HỆ ĐIỀU HÀNH

Phạm Đăng Hải
haipd@soict.hut.edu.vn

Bộ môn Khoa học Máy tính
Viện Công nghệ Thông tin & Truyền Thông

Ngày 13 tháng 8 năm 2014

Chương 2 Quản lý tiến trình

Giới thiệu

- Khi chương trình đang thực hiện
  - Được cung cấp tài nguyên (*CPU, bộ nhớ, thiết bị vào/ra...*) để hoàn thành công việc
    - Tài nguyên được cấp khi bắt đầu chương trình hay trong khi chương trình đang thực hiện
  - Gọi là tiến trình (*process*)
- Hệ thống bao gồm tập các tiến trình thực hiện đồng thời
  Tiến trình hệ điều hành Thực hiện mã lệnh hệ thống
  Tiến trình người dùng Thực hiện mã lệnh người dùng
- Tiến trình có thể chứa một hoặc nhiều luồng điều khiển
- Trách nhiệm của Hệ điều hành: Đảm bảo hoạt động của tiến trình và tiểu trình (*luồng*)
  - Tạo/xóa tiến trình (người dùng, hệ thống)
  - Điều phối tiến trình
  - Cung cấp cơ chế đồng bộ, truyền thông và ngăn ngừa tình trạng bế tắc giữa các tiến trình

Nội dung chính

Nội dung chính

Tiến trình

- Trạng thái hệ thống
  - **Vi xử lý:** Giá trị các thanh ghi
  - **Bộ nhớ:** Nội dung các ô nhớ
  - **Thiết bị ngoại vi:** Trạng thái thiết bị
- Thực hiện chương trình $\Rightarrow$ *Trạng thái hệ thống thay đổi*
  - Thay đổi rời rạc, theo từng câu lệnh được thực hiện

Tiến trình là một dãy thay đổi trạng thái của hệ thống

- Chuyển từ trạng thái này sang trạng thái khác được thực hiện theo yêu cầu nằm trong chương trình của người sử dụng
- Xuất phát từ một trạng thái ban đầu

Tiến trình là sự thực hiện chương trình

Tiến trình >< chương trình

- **Chương trình:** thực thể thụ động (*nội dung file trên đĩa*)
  - Mã chương trình: Lệnh máy (*CD2190EA...*)
  - Dữ liệu: Biến được lưu trữ và sử dụng trong bộ nhớ
    - Biến toàn cục
    - Biến được cung cấp động (*malloc, new,..*)
    - Biến stack (*tham số hàm, biến cục bộ*)
  - Thư viện liên kết động (*DLL*)
    - Không được dịch & liên kết cùng với chương trình

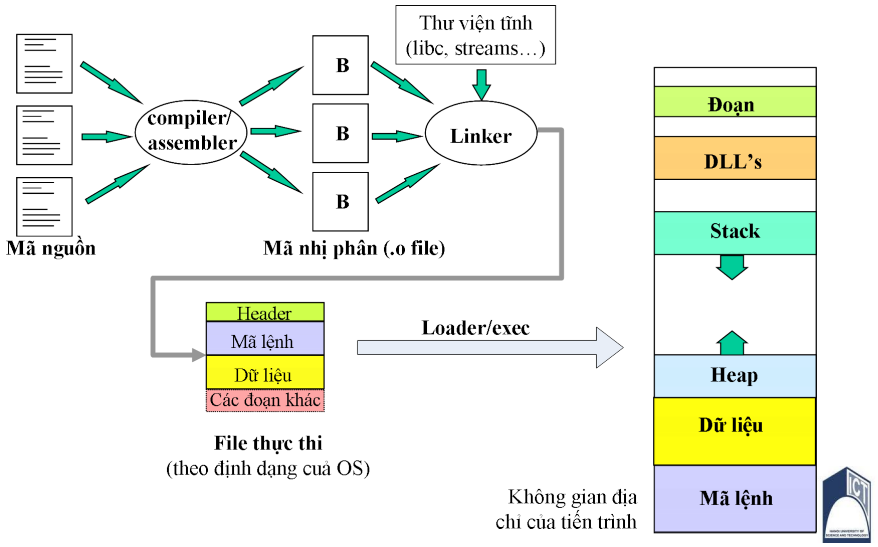
Khi chương trình đang thực hiện, tài nguyên tối thiểu cần có

  - Bộ nhớ cho mã chương trình và dữ liệu
  - Các thanh ghi của VXL phục vụ cho quá trình thực hiện
- **Tiến trình:** thực thể chủ động (*bộ đếm lệnh, tập tài nguyên*)

Một chương trình có thể

- Chỉ là một phần của trạng thái tiến trình
  - Một chương trình, nhiều tiến trình(bộ dữ liệu khác nhau)
    gcc hello.c ↵ || gcc baitap.c ↵
- Gọi tới nhiều tiến trình

Dịch và thực hiên một chương trình



Thực hiện một chương trình

- Hệ điều hành tạo một tiến trình và phân phối vùng nhớ cho nó
- Bộ thực hiện (*loader/exec*)
  - Đọc và dịch (*interprets*) file thực thi (*header file*)
  - Thiết lập không gian địa chỉ cho tiến trình để chứa mã lệnh và dữ liệu từ file thực thi
  - Đặt các tham số dòng lệnh, biến môi trường *(argc, argv, envp)* vào stack
  - Thiết lập các thanh ghi của VXL tới các giá trị thích hợp và gọi hàm "*_start()*" (*hàm của hệ điều hành*)
- Chương trình bắt đầu thực hiện tại "*_start()*". Hàm này gọi tới hàm **main()**(*hàm của chương trình*)
  ⇒*"Tiến trình" đang thực hiện, không còn đề cập đến "chương trình"* nữa
- Khi hàm **main()** kết thúc, OS gọi tới hàm "*_exit()*" để hủy bỏ tiến trình và thu hồi tài nguyên

Tiến trình là chương trình đang thực hiện

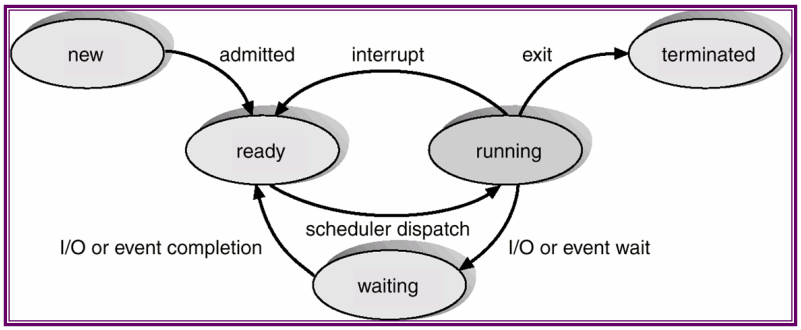
Trạng thái tiến trình

Khi thực hiện, tiến trình thay đổi trạng thái

- Khởi tạo (New) Tiến trình đang được khởi tạo
- Sẵn sàng (Ready) Tiến trình đang đợi sử dụng processor vật lý
- Thực hiện (Running) Các câu lệnh của tiến trình đang được thực hiện
- Chờ đợi (Waiting) Tiến trình đang chờ đợi một sự kiện nào đó xuất hiện (*sự hoàn thành thao tác vào/ra*)
- Kết thúc (Terminated) Tiến trình thực hiện xong

Trạng thái của tiến trình là một phần trong hoạt động hiện tại của tiến trình

Lưu đồ thay đổi trạng thái tiến trình (Silberschatz 2002)

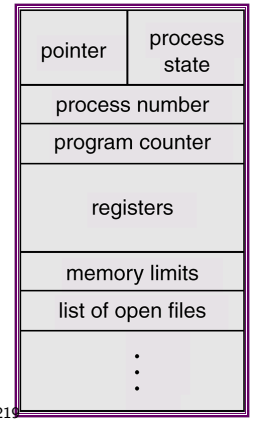


Hệ thống có một processor

- Có duy nhất một tiến trình ở trạng thái thực hiện
- Có thể có nhiều tiến trình ở trạng thái chờ đợi hoặc sẵn sàng

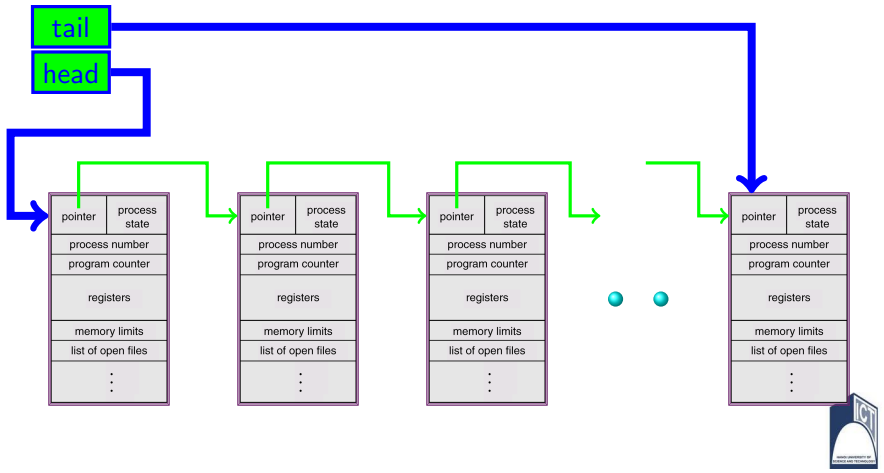
Khối điều khiển tiến trình (PCB: Process Control Block)

- Mỗi tiến trình được thể hiện trong hệ thống bởi một khối điều khiển tiến trình
- PCB: cấu trúc thông tin cho phép xác định duy nhất một **tt**



- Trạng thái tiến trình
- Bộ đếm lệnh
- Các thanh ghi của CPU
- Thông tin dùng để điều phối tiến trình
- Thông tin quản lý bộ nhớ
- Thông tin tài nguyên có thể sử dụng
- Thông tin thống kê
- ...
- Con trỏ tới một PCB khác

Danh sách tiến trình



Tiến trình đơn luồng và tiến trình đa luồng

- **Tiến trình đơn luồng :** Là chương trình thực hiện chỉ một luồng thực thi

  Có một luồng câu lệnh thực thi

  ⇒ Cho phép thực hiện chỉ một nhiệm vụ tại một thời điểm

- **Tiến trình đa luồng :** Là tiến trình có nhiều luồng thực thi

  ⇒ Cho phép thực hiện nhiều hơn một nhiệm vụ tại một thời điểm

1. **Tiến trình**
   - Khái niệm tiến trình
   - Điều phối tiến trình (Process Scheduling)
   - Thao tác trên tiến trình
   - Hợp tác tiến trình
   - Truyền thông liên tiến trình

Giới thiệu

Mục đích Sử dụng tối đa thời gian của CPU

⇒ Cần có nhiều tiến trình trong hệ thống

Vấn đề Luân chuyển CPU giữa các tiến trình

⇒ Phải có hàng đợi cho các tiến trình

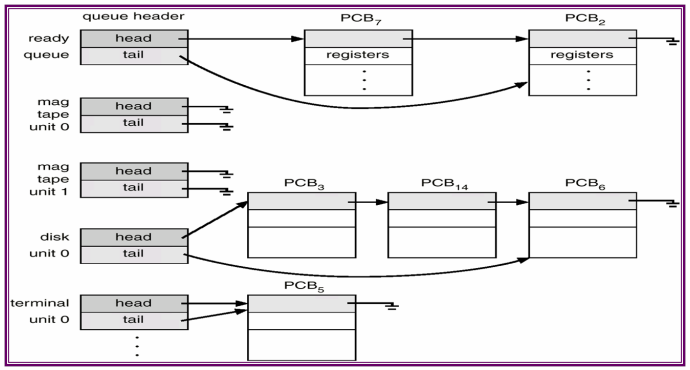
Hệ thống một processor

⇒ Một tiến trình thực hiện

⇒ Các tiến trình khác phải đợi tới khi CPU tự do

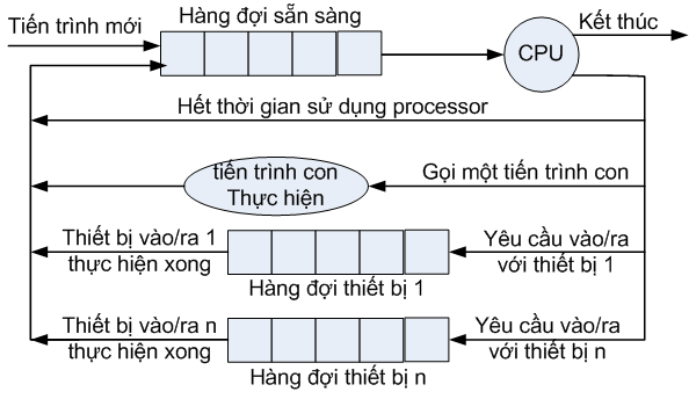
Các hàng đợi tiến trình I

- Hệ thống có nhiều hàng đợi dành cho tiến trình
  - **Job-queue** Tập các tiến trình trong hệ thống
  - **Ready-Queue** Tập các tiến trình tồn tại trong bộ nhớ, đang sẵn sàng và chờ đợi để được thực hiện
  - **Device queues** Tập các tiến trình đang chờ đợi một thiết bị vào ra. Phân biệt hàng đợi cho từng thiết bị



Các hàng đợi tiến trình II

- Các tiến trình di chuyển giữa hàng đợi khác nhau

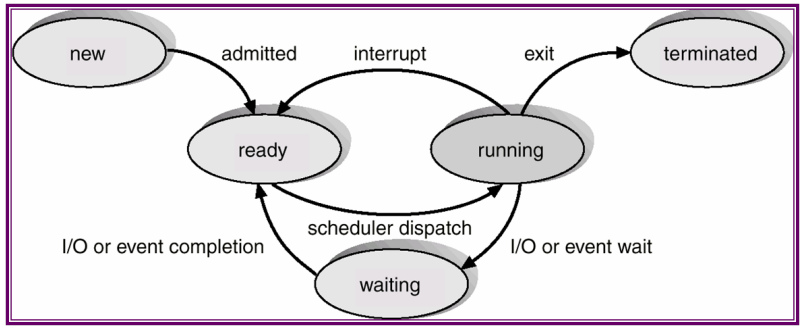


- Tiến trình mới tạo, được đặt trong hàng đợi sẵn sàng, và đợi cho tới khi được lựa chọn để thực hiện

Các hàng đợi tiến trình III

- Tiến trình đã được chọn và đang thực hiên
  1. Đưa ra một yêu cầu vào ra: đợi trong một hàng đợi thiết bị
  2. Tạo một tiến trình con và đợi tiến trình con kết thúc
  3. Hết thời gian sử dụng CPU, phải quay lại hàng đợi sẵn sàng
- Trường hợp (1&2) sau khi sự kiện chờ đợi hoàn thành,
  - Tiến trình sẽ chuyển từ trạng thái đợi sang trạng thái sẵn sàng
  - Tiến trình quay lại hàng đợi sẵn sàng
- Tiến trình tiếp tục chu kỳ (*sẵn sàng, thực hiện, chờ đợi*) cho tới khi kết thúc
  - Xóa khỏi tất cả các hàng đợi
  - PCB và tài nguyên đã cấp được giải phóng

Bộ điều phối (Scheduler)



Lựa chọn tiến trình trong các hàng đợi

- Điều phối công việc (*Job scheduler; Long-term scheduler*)
- Điều phối CPU (*CPU scheduler; Short-term scheduler*)

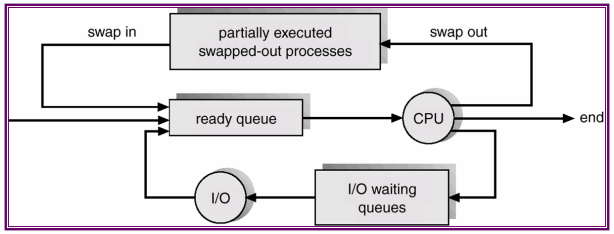
Điều phối công việc

- Chọn các tiến trình từ hàng đợi tiến trình được lưu trong các vùng đệm (*đĩa từ*) và đưa vào bộ nhớ để thực hiện
- Thực hiện không thường xuyên (*đơn vị giây/phút*)
- Điều khiển mức độ đa chương trình (*số t/trình trong bộ nhớ*)
- Khi mức độ đa chương trình ổn định, điều phối công việc được gọi chỉ khi có tiến trình rời khỏi hệ thống
- Vấn đề lựa chọn công việc
  - Tiến trình thiên về vào/ra: sử dụng ít thời gian CPU
  - Tiến trình thiên về tính toán: sử dụng nhiều thời gian CPU
  - Cần lựa chọn lẫn cả 2 loại tiến tình
    ⇒ *tt vào ra:* hàng đợi sẵn sàng rỗng, lãng phí CPU
    ⇒ *tt tính toán:* hàng đợi thiết bị rỗng, lãng phí thiết bị

Điều phối CPU

- Lựa chọn một tiến trình từ hàng đợi các tiến trình đang sẵn sàng thực hiện và phân phối CPU cho nó
- Được thực hiện thường xuyên (*VD: 100ms/lần*)
  - Tiến trình thực hiện vài ms rồi thực hiện vào ra
  - Lựa chọn tiến trình mới, đang sẵn sàng
- Phải thực hiện nhanh
  - 10ms để quyết định ⇒10/(110)=9% thời gian CPU lãng phí
- Vấn đề luân chuyển CPU từ tiến trình này tới tiến trình khác
  - Phải lưu trạng thái của tiến trình cũ (*PCB*) và khôi phục trạng thái cho tiến trình mới
  - Thời gian luân chuyển là lãng phí
  - Có thể được hỗ trợ bởi phần cứng
- Vấn đề lựa chọn tiến trình (*điều phối CPU*)

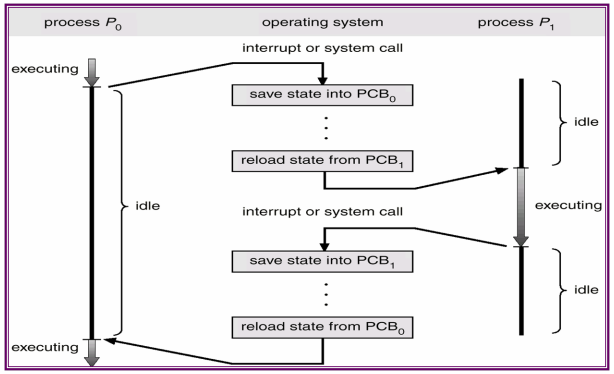
Swapping tiến trình (Medium-term scheduler)



- Nhiệm vụ
  - Đưa t/trình ra khỏi bộ nhớ (*làm giảm mức độ đa chương trình*)
  - Sau đó đưa tiến trình quay trở lại (*có thể ở vị trí khác*) và tiếp tục thực hiện
- Mục đích: Giải phóng vùng nhớ, tạo vùng nhớ tự do rộng hơn

Chuyển ngữ cảnh (context switch)

- Chuyển CPU từ tiến trình này sang tiến trình khác (*hoán đổi tiến trình thực hiện*)
- Thực hiện khi xuất hiện tín hiệu ngắt (*ngắt thời gian*) hoặc tiến trình đưa ra lời gọi hệ thống (*thực hiện và ra*)
- Lưu đồ của chuyển CPU giữa các t/trình *(Silberschatz 2002)*



Thao tác trên tiến trình

- Tạo tiến trình
- Kết thúc tiến trình

Tạo tiến trình

- Tiến trình có thể tạo nhiều tiến trình mới cùng hoạt động (CreateProcess(), fork())
  - Tiến trình tạo: tiến trình cha
  - Tiến trình được tạo: tiến trình con
- Tiến trình con có thể tạo tiến trình con khác ⇒Cây tiến trình
- Vấn đề phân phối tài nguyên
  - Tiến trình con lấy tài nguyên *từ hệ điều hành*
  - Tiến trình con lấy tài nguyên *từ tiến trình cha*
    - Tất cả các tài nguyên
    - Một phần tài nguyên của tiến trình cha (*ngăn ngừa việc tạo quá nhiều tiến trình con*)
- Vấn đề thực hiện
  - Tiến trình cha tiếp tục thực hiện đồng thời với tiến trình con
  - Tiến trình cha đợi tiến trình con kết thúc

Kết thúc tiến trình

- Hoàn thành câu lệnh cuối và yêu cầu HĐH xóa nó (*exit*)
  - Gửi trả dữ liệu tới tiến trình cha
  - Các tài nguyên đã cung cấp được trả lại hệ thống
- Tiến trình cha có thể kết thúc sự thực hiện của tiến trình con
  - Tiến trình cha phải biết định danh tiến trình con ⇒ tiến trình con phải gửi định danh cho tiến trình cha khi được khởi tạo
  - Sử dụng lời gọi hệ thống (*abort*)
- Tiến trình cha kết thúc tiến trình con khi
  - Tiến trình con sử dụng vượt quá mức tài nguyên được cấp
  - Nhiệm vụ cung cấp cho tiến trình con không còn cần thiết nữa
  - Tiến trình cha kết thúc và hệ điều hành không cho phép tiến trình con tồn tại khi tiến trình cha kết thúc
    ⇒Cascading termination. *VD, kết thúc hệ thống*

Một số hàm với tiến trình trong WIN32 API

- CreateProcess(...)
  - **LPCTSTR** Tên của chương trình được thực hiện
  - **LPTSTR** Tham số dòng lệnh
  - **LPSECURITY_ATTRIBUTES** Thuộc tính an ninh t/trình
  - **LPSECURITY_ATTRIBUTES** Thuộc tính an ninh luồng
  - **BOOL** Cho phép kế thừa các thẻ thiết bị (*TRUE/FALSE*)
  - **DWORD** Cờ tạo tiến trình (*VD CREATE_NEW_CONSOLE*)
  - **LPVOID** Trỏ tới khối môi trường
  - **LPCTSTR** Đường dẫn đầy đủ đến chương trình
  - **LPSTARTUPINFO** Cấu trúc thông tin cho tiến trình mới
  - **LPPROCESS_INFORMATION** Thông tin về tiến trình mới
- TerminateProcess(HANDLE hProcess, UINT uExitCode)
  - **hProcess** Thẻ tiến trình bị kết thúc đóng
  - **uExitCode** Mã kết thúc tiến trình
- WaitForSingleObject(HANDLE hHandle, DWORD dwMs)
  - **hHandle** Thẻ đối tượng
  - **dwMs** Thời gian chờ đợi (*INFINITE*)

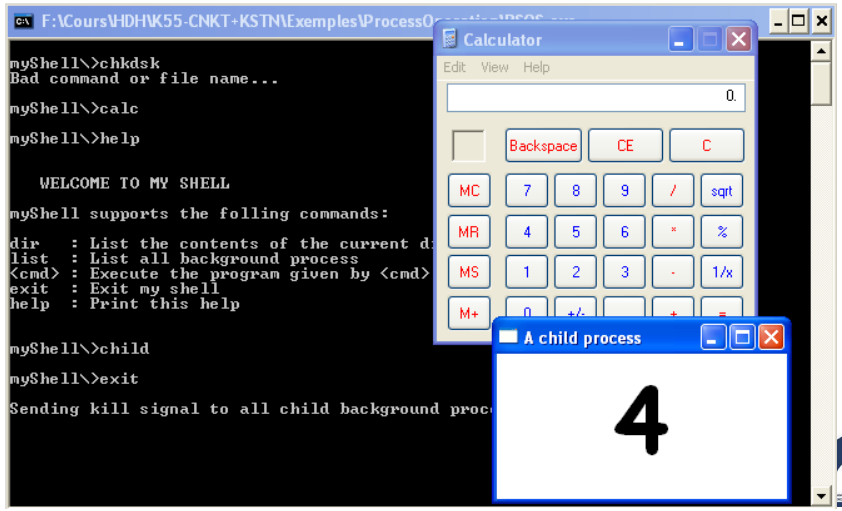
Ví dụ

```
#include <windows.h>
#include <stdio.h>
int main(){
   STARTUPINFO si;
   PROCESS_INFORMATION  pi;
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);

   CreateProcess("Child.exe",NULL,NULL,NULL,FALSE,
          CREATE_NEW_CONSOLE,NULL,NULL,&si,&pi);
   WaitForSingleObject(pi.hProcess,10000);//INFINITE

   TerminateProcess(pi.hProcess, 0);
   CloseHandle(pi.hProcess);
   CloseHandle(pi.hThread);
   return 0;
}
```

Project 1: Tiny shell

- Giới thiệu
  - Thiết kế và cài đặt một *shell* đơn giản (myShell)
- Mục đích
  - Nghiên cứu các API quản lý tiến trình trong Windows
  - Hiểu cách cài đặt và các thức shell làm việc
- Nội dung
  - **Shell** nhận lệnh, phân tích và tạo tiến trình con thực hiện
    - foreground mode: **Shell** phải đợi tiến trình kết thúc
    - background mode: **Shell** và tiến trình thực hiện song song
  - **Shell** chứa các câu lệnh quản lý tiến trình
    - **List:** in ra DS tiến trình (process Id, Cmd name, status)
    - **Kill**, **Stop**, **Resume..** một background process
  - **Shell** hiểu một số lệnh đặc biệt (*exit, help, date, time, dir,..*)
    - **path/addpath** : xem và đặt lại biến môi trường
  - **Shell** có thể nhận tín hiệu ngắt từ bàn phím để hủy bỏ foreground process đang thực hiện (CRTL+C)
  - **Shell** có thể thực hiện được file *.bat

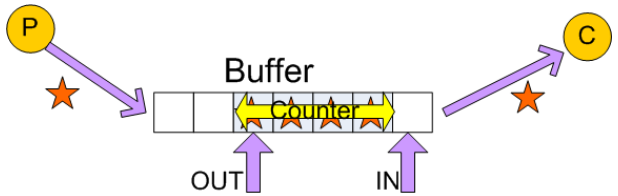
Project 1: Tiny shell → Ví dụ



Phân loại tiến trình

- Các tiến trình tuần tự
  - Điểm bắt đầu của tiến trình này nằm sau điểm kết thúc của tiến trình kia
- Các tiến trình song song
  - Điểm bắt đầu của tiến trình này **nằm giữa điểm bắt đầu và kết thúc** của tiến trình kia
  - **Độc lập:** Không ảnh hưởng tới hoặc bị ảnh hưởng bởi tiến trình khác đang thực hiện trong hệ thống
  - **Có hợp tác:** Ảnh hưởng tới hoặc chịu ảnh hưởng bởi tiến trình khác đang thực hiện trong hệ thống
  - Hợp tác tiến trình nhằm
    - Chia sẻ thông tin
    - Tăng tốc độ tính toán:
    - Module hóa
    - Tiện dụng
  - Hợp tác tiến trình đòi hỏi cơ chế cho phép
    - Truyền thông giữa các tiến trình
    - Đồng bộ hóa hoạt động của các tiến trình

Bài toán người sản xuất (producer)-người tiêu thụ(consumer) I



- Hệ thống gồm 2 tiến trình
  - **Producer** sản xuất ra các sản phẩm
  - **Consumer** tiêu thụ các sản phẩm được sản xuất ra
- Ứng dụng
  - Chương trình in (*producer*) sản xuất ra các ký tự được tiêu thụ bởi bộ điều khiển máy in (consumer)
  - Trình dịch (*producer*) *sản xuất* ra mã hợp ngữ, trình hợp ngữ (*consumer/producer*) *tiêu thụ* mã hợp ngữ rồi *sản xuất* ra module đối tượng được bộ thực hiện (*consumer*) *tiêu thụ*

Bài toán người sản xuất (producer)-người tiêu thụ(consumer) II

- *Producer* và *Consumer* hoạt động đồng thời
- Sử dụng vùng đệm dùng chung (**Buffer**) chứa sản phẩm được điền vào bởi producer và được lấy ra bởi consumer
  - **IN** Vị trí trống kế tiếp trong vùng đệm;
  - **OUT** Vị trí đầy đầu tiên trong vùng đệm.
  - **Counter** Số sản phẩm trong vùng đệm
- *Producer* và *Consumer* phải đồng bộ
  - Consumer không cố gắng tiêu thụ một sản phẩm chưa được sản xuất
- Vùng đệm dung lượng vô hạn
  - Khi Buffer rỗng, Consumer phải đợi
  - *Producer* không phải đợi khi đặt sản phẩm vào buffer
- Vùng đệm dung lượng hữu hạn
  - Khi Buffer rỗng, Consumer phải đợi
  - *Producer* phải đợi nếu vùng đệm đầy

Bài toán người sản xuất (producer)-người tiêu thụ(consumer) III

Producer

```
while(1){
   /*produce an item in nextProduced*/
   while (Counter == BUFFER_SIZE)  ; /*do nothing*/
   Buffer[IN] = nextProduced;
   IN = (IN + 1) % BUFFER_SIZE;
   Counter++;
}
```

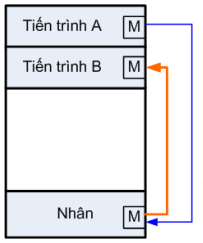
Consumer

```
while(1){
   while(Counter == 0) ; /*do nothing*/
   nextConsumed = Buffer[OUT];
   OUT =(OUT + 1) % BUFFER_SIZE;
   Counter--;
   /*consume the item in nextConsumed*/
}
```

Trao đổi giữa các tiến trình

- Dùng mô hình bộ nhớ phân chia
  - Các tiến trình chia sẻ vùng nhớ chính
  - Mã cài đặt được viết tường minh bởi người lập trình ứng dụng
  - Ví dụ: Bài toán Producer-Consumer

- Dùng mô hình truyền thông liên tiến trình (*Interprocess communication*)
  - Là cơ chế cho phép các tiến trình truyền thông và đồng bộ các hoạt động
  - Thường được sử dụng trong các hệ phân tán khi các tiến trình truyền thông nằm trên các máy khác nhau (*chat*)
  - Đảm bảo bởi hệ thống truyền thông điệp



Hệ thống truyền thông điệp

- Cho phép các tiến trình trao đổi với nhau không qua sử dụng các biến phân chia
- Yêu cầu 2 thao tác cơ bản
  - **Send (msg)** Các msg có kích thước cố định hoặc thay đổi
    - Cố định : dễ cài đặt mức hệ thống, nhiệm vụ lập trình khó
    - Thay đổi: cài đặt mức hệ thống phức tạp, lập trình đơn giản
  - **Receive (msg)**
- Nếu 2 tiến trình P và Q muốn trao đổi, chúng cần
  - Thiết lập một liên kết truyền thông (*vật lý/logic*) giữa chúng
  - Trao đổi các messages nhờ các thao tác *send/receive*
- Các vấn đề cài đặt
  - Các liên kết được thiết lập như thế nào?
  - Một liên kết có thể dùng cho nhiều hơn 2 tiến trình?
  - Bao nhiêu liên kết có thể tồn tại giữa mọi cặp tiến trình?
  - Kích thước thông báo mà liên kết chấp nhận cố định/thay đổi?
  - Liên kết một hay hai chiều?

Truyền thông trực tiếp

- Các tiến trình phải gọi tên tiến trình nhận/gửi một cách tường minh
  - send (P, message) - gửi một thông báo tới tiến trình P
  - receive(Q, message) - Nhận một thông báo từ tiến trình Q
- Tính chất của liên kết truyền thông
  - Các liên kết được thiết lập tự động
  - Một liên kết gắn chỉ với cặp tiến trình truyền thông
  - Chỉ tồn tại một liên kết giữa cặp tiến trình
  - Liên kết có thể là một chiều, nhưng thường hai chiều

Truyền thông gián tiếp

- Các thông điệp được gửi/nhận tới/từ các hòm thư (*mailboxes*), cổng (*ports*)
  - Mỗi hòm thư có định danh duy nhất
  - Các tiến trình có thể trao đổi nếu chúng dùng chung hòm thư
- Tính chất các liên kết
  - Các liên kết được thiết lập chỉ khi các tiến trình dùng chung hòm thư
  - Một liên kết có thể được gắn với nhiều tiến trình
  - Mỗi cặp tiến trình có thể dùng chung nhiều liên kết truyền thông
  - Liên kết có thể một hay hai chiều
- Các thao tác
  - Tạo hòm thư
  - Gửi/nhận thông báo qua hòm thư
    - **send(A, msg)**: Gửi một msg tới hòm thư A
    - **receive(A, msg)**: Nhận một msg từ hòm thư A
  - Hủy bỏ hòm thư

Vấn đề đồng bộ hóa

- Truyền thông điệp có thể phải chờ đợi (*blocking*), hoặc không chờ đợi (*non blocking*)

  Blocking Truyền thông đồng bộ

  Non-blocking Truyền thông không đồng bộ
- Các thủ tục **send()** và **receive()** có thể bị chờ đợi hoặc không chờ đợi
  - **Blocking send** Tiến trình gửi thông báo và đợi cho tới khi msg được nhận bởi tiến trình nhận hoặc bởi hòm thư
  - **Non blockking send** Tiến trình gửi thông báo và tiếp tục làm việc
  - **Blocking receive** Tiến trình nhận phải đợi cho tới khi có thông báo
  - **Non-blocking receive** Tiến trình nhận trả về hoặc một thông báo có giá trị, hoặc một giá trị *null*

Vùng đệm

- Các thông điệp trao đổi giữa các tiến trình được lưu trong hàng đợi tạm thời
- Hàng đợi có thể được cài đặt theo
  - Khả năng chứa 0 (*Zero capacity*): Độ dài hàng đợi là 0
    - Không tồn tại thông điệp trong đường liên kết
      ⇒ Sender phải đợi cho tới khi thông điệp được nhận
  - Khả năng chứa có giới hạn(*Bound capacity*)
    - Hàng đợi có độ dài n ⇒ chứa nhiều nhất n thông điệp
    - Nếu hàng đợi không đầy, thông điệp sẽ được lưu vào trong vùng đệm và Sender tiếp tục bình thường
    - Nếu hàng đợi đầy, sender phải đợi cho tới khi có chỗ trống
  - Khả năng chứa không giới hạn (*Unbound capacity*)
    - Sender không bao giờ phải đợi

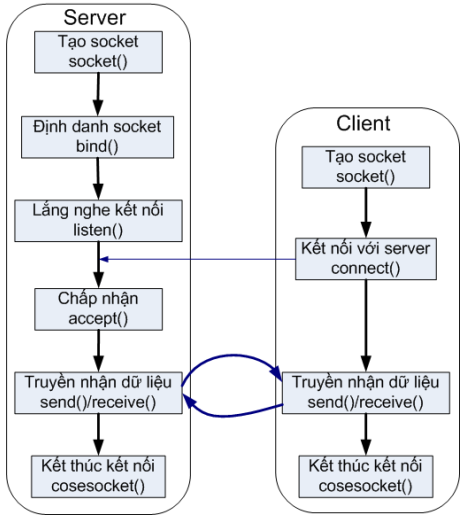
Truyền thông trong hệ thống Client-Server

- Socket
- RPC (Remote Procedure Calls)
- RMI (Remote Method Invocation) Cơ chế truyền thông của Java

Socket

- Được xem như đầu mút cho truyền thông, qua đó các ứng dụng gửi/nhận dữ liệu qua mạng
  - Truyền thông thực hiện giữa các cặp Sockets
- Bao gồm cặp địa chỉ IP và cổng. Ví dụ: **161.25.19.8:1625**
  - Địa chỉ IP: Địa chỉ của máy trong mạng
  - Cổng (port): Định danh tiến trình tham gia trao đổi trên máy
- Các loại sockets
  - Stream Socket: Dựa trên giao thức TCP/IP →Truyền dữ liệu tin cậy
  - Datagram Socket: Dựa trên giao thức UDP/IP →Truyền dữ liệu không tin cậy
- Win32 API: **Winsock**
  - Windows Sockets Application Programming Interface

Thiết lập quá trình trao đổi dữ liệu



Một số hàm trong Winsock API 32

socket() Tạo socket truyền dữ liệu
bind() Định danh cho socket vừa tạo (*gán cho một cổng*)
listen() Lắng nghe một kết nối
accept() Chấp nhận một kết nối
connect() kết nối với server.
send() Gửi dữ liệu với stream socket.
sendto() Gửi dữ liệu với datagram socket.
receive() Nhận dữ liệu với stream socket.
recvfrom() Nhận dữ liệu với datagram socket.
closesocket() Kết thúc một socket đã tồn tại.
.....

Bài tập

- Tìm hiểu các phương pháp truyền thông Client-Server
- Viết chương trình giải quyết bài toán Producer-Consumer

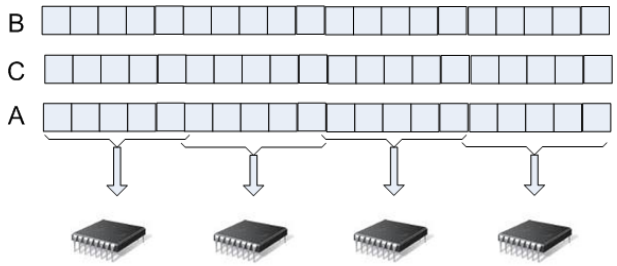
Nội dung chính

Ví dụ: Vector

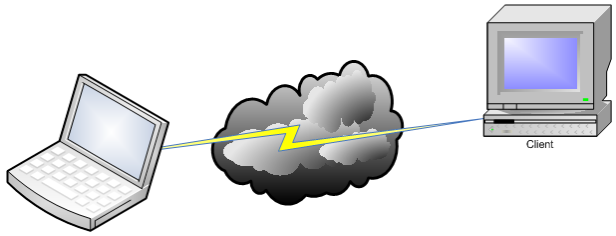
Tính toán trên vector kích thước lớn

```
for(k = 0; k < n; k++){
    a[k] = b[k] * c[k];
}
```

Với hệ thống nhiều vi xử lý



Ví dụ: Chat



Process Q

```
while(1){
  Receive(P,Msg);
  PrintLine(Msg);
  ReadLine(Msg);
  Send(P,Msg);
}
```

Vấn đề nhận Msg

- ▶Blocking Receive
- ▶Non-blocking Receive

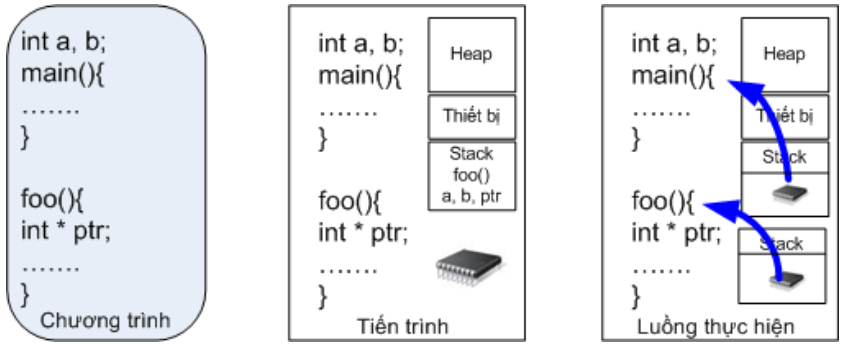
Giải quyết

Thực hiện song song Receive & Send

Process P

```
while(1){
  ReadLine(Msg);
  Send(Q,Msg);
  Receive(Q,Msg);
  PrintLine(Msg);
}
```

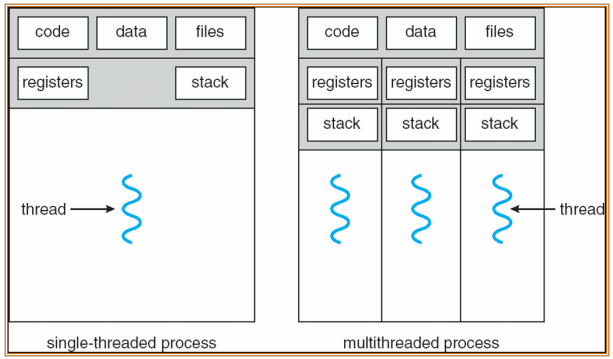

Chương trình - Tiến trình - Luồng



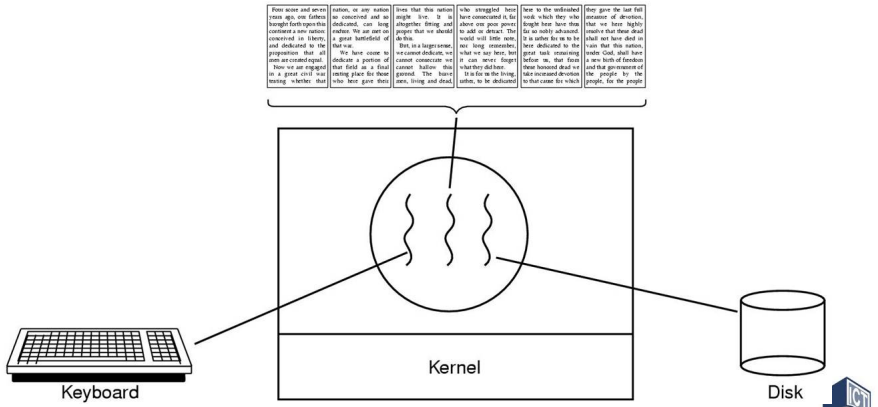
- Chương trình: Dãy lệnh, các biến,..
- Tiến trình: Chương trình đang thực hiện: Stack, t/bị, VXL,..
- Luồng: C/trình đang thực hiện trong ngữ cảnh tiến trình
  - Nhiều processor → Nhiều luồng, mỗi luồng trên một VXL
    - Khác nhau về giá trị các thanh ghi, nội dung stack

Tiến trình đơn luồng và đa luồng

- Hệ điều hành truyền thống (MS-DOS, UNIX)
  - Tiến trình có một luồng điều khiển (heavyweight process)
- Hệ điều hành hiện nay (Windows, Linux)
  - Tiến trình có thể gồm nhiều luồng
  - Có thể thực hiện nhiều nhiệm vụ tại một thời điểm



Ví dụ: Word processor (Tanenbaum 2001)



Khái niệm luồng

- Là đơn vị sử dụng CPU cơ bản, gồm
  - Định danh luồng (*ID Thread*)
  - Bộ đếm chương trình (*Program Computer*)
  - Tập các thanh ghi (*Rigisters*)
  - Không gian stack
- Chia sẻ cùng các luồng khác trong cùng một tiến trình
  - Đoạn mã lệnh
  - Đoạn dữ liệu (*đối tượng toàn cục*)
  - Các tài nguyên hệ điều hành khác (*file đang mở*)
- Các luồng có thể thực hiện cùng đoạn mã với ngữ cảnh (*Tập thanh ghi, Bộ đếm chương trình, stack*) khác nhau
- Còn được gọi là tiến trình nhẹ (**LWP**: *Lightweight Process*)
- Một tiến trình có ít nhất là một luồng

Tiến trình >< Luồng

Tiến trình

- Tiến trình có đoạn mã/dữ liệu/heap & các đoạn khác
- Phải có ít nhất một luồng trong mỗi tiến trình
- Các luồng trong phạm vi một tiến trình chia sẻ mã/dữ liệu/heap, vào/ra nhưng có stack và tập thanh ghi riêng
- Thao tác khởi tạo, luân chuyển tiến trình tốn kém
- Bảo vệ tốt do có không gian địa chỉ riêng
- Khi tiến trình kết thúc, các tài nguyên được đòi lại và các luồng phải kết thúc theo

Luồng

- Luồng không có đoạn dữ liệu hay heap riêng
- Luồng không đứng riêng mà nằm trong một tiến trình
- Có thể tồn tại nhiều luồng trong mỗi tiến trình. Luồng đầu là luồng chính và sở hữu không gian stack của tiến trình
- Thao tác khởi tạo và luân chuyển luồng không tốn kém
- Không gian địa chỉ chung, cần phải bảo vệ
- Luồng kết thúc, stack của nó được thu hồi

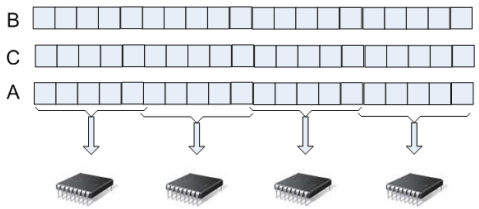
Lợi ích của lập trình đa luồng

- Tăng tính đáp ứng với người dùng
  - Cho phép chương trình vẫn thực hiện ngay khi một phần đang chờ đợi (*block*) hoặc đang thực hiện tính toán tăng cường
  - Ví dụ trình duyệt Web (*Web browser*) đa luồng
    - Một luồng tương tác với người dùng
    - Một luồng thực hiện nhiệm vụ tải dữ liệu
- Chia sẻ tài nguyên
  - Các luồng chia sẻ bộ nhớ và tài nguyên của tiến trình chứa nó
    - Tốt cho các thuật toán song song (*sử dụng chung các CTDL*)
    - Trao đổi giữa các luồng thông qua bộ nhớ phân chia
  - Cho phép một ứng dụng chứa nhiều luồng hoạt động trong cùng không gian địa chỉ
- Tính kinh tế
  - Các thao tác khởi tạo, hủy bỏ và luân chuyển luồng ít tốn kém
    - Minh họa được tính song song trên bộ đơn VXL do thời gian luân chuyển CPU nhanh (*Thực tế chỉ một luồng thực hiện*)
- Sử dụng kiến trúc nhiều vi xử lý
  - Các luồng chạy song song thực sự trên các bộ VXL khác nhau.

Lợi ích của lập trình đa luồng → Ví dụ

Tính toán trên vector

```
for(k = 0; k < n; k++){
    a[k] = b[k] * c[k];
}
```



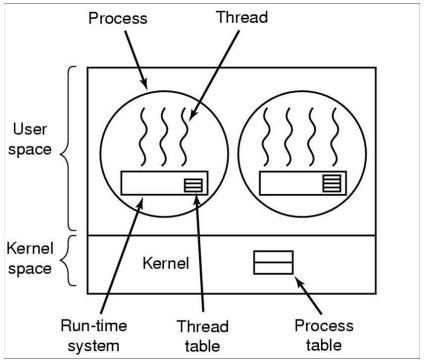
Mô hình đa luồng

```
void fn(a,b)
   for(k = a; k < b; k++){
        a[k] = b[k] * c[k];
   }
void main(){
  CreateThread(fn(0, n/4));
  CreateThread(fn(n/4, n/2));
  CreateThread(fn(n/2, 3n/4));
  CreateThread(fn(3n/4, n));
  }
```

Câu hỏi

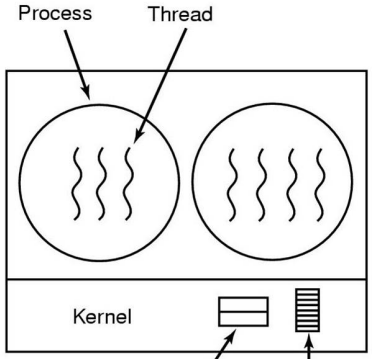
Tạo 4 tiến trình-*CreateProcess()* thay cho 4 luồng-*CreateThread()*

Cài đặt luồng



Cài đặt trong không gian người dùng

Cài đặt trong không gian nhân



Luồng người dùng (User -Level Threads)

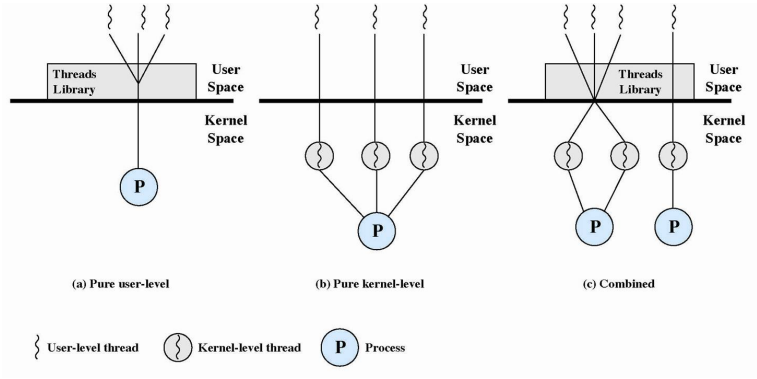
- Quản lý các luồng được thực hiện bởi chương trình ứng dụng
- Nhân hệ thống không biết gì về sự tồn tại luồng
  - Điều phối tiến trình như một đơn vị duy nhất
  - Gán cho mỗi tiến trình một trạng thái duy nhất
    - Sẵn sàng, chờ đợi, thực hiện,..
- Chương trình ứng dụng được lập trình theo mô hình đa luồng bởi sử dụng *thư viện luồng*
  - Thư viện hỗ trợ tạo, hủy bỏ, truyền thông điệp giữa các luồng, điều phối, lưu trữ, khôi phục trạng thái (*context*) luồng ,..
- Ưu điểm
  - Nhanh chóng trong tạo và quản lý luồng
- Nhược điểm
  - Khi một luồng rơi vào trạng thái chờ đợi, tất cả các luồng trong cùng tiến trình bị chờ đợi theo ⇒Không tận dụng được ưu điểm của mô hình lập trình đa luồng

Luồng mức hệ thống (Kernel - Level threads)

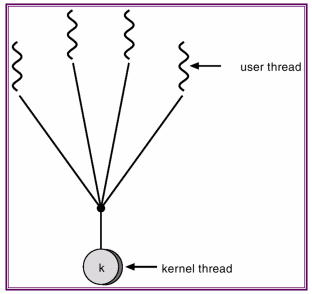
- Nhân duy trì thông tin về tiến trình và các luồng
- Quản lý luồng được thực hiện bởi nhân
  - Không tồn tại các mã quản lý luồng trong ứng dụng
  - Điều phối luồng được thực hiện bởi nhân, dựa trên các luồng
- Nhược điểm:
  - Chậm trong tạo và quản lý luồng
- Ưu điểm:
  - Một luồng chờ đợi vào ra, không ảnh hưởng tới luồng khác
  - Trong môi trường đa VXL, nhân có thể điều phối các luồng cho các VXL khác nhau
- Hệ điều hành: Windows NT/2000/XP, Linux, OS/2,..

Giới thiệu

Nhiều hệ thống hỗ trợ cả luồng mức người dùng và luồng mức hệ thống ⇒ Nhiều mô hình đa luồng khác nhau

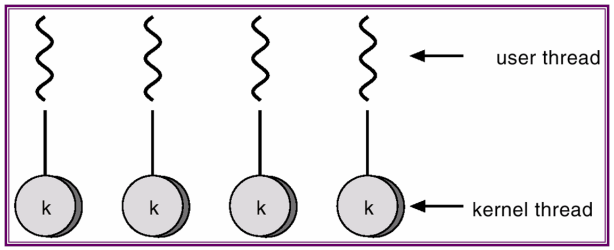


Mô hình nhiều-một



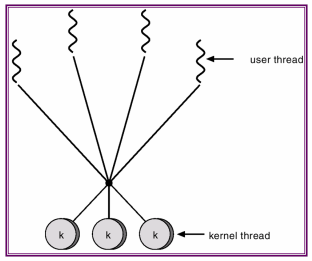
- Ánh xạ nhiều luồng mức người dùng tới một luồng mức hệ thống
- Quản lý luồng được thực hiện trong không gian người dùng
  - Hiệu quả
  - Cho phép tạo nhiều luồng tùy ý
  - Toàn bộ tiến trình sẽ bị khóa nếu một luồng bị khóa
- Không thể chạy song song trên các máy nhiều vi xử lý (*Chỉ một luồng có thể truy nhập nhân tại một thời điểm*)
- Dùng trong hệ điều hành không hỗ trợ luồng hệ thống

Mô hình một-một



- Ánh xạ mỗi luồng mức người dùng tới một luồng hệ thống
  - Cho phép thực hiện luồng khác khi một luồng bị chờ đợi
  - Cho phép chạy song song đa luồng trên máy nhiều vi xử lý
- Tạo luồng mức người dùng đòi hỏi tạo một luồng mức hệ thống tương ứng
  - Ảnh hưởng tới hiệu năng của ứng dụng
  - Chi phí cao ⇒ Giới hạn số luồng được hệ thống hỗ trợ
- Được sử dụng trong Window NT/2000/XP

Mô hình nhiều-nhiều



- Nhiều luồng mức người dùng ánh xạ tới một số nhỏ luồng mức hệ thống
- Số lượng luồng nhân có thể được xác định theo máy hoặc theo ứng dụng
  - VD: Được cấp nhiều luồng nhân hơn trên hệ thống nhiều VXL
- Có được ưu điểm của 2 mô hình trên
  - Cho phép tạo nhiều luồng mức ứng dụng theo yêu cầu
  - Các luồng nhân tương ứng có thể chạy song song trên hệ nhiều VXL
  - Một luồng bị khóa, nhân có thể cho phép luồng khác thực hiện
- Ví dụ: UNIX

Một số hàm với luồng trong WIN32 API

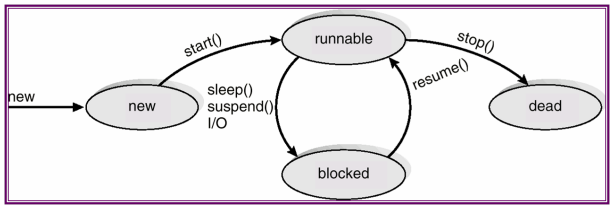
- HANDLE CreateThread(...);
  - **LPSECURITY_ATTRIBUTES** lpThreadAttributes,
    ⇒Trỏ tới cấu trúc an ninh: thẻ trả về có thể được kế thừa?
  - **DWORD** dwStackSize,
    ⇒Kích thước ban đầu của stack cho luồng mới
  - **LPTHREAD_START_ROUTINE** lpStartAddress,
    ⇒Trỏ tới hàm được thực hiện bởi luồng mới
  - **LPVOID** lpParameter,
    ⇒Trỏ tới các biến được gửi tới luồng mới (*tham số của hàm*)
  - **DWORD** dwCreationFlags,
    ⇒Phương pháp tạo luồng
    - CREATE_SUSPENDED : Luồng ở trạng thái tạm ngừng
    - 0: Luồng được thực hiện ngay lập tức
  - **LPDWORD** lpThreadId
    ⇒Biến ghi nhận định danh luồng mới
- Kết quả trả về: Thẻ của luồng mới hoặc giá trị NULL nếu không tạo được luồng mới

Ví dụ

```
#include <windows.h>
#include <stdio.h>
void Routine(int *n){
   printf("My argument is %d\n", &n);
}
int main(){
   int i, P[5];     DWORD Id;
   HANDLE hHandles[5];
   for (i=0;i < 5;i++) {
      P[i] = i;
      hHandles[i] = CreateThread(NULL,0,
             (LPTHREAD_START_ROUTINE)Routine,&P[i],0,&Id);
      printf("Thread %d was created\n",Id);
   }
   for (i=0;i < 5;i++)
       WaitForSingleObject(hHandles[i],INFINITE);
   return 0;
}
```

Java Threads

- Được cài đặt bởi
  - Mở rộng lớp Thread (*Thread class*)
  - Cài đặt giao diện có thể thực thi được (*Runnable interface*)
- Được quản lý bởi máy ảo Java (*Java Virtual Machine*)
- Các trạng thái có thể



- Tồn tại một phương thức **run()**, sẽ được thực hiện trên JVM
- Luồng được thực hiện bởi gọi phương thức **start()**
  - Cung cấp vùng nhớ và khởi tạo luồng mới trong máy ảo Java
  - Gọi tới phương thức **run()**

Ví dụ

```
class Sum extends Thread{
   int low, up, S;
   public Sum(int a, int b){
      low = a; up = b; S= 0;
      System.out.println("This is Thread "+this.getId());
   }
   public void run(){
   for(int i= low; i < up; i ++) S+= i;
      System.out.println(this.getId()+ " : " + S);
   }
}
public class Tester {
      public static void main(String[] args) {
         Sum T1 = new Sum(1,100); T1.start();
         Sum T2 = new Sum(10,200); T2.start();
         System.out.println("Main process terminated");
   }
}
```

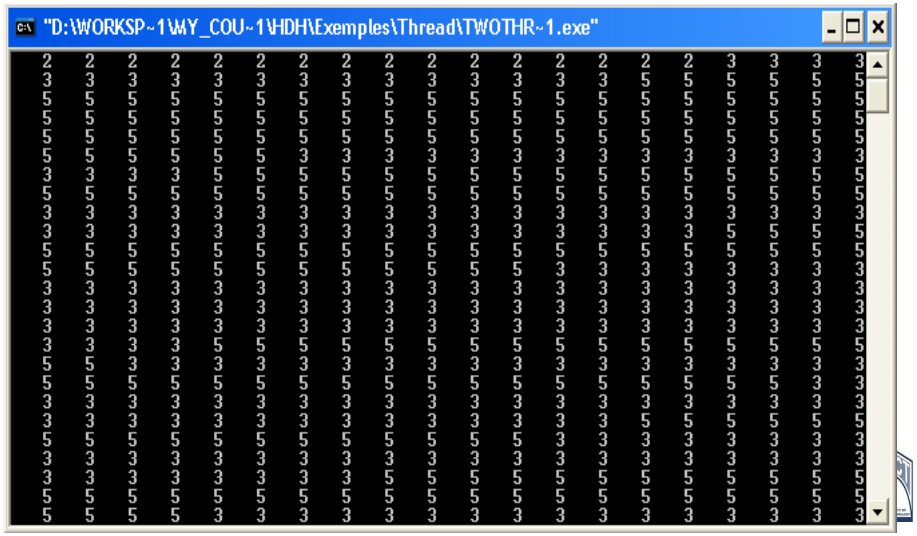
2 Luồng (Thread)

- Giới thiệu
- Mô hình đa luồng
- Cài đặt luồng với Windows
- Vấn đề đa luồng

Ví dụ

```
#include <windows.h>
#include <stdio.h>
int x = 0, y = 1;
void T1(){
    while(1){ x = y + 1; printf("%4d", x); }
}
void T2(){
    while(1){ y = 2;      y = y * 2; }
}

int main(){
 HANDLE h1, h2;   DWORD Id;
 h1=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)T1,NULL,0,&Id);
 h2=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)T2,NULL,0,&Id);
 WaitForSingleObject(h1,INFINITE);
 WaitForSingleObject(h2,INFINITE);
 return 0;
}
```

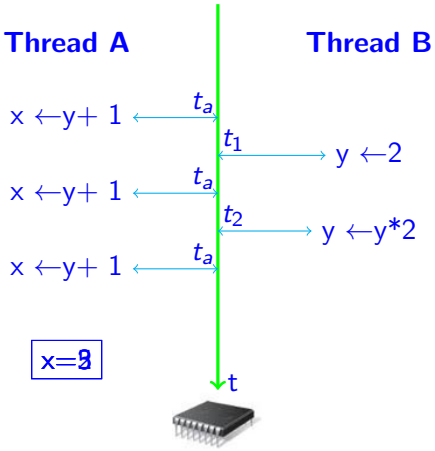
Kết quả thực hiện



Giải thích

Shared int y = 1	
Thread T_1	**Thread** T_2
x ←y+ 1	y ←2 y ← y * 2
x = ?	

Kết quả thực hiện các luồng song song phụ thuộc trật tự truy nhập biến dùng chung giữa chúng



Bài tập

- Cài đặt bài toán Producer-Consumer sử dụng khái niệm luồng
- Viết chương trình trao đổi thông báo giữa 2 máy (*chat*)

Nội dung chính

3 Điều phối CPU

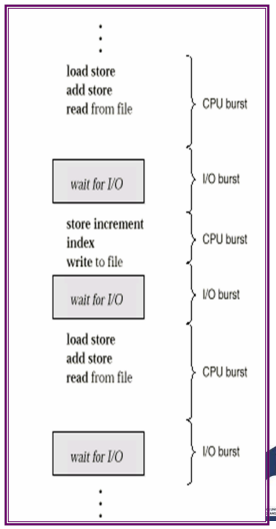
- Các khái niệm cơ bản
- Tiêu chuẩn điều phối
- Các thuật toán điều phối CPU
- Điều phối đa xử lý

Giới thiệu

- Hệ thống có một *processor* → Chỉ có một tiến trình được thực hiện tại một thời điểm
- Tiến trình được thực hiện (*chiếm dụng VXL*) cho tới khi phải chờ đợi một thao tác vào ra
  - Hệ đơn chương trình: CPU không được sử dụng ⇒Lãng phí
  - Hệ đa chương trình: cố gắng sử dụng CPU (*đang rảnh rỗi*) cho các tiến trình khác (*đang chờ đợi*)
    - Cần nhiều tiến trình sẵn sàng trong bộ nhớ tại một thời điểm
    - Khi một tiến trình phải chờ, hệ điều hành lấy lại processor để phân cho tiến trình khác
- Điều phối *processor* quan trong với hệ điều hành đa nhiệm
  - Luân chuyển CPU giữa các tiến trình → khai thác hệ thống hiệu quả hơn
- Điều phối processor là nền tảng trong thiết kế hệ điều hành

Chu kỳ thực hiện CPU - I/O

- Tiến trình là chuỗi luân phiên giữa chu kỳ tính toán và chờ đợi vào/ra
  - Bắt đầu bởi chu kỳ tính toán
  - Tiếp theo chu kỳ đợi vào/ra
  - Tính toán→ đợi vào/ra → tính toán → đợi vào/ra → ...
  - Kết thúc: Tính toán (*yêu cầu hệ thống kết thúc thực hiện*)
- Phân biệt các kiểu tiến trình
  - Dựa trên sự phân bổ thời gian cho các chu kỳ CPU & vào/ra
    - Tiến trình tính toán (*CPU-bound process*) có vài chu kỳ CPU dài
    - Tiến trình vào ra (*I/0-bound process*) có nhiều chu kỳ CPU ngắn
  - Để chọn giải thuật điều phối thích hợp



Bộ điều phối CPU

- Lựa chọn một trong số các tiến trình đang sẵn sàng trong bộ nhớ và cung cấp CPU cho nó
  - Các tiến trình phải sắp hàng trong hàng đợi
    - Hàng đợi FIFO, Hàng đợi ưu tiên, DSLK đơn giản ...
- Quyết định điều phối CPU xảy ra khi tiến trình
  1. Chuyển từ trạng thái thực hiện sang trạng thái chờ đợi (*y/c vào/ra*)
  2. Chuyển từ trạng thái thực hiện sang trạng thái sẵn sàng (*hết thời gian sử dụng CPU → ngắt thời gian*)
  3. Chuyển từ trạng thái chờ đợi sang trạng thái sẵn sàng (*hoàn thành vào/ra*)
  4. Tiến trình kết thúc
- Ghi chú
  - Trường hợp 1&4
    ⇒Điều phối không trưng dụng (*non-preemptive*)
  - Trường hợp khác
    ⇒Điều phối trưng dụng (*preemptive*)

Điều phối trưng dụng và không trưng dụng

- Điều phối không trưng dụng
  - Tiến trình chiếm CPU cho tới khi giải phóng bởi
    - Kết thúc nhiệm vụ
    - Chuyển sang trạng thái chờ đợi
  - Không đòi hỏi phần cứng đặc biệt (*đồng hồ*)
  - Ví dụ: DOS, Win 3.1, Macintosh
- Điều phối trưng dụng
  - Tiến trình chỉ được phép thực hiện trong khoảng thời gian
  - Kết thúc khoảng thời gian được định nghĩa trước, ngắt thời gian xuất hiện, bộ điều vận (*dispatcher*) được kích hoạt để quyết định hồi phục lại tiến trình hay lựa chọn tiến trình khác
  - Bảo vệ CPU khỏi các tiến trình "*đói-CPU*"
  - Vấn đề dữ liệu dùng chung
    - Tiến trình 1 đang cập nhật DL thì bị mất CPU
    - Tiến trình 2, được giao CPU và đọc DL đang cập nhật
  - Ví dụ: Hệ điều hành đa nhiệm WinNT, UNIX

3 Điều phối CPU

Tiêu chuẩn điều phối I

- **Sử dụng CPU** (*Lớn nhất*)
  - Mục đích của điều độ là làm CPU hoạt động nhiều nhất có thể
  - Độ sử dụng CPU thay đổi từ 40% (*hệ thống tải nhẹ*) đến 90% (*hệ thống tải nặng*).
- **Thông lượng (throughput)** (*Lớn nhất*)
  - Số lượng tiến trình hoàn thành trong một đơn vị thời gian
    - Các tiến trình dài: 1 tiến trình/giờ
    - Các tiến trình ngắn: 10 tiến trình/giây
- **Thời gian hoàn thành** (*Nhỏ nhất*)
  - Khoảng thời gian từ thời điểm gửi đến hệ thống tới khi quá trình hoàn thành
    - Thời gian chờ đợi để đưa tiến trình vào bộ nhớ
    - Thời gian chờ đợi trong hàng đợi sẵn sàng
    - Thời gian chờ đợi trong hàng đợi thiết bị
    - Thời gian thực hiện thực tế

Tiêu chuẩn điều phối II

- **Thời gian chờ đợi** (*Nhỏ nhất*)
  - Tổng thời gian chờ trong hàng đợi sẵn sàng (*Giải thuật điều độ CPU không ảnh hưởng tới các tiến trình đang thực hiện hay đang đợi thiết bị vào ra*)
- **Thời gian đáp ứng** (*Nhỏ nhất*)
  - Từ lúc gửi câu hỏi cho tới khi câu trả lời đầu tiên được tạo ra
    - Tiến trình có thể tạo kết quả ra từng phần
    - Tiến trình vẫn tiếp tục tính toán kết quả mới trong khi kết quả cũ được gửi tới người dùng

- Giả thiết: Các tiến trình chỉ có một chu kỳ tính toán (ms)
- Đo đạc: Thời gian chờ đợi trung bình

3 Điều phối CPU

- Các khái niệm cơ bản
- Tiêu chuẩn điều phối
- Các thuật toán điều phối CPU
- Điều phối đa xử lý

Đến trước phục vụ trước (FCFS: First Come, First Served)

- Nguyên tắc:
  - Tiến trình được quyền sử dụng CPU theo trình tự xuất hiện
  - Tiến trình sở hữu CPU tới khi kết thúc hoặc chờ đợi vào ra
- Ví dụ

Tiến trình	Thời gian
P_1	24
P_2	3
P_3	3

- Đặc điểm
  - Đơn giản, dễ thực hiện
  - Tiến trình ngắn phải chờ đợi như tiến trình dài
    - Nếu P_1 thực hiện sau cùng ?

Công việc ngắn trước (SJF: Shortest Job First)

- Nguyên tắc
  - Mỗi tiến trình lưu trữ thời gian của chu kỳ sử dụng CPU tiếp theo
  - Tiến trình có thời gian sử dụng CPU ngắn nhất sẽ sở hữu CPU
  - Hai phương pháp
    - Không trưng dụng CPU
    - Có trưng dụng CPU (*SRTF: Shortest Remaining Time First*)
- Ví dụ

Tiến trình	Thời gian	Thời điểm đến
P_1	8	0.0
P_2	4	1.0
P_3	9	2.0
P_4	5	3.0

- Đặc điểm
  - SJF (SRTF) là tối ưu: Thời gian chờ đợi trung bình nhỏ nhất
  - Không thể biết chính xác thời gian của chu kỳ sử dụng CPU
    - Dự báo dựa trên những giá trị trước đó

Điều phối có ưu tiên (Priority Scheduling)

- Nguyên tắc
  - Mỗi tiến trình gắn với một sô hiệu ưu tiên (số nguyên)
  - CPU sẽ được phân phối cho tiến trình có độ ưu tiên cao nhất
  - **SJF**: độ ưu tiên gắn liền với thời gian thực hiện
  - Hai phương pháp
    - Không trưng dụng CPU
    - Có trưng dụng CPU
- Ví dụ

Tiến trình	Thời gian	Độ ưu tiên
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Vấn đề "**Nạn đói**": Tiến trình có độ ưu tiên thấp phải chờ đợi lâu (*thậm chí không được thực hiện*)
- Giải pháp tăng dần độ ưu tiên **tt** theo t/gian trong hệ thống

Vòng tròn (RR: Round Robin Scheduling)

- Nguyên tắc
  - Mỗi tiến trình được cấp một lượng tử thời gian τ để thực hiện
  - Khi hết thời gian, tiến trình bị trưng dụng processor và được đưa vào cuối hàng đợi sẵn sàng
  - Nếu có n tiến trình, thời gian chờ đợi nhiều nhất $(n-1)\tau$
- Ví dụ

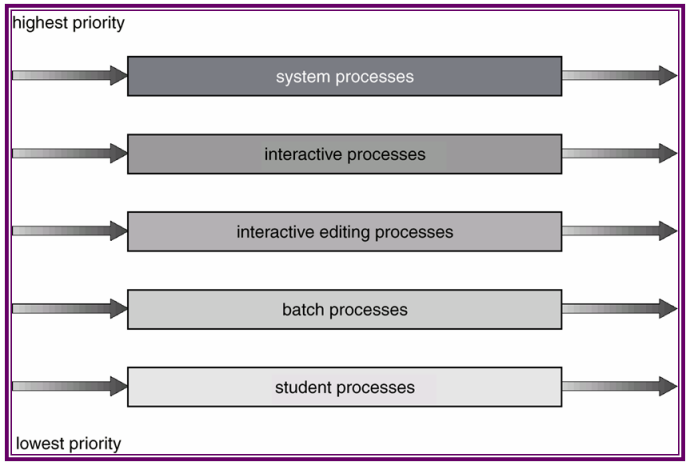
Tiến trình	Thời gian
P_1	24
P_2	3
P_3	3

  Lượng tử thời gian $\tau = 4 \Rightarrow \bar{t}_{wait} = 5.66$
- Vấn đề: Lựa chọn lượng tử thời gian τ
  - τ lớn: FCFS
  - τ nhỏ: Hãy phải luân chuyển CPU
  - Thông thường τ = 10-100ms

Điều phối hàng đợi đa mức (Multilevel Queue Scheduling)

- Hàng đợi sẵn sàng được phân chia thành nhiều hàng đợi nhỏ
- Tiến trình được ấn định **cố định** cho một hàng đợi
  - Dựa vào tính chất như độ ưu tiên, kiểu tiến trình..
- Mỗi hàng đợi sử dụng thuật toán điều độ riêng
- Cần điều phối giữa các hàng đợi
  - Điều phối có trưng dụng, độ ưu tiên cố định
    - Tiến trình hàng đợi độ ưu tiên thấp chỉ được thực hiện khi các hàng đợi có độ ưu tiên cao rỗng
    - Tiến trình độ ưu tiên mức cao, trưng dụng tiến trình độ ưu tiên mức thấp
    - Có thể gặp tình trạng *starvation*
  - Phân chia thời gian giữa các hàng đợi
    - Hàng đợi cho *foreground process*, chiếm 80% thời gian CPU cho RR
    - Hàng đợi cho *background process*, chiếm 20% thời gian CPU cho FCFS

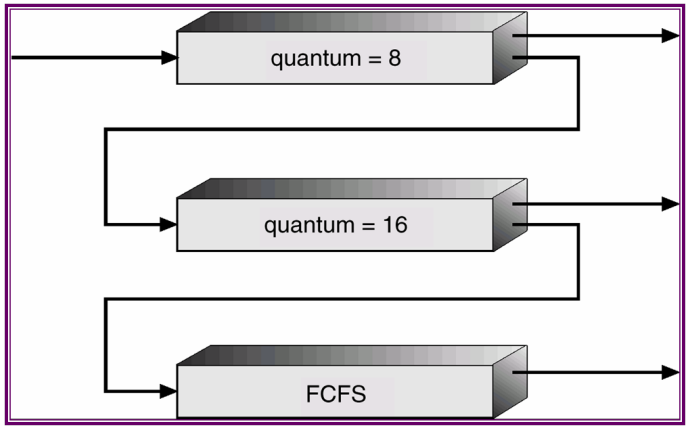
Điều phối hàng đợi đa mức → Ví dụ



Hàng đợi hồi tiếp đa mức (Multilevel Feeedback Queue)

- Cho phép các tiến trình được dịch chuyển giữa các hàng đợi
- Phân chia tiến trình theo đặc điểm sử dụng VXL
  - Nếu dùng quá nhiều thời gian của VXL → Chuyển xuống hàng đợi có độ ưu tiên thấp
  - Tiến trình vào ra nhiều → hàng đợi có độ ưu tiên cao
  - Tiến trình đợi quá lâu tại hàng đợi có độ ưu tiên thấp → Chuyển lên hàng đợi độ ưu tiên cao
    - Ngăn ngừa tình trạng "*đói CPU*"
- Được định nghĩa bởi các tham số
  - Số hàng đợi
  - Thuật toán điều độ cho mỗi hàng đợi
  - Điều kiện để tiến trình được chuyển lên/xuống hàng đợi có độ ưu tiên cao/thấp hơn
  - Phương pháp xác định một hàng đợi khi tiến trình cần phục vụ

Hàng đợi hồi tiếp đa mức → Ví dụ



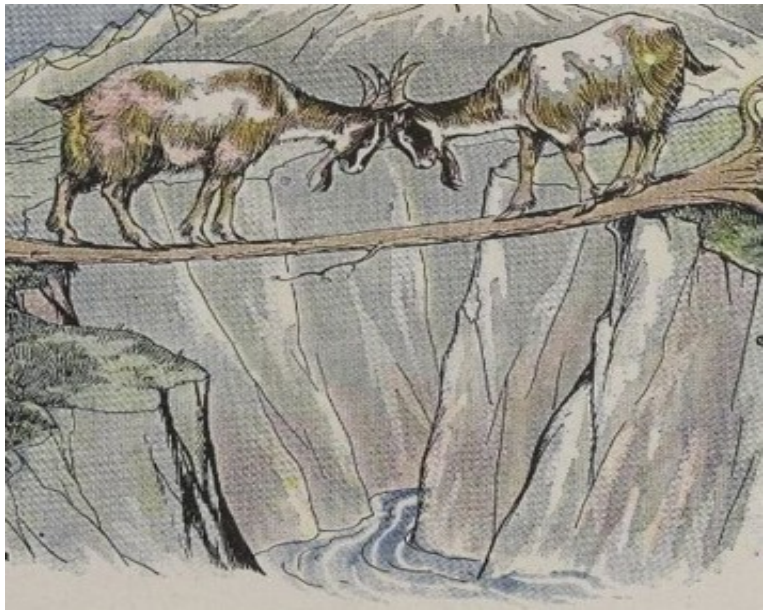
Vấn đề

- Điều phối phức tạp hơn so với trường hợp có một VXL
- Vấn đề chia sẻ tải
  - Mỗi VXL có một hàng đợi sẵn sàng riêng
    - Tồn tại VXL rảnh rỗi với hàng đợi rỗng trong khi VXL khác phải tính toán nhiều
  - Hàng đợi sẵn sàng dùng chung
    - Vấn đề dùng chung cấu trúc dữ liêu (*hàng đợi*) :
      →Một tiến trình được lựa chọn bởi 2 processors hoặc
      →Một tiến trình bị thất lạc trên hàng đợi
- Đa xử lý không đối xứng
  - Chỉ có một processor truy nhập hàng đợi hủy bỏ vấn đề dùng chung cơ sở dữ liêu
  - Có thể tắc nghẽn tại một processor

Bài tập

- Viết chương trình mô phỏng hàng đợi hồi tiếp đa mức

Kết luận



(*Nguồn: http://sedition.com/a/393*)

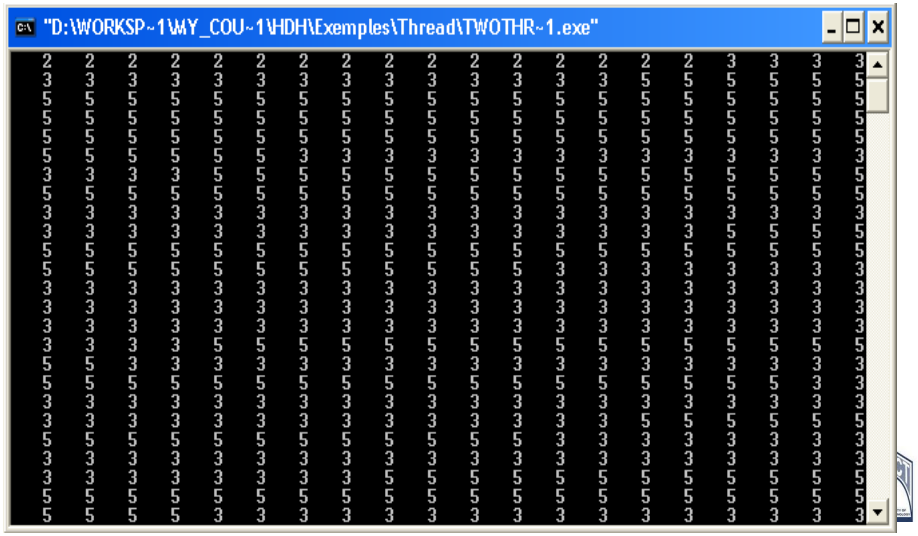
4 Tài nguyên găng và điều độ tiến trình

- Khái niệm tài nguyên găng
- Phương pháp khóa trong
- Phương pháp kiểm tra và xác lập
- Kỹ thuật đèn báo
- Ví dụ về đồng bộ tiến trình
- Công cụ điều độ cấp cao

Ví dụ: Luồng song song

```
#include <windows.h>
#include <stdio.h>
int x = 0, y = 1;
void T1(){
    while(1){ x = y + 1; printf("%4d", x); }
}
void T2(){
    while(1){ y = 2;       y = y * 2; }
}
int main(){
  HANDLE h1, h2;   DWORD ThreadId;
  h1 = CreateThread(NULL,0,T1, NULL,0,&ThreadId);
  h2 = CreateThread(NULL,0,T2, NULL,0,&ThreadId);
  WaitForSingleObject(h1,INFINITE);
  WaitForSingleObject(h2,INFINITE);
  return 0;
}
```

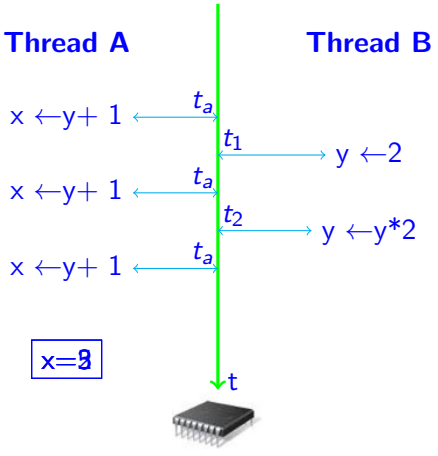
Kết quả thực hiện



Luồng song song

Shared int y = 1	
Thread T_1	**Thread** T_2
x ←y+ 1	y ←2 y ← y * 2
x = ?	

Kết quả thực hiện các luồng song song phụ thuộc trật tự truy nhập biến dùng chung giữa chúng



Producer-Consumer

Producer

```
while(1){
  /*produce an item */
  while(Counter==BUFFER_SIZE);
  Buffer[IN] = nextProduced;
  IN = (IN+1)%BUFFER_SIZE;
  Counter++;
}
```

Consumer

```
while(1){
  while(Counter == 0);
  nextConsumed = Buffer[OUT];
  OUT=(OUT+1)%BUFFER_SIZE;
  Counter--;
  /*consume the item*/
}
```

Nhận xét

- *Producer* sản xuất một sản phẩm
- *Consumer* tiêu thụ một sản phẩm

⇒Số sản phẩm còn trong **Buffer** không thay đổi

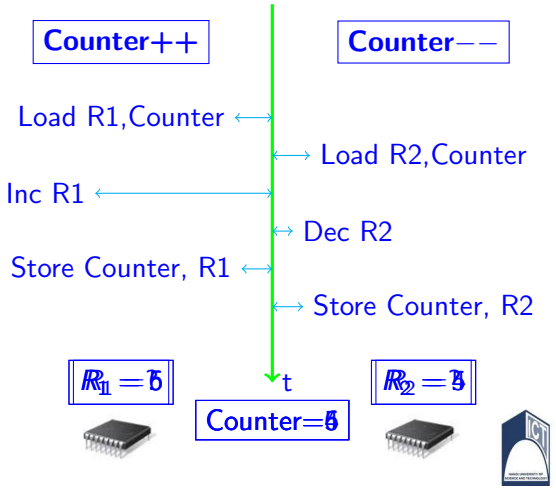
Producer-Consumer

Counter++

Load R1, Counter
Inc R1
Store Counter, R1

Counter−−

Load R2, Counter
Dec R2
Store Counter, R2



Định nghĩa

Tài nguyên

Tất cả những gì cần thiết cho thực hiện tiến trình

Tài nguyên găng

- Tài nguyên hạn chế về khả năng sử dụng chung
- Cần đồng thời cho nhiều tiến trình

Tài nguyên găng có thể là thiết bị vật lý hay dữ liệu dùng chung

Vấn đề

Dùng chung tài nguyên găng có thể dẫn đến không đảm bảo tính toàn vẹn dữ liệu
⇒ Đòi hỏi cơ chế đồng bộ hóa các tiến trình

Điều kiện cạnh tranh (Race condition)

- Tình trạng trong đó kết quả của việc nhiều tiến trình cùng truy nhập tới dữ liệu phân chia phụ thuộc vào trật tự của các truy nhập
  - Làm cho chương trình không xác định
- Ngăn ngừa điều kiện cạnh tranh được thực hiện bởi đồng bộ hóa (**synchronize**) các tiến trình thực hiện đồng thời
  - Chỉ một tiến trình truy nhập tới dữ liệu phân chia tại một thời điểm
    - Biến *counter* trong v/đề Producer-Consumer
  - Đoạn lệnh truy nhập tới dữ liệu phân chia trong các tiến trình phải thực hiện theo thứ tự xác định
    - VD Lệnh x←y+1 trong Thread T_1 chỉ thực hiện khi cả 2 lệnh của Thread T_2 đã thực hiện xong

Đoạn găng (Critical section)

- Đoạn găng (*chỗ hẹp*) là đoạn chương trình sử dụng tài nguyên găng
  - Đoạn chương trình thực hiện truy nhập và thao tác trên dữ liệu dùng chung
- Khi có nhiều tiến trình sử dụng tài nguyên găng thì phải điều độ
  - Mục đích: đảm bảo không có quá một tiến trình nằm trong đoạn găng

Yêu cầu của chương trình điều độ

- **Loại trừ lẫn nhau** (*Mutual Exclusion*) Mỗi thời điểm, tài nguyên găng không phải phục vụ một số lượng tiến trình vượt quá khả năng của nó
  - Một tiến trình đang thực hiện trong đoạn găng (*sử dụng tài nguyên găng*) $\Rightarrow$ Không một tiến trình nào khác được quyền vào đoạn găng
- **Tiến triển** (*Progress*) Tài nguyên găng còn khả năng phục vụ và tồn tại tiến trình muốn vào đoạn găng, thì tiến trình đó phải được sử dụng tài nguyên găng
- **Chờ đợi hữu hạn** (*Bounded Waiting*) Nếu tài nguyên găng hết khả năng phục vụ và vẫn tồn tại tiến trình muốn vào đoạn găng, thì tiến trình đó phải được xếp hàng chờ đợi và sự *chờ đợi là hữu hạn*

Quy ước

- Có 2 tiến trình P_1&P_2 thực hiện đồng thời
- Các tiến trình dùng chung một tài nguyên găng
- Mỗi tiến trình đặt đoạn găng ở đầu, tiếp theo là phần còn lại
  - Tiến trình phải xin phép trước khi vào đoạn găng *{phần vào}*
  - Tiến trình khi thoát khỏi đoạn găng thực hiện *{phần ra}*
- Cấu trúc tổng quát của một tiến trình

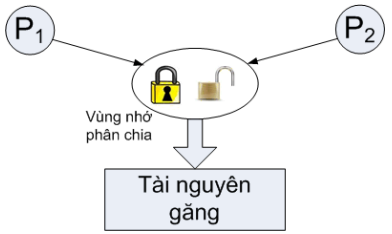
```
do{
    Phần vào
        {Đoạn găng của tiến trình}
    Phần ra
        {Phần còn lại của tiến trình}
}while(1);
```

Phân loại các phương pháp

- **Các công cụ cấp thấp**
  - Phương pháp khóa trong
  - Phương pháp kiểm tra và xác lập
  - Kỹ thuật đèn báo
- **Các công cụ cấp cao**
  - Monitor

4 Tài nguyên găng và điều độ tiến trình

Nguyên tắc



- Mỗi t/trình dùng một byte trong vùng nhớ chung làm khóa
  - Tiến trình vào đoạn găng, đóng khoá (byte khóa: *true*)
  - Tiến trình thoát khỏi đoạn găng, mở khóa (byte khóa: *false*)
- Tiến trình muốn vào đoạn găng: kiểm tra khóa của tiến trình còn lại
  - Đang khóa ⇒ Đợi
  - Đang mở ⇒ Được quyền vào đoạn găng

Thuật toán điều độ

- Share var C_1, C_2 **Boolean** // *Các biến dùng chung làm khóa*
- Khởi tạo $C_1 = C_2 = false$ // *Tài nguyên găng đang tự do*

Process P_1

do{
  while($C_2 == true$);
  $C_1 \leftarrow true$; $C_1 \leftarrow true$;
  while($C_2 == true$);
  {Đoạn găng của tiến trình P_1}
  $C_1 \leftarrow false$;
  {Phần còn lại của tiến trình P_1}
}**while(1)**;

Process P_2

do{
  while($C_1 == true$);
  $C_2 \leftarrow true$; $C_2 \leftarrow true$;
  while($C_1 == true$);
  {Đoạn găng của tiến trình P_2}
  $C_2 \leftarrow false$;
  {Phần còn lại của tiến trình P_2}
}**while(1)**;

Nhận xét

- Điều độ chưa hợp lý
  - Hai t/trình yêu cầu tài nguyên tại một thời điểm
    - Vấn đề loại trừ lẫn nhau (*trường hợp 1*)
    - Vấn đề tiến triển (*trường hợp 2*)
- Nguyên nhân: Do tách rời giữa
  - Kiểm tra quyền vào đoạn găng
  - Xác lập quyền sử dụng tài nguyên găng

Thuật toán Dekker

Sử dụng biến *turn* để chỉ ra tiến trình được quyền ưu tiên

Process P_1

```
do{
  C1 ← true;
  while(C2==true){
    if(turn == 2){
      C1 ← false;
      while(turn ==2);
      C1 ← true;
    }
  }
  {Đoạn găng của tiến trình P1}
  turn = 2;
  C1 ← false;
  {Phần còn lại của tiến trình P1}
}while(1);
```

Process P_2

```
do{
  C2 ← true;
  while(C1==true){
    if(turn == 1){
      C2 ← false;
      while(turn ==1);
      C2 ← true;
    }
  }
  {Đoạn găng của tiến trình P2}
  turn = 1;
  C2 ← false;
  {Phần còn lại của tiến trình P2}
}while(1);
```

Nhận xét

- Điều độ hợp lý cho mọi trường hợp
- Không đòi hỏi sự hỗ trợ đặc biệt của phần cứng nên có thể thực hiện bằng ngôn ngữ bất kỳ
- Quá phức tạp khi số tiến trình và số tài nguyên tăng lên
- Phải chờ đợi tích cực (*busy waiting*) trước khi vào đoạn găng
  - Khi chờ đợi vẫn phải thực hiện kiểm tra quyền vào đoạn găng
    - Lãng phí thời gian của processor

Ghi chú: Thuật toán có thể thực hiện sai trong một số trường hợp

- CPU cho phép thực hiện các lệnh không đúng trật tự
- Chương trình dịch thực hiện tối ưu hóa khi sinh mã
  - Các mã bất biến bên trong vòng lặp được đưa ra ngoài

Nguyên tắc

- Sử dụng sự hỗ trợ từ phần cứng
- Phần cứng cung cấp các câu lệnh xử lý không tách rời
  - Kiểm tra và thay đổi nội dung của một word

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

  - Hoán đổi nội dung của 2 word khác nhau

```
void Swap(VAR boolean , VAR boolean b) {
  boolean temp = a;
  a = b;
  b = temp;
}
```

- Xử lý không tách rời (*atomically*)
  - Khối lệnh không thể bị ngắt trong khi đang thực hiện
  - Được gọi đồng thời, sẽ được thực hiện theo thứ tự bất kỳ

Thuật toán với lệnh TestAndSet

- Biến phân chia **Boolean: Lock**: trạng thái của tài nguyên:
  - Bị khóa (*Lock=true*)
  - Tự do (*Lock=false*)
- Khởi tạo: *Lock* = *false* ⇒ Tài nguyên tự do
- Thuật toán cho tiến trình P_i

```
do{
        while(TestAndSet(Lock));
            {Đoạn găng của tiến trình}
        Lock = false;
            {Phần còn lại của tiến trình}
}while(1);
```

Thuật toán với lệnh Swap

- Biến phân chia **Lock** cho biết trạng thái tài nguyên
- Biến địa phương cho mỗi tiến trình: **Key**: Boolean
- Khởi tạo: *Lock* = *false* ⇒ Tài nguyên tự do
- Thuật toán cho tiến trình P_i

```
do{
        key = true;
        while(key == true)
              swap(Lock, Key);
            {Đoạn găng của tiến trình}
        Lock = false;
            {Phần còn lại của tiến trình}
}while(1);
```

Nhận xét

- Đơn giản, không phức tạp khi số tiến trình và số đoạn găng tăng lên
- Các tiến trình phải chờ đợi tích cực trước khi vào đoạn găng
  - Luôn kiểm tra xem tài nguyên găng đã được giải phóng chưa ⇒ Sử dụng Processor không hiệu quả
- Không đảm bảo yêu cầu chờ đợi hữu hạn
  - Tiến trình được vào đoạn găng tiếp theo, sẽ phụ thuộc thời điểm giải phóng tài nguyên của tiến trình đang chiếm giữ ⇒ Cần khắc phục

Thuật toán cho nhiều tiến trình

- **Nguyên tắc:** Tiến trình khi ra khỏi đoạn găng sẽ tìm tiến trình đang đợi để trao tài nguyên cho nó
- Dùng biến toàn cục Waiting[n] lưu trạng thái mỗi tiến trình
- Sơ đồ cho tiến trình P_i

```
do{
        Waiting[i] = true;
        While(Waiting[i] && TestAndSet(Lock)) ;
        Waiting[i] = False;
    {Đoạn găng của tiến trình}
        j = (i+1) % N;
        while ( (j !=i) && (! Waiting[j])) j = (j+1) % N;
        if (j == i) Lock = false;
        else Waiting[j] = false;
    {Phần còn lại của tiến trình}
}while(1);
```

4 Tài nguyên găng và điều độ tiến trình

- Khái niệm tài nguyên găng
- Phương pháp khóa trong
- Phương pháp kiểm tra và xác lập
- Kỹ thuật đèn báo
- Ví dụ về đồng bộ tiến trình
- Công cụ điều độ cấp cao

Đèn báo (Semaphore)

- Là một biến nguyên S, khởi tạo bằng *khả năng phục vụ* của tài nguyên nó điều độ
  - Số tài nguyên có thể phục vụ tại một thời điểm (*VD 3 máy in*)
  - Số đơn vị tài nguyên có sẵn (*VD 10 chỗ trống trong buffer*)
- Chỉ có thể thay đổi giá trị bởi 2 thao tác cơ bản P và V
  - Thao tác P(S) (*wait(S)*)

```
wait(S) {
   while(S ≤ 0) no-op;
   S − −;
}
```

  - Thao tác V(S) (*signal(S)*)

```
signal(S) {
   S + +;
}
```

  - Các thao tác P(S) và V(S) xử lý không tách rời
- Đèn báo là công cụ điều độ tổng quát

Sử dụng đèn báo I

- Điều độ nhiều tiến trình qua đoạn găng
  - Sử dụng biến phân chia mutex kiểu Semaphore
  - Khởi tạo mutex bằng 1
  - Thuật toán cho tiến trình P_i

```
do{
wait(mutex);
    {Đoạn găng của tiến trình}
signal(mutex)
    {Phần còn lại của tiến trình}
}while(1);
```

Sử dụng đèn báo II

- Điều độ thứ tự thực hiện bên trong các tiến trình
  - Hai tiến trình P_1 và P_2 thực hiện đồng thời
    - P_1 chứa lệnh S_1, P_2 chứa lệnh S_2 .
    - Yêu cầu S_2 được thực hiện chỉ khi S_1 thực hiện xong
  - Sử dụng đèn báo *synch* được khởi tạo giá trị 0
  - Đoạn mã cho P_1 và P_2

P_1	P_2
Phần đầu	Phần đầu
S_1	wait(synch)
Signal(synch)	S_2
Phần cuối	Phần cuối

Hủy bỏ chờ đợi tích cực

- Sử dụng 2 thao tác đơn giản
  - **block()** Ngừng tạm thời tiến trình đang thực hiện
  - **wakeup(P)** Thực hiện tiếp t/trình **P** dừng bởi lệnh *block()*
- Khi tiến trình gọi **P(S)** và đèn báo S không dương
  - Tiến trình phải dừng bởi gọi tới câu lệnh *block()*
  - Lệnh *block()* đặt tiến trình vào hàng đợi gắn với đèn báo S
  - Hệ thống lấy lại CPU giao cho tiến trình khác (*điều phối CPU*)
  - Tiến trình chuyển sang trạng thái chờ đợi (*waiting*)
  - Tiến trình nằm trong hàng đợi đến khi tiến trình khác thực hiện thao tác **V(S)** trên cùng đèn báo S
- Tiến trình đưa ra lời gọi **V(S)**
  - Lấy một tiến trình trong hàng đợi ra (nếu có)
  - Chuyển tiến trình lấy ra từ trạng thái chờ đợi sang trạng thái sẵn sàng và đặt lên hàng đợi sẵn sàng bởi gọi tới *wakeup(P)*
  - Tiến trình mới sẵn sàng có thể trưng dụng CPU từ tiến trình đang thực hiên nếu thuật toán điều phối CPU cho phép

Cài đặt đèn báo

Semaphore S

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

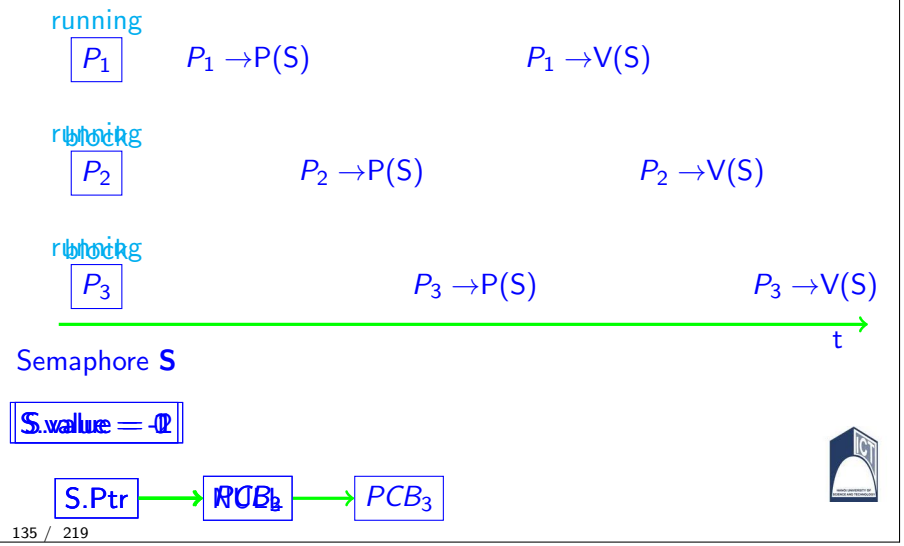
wait(S)/P(S)

```
void wait(Semaphore S) {
  S.value−−;
  if(S.value < 0) {
     Thêm tiên trình vào S.Ptr
     block();
  }
}
```

signal(S)/V(S)

```
void signal(Semaphore S) {
 S.value++;
 if(S.value ≤ 0) {
   Lấy ra tiến trình P từ S.Ptr
   wakeup(P);
 }
}
```

Ví dụ điều độ



Nhận xét

- Dễ dàng áp dụng cho các hệ thống phức tạp
- Không tồn tại hiện tượng chờ đợi tích cực
- Hiệu quả sử dụng phụ thuộc vào người dùng

P(S) {Đoạn găng} V(S)	V(S) {Đoạn găng} P(S)	P(S) {Đoạn găng} P(S)
Điều độ đúng	Nhầm vị trí	Nhầm lệnh

- Các phép xử lý P(S) và V(S) là không phân chia được
  ⇒bản thân P(S) và V(S) cũng là 2 tài nguyên găng
  ⇒Cũng cần điều độ.
  - Hệ thống một VXL: Cấm ngắt khi thực hiện wait(), signal()
  - Hệ thống nhiều vi xử lý
    - Không thể cấm ngắt trên VXL khác
    - Có thể dùng phương pháp khoa trong ⇒ Hiện tượng chờ đợi tích cực, nhưng thời gian chờ đợi ngắn (*10 lệnh*)

Đối tượng Semaphore trong WIN32 API

- **CreateSemaphore(...)** : Tạo một Semaphore
  - **LPSECURITY_ATTRIBUTES** lpSemaphoreAttributes
    ⇒ Trỏ tới cấu trúc an ninh, thẻ trả về được kế thừa?
  - **LONG** InitialCount,
    ⇒ Giá trị khởi tạo cho đối tượng Semaphore
  - **LONG** MaximumCount,
    ⇒ Giá trị lớn nhất của đối tượng Semaphore
  - **LPCTSTR** lpName
    ⇒ Tên của đối tượng Semaphore

  Ví dụ CreateSemaphore(NULL,0,1,NULL);
  Trả về thẻ (*HANDLE*) của đối tượng Semaphore hoặc NULL
- **WaitForSingleObject**(HANDLE h, DWORD time)
- **ReleaseSemaphore** (...)
  - **HANDLE** hSemaphore, ⇐Thẻ của đối tượng Semaphore
  - LONG lReleaseCount, ⇐Giá trị được tăng lên,
  - LPLONG lpPreviousCount ⇐Giá trị trước đó

  Ví dụ: *ReleaseSemaphore(S, 1, NULL);*

Ví dụ 1

```
#include <windows.h>
#include <stdio.h>
int x = 0, y = 1;
HANDLE S1, S2;
void T1();
void T2();
int main(){
   HANDLE h1, h2;
   DWORD ThreadId;
   S1 = CreateSemaphore( NULL,0, 1,NULL);
   S2 = CreateSemaphore( NULL,0, 1,NULL);
   h1 = CreateThread(NULL,0,T1, NULL,0,&ThreadId);
   h2 = CreateThread(NULL,0,T2, NULL,0,&ThreadId);
   getch();
   return 0;
}
```

Ví dụ 1 (tiếp tục)

```
void T1(){
    while(1){
        WaitForSingleObject(S1,INFINITE);
        x = y + 1;
        ReleaseSemaphore(S2,1,NULL);
        printf("%4d",x);
    }
}
void T2(){
    while(1){
        y = 2;
        ReleaseSemaphore(S1,1,NULL);
        WaitForSingleObject(S2,INFINITE);
        y = 2 * y;
    }
}
```

Ví dụ 2

```
#include <windows.h>
#include <stdio.h>

#define Max 5000000
#define numThreads 10
int Counter;
HANDLE S;

void counterThread(){
    int i, temp;
    for(i=0; i < Max; i++) {
       WaitForSingleObject(S,INFINITE); //P(S)
       temp = Counter;
       temp = temp + 1 ;
       Counter = temp;
      ReleaseSemaphore(S,1,NULL); //V(S)
   }
}
```

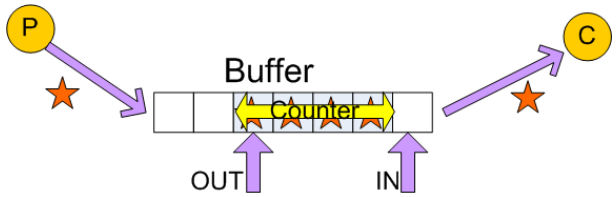
Ví dụ 2 (tiếp tục)

```
int main(){
  HANDLE hThreads[numThreads];
  DWORD Id;
  int i;

  S = CreateSemaphore(NULL,1,1,NULL);
  for(i=0; i < numThreads;i++)
    hThreads[i] = CreateThread(NULL,0,
      (LPTHREAD_START_ROUTINE)counterThread,NULL,0,&Id);
  WaitForMultipleObjects(numThreads, hThreads,
                                    TRUE, INFINITE);

  printf("\nKet qua : %d\n", Counter);
  return 0;
}
```

Một số bài toán kinh điển

- Người sản xuất-người tiêu thụ (*Producer-Consumer*)
- Triết gia ăn tối (*Dining Philosophers*)
- Người đọc và biên tập viên (*Readers-Writers*)
- Người thợ cắt tóc ngủ gật (*Sleeping Barber*)
- Bathroom Problem
- Đồng bộ theo Barriers
- ...

Vấn đề sản xuất-tiêu thụ 1




```
do{
 {Tạo phần tử mới}
 while(Counter==SIZE);
 if(Counter==SIZE) block();
 {Đặt phần tử mới vào Buffer
   IN = (IN+1)%SIZE;}
 Counter++;
 if(Counter==1)
    wakeup(Consumer);
} while (1);
```

Producer

```
do{
 while(Counter == 0);
 if(Counter == 0); block()
 {Lấy 1 phần tử trong Buffer
   OUT=(OUT+1)%SIZE;}
 Counter--;
 if(Counter==SIZE-1)
    wakeup(Producer);
 {Xử lý phần tử vừa lấy ra}
} while (1);
```

Consumer

Vấn đề sản xuất-tiêu thụ 2

- **Giải pháp:** Dùng một đèn báo *Mutex* để điều độ biến *Counter*
- **Khởi tạo:** *Mutex←1*

```
do{
 {Tạo phần tử mới}
 if(Counter==SIZE) block();
 {Đặt phần tử mới vào Buffer}
 wait(Mutex);
 Counter++;
 signal(Mutex);
 if(Counter==1)
    wakeup(Consumer);
} while (1);
```

Producer

```
do{
 if(Counter == 0); block()
 {Lấy 1 phần tử trong Buffer}
 wait(Mutex);
 Counter−−;
 signal(Mutex);
 if(Counter==SIZE - 1)
    wakeup(Producer);
 {Xử lý phần tử vừa lấy ra}
} while (1);
```

Consumer

- **Vấn đề:** Giả thiết *Counter=0*
  - *Consumer* kiểm tra *counter* ⇒ gọi thực hiện lệnh block()
  - *Producer* Tăng counter lên 1 và gọi wakeup(*Consumer*)
  - *Consumer* chưa bị block ⇒Câu lệnh wakeup() bị bỏ qua

Vấn đề sản xuất-tiêu thụ 3

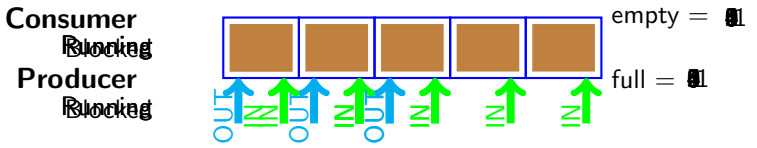
- Giải pháp: Sử dụng 2 đèn báo full, empty được khởi tạo
  - full ← 0 : Số phần tử trong hòm thư
  - empty ← BUFFER_SIZE: Số chỗ trống trong hòm thư

```
do{
 {Tạo phần tử mới}
 wait(empty);
 {Đặt phần tử mới vào Buffer}
 signal(full);
} while (1);
```

Producer

```
do{
 wait(full);
 {Lấy 1 phần tử trong Buffer}
 signal(empty);
 {Xử lý phần tử vừa lấy ra}
} while (1);
```

Consumer



Vấn đề sản xuất-tiêu thụ 4

- **Vấn đề:** Khi có nhiều *Producers* và *Consumers*, các biến **IN**, **OUT** trở thành tài nguyên găng giữa chúng
- **Giải quyết:** Dùng đèn báo thứ 3 (*mutex ← 1*) để đồng bộ giữa các tiến trình cùng loại

```
do{
 {Tạo phần tử mới}
 wait(empty);
 wait(mutex);
 {Đặt phần tử mới vào Buffer}
 signal(mutex);
 signal(full);
} while (1);
```

Producer

```
do{
 wait(full);
 wait(mutex);
 wait(mutex);
 wait(full);
 {Lấy 1 phần tử trong Buffer}
 signal(mutex);
 signal(empty);
 {Xử lý phần tử vừa lấy ra}
} while (1);
```

Consumer

Người đọc và biên tập viên

- Nhiều tiến trình (*Readers*) cùng truy nhập một cơ sở dữ liệu (*CSDL*)
- Một số tiến trình (*Writers*) cập nhật cơ sở dữ liệu
- Cho phép số lượng tùy ý các tiến trình *Readers* cùng truy nhập CSDL
  - Đang tồn tại một tiến trình *Reader* truy cập CSDL, mọi tiến trình *Readers* khác mới xuất hiện đều được truy cập CSDL (*Tiến trình Writers phải xếp hàng chờ đợi*)
- Chỉ cho phép một tiến trình *Writers* cập nhật CSDL tại một thời điểm.
- Vấn đề không trưng dụng. Các tiến trình ở trong đoạn găng mà không bị ngắt

CÀI ĐẶT BÀI TOÁN !!

Người thợ cắt tóc ngủ gật

- N ghế đợi dành cho khách hàng
- Một người thợ chỉ có thể cắt tóc cho một khách hàng tại một thời điểm
  - Không có khách hàng đợi, thợ cắt tóc ngủ
- Khi một khách hàng tới
  - Nếu thợ cắt tóc đang ngủ⇒Đánh thức anh ta dậy làm việc
  - Nếu thợ cắt tóc đang làm việc
    - Không còn ghế đợi trống ⇒ bỏ đi
    - Còn ghế đợi trống⇒ Ngồi đợi

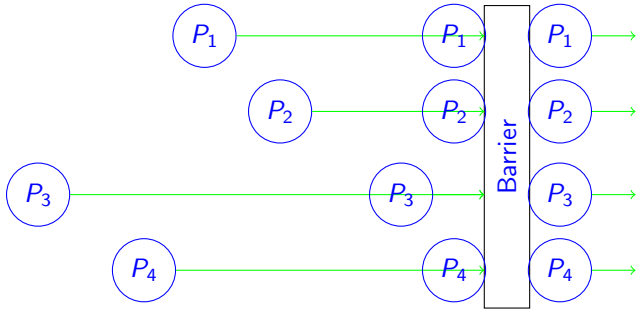
TÌM HIỂU VÀ CÀI ĐẶT

Bathroom Problem

- Thường dùng cho mục đích minh họa vấn đề phân phối tài nguyên trong nghiên cứu hệ điều hành và tính toán song song
- Bài toán
  - A bathroom is to be used by both men and women, but not at the same time
  - If the bathroom is empty, then anyone can enter
  - If the bathroom is occupied, then only a person of the same sex as the occupant(s) may enter
  - The number of people that may be in the bathroom at the same time is limited
- Yêu cầu cài đặt bài toán thỏa mãn các ràng buộc
  - Có 2 kiểu tiến trình male() và female()
  - Mỗi t/trình ở trong *Bathroom* một khoảng t/gian ngẫu nhiên

CÀI ĐẶT BÀI TOÁN !!

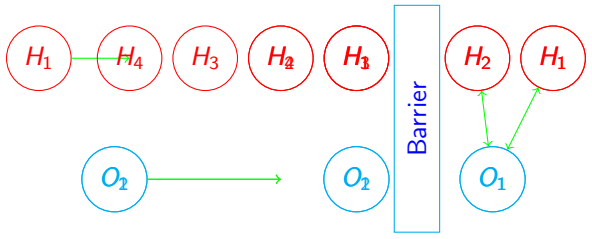
Đồng bộ barriers



- Các tiến trình hướng tới một Ba-ri-e chung
- Khi đạt tới Ba-ri-e, tất cả các tiến trình đều bị *block* ngoại trừ tiến trình đến cuối cùng
- Khi tiến trình cuối tới, đánh thức tất cả các tiến trình đang bị *block* và cùng vượt qua Ba-ri-e

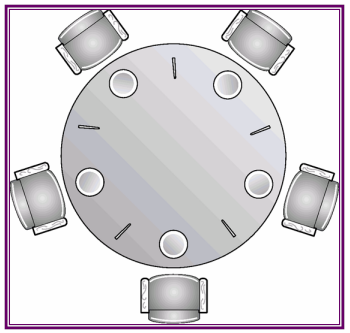
Bài toán tạo phân tử H_2O

- Có 2 kiểu tiến trình (luồng): *oxygen* and *hydrogen*
- Để kết hợp các tiến trình thành một phân tử nước, cần một Ba-ri-e để các tiến trình phải đợi cho tới khi một phân tử nước sẵn sàng được tạo ra.
- Khi mỗi tiến trình vượt qua Ba-ri-e, nó phải kích hoạt liên kết.
- Tất cả các tiến trình trong cùng một phân tử nước phải tạo liên kết, trước khi một tiến trình của phân tử nước khác gọi tới thủ tục tạo liên kết



Vấn đề triết gia ăn tối

- Bài toán đồng bộ hóa tiến trình nổi tiếng, thể hiện tình trạng nhiều tiến trình phân chia nhiều tài nguyên



- 5 triết gia ăn tối quanh một bàn tròn
  - Trước mỗi triết gia là một đĩa mì
  - Giữa 2 đĩa kề nhau là một cái dĩa (*fork*)
- Các triết gia thực hiện luân phiên, liên tục 2 việc :*Ăn* và *Nghĩ*
- Mỗi triết gia cần 2 cái dĩa để ăn
  - Chỉ lấy một dĩa tại một thời điểm
    - Cái bên trái rồi tới cái bên phải
- Ăn xong, triết gia để dĩa vào vị trí cũ
- Yêu cầu: viết chương trình đồng bộ bữa tối của 5 triết gia

Vấn đề triết gia ăn tối: Phương pháp đơn giản

- Mỗi chiếc dĩa là một tài nguyên găng, được điều độ bởi một đèn báo **fork[i]**
- **Semaphore** fork[5] = {1, 1, 1, 1, 1};
- Thuật toán cho Triết gia P_i

```
do{
 wait(fork[i])
    wait(fork[(i+1)% 5]);
          { Ăn}
    signal(fork[(i+1)% 5]);
 signal(fork[i]);
        {Nghĩ}
} while (1);
```

- Nếu tất cả các triết gia cùng muốn ăn
  - Cùng lấy chiếc dĩa bên trái (gọi tới: *wait(fork[i])*)
  - Cùng đợi lấy chiếc dĩa bên phải (gọi tới: *wait(fork[(i+1)%5])*)
- **Bế tắc (deadlock)**

Vấn đề triết gia ăn tối: Giải pháp 1

- Chỉ cho phép một nhà triết học lấy dĩa tại một thời điểm
- **Semaphore** mutex ← 1;
- Thuật toán cho Triết gia P_i

```
do{
   wait(mutex)
      wait(fork[i])
      wait(fork[(i+1)% 5]);
   signal(mutex)
         { Ăn}
      signal(fork[(i+1)% 5]);
      signal(i);
         {Nghĩ}
} while (1);
```

- **Có thể làm cho 2 triết gia không kề nhau cùng được ăn tại một thời điểm (P_1: ăn, P_2: chiếm mutex⇒ P_3 đợi)**

Vấn đề triết gia ăn tối: Giải pháp 2

- Thứ tự lấy dĩa của các triết gia khác nhau
  - Triết gia số hiệu chẵn lấy dĩa trái trước
  - Triết gia số hiệu lẻ lấy dĩa phải trước
- Thuật toán cho Triết gia P_i

```
do{
   j = i%2
   wait(fork[(i + j)%5])
      wait(fork[(i+1 - j)% 5]);
         { Ăn}
      signal(fork[(i+1 - j)% 5]);
   signal((i + j)%5);
         {Nghĩ}
} while (1);
```

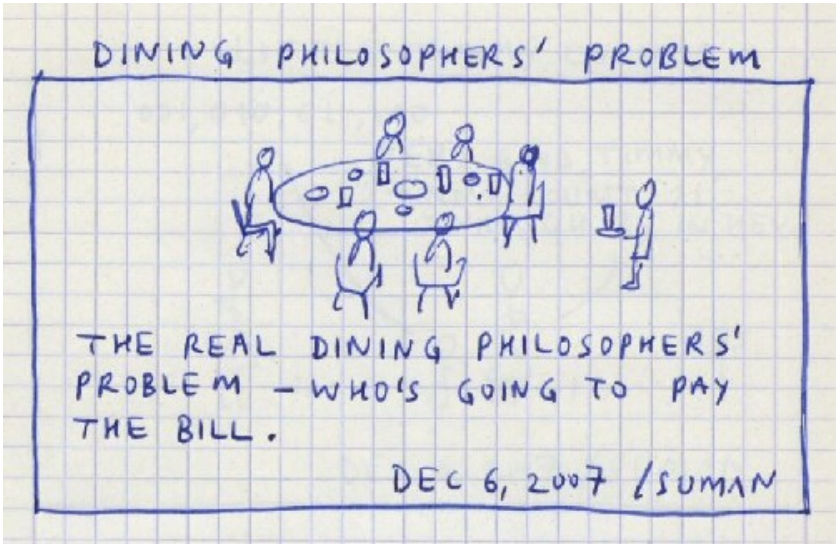
- Giải quyết được vấn đề bế tắc

Vấn đề triết gia ăn tối: Một số giải pháp khác

- Trả lại dĩa bên trái nếu không lấy được cái bên phải
  - Kiểm tra dĩa phải sẵn sàng trước khi gọi wait(fork[(i+1)%5])
  - Nếu không sẵn có: trả lại dĩa trái, đợi một thời gian rồi thử lại
  - Không bị bế tắc, nhưng không tiến triển:**nạn đói (starvation)**
  - Thực hiện trong thực tế, nhưng không đảm bảo về lý thuyết
- Sử dụng đèn báo đồng thời $P_{Sim}(S_1, S_2, \ldots, S_n)$
  - Thu được tất cả đèn báo cùng một thời điểm hoặc không có bất kỳ đèn báo nào
  - Thao tác $P_{Sim}(S_1, S_2, \ldots, S_n)$ sẽ block() tiến trình/luồng gọi khi có bất kỳ một đèn báo nào không thể thu được
  - Thuật toán

```
P_Sim(fork[i], fork[(i+1)% 5]);
{ Ăn}
V_Sim(fork[i], fork[(i+1)% 5]);
```

  - Khó cài đặt đèn báo đồng thời
- Giải pháp đề xuất bởi Tanenbaum (*Tanenbaum 2001*)
- Các công cụ điều độ cấp cao



(*http://www.codinghorror.com/blog/2008/08/deadlocked.html*)

Giới thiệu

- Kỹ thuật đèn báo là cơ chế hiệu quả trong điều độ tiến trình
- Sử dụng đèn báo (công cụ cấp thấp)
  - Người dùng phải biết về tài nguyên để điều độ
    - Có phải tài nguyên găng không?
  - Đặt các câu lệnh điều độ trong chương trình
    ⇒Nếu sử dụng nhầm có thể dẫn tới kết quả sai, khó gỡ rối
- Nhận biết và điều độ tài nguyên găng: trách nhiệm của hệ thống
- Công cụ thường dùng
  - Vùng găng
  - Monitor

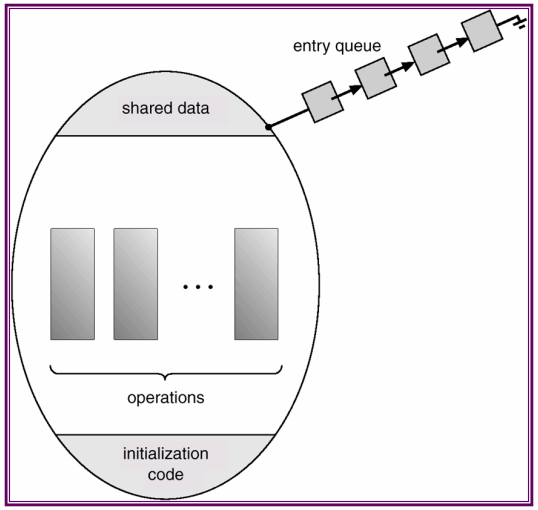
Monitor

```
monitor monitorName{
   Khai báo các biến dùng chung
   procedure P1(...){
       ...
   }
   ...
   procedure Pn(...){
       ...
   }

   {
       Mã khởi tạo
   }
} ;
```

Cú pháp của Monitor

- Là một kiểu dữ liệu đặc biệt, được đề nghị bởi HOARE 1974
- Bao gồm các thủ tục, dữ liệu cục bộ, đoạn mã khởi tạo
- Các tiến trình chỉ có thể truy nhập tới các biến bởi gọi tới các thủ tục trong Monitor
- Tại một thời điểm chỉ có một tiến trình được quyền sử dụng Monitor
  - Tiến trình khác muốn sử dụng, phải chờ đợi
- Cho phép các tiến trình đợi trong Monitor
  - Sử dụng các biến điều kiện (*condition variable*)

Mô hình



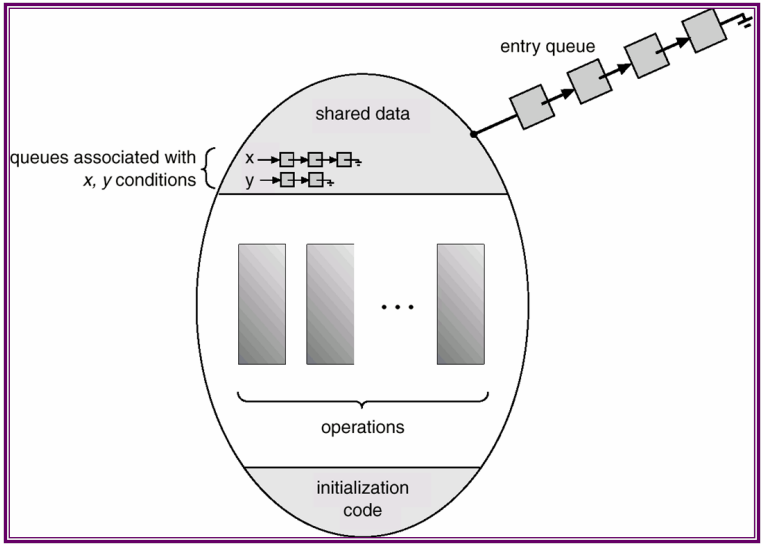
Biến điều kiện

- Thực chất là tên của một hàng đợi
- Khai báo: **condition** x, y;
- Các biến điều khiển chỉ có thể được sử dụng với 2 thao tác

wait() Được gọi bởi các thủ tục của *Monitor* (**Cú pháp**:*x.wait() hoặc wait(x)*) cho phép tiến trình đưa ra lời gọi bị tạm dừng (*block*) cho tới khi được một tiến trình khác kích hoạt bởi gọi tới *signal()*

signal() Được gọi bởi các thủ tục của *Monitor* (**Cú pháp**: *x.signal() hoặc signal(x)*) kích hoạt chính xác một tiến trình đang đợi tại biến điều kiện x (*nằm trong hàng đợi x*) ra tiếp tục hoạt động. Nếu không có tiến trình nào đang đợi, thao tác không có hiệu lực (*bị bỏ qua*)

Mô hình



Sử dụng Monitor: một tài nguyên chung

```
Monitor Resource{
  Condition Nonbusy;
  Boolean Busy
//-- Phần dành người dùng --
  void Acquire(){
    if(busy) Nonbusy.wait();
      busy=true;
  }
  void Release(){
    busy=false
    signal(Nonbusy)
  }
//---- Phần khởi tạo ----
  busy= false;
  Nonbusy = Empty;
}
```

Cấu trúc tiến trình

```
while(1){
    ...
    Resource.Acquire()
      {Sử dụng tài nguyên}
    Resource.Release()
    ...
}
```

Sử dụng Monitor: Bài toán Producer - Consumer

```
Monitor ProducerConsumer{
  Condition Full, Empty;
  int Counter ;
    void Put(Item){
      if(Counter=N) Full.wait();
      {Đặt Item vào Buffer};
      Counter++;
      if(Counter=1)Empty.signal()
  }
  void Get(Item){
    if(Counter=0) Empty.wait()
        {Lấy Item từ Buffer};
    Counter--;
    if(Counter=N-1)Full.signal()
  }
  Counter=0;
  Full, Empty = Empty;
}
```

ProducerConsumer M;

Producer

```
while(1){
    Item =Sản phẩm mới
    M.Put(Item)
    ...
}
```

Consumer

```
while(1){
    M.Get(&Item)
    {Sử dụng Msg}
    ...
}
```


Kết luận

(*Nguồn: internet*)

Giới thiệu

- Hệ thống gồm nhiều tiến trình hoạt động đồng thời cùng sử dụng tài nguyên
  - Tài nguyên có nhiều loại (*VD: CPU, bộ nhớ,..*).
  - Mỗi loại tài nguyên có nhiều đơn vị (*VD: 2 CPU, 5 máy in..*)
- Mỗi tiến trình thường gồm dãy liên tục các thao tác
  - Đòi hỏi tài nguyên: Nếu tài nguyên không có sẵn (*đang được s/dụng bởi tiến trình khác*) $\Rightarrow$ tiến trình yêu cầu phải đợi
  - Sử dụng tài nguyên theo yêu cầu (*in ấn, đọc dữ liệu...*)
  - Giải phóng tài nguyên được cấp
- Khi các tiến trình dùng chung ít nhất 2 tài nguyên, hệ thống có thể gặp *"nguy hiểm"*
- Xét ví dụ:
  - Hệ thống có hai tiến trình P_1 & P_2
  - Hai tiến trình P_1 & P_2 dùng chung hai tài nguyên R_1 & R2
    - R_1 được điều độ bởi đèn báo S_1 ($S_1 \leftarrow 1$)
    - R_2 được điều độ bởi đèn báo S_2 ($S_2 \leftarrow 1$)

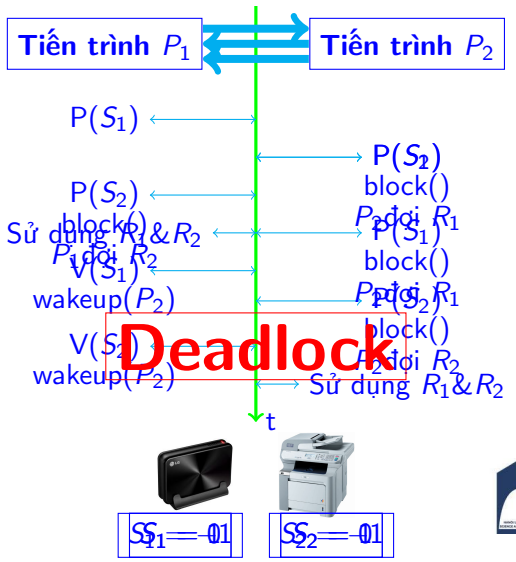
Ví dụ

$P(S_1)$
$P(S_2)$
{Sử dụng R_1&R_2}
$V(S_1)$
$V(S_2)$

Tiến trình P_1

$P(S_1)$
$P(S_2)$
$P(S_2)$
$P(S_1)$
{Sử dụng R_1&R_2}
$V(S_1)$
$V(S_2)$

Tiến trình P_2



Định nghĩa

Bế tắc là tình trạng

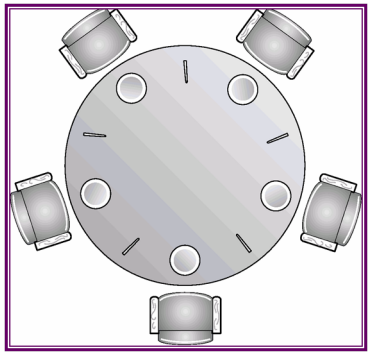
- Hai hay nhiều tiến trình cùng chờ đợi một sự kiện nào đó xảy ra
- Nếu không có sự tác động gì từ bên ngoài, thì sự chờ đợi đó là vô hạn

Điều kiện cần

Cần có 4 điều kiện sau, không được thiếu điều kiện nào

- **Có tài nguyên găng**
  - Tài nguyên được sử dụng theo mô hình không phân chia được
    - Chỉ có một tiến trình dung tài nguyên tại một thời điểm
    - Tiến trình khác cũng yêu cầu tài nguyên ⇒ yêu cầu phải được hoãn lại tới khi tài nguyên được giải phóng
- **Chờ đợi trước khi vào đoạn găng**
  - Tiến trình không được vào đoạn găng phải xếp hàng chờ đợi.
  - Trong khi chờ đợi vẫn chiếm giữ các tài nguyên được cung cấp
- **Không có hệ thống phân phối lại tài nguyên găng**
  - Tài nguyên không thể được trưng dụng
  - Tài nguyên được giải phóng chỉ bởi tiến trình đang chiếm giữ khi đã hoàn thành nhiệm vụ
- **Chờ đợi vòng tròn**
  - Tồn tại tập các tiến trình $\{P_0, P_2, \ldots, P_n\}$ đang đợi nhau theo kiểu: $P_0 \to R_1 \to P_1; P_1 \to R_2 \to P_2; \ldots P_{n-1} \to R_n \to P_n; P_n \to R_0 \to P_0$
  - Chờ đợi vòng tròn tạo ra chu trình không kết thúc

Ví dụ: Bài toán triết gia ăn tối



Tài nguyên găng

Chờ đợi trước khi vào đoạn găng

Trưng dụng tài nguyên găng

Chờ đợi vòng tròn

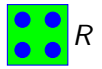
Đồ thị cung cấp tài nguyên (Resource Allocation Graph)

- Dùng để mô hình hóa tình trạng bế tắc trong hệ thống
- Là độ thị định hướng gồm tập đỉnh V và tập cung E
- Tập đỉnh V được chia thành 2 kiểu đỉnh
  - $P = \{P_1, P_2, \ldots P_n\}$ Tập chứa tất cả các tiến trình trong hệ thống
  - $R = \{R_1, R_2, \ldots R_m\}$ Tập chứa tất cả các kiểu tài nguyên trong hệ thống
- Tập các cung E gồm 2 loại
  - **Cung yêu cầu:** đi từ tiến trình P_i tới tài nguyên R_j: $P_i \rightarrow R_j$
  - **Cung sử dụng:** Đi từ tài nguyên R_j tới tiến trình P_i: $R_j \rightarrow P_i$
- Khi một tiến trình P_i yêu cầu tài nguyên R_j
  1. Cung yêu cầu $P_i \rightarrow R_j$ được chèn vào đồ thị
  2. Nếu yêu cầu được thỏa mãn, cung yêu cầu chuyển thành cung sử dung $R_j \rightarrow P_i$
  3. Khi tiến trình P_i giải phóng tài nguyên R_j, cung sử dụng $R_j \rightarrow P_i$ bị xóa khỏi đồ thị

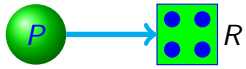
Đồ thị cung cấp tài nguyên : Biểu diễn đồ trong đồ thị

- Đỉnh *kiểu tiến trình* được thể hiện bằng hình tròn

- Đỉnh *kiểu tài nguyên* được thể hiện bằng hình chữ nhật
  Mỗi đơn vị của kiểu tài nguyên được biểu thị bằng một dấu chấm trong hình chữ nhật

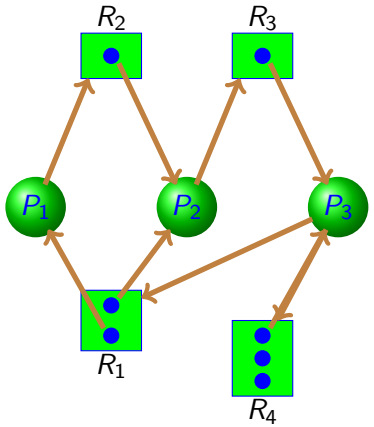


- Cung yêu cầu đi từ đỉnh tiến trình tới đỉnh tài nguyên



- Cung sử dụng xuất phát từ dấu chấm bên trong đỉnh tài nguyên tới đỉnh tiến trình

Đồ thị cung cấp tài nguyên : Ví dụ

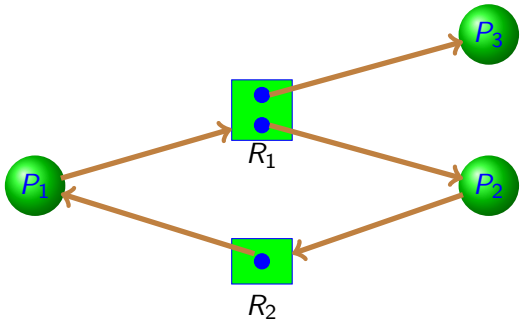


- Trạng thái hệ thống
  - 3 tiến trình P_1, P_2, P_3
  - 4 tài nguyên R_1, R_2, R_3, R_4
- P_3 yêu cầu tài nguyên R_4
  - Xuất hiện cung yêu cầu $P_3 \rightarrow R_4$
  - Cung yêu cầu $P_3 \rightarrow R_4$ chuyển thành cung sử dụng $R_4 \rightarrow P_3$
- P_3 Giải phóng tài nguyên R_4
  - Cung sử dụng $R_4 \rightarrow P_3$ bị xóa khỏi đồ thị
- P_3 yêu cầu tài nguyên R_1
  - Xuất hiện cung yêu cầu $P_3 \rightarrow R_1$
  - Trên đồ thị xuất hiện chu trình
  - Hệ thống bế tắc

Chu trình trên đồ thị và tình trạng bế tắc có liên quan?

Đồ thị cung cấp tài nguyên : Lập luận cơ bản

Đồ thị có chu trình nhưng hệ thống không bế tắc



- Đồ thị không chứa chu trình, không bế tắc
- Nếu đồ thị chứa đựng chu trình
  - Nếu tài nguyên chỉ có 1 đơn vị ⇒ Bế tắc
  - Nếu tài nguyên có nhiều hơn 1 đơn vị: có khả năng bế tắc

Phương pháp

1. **Phòng ngừa**
   - Áp dụng các biện pháp để đảm bảo hệ thống không bao giờ rơi vào tình trạng bế tắc
   - Tốn kém
   - Áp dụng cho hệ thống hay xảy ra bế tắc và tổn thất do bế tắc gây ra lớn
2. **Phòng tránh**
   - Kiểm tra từng yêu cầu tài nguyên của tiến trình và không chấp nhận yêu cầu nếu việc cung cấp tài nguyên *có khả năng* dẫn đến tình trạng bế tắc
   - Thường yêu cầu các thông tin phụ trợ
   - Áp dụng cho hệ thống ít xảy ra bế tắc nhưng tổn hại lớn
3. **Nhận biết và khắc phục**
   - Cho phép hệ thống hoạt động bình thường ⇒*có thể rơi vào tình trạng bế tắc*
   - Định kỳ kiểm tra xem bế tắc có đang xảy ra không
   - Nếu đang bế tắc, áp dụng các biện pháp loại bỏ bế tắc
   - Áp dụng cho hệ thống ít xảy ra bế tắc và thiệt hại không lớn

Nguyên tắc

Tác động vào 1 trong 4 điều kiện cần của bế tắc để nó không xảy ra

Tài nguyên găng

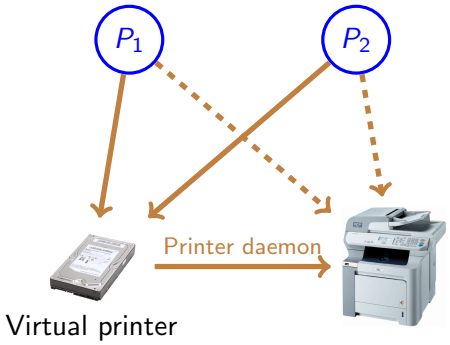
Chờ đợi trước khi vào đoạn găng

Trưng dụng tài nguyên găng

Chờ đợi vòng tròn

Điều kiện tài nguyên găng

- Giảm bớt mức độ găng của hệ thống
  - Tài nguyên phân chia được (*file chỉ đọc*): Sử dụng đồng thời
  - Tài nguyên không phân chia được: Sử dụng không đồng thời
- Kỹ thuật SPOOL(*Simultaneous peripheral operation on-line*)
  - Không phân phối tài nguyên khi không thực sự cần thiết
  - Chỉ một số ít tiến trình có khả năng yêu cầu tài nguyên



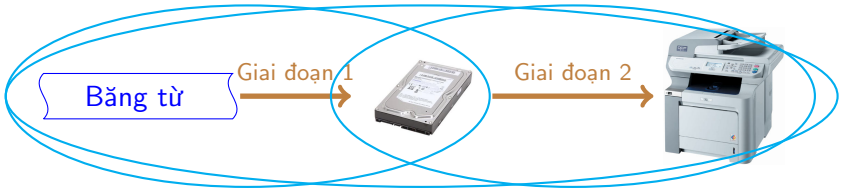
- Chỉ *printer daemon* mới làm việc với máy in ⇒ Bế tắc cho tài nguyên máy in bị hủy bỏ
- Không phải tài nguyên nào cũng dùng kỹ thuật SPOOL được

Điều kiện chờ đợi trước khi vào đoạn găng

Nguyên tắc: Đảm bảo một tiến trình xin tài nguyên chỉ khi không sở hữu bất kỳ tài nguyên nào khác

- Cung cấp trước
  - Tiến trình xin toàn bộ tài nguyên ngay từ đầu và chỉ thực hiện khi đã có đầy đủ tài nguyên
  - Hiệu quả sử dụng tài nguyên thấp
    - Tiến trình chỉ sử dụng tài nguyên ở giai đoạn cuối?
    - Tổng số tài nguyên đòi hỏi vượt quá khả năng của hệ thống?
- Giải phóng tài nguyên
  - Tiến trình giải phóng tất cả tài nguyên trước khi xin (*xin lại*) tài nguyên mới
  - Nhận xét
    - Tốc độ thực hiện tiến trình chậm
    - Phải đảm bảo dữ liệu được giữ trong tài nguyên tạm giải phóng không bị mất

Điều kiện chờ đợi trước khi vào đoạn găng: minh họa



- Tiến trình gồm 2 giai đoạn
  - Sao chép dữ liệu từ băng từ sang một file trên đĩa từ
  - Sắp xếp dữ liệu trong file và đưa ra máy in
- Phương pháp cung cấp trước
  - Xin cả băng từ, file trên đĩa và máy in
  - **Lãng phí** máy in giai đoạn đầu, băng từ giai đoạn cuối
- Phương pháp giải phóng tài nguyên
  - Xin băng từ và file trên đĩa cho giai đoạn 1
  - Giải phóng băng từ và file trên đĩa
  - Xin file trên đĩa và máy in cho giai đoạn 2(**Nếu không được?**)

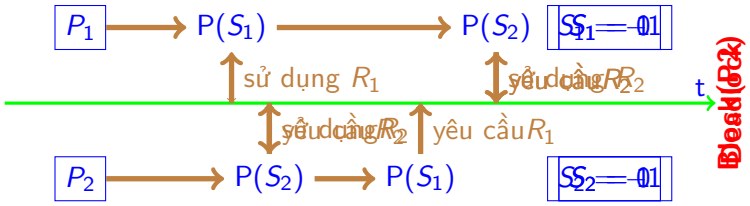
Điều kiện trưng dụng tài nguyên găng

- **Nguyên tắc:** cho phép trưng dụng tài nguyên khi cần thiết
- Tiến trình P_i xin tài nguyên R_j
  - ★ R_j **sẵn có:** Cung cấp R_j cho P_i
  - ★ R_j **không sẵn:** (R_j *bị chiếm bởi tiến trình* P_k)
    - P_k đang đợi tài nguyên
      - Trưng dụng R_j từ P_k và cung cấp cho P_i theo yêu cầu
      - Thêm R_j vào danh sách các tài nguyên đang thiếu của P_k
      - P_k được thực hiện trở lại khi
        - ♣ Có được tài nguyên đang thiếu
        - ♣ Đòi lại được R_j
    - P_k đang thực hiện
      - P_i phải đợi (*không giải phóng tài nguyên*)
      - Cho phép trưng dụng tài nguyên nhưng **chỉ khi cần thiết**
- Chỉ áp dụng cho các tài nguyên có thể lưu trữ và khôi phục trạng thái dễ dàng (*CPU, không gian nhớ*). Khó có thể áp dụng cho các tài nguyên như máy in
- Một tiến trình bị trưng dụng nhiều lần ?

Điều kiện chờ đợi vòng tròn

- Đặt ra một thứ tự toàn cục của tất cả các kiểu tài nguyên
  - R = $\{R_1, R_2, \ldots R_n\}$ Tập tất cả các kiểu tài nguyên
  - Xây dựng *hàm trật tự* $f : R \to \mathbb{N}$
    - Hàm f được xây dựng dựa trên trật tự sử dụng các tài nguyên
      - ⋆ f(Băng từ) = 1
      - ⋆ f(Đĩa từ) = 5
      - ⋆ f(Máy in) = 12
- Tiến trình chỉ được yêu cầu tài nguyên theo trật tự tăng
  - Tiến trình chiếm giữ tài nguyên kiểu R_k chỉ được xin tài nguyên kiểu R_j thỏa mãn $f(R_j) > f(R_k)$
  - Tiến trình yêu cầu tới tài nguyên R_k sẽ phải giải phóng tất cả tài nguyên R_i thỏa mãn điều kiện $f(R_i) \geq f(R_k)$
- Chứng minh
  - Giả thiết bế tắc xảy ra giữa các tiến trình $\{P_1, P_2, \ldots P_m\}$
    - $R_1 \to P_1 \to R_2 \to P_2 \Rightarrow f(R_1) < f(R_2)$
    - $R_2 \to P_2 \to R_3 \to P_3 \Rightarrow f(R_2) < f(R_3) \ldots$
    - $R_m \to P_m \to R_1 \to P_1 \Rightarrow f(R_m) < f(R_1)$
  - $f(R_1) < f(R_2) < \ldots < f(R_m) < f(R_1) \Rightarrow$ **Vô lý**

Ví dụ



Nhận xét:

Biết được chuỗi yêu cầu/giải phóng tài nguyên của các tiến trình, hệ thống có thể đưa ra được chiến lược phân phối tài nguyên (*chấp thuận hay phải đợi*) cho mọi yêu cầu để bế tắc không xảy ra.

Nguyên tắc

- Phải biết trước các thông tin về tiến trình và tài nguyên
  - Tiến trình phải khai báo lượng tài nguyên lớn nhất mỗi loại sẽ yêu cầu khi thực hiện
- Quyết định dựa trên kết quả kiểm tra *trạng thái cung cấp tài nguyên (Resource-Allocation State)* -Trạng thái hệ thống
  - Trạng thái cung cấp tài nguyên xác định bởi các thông số
    - Số đơn vị tài nguyên có sẵn trong hệ thống
    - Số đơn vị tài nguyên đã được cấp cho mỗi tiến trình
    - Số đơn vị tài nguyên lớn nhất mỗi tiến trình có thể yêu cầu
  - Nếu **hệ thống an toàn**, sẽ đáp ứng cho yêu cầu
- Thực hiện kiểm tra mỗi khi nhận được yêu cầu tài nguyên
  - Mục đích: Đảm bảo trạng thái hệ thống luôn an toàn
    - Thời điểm ban đầu (*chưa c/cấp tài nguyên*), hệ thống an toàn
    - Hệ thống chỉ cung cấp tài nguyên khi vẫn đảm bảo an toàn

  ⇒Hệ thống chuyển từ trạng thái an toàn này sang trạng thái an toàn khác

Trạng thái an toàn của hệ thống là gì?

Trạng thái an toàn

Trạng thái của hệ thống là an toàn khi

- Có thể cung cấp tài nguyên cho từng tiến trình (*đến yêu cầu lớn nhất*) theo một trật tự nào đấy mà không xảy ra bế tắc
- Tồn tại **chuỗi an toàn** của tất cả các tiến trình

Chuỗi an toàn

Chuỗi tiến trình P=$\{P_1, P_2, \ldots, P_n\}$ là an toàn nếu

Với mỗi tiến trình P_i, mọi yêu cầu tài nguyên trong tương lai đều có thể đáp ứng nhờ vào

- Lượng tài nguyên hiện có trong hệ thống
- Tài nguyên đang chiếm giữ bởi tất cả các tiến trình $P_j (j < i)$

Trong chuỗi an toàn, khi P_i yêu cầu tài nguyên

- Nếu không thể đáp ứng ngay lập tức, P_i đợi cho tới khi P_j kết thúc $(j < i)$
- Khi P_j kết thúc và giải phóng tài nguyên, P_i sẽ nhận được tài nguyên cần thiết, thực hiện, giải phóng các tài nguyên đã được cung cấp và kết thúc

Trong chuỗi an toàn

- Khi P_i kết thúc và giải phóng tài nguyên $\Rightarrow$ P_{i+1} sẽ nhận được tài nguyên cần thiết và kết thúc được ...
- Tất cả các tiến trình trong chuỗi an toàn đều kết thúc được
- **Lưu ý:** P_1 *chỉ có thể kết thúc bởi tài nguyên hệ thống đang*

Ví dụ minh họa

- Xem xét hệ thống gồm
  - 3 tiến trình P_1, P_2, P_3 và 1 tài nguyên R có 12 đơn vị
  - Các tiến trình (P_1, P_2, P_3) có thể yêu cầu tối đa tới $(10, 4, 9)$ đơn vị tài nguyên R
  - Tại thời điểm t_0, các tiến trình (P_1, P_2, P_3) đã được cấp $(5, 2, 2)$ đơn vị tài nguyên R

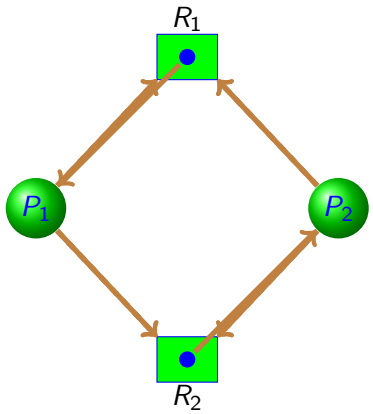
Phòng tránh bế tắc

- Nhận xét
  - Hệ thống an toàn $\Rightarrow$ Các tiến trình đều có thể kết thúc được $\Rightarrow$ không xảy ra bế tắc
  - Hệ thống không an toàn $\Rightarrow$ Có khả năng xảy ra bế tắc
- Phương pháp
  - Không để hệ thống rơi vào tình trạng không an toàn
  - Kiểm tra mọi yêu cầu tài nguyên
    - Nếu hệ thống vẫn an toàn khi cung cấp $\Rightarrow$ Cung cấp
    - Nếu hệ thống không an toàn khi cung cấp $\Rightarrow$ Phải đợi
- Thuật toán
  - Thuật toán dựa vào đồ thị cung cấp tài nguyên
  - Thuật toán người quản lý nhà băng

Thuật toán dựa vào đồ thị cung cấp tài nguyên

- Sử dụng khi mỗi kiểu tài nguyên chỉ có 1 đơn vị
  - Có chu trình, sẽ có bế tắc
- Thêm vào đồ thị loại cung mới: **cung đòi hỏi** $P_i \rightarrow R_j$
  - Cùng hướng với cung yêu cầu, thể hiện trong đồ thị $--->$
  - Cho biết P_i có thể yêu cầu R_j trong tương lai
- Tiến trình khi tham gia hệ thống, phải thêm tất cả các cung đòi hỏi tương ứng vào đồ thị
  - Khi P_i yêu cầu R_j, cung đòi hỏi $P_i \rightarrow R_j$ chuyển thành cung yêu cầu $P_i \rightarrow R_j$
  - Khi P_i giải phóng R_j, cung sử dụng $R_j \rightarrow P_i$ chuyển thành cung đòi hỏi $P_i \rightarrow R_j$
- **Thuật toán:** Yêu cầu tài nguyên R_j của tiến trình P_i được thỏa mãn chỉ khi việc chuyển cung yêu cầu $P_i \rightarrow R_j$ thành cung sử dụng $R_j \rightarrow P_i$ không tạo chu trình trên đồ thị
  - Không chu trình: Hệ thống an toàn
  - Có chu trình: Việc cung cấp tài nguyên đẩy hệ thống vào tình trạng không an toàn

Ví dụ



- Hệ thống: 2 tiến trình P_1, P_2 và 2 tài nguyên R_1, R_2, mỗi loại 1 đơn vị
  - P_1 có thể xin R_1, R_2 trong tương lai
  - P_1 có thể xin R_1, R_2 trong tương lai
- P_1 yêu cầu tài nguyên R_1
  - Cung đòi hỏi trở thành cung yêu cầu
- Yêu cầu của P_1 được đáp ứng
  - Cung yêu cầu thành cung sử dụng
- P_2 yêu cầu tài nguyên $R_2 \Rightarrow$ cung đòi hỏi trở thành cung yêu cầu $P_2 \rightarrow R_2$
  - Nếu đáp ứng
    $\Rightarrow$Cung yêu cầu thành cung sử dụng
    $\Rightarrow$ Khi P_1 yêu cầu $R_2 \Rightarrow P_1$ phải đợi
    $\Rightarrow$ Khi P_2 yêu cầu $R_1 \Rightarrow P_2$ phải đợi
    Hệ thống bế tắc
  - Yêu cầu của P_2 không được đáp ứng

Thuật toán người quản lý nhà băng: Giới thiệu

- Thích hợp cho các hệ thống gồm các kiểu tài nguyên có nhiều đơn vị
- Một tiến trình mới xuất hiện trong hệ thống cần khai báo số đơn vị lớn nhất của mỗi kiểu tài nguyên sẽ sử dụng
  - Không được vượt quá tổng số tài nguyên của hệ thống
- Khi một tiến trình yêu cầu tài nguyên, hệ thống kiểm tra liệu đáp ứng cho yêu cầu hệ thống có còn an toàn không
  - Nếu hệ thống vẫn an toàn $\Rightarrow$ Cung cấp tài nguyên cho yêu cầu
  - Nếu hệ thống không an toàn $\Rightarrow$ Tiến trình phải đợi
- Thuật toán cần
  - Các cấu trúc dữ liệu biểu diễn trạng thái phân phối tài nguyên
  - Thuật toán kiểm tra tình trạng an toàn của hệ thông
  - Thuật toán yêu cầu tài nguyên

Các cấu trúc dữ liệu I

Hệ thống

n số tiến trình trong hệ thống

m số kiểu tài nguyên trong hệ thống

Các cấu trúc dữ liệu

Available Vector chiều dài m cho biết số đơn vị tài nguyên sẵn có trong hệ thống. (Available[3] = 8 $\Rightarrow$?)

Max Ma trận $n * m$ cho biết số lượng lớn nhất mỗi kiểu tài nguyên của từng tiến trình. (Max[2,3] = 5 $\Rightarrow$?)

Allocation Ma trận $n * m$ cho biết số lượng mỗi kiểu tài nguyên đã cấp cho tiến trình. (Allocation[2,3] = 2 $\Rightarrow$?)

Need Ma trận $n * m$ chỉ ra số lượng mỗi kiểu tài nguyên còn cần đến của từng tiến trình. Need[2,3] = 3 $\Rightarrow$?)
Need[i][j] = Max[i][j] - Allocation[i][j]

Các cấu trúc dữ liệu II

Quy ước

- X, Y là các vector độ dài n
  - $X \leq Y \Leftrightarrow X[i] \leq Y[i] \quad \forall i = 1, 2, \ldots, n$
- Các dòng của ma trận *Max,Yêu cầu, Cung cấp* được xử lý như các vector
- Thuật toán tính toán trên các vector

Các cấu trúc cục bộ

Work vector độ dài m cho biết mỗi tài nguyên còn bao nhiêu

Finish vector độ dài n, kiểu *logic* cho biết tiến trình có chắc chắn kết thúc không

Thuật toán kiểm tra An toàn

```
BOOL Safe(Current Resource-Allocation State){
  Work←Available
  for (i : 1 → n) Finish[i]←false
  flag← true
  While(flag){
    flag←false
    for (i : 1 → n) do
      if(Finish[i]=false AND Need[i] ≤Work){
        Finish[i]← true
        Work ← Work+Allocation[i]
        flag← true
      }//endif
  }//endwhile
  for (i : 1 → n) if (Finish[i]=false)return false
  return true;
}//End function
```

Ví dụ minh họa

- Xét hệ thống gồm 5 tiến trình P_0, P_1, P_2, P_3, P_4 và 3 tài nguyên R_0, R_1, R_2
  - Tài nguyên R_0 có 10 đơn vị, R_1 có 5 đơn vị, R_2 có 7 đơn vị
- Yêu cầu tài nguyên lớn nhất và lượng tài nguyên đã cấp của mỗi tiến trình

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3
Max			

	R_0	R_1	R_2
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2
Allocation			

- Hệ thống có an toàn?
- Tiến trình P_1 yêu cầu thêm 1 đơn vị R_0 và 2 đơn vị R_2?
- Tiến trình P_4 yêu cầu thêm 3 đơn vị R_0 và 3 đơn vị R_1?
- Tiến trình P_0 yêu cầu thêm 2 đơn vị R_1. Cung cấp?

Ví dụ minh họa : Kiểm tra tính an toàn

- Số tài nguyên còn sẵn trong hệ thống (R_0, R_1, R_2) =(3, 3, 2)
- Yêu cầu còn lại của mỗi tiến trình (*Need = Max - Allocation*)

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3
Max			

	R_0	R_1	R_2
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2
Allocation			

	R_0	R_1	R_2
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1
Need			

Thực hiện thuật toán an toàn

Tiến trình	P_0	P_1	P_2	P_3	P_4
Finish	F F T	F	F	F F T	F F T
Work	(3, 3, 2)	(5, 3, 2)	(7, 4, 3)	(7, 4, 5)	(7, 5, 5) (10, 5, 7)
Hệ thống an toàn (P_1, P_3, P_4, P_0, P_2)					

Thuật toán yêu cầu tài nguyên

- Request[i] Vector yêu cầu tài nguyên của tiến trình P_i
  - Request[3,2] = 2: Tiến trình P_3 yêu cầu 2 đơn vị tài nguyên R_2
- Khi P_i yêu cầu tài nguyên, hệ thống thực hiện
  1. if(Request[i]>Need[i])
     Error(*Yêu cầu vượt quá khai báo tài nguyên*)
  2. if(Request[i]>Available)
     Block(*Không đủ tài nguyên, tiến trình phải đợi*)
  3. Thiết lập trạng thái phân phối tài nguyên mới cho hệ thống
     - Available = Available - Request[i]
     - Allocation[i] = Allocation[i] + Request[i]
     - Need[i] = Need[i] - Request[i]
  4. Phân phối tài nguyên dựa trên kết quả kiểm tra tính an toàn của trạng thái phân phối tài nguyên mới
     if(Safe(New Resource Allocation State))
     Phân phối cho P_i theo yêu cầu
     else
     Tiến trình P_i phải đợi
     Khôi phục lại trạng thái cũ (*Available, Allocation,Need*)

Ví dụ minh họa : P_1 yêu cầu $(1, 0, 2)$

- Request[1]≤Available $((1, 0, 2) \leq (3, 3, 2)) \Rightarrow$ Có thể cung cấp
- Nếu cung cấp : Available = (2 , 3, 0)

	R_0	R_1	R_2
P_0	0	1	0
P_1	3	0	2
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2
Allocation			

	R_0	R_1	R_2
P_0	7	4	3
P_1	0	2	0
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1
Need			

Thực hiện thuật toán an toàn

Tiến trình	P_0	P_1	P_2	P_3	P_4
Finish	F F T	F	F	F F T	F F T
Work	(2, 3, 0)	(5, 3, 2)	(7, 4, 3)	(7, 4, 5)	(7, 5, 5) (10, 5, 7)
Yêu cầu được chấp nhận					

Ví dụ minh họa (tiếp tục)

- Tiến trình P_4 yêu cầu thêm 3 đơn vị R_0 và 3 đơn vị R_2
  - Request[4] = (3, 0, 3)
  - Available = (2, 3, 0)

  $\Rightarrow$ Không đủ tài nguyên, P_4 phải đợi

- Tiến trình P_0 yêu cầu thêm 2 đơn vị R_1
  - Request[0]≤Available $((0, 2, 0) \leq (2, 3, 0)) \Rightarrow$ Có thể cung cấp
  - Nếu cung cấp : Available = (2 , 1, 0)
  - Thực hiện thuật toán an toàn
    $\Rightarrow$ Tất cả các tiến trình đều có thể không kết thúc
    $\Rightarrow$ Nếu chấp nhận, hệ thống rơi vào trạng thái không an toàn

  $\Rightarrow$ Đủ tài nguyên nhưng không cung cấp. P_0 phải đợi

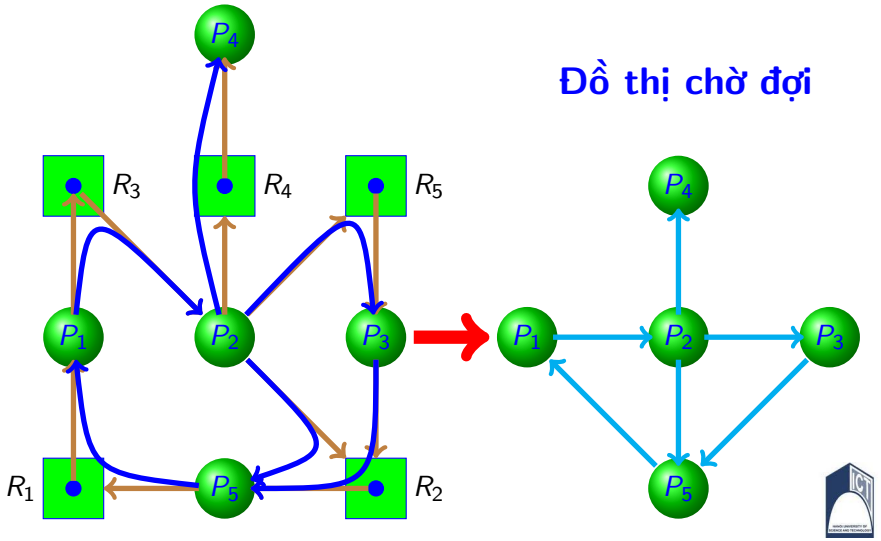
Giới thiệu

- Nguyên tắc
  - Không áp dụng các biện pháp phòng ngừa hoặc phòng tránh, để cho bế tắc xảy ra
  - Định kỳ kiểm tra xem bế tắc có đang xảy ra không. Nếu có tìm cách khắc phục
  - Để thực hiện, hệ thống phải cung cấp
    - Thuật toán xác định hệ thống đang bế tắc không
    - Thuật toán chữa bế tắc
- Nhận biết bế tắc
  - Thuật toán dựa trên đồ thị cung cấp tài nguyên
  - Thuật toán chỉ ra bế tắc tổng quát
- Khắc phục bế tắc
  - Kết thúc tiến trình
  - Trưng dụng tài nguyên

Thuận toán dựa trên đồ thị cung cấp tài nguyên

- Áp dụng khi mỗi tài nguyên trong hệ thống có một đơn vị
- Kiểm tra hệ thống có bế tắc bằng cách kiểm tra chu trình trên đồ thị
  - Nếu trên đồ thị có chu trình, hệ thống đang bế tắc
- Định kỳ gọi tới các thuật toán kiểm tra chu trình trên đồ thị
  - Thuật toán đòi hỏi n^2 thao tác (*n: số đỉnh của đồ thị*)
- Sử dụng đồ thị chờ đợi - phiên bản thu gọn của đồ thị cung cấp tài nguyên
  - Chỉ có các đỉnh dạng tiến trình
  - Cung chờ đợi $P_i \rightarrow P_j$: *Tiến trình P_i đang đợi tiến trình P_j giải phóng tài nguyên P_i cần*
  - Cung chờ đợi $P_i \rightarrow P_j$ tồn tại trên đồ thị đợi khi và chỉ khi trên đồ thị phân phối tài nguyên tương ứng tồn tại đồng thời cung yêu cầu $P_i \rightarrow R$ và cung sử dụng$R \rightarrow P_j$

Đồ thị chờ đợi: Ví dụ



Thuật toán chỉ ra bế tắc tổng quát : Giới thiệu

- Sử dụng cho các hệ thống có các kiểu tài nguyên gồm nhiều đơn vị
- Thuật toán tương tự thuật toán người quản lý nhà băng
- Các cấu trúc dữ liệu
  - **Available** Vector độ dài m: Tài nguyên sẵn có trong hệ thống
  - **Allocation** Ma trận $n * m$: Tài nguyên đã cấp cho tiến trình
  - **Request** Ma trận $n * m$ Tài nguyên tiến trình yêu cầu
- Các cấu trúc cục bộ
  - **Work** Vector độ dài m cho biết tài nguyên hiện đang có
  - **Finish** Vector độ dài n cho biết tiến trình **có thể** kết thúc không
- Các qui ước
  - Quan hệ $\leq$ giữa các Vector
  - Xử lý các dòng ma trận $n * m$ như các vector

Thuật toán chỉ ra bế tắc tổng quát

```
BOOL Deadlock(Current Resource-Allocation State){
  Work←Available
  For (i : 1 → n)
    if(Allocation[i]≠ 0) Finish[i]←false
    else Finish[i]=true;//Allocation= 0 không nằm trong chu trình đợi
  flag← true
  While(flag){
    flag←false
    for (i : 1 → n) do//Giả thiết tối ưu, đây là yêu cầu cuối
      if(Finish[i]=false AND Request[i] ≤Work){
        Finish[i]← true
        Work ← Work+Allocation[i]
        flag← true
      }//endif
  }//endwhile
  for (i : 1 → n) if (Finish[i]=false) return true;
  return false;//Finish[i] = false, tiến trình Pi đang bị bế tắc
}//End function
```

Ví dụ minh họa

- 5 tiến trình P_0, P_1, P_2, P_3, P_4; 3 tài nguyên R_0, R_1, R_2
  - Tài nguyên R_0 có 7 đơn vị, R_1 có 2 đơn vị, R_2 có 6 đơn vị
- Trạng thái cung cấp tài nguyên tại thời điểm t_0

	R_0	R_1	R_2
P_0	0	1	0
P_1	2	0	0
P_2	3	0	3
P_3	2	1	1
P_4	0	0	2

Allocation

	R_0	R_1	R_2
P_0	0	0	0
P_1	2	0	2
P_2	0	0	0
P_3	1	0	0
P_4	6	0	2

Request

- Tài nguyên hiện có $(R_0, R_1, R_2) = (0, 0, 0)$

Thực hiện thuật toán chỉ ra bế tắc

Tiến trình	P_0	P_1	P_2	P_3	P_4
Finish	F **F** T	F **F** T	F **F** T	F **F** T	F **F** T
Work	(0, 0, 0)	(0, 1, 0)	(3, 1, 3)	(5, 2, 4)	(7, 2, 4) (7, 2, 6)

Hệ thống không bế tắc $(P_0, P_2, P_3, P_1, P_4)$

Ví dụ minh họa (tiếp)

- P_2 yêu cầu thêm 1 đơn vị tài nguyên R_2
- Trạng thái cung cấp tài nguyên tại thời điểm t_1

	R_0	R_1	R_2
P_0	0	1	0
P_1	2	0	0
P_2	3	0	3
P_3	2	1	1
P_4	0	0	2

Allocation

	R_0	R_1	R_2
P_0	0	0	0
P_1	2	0	2
P_2	0	0	**1**
P_3	1	0	0
P_4	6	0	2

Request

Thực hiện thuật toán chỉ ra bế tắc

Tiến trình	P_0	P_1	P_2	P_3	P_4
Finish	F **F** T	F **F**	F **F**	F **F**	F **F**
Work	(0, 0, 0) (0, 1, 0)				

P_0 có thể kết thúc nhưng hệ thống đang bế tắc.
Các tiến trình đang chờ đợi lẫn nhau (P_1, P_2, P_3, P_4)

Khắc phục bế tắc: Phương pháp kết thúc tiến trình

Nguyên tắc: Hủy bỏ các tiến trình đang trong tình trạng bế tắc và lấy lại tài nguyên đã cấp cho tiến trình bị hủy bỏ

- Hủy bỏ tất cả các tiến trình
  - Nhanh chóng hủy bỏ bế tắc
  - Quá tốn kém
    - Các tiến trình bị hủy bỏ có thể gần kết thúc
- Hủy bỏ lần lượt tiến trình cho tới khi bế tắc không xảy ra
  - Sau khi hủy bỏ, phải kiểm tra xem bế tắc còn tồn tại không
    - Thuật toán kiểm tra bế tắc có độ phức tạp $m * n^2$
  - Cần chỉ ra thứ tự tiến trình bị hủy bỏ để phá vỡ bế tắc
    - Độ ưu tiên của tiến trình.
    - Tiến trình đã tồn tại bao lâu, còn bao lâu nữa thì kết thúc
    - Tài nguyên tiến trình đang chiếm giữ, còn cần để kết thúc
    - ...
- Vấn đề hủy bỏ tiến trình
  - Tiến trình đang cập nhật file ⇒ File không hoàn chỉnh
  - Tiến trình sử dụng máy in ⇒ Reset trạng thái máy in

Khắc phục bế tắc: Phương pháp trưng dụng tài nguyên

Nguyên tắc:
Trưng dụng liên tục một vài tài nguyên từ một số tiến trình đang bế tắc cho các tiến trình khác đến khi bế tắc được hủy bỏ

Các vấn đề cần quan tâm

1. Lựa chọn nạn nhân (*victim*)
   - Tài nguyên nào và tiến trình nào được chọn?
   - Trật tự trưng dụng để chi phí nhỏ nhất?
   - Lượng tài nguyên nắm giữ, thời gian sử dụng...
2. Quay lui (*Rollback*)
   - Quay lui tới một trạng thái an toàn trước đó và bắt đầu lại
   - Yêu cầu lưu giữ thông tin trạng thái của t/trình đang thực hiện
3. Đói tài nguyên (Starvation)
   - Một tiến trình bị trưng dụng quá nhiều lần ⇒chờ đợi vô hạn
   - **Giải pháp:** ghi lại số lần bị trưng dụng

Tổng kết

- Bế tắc là tình trạng 2 hay nhiều tiến trình cùng chờ đợi độc lập một sự kiện chỉ có thể xảy ra bởi sự hoạt động của các tiến trình đang đợi
- Bế tắc xảy ra khi hội đủ 4 điều kiện
  - Tồn tại tài nguyên găng
  - Phải chờ đợi trước khi vào đoạn găng
  - Không tồn tại hệ thống phân phối lại tài nguyên
  - Tồn tại hiện tượng chờ đợi vòng tròn
- Để xử lý bế tắc có 3 lớp thuật toán
  - Phòng ngừa bế tắc
    - Tác động vào các điều kiện xảy ra bế tắc
  - Dự báo và phòng tránh
    - Ngăn ngừa hệ thống rơi vào tình trạng có thể dẫn đến bế tắc
  - Nhận biết và khắc phục
    - Cho phép bế tắc xảy ra, chỉ ra bế tắc và khắc phục sau

Kết luận