Computação em Larga Escala

Concurrency 1

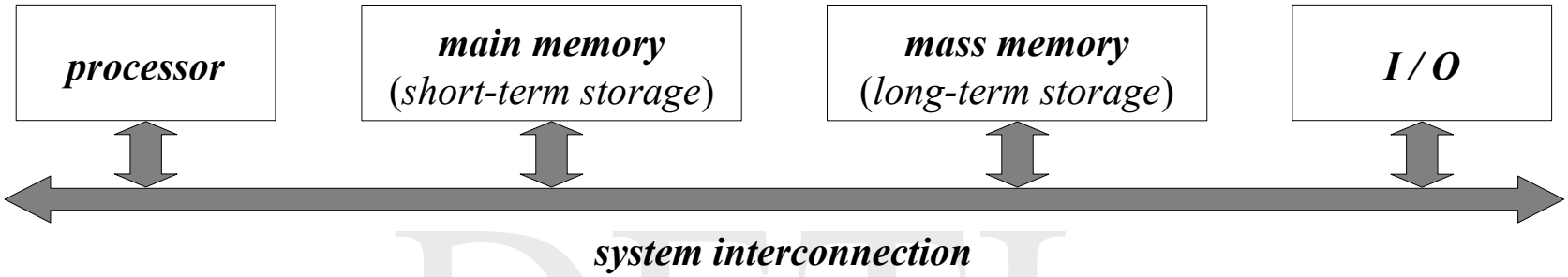
António Rui Borges

Summary

- *Computer architecture overview*
- *Program vs. Process*
  - *Characterization of a multiprogrammed environment*
- *Processes vs. Threads*
  - *Characterization of a multithreaded environment*
- *Suggested reading*

Computer architecture overview - 1

At top level, the structure of a computer system may be perceived as



- ***processor*** (or ***central processing unit*** – CPU) – controls the operation of the computer and performs its data processing functions
- ***main memory*** – stores data during processing; it has a *volatile* nature
- ***mass memory*** – stores data in-between processing runs and allows that large amounts of data be retrieved and eventually updated during processing; it has a *non-volatile* nature and is perceived as a special I/O device
- ***I/O*** – moves data between the computer system and its external environment
- ***system interconnection*** – ensures that data communication takes place among the other components; it is usually implemented as a *bus*.

Computer architecture overview - 2

In order to have the computer system to carry out a specific task, one has to supply a group of instructions that taken together constitute the *program* to be executed. The key idea here is how to represent instructions?

Suppose they could be represented in a form suitable for being stored in the main memory alongside the data, thus constituting a *special* kind of data in some abstract sense. Then, a computer system could get its instructions by reading them from the main memory and a program could be set or altered by setting the values of that portion of the memory.

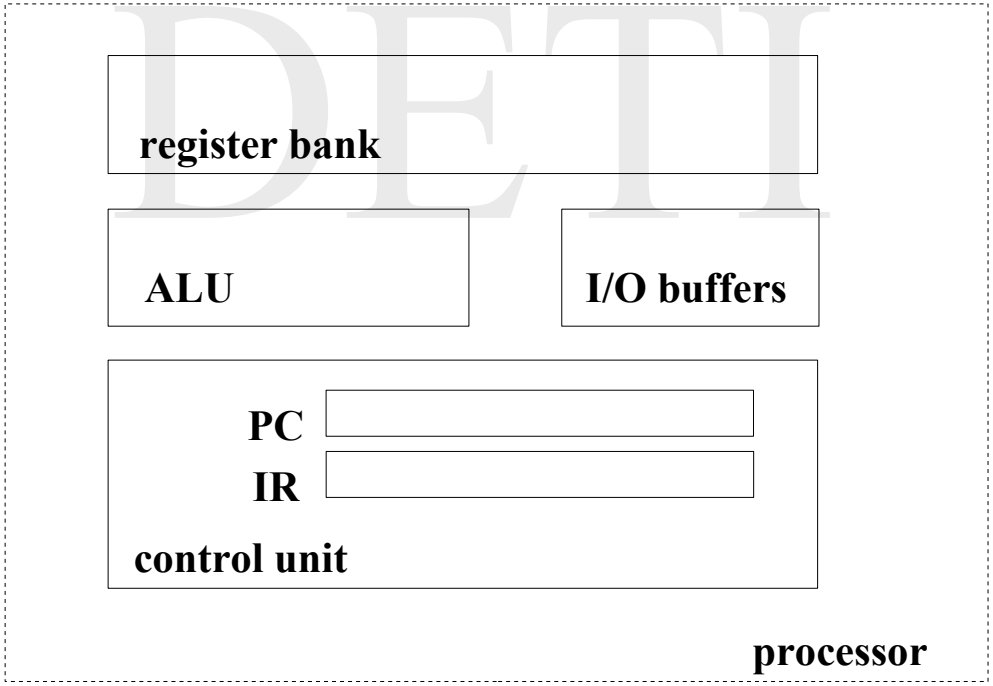
This idea, developed independently by John von Neumann and Alan Turing, has come to be known as the *stored-program concept* and has been universally adopted by computer designers. This is the reason why computer systems are sometimes called *von Neumann machines*.

Computer architecture overview - 3

In this sense, although a computer system is a very complex digital system, it may be perceived as a system which toggles continuously between two basic internal states: *instruction fetch*, where the processor accesses memory to get the next instruction, and *instruction execution*, where the processor decodes the just retrieved instruction and performs the corresponding operation.

Computer architecture overview - 4

In a simplified way, the processor itself may be seen of consisting of a *control unit*, which takes mainly care of the *instruction fetch* and the *instruction decoding* phases; of a *arithmetic / logic unit*, which performs the prescribed operations; of a *register bank*, which stores temporary data; and of *I/O buffers*, which enable communication with the other components of the computer system.



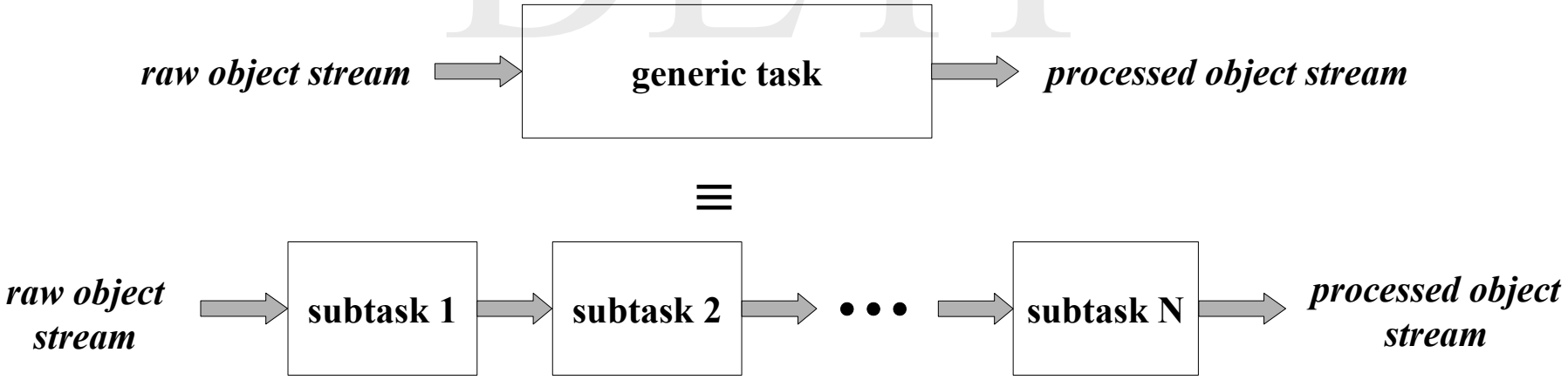
Computer architecture overview - 5

The *instruction set* of any processor always comprises instructions of the type

- *data movement* – transfer of data between some register of the register bank and main memory or some I/O controller
- *arithmetic* / *logic* – arithmetic instructions (add, subtract, multiply, divide), either in fixed or floating point format, logic instructions (not, and, or, x-or) and shifting / rotating register contents
- *branching* – modifying the strict sequentiality of instruction execution, either unconditionally or dependent on some condition
- *subroutine calling* – execution of subprograms (autonomous code segments within the whole group of supplied instructions), either unconditionally or dependent on some condition.

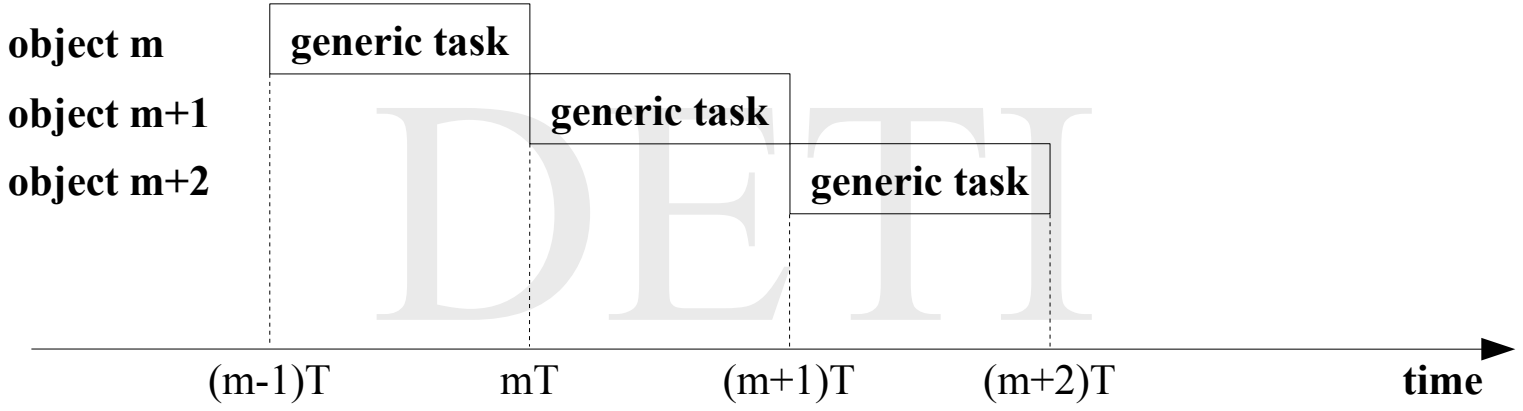
Computer architecture overview - 6

Pipelining is an implementation technique where the execution of a generic task on the objects of a stream is converted into a tandem of independent subtasks which operate simultaneously on successive objects of the stream. Each of the individual subtasks, called *pipe stages* or *segments*, is performed in sequence and represents a definite fraction of the whole task. Their combined ordered execution is equivalent to the execution of the original task on every object of the stream.



Computer architecture overview - 7

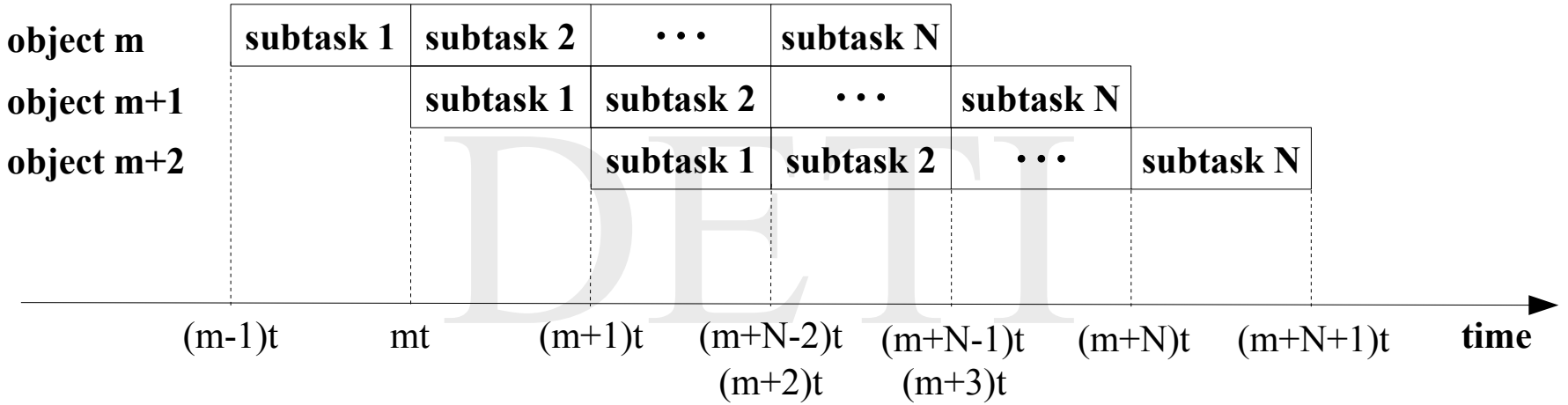
non-pipelined version



$$\text{throughput} = \frac{1}{T}$$

$$\text{execution time for a M object stream} = M \cdot T$$

N-stage pipelined version



t_n , with $n = 1,2, \cdots, N$ – execution time for subtask n

throughput $= \frac{1}{t}$, where $t = max(t_1, t_2, \cdots, t_N)$

execution time for a M object stream $= (N - 1 + M) \cdot t$

Computer architecture overview - 9

The *speed up* obtained from the execution of a N-stage pipeline implementation over the non-pipelined execution of the same task is expressed by

$$\text{speed up}_{\text{N-stage pipeline}} = \frac{\text{execution time of the non-pipelined implementation}}{\text{execution time of the N-stage pipelined implementation}} =$$

$$= \frac{M \cdot T}{(N - 1 + M) \cdot t} = \frac{M \cdot N \cdot t^*}{(N - 1 + M) \cdot t} \approx N \cdot \frac{t^*}{t} \quad ,$$

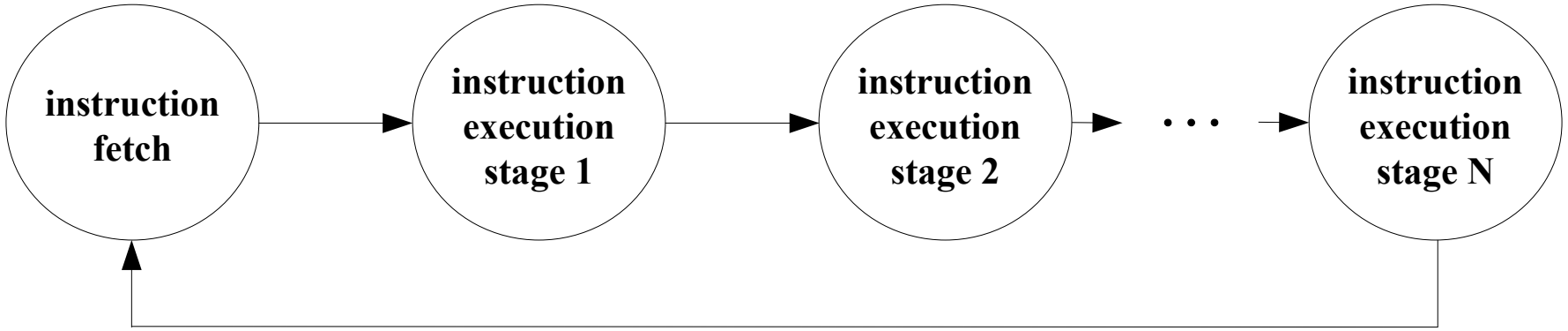
where $T = N \cdot t^*$ and $N << M$.

Ideally, if the pipe stages are almost perfectly balanced and the overhead involved in pipelining is negligible, then the ratio t^*/t approaches unity and the speed up is approximately equal to the number stages used on the partition of the original task into subtasks.

Computer architecture overview - 10

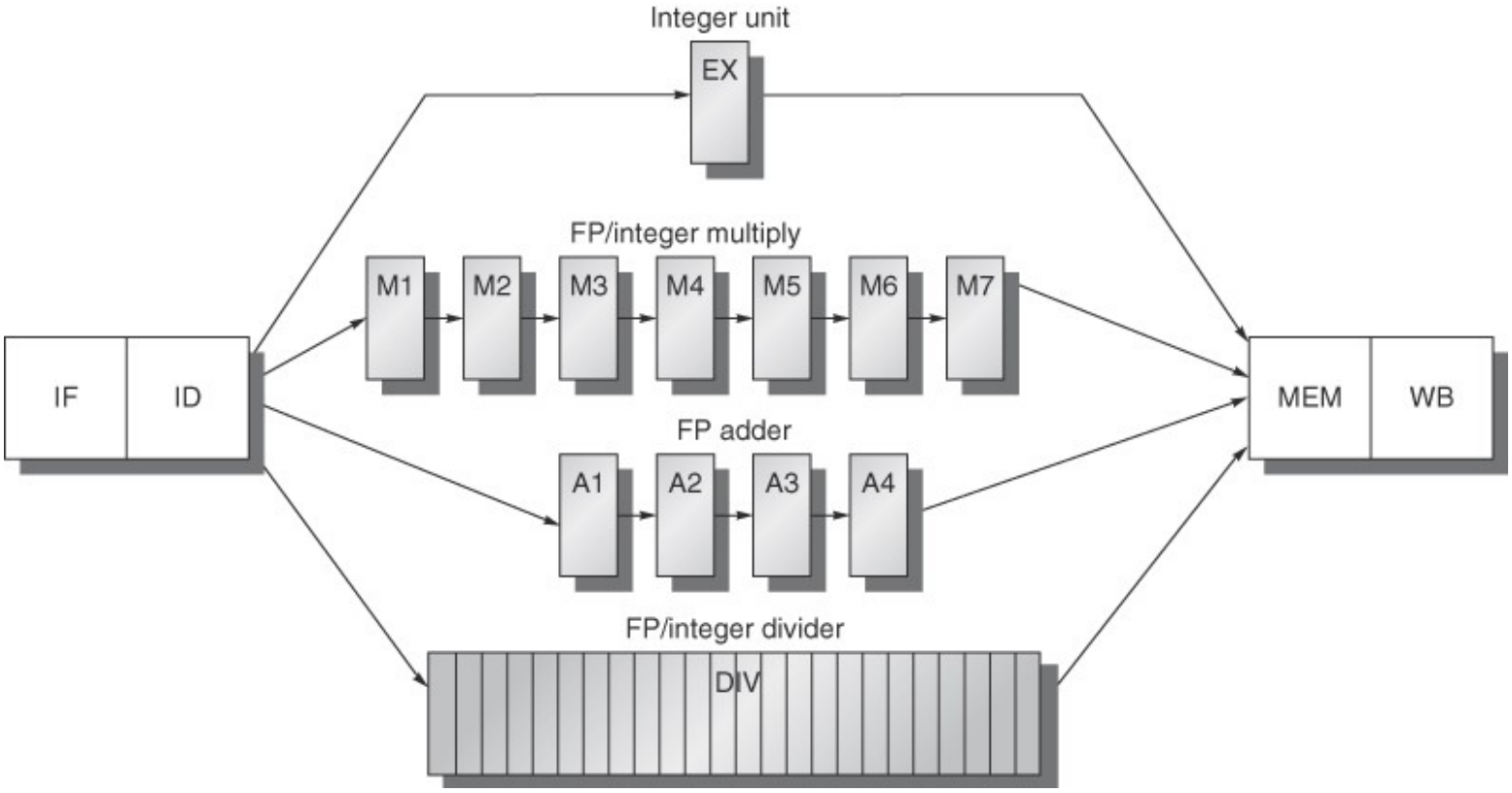
All processors since 1985 have adopted pipelining as a means for overlapping the execution of instructions and for improving performance. This potential overlap of instructions during their own execution is called *instruction-level parallelism* (ILP) because the instructions are in some sense processed in parallel through the activity decomposition that takes place.

A common approach for doing this is by decomposing the *instruction execution* phase of the processor cycle into several stages.



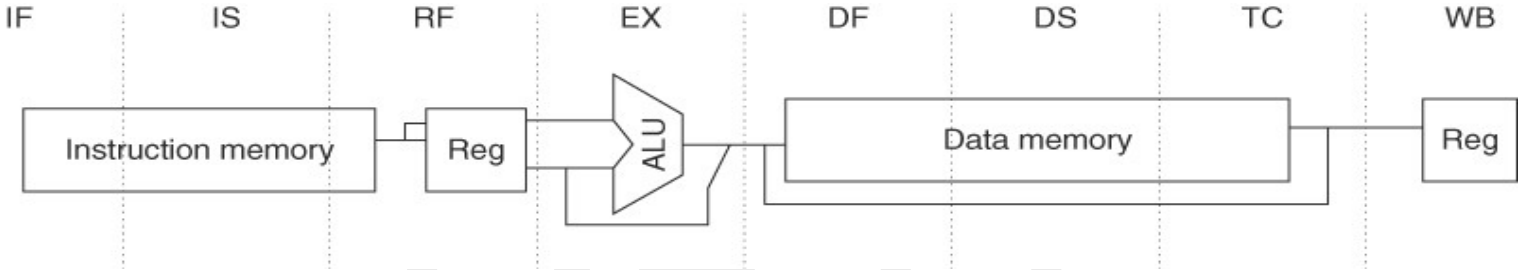
The classical 5-stage pipeline supporting multiple outstanding FP operations

Source: Computer Architecture: A Quantitative Approach



R4000 eight-stage pipeline organization

Source: Computer Architecture: A Quantitative Approach



IF – first half of instruction fetch
IS – second half of instruction fetch
RF – instruction cache hit detection + instruction decode and register fetch + hazard checking
EX – execution, including ALU operation, effective address and branch target computation and condition evaluation
DF – first half of data fetch
DS – second half of data fetch
TC – data cache hit detection
WB – register write back

Computer architecture overview - 13

To further improve performance, new techniques of *instruction-level parallelism* were introduced in the design of present day processors: *out-of-the order* instruction execution, *speculative* processing and *multiple issue*.

Primary approaches in use for multiple issue processors

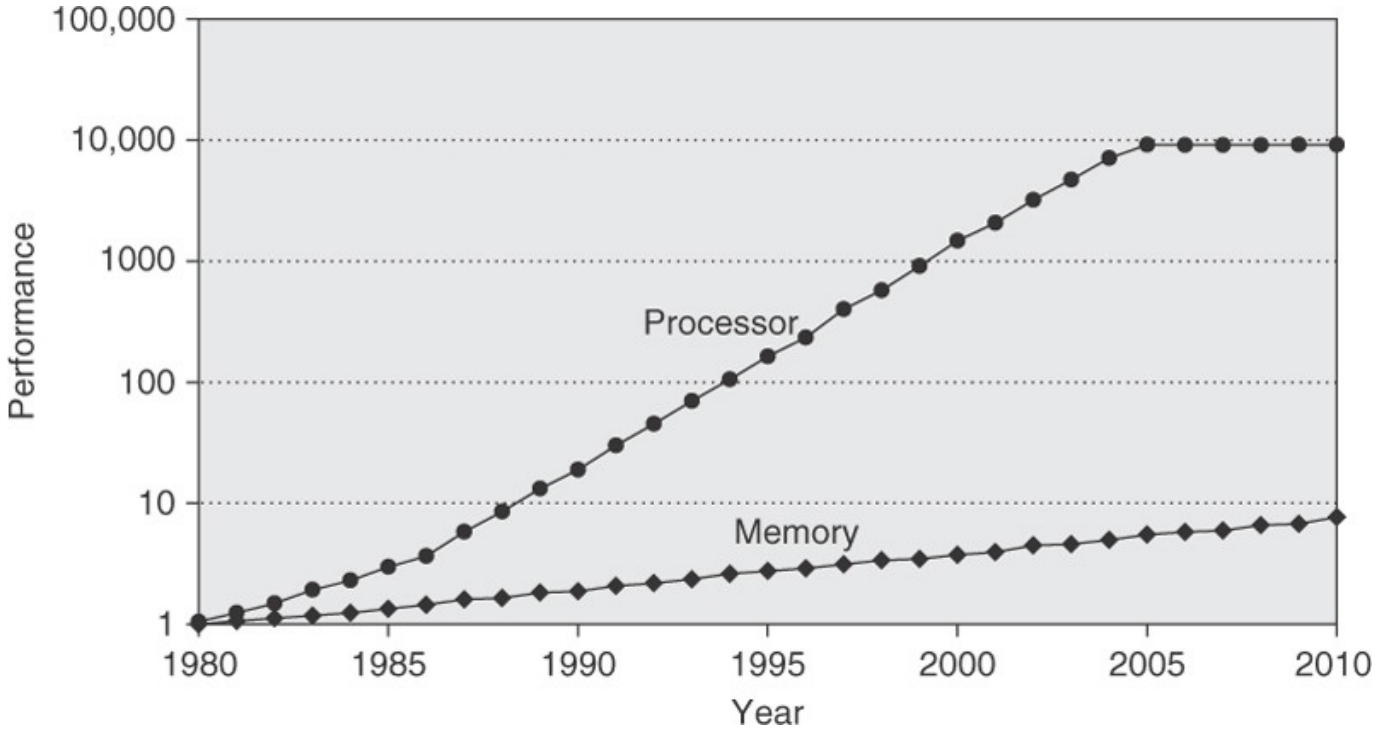
Source: Computer Architecture: A Quantitative Approach

Common name	*Issue structure*	*Hazard detection*	*Scheduling*	*Distinguishing characteristic*	*Examples*
superscalar (static)	dynamic	hardware	static	in-order execution	mostly in embedded environments: MIPS and ARM (including ARM Cortex-A8)
superscalar (dynamic)	dynamic	hardware	dynamic	some out-of-order execution, but no speculation	none at the present time
superscalar (speculative)	dynamic	hardware	dynamic with speculation	out-of-order execution with speculation	Intel Core i3, i5, i7, AMD Phenom and IBM POWER 7
VLIW / LIW	static	primarily sofware	static	all hazards determined and indicated by the compiler (often implicitly)	mostly in signal processing: TI C6x
EPIC	primarily static	primarily sofware	mostly static	all hazards determined and indicated by the compiler (explicitly)	Itanium

Computer architecture overview - 14

Over time performance variation of single processor vs. memory

Source: Computer Architecture: A Quantitative Approach



processor curve – average number of memory requests per second
memory curve – inverse of DRAM access latency

Computer architecture overview - 15

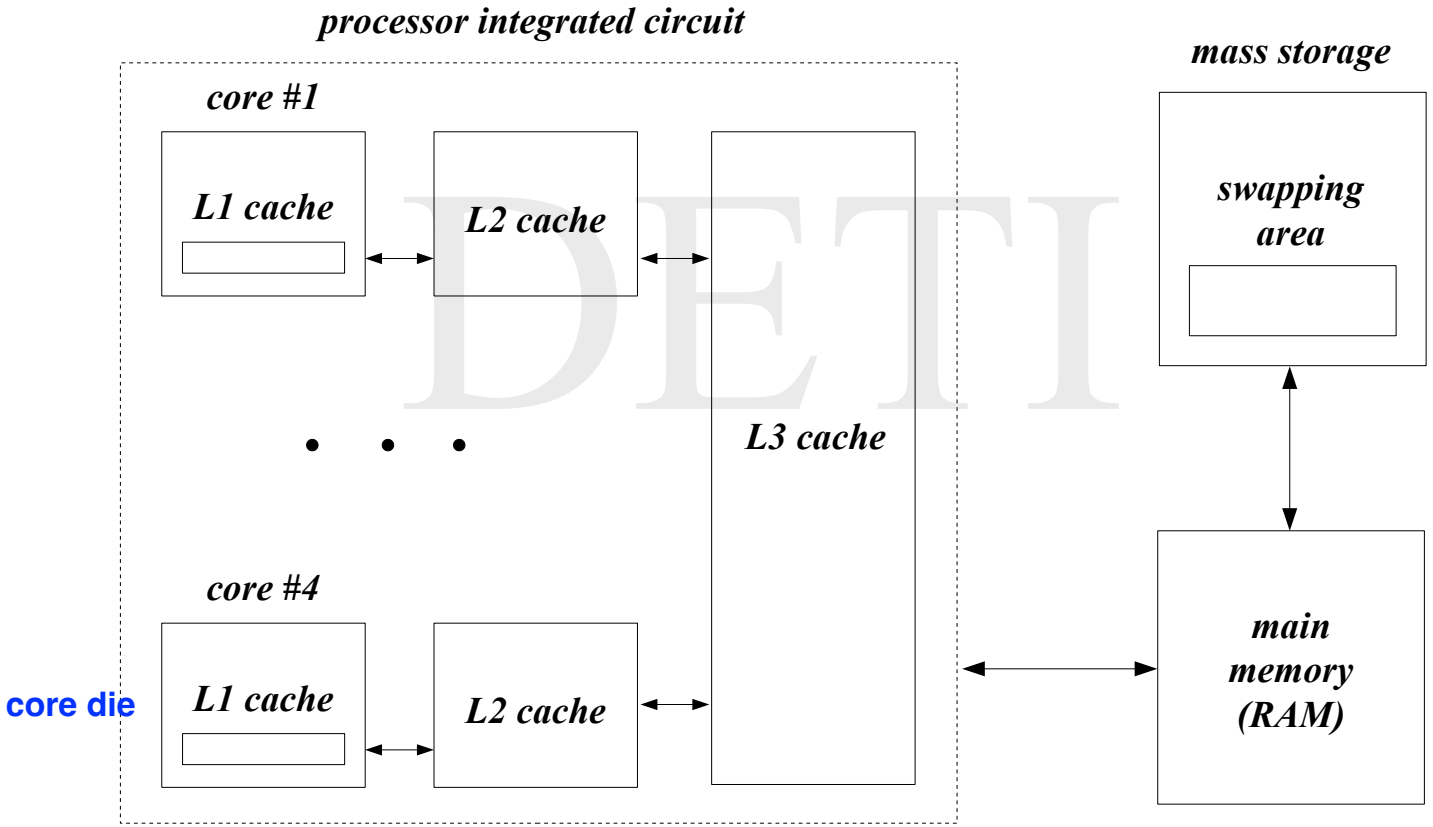
Technical specifications of a typical memory hierarchy

Source: adapted from Computer Architecture: A Quantitative Approach

Name	*Typical size*	*Implementation technology*	*Access time (ns)*	*Bandwidth (MB/s)*	*Managed by*	*Backed by*
register bank	less 1KB	multiport custom design – CMOS	0,15 – 0,30	$10^5 – 10^6$	compiler	cache
cache	32 KB – 8 MB	on/off-chip CMOS SRAM	0,5 – 15	$10^4 – 4 \times 10^4$	hardware	main memory
main memory	less 512 KB	CMOS DRAM	30 – 200	$5 \times 10^3 – 2 \times 10^4$	operating system	swapping area
swapping area	greater 1 TB	semiconductor / magnetic	3×10^4 - 5×10^6	50 – 500	operating system	long term storage

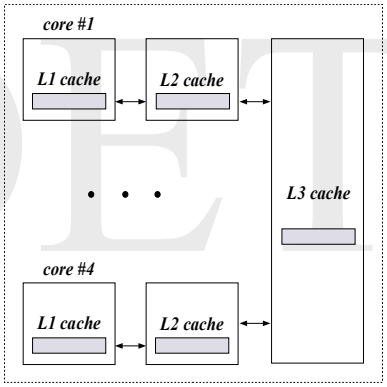
Computer architecture overview - 16

Typical memory hierarchy for a 4-core processor system



Computer architecture overview - 17

For a multicore processor, where multiple simultaneous processes may be competing for access to shared data, protected or not by critical regions, a further problem arises which has to do with *cache coherence*. In such a situation, copies of the same memory block may be stored in lines of level 1 or level 2 caches associated with different processors.



In order to ensure that all processors always see the same data, a *write-through* policy should be implemented for level 1 and level 2 caches and when a *write* takes place, all copies at these levels should be made *stale* so that a transfer from level 3 cache is carried out if a subsequent *read* arises.

Computer architecture overview - 18

Version 1

```
#define N    20000
int x[N],
    y[N];        // elements of the array y are initialized to zero
int a[N][N];
int i, j;

for (j = 0; j < N; j++)
  for (i = 0; i < N; i++)
    y[i] += a[i][j] * x[j];
```


Version 2

```
#define N    20000
int x[N],
    y[N];        // elements of the array y are initialized to zero
int a[N][N];
int i, j;

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += a[i][j] * x[j];
```

Computer architecture overview - 19

Compilation without optimization

```
[ruib@ruib-laptop desempenho]$ gcc -Wall -O0 -o testCompiler1 testCompiler1.c
[ruib@ruib-laptop desempenho]$ ./testCompiler1
Elapsed time = 4.270206 s
[ruib@ruib-laptop desempenho]$ gcc -Wall -O0 -o testCompiler2 testCompiler2.c
[ruib@ruib-laptop desempenho]$ ./testCompiler2
Elapsed time = 0.788911 s
```


Compilation with optimization

```
[ruib@ruib-laptop desempenho]$ gcc -Wall -O3 -o testCompiler1 testCompiler1.c
[ruib@ruib-laptop desempenho]$ ./testCompiler1
Elapsed time = 2.141684 s
[ruib@ruib-laptop desempenho]$ gcc -Wall -O3 -o testCompiler2 testCompiler2.c
[ruib@ruib-laptop desempenho]$ ./testCompiler2
Elapsed time = 0.137255 s
```

Program vs. Process

Generally speaking, a *program* can be defined as a sequence of instructions which describes the execution of a certain task in a computer. However, for this task to be *in fact* carried out, the corresponding program must be executed.

A program execution is called a *process*.

Representing an activity that is taking place, a *process* is characterized by

- the *addressing space* – the code and the current value of all its associated variables
- the *processor context* – the current value of all the processor internal registers
- the *I/O context* – all the data that are being transferred to the input and from the output devices
- the *state* of the execution.

Process state diagram - 1

A process may be in different situations, called *states*, along its existence.

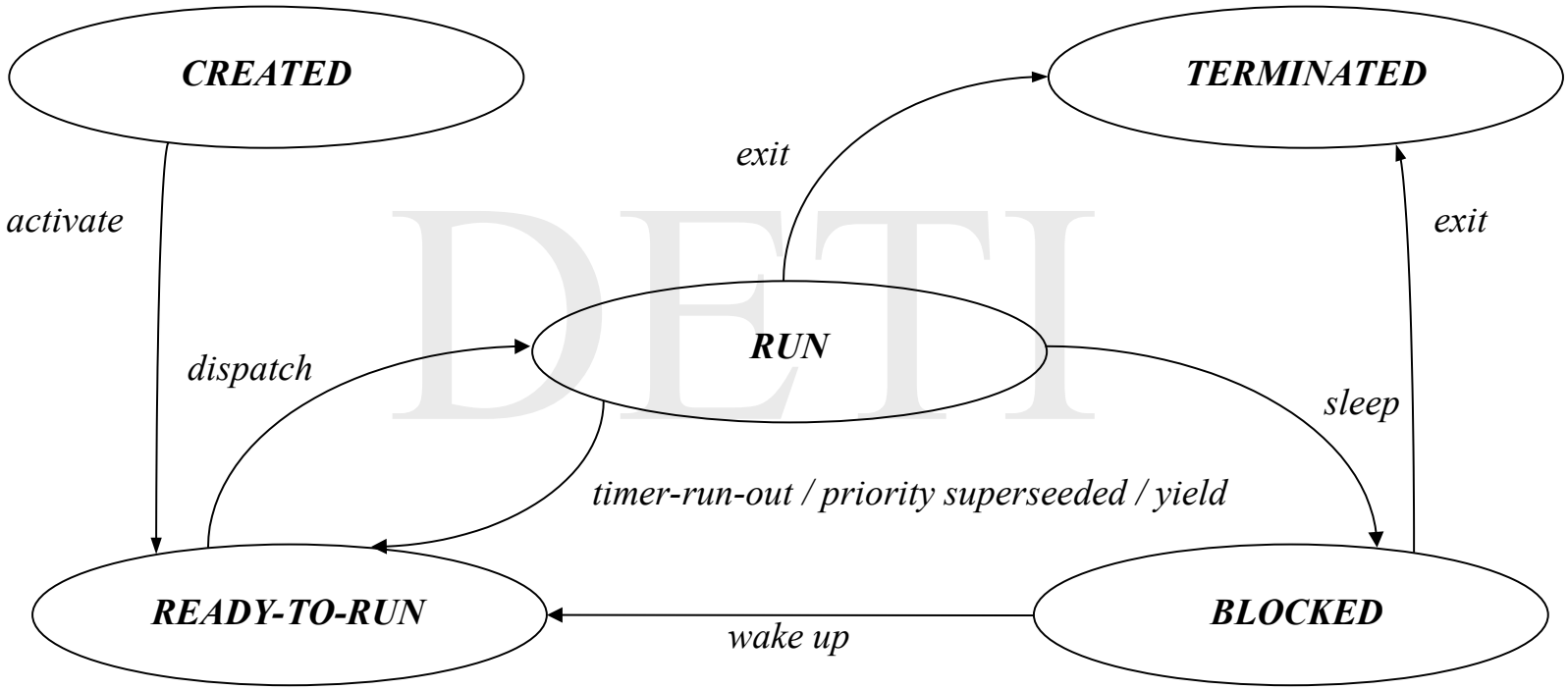
The most important states are the following

- *run* – when it holds the processor and is, therefore, in execution
- *ready-to-run* – when it waits for the assignment of the processor to start or resume execution
- *blocked* – when it is prevented to proceed until an external event occurs (access to a resource, completion of an input/output operation, etc).

State transitions are usually triggered by an external source, the operating system, but may be triggered by the process itself in some instances.

The part of the operating system which deals with [process] state transitions is called the *scheduler* (*processor scheduler*, in this case), and forms an integral portion of its nucleus, the *kernel*, which is responsible for exception handling and for scheduling the assignment of the processor and all other system resources to the processes.

Process state diagram - 2



Process state diagram - 3

activate – a process is created and placed in the *ready-to-run queue* waiting to be scheduled for execution

dispatch – one of the processes of the *ready-to-run queue* is selected by the scheduler for execution

timer-run-out – the process in execution exhausted the slot of processor time which was assigned to it (*preemptive scheduling*)

priority superseded – the process in execution looses the processor because the *ready-to-run queue* now contains a process of higher priority that requires the processor (*preemptive scheduling*)

yield – the process releases voluntarily the processor to allow other processes to be executed (*non-preemptive scheduling*)

sleep – the process is prevented to proceed and must wait for an external event to occur

wake up – the external event the process was waiting for has occurred

exit – the process has terminated its execution and waits for the resources that were assigned to it to be released

Processes vs. Threads - 1

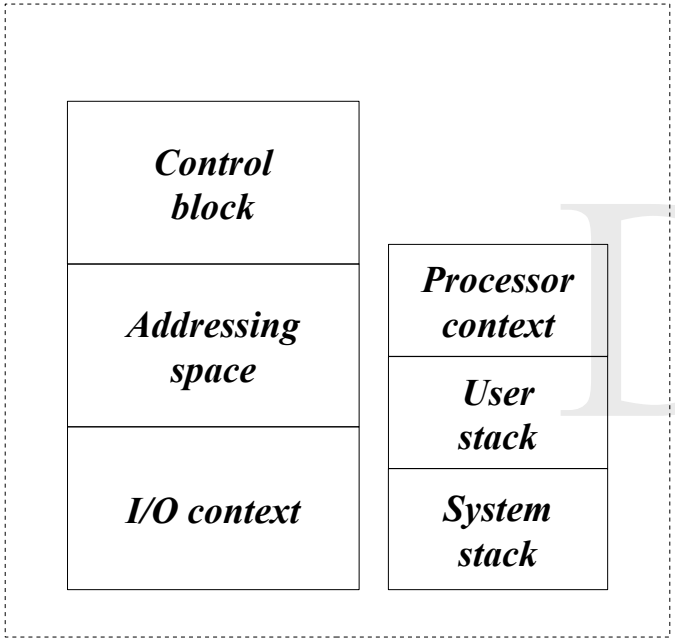
The concept of *process* embodies the following properties

- *resource ownership* – a private addressing space and a private set of communication channels with the input and output devices
- *thread of execution* – a *program counter* which points to the instruction that must be executed next, a set of *internal registers* which contain the current values of the variables being processed and a *stack* which keeps the history of execution (a *frame* for each routine that was called and has not yet returned).

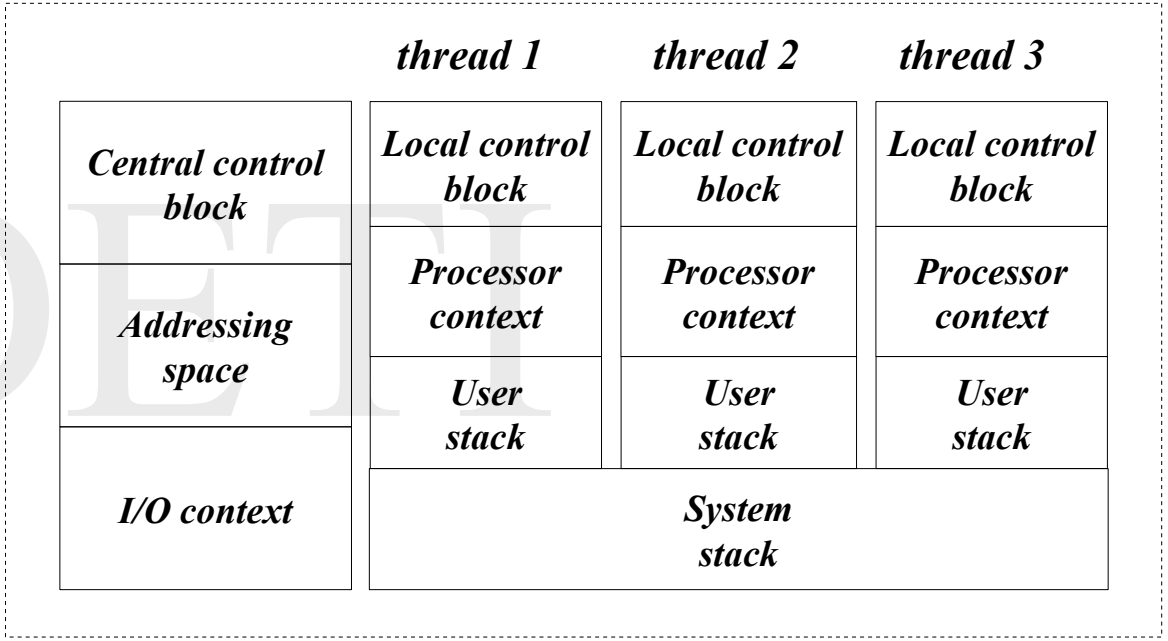
These properties, although taken together in a *process*, can be treated separately by the execution environment. When this happens, *processes* are envisaged as grouping a set of resources and *threads*, also known as *light weight processes*, represent runnable independent entities within the context of a single process.

Multithreading means, then, an environment where it is possible to create multiple *threads of execution* within the same process.

Processes vs. Threads - 2



Single threading

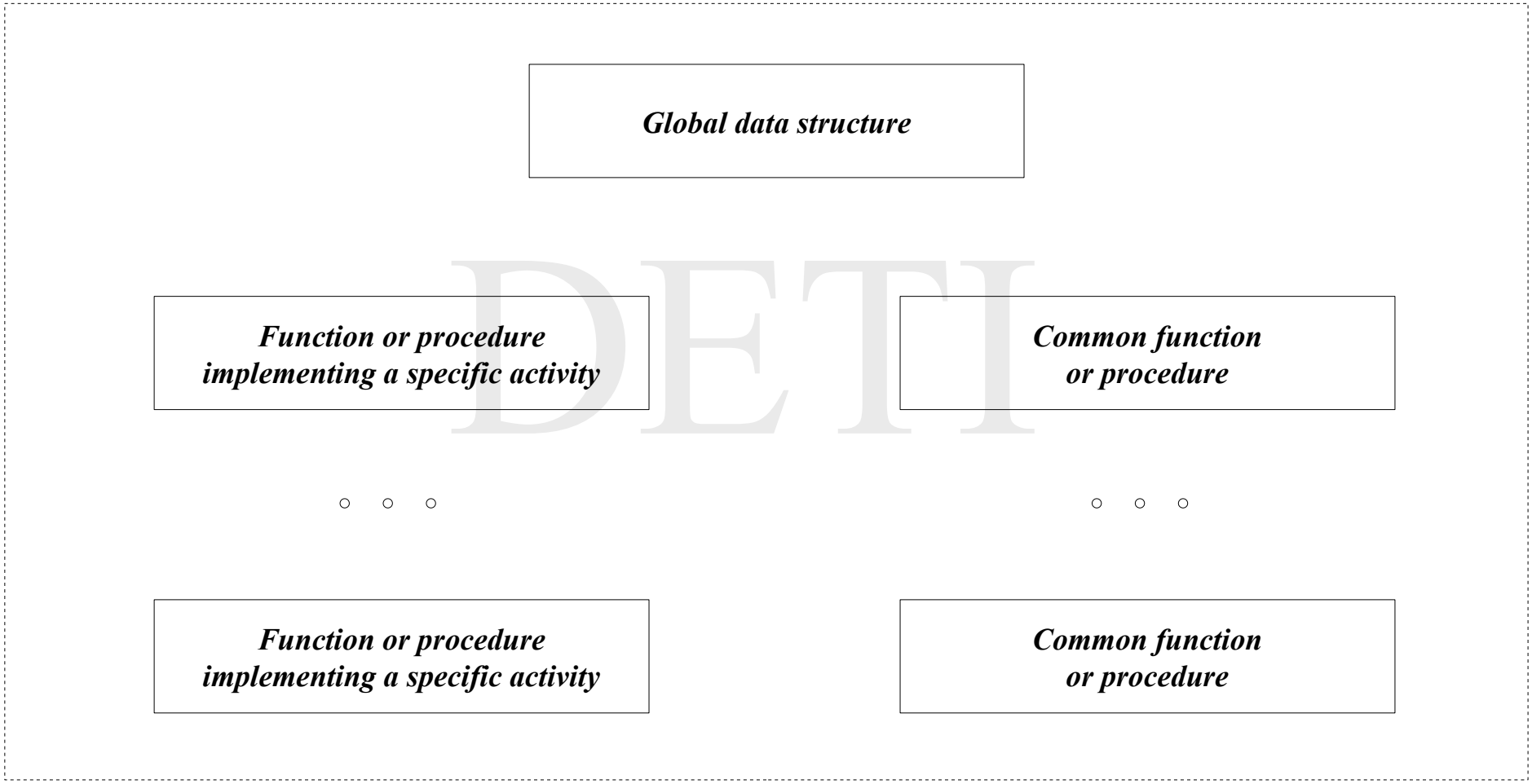


Multithreading

Advantages of a multithreaded environment

- *greater simplicity in solution decomposition and greater modularity in its implementation* – programs which involve multiple activities and service multiple requests are easier to design and implement in a concurrent perspective than in a pure sequential one
- *better management of computer system resources* – sharing the addressing space and the I/O context among the *threads* of an application results in decreasing the complexity of managing main memory occupation and access to the input / output devices
- *greater efficiency and speed of execution* – a solution decomposition based on *threads*, by opposition to one based on processes, requires less resources of the operating system, enabling that operations like process creation and termination and a context commutation to become less heavy and, thus, more efficient; furthermore, it becomes possible in symmetric multiprocessing to schedule for parallel execution multiple *threads* belonging to the same application, thus increasing the speed of execution

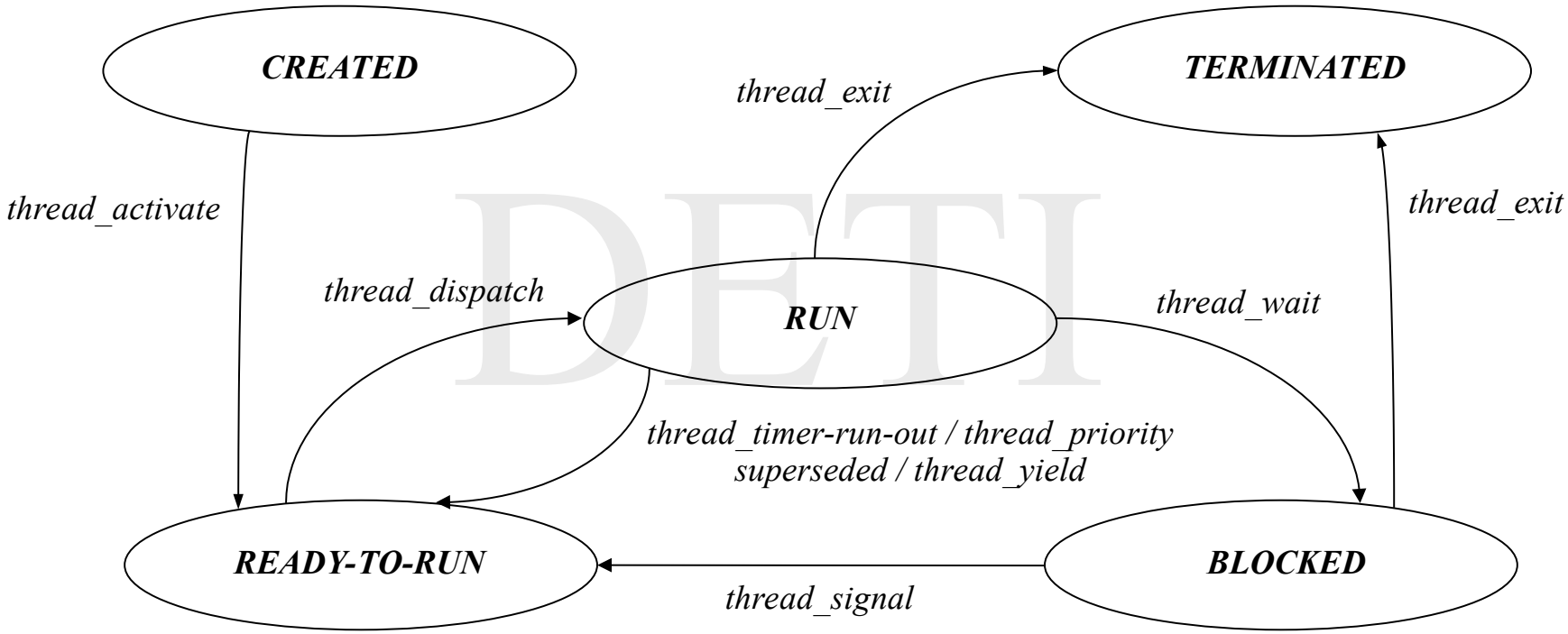
Organization of a multithreaded program - 1



Organization of a multithreaded program - 2

- each *thread* is typically associated with the execution of a *function or procedure implementing a specific activity*
- the *global data* structure forms an information sharing space, defined in terms of variables and communication channels with the input/output devices, to be accessed by the multiple *threads* that coexist at a given time for writing and for reading
- the *main program*, represented in the diagram by a *function or procedure implementing a specific activity*, constitutes the first *thread* to be created and, typically, the last *thread* to be concluded

Thread state diagram



Support to the implementation of a multithreaded environment

- *user level threads – threads* are implemented by a specific library at user level which brings support to the creation, management and scheduling of *threads without* kernel interference; this generates a very versatile and portable, but inefficient, implementation since, as the kernel perceives only processes, when a particular *thread* invokes a blocking *system call*, all the process is blocked, even if there were *threads* ready to be run
- *kernel level threads – threads* are implemented at kernel level by directly providing the operations for the creation, management and scheduling of *threads*; the implementation is operating system specific, but the blocking of a particular *thread* does not affect the dispatching of the remaining for execution and parallel execution in a multicore processor becomes possible

Suggested reading

- *Computer Organization and Architecture: Designing for Performance*, Stalling W., 9th Edition, Prentice Hall, 2013
  - Chapter 1: *Introduction*
  - Chapter 2: *Computer Evolution and Performance*