Software Platforms

Threads

LM in Computer Engineering

Massimo Maresca

OS and Processes

- Processes: abstract, discrete units of work, implemented on separate address spaces and working concurrently.
- OS makes a computer appear as a set of Virtual Machines (according to the original terminology). Every user sees a private machine.
- Programmers are not aware of concurrency. They just develop sequential programs.
- Key Issue: Memory Management. Processes are not supposed to share data, they just interact through OS primitives.

Tasks and Threads

- Concurrency from OS to application programs or, more appropriately, to Software Platforms.
- Tasks: abstract, discrete units of work, implemented in processes.
- From Wikipedia: In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.
- While “processes” do not share memory, on the contrary “threads” do.
- Multiple threads are more efficient than a single thread because of:
  - Hardware parallelism
  - Overlapping I/O
- We can test hardware parallelism by looking at the performance of a program which does not do I/O in a number of instances at different parallelization levels.

Threads: Concurrency and Parallelism

- In addition to supporting concurrency at the process level, Threads match the parallel nature of current generation CPUs.
- As a consequence multiple threads are more efficient than a single thread because of:
  - Overlapping I/O (Concurrency)
  - CPU Architectures (Parallelism)
- We can test hardware parallelism by looking at the performance of a program which does not do I/O in a number of instances at different parallelization levels.

A Simple Thread

```
public class MyThread extends Thread {
  int n;
  MyThread(int i){
     n=i;
   }

 public void run (){
     String threadName = Thread.currentThread().getName();
     for (int i=0; i < 5; i++) {
          Thread.currentThread().sleep(1000);
          int val = n*10+i;
          System.out.println(threadName + " " + val);
     }
   }
}
```

```
public class ThreadMain {
  public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
      System.out.println("Generating Thread n. "+ i);
      MyThread T= new MyThread(i);
      T.start();
    }
  }
}
```

Program: ASimpleThread

Task execution: Alternatives

- Alternatives
  - Sequential (Single Threaded)
  - Concurrent/Parallel (Thread per Task)
    - Task processing is offloaded from the main thread, thus enabling the main loop to resume waiting for the next request.
    - Tasks can be processed in parallel enabling multiple requests to be serviced simultaneously.
- Problems with Thread per Task
  - Thread life cycle overhead
  - Resource consumption
  - Stability

Number of CPUs

```
public class Ncpus {
        public static void main(String[] args) {
                int n_cpus = Runtime.getRuntime().availableProcessors();
                System.out.println(n_cpus);
        }
}
```

Program: Number of CPUs

Time consuming loop

```
void handleRequest(int iterationNumber, int sleepTime, int fakeParam) {
        int accumulator = 0;
        for (int i=0; i < iterationNumber; i++) {
                for (int j=0; j<1000; j++) {
                        accumulator+=(i*j);
                        if (accumulator > 1000000000)
                                accumulator = accumulator/1000;
                        if (accumulator == fakeParam) {
                                System.out.println(accumulator);
                        }
                }
        }
        if (sleepTime != 0) {
                try {
                        Thread.sleep(sleepTime);
                } catch (Exception e) {
                        System.out.println("Errror in sleep");
                        System.exit(-1);
                }
        }
}
```

questo tiene occupata la cpu

mentre questo la cpu non è occupata è come l'I/O

Single Thread

se x è il tempo di esecuzione della richiesta. per farne 2 ci mette 2x

```
// Main
                for (int i = 0; i < taskNumber; i++){
                        handleRequest(iterationNumber, sleepTime, -1);
                }
```

Program: SingleThreadedServer

Multiple Threads

ogni operazione è svolta da un thread diverso, quindi in teoria viene eseguito tutto in parallelo.

questo perche i computer moderni hanno piu cpu in modo da svolgere realmente operazioni in parallelo

```
public class HandleRequestThread implements Runnable
        private int iterationNumber;
        private int sleepTime;
        public HandleRequestThread(int iterationNumber, int sleepTime) {
                this.iterationNumber = iterationNumber;
                this.sleepTime = sleepTime;
        }
        public void run() {
                handleRequest(iterationNumber, sleepTime, -1);
        }
    }
// Main
for (int threadIndex =0; threadIndex < taskNumber ; threadIndex ++) {
        threads[threadIndex]= new Thread(new HandleRequestThread(iterationNumber, sleepTime));
        threads[threadIndex].start();
}
```

poi in pratica non è perfettamente tutto in parallelo

e quando il numero di thread supera il numero di cpu?
ci saranno ovviamente delle code. se però come nell'esempio di prima usiamo sleep
in 100 thread per 1s ci mette 1s poiché lo sleep non impiega la cpu in nulla.

quindi in generale dipende se il task usa cpu consuming operation

Problema: -tanti threads - tempo di creazione di tutti questi thread

Program: ThreadPerTaskServer

Thread Pool

- The Executor interface

i threads hanno due modalita di esecuzione parallelismo o concorrenza.
Paralllelismo=threads eseguiti su cpu diverse.
Concorrenza=threads eseguiti sulla stessa cpu che quindi dovranno condividere le risorse

```
public interface Executor {
    void execute (Runnable command);
}
```

Decouple thread activation mechanism from requestors.

Program: ThreadPool

See: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html

Optimal Thread Pool Size

$$N_{threads} = N_{Cpu} * UCp_u * \left(1 + \frac{W}{C}\right)$$

N_{Cpu}= Number of CPU
U_{Cpu} = Target CPU Utilization
$\frac{W}{C}$ = ratio of wait time to compute time

Thread Pool

```
ExecutorService pool = Executors.newFixedThreadPool(poolSize);
for (int threadNumber = 0; threadNumber < taskNumber; threadNumber++){
        Runnable h = new  HandleRequestThread(iterationNumber, sleepTime);
        pool.execute(h);
}
pool.shutdown();
```

ci sono piu task che threads. ES: 1k tasks che competono per poter utilizzare una pool di 10 threads

How a Thread Pool works

- Adoption of a BlockingQueue to store Tasks
- Submission of Tasks to the BlockingQueue (block on enqueue)
- Presence of a set of Workers that take care of Task execution
- Extraction of Tasks for execution (block on dequeue)

Program: ThreadPoolImplementation

See:

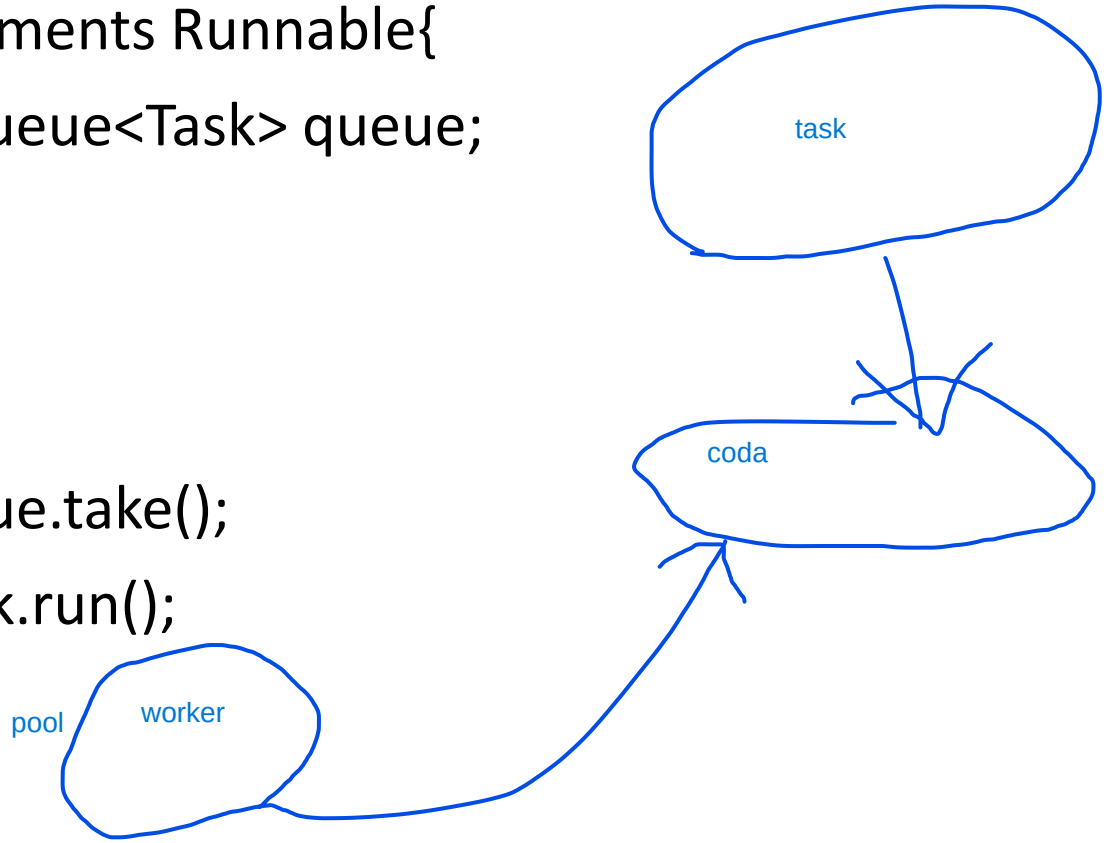
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html

How a Thread Pool works: Worker

come funziona la thread pool senza (fatta da zero senza libreria) si ha dei thread che runnano il codice detto worker. Worker ogni volta che è libero controlla la coda per controllare se c'è un task da eseguire se c'è lo esegue

```
public class Worker implements Runnable{
        private BlockingQueue<Task> queue;
        public void run() {
                Task task;
        while(true) {
                task = queue.take();
                        task.run();
    }
```



le task invece si inseriranno nella coda quando hanno bisogno di essere eseguita.

How a Thread Pool works: Starting the workers

```
BlockingQueue<Task> queue = new ArrayBlockingQueue<>(10);
TaskSource taskSource = new TaskSource (queue);
Worker worker = new Worker (queue);

for (int i = 0; i < nWorkers; i++) {
                    new Thread(worker).start();
}
```

How a Thread Pool works: Feeding the Task Queue

```
private BlockingQueue<Task> queue;

…..

Task task = new Task ("Task n. "+i);
 queue.put(task);
…..
```

Socket with Threads

```
……
try {
        ServerSocket sSoc = new ServerSocket(9000);
        while (true){
                Socket inSoc = sSoc.accept();
                MyThread T= new MyThread(inSoc);
                T.start();
        }
} catch (Exception e) {
        ….
}
…..
```

Program: SocketWithThreads

Socket with Threads and Pools

è il metodo migliore la pool cosi si evita di avere troppi threads, che tanto non vengono sfruttati

```
……
int poolSize = Integer.parseInt(args[0]);
ExecutorService pool = Executors.newFixedThreadPool(poolSize);
try {
        ServerSocket sSoc = new ServerSocket(9000);
        while (true){
                Socket inSoc = sSoc.accept();
                System.out.println("Generating Thread n. "+ i);
                MyThread T= new MyThread(inSoc, i);
                pool.execute(T);
        }
…..
```

inoltre la creazione di un thread richiede tempo, con la pool si risparmia il tempo di creazione dei threads

Program: SocketWithThreadsAndPools