Chương 9: Ngôn ngữ lập trình song song

Giảng viên: Ph.D Nguyễn Văn Hòa

Khoa KT-CN-MT – ĐH An Giang

Nội dung

- Giới thiệu
- SubProprogam-level
- Semaphores
- Chương trình giám sát (monitor)
- Truyền thông điệp (massage passing)
- Luồng (Java thread)

Giới thiệu

- Sự tương tranh (concurrency) có thể xảy ra ở 4 mức sau:
  1. Lệnh mã máy
  2. Câu lệnh của NN LT cấp cao (lệnh lặp)
  3. Chương trình con
  4. Chương trình
- Vì không có một NN LT nào hỗ trợ tương tranh ở mức chương trình, và lệnh mã máy nên 2 sự tương tranh này không được trình bày ở chương này

Giới thiệu (tt)

- ĐN: Thread điều khiển trong một chương trình là thứ tự các điểm cần đến của CT
- Phân loại sự tương tranh:
  1. Tương tranh vật lý (physical concurrency) – Multiple processors độc lập (điều khiển với multiple threads)
  2. Trương tranh logic (logical concurrency) – Sự tương tranh này xuất hiện khi có sự chia sẽ trên cùng một processor (Một phần mền có thể được thiết kết với multiple thread)

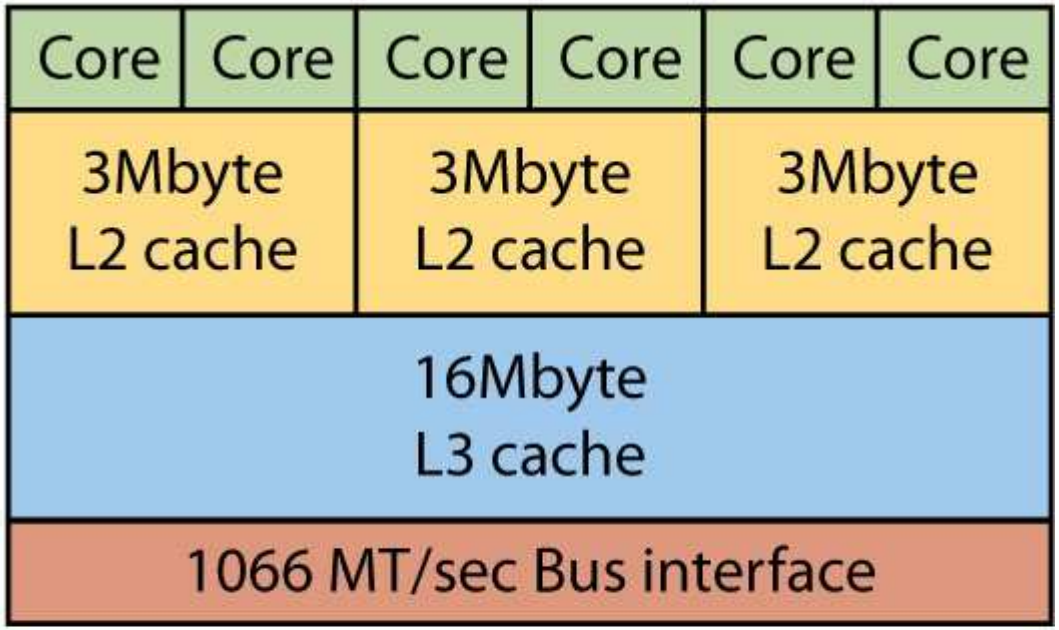
Giới thiệu (tt)

- Tại sao phải học sự tương tranh trong NN LT
  1. Rất hữu dụng cho việc thiết kế chương trình hỗ trợ tính toán song song
  2. Máy tính hỗ trợ tương tranh vật lý (multi-core processors) rất phổ biến

Kiến trúc máy tính multi-core

Single instruction multiple data (SIMD)

Multiple Instruction multiple data (MIMD)



Intel's hex core Xeon includes a large L3 cache.

Tương tranh ở mức chương trình con

- ĐN: Một công việc (task) hoặc tiến trình (process) là một đơn vị chương trình được thực hiện đồng thời với những chương trình khác
- Task khác với chương trình con như thế nào?
  - Task có thể được bắt đầu ở thời điểm tường minh
  - Khi một chương trình bắt đầu thực thi một task, thông thường thì không bị đình hoãn
  - Khi việc thực thi một task kết thúc thì không nhất thiết phải trả quyền điều khiển cho caller
- Các công việc (tasks) có thể trao đổi qua lại

Tương tranh ở mức CT con (tt)

- Thông thường có 2 loại tasks
  - Heavyweight tasks thực thi với không gian địa chỉ và run-time stack của chính nó
  - Lightweight tasks luôn luôn thực thi với cùng không gian địa chỉ và cùng run-time stack

Tương tranh ở mức CT con (tt)

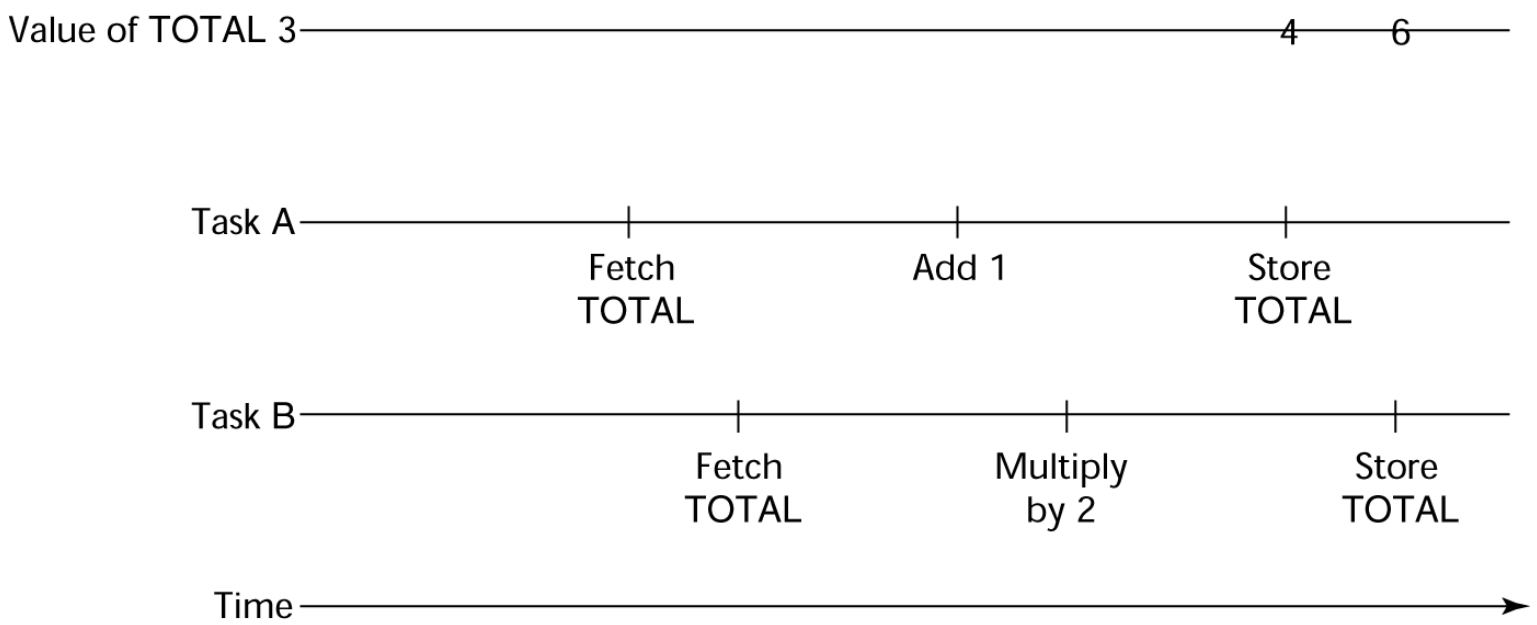
- ĐN: một task riêng biệt (disjoint) nếu như nó không giao tiếp hoặc ảnh hưởng đến sự thực thi của một task nào đó trong một chương trình bất kỳ
- Một task giao tiếp với một task khác thì cần thiết phải có sự đồng bộ hóa (synchronization)
  - Sự giao tiếp có thể bằng:

  1. Chia sẽ các biến không cục bộ
  2. Tham số
  3. Truyền thông điệp

Tương tranh ở mức CT con (tt)

- Các kiểu đồng bộ hóa :
  1. Hợp tác (Cooperation)
     - Task A phải đợi cho đến khi task B hoàn thành một vài tác vụ nhất định nào đó trước khi task A có thể thực hiện tiếp tục → mô hình producer-consumer
  2. Cạnh tranh (competition)
     - Khi hai hoặc nhiều tasks cùng dùng chung một tài nguyên (resource) nhưng tài nguyên này không thể dùng đồng thời được
     - Sự canh tranh thường được cung cấp bởi quyền truy cập loại trừ lẫn nhau

Sự cần thiết của đồng bộ hóa trong cạnh tranh



Tương tranh ở mức CT con (tt)

- Việc đồng bộ hóa đòi hỏi một cơ chế của sự trị hoãn việc thực thi các task
- Sự điều khiển việc thực thi được điều hành bởi một chương trình, gọi là scheduler, có nhiệu vụ sắp đặt việc thực thi task vào những processors sẵn có

Tương tranh ở mức CT con (tt)

- Các Task có thể ở một trong vài trạng thái sau đây:
  1. New – mới khởi tạo nhưng chưa được thực hiện
  2. Runnable hoặc ready – sẵn sàng để chạy nhưng chưa chạy (vì không có processor sẵn có)
  3. Running
  4. Blocked – đã chạy, nhưng không thể tiếp tục vì đang đợi vài sự kiện nào đó xảy ra)
  5. Dead

Tương tranh ở mức CT con (tt)

- Liveness là đặc điểm mà một chương trình có thể có hoặc không
  - Trong code tuần tự, nghĩa là nếu một CT tiếp tục thực thi → dẫn đến sự cạnh tranh
  - Trong môi trường tương tranh, một task có thể dễ dàng mất liveness của nó
  - Nếu tất cả các task trong môi trường tương tranh đều mất liveness của chúng, trường hợp này gọi là deadlock

Tương tranh ở mức CT con (tt)

- Các NN LT hỗ trợ tương tranh đều có 2 cơ chế: đồng bộ hóa cạnh tranh và đồng bộ hóa hợp tác
- Các yếu tố khi thiết kế tương tranh:
  1. Sự đồng bộ hóa hợp tác được cung cấp như thế nào?
  2. Sự đồng bộ hóa tương tranh được cung cấp như thế nào?
  3. Cách gì và khi nào một task bắt đầu và kết thúc thực thi?
  4. Các task có được sinh ra một cách tĩnh hay động?

Tương tranh ở mức CT con (tt)

- Các phương thức đồng bộ hóa:
  1. Semaphores
  2. Chương trình giám sát (Monitors)
  3. Truyền thông điệp (Message Passing)

Semaphores

- Dijkstra - 1965
- Một semaphore là một cấu trúc dữ liệu chứa một counter và một hàng đợi (queue) nhằm lưu trữ các mô tả của task
- Các semaphores được dùng để cài đặt các bảo vệ trong code có sự truy nhập các cấu trúc dữ liệu chia sẽ
- Các semaphores chỉ có 2 thao tác vụ, wait và release (hay được gọi P và V bởi Dijkstra)
- Các semaphores có thể được dùng trong cả đồng bộ hóa cạnh tranh và hợp tác

Semaphores

- Đồng bộ hóa hợp tác với Semaphores
  - Example: Một buffer chia sẽ
  - Buffer được cài đặt với hai tác vụ **`DEPOSIT`** và **`FETCH`** như là hai cách thức để truy nhập buffer
  - Sử dụng hai semaphores cho sự hợp tác: **`emptyspots`** và **`fullspots`**
  - Counter của hai semaphore dùng để lưu trữ số empty spots và full spots trong buffer

Semaphores

- Trước tiên **`DEPOSIT`** phải kiểm tra **`emptyspots`** xem có còn khoảng trống trong buffer không
- Nếu còn khoảng trống thì counter của emtyspots giảm đi một và giá trị được đưa vào buffer
- Nếu không còn khoảng trống thì chương trình gọi DEPOSIT được đặt trong hàng đợi cho đến khi có một emtyp spot rỗng
- Khi kết thúc **`DEPOSIT`**, giá trị của counter của **`fullspots`** được tăng lên một

Semaphores

- Trước tiên **`FETCH`** phải kiểm tra **`fullspots`** xem có còn giá trị nào trong buffer không
  - Nếu còn thì một giá trị được lấy ra và counter của **`fullspots`** bị giảm đi một
  - Nếu không còn giá trị nào thì tiến trình của FETCH được đặt trong hàng đợi cho đến khi có một giá trị xuất hiện
  - Khi kết thúc **`FETCH`**, tăng counter của **`emptyspots`** lên một
- Hai thao tác **`FETCH`** và **`DEPOSIT`** trên các semaphore thành công thông qua hai thao tác wait và release

Semaphores

wait(aSemaphore)

if aSemaphore's counter > 0 then

Decrement aSemaphore's counter

else

Put the caller in aSemaphore's queue

Attempt to transfer control to some ready task

(If the task ready queue is empty, deadlock occurs)

end

Semaphores

release(aSemaphore)

if aSemaphore's queue is empty {no one waiting}
then
Increment aSemaphore's counter
else
Put the calling task in the task ready queue
Transfer control to a task from aSemaphore's queue
end

Producer Consumer Code

```
semaphore fullspots, emptyspots;
fullspots.count = 0;
emptyspots.count = BUFLEN;
task producer;
   loop
   -- produce VALUE --
   wait (emptyspots); {wait for space}
   DEPOSIT(VALUE);
   release(fullspots); {increase
filled}
   end loop;
end producer;
```

Producer Consumer Code

```
task consumer;
   loop
   wait (fullspots);{to make sure it is not empty}
   FETCH(VALUE);
   release(emptyspots); {increase empty space}
   -- consume VALUE --
   end loop;
end consumer;
```

Semaphores

- Đồng bộ hóa cạnh tranh với semaphores
  - Semaphore thứ ba, có tên là **`access`**, dùng để kiểm soát truy cập (đồng bộ hóa cạnh tranh)
    - Counter của **`access`** sẽ chỉ có hai giá trị 0 và 1
    - Tương đương như là một semaphore nhị phân (binary semaphore)
    - Giá trị khởi tạo của **`access`** phải là 1, đồng nghĩa là tài nguyên đang ở trạng thái sẵn sàng. 0 nghĩa là bận

Code của Producer-Consumer

```
semaphore access, fullspots, emptyspots;
access.count = 1;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
    loop
    -- produce VALUE --
    wait(emptyspots); {wait for space}
    wait(access);     {wait for access)
    DEPOSIT(VALUE);
    release(access); {relinquish access}
    release(fullspots); {increase filled space}
    end loop;
end producer;
```

Code của Producer-Consumer

```
task consumer;
   loop
   wait(fullspots);{make sure it is not empty}
   wait(access);    {wait for access}
   FETCH(VALUE);
   release(access); {relinquish access}
   release(emptyspots); {increase empty space}
   -- consume VALUE --
   end loop;
end consumer;
```

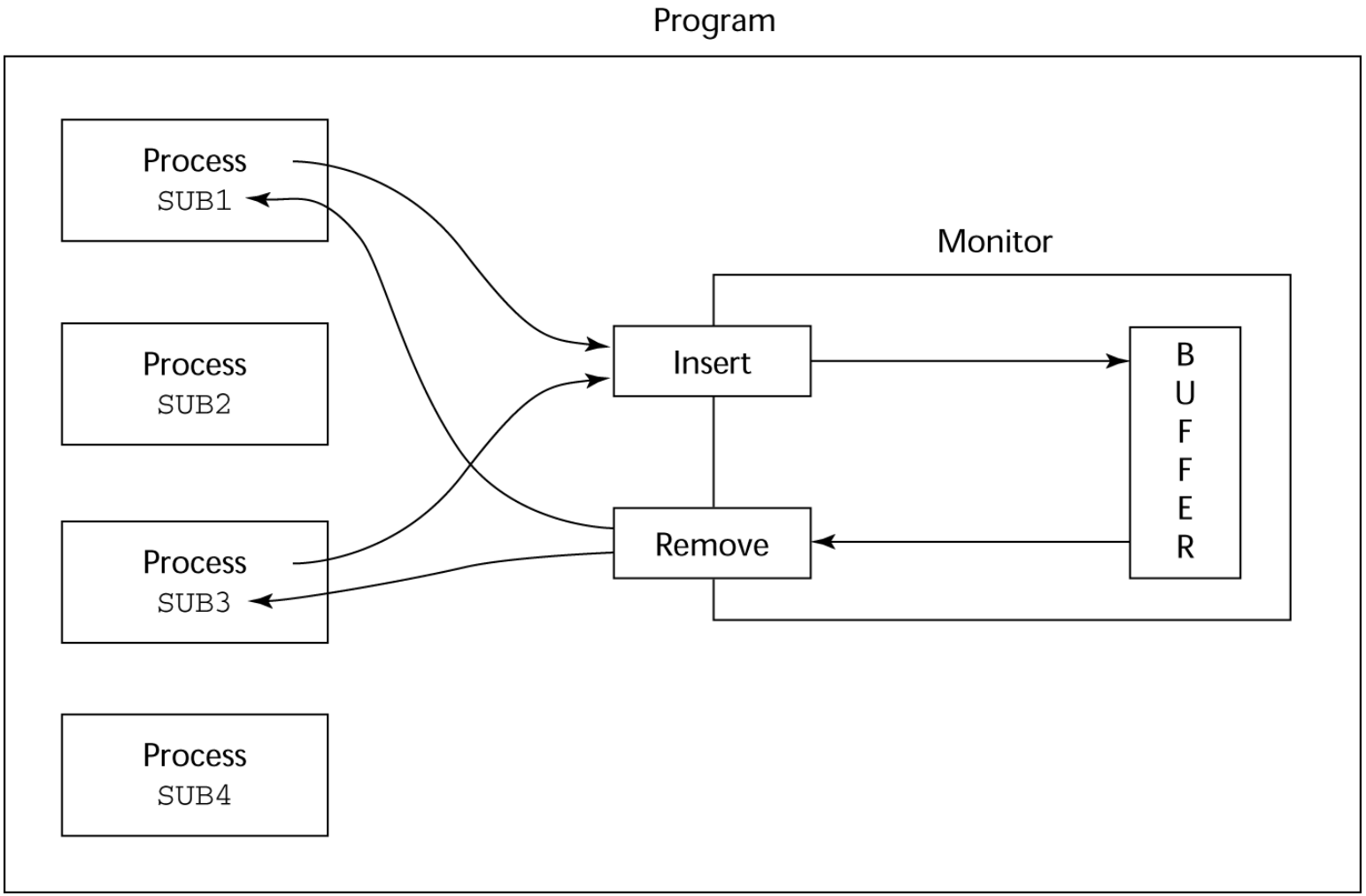
Semaphores : nhận xét

- Môi trường lập trình không an toàn (Unsafe)
  - Sử dụng sai các semaphores có thể là nguyên nhân thất bại trong đồng bộ hóa hợp tác, e.g., buffer sẽ bị tràn (overflow) nếu không có dòng code `wait(emptyspots)` trong producer task. Hoặc buffer sẽ bị underflow nếu không có dòng code `wait(fullspots)`
  - Trình biên dịch không thể kiểm tra việc dùng sai
- Sự tin cậy
  - Sử dụng sai có thể là nguyên nhân thất bại của đồng bộ hóa cạnh tranh, e.g., chương trình sẽ bị deadlock nếu loại bỏ dòng code `release(access)`

Chương trình giám sát (Monitors)

- NNLT : concurrent Pascal, Modula, Mesa, tiếp theo là C#, Ada and Java
- Ý tưởng: bao đống dữ liệu chia sẽ và giới hạn các thao tác truy nhập
- CT giám sát là một trù tường hóa dữ liệu cho những dữ liệu chia sẽ

Monitor Buffer Operation



Đồng bộ hóa cạnh tranh

- Dữ liệu chia sẽ được đặt bên trong CT giám sát (tốt hơn là đặt trong các client)
- Tất cả các truy nhập đều diễn ra ở trong CT giám sát
  - Việc cài đặt CT giám sát phải bảo đảm các truy cập được đồng bộ bằng cách chỉ có một truy cập tại một thời điểm nhất định
  - Nếu CT giám sát bận vào thời điểm gọi, thì các lời gọi sẽ được đặt vào trong hàng đợi

Đồng bộ hóa hợp tác

- Sự hợp tác giữa các tiến trình (processes) vẫn là một tác vụ trong lập trình
  - Lập trình viên phải bảo đảm là không xảy ra underflow và overflow trong một buffer chia sẽ

Chương Trình giám sát: nhận xét

- Hỗ trợ tốt cho sự đồng bộ hóa cạnh tranh
- Đối với đồng bộ hóa hợp tác thì tương tự như semaphore nên → sẽ gặp các vấn đề như semaphore

Truyền thông điệp

- Được đưa ra bởi Hansen & Hoare vào 1978
- Vấn đề: làm thế nào giải quyết vấn đề khi có nhiều yêu cầu giao tiếp từ nhiều task với một task cho trước
  - Vài dạng của cơ chế không quyết định cho sự công bằng
  - Guarded commands của Dijkstra: kiểm soát truyền thông điệp
- Ý tưởng chính: giao tiếp giữa các task giống như đến phòng mạch
  - Phần lớn thời gian BS đợi bệnh nhân
  - Hoặc bệnh nhân đợi BS, BS sẽ khám bệnh cho bệnh nhân nếu cả hai đều rãnh
  - Hoặc lấy cái hẹn

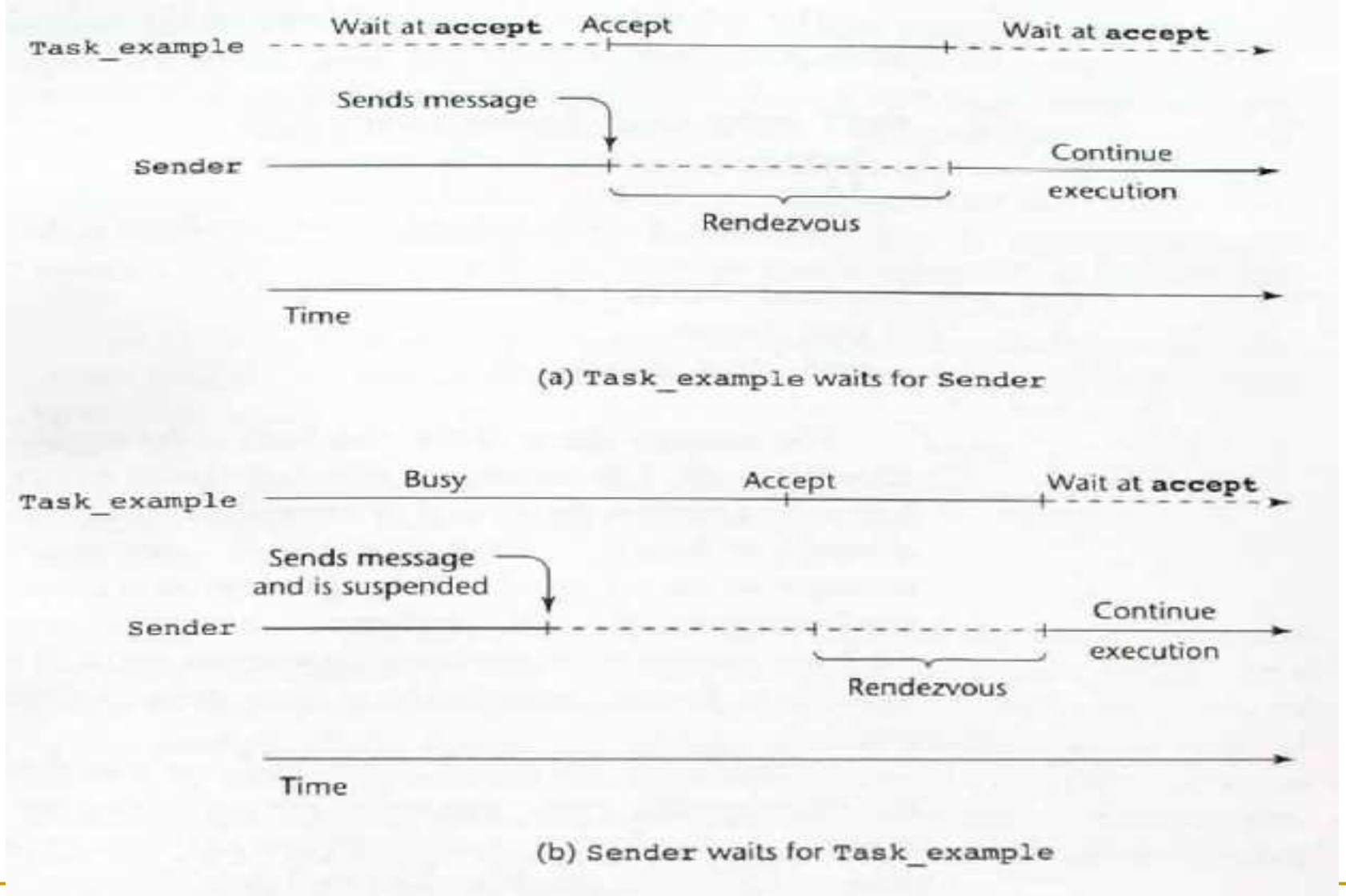
Truyền thông điệp (tt)

- Truyền thông điệp là mô hình tương tranh
  - Có thể là mô hình của cả semaphore và CT giám sát
  - Không chỉ cho đồng bộ hóa cạnh tranh
  - Truyền thông điệp đồng bộ, khi bận các task không muốn bị gián đoạn

Truyền thông điệp

- Trong phạm vi tasks, chúng ta cần:
  a. Một cơ chế để cho phép một task biểu thị khi nào nó sẵn sàng nhận các thông điệp
  b. Các tasks cần cách ghi nhớ các task khác đang đợi nó nhận message và có sự lựa chọn các message tiếp theo
- ĐN: Khi message của một task được nhận bởi một task nào đó, thì sự truyền message được gọi là rendezvous

VD về Rendezvous



Tương tranh trong Java: Java thread

- Khi chương trình Java thực thi hàm main() tức là tạo ra một luồng chính (main thread)
- Trong luồng main
  - Có thể tạo các luồng con
  - Khi luồng main ngừng thực thi, chương trình sẽ kết thúc
- Luồng có thể được tạo ra bằng 2 cách:
  - Tạo lớp dẫn xuất từ lớp Thread
  - Tạo lớp hiện thực giao tiếp Runnable

Tương tranh trong Java (tt)

- VD

```
public class Mythread extends Thread{
  private String data
  public Mythread(String data){
    this.data = data;
  }
  public void run(){
    System.out.println(‘‘I am a thread!’’);
    System.out.println(‘‘The data is:’’,data);
  }
}
```

Tương tranh trong Java (tt)

- Tạo ra một thể hiện của lớp Thread (hoặc dẫn xuất của nó) và gọi phương thức start()

```
public class ExampleThread
{
    public static void main(String[] args) {
        Thread myThread = new MyThread("my data");
        myThread.start();
        System.out.println("I am the main thread");
    }
}
```

- Khi gọi myThread.start() một luồng mới tạo ra và chạy phương thức run() của myThread.

Tương tranh trong Java: Runnable

- Giao tiếp Runnable
  - Ngoài tạo luồng bằng cách thừa kế từ lớp Thread, cũng có một cách khác để tạo luồng trong Java
  - Luồng có thể tạo bằng cách tạo lớp mới hiện thực giao tiếp Runnable và định nghĩa phương thức:

    public abstract void run()
  - Điều này đặc biệt hữu ích nếu muốn để tạo ra một đối tượng Thread nhưng muốn sử dụng một lớp cơ sở khác Thread

Tương tranh trong Java: Runnable

- VD

```
class MyThreadRbl extends JFrame implements Runnable
{
    private String data;

    public MyThreadRbl(String data) {
        this.data = data;
    }

    public void run() {
        System.out.println("I am a thread");
        System.out.println("The data is : " + data);
    }
}
```

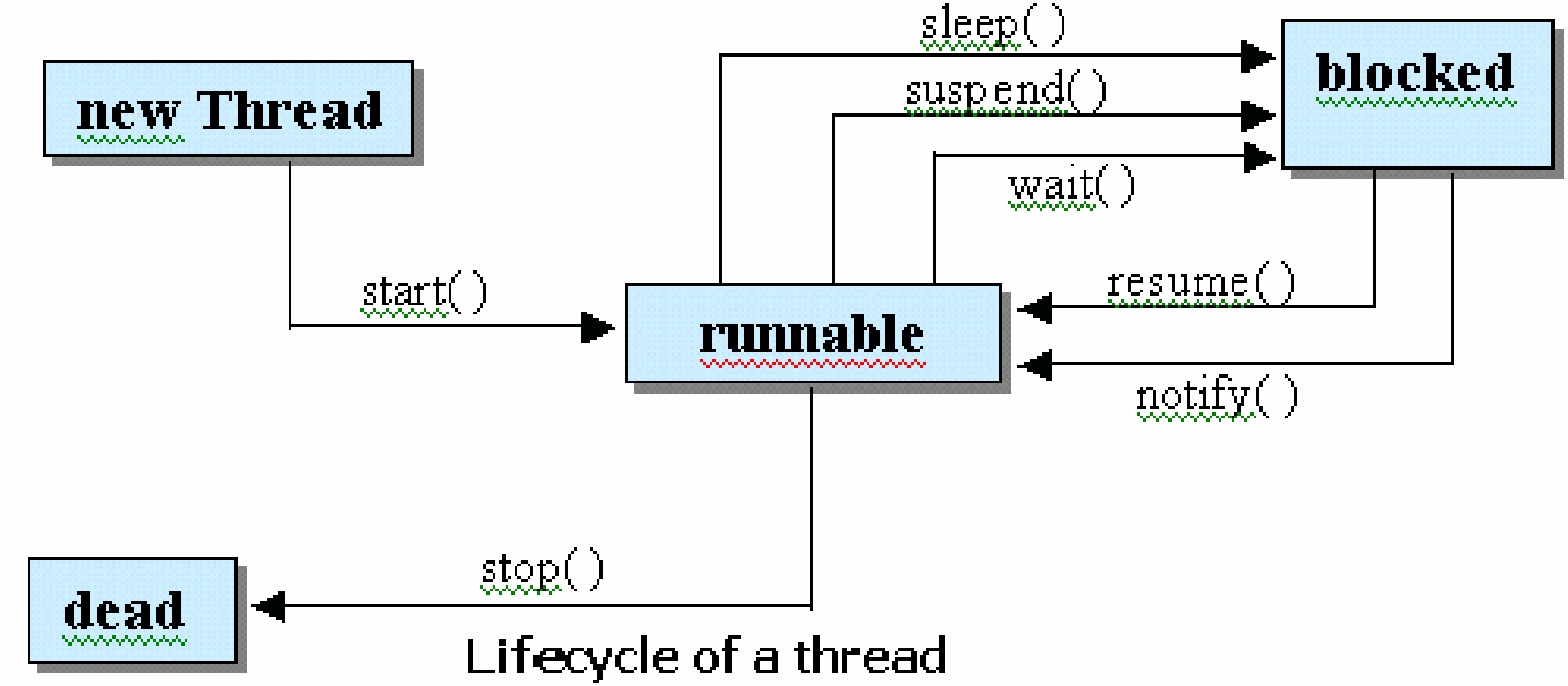
Tương tranh trong Java: Runnable

- Để tạo ra một luồng mới từ một đối tượng hiện thực giao tiếp Runnable, cần phải khởi tạo một đối tượng Thread mới với đối tượng Runnable như đích của nó

```
public class MyThreadStart
{
    public static void main(String[] args) {
        MyThreadRbl thrbl = new MyThreadRbl("my data");
        Thread myThread = new Thread(thrbl);
        myThread.start();
        System.out.println("I am the main thread!");
    }
}
```

- Khi gọi start() trên đối tượng luồng sẽ tạo ra một luồng mới và phương thức run() của đối tượng Runnable sẽ được thực hiện

Tương tranh trong Java: điều khiển



Lifecycle of a thread

Tương tranh trong Java: điều khiển

- Khi một luồng giành quyền sử dụng CPU, nó sẽ thực hiện cho đến khi một sự kiện sau xuất hiện:
  - Phương thức run() kết thúc
  - Một luồng quyền ưu tiên cao hơn
  - Nó gọi phương thức sleep() hay yield() – nhượng bộ
- Khi gọi yield(), luồng đưa cho các luồng khác với cùng quyền ưu tiên cơ hội sử dụng CPU. Nếu không có luồng nào khác cùng quyền ưu tiên tồn tại, luồng tiếp tục thực hiện
- Khi gọi sleep(), luồng ngủ trong một số mili-giây xác định, trong thời gian đó bất kỳ luồng nào khác có thể sử dụng CPU

Tương tranh trong Java: điều khiển

- Phương thức join()
  - Khi một luồng (A) gọi phương thức join() của một luồng nào đó (B), luồng hiện hành (A) sẽ bị khóa chờ (blocked) cho đến khi luồng đó kết thúc (B).

Đồng bộ hóa cạnh tranh với java

- Dùng từ khóa synchronized trước tên các hàm khi định nghĩa để cấm các lớp khác thực thi các hàm này khi nó đang thực thi

```
class ManageBuf{
  private int [100] buf;
  …
  public synchonized void deposit(int item){…}
  public synchonized void fetch(int item){…}
  …
}
```

Đồng bộ hóa hợp tác với java

- Các phương thức wait(), notify() và notifyAll() được sử dụng để thóa khóa trên một đối tượng và thông báo các luồng đang đợi chúng có thể có lại điều khiển
- wait() được gọi trong vòng lập
- notify() thông báo cho thread đang chờ đợi là sự kiện đang đợi đã xãy ra
- notifyall() đánh thức các thread đang đợi là có thể thực thi sau lệnh wait()

Đồng bộ hóa hợp tác với java: VD

```
public synchronized int get() {
    while (available == false) {
        try {
            //wait for Producer to put value
            wait();
        } catch (InterruptedException e) { }
    }
    available = false;
    //notify Producer that value has been retrieved
    notifyAll();
    return contents;
}
```

Đồng bộ hóa hợp tác với java: VD

```
public synchronized void put(int value) {
    while (available == true) {
        try {
            //wait for Consumer to get value
            wait();
        } catch (InterruptedException e) { }
    }
    contents = value;
    available = true;
    //notify Consumer that value has been set
    notifyAll();
}
```

SimpleThread

```
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
```

TwoThreads

```
class TwoThreadsTest {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
```