HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

BÀI GIẢNG MÔN

HỆ ĐIỀU HÀNH

Giảng viên: ThS. Nguyễn Thị Ngọc Vinh

Bộ môn: Khoa học máy tính- Khoa CNTT1

Học kỳ/Năm biên soạn: I/ 2009 - 2010

CHƯƠNG 4: QUẢN LÝ TIẾN TRÌNH

NỘI DUNG

1. Các khái niệm liên quan đến tiến trình
2. Luồng (thread)
3. Điều độ tiến trình
4. Đồng bộ hóa các tiến trình đồng thời
5. Tình trạng bế tắc và đói

I. CÁC KHÁI NIỆM LIÊN QUAN ĐẾN TIẾN TRÌNH

1. Tiến trình là gì?

- *Tiến trình là một chương trình đang trong quá trình thực hiện*

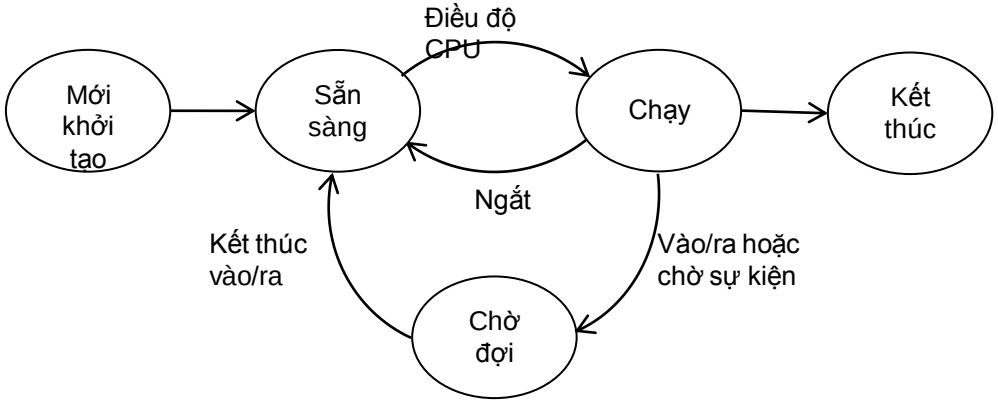
Chương trình	Tiến trình
Thực thể tĩnh	Thực thể động
Không sở hữu tài nguyên cụ thể	Được cấp một số tài để chứa tiến trình và thực hiện lệnh

- Tiến trình được sinh ra khi chương trình được tải vào bộ nhớ để thực hiện
  - Tiến trình người dùng
  - Tiến trình hệ thống

2. Trạng thái của tiến trình

- Phân biệt theo 2 trạng thái: chạy và không chạy
- => Không phản ánh đầy đủ thông tin về trạng thái tiến trình
- => Mô hình 5 trạng thái: *mới khởi tạo, sẵn sàng, chạy, chờ đợi, kết thúc*

- **Mới khởi tạo:** tiến trình đang được tạo ra
- **Sẵn sàng**: tiến trình chờ được cấp CPU để thực hiện lệnh của mình
- **Chạy**: lệnh của tiến trình được CPU thực hiện
- **Chờ đợi**: tiến trình chờ đợi một sự kiện gì đó xảy ra (blocked)
- **Kết thúc:** tiến trình đã kết thúc việc thực hiện nhưng vẫn chưa bị xóa



3. Thông tin mô tả tiến trình

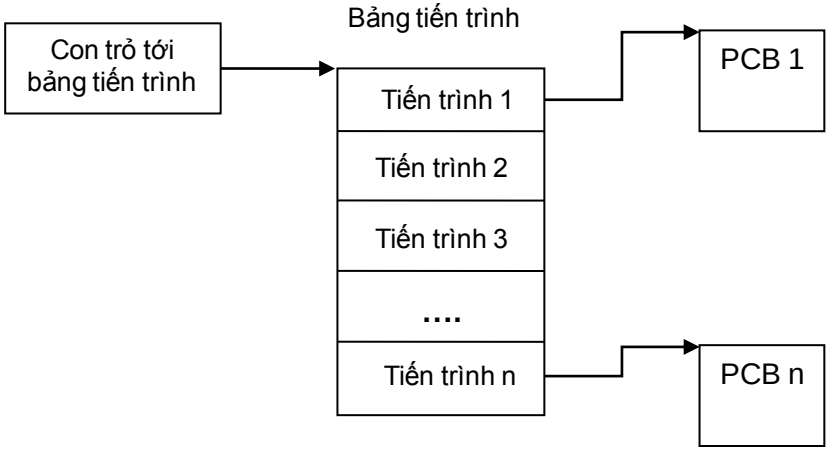
- Được lưu trong một cấu trúc dữ liệu gọi là khối quản lý tiến trình - PCB (Process Control Block)
- Các thông tin chính trong PCB:
  - Số định danh của tiến trình (PID)
  - Trạng thái tiến trình
  - Nội dung một số thanh ghi CPU:
    - Thanh ghi con trỏ lệnh: trỏ tới lệnh tiếp theo
    - Thanh ghi con trỏ ngăn xếp
    - Các thanh ghi điều kiện và trạng thái
    - Các thanh ghi đa năng

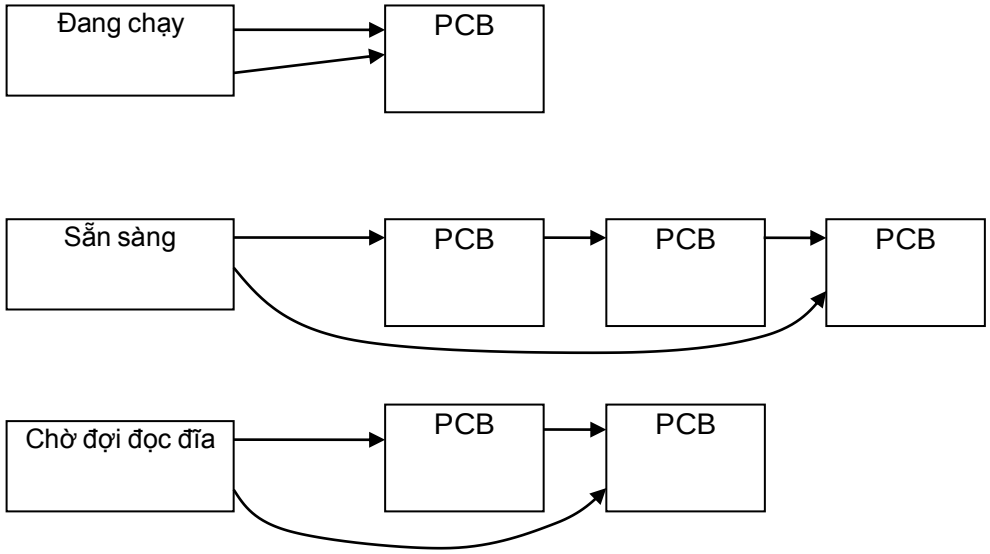
3. Thông tin mô tả tiến trình

- PCB:
  - Thông tin phục vụ điều độ tiến trình: mức độ ưu tiên của tiến trình, vị trí trong hàng đợi, …
  - Thông tin về bộ nhớ của tiến trình
  - Danh sách các tài nguyên khác: các file đang mở, thiết bị vào ra mà tiến trình sử dụng
  - Thông tin thống kê phục vụ quản lý: thời gian sử dụng CPU, giới hạn thời gian

4. Bảng và danh sách tiến trình

- Sử dụng bảng tiến trình chứa con trỏ tới PCB của toàn bộ tiến trình có trong hệ thống
- PCB của các tiến trình cùng trạng thái hoặc cùng chờ 1 tài nguyên nào đó được liên kết thành 1 danh sách





1. Tạo mới tiến trình:
   - Gán số định danh cho tiến trình được tạo mới và tạo một ô trong bảng tiến trình
   - Tạo không gian nhớ cho tiến trình và PCB
   - Khởi tạo PCB
   - Liên kết PCB của tiến trình vào các danh sách quản lý

2. Kết thúc tiến trình:

- Kết thúc bình thường: yêu cầu HDH kết thúc mình bằng cách gọi lời gọi hệ thống exit()
- Bị kết thúc:
  - Bị tiến trình cha kết thúc
  - Do các lỗi
  - Yêu cầu nhiều bộ nhớ hơn so với số lương hệ thống có thể cung cấp
  - Thực hiện lâu hơn thời gian giới hạn
  - Do quản trị hệ thống hoặc hệ điều hành kết thúc

3. Chuyển đổi giữa các tiến trình:

- Thông tin về tiến trình hiện thời (chứa trong PCB) được gọi là ngữ cảnh (context) của tiến trình
- Việc chuyển giữa tiến trình còn được gọi là chuyển đổi ngữ cảnh
- Xảy ra khi:
  - Có ngắt
  - Tiến trình gọi lời gọi hệ thống
- Trước khi chuyển sang thực hiện tiến trình khác, ngữ cảnh được lưu vào PCB
- Khi được cấp phát CPU thực hiện trở lại, ngữ cảnh được khôi phục từ PCB vào các thanh ghi và bảng tương ứng

3. Chuyển đổi giữa các tiến trình:

- Thông tin nào phải được cập nhật và lưu vào PCB khi chuyển tiến trình?
- => Tùy từng trường hợp:
- Hệ thống chuyển sang thực hiện ngắt vào/ra rồi quay lại thực hiện tiếp tiến trình:
  - Ngữ cảnh gồm thông tin có thể bị hàm xử lý ngắt thay đổi
  - => nội dung thanh ghi, trạng thái CPU

3. Chuyển đổi giữa các tiến trình:

- Sau khi thực hiện ngắt, hệ thống thực hiện tiến trình khác
  - Thay đổi trạng thái tiến trình
  - Cập nhật thông tin thống kê trong PCB
  - Chuyển liên kết PCB của tiến trình vào danh sách ứng với trạng thái mới
  - Cập nhật PCB của tiến trình mới được chọn
  - Cập nhật nội dung thanh ghi và trạng thái CPU
- => Chuyển đổi tiến trình đòi hỏi thời gian

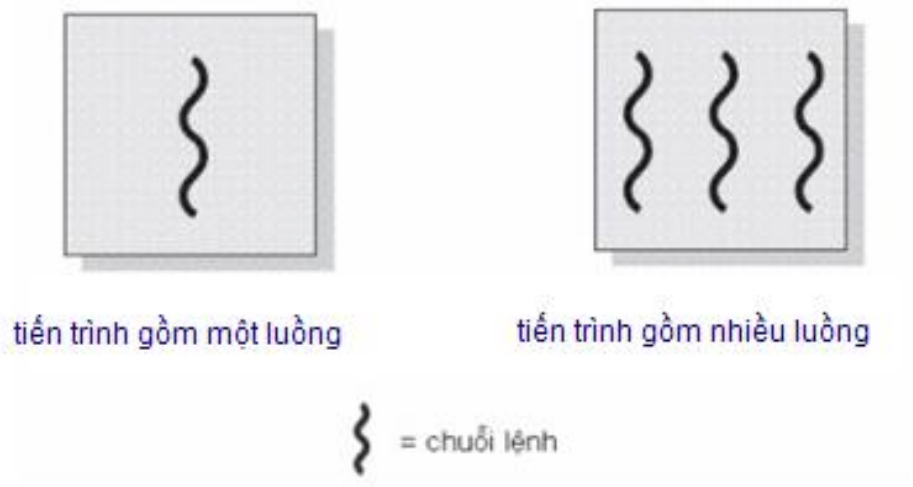
1. Khái niệm

- Tiến trình được xem xét từ 2 khía cạnh:
  - Tiến trình là 1 đơn vị sở hữu tài nguyên
  - Tiến trình là 1 đơn vị thực hiện công việc tính toán xử lý
- Các HDH trước đây: mỗi tiến trình chỉ tương ứng với 1 đơn vị xử lý duy nhất
- => Tiến trình không thể thực hiện nhiều hơn một công việc cùng một lúc

II. LUỒNG THỰC HIỆN

1. Khái niệm (tt)

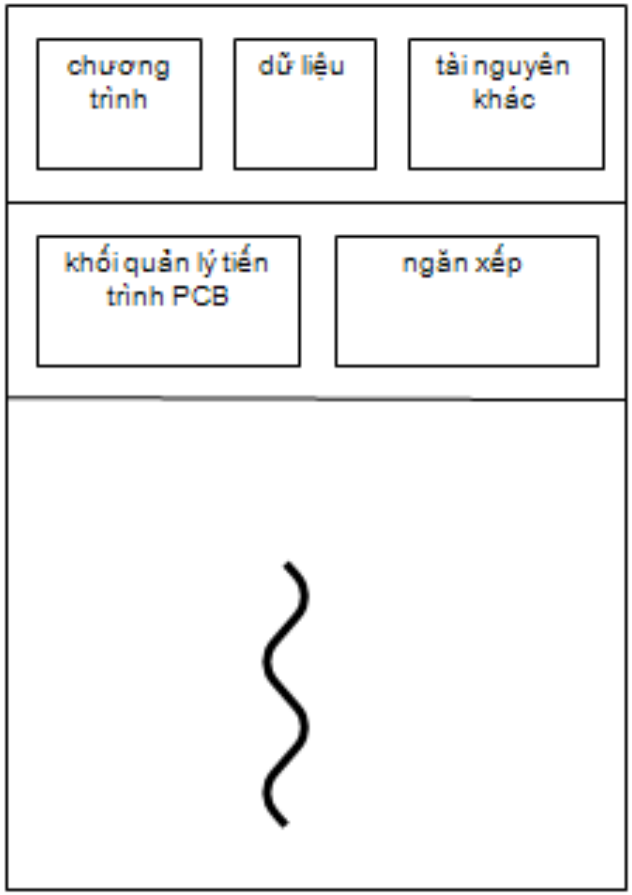
- HDH hiện đại: cho phép tách riêng vai trò thực hiện lệnh của tiến trình
- **Mỗi đơn vị thực hiện lệnh của tiến trình, tức là 1 chuỗi lệnh được cấp phát CPU để thực hiện độc lập được gọi là một *luồng thực hiện***
- HDH hiện nay thường hỗ trợ đa luồng (multithreading)



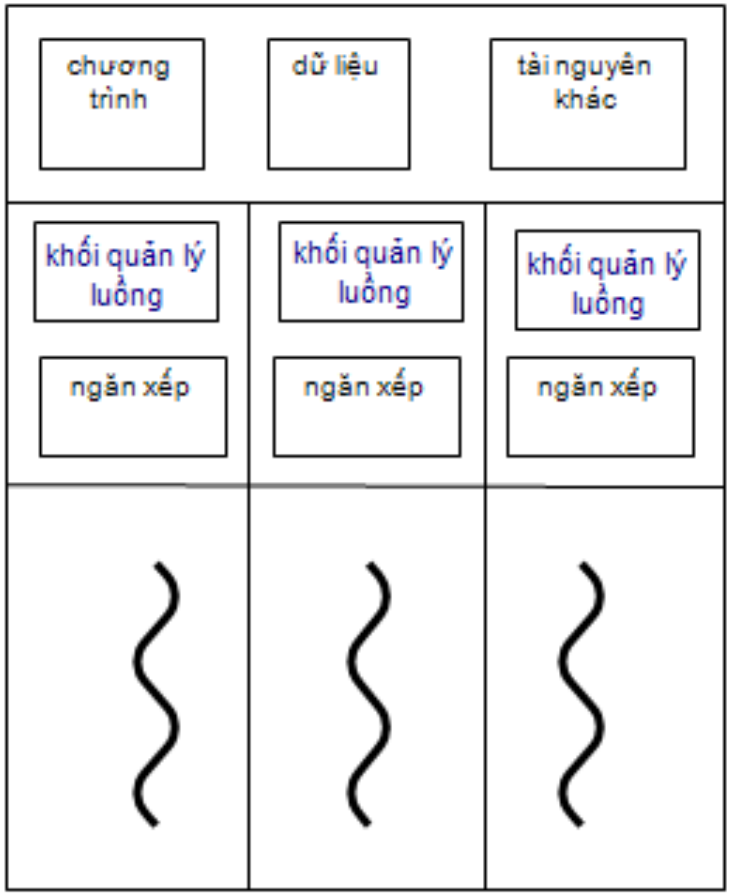
- Trong hệ thống cho phép đa luồng, tiến trình vẫn là 1 đơn vị để HDH phân phối tài nguyên
- Mỗi tiến trình sở hữu chung một số tài nguyên:
  - Không gian nhớ của tiến trình (logic): chứa CT, dữ liệu
  - Các tài nguyên khác: các file đang mở, thiết bị I/O

- Mô hình đơn luồng:
  - Tiến trình có khối quản lý PCB chứa đầy đủ thông tin trạng thái tiến trình, giá trị thanh ghi
  - Ngăn xếp chứa tham số, trạng thái hàm/ thủ tục/ chương trình con
  - Khi tiến trình thực hiện, nó sẽ làm chủ nội dung các thanh ghi và con trỏ lệnh

- Mô hình đa luồng:
  - Mỗi luồng cần có khả năng quản lý con trỏ lệnh, nội dung thanh ghi
  - Luồng cũng có trạng thái riêng
  - => chứa trong khối quản lý luồng
  - Luồng cũng cần có ngăn xếp riêng
  - Tất cả các luồng của 1 tiến trình chia sẻ không gian nhớ và tài nguyên
  - Các luồng có cùng không gian địa chỉ và có thể truy cập tới dữ liệu của tiến trình



Tiến trình đơn luồng



Tiến trình nhiều luồng

- Mô hình đơn dòng:
  - Tiến trình có khối quản lý PCB chứa đầy đủ thông tin trạng thái tiến trình, giá trị thanh ghi
  - Ngăn xếp chứa tham số, trạng thái hàm/ thủ tục/ chương trình con
  - Khi tiến trình thực hiện, nó sẽ làm chủ nội dung các thanh ghi và con trỏ lệnh

3. Ưu điểm của mô hình đa dòng

- Tăng hiệu năng và tiết kiệm thời gian
- Dễ dàng chia sẻ tài nguyên và thông tin
- Tăng tính đáp ứng
- Tận dụng được kiến trúc xử lý với nhiều CPU
- Thuận lợi cho việc tổ chức chương trình

- Có thể tạo và quản lý dòng ở 2 mức:
  - Mức người dùng
  - Mức nhân
- Dòng mức người dùng: được tạo ra và quản lý không có sự hỗ trợ của HDH
- Dòng mức nhân: được tạo ra và quản lý bởi HDH

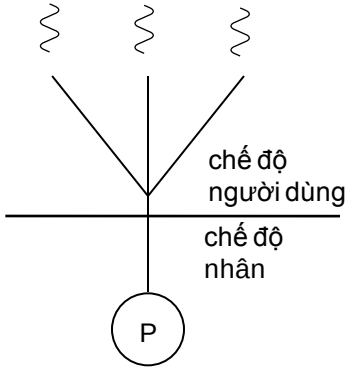
- Do trình ứng dụng tự tạo ra và quản lý
- Sử dụng thư viện do ngôn ngữ lập trình cung cấp
- HDH vẫn coi tiến trình như một đơn vị duy nhất với một trạng thái duy nhất
- Việc phân phối CPU được thực hiện cho cả tiến trình

- Ưu điểm:
  - Việc chuyển đổi dòng không đòi hỏi chuyển sang chế độ nhân => tiết kiệm thời gian
  - Trình ứng dụng có thể điều độ theo đặc điểm riêng của mình, không phụ thuộc vào cách điều độ của HDH
  - Có thể sử dụng trên cả những HDH không hỗ trợ đa dòng

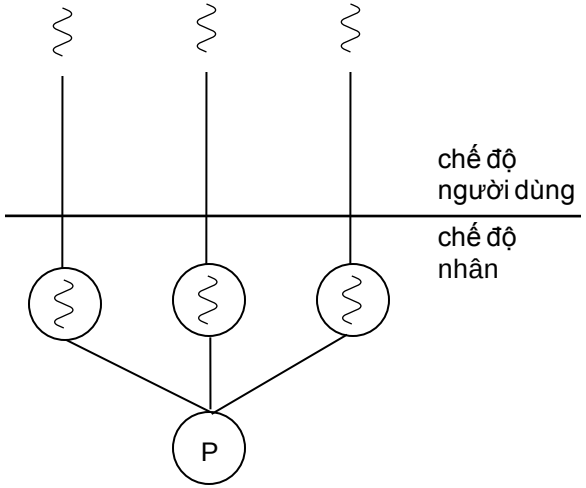
- Nhược điểm:
  - Khi một dòng gọi lời gọi hệ thống và bị phong tỏa, toàn bộ tiến trình bị phong tỏa
  - => không cho phép tận dụng ưu điểm về tính đáp ứng của mô hình đa dòng
  - Không cho phép tận dụng kiến trúc nhiều CPU

- HDH cung cấp giao diện lập trình: gồm các lời gọi hệ thống mà trình ứng dụng có thể yêu cầu tạo/ xóa dòng
- Tăng tính đáp ứng và khả năng thực hiện đồng thời của các dòng trong cùng tiến trình
- Tạo và chuyển đổi dòng thực hiện trong chế độ nhân => tốc độ chậm

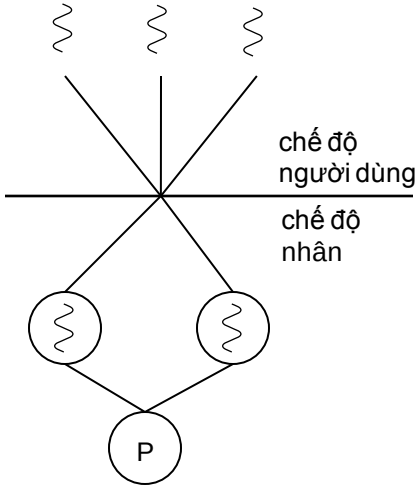
- Dòng mức người dùng được tạo ra trong chế độ người dùng nhờ thư viện
- Dòng mức người dùng được ánh xạ lên số lượng tương ứng hoặc ít hơn các dòng mức nhân



a) Mô hình mức người dùng



b) Mô hình mức nhân



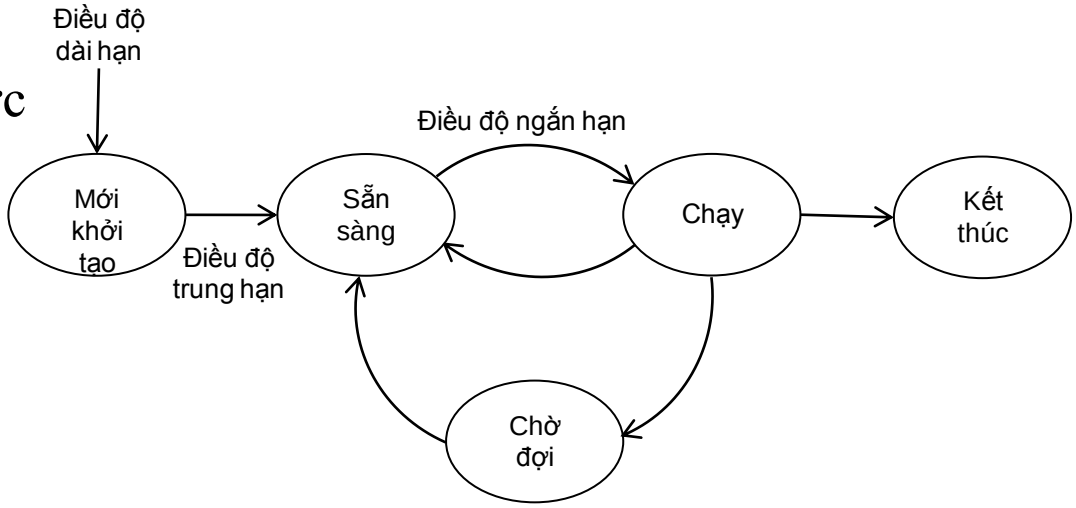
c) Mô hình kết hợp

1. Khái niệm điều độ

- *Điều độ* (scheduling) hay *lập lịch* là quyết định tiến trình nào được sử dụng tài nguyên phần cứng khi nào, trong thời gian bao lâu
- Tập trung vào vấn đề điều độ đối với CPU
- => Quyết định thứ tự và thời gian sử dụng CPU
- Điều độ tiến trình và điều độ dòng:
  - Hệ thống trước kia: tiến trình là đơn vị thực hiện chính => điều độ thực hiện với tiến trình
  - Hệ thống hỗ trợ dòng: dòng mức nhân là đơn vị HDH cấp CPU
  - => Sử dụng thuật ngữ điều độ tiến trình rộng rãi ⇔ điều độ dòng

1. Điều độ dài hạn và ngắn hạn

- Điều độ dài hạn:
  - Thực hiện khi mới tạo ra tiến trình
  - HDH quyết định tiến trình có được thêm vào danh sách đang hoạt động?
  - Ảnh hưởng tới mức độ đa chương trình
- Điều độ trung hạn:
  - Quyết định tiến trình có được cấp MEM để thực hiện?
- Điều độ ngắn hạn:
  - Quyết định tiến trình nào được cấp CPU để thực hiện
  - Thực hiện với tiến trình ở trạng thái sẵn sàng



2. Điều độ có phân phối lại và không phân phối lại:

- Điều độ có phân phối lại (preemptive):
  - HDH có thể sử dụng cơ chế ngắt để thu hồi CPU của một tiến trình đang trong trạng thái chạy
- Điều độ không phân phối lại (nonpreemptive):
  - Tiến trình đang ở trạng thái chạy sẽ được sử dụng CPU cho đến khi xảy ra một trong các tình huống sau:
    - Tiến trình kết thúc
    - Tiến trình phải chuyển sang trạng thái chờ đợi do thực hiện I/O
  - => Điều độ hợp tác: chỉ thực hiện được khi tiến trình hợp tác và nhường CPU
  - Nếu tiến trình không hợp tác, dùng CPU vô hạn => các tiến trình khác không được cấp CPU

2. Điều độ có phân phối lại:

- HDH chủ động hơn, không phụ thuộc vào hoạt động của tiến trình
- Đảm bảo chia sẻ thời gian thực sự
- Đòi hỏi phần cứng có bộ định thời gian và một số hỗ trợ khác
- Vấn đề quản lý tiến trình phức tạp hơn

1. Lượng tiến trình được thực hiện xong:
   - Số lượng tiến trình thực hiện xong trong 1 đơn vị thời gian
   - Đo tính hiệu quả của hệ thống
2. Hiệu suất sử dụng CPU
3. Thời gian vòng đời trung bình của tiến trình:
   - Từ lúc có yêu cầu tạo tiến trình đến khi kết thúc
4. Thời gian chờ đợi:
   - Tổng thời gian tiến trình nằm trong trạng thái sẵn sàng và chờ cấp CPU
   - Ảnh hưởng trực tiếp của thuật toán điều độ tiến trình

5. Thời gian đáp ứng
6. Tính dự đoán được:
   - Vòng đời, thời gian chờ đợi, thời gian đáp ứng phải ổn định, không phụ thuộc vào tải của hệ thống
7. Tính công bằng
   - Các tiến trình cùng độ ưu tiên phải được đối xử như nhau

1. Thuật toán đến trước phục vụ trước (FCFS):

- Tiến trình yêu cầu CPU trước sẽ được cấp trước
- HDH xếp các tiến trình sẵn sàng vào hàng đợi FIFO
- Tiến trình mới được xếp vào cuối hàng đợi
- Đơn giản, đảm bảo tính công bằng
- Thời gian chờ đợi trung bình lớn
- => Ảnh hưởng lớn tới hiệu suất chung của toàn hệ thống
- Thường là thuật toán điều độ không phân phối lại

2. Điều độ quay vòng (RR: round robin):

- Sửa đổi FCFS dùng cho các hệ chia sẻ thời gian
- Có thêm cơ chế phân phối lại bằng cách sử dụng ngắt của đồng hồ
- Hệ thống xác định những khoảng thời gian nhỏ gọi là *lượng tử/ lát cắt thời gian t*
- Khi CPU được giải phóng, HDH đặt thời gian của đồng hồ bằng độ dài lượng tử, lấy tiến trình ở đầu hàng đợi và cấp CPU cho nó
- Tiến trình kết thúc trước khi hết thời gian t: trả quyền điều khiển cho HDH

2. Điều độ quay vòng (tt)

- Hết lượng tử thời gian mà tiến trình chưa kết thúc:
  - Đồng hồ sinh ngắt
  - Tiến trình đang thực hiện bị dừng lại
  - Quyền điều khiển chuyển cho hàm xử lý ngắt của HDH
  - HDH chuyển tiến trình về cuối hàng đợi, lấy tiến trình ở đầu và tiếp tục
- Cải thiện thời gian đáp ứng so với FCFS
- Thời gian chờ đợi trung bình vẫn dài
- Lựa chọn độ dài lượng tử thời gian?

3. Điều độ ưu tiên tiến trình ngắn nhất (SPF)

- Chọn trong hàng đợi tiến trình có chu kỳ sử dụng CPU tiếp theo ngắn nhất để phân phối CPU
- Nếu có nhiều tiến trình với chu kỳ CPU tiếp theo bằng nhau, chọn tiến trình đứng trước
- Thời gian chờ đợi trung bình nhỏ hơn nhiều so với FCFS
- Khó thực hiện vì phải biết độ dài chu kỳ CPU tiếp:
  - Trong các hệ thống xử lý theo mẻ: dựa vào thời gian đăng kí tối đa do lập trình viên cung cấp
  - Dự đoán độ dài chu kỳ CPU tiếp theo: dựa trên độ dài TB các chu kỳ CPU trước đó
- Không có phân phối lại

4. Điều độ ưu tiên thời gian còn lại ngắn nhất

- SFP có thêm cơ chế phân phối lại (SRTF)
- Khi 1 tiến trình mới xuất hiện trong hàng đợi, HDH so sánh thời gian còn lại của tiến trình đang chạy với thời gian còn lại của tiến trình mới xuất hiện
- Nếu tiến trình mới xuất hiện có thời gian còn lại ngắn hơn, HDH thu hồi CPU của tiến trình đang chạy, phân phối cho tiến trình mới
- Thời gian chờ đợi trung bình nhỏ
- HDH phải dự đoán độ dài chu kỳ CPU của tiến trình
- Việc chuyển đổi tiến trình ít hơn so với RR

5. Điều độ có mức ưu tiên

- Mỗi tiến trình có 1 mức ưu tiên
- Tiến trình có mức ưu tiên cao hơn sẽ được cấp CPU trước
- Các tiến trình có mức ưu tiên bằng nhau được điều độ theo FCFS
- Mức ưu tiên được xác định theo nhiều tiêu chí khác nhau

IV. ĐỒNG BỘ HÓA CÁC TIẾN TRÌNH ĐỒNG THỜI

- Những tiến trình cùng tồn tại được gọi *là tiến trình đồng thời /tiến trình tương tranh*
- Quản lý tiến trình đồng thời là vấn đề quan trọng:
  - Liên lạc giữa các tiến trình
  - Cạnh tranh và chia sẻ tài nguyên
  - Phối hợp và đồng bộ hóa các tiến trình
  - Vấn đề bế tắc
  - Đói tài nguyên (starvation)

1. Tiến trình cạnh tranh tài nguyên với nhau:

- Đảm bảo ***loại trừ tương hỗ*** (mutual exclusion):
  - Khi các tiến trình cùng truy nhập tài nguyên mà khả năng chia sẻ của tài nguyên đó là có hạn
  - => phải đảm bảo tiến trình này đang truy cập tài nguyên thì tiến trình khác không được truy cập
  - Tài nguyên đó gọi là ***tài nguyên nguy hiểm***. Đoạn chương trình có yêu cầu sử dụng tài nguyên nguy hiểm gọi là ***đoạn nguy hiểm*** (critical section)
  - Mỗi thời điểm chỉ có 1 tiến trình nằm trong đoạn nguy hiểm => loại trừ lẫn nhau

1. Tiến trình cạnh tranh tài nguyên với nhau (tt):

- Không để xảy ra ***bế tắc*** (deadlock):
  - Bế tắc: tình trạng hai hoặc nhiều tiến trình không thể thực hiện tiếp do chờ đợi lẫn nhau
- Không để ***đói*** tài nguyên (starvation):
  - Tình trạng chờ đợi quá lâu mà không đến lượt sử dụng tài nguyên

2. Tiến trình hợp tác với nhau qua tài nguyên chung:

- Có thể trao đổi thông tin bằng cách chia sẻ vùng nhớ dùng chung (biến toàn thể)
- Đòi hỏi đảm bảo loại trừ tương hỗ
- Xuất hiện tình trạng bế tắc và đói
- Yêu cầu đảm bảo tính nhất quán dữ liệu
- ***Điều kiện chạy đua*** (race condition): tình huống mà một số dòng /tiến trình đọc, ghi dữ liệu sử dụng chung và kết quả phụ thuộc vào thứ tự các thao tác đọc, ghi
- => Đặt thao tác truy cập và cập nhật dữ liệu dùng chung vào đoạn nguy hiểm

3. Tiến trình có liên lạc nhờ gửi thông điệp:

- Có thể trao đổi thông tin trực tiếp với nhau bằng cách gửi thông điệp (message passing)
- Không có yêu cầu loại trừ tương hỗ
- Có thể xuất hiện bế tắc và đói

2. Yêu cầu với giải pháp cho đoạn nguy hiểm

- Loại trừ tương hỗ
- Tiến triển
- Chờ đợi có giới hạn
- Các giả thiết:
  - Giải pháp không phụ thuộc vào tốc độ của các tiến trình
  - Không tiến trình nào được phép nằm quá lâu trong đoạn nguy hiểm
  - Thao tác đọc và ghi bộ nhớ là thao tác nguyên tử (atomic) và không thể bị xen ngang giữa chừng

- Các giải pháp được chia thành 3 nhóm chính:
  - Nhóm giải pháp phần mềm
  - Nhóm giải pháp phần cứng
  - Nhóm sử dụng hỗ trợ của HDH hoặc thư viện ngôn ngữ lập trình

2. Yêu cầu với giải pháp cho đoạn nguy hiểm

- Loại trừ tương hỗ
- Tiến triển
- Chờ đợi có giới hạn
- Các giả thiết:
  - Giải pháp không phụ thuộc vào tốc độ của các tiến trình
  - Không tiến trình nào được phép nằm quá lâu trong đoạn nguy hiểm
  - Thao tác đọc và ghi bộ nhớ là thao tác nguyên tử (atomic) và không thể bị xen ngang giữa chừng

3. Giải thuật peterson

- Giải pháp thuộc nhóm phần mềm
- Đề xuất ban đầu cho đồng bộ 2 tiến trình
- P0 và P1 thực hiện đồng thời với một tài nguyên chung và một đoạn nguy hiểm chung
- Mỗi tiến trình thực hiện vô hạn và xen kẽ giữa đoạn nguy hiểm với phần còn lại của tiến trình
- Yêu cầu 2 tiến trình trao đổi thông tin qua 2 biến chung:
  - Turn: xác định đến lượt tiến trình nào được vào đoạn nguy hiểm
  - Cờ cho mỗi tiến trình: flag[i]=true nếu tiến trình thứ i yêu cầu được vào đoạn nguy hiểm

3. Giải thuật peterson (tt)

bool flag[2]; int turn;

```
Void P0() {
    for(;;)  { //lặp vô hạn
        flag[0] = true;
        turn = 1;
        while(flag[1] && turn ==1) ;
//lặp đến khi điều kiện không thỏa
         <đoạn nguy hiểm>
          flag[0] = false;
         <phần còn lại của tiến trình>
    }
}
```

```
Void P1() {
    for(;;)  {  //lặp vô hạn
        flag[1] = true;
        turn = 0;
        while(flag[0] && turn ==0) ;
//lặp đến khi điều kiện không thỏa
         <đoạn nguy hiểm>
          flag[1] = false;
         <phần còn lại của tiến trình>
    }
}
```

```
Void main() {
        flag[0] = flag[1] = false;  turn =0;
//tắt tiến trình chính, chạy đồng thời 2 tiến trình P0 và P1
        startProcess (P0); startProcess(P1);
}
```

3. Giải thuật peterson (tt)

- Thỏa mãn các yêu cầu:
  - Điều kiện loại trừ tương hỗ
  - Điều kiện tiến triển:
    - P0 chỉ có thể bị P1 ngăn cản vào đoạn nguy hiểm nếu flag[1] = true và turn =1 luôn đúng
    - Có 2 khả năng với P1 ở ngoài đoạn nguy hiểm:
      - P1 chưa sẵn sàng vào đoạn nguy hiểm => flag[1] = false, P0 có thể vào ngay đoạn nguy hiểm
      - P1 đã đặt flag[1]=true và đang trong vòng lặp while => turn = 1 hoặc 0
        - Turn = 0: P0 vào đoạn nguy hiểm ngay
        - Turn = 1: P1 vào đoạn nguy hiểm, sau đó đặt flag[1] = false => quay lại khả năng 1
  - Chờ đợi có giới hạn

3. Giải thuật peterson (tt)

- Sử dụng trên thực tế tương đối phức tạp
- Đòi hỏi tiến trình đang yêu cầu vào đoạn nguy hiểm phải nằm trong trạng thái ***chờ đợi tích cực***
- Chờ đợi tích cực: tiến trình vẫn phải sử dụng CPU để kiểm tra xem có thể vào đoạn nguy hiểm?
- => Lãng phí CPU

1. Cấm các ngắt:

- Tiến trình đang có CPU: thực hiện cho đến khi tiến trình đó gọi dịch vụ hệ điều hành hoặc bị ngắt
- => cấm không để xẩy ra ngắt trong thời gian tiến trình đang ở trong đoạn nguy hiểm để truy cập tài nguyên
- Đảm bảo tiến trình được thực hiện trọn vẹn đoạn nguy hiểm và không bị tiến trình khác chen vào
- Đơn giản
- Giảm tính mềm dẻo của HDH
- Không áp dụng với máy tính nhiều CPU

2. Sử dụng lệnh máy đặc biệt:

- Phần cứng được thiết kế có một số lệnh máy đặc biệt
- 2 thao tác kiểm tra và thay đổi giá trị cho một biến, hoặc các thao tác so sánh và hoán đổi giá trị hai biến, được thực hiện trong cùng một lệnh máy
- => Đảm bảo được thực hiện cùng nhau mà không bị xen vào giữa – thao tác nguyên tử (atomic)
- Gọi là lệnh "kiểm tra và xác lập" – Test_and_Set

2. Sử dụng lệnh máy đặc biệt (tt):

- Logic của lệnh Test_and_Set:

```
Bool Test_and_Set(bool & val)
{
        bool temp = val;
        val = true;
        return temp;
}
```

2. Sử dụng lệnh máy đặc biệt (tt):

```
const int n; //n là số lượng tiến trình
bool lock;
void P(int i){    //tiến trình P(i)
    for(;;){    //lặp vô hạn
        while(Test_and_Set(lock));//lặp đến khi điều kiện không thỏa
        <Đoạn nguy hiểm>
        lock = false;
        <Phần còn lại của tiến trình>
    }
}
void main(){
            lock = false;
//tắt tiến trình chính, chạy đồng thời n tiến trình
            StartProcess(P(1)); ….    StartProcess(P(n));
}
```

2. Sử dụng lệnh máy đặc biệt (tt):

- Ưu điểm:
  - Việc sử dụng tương đối đơn giản và trực quan
  - Có thể dùng để đồng bộ nhiều tiến trình
  - Có thể sử dụng cho trường hợp đa xử lý với nhiều CPU nhưng có bộ nhớ chung
- Nhược điểm:
  - Chờ đợi tích cực
  - Có thể gây đói

5. Cờ hiệu

- Cờ hiệu S là 1biến nguyên được khởi tạo bằng khả năng phục vụ đồng thời của tài nguyên
- Giá trị của S chỉ có thể thay đổi nhờ gọi 2 thao tác là Wait và Signal
  - Wait(S):
    - Giảm S đi 1 đơn vị
    - Nếu giá trị của S<0 thì tiến trình gọi wait(S) sẽ bị phong tỏa
  - Signal(S):
    - Tăng S lên 1 đơn vị
    - Nếu giá trị của S≤0: 1 trong các tiến trình đang bị phong tỏa được giải phóng và có thể thực hiện tiếp

5. Cờ hiệu (tt)

- Wait và Signal là những thao tác nguyên tử
- Để tránh tình trạng chờ đợi tích cực, sử dụng 2 thao tác phong tỏa và đánh thức:
  - Nếu tiến trình thực hiện thao tác wait, giá trị cờ hiệu âm thì thay vì chờ đợi tích cực nó sẽ bị phong tỏa (bởi thao tác block) và thêm vào hàng đợi của cờ hiệu
  - Khi có 1 tiến trình thực hiện thao tác signal thì 1 trong các tiến trình bị khóa sẽ được chuyển sang trạng thái sẵn sàng nhờ thao tác đánh thức (wakeup) chứa trong signal

5. Cờ hiệu (tt)

```
struct semaphore {
        int value;
        process *queue;//danh sách chứa các tiến trình bị phong tỏa
};
void Wait(semaphore& S) {
        S.value--;
        if (S.value < 0) {
        Thêm tiến trình gọi Wait vào S.queue
        block();   //phong tỏa tiến trình
        }
}
void Signal(semaphore& S) {
        S.value++;
        if (S.value <= 0) {
        Lấy một tiến trình P từ S.queue
        wakeup(P);
        }
}
```

5. Cờ hiệu (tt)

- Cờ hiệu được tiến trình sử dụng để gửi tín hiệu trước khi vào và sau khi ra khỏi đoạn nguy hiểm
- Khi tiến trình cần truy cập tài nguyên, thực hiện thao tác Wait của cờ hiệu tương ứng
  - Giá trị cờ hiệu âm sau khi giảm:
    - Tài nguyên được sử dụng hết khả năng
    - Tiến trình thực hiện Wait sẽ bị phong tỏa đến khi tài nguyên được giải phóng
    - Nếu tiến trình khác thực hiện Wait trên cờ hiệu, giá trị cờ hiệu sẽ giảm tiếp
    - Giá trị tuyệt đối của cờ hiệu âm tương ứng với số tiến trình bị phong tỏa
  - Sau khi dùng xong tài nguyên, tiến trình thực hiện thao tác Signal trên cùng cờ hiệu: tăng giá trị cờ hiệu và cho phép một tiến trình đang phong tỏa được thực hiện tiếp

5. Cờ hiệu (tt)

```
const int n; //n là số lượng tiến trình
semaphore S = 1;
void P(int i){    //tiến trình P(i)
    for(;;){    //lặp vô hạn
        Wait(S);
        <Đoạn nguy hiểm>
        Signal(S);
        <Phần còn lại của tiến trình>
    }
}
void main(){
//tắt tiến trình chính, chạy đồng thời n tiến trình
            StartProcess(P(1));
            ...
            StartProcess(P(n));
}
```

1. Bài toán triết gia ăn cơm:
   - 5 triết gia ngồi trên ghế quanh 1 bàn tròn
   - Trên bàn có 5 cái đũa: bên phải và bên trái mỗi người có 1 cái
   - Triết gia có thể nhặt 2 chiếc đũa theo thứ tự bất kì: phải nhặt từng chiếc một và đũa không nằm trong tay người khác
   - Khi cầm được cả 2 đũa: triết gia bắt đầu ăn và không đặt đũa trong thời gian ăn
   - Sau khi ăn xong, triết gia đặt 2 đũa xuống bàn
   - => 5 triết gia như 5 tiến trình đồng thời với tài nguyên nguy hiểm là đũa và đoạn nguy hiểm là đoạn dùng đũa để ăn
   - => Mỗi đũa được biểu diễn bằng 1 cờ hiệu
   - Nhặt đũa: wait(); đặt đũa xuống: signal()

1. Bài toán triết gia ăn cơm:

```
semaphore chopstick[5] = {1,1,1,1,1,1};
void Philosopher(int i){          //tiến trình P(i)
    for(;;){    //lặp vô hạn
        Wait(chopstick[i]);                    //lấy đũa bên trái
        Wait(chopstick[(i+1)%5]);              //lấy đũa bên phải
        <Ăn cơm>
        Signal(chopstick[(i+1)%5]);
        Signal(chopstick[i]);
        <Suy nghĩ>
    }
}
void main(){
// chạy đồng thời 5 tiến trình
            StartProcess(Philosopher(0));
            ...
            StartProcess(Philosopher (4));
}
```

2. Bài toán người sản xuất, người tiêu dùng với bộ đệm hạn chế:
   - Người sản xuất: tạo ra sản phẩm, xếp nó vào 1 chỗ gọi là bộ đệm, mỗi lần 1 sản phẩm
   - Người tiêu dùng: lấy sản phẩm từ bộ đệm, mỗi lần 1 sản phẩm, để sử dụng
   - Dung lượng bộ đệm hạn chế, chứa tối đa N sản phẩm

2. Bài toán người sản xuất, người tiêu dùng với bộ đệm hạn chế:

- 3 yêu cầu đồng bộ:
  1. Người sản xuất và tiêu dùng không được sử dụng bộ đệm cùng lúc
  2. Khi bộ đệm rỗng, người tiêu dùng không nên cố lấy sản phẩm
  3. Khi bộ đệm đầy, người sản xuất không được thêm sản phẩm
- Giải quyết bằng cờ hiệu:
  1. Yêu cầu 1: sử dụng cờ hiệu lock khởi tạo bằng 1
  2. Yêu cầu 2: cờ hiệu empty, khởi tạo bằng 0
  3. Yêu cầu 3: cờ hiệu full, khởi tạo bằng N

2. Bài toán người sản xuất, người tiêu dùng:

Const int N; // kích thước bộ đệm
Semaphore lock = 1;
Semaphore empty = 0;
Semaphore full = N

```
Void producer () {
    for (; ;) {
        <sản xuất>
        wait (full);
        wail (lock);
        <thêm 1 sản phẩm vào bộ đệm>
        signal (lock);
        signal (empty);
    }
}
```

```
Void consumer() {
    for (; ;) {
        wait (empty);
        wail (lock);
        <lấy 1 sản phẩm từ bộ đệm>
        signal (lock);
        signal (full);
        <tiêu dùng>
    }
}
```

```
Void main() {
   startProcess(producer);  startProcess(consumer);
}
```

7. Monitor

- Được định nghĩa dưới dạng một kiểu dữ liệu trừu tượng của ngôn ngữ lập trình bậc cao
- Gồm một dữ liệu riêng, hàm khởi tạo, và một số hàm hoặc phương thức để truy cập dữ liệu:
  - Tiến trình/dòng chỉ có thể truy cập dữ liệu của monitor thông qua các hàm hoặc phương thức của monitor
  - Tại mỗi thời điểm:
    - Chỉ một tiến trình được thực hiện trong monitor
    - Tiến trình khác gọi hàm của monitor sẽ bị phong tỏa, xếp vào hàng đợi của monitor để chờ cho đến khi monitor được giải phóng
  - => Đảm bảo loại trừ tương hỗ đối với đoạn nguy hiểm
  - Đặt tài nguyên nguy hiểm vào trong monitor

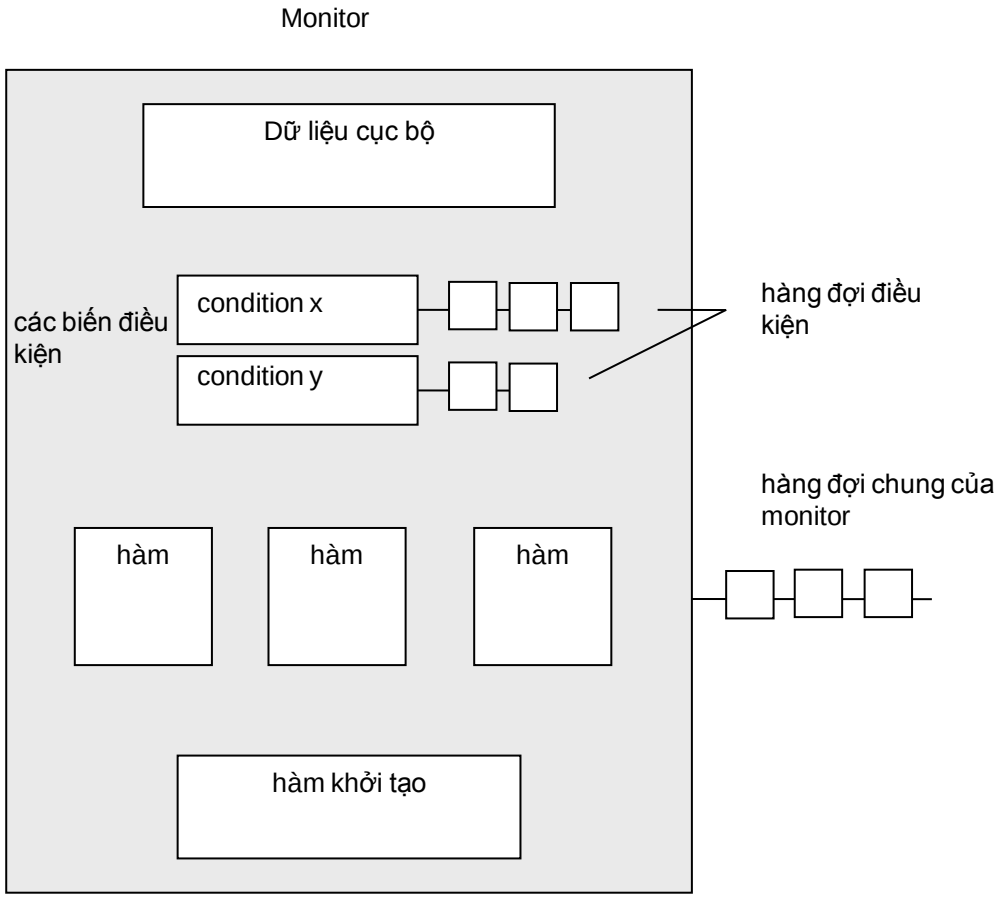
7. Monitor (tt)

- Tiến trình đang thực hiện trong monitor và bị dừng lại để đợi sự kiện => trả lại monitor để tiến trình khác dùng
- Tiến trình chờ đợi sẽ được khôi phục lại từ điểm dừng sau khi điều kiện đang chờ đợi được thỏa mãn
- => Sử dụng các biến điều kiện
- Được khai báo và sử dụng trong monitor với 2 thao tác: cwait() và csignal()

7. Monitor (tt)

- x.cwait():
  - Tiến trình đang ở trong monitor và gọi cwait bị phong tỏa cho tới khi điều kiện x xẩy ra
  - Tiến trình bị xếp vào hàng đợi của biến điều kiện x
  - Monitor được giải phóng và một tiến trình khác sẽ được vào
- x.csignal():
  - Tiến trình gọi csignal để thông báo điều kiện x đã thỏa mãn
  - Nếu có tiến trình đang bị phong tỏa và nằm trong hàng đợi của x do gọi x.cwait() trước đó sẽ được giải phóng
  - Nếu không có tiến trình bị phong tỏa thì thao tác csignal sẽ không có tác dụng gì cả

7. Monitor (tt)


Monitor
Dữ liệu cục bộ
các biến điều kiện
condition x
condition y
hàng đợi điều kiện
hàng đợi chung của monitor
hàm
hàm
hàm
hàm khởi tạo

7. Monitor (tt)

```
monitor BoundedBuffer
        product buffer[N];    //bộ đệm chứa N sản phẩm kiểu product
        int count;            //số lượng sản phẩm hiện thời trong bộ đệm
        condition notFull, notEmpty;    //các biến điều kiện
public:
        boundedbuffer( ) {  //khởi tạo
              count = 0;
        }
        void append (product x) {
              if (count == N)
                        notFull.cwait ( );  //dừng và chờ đến khi buffer có chỗ
              <Thêm một sản phẩm vào buffer>
              count++;
              notEmpty.csignal  ();
}
```

7. Monitor (tt)

```
        product take ( ) {
            if (count == 0)
                    notEmptry.cwait ();  //chờ đến khi buffer không rỗng
            <Lấy một sản phẩm x từ buffer>
            count --;
            notFull.csignal ( );
        }
}

void producer ( )  {   //tiến trình người sản xuất
        for (;;){
            <Sản xuất sản phẩm x>
            BoundedBuffer.append (x);
        }
}
```

7. Monitor (tt)

```
void consumer ( )  {   //tiến trình người tiêu dùng
        for (;;){
              product x = BoundedBuffer.take ();
              <Tiêu dùng x>
        }
}
void main() {
        Thực hiện song song producer và consumer.
}
```

V. BẾ TẮC (DEADLOCK)

1. Định nghĩa

- Tình trạng một nhóm tiến trình có cạnh tranh về tài nguyên hay có hợp tác phải dừng vô hạn
- Do tiến trình phải chờ đợi một sự kiện chỉ có thể sinh ra bởi tiến trình khác cũng đang trong trạng thái chờ đợi

- Đồng thời xảy ra 4 điều kiện:
  1. Loại trừ tương hỗ: có tài nguyên nguy hiểm, tại 1 thời điểm duy nhất 1 tiến trình sử dụng
  2. Giữ và chờ: tiến trình giữ tài nguyên trong khi chờ đợi
  3. Không có phân phối lại (no preemption): tài nguyên do tiến trình giữ không thể phân phối lại cho tiến trình khác trừ khi tiến trình đang giữ tự nguyện giải phóng tài nguyên
  4. Chờ đợi vòng tròn: tồn tại nhóm tiến trình P1, P2, …, Pn sao cho P1 chờ đợi tài nguyên do P2 đang giữ, P2 chờ tài nguyên do P3 đang giữ, …, Pn chờ tài nguyên do P1 đang giữ

- Giải quyết vấn đề bế tắc theo cách:
  - Ngăn ngừa (deadlock prevention): đảm bảo để một trong bốn điều kiện xẩy ra bế tắc không bao giờ thỏa mãn
  - Phòng tránh (deadlock avoidance): cho phép một số điều kiện bế tắc được thỏa mãn nhưng đảm bảo để không đạt tới điểm bế tắc
  - Phát hiện và giải quyết (deadlock detection): cho phép bế tắc xẩy ra, phát hiện bế tắc và khôi phục hệ thống về tình trạng không bế tắc

- Đảm bảo ít nhất 1 trong 4 điều kiện không xảy ra
- Loại trừ tương hỗ: không thể ngăn ngừa
- Giữ và chờ:
  - Cách 1:
    - Yêu cầu tiến trình phải nhận đủ toàn bộ tài nguyên cần thiết trước khi thực hiện tiếp
    - Nếu không nhận đủ, tiến trình bị phong tỏa để chờ cho đến khi có thể nhận đủ tài nguyên
  - Cách 2:
    - Tiến trình chỉ được yêu cầu tài nguyên nếu không giữ tài nguyên khác
    - Trước khi yêu cầu thêm tài nguyên, tiến trình phải giải phóng tài nguyên đã được cấp và yêu cầu lại (nếu cần) cùng với tài nguyên mới

- Không có phân phối lại:
  - Cách 1:
    - Khi một tiến trình yêu cầu tài nguyên nhưng không được do đã bị cấp phát, HDH sẽ thu hồi lại toàn bộ tài nguyên nó đang giữ
    - Tiến trình chỉ có thể thực hiện tiếp sau khi lấy được tài nguyên cũ cùng với tài nguyên mới yêu cầu
  - Cách 2:
    - Khi tiến trình yêu cầu tài nguyên, nếu còn trống, sẽ được cấp phát ngay
    - Nếu tài nguyên do tiến trình khác giữ mà tiến trình này đang chờ cấp thêm tài nguyên thì thu hồi lại để cấp cho tiến trình yêu cầu
    - Nếu hai điều kiện trên đều không thỏa thì tiến trình yêu cầu tài nguyên phải chờ

- Chờ đợi vòng tròn:
  - Xác định thứ tự cho các dạng tài nguyên và chỉ cho phép tiến trình yêu cầu tài nguyên sao cho tài nguyên mà tiến trình yêu cầu sau có thứ tự lớn hơn tài nguyên mà nó yêu cầu trước
  - Giả sử trong hệ thống có n dạng tài nguyên ký hiệu R1, R2, …, Rn
  - Giả sử những dạng tài nguyên này được sắp xếp theo thứ tự tăng dần của chỉ số
  - Nếu tiến trình đã yêu cầu một số tài nguyên dạng Ri thì sau đó tiến trình chỉ được phép yêu cầu tài nguyên dạng Rj nếu $j > i$
  - Nếu tiến trình cần nhiều tài nguyên cùng dạng thì tiến trình phải yêu cầu tất cả tài nguyên dạng đó cùng một lúc

- Ngăn ngừa bế tắc:
  - Sử dụng quy tắc hay ràng buộc khi cấp phát tài nguyên để ngăn ngừa điều kiện xẩy ra bế tắc
  - Sử dụng tài nguyên kém hiệu quả, giảm hiệu năng của tiến trình
- Phòng tránh bế tắc:
  - Cho phép 3 điều kiện đầu xẩy ra và chỉ đảm bảo sao cho trạng thái bế tắc không bao giờ đạt tới
  - Mỗi yêu cầu cấp tài nguyên của tiến trình sẽ được xem xét và quyết định tùy theo tình hình cụ thể
  - HDH yêu cầu tiến trình cung cấp thông tin về việc sử dụng tài nguyên (số lượng tối đa tài nguyên tiến trình cần sử dụng)

4. Phòng tránh bế tắc – thuật toán người cho vay

- Khi tiến trình muốn khởi tạo, thông báo dạng tài nguyên và số lượng tài nguyên tối đa cho mỗi dạng sẽ yêu cầu
- Nếu số lượng yêu cầu không vượt quá khả năng hệ thống, tiến trình sẽ được khởi tạo
- *Trạng thái* được xác định bởi tình trạng sử dụng tài nguyên hiện thời trong hệ thống:
  - Số lượng tối đa tài nguyên mà tiến trình yêu cầu:
    - Dưới dạng ma trận M[n][m]: n là số lượng tiến trình, m: số tài nguyên
    - M[i][j]: số lượng tài nguyên tối đa dạng j mà tiến trình i yêu cầu

- Số lượng tài nguyên còn lại:
  - Dưới dạng vectơ A[m]
  - A[j] là số lượng tài nguyên dạng j còn lại và có thể cấp phát
- Lượng tài nguyên đã cấp cho mỗi tiến trình:
  - Dưới dạng ma trận D[n][m]
  - D[i][j] là lượng tài nguyên dạng j đã cấp cho tiến trình i
- Lượng tài nguyên còn cần cấp
  - Dưới dạng ma trận C[n][m]
  - C[i][j]=M[i][j]-D[i][j] là lượng tài nguyên dạng j mà tiến trình i còn cần cấp

- *Trạng thái an toàn*: trạng thái mà từ đó có ít nhất một phương án cấp phát sao cho bế tắc không xẩy ra
- Cách phòng tránh bế tắc:
  - Khi tiến trình có yêu cầu cấp tài nguyên, hệ thống giả sử tài nguyên được cấp
  - Cập nhật lại trạng thái & xác định xem trạng thái đó có an toàn?
    - Nếu an toàn, tài nguyên sẽ được cấp thật
    - Ngược lại, tiến trình bị phong tỏa &chờ tới khi có thể cấp phát an toàn

- Hệ thống có 3 dạng tài nguyên X, Y, Z với số lượng ban đầu X=10, Y=5, Z=7
- 4 tiến trình P1, P2, P3, P4 có yêu cầu tài nguyên tối đa cho trong bảng
- Xét trạng thái sau của hệ thống:
  - Là trạng thái an toàn
  - Có thể tìm ra cách cấp phát không dẫn đến bế tắc, VD: P2, P4, P3, P1

	X	Y	Z
P1	7	5	3
P2	3	2	2
P3	9	0	2
P4	2	2	2

Yêu cầu tối đa

	X	Y	Z
P1	0	1	0
P2	2	0	0
P3	3	0	2
P4	2	1	1

Đã cấp

X	Y	Z
3	3	4

Còn lại

	X	Y	Z
P1	7	4	3
P2	1	2	2
P3	6	0	0
P4	0	1	1

Còn cần cấp

- P1 yêu cầu cấp phát 3 tài nguyên dạng Y, tức là yêu cầu = (0,3,0). Nếu yêu cầu thỏa mãn, hệ thống chuyển sang trạng thái:
- Trạng thái không an toàn
- => yêu cầu (0,3,0) bị từ chối

X	Y	Z
3	0	4

Còn lại

	X	Y	Z
P1	7	1	3
P2	1	2	2
P3	6	0	0
P4	0	1	1

Còn cần cấp

- Thuật toán xác định trạng thái an toàn:

1. Khai báo mảng W kích thước m và mảng F kích thước n. (F[i] chứa tình trạng tiến trình thứ i đã kết thúc hay chưa, W chứa lượng tài nguyên còn lại)

   Khởi tạo W=A và F[i]=false (i=0,…,n-1)
2. Tìm i sao cho:

   F[i] = false và C[i][j] ≤ W[j] với mọi j=0,…,m-1

   Nếu không có i như vậy thì chuyển sang bước 4
3. a) W = W + D[i]

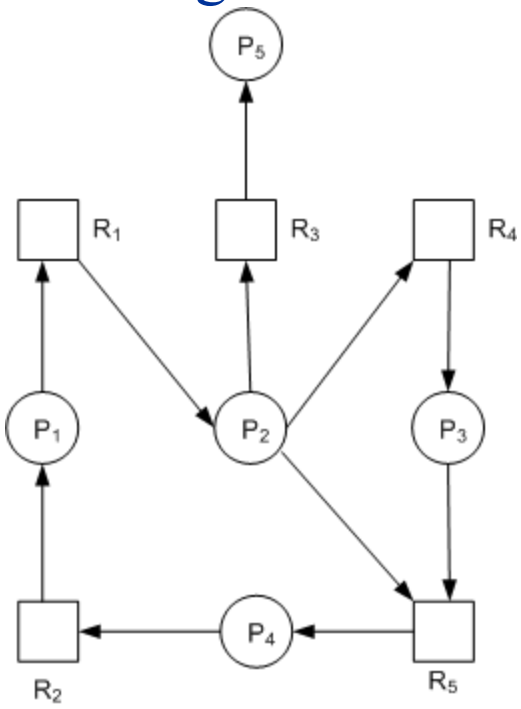
   b) F[i] = true

   c) Quay lại bước 2
4. If F[i] = true với mọi i =0,…,n-1 thì trạng thái an toàn

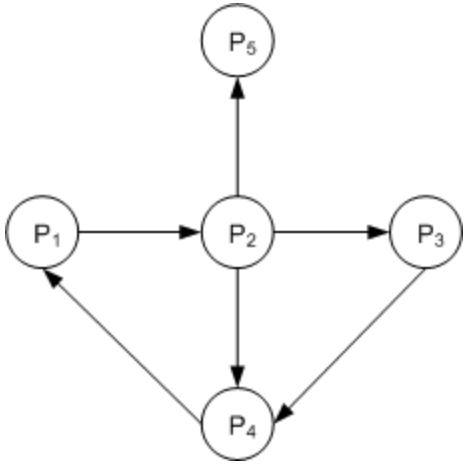
   Else trạng thái không an toàn

- Hệ thống không thực hiện biện pháp ngăn ngừa/phòng tránh => bế tắc có thể xảy ra
- Hệ thống định kỳ kiểm tra để phát hiện có tình trạng bế tắc hay không?
- Nếu có, hệ thống sẽ xử lý để khôi phục lại trạng thái không có bế tắc

- Phát hiện bế tắc:
  - Trường hợp mỗi dạng tài nguyên chỉ có một tài nguyên duy nhất =>sử dụng đồ thị biểu diễn quan hệ chờ đợi lần nhau giữa tiến trình
  - Xây dựng đồ thị cấp phát tài nguyên:
    - Các nút là tiến trình và tài nguyên
    - Tài nguyên được nối với tiến trình bằng cung có hướng nếu tài nguyên được cấp cho tiến trình đó
    - Tiến trình được nối với tài nguyên bằng cung có hướng nếu tiến trình đang được cấp tài nguyên đó



- Phát hiện bế tắc:
  - Đồ thị chờ đợi:
    - Được xây dựng từ đồ thị cấp phát tài nguyên bằng cách bỏ đi các nút tương ứng với tài nguyên và nhập các cung đi qua nút bị bỏ
    - Cho phép phát hiện tình trạng tiến trình chờ đợi vòng tròn là điều kiện đủ để sinh ra bế tắc
    - Sử dụng thuật toán phát hiện chu trình trên đồ thị có hướng để phát hiện bế tắc trên đồ thị chờ đợi



- Thời điểm phát hiện bế tắc:
  - Bế tắc chỉ có thể xuất hiện sau khi một tiến trình nào đó yêu cầu tài nguyên và không được thỏa mãn
  - => Chạy thuật toán phát hiện bế tắc mỗi khi có yêu cầu cấp phát tài nguyên không được thỏa mãn
  - => Cho phép phát hiện bế tắc ngay khi vừa xẩy ra
  - Chạy thường xuyên làm giảm hiệu năng hệ thống
  - => Giảm tần suất chạy thuật toán phát hiện bế tắc:
    - Sau từng chu kỳ từ vài chục phút tới vài giờ
    - Khi có một số dấu hiệu như hiệu suất sử dụng CPU giảm xuống dưới một ngưỡng nào đó

- Xử lý khi bị bế tắc:
  - Kết thúc tất cả tiến trình đang bị bế tắc
  - Kết thúc lần lượt từng tiến trình đang bị bế tắc đến khi hết bế tắc:
    - HDH phải chạy lại thuật toán phát hiện bế tắc sau khi kết thúc 1 tiến trình
    - HDH có thể chọn thứ tự kết thúc tiến trình dựa trên tiêu chí nào đó
  - Khôi phục tiến trình về thời điểm trước khi bị bế tắc sau đó cho các tiến trình thực hiện lại từ điểm này:
    - Đòi hỏi HDH lưu trữ trạng thái để có thể thực hiện quay lui và khôi phục về các điểm kiểm tra trước đó
    - Khi chạy lại, các tiến trình có thể lại rơi vào bế tắc tiếp
  - Lần lượt thu hồi lại tài nguyên từ các tiến trình bế tắc cho tới khi hết bế tắc

6. Ngăn ngừa bế tắc cho bài toán triết gia ăn cơm

- Đặt hai thao tác lấy đũa của mỗi triết gia vào đoạn nguy hiểm để đảm bảo triết gia lấy được hai đũa cùng một lúc
- Quy ước bất đối xứng về thứ tự lấy đũa: ví dụ người có số thứ tự chẵn lấy đũa trái trước đũa phải, người có số thứ tự lẻ lấy đũa phải trước đũa trái
- Tại mỗi thời điểm chỉ cho tối đa bốn người ngồi vào bàn:
  - Sử dụng thêm một cờ hiệu *table* có giá trị khởi tạo bằng 4
  - Triết gia phải gọi thao tác *wait(table)* trước khi ngồi vào bàn và lấy đũa

6. Ngăn ngừa bế tắc cho bài toán triết gia ăn cơm

```
semaphore chopstick[5] = {1,1,1,1,1,1};
semaphore table = 4;
void Philosopher(int i){          //tiến trình P(i)
    for(;;){    //lặp vô hạn
        wait(table);
        wait(chopstick[i]);                    //lấy đũa bên trái
        wait(chopstick[(i+1)%5]);              //lấy đũa bên phải
        <Ăn cơm>
        signal(chopstick[(i+1)%5]);
        signal(chopstick[i]);
        signal(table);
        <Suy nghĩ>
    }
}
void main(){
            StartProcess(Philosopher(1));
            ...
            StartProcess(Philosopher (5));
}
```