;
    for (0 to N)
        send(producer, Request);
    while (TRUE)
    {
        receive(producer, &m);
        remove_item(&m,&item);
        send(producer, Request);
        consumer_item(item);
    }
}
```


58

Bài toán kinh điển Readers & Writers

- Mô tả:
  - N tiến trình Ws và Rs hoạt động đồng hành.
  - Rs và Ws chia sẻ CSDL.
  - W cập nhật nội dung CSDL.
  - Rs truy cập nội dung CSDL.
- Tình huống:
  - Các Rs cùng truy cập CSDL ?
  - W đang cập nhật CSDL thì các Rs truy cập CSDL ?
  - Các Rs đang truy cập CSDL thì W muốn cập nhật CSDL ?



- Yêu cầu:
  - W không được cập nhật dữ liệu khi có ít nhất một R đang truy xuất CSDL (Mutual Exclusion).
  - Rs không được truy cập CSDL khi một W đang cập nhật nội dung CSDL (Mutual Exclusion).
  - Tại một thời điểm, chỉ cho phép một W được sửa đổi nội dung CSDL (Mutual Exclusion).

Readers & Writers – Giải pháp Semaphore (1/4)

- Các biến dùng chung giữa Rs và Ws
  - semaphore db = 1; // Kiểm tra truy xuất CSDL

```
Reader()
{
   down(&db);
   read-db(Database);
   up(&db);
}
```

```
Writer()
{
   down(&db);
   write-db(Database);
   up(&db);
}
```

- **Chuyện gì xảy ra ?**
  - **Chỉ có 1 Reader được đọc CSDL tại 1 thời điểm !**

fit@hcmus

Readers & Writers – Giải pháp Semaphore (2/4)

- Thêm biến dùng chung giữa các Rs.
  - int rc; // Số lượng tiến trình Reader

```
Reader()
{
   if (rc == 0)
         down(&db);
   rc = rc +1;
   read-db(Database);
   rc = rc – 1;
   if (rc == 0)
         up(&db);
}
```

```
Writer()
{
   down(&db);
   write-db(Database);
   up(&db);
}
```

- **Đúng chưa ?**
  - **rc là biến dùng chung giữa các Reader...**

  → CS đó

61

Readers & Writers – Giải pháp Semaphore (3/4)

- Thêm biến dùng chung giữa các Rs.
  - semaphore mutex1 = 1; // Kiểm tra truy xuất rc
  - semaphore mutex2 = 1; // Kiểm tra truy xuất rc

```
Reader()
{
    down(&mutex1);
    if (rc == 0)
            down(&db);
    rc = rc +1;
    up(mutex1);
    read-db(Database);
    down(mutex2);
    rc = rc – 1;
    if (rc == 0)
            up(&db);
    up(mutex2);
}
```

```
Writer()
{
    down(&db);
    write-db(Database);
    up(&db);
}
```

Đúng chưa ?

62

Readers & Writers – Giải pháp Semaphore (4/4)

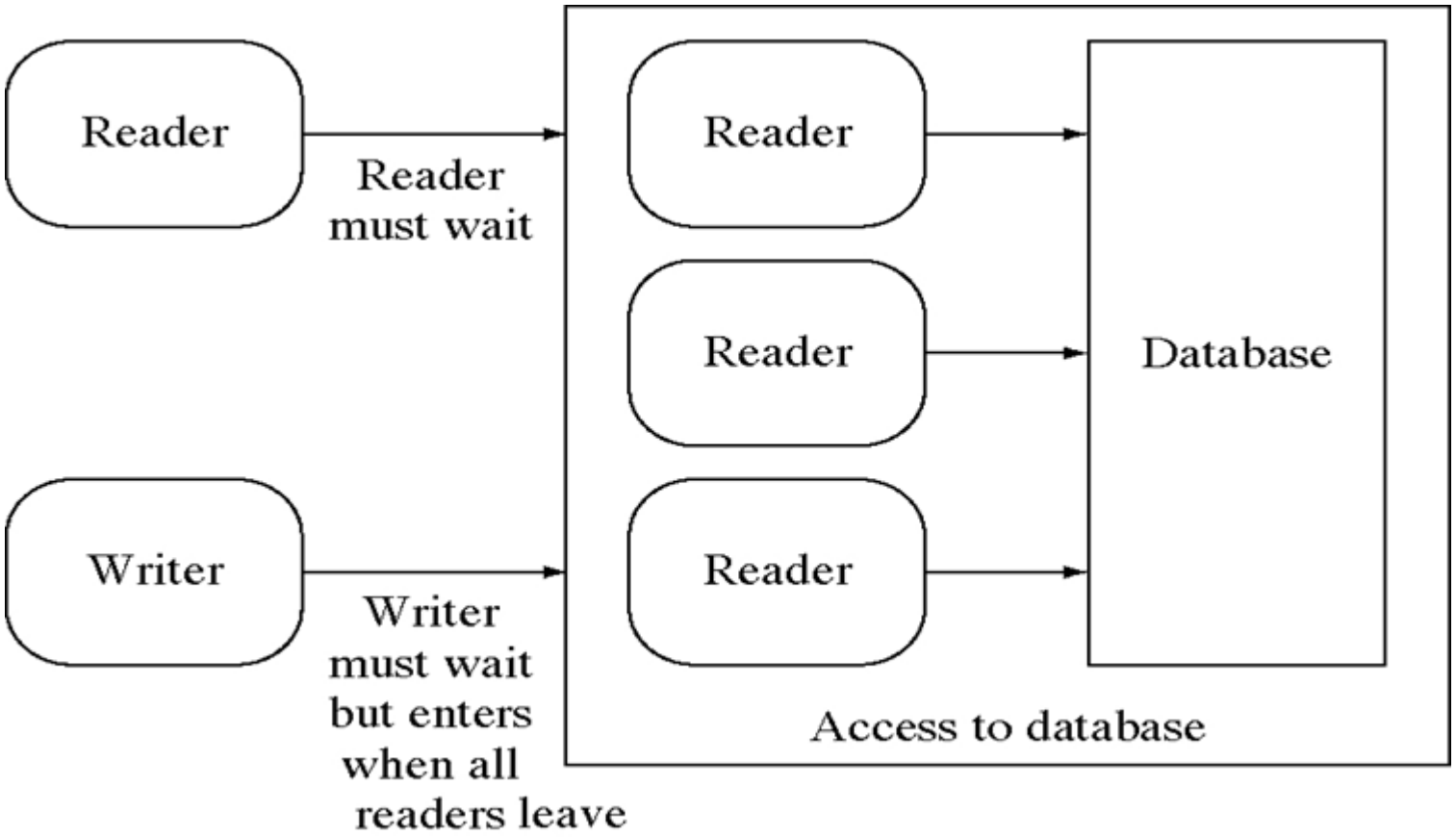
- Thêm biến dùng chung giữa các Rs.
  - semaphore mutex = 1;
    // Kiểm tra truy xuất rc

```
Reader()
{
    down(&mutex);
    if (rc == 0)
            down(&db);
    rc = rc +1;
    up(mutex);
    read-db(Database);
    down(mutex);
    rc = rc – 1;
    if (rc == 0)
            up(&db);
    up(mutex);
}
```

```
Writer()
{
    down(&db);
    write-db(Database);
    up(&db);
}
```


63

Readers & Writers – Vấn đề



- Tham khảo bài báo của Courtois và các cộng sự về giải pháp ưu tiên cho writers.

Readers & Writers – Giải pháp Monitor

```
monitor ReaderWriter
condition   OKWrite, OKRead;
int         rc = 0;
Boolean     busy = false;
procedure BeginRead() {
    if (busy)
        wait(OKRead);
    rc++;
    signal(OKRead);
}
procedure FinishRead() {
    rc--;
    if (rc == 0)
        signal(OKWrite);
}
```

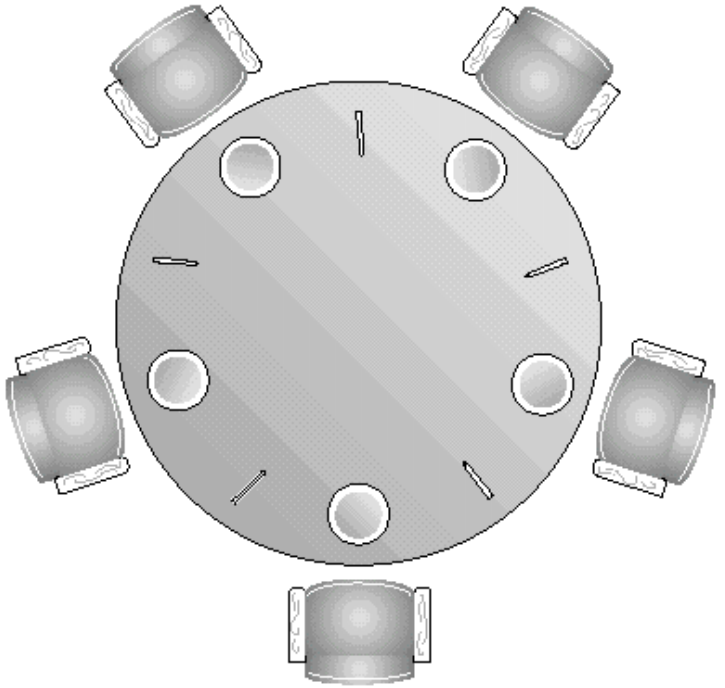
```
procedure BeginWrite()
{
    if (busy || rc != 0)
        wait(OKWrite);
    busy = true;
}
procedure FinishWrite()
{
    busy = false;
    if (OKRead.Queue)
        signal(OKRead);
    else
        signal(OKWrite);
}
end monitor;
```

```
Reader()
{
    RW.BeginRead();
    Read-db(Database);
    RW.FinishRead();
}
```

```
Writer();
{
    RW.BeginWrite();
    Write-db(Database);
    RW.FinishWrite();
}
```

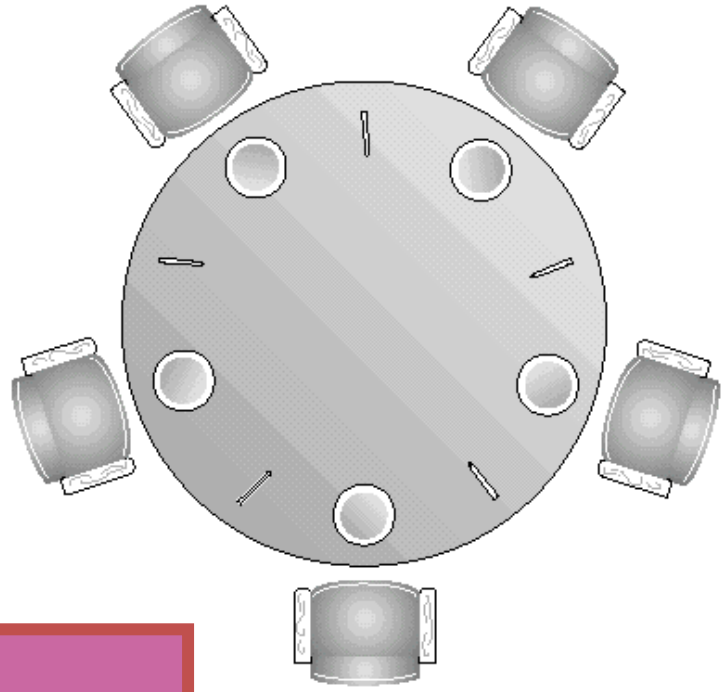
Bài toán kinh điển Dining Philosophers

- Mô tả
  - Năm triết gia ngồi chung quanh bàn ăn món spaghetti.
  - Trên bàn có 5 cái nĩa được đặt giữa 5 cái đĩa (xem hình).
  - Để ăn món spaghetti mỗi người cần có 2 cái nĩa.
  - Triết gia thứ i:
    - Thinking...
    - Eating...
- Tình huống:
  - 2 triết gia "giành giật" cùng 1 cái nĩa.
  - Tranh chấp.
- Yêu cầu:
  - Cần đồng bộ hoá hoạt động của các triết gia sao cho không deadlock và starvation.



Dining Philosophers – Giải pháp Semaphore

```
semaphore fork[5] = 1;
Philosopher (i) {
        while (true) {
                think();
                down(fork[i]);
                down(fork[i+1 mod 5]);
                eat();
                up(fork[i]);
                up(fork[i+1 mod 5]);
        }
}
```



Deadlock

- Xem thêm giải pháp trong giáo trình…