Chương 7

Lập trình song song bằng Java

7.0 Dẫn nhập
7.1 Tổng quát về lập trình song song
7.2 Lập trình multi-process bằng class Process
7.3 Lập trình multi-thread bằng class Thread
7.4 Demo tính hiệu quả của multi-thread
7.5 Demo vấn đề tương tranh giữa các thread đồng thời
7.6 Demo việc giải quyết tương tranh giữa các thread
7.7 Kết chương

7.0 Dẫn nhập

- Chương này giới thiệu các ưu khuyết điểm của 2 thuật giải tuần tự và song song trong việc giải quyết bài toán xác định, từ đó cho thấy tính hiệu quả trong việc dùng thuật giải song song để giải quyết bài toán.
- Chương này giới thiệu các đối tượng JDK để quản lý process và thread, phục vụ lập trình song song theo kỹ thuật multi-process và multi-threads.
- Chương này cũng giới thiệu vấn đề tương tranh giữa các process hay các thread trong việc truy xuất tài nguyên dùng chung, cách giải quyết tương tranh bằng phương pháp dùng class semaphore của JDK.

7.1 Tổng quát về lập trình song song

- Để giải quyết bài toán nào đó, ta thường dùng giải thuật tuần tự nhờ tính dễ hiểu, dễ kiểm soát của nó. Chương trình dùng thuật giải tuần tự khi chạy trở thành process mono-thread hay process tuần tự : nó bắt đầu chạy từ lệnh đầu tiên của phần mềm, hết lệnh này đến lệnh khác cho đến khi thực hiện xong lệnh cuối cùng thì kết thúc.
- Process tuần tự hoạt động không hiệu quả vì không dùng triệt để các CPU trên máy tính vật lý (mỗi thời điểm chỉ dùng 1 CPU). Lưu ý rằng hiện nay các máy PC, smartphone hay tablet đều dùng CPU đa nhân. Thí dụ galaxy S4 ở thị trường Việt Nam có 8 nhân.
- Để máy giải quyết bài toán hiệu quả hơn, ta nên dùng thuật toán song song bằng cách nhận dạng các hoạt động có thể thực hiện đồng thời rồi nhờ nhiều CPU thực hiện chúng đồng thời.

7.2 Lập trình multi-process bằng class Process

- Một trong các phương pháp hiện thực thuật toán song song là lập trình multi-process và multi-thread.
- Môi trường JDK cung cấp 2 class tên là Process và Runtime để giúp ta lập trình multi-process dễ dàng.
- Class Process thuộc package java.lang, nó chứa các thuộc tính và tác vụ giúp ta quản lý process dễ dàng, thuận lợi.
- Class Runtime cũng thuộc package java.lang, nó giúp ta khởi chạy ứng dụng bất kỳ, ứng dụng được khởi chạy này sẽ trở thành process mới, tham khảo của đối tượng Process này sẽ được trả về để ta lưu giữ hầu quản lý tiếp process đó.

7.2 Lập trình multi-process bằng class Process

- Thí dụ sau là đoạn code giúp ta khởi chạy 1 chương trình :

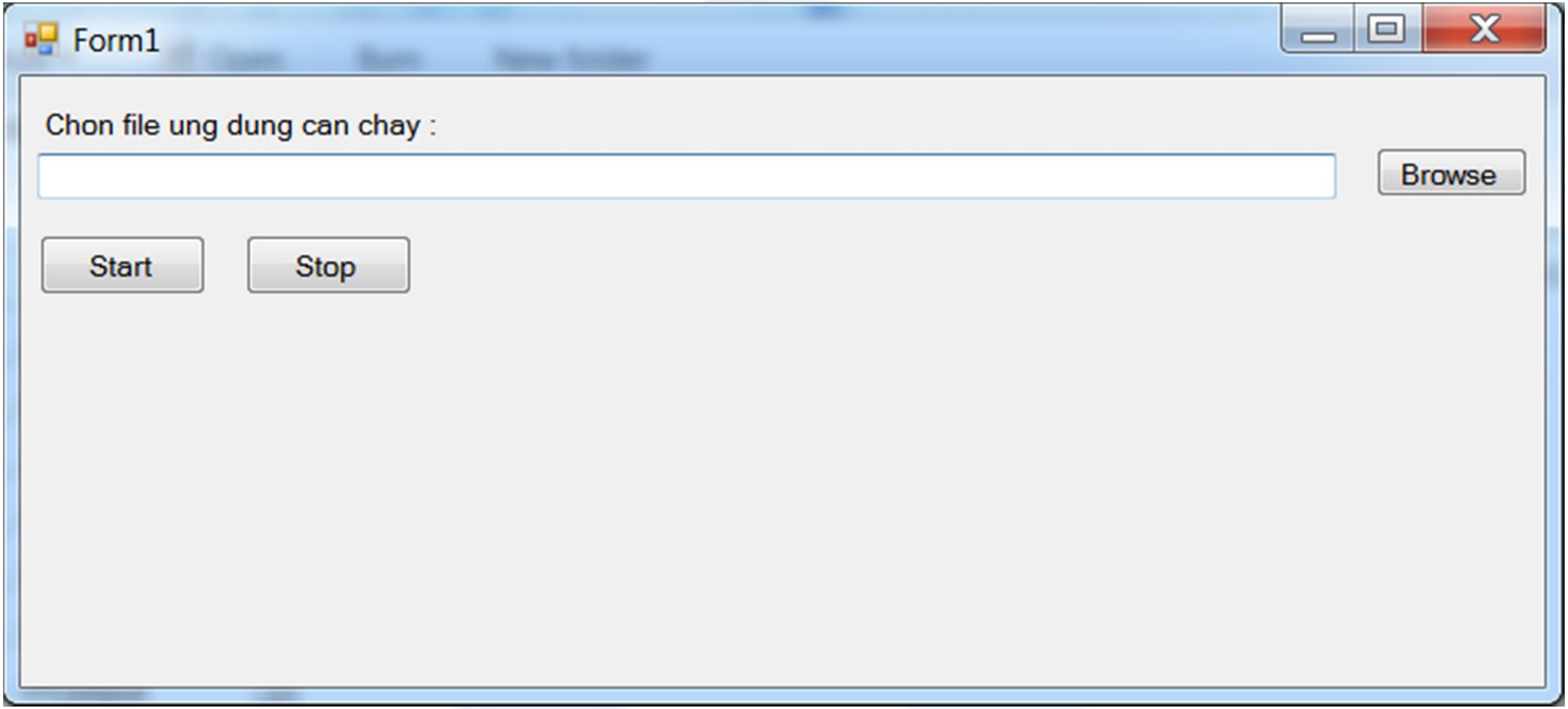
```
Process prc;
try {
   prc = Runtime.getRuntime().exec("d:\\MyProgs\\myprog.exe");
} catch (Exception e) { //xử lý lỗi }
```

- Sau khi được kích hoạt, process sẽ chạy song hành và độc lập với process kích hoạt nó cho đến hết thuật giải của nó. Tuy nhiên, từ bên ngoài ta có thể giết tường minh process nhờ tác vụ destroy() :

```
prc.destroy();
```

7.2 Lập trình multi-process bằng class Process

- Ta hãy thử viết 1 ứng dụng quản lý process đơn giản có form giao diện như sau :



7.2 Lập trình multi-process bằng class Process

- Code của hàm xử lý button Start như sau :

```
Process prc;
private void btnStartActionPerformed(java.awt.event.ActionEvent
 evt) {
 try {
    //khởi chạy chương trình do người dùng chọn
    prc = Runtime.getRuntime().exec(txtPath.getText());
 } catch (Exception e) { //xử lý lỗi }
```

- Code của hàm xử lý button Stop như sau :

```
private void btnStopActionPerformed(java.awt.event.ActionEvent
 evt) { prc.destroy();
 }
```

7.3 Lập trình multi-threads bằng class Thread

- Môi trường JDK cung cấp class tên là Thread để giúp ta lập trình multi-thread dễ dàng.
- Class Thread thuộc package java.lang, nó chứa các thuộc tính và tác vụ giúp ta quản lý thread dễ dàng, thuận lợi.
- Thường mỗi thread sẽ chạy đoạn code được miêu tả trong 1 hàm chức năng xác định. Thí dụ khi process được kích hoạt, HĐH sẽ tạo tường minh thread ban đầu cho process đó, thread chính này sẽ chạy đoạn code của hàm main của class ứng dụng.
- Để tạo thread mới, ta có thể dùng lệnh :

```
//MyThread là class con của Thread mà ta định nghĩa để quản lý
//thread mà ta muốn khởi chạy
MyThread t = new MyThread (các tham số cần truyền);
```

7.3 Lập trình multi-threads bằng class Thread

- Để kích hoạt chạy thread, ta có thể gọi tác vụ Start :

  t.Start ();

- Lưu ý tác vụ mà thread sẽ chạy phải là tác vụ run trong class MyThread có đặc tả như sau :

```
//tác vụ mà thread sẽ chạy
public void run(các tham số cần nhận) {
    ...
}
```

7.3 Lập trình multi-threads bằng class Thread

- Để tạm dừng thread, ta có thể gọi tác vụ suspend :

  t.suspend();
- Để chạy tiếp thread, ta có thể gọi tác vụ resume :

  t.resume();
- Để thread tạm dừng n miliseconds rồi chạy tiếp, ta có thể gọi tác vụ sleep :

  t.sleep(n);
- Để dừng và xóa thread, ta có thể gọi tác vụ stop :

  t.stop();
- Để thay đổi quyền ưu tiên của thread, ta có thể gọi tác vụ setPriority :

  t.setPriority(chỉ số quyền ưu tiên);

7.3 Lập trình multi-threads bằng class Thread

- Trên Windows, mỗi process có thể ở 1 trong 6 cấp quyền ưu tiên sau đây :
  - IDLE_PRIORITY_CLASS
  - BELOW_NORMAL_PRIORITY_CLASS
  - NORMAL_PRIORITY_CLASS
  - ABOVE_NORMAL_PRIORITY_CLASS
  - HIGH_PRIORITY_CLASS
  - REALTIME_PRIORITY_CLASS
- Cấp quyền ưu tiên của process sẽ quyết định các thread trong process đó chạy theo quyền ưu tiên như thế nào.

7.3 Lập trình multi-threads bằng class Thread

- Trên Windows, mỗi thread trong process có thể ở 1 trong 7 cấp quyền ưu tiên sau đây :
  - THREAD_PRIORITY_IDLE
  - THREAD_PRIORITY_LOWEST
  - THREAD_PRIORITY_BELOW_NORMAL
  - THREAD_PRIORITY_NORMAL
  - THREAD_PRIORITY_ABOVE_NORMAL
  - THREAD_PRIORITY_HIGHEST
  - THREAD_PRIORITY_TIME_CRITICAL
- Cấp quyền ưu tiên của process sẽ quyết định các thread với từng cấp quyền ưu tiên trên đây sẽ được xử lý như thế nào.

7.3 Lập trình multi-threads bằng class Thread

Process priority class	Thread priority level	Base priority
IDLE_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	2
	THREAD_PRIORITY_BELOW_NORMAL	3
	THREAD_PRIORITY_NORMAL	4
	THREAD_PRIORITY_ABOVE_NORMAL	5
	THREAD_PRIORITY_HIGHEST	6
	THREAD_PRIORITY_TIME_CRITICAL	15

7.3 Lập trình multi-threads bằng class Thread

Process priority class	Thread priority level	Base priority
BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	4
	THREAD_PRIORITY_BELOW_NORMAL	5
	THREAD_PRIORITY_NORMAL	6
	THREAD_PRIORITY_ABOVE_NORMAL	7
	THREAD_PRIORITY_HIGHEST	8
	THREAD_PRIORITY_TIME_CRITICAL	15

7.3 Lập trình multi-threads bằng class Thread

Process priority class	Thread priority level	Base priority
NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	6
	THREAD_PRIORITY_BELOW_NORMAL	7
	THREAD_PRIORITY_NORMAL	8
	THREAD_PRIORITY_ABOVE_NORMAL	9
	THREAD_PRIORITY_HIGHEST	10
	THREAD_PRIORITY_TIME_CRITICAL	15

7.3 Lập trình multi-threads bằng class Thread

Process priority class	Thread priority level	Base priority
ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	8
	THREAD_PRIORITY_BELOW_NORMAL	9
	THREAD_PRIORITY_NORMAL	10
	THREAD_PRIORITY_ABOVE_NORMAL	11
	THREAD_PRIORITY_HIGHEST	12
	THREAD_PRIORITY_TIME_CRITICAL	15

7.3 Lập trình multi-threads bằng class Thread

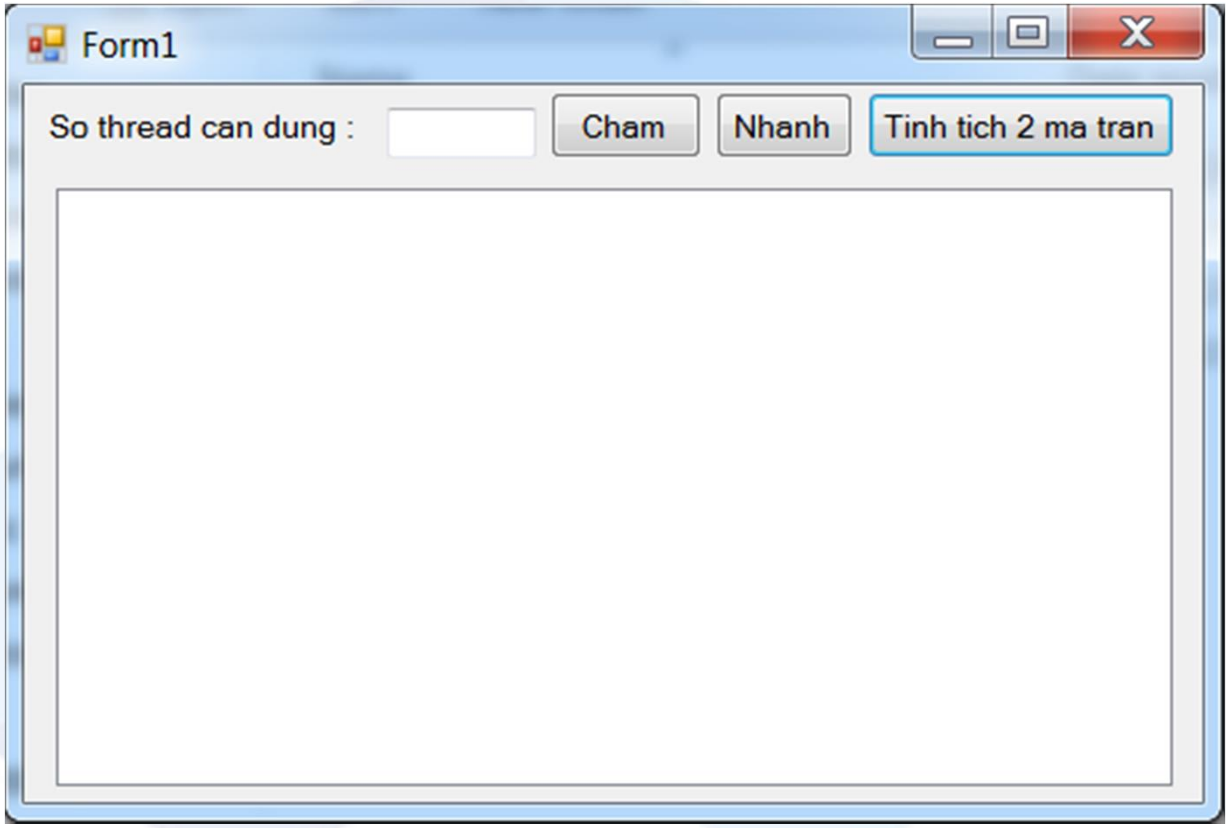
Process priority class	Thread priority level	Base priority
HIGH_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	11
	THREAD_PRIORITY_BELOW_NORMAL	12
	THREAD_PRIORITY_NORMAL	13
	THREAD_PRIORITY_ABOVE_NORMAL	14
	THREAD_PRIORITY_HIGHEST	15
	THREAD_PRIORITY_TIME_CRITICAL	15

7.3 Lập trình multi-threads bằng class Thread

Process priority class	Thread priority level	Base priority
REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	16
	THREAD_PRIORITY_LOWEST	22
	THREAD_PRIORITY_BELOW_NORMAL	23
	THREAD_PRIORITY_NORMAL	24
	THREAD_PRIORITY_ABOVE_NORMAL	25
	THREAD_PRIORITY_HIGHEST	26
	THREAD_PRIORITY_TIME_CRITICAL	31

7.4 Demo tính hiệu quả của multi-thread

- Để demo tính hiệu quả của lập trình multi-thread trên các máy tính đa nhân, ta hãy thử viết ứng dụng tính tích của 2 ma trận có kích thước lớn (thí dụ 2000*2000). Ta thiết kế form ứng dụng như sau :



7.4 Demo tính hiệu quả của multi-thread

- Ứng dụng cho phép người dùng nhập số thread cần dùng (cnt), chia ma trận tích thành N/cnt nhóm hàng rồi tạo thread con để tính dùm từng nhóm hàng. Sau khi tính ma trận tích xong, ứng dụng sẽ hiển thị cho người dùng thấy thời gian tính toán để họ đánh giá độ hiệu quả của thuật giải theo số lượng thread được dùng.

7.4 Demo tính hiệu quả của multi-thread

```
public void TinhTich (int sr, int er, int id) {
    //ghi nhận thời điểm bắt đầu chạy
    long t1 = Calendar.getInstance().getTimeInMillis();
    int h, c, k;
    for (h = p.sr; h < p.er; h++) //lặp theo hàng
        for (c = 0; c < N; c++) { //lặp theo cột
            double s = 0;
            for (k = 0; k < N; k++)
                s = s + A[h, k] * B[k, c];
            C[h, c] = s;
        }
    stateLst[id] = 1;    //ghi nhận trạng thái hoàn thành
    long t2 = Calendar.getInstance().getTimeInMillis();
    dateLst[id] = t2-t1;//tính thời gian chạy
}
```

7.4 Demo tính hiệu quả của multi-thread

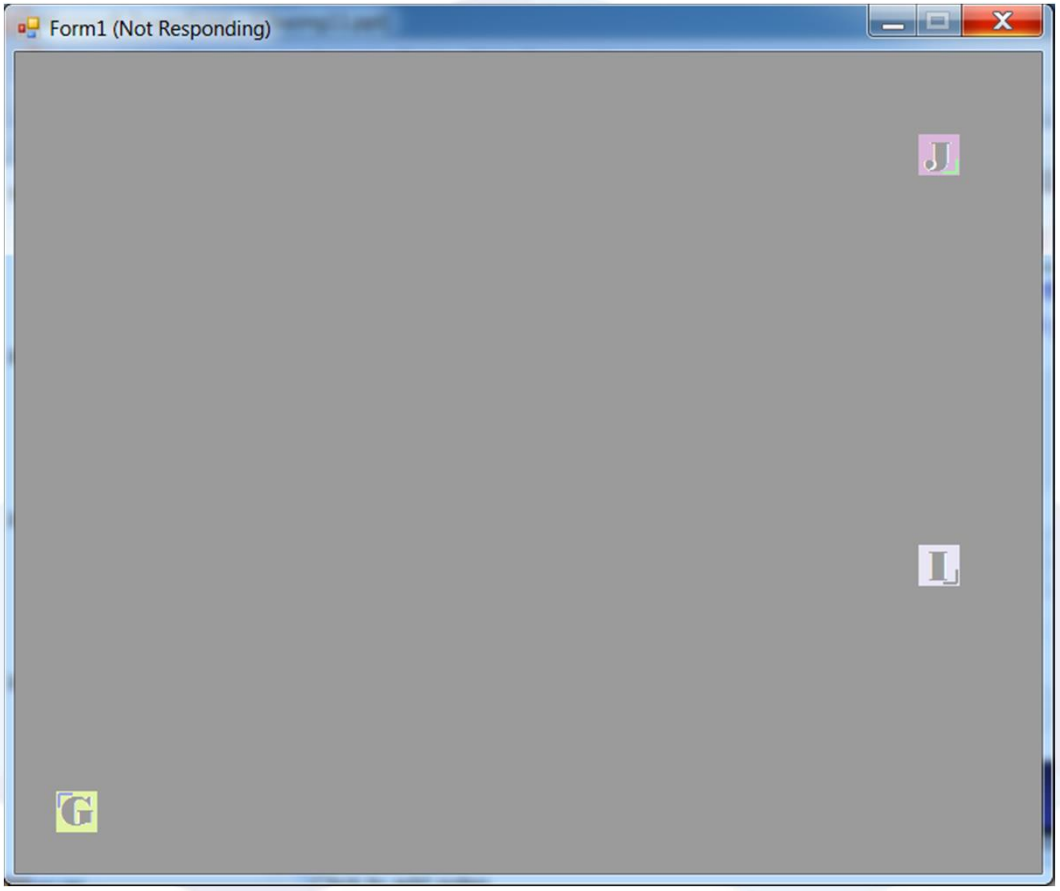
```
//xác định số thread được yêu cầu chạy
int cnt = Integer.valueOf(txtThreads.getText().toString());
//ghi nhận thời điểm bắt đầu tính tích
long t1 = Calendar.getInstance().getTimeInMillis();
if (cnt == 1) { //dùng thuật giải tuần tự
    TinhTich(0, N, 0);
} else { //dùng thuật giải // : 1 thread hiện hành + cnt-1 thread con
    int i;
    for (i = 0; i < cnt-1; i++) {
        stateLst[i] = 0;
        MyThread  t = new MyThread(i*N/cnt, (i+1)*N/cnt,i,this);
        t.start();    //khởi chạy thread vừa tạo
    }
```

7.4 Demo tính hiệu quả của multi-thread

```
    //thread cha tính N/cnt hàng cuối của ma trận tích
    TinhTich((cnt-1)*N/cnt, N, cnt-1);
    //rồi lặp đợi từng thread con hoàn thành nhiệm vụ
    for (i = 0; i < cnt-1; i++)
        while (stateLst[i] == 0); //chờ thread con i hoàn thành
}
//tính tổng thời gian tính tích 2 ma trận
long diff = Calendar.getInstance().getTimeInMillis() - t1;
//đổi ra số phút, số giây, số ms rồi hiển thị lên ListBox
long diffS = (diff / 1000)%60; long diffM = diff / (60 * 1000);
diff = diff % 1000;
buf = cnt+" threads ==> Thời gian chạy là  "+ diffM+ " phút "+ diffS+ "
   giây "+diff+ " ms";
lmKetqua.addElement(buf);
```

7.5 Demo vấn đề tương tranh giữa các thread

- Để demo vấn đề tương tranh giữa các thread, ta hãy thử viết ứng dụng quản lý các thread với giao diện như sau :



7.5 Demo vấn đề tương tranh giữa các thread

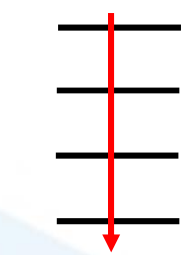
- Form là ma trận gồm nhiều cell, mỗi cell chứa được icon ảnh cho 1 thread đang chạy. Lúc đầu, chưa có thread nào chạy hết. Người dùng có thể ấn phím để quản lý các thread như sau :
  - Ấn phím từ A-Z để kích hoạt chạy thread có tên tương ứng.
  - Ấn phím Ctrl-Alt-X để tạm dừng chạy thread X.
  - Ấn phím Alt-X để chạy tiếp thread X.
  - Ấn phím Shift-X để tăng độ ưu tiên chạy cho thread X.
  - Ấn phím Ctrl-X để giảm độ ưu tiên chạy cho thread X.
  - Ấn phím Ctrl-Shift-X để dừng và thoát thread X.
- Để thấy trực quan sự hoạt động của thread, hoạt động của thread là hiển thị icon miêu tả mình lên form, icon này sẽ chạy theo 1 phương xác định, khi đụng thành form thì dội lại theo nguyên lý vật lý.

7.5 Demo vấn đề tương tranh giữa các thread

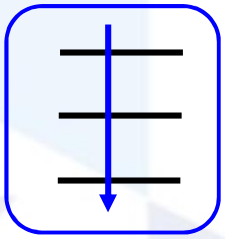
- Trong ứng dụng này, các cell trên form là các tài nguyên dùng chung của các thread. Ta dùng thuật ngữ "tương tranh" giữa các thread để miêu tả sự truy xuất đồng thời của các thread trên các tài nguyên dùng chung.
- Quan sát quỹ đạo chạy icon của từng thread, ta thấy thỉnh thoảng có icon của thread này đè mất icon của thread khác. Đây là hiện tượng lỗi không mong muốn do các thread được quyền tự do chiếm dụng từng cell của form để hiển thị icon của mình mà không hề quan tâm xem cell đó đang được chiếm giữ bởi thread khác không.
- Cần có biện pháp quản lý tương tranh sao cho các thread không được quyền truy xuất tài nguyên dùng chung đồng thời. Phương pháp phổ biến nhất để giải quyết tương tranh là phương pháp loại trừ tương hỗ.

7.6 Demo việc giải quyết tương tranh giữa các thread

- Ta gọi đoạn code truy xuất tài nguyên dùng chung là CS (Critical section).
- Mỗi lần muốn vào vùng CS, ta phải gọi hàm In_Control() để kiểm soát việc thi hành tuần tự vùng CS.
- Khi hoàn thành vùng CS, ta phải gọi hàm Out_Control() để thông báo cho các thread khác đang chờ để chúng kiểm tra lại việc đi vào.

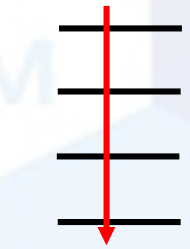


In_Control();



Vùng CS truy xuất tài nguyên dùng chung

Out_Control();



7.6 Demo việc giải quyết tương tranh giữa các thread

- Phương pháp loại trừ tương hỗ phổ dụng hiện nay là dùng semaphore nhị phân (Mutex). Semaphore là 1 đối tượng đơn giản chứa :
  - 1 thuộc tính kiểu nguyên dương (s), ta còn gọi nó là biến semaphore. Nếu là semaphore nhị phân thì s chỉ có thể chứa 2 giá trị 0 và 1.
  - Tác vụ down(), có nhiệm vụ giảm s 1 đơn vị và luôn phải hoàn thành. Do đó trong trường hợp s = 0, tác vụ down sẽ phải ngủ chờ đến khi s <> 0 thì cố gắng thực hiện lại… ➔ Thời gian thi hành tác vụ down là không xác định.
  - Tác vụ up(), có nhiệm vụ tăng s 1 đơn vị và luôn phải hoàn thành. Trong trường hợp s = 0, tác vụ up sẽ phải đánh thức các thread đang ngủ chờ down s dậy.

7.6 Demo việc giải quyết tương tranh giữa các thread

- Môi trường JDK cung cấp class Semaphore để quản lý semaphore. Ta kết hợp mỗi tài nguyên dùng chung 1 semaphore nhị phân m với giá trị đầu = 1.
- Tác vụ down() được hiện thực với tên là acquire(), do đó hàm In_Control() sẽ là lệnh m.acquire(); Thread nào thực hiện lệnh này đầu tiên sẽ thành công ngay và sẽ chạy được đoạn lệnh CS truy xuất tài nguyên tương ứng. Các thread khác thực hiện lệnh trên để truy xuất tài nguyên dùng chung sẽ thất bại và bị ngủ trong khi thread đầu chưa hoàn thành truy xuất.
- Tác vụ up() được hiện thực với tên là release, do đó hàm Out_Control() sẽ là lệnh m.release(). Nó sẽ đánh thức các thread đang ngủ chờ nếu có.

7.6 Demo việc giải quyết tương tranh giữa các thread

```
//xin khóa truy xuất cell bắt đầu (x1,y1)
mutList[y1][x1].acquire();
while (fstart) { //lặp trong khi chưa có yêu cầu kết thúc
    //hiển thị icon miêu tả mình lên cell (x1,y1)
    //thực hiện công việc của thread
    //xác định vị trí mới của thread (x2,y2)
    //xin khóa truy xuất cell (x2,y2)
    while (true) {
        kq = mutList[y2][x2].tryAcquire(30, TimeUnit.MILLISECONDS));
        if (kq==true || fstart==false) break;
    }
    // Xóa icon của thread ở vị trí cũ (x1,y1)
    //giải phóng cell (x1,y1) cho các thread khác truy xuất
    mutList[y1][x1].release();
    x1 = x2; y1 = y2;
```

7.6 Demo việc giải quyết tương tranh giữa các thread

- Theo thuật giải của slide trước, đầu tiên thread i sẽ chờ xin truy xuất cell xuất phát (x1,y1). Nếu không được, nó sẽ chờ mãi cho đến khi được thì chạy tiếp. Trong lúc thread i chờ, không ai chờ thread i vì thread i chưa chiếm dụng cell dùng chung nào cả.
- Sau khi khóa được cell (x1, y1), thread i hiển thị icon lên cell này, tính toán công việc, xác định cell kế (x2,y2) cần chuyển đến.
- Thread i sẽ chờ xin truy xuất cell kế (x2,y2) trong lúc giữ cell (x1,y1). Nếu không được, nó sẽ chờ mãi cho đến khi được thì chạy tiếp, trong khoản thời gian này nếu có thread khác (j) cần cell (x1,y1) thì phải chờ thread i.
- Nếu thread j đang chiếm cell (x2, y2) thì deadlock đã xảy ra : thread i và j chờ nhau vô tận.
- Hiện các HĐH phổ biến không giải quyết deadlock, do đó ứng dụng và người dùng phải tự giải quyết lấy.

7.7 Kết chương

- Chương này đã giới thiệu các ưu khuyết điểm của 2 thuật giải tuần tự và song song trong việc giải quyết bài toán xác định, từ đó cho thấy tính hiệu quả trong việc dùng thuật giải song song để giải quyết bài toán.
- Chương này đã giới thiệu các đối tượng JDK để quản lý process và thread, phục vụ lập trình song song theo kỹ thuật multi-process và multi-threads.
- Chương này cũng đã giới thiệu vấn đề tương tranh giữa các process hay các thread trong việc truy xuất tài nguyên dùng chung, cách giải quyết tương tranh bằng phương pháp dùng class semaphore của JDK.