OPI
Lecture 16
Thread Programming

Mette Jaquet

Jeffrey Sarnat

IT University Copenhagen

Today's schedule

Goal

- To give a basic understanding of “threads”, how they are used and what some of the challenges are.

Contents

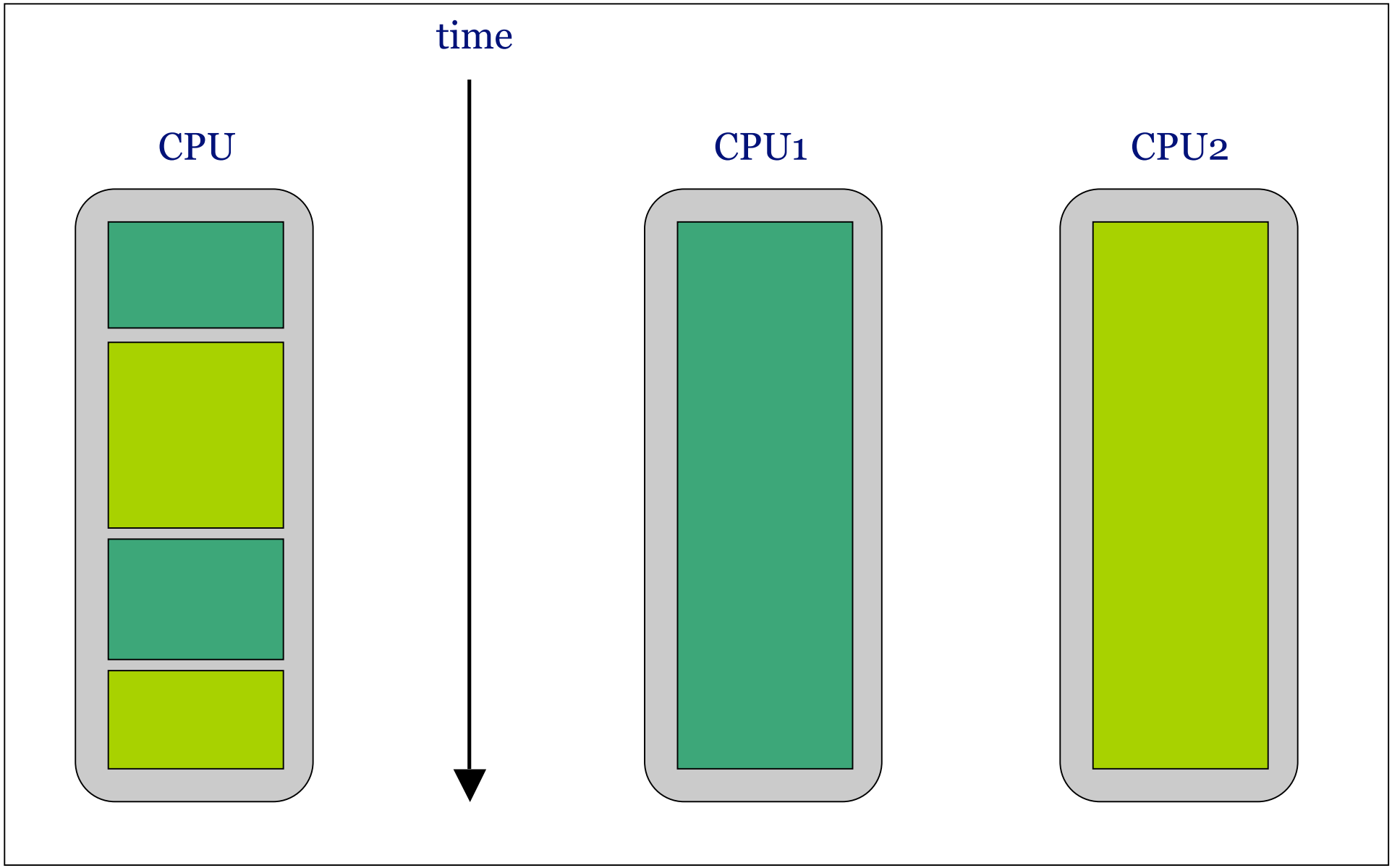
- What is a thread ?
- How are threads created and used ?
- Issues to consider when using threads
  - Communicating between threads
  - Scheduling threads
  - Synchronization
  - Deadlocks and other pitfalls to avoid

What is a thread ?

- A thread is shorthand for *thread of execution*
- In sequential programming there is only one thread.
- Sometimes it takes more than one thread to solve a problem…

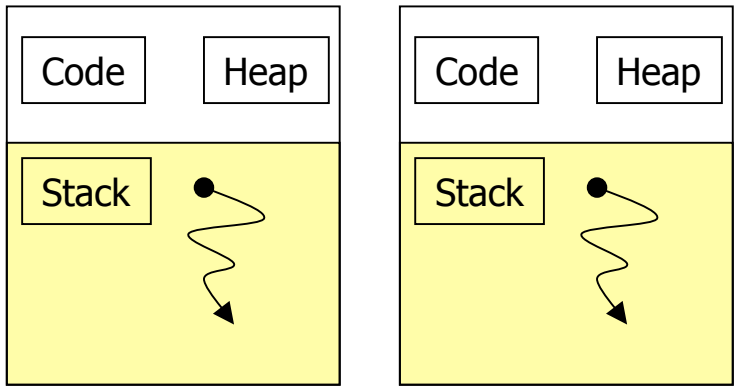
An object:	A thread:
Is passive	Is active
Points to code and data	Runs code
	Manipulates objects

Concurrency vs. Parallelism

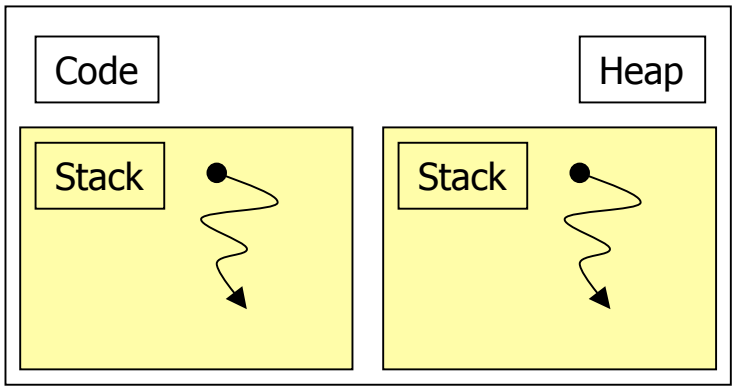


Multitasking vs. Multithreading

- *Multitasking* is a computers ability to do multiple program executions concurrently
- Each program has its own code, stack and heap

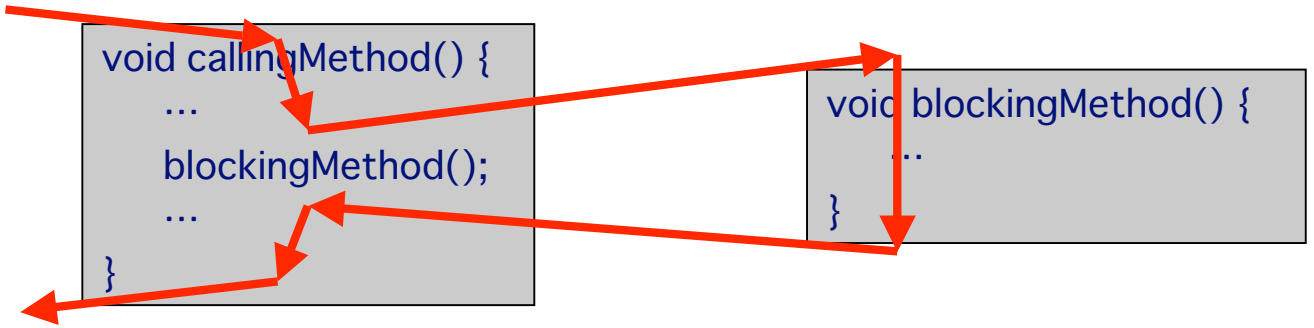


- *Multithreading* is having multiple threads in a program
- All threads access the same code and share the same heap
- Each thread has its own stack for local variables and method arguments

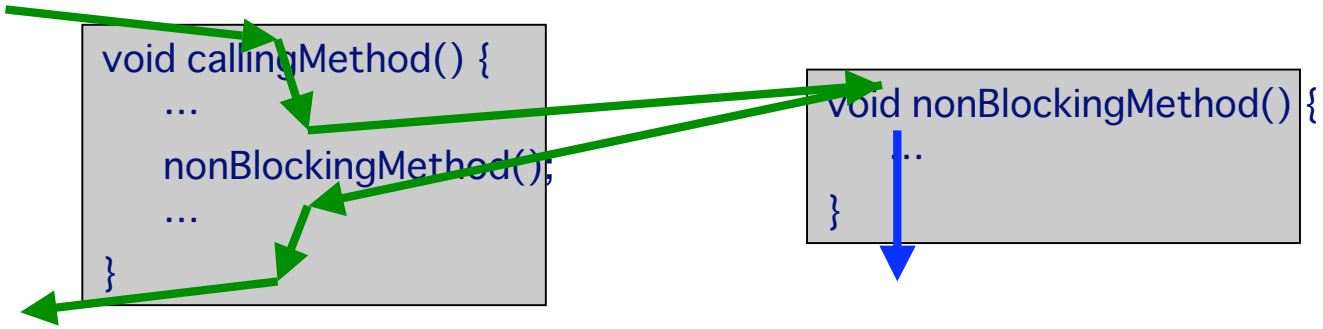


Blocking vs. non-blocking method calls

A blocking (synchronous) method call:



A non-blocking (asynchronous) method call:



A new thread executes the method

Usage of threads

- Responsive applications
- Monitoring the state of resources
  - Databases, servers etc.
- Problems of a parallel nature
  - I.e. simulations like Secretary.java
- Listening for events
  - GUI
  - Network packages
- Operations that take a while to complete
  - Printing

Two ways of creating threads

By extending Thread:

Declaring:

```
class MyThread extends Thread {
    public void run() {
            …..   //  loop doing something
    }
}
```

Starting:

```
MyThread threadObj = new MyThread();
threadObj.start();
```

By implementing Runnable

Declaring:

```
class MyRunnable implements Runnable {
    public void run() {
    …..   //  loop doing something
    }
}
```

Starting:

```
MyRunnable runnableObj = new MyRunnable();
new Thread(runnableObj).start();
```

Thread constructors

- Thread()
- Thread(**Runnable** target)
- Thread(**String** name)
- Thread(**Runnable** target, **String** name)

If a Runnable is passed to the constructor it will be the run() method on this object that is invoked when the thread is started.

If no Runnable argument is given the run() method on the Thread object is invoked.

Note that a Thread can be given name. This can be quite useful when debugging.

Starting a thread

When working with Threads in Java you always *implement* run() but *call* start()

Calling Thread.start() will create a new thread and enable it.

If the Thread object is a subclass of Thread the overridden run() method will be called.

If the Thread instance was constructed using a Runnable, run() on the Runnable object will be called.

The method run() is for a thread what main() is for objects.

A thread will start by running the code in the run() method.

It may continue running code in other methods, but it will always return and exit from the run() method.

When the code in the run() method is done executing the thread ceases to exist, but the Thread object still exists.

A ThreadPrinterExample

The two different ways of starting threads are shown to the right:

What will happen when the main() method below is executed?

```
class ThreadPrinterExample {
    public static void main(String[] args) {
                Print p0 = new Print();
                Printer p1 = new Printer('|');
                Printer p2 = new Printer('-');
                p1.start();
                p2.start();
                new Thread(p0).start();
    }
}
```

```
class Print implements Runnable {
    public void run() {
                for(int i = 0; i < 50; i++)
                System.out.print("R");
    }

    public void start() {
                System.out.println("Hi there!");
    }
}
```

```
class Printer extends Thread {
    char character;

    Printer(char character) { this.character = character; }

    public void run () {
                for(int i = 0; i < 50; i++)
                System.out.print(character);
    }
}
```

Joining threads

Two threads can be merged by calling join().

The calling thread will wait for the thread join() is called on to die before it continues.

So if threadA calls threadB.join(), then threadA will be the one waiting for threadB to die.

If join(long millis) is called, the thread will sleep till the thread it was called on dies, or the given time has elapsed.

Daemon threads

Normally a program will continue to run as long as there are any live threads.

An exception to this is daemon threads.

A daemon thread is made by setting Thread.setDaemon(true)

If a new thread is created by a daemon thread it will itself be a daemon thread.

Daemon threads are usually used for processes that run in the background for an extended period of time. I.e. monitoring state of resources or collecting garbage.

Scheduling

Thread scheduling determines how threads are allocated CPU time.
Some approaches are:

Preemptive scheduling – the scheduler pauses (preempts) the running thread to allow others to execute.

Non-preemptive scheduling – a running thread is never interrupted. The running thread must yield control to the CPU.

Both types can cause *livelocks* where threads are assigned CPU time but never progress.

Non-preemptive scheduling may also cause thread *starvation* where low-priority threads never are assigned CPU time.

Priorities

In Java scheduling is preemptive and based on priorities of threads.

When the scheduler assigns control to a thread it generally favours the one with the highest priority.

The methods used to influence thread-scheduling with priorities are:

setPriority(int newPriority)
getPriority()
Thread.yield()

The yield() method will voluntarily give up the control and let the scheduler activate another thread.

Inappropriate use of priorities might lead to *thread-starvation*

The challenge of sharing data

Local variables and method arguments are placed on the stack and are not available to other threads.

But objects and their fields are located in the heap and can be accessed by multiple threads.

When multiple threads read, modify and update data simultaneously a dependency on timing is introduced.

A situation may occur where the values that are updated depend on the timing. When different threads *race* to store their values it is called a *race condition*.

An example of a race condition

Imagine a bank account with an initial balance of 500.

Two transactions A and B are requested.

Transaction A deposits 200 units.

Transaction B withdraws 100 units.

What is the balance after the two transactions are done ?

The answer should be 600, but it will depend on the timing...

Thread A: // +200

- Get balance (500)
- Calculate new balance (700)
- Save new balance (700)

Thread B: // -100

- Get balance (500)
- Calculate new balance (400)
- Save new balance (400)

So what we need is..

A semaphore !

Restricting access

Some classical ways of restricting access to critical regions are:

Semaphores

- Are like flags signaling availability. They can be *counting semaphores* allowing a finite number of threads to enter at the same time, or *binary semaphores* allowing only one

Locks

- Are binary semaphores that can only be released by the thread holding the lock

Monitors

- Are encapsulations of resources that can only be accessed under certain conditions. A conditional expression determines if a given thread may enter the monitor or not

Synchronization

Some sequences of instructions depend on the state of an object to be unchanged.

In Java the keyword *synchronized* is used to mark such critical regions of code that have to be executed in an atomic manner.

Java has a single lock associated with every object, array and class, and any thread entering a synchronized area of code is blocked if there is already another thread holding the lock on the requested object.

Statements can be synchronized…

```
synchronized (this) {
        // critical code
}
```

..or methods can be synchronized

```
synchronized void myMethod () {
        // critical code
}
```

Deadlocks

- Most commonly occur when threads are mutually blocking each other.
- Not normally detected by runtime system
- Often not detectable because it only shows under certain timing constraints.
- Avoid by design !

An Example:

Thread 1

```
syncronize(A){
        syncronize(B) {
            …
        }
}
```

Thread 2

```
syncronize(B){
        syncronize(A){
            …
        }
}
```

Avoiding deadlocks

- A rule of thumb to avoid having deadlocks, is to avoid having code in a restricted area that can halt a thread.

- If multiple locks are required, one way to avoid deadlocks is to use hierarchical locking where locks are always requested in the same order

It would not have been a problem if the previous example had been:

Thread 1

```
syncronize(A){
        syncronize(B) {
            …
        }
}
```

Thread 2

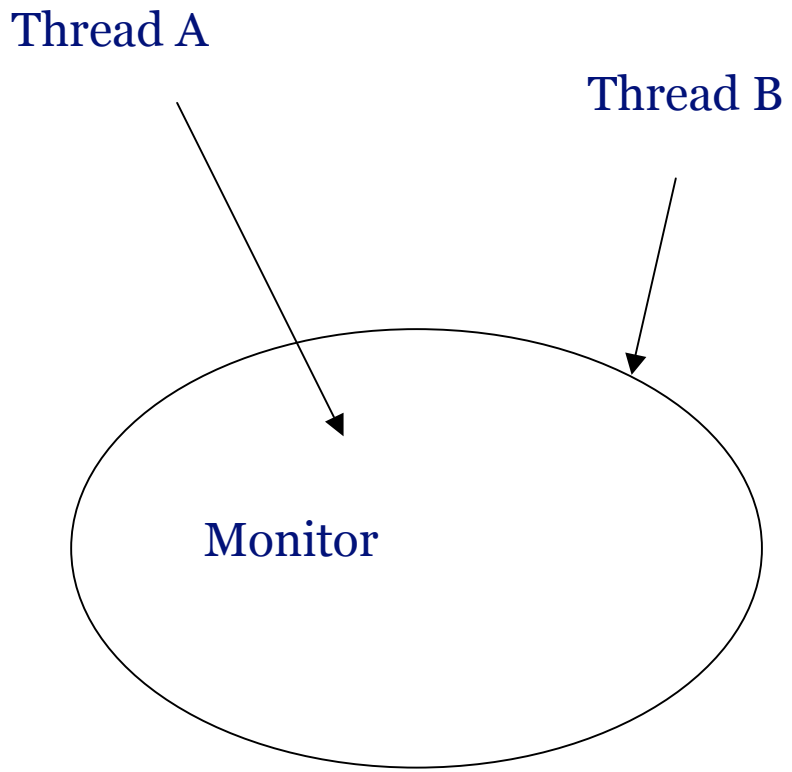
```
syncronize(A){
        syncronize(B){
            …
        }
}
```

Monitors

A monitor encapsulates an object ensuring that only one thread at a time can access the monitor.

Other threads will be blocked until the monitor is available.

In Java a monitor can be implemented by making all fields of a class private and all areas that manipulate the fields synchronized.



Wait & Notify

The following methods on the Object class can be used for monitors:

void wait(): Causes the calling thread to wait until another thread calls notify() or notifyAll() on the object wait() was called on. A thread can only call wait() on an object o if it has the lock on o. This means wait() can only be called inside a synchronized block of code.

void wait(long timeout): As wait() but with a maximum limit of time to wait.

void notify(): Wakes a thread waiting for the object. If more than one thread is waiting only one is notified.

void notifyAll(): Wakes all threads waiting for an object.

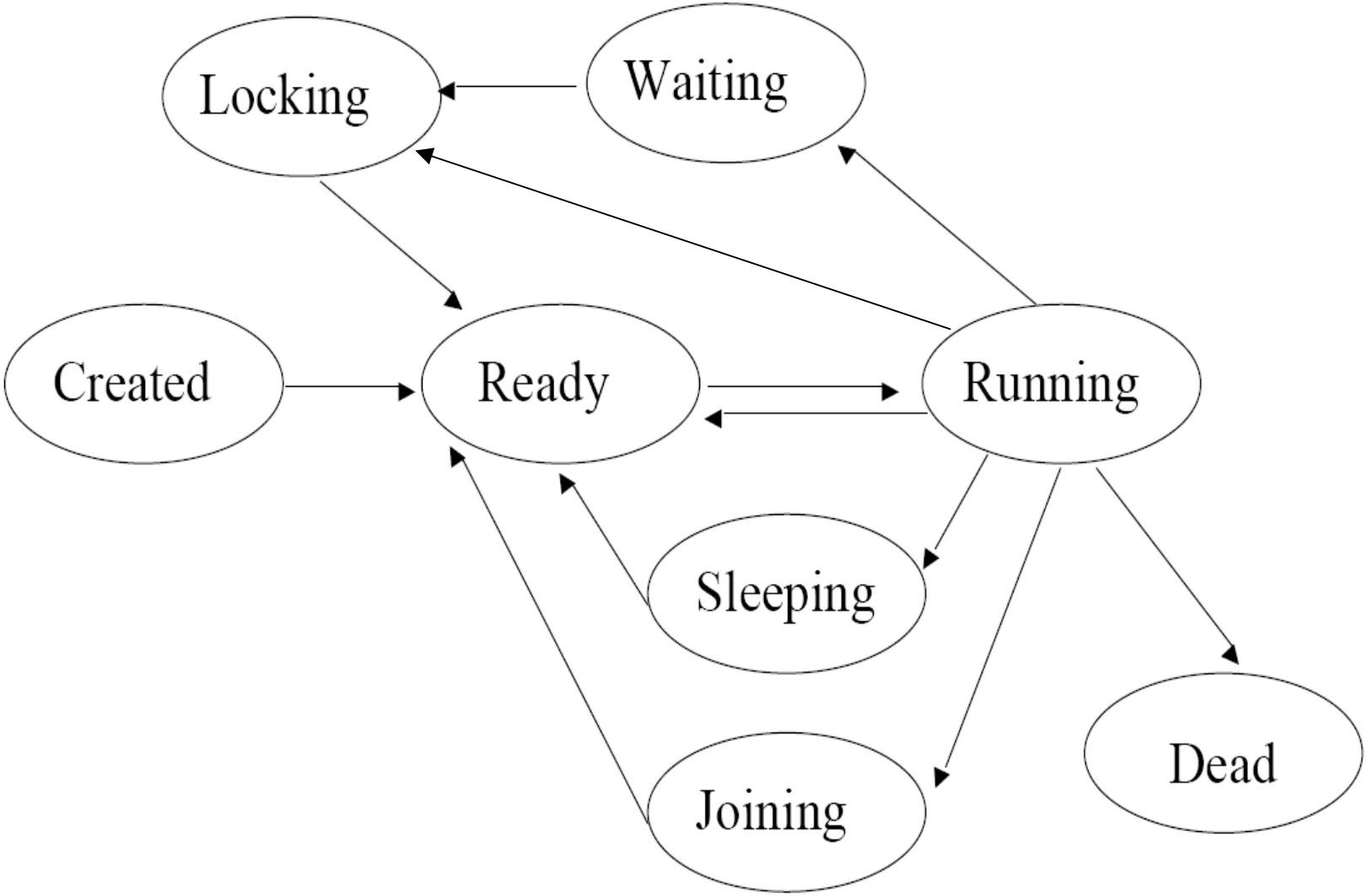
Inter-thread communication

A few common patterns of inter-thread communications are:

Wait & Notify	Blocks current thread with monitor.wait() and wakes up another thread with monitor.notify() Thread must hold lock when calling wait or notify
Send & Receive	Blocks current thread with socket.receive() and wakes up another thread with socket.send() Can transfer information in the data being sent Also works between different programs. Not dependent on a monitor
Agent	An agent has its own life = its own thread You can stimulate it and get responses asynchronously. Sending a stimulus does not wait for a response
Worker-thread	All information is provided before the thread starts.

States of a thread

- The life cycle of a thread can be described as



Blocking and unblocking

Thread state	Caused by	Changed by
Sleeping	Thread.sleep()	Time passed
Joining	otherThread.join()	otherThread dies
Waiting	object.wait()	object.notify()
Locking	synchronized(object) or waiting for i.e. I/O	Object gets unlocked or resource available

Interrupting a thread

- Interrupting a thread is another way of unblocking a thread and changing its state.
- What state a thread will enter depends on its state when interrupt() is called.
- If it is sleeping or joining it will be marked as ready again.
- If it is waiting for a lock it will become locking.
- If a thread is either running, ready or locking the interrupted status will be set true but no InterruptedException is thrown

Methods on Thread:

void **interrupt()**: Interrupts the thread it is called on.

static boolean **interrupted()**: Returns the interrupted status of the currently running thread resets the value.

boolean **isInterrupted()**: Returns the interrupted status on the thread it is called on – without changing the value.

An **InterruptedException** is thrown when interrupt() is called on a blocked thread and it becomes running.

Stopping a thread

- For a thread to stop in an orderly manner it needs to release its monitors and notify any waiting objects before it dies.
- There is a method on Thread called stop(), but it is deprecated and unsafe to use. If called it will stop the thread, but it might leave the program in an inconsistent state.
- The recommended way to stop a thread is by using a private variable that is checked regularly and changed if the thread is to stop.

```
class MyThread {
    private Thread running;

    public void stop() {
        running = null;
    }

    public void run() {
        Thread thisThread = Thread.currentThread();
        while (running == thisThread) {
                // do stuff
        }
    }
}
```

Is Santa thread-safe ?

In lecture 2 Santa was given as an example of a Singleton object (an object you only allow a single instance of)

Is Santa thread-safe?

```
public class Santa {
   private static Santa santa;

   public static Santa theOneAndOnly(){
        if (santa == null) {     // lazy instantiation
              santa = new Santa();
        }
        return santa;
   }

   private Santa() {
              ….
   }
```

A summary of pitfalls

A summary of the pitfalls to avoid when using threads includes:

- **Race conditions** that can leave data inconsistent
- **Deadlocks** where blocked threads are waiting for each other
- **Livelocks** where threads are given CPU time, but are unable to progress
- **Thread thrashing** where excessive thread swapping leads to poor performance
- **Thread starvation** where threads with a low priority never get CPU time
- **Semantic behavior** is changed

So when doing thread programming try to enforce:

Safety and **Liveness**

Useful tools

- Some solutions to some specific classical concurrency problems:
  - java.util.concurrent
  - java.util.concurrent.locks
  - java.util.concurrent.atomic
- Non-blocking input and output
  - java.nio
- Explicit scheduling
  - java.util.Timer
  - java.util.TimerTask

A last word on performance

Using threads has an impact on performance.

But you can minimize the effects by:

- Limiting the number of threads
- Making as few areas as possible synchronized
- Using fine-grained locks in order to decrease the time threads wait for a lock
- Prioritizing threads to make selected ones wait less than others.
- Consider using shallow copies (clones) when running through i.e. Collections to avoid synchronizing large blocks of code.