v1.0 BETA

Sistemi Concorrenti e di Rete LS

II Facoltà di Ingegneria - Cesena

a.a 2008/2009

[module 1.3]

CONCURRENT LANGUAGES AND MACHINES

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)

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 Intruction, 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**
    - 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
- 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
  - **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 connecred by LAN and/or by WAN, without a common point of administration

MPP EXAMPLE: THINKING MACHINE CM-5

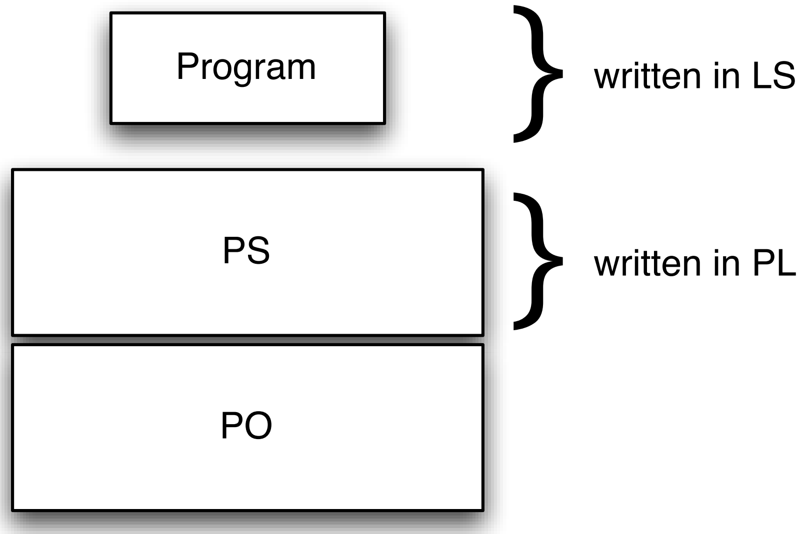
- MIMD, 512 Processors

FROM PHYSICAL TO ABSTRACT MACHINES

- **Abstract machine** or **abstract processor**
  - an entity that can execute the instructions of a specific source programming language
    - generalization of the notion of processor
  - can be realized on top of lower level processor, which can be physical or abstract
    - the lower level processor has its own programming language
- Different kinds of techniques / architectures to build abstract machines / processors on top of lower level processors
  - hardware
    - maximum efficiency, minimum flexibility
  - software
    - **interpreters, compilers, virtual machines**

ABSTRACT MACHINES

- Terms
  - PS = abstract processor / machines
  - LS = programming language to write programs on top of PS
  - PO = lower level processor
  - LO = programming language to write programs on top of PO



ABSTRACT MACHINE ARCHITECTURE: INTERPRETERS & COMPILERS

- **Interpreter**
  - a LO program that simulates PS on PO, interpreting LS
  - very flexible, but also very inefficient
- **Compiler**
  - the process PS is completely virtual, without an interpreter
  - LS is translated into a functionally equivalent program, written (compiled) in LO so as to run directly on PO
    - high efficiency + more resource consuming
    - less dynamism and portability

ABSTRACT MACHINE ARCHITECTURE: VIRTUAL MACHINES

- **Virtual Machine**
  - an abstract processor **PI** between PS and PO, executing programs written in a **LI** language
  - LS is translated into LI and executed onto an interpreter of LI - i.e. a simulator of the PI processor - running directly on PO
    - PI extends the functionalities of the physical machine PO so as to make it easier the translation of the source language
    - at the same time it makes it ieasier the portability of the language on different POs
- Examples
  - JVM, CLR, Erlang Virtual Machine
- Advantages
  - LI/PI is higher-level than LO/PO

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 and *message passing* (local memory) model
  - **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

CONCURRENT 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
- 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

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, C#
- Research landscape
  - actor-based models
  - coordination models and languages
  - agent-oriented approaches

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

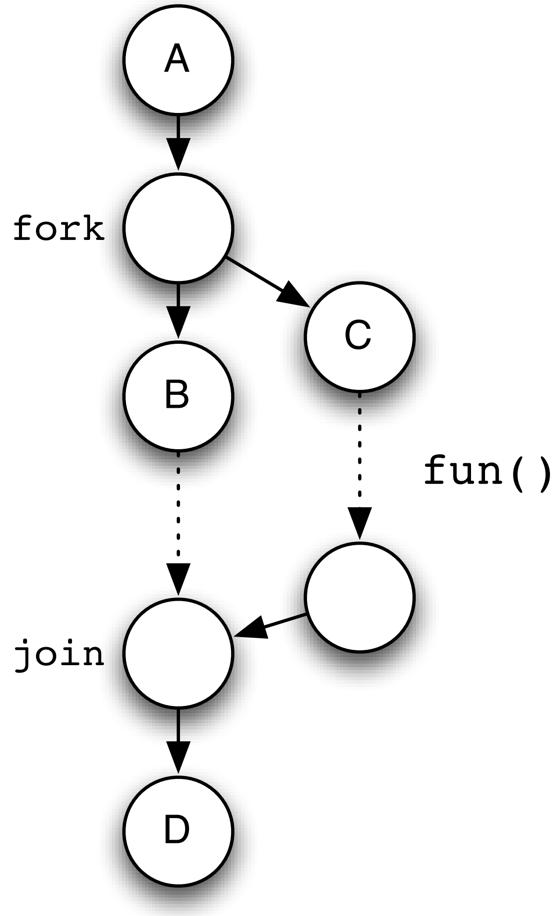
  > it results in a bifurcation of the program control flow
    - the new process (child) is executed asynchronously with respect to the generating process (parent) and existing processes
- **join** primitive
  – it detects when a process created by a fork has terminated and it synchronize current control 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

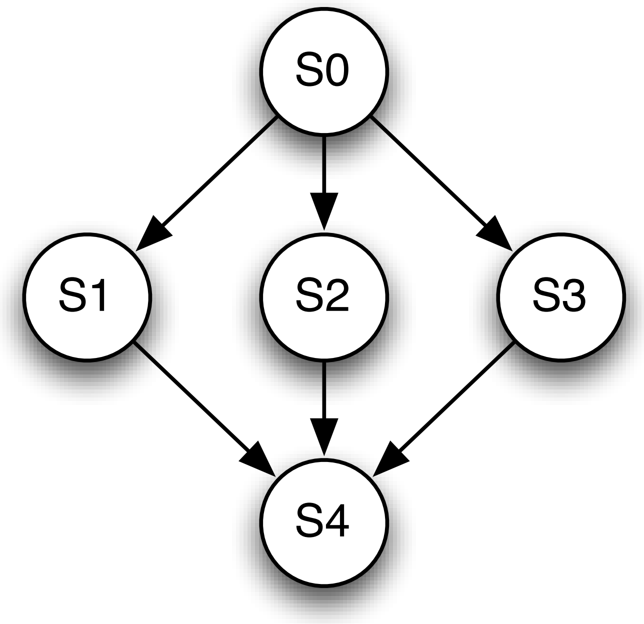
- an instruction Si can be as complex as a full program (it can include nested cobegin/coend)

- 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
- First concurrent languages
  - **Concurrent Pascal** (70ies)
- Modern languages
  - OCCAM (1980...OCCAM3 ~90ies)
  - SR (90ies)
  - **ADA** (~1980 up today with new versions - ADA95 with OO),
  - **Erlang** (end of 90ies up today)
    - used in particular by telecom industries

CONCURRENT PASCAL

- Designed by Per Brinch Hansen (~1975) as a language to write concurrent programs like operating systems and real-time monitoring systems, for shared-memory contexts

- Extension of Pascal language with first-class constructs to define **processes** and **monitors**
  - procedural/imperative language + process abstraction
  - monitor as data structure encapsulating and enforcing mutual exclusion and synchronization
- Process body specified as a procedure
  - declaration of the process type
  - dynamic creation of process instances

PROCESSES IN CONCURRENT PASCAL

```
type myProcess = process()
begin
  cycle
     <process actions>
  end
end;

var proc: myProcess;

begin
  init myProcess();
end;
```

ADA

- Introduced at the end of 70ies / 80ies as the reference language for (concurrent and sequential) programming inside US DoD
  - for programming in-the-large (being high-level, structural programming with OO elements, strong typing..)
  - for concurrent programming, to develop critical and real-time systems (such as aircraft controllers)
  - Ada was named after the Countess Ada Lovelace (1815-1852), who is often credited with inventing computer programming (actually based on the work on the “Analytical machine” of Charles Babbage)
- Processes in Ada are called **task**
  - task body specified as a procedure
    - declaration of the task type and definition of the task body
    - dynamic creation of task instances
  - task *entry* for task communication
    - operations served by tasks
- Ada is an international standard; the current version (known as Ada 2005) is defined by joint ISO/ANSI standard (ISO-8652:1995), combined with major Amendment ISO/IEC 8652:1995/Amd 1:2007.

TASK IN ADA

```
task <identifier> is
  <entry declarations>
end;

task body <identifier> is
  <local declarations>
begin
  <statements>
end <identifier>;
```

```
task Worker is
  entry DOTASK(T is Task)
end;

task body Worker is
 T: Task;
begin
 loop
  ...
  accept DOTASK(T);
  ...
 end loop;
end Worker;
```

COUNTER EXAMPLE IN ADA

```
with Ada.Text_IO;
use Ada.Text_IO;
procedure Count is
  N: Integer := 0;
  pragma Volatile(N)
  task type Count_Task;
  task body Count_Task is
    Temp: Integer;
  begin
    for I in 1..10 loop
      Temp := N;
      N := Temp + 1;
    end loop
  end Count_Task;
begin
  declare
    P,Q: Count_Task;
    begin
      null;
    end;
    Put_Line(“The value of N is “ & Integer’Image(N));
end Count;
```

ERLANG

- Functional language providing a native support for concurrent programming based on **processes** and process asynchronous communication through message passing
  - developed in Ericsson since 1987 for building telecom applications
  - along with ADA, it can be considered the most used and robust concurrent programming language adopted by the industry
- BEAM concurrent virtual machine
  - BEAM stands for Bogdan/Björn's Erlang Abstract Machine
  - completely abstract / virtual notion of process
    - not related to OS process or OS threads
  - extremely efficient process management
    - hundred of thousands processes can be created on a single host

ERLANG AS A FUNCTIONAL LANGUAGE

- An Erlang program describes a series of *functions*
  - operators as special kind of functions
  - each function uses *pattern matching* to determine which function to execute
    - variables start with upper-case
  - no global variables

```
fact(0) -> 1;
fact(N) -> N * fact(N - 1).
```

```
fib(1) -> 1;
fib(2) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```

- Modules are used to package functions

```
-module(math).
-export([fact/1]).
-export([fib/1]).

fact(0) -> 1;
fact(N) -> N * fact(N - 1);

fib(1) -> 1;
fib(2) -> 1;
fib(N) -> fib(N-1) + fib(N-2);
```

ERLANG AS A FUNCTIONAL LANGUAGE

- Calling functions

```
X = math:fact(100).
```

- Compiling and executing programs (..is calling functions..)

```
Erlang (BEAM) emulator version 5.6.4 [source] [smp:2] [async-
threads:0] [kernel-poll:false]

Eshell V5.6.4  (abort with ^G)
1> c(math).
{ok,math}
2> Res = math:fact(15).
1307674368000
```

ERLANG AS A FUNCTIONAL LANGUAGE

- *Tuple* and *list* as primitive structured data structures
  - besides atomic data structures (atoms), such as symbols, constants, numbers and strings
- Tuples
  - record-like ordered structure with a fixed number of elements
    - e.g. Point = { point, 10, 20 }
  - support for pattern matching
    - e.g. {point, X, Y } = Point
      - X is bound to 10 and Y to 20
- Lists
  - to store a variable number of data items
    - e.g. ThingsToBuy = [{ apples, 10 }, { pears, 6 }, { smirnoff, 2 }]
  - head and tail notation: [H | T]
    - e.g. ThingsToBuy = [{apples, X} | T]
      - X is bound to 10 and T to [{ pears, 6 }, { smirnoff, 2 }]

ERLANG FOR CONCURRENT PROGRAMMING

- A process is a computational activity whose computational behaviour is given by some specific function
- The **spawn** primitive to launch a process, getting its PID
  - specifying the function module, function name and parameters

```
Pid = spawn(math, fact,[999]).
```

- Processes can communicate solely through message passing
  - **!** operator to send a message

```
Pid ! Message
```

  - **receive** construct to receive a message, specifying a pattern

```
receive
 Pattern1 [when Guard1 ] -> Expression1;
 Pattern2 [when Guard2 ] -> Expression2;
 ...
end
```

A SIMPLE EXAMPLE

```
-module(area_server0).
-export([loop/0]).

loop() ->
  receive
    { rectangle, Width, Ht} ->
         io:format("Area of rectangle is ~p~n",[Width*Ht]),
         loop();
    { circle, R } ->
         io:format("Area of rectangle is ~p~n",[3.14159*R*R]),
         loop();
    Other ->
         io:format("I don't know what the area of a ~p is ~n",[Other]),
         loop()
  end.
```

```
20> c(area_server0).
{ok,area_server0}
21> Pid = spawn(fun area_server0:loop/0).
<0.79.0>
22> Pid ! {rectangle, 3, 4}.
Area of rectangle is 12
{rectangle,3,4}
23> Pid ! {circle,1}.
Area of rectangle is 3.14159
{circle,1}
24> Pid ! {triangle,1,4}.
I don't know what the area of a triangle is
{triangle,1,4
```

HOW DOES IT TAKE TO CREATE A PROCESS (...WHEN PROCESSES ARE VIRTUAL...)

```
-module(processes).
-export([max/1]).

%% max(N)
%%  Create N processes then destroy them
%%  See how much time this takes

max(N) ->
  Max = erlang:system_info(process_limit),
  io:format("Maximum allowed processes:~p~n",[Max]),
  statistics(runtime),
  statistics(wall_clock),
  L = for(1, N, fun() -> spawn(fun() -> wait() end) end),
  {_, Time1} = statistics(runtime),
  {_, Time2} = statistics(wall_clock),
  lists:foreach(fun(Pid) -> Pid ! die end, L),
  U1 = Time1 * 1000 / N,
  U2 = Time2 * 1000 / N,
  io:format("Process spawn time=~p (~p) microseconds~n",[U1, U2]).

wait() ->
  receive
    die -> void
  end.

for(N, N, F) -> [F()];
for(I, N, F) -> [F()|for(I+1, N, F)].
```

```
1> processes:max(20000).
Maximum allowed processes:32768
Process spawn time=5.5 (9.4) microseconds
ok
```

COUNTER EXAMPLE IN ERLANG

```
-module(counter).
-export([start/0]).

start() -> loop(0).
loop(Sum) ->
  receive
    {inc} ->
      loop(Sum+1);
    {getValue, Pid} ->
      Pid ! {count_value, Sum},
      loop(Sum)
  end.
```

```
-module(counter_user).
-export([start/2]).

start(Counter,N) -> loop(Counter,0,N).

loop(_,N,N).
loop(Counter,I,N) ->
  Counter ! {inc},
  loop(Counter,I+1,N).
```

```
1> Pid = spawn(counter,start,[]),
   spawn(counter_user,start,[Pid,100)), spawn(counter_user,start,[Pid,100]).
```

- Note the “everything is process” philosophy
  - no shared memory, then a counter is a process...

PROCESSES 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 (=> local variables, activation records for method invocation)
    - 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
    - what kind of specialization is this??
- 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

- Introducing *higher-level* first-class abstractions for organizing a large-scale concurrent software systems (from 80ies...)
  - **actor**-based models
  - **active-objects**
  - coordination models and languages
  - **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)
- Implemented as a pattern on top of existing languages
  - many Java-based frameworks

ACTORS IN ACT3

```
(define (Factorial( ))
  (Is-Communication (a doit (with customer ≡m)
                              (with number ≡n)) do
      (become Factorial)
      (if (NOT (= n 0))
        (then (send m 1))
        (else (let (x = (new FactCust (with customer m)
                                  (with number n)))
                  (send Factorial (a do (with customer x)
                                  (with number n-1))))))))
```

```
(define (Account (with Balance ≡b))
   (Is-Request (a Balance) do (reply b))
   (Is-Request (a Deposit (with Amount ≡a)) do
       (become (Account (with Balance (+ b a))))
       (reply (a Deposit-Receipt (with Amount a))))
   (Is-Request (a Withdrawal (with Amount ≡a)) do
       (if (> a b)
        (then do (complain (an Overdraft)))
        (else do
          (become (Account (with Balance (- b a))))
          (reply (a Withdrawal-Receipt (with Amount a)))))))
```

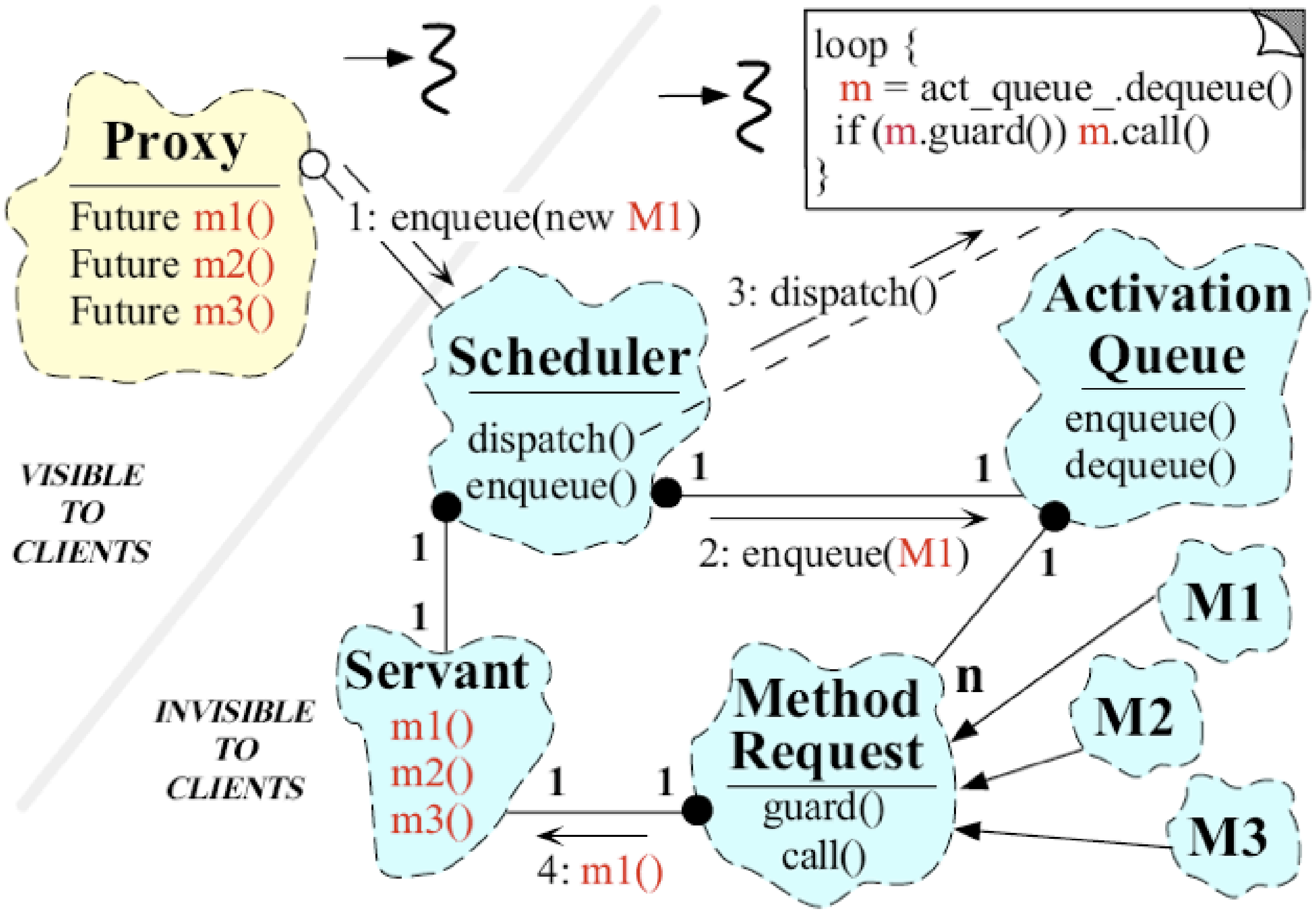
ACTOR BASIC PRIMITIVES

- Only three primitives (actions) to compose 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 behaviour
    - it is to concurrent programming what procedure abstraction is to sequential programming
  - **become**
    - specify a new behaviour (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

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]
  - Active Objects as a pattern [LAV-96]
    - can be implemented on top of sequential OO languages with a basic thread support

ACTIVE-OBJECT COMPONENTS



AGENT-ORIENTED COMPUTING

- The notion of agent (and multi-agent system) has been introduced in several research contexts, with different acceptations
  - (distributed) artificial intelligence, complex systems modelling and simulation, mobile technology, software engineering...
  - *agent-oriented computing*
    - introducing agent-orientation as a general-purpose programming paradigm for developing software systems
- Basic abstractions
  - **agents**
    - autonomous entities designed to pro-actively do some kind of work, encapsulating the logic and control of their activities
      - goal-oriented / task-oriented behaviour
    - interacting with their *computational environment*
      - actions and perceptions
    - interacting with other agents through some ACL
      - asynchronous message passing
  - **agents environment**
    - target of agent actions and source of agent perceptions
    - what can be used by agents to achieve their objective

AMONG RECENT RESEARCH WORKS...

- **Polyphonic C#** [BEN-04]
  - C# extension with new asynchronous concurrency abstractions, based on the join calculus
    - synchronous and asynchronous methods
    - *chord* basic synchronization mechanism
  - applicable both to multithreaded applications running on a single machine and to the orchestration of asynchronous, event-based applications communicating over a wide area network
- **Map-reduce** framework [DEA-04]
  - software framework to support parallel computations over large (multiple petabyte) data sets on clusters of computers
- **simpA** agent-oriented programming framework [RV-07]
  - a framework on top of Java providing agent-oriented concepts to program concurrent applications

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
- **[DEA-04]**
  - MapReduce: Simplified Data Processing on Large Clusters. Jeffrey Dean and Sanjay Ghemawat. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
- **[BEN-04]**
  - Nick Benton, Luca Cardelli and Cédric Fournet. Modern Concurrency Abstractions for C#. ACM Transactions on Programming Languages and Systems (TOPLAS) 26(5) pp.269-804. September 2004.
- **[RV-07]**
  - A. Ricci and M. Viroli. simpA: An agent-oriented approach for prototyping concurrent applications on top of Java. 5th International Conference, Principles and Practice of Programming in Java (PPPJ 2007), pages 185–194, PPPJ 2007