Multithreading

Banyak persoalan dalam pemrograman membutuhkan kemampuan suatu program untuk melakukan beberapa hal sekaligus, atau memberikan penanganan segera terhadap suatu kejadian/ event tertentu dengan menunda aktivitas yang sedang dijalankan untuk menangani event tersebut dan akhirnya kembali melanjutkan aktivitas yang tertunda.

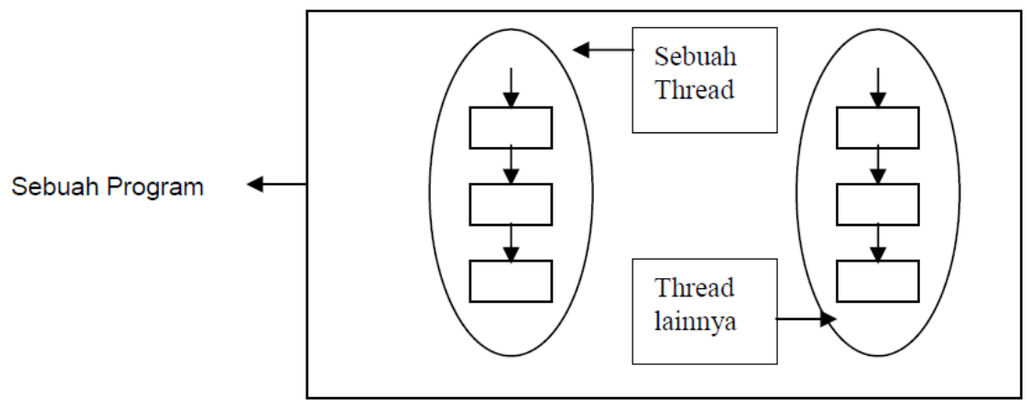
Contoh, dalam sistem aplikasi jaringan, kita dapat membuat suatu program melakukan komputasi lokal dengan data yang sudah didapat dari jaringan, pada saat program tersebut menunggu datangnya tambahan data dari jaringan. Tanpa multithreading, program tersebut harus melakukannya secara sekuensial dalam sebuah alur program tunggal (yaitu alur control utama), yang diawali dengan penantian tibanya keseluruhan data, baru kemudian komputasi. Pada masa penantian tersebut, komputer berada pada keadaan idle yang menyebabkan ketidakefisienan pada keseluruhan program.

Dengan multithreading kita dapat menciptakan dua thread secara dinamis, yaitu thread yang berjaga dipintu gerbang, menunggu masuknya data., dan thread yang melakukan komputasi lokal atas data yang sudah tersedia.

Multithreading dan Java

Thread (seringkali disebut juga lightweight process atau execution context) adalah sebuah single sequential flow of control didalam sebuah program. Secara sederhana, thread adalah sebuah subprogram yang berjalan didalam sebuah program.

Seperti halnya sebuah program, sebuah thread mempunyai awal dan akhir. Sebuah program dapat mempunyai beberapa thread didalamnya. Jadi perbedaannya program yang multithreaded mempunyai beberapa flow of control yang berjalan secara konkuren atau paralel sedangkan program yang singlethreaded hanya mempunyai satu flow of control.



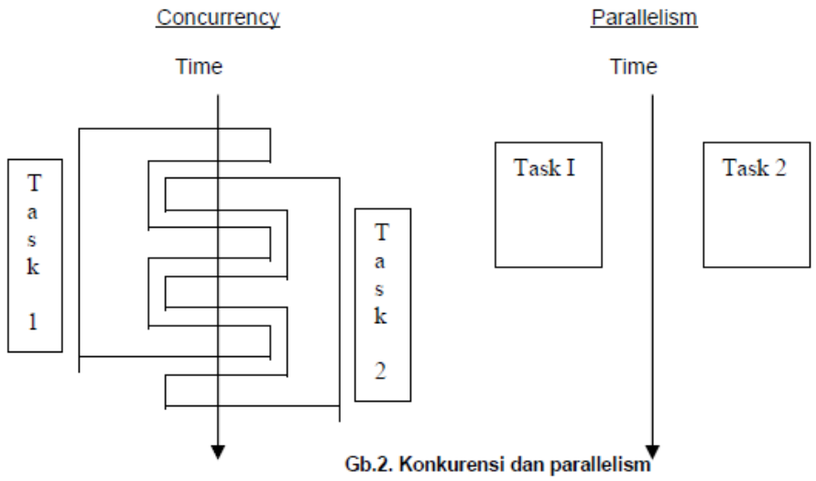
Gb.1. Dua thread dalam satu program

Dua program yang dijalankan secara terpisah (dari command line secara terpisah), berada pada dua address space yang terpisah. Sebaliknya, kedua thread pada gambar diatas berada pada address space yang sama (address space dari program dimana kedua thread tersebut dijalankan).

Kalau program itu berjalan diatas mesin dengan single processor, maka thread-thread itu dijalankan secara Konkuren (dengan mengeksekusi secara bergantian dari satu thread ke thread yang lainnya). Jika program itu berjalan diatas mesin dengan multiple processor, maka thread-thread itu bisa dijalankan secara paralel (masing-masing thread berjalan di processor yang terpisah).

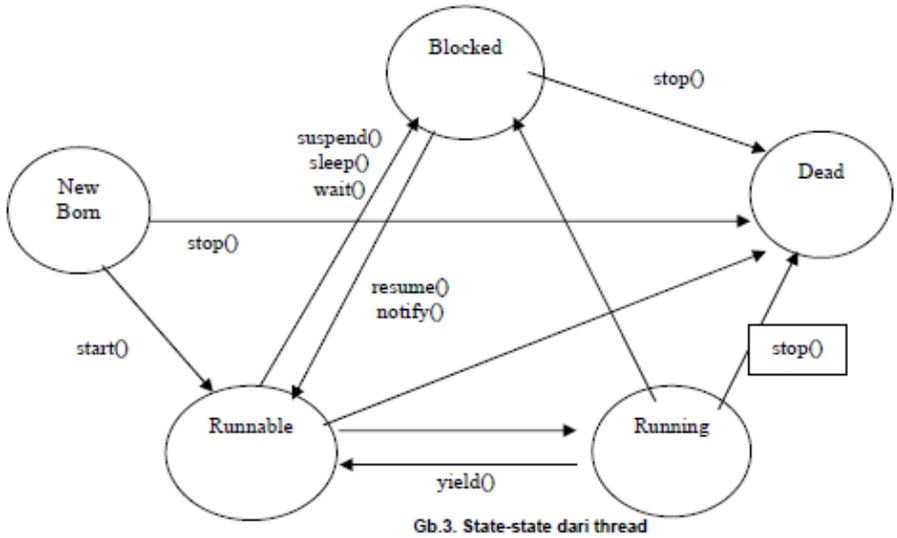
Gambar 2 dapat menjelaskan perbedaan antara konkurensi dan parallelism. Bahasa Java mempunyai kemampuan multithreading built-in, pada Java Virtual Macjine terdapat thread scheduler yang menentukan thread mana yang beraksi pada selang waktu tertentu. Scheduler pada JVM mendukung preemptive multithreading, yaitu suatu thread dengan prioritas tinggi dapat menyeruak masuk dan menginterupsi thread yang sedang beraksi, kemampuan ini sangat menguntungkan dalam membuat aplikasi real-time.

Scheduler pada JVM juga mendukung non-preemptive multithreading (atau sering disebut juga cooperative multithreading), yaitu thread yang sedang beraksi tidak dapat diinterupsi, ia akan menguasai waktu CPU, sampai ia menyelesaikan tugasnya atau secara eksplisit merelakan diri untuk berhenti dan memberi kesempatan bagi thread yang lain.



Gb.2. Konkurensi dan parallelism

Daur Hidup sebuah Thread



Gb.3. State-state dari thread

Newborn

Sebuah thread berada pada state ini ketika dia di instantiasi. Sebuah ruangan dimemori telah dialokasikan untuk thread itu,dan telah menyelesaikan tahap inisialisasinya.

```
……
Thread timerThread = new TimerThread();
……..
```

Pada state ini, timeThread belum masuk dalam skema penjadwalan thread scheduler.

Runnable

Pada state ini, sebuah thread berada dalam skema penjadwalan, akan tetapi dia tidak sedang beraksi. Kita bisa membuat timerThread yang kita buat sebelumnya masuk ke state runnable dengan :

```
…..
timerThread.start();
……
```

Kapan tepatnya timerThread beraksi, ditentukan oleh thread scheduler.

Running

Pada state ini, thread sedang beraksi. Jatah waktu beraksi bagi thread ini ditentukan oleh thread scheduler. Pada kasus tertentu, thread scheduler berhak meng-interupsi kegiatan dari thread yang sedang beraksi (misalnya ada thread lainnya dengan prioritas yang lebih tinggi).

Thread dalam keadaan running bisa juga lengser secara sukarela, dan masuk kembali ke state runnable, sehingga thread lain yang sedang menunggu giliran(runnable) memperoleh kesempatan untuk beraksi. Tinddakan thread yang lengser secara sukarela itu biasanya disebut yield-ing.

```
public void run() {
…….
Thread.yield();
…….
}
```

Blocked

Pada tahap ini thread sedang tidak beraksi dan diabaikan dalam penjadwalan thread scheduler. Thread yang sedang terblok menunggu sampai syarat-syarat tertentu terpenuhi, sebelum ia kembali masuk kedalam skema penjadwalan thread scheduler (masuk state runnable lagi). Suatu thread menjadi terblok karena hal-hal berikut :

a. Thread itu tidur untuk jangka waktu tertentu, seperti berikut :

```
public void run() {
…… try {
thread.slepp(3000); //thread yg sedang beraksi akan tidur selama 3000 milisecond=3menit
}
catch (InterruptedException e) {……. }
```

b. Thread itu di- suspend(). Thread yang ter-suspend() itu bisa masuk kembali ke state runnable bila ia resume(). seperti hal berikut :

```
……
//timerThread akan segera memasuki state blocked
timerThread.suspend();
……….
timerThread.resume();
//timerThread kembali masuk state runnable
……
```

c. Bila thread tersebut memanggil method wait() dari suatu object yang sedang ia kunci. Thread tersebut bisa kembali memasuki state runnable bila ada thread lain yang memanggil method notify() atau notifyAll() dari object tersebut.

d. Bila thread ini menunggu selesainya aktifitas yang berhubungan dengan I/O. Misalnya, jika suatu thread menunggu datangnya bytes dari jaringan komputer maka secara otomatis thread tersebut masuk ke state blocked.

e. Bila suatu thread mencoba mengakses critical section dari suatu object yang sedang dikunci oleh thread lain. Critical section adalah method/blok kode yang ditandai dengan kata synchronized.

Dead

Suatu thread secara otomatis disebut mati bila method run() – nya sudah dituntaskan (return dari method run()). Contoh dibawah ini adalah thread yang akan mengecap state running hanya sekali saat thread scheduler memberinya kesempatan untuk running, ia akan mencetak “ I’m doing something….something stupid….but I’m proud of It”… kemudian mati.

```
public class MyThread extends Thread {
…..
public void run() {
System.out.print(“I’m doing something…”);
System.out.print(“something stupid…”);
System.out.println(“but I’m proud of It…”);
// MyThread akan mati begitu baris diatas selesai dieksekusi
} …… }
```

contoh :

```
1. class theThread implements Runnable{
2. Thread t;
3.
4. theThread(){ t = new Thread(this, "DemoThread");
5. System.out.println("Child Thead : " + t);
6. t.start(); }
7.
8. public void run(){
9. try{ for(int i=5;i>0;i--){
10. System.out.println("Child Thread :" + i);
11. Thread.sleep(500); }
12. }
13. catch(InterruptedException e){
14. System.out.println("Child Interrupted"); }
15. System.out.println("Exiting Child Thread"); }
16. }
17. class MyThread{
18. public static void main(String args[]){
19. new theThread();
20. try{ for(int i=5;i>0;i--){
21. System.out.println("Main Thread :" + i);
22. Thread.sleep(1000); }
23. }
24. catch(InterruptedException e){
25. System.out.println("main Interrupted"); }
26. System.out.println("Exiting Main Thread"); }
27. }
```

Output :

```
Child Thread : Thread(Demo Thread, 5, main)
Main Thread : 5
Child Thread : 5
Child Thread : 4
Main Thread : 4
Child Thread : 3
Child Thread : 2
Main Thread : 3
Child Thread : 1
Exiting Child Thread
Main Thread : 2
Main Thread : 1
Exiting Main Thread
```