جامعة الجزيرة الخاصة
ALJAZEERA PRIVATE UNIVERSITY

Threads

Threads

- Indrodaction
- Multi-threading (Ex)
- Threads & process
- Multi-threaded Server Architectures
- Thread Implementation
- Thread Scheduling
- Threads in Java
  - Thread life-cycle
  - Thread Building
  - Mutual Exclusion
  - Synchronization

Introducion

- الاتصالات بين الفعاليات تلعب دوراً هاماً في النظم الموزعة.
  - إدارة الفعاليات في النظم الموزعة تلعب دوراً مهماَ و حرجاً.
- التطبيقات الموزعة تستخدم IPC لتحقيق التعاون بين أجزاء النظام
  - سيئة = التكلفة العالية كون الـ IPC تمر بالـ Kernel ذهاباً و إياباً
- استخدام الـ multi-threaded في نموذج الـ C/S
  - يتم التعويض عن الاتصالات باستخدام الـ Data Sharing (Common Variables) و ضمن الـ user-level
  - يسمح في كلا الجهتين تغطية الاتصالات بالمعالجة المحلية = أداء أعلى للنظام
  - تسمح باستثمار الـ Parallelism عند تنفيذ البرنامج على حاسب متعدد المعالجات
  - البساطة في بناء التطبيقات = مهمات (Threads)
  - سيئة = blocking system call يعرقل الـ process بكاملها

Threads

- Thread Goal =
  - Maximize the concurrent execution between operations
  - Overlap computations with I/O operations
  - Concurrent processing on multiprocessors (overlap computations with other computations)
- Servers != bottlenecks
- Process allows resource sharing between Threads

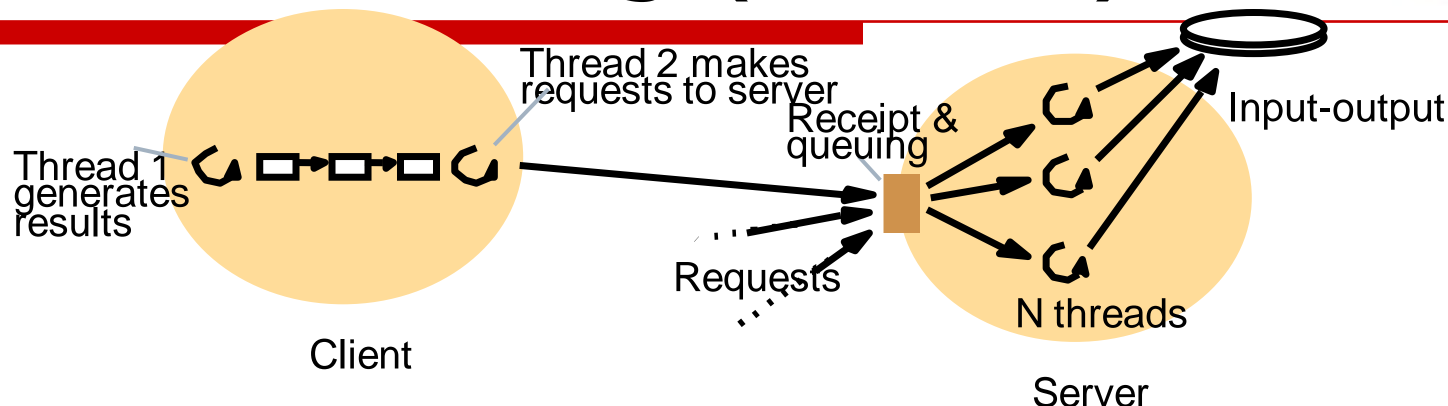
Threads & Processes (1/2)

- Process
  - Process لها فضاء عنونة خاص بها
  - Process لها آليات اتصال خاصة بها (sockets)
  - تشارك الموارد (ملفات) و آليات تزامن (semaphores)
  - Process تملك عدة threads و حالاتها
- Thread
  - Priority scheduling
  - Execution state (blocked, runnable …processor register values)

Threads & Processes (2/2)

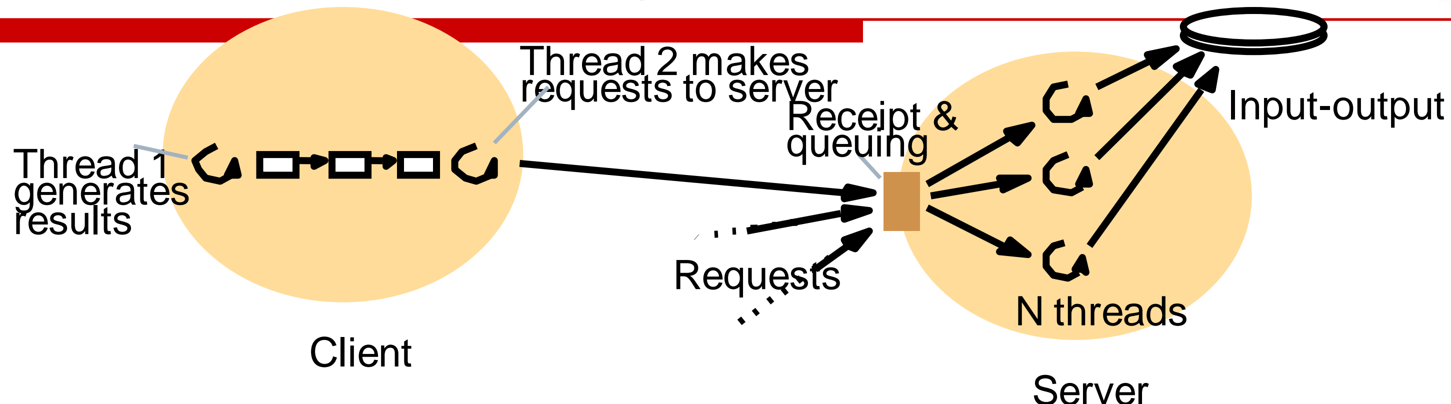
- إنشاء الـ Threads ضمن Process أقل كلفة من إنشاء Process
  - إنشاء Process = حوال ١١ ميلي ثانية
  - إنشاء Thread = حوالي ١ ميلي ثانية
- المبادلة بين الـ Threads ضمن نفس الـ Process أرخص بكثير من مبادلة الـ processes عبر الـ kernel
  - المبادلة بين الـ processes = حوالي ١.٨ ميلي ثانية
  - المبادلة بين الـ threads = حوالي ٠.٤ ميلي ثانية و عند مستوى الـ user-level و بدون الدخول في الـ kernel= حوالي ٠.٠٤ ميلي ثانية
- الـ threads تتشارك المعطيات بفعالية أكبر عبر متحولات مشتركة (conflict access)

Multi-threading (Ex 1/2)



- Request = 2ms processing + 8 ms I/O
  - Mono-thread Server ➔ throughput = 100 client / sec
  - Multi-threaded Server (**2** threads) = when 1 thread blocked on I/O. Then, the another can process another request ➔ throughput = 125 client / sec
  - Multi-threaded Server (**2** threads) + (caches, hit rate = 75% → 2ms disk access, 0.5 ms search in cashes) ➔ throughput = 400 client / sec

Multi-threading (Ex 2/2)



- Request = 2ms processing + 8 ms I/O
  - Multi-threaded Server (**2** threads) + (caches, hit rate = 75% → 2ms disk access, 0.5 ms search in cashes) + 2 processors ➔ throughput = 444 client / sec
  - Multi-threaded Server (more threads of 3) + (caches, hit rate = 75% → 2ms disk access, 0.5 ms search in cashes + 2 processors ➔ throughput = 500 client / sec

Multi-threaded Server Architectures

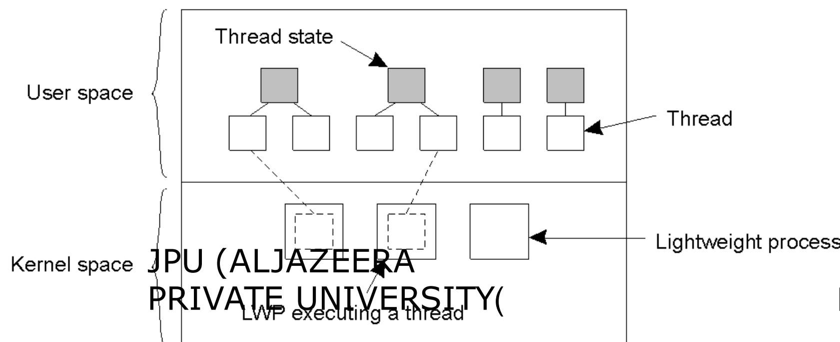
- Thread-per-Request
  - New worker Thread for each request
  - Pb = overheads of thread creation & destruction
- Fixed pool of worker Threads
  - Thread (I/O thread) to receive the requests & put them in queue
  - Server creates a fixed pool of worker Threads
  - Worker thread scan the request queue to process requests
  - Adv = Client class → priority request → several queues of priority
  - Pb = nb threads in the pool is fixed
  - Pb = shared queue between I/O thread & worker Threads
- ….

Thread Implementation (1/2)

- User-level library
  - إنشاء و إزالة الـ Threads (رخيص) = تخصيص و تحرير ذاكرة مكدس الـ thread
  - المبادلة بين الـ threads (رخيص) = تخزين / تحميل مسجلات الـ CPU
  - سيئة = blocking system call يعرقل الـ process بكاملها
- System kernel
  - إنشاء و إزالة الـ Threads (مكلف)
  - المبادلة بين الـ threads (مكلف)
  - ميزة = blocking system call لا يعرقل الـ process بكاملها

Thread Implementation (2/2)

- حل هجين = Lightweight Processes (lwp)
  - الـ Process تملك عدة lwp + thead user-level pkg لإنشاء و إزالة و مزامنة الـ threads
  - يتم ربط كل thread مع lwp (kernel-level) دون معرفة المبرمج
  - Lwp (عند الإنشاء، التبديل و العرقلة) تنفذ مجدول الـ threads في الـ user-level للبحث عن Threadو تنفيذها
  - المزايا =
    - بامتلاك عدد كافٍ من الـ lwp فإن الـ blocking sys call لا يعرقل كامل الـ process
    - يمكن الاستفادة من multiprocessors من خلال تنفيذ عدة lwp على عدة آلات و بشكل شفاف عن التطبيق
    - التطبيق لا يرى الـ lwp و لكن الـ threads
  - السيئة =
    - عمليات إدارة الـ threads أكثر كلفة منها في الـ kernel-level



Thread Scheduling

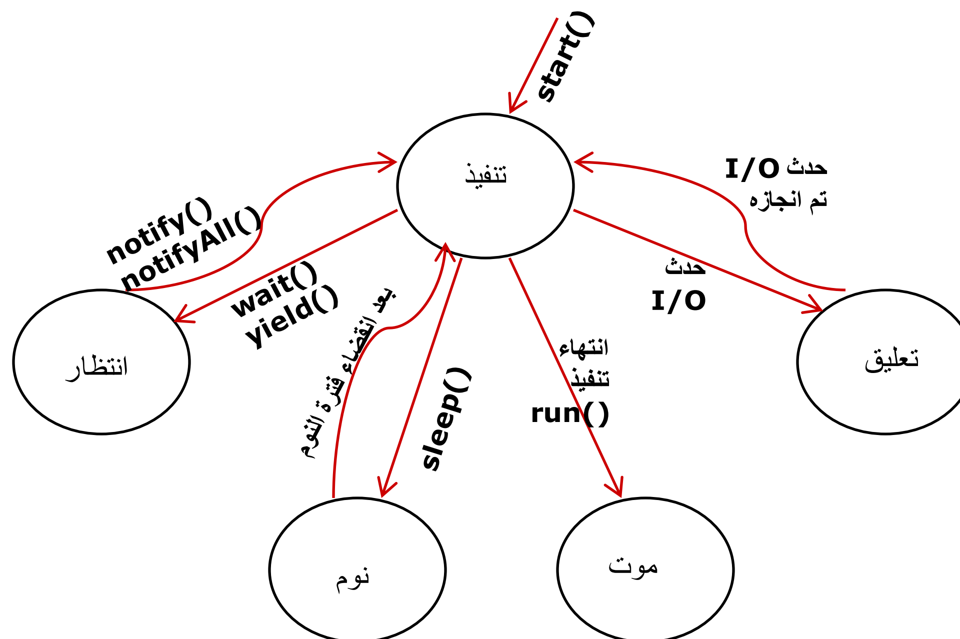
- Non-preemptive Scheduling = الـ Thread تتابع تنفيذها حتى تصادف Threading System Call
  - مقطع الكود بدون threading system calls = مقطع حرج
  - سيئة ١ = يتوجب على المبرمج استدعاء yield() حتى يسمح للـ threads الأخرى بالتقدم
  - سيئة ٢ = لا يمكن استخدامها في الـ real-time systems
  - سيئة ٣ = لا يمكن الاستفادة من الـ multiprocessors
- Preemptive Scheduling = يمكن مقاطعة الـ Thread عند أي نقطة (مثلاً ورود Thread ذات أولوية أعلى كما في الـ Real-time Systems)

Threads in Java

- الـ Threads يتم تنفيذها ضمن نفس مساحة العنونة للـ JVM
- الـ Threads تنفذ بشكل تسايري (Concurrency)
  - ضمان لتتابع التعليمات ضمن الـ Thread الواحدة
  - لا ضمان لترتيب التعليمات بين عدة Threads
- تنفيذ برنامج جافا = إطلاق Thread لتنفيذ تعليمات main()
- الصف java.lang.Thread يتحكم بسلوك الـ Thread و لا يمثل الـ Thread نفسها
- يمكننا دوماً معرفة الغرض الحاكم للـ Thread الحالية من خلال :
  - Static void Thread.currentThread()
- الـ Thread =
  - Thread Controller = غرض من الصف Thread
  - Thread Object= الغرض الذي يمثل تعليمات الـ Thread
    - غرض من صف ينفذ الواجهة Runnable
  - الـ Thread نفسها = تسلسل تنفيذ تعليمات الـ Thread
    - تقلع عند استدعاء start() على Thread Controller و تنفذ تعليمات run() و تنافس الـ Threads للحصول على المعالج

Thread life-cycle

- حياة الـ Thread تنتهي بنهاية run() و تختفي من الـ JVM و يبقى :
  - Thread Controller لاستدعاء عمليات إدارة الـ Thread (مثلاً isAlive())
  - Thread Object لاستدعاء عمليات خاصة بالتطبيق (مثلاً معرفة نتائج حسابات الـ Thread)



Thread Building by Interfaces

```
class MyRunnable implements Runnable
{
        public void run() {
                for(int i=0; i < 5; i++) {
                        S.O.P("MyThread " + i);
                        try {
                                Thread.sleep(500);
                        } catch (InterruptedException e)
                        {S.O.P("Interruption …. ");}
                }
                S.O.P("MyThread finishes   ….. ");
        }
}
```

```
public class Test{
        public static void main(String[] args) {
                Runnable myRun = new MyRunnable();
                Thread myTh = new Thread(myRun);
                myTh.start();
                for(int i=0; i < 5; i++) {
                        S.O.P("Main Thread" + i);
                        try {
                                Thread.sleep(500);
                        } catch (InterruptedException e)
                        {S.O.P("Interruption …. ");}
                }
                S.O.P("Main Thread finishes   ….. ");
        }
}
```

Thread Building by Inheretence

```
class MyThread extends Thread
{
        public void run() {
                for(int i=0; i < 5; i++) {
                        S.O.P("MyThread " + i);
                        try {
                                Thread.sleep(500);
                        } catch (InterruptedException e)
                        {S.O.P("Interruption .... ");}
                }
                S.O.P("MyThread finishes   ..... ");
        }
}
```

```
public class Test{
        public static void main(String[] args) {
                Thread myTh = new MyThread();
                myTh.start();
                for(int i=0; i < 5; i++) {
                        S.O.P("Main Thread" + i);
                        try {
                                Thread.sleep(500);
                        } catch (InterruptedException e)
                        {S.O.P("Interruption .... ");}
                }
                S.O.P("Main Thread finishes   ..... ");
        }
}
```

Mutual Exclusion

```
class Point{
        protected int x;
        protected int y;
        public void moveTo(int x, int y){
                this.x = x;
                this.y = y;
        }
        public String toString() {
                return "(" + x + "," + y + ")";
        }
}
```

```
public class Init implements Runnable {
        Point p;
        int val;
        public Init(Point p, int v){
                this.p = p;        this.val = v;
        }
        public void run() {
                for( ; ;) {p.moveTo(val, val);
                S.O.P(p);}
        }
}
```

```
public class Test{
        public static void main(String[] args) {
                Point p = new Point();
                Init init1 = new Init(p, 1);
                Init init2 = new Init(p, 2);

                Thread Th1 = new Thread(init1);
                Thread Th2 = new Thread(init2);

                Th1.start();
                Th2.start();
        }
}
```

```
(1, 1)
(1, 1)
....
(2, 2 )
....
(2, 1)
(1, 2)
......
```

Mutual Exclusion in Java

- Mutual exclusion in Java --- is based on --- monitor
- Critical section can be protected using a monitor
  - The thread enters critical section → takes the monitor
  - Another Thread wants entering same section → waits in queue for the monitor
  - When the first thread leaves the critical section → free the monitor
- ***synchronized(obj) {***

```
synchronized(obj) {
        .....
        // critical section
        .....
}
```

- ***public void moveTo(int x, int y) {***

```
public  void moveTo(int x, int y) {
        synchronized(this) {
                this.x = x;      this.y = y;
        }
}
```

Thread Synchronization

- **wait()**
  - Blocks the current Thread in the monitor's queue
  - Frees the monitor
  - The current thread waits notification on the monitor
  - When this Thread receives a notification → it re-demands the monitor to enter the synchronization region
- **notify()** is invoked on obj to wake up Thread waiting on the obj's monitor
- **Remark 1** : notification may be lost if the thread does not in sleep
  - Synchronization condition → in the synchronization region
  - Notify() → the last statement in the synchronization region
- **Remark 2** : sleep() & yield() do not free the monitor

Thread Synchronization (Ideal Example)

```
class CircularCounter{
        private int max, value =0;
        public CircularCounter(int max) { this.max = max;}
        public int getValue() {return value;}
        public synchronized void inc() {
                value = (value + 1) % max;
        }
}
```

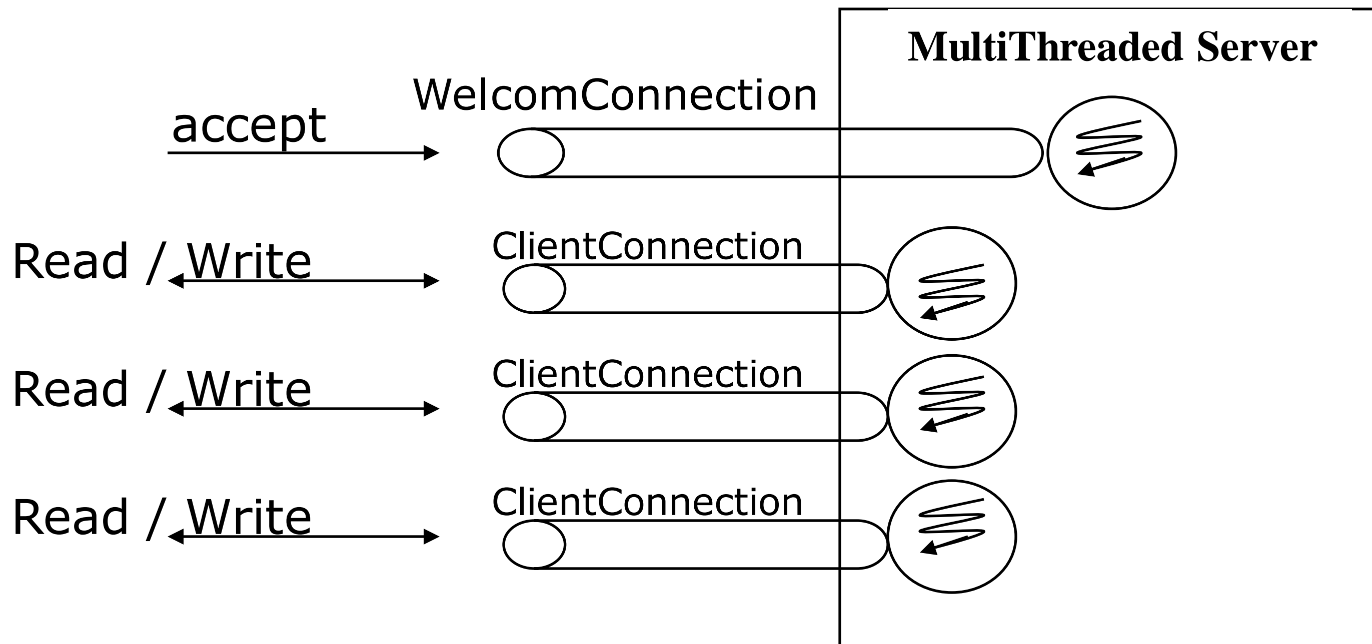
```
public class Test{
        public static void main(String[] args) {
        CircularCounter c = new CircularCounter(5);
        NumberedThread th0 = new NumberedThread(c, 0);
        …..
        NumberedThread th4 = new NumberedThread(c, 4);

        th0.start();
        …
        th4.start();
        }
}
```

```
class NumberedThread extends Thread{
        private CircularCounter c;
        private int ord;
        public NumberedThred (CircularCounter c, int ord) {
        this.c =c; this.ord = ord;
        }

        public void run() {
                while(true){
                synchronized(c){
                while(ord != c.getValue()){
                                try{ c.wait();}
                                catch(Exception e) {}}
                                S.O.P("T"+ord);
                                c.inc();
                                c.notifyAll();
                }
                }
                }
        }
}
```

Thread per Connection



Thread per Connection

```
import java.net.*;
class ClientManager extends Thread{
        Socket client;
        public ClientManager(Socket s)
        {
                client = s;
        }
        public void run() {
                // … process the client request
                // get input & output streams
                client.close();
                }
        }
}
```

```
import java.net.*;
public class MultithreadedTCP{
        public static void main(String[] args) {
                ServerSocket server = new ServerSocket(port);

                while(true) {
                                S.O.P("Server is ready …");
                                Socket client = server.accept();
                                new ClientManager(client). start();
                }
                server.close();
        }
}
```

Thread Pools

- Threads تحسن من أداء النظام خصوصاً مع I/O
- Threads لها كلفة (لدى الإنشاء و الإزالة و المبادلة) خصوصاً عندما : تنتهي بسرعة و/أو نخصص الآلاف منها بالدقيقة
- Thread Pools=
  - الـ threads لا تموت بعد إنهاء عملها (لا حاجة لإقلاع الـ Threads)
  - المهام في رتل و الـ Threads تبحث عن مهمة في الرتل لإنجازها
  - Method 1 =
    - إقلاع عدد ثابت من الـ Threads
    - task pool فارغة فإن الـ Threads تنتظر على الـ pool
    - Thread تنهي عملها تعود إلى Task pool للبحث عن مهمة جديدة
  - Method 2 =
    - Threads في الـ pool و البرنامج الرئيسي يسحب الـ Threads و يربطهم بالمهام
    - يمكن إضافة Thread جديدة عندما لا Threads بالـ pool و لدينا مهمة ضرورية

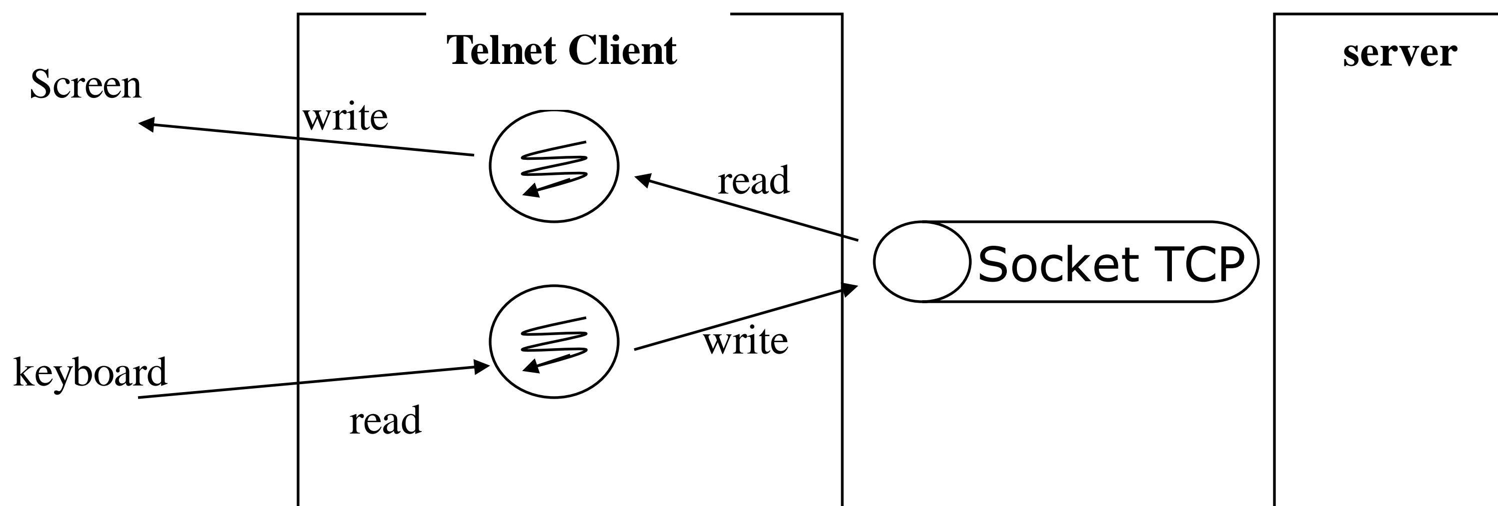
Thread Pools

```
Class WorkQueue{
        public synchronized void add (Socket s) {
                queue.addLast(r);
                queue.notifyAll();
        }
}
```

```
Class TCPThreadPool{
    public static void main(String[] args){
        WorkQueue q = new WorkQueue();
        threads = new PoolWorker[nThreads];
        for(int i=0; i<nThreads; i++){
                threads[i] = new PoolWorker(q);
                threads[i].start();
        }
    ServerSocket ws = new ServerSocket(6789);
    while(true){
        Socket s = ws.accept();
        q.add(s);
        }
    }
}
```

```
class PoolWorker extends Thread{
    WorkQueue queue;
    public PoolWorker(WorkQueue queue){
        this.queue =queue;
    }
    public void run() {
        Runnable r;
        while(true){
            synchronized(queue){
                while(queue.isEmpty()){
                    try{queue.wait();}
                    catch(InterruptedException e) {}
                }
                s = (Socket) queue.removeFirst();       }
            try{DataInputStream is = ..;
                DataOutputStream = ….;
                //the problem protocol}
            catch(RunTimeException ex) {}
        }
    }
}
```

Telnet Client



Telnet Client

```
import java.io.*;
class RedirectStream extends Thread{
	BufferedReader rs = null;
	PrintStream ws = null;
	public RedirectStream(InputStream is, OutputStream os)
	{
		rs = new BufferedReader(is);
		ws = new PrintStream(os);
		start();
	}
	public void run() {
		String s;
		while((s=rs.readLine()) != null) {
				ws.println(s);
		}
	}
}
```

```
import java.io.*;
import java.net.*;
public class Telnet{
	public static void main(String[] args) {
		Socket s = new Socket(TelnetServer, port);

		RedirectStream SktToScreen = new
		RedirectStream(s.getInputStream(), System.out);

		RedirectStream KeyboardToSkt = new
		RedirectStream(System.in, s.getOutputStream());

		SktToScreen.join();
		keyboardToSkt.Interrupt();
		}
}
```

Drafts …………

Thread Pools

```
Class WorkQueue{
        private final int nThreads;
        private final PoolWorker[] threads;
        private LinkedList queue;

        public WorkQueue(int nThreads){
                this.nThreads = nThreads;
                queue = new LinkedList();
                threads = new PoolWorker[nThreads];
                for(int i=0; i<nThreads; i++){
                                threads[i] = new PoolWorker();
                                threads[i].start();
                }
        }
        public void execute (Runnable r) {
                synchronized(queue) {
                                queue.addLast(r);
                                queue.notifyAll();
                }
        }
}
```

```
class PoolWorker extends Thread{
        public void run() {
                Runnable r;
                while(true){
                    synchronized(queue){
                        while(queue.isEmpty()){
                            try{queue.wait();}
                            catch(InterruptedException e) {}
                        }
                        r = (Runnable) queue.removeFirst();
                    }
                    try{r.run();}
                    catch(RunTimeException ex) {}
                }
        }
}
```