# CS331 Tutorial 2

# Java Thread Programming

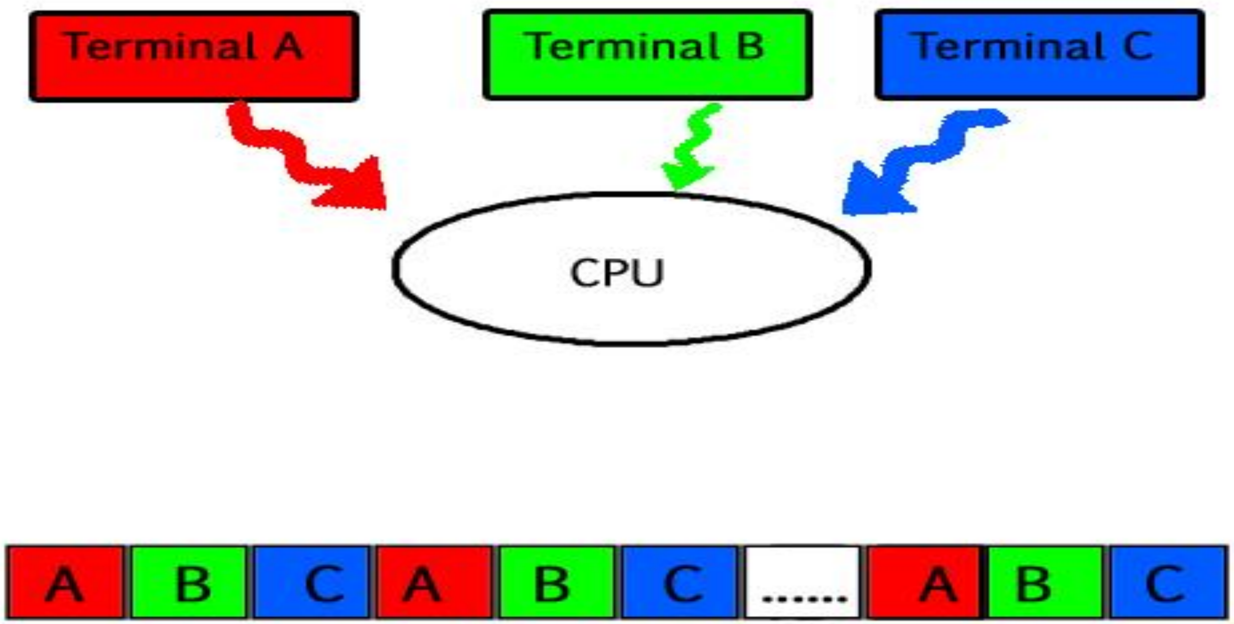
A. Sahu
Dept of Comp. Sc. & Engg.
Indian Institute of Technology Guwahati

# Outline

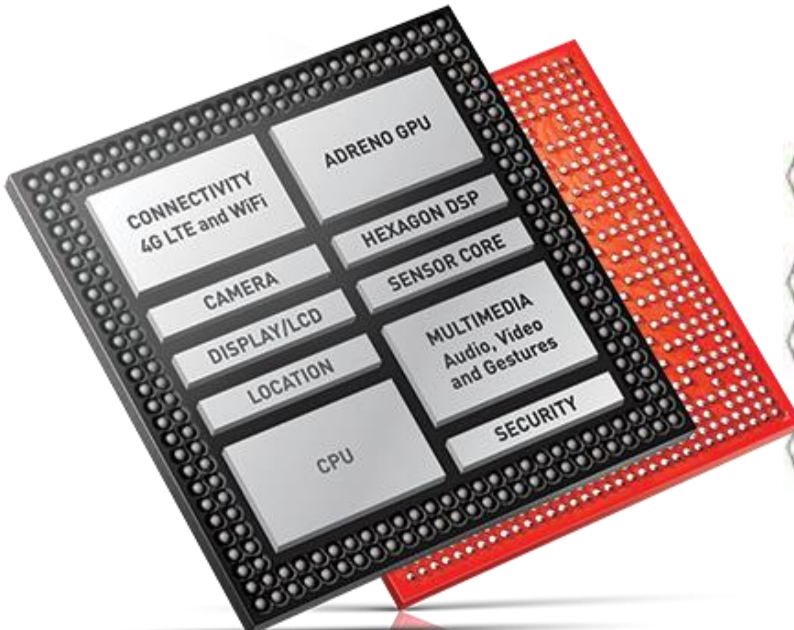
- Motivation for Concurrent Programming
  - Earlier Goal : Many thing want to run simultaneously
    - Multi-Tasking using Single Cores: Surfing email at the time of listening songs
  - High Performance as Multi-Cores
    - Multiprocessor programming == > Threading
    - Multi-computer programming == > Multi-processing/process level parallelism
- Threading Methodology
- 8 Rules Designing Mutli-Threaded APPs
- Java Threading Examples

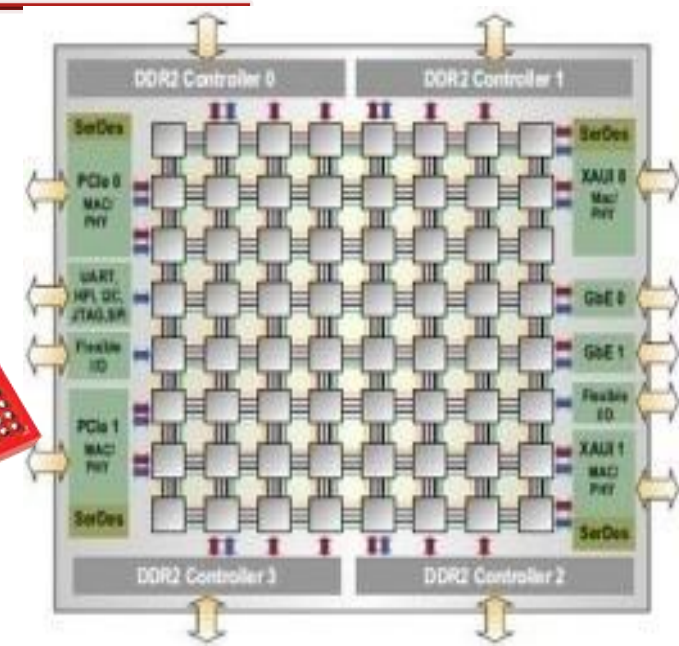
# Multi Tasking on  a Single Core PC

- Emulating concurrency on single core
- Giving user with experience of multiple task running on single core : **Surfing email at the time of listening songs**
- **Interleaving** is the best way to achieve

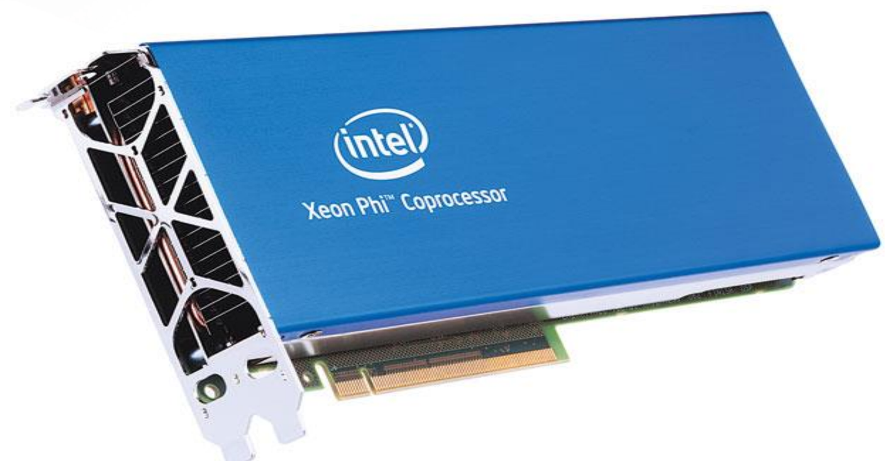


# Example of Multicore




Tilera64

# Why Multicore ?

- Many applications are highly parallel
  - Take benefit of all parallelism (instruction, data and thread)
- Multiprocessors
  - Flexible, programmable, high performance
  - Take benefit of all parallelism (instruction, data and thread)
  - Likely to be cost/power effective solutions

# Why Multicore ?

- Multiprocessors are likely to be cost/power effective solutions
  - **Share lots of resources**
    - ***Personal room is costlier than dormitory***
    - ***You cannot allocate a Bungalow to each student: it will too costly***
      - ***Hostel room with shared facility is sufficient***
  - Need not require very high frequency to run
  - Lots of replication makes easy to manage and cost effective in design

# Multicore Difficulties I

- Multiprocessors are likely to be cost/power effective solutions
  - Because it share lots of resources
    - ***Personal room is costlier than dormitory***
  - Sharing resource arise many other problems
    - Critical Sections
      - Lock and Barrier Design
    - Coherence
      - Shared data at all placed should be same
    - Consistency
      - Order should be similar to serial (ROB)
    - One processor Interference others
      - Share efficiently using some policy

# Multicore Difficulties II

- Task scheduling in multiprocessors
  - Deterministic task scheduling on multiprocessor with more than 2 processor is NP-Complete problem
  - 4 Tasks (A,B, C and D), 3 Processor
    - {A,B,C,D}{}{}, {A,B,C}{D}{}, ......Exponential Number of Solutions

# Multicore Difficulties III

- Many applications are highly parallel
  - Take benefit of all parallelism (instruction, data and thread)
  - Most of the coder write sequential code
  - Who will extract parallelism from applications ?
  - There is no successful auto-parallelisation tool till date
    - » Attempts: Cetus, SUIF, SolarisCC

# Threading

# Traditional View of a Process

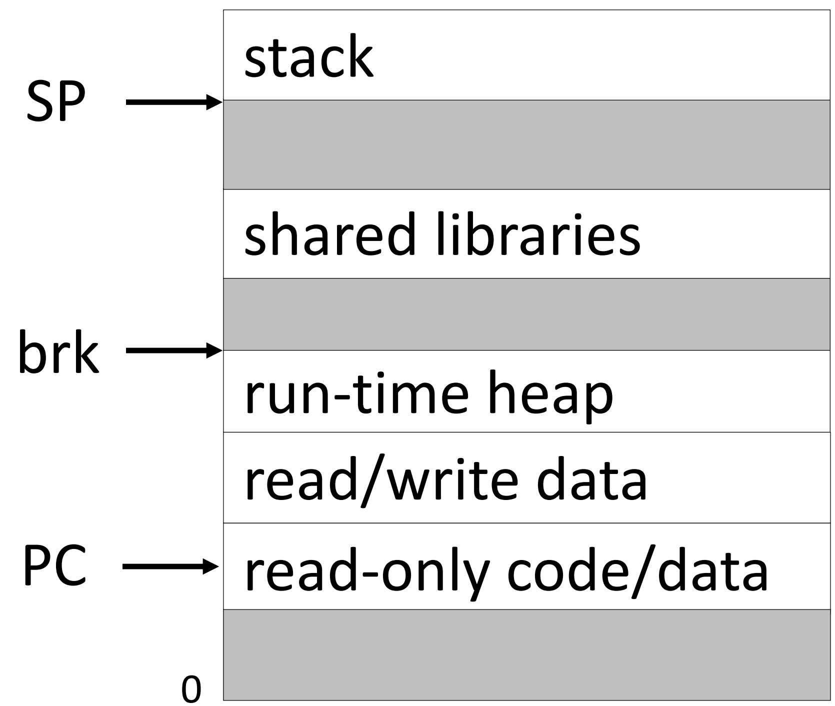
- Process = process context + code, data, and stack

Process context

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

Kernel context:
- VM structures (VMem)
- Descriptor table
- brk pointer

Code, data, and stack

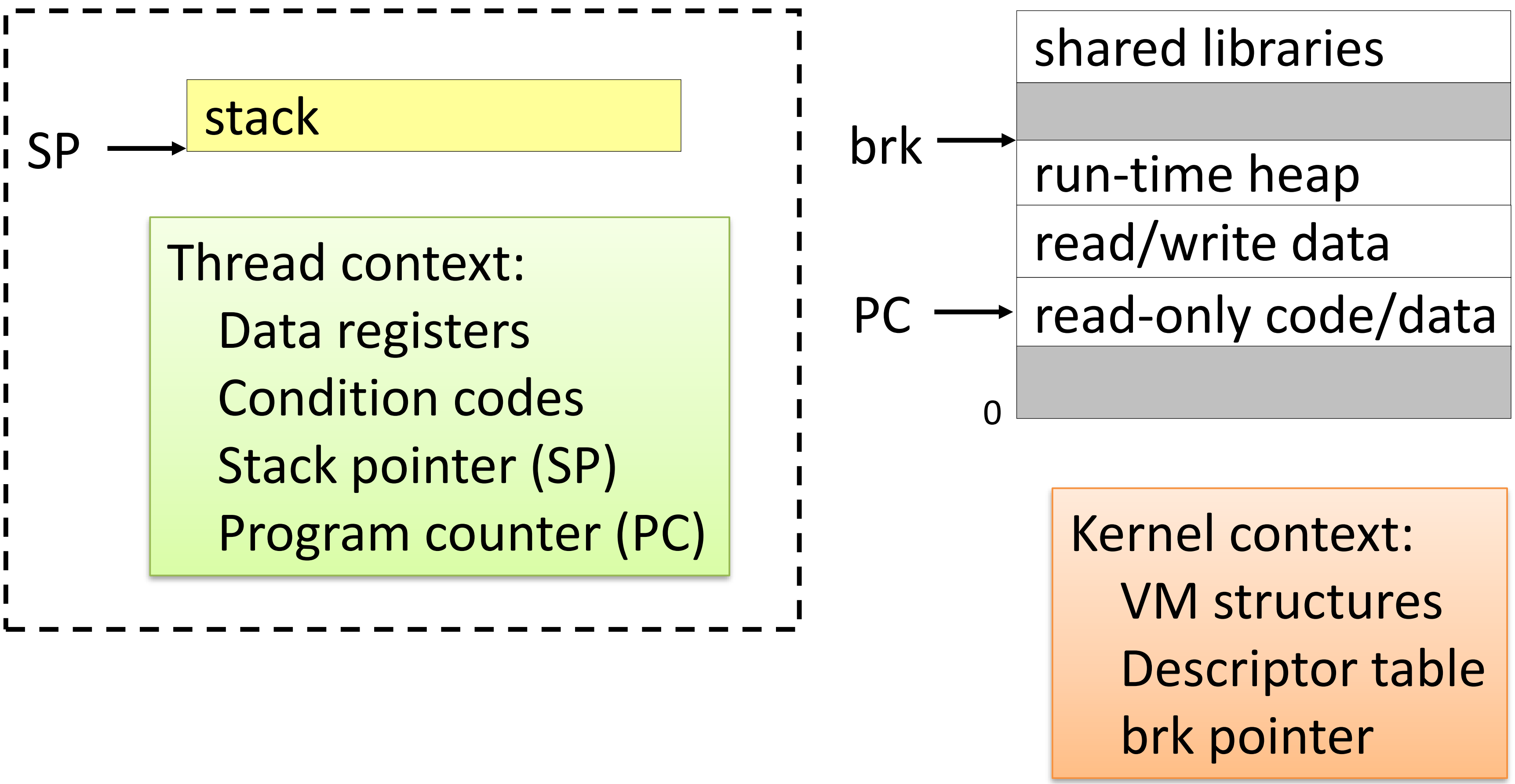


# Alternate View of a Process

- Process = thread+ code, data & kernel context

Thread (main thread)

Code and Data



# A Process With Multiple Threads

- Multiple threads can be associated with a process
  - Each thread has its *own* logical control flow (sequence of PC values)
  - Each thread *shares* the same code, data, and kernel context
  - Each thread has its own thread id (TID)

# A Process With Multiple Threads

Thread 1 (main thread)

stack 1

Thread 1 context:
- Data registers
- Condition codes
- SP1
- PC1

Shared code and data

| |
|---|
| shared libraries |
| |
| run-time heap |
| read/write data |
| read-only code/data |
| 0 |

Kernel context:
- VM structures
- Descriptor table
- brk pointer

Thread 2 (peer thread)

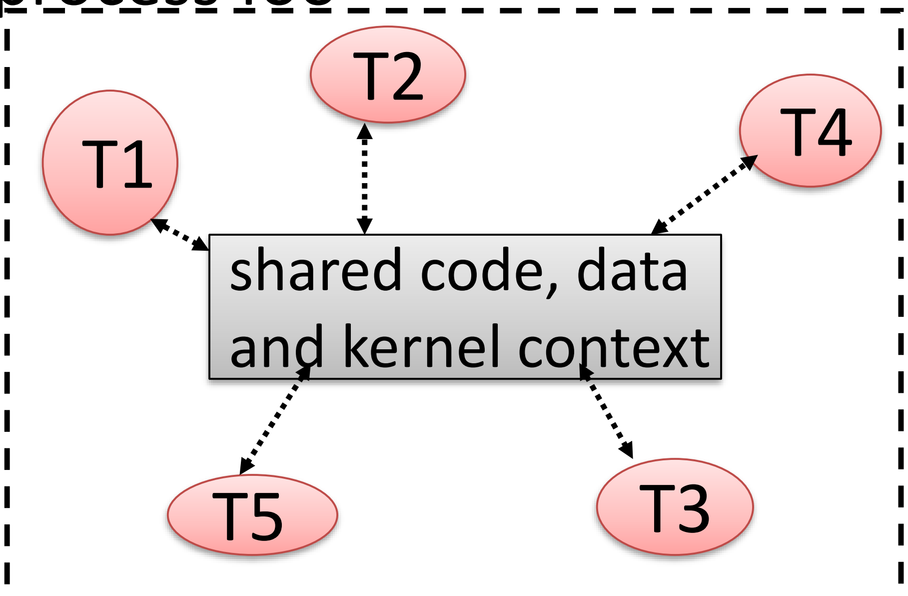
stack 2

Thread 2 context:
- Data registers
- Condition codes
- SP2
- PC2

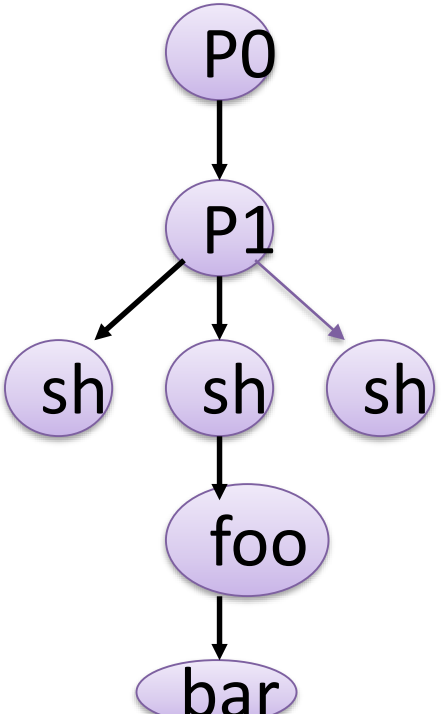
# Logical View of Threads

- Threads associated with a process form a pool of peers
  – Unlike processes, which form a tree hierarchy

Threads associated with process foo




Process hierarchy



# Threads vs. Processes

- How threads and processes are similar
  - Each has its own logical control flow
  - Each can run concurrently
  - Each is context switched

# Threads vs. Processes

- How threads and processes are different
  - Threads share code and data, processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) is twice as expensive as thread control
    - Linux/Pentium III numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles to create and reap a thread

# Threading Language and Support

- Pthread: POSIX thread
  - Popular, Initial and Basic one
- Improved Constructs for threading
  - c++ thread : available in c++11, c++14
  - **Java thread : very good memory model**
    - **Atomic function, Mutex**
- Thread Pooling and higher level management
  - OpenMP (loop based)
  - Cilk (dynamic DAG based)

# Threading Methodology

# Threading Methodology

- **Does not recommend going straight to concurrency!**
- First produce a tested single-threaded program
  - Use reqs./ design/ implement /test/ tune/ maintenance steps
- Then to create a concurrent system from the former, do
  1. Analysis: Find computations that are independent of each other
     1. AND take up a large amount of serial execution time (80/20 rule)
  2. Design and Implement: straightforward Test for Correctness: Verify that concurrent code produces correct output
  3. Tune for performance: once correct, find ways to speed up

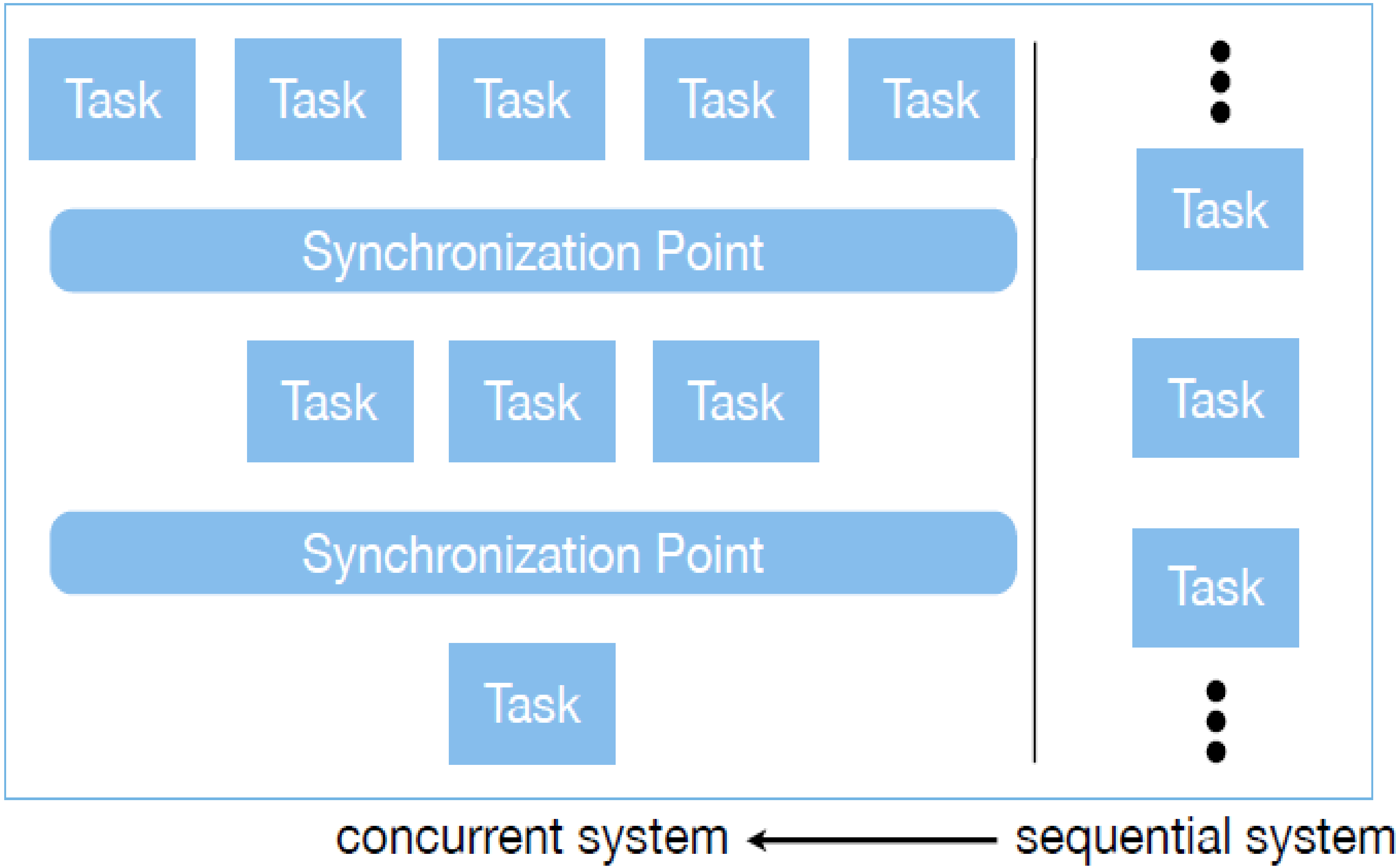
# Performance Tuning

- Tuning threaded code typically involves
  - identifying sources of contention on locks (synchronization)
  - identifying work imbalances across threads
  - reducing overhead
- Testing and Tuning
  - Whenever you tune a threaded program, you must test it again for correctness
- Going back further
  - if you are unable to tune system performance,
  - you may have to re-design and re-implement

# Design Models

- Two primary design models for concurrent algorithms
- Task Decomposition
  - identify tasks (computations) that can occur in any order
  - assign such tasks to threads and run concurrently
- Data Decomposition
  - program has large data structures where individual data elements can largely be calculated independently
  - data decomposition implies task decomposition in these cases

# Task Decomposition



# Eight rules of  Designing multithreaded APPS

# 1/8 Rules: Designing multithreaded APPS

- **Identify Truly Independent Computations**
- If you can’t identify (in a single threaded application) computations that can be done in parallel, you’re out of luck
- Some situations that indeed can’t be made parallel

# 2/8 Rules: Designing multithreaded APPS

- **Implement Concurrency at the Highest Level Possible**
- When discussing “What’s Not Parallel” a common refrain was “you can’t make this parallel,
  - So see if its part of a larger computation that CAN be made parallel”
- This is such good advice, it was promoted to being a guideline!
  - Two approaches: bottom up, top down

# 2/8 Rules: Bottom UP

- Our methodology says to create a concurrent program
  - Start with a tuned, single-threaded program and
  - Use a profiler to find out where it spends most of its time
- In the bottom-up approach, you start at those “hot spots” and work up; typically, a hotspot will be a loop of some sort
  - See if you can thread the loop
  - If not, move up the call chain, looking for the next loop and see if it can be made parallel...
  - If so, still look up the call chain for other opportunities, first.
    - Why? Granularity! You want coarse-grained tasks for your thread

# 2/8 Rules: Top Down

- With knowledge of the location of the hot spot
- Start by looking at the whole application and see if there are parallelization opportunities on the large-scale structure that contains the hot spot
  - if so, you’ve probably found a nice coarse-grained task to assign to your threads
  - If not, move lower in the code towards the hot spot, looking for the first opportunity to make the code concurrent

# 3/8 Rules: Designing multithreaded APPS

- **Plan Early for Scalability**
- The number of cores will keep increasing
- You should design your system to take advantage of more cores as they become available
  - Make the number of cores an input variable and design from there
- In particular, designing systems via data decomposition techniques will provide more scalable systems
  - humans are always finding more data to process!
- More data, more tasks; if more cores arrive, you’re ready

# 4/8 Rules: Designing multithreaded APPS

- **Make use of Thread-Safe Libraries Wherever Possible**
- First, software reuse!
  - Don’t fall prey to Not Invented Here Syndrome
  - if code already exists to do what you need, use it!
- Second, more libraries are becoming multithread aware
  - That is, they are being built to perform operations concurrently
- Third, if you make use of libraries, ensure they are thread-safe; if not, you’ll need to synchronize calls to the library
  - Global variables hiding in the library may prevent even this, if the code is not reentrant ; if so, you may need to abandon it

# 5/8 Rules: Designing multithreaded APPS

- **Use the Right Threading Model**
- Avoid the use of explicit threads if you can get away with it
- They are hard to get right
- Look at libraries that abstract away the need for explicit threads
  - OpenMP, Cilk and Intel Threading Building Blocks
  - Scala's agent model, Go's goroutines and Clojure's concurrency primitives
- All of these models hide explicit threads from the programmer Right Threading Model

# 6/8 Rules: Designing multithreaded APPS

- **Never Assume a Particular Order of Execution**
- With multiple threads, as we've seen, the scheduling of atomic statements is nondeterministic
- If you care about the ordering of one thread's execution with respect to another, you have to impose synchronization
- But, to **get the best performance**, you want to **avoid synchronization** as much as possible
- In particular, you want high granularity tasks that don't require synchronization
  – This allows your cores to run as fast as possible on each task they're given

# 7/8 Rules: Designing multithreaded APPS

- **Use Thread-Local Storage Whenever Possible or Associate Locks with specific data**
- Related to Rule 6; the more your threads can use thread-local storage, the less you will need synchronization
- Otherwise, associate a single lock with a single data item
  - in which a data item might be a huge data structure
- This makes it easier for the developer to understand the system;
  - “if I need to update data item A, then I need to acquire lock A first”

# 8/8 Rules: Designing multithreaded APPS

- **Dare to Change the Algorithm for a Better Chance of Concurrency**
- Sometimes a tuned, single-threaded program makes use of an algorithm which is not amenable to parallelization
- They might have picked that algorithm for performance reasons
  - Strassen's Algorithm $O(n^{2.81})$ vs. the triple-nested loop algorithm to perform matrix multiplication $O(n^3)$
- Change the algorithm used by the single-threaded program to see if you can then make that new algorithm concurrent
  - BUT: when measuring speedup, compare to the original!!

# Thread Programing In Java

# What Is a Java Thread?

- A thread is actually a lightweight process.
- Java provides built-in support for **multithreaded programming.**
- A multithreaded program contains two or more parts that can run **concurrently**.
  - Each part of such a program is called a thread and
  - each thread defines a separate path of the execution.
- Thus, multithreading is a specialized form of multitasking.

# Java Thread Model

- The Java run-time system depends on threads for many things.
- Threads reduce inefficiency by preventing the waste of CPU cycles.
- Threads exist in several states:
  - **New** - When we create an instance of Thread class, a thread is in a new state.
  - **Running -** The Java thread is in running state.
  - **Suspended** - A running thread can be **suspended**, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off.
  - **Blocked** - A Java thread can be blocked when waiting for a resource.
  - **Terminated** - A thread can be terminated, which halts its execution immediately at any given time.

# Multithreading in Java: Thread Class and Runnable Interface

- Java's multithreading system is built upon the Thread class
- Its methods, and its companion interface, **Runnable**.
- To create a new thread, your program will
  - either extend **Thread**
  - or **implement** the **Runnable interface**.

# Thread class methods to manage threads

| | |
|---|---|
| getName() | Obtain thread's name |
| getPriority() | Obtain thread's priority |
| isAlive() | Determine if a thread is still running |
| join() | Wait for a thread to terminate |
| run() | Entry point for the thread |
| sleep() | Suspend a thread for a period of time |
| start() | Start a thread by calling its run method |

# Main Java Thread

- Thread and Runnable interface to create and manage threads, beginning with the **main java thread**.
- **Why Is Main Thread So Important?**
  – Because it affects the other 'child' threads.
  – Because it performs various shutdown actions.
  – Because it's created automatically when your program is started.

# How to Create a Java Thread

- Java lets you create a thread one of two ways:
  - By **implementing** the **Runnableinterface**.
  - By **extending** the **Thread.**
- **Runnable Interface**
  - The easiest way to create a thread is to create a class that implements the **Runnable** interface.
  - To implement Runnable interface, a class need only implement a single method called run( )
- **Extending Java Thread**
  - Override the run() method and then to create an instance of that class.
  - The run() method is what is executed by the thread after you call start()

# Create Java Thread: Runnable Interface

```
public class MyThIR implements Runnable{
      public void run(){
            System.out.println("Child
                  thread is running...");
      }
}
public static void main(String args[]){
      MyThIR  m1=new  MyThIR();
      Thread t1 =new Thread(m1);
      t1.start();
 }
```

# Create Java Thread: Extending Java Thread

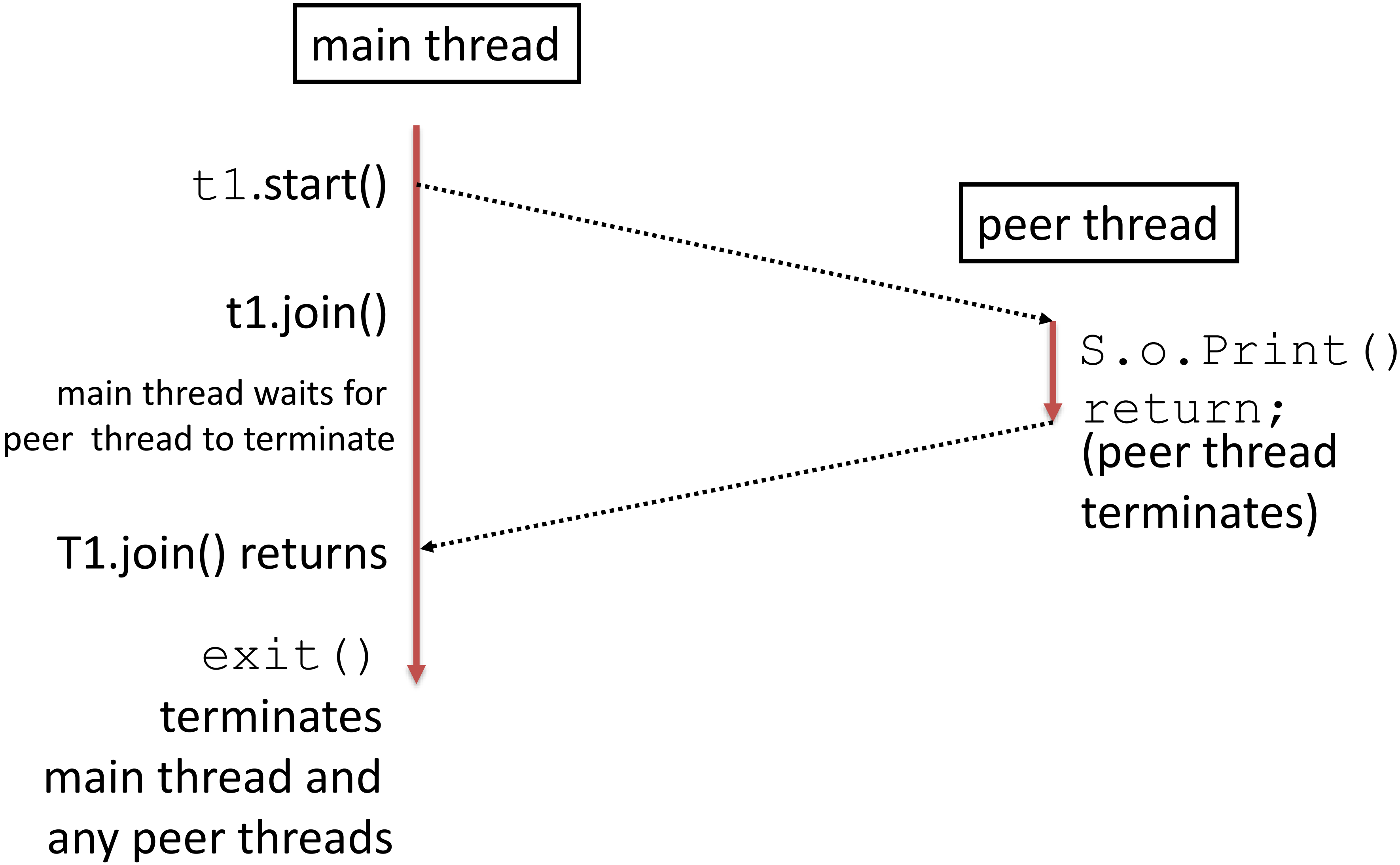
```
public class MyTh_EJT extends Thread{
      public void run(){
            System.out.println("Child
                  thread is running...");
      }
}

public static void main(String args[]){
      MyTh_EJT  t1 =new  MyTh_EJT();
      t1.start();
}
```

# Create  Java Thread: Extending Java Thread

```
public class MyTh_EJT extends Thread{
      public void run(){
            System.out.println("Child
                  thread is running...");
            return;
      }
}
public static void main(String args[]){
      MyTh_EJT  t1 =new  MyTh_EJT();
      t1.start();
      t1.join();
}
```

# Execution of Threaded “child thread”



# VectorSum Serial

```
int A[VSize], B[VSize], C[VSize];
void VectorSumSerial(){
   for( int j=0;j<SIZE;j++)
     A[j]=B[j]+C[j];
}
```

**Suppose Size=1000**

| | | | |
|---|---|---|---|
| **0-249** | **250-499** | **500-749** | **750-999** |
| **T1** | **T2** | **T3** | **T4** |

# VectorSum Serial

```
int A[VSize], B[VSize], C[VSize];

void VectorSumSerial(){
   for( int j=0;j<SIZE;j++)
     A[j]=B[j]+C[j];
}
```

- Independent
- Divide work into equal  for each thread
- Work per thread: Size/numThread

# VectorSum Parallel

```
void DoVectorSum(int TID){
    int j, SzPerthrd, LB, UB;
     SzPerthrd=(VSize/NUM_THREADS);
     LB= SzPerthrd*TID;UB=LB+SzPerthrd;

    for(j=LB;j<UB;j++)
     A[j]=B[j]+C[j];
}
```

# VectorSum Parallel : Java Threads

```
public class MyTh_EJT extends Thread{
      public void run(int TID){
            DoVectorSum(TID)
            return;
      }
}
public static void main(String args[]){
   int NumOfTasks = Integer.parseInt(args[0]);
   for ( int i=0; i < NumOfTasks; i++) {
       MyTh_EJT t1=new  MyTh_EJT();
       t1.start(i);
      }
}
```

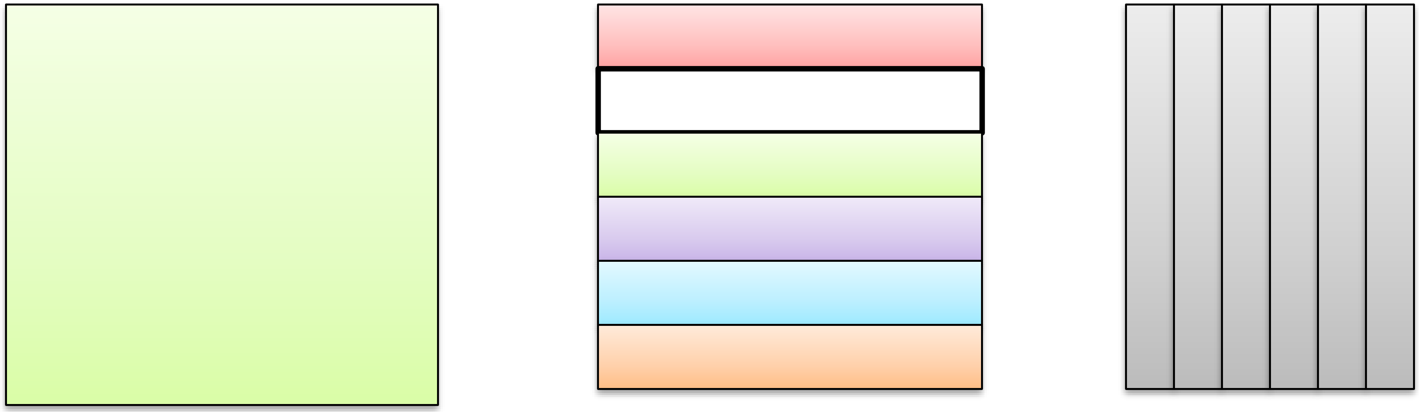
# VectorSum Parallel : Java Threads

```
Complete java program will be
uploaded to MS Team
```

# Matrix multiply and threaded matrix multiply

- Matrix multiply: C = A × B

$$C[i,j] = \sum_{k=1}^{N} A[i,k] \times B[k,j]$$



# Matrix multiply and threaded matrix multiply

- Matrix multiply: C = A × B

$$C[i,j] = \sum_{k=1}^{N} A[i,k] \times B[k,j]$$

- Divide the whole rows to T chunks
  - Each chunk contains : N/T rows, Assume N%T=0

# Four ways to implement a synchronized counter in Java

- **Suppose there is a Shared counter and every threads are attempting to manipulate**
  1. **Synchronized Block**
  2. **Atomic Variable**
  3. **Concurrent Lock**
  4. **Semaphore**

# Simple Java Ctr: without Lock

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Counter implements Runnable {
    private static int counter = 0;
    private static final int limit = 1000;
    private static final int threadPoolSize = 5;

   public static void main(String[] args) {
        ExecutorService ES = Executors.newFixedThreadPool(threadPoolSize);
        for (int i = 0; i < threadPoolSize; i++) { ES.submit(new Counter());   }
        ES.shutdown();
    }
}
```

# Simple Java Ctr: without Lock

```
    @Override
    public void run() {
        while (counter < limit) {
            increaseCounter();
        }
    }
    private void increaseCounter() {
        System.out.println(Thread.currentThread().getName() + " : " + counter);
        counter++;
    }
}
```

# Simple Java Ctr: with Sync Lock

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Counter implements Runnable {
    private static int counter = 0;
    private static final int limit = 1000;
    private static final int threadPoolSize = 5;
    private static final Object lock = new Object();
   public static void main(String[] args) {
       ExecutorService ES = Executors.newFixedThreadPool(threadPoolSize);
       for (int i = 0; i < threadPoolSize; i++) { ES.submit(new Counter());   }
       ES.shutdown();
    }
}
```

# Simple Java Ctr: without Lock

```
  @Override
  public void run() {
    while (counter < limit) {
      increaseCounter();
    }
  }
  private void increaseCounter() {
    synchronized (lock) {
    System.out.println(Thread.currentThread().getName() + " : " + counter);
    counter++;
    }
  }
}
```

# Simple Java Ctr: with atomic Int

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicInteger;
class Counter implements Runnable {
    private static AtomicInteger counter;
    private static final int limit = 1000;
    private static final int threadPoolSize = 5;
    public static void main(String[] args) {
        ExecutorService ES = Executors.newFixedThreadPool(threadPoolSize);
        for (int i = 0; i < threadPoolSize; i++) { ES.submit(new Counter());   }
        ES.shutdown();
    }
}
```

# Simple Java Ctr: atomic int

```
  @Override
  public void run() {
    while (counter < limit) {
      increaseCounter();
    }
  }
  private void increaseCounter() {
   System.out.println(Thread.currentThread().getName() + " : " + counter);
    counter.getAndIncrement();


  }
}
```

# Thanks