HỆ ĐIỀU HÀNH

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

Chương 2

Quản lý tiến trình

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

Tài liệu tham khảo

- Andrew S. Tanenbaum, Modern Operating Systems, Pearson, 2015
  - Chương 2
- Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating System Concepts, John Wiley, 2013
  - Chương 3, 4, 5, 6
- William Stallings,Operating Systems: Internals and Design Principles, Pearson, 2015
  - Chương 3, 4

5

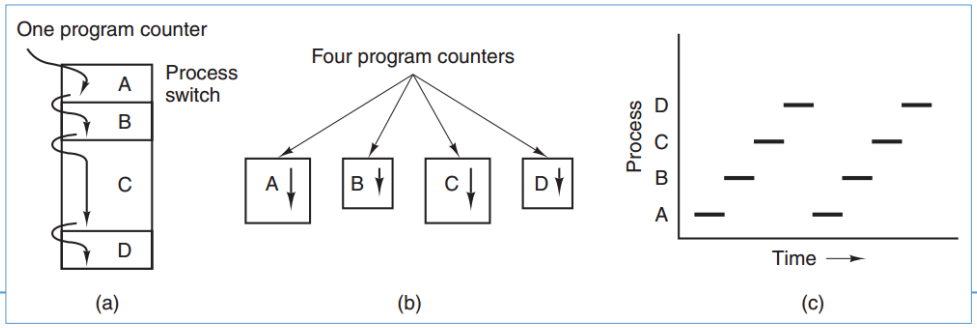
2.1. Tiến trình

- Mô hình
- Hiện thực

6

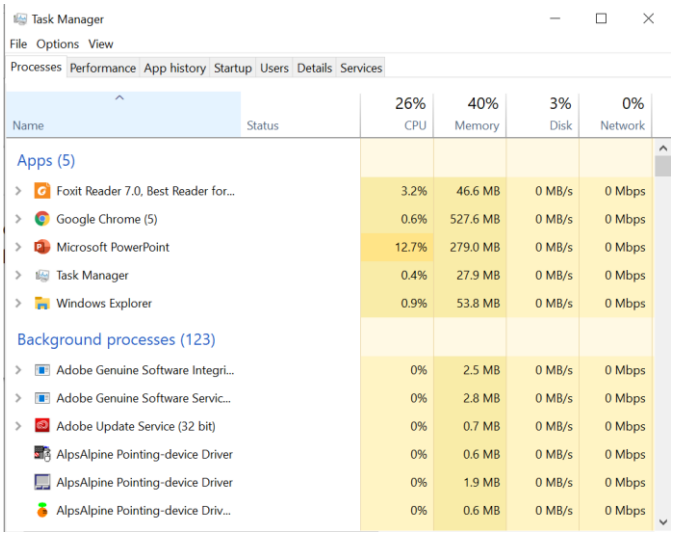
2.1.1 Mô hình Tiến trình

- Chương trình phần mềm khi thực thi phải được nạp vào bộ nhớ chính
- CPU đọc các chỉ thị từ bộ nhớ chính để xử lý
- 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)**



Mô hình Tiến trình (tt)

- 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
  - ...



Mô hình Tiến trình (tt)

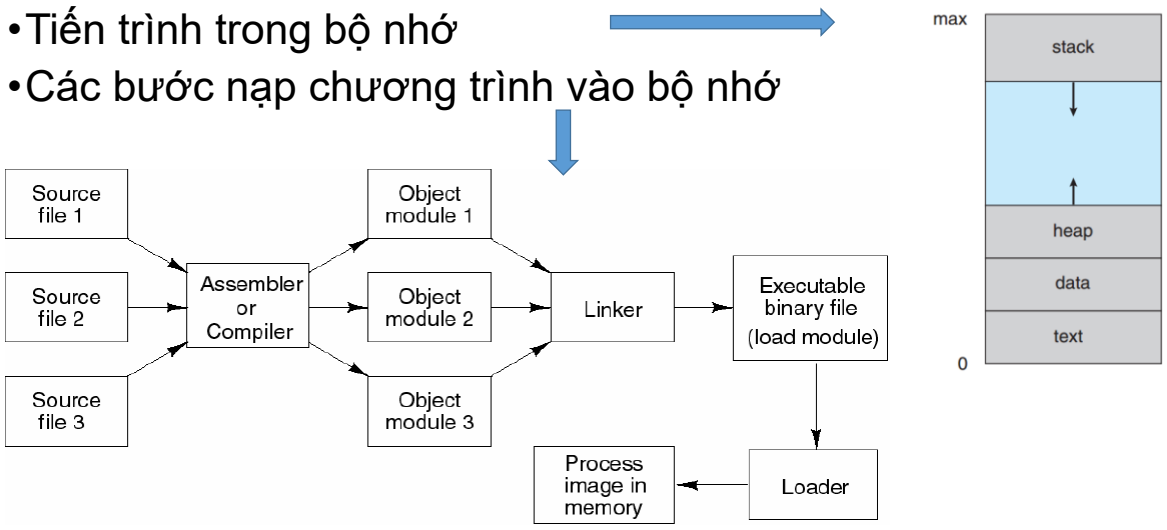
- Một tiến trình bao gồm:
  - Text section (program code), data section (chứa global variables)
  - program counter (PC), process status word (PSW), stack pointer (SP), memory management registers,…
- Phân biệt chương trình và tiến trình:
  - Chương trình: là một thực thể thụ động (tập tin có chứa một 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.

9

9

Mô hình Tiến trình (tt)

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



10

10

2.1.2 Hiện thực Tiến trình

- Vai trò của hệ điều hành trong quản lý tiến trình
- Tạo tiến trình
- Các trạng thái của tiến trình
- Các thao tác với tiến trình
- Thực thi các tiến trình
- PCB – Process Control Block

Vai trò của hệ điều hành trong quản lý tiến trình

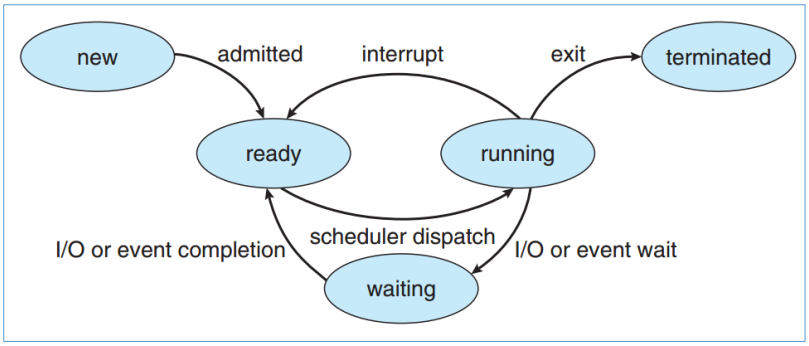
- Tạo và hủy tiến trình của người dùng và tiến trình hệ thống
- Dừng, khôi phục (resume) tiến trình
- Cung cấp các cơ chế để đồng bộ hóa tiến trình
- Cung cấp các cơ chế liên lạc giữa các tiến trình
- Cung cấp cơ chế xử lý bế tắc (deadlock)

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,…)

Các trạng thái của tiến trình

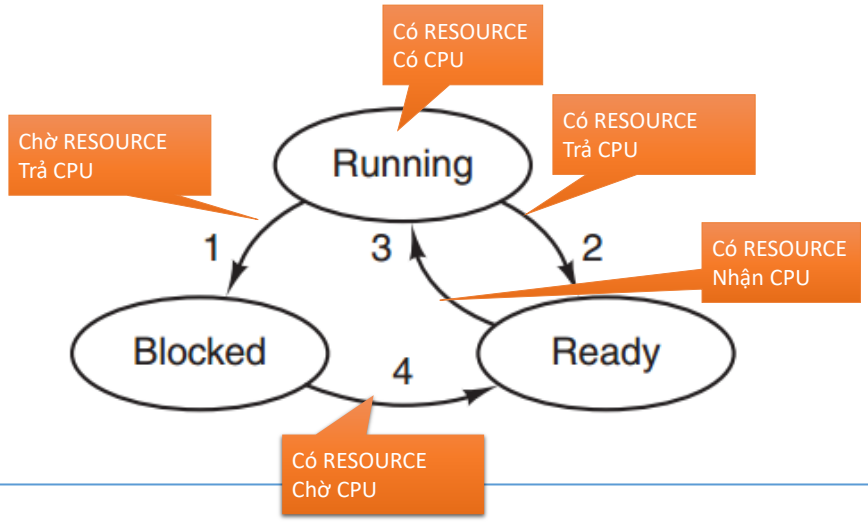
- New: tiến trình đang được tạo lập.
- Ready: tiến trình chờ được cấp phát CPU.
- Running: các chỉ thị của tiến trình đang được xử lý.
- Waiting: tiến trình chờ được cấp phát một tài nguyên, hay chờ một sự kiện xảy ra.
- Terminated: tiến trình kết thúc xử lý.



Ex: Operating Systems: Internals and Design Principles, **Eighth Edition page 142**

Các trạng thái của tiến trình (tt)

- Quá trình chuyển đổi trạng thái của tiến trình



15

15

Các trạng thái của tiến trình (tt)

- Các trạng thái có thể chuyển đổi của tiến trình
  - Null → New
  - New → Ready
  - Ready → Running
  - Running → Exit
  - Running → Ready
  - Running → Blocked
  - Blocked → Ready
  - Ready → Exit
  - Blocked → Exit

16

16

Các trạng thái của tiến trình (tt)

- Ví dụ về quá trình chuyển đổi trạng thái của tiến trình
  - Cho đoạn chương trình C trên Linux:
  - 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

17

17

Các thao tác với tiến trình

- Tiến trình được tạo khi:
  - Hệ thống khởi động
  - Một tiến trình đang chạy thực hiện lời gọi hệ thống tạo tiến trình mới.
  - Người dùng tạo
  - Bắt đầu một batch job

18

18

Các thao tác với tiến trình (tt)

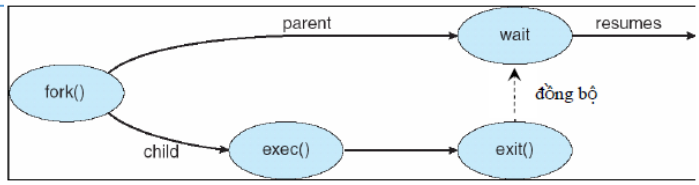
- Tiến trình mới:
  - Nhận tài nguyên từ HĐH hoặc từ tiến trình cha
  - Tài nguyên: nếu tiến trình con được tạo từ tiến trình cha:
    - Tiến trình cha và con chia sẻ mọi tài nguyên
    - Tiến trình con chỉ chia sẻ một phần tài nguyên của cha
  - Trình tự thực thi:
    - Tiến trình cha và con thực thi 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 (tt)

```
#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 (tt)

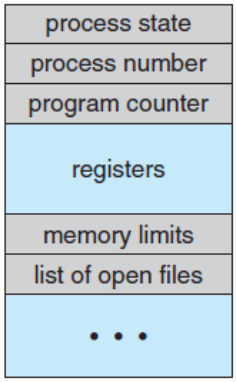
- Kết thúc tiến trình
  - Tiến trình kết thúc trong các trường hợp:
    - Tiến trình tự kết thúc bình thường: tiến trình kết thúc khi thực thi lệnh cuối và gọi exit
    - Tiến trình bị lỗi và tự kết thúc (do trình biên dịch xử lý)
    - Tiến trình bị lỗi và buộc phải kết thúc do phần cứng (ví dụ thực hiện một lệnh không hợp lệ, tham chiếu đến bộ nhớ không tồn tại, lỗi chia cho không, lỗi I/O,...) → phát sinh một ngắt (interrupted)
    - Tiến trình kết thúc do tiến trình khác (ví dụ tiến trình cha)
      - Tiến trình cha kết thúc → tiến trình con cũng kết thúc
      - Tiến trình cha gọi abort với tham số là pid của tiến trình cần được kết thúc
    - Do người dùng
  - Khi tiến trình kết thúc, HĐH thu hồi tất cả các tài nguyên của tiến trình

Thực thi các tiến trình

- Để thực thi được mô hình tiến trình, HĐH duy trì một bảng gọi là bảng tiến trình.
- Mỗi phần tử trong bảng là một khối quản lý tiến trình (Process Control Block - PCB)
- Mỗi PCB quản lý một tiến trình
- Mỗi tiến trình khi được tạo sẽ được cấp phát một (PCB)

PCB – Process Control Block

- Là một cấu trúc dữ liệu lưu các thông tin:
  - Định danh (Process Number)
  - Trạng thái tiến trình (Process State)
  - Bộ đếm chương trình (Program counter)
  - Các thanh ghi (CPU registers)
  - 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

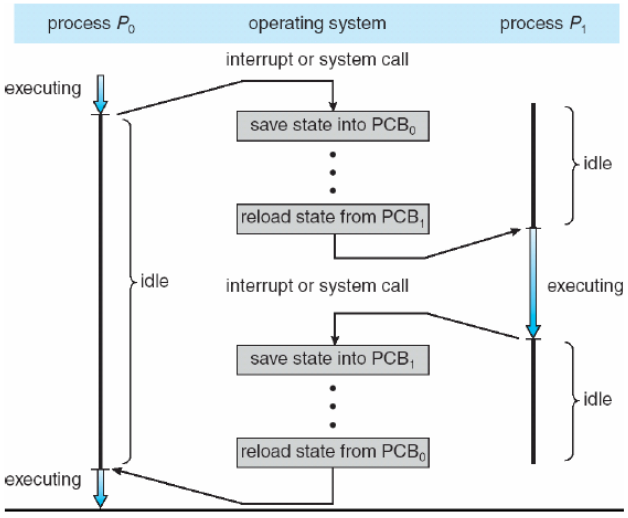


Process control block (PCB).

23

PCB (tt)

- Sơ đồ chuyển CPU giữa các tiến trình



24

2.2. Luồng (thread)

- Mô hình
- Hiện thực

25

25

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 có một không gian địa chỉ và một dòng xử lý duy nhất → 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

26

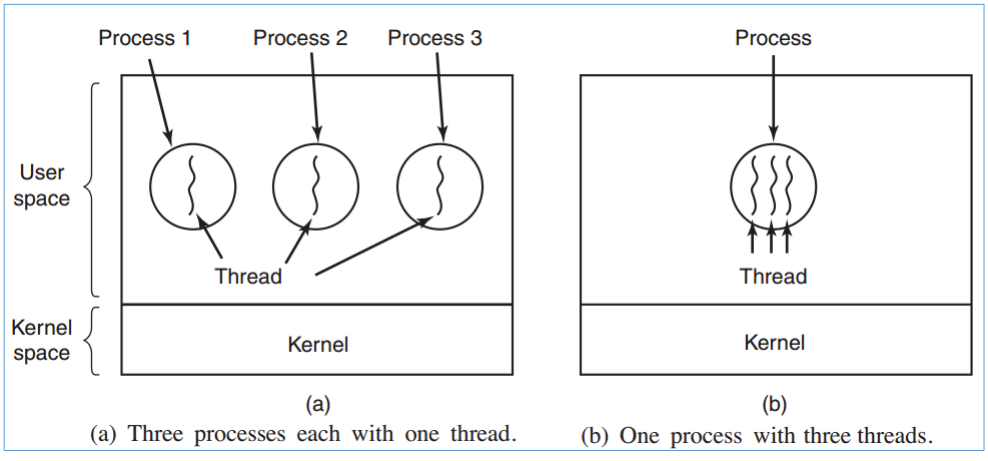
26

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

Mô hình luồng (tt)

- Đơn luồng và đa luồng

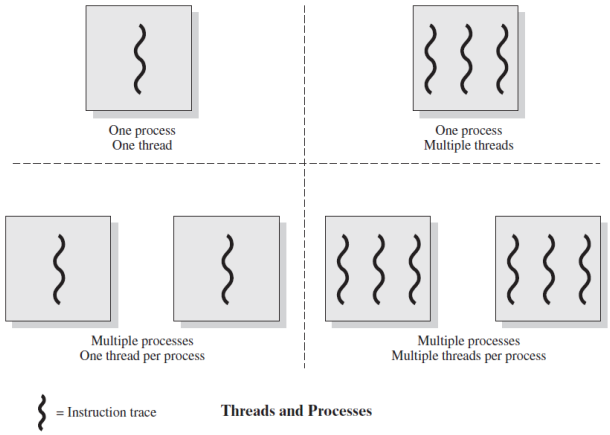


(a) Three processes each with one thread. (b) One process with three threads.

Andrew S. Tanenbaum, Modern Operating Systems, Pearson, 2015.

Mô hình luồng (tt)

- Mô hình luồng phụ thuộc vào hệ điều hành

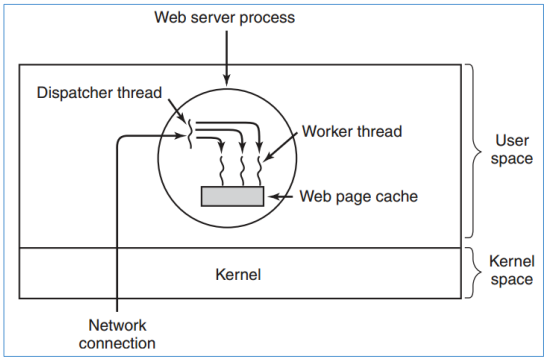


29

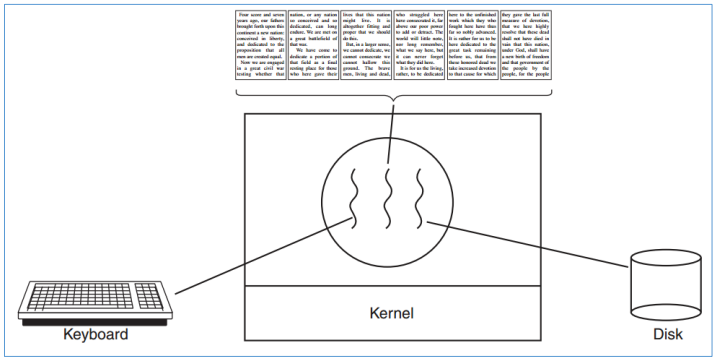
29

Mô hình luồng (tt)

- Web server xử lý đa luồng



Ứng dụng Word với ba luồng xử lý



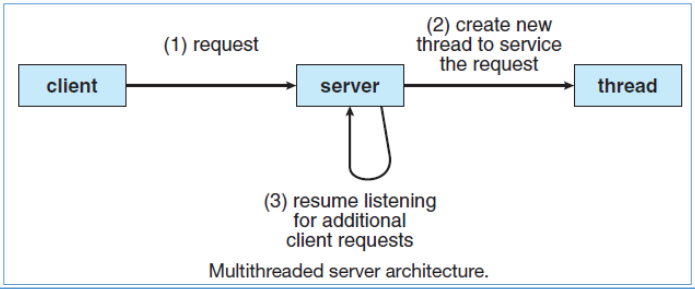
Andrew S. Tanenbaum, Modern Operating Systems, Pearson, 2015.

30

30

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



Multithreaded server architecture.

31

Mô hình luồng (tt)

- Các trạng thái của luồng:
  - Spawn: luồng được tạo.
    - Khi một tiến trình mới được tạo, một luồng chính cũng được tạo ra.
    - Một luồng trong tiến trình có thể tạo ra một luồng khác, cung cấp con trỏ lệnh, các đối số, thanh ghi ngữ cảnh, không gian stack và luồng mới tạo được đưa vào ready queue.
  - Blocked:
    - Khi một luồng phải chờ một sự kiện → blocked (lưu các thanh ghi người dùng, bộ đếm chương trình, con trỏ stack).
    - CPU có thể chuyển sang thực thi một luồng khác trong cùng tiến trình hoặc tiến trình khác.
  - Unblock: khi sự kiện mà luồng đang chờ đã sẵn sàng, luồng chuyển sang trạng thái ready (được đưa vào ready queue)
  - Finish: khi luồng kết thúc

32

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

33

33

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.

34

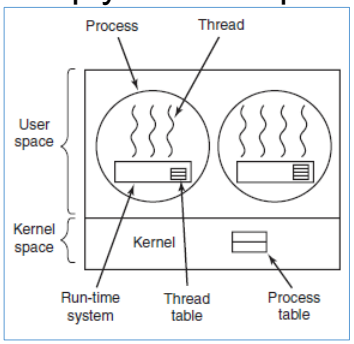
34

2.2.2 Hiện thực luồng

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

Phân loại luồng

- **User-Level Thread (ULT)**
  - Việc quản lý luồng được thực hiện bởi ứng dụng
  - Bất kỳ ứng dụng nào cũng có thể được lập trình đa luồng bằng cách sử dụng Thread library, là một thư viện các quy trình để quản lý ULT.
  - Vai trò của 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



Phân loại luồng (tt)

- **User-Level 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)

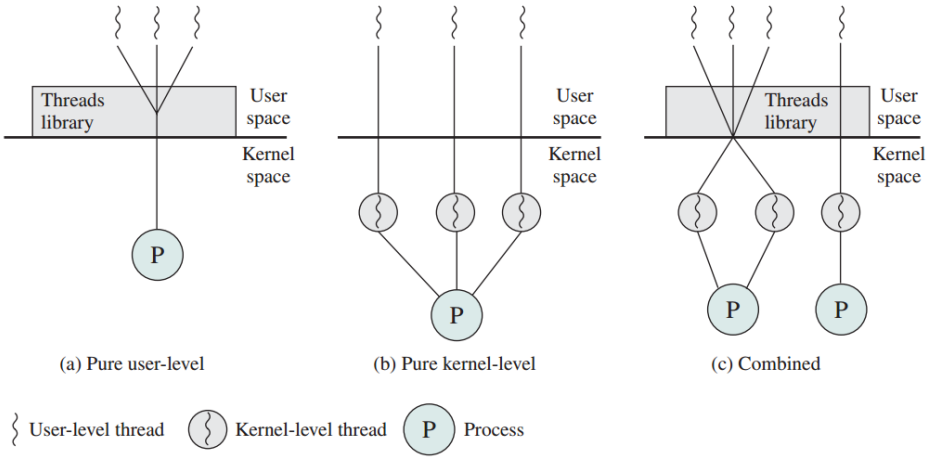
- **Kernel-Level Thread:**
  - Kernel thread được hỗ trợ trực tiếp bởi nhân hệ điều hành.
  - Kernel thực hiện việc tạo luồng, định thời và quản lý trong không gian kernel.
  - Bởi vì việc quản lý thread được thực hiện bởi hệ điều hành, kernel thread được tạo và quản lý chậm hơn user thread.
  - Kernel 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

Phân loại luồng (tt)

- User-Level and Kernel-Level Threads



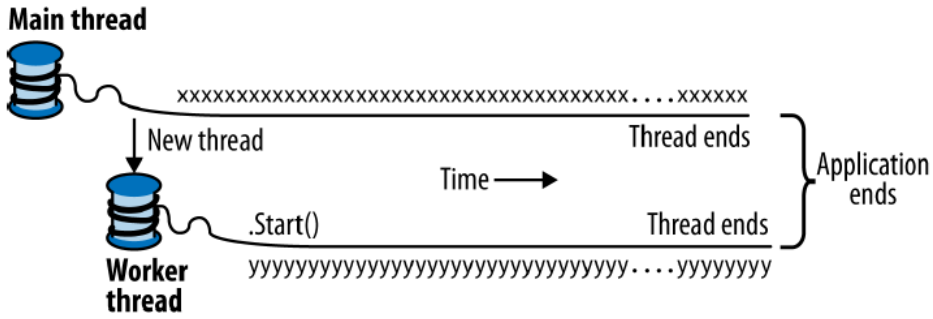
Nguồn: William Stallings,Operating Systems: Internals and Design Principles, Pearson, 2015

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
    - …

Tạo luồng

- Khi tiến trình thực thi, một luồng chính được tạo
- Luồng chính có thể tạo các luồng con



41

41

Tạo luồng (tt)

```
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:
xxxxxxxxxxxxxxxxyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyy
```

42

42

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

43

43

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

44

44

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.

45

45

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

46

46

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 gởi đi bởi :
  - Phần cứng (ví dụ lỗi do các phép tính số học)
  - Kernel gởi đến một tiến trình (ví dụ: báo cho tiến trình khi có một thiết bị I/O rỗi).
  - Một tiến trình gởi 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)

47

47

Tín hiệu (Signal)

- Một số tín hiệu của UNIX

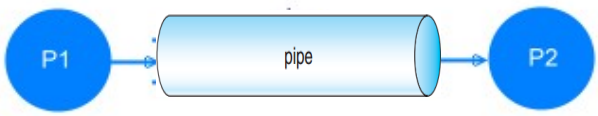
Tín hiệu	Mô tả
SIGINT	Người dùng nhấn phím DEL để ngắt xử lý tiến trình
SIGQUIT	Yêu cầu thoát xử lý
SIGILL	Tiến trình xử lý một chỉ thị bất hợp lệ
SIGKILL	Yêu cầu kết thúc một tiến trình
SIGFPT	Lỗi floating – point xảy ra (chia cho 0)
SIGPIPE	Tiến trình ghi dữ liệu vào pipe mà không có reader
SIGSEGV	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

48

48

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)



49

49

Pipe (tt)

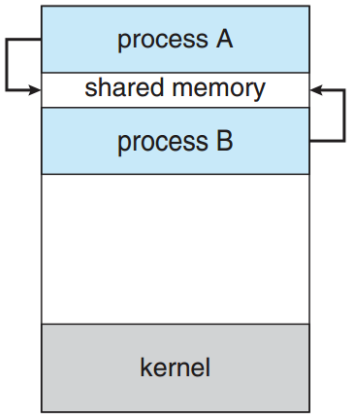
- Một tiến trình có thể sử dụng một pipe do nó tạo ra hay kế thừa từ tiến trình cha.
- Vai trò của Hệ điều hành:
  - Cung cấp các lời gọi hệ thống (hàm) read/write cho các tiến trình thực hiện thao tác đọc/ghi dữ liệu trong pipe
  - Đồng bộ hóa việc truy xuất :
    - Nếu pipe trống: tiến trình đọc sẽ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

50

50

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.

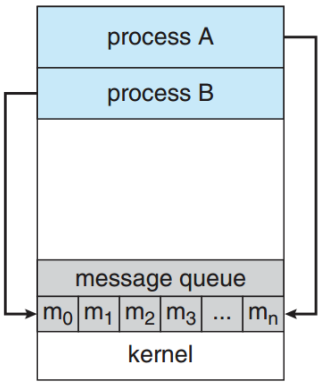


51

51

2.3.1.4 Trao đổi thông điệp (Message)

- Các tiến trình liên lạc với nhau thông qua việc gởi thông điệp.
- HĐH cung cấp các hàm IPC chuẩn (Interprocess communication):
  - Send ([destination], message): gởi 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.



52

52

2.3.1.5 Sockets

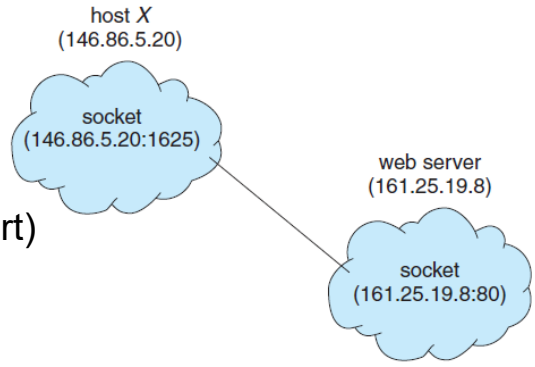
- Được dùng để trao đổi dữ liệu giữa các máy tính trên mạng
- Một socket là một thiết bị truyền thông hai chiều để gửi và nhận dữ liệu giữa các máy trên mạng.
- Mỗi socket là một đầu nối giữa một máy đến một máy tính khác
- Thao tác đọc/ghi là sự trao đổi dữ liệu ứng dụng trên nhiều máy khác nhau.
- 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) .

53

53

Sockets (tt)

- Truyền thông giữa hai tiến trình dùng socket



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

54

54

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)

55

55

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
n = counter;	n = counter;
n = n +1;	n = n + 1;
counter = n;	counter = n;

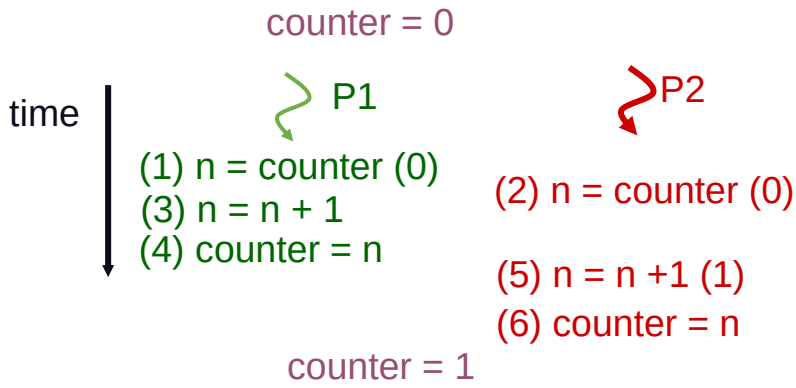
counter?

56

56

Race Condition (tt)

- Đếm số người truy cập website: tình huống 1

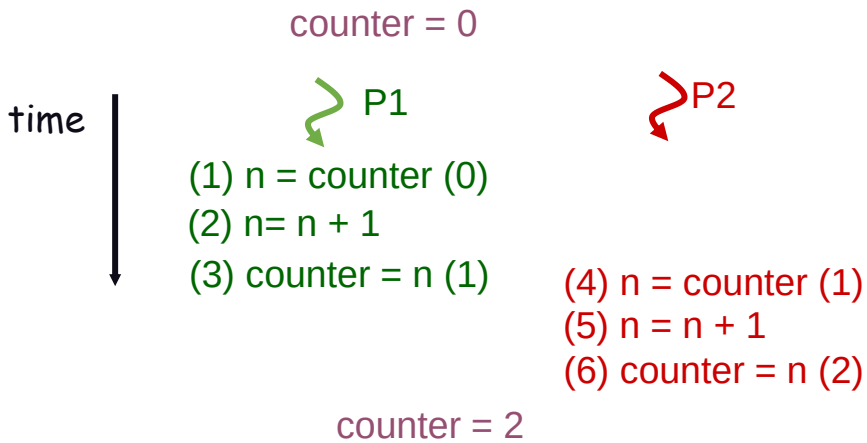


57

57

Race Condition (tt)

- Đếm số người truy cập website: tình huống 2



58

58

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?

59

59

Race Condition (tt)

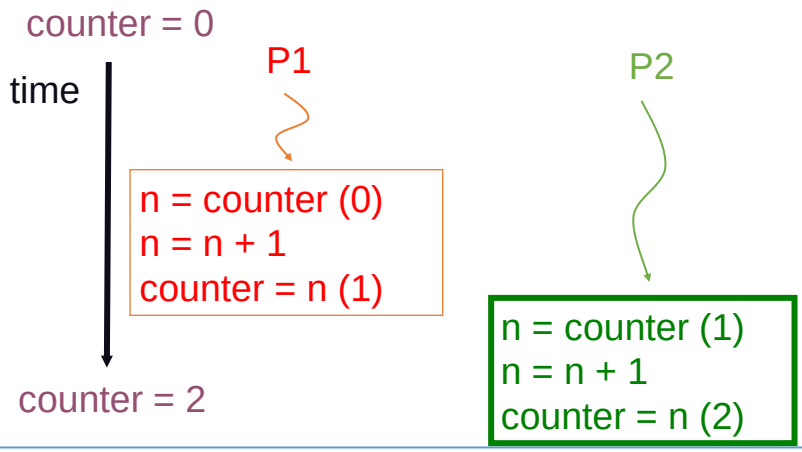
- Lý do xảy ra Race condition:
  - Hai hoặc nhiều tiến trình cùng đọc/ghi dữ liệu trên một vùng nhớ chung
  - Một tiến trình xen vào quá trình truy xuất tài nguyên của một tiến trình khác
- Giải pháp: đảm bảo cho một tiến trình hoàn tất việc truy xuất tài nguyên chung trước khi có tiến trình khác can thiệp.

60

60

Race Condition (tt)

- Giải pháp cho ví dụ 1:

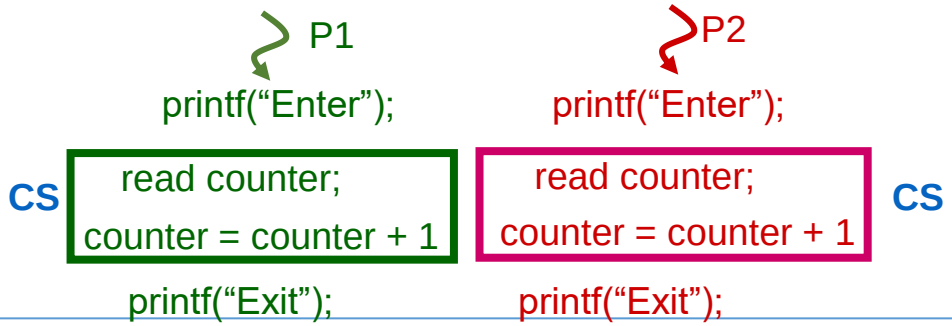


61

61

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

- Miền găng (CS) là đoạn chương trình có khả năng gây ra tình trạng race condition
- Để không xảy ra tình trạng Race condition → bảo đảm tính "độc quyền truy xuất" (Mutual Exclusion) cho miền găng (CS)



62

62

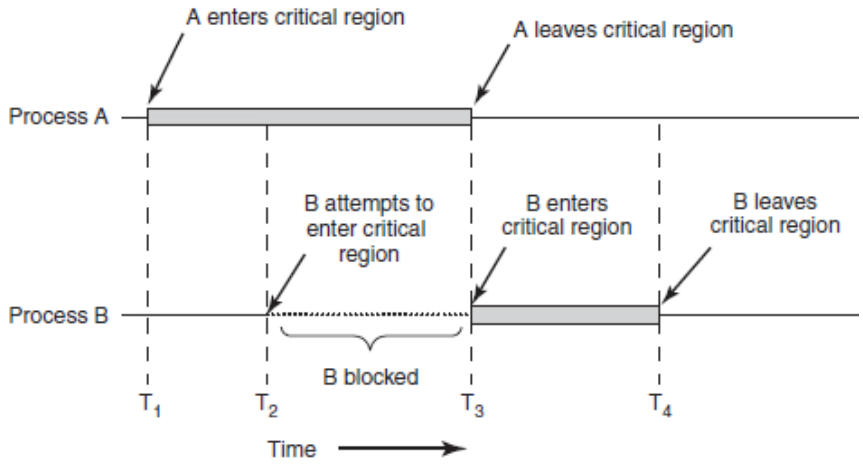
Giải pháp cho vấn đề tranh chấp tài nguyên

- **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.

63

63

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



Mutual exclusion using critical regions.

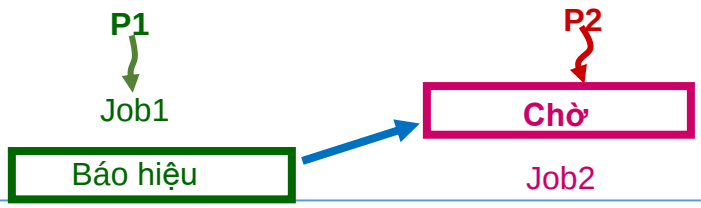
64

64

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

- **Mô hình tổ chức phối hợp giữa hai tiến trình**:
  - Thêm các đoạn code đồng bộ hóa vào 2 chương trình gốc
  - Không áp dụng cho nhiều tiến trình



65

65

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**
    - Cấm ngắt
    - Chỉ thị TSL
- **Nhóm giải pháp Sleep & Wakeup**
  - Mutex
  - Semaphore
  - Monitor
  - Message

66

66

2.3.3.1 Các giải pháp "Busy waiting"

While (chưa có quyền) donothing()

CS;

Từ bỏ quyền sử dụng CS

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

67

67

Sử dụng biến cờ hiệu

- Sử dụng biến **lock** làm khóa chốt, khởi động là 0
  - lock = 0: CS rảnh
  - lock =1: CS bận (có tiến trình đang truy cập)
- Tiến trình muốn vào miền găng phải kiểm tra giá trị của lock
  - lock = 0:
    - đặt lock = 1
    - vào miền găng
    - sau khi ra khỏi miền găng đặt lock = 0.
  - lock = 1: chờ

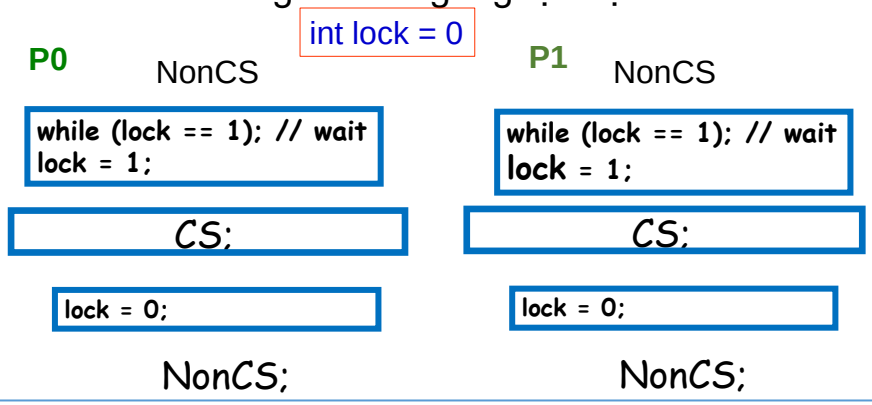
```
while (TRUE) {
        while (lock == 1); // wait
        lock = 1;
        critical-section ();
        lock = 0;
        Noncritical-section ();
}
```

68

68

Sử dụng biến cờ hiệu (tt)

- Nhận xét:
  - Có thể áp dụng cho nhiều tiến trình
  - Hai tiến trình có thể cùng ở miền găng tại một thời điểm



69

69

Kiểm tra luân phiên

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

70

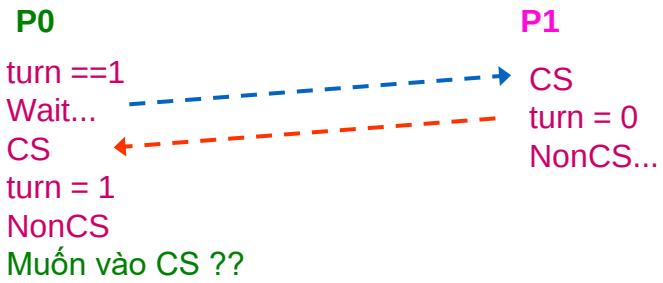
70

Kiểm tra luân phiên (tt)

- Nhận xét:
  - Chỉ áp dụng 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

Kiểm tra luân phiên (tt)

- Vi phạm Progress : int turn = 1



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

Peterson

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

73

73

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;
}
```

74

74

Peterson (tt)

- Nhận xét: đáp ứng được cả 3 điều kiện
  - Mutual Exclusion :
    - Pi chỉ có thể vào CS khi: interested [j] == False, 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 => tiến trình đối phương không khoá lẫn nhau được.
  - Bounded Wait : interested [i] và turn đều có thay đổi giá trị
  - Không thể mở rộng cho N tiến trình

75

75

Cấm ngắt

- Cho phép tiến trình cấm tất cả các ngắt (kể cả ngắt đồng hồ) trước khi vào CS, và phục hồi ngắt khi ra khỏi CS
- → hệ thống không thể tạm dừng hoạt động của tiến trình đang xử lý để cấp phát CPU cho tiến trình khác.
- Nhận xét:
  - 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 vào CS → không đảm bảo Mutual Exclusion.

NonCS;

Disable Interrupt;

CS;

Enable Interrupt;

NonCS;

76

76

Chỉ thị TSL (Test-and-Set)

- Giải pháp này yêu cầu sự trợ giúp của cơ chế phần cứng.
- Hệ thống cung cấp một lệnh đặc biệt cho phép kiểm tra và cập nhật nội dung một vùng nhớ chung.
- 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
  - Tiến trình muốn vào CS phải kiểm tra nếu lock = FALSE →vào CS

77

77

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);
```

78

78

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 do 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**

79

79

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 vào CS) Sleep() ;

CS

Wakeup (other process);

80

80

Sleep & Wake up (tt)

- Hệ Điều hành hỗ trợ 2 hàm đặc quyền:
  - Sleep(): Tiến trình tự gọi hàm này để 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 (P) để đánh thức một tiến trình khác (P) đang bị khóa, để tiến trình này vào CS

81

81

Sleep & Wake up (tt)

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

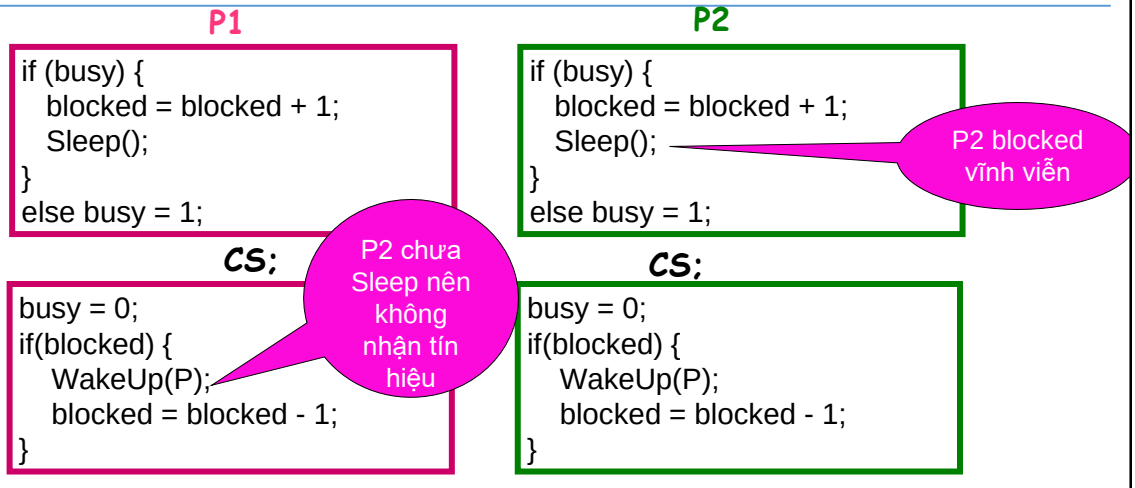
82

82

Sleep & Wake up (tt)

- Có thể xảy ra mâu thuẫn truy xuất
  - Giả sử P1 vào CS, khi chưa xử lý xong P1 hết thời gian sử dụng CPU, P2 được kích hoạt.
  - P2 muốn vào CS nhưng kiểm tra thấy P1 đang ở trong CS → P2 tăng giá trị biến *blocked* và ***sẽ*** gọi *Sleep* để tự khoá.
  - Trước khi P2 gọi *Sleep*, P1 lại được tái kích hoạt, xử lý xong và 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

Vấn đề với Sleep & WakeUp



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

85

85

Mutex locks (tt)

```
mutex_lock() {
  while (!available); /* busy wait */
  available = false;
}

mutex_unlock() {
  available = true;
}
```

```
do {
        mutex_lock ();
        Critical_section
        mutext_unlock ();
        Noncritical_section
} while (true);
```

86

86

Semaphore

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

87

87

Cài đặt Semaphore (Sleep & Wakeup)

```
struct semaphore {
      int count;
      queueType queue;
};
```

```
void Wait(semaphore s){
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
```

```
void Signal(semaphore s){
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

88

88

Sử dụng Semaphore

- Đảm bảo Mutual Exclusion
  - Nhiều tiến trình cùng truy xuất đến CS mà không xảy ra tranh chấp
  - n tiến trình cùng sử dụng một semaphore s, count được khởi gán là 1.

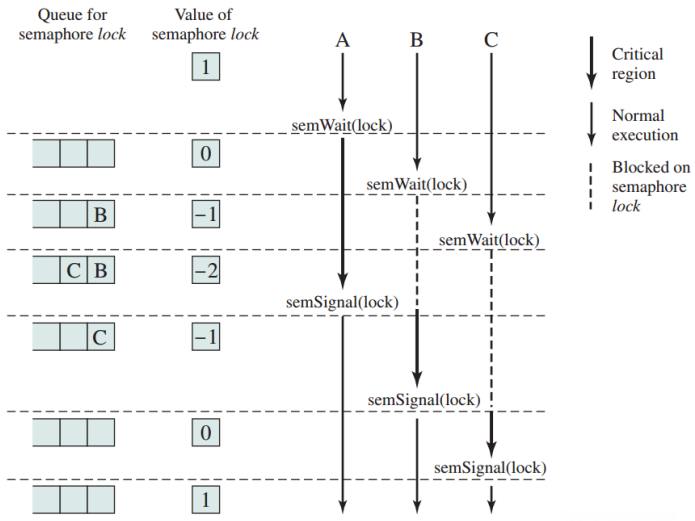
```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true) {
       Wait(s);
       /* critical section */;
       Signal(s);
       /* remainder */;
    }
}
```

89

89

Sử dụng Semaphore

- Mutual Exclusion



William Stallings,Operating Systems: Internals and Design Principles, Pearson, 2015

90

90

Sử dụng Semaphore

- Đảm bảo đồng bộ hóa:
  - 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 count = 0.

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

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

91

91

Nhận xét Semaphore

- Semaphore được xem như là một resource
  - Các tiến trình "yêu cầu" semaphore s: gọi Wait (s)
  - Nếu không hoàn tất được Wait (s): chưa được cấp resource → Blocked, được đưa vào s.queue
- Cần có sự hỗ trợ của HĐH: Sleep() & Wakeup()
- Là một cơ chế tốt để thực hiện đồng bộ
- Mở rộng cho N tiến trình
- Khó sử dụng đúng: nếu thiếu hoặc đặt sai vị trí down và up

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

92

92

Ví dụ về Semaphores
The Producer-Consumer Problem

Buffer

Producer

Consumer

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

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

93

93

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? */
     }
}
```

94

94

Semaphores - The Producer-Consumer Problem (tt)

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

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

95

95

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.

96

96

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 */
    }
}
```

97

97

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 */
    }
}
```

98

98

Monitor

- Đề xuất bởi Hoare (1974) & Brinch (1975)
- Là giải pháp 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

99

99

Monitor

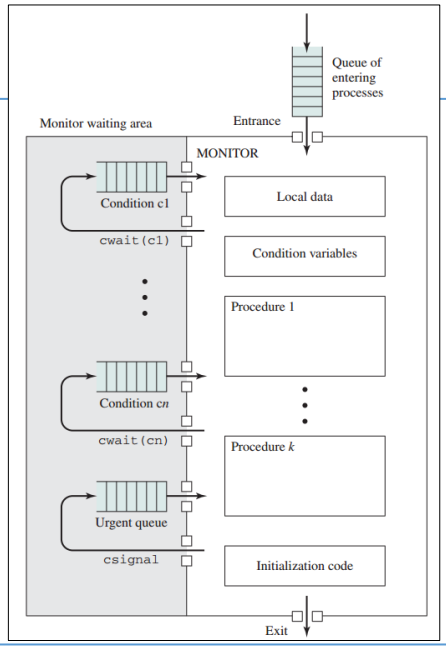
- Các đặc điểm chính của Monitor:
  - Các biến dữ liệu cục bộ chỉ có thể truy cập được bằng thủ tục bên trong Monitor.
  - Một tiến trình hoạt động bên trong monitor bằng cách gọi các phương thức trong monitor.
  - Tại một thời điểm, chỉ có một tiến trình duy nhất được hoạt động bên trong một monitor
  - Các tiến trình không thể vào monitor →đưa vào hàng đợi

100

100

Monitor

- Cấu trúc Monitor:



101

101

Monitor (tt)

- Monitor hỗ trợ đồng bộ hóa bằng các biến điều kiện
- Biến điều kiện là một kiểu dữ liệu đặc biệt, chỉ được truy cập bên trong Monitor
- Việc đồng bộ được thực hiện bởi hai hàm:
  - c.Wait(): P bên trong monitor gọi hàm sẽ bị blocked → hàng đợi trên biến c cho đến khi được một P khác đánh thức
  - c.Signal(): giải phóng 1 P đang bị blocked trên biến điều kiện c
  - c.queue: danh sách các P blocked trên biến điều kiện c

102

102

Monitor (tt)

- Cài đặt monitor

```
monitor monitor_name {
  /* shared variable declarations */
  function Procedure1 (...) {
       ...
  }
  function Procedure2 (...) {
       ...
  }
       ...
  function Proceduren (...) {
       ...
  }
  initialization code (...) {
       ...
  }
}
```

103

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);
       }
}
```

104

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>.Procedurei (); //critical-section();
      Noncritical-section ();
}
```

105

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.

106

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:
  - Gởi 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 →gởi 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 → gởi một thông điệp đến tiến trình đang bị khóa trên tài nguyên đó để đánh thức tiến trình này.

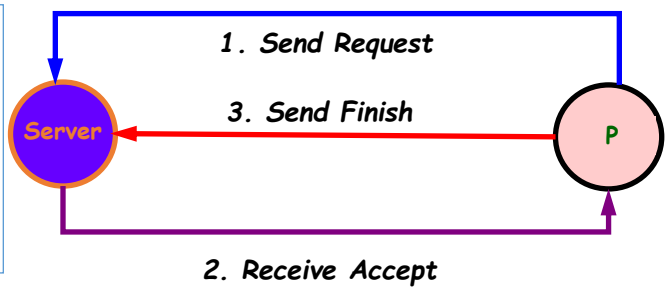
107

107

Message (tt)

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

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



108

108

Các bài toán đồng bộ hoá kinh điển

- Producer – Consumer
- Readers – Writers
- Dinning Philosophers

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

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)

111

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 điều hành xử lý theo lô**
  - Tối ưu thông lượng tối đa (throughput)
  - Giảm thiểu turnaround time: Tquit – Tarrive
  - Tận dụng CPU
- **Hệ điều hành tương tác**
  - Đáp ứng nhanh: giảm thiểu thời gian chờ
  - Cân đối mong muốn của người dùng
- **Hệ điều hành thời gian thực**
  - Tránh mất dữ liệu
  - Đảm bảo chất lượng trong các hệ thống đa phương tiện

112

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 chờ I/O hay các dịch vụ của hệ thống (độc chiếm CPU)
  - Điều phối không độc quyền (preemptive scheduling): tiến trình đang running phải trả CPU khi:
    - Xử lý xong (kết thúc)
    - Bị block chờ I/O hay các dịch vụ của hệ thống
    - Hết thời gian sử dụng CPU
    - Có tiến trình có độ ưu tiên hơn vào ReadyQueue

113

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

114

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)

115

115

2.4.3 Các giải thuật điều phối

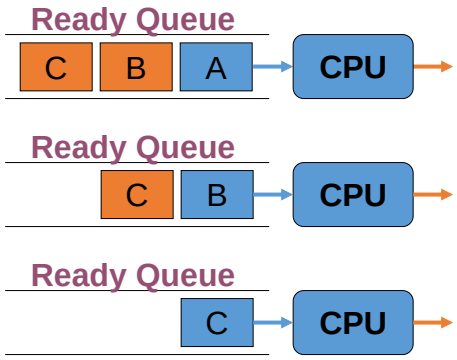
- Điều phối trong Hệ điều hành xử lý theo lô (Batch Systems)
  - First-Come, First-Served (FCFS)
  - Shortest Job First (SJF)
  - Shortest Remaining Time Next (SRTF)
- Điều phối trong Hệ điều hành tương tác (Interactive Systems)
  - Round Robin
  - Priority
  - HRRN
  - Multiple Queues

116

116

2.4.3.1 First-Come First-Served (FCFS)

- Tiến trình vào hàng đợi (ready queue) trước sẽ nhận CPU trước.
- Thời điểm lựa chọn tiến trình
  - Độc quyền



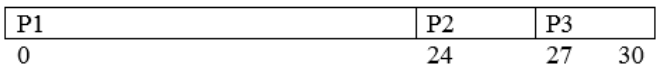
117

FCFS (tt)

- Ví dụ:

P	Arrival Time	Service Time
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
Avg = (23 + 25) /3 =16

Thời gian hoàn thành:
P1: 24 – 0 = 24
P2: 27 – 1 = 26
P3: 30 – 2 = 28
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
Avg = (23 + 25) /3 =16

118

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

119

119

2.4.3.2 Shortest Job First (SJF) Shortest Remaining Time Next (SRT)

- Shortest Job First
  - Chọn tiến trình có thời gian xử lý ngắn nhất
  - Thời điểm lựa chọn tiến trình: độc quyền (non-preemptive)
- Shortest Remaining Time Next
  - Chọn tiến trình có thời gian xử lý còn lại ngắn nhất
  - Thời điểm lựa chọn tiến trình: không độc quyền (khi có một tiến trình mới vào RQ, thời gian xử lý còn lại của các tiến trình được tính toán lại và tiến trình có thời gian xử lý còn lại ngắn nhất sẽ được nhận CPU)

120

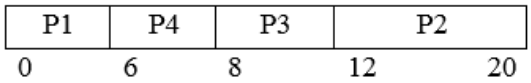
120

SJF – SRT (tt)

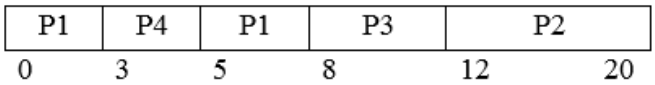
- Ví dụ

P	Arrival Time	Service Time
P1	0	6
P2	1	8
P3	2	4
P4	3	2

- Thời điểm lựa chọn độc quyền



- Thời điểm lựa chọn không độc quyền



121

121

SJF – SRT (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
  - Thời gian sử dụng CPU còn lại cho lần tiếp theo của mỗi tiến trình được tính theo công thức: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$
    - t_n là độ dài của thời gian xử lý lần thứ n
    - t_{n+1} là giá trị dự đoán cho lần xử lý tiếp theo
    - $0 <= \alpha <= 1$

122

122

2.4.3.3 Điều phối với độ ưu tiên (Priority)

- Mỗi tiến trình được gán một độ ưu tiên để 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

123

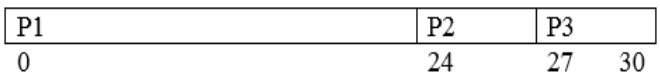
123

Điều phối với độ ưu tiên (tt)

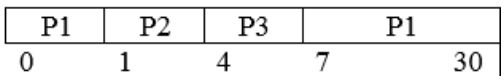
- Ví dụ:

P	Arrival Time	Priority	Service Time
P1	0	3	24
P2	1	1	3
P3	2	2	3

- Độ ưu tiên độc quyền



- Độ ưu tiên không độc quyền



124

124

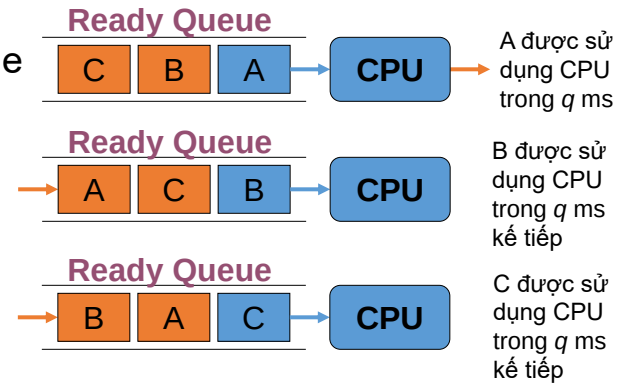
Điều phối với độ ưu tiên (tt)

- 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
    - Tỉ lệ thời gian dùng cho I/O trên thời gian sử dụng CPU
    - Các yêu cầu bên ngoài

125

2.4.3.4 Round Robin (RR) Điều phối xoay vòng

- Mỗi tiến trình chỉ sử dụng một lượng q (quantum) 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



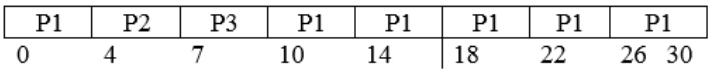
126

Round Robin (tt)

- Ví dụ:
  - RR (q=4)

P	Arrival Time	Service Time
P1	0	24
P2	1	3
P3	2	3

- Biểu đồ Gantt:



127

127

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 => FCFS (giảm tính tương tác)
    - q quá nhỏ => chủ yếu thực hiện chuyển đổi ngữ cảnh (context switching)
    - Thường q = 10 -100 milliseconds

128

128

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

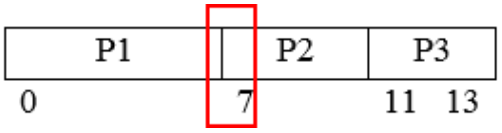
129

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

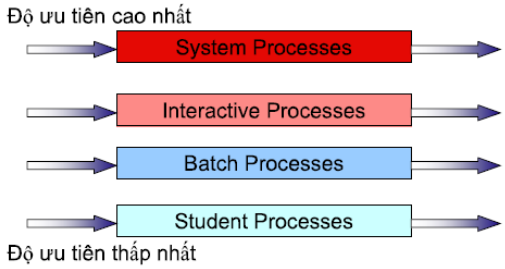
- Độ ưu tiên tại thời điểm (7):
  - P2: (6+4)/4 = 2.5
  - P3: (2+2)/2 = 2
  - → chọn P2



130

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:



131

131

Multilevel Queue Scheduling (tt)

- Fixed priority scheduling:
  - Chọn tiến trình trong hàng đợi có độ ưu tiên từ cao đến thâp.
  - 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

132

132

2.4.3.7 Multilevel Feedback Queue

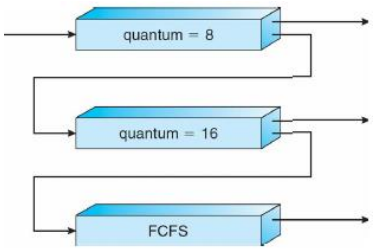
- Trong hệ thống Multilevel Feedback Queue, các tiến trình có thể được di chuyển giữa các queue tùy theo đặc tính của nó tại mỗi thời điểm, Ví dụ:
  - Tiến trình sử dụng CPU quá lâu → chuyển sang một hàng đợi có độ ưu tiên thấp hơn
  - Tiến trình chờ quá lâu trong một hàng đợi có độ ưu tiên thấp → chuyển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

133

133

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 *Q*0. Khi đến lượt, sẽ được cấp phát quantum là 8 ms, nếu chưa xử lý xong → chuyển xuống cuối hàng đợi *Q*1
  - Tại Q1, tiến trình thực thi sẽ được cấp phát quantum là 16ms, nếu chưa xử lý xong → chuyển xuống cuối hàng đợi *Q*2

134

134

Bài tập

- Cho các tiến trình và thời gian đến hang đợi, thời gian sử dụng CPU mô tả trong bảng sau

P	Arrival time	CPU burst	Priority
P1	1	3	3
P2	2	1	2
P3	3	5	1
P4	4	4	4
P5	5	2	5

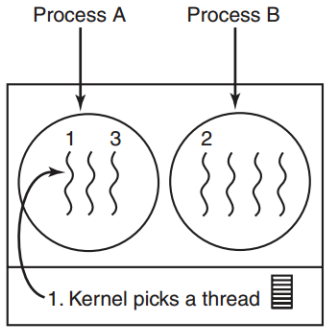
- Vẽ biểu đồ Gantt, tính thời gian đáp ứng, thời gian hoàn thành, thời gian chờ với các thuật toán điều phối:
  - FCFS
  - Priority (thời điểm lựa chọn độc quyền)
  - Priority (thời điểm lựa chọn không độc quyền)
  - SJF (thời điểm lựa chọn độc quyền)
  - SRT (thời điểm lựa chọn không độc quyền)
  - HRRN
  - Round Robin (với q = 2)

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

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



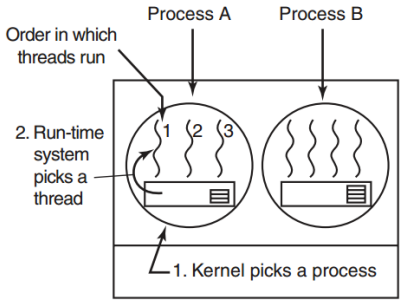
Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

137

137

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



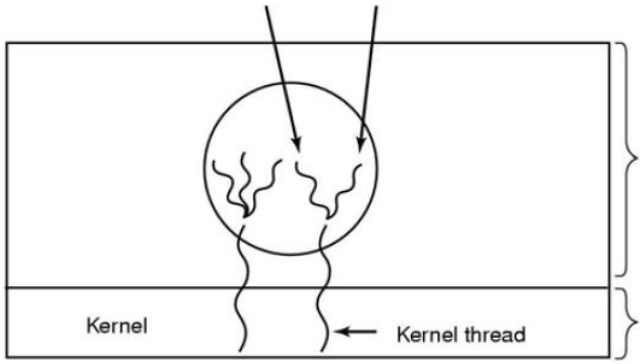
Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3

138

138

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



139