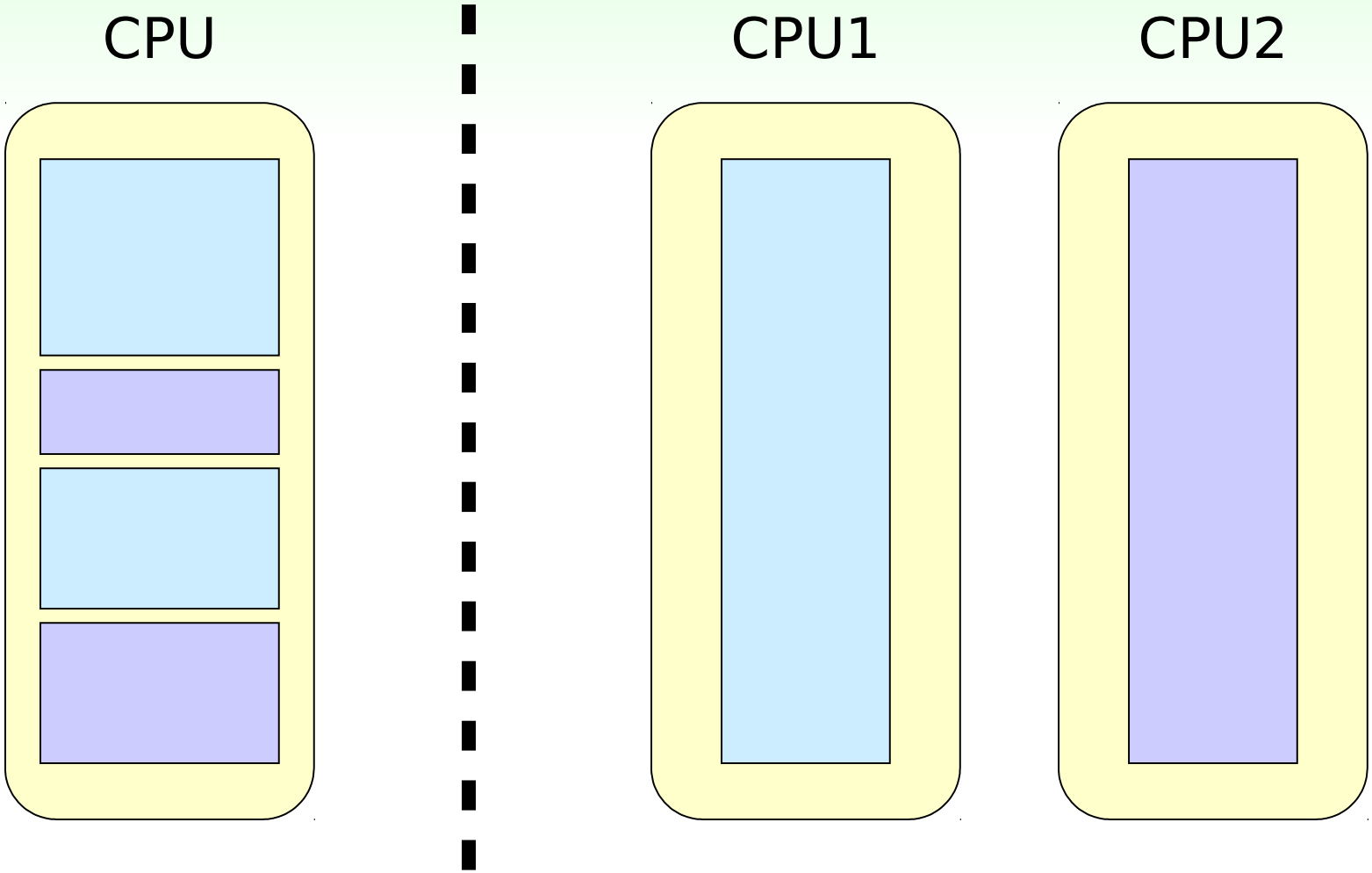
Java Threads

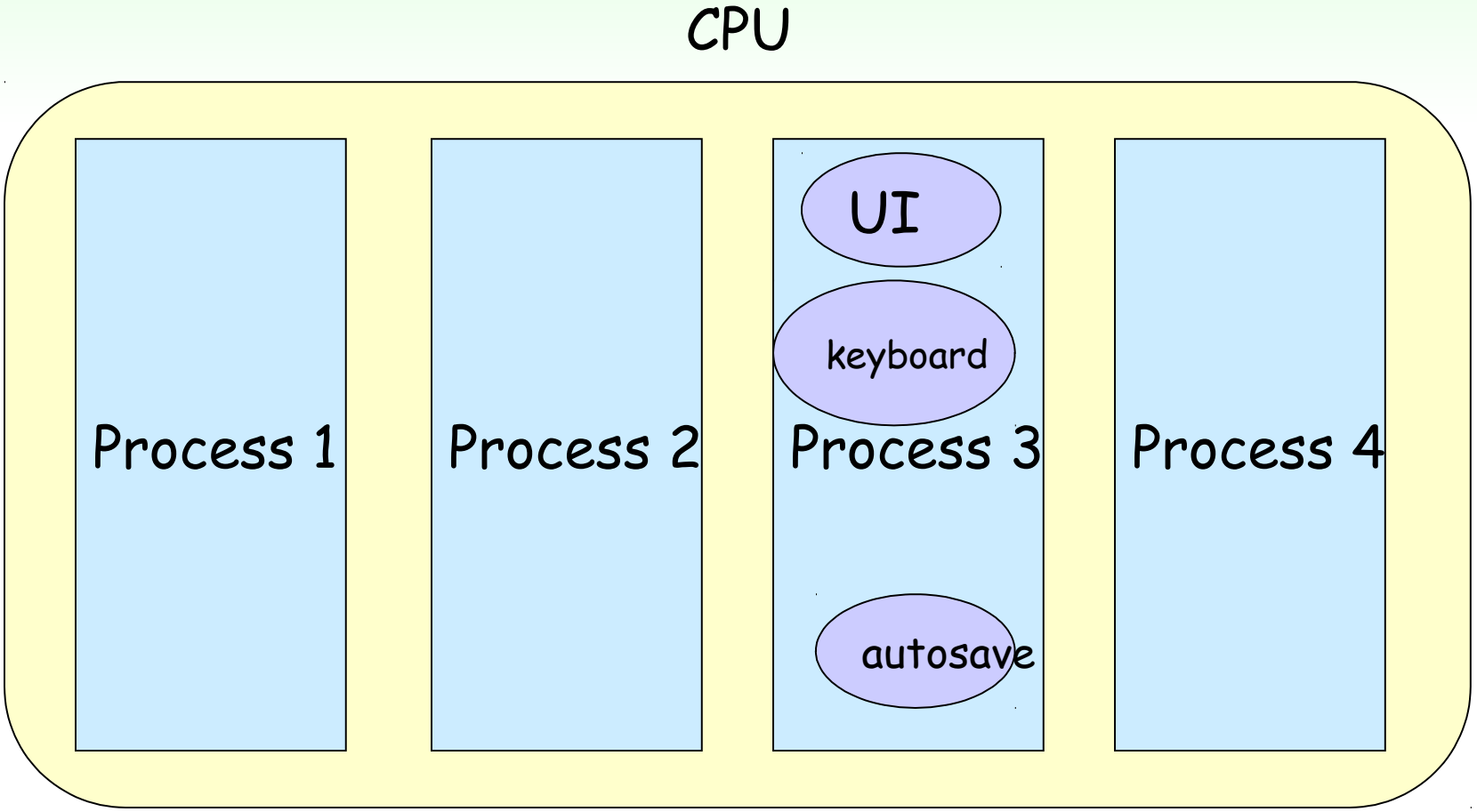
Multitasking and Multithreading

- Multitasking refers to a computer's ability to perform multiple jobs concurrently
  - more than one program are running concurrently, e.g., UNIX
- A thread is a single sequence of execution within a program
- Multithreading refers to multiple threads of control within a single program
  - each program can run multiple threads of control within it, e.g., Web Browser

Concurrency vs. Parallelism



Threads and Processes



What are Threads Good For?

- To maintain responsiveness of an application during a long running task.
- To enable cancellation of separable tasks.
- Some problems are intrinsically parallel.
- To monitor status of some resource (DB).
- Some APIs and systems demand it: Swing.

Application Thread

- When we execute an application:
  - The JVM creates a Thread object whose task is defined by the **main()** method
  - It starts the thread
  - The thread executes the statements of the program one by one until the method returns and the thread dies

Multiple Threads in an Application

- Each thread has its private run-time stack
- If two threads execute the same method, each will have its own copy of the local variables the methods uses
- However, all threads see the same dynamic memory (heap)
- Two different threads can act on the same object and same static fields concurrently

Thread Methods

void start()

- Creates a new thread and makes it runnable
- This method can be called only once

void run()

- The new thread begins its life inside this method

void stop() (deprecated)

- The thread is being terminated

Thread Methods

- **yield()**
  - Causes the currently executing thread object to temporarily pause and allow other threads to execute
  - Allow only threads of the same priority to run
- **sleep(int *m*)/sleep(int *m*,int *n*)**
  - The thread sleeps for *m* milliseconds, plus *n* nanoseconds

Creating Threads

- There are two ways to create our own **Thread** object
  1. Subclassing the **Thread** class and instantiating a new object of that class
  2. Implementing the **Runnable** interface
- In both cases the **run()** method should be implemented

Extending Thread

```
public class ThreadExample extends Thread {
   public void run () {
      for (int i = 1; i <= 100; i++) {
         System.out.println(“Thread: ” + i);
      }
   }
}
```

Implementing Runnable

```
public class RunnableExample implements Runnable {
    public void run () {
        for (int i = 1; i <= 100; i++) {
            System.out.println (“Runnable: ” + i);
        }
    }
}
```

A Runnable Object

- The Thread object's **run()** method calls the Runnable object's **run()** method

- Allows threads to run inside any object, regardless of inheritance

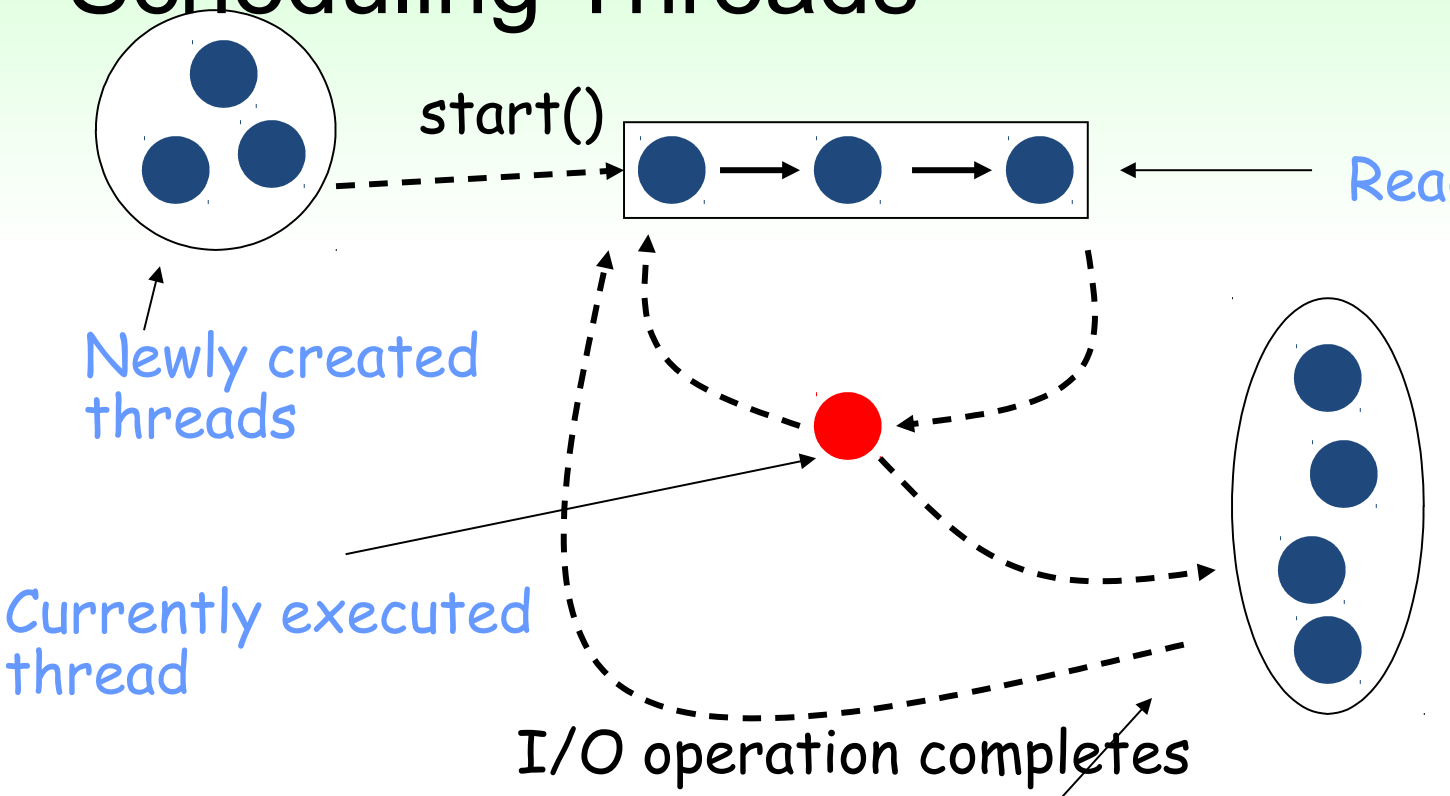
Example – a servlet that is also a thread

Starting the Threads

```
public class ThreadsStartExample {
        public static void main (String argv[]) {
                new ThreadExample ().start ();
                new Thread(new RunnableExample ()).start ();
        }
}
```

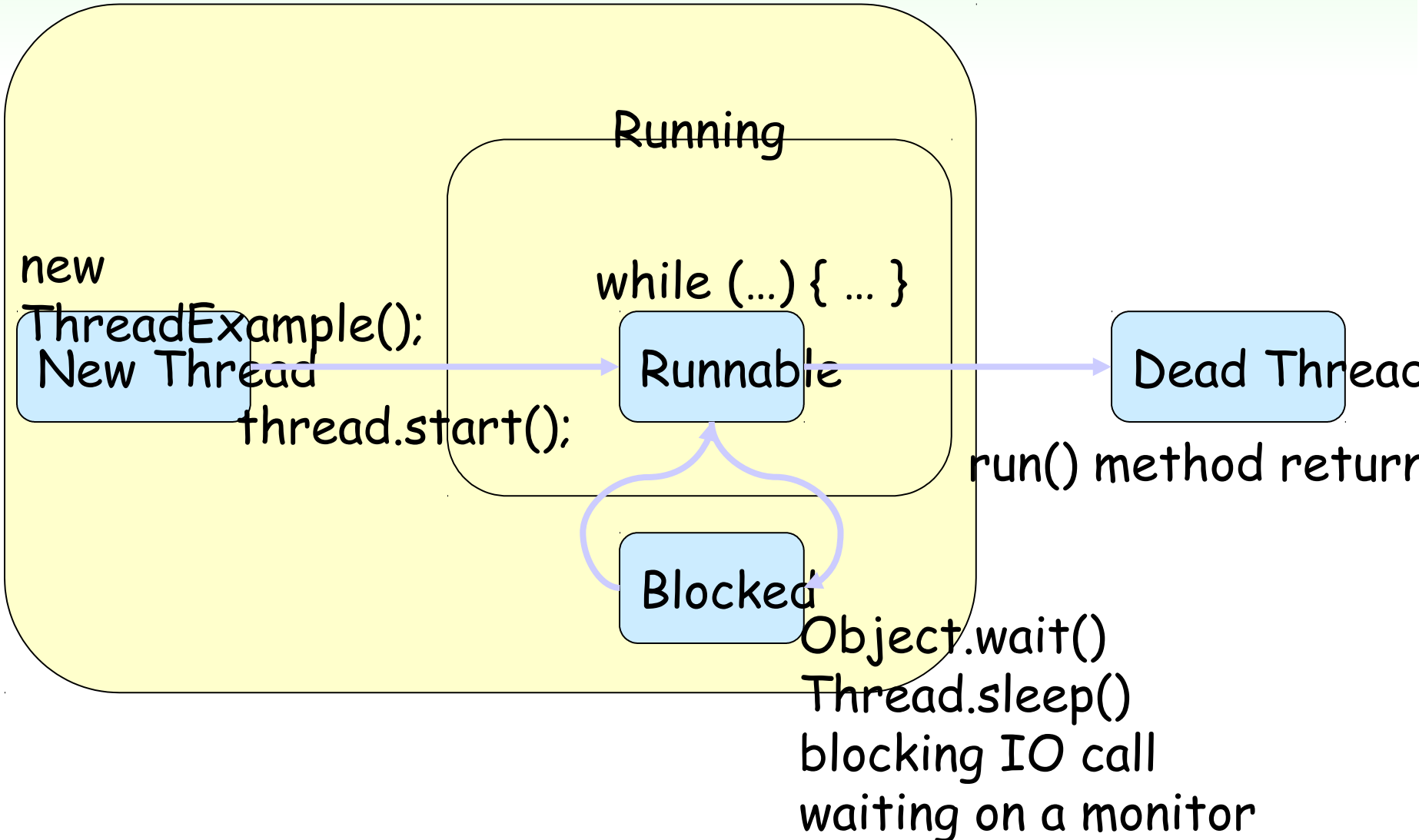
RESULT

Scheduling Threads



- Waiting for I/O operation to be completed
- Waiting to be notified
- Sleeping
- Waiting to enter a synchronized section

Thread State Diagram



Example

```
public class PrintThread1 extends Thread {
    String name;
    public PrintThread1(String name) {
        this.name = name;
    }
    public void run() {
        for (int i=1; i<500 ; i++) {
            try {
                sleep((long)(Math.random() * 100));
            } catch (InterruptedException ie) { }
            System.out.print(name);
        } }
```

Example (cont)

```
    public static void main(String args[]) {
        PrintThread1 a = new PrintThread1("*");
        PrintThread1 b = new PrintThread1("-");
        PrintThread1 c = new PrintThread1("=");
        a.start();
        b.start();
        c.start();
    }
}
```