Concurrency means that several activities are executed simultaneously. Concurrency allows us to reduce

the execution time of a data-intensive problem, as discussed in Section 2.1. To exploit concurrency, often we have to take a fresh look at the problem and design a parallel algorithm. In other instances we can

still use the sequential algorithm in the context of the SPMD paradigm.

Concurrency is a critical element of the design of system software. The kernel of an operating system

exploits concurrency for virtualization of system resources such as the processor and the memory.

Virtualization, covered in depth in Section 5.1, is a system design strategy with a broad range of

objectives, including:

• Hiding latency and performance enhancement (e.g., schedule a ready-to-run thread when the current

thread is waiting for the completion of an I/O operation).

• Avoiding limitations imposed by the physical resources (e.g., allow an application to run in a virtual

address space of a standard size rather than be restricted by the physical memory available on a

system).

• Enhancing reliability and performance, as in the case of RAID systems mentioned in Section 3.5.

Sometimes concurrency is used to describe activities that appear to be executed simultaneously,

though only one of them may be active at any given time, as in the case of processor virtualization,

when multiple threads appear to run concurrently on a single processor. A thread can be suspended

due to an external event and a context switch to a different thread takes place. The state of the first

thread is saved and the state of another thread ready to proceed is loaded and the thread is activated.

The suspended thread will be reactivated at a later point in time.

Dealing with some of the effects of concurrency can be very challenging. Context switching could

involve multiple components of an OS kernel, including the Virtual Memory Manager (VMM), the

Exception Handler (EH), the Scheduler (S), and the Multilevel Memory Manager (MLMM). When a

page fault occurs during the fetching of the next instruction, multiple context switches are necessary,

as shown in Figure 2.11.

Concurrency is often motivated by the desire to enhance system performance. For example, in a

pipelined computer architecture, multiple instructions are in different phases of execution at any given

time. Once the pipeline is full, a result is produced at every pipeline cycle; an n-stage pipeline could

potentially lead to a speed-up by a factor of n. There is always a price to pay for increased performance,

and in this example it is design complexity and cost. An n-stage pipeline requires n execution units, one

for each stage, as well as a coordination unit. It also requires careful timing analysis in order to achieve

the full speed-up.

This example shows that the management and coordination of the concurrent activities increase the

complexity of a system. The interaction between pipelining and virtual memory further complicates

the functions of the kernel; indeed, one of the instructions in the pipeline could be interrupted due to a

page fault, and the handling of this case requires special precautions, since the state of the processor is

difficult to define.

In the early days of computing, concurrency was analyzed mostly in the context of the system software;

nowadays concurrency is a ubiquitous feature ofmany applications. Embedded systems are a class

of concurrent systems used not only by the critical infrastructure but also by the most diverse systems,

from ignition in a car to oil processing in a refinery, from smartmeters to coffeemakers. Embedded controllers

for reactive real-time applications are implemented as mixed software/hardware systems [294].

Concurrency is exploited by application software to speed up a computation and to allow a number

of clients to access a service. Parallel applications partition the workload and distribute it to multiple threads running concurrently. Distributed applications, including transaction management systems and

applications based on the client-server paradigm discussed in Section 2.13, use concurrency extensively

to improve the response time. For example, a Web server spawns a new thread when a new request

is received; thus, multiple server threads run concurrently. A main attraction for hosting Web-based applications is the cloud elasticity – the ability of a service running on a cloud to acquire resources as

needed and to pay for these resources as they are consumed.

Communication channels allow concurrent activities to work in concert and to coordinate. Communication

protocols allow us to transform noisy and unreliable channels into reliable ones that deliver

messages in order. As mentioned earlier, concurrent activities communicate with one another via shared

memory or viamessage passing. Multiple instances of a cloud application, a server and the clients of the

service it provides, and many other applications communicate via message passing. TheMessage Passing

Interface (MPI) supports both synchronous and asynchronous communication, and it is often used

by parallel and distributed applications.Message passing enforces modularity, aswe see in Section 2.13,

and prevents the communicating activities from sharing their fate; a server could fail without affecting

the clients that did not use the service during the period the server was unavailable.

The communication patterns in the case of a parallel application are more structured, whereas patterns

of communication for concurrent activities of a distributed application are more dynamic and

unstructured. Barrier synchronization requires the threads running concurrently to wait until all of them

have completed the current task before proceeding to the next. Sometimes one of the activities, a coordinator,

mediates communication among concurrent activities; in other instances individual threads

communicate directly with one another.