ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC BÁCH KHOA

NGUYÊN LÝ HỆ ĐIỀU HÀNH

Khoa Công nghệ thông tin
ThS. Nguyễn Thị Lệ Quyên

Nội dung môn học

- Giới thiệu
- **Tiến trình và luồng**
- ***Thi giữa kỳ***
- Quản lý bộ nhớ
- Vào/ Ra
- Hệ thống file
- Thực hành

TIẾN TRÌNH VÀ LUỒNG

Nội dung

- **Tiến trình**
- Luồng
- *Giao tiếp liên tiến trình*
- *Lập lịch*

Định nghĩa

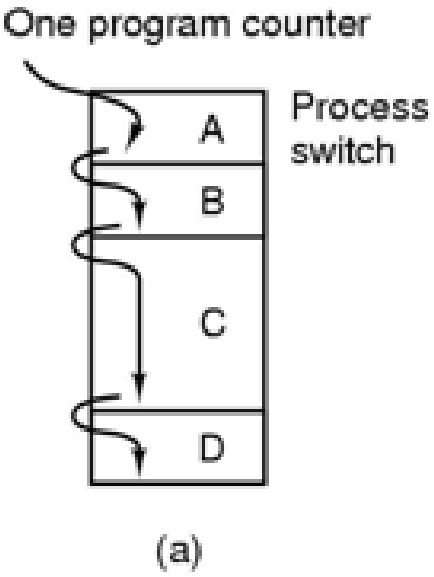
- **Tiến trình:** 1 chương trình đang thực thi
- **Đặc trưng:**
  - Một chương trình được nạp vào bộ nhớ và thực thi
  - Liên kết với mỗi tiến trình là tập hợp các tài nguyên như mã thực thi, dữ liệu, ngăn xếp, giá trị thanh ghi CPU, PC và các thông tin khác để chạy chương trình
  - Liên kết với mỗi tiến trình là không gian địa chỉ của nó (tức là tất cả các vị trí bộ nhớ mà tiến trình có thể đọc/ ghi)
  - Trong một số HDH, 1 tiến trình có thể tạo ra tiến trình con của nó
- Một số HDH hỗ trợ khả năng thực hiện các hoạt động đồng thời ngay cả khi chỉ có 1 CPU (giả song song) dựa trên cơ chế chia sẻ thời gian

Process Model

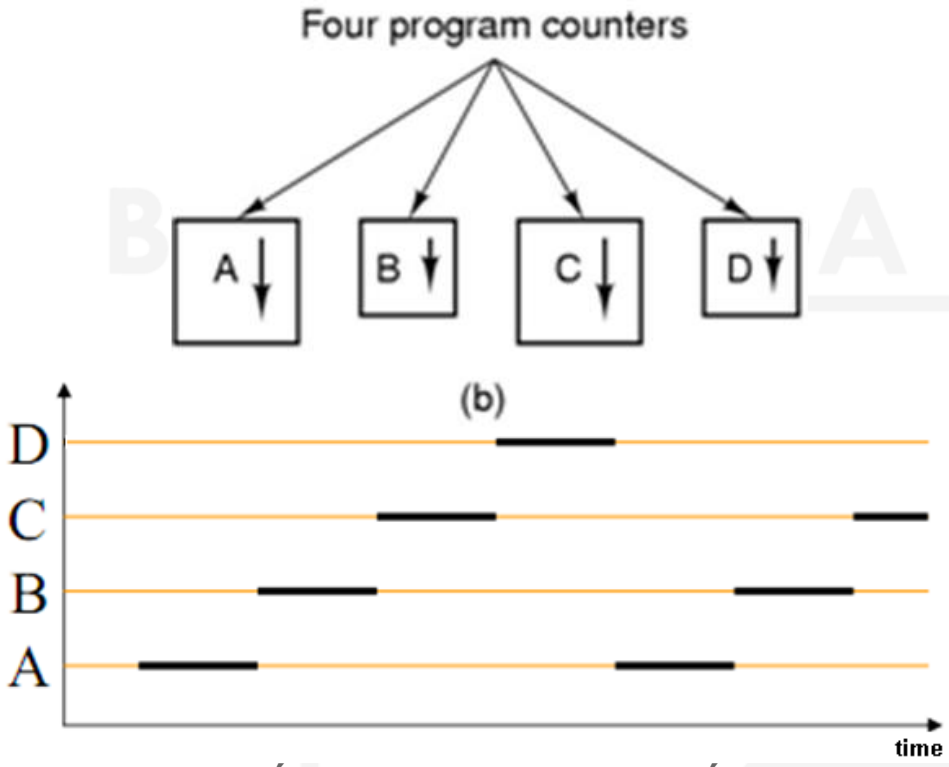
- Các máy tính đời đầu chỉ cho phép thực thi 1 chương trình tại mỗi thời điểm
- Một tiến trình về cơ bản là một hoạt động, có có một chương trình, input, output, trạng thái
- Có 2 khái niệm cơ bản:
  - **Khi nào một tiến trình chạy?** → Thực thi tuần tự: không có sự đồng thời, mọi thứ diễn ra tuần tự (chỉ có 1 CPU và 1 PC vật lý)
  - **HDH quản lý 1 tiến trình như thế nào?** → Trạng thái tiến trình (process state): mọi thứ mà tiến trình tương tác (các thanh ghi, bộ nhớ, file, ...)

Process Model: PC

- Bộ đếm chương trình: 1 biến duy trì thứ tự câu lệnh thực thi (địa chỉ câu lệnh thực thi tiếp theo)



Trong hệ thống đơn nhiệm như MS-DOS, chỉ cần 1 PC



Trong hệ thống đa nhiệm với chế độ giả song song sử dụng chia sẻ thời gian, mỗi bộ đếm được duy trình cho một chương trình

Process Model

- Hệ thống đơn xử lý: Giả song song
- Đa tiến trình:
  - Chuyển đổi giữa các tiến trình (CPU chuyển qua lại giữa các tiến trình). Một bộ xử lý có thể được chia sẻ giữa nhiều tiến trình → Thuật toán lập lịch, phân chia thời gian
  - Thời gian dành cho context switch tốn chi phí vì hệ thống không hoạt động hữu ích trong thời gian chuyển đổi. Nó thay đổi tùy thuộc vào phần cứng của máy
  - Nó trở thành một nút cổ chai về hiệu suất
- **Context switch:** Chuyển CPU sang tiến trình khác
  - Nhiệm vụ: (1) Lưu trạng thái của tiến trình hiện tại (2) Nạp trạng thái đã lưu của tiến trình mới vào

Tiến trình được tạo ra như thế nào?

Các sự kiện tạo tiến trình

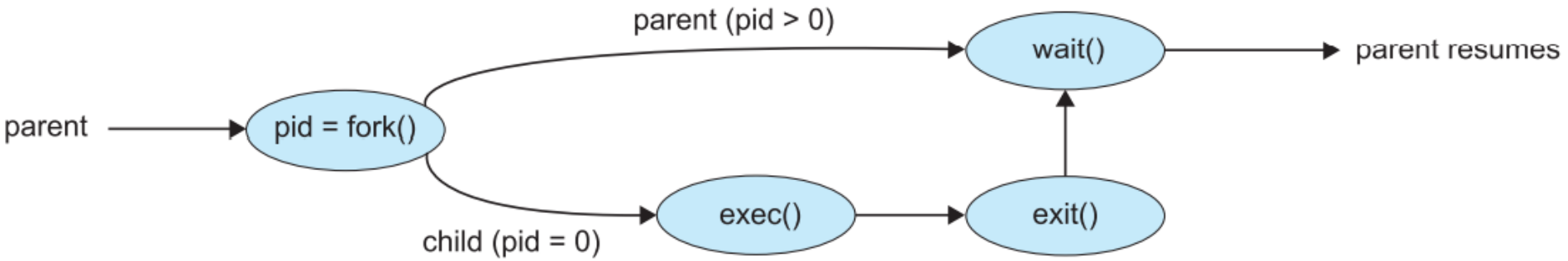
- **Khởi tạo hệ thống:** một số tiến trình được tạo có thể là foreground process (ứng dụng) hoặc background process (dịch vụ)
- **Thực thi system call tạo tiến trình bằng 1 tiến trình đang chạy**: khi system call tạo tiến trình được thực thi (fork() hoặc exec() trong UNIX, CreateProcess() trong Win32)
- **Người dùng yêu cầu tạo một tiến trình mới**
- **Bắt đầu một công việc hàng loạt** (batch job)

Tạo tiến trình

- Tạo 1 tiến trình riêng biệt bằng system call fork() trong UNIX

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Tạo tiến trình



Tạo tiến trình

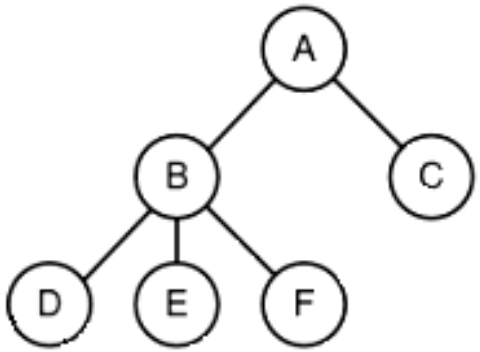
- Tạo tiến trình riêng biệt sử dụng Windows API

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Dừng tiến trình

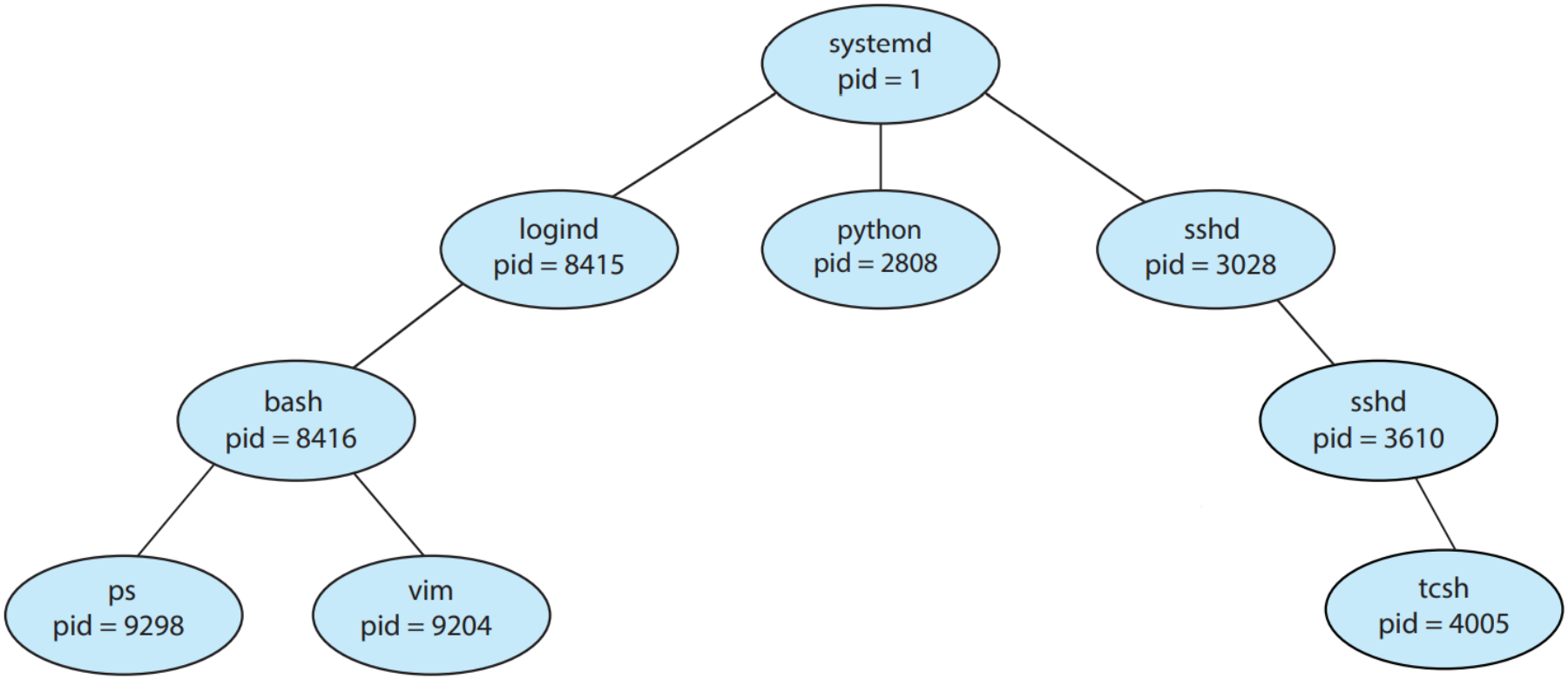
- Sau khi tiến trình được tạo, nó thường bị dừng bởi những nguyên nhân sau:
  - **Dừng bình thường** (normal exit) (nhiệm vụ hoàn thành, tự nguyện)
  - **Dừng do lỗi** (tự nguyện)
    - VD: file không tồn tại, input không đầy đủ hoặc không chính xác
  - **Dừng do lỗi nghiêm trọng** (không tự nguyện) – program bug
    - VD: câu lệnh không hợp lệ, chia cho số 0, ...
  - **Bị dừng bởi một tiến trình khác** (không tự nguyện)
    - Sử dụng system call kill trong Unix hoặc TerminateProcess trong Win32 (trong một vài hệ thống thì tiến trình cha bị dừng)
- **Tự nguyện** – sử dụng system call đặc biệt
- **Không tự nguyện** – nhận gián đoạn (ngoại lệ exception)

Phân cấp tiến trình



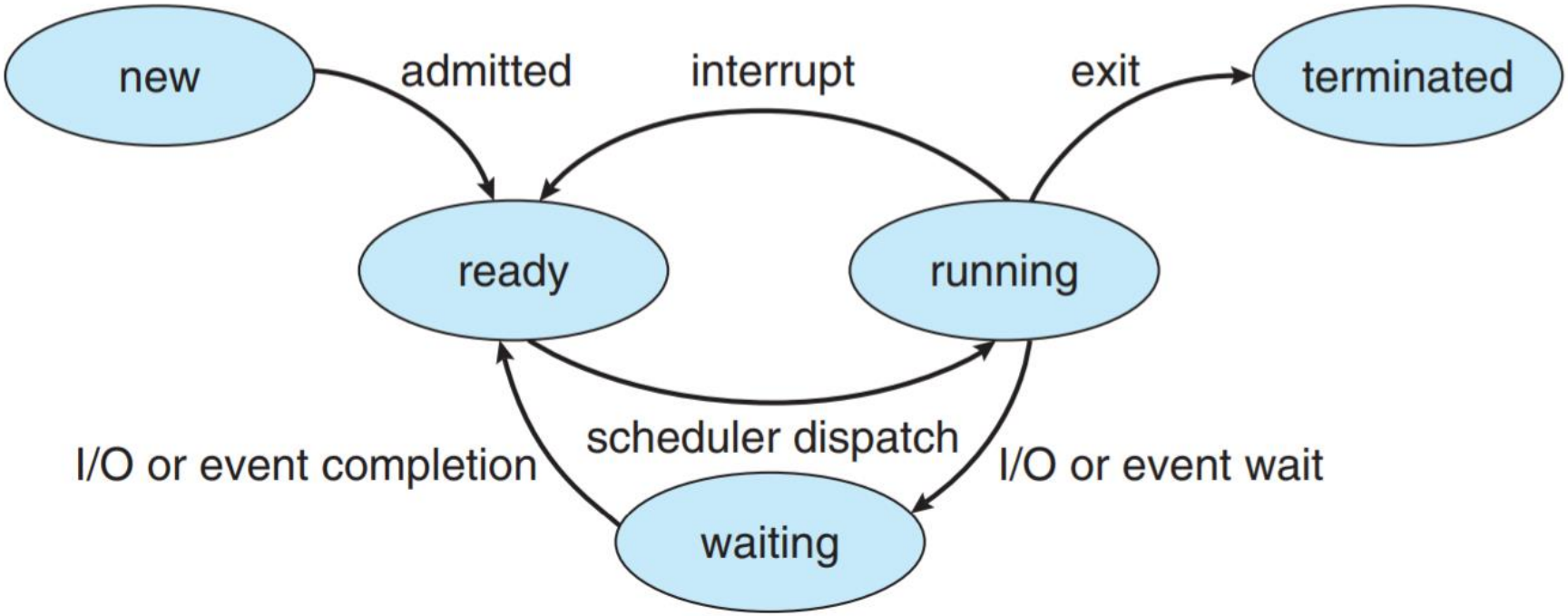
- Phân cấp tiến trình
  - Mỗi quan hệ cha con
  - Tiến trình con cũng có thể tạo tiến trình mới
  - Cây tiến trình
- **Trong HDH Unix**
  - **1 tiến trình có thể tạo tiến trình khác**, tiến trình cha và tiến trình con tiếp tục được liên kết theo những cách nhất định
  - 1 tiến trình cùng với tất cả tiến trình con và các hậu duệ tiếp theo của nó cùng nhau tạo thành một nhóm tiến trình (process group)
- **Trong HDH Windows**
  - **Tất cả các tiến trình đều bình đẳng** (ko có khái niệm về phân cấp)
  - Tiến trình ban đầu được cấp 1 token đặc biệt (được gọi là handle, process ID) có thể được sử dụng để kiểm soát tiến trình nó tạo ra

Phân cấp tiến trình



Trạng thái tiến trình (Process state)

- Sơ đồ trạng thái tiến trình



Process state

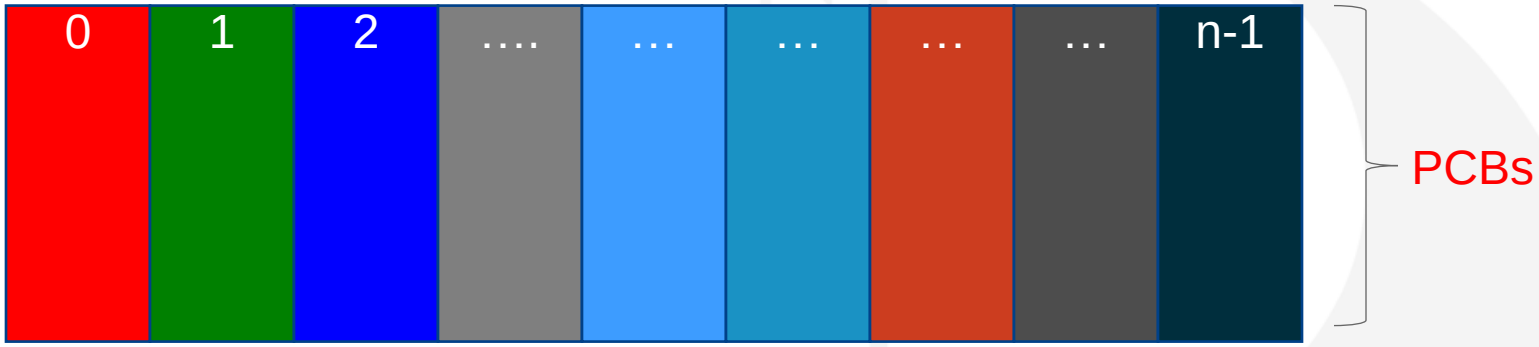
- **New:** Đang chờ phân bổ tài nguyên (hoặc tiến trình đang được tạo)
- **Ready:** Có thể chạy và chờ đến lượt (chờ CPU) vì một tiến trình khác đang chạy.
- **Running**: Đang sử dụng CPU, các lệnh của nó đang được thực thi.
- **Waiting:** Không thể chạy cho đến khi xảy ra một số sự kiện bên ngoài, như chờ dữ liệu được nhập từ bàn phím, mạng, dữ liệu được ghi vào ổ đĩa,…
- **Terminated:** Dữ nguyên thông tin về trạng thái dừng (hoặc chương trình đã thực hiện xong)

Trạng thái chuyển tiếp (Transition State)

- New to Ready
- Ready to Running (dispatch)
  - Đến lượt nó được chạy.
  - Được chọn bởi bộ lập lịch.
- Running to Ready (interrupt)
  - time slice expired
  - Bị tạm dừng bởi bộ lập lịch
- Running to Waiting (block)
  - Chờ đợi một sự kiện nào đó xảy ra
- Waiting to Ready (ready)
  - Sự kiện được chờ đợi đã xảy ra
- Running to Terminated (exit)

Triển khai tiến trình

- HDH duy trì 1 danh sách các khối điều khiển tiến trình **P**rocess **C**ontrol **B**locks (PCB)
  - Mỗi mục PCB chứa thông tin về tiến trình
  - Nó được sử dụng như một kho lưu trữ khi tiến trình bị tạm dừng hoặc chặn
  - Cấu trúc của PCB



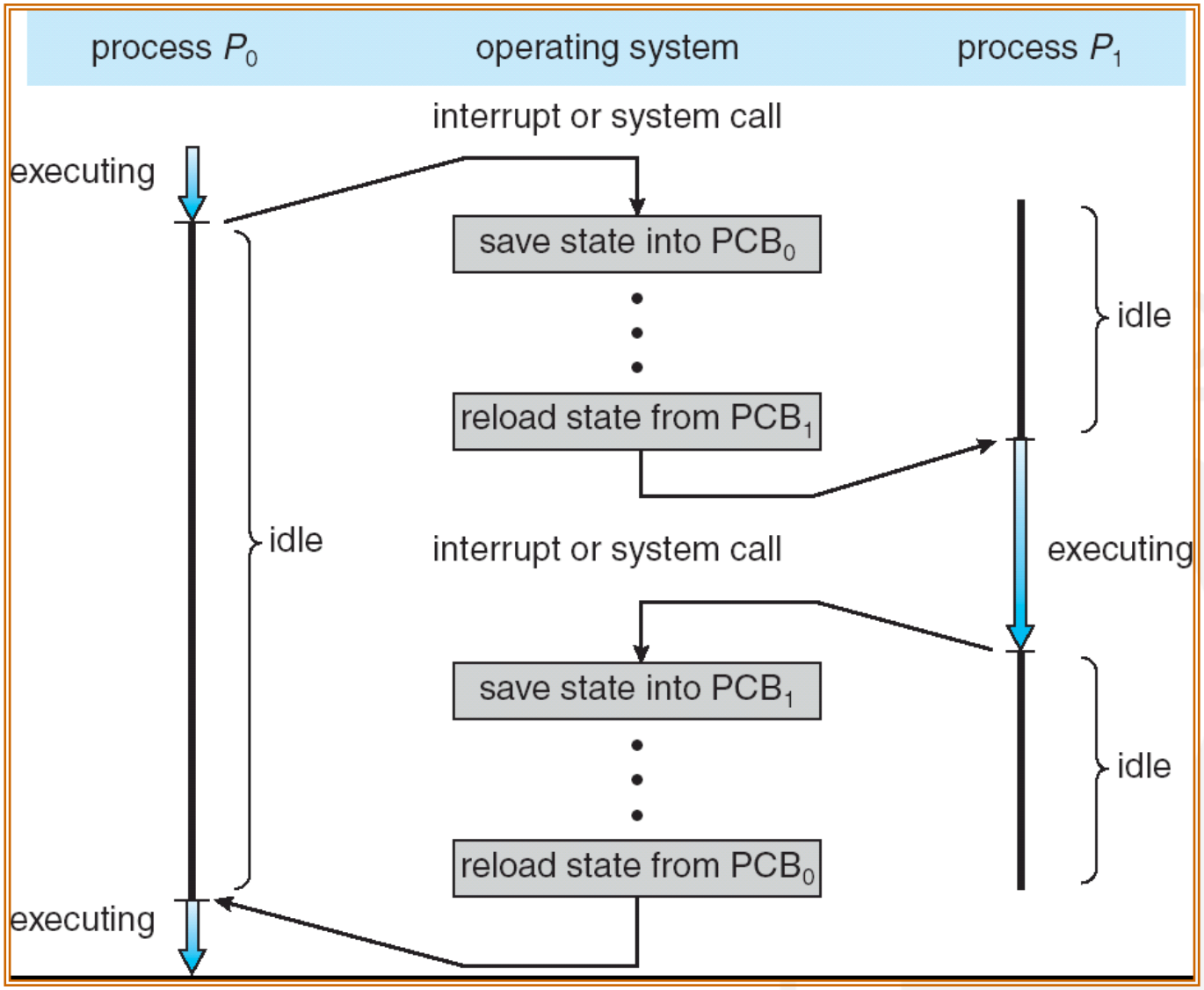
Bộ lập lịch sẽ sử dụng bảng này để chọn tiến trình hiện tại

Cấu trúc PCB

Figure: Some of the fields of a typical process table entry

Process Management	Memory Management	File Management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters ProcessID Parent process Process group Signals Time when process started CPU time used Children CPU's time Time of next alarm	Pointer to text segment info Pointer to data segment info Pointer to stack segment info	Root directory Working directory File descriptors UserID GroupID

Chuyển đổi giữa các tiến trình



Nội dung

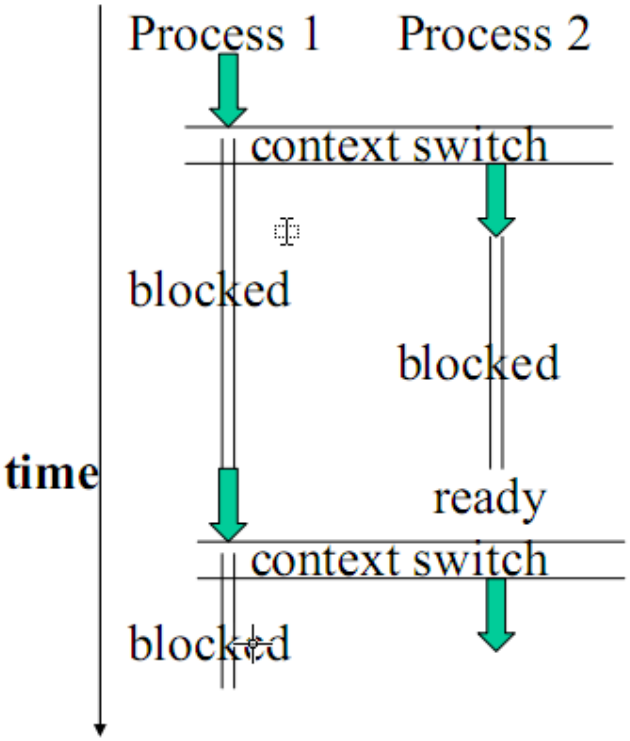
- Tiến trình
- **Luồng**
- Giao tiếp liên tiến trình
- Lập lịch

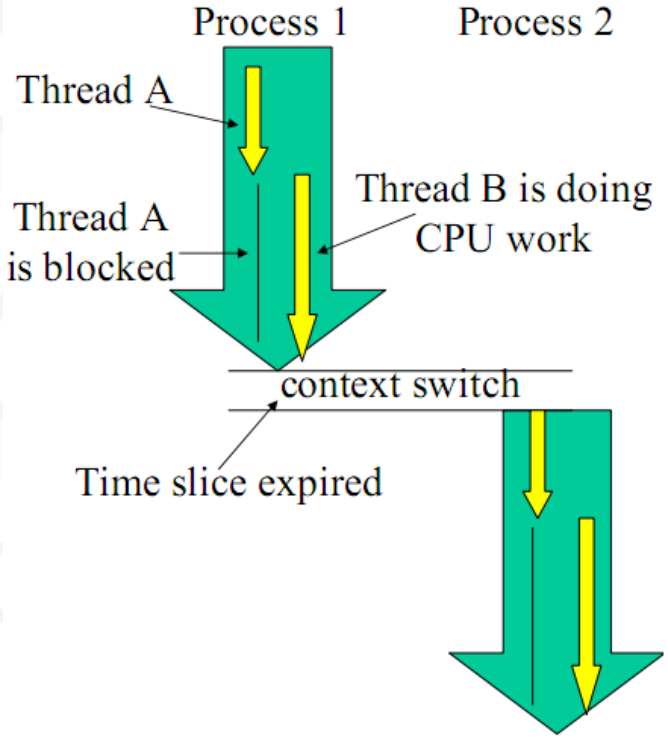
Luồng

- Mỗi tiến trình có 1 không gian địa chỉ khác nhau, CPU chỉ được cấp phát cho một tiến trình tại 1 thời điểm → **Context switch**
- Có thể có 1 tiến trình trong đó một số tác vụ cần được thực hiện đồng thời → Cần có luồng
- Nếu luồng không được triển khai
  - **Trong dịch vụ mạng**
    - Server chỉ có thể phục vụ 1 client trong 1 thời điểm
  - **Trong Word**
    - Word cần hỗ trợ vài tính năng như tự động lưu file mỗi 5 phút, đọc người dùng gõ trên bàn phím và hiển thị đồ họa, kiểm tra ngữ pháp, chính tả, ...
    - → khi việc lưu tự động được thực hiện, việc đọc hoặc hiển thị có thể không được tiến hành

Luồng

- Cần có nhiều luồng điều khiển trong cùng một không gian địa chỉ chạy gần như song song (quasi-parallel) như thể chúng là các tiến trình riêng biệt
- Có nhiều luồng chạy đồng thời trong một tiến trình cũng tương tự như có nhiều tiến trình chạy song song trong một máy tính (kỹ thuật đa luồng)

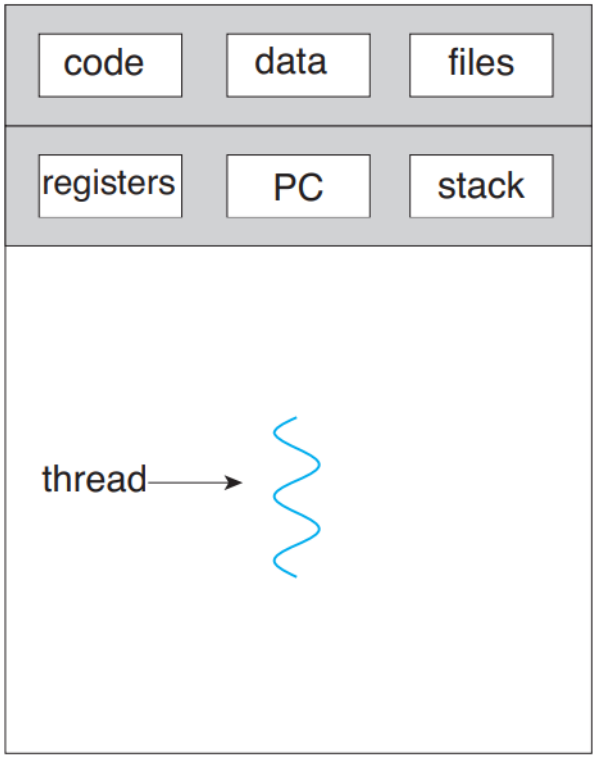




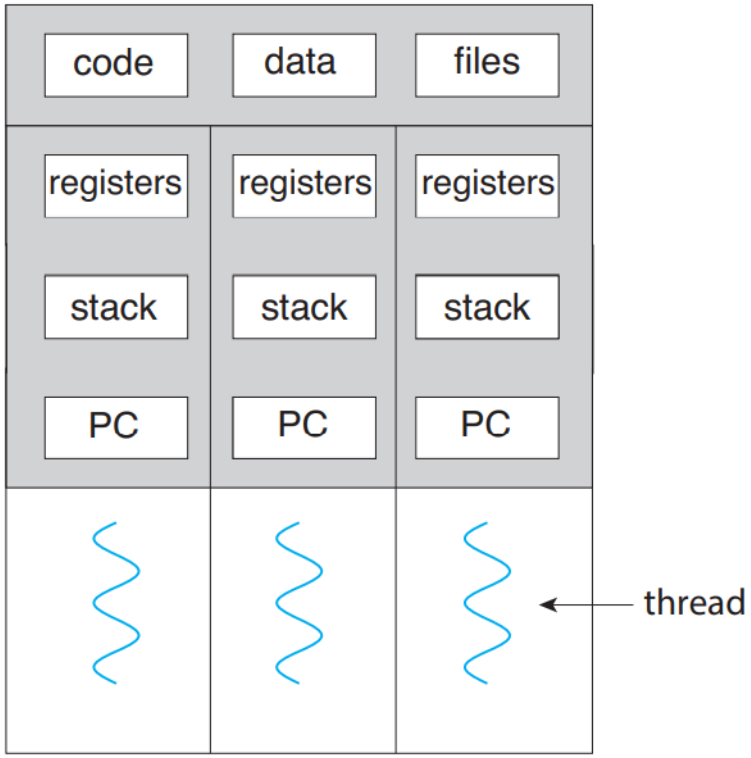
Luồng

- Luồng là 1 đơn vị thực thi của tiến trình → a function
- Cũng được gọi là mini-process, light-process
- Thuộc tính của luồng:
  - Mô tả việc thực hiện tuần tự trong một tiến trình
  - Chia sẻ cùng 1 không gian địa chỉ và tài nguyên của tiến trihf
  - Mỗi luồng có 1 PC, các thanh ghi và stack thực thi riêng
  - Không có sự bảo vệ giữa các luồng trong 1 tiến trình.
  - Chứa một số thuộc tính của tiến trình

Luồng



single-threaded process

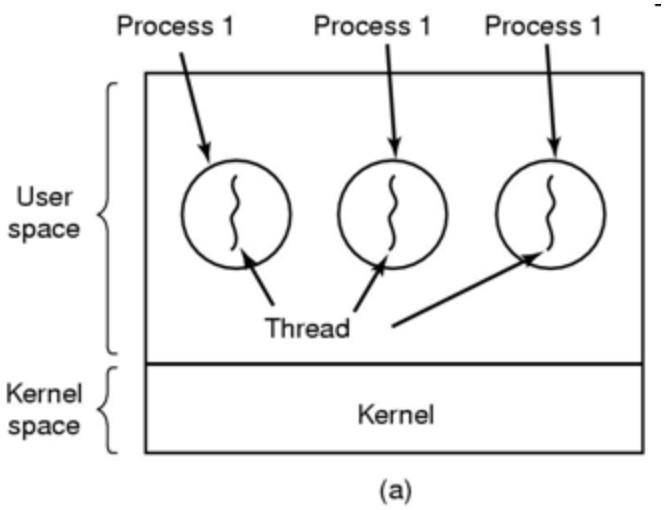


multithreaded process

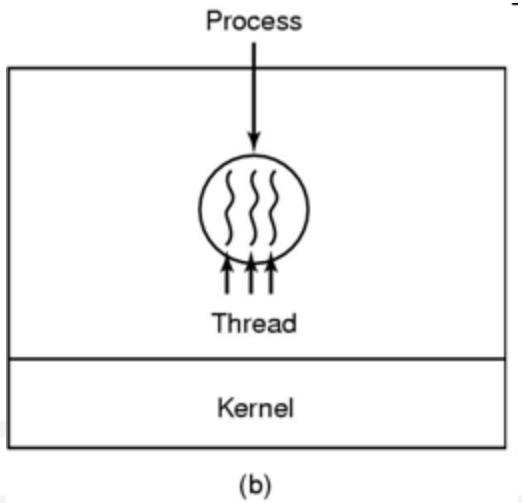
Đa luồng

- HDH hỗ trợ cho phép nhiều luồng thực thi trong 1 tiến trình
- MS-DOS hỗ trợ đơn luồng
- UNIX hỗ trợ nhiều tiến trình người dùng nhưng chỉ hỗ trợ một luồng cho mỗi tiến trình
- Windows 2000, Solaris, Linux, Mach và OS/2 hỗ trợ đa luồng
- Đa luồng thực sự hiệu quả trong các bộ xử lý đa luồng vì các luồng có thể thực thi đồng thời
- ...

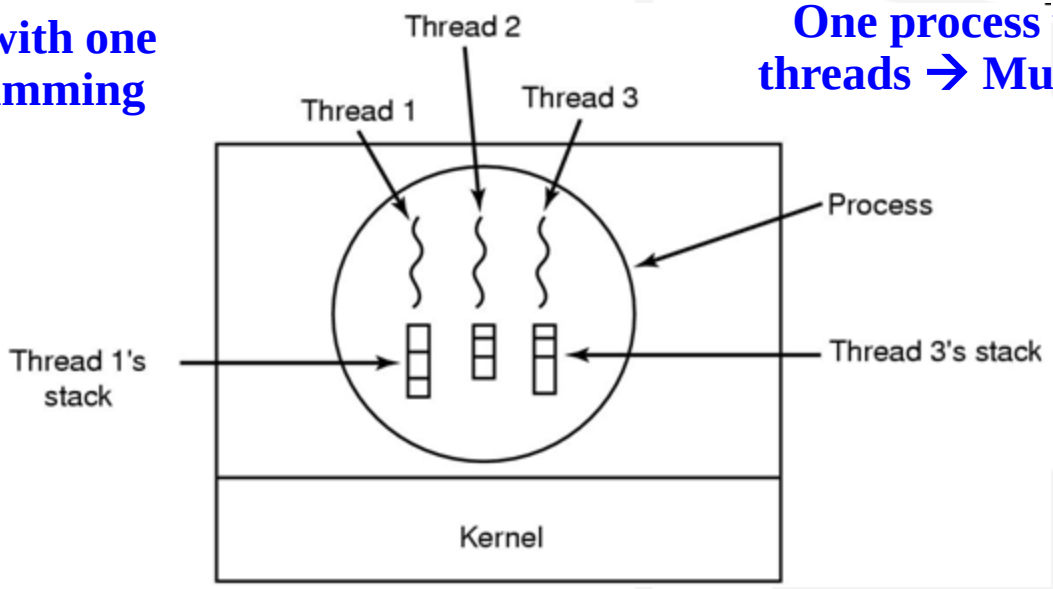
Mô hình luồng



Three processes, each with one thread → Multiprogramming



One process with three threads → Multithreading



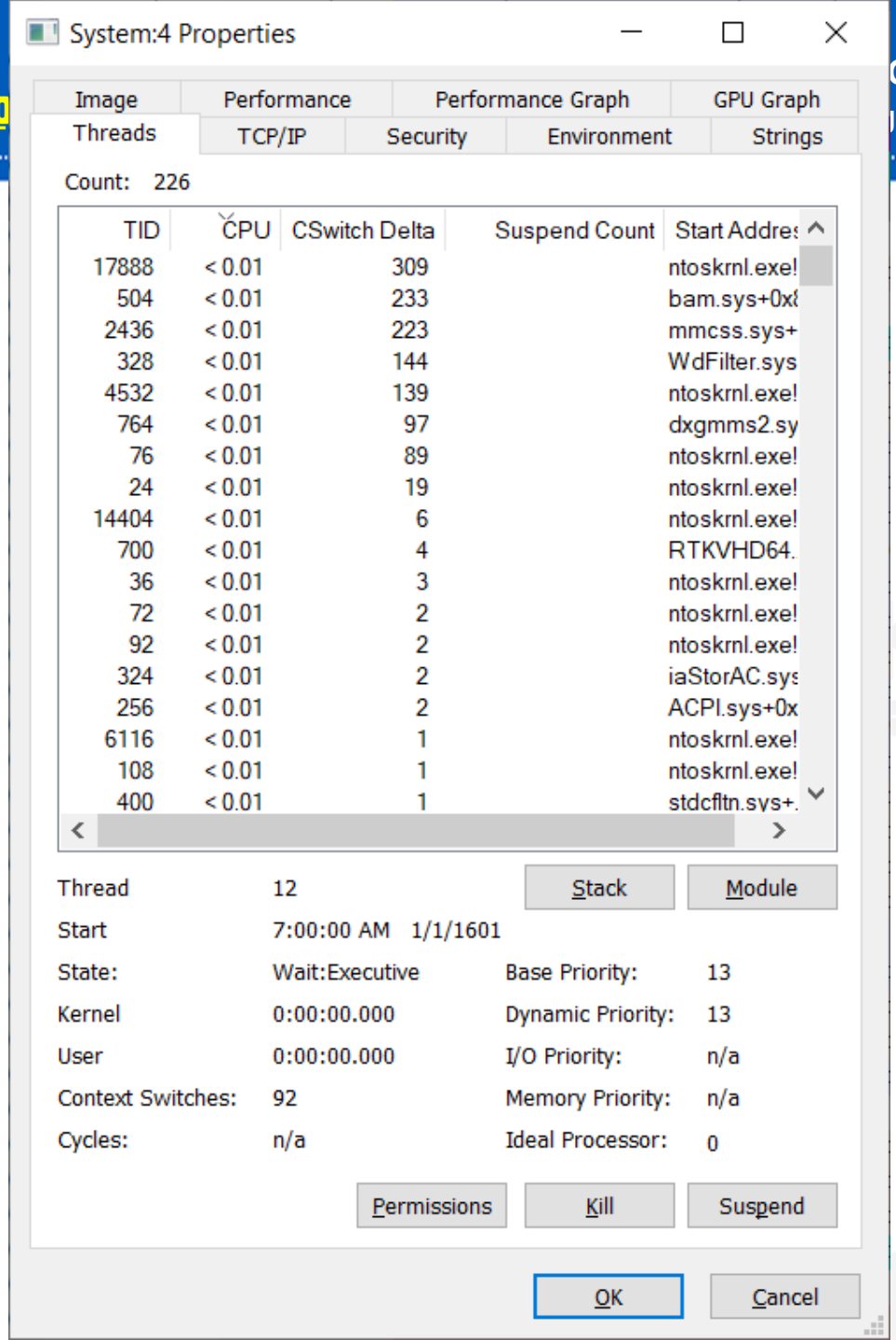
Each thread has it's own stack

Luồng

- Khả năng đáp ứng và chia sẻ tài nguyên tốt hơn
- Một chương trình có thể tiếp tục chạy ngay cả khi một phần của nó bị chặn
- Hiệu suất của ứng dụng có thể cải thiện vì có thể chồng chéo I/O và CPU
- Kinh tế:
  - Việc phân bổ bộ nhớ và tài nguyên để tạo tiến trình (nhanh hơn, dễ dàng hơn) rất tốn kém
  - Tạo luồng có thể nhanh hơn 100 lần so với tạo tiến trình
- Hữu ích trên các hệ thống nhiều CPU
- Ít thời gian hơn để dừng 1 luồng so với dừng 1 tiến trình
- Ít thời gian hơn để chuyển đổi giữa 2 luồng trong cùng 1 tiến trình (phục vụ nhiều tác vụ có cùng mục đích)
- Vì các luồng trong cùng 1 tiến trình chia sẻ bộ nhớ và file nên chúng có thể giao tiếp với nhau mà không cần gọi kernel

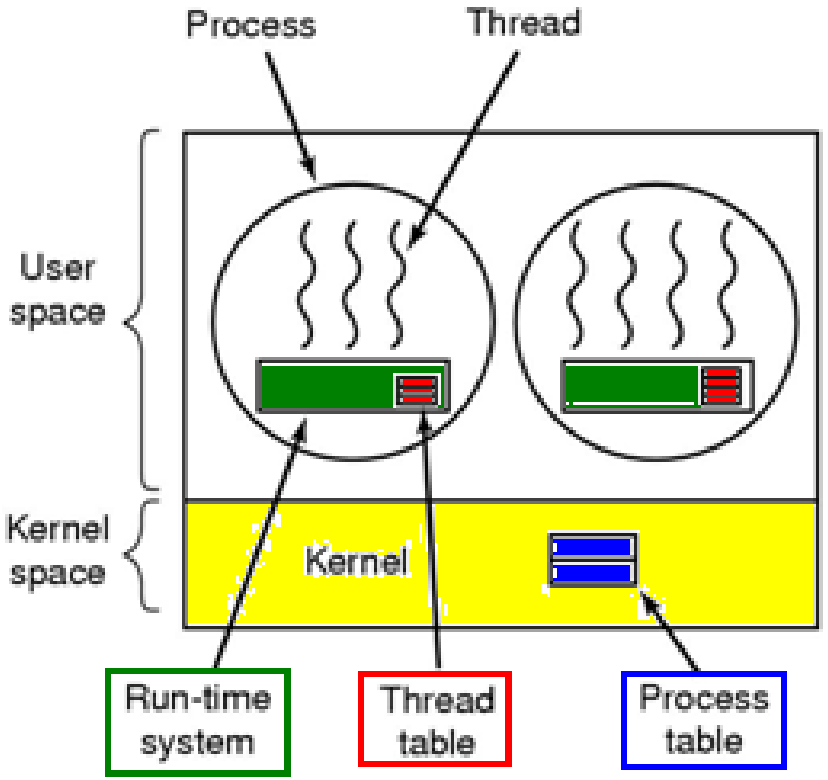
Luồng

- Có 1 số vấn đề phức tạp:
  - VD vì chia sẻ dữ liệu nên 1 luồng có thể đọc và 1 luồng khác có thể ghi vào cùng 1 vị trí → cần phải cẩn thận


System:4 Properties
Image
Performance
Performance Graph
GPU Graph
Threads
TCP/IP
Security
Environment
Strings
Count: 226
TID CPU CSwitch Delta Suspend Count Start Addres
17888 < 0.01 309 ntoskrnl.exe!
504 < 0.01 233 bam.sys+0x8
2436 < 0.01 223 mmcss.sys+
328 < 0.01 144 WdFilter.sys
4532 < 0.01 139 ntoskrnl.exe!
764 < 0.01 97 dxgmms2.sy
76 < 0.01 89 ntoskrnl.exe!
24 < 0.01 19 ntoskrnl.exe!
14404 < 0.01 6 ntoskrnl.exe!
700 < 0.01 4 RTKVHD64.
36 < 0.01 3 ntoskrnl.exe!
72 < 0.01 2 ntoskrnl.exe!
92 < 0.01 2 ntoskrnl.exe!
324 < 0.01 2 iaStorAC.sys
256 < 0.01 2 ACPI.sys+0x
6116 < 0.01 1 ntoskrnl.exe!
108 < 0.01 1 ntoskrnl.exe!
400 < 0.01 1 stdcfltn.sys+.
Thread 12
Stack
Module
Start 7:00:00 AM 1/1/1601
State: Wait:Executive
Base Priority: 13
Kernel 0:00:00.000
Dynamic Priority: 13
User 0:00:00.000
I/O Priority: n/a
Context Switches: 92
Memory Priority: n/a
Cycles: n/a
Ideal Processor: 0
Permissions
Kill
Suspend
OK
Cancel

Triển khai Luồng tại User Space

- Kernel không biết gì về luồng
  - Cách tiếp cận này phù hợp với HDH không hỗ trợ thread
  - Thread được triển khai bởi 1 thư viện cấp người dùng (user-level library) (với code và cấu trúc dữ liệu)
- Các thread chạy trên runtime system (là tập hợp các thủ tục quản lý thread)
- Mỗi tiến trình có 1 bảng thread riêng

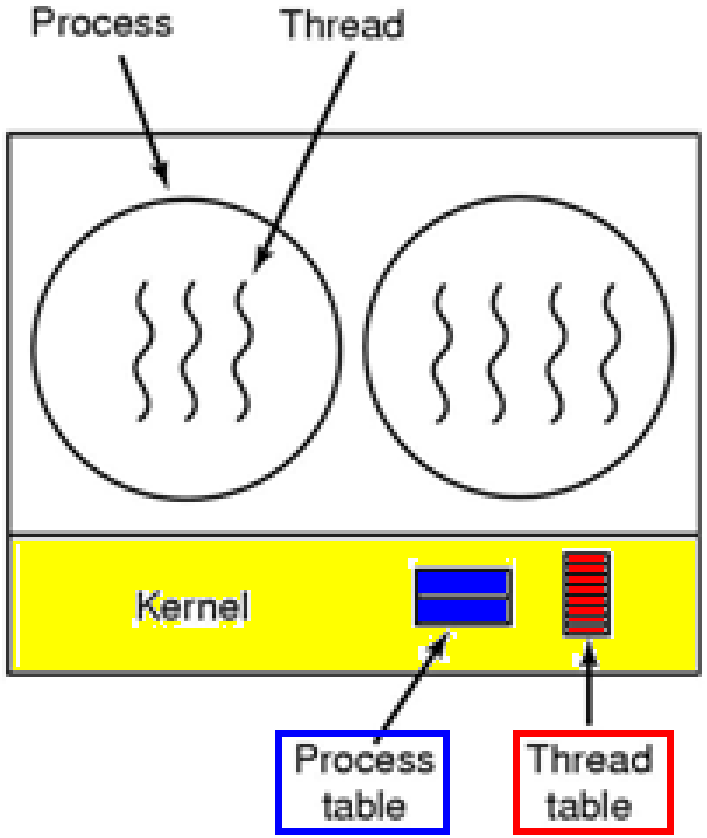


Triển khai Luồng tại User Space

- **Ưu điểm**
  - Nhanh hơn: Chuyển đổi và lập lịch thread nhanh hơn (vì được thực hiện ở user mode) so với phải dùng trap để chuyển sang kernel mode
  - Linh hoạt, mở rộng tốt hơn: Mỗi tiến trình có thể có thuật toán lập lịch riêng. Nó có thể thay đổi không gian bảng và không gian ngăn xếp 1 cách linh hoạt
- **Nhược điểm**
  - Thực hiện chặn system call rất phức tạp → thay vì chặn thread thì tiến trình bị chặn
  - Nhu cầu 1 thread tự nguyện bỏ CPU → HDH ko biết điều này, vì vậy nếu bất kỳ thread cấp người dùng bị chặn, toàn bộ tiến trình sẽ bị chặn
  - Developers muốn thread chính xác trong ứng dụng → thực hiện system call liên tục

Triển khai thread tại kernel

- Kernel biết về thead và quản lý thread (ko cần runtime system)
- Kernel lập lịch tất cả thread
- Kernel có 1 bảng thread (sử dụng lời gọi kernel để tạo hoặc hủy thread)

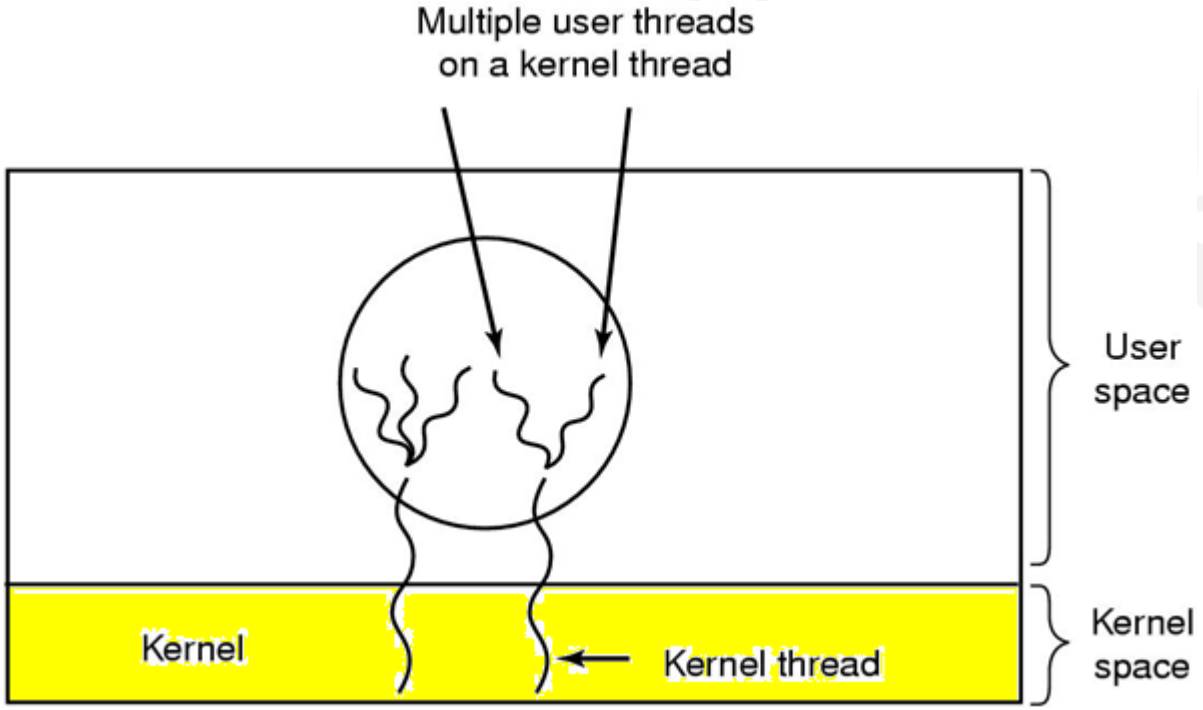


Triển khai thread tại kernel

- Ưu điểm
  - Kernel có thể chuyển đổi giữa các thread thuộc các tiến trình khác nhau → không gặp vấn đề với việc chặn system call
  - Hữu ích nếu có hỗ trợ đa xử lý (nhiều CPU)
- Nhược điểm
  - Chi phí lớn hơn (thời gian và nguồn lực để quản lý việc tạo và dừng thread)
  - Tạo, lưu thread chậm (cần system call)
- Vấn đề:
  - Điều gì xảy ra khi tiến trình đa luồng fork?
  - Khi có tin hiệu đến, thread nào sẽ xử lý nó?

Triển khai kết hợp

- Kết hợp ưu điểm của cả user-level thread và kernel-level thread



Sử dụng kernel-level thread và sau đó ghép user-level thread vào một số hoặc tất cả các kernel thread (tính linh hoạt cao nhất)

3 thư viện thread nguyên thủy

- POSIX Pthreads (UNIX)
  - Có thể được cung cấp dưới dạng thư viện user-level hoặc kernel-level
- Win32 threads (Windows)
  - Thư viện kernel-level, có sẵn trên hệ thống Windows
- Java threads (JAVA)
  - JVM đang chạy trên HDH host, triển khai phụ thuộc vào hệ thống host (API Win32 hoặc Pthreads)

Nội dung

- Tiến trình
- Luồng
- **Giao tiếp liên tiến trình**
- Lập lịch

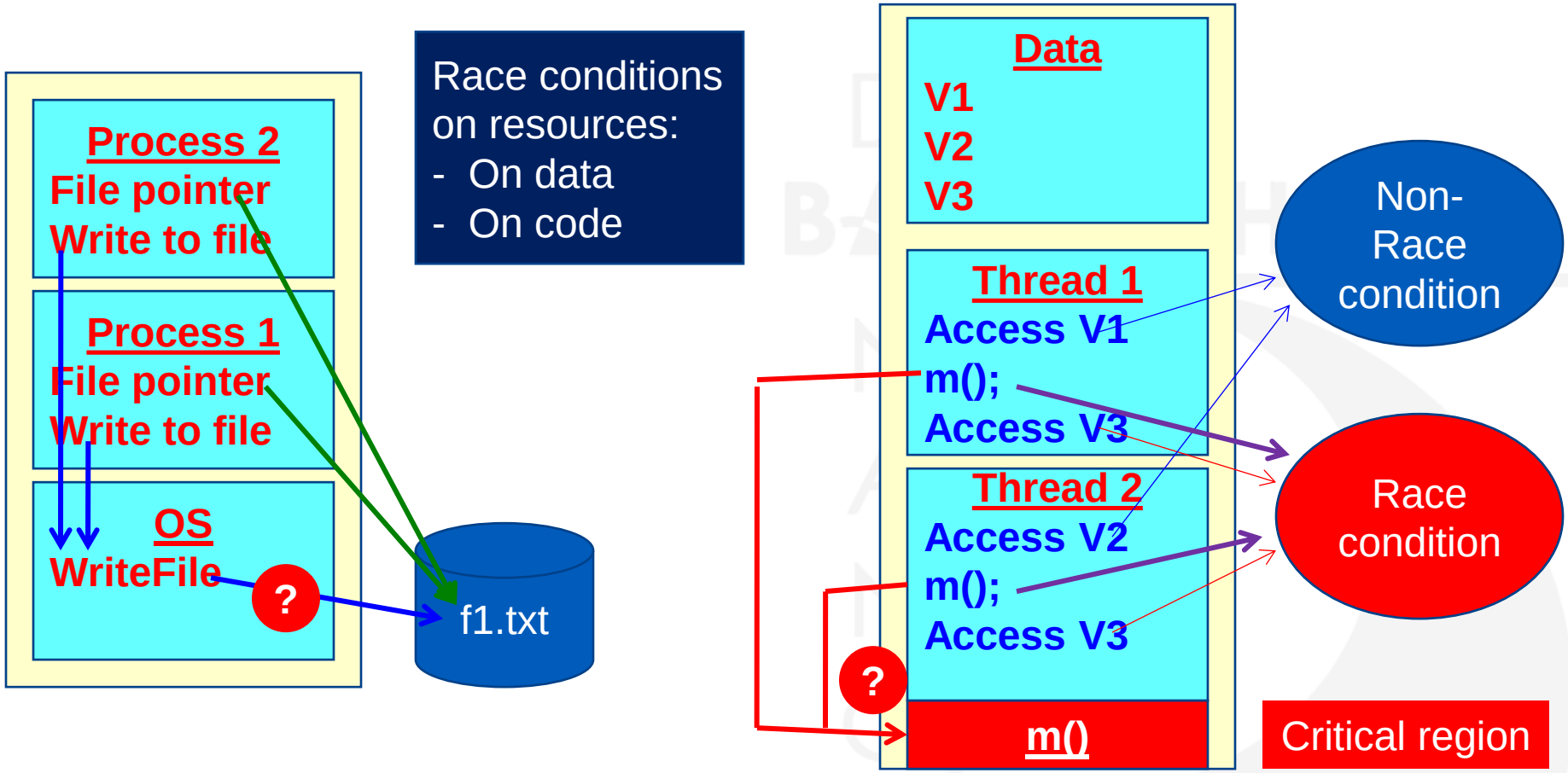
Giao tiếp liên tiến trình

- InterProcess Communication (IPC)
- Làm thế nào 1 tiến trình có thể truyền thông tin đến 1 tiến trình khác?
- Waiting: Trình tự thích hợp khi có sự phụ thuộc: nếu tiến trình A tạo ra dữ liệu và tiến trình B in chúng, thì B phải đợi cho đến khi A tạo ra dữ liệu trước khi bắt đầu in

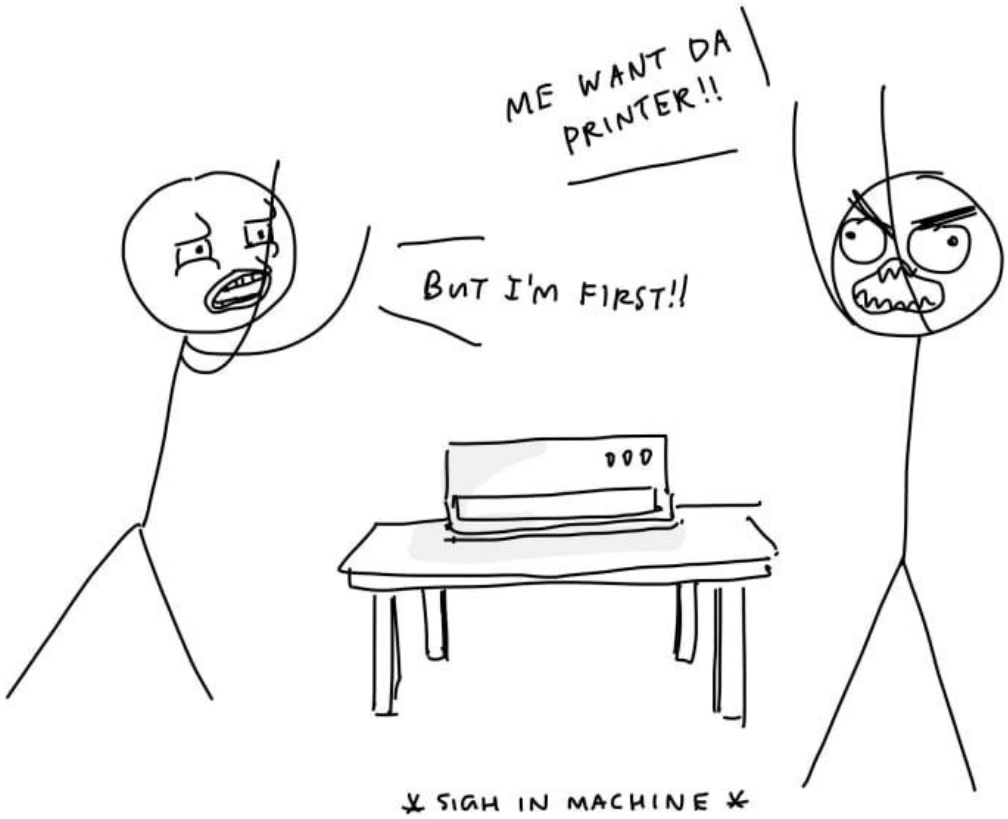
Giao tiếp liên tiến trình

- Busy Waiting
- Sleep & Wakeup
- Semaphores
- Monitors
- Message Passing

Race conditions

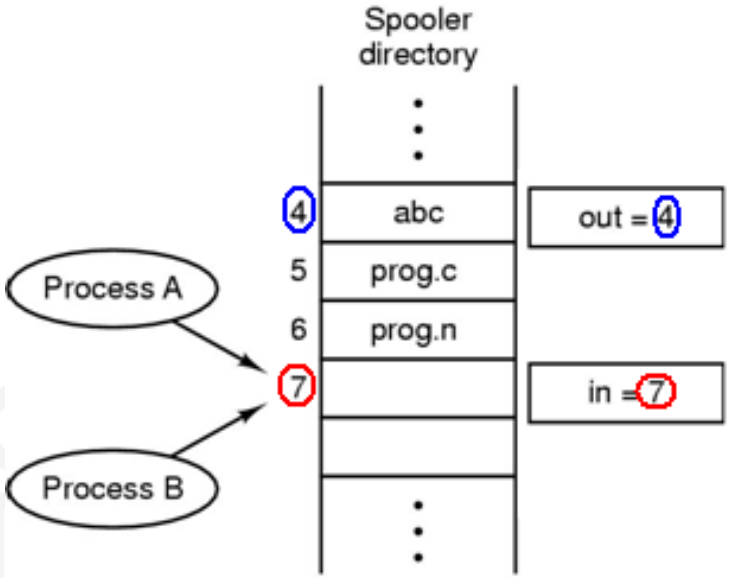


Race conditions



Race conditions

- Hai hoặc nhiều thread truy cập đồng thời vào 1 tài nguyên được chia sẻ:
  - Trạng thái cuối cùng của tài nguyên là không thể đoán trước và có thể không nhất quán
  - Nếu tài nguyên là code fragment thì không thể dự đoán được kết quả



Hai tiến trình muốn truy cập bộ nhớ dùng chung cùng một lúc

- VD: Hàng đợi được sử dụng để quản lý thư mục bộ đệm. Các biến dùng chung để quản lý hàng đợi: **out**: chỉ mục cho việc xếp hàng, **in**: chỉ mục cho việc xếp hàng. Hai thread đồng thời đưa 1 chuỗi vào hàng đợi.

Race conditions

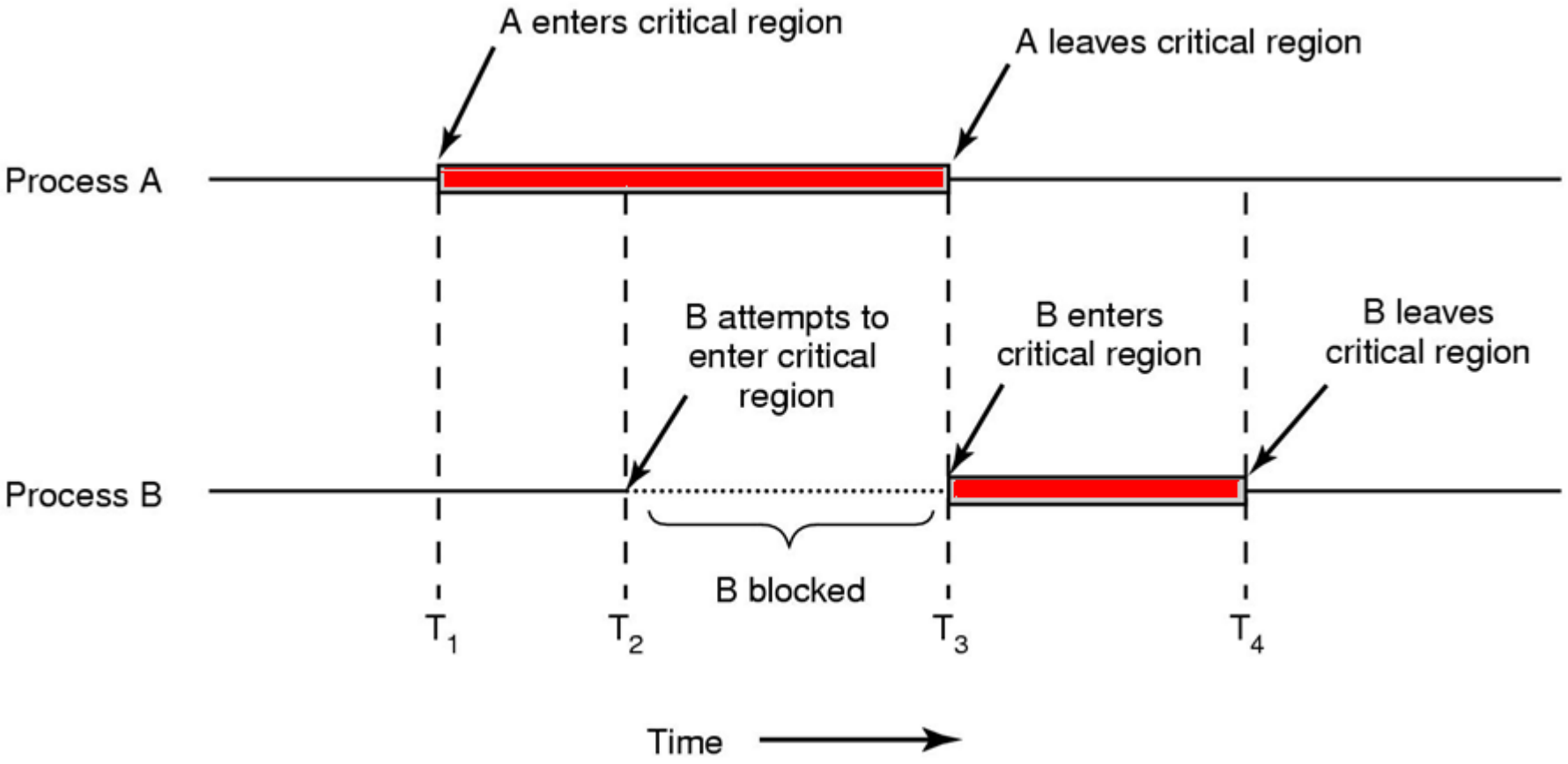
- VD khác trong thực tế
  - Tài khoản ngân hàng có số dư 800, có thể rút bằng 2 thẻ ATM tại 2 địa điểm khác nhau cùng lúc
  - Đầu tiên, user1 đẩy thẻ vào máy ATM và kiểm tra số dư tài khoản. Tiến trình P1 được tạo đồng thời trên máy chủ. Khi đó kết quả kiểm tra là 800. Người dùng chọn rút 400.
  - Ở vị trí khác, user2 cũng thực hiện những việc tương tự như user1 và tiến trình P2 được tạo. User2 chọn rút 500.
  - Trong trường nếu, nếu P1 hết thời gian, P2 được phục vụ trước sau đó người dùng 2 nhận được 500. Sau đó user1 nhận được 400 hoặc nhận được thông báo lỗi.

Critical Regions (vùng găng)

Critical Regions (vùng găng)

- Phần chương trình nơi bộ nhớ dùng chung được truy cập (vùng mã nơi xuất hiện race condition)
- Cùng 1 đoạn code được thực thi bởi 2 hoặc nhiều process/thread.
- Là đoạn code mà việc thực thi phải được quy định để đảm bảo kết quả có thể dự đoán được
- Để tránh race condition, có thể sắp xếp các vấn đề sao cho không có 2 tiến trình nào nằm trong critical region
- Điều kiện cần thiết kể tránh tình trạng race condition:
  - Không có 2 tiến trình nào có thể xảy ra đồng thời bên trong vùng găng
  - Không thể đưa ra giả định về tốc độ hoặc số lượng CPU
  - Không có tiến trình nào chạy bên ngoài critical section có thể chặn tiến trình khác
  - Không có tiến trình nào phải đợi mãi mãi để vào critical section

Critical Regions



Mutual exclusion using critical regions

Mutual Exclusion with Busy Waiting

Mutual Exclusion with Busy Waiting

- Nếu 1 thread/process đang thực thi trong critical section, không có thread/process nào khác có thể thực thi trong critical section của chúng (Chỉ có 1 tiến trình có thể sử dụng tài nguyên được chia sẻ tại một thời điểm)
- Sử dụng một biến bổ sung để khóa tài nguyên chia sẻ chung (spin lock: khóa xoay vòng giúp các tiến trình/ luồng lần lượt truy xuất tài nguyên chung)
- Đề xuất để đạt được mutual exclusion
  - Vô hiệu hóa ngắt (disabling interrupts)
  - Biến khóa (lock variables)
  - Luân phiên nghiêm ngặt (strict alternation)
  - Giải pháp của Peterson)
  - Câu lệnh TSL (kiểm tra và đặt khóa – test and set lock)

Một cách để tránh race condition là tạo một hoặc nhiều biến bổ sung để tạo một tài nguyên chung truy cập nối tiếp. Trước khi truy cập các tài nguyên chung, tất cả các biến bổ sung phải được kiểm tra.

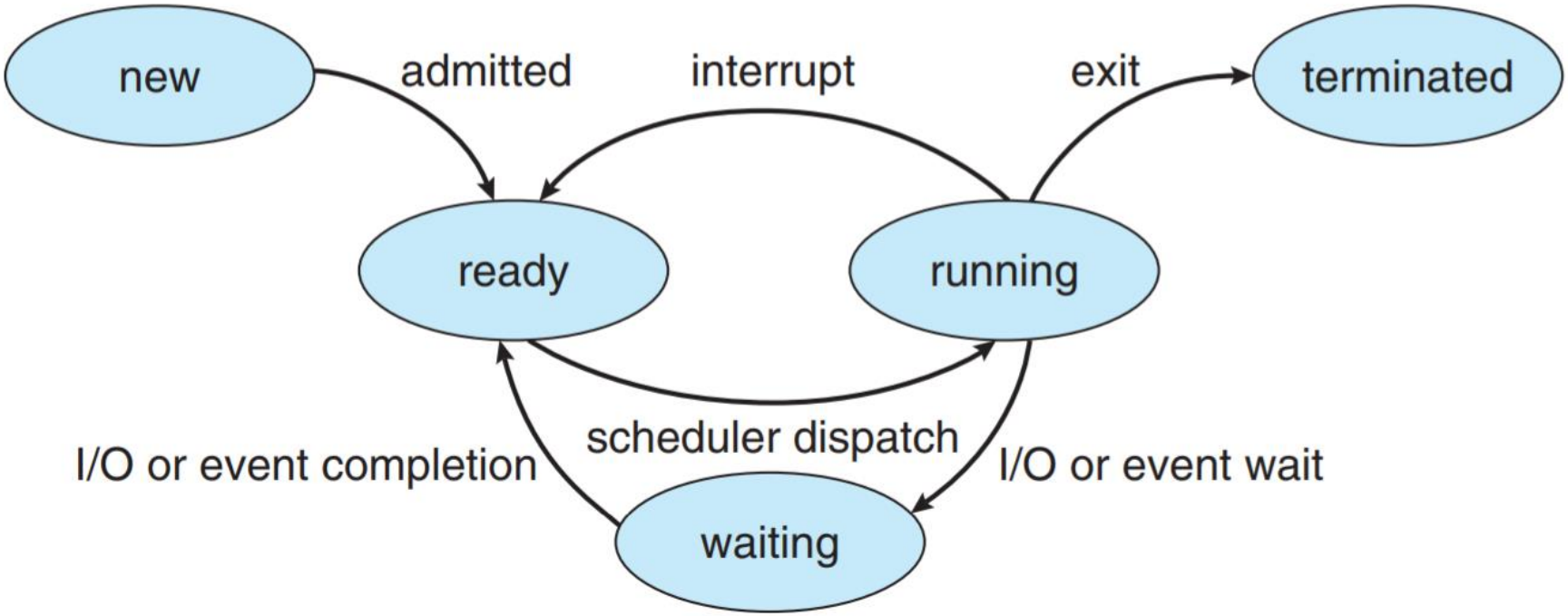
Busy Waiting: A Study Problem

- Yêu cầu về độ chính xác:
  - Không bao giờ có nhiều hơn 1 người mua sữa
  - Có người mua nếu cần
- Giải pháp cho trường hợp này (mutual exclusion) được gọi là đồng bộ hóa (synchronization)

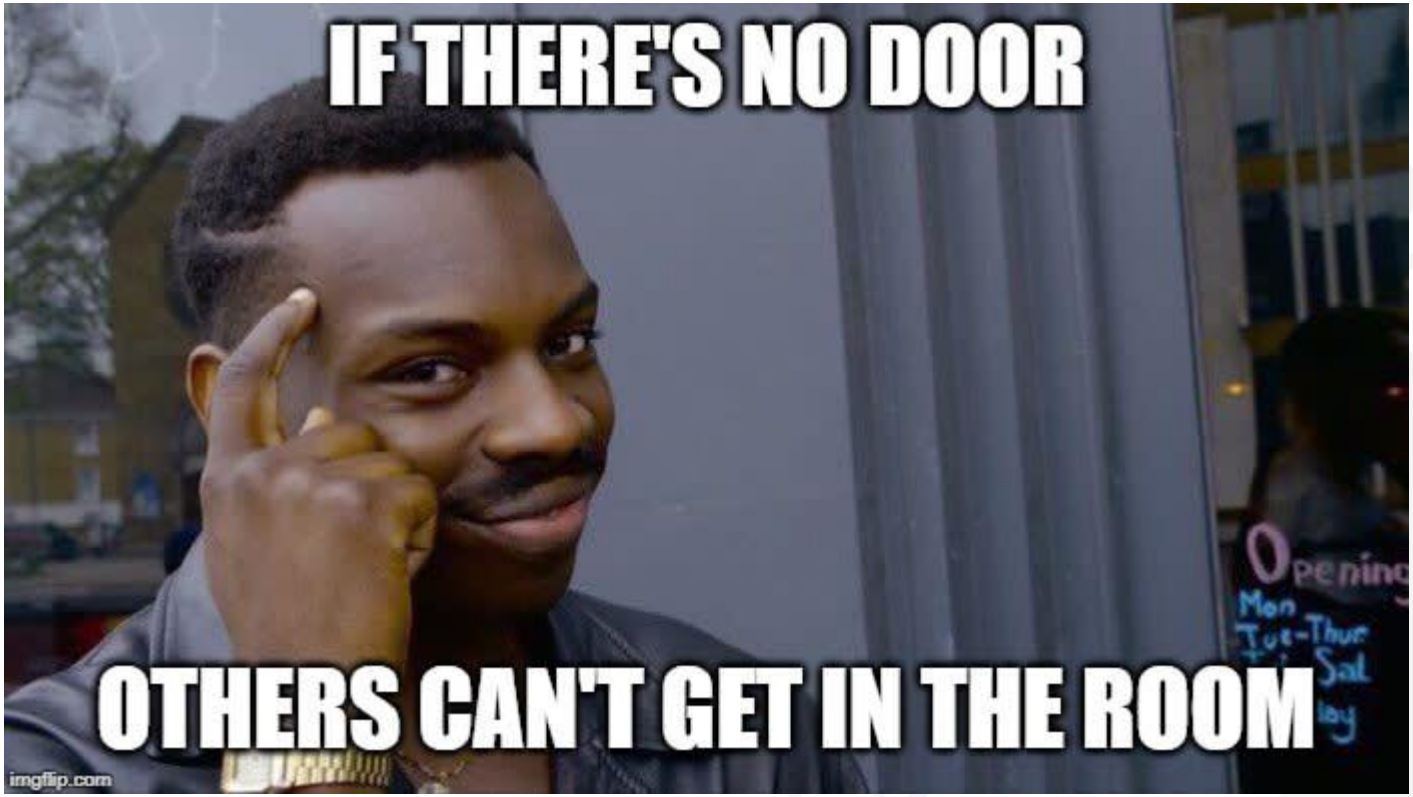
Time	Person A	Person B
3:00	Look in fridge. Run out of milk.	
3:05	Go to the store.	
3:10	Arrive at store.	Look in fridge. Run out of milk.
3:15	Buy milk.	Go to the store.
3:20	Arrive home, put milk away.	Arrive at store.
3:25		Buy milk.
3:30		Arrive home, put milk away.
Not a good cooperation => Too much milk.		

Trạng thái tiến trình (Process state)

- Sơ đồ trạng thái tiến trình



Busy Waiting: Disabling Interrupts



Busy Waiting: Disabling Interrupts

- Interrupt: Tín hiệu được gửi tới CPU từ thiết bị I/O để thông báo rằng hoạt động I/o đã kết thúc
- Trên hệ thống 1 bộ vi xử lý, mỗi tiến trình
  - Vô hiệu hóa tất cả các ngắt ngay sau khi vào critical region → Bộ lập lịch phải đợi cho đến khi tất cả các code trong critical region hoàn thành/ chấm dứt
  - Kích hoạt lại các ngắt
  - → 1 tiến trình có thể kiểm tra và cập nhật bộ nhớ dùng chung mà không sợ các tiến trình khác xen vào

Busy Waiting: Disabling Interrupts

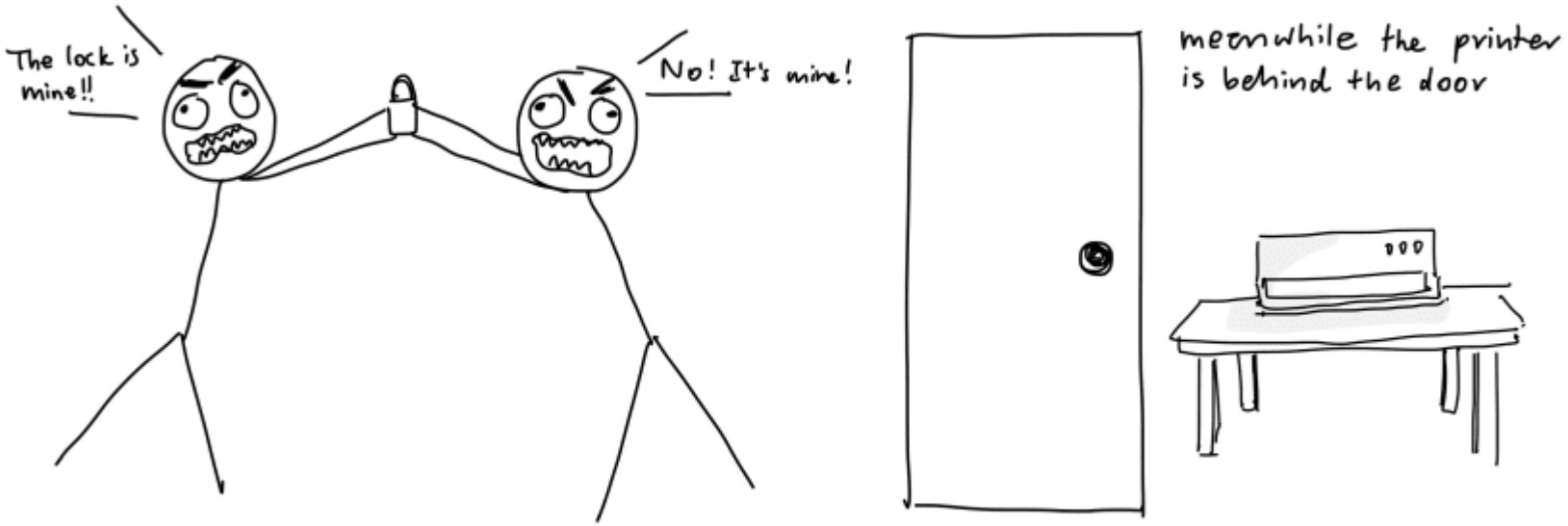
- **Nhược điểm:**
  - Cho phép tiến trình người dùng khả năng tắt ngắt (nếu tiến trình bị hủy trong khi đang thực hiện critical region → hệ thống sẽ bị chặn vô thời hạn)
  - Trên bộ đa xử lý: Việc ngắt vô hiệu hóa chỉ ảnh hưởng đến CPU đã thực thi lệnh vô hiệu hóa trong khi các CPU khác vẫn tiếp tục chạy và có thể truy cập vào bộ nhớ dùng chung

Busy Waiting: Lock Variables

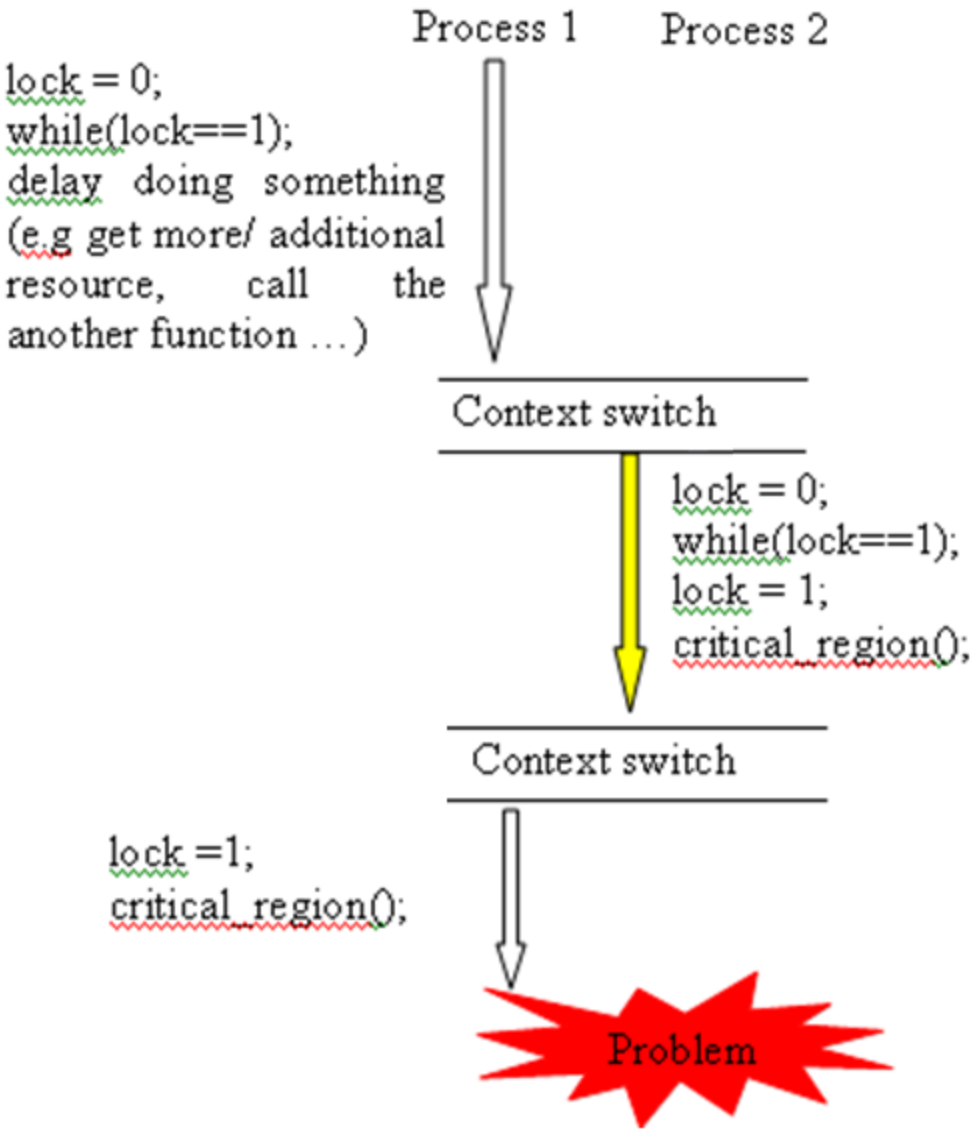
- 1 biến bổ sung lock được sử dụng như 1 cờ (flag) cho phép các luồng/tiến trình vào critical region

```
// Code of a thread
while(TRUE) {
  while (lock == 1);     //waiting until lock is set to 0
  lock = 1;              // set the flag on to enter the CR
  critical_region();     // enter the CR
  lock = 0;    // clear the flag just before going out the CR
  nonCritical_region(); // going out the CR
}
```

Busy Waiting: Lock Variables



Busy Waiting: Lock Variables



- Giải pháp này thỉnh thoảng thất bại: cả 2 tiến trình có thể đồng thời vào critical region của chúng.
  - Tiến trình A đọc lock và thấy nó bằng 0
  - Trước khi gán lock thành 1, tiến trình B được bộ lập lịch chọn và chạy, gán lock thành 1, (sau đó tiến trình A mới gán lock bằng 1)

Busy Waiting : Strict Alternation

Code of the Process 0

```
while (TRUE) {
    while (turn != 0) /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1) /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Code of the Process 1

A proposed solution to the critical region problem

```
// Code of a thread
while(TRUE) {
  while (lock == 1);     //waiting until lock is set to 0
  lock = 1;              // set the flag on to enter the CR
  critical_region();     // enter the CR
  lock = 0;    // clear the flag just before going out the CR
  nonCritical_region(); // going out the CR
}
```

Busy Waiting : Strict Alternation

Busy Waiting : Strict Alternation

- 2 tiến trình luân phiên nhau đi vào critical region của chúng (biến turn như lock)
- Vấn đề:
  - Việc kiểm tra 1 biến cho đến khi 1 giá trị nào đó xuất hiện gọi là busy waiting (lãng phí thời gian của CPU)
  - 1 tiến trình đang bị chặn bởi 1 tiến trình khác không nằm trong critical region của nó
    - 1 tiến trình chậm hơn nhiều so với các tiến trình khác
    - Khi 1 tiến trình dừng, những tiến trình khác sẽ bị chặn vĩnh viễn

Busy Waiting : Peterson's Solution

- Kết hợp ý tưởng dùng turn với ý tưởng lock các biến và sự luân phiên nghiêm ngặt (strict alternation)
- Sử dụng 2 lock: 2 biến turn và interested[i] được sử dụng cho tiến trình i

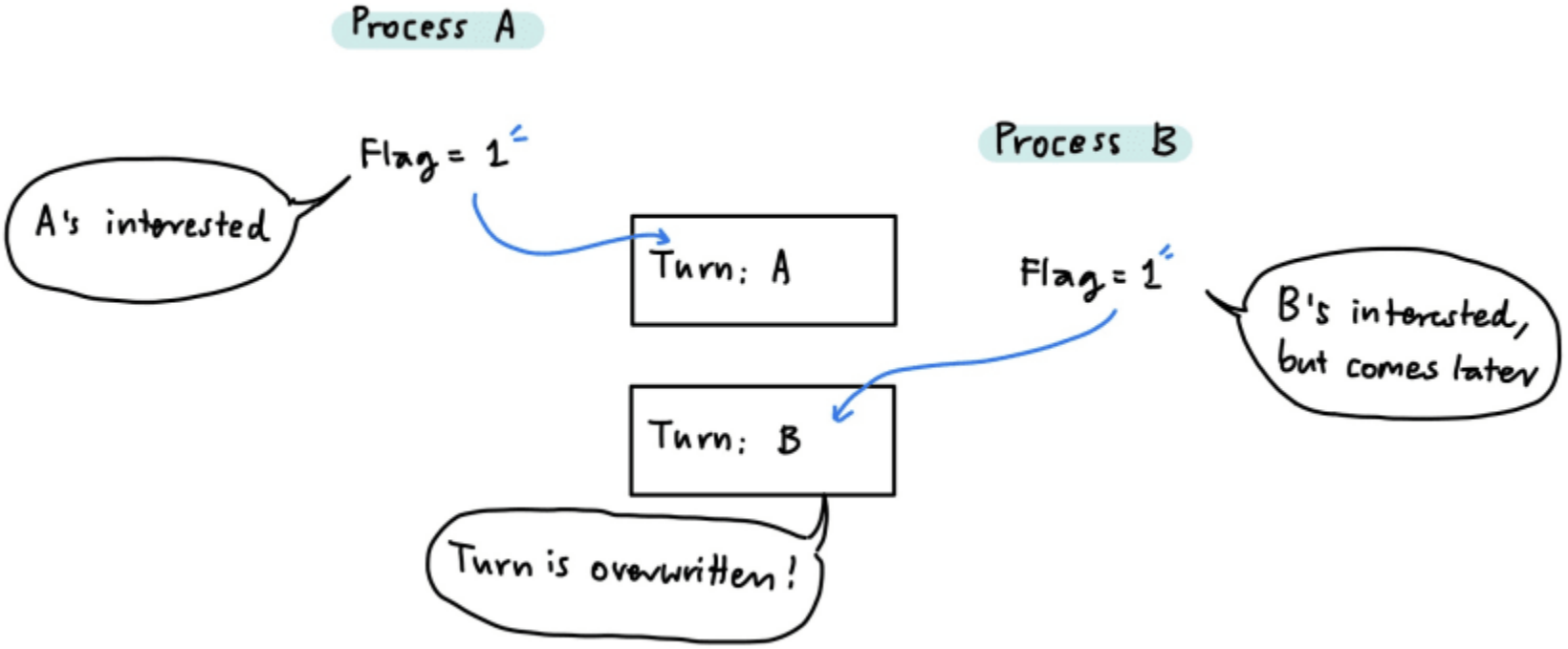
Busy Waiting : Peterson's Solution

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
      int other;                        /* number of the other process */

      other = 1 – process;              /* the opposite of process */
      interested[process] = TRUE;       /* show that you are interested */
      turn = process;                   /* set flag */
      while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
      interested[process] = FALSE;      /* indicate departure from critical region */
}
```

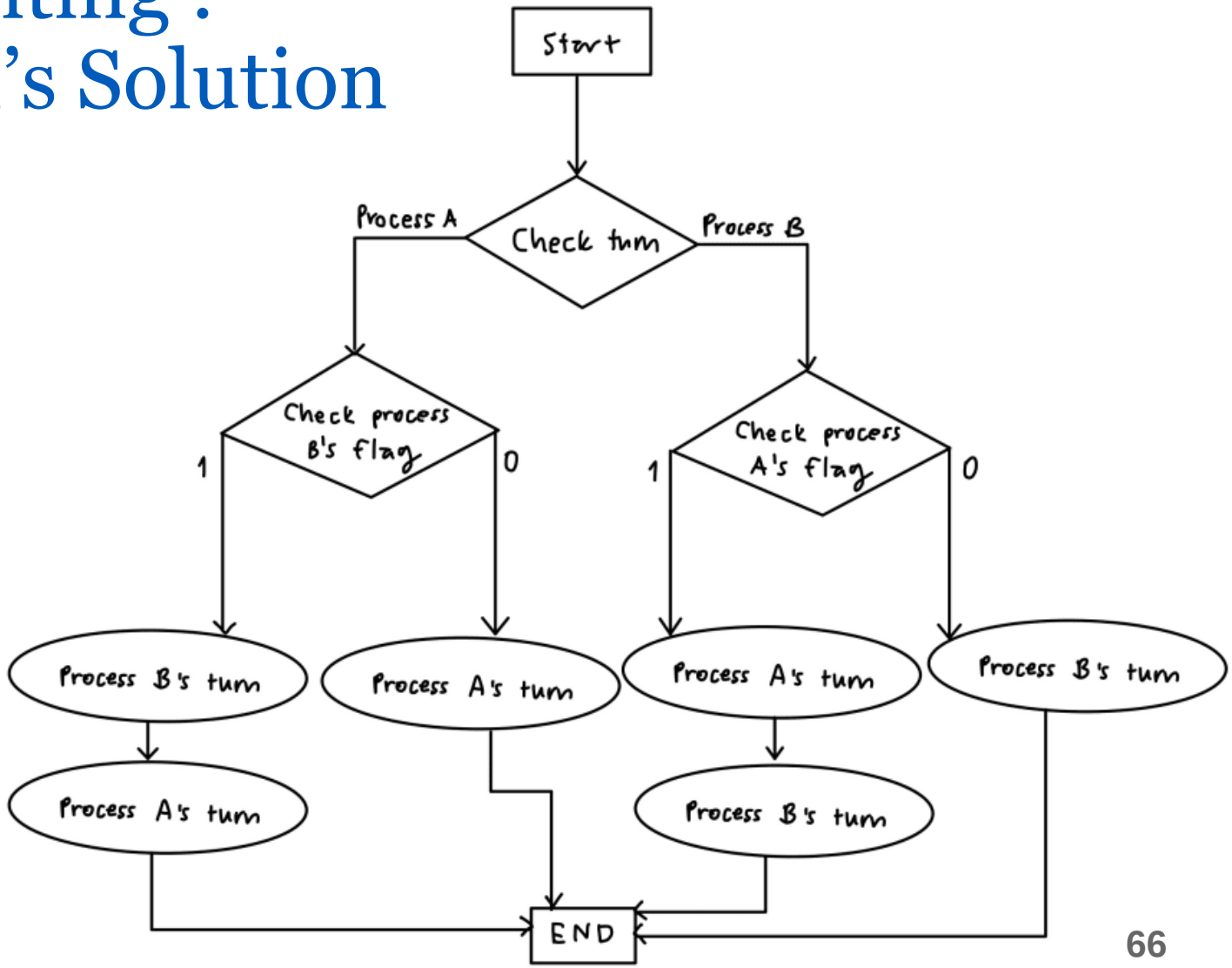
Solution for process P_i

```
do {
     enter_region(i);
     critical_region();
     leave_region(i);
     nonCritical_region();
     }
 while (TRUE);
```

Busy Waiting : Peterson's Solution



Busy Waiting : Peterson's Solution



Busy Waiting : Peterson's Solution

- Loại trừ tương tranh (Mutual exclusion) được bảo tồn
- P_i vào critical region của nó chỉ khi interested[other]=false hoặc turn=other.
- Nếu 2 tiến trình cùng muốn vào critical region cùng 1 lúc, thì interested[i]=interested[other]=true.
- Tuy nhiên, giá trị của turn chỉ có thể là 0 hoặc 1 vì vậy, 1 trong các tiến trình phải thực thi thành công câu lệnh while (để vào critical region của nó) và tiến trình còn lại phải đợi đến khi tiến trình rời khỏi critical region → loại trừ tương tranh được duy trì

Busy Waiting : The TSL Instruction

- TSL: Test and Set Lock, a pre-define assembly instruction → Hard lock
- Instruction form: TSL RIGISTER, LOCK
- Câu lệnh thực hiện 2 bước: đọc biến LOCK từ thanh ghi sau đó tự động lưu giá trị khác 0 (1) tại địa chỉ bộ nhớ của lock.
- Cả 2 bước đều được thực hiện atomically (độc lập – không bộ vi xử lý nào có thể truy cập từ bộ nhớ cho đến khi câu lệnh kết thúc)
- Nếu LOCK = 0, 1 tiến trình sẽ vào critical region. Khi nó hoàn thành, LOCK được gán bằng 0.

enter_region:	leave_region:
TSL REGISTER, LOCK	MOVE LOCK, #0
CMP REGISTER, #0	RET
JNE enter_region	
RET	

Busy Waiting : The TSL Instruction

- The XCHG instruction (on Intel x86 CPU): an alternative to TSL:

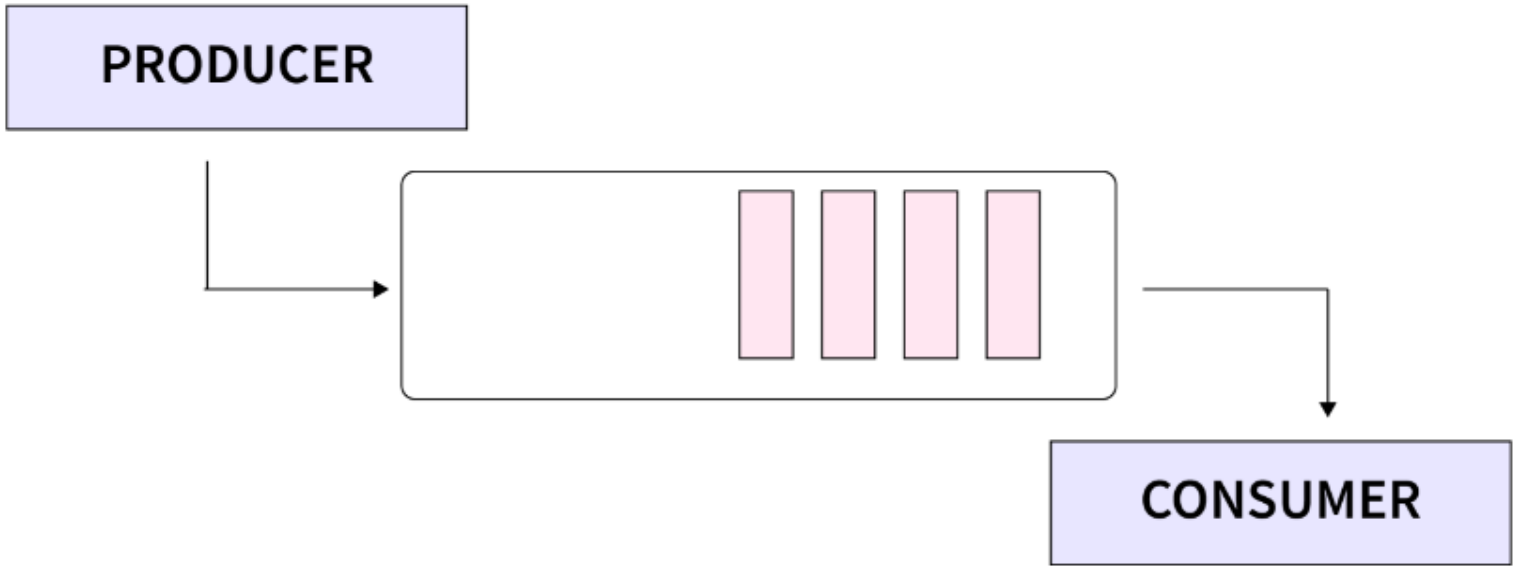
```
Enter_region:
    MOVE  REGISTER, #1
    XCHG REGISTER, LOCK
    CMP REGISTER, $0
    JNE  Enter_region
    RET
```

```
Leave_region:
    MOVE LOCK, #0
    RET
```

Sleep and Wakeup

- Cả Peterson và TSL đều có nhược điểm vì cần busy waiting → Lãng phí tài nguyên CPU, đảo ngược ưu tiên.
- Giải pháp: sử dụng sleep và wakeup để chặn trực tiếp thay vì lãng phí thời gian CPU khi các tiến trình không được vào critical region của chúng.
- Sleep là 1 system call khiến caller bị chặn, nghĩa là tạm dừng cho đến khi tiến trình khác đánh thức nó.
- Wakeup có 1 tham số (tiến trình được đánh thức → ready state).

Bài toán nhà sản xuất nhà tiêu thụ



Sleep & Wakeup: Producer-Consumer Problem

- Vấn đề về bộ đệm giới hạn
  - 2 tiến trình cùng có kích thước bộ đệm cố định.
  - Nhà sản xuất đưa thông tin vào bộ đệm.
  - Nhà tiêu thụ lấy nó ra.
- Điều kiện ngủ
  - Đối với nhà sản xuất, buffer đầy.
  - Đối với nhà tiêu thụ, buffer rỗng.
- Điều kiện thức
  - Đối với nhà sản xuất, buffer còn chỗ trống.
  - Đối với nhà tiêu thụ, có thông tin trong buffer.

Sleep & Wakeup: Producer-Consumer Problem

- Có 2 vấn đề:
  - Không thể kiểm soát việc truy cập đồng thời vào biến count
  - Tín hiệu wakeup có thể bị mất

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                item = produce_item( );         /* generate next item */
                if (count == N) sleep( );       /* if buffer is full, go to sleep */
                insert_item(item);              /* put item in buffer */
                count = count + 1;              /* increment count of items in buffer */
                if (count == 1) wakeup(consumer);   /* was buffer empty? */
        }
}


void consumer(void)
{
        int item;

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

Sleep & Wakeup: Producer-Consumer Problem

- Các hàm thực thi không chính xác tại các câu lệnh count=count+1 của nhà sản xuất cà count=count-1 của nhà tiêu thụ nếu các câu lệnh chạy đồng thời
- Giả sử giá trị của biến counter là 5
- Khi đó giá trị của biến counter có thể là 4, 5 hoặc 6
- 2 câu lệnh có thể được triển khai bằng ngôn ngữ máy như sau:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Sleep & Wakeup: Producer-Consumer Problem

- Vấn đề 1:
  - Không thể kiểm soát việc truy cập đồng thời vào biến count

T_0: producer execute $register_1 = count$ {$register_1 = 5$}
T_1: producer execute $register_1 = register_1 + 1$ {$register_1 = 6$}
T_2: consumer execute $register_2 = count$ {$register_2 = 5$}
T_3: consumer execute $register_2 = register_2 - 1$ {$register_2 = 4$}
T_4: producer execute $count = register_1$ {$count = 6$ }
T_5: consumer execute $count = register_2$ {$count = 4$}

Sleep & Wakeup: Producer-Consumer Problem

- Vấn đề 2:
  - Tín hiệu wakeup có thể bị mất
  - Vấn đề:
    - Bộ đệm rỗng và nhà tiêu thụ đọc count=0 -> nhà tiêu thụ sleep, nhà sản xuất wakeup và thêm data
      - Count=1, nhà tiêu thụ đang sleep, nhà sản xuất gọi nhà tiêu thụ dậy
      - Nhà tiêu thụ thực tế chưa sleep nên tín hiệu đánh thức bị mất

Semaphores

- Semaphore: Bộ đếm không âm
- Biến semaphore có thể có:
  - Giá trị 0: không có wakeup nào được lưu
  - Giá trị dương: 1 hoặc nhiều wakeup đang chờ xử lý
- Semaphore giải quyết được vấn đề mất tín hiệu wakeup

Semaphores

- 2 hành động
  - Down (sleep)
    - Kiểm tra giá trị semaphore có lớn hơn 0, nếu đúng, giảm giá trị và tiếp tục, ngược lại, chặn tiến trình hiện tại
    - Kiểm tra, thay đổi giá trị và có thể chuyển sang chế độ sleep, tất cả đều được thực hiện atomic và đơn lẻ
    - Khi 1 hoạt động semaphore đã bắt đầu, không có tiến trình nào có thể truy cập vào semaphore cho đến khi hoạt động đó hoàn thành hoặc bị chặn
  - Up (wakeup)
    - Tăng giá trị của semaphore và wakeup tiến trình đang ngủ

Semaphores: Solving Producer-Consumer Problem

- Sử dụng 3 semaphore
  - Mutex
  - Empty
  - Full

```
#define N 100                       /* number of slots in the buffer */
typedef int semaphore;              /* semaphores are a special kind of int */
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N;                /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                  /* TRUE is the constant 1 */
        item = produce_item( );     /* generate something to put in buffer */
        down(&empty);               /* decrement empty count */
        down(&mutex);               /* enter critical region */
        insert_item(item);          /* put new item in buffer */
        up(&mutex);                 /* leave critical region */
        up(&full);                  /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                  /* infinite loop */
        down(&full);                /* decrement full count */
        down(&mutex);               /* enter critical region */
        item = remove_item( );      /* take item from buffer */
        up(&mutex);                 /* leave critical region */
        up(&empty);                 /* increment count of empty slots */
        consume_item(item);         /* do something with the item */
    }
}
```

Semaphores: Solving Producer-Consumer Problem

- Có 2 cách sử dụng semaphore
  - Mutual exclusion – binary semaphore
    - Chỉ 1 tiến trình có thể vào critical region (đọc hoặc ghi bộ đệm) tại một thời điểm
  - Synchoronization – Condition checking
    - Đảm bảo nhà sản xuất dừng chạy khi buffer đầy và nhà tiêu thụ ngừng khi buffer rỗng

Semaphores: Solving Producer-Consumer Problem

- Dẫn tới những giải pháp phức tạp, lắt léo, có thể gây ra deadlock
  - 2 down đảo ngược thứ tự trong nhà sản xuất
  - Mutex giảm trước khi rỗng
  - Nếu buffer đầy, mutex = 0 trước khi empty giảm → nhà sản xuất bị chặn
  - Tiếp theo nhà tiêu thục cố chạy, giảm mutex xuống 0 → nhà tiêu thụ cũng bị chặn
  - Cả 2 nhà sản xuất và tiêu thụ bị chặn vĩnh viễn
  - Cẩn thận khi sử dụng semaphore

```
void producer(void){
    int item;
    while(TRUE){
      item=produce_item();
      down(&mutex);
      down(&empty);
      insert_item(item);
      up(&mutex);
      up(&full);
    }
}
```

Mutexes

- Mutex: Binary flag (binary semaphore)
- 1 mutex là một biến có thể ở một trong hai trạng thái
  - Unlock (0): thread tự do vào critical region
  - Lock (1): thread bị chặn cho đến khi thread trong critical region hoàn thành
- 2 thủ tục được sử dụng
  - Mutex_lock: được gọi khi 1 thread cần vào critical region
  - Mutex_unlock: được gọi khi 1 thread trong critical region hoàn thành

```
mutex_lock:
        TSL REGISTER,MUTEX  | copy mutex to register and set mutex to 1
        CMP REGISTER,#0     | was mutex zero?
        JZE ok              | if it was zero, mutex was unlocked, so return
        CALL thread_yield   | mutex is busy; schedule another thread
        JMP mutex_lock      | try again
ok:     RET                 | return to caller; critical region entered
```

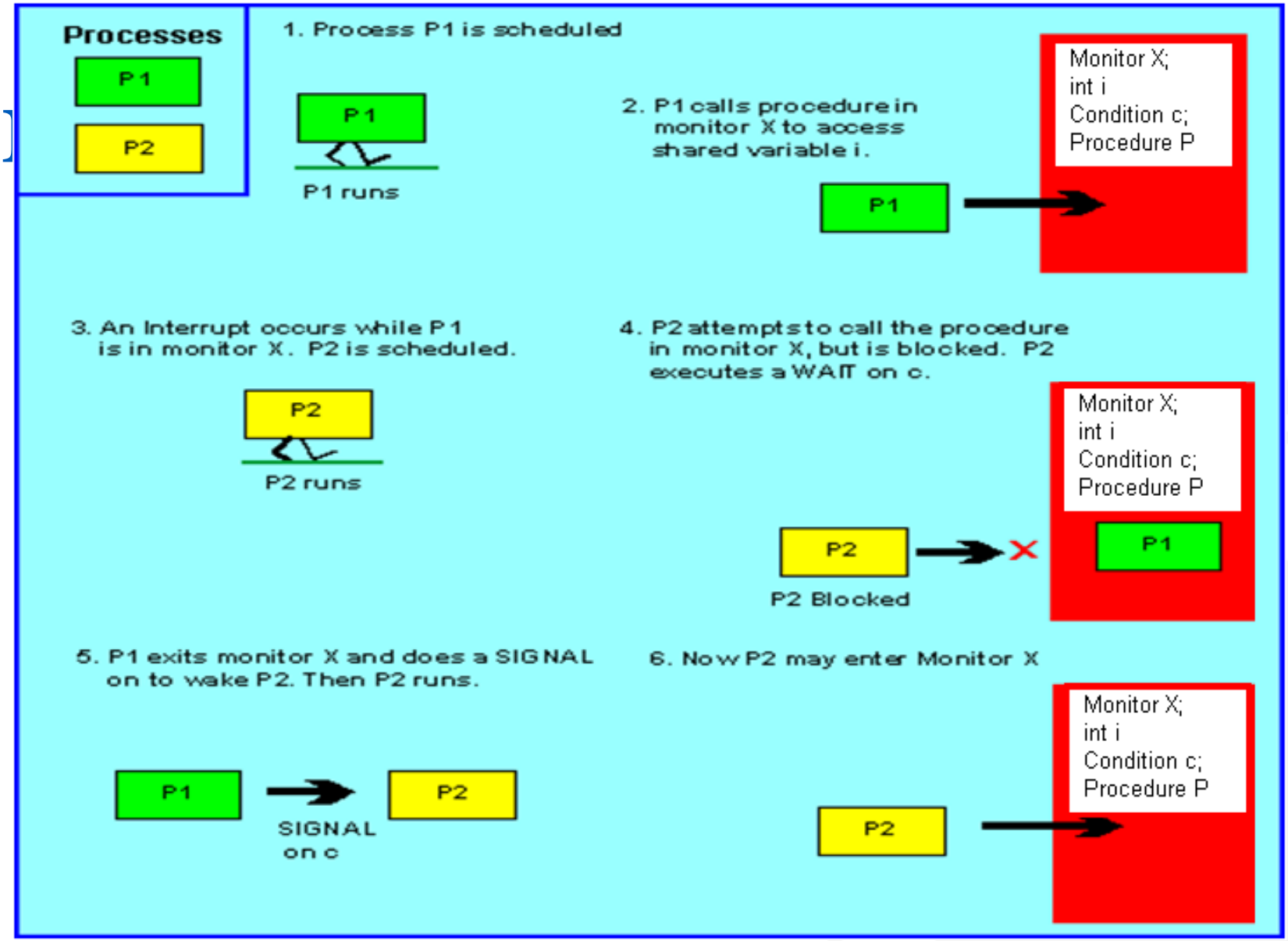
```
mutex_unlock:
        MOVE MUTEX,#0       | store a 0 in mutex
        RET                 | return to caller
```

Monitors

- Là tập hợp các thủ tục, biến và cấu trúc dữ liệu được nhóm lại với nhau trong một loại modun hoặc gói đặc biệt
- Các tiến trình có thể gọi các thủ tục trong monitor bất cứ khi nào chúng muốn, nhưng chúng không thể truy cập trực tiếp vào cấu trúc dữ liệu bên trong của monitor từ các thủ tục được khai báo bên ngoài monitor (đóng gói)
- Chỉ một tiến trình có thể hoạt động trong monitor bất cứ lúc nào (mutex)
- Monitor là cấu trúc ngôn ngữ lập trình
- Khi một tiến trình gọi 1 thủ tục monitor, nó phải kiểm tra xem liệu có thủ tục nào khác hiện đang hoạt động trong monitor hay không. Nếu có, tiến trình gọi sẽ tạm dừng cho đến khi người kia rời đi.
- Monitor được thực hiện bằng cách sử dụng:
  - Mutual exclusion được đảm bảo bởi trình biên dịch
  - Biến điều kiện: cung cấp khả năng chờ đợi

Monitors: Condition Variables

- Sử dụng để đợi một điều kiện cụ thể nào đó đạt được
- 2 hoạt động
  - Wait(): tiến trình hiện tại ngủ, đang đợi (block/ waiting state)
  - Signal(): 1 tiến trình đang ngủ thức dậy
- Khi 1 tiến trình chờ, tiến trình khác có quyền vào monitor
- Sự chờ đợi phải đến trước tín hiệu và theo dõi trạng thái của từng tiến trình bằng các biến để đảm bảo tín hiệu không bị mất


Processes
P1
P2
1. Process P1 is scheduled
P1
P1 runs
2. P1 calls procedure in monitor X to access shared variable i.
P1
Monitor X;
int i
Condition c;
Procedure P
3. An Interrupt occurs while P1 is in monitor X. P2 is scheduled.
P2
P2 runs
4. P2 attempts to call the procedure in monitor X, but is blocked. P2 executes a WAIT on c.
P2
P2 Blocked
Monitor X;
int i
Condition c;
Procedure P
P1
5. P1 exits monitor X and does a SIGNAL on to wake P2. Then P2 runs.
P1
SIGNAL on c
P2
6. Now P2 may enter Monitor X
P2
Monitor X;
int i
Condition c;
Procedure P

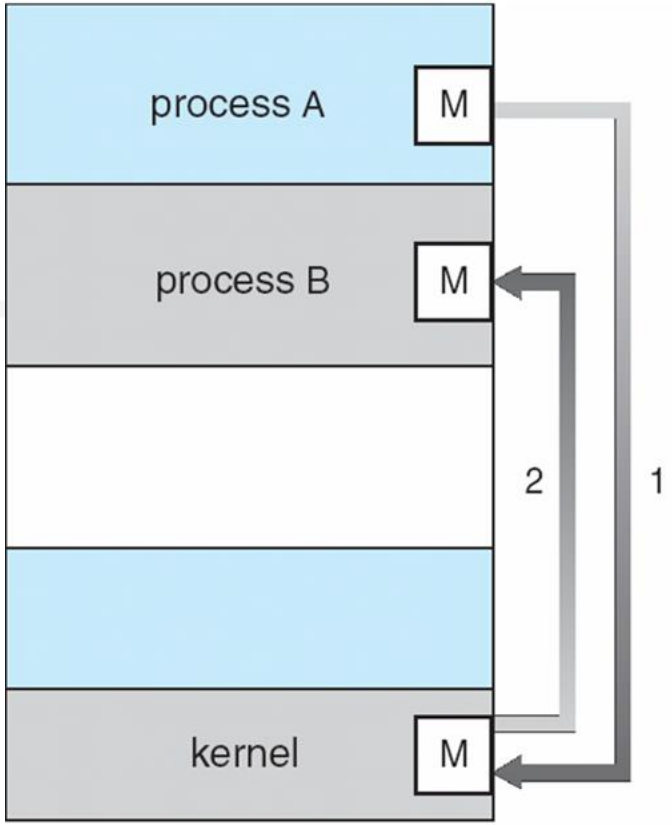
Monitor: Solving Producer-Consumer

```
monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty);
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count – 1;
        if count = N – 1 then signal(full);
    end;

    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item);
    end
end;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item);
    end
end;
```

Message Passing

- Sử dụng system calls (như semaphore) với 2 kiểu nguyên thủy
  - Send(destination, &message): gửi tin nhắn đến nơi nhận
  - Receive(source, &message): nhận tin nhắn từ nguồn. Nếu ko có tin nhắn, ng nhận có thể chặn cho đến khi có tin nhắn đến
- Hệ thống truyền tin nhắn được sử dụng cho giao tiếp các tiến trình trên các máy khác nhau được kết nối bởi mạng
- Để tránh mất tin, người gửi và người nhận có thể đồng ý rằng ngay sau khi nhận được tin nhắn, người nhận sẽ gửi lại 1 tin nhắn xác nhận đặc biệt



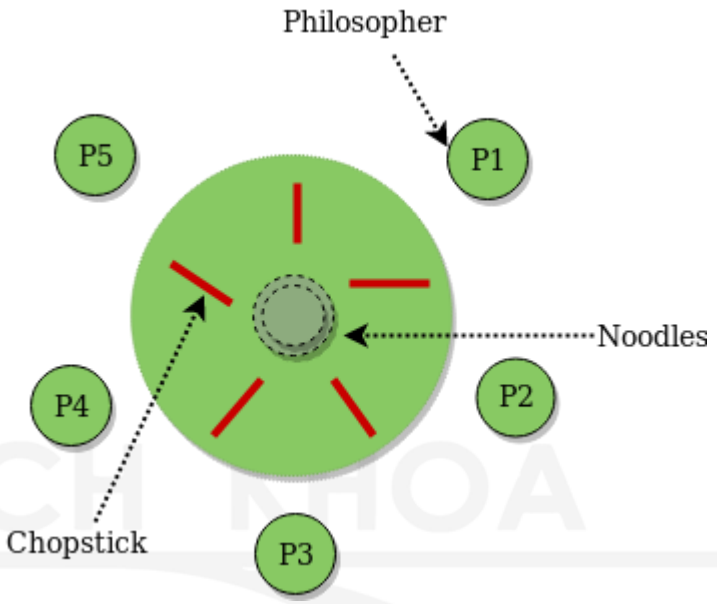
Message Passing: Solving Producer-Consumer Problem

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
        int item;
        message m;                              /* message buffer */

        while (TRUE) {
                item = produce_item( );         /* generate something to put in buffer */
                receive(consumer, &m);          /* wait for an empty to arrive */
                build_message(&m, item);        /* construct a message to send */
                send(consumer, &m);             /* send item to consumer */
        }
}

void consumer(void)
{
        int item, i;
        message m;

        for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
        while (TRUE) {
                receive(producer, &m);          /* get message containing item */
                item = extract_item(&m);        /* extract item from message */
                send(producer, &m);             /* send back empty reply */
                consume_item(item);             /* do something with the item */
        }
}
```

Classical IPC Problems

- 2 bài toán kinh điển trong giao tiếp liên tiến trình:
  - Bài toán Bữa tối của các triết gia
  - Bài toán reader và writer

Bữa tối của các triết gia



- 5 ông triết gia với 5 chiếc đũa
- Triết gia muốn ăn được phải đủ 2 chiếc đũa.
- Triết gia có 2 trạng thái
  - Ăn (không suy nghĩ)
    - Khi ông đói, ông cố lấy 2 chiếc đũa gần ông. Ông sẽ ko thể lấy nếu chiếc đũa đã bị ông khác cầm
  - Suy nghĩ (không ăn)
    - Khi ông ăn xong, ông sẽ đặt cả 2 chiếc đũa xuống lại 2 bên và bắt đầu suy nghĩ

Bữa tối của các triết gia

- Cần phải có sự phân bổ một số tài nguyên giữa các tiến trình
  - Deadlock (tất cả cùng lấy đũa bên trái của mình cùng lúc. Không ai có thể lấy đũa bên phải)
  - Starvation (tất cả cùng lúc bắt đầu ăn cùng lúc, cầm đũa bên trái lên, thấy đũa bên phải không lấy được, đặt đũa bên trái xuống, chờ đợi và lại cầm đũa bên trái lên ... Lặp lại mãi mãi)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
        while (TRUE) {
                think( );                       /* philosopher is thinking */
                take_fork(i);                   /* take left fork */
                take_fork((i+1) % N);           /* take right fork; % is modulo operator */
                eat( );                         /* yum-yum, spaghetti */
                put_fork(i);                    /* put left fork back on the table */
                put_fork((i+1) % N);            /* put right fork back on the table */
        }
}
```

Bữa tối của các triết gia

- Giải pháp 1: A random solution – tại 1 thời điểm, chỉ có 1 ông được ăn → Hiệu suất thấp
  - Triết gia sẽ đợi 1 khoảng thời gian ngẫu nhiên thay vì cùng 1 thời điểm sau khi không lấy được đũa bên tay phải
  - Tuy nhiên không phù hợp với task cần sự chính xác về mặt thời gian
- Giải pháp 2: Áp dụng binary semaphore → hiệu suất cao hơn
  - Trước khi bắt đầu lấy đũa, triết gia sẽ giảm giá trị của mutex
  - Sau khi lấy được thì sẽ tăng giá trị mutex
  - Tuy nhiên sẽ có bug trong thực tế, chỉ 1 ông triết gia có thể ăn thay vì 2 ông
  - Giải pháp: Sử dụng 1 mảng trạng thái và semaphore kết hợp với semaphore nhị phân cho phép song song tối đa cho số lượng triết gia

Bữa tối của các triết gia

```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N–1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      1                   /* philosopher is trying to get forks */
#define EATING      2                   /* philosopher is eating */
typedef int semaphore;                  /* semaphores are a special kind of int */
int state[N];                           /* array to keep track of everyone's state */
semaphore mutex = 1;                    /* mutual exclusion for critical regions */
semaphore s[N];                         /* one semaphore per philosopher */

void philosopher(int i)                 /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                      /* repeat forever */
        think( );                       /* philosopher is thinking */
        take_forks(i);                  /* acquire two forks or block */
        eat( );                         /* yum-yum, spaghetti */
        put_forks(i);                   /* put both forks back on table */
    }
}
```

```
void take_forks(int i)          /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);           /* enter critical region */
        state[i] = HUNGRY;      /* record fact that philosopher i is hungry */
        test(i);                /* try to acquire 2 forks */
        up(&mutex);             /* exit critical region */
        down(&s[i]);            /* block if forks were not acquired */
}
void put_forks(i)               /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);           /* enter critical region */
        state[i] = THINKING;    /* philosopher has finished eating */
        test(LEFT);             /* see if left neighbor can now eat */
        test(RIGHT);            /* see if right neighbor can now eat */
        up(&mutex);             /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING &&
                        state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Bài toán reader và writer

- Mô hình truy cập vào CSDL(file). Reader – tiến trình đọc dữ liệu. Writer – tiến trình sửa đổi dữ liệu
- Nếu 2 reader truy cập vào tài nguyên dùng chung cùng 1 lúc sẽ không có vấn đề gì xảy ra. Cho phép nhiều reader nhưng không cho phép nhiều writer. Chỉ 1 người được phép sửa đổi trên CSDL/file tại 1 thời điểm
- Giải pháp: Bài toán này có nhiều biến thể, tất cả đều liên quan đến độ ưu tiên
  - The first readers-writers problem:
    - Không reader nào phải chờ đợi trừ khi write đã sử dụng dữ liệu
    - Không reader nào phải đợi reader khác đọc xong hoặc reader có độ ưu tiên cao hơn
  - The second readers-writers problem:
    - Một khi writer sẵn sàng, writer sẽ thực hiện việc sửa đổi ngay khi có thể

Bài toán reader và writer

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
        while (TRUE) {                  /* repeat forever */
                down(&mutex);           /* get exclusive access to 'rc' */
                rc = rc + 1;            /* one reader more now */
                if (rc == 1) down(&db); /* if this is the first reader ... */
                up(&mutex);             /* release exclusive access to 'rc' */
                read_data_base( );      /* access the data */
                down(&mutex);           /* get exclusive access to 'rc' */
                rc = rc – 1;            /* one reader fewer now */
                if (rc == 0) up(&db);   /* if this is the last reader ... */
                up(&mutex);             /* release exclusive access to 'rc' */
                use_data_read( );       /* noncritical region */
        }
}
```

Bài toán reader và writer

```
void writer(void)
{
        while (TRUE) {                  /* repeat forever */
                think_up_data( );       /* noncritical region */
                down(&db);              /* get exclusive access */
                write_data_base( );     /* update the data */
                up(&db);                /* release exclusive access */
        }
}
```

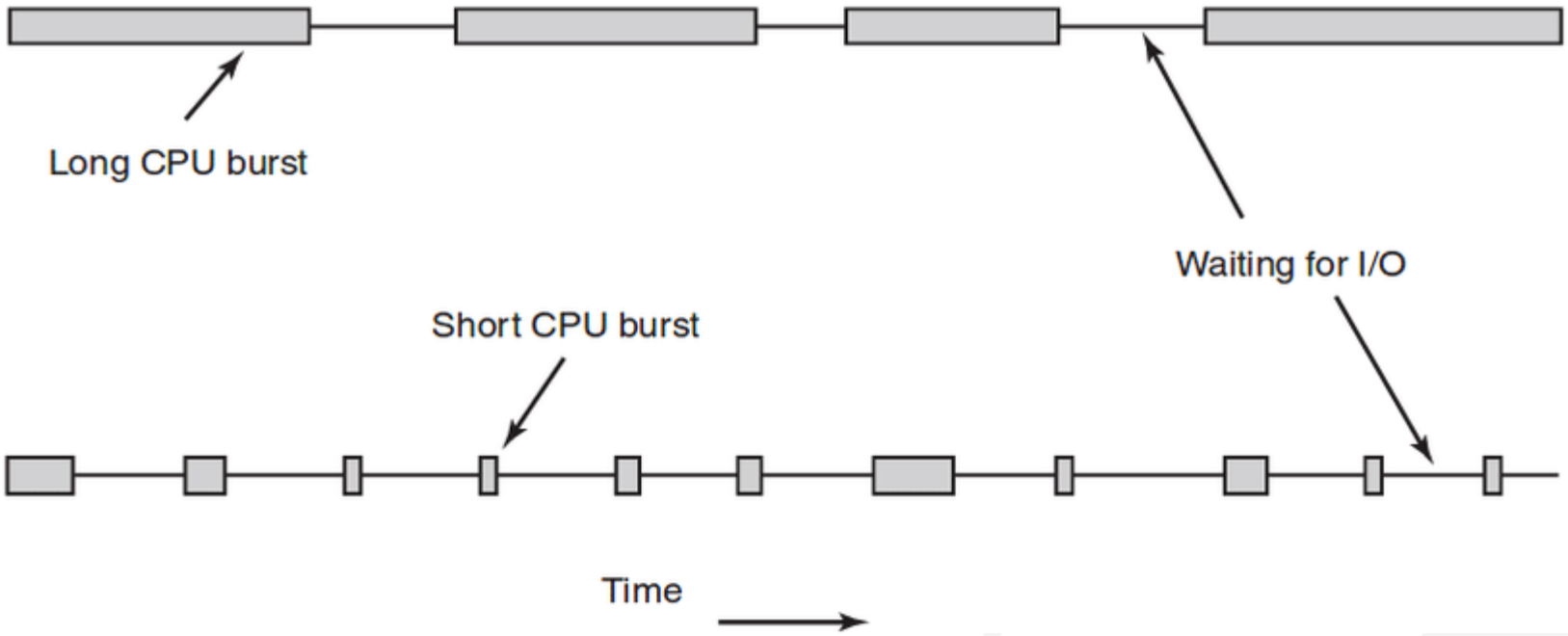
Nội dung

- Tiến trình
- Luồng
- Giao tiếp liên tiến trình
- **Lập lịch**

Giới thiệu lập lịch

- Scheduler
  - Thành phần quyết định tiến trình nào sẽ được chạy và trong bao lâu dựa trên một số thuật toán lập lịch
- Nếu HDH hỗ trợ kernel-level thread thì thread cũng được lập lịch
- Lịch sử
  - Ở máy tính đời đầu, CPU là nguồn tài nguyên khan hiếm → việc lập lịch là điều quan trọng nhất
  - Ngày nay, CPU không còn là nguồn tài nguyên khan hiếm nữa. Hơn nữa, trên PC không có nhiều người dùng cạnh tranh. Tuy nhiên, thuật toàn lập lịch đã trở nên phức tạp hơn.

Process behavior



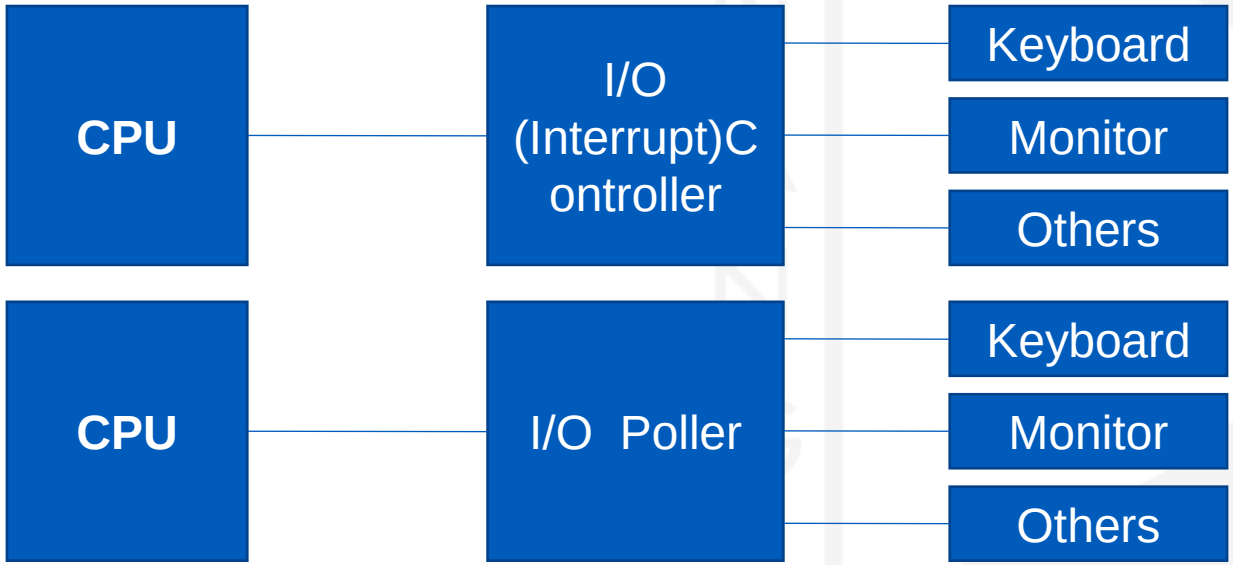
Process behavior

- Tất cả tiến trình thực thi bao gồm 1 chu kỳ gồm các tính toán (CPU thực thi) và yêu cầu I/O (I/O wait).
- Dựa trên hoạt động của CPU, tiến trình phân thành 2 loại:
  - Compute-bound process
    - Dành phần lớn thời gian cho việc tính toán
    - Thời gian sử dụng CPU kéo dài và do đó thời gian chờ I/O ít
  - I/O bound process
    - Dành phần lớn thời gian chờ đợi I/O
    - Có thời gian sử dụng CPU ngắn và do đó chờ I/O thường xuyên

Khi nào lập lịch?

- Tạo tiến trình
  - Cần đưa ra quyết định chạy tiến trình cha hay tiến trình con
- Chấm dứt tiến trình
  - Tiến trình có thể không chạy nữa nên những tiến trình khác phải được chọn từ tập tiến trình sẵn sàng (ready process). Nếu không có tiến trình sẵn sàng, tiến trình nhàn rỗi (idle) do hệ thống cung cấp sẽ chạy
- Chặn tiến trình
  - Khi 1 tiến trình bị chặn, một tiến trình khác phải được chọn để chạy
- Ngắt do thiết bị I/O hoặc phần cứng

- Nội dung: Làm thế nào CPU biết được thiết bị I/O đã hoàn thành công việc của nó.
- Phụ thuộc vào kiến trúc CPU, có 2 cách sau:
  - Sử dụng ngắt (interrupt), 1 tín hiệu được gởi tới CPU từ thiết bị (Intel CPU)
  - CPU quét định kỳ thanh ghi/ bộ đệm của từng thiết bị I/O → Cần xung định kỳ
- Những điều này ảnh hưởng đến thuật toán được chọn để lập lịch



Khi nào lập lịch?

- Sử dụng ngắt
  - Nếu ngắt đến từ thiết bị I/O đã hoàn thành thì một số tiến trình bị chặn chờ I/O hiện có thể sẵn sàng chạy
- Nếu phần cứng ngắt định kỳ
  - Thuật toàn lập lịch ưu tiên (preemptive)/ không ưu tiên (non-preemptive)
  - Các quyết định lập lịch CPU có thể diễn ra khi một tiến trình:
    - Chuyển từ trạng thái running sang waiting (I/O hoặc chờ các tiến trình con)
    - Chuyển từ trạng thái running sang ready (bị interrupt)
    - Chuyển từ trạng thái waiting sang ready (hoàn thành I/O)
    - Chấm dứt

Khi nào lập lịch?

- Thuật toán lập lịch không ưu tiên
  - Chọn 1 tiến trình và để nó chạy cho đến khi bị chặn hoặc tự nguyên giải phóng CPU (không bị ép, không có quyết định lập lịch)
  - Khi 1 tiến trình ở trạng thái running, nó sẽ tiếp tục cho đến khi kết thục hoặc bị chặn do I/O
  - Áp dụng cho batch system
  - VD Shortest Job First (SJF) và Priority (phiên bản nopreemptive), ...

Khi nào lập lịch?

- Thuật toán lập lịch ưu tiên
  - Tiến trình có thể chạy liên tục trong 1 khoảng thời gian tối đa cố định. Nếu nó chuyển sang trạng thái treo (suspended), bộ lập lịch sẽ chọn 1 tiến trình khác để chạy (Cần 1 bộ hẹn giờ)
  - Tiến trình đang chạy có thể bị ngắt và chuyển sang trạng thái ready
  - Cho phép dịch vụ tốt hơn vì bất kỳ tiến trình nào cũng không thể độc quyền sử dụng CPU trong thời gian dài.
  - Áp dụng cho hệ thống chia sẻ thời gian (time-sharing) và thời gian thực (real time)
  - VD Round Robin (RR), Shortest Remainning TimeFirst(SRTF), Priority (phiên bản preemptive), ...

Phân loại thuật toán lập lịch

- Batch system – hệ thống xử lý dữ liệu theo các nhóm riêng biệt của các hoạt động đã được lên lịch trước đó thay vì tương tác hoặc theo thời gian thực
  - Thuật toán không ưu tiên hoặc các thuật toán ưu tiên với khoảng thời gian dài cho mỗi tiến trình. Giảm chuyển đổi tiến trình và tăng hiệu suất
- Interactive system
  - Các thuật toán ưu tiên là cần thiết để ngăn chặn tình trạng ưu tiên giữ cho một tiến trình không chiếm dụng CPU và từ chối dịch vụ đối với các tiến trình khác.
- Real-time system
  - Các thuật toán ưu tiên được sử dụng nhưng thỉnh thoảng không cần thiết vì các tiến trình không chạy trong thời gian dài và thường thực hiện xong công việc và nhanh chóng bị chặn.

Tiêu chí/ Thuộc tính và khái niệm

- Fairness – các tiến trình tương đương có thời gian CPU tương đương nhau
- Policy enforcement (thực thi chính sách) – nếu chính sách là các tiến trình kiểm soát an toàn có thể chạy bất cứ khi nào chúng muốn, ngay cả khi phải trễ 30s thì bộ lập lịch phải đảm bảo chính sách này được thực thi
- Chính sách và cơ chế
  - Chính sách – những gì cần phải làm
  - Cơ chế - cách thức thực hiện chính sách/ thực hiện chính sách như thế nào
  - VD:
    - Timer được xây dựng để đảm bảo bảo vệ CPU (cơ chế)
    - Quyết định về thời gian đặt timer cho một người dùng cụ thể (chính sách)

Tiêu chí/ Thuộc tính và khái niệm

- Trong lập lịch, nếu kernel áp dụng bộ lập lịch ưu tiên với k mức độ ưu tiên
  - Cơ chế
    - Xây dựng 1 danh sách ready process được lập chỉ mục (index) theo mức độ ưu tiên
    - Bộ lập lịch tìm kiếm trên danh sách đó mức độ ưu tiên cao nhất đến thấp nhất → chọn tiến trình đầu tiên
  - Chính sách thiết lập mức độ ưu tiên tương ứng với những người dùng khác nhau

Khái niệm

- Throughput (năng suất) – số lượng tiến trình hoàn thành thực thi mỗi đơn vị thời gian
  - VD đối với tiến trình chạy lâu, tốc độ có thể là một tiến trình mỗi giờ. Đối với tiến trình chạy ngắn hơn, có thể 10 tiến trình mỗi giây.
- Turnaround time (tổng thời gian thực thi) – Lượng thời gian để thực hiện một tiến trình cụ thể
  - Khoảng thời gian chờ đợi trong hàng đợi sẵn sàng + thời gian thực thi trên CPU + thời gian cho I/O + ...
  - Là thời gian từ khi tiến trình được khởi chạy cho đến khi hoàn thành (=thời gian hoàn thành – thời gian đến)

Khái niệm

- CPU utilization: Sử dụng CPU
  - Dao động từ 0 đến 100%. Nên nằm trong khoảng 40% (lightly loaded system) tới 90% (heavily used system)
- Reponse time (Thời gian đáp ứng) – lượng thời gian cần thiết kể từ khi gửi yêu cầu cho đến khi có phản hồi đầu tiên.
  - Trong interactive system, turnaround time có thể không phải là tiêu chí tốt nhất
  - Thông thường, một tiến trình có thể tạo ra một số output sớm và có thể tiếp tục tính toán các kết quả mới
- Proportionality (tính cân đối) – khi một yêu cầu được coi là phức tạp, mất nhiều thời gian, user chấp nhận nhưng một yêu cầu đơn giản mà mất nhiều thời gian sẽ gây khó chịu, bực mình

Scheduling Algorithm Goals

- Tất cả hệ thống
  - Fairness – cung cấp cho mỗi tiến trình phần CPU công bằng
  - Policy enforcement – đảm bảo các chính sách đưa ra được thực hiện
  - Balance – giữ cho tất cả các bộ phận của hệ thống luôn bận (busy)
- Batch system
  - Throughput – tối đa hóa công việc mỗi giờ
  - Turnaround time – giảm thiểu thời gian từ khi khởi chạy đến khi kết thúc
  - CPU utilization – giữ cho CPU luôn bận rộn

Scheduling Algorithm Goals

- Interactive system
  - Response time – phản hồi với yêu cầu một cách nhanh chóng
  - Proportionality (tính cân đối) – nếu có thể, đáp ứng mong đợi của user
- Real-time system
  - Meeting deadlines – tránh mất dữ liệu
  - Predictability – tránh suy giảm chất lượng trong hệ thống đa phương tiện

Lập lịch trong Batch system

- First come first served (FCFS)
- Time-based:
  - Shortest Job First (SJF)
  - Shortest Remaining Time Next (SRT)
- Evaluation
  - Average waiting time
  - Average turnaround time

FCFS

- Tiến trình được dùng CPU theo thứ tự chúng được yêu cầu
- FCFS
  - Thuật toán lập lịch đơn giản nhất
  - Thuật toán lập lịch không ưu tiến
- Cần 1 hàng đợi những tiến trình sẵn sàng
  - Nếu tiến trình vào trạng thái ready, nó sẽ được đưa vào cuối hàng đợi
  - Nếu CPU rảnh, tiến trình ở đầu hàng đợi sẽ được chọn
  - Nếu 1 tiến trình đang chạy bị chặn, nó sẽ bị chuyển xuống cuối danh sách hàng đợi
- Cơ chế time-sliced không được sử dụng

FCFS

- Ưu điểm: Dễ hiểu, dễ triển khai
- Nhược điểm:
  - Tất cả các tiến trình khác phải đợi 1 tiến trình lớn (chạy lâu) → kết quả việc sử dụng CPU hiệu quả thấp
  - Tình trạng trên dược gọi là convoy effect (hiệu ứng hộ tống)

FCFS Algorithm Evaluation

- Ex: 3 processes are ready in order and their process time: P1 (24 ms), P2(3 ms), P3 (3 ms)

Process	Burst time	Waiting time	Turnaround time
P1	24	0	24
P2	3	24	27
P3	3	27	30
Sum	30	51	81

Average waiting time = 51/3= 17
Average Turnaround time: = 81/3 = 27

Do yourself: 3 processes:
(Process:BurstTime) in order (P2:3), (P3:3), (P1:24)

FCFS Algorithm Evaluation

- Ex: 3 processes are ready in order and their process time: P1 (24 ms), P2(3 ms), P3 (3 ms)

Process	Burst time	Waiting time	Turnaround time
P2	3	0	3
P3	3	3	6
P1	24	6	30
Sum	30	9	39

Average waiting time = 9/3= 3
Average Turnaround time: = 39/3 = 13

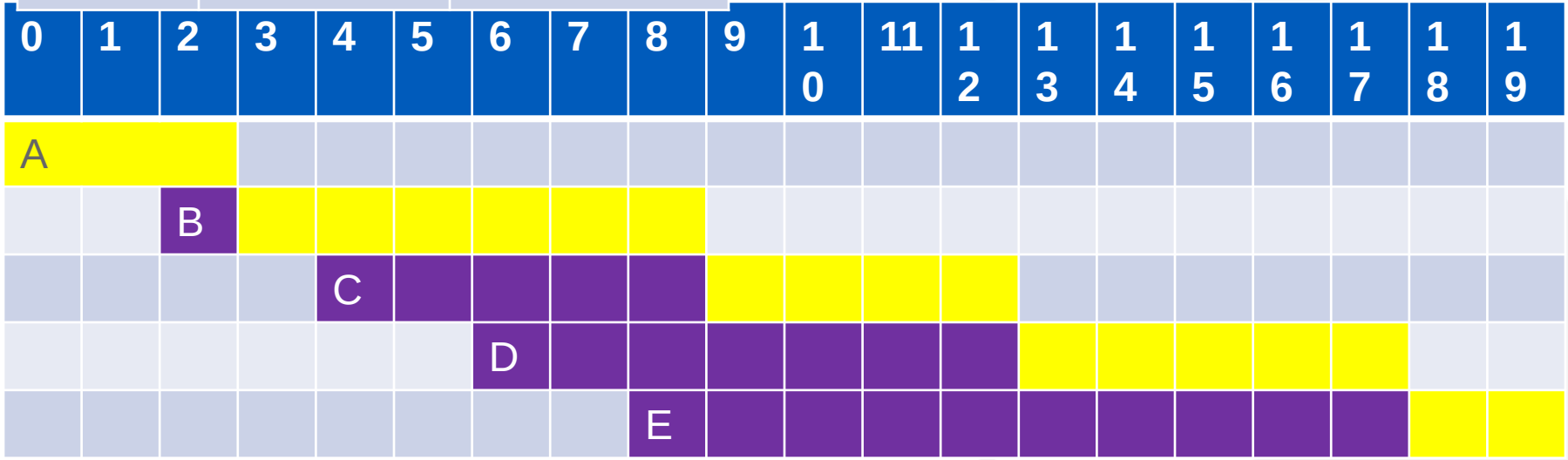
Do yourself: 3 processes:
(Process:BurstTime) in order (P2:3), (P3:3), (P1:24)

FCFS Evaluataion: Do yourself

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Avg waiting time = ?

Avg turnaround time = ?



Turn-around time = time of complete – arrival time

The Shortest Job First (SJF) Algorithm

- Thời gian chạy được biết trước (không ưu tiên)
- Khi một số công việc như nhau đang ở hàng đợi chờ bắt đầu, bộ lập lịch sẽ chọn công việc có thời gian ngắn nhất trước tiên
  - Thuật ngữ thích hợp hơn – shortest next CPU burst scheduling algorithm
  - Khi CPU khả dụng, tiến trình có thời gian chạy nhỏ nhất sẽ được chọn
  - → Sắp xếp các tiến trình dựa trên thời gian chạy tăng dần

SJF Algorithm Evaluation

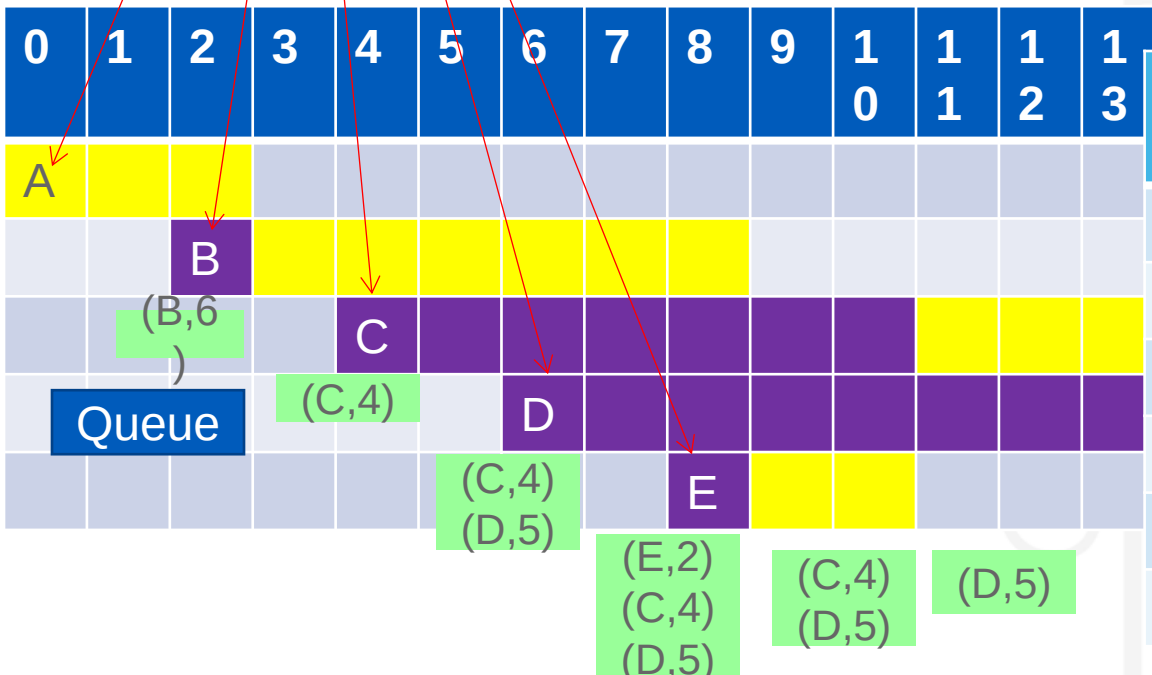
- Ex: 4 simultanous incoming processes
  - (Process:BurstTime) (P1:6), (P2:8), (P3:7), (P4: 3)
  - Sorted list: P4(3), P1(6), P3(7), P2(8)

Process	Burst time	Waiting time	Turnaround time
P4	3	0	3
P1	6	3	9
P3	7	9	16
P2	8	16	24
Sum	24	28	52
Avg		**28/4= 7**	**52/4= 13**

- **Do yourself:** 6 Simultaneous incoming processes
  - (Process:BurstTime) (P1:5), (P2:8), (P3:6), (P4: 3), (P5:26), (P6: 6)

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

P	BT	WT	TT
A	3	0	3-0=3
B	6	3-2 =1	9-2=7
E	2	9-8=1	11-8=3
C	4	11-4=7	15-4=11
D	5	15-6=9	20-6=14
Sum	20	18	38
AVG		**3.6**	**7.6**



Process	Arrival Time	Service Time
A	0	5
B	1	3
C	2	2
D	3	1
E	4	7
F	5	4

SJF Algorithm Evaluation

Do yourself: 6 incoming processes

- Average waiting time= ?
- Average turnaround time = ?

Process	Arrival Time	Service Time
A	0	5
B	1	3
C	2	2
D	3	1
E	4	7
F	5	4

The SRT Algorithm

- Shortest Remainning Next
- Là phiên bản ưu tiên của SJF
- Bộ lập lịch luôn chọn tiến trình có thời gian chạy còn lại ngắn nhất
- Ưu tiên tiến trình đang thực thi, nếu thời gian chạy của tiến trình mới đến ngắn hơn phần còn lại của tiến trình hiện đang thực thi
- Khi một tiến trình mới đến, tổng thời gian của nó được so sánh với thời gian còn lại của tiến trình hiện tại
- Nếu tiến trình mới cần ít thời gian hơn để hoàn thành thì tiến trình hiện tại sẽ tạm dừng, và tiến trình mới bắt đầu

SRT Algorithm Evaluati

- Ex: (Process:ArrivalTime:BurstTime):

  (P1:0:9), (P2:2:4), (P3:4:1), (P4:5:4)

Process	Time	Execute	Process:remainin
P1:9 →	0	P1	P1:9
	1	P1	P1:8
P2:4 →	2	P2	P2:4, P1:7
	3	P2	P2:3, P1:7
P3:1 →	4	P3	P3:1, P2:2, P1:7
P4:4 →	5	P2	P2:2, P4:4, P1:7
	6	P2	P2:1, P4:4, P1:7
	7..10	P4	P4:4→0, P1:7
	11.. 17	P1	P1:7 → 0
	18		finished

P	BT	WT	TT
P1	9	0+(11-2)=9	18
P2	4	0+1=1	5
P3	1	0	1
P4	4	2	6
Sum	18	12	30
Avg		**12/4=3**	**30/4=7.5**

Process	Arrival Time	Service Time
A	0	5
B	1	3
C	2	2
D	3	1
E	4	7
F	5	4

SRT Algorithm Evaluation

Do yourself: 6 incoming processes

- Average waiting time= ?
- Average turnaround time = ?

Process	Arrival Time	Service Time
A	0	5
B	1	3
C	2	2
D	3	1
E	4	7
F	5	4

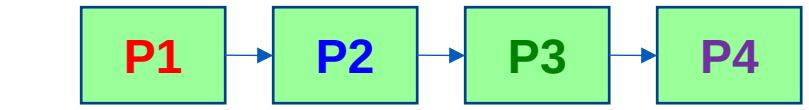
Lập trình trong interactive system

- Interactive program: 1 chương trình trao đổi output và input với user, thường là user xem mà hình và sử dụng input devices như bàn phím, chuột để cung cấp phản hồi cho chương trình
- Interactive system: HDH hỗ trợ các tiến trình tương tác.
- Mô hình cho tiến trình tương tác:
  - Đợi lệnh, thực thi lệnh, ...
  - Thời gian thực thi lệnh có thể được ước tính
- Thuật toán:
  - RR scheduling, Priority scheduling, Multiple queues scheduling, Shortest Process Next, Guaranteed Scheduling, Lottery scheduling, Fair-Share Scheduling

RR

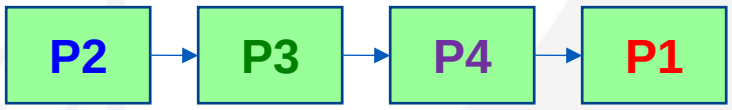
- Cũ nhất, đơn giản nhất, công bằng nhất, sử dụng rộng rãi nhất
- Giả định: Tất cả các tiến trình đều quan trọng như nhau
- Mỗi tiến trình được ấn định một khoảng thời gian bằng nhau (quantum/ time-slice)
- Khi tiến trình sử dụng hết quantum hoặc bị chặn, nó sẽ được chuyển đến cuối hàng đợi và CPU được gán cho tiến trình ở đầu hàng đợi → Hàng đợi được coi là hàng đợi vòng tròn
- Tương tự như FCFS

Quantum=3



Current process

t=0



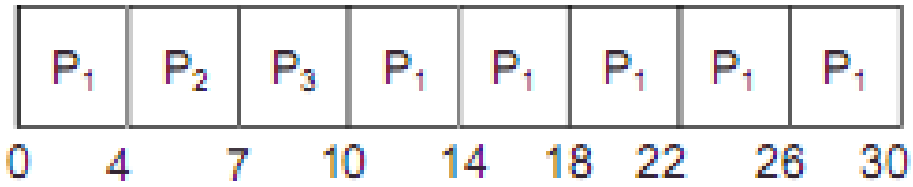
Current process

t=3

RR

- Độ dài của quantum
  - Quá ngắn → giảm hiệu suất CPU và dẫn tới quá nhiều tiến trình chuyển đổi. VD quantum=4ms, switching overhead=1ms
    - CPU waste = 1/(1+4)=20%
  - Quá dài → phản hồi chậm cho những yêu cầu phản hồi ngắn (chung nhược điểm với FCFS). VD quantum=99ms, switching overhead=1ms
    - CPU waste=1%
  - 20-50msec là khoảng thời gian hợp lý

RR



- Ex: With quantum=4, switching overhead=0,
- 3 processes come:
- (Process:BurstTime) (P1:24), (P2:3), (P3:3)
  - Average waiting time = (6 + 4 + 7)/3 = 5.7
  - Average turnaround time = (30 + 7 + 10)/ 3 = 15.6
- Ex: With quantum=4, switching overhead=1
  - Average waiting time = ((13-4) + (5-0) + (9-0))/3 = 23/3 = 7.8
  - Average turnaround time = ((33-0) + (8-0) + (12-0))/ 3 = 53/3=17.8

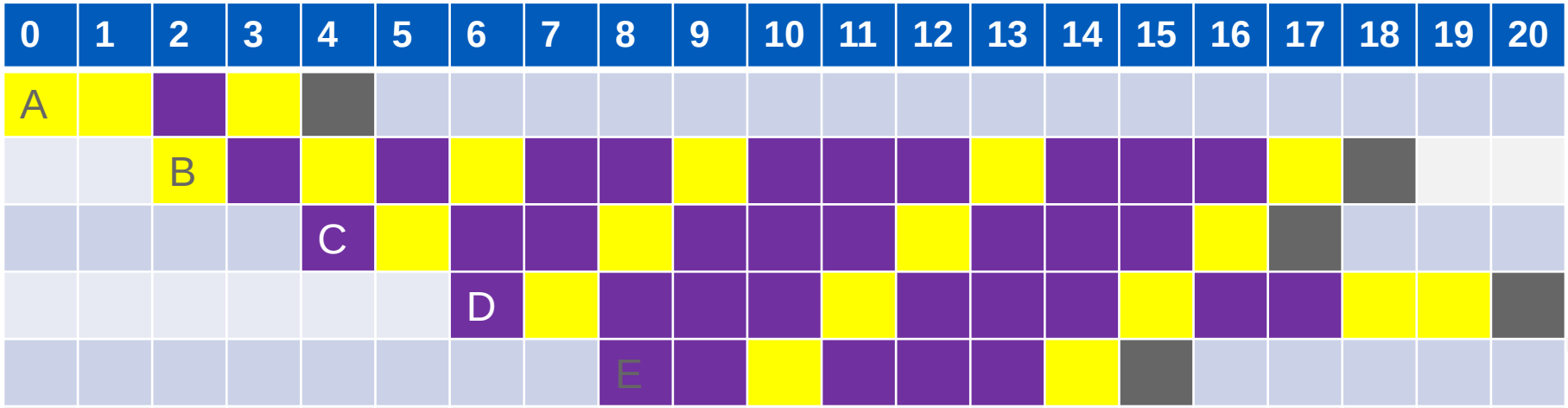
(P1:2), (P2:5), (P3:7), (P4:9), (P5:27), (P6:8)

0	4	5	8	9	12	13	33
P1	Switch	P2	Switch	P3	Switch	P1	Finished

RR

- Do yourself:
- With quantum=5, switching overhead=0, 6 processes come with burst time of each:
- (P1:2), (P2:5), (P3:7), (P4:9), (P5:27), (P6:8)
  - Average waiting time =?
  - Average turnaround time = ?
- The same problem but switching overhead = 1.

RR



Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Quantum=1

Avg waiting time = 6.8

Avg turnaround time = 10.8

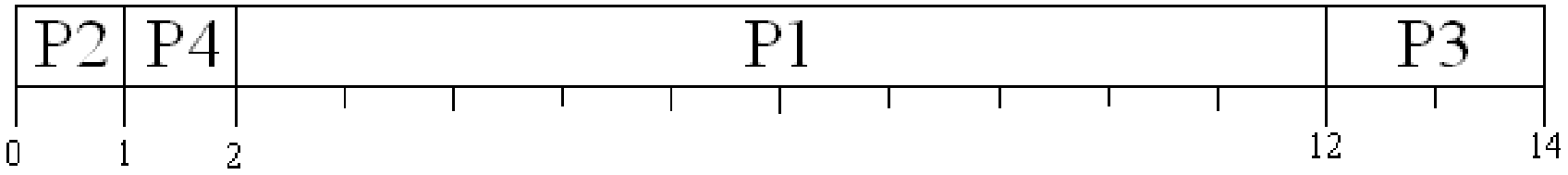
Check these results.

Priority Scheduling

- Mỗi tiến trình có một mức độ ưu tiên – yếu tố ngoài
- Tiến trình có thể chạy có độ ưu tiên cao nhất (giá trị tối thiểu) luôn được chạy
- Tạo cơ hội cho các tiến trình khác → thay đổi mức độ ưu tiên của tiến trình đang chạy hoặc gán nó 1 lượng quantum
- Phân công ưu tiên:
  - Statically – được xác định trước cho mỗi tiến trình.
  - Dynamically – được hệ thống chỉ định để đạt được các mục tiêu nhất định. VD tăng độ ưu tiên cho các tiến trình I/O-bound process

Priority Scheduling

- Các lớp ưu tiên (một số tiến trình được gán cùng mức độ ưu tiên)
  - Sử dụng xếp lịch ưu tiên giữa các lớp (4 lớp 1-4). Số càng nhỏ thì mức độ ưu tiên càng cao
  - Sử dụng một thuật toán lập lịch khác nhau cho mỗi lớp
- VD: (Process:BurstTime:Priority)
  - (P1:10:3), (P2:1:1), (P3:2:4), (P4:1:2) → P2, P4, P1, P3
  - Average waiting time: (2 + 0 + 12 + 1)/ 4 = 3.75
  - Average turnaround time: (12 + 1 + 14 + 2)/ 4 = 7.25



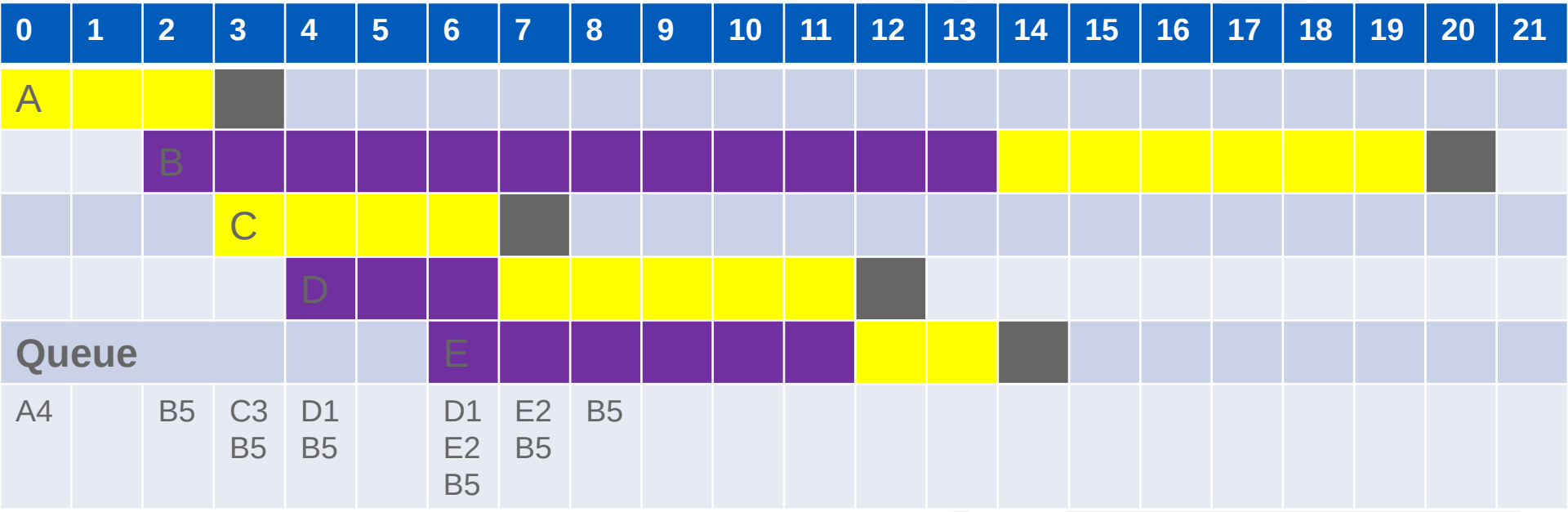
Priority Scheduling: Example

Process	Arrival time	Service time	Priority
A	0	3	4
B	2	6	5
C	3	4	3
D	4	5	1
E	6	2	2

Avg waiting time = 4.2

Avg turnaround time = 8.2

Check results!



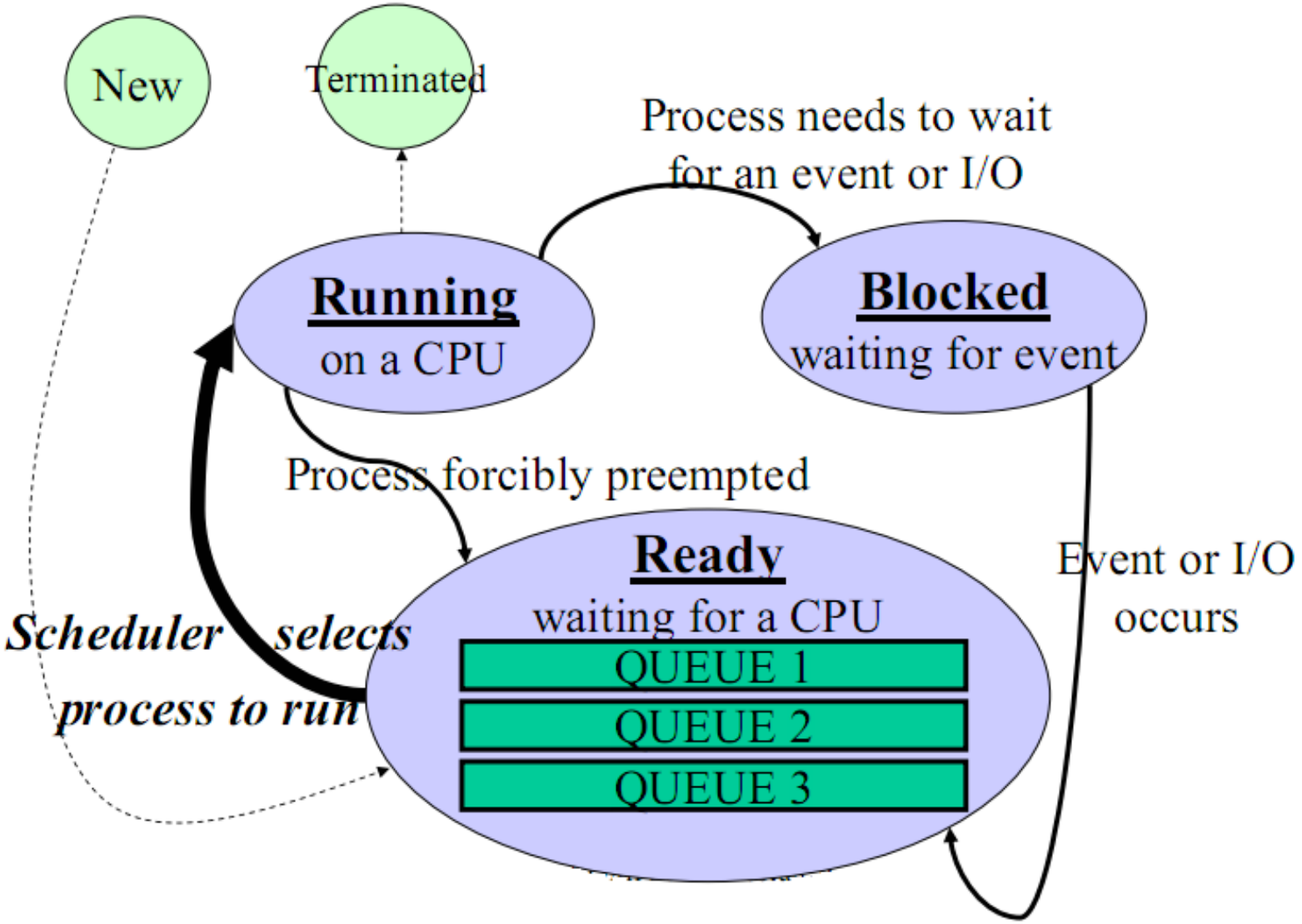
Priority Scheduling

- Vấn đề: Starvation
  - Các tiến trình có mức ưu tiên thấp có thể phải chờ vô thời gian (không thể được thực thi nếu hệ thống xảy ra lỗi trong thời gian chạy) đối với CPU bởi các tiến trình có mức ưu tiên cao hơn
- Giải pháp: Aging
  - Kỹ thuật tăng dần mức độ ưu tiên của các tiến trình chờ trong hệ thống trong thời gian dài (sử dụng ngắt đồng hồ- clock interrupt)
  - VD cứ sau 15', giảm mức độ ưu tiên của quy trình chờ từ (1→127) nghĩa là mức độ ưu tiên với 127 → 1 ít nhất là 32 giờ

Multiple Queues

- Một cách đơn giản để ánh xạ mức độ ưu tiên vào các quyết định lập lịch thực tế là cấp cho mỗi tiến trình một time slice liên quan đến mức độ ưu tiên của nó (VD time slice nhiều hơn cho các tiến trình có độ ưu tiên cao hơn)
- Một cách tiếp cận thuận tiện hơn là xem trạng thái sẵn sàng không chỉ là 1 hàng đợi các tiến trình mà còn là nhiều hàng đợi, mỗi hàng đợi có mức độ ưu tiên riêng
- Tùy chọn:
  - Các tiến trình của hàng đợi có mức độ ưu tiên cao hơn có thể phải hoàn thành trước khi các tiến trình của hàng đợi có mức độ ưu tiên thấp hơn bắt đầu chạy
  - Hàng đợi có mức độ ưu tiên cao hơn có thể nhận được nhiều thời gian hơn
  - Các tiến trình có thể di chuyển giữa các hàng đợi (ưu tiên được điều chỉnh động)

Multiple Queues



Guaranted Scheduling

- Nếu hệ thống 1 người dùng có n tiến trình đang chạy thì mỗi tiến trình sẽ nhận được $1/n$ chu kỳ CPU (công bằng)
- Cơ chế
  - Ban đầu, mỗi tiến trình được phân bổ $1/n$ thời gian CPU
  - Hệ thống phải theo dõi lượng thời gian CPU mà mỗi tiến trình đã có kể từ khi tạo
  - Sau đó hệ thống tính toán lượng thời gian mà mỗi CPU được hưởng, cụ thể là thời gian kể từ khi tạo chia cho n → $t_{real}/t_{entitled}$, tỷ lệ biểu thị hiệu suất khi một tiến trình được phân bổ thời gian CPU.
  - Sau đó thuật toán sẽ chạy tiến trình có tỷ lệ thấp nhất (tiến trình sử dụng CPU với thời gian ngắn nhất) cho đến khi tỷ lệ của nó vượt lên trên đối thủ cạnh tranh gần nhất)

Lottery Scheduling

- Cho các tiến trình vé số
- Các tiến trình quan trọng có thể được tặng thêm vé (để tăng tỷ lệ thắng)
- Bất cứ khi nào phải đưa ra quyết định lập lịch, một vé số sẽ được chọn ngẫu nhiên và tiến trình giữ vé đó sẽ nhận được tài nguyên
- Lottery scheduling có tính phản hồi cao
- Lottery scheduling có thể được sử dụng để giải quyết các vấn đề khó giải quyết bằng các phương pháp khác

Fair-Share Scheduling

- Sự công bằng dựa trên số lượng người dùng
- Tính huống: Mỗi tiến trình được lập lịch riêng, bất kể chủ sở hữu là a: nếu người dùng 1 khởi chạy 9 tiến trình và người dùng 2 khởi chạy 1 tiến trình, với RR hoặc độ ưu tiên bằng nhau, user 1 sẽ chiếm 90% CPU và user 2 10%
- Để ngăn chặn tình trạng này, một số hệ thống sẽ tính đến người sở hữu tiến trình trước khi lập lịch cho tiến trình. Trong mô hình này, mỗi người dùng được phân bổ 1 phần CPU và bộ lập lịch sẽ chọn các tiến trình. Mỗi người sẽ nhận được 50% CPU bất kể họ chạy bao nhiêu tiến trình

Lập trình trên real-time system

- Bối cảnh:
  - Hệ thống thời gian thực là hệ thống trong đó thời gian đóng vai trò thiết yếu
  - Một hệ thống thời gian thực có 2 loại là hard real time và soft real time
  - Một hệ thống thời gian thực chia chương trình thành nhiều tiến trình có hành vi có thể dự đoán và biết trước
  - Các tiến trình thường tồn tại trong thời gian ngắn và có thể hoàn thành trong chưa đầy 1s
- Bộ lập lịch lên lịch cho các tiến trình sao cho đáp ứng được tất cả deadline

Review
- Hard real-time system: System in which deadlines must be met.
- Soft real-time system: System in which deadlines may not be met.

Lập trình trên real-time system

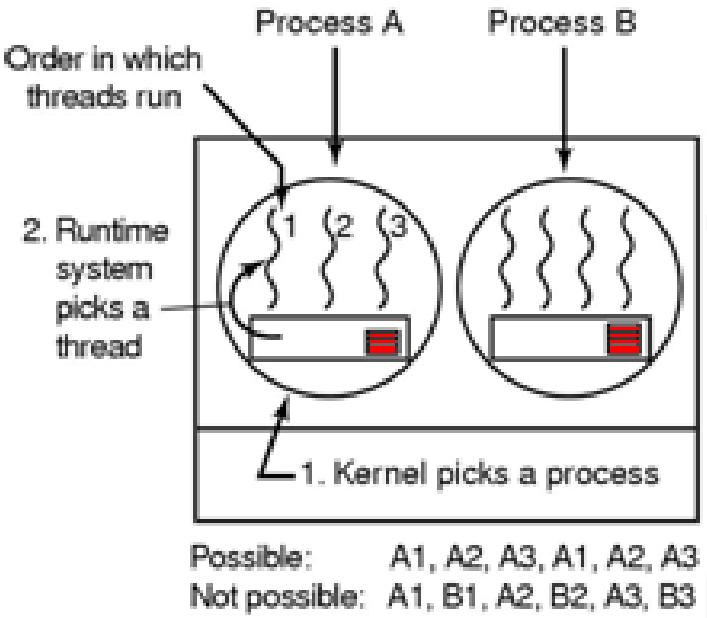
- Static scheduling (applied to hard real-time)
  - Đưa ra quyết định lập kế hoạch trước khi hệ thống bắt đầu chạy
  - Chỉ hoạt động khi có sẵn thông tin hoàn hảo về công việc phải làm và thời hạn phải đáp ứng
- Dynamic scheduling (applied to soft real-time)
  - Đưa ra quyết định lập lịch trong thời gian chạy
  - Không có hạn chế tĩnh

Thread Scheduling

- Bộ lập lịch ở kernal lập lịch tiến trình
- Bộ lập lịch ở kernel có thể hỗ trợ lập lịch thread
- Bộ lập lịch thread ở mỗi tiến trình quyết định thread nào sẽ chạy
- Các thuật toán lập lịch thread giống với lập lịch tiến trình

User-Level Thread Scheduling

- Nếu bất kỳ thread nào có thời gian chạy dài, thread sẽ tiêu tốn toàn bộ thời gian của tiến trình (cho đến khi nào kết thúc) vì user mode không được hỗ trợ clock interrupt
- Mặt khác, mỗi thread chạy một lúc, sau đó đưa CPU trở lại bộ lập lịch thread trước khi kernel phân bổ quantum cho tiến trình khác
- RR và priority scheduling thường được áp dụng.
- Ứng dụng triển khai bộ lập lịch luồng vì chỉ nó biết tất cả luồng



Kernel-Level Thread Scheduling

- Bộ lập lịch tại kernel lập lịch luồng → kernel yêu cầu full text switch (đổi memory map, vô hiệu hóa cache...), đặc biệt là việc chuyển đổi thread của tiến trình này sang thread của tiến trình khác tốn nhiều chi phí.
- Kernel không biết từng thread làm gì
- → User-level thread có hiệu suất cao hơn kernel-level thread trong lập lịch