# COMP2121: Principles of Distributed Systems and Networks

## Concurrency

Unit coordinator Dr. Vincent Gramoli
School of Information Technologies

THE UNIVERSITY OF SYDNEY

# Introduction

- Previously: a distributed system gives the illusion of a single system to a user thanks to transparency

- Now: a single system gives the illusion of multiple resources to multiple users thanks to concurrency

# Outline

- A Very Brief History
- UNIX Processes and Threads
- Java Threads
- Multithreading
- Multithreaded Server
- Virtualization

# A Very Brief History

Dr Vincent Gramoli | Lecturer
School of Information Technologies

THE UNIVERSITY OF SYDNEY

# Evolution of operating systems (OSes)

A brief history

- The hardware evolved
  - Hardware was expensive, humans were cheap (e.g., Multics 1960's)
  - Hardware cheaper, humans expensive (e.g., desktop computer 1980's)
  - Hardware very cheap, humans very expensive (e.g., handheld devices 2000's)

- The OS interaction had to adapt
  - Batch: one execution at a time
  - Multiprogramming: multiple program executions simultaneously
  - Timeshare: split time into slots allocated for different program execution
  - GUI: multiple interfaces to access a system from different end-points
  - Ubiquitous devices: each user possesses her own computational device

# Uniprogramming vs. Multiprogramming

Operating systems

- Uniprogramming: one program execution at a time
  - MS/DOS
  - early Macintosh
  - Batch processing
  - No longer acceptable!

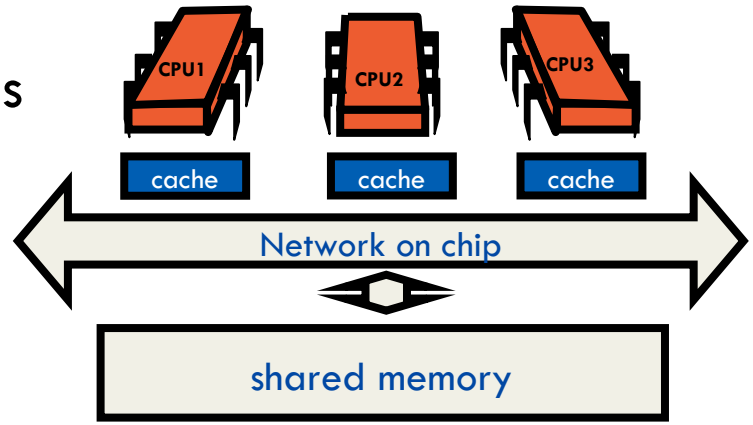
- Many program executions on personal computers:
  - Browsing the web
  - Sending emails
  - Typing a letter
  - Burning a DVD

# Uniprogramming vs. Multiprogramming

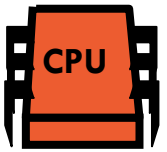
Operating systems

- Multiprogramming: more than one program execution at a time
  - Multics
  - UNIX/Linux
  - Windows NT
  - Mac OS X

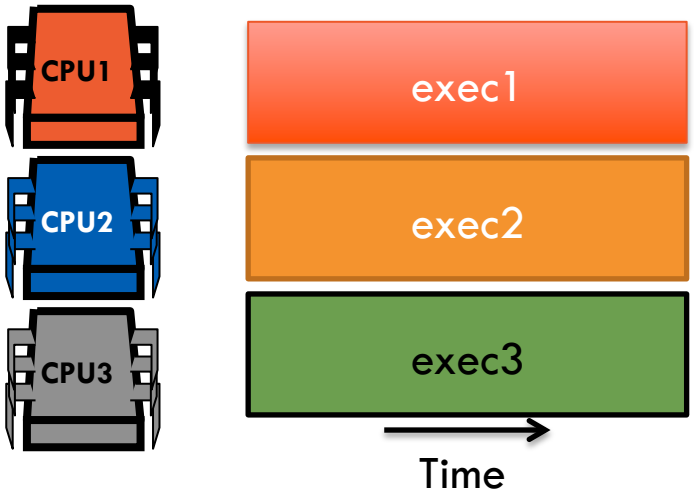
- Multi/many-cores



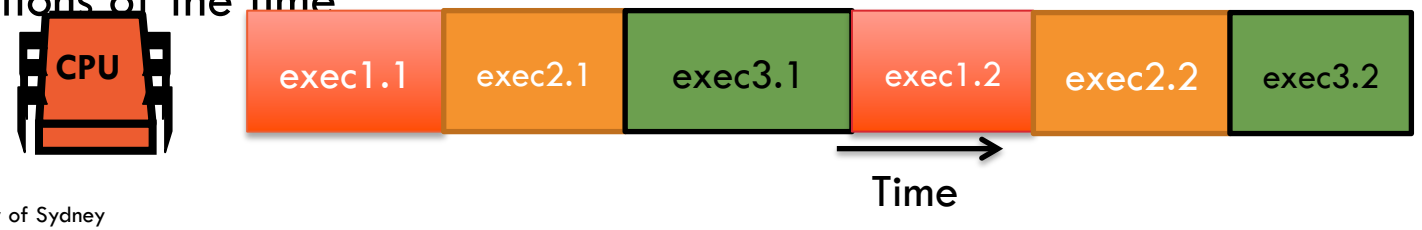
# Concurrency




- Assume a single central processing unit (CPU)
- Problem: How to give the illusion to the users of multiple CPUs?




- Solution: scheduling program executions, one after the other during small fractions of the time



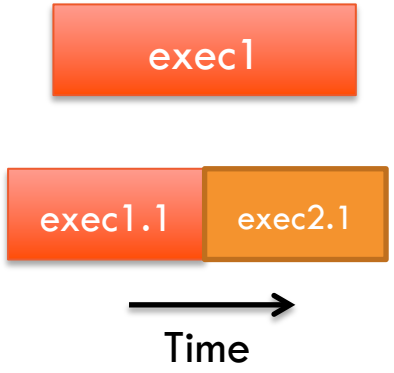
# Processes and Threads

Dr Vincent Gramoli | Lecturer
School of Information Technologies

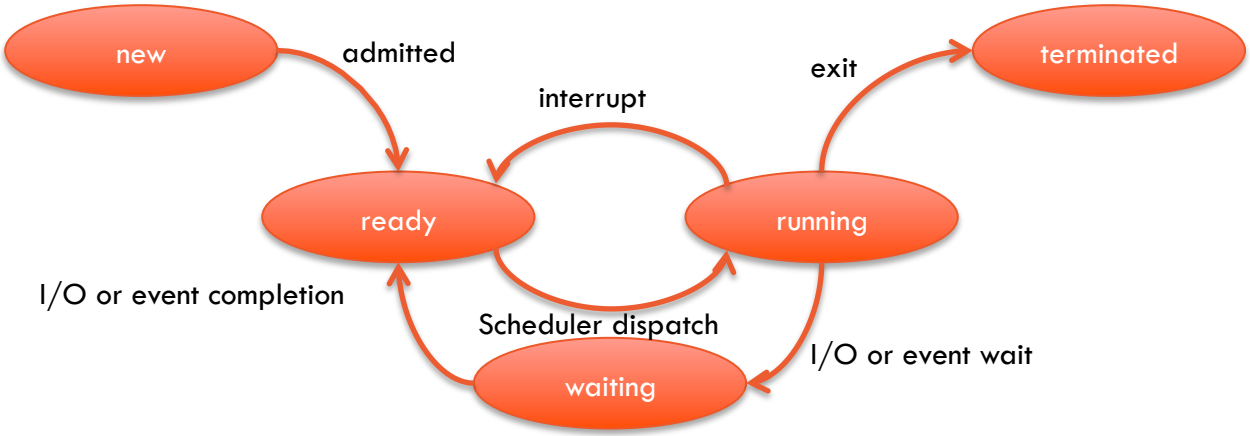
THE UNIVERSITY OF SYDNEY

# Process

*Process*: an abstraction representing a program execution



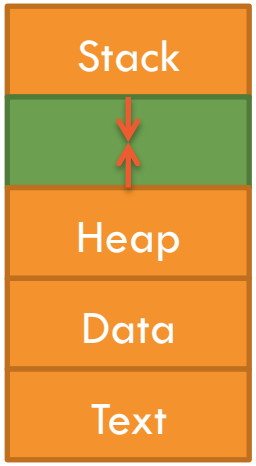

- When to *switch* from one process to another?
  - timer interrupt goes off, explicit yield, I/O, etc.
- Each virtual CPU needs to store the *process context*:
  - program counter, stack pointer, other registers
- A process switches from a state to another:



# Address space

*Address space:* A unit of management of a process's virtual memory

- Process address space:
  - Stack: temporary data extensible towards lower virtual addresses
  - Heap: memory allocated dynamically extensible to higher virtual addresses
  - Data section: global variable
  - Text region: program code



# Process management

Process creation

- Process allocation: the act of choosing the host of the process
  - The System V ("5") provides command to execute a process at an idle workstation
  - The Amoeba system chooses a host from a pool of processors to execute it
- Execution environment: an address space w/ initialized content & open files
  - Static: program text, heap and stack regions created from a list
  - Dynamic: UNIX fork shares program text and copies stack and heap regions

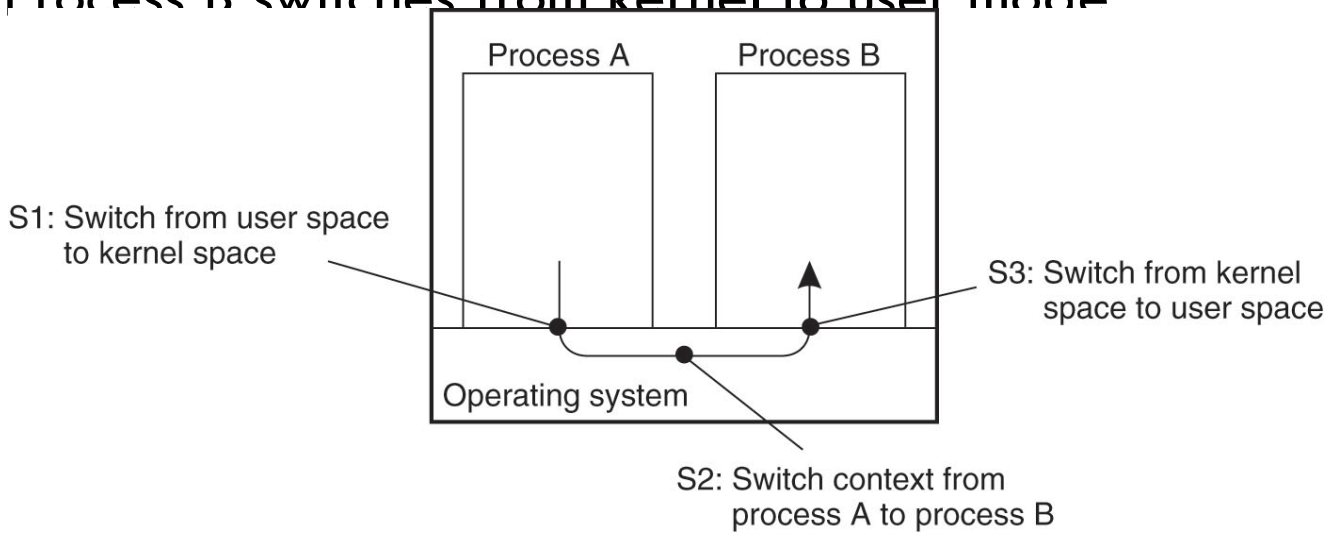
Interprocess communication (IPC):

- Pipes/Message queues/Shared memory segments
- Requires costly context switches

# Context-switch

An interrupt requires kernel intervention

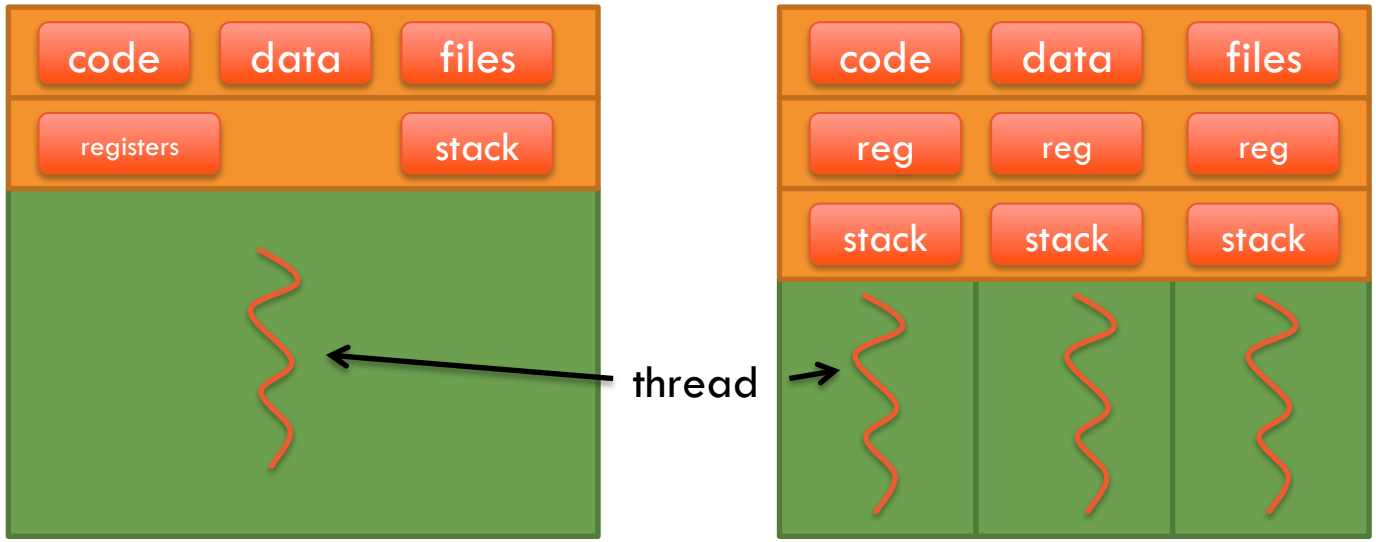
- S1. Process A switches from user to kernel mode
  - Changing the memory map in the MMU (memory management unit)
  - Flushing the TLB (translation lookaside buffer)
- S2. Context-switch (swapping processes from disk to main memory)
- S3. Process B switches from kernel to user mode



# Thread

*Threads*: smallest unit of CPU utilization

- Portable OS Interface (POSIX) Threads
  - Current activity: program counter
  - Stack: temporary data
  - No data, code, files (but it shares them with other threads)




Single-threaded process

Multithreaded process

# Thread

*Threads*: smallest unit of CPU utilization

- Portable OS Interface (POSIX) Threads
  - Current activity: program counter
  - Stack: temporary data
  - No data, code, files (but it shares them with other threads)

- Communication between threads always through memory
  - Does not need IPC
  - No context switches required (when purely at user-level)

# Thread

User-level threads library vs. kernel-scheduled thread

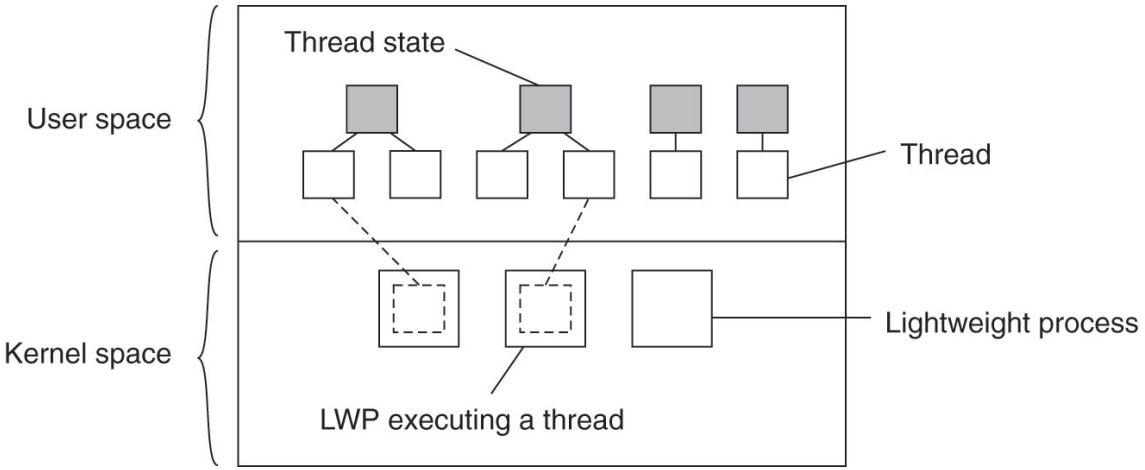
- User-level thread library:
  1. Cheap to create and destroy
  2. Blocking system call freezes the entire process of the thread (and other threads)
  3. Cheap context-switch:
     - Few instructions to store and reload CPU register values
     - No need to change memory map in MMU
     - No need to flush the TLB

- Kernel-scheduled threads:
  1. Costly to create and destroy
  2. Do not block the current process (and other threads) upon blocking system calls (I/O)
  3. Costly context-switch (similar to process context switch)
     - Needs to change memory map in MMU
     - Needs to flush the TLB

# Thread

## User-level threads library vs. kernel-scheduled thread

- Lightweight processes (LWP):
  1. Runs in the context of a process (potentially multiple LWPs per process)
  2. Current threads are in a table
  3. Once an LWP finds a runnable thread:
     - It locks the table in user-mode to update it
     - It switches context to that thread in user-mode
  4. Does not block upon system calls:
     - Passing from user to kernel mode in the LWP context
     - If the LWP is blocked, the kernel context-switches to another LWP
     - Implying a context-switch also at the user level



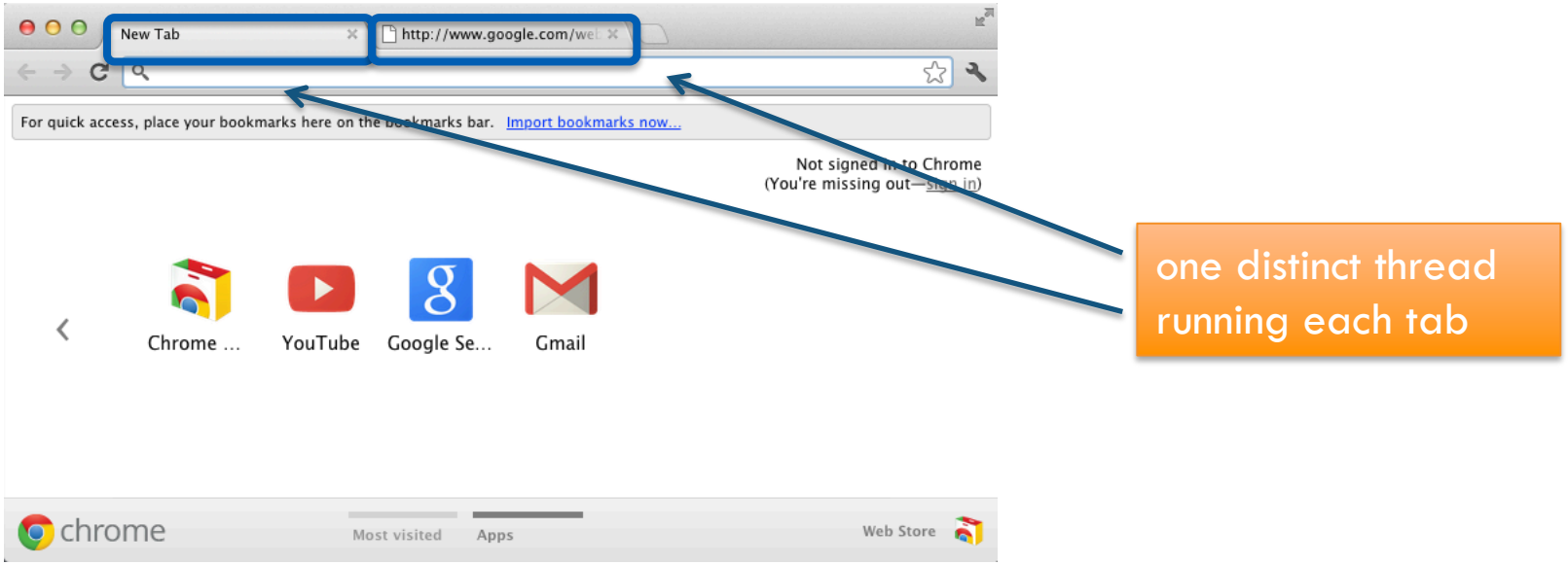
# Process vs. threads

- Processes
  - Isolated: prevents one process from interfering with another
  - Inefficient: starting/terminating a process and context switches are costly
- Threads
  - Non-isolated: avoiding incorrect interferences makes programming harder
  - Efficient: a thread is a lightweight version of a process

# Client multi-threading

Example: The Google Chrome web browser

- Multithreading: each tab runs its own thread
- A JavaScript error does not crash the main Chrome process:
  Only one tab freezes, the user can close it independently of others
- Takes benefit of multicore architectures, each thread runs on a separate core



# Java Threads

Dr Vincent Gramoli | Lecturer
School of Information Technologies

THE UNIVERSITY OF SYDNEY

# Threads in Java

They are more threads than you see

- Java Virtual Machine (JVM) thread: main thread, responsible for executing VM operations
- Periodic task thread: simulate timer interrupts to execute periodic operations, like scheduling other threads
- Garbage collection threads: parallel and concurrent garbage collection
- Compiler threads: runtime compilation of bytecode to native code
- Signal dispatcher thread: redirects process directed signals to Java level signal handling methods
- Graphical user interface: Java AWT and Swing creates threads for refreshing visual representation
- Java Servlets and RMI: create pools of threads and invoke methods within these

# Thread API

Synchronization calls

- thread.join([long milliseconds])
  Blocks the calling thread until the specified thread has terminated (for up to the specified milliseconds)
- thread.interrupt()
  Interrupts thread: causes it to return from a blocking method call such as sleep()
- object.wait([long milliseconds])
  Blocks the calling thread until a call made to notify() or notifyAll() on objects wakes the thread or the thread is interrupted (or the specified milliseconds have elapsed)
- object.notify()/object.notifyAll()
  Wakes, respectively, one (arbitrarily chosen) or all threads that have called wait on object.

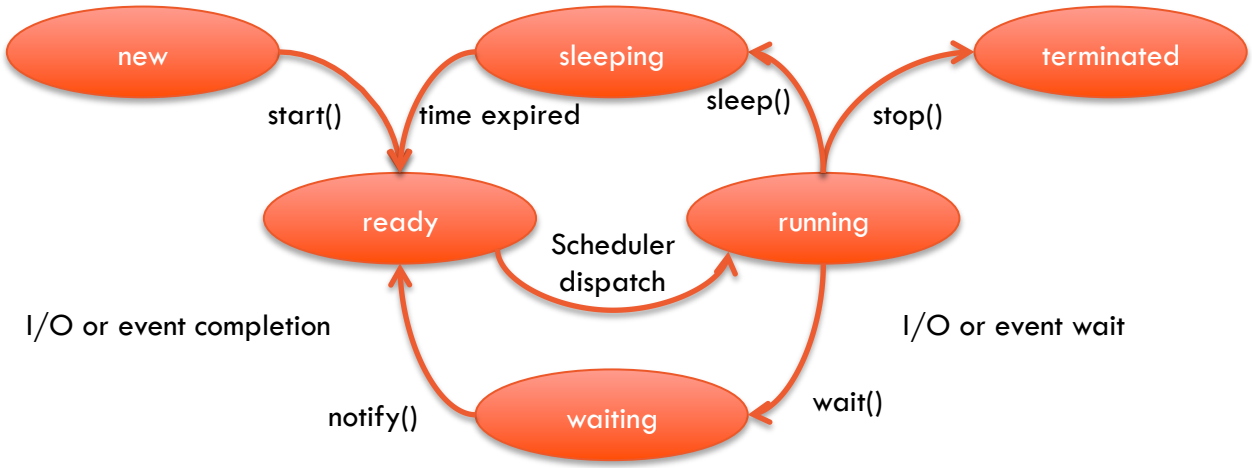
# Thread API

Thread lifecycle

- NEW
  A thread that is just instantiated is in *new* state. When a start() method is invoked, the thread moves to the ready state from which it is automatically moved to runnable state by the thread scheduler.
- RUNNABLE (ready → running)
  A thread executing in the JVM is ready to be placed in the running state by the scheduler
- BLOCKED
  A thread that is blocked waiting for a monitor lock is in this state. This can also occur when a thread performs an I/O operation and moves to next (runnable) state.
- WAITING
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- SLEEPING
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- TERMINATED (dead)
  A thread that has excited is in this state.

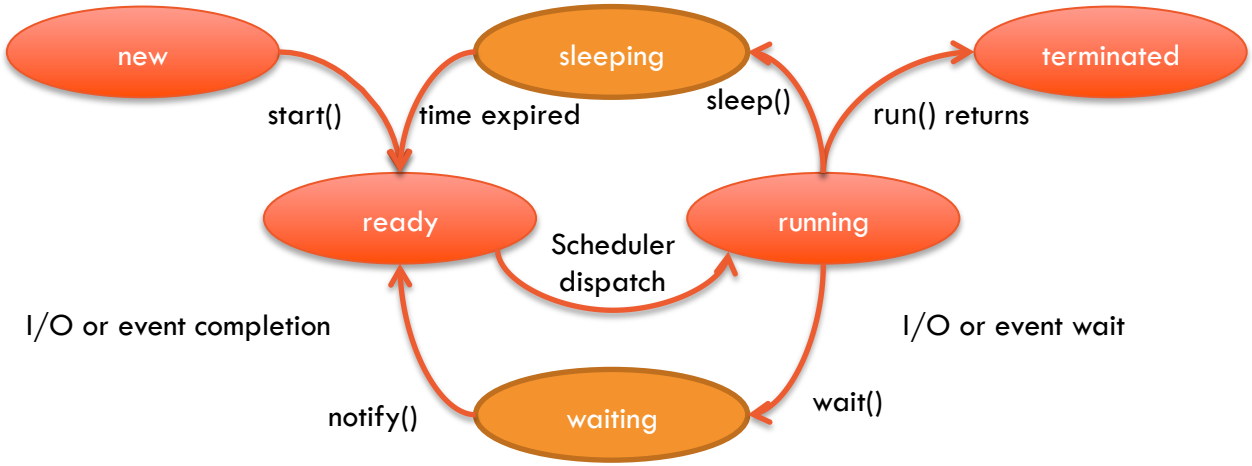
# Thread API

Thread lifecycle (con't)



# Thread API

Thread lifecycle (con't)



# Programming threads

Creating Java threads

There are two ways:

1. Deriving a class from the Thread class and overriding run()

2. Writing a class that implements the Runnable interface
   - Must redefine the run method
   - The code within the run method executes as a separate thread

```
class MyThread implements Runnable {
    public void run() {
        /* here goes my thread code */
    }
}
```

# Programming threads

Creating Java threads

There are two ways:

1. Deriving a class from the Thread class and overriding run()

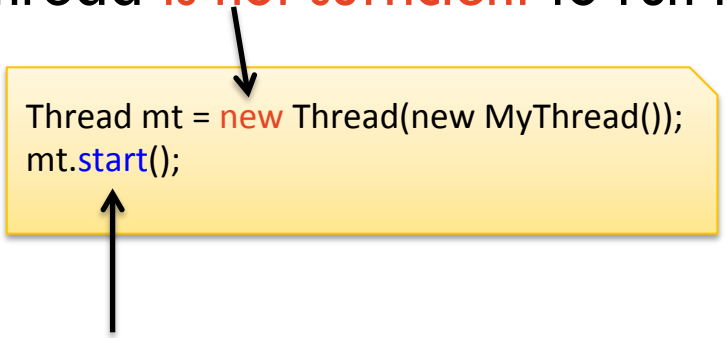
2. Writing a class that implements the Runnable interface
   - Must redefine the run method
   - The code within the run method executes as a separate thread

```
class MyThread implements Runnable {
    public void run() {
        /* here goes my thread code */
    }
}
```

# Programming threads

Running Java threads

- Instantiating a thread is not sufficient to run it

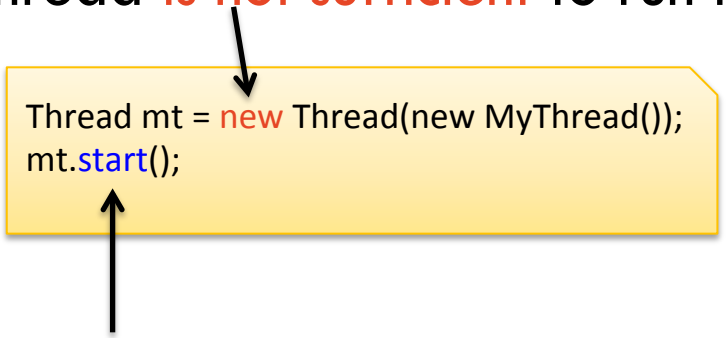



- One has to call its start() method, that will implicitly call its run() method

# Programming threads

Running Java threads

- Instantiating a thread is not sufficient to run it




- One has to call its start() method, that will implicitly call its run() method

*Actually, it may not even start right after: but it is ready to be run by the scheduler*

# Summary

- A thread is generally faster than a process, but shares its data with other threads

- To prepare a Java thread for execution, (1) implements the Runnable interface, (2) allocate a new thread, and (3) starts it!

# Multithreading

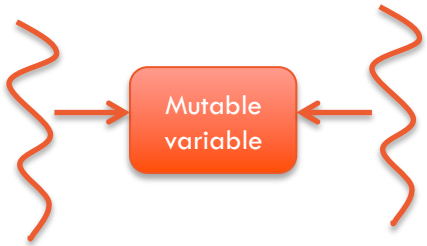
Dr Vincent Gramoli | Lecturer
School of Information Technologies

THE UNIVERSITY OF SYDNEY

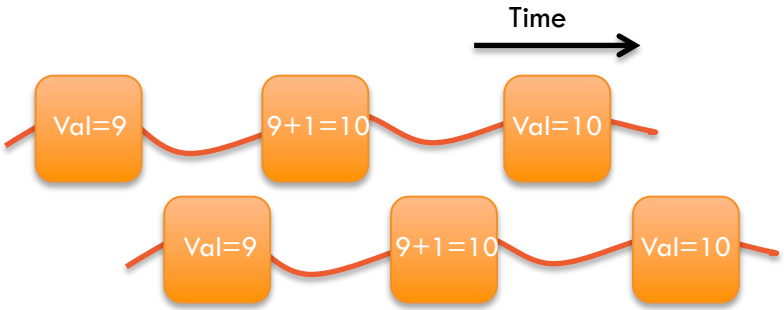
# Thread safety

*Thread safety* is about protecting data from uncontrolled concurrent access

- An *object state* is its data, stored in *state variables* such as instances or static fields
- A *shared variable* is a variable that could be accessed by multiple threads
- A *mutable variable* is a variable whose value may change
- Whenever more than one thread access a given state variable and one of them might write to it, they all must coordinate their access to it using synchronization
- If multiple threads access the same mutable state variable without appropriate synchronization, the program is broken.




```
public class UnsafeSequence {
  private int value;

  // Returns a unique value
  public int getNext() {
    return value++;
  }
}
```




Multiple threads accessing a mutable variable may produce erroneous results

# Thread safety

*Thread safety* is about protecting data from uncontrolled concurrent access

- Potential solutions:
  - Do not share the state variable among threads, make it thread specific;
  - Make the state variable immutable; or
  - Use synchronization to access the state variable

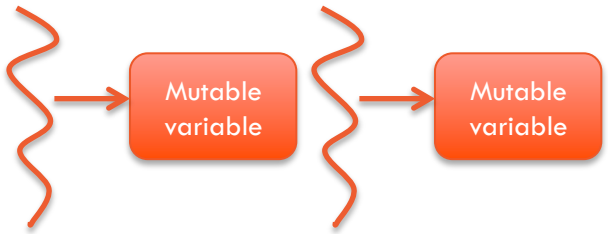
```
public class UnsafeSequence {
  private int value;

  // Returns a unique value
  public synchronized int getNext() {
    return value++;
  }
}
```


Using synchronization

# Thread safety

## Thread-specific data

- Threads belonging to the same process share data
- How to declare data to be "thread-specific"?
  - By inheriting from Thread into MyThread in which data is declared
  - By using the ThreadLocal class initialized by initialValue()/set() and accessed through get()/set()



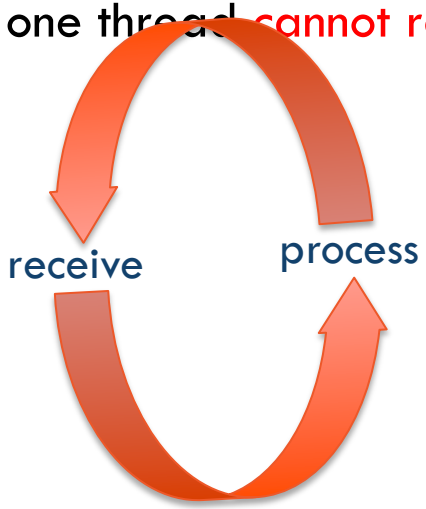

Thread-specific variables

```
public class Service {
 private static ThreadLocal errorCode = new ThreadLocal();

 private static void transaction() {
  try {
    // some operation where an error may occur
  }  catch (Exception e) {
    errorCode.set(e);
  }

  // get the error code for this transaction
  public static Object getErrorCode() {
     return errorCode.get();
  }
}
```

# Producer/consumer threads

Example: producer/consumer threads

- Problem:
  - Assume that you may receive requests almost simultaneously
  - It takes you some non-null amount of time to process a request
  - Yet, you would like to eventually process all requests
  - How to proceed when one thread cannot receive while processing a request?

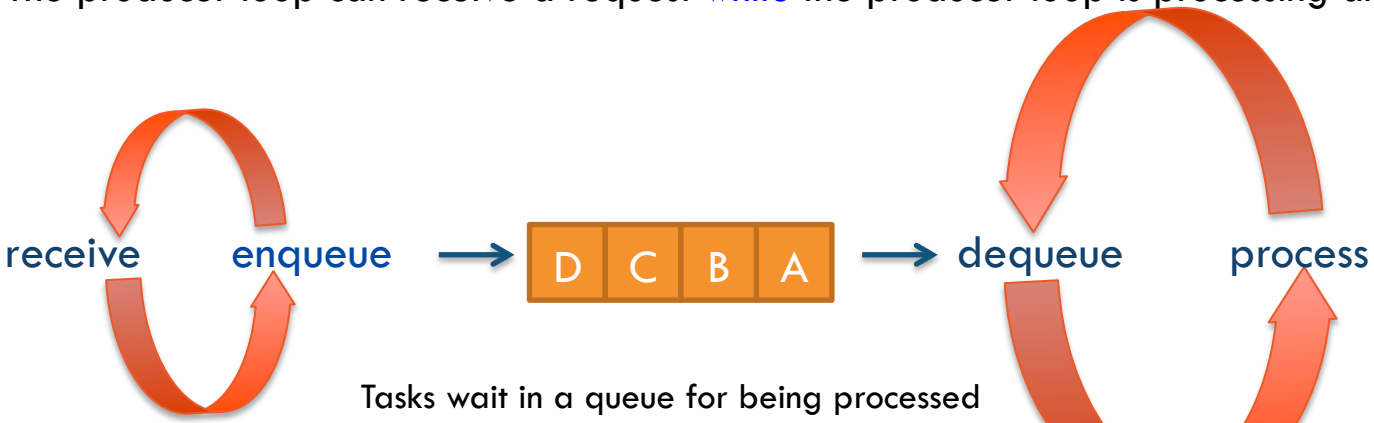



By the time it can receive again, some request may be lost

# Producer/consumer threads

Example: producer/consumer threads (con't)

- Solution: use multithreading in a consumer producer scenario
  - Ideal for asynchronous communication (mailbox analogy ≠ phone analogy) between threads
  - Decouples the consuming and production that run at different rates
  - One thread typically passes the work that arrives to some other thread
  - The producer loop can receive a request while the producer loop is processing another



# Producer/consumer threads

Example: producer/consumer threads (con't)

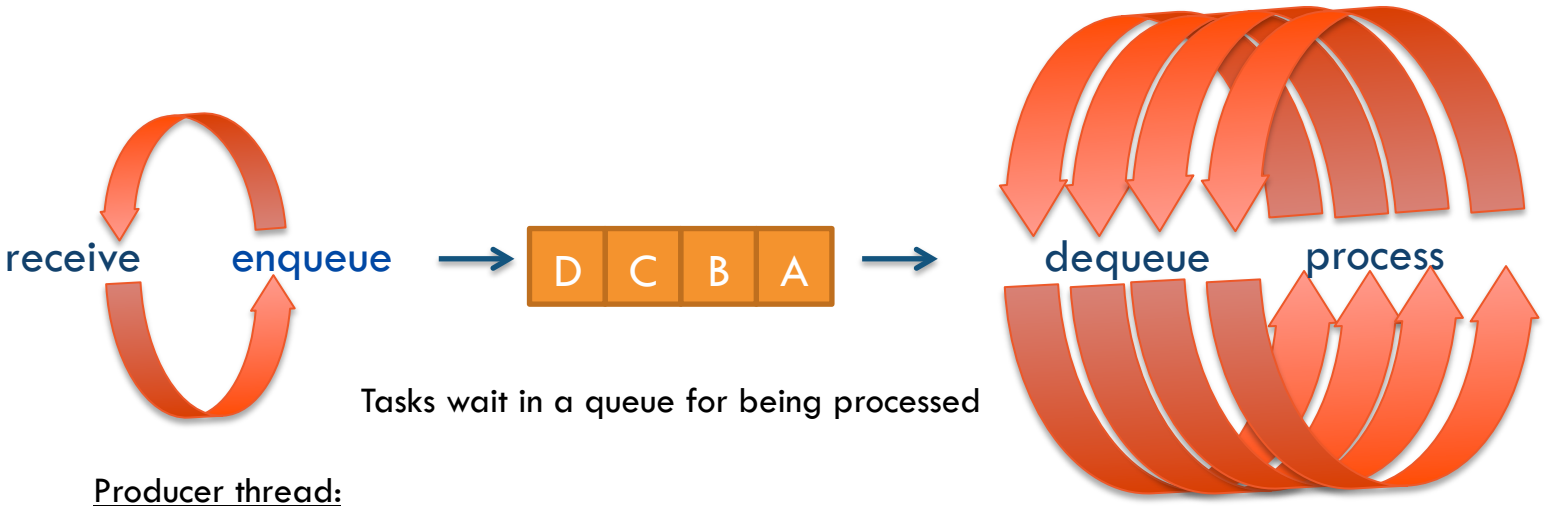
- One thread acts as a consumer
- Another thread acts as a producer

```
import java.util.Date;

public class Factory {
  public static void main(String args[]) {
    // create the message queue
    Channel<Date> queue = new MessageQueue<Date>();

    // create the producer and consumer threads and pass
    // each thread a reference to the MessageQueue object.
    Thread producer = new Thread(new Producer(queue));
    Thread consumer = new Thread(new Consumer(queue));

    producer.start();
    consumer.start();
}
```

# Producer/consumer threads

Example: producer/consumer threads (con't)

- Speeding up the request processing using multiple consumer threads



# Multithreaded Server

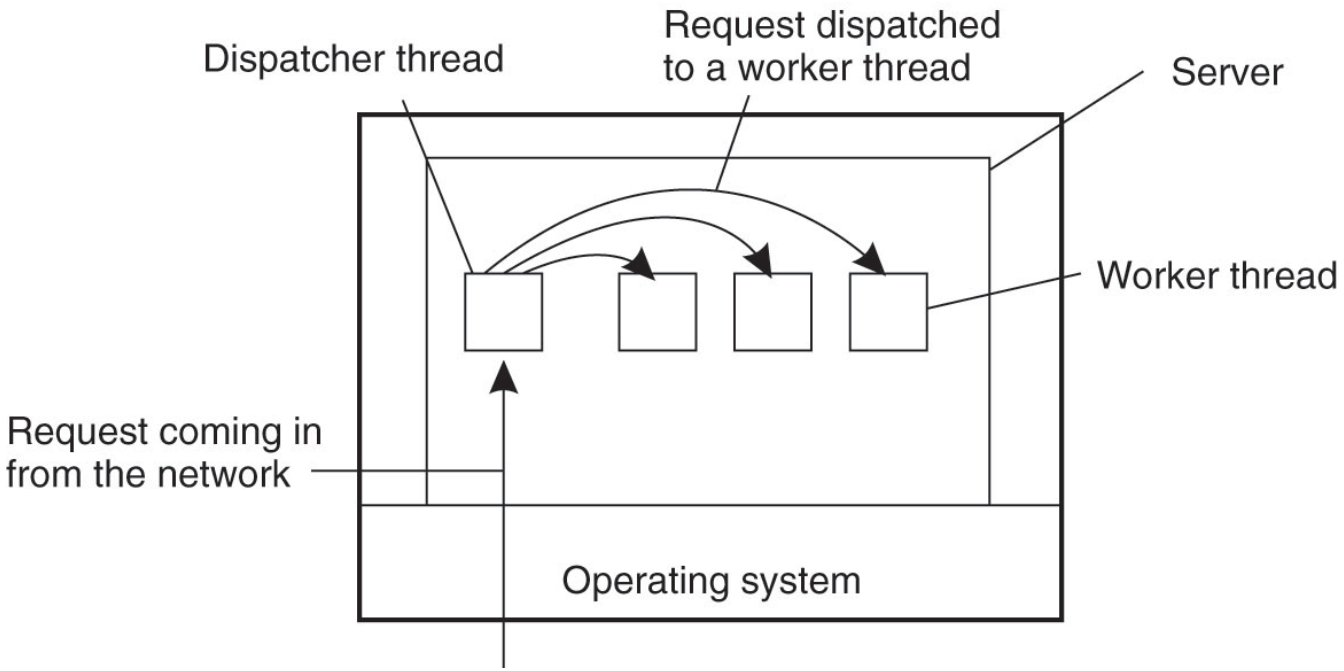
Dr Vincent Gramoli | Lecturer
School of Information Technologies

THE UNIVERSITY OF
SYDNEY

# Multithreaded server

A multithreaded server organized in a dispatcher/worker model

- One thread, the *dispatcher*, reads incoming clients requests
- The server chooses an *idle* worker thread and hands it the request



# Multithreaded server

Other alternatives

- Single-threaded server: one loop iteration must treat a request before starting to treat another one ⇒ potential request losses
- Finite state machine: the server treats the request if the requested info is in the cache, otherwise it does not wait for I/O to disk to complete but stores the state of the request in a table. It uses non-blocking system calls.

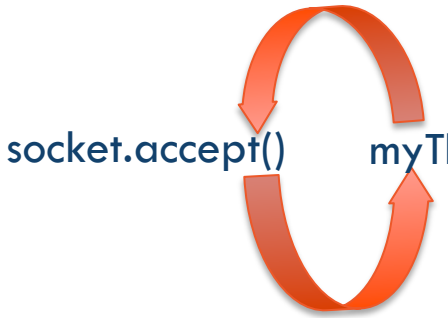
| Model | Characteristics |
|---|---|
| Threads | Parallelism, blocking system calls |
| Single-threaded process | No parallelism, blocking system calls |
| Finite-state machine | Parallelism, nonblocking system calls |

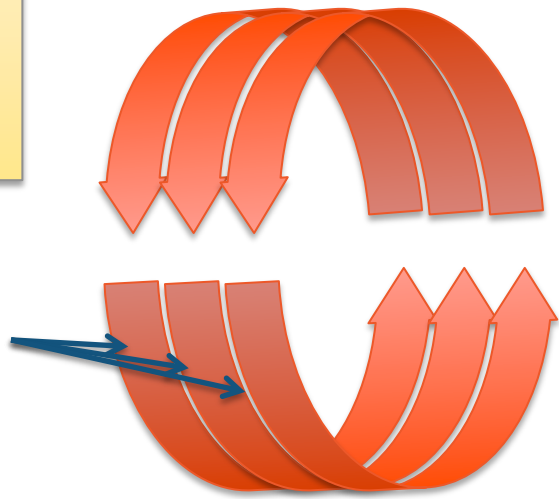
# Multithreaded server

Creating threads upon request

- For each request that arrives at the server
- Creates the request in a separate thread

```
...
   // keep treating incoming request
   while (true) {
     // block until accepting a new request
     try { client.socket = this.serverSocket.accept();
     } catch (IOException ioe) {System.out.println(ioe); }
     // start a new task in a new thread
     new Thread(new MyThread(new Task()).start();
   }
...
```



Main process thread: spawns one thread upon request

3 threads for 3 tasks

# Multithreaded server

Thread pool

- Creates a pool of threads
- That the server can (re)use
- No need to initialize many threads
- Nor to destroy many threads

```
import java.util.concurrent.*;

public class TPExample {

   public static void main(String[] args) {
     int numTasks = Integer.parseInt(args[0].trim());

     // create the thread pool
     ExecutorService pool = Executors.newCachedThreadPool();

     // Run each task using a thread in the pool
     for (int i=0; i<numTasks; i++)
        pool.execute(new Task());

     // Shut down the pool. This shuts down the pool only
     // after all threads have completed.
     pool.shutdown();
   }
}
```

# Conclusion

- Concurrency has been used for decades since hardware was powerful enough to address multiple humans need
- Processes are heavy but protected whereas threads are lightweight but non-protected
- Threads require synchronization to be protected from each other
- Java uses a process virtual machine but maps its virtual Java threads to real OS threads