



# HỆ ĐIỀU HÀNH

# Operating Systems

1

## NỘI DUNG

---

- Chương 1: Tổng quan
- Chương 2: Quản lý tiến trình**
- Chương 3: Deadlock
- Chương 4: Quản lý bộ nhớ
- Chương 5: Hệ thống file
- Chương 6: Quản lý nhập xuất

---

2

2

## Chương 2

# Quản lý tiến trình



3  
www.cunghodaptrinh.com

3

## Nội dung

---

1. Tiến trình (process)
2. Luồng (thread)
3. Truyền thông giữa các tiến trình
4. Đồng bộ hóa tiến trình
5. Điều phối tiến trình.

---

4

4

## 2.1. Tiến trình

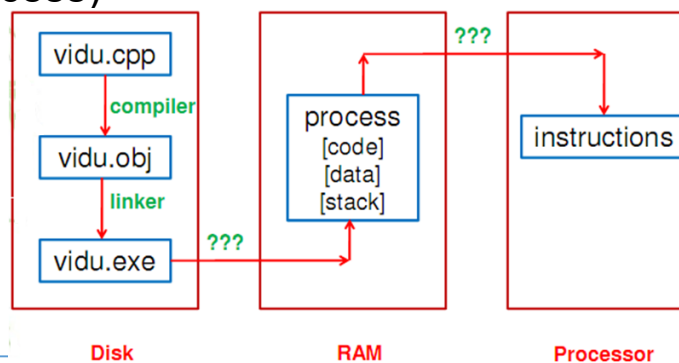
- Mô hình
- Hiện thực

5

5

### 2.1.1 Mô hình Tiến trình

- Hệ thống máy tính hiện đại cho phép nhiều chương trình được nạp vào bộ nhớ và thực hiện đồng thời → cần có cơ chế kiểm soát hoạt động của các chương trình khác nhau → khái niệm tiến trình (process)

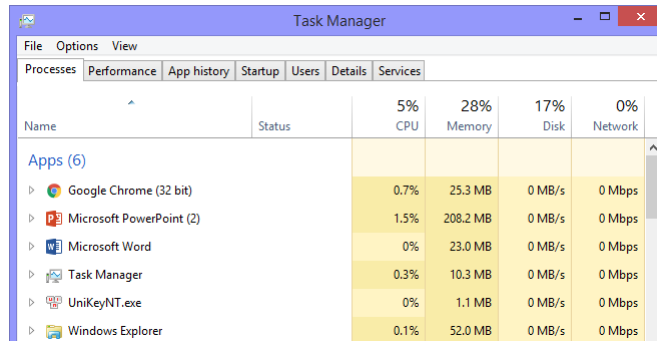


6

6

## Mô hình Tiến trình

- Tiến trình là một chương trình đang được thực thi
- Một tiến trình cần sử dụng các tài nguyên: CPU, bộ nhớ, tập tin, thiết bị nhập xuất để hoàn tất công việc của nó



Name	Status	CPU	Memory	Disk	Network
<b>Apps (6)</b>					
Google Chrome (32 bit)		0.7%	25.3 MB	0 MB/s	0 Mbps
Microsoft PowerPoint (2)		1.5%	208.2 MB	0 MB/s	0 Mbps
Microsoft Word		0%	23.0 MB	0 MB/s	0 Mbps
Task Manager		0.3%	10.3 MB	0 MB/s	0 Mbps
UniKeyNT.exe		0%	1.1 MB	0 MB/s	0 Mbps
Windows Explorer		0.1%	52.0 MB	0 MB/s	0 Mbps

7

7

## Mô hình Tiến trình

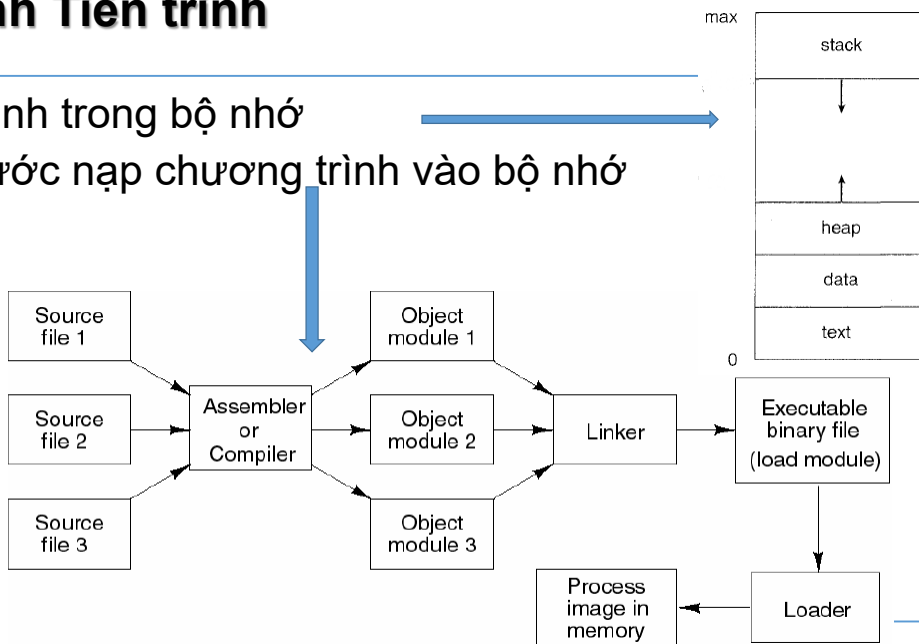
- Một tiến trình bao gồm:
  - Text section (program code), data section (chứa global variables)
  - program counter (PC), process status word (PSW), stack pointer (SP), memory management registers,...
- Phân biệt chương trình và tiến trình:
  - Chương trình: là một thực thể thụ động (tập tin có chứa một danh sách các lệnh được lưu trữ trên đĩa - file thực thi)
  - Tiến trình: là một thực thể hoạt động, với một bộ đếm chương trình xác định các chỉ lệnh kế tiếp để thực hiện và một số tài nguyên liên quan.
  - Một chương trình sẽ trở thành một tiến trình khi một tập tin thực thi được nạp vào bộ nhớ.

8

8

## Mô hình Tiến trình

- Tiến trình trong bộ nhớ
- Các bước nạp chương trình vào bộ nhớ



9

### 2.1.2 Hiện thực Tiến trình

- Vai trò của hệ điều hành trong quản lý tiến trình
- Tạo tiến trình
- Trạng thái tiến trình – chuyển đổi trạng thái tiến trình
- Khối quản lý tiến trình (PCB – Process Control Block)
- Các thao tác với tiến trình
- Hoạt động của tiến trình
- Cấp phát tài nguyên cho tiến trình

10

## Vai trò của hệ điều hành trong quản lý tiến trình

- Tạo và hủy tiến trình của người dùng và tiến trình hệ thống
- Dừng, khôi phục (resume) tiến trình
- Cung cấp các cơ chế để đồng bộ hóa tiến trình
- Cung cấp các cơ chế liên lạc giữa các tiến trình
- Cung cấp cơ chế xử lý bế tắc (deadlock)

11

11

## Tạo Tiến trình

- Các bước hệ điều hành khởi tạo tiến trình:
  - Cấp phát định danh duy nhất cho tiến trình (process number, process identifier, pid)
  - Cấp phát không gian nhớ để nạp tiến trình
  - Khởi tạo khối dữ liệu Process Control Block (PCB) lưu các thông tin về tiến trình được tạo.
  - Thiết lập các mối liên hệ cần thiết (vd: sắp PCB vào hàng đợi định thời,...)

12

12

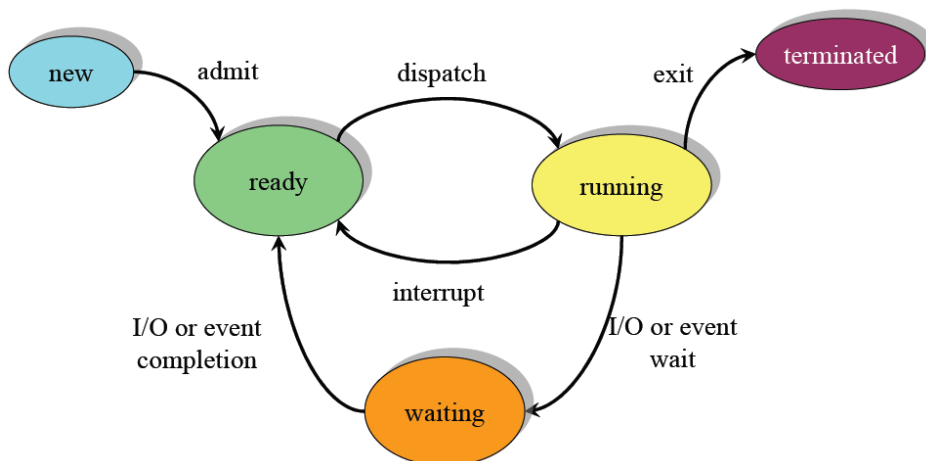
## Các trạng thái của tiến trình

- **Mới tạo**: tiến trình đang được tạo lập.
- **Running**: các chỉ thị của tiến trình đang được xử lý.
- **Blocked (waiting)**: tiến trình chờ được cấp phát một tài nguyên, hay chờ một sự kiện xảy ra .
- **Ready**: tiến trình chờ được cấp phát CPU để xử lý.
- **Kết thúc**: tiến trình hoàn tất xử lý.

13

13

## Các trạng thái của tiến trình



14

14

## Các trạng thái của tiến trình

### • Các trạng thái của tiến trình:

- Tiến trình mới tạo được đưa vào hệ thống
- Bộ điều phối cấp phát cho tiến trình một khoảng thời gian sử dụng CPU
- Tiến trình kết thúc
- Tiến trình yêu cầu một tài nguyên nhưng chưa được đáp ứng hoặc tiến trình phải chờ một sự kiện hay thao tác nhập/xuất.
- Bộ điều phối chọn một tiến trình khác để cho xử lý .
- Tài nguyên mà tiến trình yêu cầu đã sẵn sàng để cấp phát; hay sự kiện hoặc thao tác nhập/xuất tiến trình đang đợi hoàn tất.

15

15

## Các trạng thái của tiến trình

### • Ví dụ: đoạn chương trình C

### • Biên dịch

- gcc test.c -o test

### • Thực thi chương trình test

- ./test

```
/* test.c */
int main(int argc, char** argv)
{
    printf("Hello \n");
    exit (0);
}
```

### • Tiến trình test được tạo ra, thực thi và kết thúc.

### • Trạng thái của tiến trình Test:

- new → ready → running → waiting (do chờ I/O khi gọi printf) → ready → running → terminated

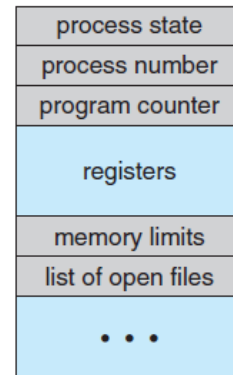
16

16



## Khối quản lý tiến trình (PCB – Process Control Block)

- Mỗi tiến trình được tạo được cấp phát một (PCB) - là một cấu trúc dữ liệu lưu các thông tin:
  - Định danh (Process Number)
  - Trạng thái tiến trình (Process State)
  - Bộ đếm chương trình
  - Các thanh ghi
  - Thông tin lập thời biểu CPU: độ ưu tiên, con trỏ đến hàng đợi,...
  - Thông tin quản lý bộ nhớ
  - Thông tin tài khoản: lượng CPU, thời gian sử dụng,
  - Thông tin trạng thái I/O



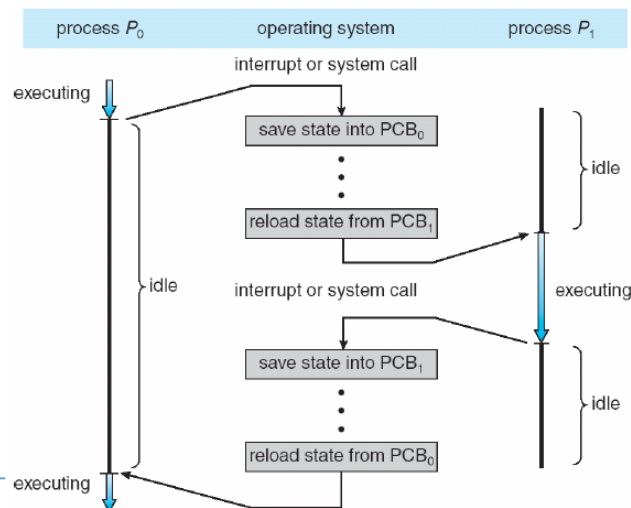
Process control block (PCB).

17

17

## PCB (tt)

- Sơ đồ chuyển CPU giữa các tiến trình



18

18

## Các thao tác với tiến trình

### • Tạo tiến trình mới

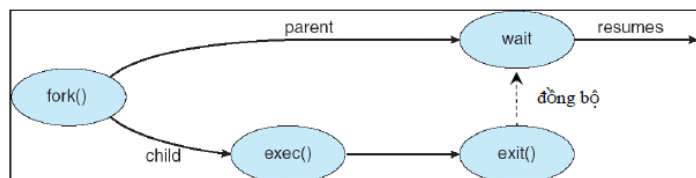
- Tiến trình con nhận tài nguyên từ HĐH hoặc từ tiến trình cha
- Tài nguyên của tiến trình con được chia sẻ từ tiến trình cha:
  - Tiến trình cha và con chia sẻ mọi tài nguyên
  - Tiến trình con chia sẻ một phần tài nguyên của cha
- Trình tự thực thi
  - Tiến trình cha và con thực thi đồng thời (concurrently)
  - Tiến trình cha đợi đến khi các tiến trình con kết thúc
- Không gian địa chỉ (address space)
  - Không gian địa chỉ của tiến trình con được nhân bản từ cha
  - Không gian địa chỉ của tiến trình con được khởi tạo từ template

19

19

## Các thao tác với tiến trình

```
#include <stdio.h>
#include <unistd.h>
int main (int argc, char *argv[]){
    int pid;
    pid = fork();
    if (pid > 0) {
        printf("This is parent process");
        wait (NULL);
        exit(0);
    }
    else if (pid == 0) {
        printf("This is child process");
        execlp("/bin/ls", "ls", NULL);
        exit(0);
    }
    else {
        printf("Fork error\n");
        exit(-1);
    }
}
```



20

20

## Các thao tác với tiến trình

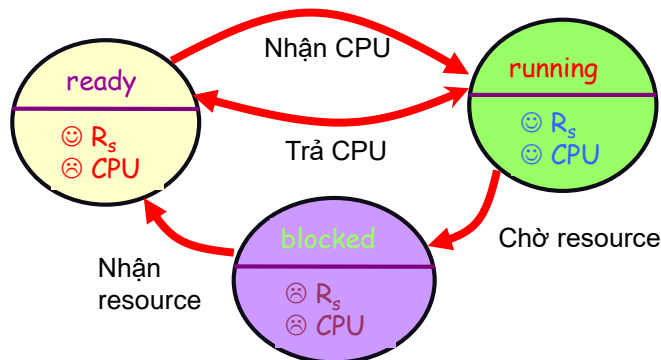
### • Kết thúc tiến trình

- Tiến trình tự kết thúc: tiến trình kết thúc khi thực thi lệnh cuối và gọi **exit**
- Tiến trình kết thúc do tiến trình khác (ví dụ tiến trình cha)
  - Gọi **abort** với tham số là pid của tiến trình cần được kết thúc
- Hệ điều hành thu hồi tất cả các tài nguyên của tiến trình kết thúc (vùng nhớ, I/O buffer,...)

21

21

## Hoạt động của các tiến trình

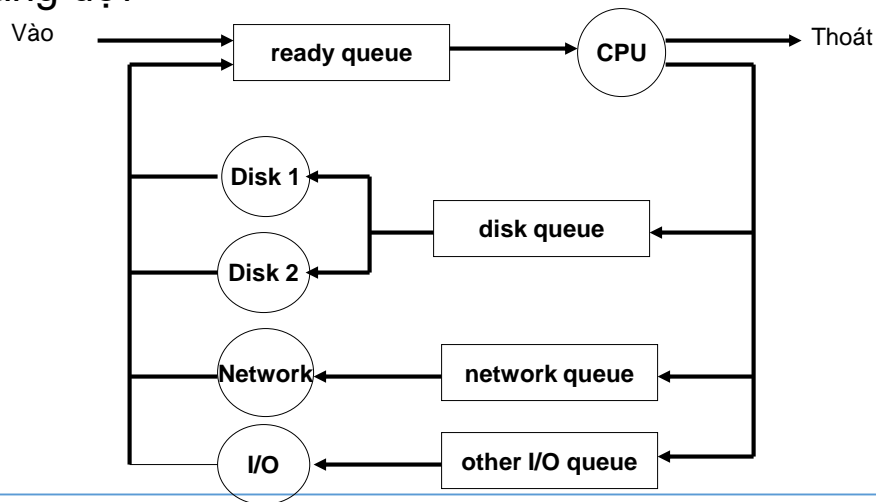


22

22

## Hoạt động của các tiến trình (tt)

### • Sơ đồ hàng đợi



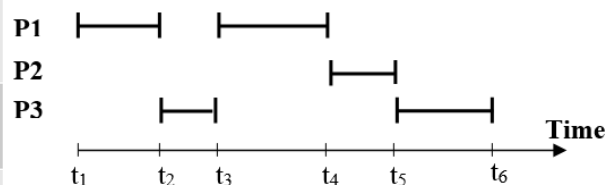
23

23

## Hoạt động của các tiến trình (tt)

### • Ví dụ: chuyển processor giữa 3 tiến trình :

Thời điểm	Trạng thái các tiến trình
$t_1$	$P_1$ : được cấp CPU
$t_2$	$P_1$ : bị thu hồi CPU (khi chưa kết thúc) $P_3$ : được cấp CPU
$t_3$	$P_3$ : bị thu hồi CPU (khi chưa kết thúc) $P_1$ : được cấp CPU
$t_4$	$P_1$ : kết thúc và trả lại CPU $P_2$ : được cấp CPU
$t_5$	$P_2$ : kết thúc và trả lại CPU $P_3$ : được cấp CPU
$t_6$	$P_3$ : kết thúc và trả lại CPU



24

24

## Cấp phát tài nguyên cho tiến trình

- Mục tiêu cấp phát tài nguyên:
  - Tối ưu hóa sự sử dụng tài nguyên
  - Bảo đảm một số lượng hợp lệ các tiến trình truy xuất đồng thời đến các tài nguyên không chia sẻ được.
  - Cấp phát tài nguyên cho tiến trình có yêu cầu trong một khoảng thời gian có thể chấp nhận được.
- Cấp phát tài nguyên:
  - Định danh tài nguyên
  - Trạng thái tài nguyên: phần nào đã/chưa cấp phát cho tiến trình
  - Hàng đợi trên một tài nguyên: danh sách các tiến trình đang chờ được cấp phát tài nguyên
  - Bộ cấp phát: là đoạn code xử lý việc cấp phát một tài nguyên

25

25

## 2.2. Luồng (thread)

- Mô hình
- Hiện thực

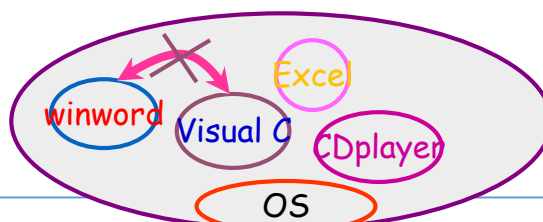
26

26

## 2.2.1 Mô hình luồng

### •Nhìn lại giải pháp đa tiến trình

- Các tiến trình độc lập, không có sự liên lạc với nhau
- Mỗi tiến trình thực thi, có một không gian địa chỉ và một dòng xử lý
- → tiến trình thực hiện chỉ có một tác vụ tại một thời điểm.
- Muốn trao đổi thông tin với nhau, các chương trình cần được xây dựng theo mô hình liên lạc đa tiến trình (IPC – Inter-Process Communication) → Phức tạp, chi phí cao



27

27

## Mô hình luồng (tt)

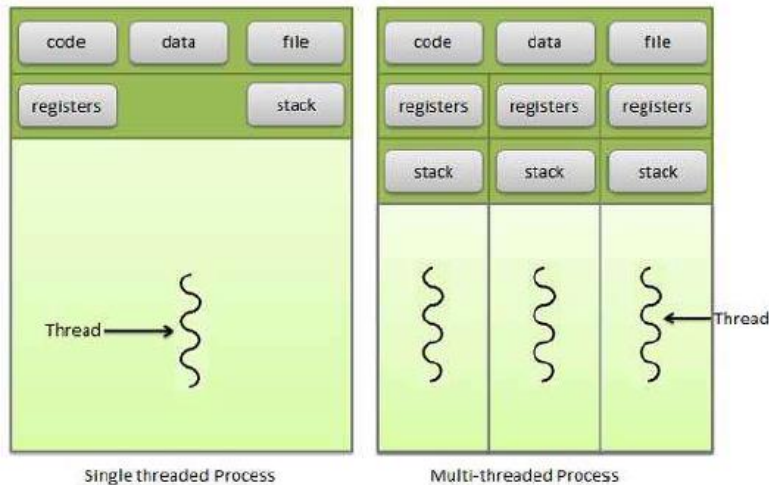
- Để tăng tốc độ và sử dụng CPU hiệu quả hơn:
    - Cần nhiều dòng xử lý cùng chia sẻ một không gian địa chỉ
    - Các dòng xử lý hoạt động song song tương tự như tiến trình phân biệt
    - Mỗi dòng xử lý được gọi là một luồng (thread)
  - Hầu hết các HĐH hiện đại đều được thực hiện theo mô hình luồng
- Ví dụ:
- Các xử lý trên máy chủ web
  - Ứng dụng Word: nhận ký tự gõ vào, kiểm tra chính tả, tự động sao lưu,....

28

28

## Mô hình luồng (tt)

### •Đơn luồng và đa luồng

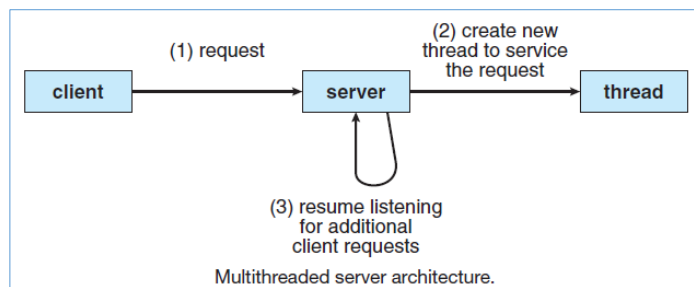


29

29

## Mô hình luồng (tt)

- Luồng là một dòng xử lý trong một tiến trình
- Mỗi tiến trình luôn có một luồng chính (dòng xử lý cho hàm main())
- Ngoài luồng chính, tiến trình còn có thể có nhiều luồng con khác
- Các luồng của một tiến trình
  - Chia sẻ không gian vùng code và data
  - Có vùng stack riêng



30

30

## Mô hình luồng (tt)

### •Ưu điểm của luồng:

- Tính đáp ứng (responsiveness): chương trình tiếp tục chạy nếu một phần của nó bị blocked hay đang thực thi một thao tác dài → đáp ứng nhanh đến người sử dụng, đặc biệt trong các hệ thống chia sẻ thời gian thực.
- Chia sẻ tài nguyên (resource sharing): các luồng chia sẻ bộ nhớ và tài nguyên của tiến trình, và các luồng có liên quan (luồng tạo ra nó) → cho phép một ứng dụng có nhiều luồng khác nhau hoạt động trong cùng một không gian địa chỉ

31

31

## Mô hình luồng (tt)

### •Ưu điểm của luồng (tt)

- Tính kinh tế (economy):
  - Cấp phát bộ nhớ và tài nguyên cho việc tạo tiến trình: tốn kém.
  - Luồng chia sẻ tài nguyên của tiến trình → tạo luồng và chuyển ngữ cảnh ít tốn thời gian (thấp hơn 5 lần so với tiến trình)
  - Thời gian tạo tiến trình gấp 30 lần so với tạo luồng
- Khả năng mở rộng (**Scalability**): tận dụng được kiến trúc đa xử lý (Utilization of multiprocessor architecture):
  - Luồng có thể chạy song song trên các CPU khác nhau.
  - Đa luồng trên máy nhiều CPU làm tăng tính đồng thời.

32

32



## 2.2.2 Hiện thực luồng

---

- Các trạng thái của luồng
- Phân loại luồng
- Mô hình đa luồng
- Khối quản lý luồng
- Tạo luồng
- So sánh luồng và tiến trình

33

33

## Các trạng thái của luồng

---

- Running: luồng đang được thực thi bởi bộ vi xử lý
- Ready: luồng đang ở trong trạng thái chờ CPU
- Blocked: luồng đang chờ một sự kiện I/O

34

34

## Chuyển đổi trạng thái của luồng

- **Spawn:**
  - Khi một tiến trình được tạo → luồng chính được tạo.
  - Một luồng có thể tạo ra các luồng khác, cung cấp một con trỏ lệnh và các thông số cho các luồng mới (thanh ghi ngữ cảnh riêng, không gian stack, hàng đợi ready)
- **Block:** khi luồng cần đợi một sự kiện → blocked (lưu thanh ghi của người sử dụng, bộ đếm chương trình và con trỏ stack) → chuyển CPU cho một tiến trình khác.
- **Unblock:** khi sự kiện mà luồng đang ở trạng thái block đang đợi xuất hiện → chuyển sang trạng thái Ready.
- **Finish:**
  - Khi một luồng hoàn tất → trả tất cả tài nguyên cho hệ điều hành.
  - Khi một tiến trình kết thúc, tất cả các luồng của tiến trình cũng kết thúc

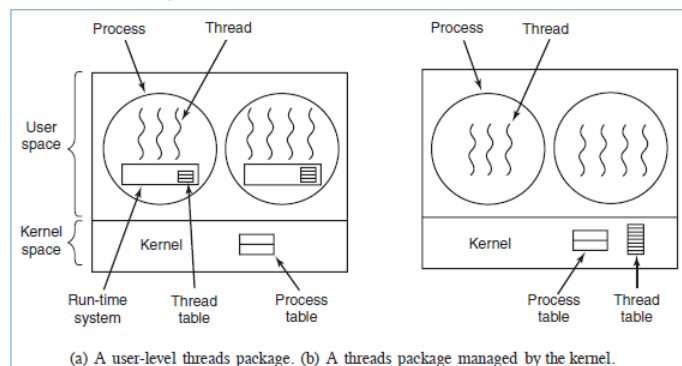
35

35

## Phân loại luồng

### • Users thread

- Luồng được đặt trong user space
- Được hiện thực bởi Thread library:
  - Khởi tạo, định thời và quản lý luồng
  - Truyền thông giữa các luồng
  - Lưu giữ và khôi phục ngữ cảnh của luồng
  - Không cần hỗ trợ từ kernel



36

36

## Phân loại luồng (tt)

### •Users thread (tt)

- Ưu điểm: tạo và quản lý nhanh.
- Nhược điểm:
  - Nếu kernel là single threaded, một luồng bị blocked → tất cả các luồng khác cũng bị blocked.
  - Không tận dụng kiến trúc nhiều CPU: hai luồng của một tác vụ không thể chạy trên hai CPU.
- Một số thư viện Users thread:
  - POSIX pthreads
  - Mach C-threads.
  - Solaris UI-threads

37

37

## Phân loại luồng (tt)

### •Kernel thread:

- Kernel thread được hỗ trợ trực tiếp bởi hệ điều hành.
- Kernel thực hiện việc tạo luồng, định thời và quản lý trong không gian kernel.
- Bởi vì việc quản lý thread được thực hiện bởi hệ điều hành, kernel thread được tạo và quản lý chậm hơn user thread.
- Kernel quản lý luồng → nếu một luồng bị blocked → có thể chuyển một luồng khác trong ứng dụng để thực thi → trong môi trường nhiều CPU, kernel có thể định thời cho luồng trên các CPU khác nhau.

38

38

## Các mô hình đa luồng

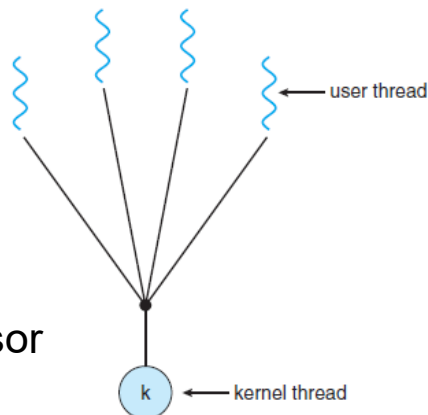
- Nhiều HĐH cung cấp hỗ trợ cho cả user thread và kernel thread, thể hiện trong ba mô hình phổ biến:
  - Many to one
  - One to one
  - Many to Many

39

39

## Mô hình many to one

- Nhiều user thread được ánh xạ đến một kernel thread.
- Việc quản lý luồng được thực trong user space → hiệu quả
- Toàn bộ tiến trình sẽ bị blocked nếu một luồng bị blocked → tại một thời điểm chỉ có một luồng có thể truy nhập kernel → Các luồng không thể chạy song song trong các hệ thống multiprocessor

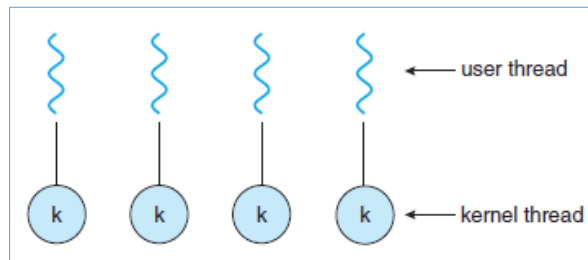


40

40

## Mô hình one to one

- Mỗi user thread được ánh xạ một kernel thread.
- Các luồng khác vẫn chạy khi một luồng bị khóa.
- Nhiều luồng được chạy song song trên nhiều CPU.
- Hạn chế: khi một user thread được tạo ra → phải tạo ra một kernel thread → cần giới hạn số luồng (được hỗ trợ do HĐH)

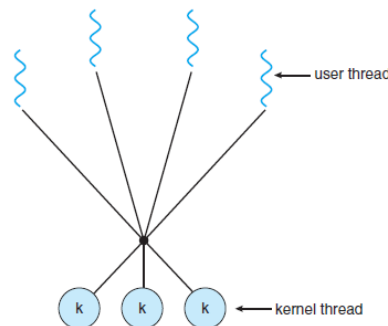


41

41

## Mô hình many to many

- N user thread được ánh xạ đến N (hoặc  $M < N$ ) kernel thread.
- Người phát triển có thể tạo ra nhiều luồng tùy ý tuy nhiên không đạt tới được sự đồng thời đúng nghĩa bởi vì tại một thời điểm kernel chỉ có thể định thời cho một luồng.
- Các kernel thread có thể chạy song song trên nhiều CPU → khi một luồng bị treo kernel có thể định thời cho luồng khác hoạt động



42

42

## Khối quản lý luồng (Thread Control Block – TCB)

- TCB chứa các thông tin của mỗi luồng
  - ID của luồng
  - Không gian lưu các thanh ghi
  - Con trỏ tới vị trí xác định trong ngăn xếp
  - Trạng thái của luồng
  - Thông tin chia sẻ giữa các luồng trong một tiến trình
    - Các biến toàn cục
    - Các tài nguyên sử dụng như tập tin,...
    - Các tiến trình con
    - Thông tin thống kê
    - ...

43

43

## Ví dụ sử dụng luồng

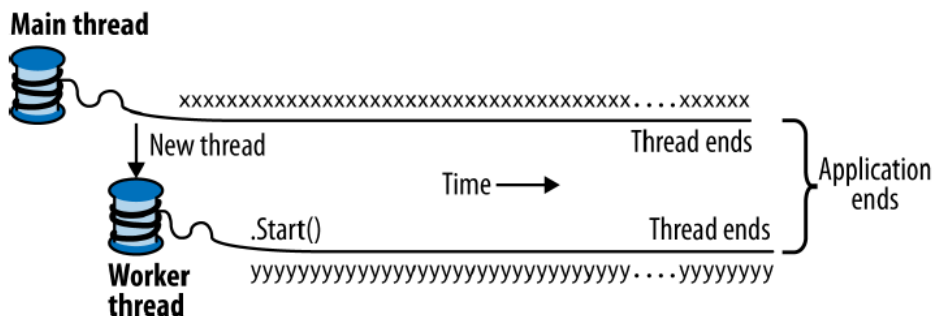
```
using System;
using System.Threading;
class ThreadTest
{
    static void Main()
    {
        Thread t = new Thread (WriteY);
        t.Start(); // running WriteY()
        for (int i = 0; i < 1000; i++) Console.Write ("x");
    }
    static void WriteY()
    {
        for (int i = 0; i < 1000; i++) Console.Write ("y");
    }
}
```

```
// Typical Output:
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
```

44

44

## Tạo luồng – ví dụ



45

45

## So sánh luồng và tiến trình

- Tại sao không dùng nhiều tiến trình để thay thế cho việc dùng nhiều luồng ?
  - Các tác vụ điều hành luồng (tạo, kết thúc, điều phối, chuyển đổi,...) ít tốn chi phí thực hiện hơn so với tiến trình
  - Liên lạc giữa các luồng thông qua chia sẻ bộ nhớ, không cần sự can thiệp của kernel

46

46

## So sánh luồng và tiến trình

Tiến trình	Luồng
Các tác vụ điều hành tiến trình tốn nhiều tài nguyên	Các tác vụ điều hành luồng ít tốn chi phí thực hiện hơn so với tiến trình
Tiến trình chuyển đổi cần tương tác với hệ điều hành.	Liên lạc giữa các luồng thông qua chia sẻ bộ nhớ, không cần sự can thiệp của kernel
Trong nhiều môi trường xử lý, mỗi tiến trình thực thi có nguồn tài nguyên bộ nhớ và tập tin riêng của mình	Tất cả các luồng có thể chia sẻ các tập tin mở, tiến trình con.
Nếu một tiến trình bị khóa (blocked), không có tiến trình nào khác có thể thực thi cho đến khi tiến trình ban đầu được unblocked	Khi một tiến trình bị khóa và chờ, luồng kế tiếp trong cùng một task có thể chạy
Nhiều tiến trình hoạt động độc lập với các tiến trình khác	Một luồng có thể đọc, ghi, thay đổi dữ liệu của luồng khác

47

47

## 2.3. Truyền thông giữa các tiến trình

### •Mục tiêu:

- Chia sẻ thông tin.
- Hợp tác hoàn thành tác vụ:
  - Chia nhỏ tác vụ để có thể thực thi song song.
  - Dữ liệu ra của tiến trình này là dữ liệu đầu vào cho tiến trình khác
- HĐH cần cung cấp cơ chế để các tiến trình có thể trao đổi thông tin với nhau.

48

48



## 2.3.1 Các dạng tương tác giữa các tiến trình

- Tín hiệu (Signal)
- Pipe
- Vùng nhớ chia sẻ
- Trao đổi thông điệp (Message)
- Sockets

49

49

### 2.3.1.1 Tín hiệu (Signal)

- Sử dụng tín hiệu để thông báo cho tiến trình khi có một sự kiện xảy ra
- Mỗi tiến trình có một bảng biểu diễn các tín hiệu khác nhau.
- Mỗi tín hiệu (*signal handler*) có một trình xử lý tương ứng.
- Các tín hiệu được gửi đi bởi :
  - Phần cứng (ví dụ lỗi do các phép tính số học)
  - Kernel gửi đến một tiến trình (ví dụ: báo cho tiến trình khi có một thiết bị I/O rồi).
  - Một tiến trình gửi đến một tiến trình khác (ví dụ: tiến trình cha yêu cầu một tiến trình con kết thúc)
  - Người dùng (ví dụ nhấn phím Ctl-C để ngắt xử lý của tiến trình)

50

50

## Tín hiệu (Signal)

### • Một số tín hiệu của UNIX

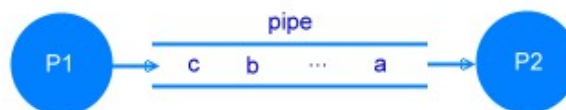
Tín hiệu	Mô tả
SIGINT	Người dùng nhấn phím DEL để ngắt xử lý tiến trình
SIGQUIT	Yêu cầu thoát xử lý
SIGILL	Tiến trình xử lý một chỉ thị bất hợp lệ
SIGKILL	Yêu cầu kết thúc một tiến trình
SIGFPT	Lỗi floating – point xảy ra ( chia cho 0)
SIGPIPE	Tiến trình ghi dữ liệu vào pipe mà không có reader
SIGSEGV	Tiến trình truy xuất đến một địa chỉ bất hợp lệ
SIGCLD	Tiến trình con kết thúc
SIGUSR1	Tín hiệu 1 do người dùng định nghĩa
SIGUSR2	Tín hiệu 2 do người dùng định nghĩa

51

51

### 2.3.1.2 Pipe

- Dữ liệu xuất của tiến trình này được chuyển đến làm dữ liệu nhập cho tiến trình kia dưới dạng **một dòng các byte**.
- Khi một pipe được thiết lập giữa hai tiến trình:
  - Một tiến trình ghi dữ liệu vào pipe
  - Một tiến trình đọc dữ liệu từ pipe.
- Thứ tự dữ liệu truyền qua pipe: FIFO
- Một pipe có kích thước giới hạn (thường là 4096 ký tự)



52

52

## Pipe (tt)

- Một tiến trình có thể sử dụng một pipe do nó tạo ra hay kế thừa từ tiến trình cha.
- Vai trò của Hệ điều hành:
  - Cung cấp các lời gọi hệ thống (hàm) read/write cho các tiến trình thực hiện thao tác đọc/ghi dữ liệu trong pipe
  - Đồng bộ hóa việc truy xuất :
    - Nếu pipe trống: tiến trình đọc sẽ bị khóa
    - Nếu pipe đầy: tiến trình ghi sẽ bị khóa
- Ví dụ sử dụng pipe: lệnh trong command line
  - ls | more (Linux)
  - dir | more (Windows)

53

53

### 2.3.1.3 Vùng nhớ chia sẻ

- Cho phép nhiều tiến trình cùng truy xuất đến một vùng nhớ chung gọi là *vùng nhớ chia sẻ (shared memory)*.
- Các tiến trình chia sẻ một vùng nhớ vật lý thông qua không gian địa chỉ của tiến trình.
- Vùng nhớ chia sẻ tồn tại độc lập với các tiến trình
- Một tiến trình muốn truy xuất đến vùng nhớ chia sẻ cần phải kết gắn vùng nhớ đó vào không gian địa chỉ riêng của tiến trình để thao tác trên đó.



54

54

### 2.3.1.4 Trao đổi thông điệp (Message)

- Các tiến trình liên lạc với nhau thông qua việc gửi thông điệp.
- HĐH cung cấp các hàm IPC chuẩn (Interprocess communication):
  - Send ([destination], message): gửi một thông điệp
  - Receive ([source], message): nhận một thông điệp
- Khi hai tiến trình muốn liên lạc với nhau:
  - Thiết lập một mối liên kết giữa hai tiến trình
  - Sử dụng các hàm IPC thích hợp để trao đổi thông điệp
  - Hủy liên kết.

55

55

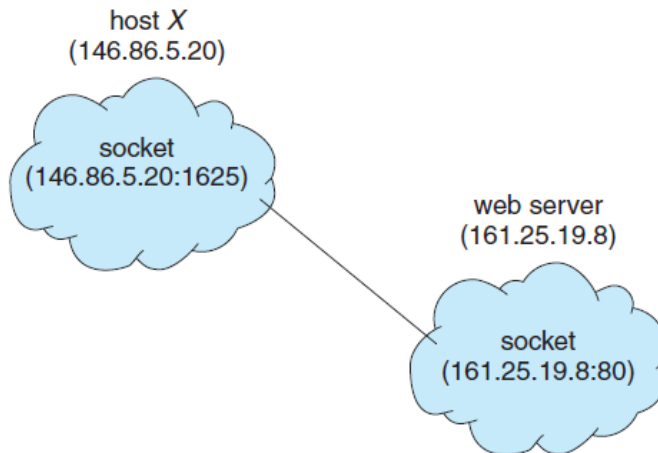
### 2.3.1.5 Sockets

- Được dùng để trao đổi dữ liệu giữa các máy tính trên mạng
- Một socket là một thiết bị truyền thông hai chiều để gửi và nhận dữ liệu giữa các máy trên mạng.
- Mỗi socket là một đầu nối giữa một máy đến một máy tính khác
- Thao tác đọc/ghi là sự trao đổi dữ liệu ứng dụng trên nhiều máy khác nhau.
- Sử dụng socket có thể mô phỏng hai phương thức liên lạc trong thực tế:
  - Liên lạc thư tín (socket đóng vai trò bưu cục)
  - Liên lạc điện thoại (socket đóng vai trò tổng đài) .

56

56

## Sockets (tt)



57

57

## Sockets (tt)

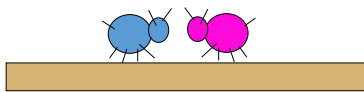
- Các bước trao đổi dữ liệu bằng socket:
  - Tạo socket
  - Gắn kết một socket với một địa chỉ (IP, Port)
  - Liên lạc: có hai kiểu liên lạc tùy thuộc vào chế độ nối kết:
    - Liên lạc trong chế độ không kết nối
    - Liên lạc trong chế độ có kết nối
  - Hủy socket

58

58

## 2.3.2 Vấn đề tranh chấp tài nguyên

- Trong một hệ thống có nhiều tiến trình cùng chạy
- Các tiến trình có thể chia sẻ tài nguyên chung (file system, CPU...)
- 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)



59

59

## Race Condition

- Ví dụ 1: đếm số người truy cập website, có 2 tiến trình chia sẻ đoạn code cập nhật biến đếm như sau:

```

counter = 0

    > P1                                > P2
    read counter;                        read counter;
counter = counter + 1;                  counter = counter + 1;

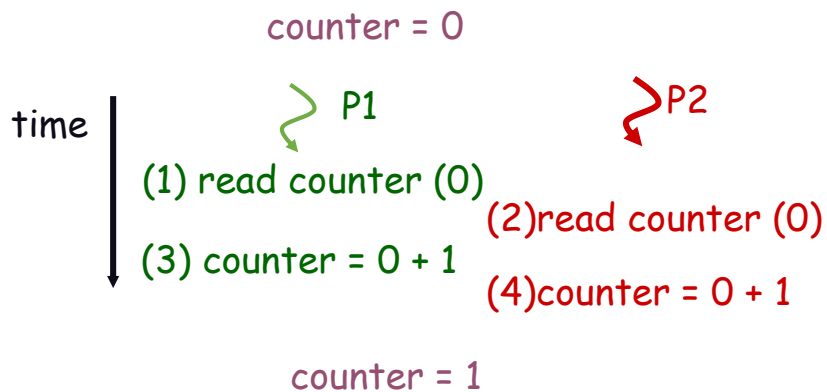
                                counter?
  
```

60

60

## Race Condition (tt)

### • Ví dụ 1 (tt)

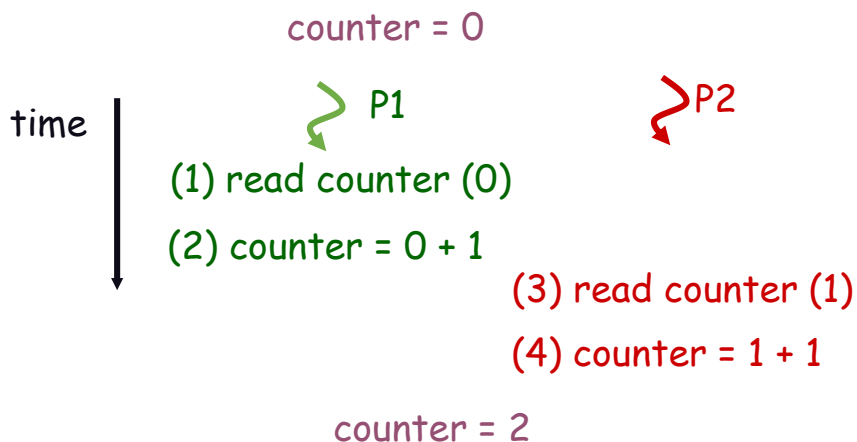


61

61

## Race Condition (tt)

### • Ví dụ 1 (tt)



62

62

## Race Condition (tt)

- Ví dụ 2: Bài toán rút tiền từ ngân hàng
  - if (tongtien >= tienrut) tong tien = tong tien –tien rut
  - Nếu có hai tiến trình cùng thực hiện?

63

63

## Race Condition (tt)

- Kết quả thực hiện tiến trình phụ thuộc vào kết quả điều phối
  - Cùng dữ liệu đầu vào, không chắc cùng dữ liệu ra
  - Khó debug lỗi sai trong xử lý đồng hành
- Lý do xảy ra Race condition?:
  - Hai tiến trình đọc và ghi dữ liệu trên cùng một vùng nhớ chung
  - Một tiến trình “xen vào” quá trình truy xuất tài nguyên của một tiến trình khác
- **Giải pháp:** cho phép tiến trình hoàn tất việc truy xuất tài nguyên chung trước khi có tiến trình khác can thiệp.

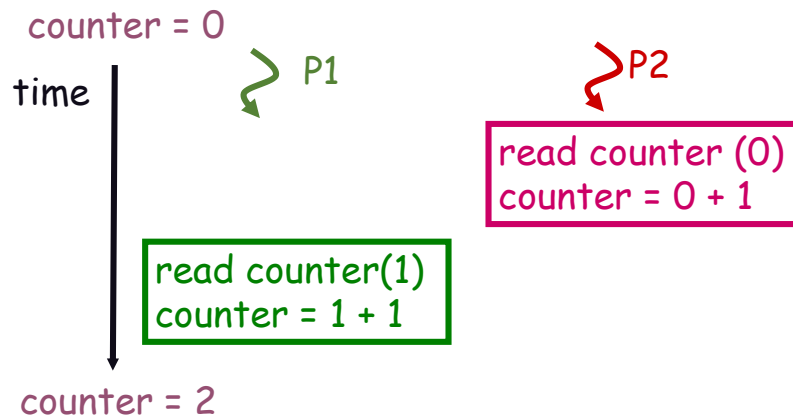
64

64



## Race Condition (tt)

- Giải pháp cho ví dụ 1:

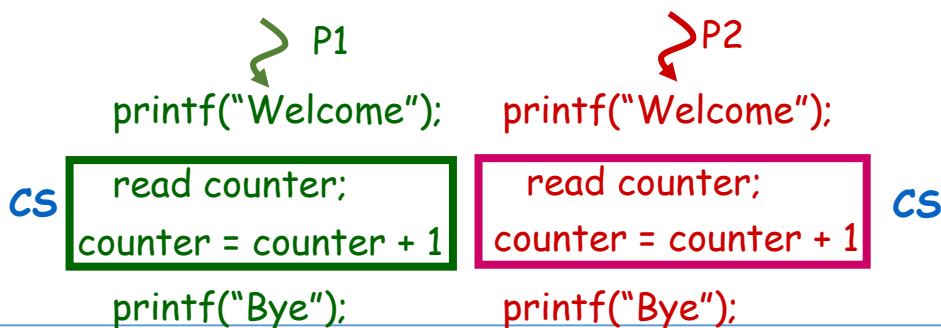


65

65

## Miền găng (Critical Section) & Khả năng độc quyền (Mutual Exclusion)

- Miền găng (CS) là đoạn chương trình có khả năng gây ra hiện tượng race condition
- Cần bảo đảm tính “độc quyền truy xuất” (Mutual Exclusion) cho miền găng (CS)



66

66

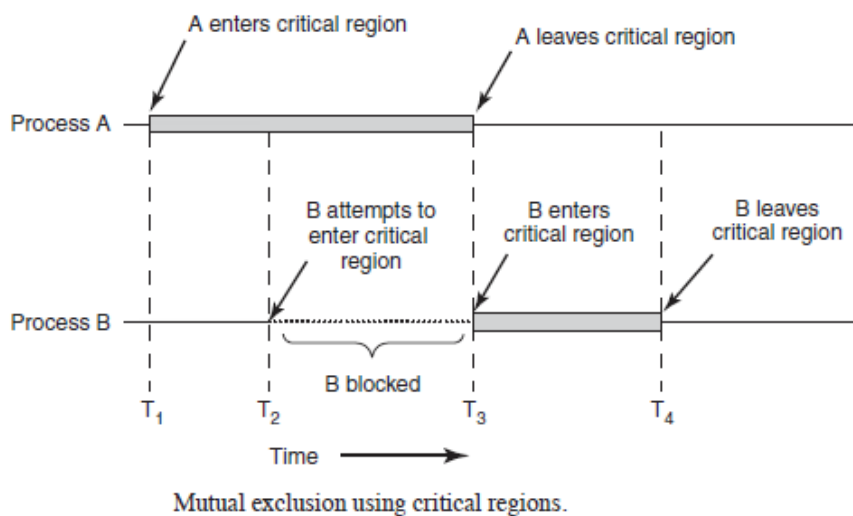
## Giải pháp cho vấn đề tranh chấp tài nguyên

- **Đồng bộ hóa tiến trình:** bảo đảm tại một thời điểm chỉ có duy nhất một tiến trình được xử lý trong miền găng.
- **Bốn mục tiêu đồng bộ hóa tiến trình:**
  1. **Mutual exclusion:** Không có hai tiến trình cùng ở trong miền găng cùng lúc.
  2. **Progress:** Một tiến trình bên ngoài miền găng không được ngăn cản các tiến trình khác vào miền găng
  3. **Bounded waiting:** Không có tiến trình nào phải chờ vô hạn để được vào miền găng.
  4. Không có giả định nào đặt ra về tốc độ của các tiến trình hoặc số lượng CPU.

67

67

## Giải pháp cho vấn đề tranh chấp tài nguyên (tt)



68

68

## Giải pháp cho vấn đề tranh chấp tài nguyên (tt)

### • Mô hình đảm bảo Mutual Exclusion:

- Thêm các đoạn code đồng bộ hóa vào chương trình gốc

**Kiểm tra quyền vào CS**

CS;

**Từ bỏ quyền vào CS**

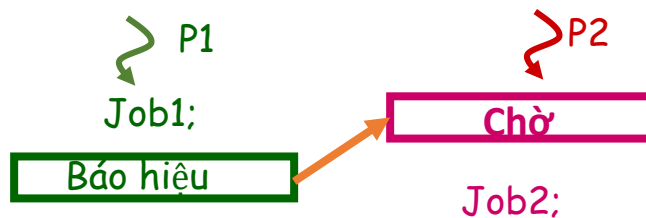
69

69

## Giải pháp cho vấn đề tranh chấp tài nguyên (tt)

### • Mô hình tổ chức phối hợp giữa hai tiến trình:

- Thêm các đoạn code đồng bộ hóa vào 2 chương trình gốc
- Không có mô hình tổng quát cho nhiều tiến trình



70

70

## 2.3.3 Đồng bộ hóa tiến trình

### •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

- Cắm ngắt
- Chỉ thị TSL

### •Nhóm giải pháp Sleep & Wakeup

- Semaphore
- Monitor
- Message

71

71

### 2.3.3.1 Các giải pháp “Busy waiting”

```
While (chưa có quyền) donothing() ;
```

CS;

Từ bỏ quyền sử dụng CS

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

72

72

## Sử dụng biến cờ hiệu

- Sử dụng biến **lock** làm khóa chốt, khởi động là 0
- P muốn vào miền găng phải kiểm tra giá trị của lock
  - lock = 0: đặt lock = 1 và đi vào miền găng, sau khi ra khỏi miền găng đặt lock = 0.
  - lock = 1: chờ

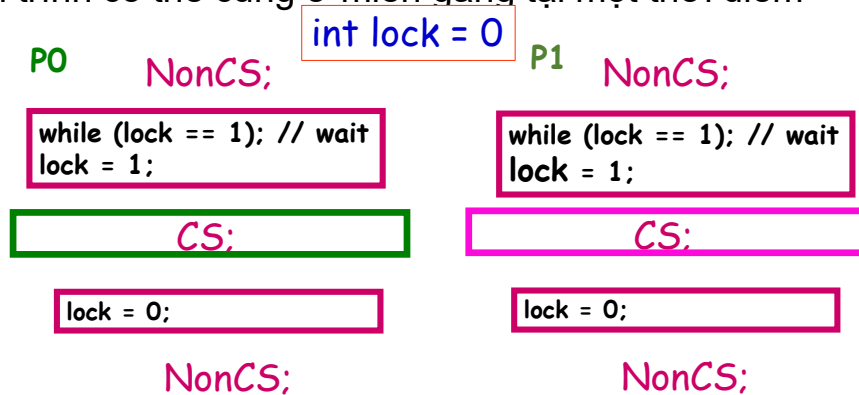
```
while (TRUE)
{
    while (lock == 1); // wait
    lock = 1;
    critical-section ();
    lock = 0;
    Noncritical-section ();
}
```

73

73

## Sử dụng biến cờ hiệu (tt)

- Nhận xét:
  - Có thể mở rộng cho N tiến trình
  - Hai tiến trình có thể cùng ở miền găng tại một thời điểm



74

74

## Kiểm tra luân phiên

- Hai tiến trình sử dụng chung biến turn, khởi động = 0
  - turn = 0: P0 được vào miền găng, P1 chờ
  - turn = 1: P1 được vào miền găng, P0 chờ
- Khi tiến trình rời khỏi miền găng:
  - P0: đặt turn = 1
  - P1: đặt turn = 0

```

while (TRUE)      P0
{
    while (turn != 0); // wait
    critical-section ();
    turn = 1;
    Noncritical-section ();
}
  
```

```

while (TRUE)      P1
{
    while (turn != 1); // wait
    critical-section ();
    turn = 0;
    Noncritical-section ();
}
  
```

75

75

## Kiểm tra luân phiên (tt)

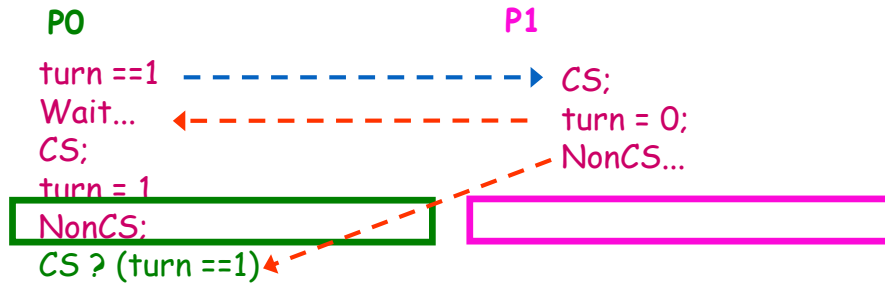
- Chỉ dành cho 2 tiến trình
- Bảo đảm Mutual Exclusion: ngăn chặn được tình trạng hai tiến trình cùng vào miền găng do kiểm tra biến turn
- Vi phạm Progress: một tiến trình ở ngoài miền găng có thể ngăn chặn tiến trình khác vào miền găng

76

76

## Kiểm tra luân phiên (tt)

- Vi phạm Progress : `int turn = 1`



**P0 không vào được CS lần 2 khi P1 dừng trong NonCS !**

77

77

## Peterson

- Kết hợp ý tưởng của 1 & 2, các tiến trình chia sẻ các biến:
  - `int turn` // đến phiên ai
  - `int interested [2]` // xác định tiến trình muốn vào CS  
`interested [ i ] = TRUE`: tiến trình  $P_i$  muốn vào miền găng.
- Khởi đầu:
  - `interested [0] = interested [1] = FALSE`;
  - `turn = 0` hay `1`.
- $P_i$  muốn vào được miền găng:
  - `interested [ i ] = TRUE`
  - `turn = i`
- $P_i$  rời khỏi miền găng:
  - `interested [ i ] = FALSE`

78

78

## Peterson (tt)

```
#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 = 1 - process; /* the opposite of process */
    interested [process] = TRUE; /* show that you are interested */
    turn = process; /* set flag */
    while (turn == process && interested[other] == TRUE) ;
}
void leave_region (int process) { /* process: who is leaving */
    interested [process] = FALSE;
}
```

79

79

## Peterson (tt)

- Nhận xét: đáp ứng được cả 3 điều kiện
  - Mutual Exclusion :
    - $P_i$  chỉ có thể vào CS khi:  $interested[j] == \text{False}$  hay  $turn == i$
    - Do biến  $turn$  chỉ có thể nhận giá trị 0 hay 1 nên chỉ có 1 tiến trình vào CS
  - Progress
    - Sử dụng 2 biến  $interested[i]$  riêng biệt  $\Rightarrow$  trạng thái đối phương không khoá mình được
  - Bounded Wait :  $interested[i]$  và  $turn$  đều có thay đổi giá trị
  - Không thể mở rộng cho  $N$  tiến trình

80

80



## Cấm ngắt

- Cho phép tiến trình cấm tất cả các ngắt trước khi vào miền găng, và phục hồi ngắt khi ra khỏi miền găng – kể cả ngắt đồng hồ
- → hệ thống không thể tạm dừng hoạt động của tiến trình đang xử lý để cấp phát CPU cho tiến trình khác.

NonCS;

Disable Interrupt;

CS;

Enable Interrupt;

NonCS;

81

81

## Cấm ngắt (tt)

- Thiếu thận trọng: cho phép tiến trình người dùng được phép thực hiện lệnh cấm ngắt.
- 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 → không đảm bảo Mutual Exclusion.

82

82

## Chỉ thị TSL (Test-and-Set)

- Giải pháp này đòi hỏi sự trợ giúp của cơ chế phần cứng.
- Hệ thống cung cấp một chỉ thị đặc biệt cho phép kiểm tra và cập nhật nội dung một vùng nhớ không chia sẻ
- Hai chỉ thị TSL xử lý đồng thời (trên hai bộ xử lý khác nhau) sẽ được xử lý tuần tự.
- Đảm bảo truy xuất độc quyền:
  - Sử dụng thêm một biến **lock**, khởi gán lock = FALSE, được kiểm tra trước khi tiến trình vào CS: lock = FALSE → vào CS

83

83

## Chỉ thị TSL (tt)

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

```
do {
    while (test and set (&lock));
    critical-section ();
    lock = false;
    Noncritical-section ();
} while (true);
```

84

84

## Chỉ thị TSL (tt)

### • Ví dụ:

```

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 not 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
  
```

Entering and leaving a critical region using the TSL instruction.

85

85

## Chỉ thị TSL (tt)

### • Nhận xét về TSL:

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

### • Nhận xét chung về các giải pháp Busy waiting

- Sử dụng CPU không hiệu quả
- Liên tục kiểm tra điều kiện khi chờ vào CS
- **Khắc phục:** khoá các tiến trình chưa đủ điều kiện vào CS, nhường CPU cho tiến trình khác → giải pháp Sleep & Wake up

86

86

## Các giải pháp “Sleep & Wake up”

- Các tiến trình phải từ bỏ CPU khi chưa được vào CS
- Khi CS trống, sẽ được đánh thức để vào CS
- Cần phải sử dụng các thủ tục do hệ điều hành cung cấp để thay đổi trạng thái tiến trình

if (chưa có quyền)  
Sleep() ;



CS:

WakeUp (somebody):



87

87

## Sleep & Wake up (tt)

- Hệ Điều hành hỗ trợ 2 hàm đặc quyền:
  - Sleep() : Tiến trình gọi hàm này sẽ chuyển sang trạng thái Blocked
  - WakeUp (P): Tiến trình P nhận trạng thái Ready
- Khi một tiến trình chưa đủ điều kiện vào CS → gọi Sleep () → chuyển sang trạng thái bị khóa cho đến khi có một tiến trình khác gọi WakeUp để giải phóng cho nó.
- Một tiến trình khi ra khỏi CS sẽ gọi WakeUp để đánh thức một tiến trình khác đang bị khóa, để tiến trình này vào CS

88

88

## Sleep & Wake up (tt)

```

int busy; //busy = 0: CS trống
int blocked; //số tiến trình bị Blocked chờ vào CS
while (TRUE) {
    if (busy){
        blocked = blocked + 1;
        sleep();
    }
    else busy = 1;
    critical-section ();
    busy = 0;
    if(blocked){
        wakeup(process);
        blocked = blocked - 1;
    }
    Noncritical-section ();
}

```

89

89

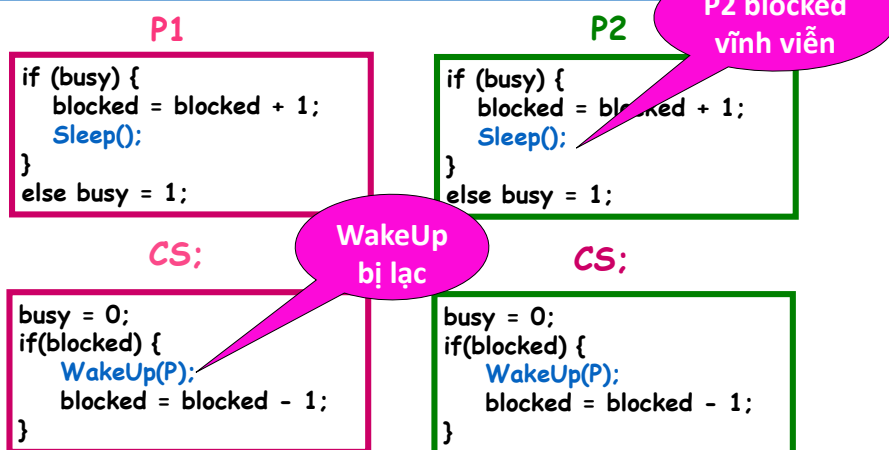
## Sleep & Wake up (tt)

- Có thể xảy ra mâu thuẫn truy xuất
  - Giả sử P1 vào CS, trước khi P1 ra khỏi CS P2 được kích hoạt.
  - P2 thử vào CS nhưng nhận thấy P1 đang ở trong CS → P2 tăng giá trị biến *blocked* và **sẽ** gọi *Sleep* để tự khóa.
  - Trước khi P2 gọi *Sleep*, P1 lại được tái kích hoạt và ra khỏi CS. Khi ra khỏi CS, P1 thấy có một tiến trình đang chờ (*blocked=1*) nên gọi *WakeUp* và giảm giá trị của *blocked* (*blocked=0*)
  - Do P2 chưa thực hiện *Sleep* nên không thể nhận tín hiệu *WakeUp*.
  - Khi P2 được tiếp tục xử lý, nó mới gọi *Sleep* và tự khóa vĩnh viễn

90

90

## Vấn đề với Sleep & WakeUp



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

91

## Mutex locks

- Là giải pháp đơn giản được sử dụng để ngăn chặn race conditions
- Mutex là một biến có hai giá trị: khóa/không khóa
- Hai hàm đặc quyền được hỗ trợ từ phần cứng:
  - mutex-lock: tiến trình muốn vào CS
  - mutex-unlock: tiến trình ra khỏi CS, mở khóa cho các tiến trình khác có thể vào CS
- Một tiến trình muốn vào CS, gọi mutex-lock và trước khi rời khỏi CS gọi mutex-unlock

92

92

## Mutex locks (tt)

```
mutex_lock() {
    while (!available); /* busy wait */
    available = false;
}
mutex_unlock() {
    available = true;
}
do {
    mutex_lock ();
    Critical_section
    mutex_unlock ();
    Noncritical_section
} while (true);
```

93

93

## Semaphore

- Được Dijkstra đề xuất vào 1965
- Một semaphore s là cấu trúc dữ liệu gồm có:
  - Giá trị nguyên dương value
  - Hàng đợi L: danh sách các tiến trình đang bị khóa trên semaphore s
- Hai phương thức:
  - Down(value):
    - nếu value > 0: value = value -1
    - Ngược lại (value = 0): tiến trình phải chờ đến khi value > 0.
  - Up(value):
    - value = value + 1
    - Nếu có một hoặc nhiều tiến trình đang chờ trên semaphore s (count (L) >0) → hệ thống sẽ chọn một trong các tiến trình trong hàng đợi này cho tiếp tục xử lý.

94

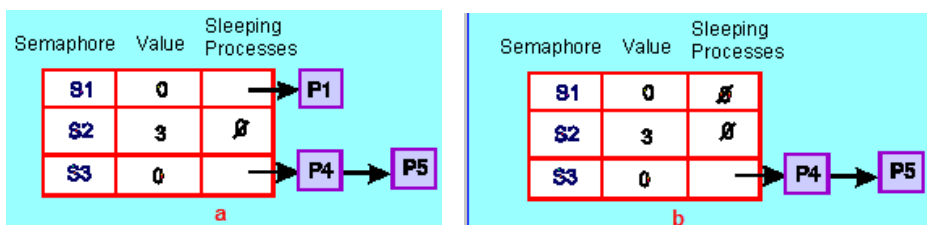
94

## Semaphore

### • hình a:

- s1: P1 bị khóa
- s2: value = 3 → 3 tiến trình có thể thực thi lệnh down và không bị khóa
- s3: P4, P5 bị khóa

### • hình b: P1 được kích hoạt (bởi lệnh Up) → nó thực thi lệnh down trên S1 và vào CS.



95

95

## Cài đặt Semaphore (Sleep & Wakeup)

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

Giá trị bên trong của semaphore

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

### • Semaphore được xem như là một resource

- 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.list

### • Cần có sự hỗ trợ của HĐH: Sleep() & Wakeup()

96

96



## Cài đặt Semaphore (Sleep & Wakeup)

### Down (S)

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

### Up(S)

```
{
  S.value ++;
  if count(S.list) > 0
  {
    Remove(P, S.list);
    Wakeup(P);
  }
}
```

97

97

## Sử dụng Semaphore

- Truy xuất độc quyền với Semaphore
  - Cho phép bảo đảm nhiều tiến trình cùng truy xuất đến CS mà không có sự mâu thuẫn truy xuất.
  - $n$  tiến trình cùng sử dụng một semaphore  $s$ , value được khởi gán là 1. Tất cả các tiến trình cần phải áp dụng cùng cấu trúc chương trình sau:

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

98

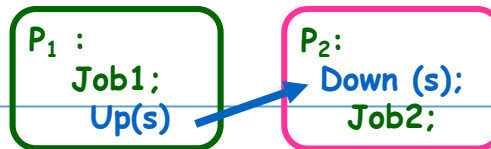
98

## Sử dụng Semaphore

- Tổ chức đồng bộ hóa với Semaphores
  - 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 value = 0.

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

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



99

99

## Nhận xét Semaphores

- Là một cơ chế tốt để thực hiện đồng bộ
  - Dễ dùng cho N tiến trình
- Khó sử dụng đúng: nếu thiếu hoặc đặt sai vị trí down và up

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

100

100

## Ví dụ về Semaphores

### The Producer-Consumer Problem



Buffer



Producer

- It has to stop when buffer is full.
- If producer is producing at that time won't allow consumer to consume data.



Consumer

- It has to stop when buffer is empty.
- If consumer is consuming that time won't allow producer to produce.

101

101

## Semaphores - The Producer-Consumer Problem (tt)

### • Sử dụng giải pháp sleep -wakeup

```
#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? */
    }
}
```

102

102

## Semaphores - The Producer-Consumer Problem (tt)

```

void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep();           /* repeat forever */
        item = remove_item();             /* if buffer is empty, got to sleep */
        count = count - 1;                 /* take item out of buffer */
        if (count == N - 1) wakeup(producer); /* decrement count of items in buffer */
        consume_item(item);               /* was buffer full? */
    }                                     /* print item */
}

```

103

103

## Semaphores - The Producer-Consumer Problem (tt)

- Vấn đề:
  - Ban đầu count= 0:
    - C(consumer) thấy count = 0 → chuẩn bị sleep, nhưng bị block.
    - P(producer): count = count + 1, gọi wakeup để đánh thức C.
    - Tuy nhiên, có thể C vẫn chưa sleep một cách hợp lý, vì vậy tín hiệu wakeup bị mất. Khi C tiếp tục chạy → kiểm tra giá trị biến count đã đọc trước đó: count= 0 → sleep.
    - Khi P lấp đầy bộ đệm (count=N) → sleep.
    - Cả P và C đều sleep.

104

104

## Semaphores - The Producer-Consumer Problem (tt)

### •Giải pháp dùng Semaphore:

```

#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 */
    }
}

```

105

105

## Semaphores - The Producer-Consumer Problem (tt)

### •Giải pháp dùng Semaphore (tt):

```

void consumer(void)
{
    int item;

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

```

106

106

# 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 đồng bộ hóa
- 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 đóng gói
- Các tiến trình có thể gọi các phương thức trong monitor nhưng không thể truy cập trực tiếp vào các cấu trúc dữ liệu bên trong của monitor

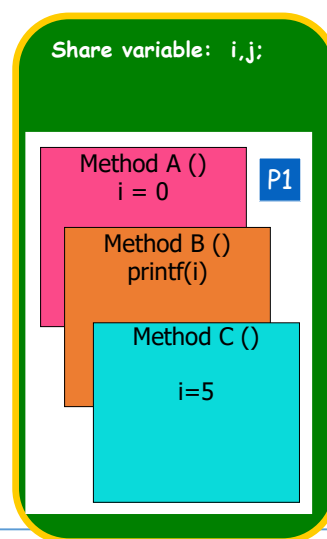
107

107

## Monitor (tt)

- Tại một thời điểm, chỉ có một tiến trình duy nhất được hoạt động bên trong một monitor
- Các tiến trình không thể vào monitor → đưa vào hàng đợi
- Ví dụ
  - P1 : M.A();
  - P6 : M.B();
  - P7 : M.A();
  - P8 : M.C();

Entry queue P6 → P7 → P8

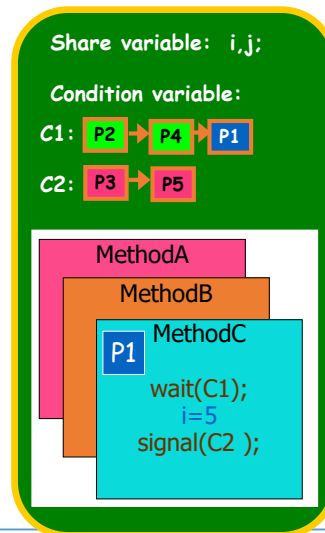


108

108

## Monitor (tt)

- Hỗ trợ đồng bộ hóa với các biến điều kiện
  - Wait (c): Tiến trình gọi hàm sẽ bị blocked → hàng đợi trên biến c
  - 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
- 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



109

109

## Monitor (tt)

- Cài đặt:

```
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);
}
```

110

110

## Monitor (tt)

- 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:

```

monitor <tên monitor >
condition <danh sách các biến điều kiện>;
<déclaration de variables>;
procedure Action_1(){
}
....
procedure Action_n() {
}
end monitor;

```

111

111

## Monitor (tt)

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

```

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

```

112

112



## Monitor (tt)

### • Nhận xét:

- Đảm bảo truy xuất độc quyền bởi trình biên dịch mà không do lập trình viên → nguy cơ thực hiện đồng bộ hóa sai giảm rất nhiều.
- Đòi hỏi phải có một ngôn ngữ lập trình định nghĩa khái niệm monitor.

113

113

## Message

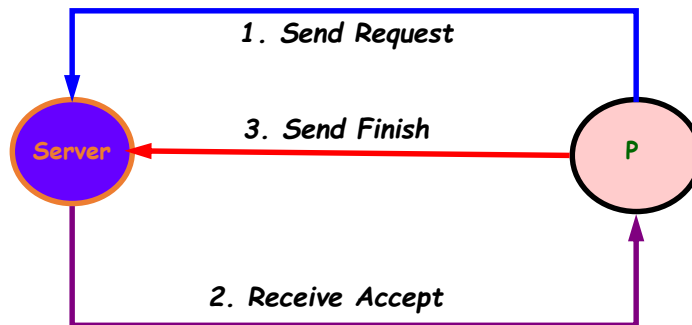
- Đồng bộ hóa trên môi trường phân tán
- Được hỗ trợ bởi HĐH
- Một tiến trình kiểm soát việc sử dụng tài nguyên và nhiều tiến trình khác yêu cầu tài nguyên.
- Tiến trình có yêu cầu tài nguyên:
  - Gửi một message đến tiến trình kiểm soát
  - → chuyển sang trạng thái blocked cho đến khi nhận được thông điệp đánh thức từ tiến trình kiểm soát
  - Khi sử dụng xong tài nguyên → gửi một thông điệp khác đến tiến trình kiểm soát để báo kết thúc truy xuất.
- Tiến trình kiểm soát:
  - Nhận được message yêu cầu tài nguyên
  - Khi tài nguyên sẵn sàng → gửi một thông điệp đến tiến trình đang bị khóa trên tài nguyên đó để đánh thức tiến trình này.

114

114

## Message (tt)

- Trao đổi thông điệp với hai primitive Send và Receive để thực hiện sự đồng bộ hóa:



115

115

## Message (tt)

```

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

116

116

## Các bài toán đồng bộ hoá kinh điển

---

- Producer – Consumer
- Readers – Writers
- Dining Philosophers

117

117

## 2.4. Điều phối tiến trình (Scheduling)

---

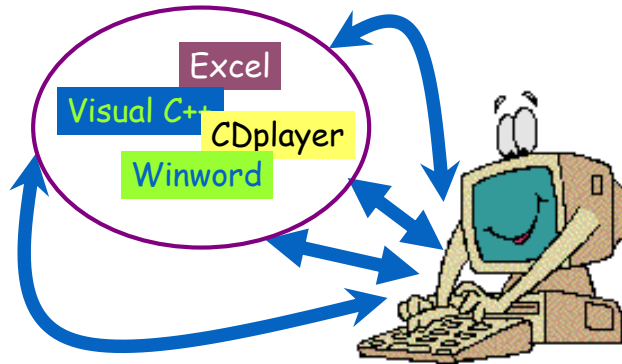
- Mục tiêu điều phối
- Các cấp điều phối
- Các giải thuật điều phối (cấp điều phối thời gian ngắn)
- Vấn đề điều phối luồng

118

118

## 2.4.1 Mục tiêu điều phối

- Môi trường đa chương

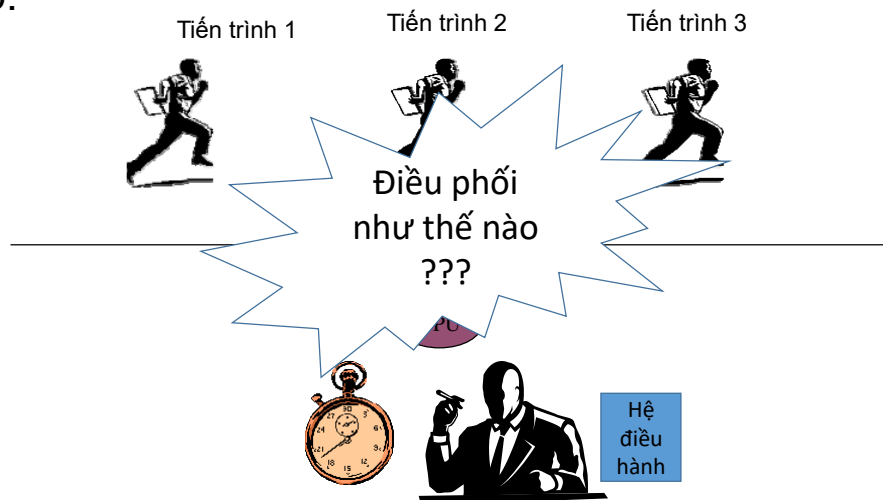


119

119

## Mục tiêu điều phối (tt)

- Giải pháp:



120

120

## Mục tiêu điều phối (tt)

### • Một số khái niệm:

- CPU utilization (% sử dụng CPU, Độ lợi CPU)
- Throughput: thông lượng tối đa
- Turnaround-time (Thời gian quay vòng – hoàn thành)
- Response time (Thời gian đáp ứng)
- Waiting time (Thời gian chờ)
- Average turn-around time (Thời gian quay vòng trung bình)

121

121

## Mục tiêu điều phối (tt)

### • Mục tiêu chung

- Công bằng sử dụng CPU, tận dụng CPU tối đa
- Cân bằng sử dụng các thành phần của hệ thống

### • Hệ thống theo lô

- Tối ưu thông lượng tối đa (throughput)
- Giảm thiểu turnaround time: T<sub>quit</sub> – T<sub>arrive</sub>
- Tận dụng CPU

### • Hệ thống tương tác

- Đáp ứng nhanh: giảm thiểu thời gian chờ
- Cân đối mong muốn của người dùng

### • Hệ thống thời gian thực

- Tránh mất dữ liệu
- Đảm bảo chất lượng trong các hệ thống đa phương tiện

122

122

## Mục tiêu điều phối (tt)

### •Tiêu chí lựa chọn tiến trình

- Chọn tiến trình vào RQ trước
- Chọn tiến trình có độ ưu tiên cao hơn

### •Thời điểm lựa chọn tiến trình

- Điều phối độc quyền (non-preemptive scheduling): tiến trình đang ở trạng thái Running sẽ tiếp tục sử dụng CPU cho đến khi kết thúc hoặc bị block vì I/O hay các dịch vụ của hệ thống (độc chiếm CPU)
- Điều phối không độc quyền (preemptive scheduling): ngoài thời điểm lựa chọn như điều phối độc quyền, tiến trình đang sử dụng CPU có thể bị ngắt (chuyển sang trạng thái Ready) khi hết thời gian qui định hoặc có tiến trình có độ ưu tiên hơn vào ReadyQueue

123

123

## 2.4.2 Các cấp điều phối

### •Điều phối tác vụ (job scheduling)

- Chọn tác vụ và nạp những tiến trình của tác vụ đó vào bộ nhớ chính để thực hiện
- Quyết định mức độ đa chương của hệ thống
- Chức năng điều phối tác vụ được kích hoạt khi tiến trình được tạo hoặc kết thúc
- Bộ điều phối tác vụ cần chọn luân phiên một cách hợp lý giữa tiến trình hướng nhập xuất (I/O bounded) và các tiến trình hướng xử lý (CPU bounded) → cân bằng hoạt động của CPU và các thiết bị ngoại vi

124

124

## Các cấp điều phối (tt)

---

- **Điều phối tiến trình** (process scheduling)

- Chọn một tiến trình ở trạng thái sẵn sàng để cấp phát CPU
- Tần suất hoạt động cao (~100ms), xảy ra khi có ngắt (đồng hồ, thiết bị ngoại vi,...) → cần tăng tốc độ của bộ điều phối tiến trình
- Một số HĐH không có bộ điều phối tác vụ, một số HĐH kết hợp cả hai cấp độ điều phối (cấp độ điều phối trung gian)

125

125

## 2.4.3 Chiến lược điều phối

---

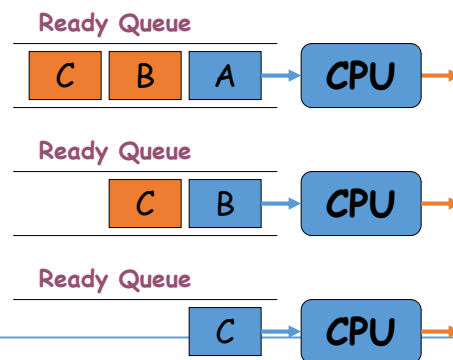
- FCFS
- SJF
- Round Robin
- Priority
- HRRN
- Multiple Queues

126

126

### 2.4.3.1 First-Come First-Served (FCFS)

- Được sử dụng trong hệ thống xử lý theo lô (Scheduling in Batch Systems)
- Tiêu chí lựa chọn tiến trình
  - Thứ tự vào hàng đợi Ready Queue
- Thời điểm lựa chọn tiến trình
  - Độc quyền



127

127

### FCFS (tt)

- Ví dụ:

P	Thời gian đến	Thời gian sử dụng CPU
P1	0	24
P2	1	3
P3	2	3

- Biểu đồ Gantt:

Thời gian đáp ứng:

$$P1: 0 - 0 = 0$$

$$P2: 24 - 1 = 23$$

$$P3: 27 - 2 = 25$$

$$\text{Avg} = (23 + 25) / 3 = 16$$

Thời gian hoàn thành:

$$P1: 24 - 0 = 24$$

$$P2: 27 - 1 = 26$$

$$P3: 30 - 2 = 28$$

$$\text{Avg} = (24 + 26 + 28) / 3 = 26.3$$

Thời gian chờ:

$$P1: 24 - 0 - 24 = 0$$

$$P2: 27 - 1 - 3 = 23$$

$$P3: 30 - 2 - 3 = 25$$

$$\text{Avg} = (23 + 25) / 3 = 16$$

P1	P2	P3
0	24	27 30

128

128



## FCFS (tt)

- Đơn giản, dễ cài đặt
- Chịu đựng hiện tượng tích lũy thời gian chờ
  - Tiến trình có thời gian xử lý ngắn đợi tiến trình có thời gian xử lý dài
  - Ưu tiên tiến trình cpu-bounded
- Có thể xảy ra tình trạng độc chiếm CPU
- Không phù hợp với các hệ phân chia thời gian

129

129

## 2.4.3.2 Điều phối với độ ưu tiên (Priority)

- Một độ ưu tiên được gán vào mỗi tiến trình
- Phân biệt tiến trình quan trọng với tiến trình bình thường
- Tiêu chí lựa chọn tiến trình
  - Tiến trình có độ ưu tiên cao nhất
- Thời điểm lựa chọn tiến trình
  - Độc quyền
  - Không độc quyền (có độ ưu tiên)

130

130

## Điều phối với độ ưu tiên (tt)

### • Ví dụ:

P	Thời gian đến	Độ ưu tiên	Thời gian sử dụng CPU
P1	0	3	24
P2	1	1	3
P3	2	2	3

### • Độ ưu tiên độc quyền

P1	P2	P3
0	24	27 30

### • Độ ưu tiên không độc quyền

P1	P2	P3	P1
0	1	4	7 30

131

131

## Điều phối với độ ưu tiên (tt)

### • Nhận xét:

- Có thể xảy ra tình trạng Starvation (đói CPU): các tiến trình độ ưu tiên thấp có thể không bao giờ thực thi được
- → Giải pháp Aging – tăng độ ưu tiên cho tiến trình chờ lâu trong hệ thống
- Gán độ ưu tiên còn dựa vào:
  - Yêu cầu về bộ nhớ
  - Số lượng file được mở
  - Tỷ lệ thời gian dùng cho I/O trên thời gian sử dụng CPU
  - Các yêu cầu bên ngoài

132

132

### 2.4.3.3 Shortest Job First (SJF)

- Là một dạng đặc biệt của giải thuật điều phối với độ ưu tiên
- Tiêu chí lựa chọn tiến trình
  - Chọn tiến trình có thời gian xử lý ngắn nhất
- Thời điểm lựa chọn tiến trình
  - Độc quyền (non-preemptive)
  - Không độc quyền (Shortest-Remaining-Time-First - SRTF)

$$p_i = \text{thời\_gian\_còn\_lại (Process}_i\text{)}$$

133

133

### Shortest Job First (tt)

- Ví dụ

P	Thời gian đến	Thời gian sử dụng CPU
P1	0	6
P2	1	8
P3	2	4
P4	3	2

- Thời điểm lựa chọn độc quyền

P1	P4	P3	P2	
0	6	8	12	20

- Thời điểm lựa chọn không độc quyền (SRTF)

P1	P4	P1	P3	P2	
0	3	5	8	12	20

134

134

## Shortest Job First (tt)

### • Nhận xét:

- Tối ưu thời gian chờ
- Cơ chế không độc quyền có thể xảy ra tình trạng “đói” (starvation) đối với các process có CPU-burst lớn khi có nhiều process với CPU-burst nhỏ đến hệ thống
- Cơ chế độc quyền không phù hợp cho hệ thống chia sẻ thời gian
- Cần xác định chiều dài thời gian sử dụng CPU cho lần tiếp theo của mỗi tiến trình:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

- $t_n$  là độ dài của thời gian xử lý lần thứ  $n$
- $\tau_{n+1}$  là giá trị dự đoán cho lần xử lý tiếp theo
- $0 \leq \alpha \leq 1$

135

135

## Shortest Job First (tt)

### • $\alpha = 0$

- $\tau_{n+1} = \tau_n$
- Không xét các giá trị thực đã dùng CPU.

### • $\alpha = 1$

- $\tau_{n+1} = t_n$
- Chỉ dùng giá trị thực mới dùng CPU.

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

### • Nếu mở rộng công thức, ta được:

- $\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots$
- $\quad \quad \quad + (1 - \alpha)^j \alpha t_{n-j} + \dots$
- $\quad \quad \quad + (1 - \alpha)^{n-1} t_1 + \tau_0$

- Vì  $\alpha$  và  $(1 - \alpha)$  nhỏ hơn hay bằng 1, mỗi giá trị kế tiếp sẽ nhỏ dần.

136

## 2.4.3.4 Round Robin (RR) Điều phối xoay vòng

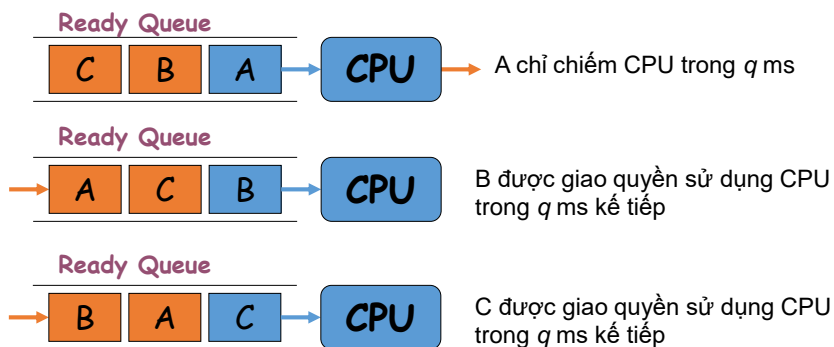
- Mỗi tiến trình chỉ sử dụng một lượng  $q$  cho mỗi lần sử dụng CPU
- Tiêu chí lựa chọn tiến trình
  - Thứ tự vào hàng đợi Ready Queue
- Thời điểm lựa chọn tiến trình
  - Không độc quyền (không có độ ưu tiên)

Quantum/  
Time slice

137

137

## Round Robin (tt)



138

138

## Round Robin (tt)

- Ví dụ:
  - RR ( $q=4$ )

P	Thời gian đến	Thời gian sử dụng CPU
P1	0	24
P2	1	3
P3	2	3

- Biểu đồ Gantt:

P1	P2	P3	P1	P1	P1	P1	P1
0	4	7	10	14	18	22	26 30

139

139

## Round Robin (tt)

- Nhận xét:
  - Loại bỏ hiện tượng độc chiếm CPU
  - Phù hợp với hệ thống tương tác người dùng
  - Thời gian chờ đợi trung bình của giải thuật RR thường khá lớn nhưng thời gian đáp ứng nhỏ
  - Hiệu quả phụ thuộc vào việc lựa chọn quantum  $q$ 
    - $q$  quá lớn  $\Rightarrow$  FCFS (giảm tính tương tác)
    - $q$  quá nhỏ  $\Rightarrow$  chủ yếu thực hiện chuyển đổi ngữ cảnh (context switching)
    - Thường  $q = 10 - 100$  milliseconds

140

140

### 2.4.3.5 Highest Response Ratio Next (HRRN)

- Cải tiến giải thuật SJF
- Định thời theo chế độ độc quyền
- Độ ưu tiên được tính theo công thức:

$$p = (tw + ts)/ts$$

- tw: thời gian chờ (waiting time)
- ts: thời gian sử dụng CPU (service time)
- Tiêu chí chọn: chọn p lớn nhất → Ưu tiên cho các tiến trình có thời gian sử dụng CPU ngắn và các tiến trình chờ lâu.
- p được tính lại khi có tiến trình kết thúc

141

141

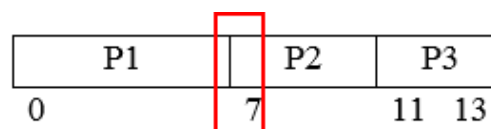
### Highest Response Ratio Next (tt)

- Ví dụ:

P	Thời gian đến	Thời gian sử dụng CPU
P1	0	7
P2	1	4
P3	5	2

- Độ ưu tiên tại thời điểm (7):

- P2:  $(6+4)/4 = 2.5$
- P3:  $(2+2)/2 = 2$
- → chọn P2

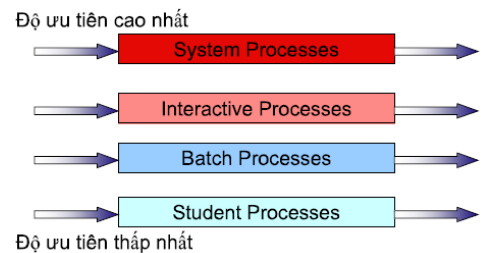


142

142

### 2.4.3.6 Multilevel Queue Scheduling

- Sử dụng nhiều hàng đợi ready (ví dụ foreground, background)
- Mỗi hàng đợi chứa các tiến trình có cùng độ ưu tiên và có giải thuật điều phối riêng, ví dụ:
  - foreground: RR
  - background: FCFS
- HĐH phải định thời cho các hàng đợi
- Ví dụ:



143

143

### Multilevel Queue Scheduling (tt)

- Fixed priority scheduling:
  - Chọn tiến trình trong hàng đợi có độ ưu tiên từ cao đến thấp.
  - Có thể có starvation.
- Time slice:
  - Mỗi hàng đợi được nhận một khoảng thời gian chiếm CPU và phân phối cho các process trong hàng đợi khoảng thời gian đó.
  - Ví dụ: 80% cho hàng đợi foreground định thời bằng RR và 20% cho hàng đợi background định thời bằng giải thuật FCFS

144

144



### 2.4.3.7 Multilevel Feedback Queue

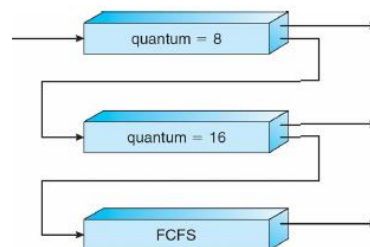
- Trong hệ thống Multilevel Feedback Queue, bộ định thời có thể di chuyển tiến trình giữa các queue tùy theo đặc tính của nó được quan sát. Ví dụ:
  - Tiến trình sử dụng CPU quá lâu → chuyển sang một hàng đợi có độ ưu tiên thấp hơn
  - Tiến trình chờ qua lâu trong một hàng đợi có độ ưu tiên thấp → chuyển lên hàng đợi có độ ưu tiên cao hơn (*aging*, giúp tránh starvation)

145

145

### Multilevel Feedback Queue (tt)

- Ví dụ: Có 3 hàng đợi
  - Q0, dùng RR với quantum 8ms
  - Q1, dùng RR với quantum 16ms
  - Q2, dùng FCFS
- Giải thuật
  - Tiến trình mới sẽ vào hàng đợi Q0. Khi đến lượt, sẽ được cấp phát quantum là 8 ms, nếu chưa xử lý xong → chuyển xuống cuối hàng đợi Q1
  - Tại Q1, tiến trình thực thi sẽ được cấp phát quantum là 16ms, nếu chưa xử lý xong → chuyển xuống cuối hàng đợi Q2



146

146

## Bài tập

### • Điều phối:

- FCFS
- Priority
  - Thời điểm lựa chọn độc quyền
  - Thời điểm lựa chọn không độc quyền
- SJF
  - Thời điểm lựa chọn độc quyền
  - Thời điểm lựa chọn không độc quyền (SRTF)
- RR ( $q=4$ )
- HRRN

Process	Arrival time	Service time	Priority
P1	0	10	4
P2	2	24	3
P3	4	5	5
P4	5	8	1
p5	6	12	2

### • Vẽ biểu đồ Gantt

- Tính thời gian đáp ứng, thời gian hoàn thành, thời gian chờ trung bình

147

147

## Bài tập

- Giả sử các tiến trình trong hệ thống với thời điểm đến, độ ưu tiên và thời gian sử dụng CPU được mô tả trong các bảng
- Sử dụng chiến lược điều phối FCFS, RR (với  $q=2$ ), Priority, SJF (độc quyền/không độc quyền(SRTF)), HRRN
- Tính thời gian chờ, thời gian đáp ứng, thời gian hoàn thành.

Process	Arrival time	Service time	Priority
P1	0	7	2
P2	2	4	1
P3	4	1	4
P4	5	4	3

148

148

## 2.4. Vấn đề điều phối luồng

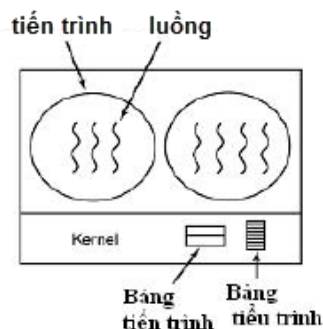
- Các luồng liên lạc với nhau thông qua các biến toàn cục của tiến trình
- Cơ chế điều phối luồng phụ thuộc vào cách cài đặt luồng:
  - Cài đặt trong kernel-space
  - Cài đặt trong user-space
  - Cài đặt trong kernel-space và user-space

149

149

## Vấn đề điều phối luồng (tt)

- Cài đặt trong kernel-space:
  - Bảng quản lý thread lưu ở phần kernel-space
  - HĐH chịu trách nhiệm điều phối thread

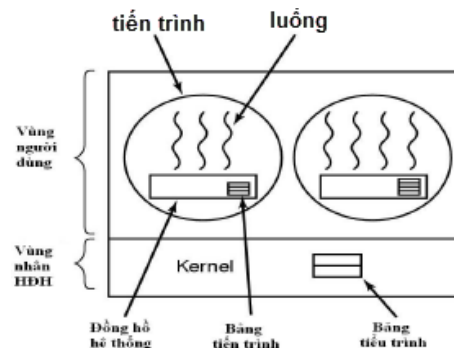


150

150

## Vấn đề điều phối luồng (tt)

- Cài đặt trong user-space
  - Bảng quản lý thread lưu ở phần user-space
  - Tiến trình chịu trách nhiệm điều phối thread



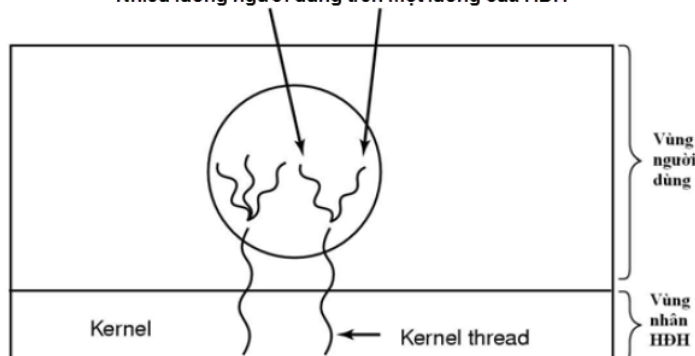
151

151

## Vấn đề điều phối luồng (tt)

- Cài đặt trong kernel-space và user-space
  - Một số luồng mức user được cài đặt bằng một luồng mức kernel
  - Một luồng của HĐH quản lý một số luồng của tiến trình

Nhiều luồng người dùng trên một luồng của HĐH



152

152