# Distributed Systems
# Concurrent Programming

**Gianpaolo Cugola**
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
cugola@elet.polimi.it
http://home.dei.polimi.it/cugola

http://corsi.dei.polimi.it/distsys

# Bibliography

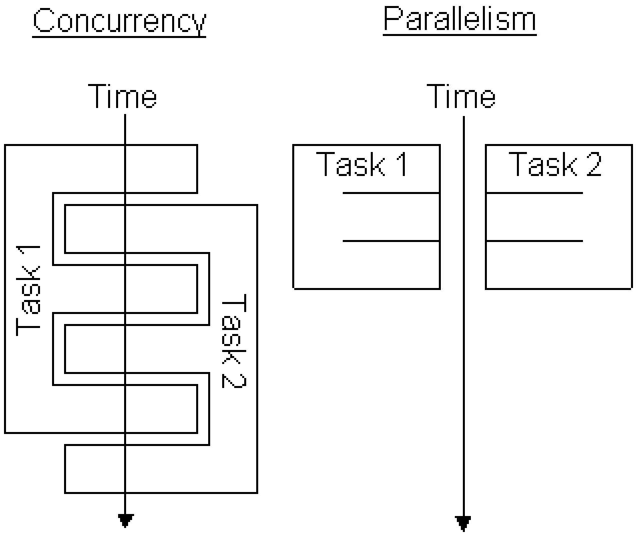
- http://www.llnl.gov/computing/tutorials/pthreads/
- http://users.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html
- Bruce Eckel, Thinking in Java (4th Edition)
- Doug Lea, Concurrent Programming in Java: Design Principles and Patterns (II edition, 1999)
- SUN Tutorial http://java.sun.com/docs/books/tutorial
- http://www.mcs.drexel.edu/~shartley/ConcProgJava
- http://www-dse.doc.ic.ac.uk/concurrency

# Contents

- Introduction to concurrency
- Concurrent programming in C
  - The pthreads
  - Synchronizing threads: Mutex and condition variables
- Concurrent programming in Java

# Concurrency

- Concurrency is an important area of computer science studied in different contexts: machine architectures, operating systems, distributed systems, database, etc
- Objects provide a way to divide a program into independent sections. Often, you also need to turn a program into separate, independently running subtasks
- Concurrency may be *physical* (parallelism) if each unit is executed on a dedicated processor or *logical* if the CPU is able to switch from one to another so that all units appear to progress simultaneously



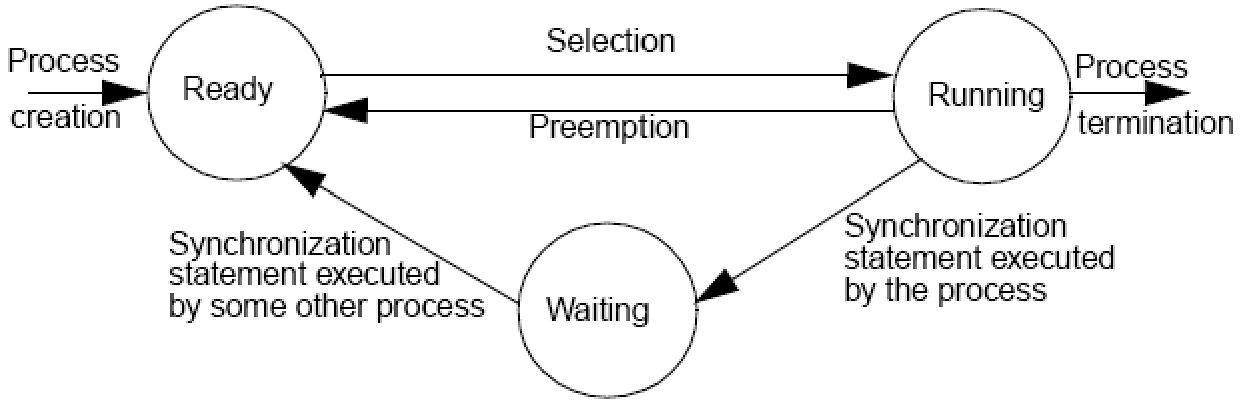
# Process & Thread

- A *process* is a self-contained running program with its own address space
  - A *multitasking operating system* is capable of running more than one process at a time by periodically switching the CPU from one task to another
- The term *thread* (a.k.a. lightweight process) instead is used when the concurrent units share a single address space

Politecnico di Milano

# Preemption

- Modern systems adopts a *preemptive* model
  - It forces a process to abandon its running state even if it could safely execute
- Usually a *time slicing* mechanism is employed where a process is suspend when a specified amount of time has expired


Process creation
Ready
Selection
Preemption
Running
Process termination
Synchronization statement executed by some other process
Waiting
Synchronization statement executed by the process

# Contents

- Introduction to concurrency
- **Concurrent programming in C**
  - **The pthreads**
  - **Synchronizing threads: Mutex and condition variables**
- Concurrent programming in Java

# Concurrency in C (in Unix/Linux)

- All Unix's are multitasking, preemptive operating systems
- Under the traditional Unix model, processes are all created in the same way, by using the `fork()` system call.
- `Fork()` produces a second copy of the calling process. This second copy is identical to the original, except that the return value of `fork()` in the child is 0, while the return value of `fork()` in the parent is the child's `pid`. So the normal use of fork() is:

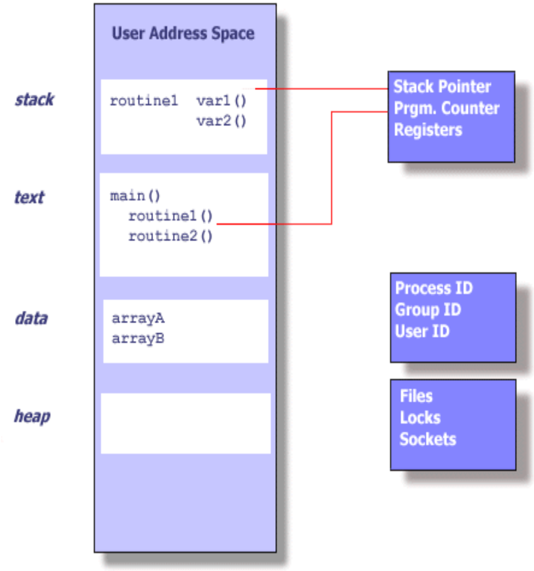
```
[do parent stuff]
ppid = fork ();
if (ppid < 0) {
        fork_error_function ();
} else if (ppid == 0) {
        child_function ();
} else {
        parent_function ();
}
```

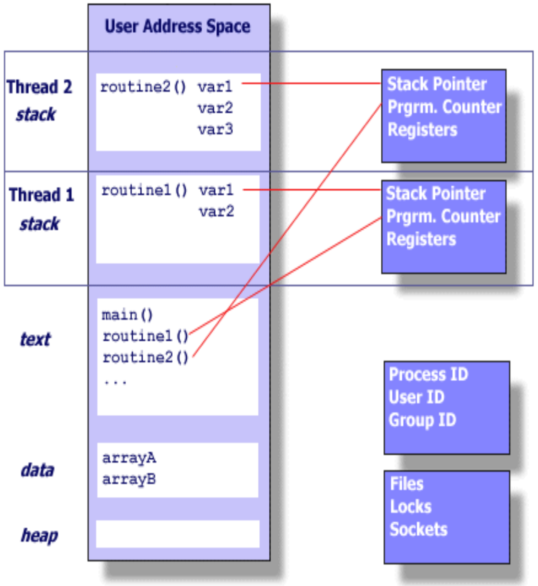
# From fork() to pthreads

- fork() returns two completely independent copies of the original process
- This independence, while providing memory protection and therefore stability, causes problems when you want to have multiple processes working on the same task/problem
  - You can use pipes or SysV IPC, but they are complex to use
- Moreover:
  - The cost of switching between multiple processes is relatively high
  - There are often severe limits on the number of processes the scheduler can handle efficiently
  - Synchronization variables, shared between multiple processes, are typically slow
- Threads share a common address space, and are often scheduled internally in a process, thereby avoiding a lot of the inefficiencies of multiple processes
- One very popular API for threading a Unix application is pthreads, also known as POSIX threads

# Pthread basics

- Thread operations include thread creation, termination, synchronization (joins), scheduling, data management, and process interaction
- Threads in the same process share: Process instructions, most data, open files (descriptors), signals and signal handlers, current working directory, user and group id
- Each thread has a unique: Thread ID, set of registers, stack pointer, stack, signal mask, priority





# Thread creation

- Syntax

```
int pthread_create (pthread_t *id, const
  pthread_attr_t *attr, void *(*routine)(void *),
  void *arg)
```

- Arguments
  - `id`: An opaque, unique identifier for the new thread returned by the subroutine
  - `attr`: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values
  - `routine`: the C routine that the thread will execute once it is created
  - `arg`: A single argument that may be passed to `routine`. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed

# Thread termination

- There are several ways in which a pthread may be terminated:
  - The thread returns from its starting routine
  - The thread makes a call to the `pthread_exit` subroutine
  - The thread is canceled by another thread via the `pthread_cancel` routine (not covered)
  - The entire process is terminated due to a call to either the exec or exit subroutines.
- With `pthread_exit` the programmer may optionally specify a termination status, which is stored as a void pointer for any thread that may join the calling thread
  - The `pthread_exit` routine does not close files; any files opened inside the thread will remain open after the thread is terminated
- Important notice:
  - If main() finishes before the threads it has created, and exits with `pthread_exit`, the other threads will continue to execute. Otherwise, they will be automatically terminated when main() finishes.

# Example: Thread creation

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid) {
   printf("\n%d: Hello World!\n", threadid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0; t<NUM_THREADS; t++){
      printf("Creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

# Exercise

- Modify the previous program to check wheather pthreads are managed in time slicing or cooperative mode in Linux

# Example: Thread argument passing (incorrect)

- How to pass argument to threads?
- Example:

```
int rc, t;
for(t=0; t<NUM_THREADS; t++) {
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL,
        PrintHello, (void *) &t);
   ...
}
```

- Wrong: The loop which creates threads modifies the contents of the address passed as an argument, possibly before the created threads can access it

# Example: Thread argument passing (correct)

- Correct solution: The calling thread uses a unique data structure for each thread, insuring that each thread's argument remains intact throughout the program

```
int *taskids[NUM_THREADS];
for(t=0; t<NUM_THREADS; t++) {
   taskids[t] = (int *) malloc(sizeof(int));
   *taskids[t] = t;
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL,
        PrintHello, (void *) taskids[t]);
   ...
}
```

# Joining threads

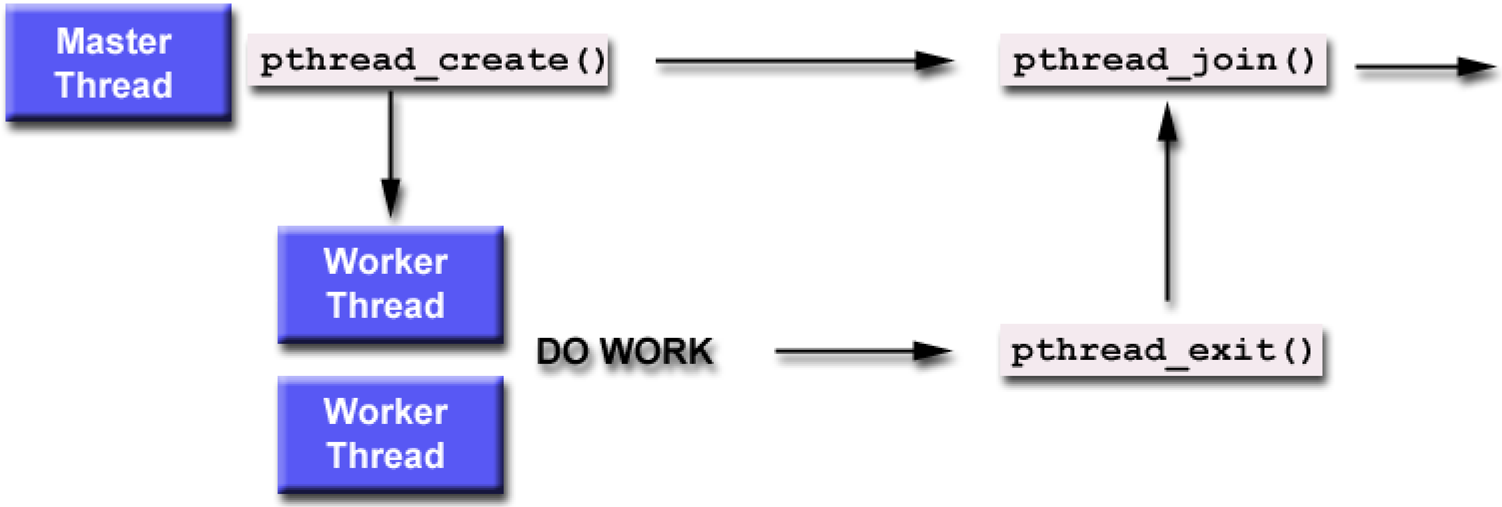
- "Joining" is one way to accomplish synchronization between threads
- Syntax

```
int pthread_join(pthread_t thid,
  void **thread_return)
```

- Semantics
  - Blocks the calling thread until the specified `thid` thread terminates
  - Through the `thread_return` argument it is possible to obtain the target thread's termination return status (if it was specified in the target thread's call to `pthread_exit`)

Politecnico
di Milano

# Joining threads



# Thread synchronization: Mutex variables

- A mutex variable acts like a "lock" protecting access to a shared data resource
- The basic concept of a mutex as used in pthreads is that only one thread can lock (or own) a mutex variable at any given time
  - Even if several threads try to lock a mutex only one thread will be successful
  - No other thread can own that mutex until the owning thread unlocks that mutex
- When protecting shared data, it is the programmer's responsibility to make sure every thread that needs to use a mutex does so
  - For example, if 4 threads are updating the same data, but only one uses a mutex, the data can still be corrupted

# Creating and destroying mutexes

- Mutex variables must be declared with type `pthread_mutex_t`
- Two ways to initialize a mutex variable:
  - Statically, when it is declared

    ```
    pthread_mutex_t mymutex =
      PTHREAD_MUTEX_INITIALIZER;
    ```

  - Dynamically, with the `pthread_mutex_init` routine
    - This method permits setting mutex object attributes
    - The `pthread_mutexattr_init` and `pthread_mutexattr_destroy` routines are used to create and destroy mutex attribute objects, respectively
- In both cases the mutex is initially unlocked
- The `pthread_mutex_destroy` routine should be used to free a mutex object which is no longer needed

# Locking and unlocking mutexes

- The `pthread_mutex_lock` routine is used by a thread to acquire a lock on a mutex variable
  - If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.
- The `pthread_mutex_trylock` routine will attempt to lock a mutex
  - If the mutex is already locked, the routine will return immediately with a "busy" error code
  - Useful in preventing deadlock conditions
- The `pthread_mutex_unlock` routine will unlock a mutex if called by the owning thread
  - An error will be returned if the mutex was already unlocked, or the mutex is owned by another thread

# Example: Using mutexes

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  double *a;
  double *b;
  double sum;
  int    veclen;
} DOTDATA;

/* Define global variables and a mutex */
#define NUMTHRDS 4
#define VECLEN 100
DOTDATA dotstr;
pthread_t callThd[NUMTHRDS];
pthread_mutex_t mutexsum;

void *dotprod(void *arg) {
  int i, start, end, offset;
  double mysum;
  offset = (int)arg;
  start  = offset * dotstr.veclen;
  end    = start + dotstr.veclen;
  mysum = 0;
  for (i=start; i<end ; i++) {
    mysum += (dotstr.a[i] * dotstr.b[i]);
  }
  pthread_mutex_lock (&mutexsum);
  dotstr.sum += mysum;
  pthread_mutex_unlock (&mutexsum);
  pthread_exit((void*) 0);
}
```

```c
int main (int argc, char *argv[]) {
  int i;
  int status;
  /* Assign storage and initialize values */
  dotstr.a = (double*) malloc
     (NUMTHRDS*VECLEN*sizeof(double));
  dotstr.b = (double*) malloc
     (NUMTHRDS*VECLEN*sizeof(double));
  for (i=0; i<VECLEN*NUMTHRDS; i++) {
    dotstr.a[i]=1;
    dotstr.b[i]=1;
  }
  dotstr.veclen = VECLEN;
  dotstr.sum=0;
  /* initialize the mutex */
  pthread_mutex_init(&mutexsum, NULL);
  /* Create threads to perform the dotproduct
     */
  for(i=0;i<NUMTHRDS;i++) {
    pthread_create(&callThd[i], NULL, dotprod,
     (void *)i);
  }
  /* Wait on the other threads */
  for(i=0;i<NUMTHRDS;i++) {
    pthread_join( callThd[i], (void
     **)&status);
  }
  /* After joining, print out the results */
  printf ("Sum =  %f \n", dotstr.sum);
  free (dotstr.a);
  free (dotstr.b);
  pthread_mutex_destroy(&mutexsum);
  pthread_exit(NULL);
}
```

# Exercise

- Implement (the scheleton) of a multithreaded TCP server which registers the highest temperature in a building
  - Each time a new TCP connection is opened (from the sensors monitoring each office) a new thread is spawned
  - Each time a new temperature is reported the thread checks if it is greater than the greatest one (saved in a global variable) and updates it as necessary
  - Periodically (once a day), the main thread writes down the highest temperature reported and set it to 0 to restart the calculation
  - Simulates connections by creating a fixed number of threads from the very beginning
  - Simulate temperature readings by periodically creating random numbers --- use `random()`

# Thread synchronization: Condition variables

- While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data
- Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met
  - A solution that can be very resource consuming since the thread would be continuously busy in this activity
- A condition variable is a way to achieve the same goal without polling
- A condition variable is always used in conjunction with a mutex lock

# Using condition variables

### Main Thread

- Declare and initialize global data/variables which require synchronization (such as "count")
- Declare and initialize a condition variable object
- Declare and initialize an associated mutex
- Create threads A and B to do work

### Thread A

- Do work up to the point where a certain condition must occur (e.g. count reaches a specific value)
- Lock associated mutex and check value of a global variable
- Call `pthread_cond_wait` to perform a blocking wait for signal from Thread-B
  - Note that a call to `pthread_cond_wait` automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B
- When signalled, wake up
  - Mutex is automatically and atomically locked
- Explicitly unlock mutex
- Continue

### Thread B

- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon
- Check value of the variable. If it fulfills the desired condition, signal Thread-A.
- Unlock mutex.
- Continue

### Main Thread

- Join and continue

# Creating and destroying condition variables

- Condition variables must be declared with type `pthread_cond_t`
- Two ways to initialize a condition variable:
  - Statically, when it is declared

    ```
    pthread_cond_t myconvar =
      PTHREAD_COND_INITIALIZER;
    ```

  - Dynamically, with the `pthread_cond_init` routine
    - This method permits setting condition variable object attributes
    - The `pthread_condattr_init` and `pthread_condattr_destroy` routines are used to create and destroy condition variable attribute objects
- The `pthread_cond_destroy` routine should be used to free a condition variable that is no longer needed

# Waiting and signaling condition variables

- The `pthread_cond_wait` routine blocks the calling thread until the specified condition is signalled
  - Must be called while mutex is locked, and it will automatically release the mutex while it waits
  - After signal is received and thread is awakened, mutex will be automatically locked for use by the thread
  - The programmer is then responsible for unlocking mutex when the thread is finished with it
- The `pthread_cond_signal` routine signal (wake up) another thread which is waiting on the condition variable
  - It should be called after mutex is locked, and must unlock mutex in order for `pthread_cond_wait` routine to complete.
- The `pthread_cond_broadcast` routine should be used instead of `pthread_cond_signal` if more than one thread is in a blocking wait state
- It is a logical error to call `pthread_cond_signal` before calling `pthread_cond_wait`

# Exercises

- Use condition variables to synchronize a producer with a consumer thread using a single integer buffer
- Do the same with multiple producers
- Use a larger buffer

# Contents

- Introduction to concurrency
- Concurrent programming in C
  - The pthreads
  - Synchronizing threads: Mutex and condition variables
- **Concurrent programming in Java**

# Concurrency in Java

- Java supports concurrency at the *language level* rather than through run time libraries (like pthreads for C/C++)
- It provides classes to instantiate and run new threads
- Plus methods to synchronize threads…
- … and to implement condition variables

# The Thread class

- The simplest way to create a thread is to inherit from `java.lang.Thread.` It has all the wiring necessary to create and run threads.
  - The most important method for `Thread` is `run()`

The thread object is defined by extending the Thread class

You must override run(), to make the thread do your bidding

Thread object is instantiated as usual through a new()

To run the thread you must invoke the start() method on it

```
public class MyThread extends Thread {
  private String message;
  public MyThread(String m) {message = m;}
  public void run() {
    for(int r=0; r<20; r++)
      System.out.println(message);
  }
}

public class ProvaThread {
  public static void main(String[] args) {
    MyThread t1,t2;
    t1=new MyThread("primo thread");
    t2=new MyThread("secondo thread");
    t1.start();
    t2.start();
  }
}
```

# Runnable interface

- You can use the alternative approach of implementing the `Runnable` interface. `Runnable` specifies only that there be a `run()` method implemented, and `Thread` also implements `Runnable`

Your class must implement Runnable interfaces

run() must be overridden

To produce a thread from a Runnable object, you must create a separate Thread object

start() method is invoked to execute the thread

```
public class MyThread implements Runnable {
  private String message;
  public MyThread(String m) {message = m;}
  public void run() {
    for(int r=0; r<20; r++)
      System.out.println(message);
  }
}

public class ProvaThread {
  public static void main(String[] args) {
    Thread t1, t2;
    MyThread r1, r2;
    r1 = new MyThread("primo thread");
    r2 = new MyThread("secondo thread");
    t1 = new Thread(r1);
    t2 = new Thread(r2);
    t1.start();
    t2.start();
  }
}
```

# Exercise

- Implement a multi-threaded program to compute the matrix product
- Each thread is responsible for a different line of the resulting matrix
- Use `join()` in the main program to wait all threads to finish before printing out the resulting matrix
  - The `join()` method is used to wait until thread is done
  - The caller of `join()` blocks until the thread finishes

# Non-determinism

- Threads execution proceeds without a predefined order
- The same code, if run on different computers, could produce different output
  - It depends on how the internal scheduling is performed, on processor features, …
- Such behavior is defined as *non-deterministic*
- Non-determinism is a *key point* in concurrency
  - It is what makes it so hard to handle concurrency

# The Java concurrency model

- Java employs a *preemptive* model
- If time-slicing is available (implementation dependent), Java ensures equal priority threads execute in round-robin fashion otherwise they run to completion
- Priority is handled differently according to the specific implementation
- Threads may explicitly relinquish control to the JVM scheduler through the yield() method

# Correctness

- A concurrent system is correct if and only if it owns the following properties:
  - *Safety*: bad things do not happen
  - *Liveness*: good things eventually happen
- Safety failures lead to unintended behavior at run time — things just start going wrong
  - Read / write conflicts
  - Write / write conflicts
- Liveness failures lead to no behavior — things just stop running
  - Locking
  - Waiting
  - I/O
  - CPU contention
  - Failure
- Sadly enough, some of the easiest things you can do to improve liveness properties can destroy safety properties, and vice versa (e.g. locking)

# J2SE 5 (a.k.a Tiger)

- J2SE5 provides a number of enhancements such as support for Metadata, Generics, Enumerated types, Autoboxing of primitive types
- It improves support for concurrency too
- Package `java.util.concurrent` includes versions of the utilities described hereafter, plus some others
- `http://java.sun.com/developer/technicalArticles/J2SE/concurrency/`

# Exclusion

- In a safe system, every object protects itself from integrity violations
- Exclusion techniques preserve object invariants and avoid effects that would result from acting upon even momentarily inconsistent state representations
- Three strategies in presence of multithreading:
  - Immutability
  - Dynamic Exclusion (locks)
  - Structural Exclusion

# Immutability

```
class StatelessObject{
  public static int add(int a, int b) {
         return a + b;
  }
}
```

Stateless Objects

```
class ImmutableAdder {
 private final int offset;

 public ImmutableAdder(int a) {
        offset = a;
}
 public int addOffset(int b) {
        return offset + b; }
}
```

Immutable Objects (e.g., **Integer**)

- PROs & CONs
  - ✓ No need for synchronizing
  - ✓ Value containers (a new object is provided when value is changed)
  - ✓ Useful for sharing objects among threads
  - ✘ Limited applicability

# Synchronization

```
public class RGBColor {
    private int r;
    private int g;
    private int b;

    public void setColor(int r, int g, int b) {
        checkRGBVals(r,  g, b);
        this.r = r;
        this.g = g;
        this.b = b;
    }
```

- Imagine you have two threads, one thread named "red" and another named "blue". Both threads are trying to set their color of the same `RGBColor` object
- If the thread scheduler interleaves these two threads in just the right way, the two threads will inadvertently interfere with each other, yielding a write/write conflict. In the process, the two threads will corrupt the object's state

# Objects and locks

- Locking serializes the execution of *synchronized* methods
- Every instance of a reference type (i.e., a class) possess a lock
  - Inherited from the `Object` class
- Scalar fields can be locked only via their enclosing objects
- Locking and arrays: Locking an array does not lock its elements

# Synchronized blocks and methods

- Entire methods or single blocks can be synchronized
  - Block synchronization takes an argument of which object to lock
  - Method synchronization uses the lock of the called object (this)

```
synchronized (object){
     // Lock is held
     …
}
// Lock is released
```

```
synchronized void f() {
     // Lock is held
     /* body */
}
// Lock is released
```

↓ Equivalent to

```
void f() {
     synchronized(this) {
          /* body */
     }
}
```

# Acquiring locks

- A lock is acquired *automatically* on entry to a `synchronized` method or block, and released on exit, even if the exit occurs due to an exception
- Locks operate on a per-thread, not per-invocation basis. A thread hitting synchronized passes if the lock is free or if the thread already possess the lock, otherwise it blocks
- A synchronized method or block obeys the acquire-release protocol only with respect to other synchronized methods and blocks on the same target object
  - Methods that are not synchronized may still execute at any time, even if a synchronized method is in progress
- Synchronized is not equivalent to atomic, but synchronization can be used to achieve atomicity

# Exercise

- Consider the following class

```
class Even {
 private int n = 0;
 public int next(){
   // POST?: next is always even
   ++n;
   Thread.sleep(500);
   ++n;
   return n;
  }
 }
```

- If multiple threads access `Even`, post-conditions may not be kept
- Write the unsynchronized and synchronized version of a program with two threads calling next concurrently

# `synchronized` & inheritance

- The `synchronized` keyword is not considered to be part of a method's signature:
  - The `synchronized` modifier is not automatically inherited when subclasses override superclass methods
  - Methods in interfaces cannot be declared as `synchronized`
- Synchronization in an inner class method is independent of its outer class
  - However, a non-static inner class method can lock its containing class, say `OuterClass`, via code blocks using:

```
synchronized(OuterClass.this) { /* body */ }
```

- Static synchronization employs the lock possessed by the `Class` object associated with the class the static methods are declared in
  - The static lock for class `C` can also be accessed inside instance methods via:

```
synchronized(C.class) { /* body */ }
```

# Key rules

1. Always lock during updates to object fields

```
synchronized (point) {
 point.x = 5; point.y = 7;
}
```

2. Always lock during access of possibly updated object fields

```
synchronized(point) {
 if(point.x > 0) {...}
}
```

3. You do not need to synchronize stateless parts of methods

```
public synchronized void f(){
  state = ...;
  operation();
 }
```

```
public void f(){
  synchronized(this) {
    state = ...;
  }
  operation();
}
```

4. Never lock when invoking methods on other objects

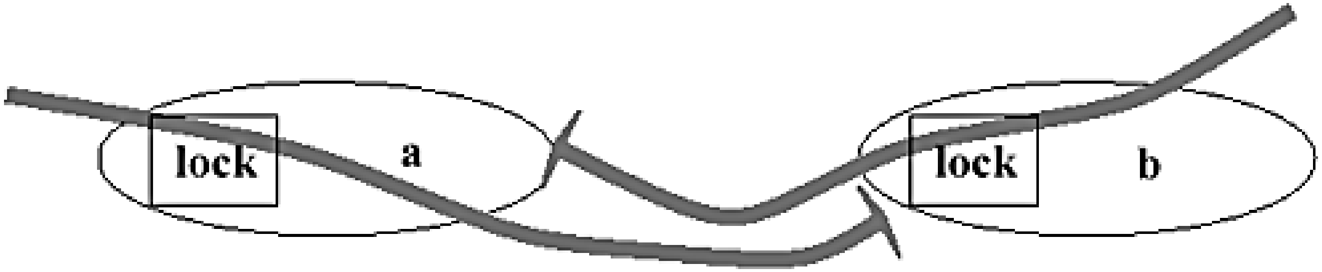
```
public synchronized void f(){
  ...
  h.foo();
}
```

```
public void f(){
  synchronized(this) {
    ...;
  }
  h.foo();
}
```

# Fully synchronized objects

- The safest (but not always the best) concurrent OO design strategy based on locking is to restrict attention to fully synchronized objects (also known as *atomic objects*) in which:
  - All methods are synchronized
  - There are no public fields or other encapsulation violations
  - All methods are finite (no infinite loops or unbounded recursion), and so eventually release locks
  - All fields are initialized to a consistent state in constructors
  - The state of the object is consistent (obeys invariants) at both the beginning and end of each method, even in the presence of exceptions

# Deadlock



- Deadlock is possible when two or more objects are mutually accessible from two or more threads, and each thread holds one lock while trying to obtain another lock already held by another thread
- Although fully synchronized atomic objects are always safe, they may lead to deadlock

# Deadlock: Example

```
class Cell {  // Do not use
 private long value;
 synchronized long getValue() { return value; }
 synchronized void setValue(long v) { value = v; }

 synchronized void swapValue(Cell other) {
  long t = getValue();
  long v = other.getValue();
  setValue(v);
  other.setValue(t);
 }
}
```

- If two generic instances of `Cell`, say `a` and `b`, are concurrently invoking `swapValue`, the program may block indefinitely because a needs `b`'s lock and vice versa

# Exercise

- Consider the following java classes:

```
class A {

  ...
  synchronized public void ma1(B b){
     ...
     b.mb2();
  }
  synchronized public void ma2() {
     ...
  }
}
```

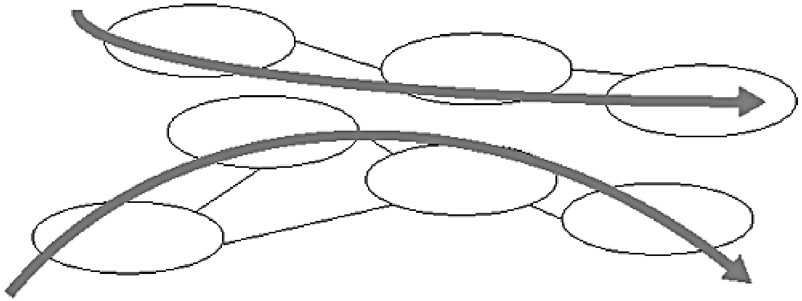
```
class B {

  ...
  synchronized public void mb1(A a){
     ...
     a.ma2();
  }
  synchronized public void mb2() {
     ...
  }
}
```

- How deadlock can arise if two thread, say X and Y, concurrently access to A and B respectively ?
- How can you effectively solve the issue ?

# Assignment & lock

- The act of setting the value of a variable (except for a long or a double) is atomic
- That means there is generally no need to synchronize access simply to set or read the value of a variable
- However, threads are allowed to hold the values of variables in local memory (e.g. in a machine register). In that case, when one thread changes the value of a variable, another thread may not see the changed value (especially true for loop)
- You may synchronize access to that variable or mark it as `volatile`, which means that every time the variable is used, it must be read from main memory

# Structural exclusion



- Confinement employs encapsulation techniques to structurally guarantee that at most one activity at a time can possibly access a given object
- This statically ensures that the accessibility of a given object is unique to a single thread without needing to rely on dynamic locking on each access
- Only one thread, or one thread at a time, can ever access a confined object

# Escaping

- The key point is to avoid references escaping from their owner `Thread`
- There are four categories to check to see if a reference `r` to an object `x` can escape from a method `m` executing within some activity:
  - `m` passes `r` as an argument in a method invocation or object constructor
  - `m` passes `r` as the return value from a method invocation
  - `m` records `r` in some field that is accessible from another activity (in the most flagrant case, static fields that are accessible anywhere)
  - `m` releases (in any of the above ways) another reference that can in turn be traversed to access `r`

# Confinement across Methods

- If a given method invocation creates an object and does not let it escape, then it can be sure that no other threads will interfere with (or even know about) its use of that object

```
class Plotter {    // Fragments
 // ...

 public void showNextPoint() {
  Point p = new Point();
  p.x = computeX();
  p.y = computeY();
  display(p);
 }

 protected void display(Point p) {
  // somehow arrange to show p.
 }
}
```

Tail call hand-off

# Confinement Across Methods - 2

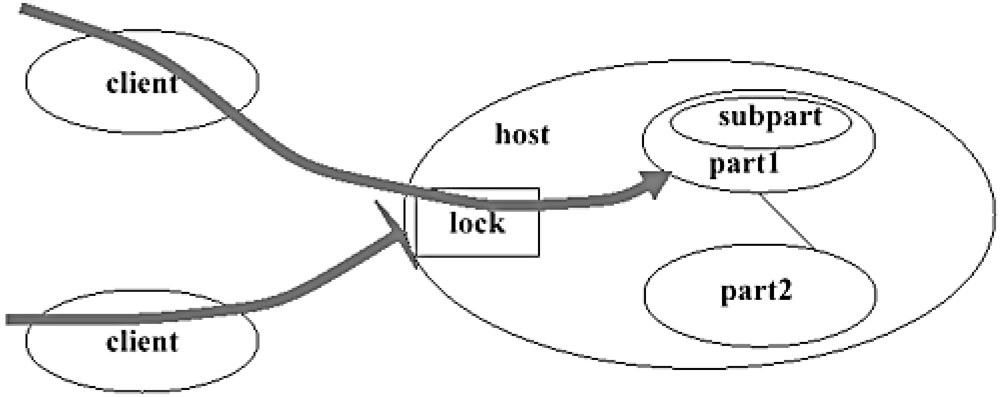
- Tail-call hand-offs do not apply if a method must access an object after a call or must make multiple calls

```
public void showNextPointV2() {
   Point p = new Point();
    p.x = computeX();
    p.y = computeY();
   display(p);
   recordDistance(p); // added }
```

- Three solutions:
  - Caller copies: `display(new Point(p.x, p.y))`
  - Receiver copies: `Point localPoint = new Point(p.x, p.y)`
  - Scalar Arguments: `display(p.x, p.y)`

# Confinement Within Object

- Sometimes it is needed to share objects among threads
- You can confine all accesses internal to that object so that no additional locking is necessary once a thread enters one of its methods



- In this way, the exclusion control for the outer Host container object automatically propagates to its internal Parts

# Host and Parts

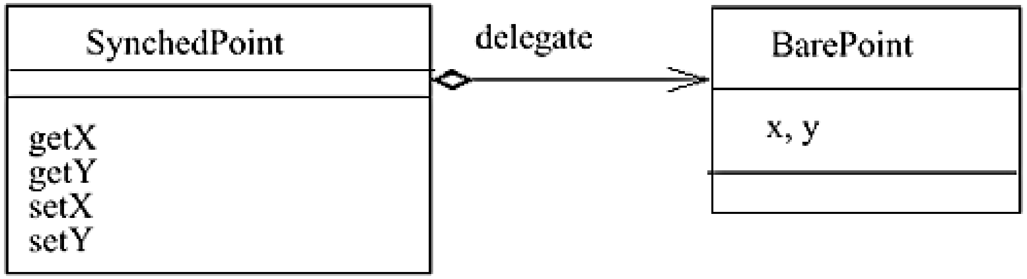
- Host object may be thought of as owning the inner Parts
- Host object constructs new instances of each Part guaranteeing that references to the Part objects are not shared by any other object
- The Host object must never leak references to any Part object:
  - it must never pass the references as arguments or return values of any method, and must ensure that the fields holding the references are inaccessible
- All appropriate methods of the host object are synchronized (while Part's are not)

# Host and Parts: Example

```
class Pixel {
    private final Point pt_;
    Pixel(int x, int y) { pt_ = new Point(x, y); }
    synchronized Point location() {
       return new Point(pt_.x, pt_.y);
    }
    synchronized void moveBy(int dx, int dy){
       pt_.x += dx; pt_.y += dy;
    }
}
```

- `Pixel` provides synchronized access to `Point` methods
  - the reference to `Point` object is immutable, but its fields are in turn mutable (and public!) so is unsafe without protection
- Must make copies of inner objects when revealing state (see `location()`)

# Adapters



```
class SynchedPoint {

 protected final BarePoint delegate =
new BarePoint();

 public synchronized double getX() {
         return delegate.x;}
 public synchronized double getY() {
         return delegate.y; }
 public synchronized void setX(double v)
{        delegate.x = v; }
 public synchronized void setY(double v)
{        delegate.y = v; }
}
```

- Adapters can be used to wrap unsynchronized ground objects within fully synchronized counterparts

# Synchronizing collections

- The `java.util.Collection` framework uses an Adapter-based scheme to allow layered synchronization of collection classes
- Except for `Vector` and `Hashtable`, the basic collection classes (such as `java.util.ArrayList`) are unsynchronized
- However, anonymous synchronized Adapter classes can be constructed around the basic classes using for example:

```
List l = Collections.synchronizedList(new ArrayList());
```

# Why synchronization is not so fine ?

- Invoking a `synchronized` method takes up to 4 times as long as unsynchronized one
- It reduces concurrency and affects performance
  - When many threads all contend for the same entry-point lock, they will spend most of their time waiting for the lock, increasing latencies and limiting opportunities for parallelism
- It may lead to deadlock and prevent liveness

USE WITH CARE

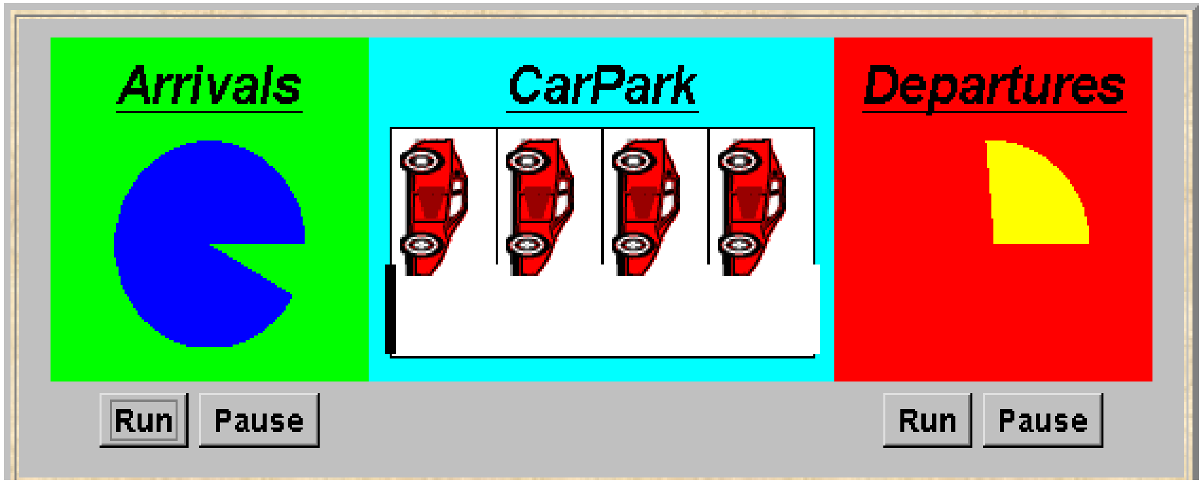
# Other Java synchronization drawbacks

- There is no way to back off from an attempt to acquire a lock if it is already held, to give up after waiting for a specified time, or to cancel a lock attempt after an interrupt
  - Difficult to recover from *liveness* problems
- There is no way to alter the semantics of a lock, for example with respect to reentrancy, read versus write protection, or fairness
- There is no access control for synchronization. Any method can perform `synchronized(obj)` for any accessible object, thus leading to potential denial-of-service problems caused by the holding of needed locks
- Synchronization within methods and blocks limits use to strict block-structured locking. For example, you cannot acquire a lock in one method and release it in another

**Most of these problems have been solved by the new classes introduced in J2SE5**

# Condition synchronization

from *http://www-dse.doc.ic.ac.uk/concurrency/book_applets/CarPark.html*



- A controller is required for a carpark, which only permits cars to enter when the carpark is not full and does not permit cars to leave when there are no cars in the carpark
- Car arrival and departure are simulated by separate threads

Politecnico
di Milano

# CarParkControl monitor

```
class CarParkControl {
   protected int spaces;
   protected int capacity;

   CarParkControl(int n)
      {capacity = spaces = n;}

   synchronized void arrive() {
      …  --spaces; …
      }

   synchronized void depart() {
      … ++spaces; …
      }
}
```

*mutual exclusion by synchronizing methods*

*condition synchronization*

*block if full (spaces==0)*

*block if empty (spaces==N)*

# Condition synchronization in Java

- Java provides a thread wait queue per monitor (actually per object) with the following methods:

```
public final void notify()
```

Wakes up a single thread that is waiting on this object's set

```
public final void notifyAll()
```

Wakes up all threads that are waiting on this object's set

```
public final void wait()
    throws InterruptedException
```

Waits to be notified by another thread. The waiting threa releases the synchronization lock associated with the monitor. When notified, the thread must wait to reacquire the monitor before resuming execution

# Condition synchronization in Java

```
public synchronized void act()
       throws InterruptedException {
  while (!cond) { wait(); }
  // modify state
  notifyAll()
}
```

- The `while` loop is necessary to retest the condition `cond` to ensure that `cond` is indeed satisfied when it re-enters the monitor
- `notifyAll` is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed

# CarParkControl in Java

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() throws InterruptedException {
    while (spaces==0) wait();
    --spaces;
    notifyAll();
  }

  synchronized void depart() throws InterruptedException {
    while (spaces==capacity) wait();
    ++spaces;
    notifyAll();
  }
}
```

# Single notifications

- You can reduce the context-switch overhead associated with notifications by using a single `notify` rather than `notifyAll`
- Single notifications can be used to improve performance when you are sure that at most one thread needs to be woken. This applies when:
  - All possible waiting threads are necessarily waiting for conditions relying on the same notifications, usually the exact same condition (e.g., single producer, several consumers)
  - Each notification intrinsically enables at most a single thread to continue. Thus it would be useless to wake up others (e.g., single producer and consumer)

# Timed waits

- Rather than waiting forever for a condition to become true in a guarded method, time-out designs place a bound on how long any given wait should remain suspended
- Time-outs are typically more useful than other techniques that detect unanticipated liveness problems (such as deadlock) because they make fewer assumptions about contexts

```
synchronized(object) {
 while(!cond) {
   object.wait(time);
 }
}
```

# Busy waits

- Implementing guards via waiting and notification methods is nearly always superior to using an optimistic-retry-style busy-wait "spinloop" of the form:

```
protected void busyWaitUntilCond() {
  while (!cond)
      Thread.yield();
}
```

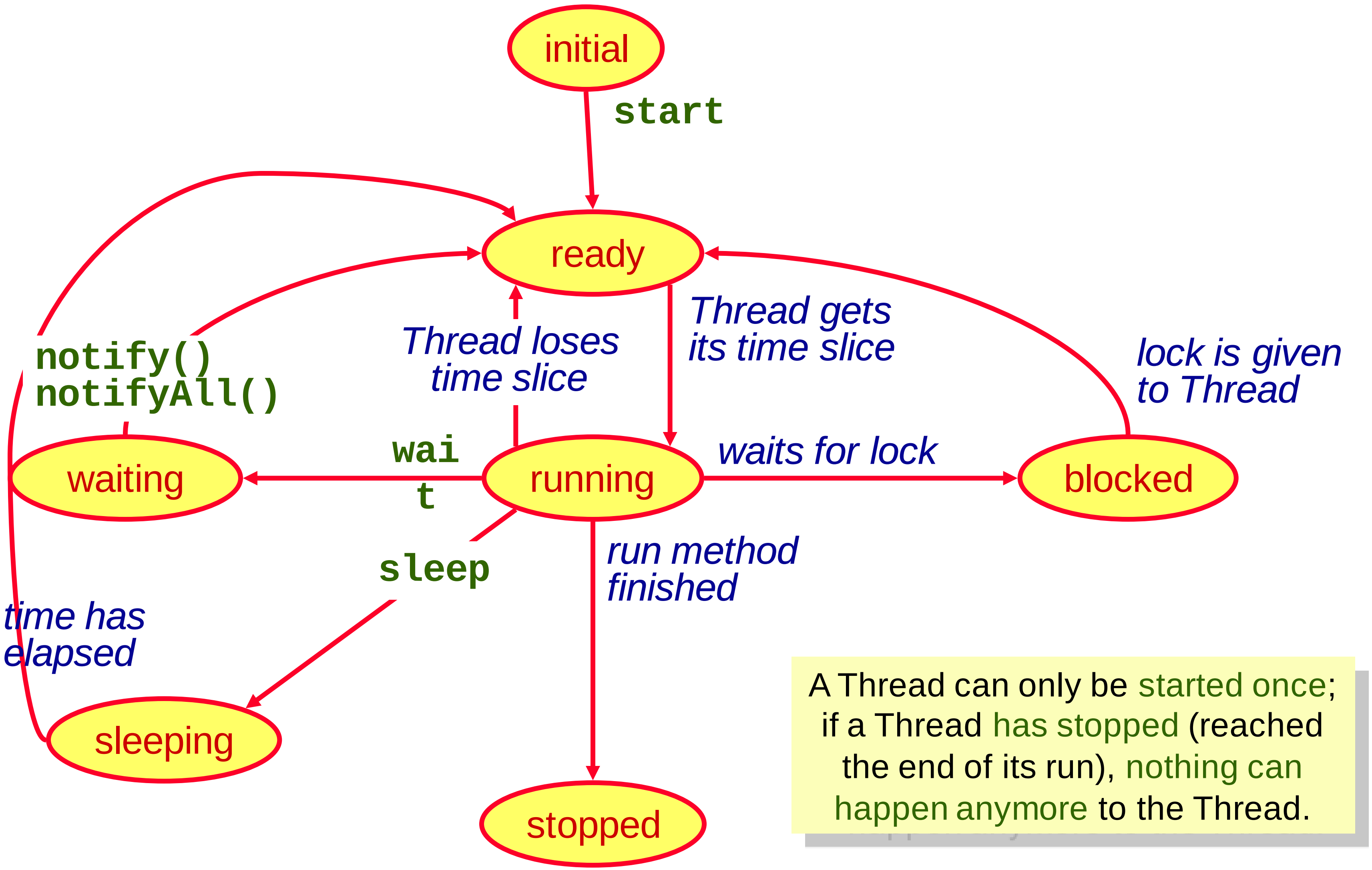
- Reasons are:
  - Efficiency
  - Scheduling
  - Triggering
  - Synchronizing actions
  - Implementations

# sleep()

- Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds
- The thread *does not* lose ownership of any monitors
- Useful for:
  - Periodic actions
    - Need to wait for some period of time before doing the action again
  - Allow other threads to run

Politecnico di Milano

# Threads' transition states

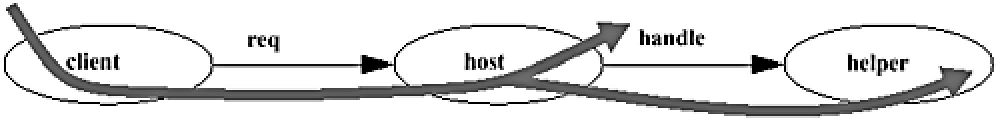


A Thread can only be started once; if a Thread has stopped (reached the end of its run), nothing can happen anymore to the Thread.

# Exercise

- Implement the scheleton of a multithreaded printer spooler
  - A thread for each connection
  - Three type of requests
    - to append a new job
    - to remove an existing job
    - to have a list of jobs
  - A single "printing" thread
    - Extracts the first job from the queue and prints it

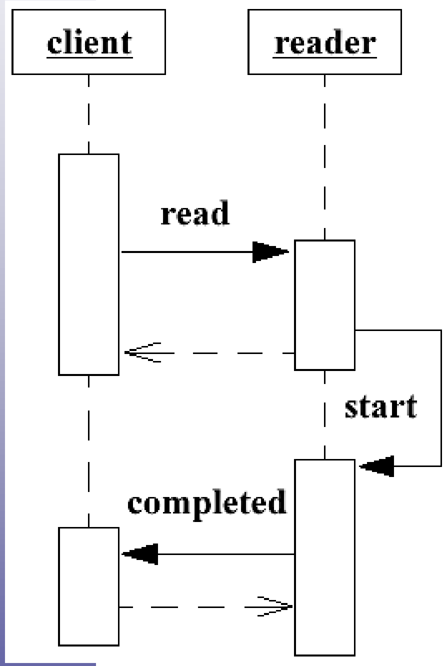
# Thread per message



```
class ThreadPerMessageHost {
 protected long localState;
 protected final Helper helper = new Helper();

 protected synchronized void updateState() {
  localState = ...;
 }

 public void req(...) {
  updateState(...);
  new Thread(new Runnable() {
   public void run() {
    helper.handle(...);
   }
  }).start();
 }
}
```

# Completion callbacks

```
interface FileReader {
  void read(String filename, FileReaderClient client);
}
interface FileReaderClient {
  void readCompleted(String filename, byte[] data);
  void readFailed(String filename, IOException ex);
}
```



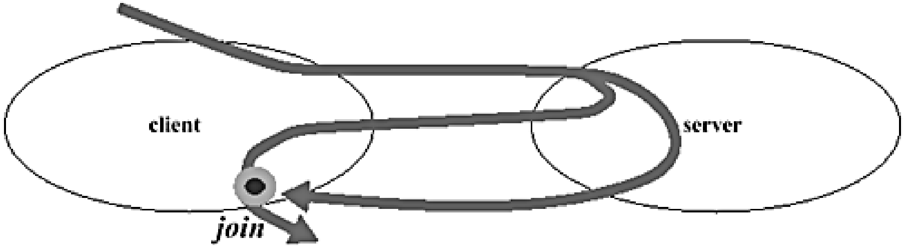
- The most natural way to deal with completion is for a client to activate a task via a oneway message to a server, and for the server later to indicate completion by sending a oneway callback message to the caller

# Completion callbacks

```
class FileReaderApp implements FileReaderClient {protected
FileReader reader = new AFileReader();

 public void readCompleted(String filename, byte[] data) {…}
 public void readFailed(String filename, IOException ex){…}
 public void actionRequiringFile() {
  reader.read("AppFile", this);
 } }
```

```
class AFileReader implements FileReader {
 public void read(final String fn, final FileReaderClient c){
  new Thread(new Runnable() {
   public void run() { doRead(fn, c); }
  }).start();
 }
 protected void doRead(String fn, FileReaderClient client) {
  byte[] buffer = new byte[1024]; // just for illustration
  try {
    FileInputStream s = new FileInputStream(fn);
    s.read(buffer);
    if (client != null) client.readCompleted(fn, buffer);
  }
  catch (IOException ex) {
   if (client != null) client.readFailed(fn, ex);
  }}}
```

# Joining threads



- While completion callbacks are very flexible, they are awkward to use when a caller just needs to wait out a particular task that it started
- This functionality is provided by `Thread.join`:
  - The `join` method blocks the caller while the target `isAlive`
  - Terminating threads automatically perform notifications
- Either `join` or explicitly coded variants can be used in designs where a client needs a service to be performed but does not immediately rely on its results or effects (a.k.a. *deferred-synchronous invocation*)
  - Similar to RPC-promise

# Exercise

- A server has a route(Message m) method, which asynchronously (i.e., starting a new thread) route the given message
  - It must return to the caller the number of neighbors who received the message
  - The caller does not need such number until a reply to the message arrive
- How to implement such interaction limiting synchronization to a minimum?

# Exercise

- Consider a cluster of PCs used for scientific simulations:
  - Each simulation is represented by a class `Task` extending `Runnable`
  - At most N tasks can be accepted simultaneously
    - Exceeding tasks will be blocked until some of the previous tasks complete
  - Since the cluster is equipped with M < N processors, only M threads may run concurrently
  - The client needs to be notified when its computation is over
- These functionalities are to be implemented from scratch:
  - Counting Semaphore (FIFO fairness required)
  - Thread Pool
  - Completion callback

# New synchronization facilities in tiger

- Java 1.5 provides new classes to ease the implementation of multithreaded programs (part of the `java.util.concurrent`, `java.util.concurrent.atomic`, and `java.util.concurrent.locks` packages)
- Most relevant features provided include:
  - The *Executor framework* for standardizing invocation, scheduling, execution, and control of asynchronous tasks according to a set of execution policies. Implementations are provided that allow tasks to be executed within the submitting thread, in a single background thread (as with events in Swing), in a newly created thread, or in a thread pool, and developers can create new instances of Executor supporting arbitrary execution policies
  - General purpose synchronization classes, including semaphores and mutexes (including read-write ones), which facilitate coordination between threads
  - A high-performance lock implementation with the same memory semantics as synchronization, but which also supports specifying a timeout when attempting to acquire a lock, multiple condition variables per lock, non-lexically scoped locks, and support for interrupting threads which are waiting to acquire a lock
  - A set of classes for atomically manipulating single variables (primitive types or references), providing high-performance atomic compare-and-set methods

# Future

```
interface ArchiveSearcher { String search(String target); }
 class App {
   ExecutorService executor = ...
   ArchiveSearcher searcher = ...
   void showSearch(final String target) throws InterruptedException
{
     FutureTask<String> future =
       new FutureTask<String>(new Callable<String>() {
         public String call() {
           return searcher.search(target);
       }});
     executor.execute(future);
     displayOtherThings(); // do other things while searching
     try {
       displayText(future.get()); // use future
     } catch (ExecutionException ex) { cleanup(); return; }
   }}
```

A task that returns a result (J2SE 5)