The Process

Process

is any program which *is executed* by computer's operation system.

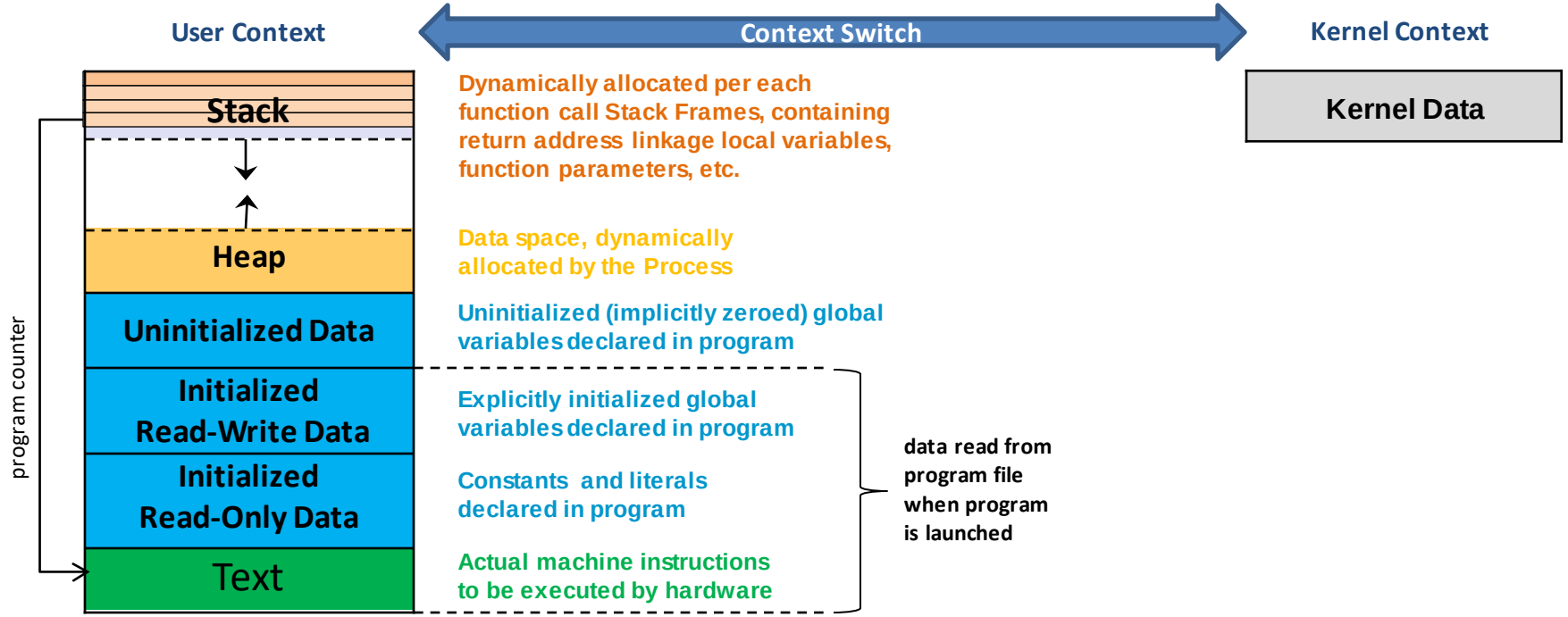
Kernel

is Operation System, that interacts with hardware and provides services like Memory Management, CPU scheduling, Filesystem, I/O Device access, etc.

System Call

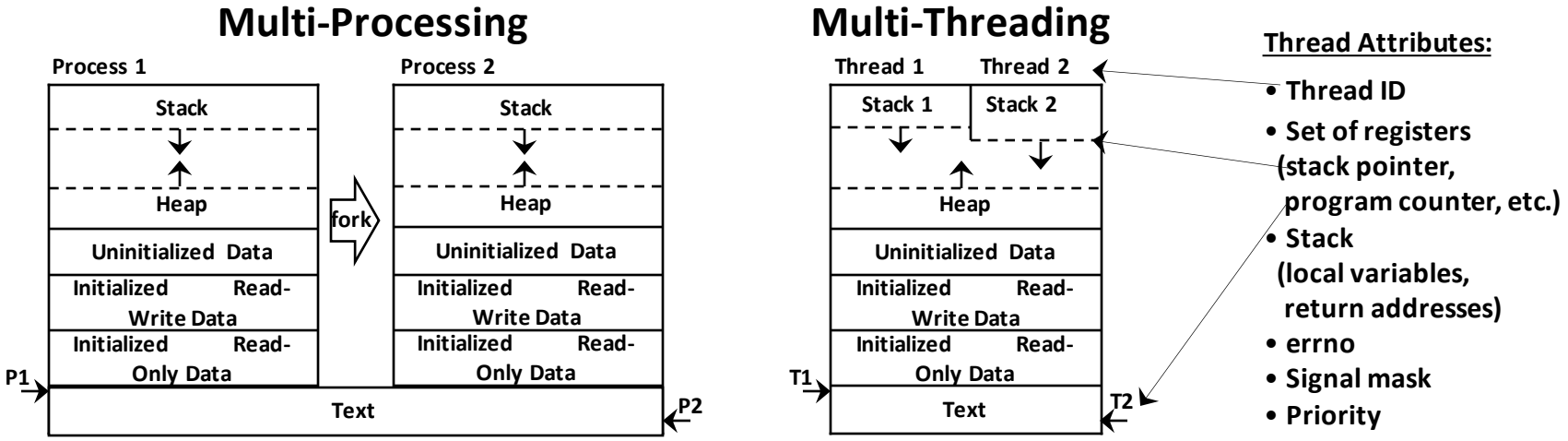
is direct entry point, provided by Kernel, through which active Process can obtain the service

Process Representation in Memory



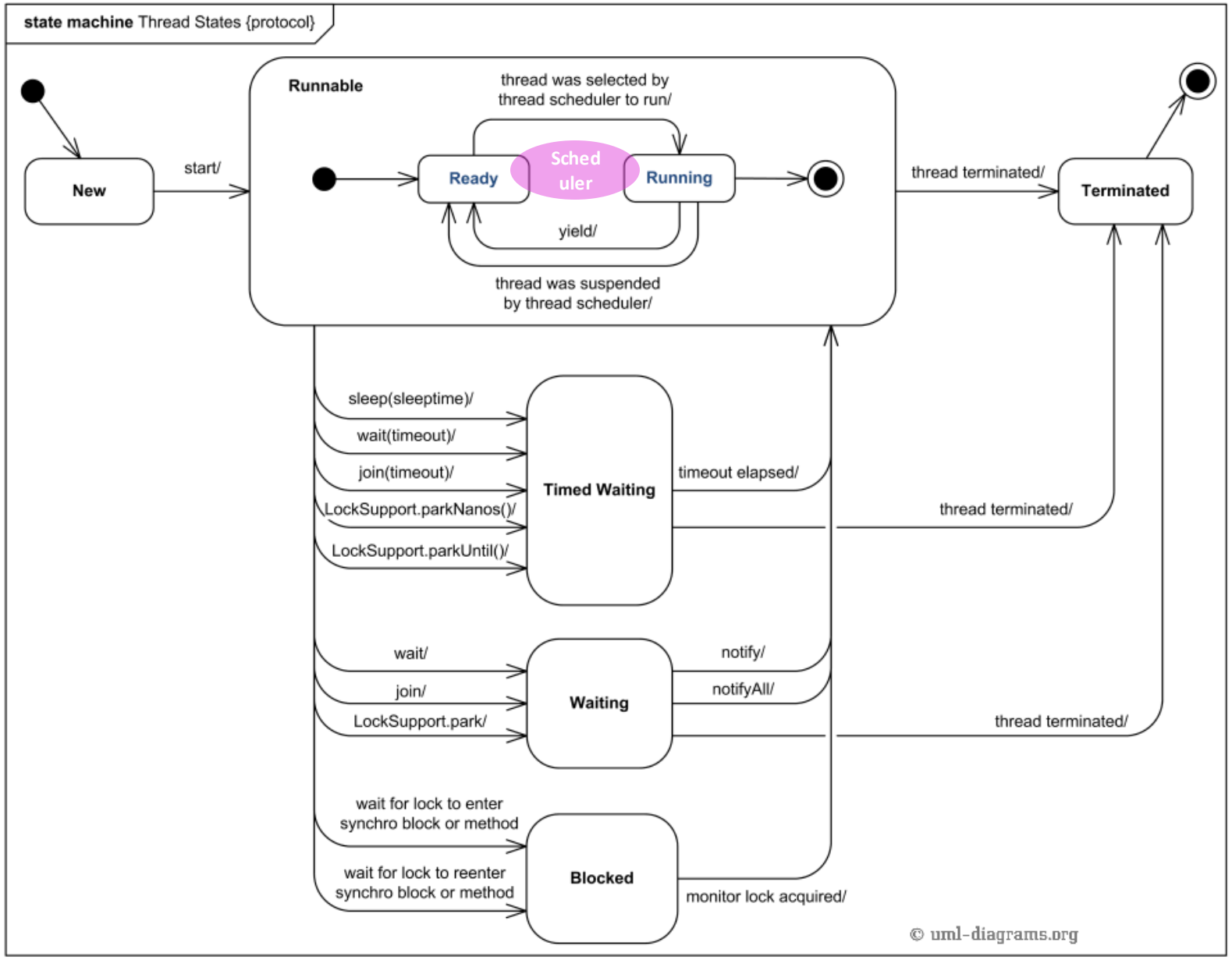
Thread

Thread is separate part of process, providing it's specific working flow, and sharing the process data and resources with other threads.



Multi-Processing	Multi-Threading
New processes created by means of expensive *fork* call, copying memory and descriptors to allocated resources.	Thread is "light-weight" process. Its creation is 10-100 times faster than process creation, and does not require copying of memory and descriptors.
Different processes have separate address spaces and resources	Multiple threads directly share memory and resources
Inter-Process communication requires usage of specific IPC mechanisms.	Shared data segment of process is used for inter-thread data exchange
Context switching is expensive.	Context switching is cheap.
Each process uses system calls to allocate its own resources (IPC, synchronization, other resources).	Thread uses system calls for synchronization needs. All other resources could be shared.
Single-threaded process could be executed each time by single CPU	Multi-threaded process can utilize multiple CPUs for simultaneous execution of multiple threads.

Java Thread State Diagram



Java Thread & Runnable

```
// sketch of standard Java core class Thread
class Thread implements Runnable {
    private Runnable r = null;

    public Thread(){}

    public Thread(Runnable r) {
       this.r = r;
    }

    @Override
    public void run() {
       if (r!=null) r.run();
    }

    public void start() {
       // Ask JVM (or Operational System)
       // to create new Stack in our process and
       // to begin run() method execution on this stack
    }
    …
}
```

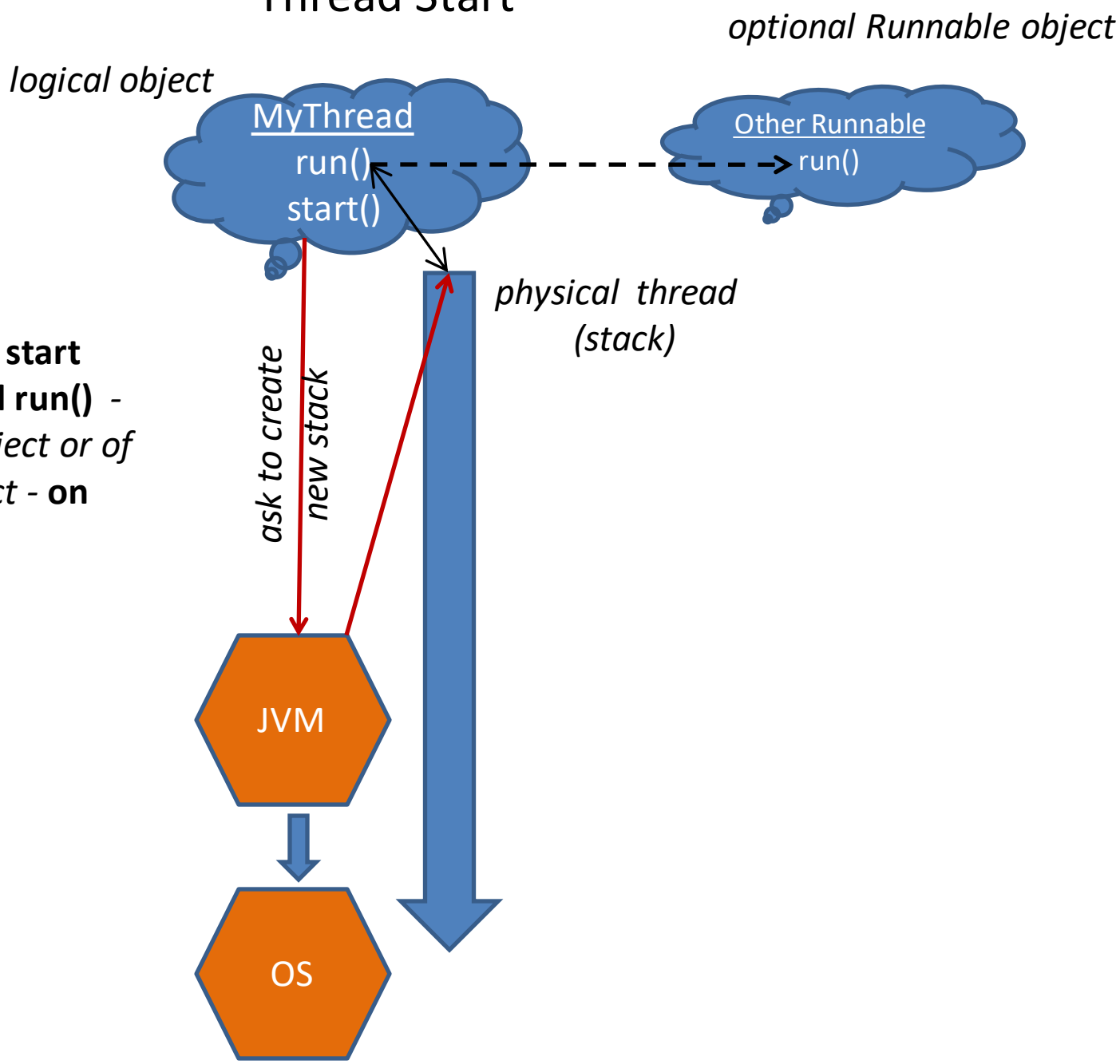
```
//===== variant A
class MyThread extends Thread{
    @Override
    public void run() {
       // do something
    }
}

//===== variant B
class MyRunnable implements Runnable {
    @Override
    public void run() {
       // do something
    }
}

//======= launch the thread =======
// variant A
Thread myThread = new MyThread();
myThread.start();

// variant B
Runnable myRunnable = new MyRunnable();
Thread coreThread = new Thread(myRunnable);
coreThread.start();
```

Thread Start



start() – **asks JVM to start execution of method run()** - *of current Thread object or of other Runnable object* - **on new Stack**

Sketch of “Home-baked” Thread Interruption

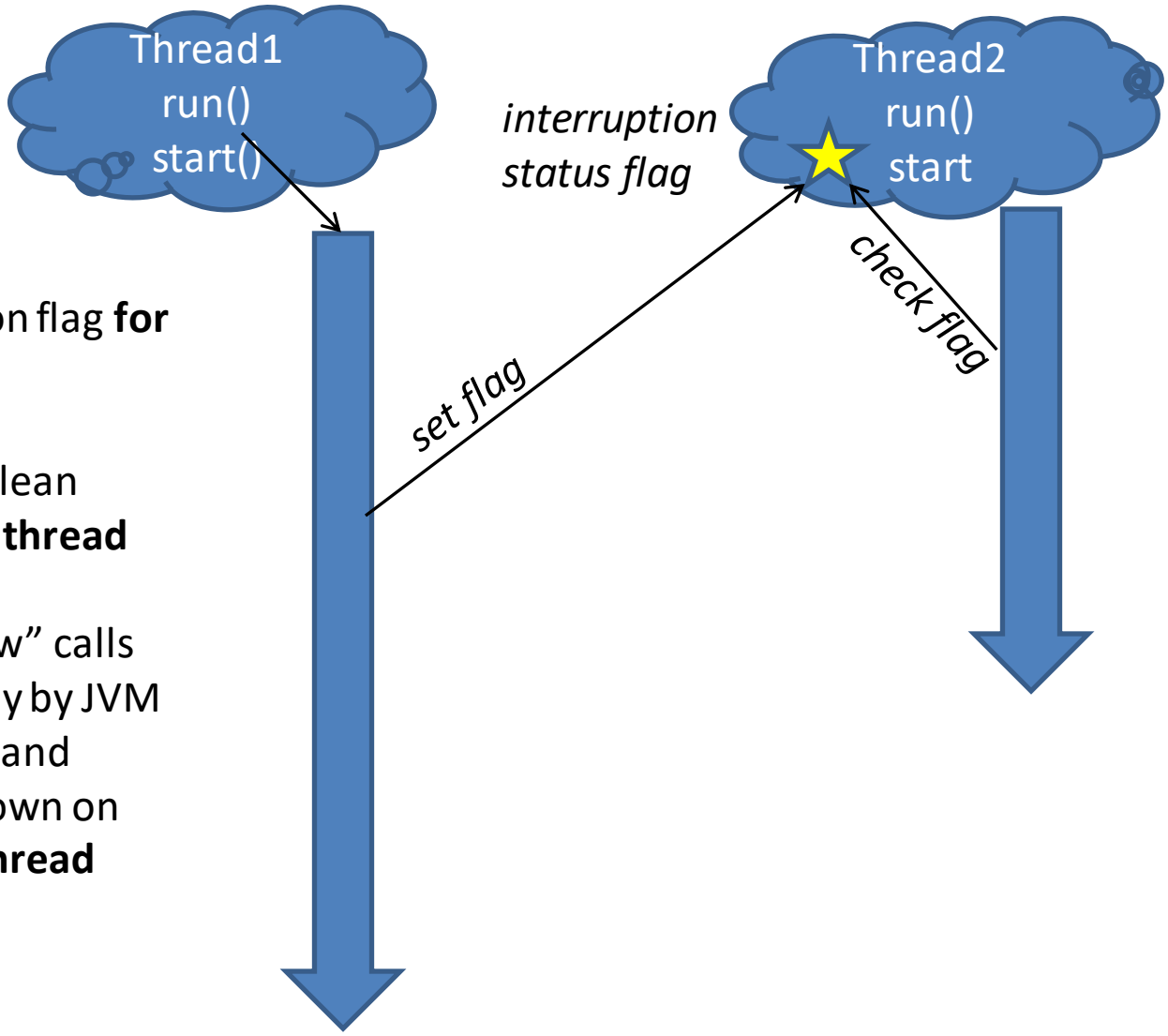
```
class MyThread {
        ….
        boolean isInterrupted = false;

        // called by interrupting thread
        void interrupt(){
           isInterrupted = true;
        }

        //  thread "sleeping"
        public void sleep(long millisPeriod) throws InterruptedException{
           long time0 = System.currentTimeMillis();
           while (currentTimeMillis() - time0 < millisPeriod){   // unjustified CPU load
              if (isInterrupted) {
                 isInterrupted = false;
                 throw new InterruptedException();
              }
           }
        }
}`
```

- In real class Thread, the methods sleep, join and other “slow calls” are executed by JVM (OS) **without CPU load**
- This also requires the “isInterrupted” flag to be maintained by JVM on per-thread basis
- When thread is not “waiting”, it could ask about flag value from JVM

Thread Interruption



interrupt() – set interruption flag **for other thread**

interrupted() – check and clean interruption flag **for calling thread**

sleep(), join(), wait() – "slow" calls are interrupted immediately by JVM if interruption flag was set, and *InterruptedException* is thrown on stack of **"slow" call caller thread**