Network Programming 2010

Pemrograman Multi-Thread

Husni
husni@if.trunojoyo.ac.id
Husni.trunojoyo.ac.id
Komputasi.wordpress.com

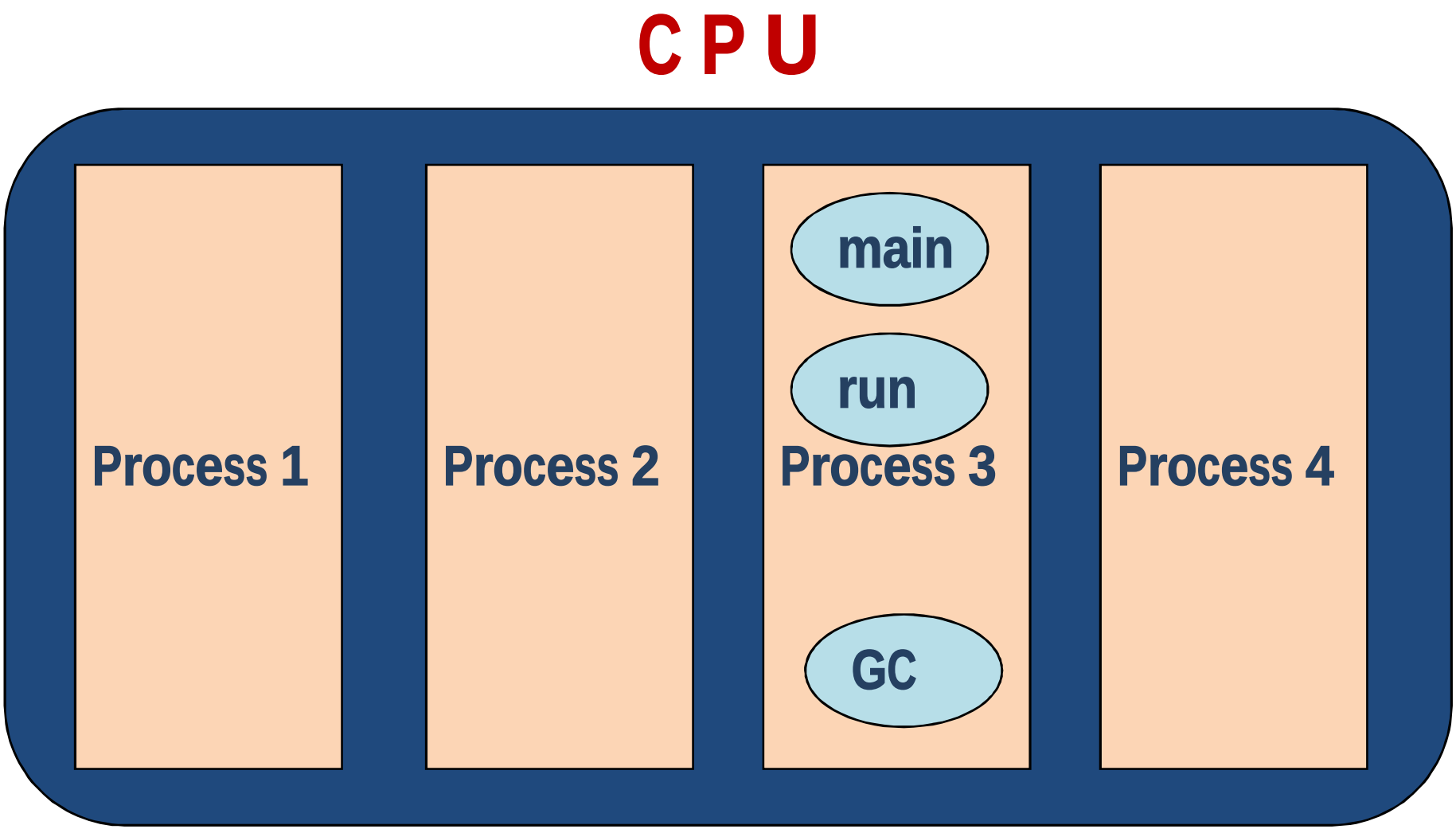
MultiThreading Menurut *Free Online Dictionary of Computing* (**FOLDOC**)

Berbagi-pakai satu Processor antara banyak *task* (atau "thread") dalam suatu cara yang dirancang untuk meminimalkan waktu yang diperlukan untuk berganti task. Ini dibangun dengan berbagi-pakai sebanyak mungkin lingkungan eksekusi program antara task-task berbeda sehingga sangat sedikit status yang perlu disimpan dan direstore ketika task-task tersebut berubah.

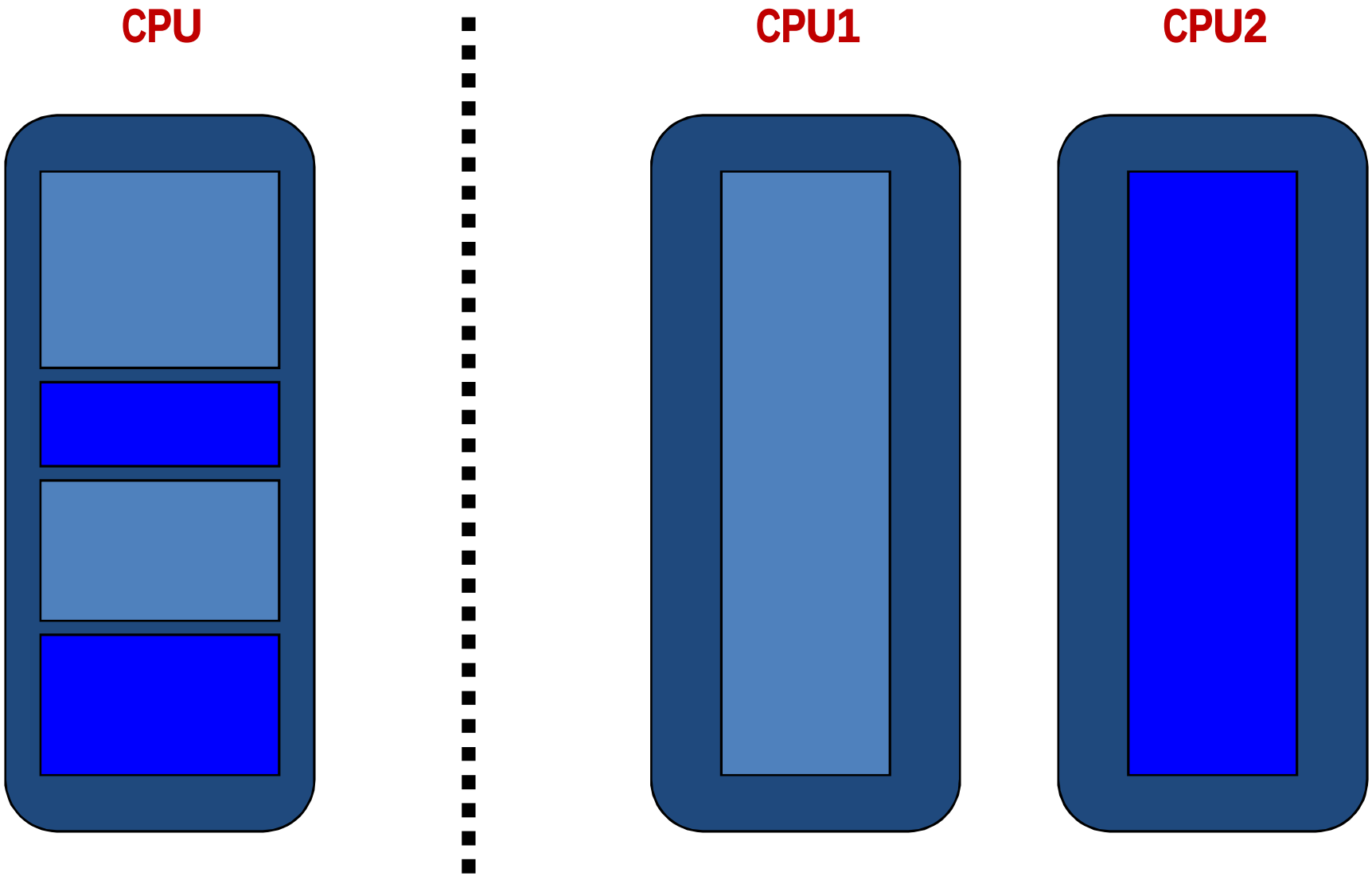
Outline

- Multithreading
- Pembuatan Thread
  - Menurunkan kelas Thread
  - Mengimplementasikan Interface Runnable
- Metode dalam Thread
- Thread Pool
- Sinkronisasi
- Komunikasi Antar Thread
- Deadlock

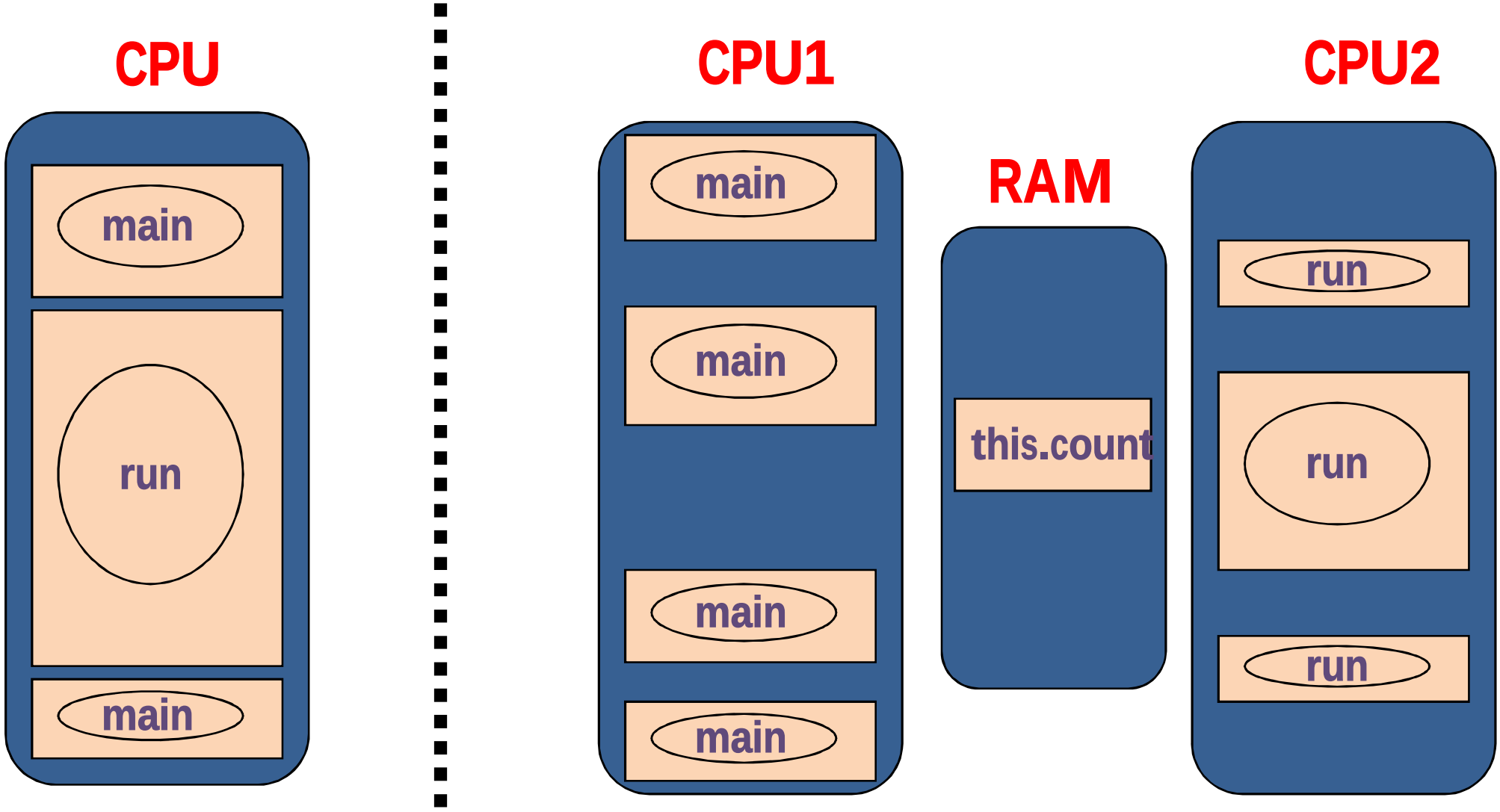
Proses & Thread



Concurrency & Paralellism



Concurrency & Paralellism



Thread

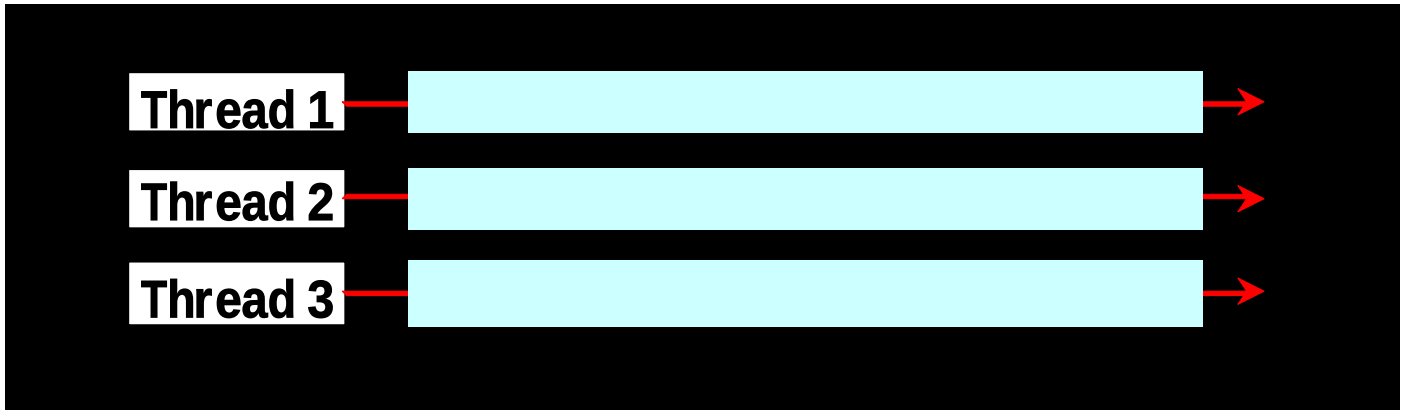
- Thread adalah aliran (*stream*) tunggal dari eksekusi dalam suatu proses.
- Process adalah program yang berjalan dalam ruang alamatnya sendiri.
- Semua program saat ini terdiri dari thread.
- Kontrol atau eksekusi utama dari semua program (sejauh ini) dikendalikan oleh suatu thread tunggal.
- Akan kita lihat: mutlithreading atau ada banyak thread berjalan dalam program yang sama.

MultiTasking & Multithreading

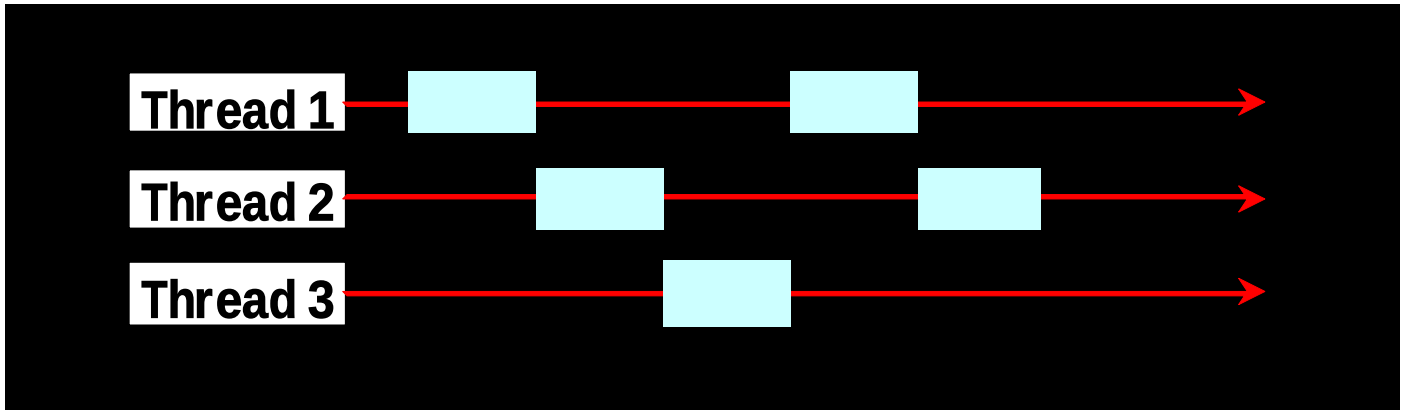
- Pengguna lebih akrab dengan multitasking.
- Multitasking:
  - Ada lebih dari satu program bekerja (seolah) pada waktu yang sama.
  - SO menjatahkan CPU ke program-program berbeda dengan suatu cara untuk memberikan pengaruh dari concurrency.
  - Ada dua jenis multitasking - preemptive dan cooperative.
- Multithreading:
  - Perluasan ide multitasking dengan membolehkan program mempunyai banyak task.
  - Setiap task di dalam program dinamakan thread.

Multi-Thread & Multi-Processor

Banyak thread pada banyak processor

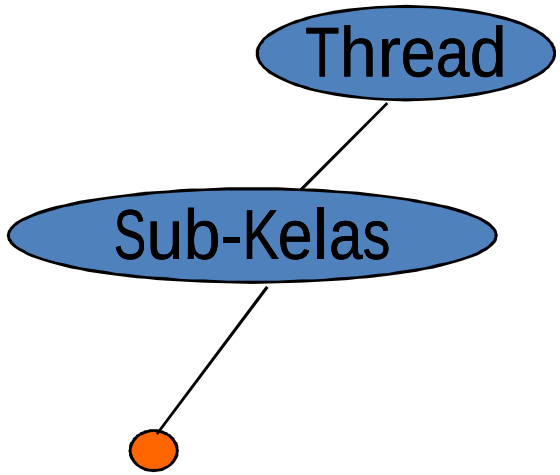


Banyak thread pada satu processor



Java & Multi-Thread

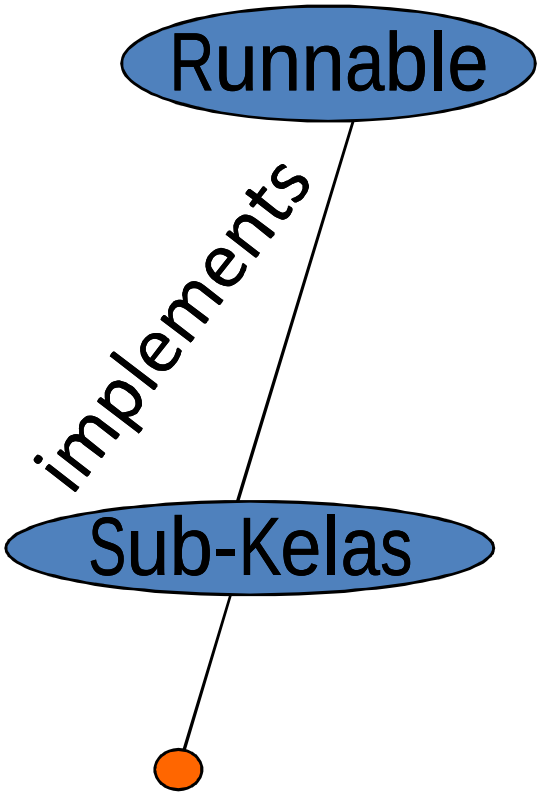
- Fitur multithreading pada java sudah *built-in*.
- Java menyediakan kelas Thread untuk menangani thread-thread dalam program.
- Ada dua cara membuat obyek Thread.
  - Membuat obyek dari sub-kelas dari kelas Thread
  - Meng-implements interface Runnable pada suatu obyek




```
class ThreadX extends  Thread {
        public void run() {
        //apa yang dikerjakan thread
        }
}
```

```
ThreadX tx = new ThreadX( );
tx.start( );
```

Implements Interface Runnable




```
class RunnableY implements  Runnable {
        public void run() {
        //apa yang dikerjakan thread
        }
}
```

```
RunnableY ry = new RunnableY();
Thread ty = new Thread(ry);
ty.start();
```

Kelas Thread

- Kelas Thread bagian dari *package* java.lang.
- Menggunakan obyek dari kelas ini, thread dapat di**stop**, di**pause** dan di**resume**
- Ada banyak *constructor* & metode pada kelas ini, di antaranya:
  - **Thread(String n)** – membuat Thread baru dengan nama n.
  - **Thread(Runnable target)** – membuat obyek Thread baru
  - **Thread(Threadgroup group, Runnable target)**

    Membuat obyek Thread baru dalam Threadgroup.

Metode Dalam Kelas Thread

Metode Statis:

- activeCount();
- currentThread();
- sleep();
- yield();

Metode Instance:

- getPriority( );
- setPriority( );
- start( );
- stop( );
- run( );
- isAlive( );
- suspend( );
- resume( );
- join( );

Diagram Status Thread

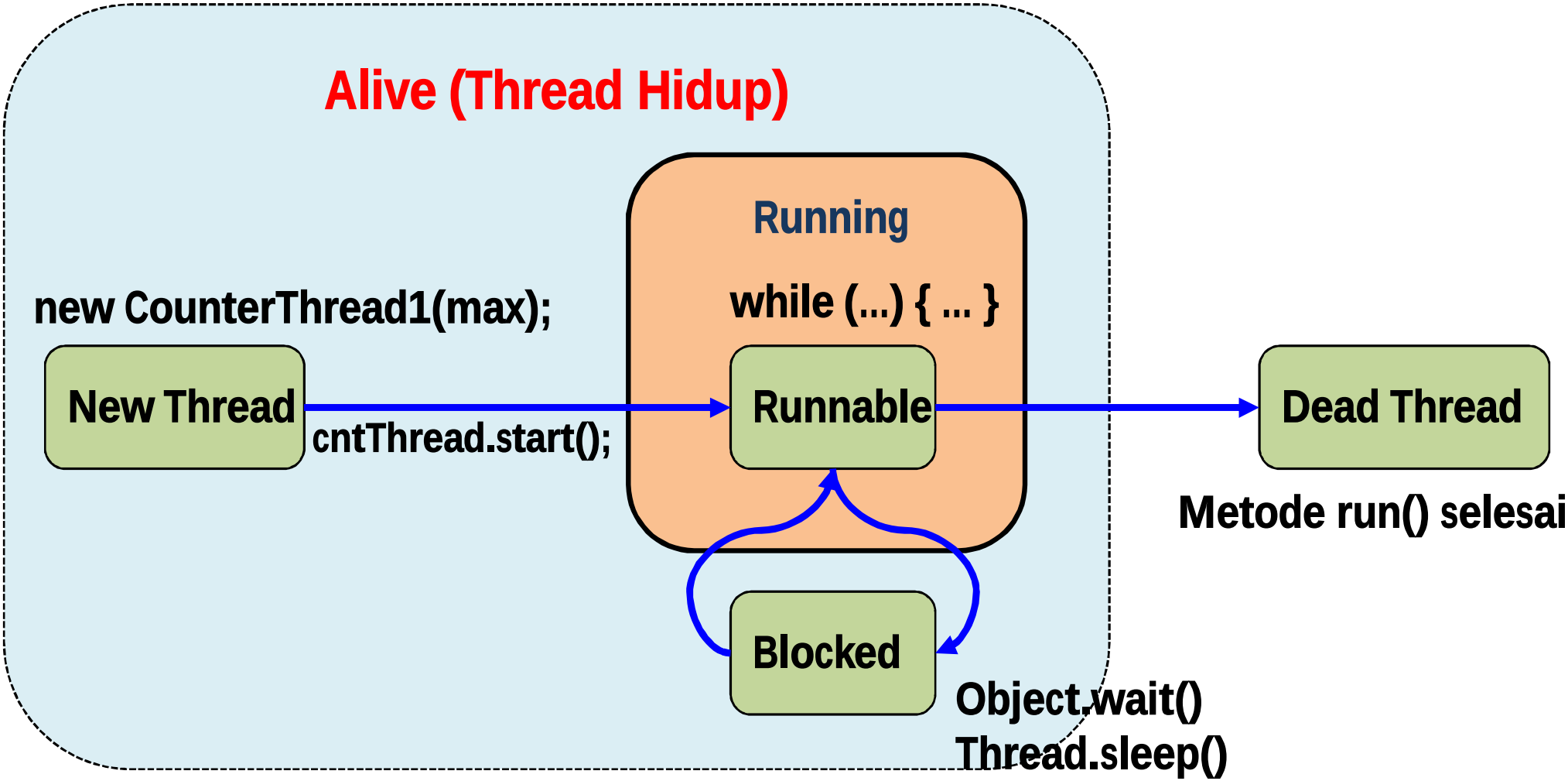
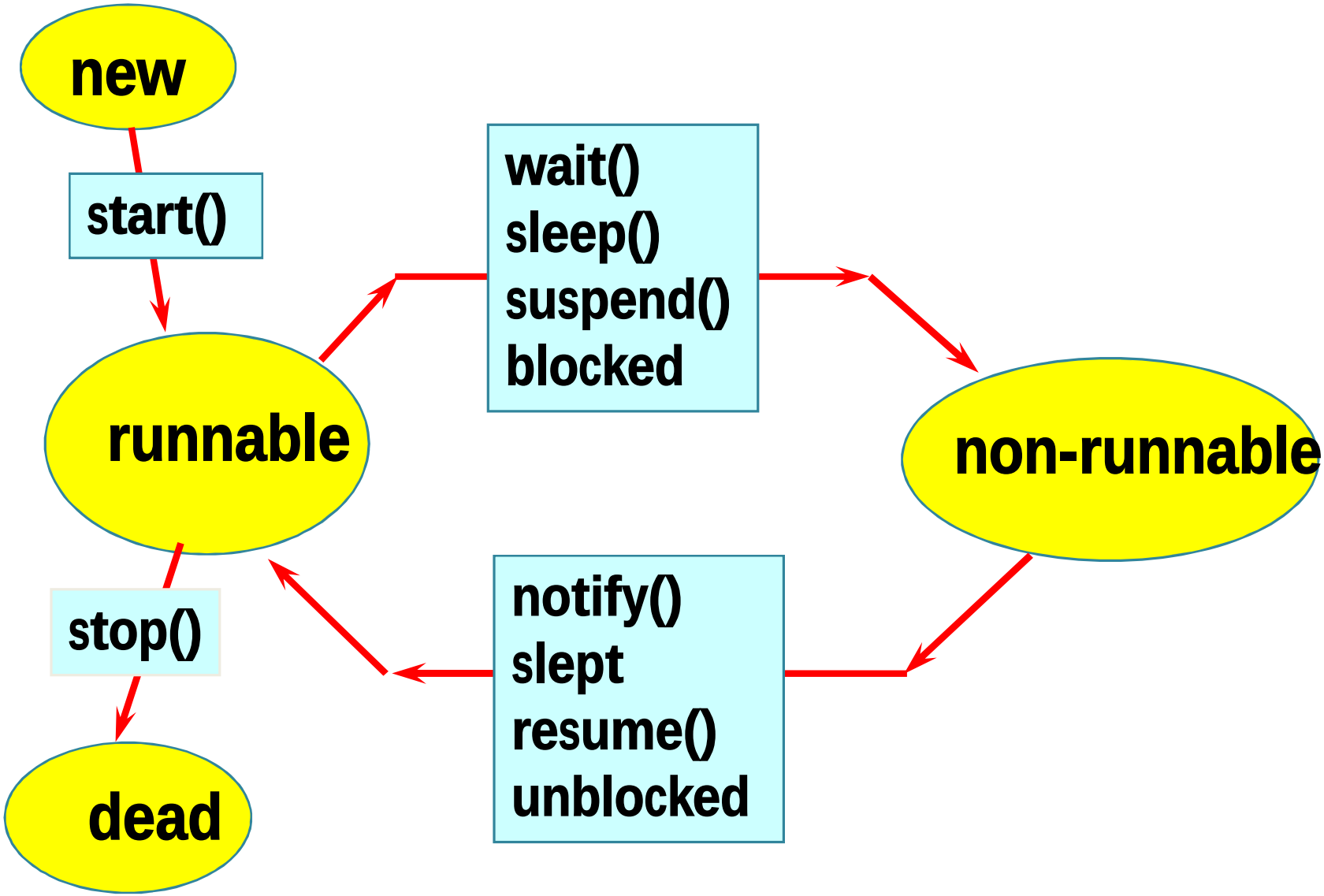
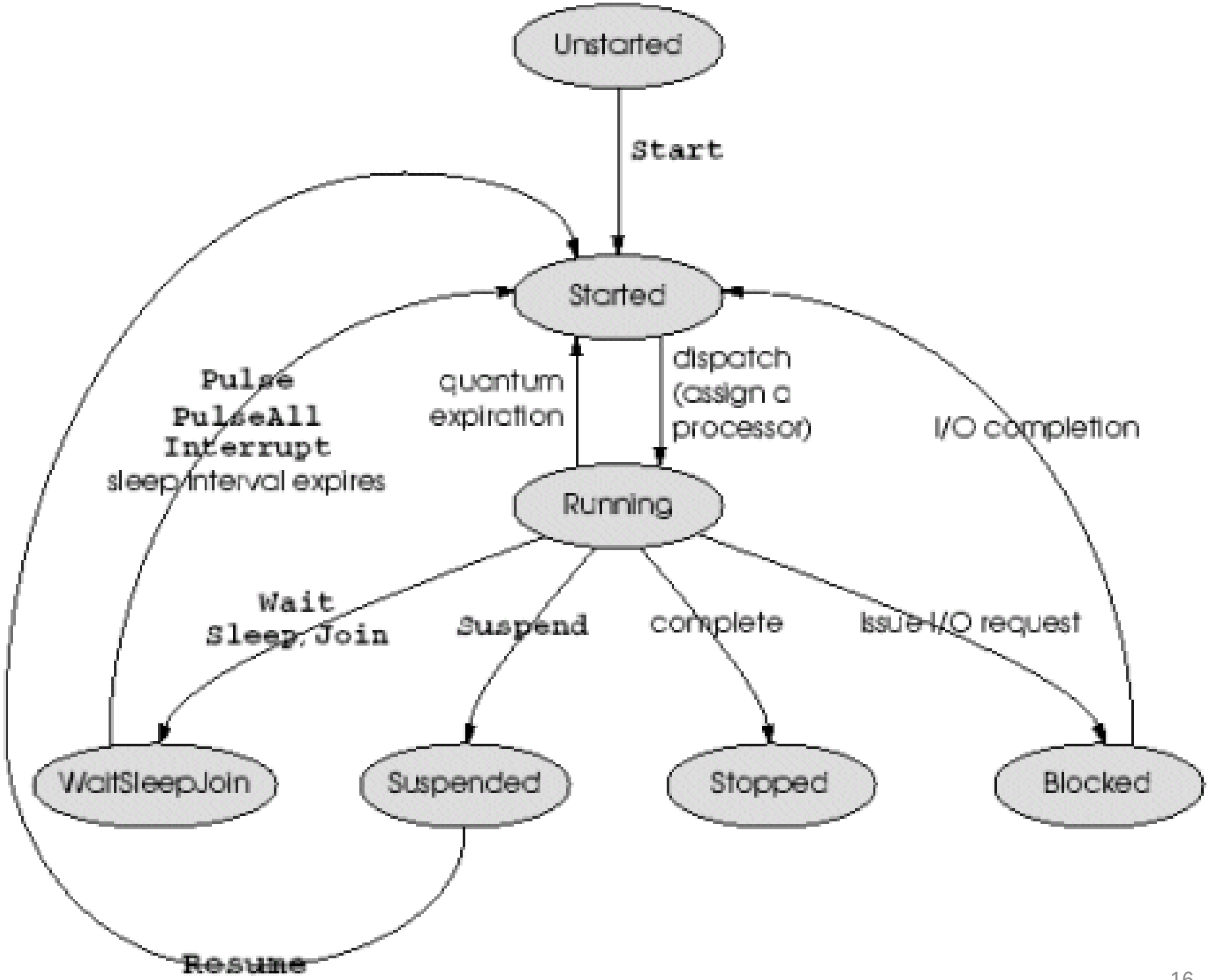
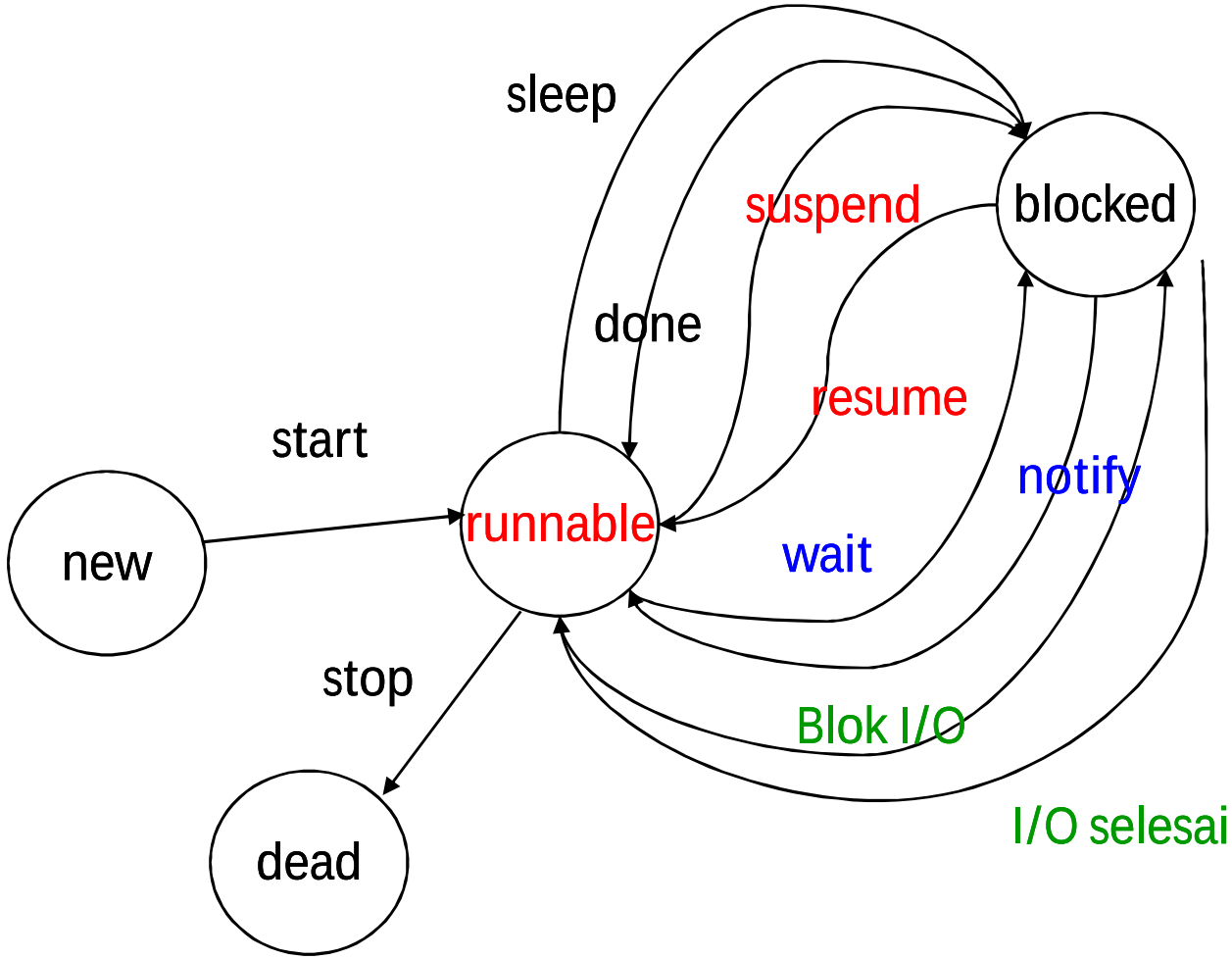


Diagram Status Thread




Unstarted
Start
Started
Pulse
PulseAll
Interrupt
sleep interval expires
quantum
expiration
dispatch
(assign a
processor)
I/O completion
Running
Wait
Sleep, Join
Suspend
complete
issue I/O request
WaitSleepJoin
Suspended
Stopped
Blocked
Resume

Properti Thread



Membuat Thread

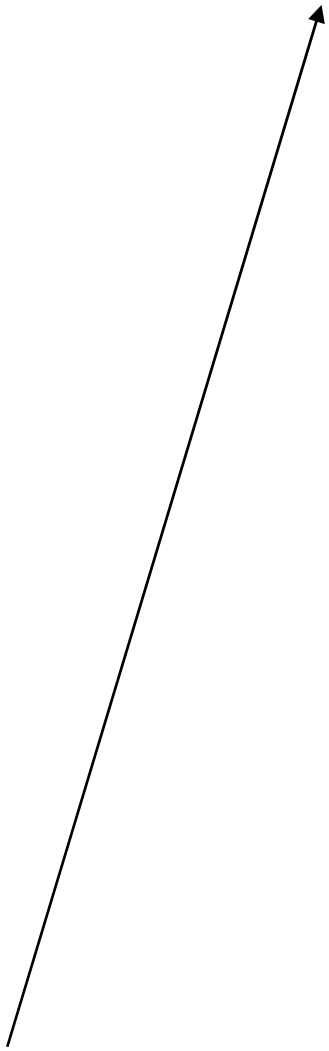
- Membuat obyek dari sub-kelas Thread. Berarti buat sub-kelas tersebut lebih dulu.
- Contoh ini membuat 5 obyek thread bernama SimpleThread. Perhatikan hasilnya! Mengapa?

```
public class SimpleThread extends Thread {
    private int countDown = 3;
    private static int threadCount = 0;
    private int threadNumber = ++threadCount;

    public SimpleThread( ) {
        System.out.println("Membuat " + threadNumber++);
    }
```

```
    public void run( ) {
        while(true) {
            System.out.println("Thread " + threadNumber +
                                    " (" + countDown + ") berjalan");
            if (--countDown == 0) return;
        }
    }

    public static void main(String[] args)  {
        for (int i = 0; i < 5; i++)
        new SimpleThread( ).start( );
        System.out.println("Semua Thread dimulai...");
    }
}
```

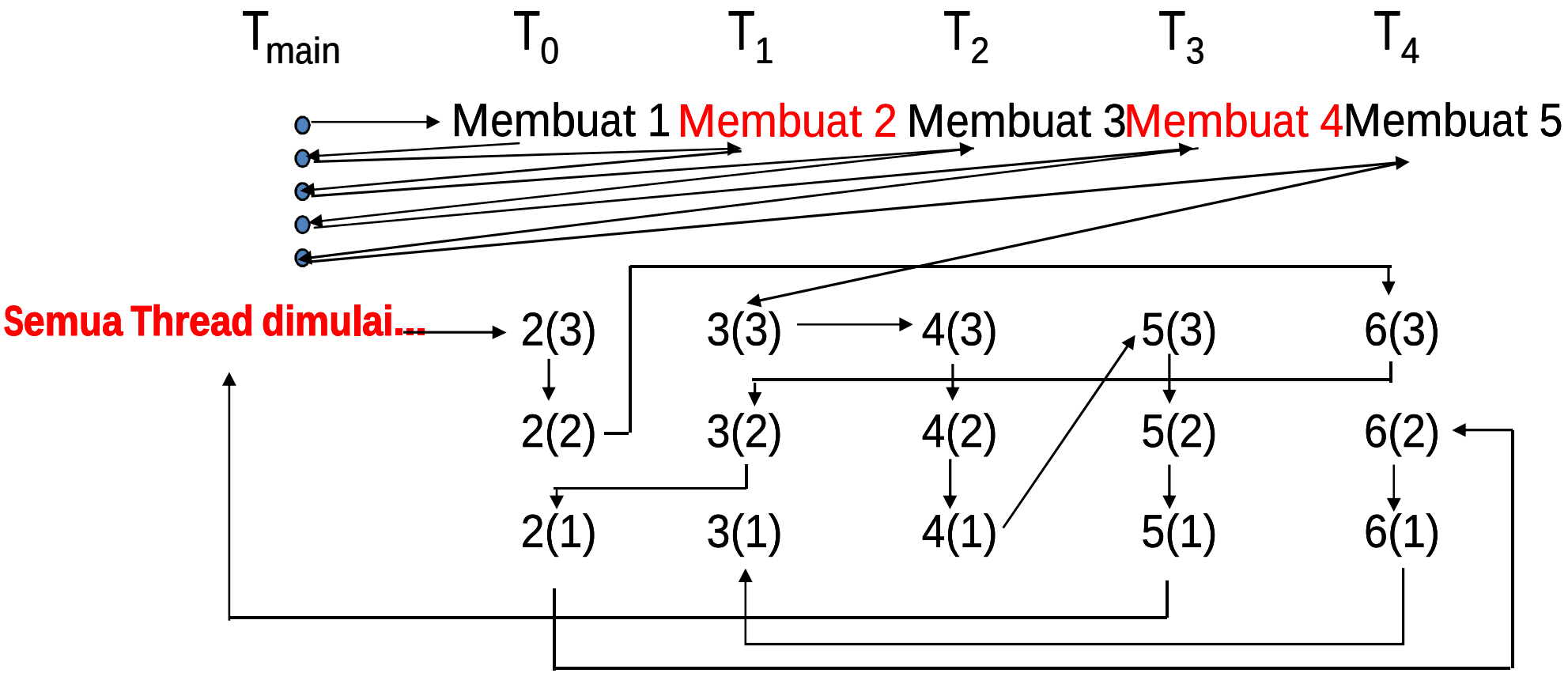
Output: Kemungkinan

Membuat 1
Membuat 2
Membuat 3
Thread 2 (3) berjalan
Thread 2 (2) berjalan
Thread 2 (1) berjalan
Membuat 4
Thread 4 (3) berjalan
Thread 4 (2) berjalan
Thread 4 (1) berjalan
Membuat 5



Semua Thread dimulai...
Thread 3 (3) berjalan
Thread 3 (2) berjalan
Thread 3 (1) berjalan
Thread 5 (3) berjalan
Thread 5 (2) berjalan
Thread 6 (3) berjalan
Thread 5 (1) berjalan
Thread 6 (2) berjalan
Thread 6 (1) berjalan

Output: Kemungkinan



Menggunakan Interface `Runnable`

- Tujuan: Membuat dan menjalankan 3 thread:
  - Thread pertama mencetak huruf ‘a’ 100 kali.
  - Thread kedua mencetak huruf ‘b’ 100 kali.
  - Thread ketiga mencetak bilangan 1 s.d 100.

```
public class TaskThreadDemo {
    public static void main(String[] args) {

        // Membuat task
        Runnable printA = new PrintChar('a', 100);
        Runnable printB = new PrintChar('b', 100);
        Runnable print100 = new PrintNum(100);

        // Membuat thread
        Thread thread1 = new Thread(printA);
        Thread thread2 = new Thread(printB);
        Thread thread3 = new Thread(print100);

        // Jalankan thread
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

```
// Tugas mencetak karakter tertentu sebanyak tertentu
class PrintChar implements Runnable {

    private char charToPrint; // karakter yang dicetak
    private int times; // banyaknya perulangan

    // constructor: karakter yang akan dicetak dan jumlahnya
    public PrintChar(char c, int t) {
        charToPrint = c;
        times = t;
    }

    // Override metode run(): apa yang dikerjakan dalam thread?
    public void run() {
        for (int i = 0; i < times; i++) {
            System.out.print(charToPrint);
        }
    }
}
```

```
// Tugas mencetak bilangan 1 s.d  n
class PrintNum implements Runnable {
    private int lastNum;

    //constructor mencetak 1, 2, ... i */
    public PrintNum(int n) {
        lastNum = n;
    }

    //apa yang dikerjakan dalam thread?
    public void run() {
        for (int i = 1; i <= lastNum; i++) {
            System.out.print(" " + i);
        }
    }
}
```

Output: Kemungkinan

- abbbbbbbbbbb 1 2 3 4 5 6aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa 7bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
- 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
- 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

Metode Statis yield()

Metode ini digunakan untuk melepas waktu (sementara) untuk thread lain. Contoh: kita mengubah isi metode run() dalam kelas PrintNum menjadi:

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    Thread.yield();
  }
}
```

Setiap kali suatu bilangan dicetak, thread print100 mengalah. Akibatnya, bilangan dicetak setelah karakter.

Output: Kemungkinan

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaab
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbb 2bbbbbbbbbbbbbbbbbbbbbbb
3b 4 5b 6b 7bbbbbbbbb 8b 9b 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32b 33 34 35
36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53
54 55 56 57 58 59 60 61 62 63 64 65 66 67

68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85
86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

Metode Statis sleep()

Metode sleep(long milidetik) menidurkan thread selama waktu tertentu. Contoh: tambahkan teks merak berikut ke dalam metode run() sebelumnya:

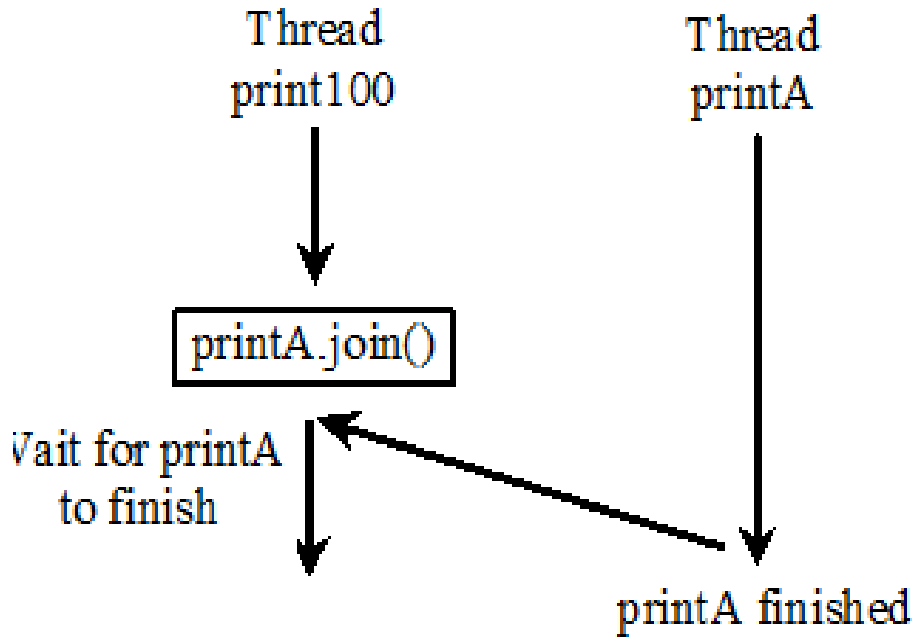
```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    try {
       if (i >= 50) Thread.sleep(1);
     }
     catch (InterruptedException ex) {
     }
  }
}
```

Setiap kali bilangan (>= 50) dicetak, thread print100 tidur selama 1 mili detik.

Metode Join()

- Metode ini dapat digunakan untuk memaksa satu thread menunggu thread lain selesai.

```
public void run() {
  Thread thread4 = new Thread(
    new PrintChar('c', 40));
  thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {
  }
```



- Bilangan setelah 50 dicetak setelah thread printA selesai.

Metode isAlive(), interrupt(), isInterrupted()

Metode **isAlive**() digunakan untuk mendapatkan starus dari suatu thread. Mengembalikan true jika thread berstatus Ready, Blocked atau Running; false jika thread masih baru, belum dimulai atau telah selesai.

Metode **interrupt**() menyela suatu thread dengan cara berikut: Jika suatu thread dalam status Ready atau Running, flag interrupted –nya diset; jika statusnya blocked, dibangunkan dan masuk ke status Ready, dan java.io.InterruptedException dibuat.

Metode **isInterrupt**() memeriksa apakah thread diinterupsi.

Metode stop(), suspend(), resume()

- Kelas Thread punya metode stop(), suspend() dan resume(). Pada Java 2, metode ini dihapus.
- Berikan nilai **null** ke suatu variabel Thread untuk meminta thread tersebut berhenti daripada menggunakan metode stop().

Prioritas Thread

- Setiap thread diberikan prioritas default **`Thread.NORM_PRIORITY`**. Prioritas dapat direset menggunakan metode `setPriority(int priority)`.

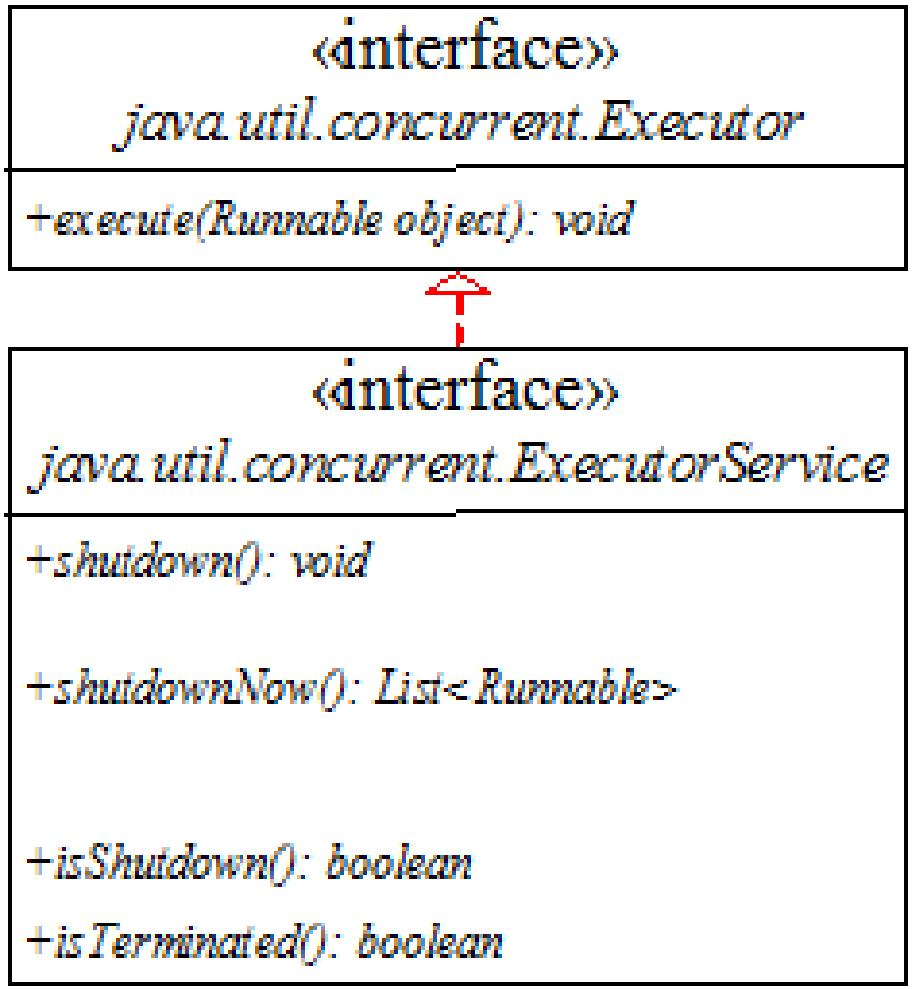
- Beberapa konstanta untuk prioritas:

```
Thread.MIN_PRIORITY
Thread.MAX_PRIORITY
Thread.NORM_PRIORITY
```

Thread Pool

- Memulai suatu thread baru untuk setiap task dapat membatasi *throughput* dan menurunkan kinerja.
- Thread pool sangat cocok untuk mengelola sejumlah task yang berjalan secara konkuren.
- JDK 1.5 menggunakan interface **Executor** untuk mengeksekusi task-task dalam suatu thread pool dan interface **ExecutorService** untuk mengelola dan mengontrol task-task.
- ExecutorService merupakan sub-interface dari Executor.

Interface Executor & ExecutorService



Executes the runnable task.

Shuts down the executor, but allows the tasks in the executor to complete. Once shutdown, it cannot accept new tasks.

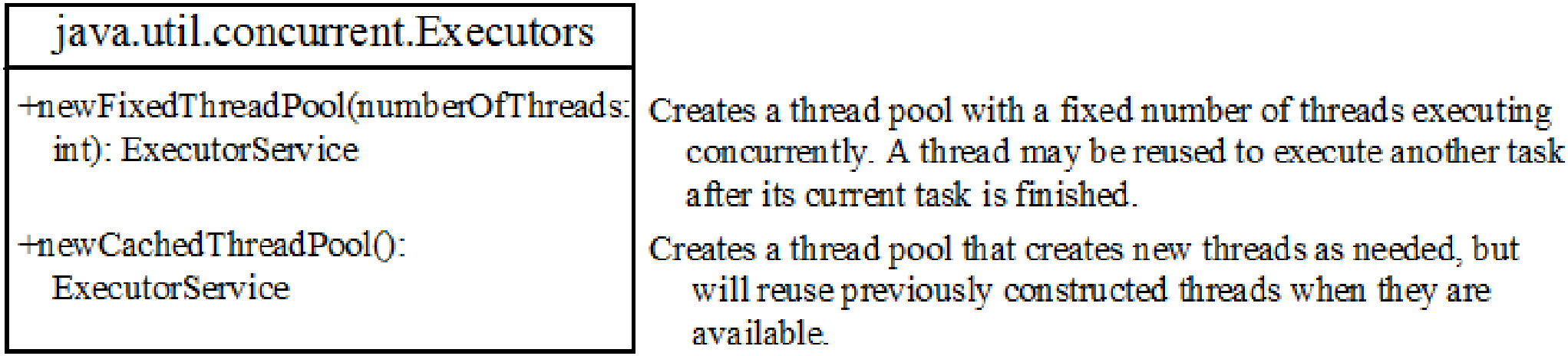
Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.

Returns true if the executor has been shutdown.

Returns true if all tasks in the pool are terminated.

Pembuatan Executor

- Gunakan metode yang statis di dalam kelas Executors.



```
import java.util.concurrent.*;

public class ExecutorDemo {
  public static void main(String[] args) {

    // buat thread pool tetap, maksimum 3 thread
    ExecutorService executor = Executors.newFixedThreadPool(3);

    // Kirim task runnable ke executor
    executor.execute(new PrintChar('a', 100));
    executor.execute(new PrintChar('b', 100));
    executor.execute(new PrintNum(100));

    // Matikan executor
    executor.shutdown();
  }
}
```

Thread Pool

- Jika baris ke-6 diganti menjadi:

  ExecutorService executor = Executors.newFixedThreadPool(1);
- Apa yang terjadi? Tuliskan kodenya dan jalankan!

- Bagaimana diganti menjadi:

  ExecutorService executor = Executors.newCachedThreadPool();

Sinkronisasi

- Sinkronisasi merupakan mekanisme untuk mengontrol eksekusi dari thread-thread berbeda sehingga ketika banyak thread mengakses variabel “shared”, dapat dipastikan eksekusinya benar.
- Java punya keyword **synchronized** – dapat digunakan untuk memperkenalkan suatu segmen kode atau metode yang akan dapat diakses hanya oleh suatu thread tunggal pada satu waktu.
- Sebelum memasuki wilayah sinkronisasi, suatu thread akan memperoleh **semaphore** yang berasosiasi dengan wilayah tersebut – jika telah diambil oleh thread lain, maka thread tersebut mem-blokir (menunggu) sampai semaphore dilepas.

```
class Account {
    private int balance = 0;

    synchronized void deposit(int amount) {
        balance += amount;
     }

    int getBalance() {   return balance;   }
}

class Customer extends Thread {
    Account account;
    Customer(Account account) { this.account = account;  }
```

```
        public  void run() {
        try  {
           for (int i = 0; i < 10000; i++) { account.deposit(10); }
        }
        catch (Exception e) {  e.printStackTrace(); }
    }  /* metode run   */
} /* kelas Customer */

public class BankDemo {
    private final static int NUMCUSTOMER = 10;
    public static void main(String args[ ]) {
        //buat account
        Account account = new Account();
```

```
        //buat dan jalankan thread customer
        Customer customer[ ] = new
    Customer[NUMCUSTOMER];
        for (int i = 0; i < NUMCUSTOMER; i++) {
          customer[i] = new Customer(account);
          customer[i].start( );
        }

        //menunggu thread customer selesai
        for (int i = 0; i < NUMCUSTOMER; i++) {
              try {  customer[i].join( );  }
              catch (InterruptedException e) { e.printStackTrace( ); }
        }
        //menampilkan saldo (balance) account
        System.out.println(account.getBalance( ) );
    }
}
```

Sinkronisasi

- Dalam Java, obyek dengan satu atau lebih metode **synchronized** disebut sebagai monitor.
- Ketika thread memanggil metode synchronized, hanya satu thread yang dibolehkan pada satu waktu, lainnya menunggu dalam antrian (*queue*).
- Dalam aplikasi jenis *producer- consumer*, thread consumer mungkin mendapatkan tidak ada cukup elemen untuk dikonsumsi.
- Tanggungjawab **monitor** untuk memastikan bahwa thread-thread yang menunggu producer **diberitahu** segera setelah elemen-elemen diproduksi.

Komunikasi Thread

- Suatu thread dapat **melepaskan lock** (kunci) sehingga thread lain berkesempatan untuk mengeksekusi metode ***synchronized***.
- Ini karena kelas **Object** mendefinisikan tiga metode yang memungkinkan **thread berkomunikasi** satu dengan lainnnya.
- Metode tersebut adalah **wait**(), **notify**() dan **notifyall**()

Metode Komunikasi Thread

- **void wait( )** – mengakibatkan thread tersebut menunggu sampai diberitahu – metode ini hanya dapat dipanggil di dalam metode *synchronized*.

  Bentuk lain dari wait() adalah
  - **void wait(long msec) throws InterruptedException**
  - **void wait(long msec, int nsec) throws InterruptedException**
- **void notify( )** – memberitahukan thread yang terpilih secara acak (*random*) yang sedang menunggu kunci pada obyek ini – hanya dapat dipanggil di dalam metode *synchronized*.
- **void notifyall( )** – memberitahukan semua thread yang sedang menunggu kunci pada obyek ini - hanya dapat dipanggil di dalam metode *synchronized*.

```
class Producer extends Thread {
    Queue queue;

    Producer (Queue queue) {
        this.queue = queue;
    }

    public void run {
        int i = 0;
        while(true) {   queue.add(i++);       }
    }
}
```

```
class Comsumer extends Thread {
    String str;
    Queue queue;

    Consumer (String str, Queue queue) {
        this.str = str;
        this.queue = queue;
    }

    public void run {
        while(true) {
          System.out.println(str + ": " + queue.remove());
        }
    }
}
```

```
class queue {
    private final static int SIZE = 10;
    int array[ ] = new int[SIZE];
    int r = 0;        int w = 0;            int count = 0;

    synchronized void add(int i) {
        //menunggu selama antrian (queue) penuh
        while (count == SIZE) {
          try {   wait( );}
          catch (InterruptedException ie) {
              ie.printStackTrace( );
              System.exit(0);
        }
    } /* metode add */
```

```
        //Tambahkan data ke array dan atur pointer write
        array[w++] =  i;
        if (w >= SIZE)    w = 0;

        ++count;      //Naikkan nilai count
        notifyAll( );  //Beritahu thread-thread yang menunggu
}

synchronized int remove( ) {
        //menunggu selama queue kosong
        while (count == 0) {
          try { wait( ); }
          catch (InterruptedException ie) {
              ie.printStackTrace( );
              System.exit(0);
          }
}
```

```
        //baca data dari array dan atur pointer read
        int element = array[r++];

        if (r >= SIZE) r = 0;

        --count;          //Turunkan count
        notifyAll( );     //Beritahu thread-thread yang menunggu
        return element;
    }
}

public ProducerConsumer {
    public static void main(String args[ ]) {
      Queue queue = new Queue( );
      new Producer(queue).start( );
      new Consumer("ConsumerA", queue).start( );
      new Consumer("ConsumerB", queue).start( );
      new Consumer("ConsumerC", queue).start( );
    }
}
```

Deadlock

- Deadlock adalah error yang dapat diselesaikan dengan multithread.
- Terjadi saat dua atau lebih thread saling menunggu selama tak terbatas untuk melepaskan kunci.
- Misal: thread-1 memegang kunci object-1 dan menunggu kunci dari object-2. Thread-2 memegang kunci object-2 dan menunggu kunci dari object-1.
- Tidak satu pun thread (ini) dapat diproses. Masing-masing selalu menunggu yang lain untuk melepaskan kunci yang dibutuhkannya.

Tugas

- Pelajari dengan baik konsep multithreading di atas.
- Perbaiki program **EchoServer** dan **EchoClient** sehingga server dapat menerima koneksi dari 5 client pada waktu yang bersamaan
- Tambahkan satu variabel di server, misalnya X yang hanya dapat diakses oleh satu client pada satu waktu. X mencatat pesan teks yang sebelumnya dikirim oleh client tertentu!