v1.0
20130407

Programmazione Avanzata e Paradigmi

Ingegneria e Scienze Informatiche - UNIBO
a.a 2013/2014
Lecturer: Alessandro Ricci

[module 2.1]
CONCURRENT PROGRAMMING INTRODUCTION

SUMMARY

- Concurrent programming
  - motivations: HW evolution
  - basic jargon
    - processes interaction, cooperation, competition,
    - mutual exclusion, synchronization
    - problems: deadlocks, starvation, livelocks
- A little bit of history
  - Dijkstra, Hoare, Brinch-Hansen
- Concurrent languages, mechanisms, abstractions
  - overview

CONCURRENCY AND CONCURRENT SYSTEMS

- Concurrency as a main concept of many domains and systems
  - operating systems, multi-threaded and multi-process programs, distributed systems, control systems, *real-time* systems,...
- General definitions
  - "*In computer science, concurrency is a property of systems in which several computational processes are executing* ***at the same time****, and potentially* ***interacting*** *with each other.*" [ROS-97]
  - "*Concurrency is concerned with the fundamental aspects of systems of multiple,* ***simultaneously active*** *computing agents, that* ***interact*** *with one another*" [CLE-96]
- Common aspects
  - systems with multiple activities or ***processes*** whose execution **overlaps in time**
  - activities can have some kind of *dependencies,* therefore can **interact**

CONCURRENT PROGRAMMING

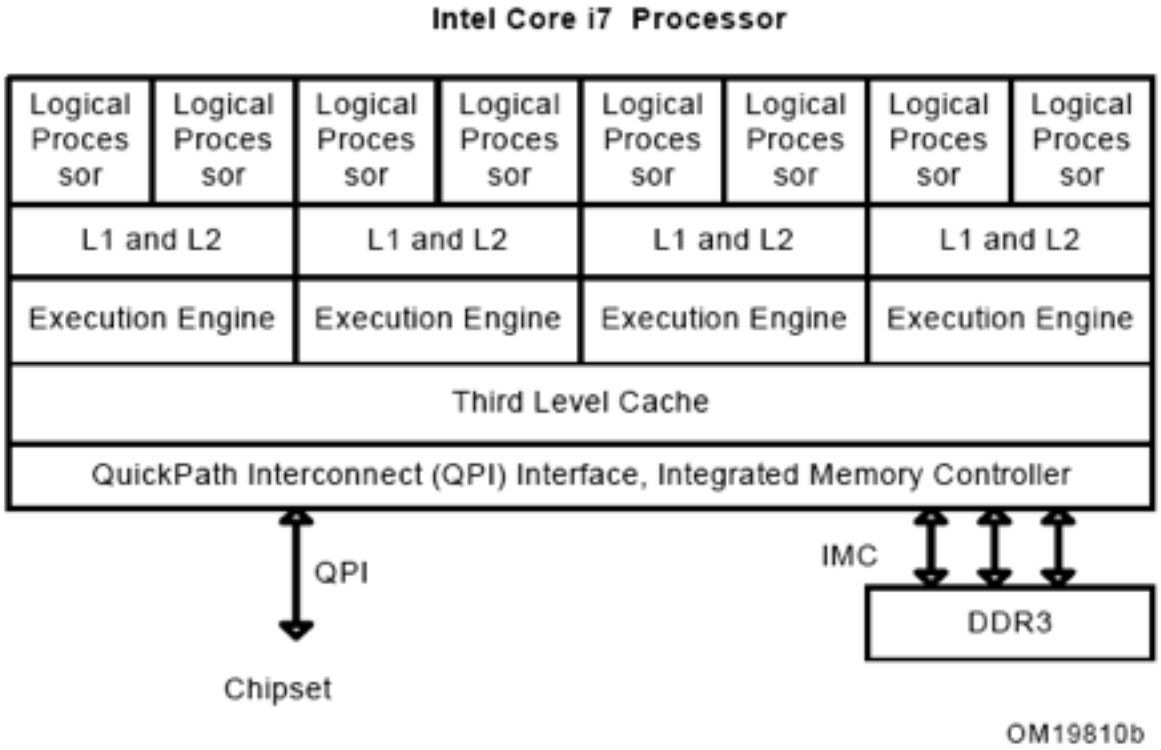
- **Concurrent programming**
  - building programs in which multiple computational activities overlap in time and typically interact in some way
- **Concurrent program**
  - finite set of *sequential* programs that can be executed in parallel, i.e. overlapped in time
    - a sequential program specifies sequential execution of a list of statements
    - the execution of a sequential program is called **process**
    - a concurrent program specifies two or more sequential programs that may be executed concurrently as *parallel processes*
  - the execution of a concurrent program is called *concurrent computation or elaboration*

CONCURRENT PROGRAMMING VS. PARALLEL PROGRAMMING

- **Parallel** programming
  - the execution of programs overlaps in time by running on separate physical processors
- **Concurrent** programming
  - the execution of programs overlaps in time *without necessarily running on separate physical processors*, by sharing for instance the same processor
    - potential or *abstract* parallelism
- **Distributed** programming
  - when processors are distributed over a network
  - no shared memory

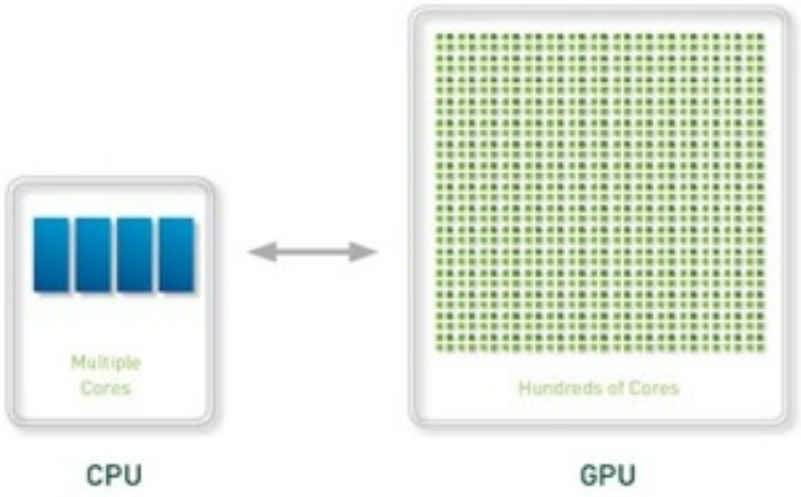
PARALLEL COMPUTERS: MULTI-CORE ARCHITECTURES

- **Chip multiprocessors - Multicore**
  - multiple cores on a single chip
    - sharing RAM, possibly sharing cache levels
    - examples: Intel Core Duo, Core i7, AMD Dual Core Opteron



PARALLEL COMPUTERS: HETEROGENEOUS CORES & MANY-CORE

- **Heterogeneous Chips Designs**
  - augmenting a standard processor with one or more specialized compute engines, called *attached processors*
    - examples: Graphical Processing Units (GPU), **GPGPU** (General-Purpose Computation on Graphics Hardware), Field-Programmable Gate Array (FPGA), Cell processors, **CUDA** architecture



PARALLEL COMPUTERS: SUPER-COMPUTERS

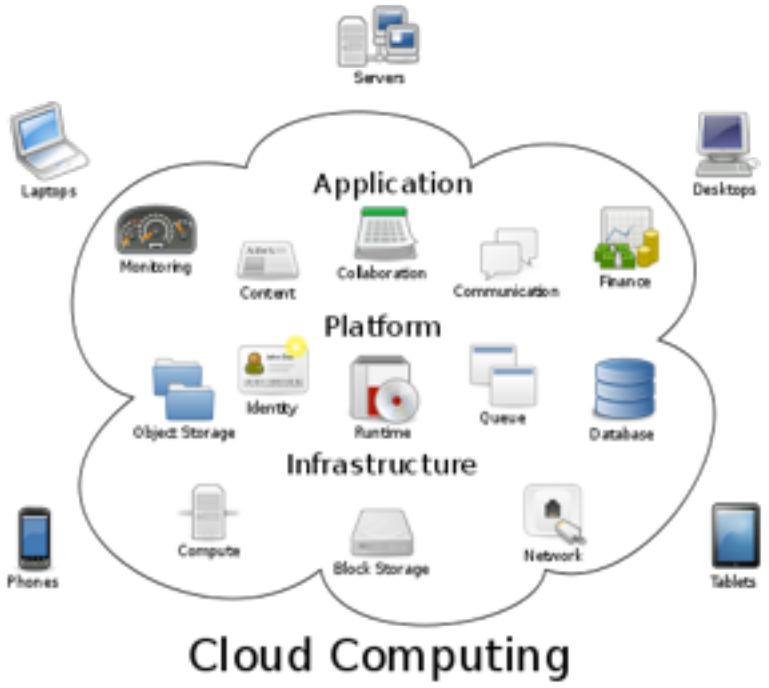
- Traditionally used by national labs and large companies
- Different kind of architectures, including clusters
- Typically large number of processors
  - example: IBM BlueGene/L
    - 65536 dual-core nodes, where each node is a 440 PowerPC (770MhZ), 512 MiB of shared RAM, a number of ports to be connected to the other nodes

PARALLEL COMPUTERS: CLUSTERS / GRID

- Made from commodity parts
  - nodes are boards containing one or few processors, RAM and sometimes a disk storage
  - nodes connected by commodity interconnect
    - e.g. Gigabit Ethernet, Myrinet, InfiniBand, Fiber Channel
- Memory not shared among the machines
  - processors communicate by message passing
- Examples
  - System X supercomputer at Virginia Tech,a 12.25 TFlops computer cluster of 1100 Apple XServe G5 2.3 GHz dual-processor machines (4 GB RAM, 80 GB SATA HD) running Mac OS X and using InfiniBand interconnect
- Grid computing

PARALLEL COMPUTERS: CLOUD COMPUTING

- Delivering computing as a service through the network
  - shared resources, software, and information are provided to computers and other devices as a metered service over a network (typically the Internet)
- X as a Service
  - Software as a Service (SAAS)
  - Platform as a Service (PAAS)
  - Infrastructure as a Service (IAAS)
- Public clouds, private clouds
- Examples
  - Amazon EC2 (Elastic Computing Cloud)
  - Microsoft Azure, Google App Engine

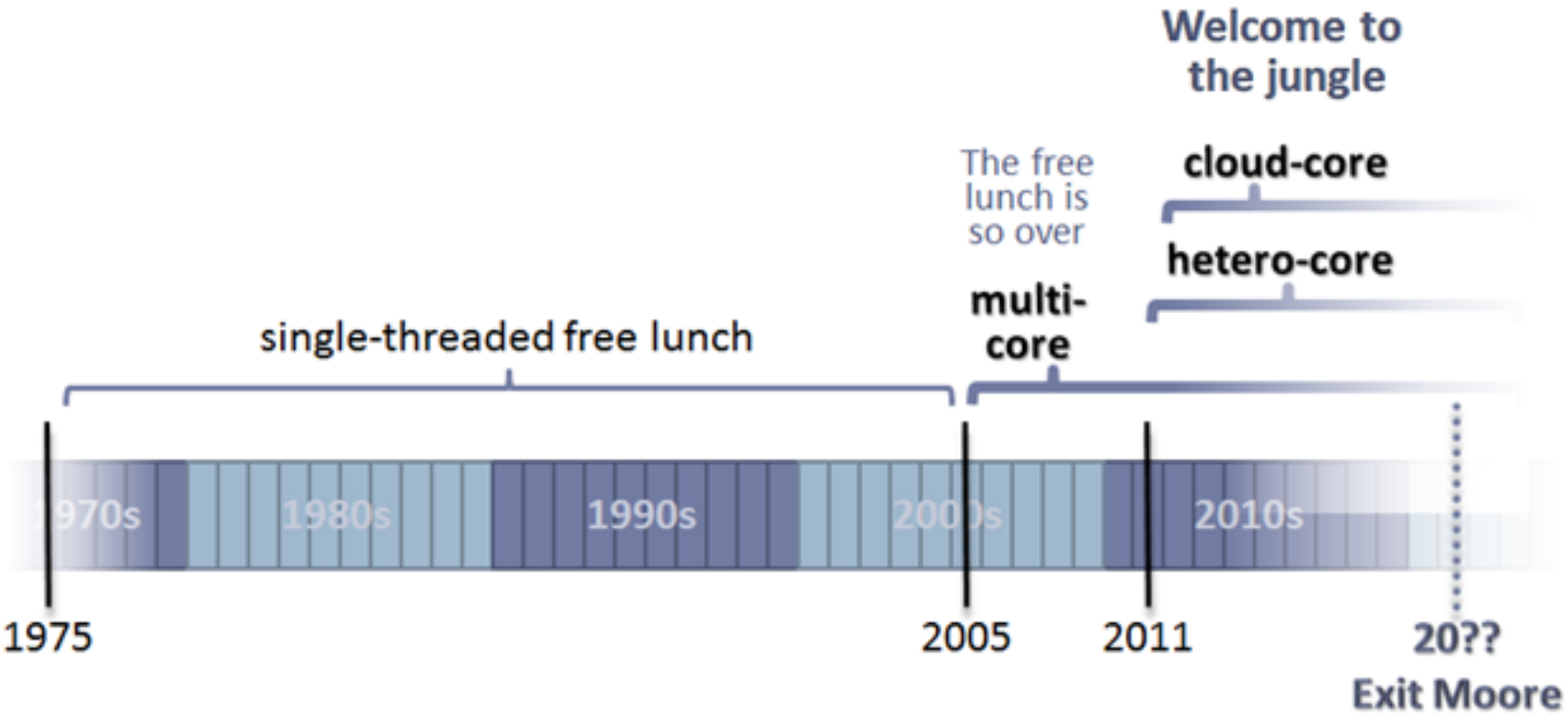


THE FASTEST

- Fastest operational supercomputer (Nov 2009): Oak Ridge National Laboratory 'Jaguar' Supercomputer
  - composed by Cray XT5 and XT4 Jaguar machines
    - based on AMD Opteron CPU - 6 cores per CPU
  - more than 224,000 cores
  - a sustained processing rate of 1.759 PFLOPS
- Fastest cluster (December 2009): Folding@home
  - reported over 7.8 petaflops of processing power
    - 2.3 petaflops of this processing power is contributed by clients running on PlayStation 3 systems - Cell microprocessor CPU (Sony, Toshiba, IBM) - 3.2 GHz PowerPC-based "Power Processing Element" (PPE) + 8 Synergistic Processing Elements (SPEs).
    - 5.1 petaflops is contributed by GPU2 client.
- (?) Google Cluster Architecture - search engine system - at Googleplex
  - estimated total processing power of between 126 and 316 teraflops, as of April 2004
  - 450,000 servers in the server farm estimated in 2006
  - recent estimation: 20 to 100 petaflops
    - ~500000 servers based on dual quad-core Xeon processors, at 2.5 GHz or 3 GHz.

“THE HARDWARE (CORE) JUNGLE”

- *“The Free Lunch is Over. Now Welcome to the Hardware Jungle” (Herber Sutter, [SUT-12])*



“THE HARDWARE JUNGLE”

Exploitability	Summary	Stages / Alternatives	Software Impact	Examples
Moore’s motherlode: Unicore	Make single core more complex to run single-threaded code faster	1970s & 1980s: Add one big feature per chip generation 1990s & 2000s: Several smaller improvements/gen	The free lunch: Ship an EXE that will just run faster on new hardware	Single-core x86, SPARC, ARM
Secondary veins: Multicore	Deliver more cores per chip	2005-20??: Deliver several big cores 2012-20??: Deliver lots of smaller cores	Must write parallel code Must write very parallel code	SPARC Niagara, x86 Intel MIC
Tertiary veins: Hetero manycore	Deliver different kinds of cores Because the cores are simpler, yields large one-time jump in #cores	Big/fast (complex) vs. small/slow (simpler) General-purpose (traditional CPU core) vs. special-purpose (e.g., GPU core)	Still less exploitable: Must write heterogeneous and locally distributed parallel code	Cell (e.g., PS3) Intel Xeon+MIC AMD and NVIDIA GPUs, incl. on-die (Fusion and Tegra 3)

FLYNN'S TAXONOMY

- Categorization of all computing systems according to the *number* of *instruction stream* and *data stream*
  - stream as a sequence of instruction or data on which a computer operate
- Four possibilities
  - **Single Instruction, Single Data (SISD)**
    - Von-Neumann model, single processor computers
  - **Single Instruction, Multiple Data (SIMD)**
    - single instruction stream concurrently broadcasted to multiple processors, each with its own data stream
    - fine grained parallelism, vector processors
  - **Multiple Instruction, Single Data (MISD)**
    - no well known systems fit this
  - **Multiple Instruction, Multiple Data (MIMD)**
    - each processor has its own stream of instructions operating on its own data

MIMD MODELS

- MIMD category can be then decomposed according to memory organization
  - **shared memory**
    - all processes (processors) share a single address space and communicate each other by writing and reading shared variables
  - **distributed memory**
    - each process (processor) has its own address space and communicate with other process by *message passing* (sending and receiving messages)

MIMD FURTHER CLASSIFICATIONS

- Two further classes for shared-memory computers
  - **SMP (Symmetric Multi-processing Architecture)**
    - all processors share a connection to a common memory and access all location memories at equal speed
  - **NUMA (Non-uniform Memory Access)**
    - the memory is shared, by some blocks of memory may be physically more closely associated with some processors than others

MIMD FURTHER CLASSIFICATIONS

- Two further classes for distributed-memory computers
  - **MPP (Massively Parallel Processors)**
    - processors and the network infrastructure are tightly coupled and specialized for a parallel computer
    - extremely scalable, thousands of processors in a single system
    - for High-Performance Computing (HPC) applications
  - **Clusters**
    - distributed-memory systems composed of off-the-shelf computers connected by an off-the-shelf network
    - e.g. Beowulf clusters (= clusters on Linux)
  - **Grid**
    - systems that use distributed, heterogeneous resources connected by LAN and/or by WAN, without a common point of administration

WHY CONCURRENT PROGRAMMING: PERFORMANCE

- **Performance improvement**
  - increased application **throughput**
    - by exploiting parallel hardware
  - increased application **responsiveness**
    - by optimizing the interplay among CPU and I/O activities

- Quantitative measurement for performance: **speedup**

$$S = \frac{T_1}{T_N}$$

N is the number of processors
T_1 is the execution time of the sequential algorithm
T_N is the execution time of the parallel algorithm with N processors

AMDAHL'S LAW

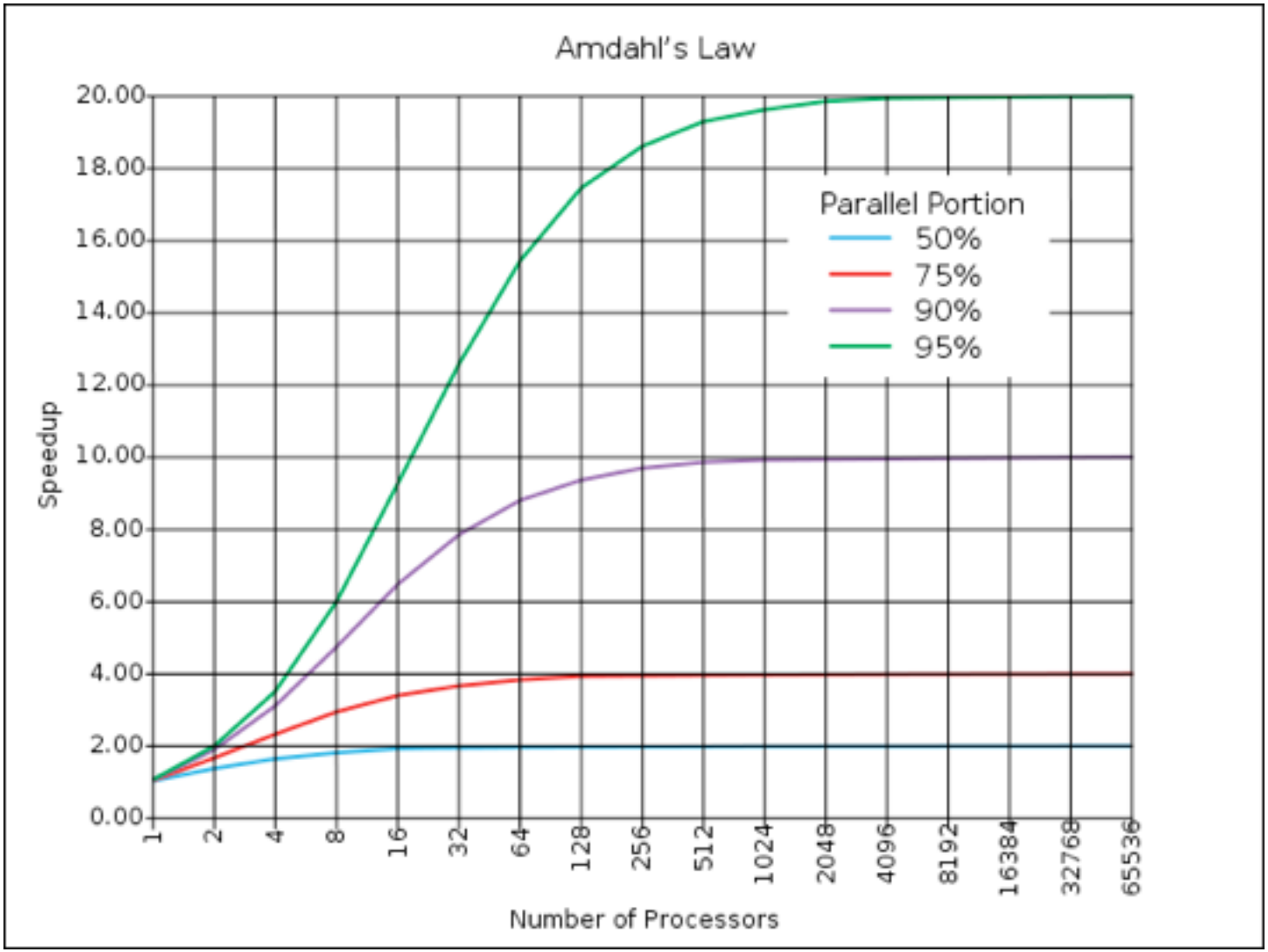
- *Maximum* speedup parallelizing a system:

$$S = \frac{1}{1 - P + \frac{P}{N}}$$

  - P is the proportion of a program that can be made parallel
  - (1-P) is then the part that cannot be parallelized

- Theoretically maximum for P = 1 (*linear speedup)*
  - actually there are specific cases with S > N (*super-linear)* speedup

AMDAHL'S LAW


Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Parallel Portion
50%
75%
90%
95%
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

THAT MEANS:

serializations / sequentializations
are poison for performances
(e.g. locking)

...but are often necessary for correctness
(e.g. safety properties)

> struggle & tradeoffs
(..and a lot of research about it)

BUT DON'T FORGET *EFFICIENCY*

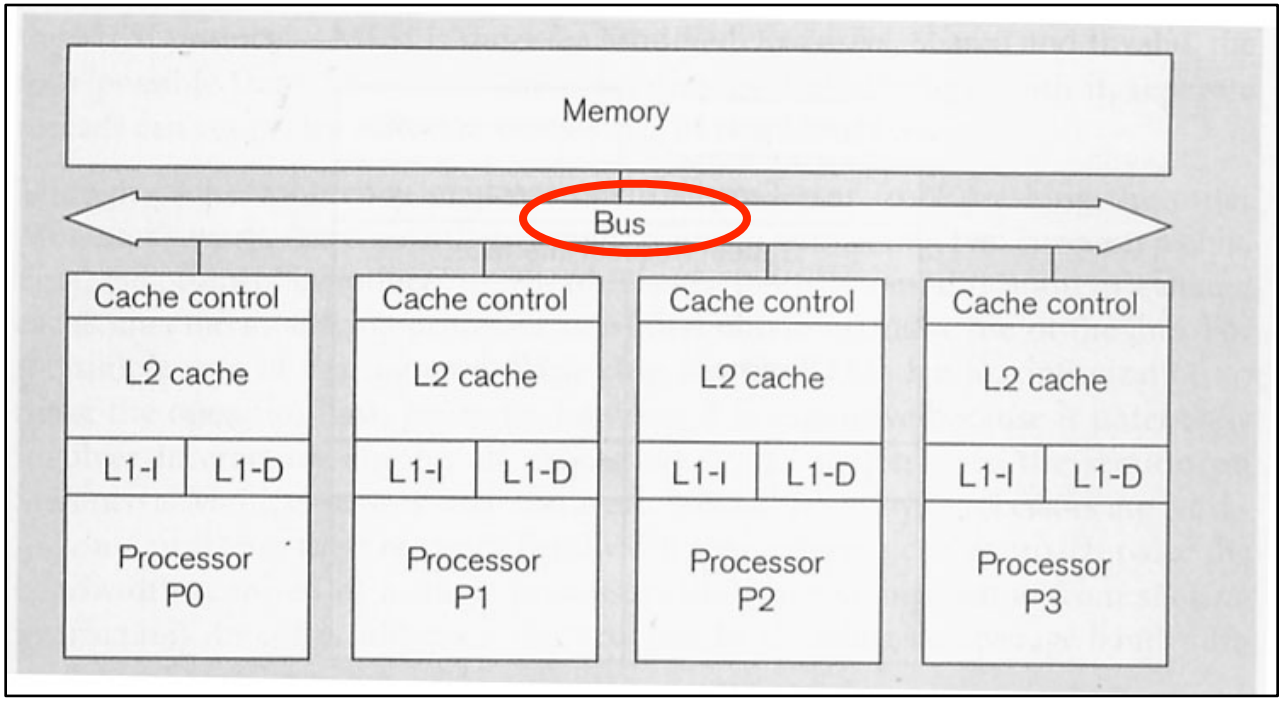
- Normalized measure of speed-up indicating *how* effectively each processor is used

$$E = \frac{S}{P}$$

- The ideal efficiency is 1 = all processors are used at full capacity
  - typically lower

A NEW BOTTLENECK: MEMORY

- Shared memory and bus as potential bottleneck
  - only one memory operation takes place at a time
  - importance of the cache
    - cache coherency protocol more and more complex and smart



WHY CONCURRENT PROGRAMMING: DESIGN & ABSTRACTION

- **Abstraction** and engineering
  - define a proper level of abstraction for programs which interact with the environment, control multiple activities and handle multiple events..
    - *objects from OOP are not enough*
- Concurrency as a tool for **software design and construction**
  - *rethinking to the way in which we solve problems*
    - basic algorithms & data structures
  - *rethinking to the way in which we design and build systems*
    - new level of abstraction
      - different kind of decomposition, modularization, encapsulation
- Affecting the full engineering spectrum
  - **modelling**, **design**, **implementation**, **verification**, **testing**

BASIC JARGON OF CONCURRENT PROGRAMMING: PROCESSES

- **Processes** ~ a sequential program in execution
  - the basic unit of a concurrent system, single thread of control
    - *logical* thread of control, not (necessarily) physical
  - sequence of instructions operating together as a group
    - unit of work *(task)*
  - *abstract / general concept*
    - *…*not necessarily related to OS processes
- ***speed independence***
  - process execution is meant to be completely *asynchronous* with each other
    - we can't do any assumption about their speed
  - **non-determinism**

BASIC JARGON OF CONCURRENT PROGRAMMING: INTERACTION

- **Process interaction**
  - any non trivial concurrent program is based on *multiple* processes that need to *interact* in some way in order to achieve the objective of the system
- Basic kinds of interaction:
  - **cooperation**
  - **competition / contention**
  - **interferences**

PROCESS INTERACTION: COOPERATION

- Refers to interactions which are both *expected* and *wanted*
  - they are part of the semantics of the concurrent program
- Two basic kinds
  - **communication**
    - concerns the need of realizing an information flow among processes, typically realized in terms of messages
    - introduction of specific supports for the exchange of messages
  - **synchronization**
    - concerns the explicit definition or presence of **temporal relationships** or dependencies among processes and among actions of distinct processes
    - introduction of specific supports for the exchange of temporal signals

PROCESS INTERACTION: CONTENTION / COMPETITION

- Refers to interactions which are *expected* and *necessary*, but *not wanted*
  - typically concerns the need of coordinating the access by multiple processes to shared resources
- Two basic class of problems
  - **mutual exclusion**
    - ruling the access to shared resources by distinct processes
  - **critical sections**
    - ruling the concurrent execution of blocks of actions by distinct processes

SYNCHRONIZATION VS. MUTUAL EXCLUSION

- Different - even if related - concepts
  - "synchronization = mutual exclusion urban legend" [BUH-05]
    - false story, still present in textbooks / research papers
  - synchronization defines a *timing relationship* among processes
    - *maintaining time-relationships which includes actions happening at the same time or happening at the same relative rate or simply some action having to occur before another* (precedence relationships)
  - mutual-exclusion defines a *restriction* on access to shared data
    - mutual-exclusion is meaningless if no shared data is involved
- Relationships
  - mutual-exclusion typically require some forms of *implicit synchronization*
    - blocking some actions, waiting for other actions to complete
  - synchronization does not necessarily require any kind of shared data and the mutual exclusion

ON THE DIFFICULTY OF SYNCHRONIZATION: TOY EXAMPLE: “BUY-THE-MILK” PROBLEM

- “Alice and Bob live together, happily without cell-phones. Both are responsible to buy the milk when it finishes...”

Time	Alice	Bob
5:00	Arrive home	
5:05	Look in the fridge; no milk	
5:10	Leave for a grocery	
5:15		Arrive home
5:20		Look in the fridge; no milk
5:25	Buy milk	Leave for grocery
5:30	Arrive home; put milk in fridge	
5:40		Buy milk
5:45		Arrive home; oh no!

A SOLUTION: NOTES IN THE FRIDGE (1/2)

- Looking for a solution to ensure that:
  - only one person buys the milk, when there is no milk
  - someone always buys the milk, when there is no milk
- Tentative solution: using notes on the fridge!

PROGRAM for Alice & Bob

```
1 if (no note) then
2   if (no milk) then
3     leave note
4     buy milk
5     remove note
6   fi
7 fi
```

  - "if you find that there is no milk and there is no note on the door of the fridge, then leave a note on the fridge's door, go and buy milk, put the milk in the fridge, and remove your note."
- Does it work? Not always actually...

A SOLUTION: NOTES IN THE FRIDGE (2/2)
(..NOT SO EASY, ACTUALLY..)

Time	Alice	Bob
5:00	Arrive home	
5:05	Look at the fridge; no note	
5:10	...ops! need a toilet	
5:15	...still at the toilet...	Arrive home
5:20	...still at the toilet...	Look at the fridge; no note
5:21	...still at the toilet...	Look in the fridge; no milk (argh)
5:22	...still at the toilet...	leave note
5:25	...still at the toilet...	go and buy milk
5:45	look in the fridge: no milk (*)	...
5:50	leave note...	

[*] Alice does not realize that a note was put on the fridge (she is not really a good observer) and strictly follows the established program

PROCESS INTERACTION: INTERFERENCES

- Refers to interactions which are *neither expected*, *nor wanted*
  - producing bad effects only when the ratio among the process speeds assumes specific values (time-dependent errors)
- The “nightmare” of concurrent programming
  - “*heisen-bugs*”
    - when debugging influence the bugs...

INTERFERENCES: RACE CONDITIONS

- **race condition** or **race hazard** or simply **race**
  - whenever two or more processes concurrently access and update shared resources, and the result of the single update depends on the specific order occurring in process access
- Related to two main types of programming errors
  - bad management of expected interactions
  - presence of spurious interactions not expected in the problem

CRITICAL SITUATIONS

- Interferences and errors in concurrent programs can lead to *critical situations* for the concurrent system in the overall
  - **Deadlock** (...or *deadly embrace* (Dijkstra))
  - **Starvation** (or *unfairness*)
  - **Livelock**

DEADLOCK

- Situation wherein two or more competing actions (processes) are waiting for the other to finish, and thus neither ever does
  - typically concerns the release of a locked shared resource, the reception of a temporal signal or a message

STARVATION

- Situation wherein a process is blocked in an infinite waiting
- *Resource starvation*
  - the process is perpetually denied in accessing necessary resources.
  - without those resources, the program can never finish its task

LIVELOCK

- A *livelock* is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing
- Livelock is a special case of resource starvation
  - the general definition only states that a specific process is not progressing

“STANDING ON THE SHOULDERS OF GIANTS”: THE ORIGIN OF CONCURRENT PROGRAMMING


Edgar W. Dijkstra
(1930-2002)


Per Brinch Hansen
(1938-2007)


Sir Anthony (Tony) Hoare
(1934)

THE INVENTION OF CONCURRENT PROGRAMMING (NOTES FROM [HAN-01)]

- One original motivation:
  developing *reliable operating systems*

- But from the beginning it was recognized that the principles of concurrent programming...
  "*have a general utility that goes beyond operating systems..* " (P.B. Hansen 1971)

1960s - 1970s

- 1961: birth of **multiprogramming**
  - Kilburn & Howarth introduce the use of **interrupts** to simulate concurrent execution of programs on the ATLAS computer
- early multiprogramming systems were programmed in assembly language **without any conceptual foundation**
  - huge and unreliable multiprogrammed OS

  => software crisis (end of the 1960s) (Naur, 1969)

  => need of having a deeper understanding of concurrent programming
- In 15 years (from ~1965 to end of the 1970s) computer scientists
  - discovered the ***fundamental concepts***
  - expressed by ***programming notations***
  - included them in ***programming languages***
  - and used these languages to write operating systems
- 1970s
  - the new programming concepts used to write first textbooks on the principle of OS and concurrent programming

THE MAIN CONCEPTS

- All the main contributions were from the three giants: Dijkstra, Hansen, Hoare

Fundamental Concepts

Asynchronous processes
Speed independence
Fair scheduling
Mutual exclusion
Deadlock prevention
Process communication
Hierarchical structure
Extensible system kernels

Programming Language Concepts

Concurrent statements
Critical regions (*~critical sections)*
Semaphores
Message buffers *(~bounded buffers)*
Conditional critical regions
Secure queueing variables
Monitors
Synchronous message communication
Remote procedure calls

CLASSIC PAPERS

1. E. W. Dijkstra, Cooperating Sequential Processes (1965).
2. E. W. Dijkstra, The Structure of the THE Multiprogramming System (1968).
3. P. Brinch Hansen, RC 4000 Software: Multiprogramming System (1969).
4. E. W. Dijkstra, Hierarchical Ordering of Sequential Processes (1971).
5. C. A. R. Hoare, Towards a Theory of Parallel Programming (1971).
6. P. Brinch Hansen, An Outline of a Course on Operating System Principles (1971).
7. P. Brinch Hansen, Structured Multiprogramming (1972).
8. P. Brinch Hansen, Shared Classes (1973).
9. C. A. R. Hoare, Monitors: An Operating System Structuring Concept (1974).
10. P. Brinch Hansen, The Programming Language Concurrent Pascal (1975).
11. P. Brinch Hansen, The Solo Operating System: A Concurrent Pascal Program (1976).
12. P. Brinch Hansen, The Solo Operating System: Processes, Monitors and Classes (1976).
13. P. Brinch Hansen, Design Principles (1977).
14. E. W. Dijkstra, A Synthesis Emerging? (1975).
15. C. A. R. Hoare, Communicating Sequential Processes (1978).
16. P. Brinch Hansen, Distributed Processes: A Concurrent Programming Concept (1978).
17. P. Brinch Hansen, Joyce|A Programming Language for Distributed Systems (1987).
18. P. Brinch Hansen, SuperPascal: A Publication Language for Parallel Scientific Computing (199
19. P. Brinch Hansen, Efficient Parallel Recursion (1995)

CONCURRENT LANGUAGES AND MACHINES

- To *describe / specify* a concurrent program we need **concurrent programming languages**
  - enabling programmers to write down programs as set of instructions to be executed concurrently
- To *execute* a concurrent program we need a **concurrent machine**
  - a machine (which can be abstract) designed to handle the execution of multiple sequential processes, by exploiting multiple processors (physical or virtual)

CONCURRENT MACHINES

- A **concurrent machine** provides:
  - a support for the execution of concurrent programs and realizing then concurrent computations
  - as many virtual processors as the number of processes composing the concurrent computation
- Providing basic mechanisms for:
  - **multiprogramming**
    - virtual processors generation and management
  - **synchronization** and **communication**
  - **access control** to resources

BASIC MECHANISMS

- **Multiprogramming**
  - set of mechanisms that make it possible to create new virtual processors and allocate physical processors of the lower-level machine to the virtual processors by means of scheduling algorithms
- **Synchronization** and **Communication**
  - two different typologies of mechanisms, related to two different *architectural models* for concurrent machines:
    - **shared memory model**
      - presence of a shared memory among the virtual processors
      - example: multi-threaded programming
    - **message passing model**
      - every virtual processor has its own memory and no shared memory among processors is present
      - every communication and interaction among processors is realized through message passing

FROM MACHINES TO PROGRAMMING LANGUAGES

- Programming languages for specifying concurrent programs on top of concurrent machines
  - programs organized as sets of sequential processes to be executed concurrently on the virtual processors of the concurrent machine
- Basic constructs for
  - **specifying concurrency**
    - creation of multiple processes
  - **specifying process interaction**
    - synchronization and communication
    - mutual exclusion

CONCURRENT PROGRAMMING LANGUAGES - DESIGN APPROACHES

- Three main design approaches
  - sequential language + library with concurrent primitives
    - e.g. C + PThreads
  - language designed for concurrency
    - e.g. OCCAM, ADA, Erlang
  - hybrid approach
    - sequential paradigm extended with a native support for concurrency
      - e.g. Java, Scala
    - library and patterns based on basic mechanisms
      - e.g. java.util.concurrent

BASIC NOTATIONS AND CONSTRUCTS:

- First proposals (back to ~1960/1970)
  - fork/join
  - cobegin/coend
- More recent proposals
  - first-class abstractions and constructs for defining *processes*
    - called also *tasks*
  - e.g. ADA, Erlang languages
- Mainstream languages
  - support for threads and **multi-threaded programming**
    - e.g. Java
  - raise of **asynchronous** & event-driven programming
- Research landscape - several proposals, among the others:
  - **actor-based** models
    - …more and more adopted also in the main stream
    - a reference model for *Concurrent OOP*
  - **active objects**
  - **STM - Software Transactional Memory**
  - **reactive programming**
  - **agent-oriented programming**

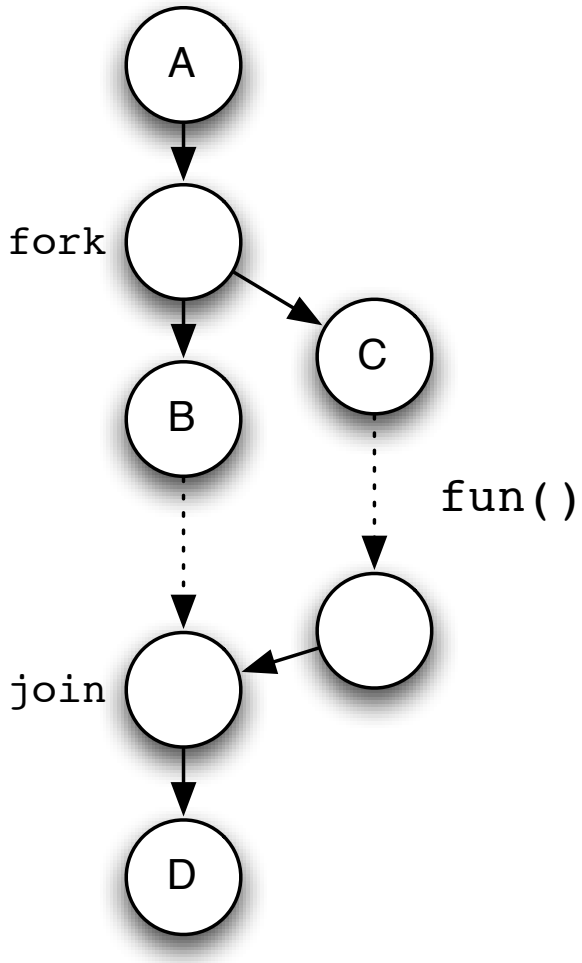
FORK / JOIN

- Among the first basic language notations for expressing concurrency (Conway 1963, Dennis 1968)
  - adopted in UNIX system / POSIX, provided by MESA language (1979)
- **fork** primitive
  - behavior similar to procedure invocation, with the difference that a new process is created and activated for executing the procedure
    - input param: procedure to be executed
    - output param: the identifier of the process created
  - > it results in a bifurcation of the program control flow
    - the new process (child) is executed asynchronously with respect to the generating process (parent) and existing processes
- **join** primitive
  - it detects when a process created by a fork has terminated and it synchronize current control flow with such event
    - input parameter: the identifier of the process to wait
  - > it results in a join of independent control flows

FORK / JOIN IN MESA

```
process p;
A: ...;
   p=fork fun;
B: ...;
   join p;
D: ....;

void fun() {
  C: ....;
}
```



FORK / JOIN: WEAKNESSES

- Pro
  - general and flexible
    - can be used to build any kind of concurrent application
- Cons
  - low-level of abstraction
    - not providing any discipline for structuring complex processes
    - error-prone
  - programs difficult to read
    - it is hard getting from the text an idea of what processes are active in a specific point of the program
  - no explicit representation of the process abstraction
    - as abstraction to organize the overall system

COBEGIN / COEND CONSTRUCT

- Construct proposed by Dijkstra (1968) to provide a discipline for concurrent programming
  - enforcing the programmer to follow a specific scheme to structure concurrent programs
- Concurrency is expressed in blocks:

```
cobegin
   S1;
   S2;
   ...
   Sn;
coend
```

- instructions S1, S2, Sn are executed in parallel

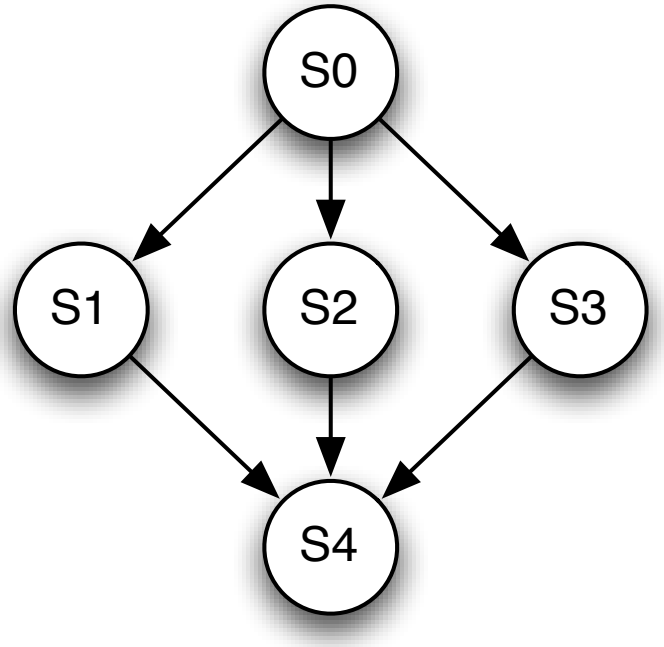
- an instruction Si can be as complex as a full program (it can include nested cobegin/coend)

- a parallel structure terminates only when all its components (processes) have terminated

- The process executing a cobegin (pared) creates as many processes (children) as the number of instructions in the body and suspends its execution until all the processes have terminated

EXAMPLE

```
S0
cobegin
   S1;
   S2;
   S3;
coend
S4;
```



COBEGIN / COEND

- Pro
  - stronger discipline in structuring a concurrent program with respect to fork/join primitives
  - programs are more readable
- Cons
  - less flexibility than fork/join
    - how to create N concurrent processes, where N is known only at runtime ?
  - also in this case we haven't an explicit abstraction encapsulating the process

LANGUAGES WITH FIRST-CLASS SUPPORT FOR PROCESSES

- Introducing a notion of process *as first-class entity* of the concurrent language (and of the concurrent machine)
  - as “modules” to organize a program (static) and the system (runtime)
  - explicit encapsulation of the control flow
- Examples
  - historical one
    - **Concurrent Pascal** (70ies)
    - OCCAM (1980...OCCAM3 ~90ies)
    - ...
  - more recent / in use examples
    - **ADA** (~1980 up today with new versions - ADA95 with OO),
    - **Erlang** (end of 90ies up today)
      - used in particular by telecom industries

CONCURRENCY IN MAINSTREAM LANGUAGES

- For the most part, mainstream languages - both procedural (like C) and Object-Oriented (Java) - provide a support for the creation and execution of processes by means of *libraries*
  - without extending the language
  - not completely true for Java

> Support for *multi-threaded programming*

  - threads as implementation of the abstract notion of process
    - also called “lightweight processes” by referring to OS “heavyweight processes”
  - not to be confused with the notion of process as defined in OS
    - process as a programming in execution, with one or multiple control flows (threads)

- Main examples
  - multi-threaded programming in Java
  - Pthread library for C/C++ language on POSIX systems

MULTITHREADED PROGRAMMING IN JAVA

- Java has been the first “mainstream” language providing a native support for concurrent programming
  - “conservative approach”
    - the language is still ~purely OO, with no explicit construct for defining processes (threads)
    - introduction of some keywords and mechanisms for concurrency
      - synchronized blocks, wait / notify mechanisms
- The abstract notion of process is implemented as a *thread,* with a direct mapping onto OS support for threads
  - thread defined by specific classes, so at runtime they are objects

THREADS IN JAVA

- Thread model
  - a thread is defined by a single control flow, sharing memory with all the other threads
    - private stack, common heap
  - each Java program contains at least one thread, corresponding to the execution of the main in the main class
  - further threads can be dynamically created and activated with program execution, running concurrently
- Thread (process) definition
  - threads are objects of classes extending Thread class provided in java.lang package
    - multiple process types can be defined, as different classes extending java.lang.Thread
- Thread (process) execution
  - thread object can be instantiated and “spawned” by invoking the **start** method, beginning the execution of the process

JAVA THREADS: SIMPLE EXAMPLE

```
class ClockVisualizer extends Thread {
  private int step;

  public ClockVisualizer(int step){
    this.step=step;
  }

  public void run(){
    while (true) {
      System.out.println(new Date());
      try {
        sleep(step);
      } catch (Exception ex){
      }
   }
  }
}

class TestClockVisualizer {
  static public void main(String[] args) throws Exception {
    ClockVisualizer clock = new ClockVisualizer(1000);
    clock.start();
  }
}
```

MULTITHREADED PROGRAMMING WITH C/C++ & Pthreads

- Defined in the POSIX (Portable Operating System Interface) context the Pthread (POSIX-thread) library provides a set of basic primitives for multithreaded programming in C / C++
  - the abstract notion of process is implemented as thread
  - differently from Java, process body is specified by means of a procedure
  - the standard defines just the interface / specification, not the implementation (which depends on the specific OS)
    - An implementation is available on every modern OS, including Solaris, Linux, Tru64 UNIX, Mac OS X and Windows
- Basic API for threads creation and synchronization
  - good tutorial: http://www.llnl.gov/computing/tutorials/pthreads/

Pthread API: SOME FUNCTIONS

- Interface defined in `pthread.h`
- Two main data types
  - **`pthread_t`**
    - thread identifier data type
  - **`pthread_attr_t`**
    - data structure for specifying thread attributes
- Among the main functions
  - thread creation (Fork)
    - **`pthread_create`**`(pthread_t* tid, pthread_attr_t* attr, void* (*func)(void*), void* arg)`
    - **`pthread_attr_init(pthread_attr_t*)`**
      - for setting up attributes
  - thread termination
    - **`pthread_exit(int)`**
  - thread join
    - `int` **`pthread_join`**`(pthread_t thread, void **value_ptr);`

AN EXAMPLE

- Creation of 5 threads running concurrently

```
#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() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

RESEARCH LANDSCAPE

- Many proposals in the last 30 years
  - most of them are extensions of sequential programming languages
- Among the main families:
  - Concurrent Object-Oriented Programming (COOP)
    - extending OO with concurrency
  - main examples
    - **actor**-based models
    - active objects
    - objects + asynchronous programming extensions
    - agent-based models

ACTORS

- Model proposed originally by **Carl Hewitt** in 1977 in the context of Distributed Artificial Intelligence [HEW-77]
  - adopted and further developed by **Gul Agha** & colleagues as a model unifying objects and concurrency [AGH-96]
- **Actor** as unique abstraction
  - *autonomous* entities, possibly distributed on different machines, executing concurrently and *communicating through asynchronous message passing*
    - no shared memory, every actor has a mailbox
- First languages
  - **ACT** family (ACT/1, ACT2, ACT/3), **ABCL** family (ABCL/1,ABCL/R3)
- Current languages
  - **Erlang** is based on Actors
- Implemented as a pattern on top of existing languages
  - many Java-based frameworks
    - es: ActorFoundry, http://osl.cs.uiuc.edu/af/
  - **Scala** language, http://www.scala-lang.org/node/242

ACTOR ABSTRACTION

- An actor is a computational entity that, **in response to a message** it receives, can concurrently:
  - **send** a finite number of messages to other Actors;
  - **create** a finite number of new Actors;
  - designate the behavior to be used for the next message it receives *(replacing behaviour)*
- There is no assumed list to the above actions and they could be carried out concurrently.
- An Actor can only communicate with Actors to which it is connected.
  - it can directly obtain information only from other Actors to which it is directly connected
  - connections can be implemented in a variety of ways:
    - direct physical attachment
    - memory or disk addresses
    - network addresses / email addresses

ACTOR BASIC PRIMITIVES

- Only three primitives (actions) to compose an actor behaviour
  - **send**
    - asynchronously sending a message to a specified actor
    - it is to concurrent programming what procedure invocation is to sequential programming
  - **create**
    - create an actor with the specified behavior
    - it is to concurrent programming what procedure abstraction is to sequential programming
  - **become**
    - specify a new behavior (local state) to be used by actor to respond to the next message it processed
    - gives actors a history-sensitive behaviour necessary for shared, mutable data objects

STATE-OF-THE-ART

- Languages
  - **Erlang**, E language, SALSA, AmbientTalk…
  - **HTML 5 WebWorker**
    - based on the actor model
  - **DART** Language for Web app programming
    - “isolates”
- Frameworks (over existing languages)
  - (on JVM) Scala Actors library, Kilim, Jetlang, **ActorFoundry**, Actor Architecture, Actors Guild, JavAct, AJ
    - survey in [KAR09]
  - (on .NET) Microsoft’s Asynchronous Agents Library, Retlang, **Orleans** (for cloud computing)
  - Act++, Thal (on C/C++), Acttalk (on Smalltalk), Stackless Python (on Python), Stage (on Ruby)….

TASTE OF ACTORS IN ACTORFOUNDRY

```
public class PingActor extends Actor {
  ActorName otherPinger;
  @message
  public void start(ActorName other) {
    otherPinger = other;
    send(otherPinger, "ping", self(), Id.stamp()+"called from " + self());
  }
  @message
  public void ping(ActorName caller, String msg) {
    send(stdout, "println", Id.stamp()+"Received ping (" + msg +") from " + caller + "...");
    send(caller, "alive", Id.stamp()+self().toString() + " is alive");
  }
  @message
  public void alive(String reply) {
    send(stdout, "println", Id.stamp()+"Received " + reply + " from pinged actor");
  }
}
```

```
public class PingBoot extends Actor {

  @message
  public void boot()  throws RemoteCodeException {
    ActorName pinger1 = null;
    ActorName pinger2 = null;

    pinger1 = create(osl.examples.ping.PingActor.class);
    pinger2 = create(osl.examples.ping.PingActor.class);

    send(pinger1, "start", pinger2);
  }
}
```

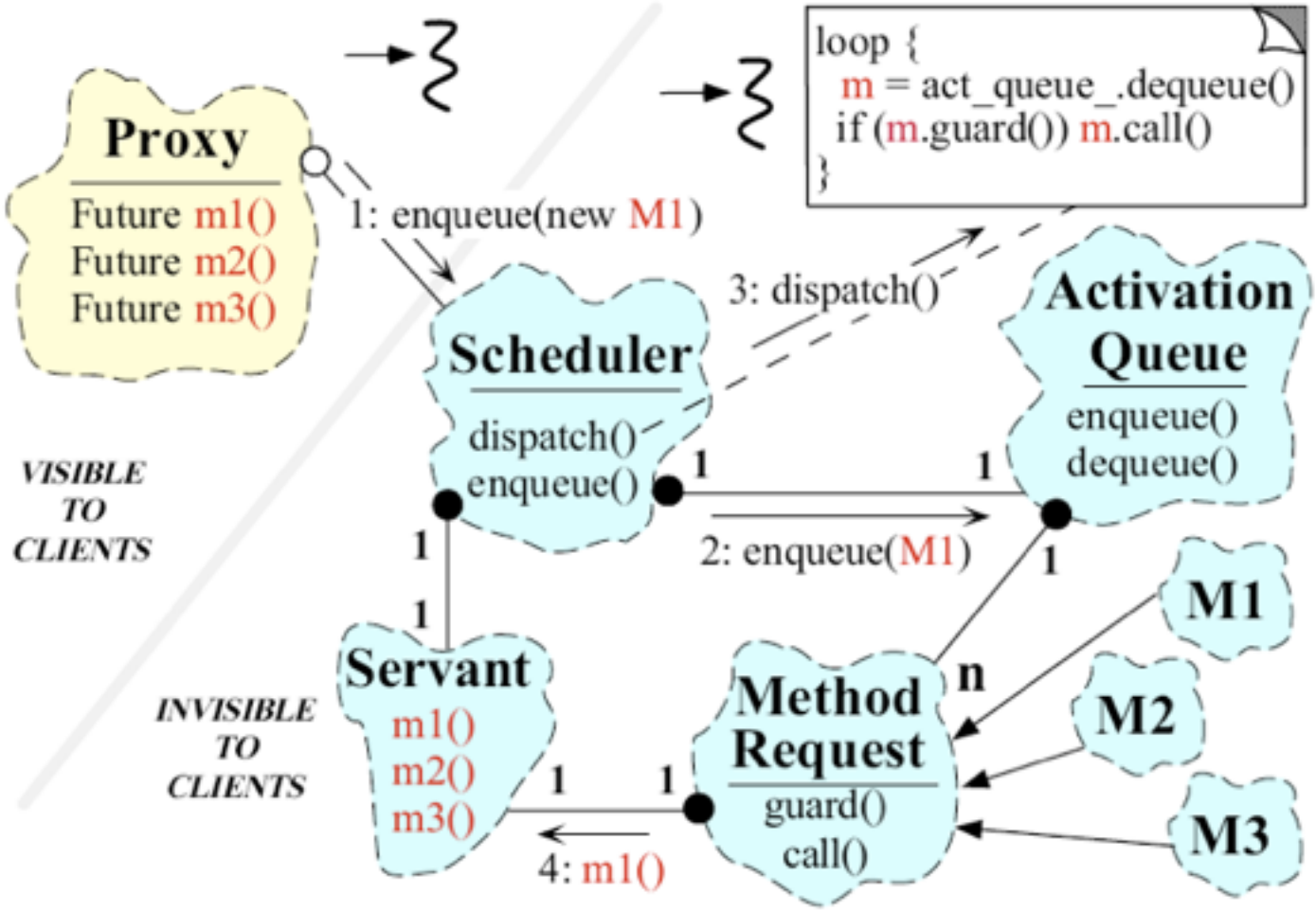
TASTE OF ACTORS IN SCALA

```
class Ping(count: int, pong: Actor) extends Actor {
  def act() {
    var pingsLeft = count - 1
    pong ! Ping
    while (true) {
      receive {
        case Pong =>
          if (pingsLeft % 1000 == 0)
            Console.println("Ping: pong")
          if (pingsLeft > 0) {
            pong ! Ping
            pingsLeft -= 1
          } else {
            Console.println("Ping: stop")
            pong ! Stop
            exit()
          }
      }
    }
  }
}
```

ACTIVE OBJECTS

- Integrating concurrency within the OO paradigm
  - active + passive objects
  - implicit thread creation + synchronization mechanisms
- Examples
  - Languages with first-class support
    - “Hybrid” language [NIE87]
    - more recent: Creol [JOH06], JCoBoxes [SCH10], ABS [JOH12]
  - Active Objects as a pattern [LAV-96]
    - can be implemented on top of sequential OO languages with a basic thread support

ACTIVE-OBJECT COMPONENTS



SUMMARY

- Concurrent programming
  - motivations: HW evolution
  - basic jargon
    - processes interaction, cooperation, competition,
    - mutual exclusion, synchronization
    - problems: deadlocks, starvation, livelocks
- A little bit of history
  - Dijkstra, Hoare, Brinch-Hansen
- Concurrent languages, mechanisms, abstractions
  - overview

BIBLIOGRAPHY

- **[HAN-73]**
  - Per Brinch Hansen - “Concurrent Programming Concepts”, *ACM Computing Surveys*, Vol. 5, No. 4, Dec. 1973
- **[HAN-01]**
  - Per Brinch Hansen - The Invention of Concurrent Programming” in “The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls”, Springer-Verlag, 2002
- **[SUT-12]**
  - Sutter’s Mill. Herb Sutter on software, hardware, and concurrency. "Welcome to the Jungle". http://herbsutter.com/welcome-to-the-jungle/
- **[AND-83]**
  - Gregory Andrews and Fred Schneider - “Concepts and Notations for Concurrent Programming”, *ACM Computing Survey*s, Vol. 15, No. 1, March 1983
- **[CLE-96]**
  - Rance Cleaveland, Scott Smolka et al - “Strategic Directions in concurrency Research”, *ACM Computing Survey*s, Vol. 28, No. 4, Dec. 1996
- **[ROS-97]**
  - Roscoe, A. W. (1997). *The Theory and Practice of Concurrency*. Prentice Hall. ISBN 0-13-674409-5.
- **[BUH-05]**
  - Peter Buhr and Ashif Harji. “Concurrent Urban Legends”. *Concurrency and Computation: Practice and Experience.* 2005. 17:1133-1172.

BIBLIOGRAPHY

- **[HEW-77]**
  - C. Hewitt. Viewing Control Structures as Pattern of Passing Messages. Journal of Artificial Intelligence, 8(3):323-364, 1977
- **[AGH-86]**
  - Gul Agha. Actors: A model of concurrent computation in distributed systems. MIT Press, 1986.
- **[NIE-87]**
  - Oscar Nierstrasz. Active Objects in Hybrid. SIGPLAN Notices, 1987
- **[LAV-96]**
  - R. Greg Lavender, Douglas C. Schmidt. Active Object An Object Behavioral Pattern for Concurrent Programming. Proc.Pattern Languages of Programs, 1996
- **[GEL-92]**
  - D. Gelernter, N. Carriero. Coordination Languages and their Significance. Communications of the ACM. Vol 33, Issue 2, Feb. 1992
- **[JOH06]**
  - Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. 2006. Creol: a type-safe object-oriented model for distributed concurrent systems. Theor. Comput. Sci. 365, 1 (November 2006), 23-66
- **[KAR09]**
  - Karmani, Shali, Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. *PPPJ 09*
- **[SCH10]**
  - Jan Schäfer and Arnd Poetzsch-Heffter. 2010. JCoBox: generalizing active objects to concurrent components. In Proceedings of the 24th European conference on Object-oriented programming (ECOOP'10), Theo D'Hondt (Ed.). Springer-Verlag, Berlin, Heidelberg, 275-299.

BIBLIOGRAPHY

- **[JOH-12]**
  - Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, Martin Steffen. ABS: A Core Language for Abstract Behavioral Specification. Lecture Notes in Computer Science Volume 6957, 2012, pp 142-164