Chapter 1: System Design Requirements

System software runs indefinitely, typically supporting other running programs and users; concept of *concurrent activity* is central.

Concurrent => 'at the same time'

└> system software has to support many users/programs at the same time => handle separate activities which are in progress at the same time

=> two activities are concurrent if, at a given time, each is at some point between its starting point and finishing point.

Inherently concurrent: system handles activities that can happen simultaneously in the world external to the computer system (kan niet anders, moet concurrent)

potentially concurrent: it might be possible for an application to work on parts of a problem in parallel by devising a concurrent algorithm for its solution (kan concurrent gemaakt worden, bv om snelheid ↗)

1.1 Inherently concurrent systems

1.1.1 Real-time and embedded systems

Real-time systems => time requirements dedicated by the environment of the computer system (nooit nul)

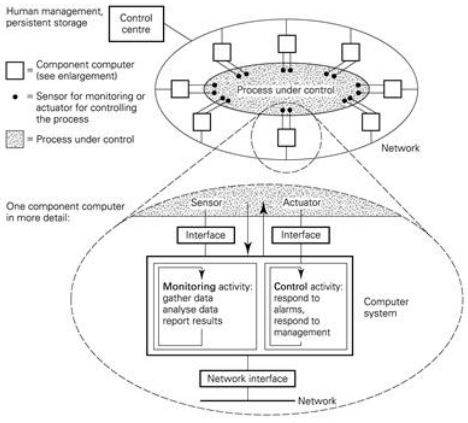
Hard real-time => timing requirements are absolute (harde grens)

Soft real-time => failing to meet a deadline will not lead to a catastrophe

bovengrens voor communicatie is snelheid van het licht

Static real-time => analysis of the activities that must be carried out by the system can be done when the system is designed

Dynamic real-time => requests may occur at irregular and unpredictable times and the system must respond dynamically and with guaranteed performance



Process control

Concurrent: veel componenten samen die tegelijk het proces ondersteunen

Data is gathered from the controlled system and analyzed

Accuracy depends on amount of data gathered and the time that can be spent on analysis

Data gathering ad analysis are periodic and predictable

Period depends on process being controlled

Periodic activity must be integrated with ability to respond to unpredictable events

Monitoring => gathering data

Control => tuning, response to alarms

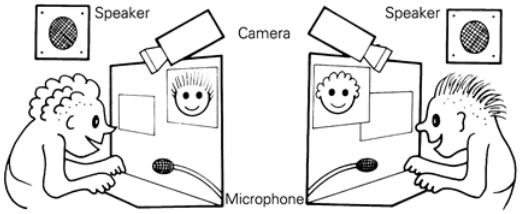
Trade-off: at times accuracy of data analysis may be sacrificed to rapid response

Not all real-time systems are distributed, multicomputer systems

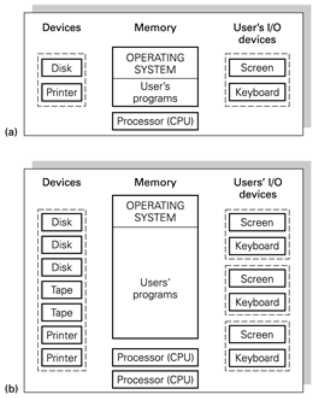
Some may have a single embedded controlling computer (embedded: systeem dat in ander systeem zit; embedded is altijd dedicated)

Multimedia support

Multimedia: ability to show moving pictures and hear voice in addition to the traditional display of text and graphics
↳> video & audio must be synchronized
Video conferencing, video phones
When multimedia data is stored, requests to deliver it are unpredictable and the system must respond dynamically => guarantees of certain quality of service
video: required transfer rate: 200 kb/sec
Voice to be sync'd with video: 64 kb/sec



1.1.2 Operating systems and distributed operating systems



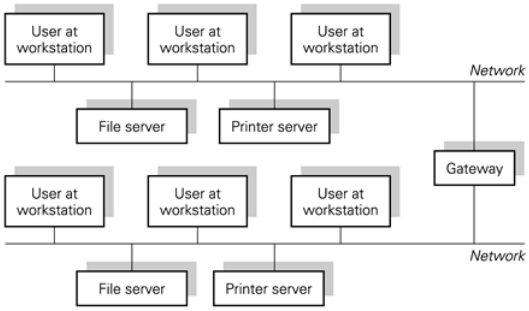
The terminal devices are packaged as separate units. There may be large numbers of them, they may be remote from the computer and they may be handled by an intermediate controlling computer, a terminal concentrator.

self-standing computer system
(a) Single-user system
(b) Multi-user system
Single-user => single computer is perceived
Multi-user => users access the system via a terminal -> separate from main memory, central processor(s) and shared devices (disks, printers,…)
System might have large numbers of terminals

Scope for concurrency in both single-user and multi-user systems
⇨ Devices tend to be very slow compared with processors
↳> OS will attend to the devices when necessary, but run programs while the devices are busy producing the next input or performing the last output
↳> OS will attempt to overlap processing and device handling wherever possible

Events handled by OS tend to be irregular and unpredictable rather than periodic, load they must handle is dynamic.
Multi-user OS must manage the sharing of system resources between users (with acceptable service for all of them), respond to potentially conflicting demands from users for the resources it manages. Requests will occur dynamically and some will involve a number of related objects which are managed by the OS.

Distributed Operating Systems



distributed systems
└> computers connected by a communications medium such as a LAN
Workstation OS = single-user system with scope for concurrency
-> likely that file storage is provided as a network-based service
also likely that a number of file servers will be needed to provide sufficient storage space and processing capacity.

These shared servers must respond to simultaneous requests from clients => concurrent systems
OS in computers part of a distributed system contain software for communications handling => a communications handling subsystem of an OS is itself a concurrent system
(subsystem: a major functional unit within a system)
There are many distributed systems connected by WANs and communications software is in general designed to allow world-wide interactions.

1.1.3 Window-based interfaces

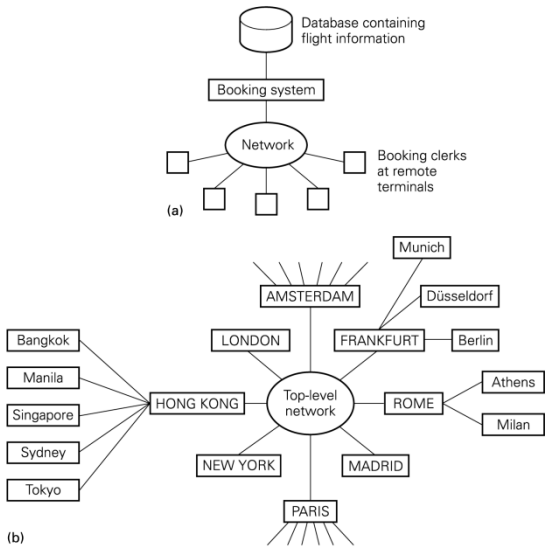
GUI, Window system: make concurrent activity explicit and natural to computer users

1.1.4 Database management and transaction processing systems

DB applications are concerned with large amounts of persistent data, that is, data on permanent storage devices that exists independently of any running program.
DBMS is a concurrent system since it may have to handle several clients simultaneously (queries, updates) (scope for concurrency). The term *transaction* is used for a request from a client to a DBMS.
In applications where it is not critical that at all times the data appears to be up to date to the current instant, updates may be batched and run when convenient. This approach simplifies the management of the DB enormously and is likely to be used if at all possible. Many of the problems associated with concurrent systems are avoided by this means.
In some systems the activities of reading and updating data are closely related and updates cannot be deferred. Each transaction needs to see the up-to-date system state.
Concurrent access is desirable for a fast response to queries and there are unlikely to be conflicting requests, i.e., requests that attempt to update and read the same part of the DB at the same time. Any number of transactions which only require to read the DB may be run in parallel. If it is possible to update the DB as well as read it, the DBMS must ensure that transactions do not interfere with each other.
Data will be copied from secondary storage into main memory and reads and updates will

use the main memory copy. Potential problem: the copy of the data held on secondary storage becomes out of date. If the system crashes, the contents of main memory may be lost. A transaction system must therefore support concurrent access and allow for system failure at any time.

figuur: (a) components of a transaction processing system for airline bookings (b) world-wide nature of such a system



Requirements for building transaction processing and DB management systems:
- there is a need to support separate activities
- there is a need to ensure that the separate activities access and update common data without interference
- there is a need to ensure that the results of transactions are recorded permanently and securely before the user is told that an operation has been done.

1.1.5 Middleware (also chapter 16)

The idea is to build a layer of software (middleware) above the heterogeneous operating systems to present a uniform platform above which distributed applications can run. This simplifies the development of distributed applications by removing the need to port each application to a range of operating systems and hardware.

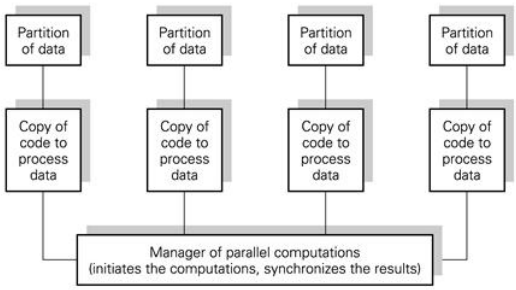
1.2 Supporting potentially concurrent systems

Applications which might benefit from concurrent implementation.
General motivation for exploiting potential concurrency:

- There is a large amount of computing to be done;
- There is a large amount of data to be processed;
- There is a real-time requirement for the results;
- Hardware is available for running the potentially concurrent components in parallel.

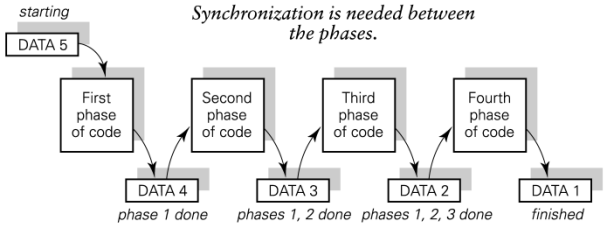
1.2.1 Replicated code, partitioned data

Simplest approach is to use a sequential algorithm but to partition the data.
figuur: if all the components must run to completion the overall time taken is that of the longest component plus the overhead of initiating the parallel computations and synchronizing on their completion. The



speed-up achieved therefore depends on being able to divide the computation into pieces with approximately equal computation time. If we can divide the computation into components which require the same amount of computation time, we have maximized the concurrency in the execution of the algorithm.
Particularly suitable for large amounts of data. While one parallel activity waits for more data to be brought in from disk, another can proceed with processing data in main memory. In some applications, once a solution is found by one component, all the others can cease their activity.

1.2.2 Pipelined processing



Another simple approach is the pipeline. If the total function to be applied to the data can be divided into distinct processing phases, different portions of data can flow along from function to function.

figuur: four-stage pipeline (example: compiler); as soon as the first program or module has passed the lexical analysis phase it may be passed on to the parsing phase while the analyser starts on the second program or module.
Requirements: separate activities have to be able to synchronize with each other: to wait for new data to become available or to wait for the next phase to finish processing its current data.

1.2.6 Requirements for supporting concurrent applications

- support for separate activities
  - monitoring & control activities, running user programs, handling devices, …
- support for the management of separate activities
  - create, run, stop, kill activities
  - indicate relative priorities of activities
- support for related activities to work together (part II)
  -activities generating part of a solution (cooperative)
  - device handlers delivering data to user programs
  - transactions on behalf of many customers (competitive)
- ability to meet timing constraints
  - real-time constraints versus reasonable response time
  - synchronized, jitter-free audio & video
- support for composite tasks (part III)
  - a single task may have several components, executed concurrently (with other tasks); the system may fail before all components are completed

1.3 Architectures for software systems

1.3.1 System classification

- SISD: single instruction stream, single data stream => single processor fetching and executing a sequence of instructions (original von Neumann model)
- SIMD (vectorprocessor): single instruction stream, multiple data stream => many processing elements but they are designed to execute the same instruction at the same time. Processors can be simple: they do not need to fetch instructions from their private memory, but receive their instructions from a central controller. They do, however, operate on separate data such as the items of vectors or arrays. (1 instructie uitvoeren op meerdere gegevens)
- MIMD: multiple instruction stream, multiple data stream => systems with more than one processor fetching and executing instructions. The instructions need not be closely related and they are executed asynchronously. Very broad category, includes networks of computers and multiprocessors built from separate computers. (meerdere instructies simultaan uitvoeren – iedere instructie op 1 gegeven; hiervoor meerdere CPU's nodig; vb: laptop = MIMD: heeft meerdere cores)

1.3.2 Conventional uniprocessors

A single uniprocessor computer may be used to implement some kinds of concurrent system. Example: a time-sharing OS. Many users are interacting with the system simultaneously and many programs are in the process of being executed at a given time. Only one processor may be transparent to the programmer. At the application software level it is as though each program is executing on a dedicated processor. The technique of **forbidding interrupts** may be used for controlling concurrency. You can prevent the processor from doing anything other than its current activity until it reaches the end of it, when you 'enable interrupts' again. This obviously has to be used with great care and for short periods of time.

1.3.3 Shared-memory multiprocessors

The conventional multiprocessor model is of a relatively small number of processors (2 – 30) executing programs from a single shared memory. Only one copy of the OS and any other system software is needed. Software from the same memory is being executed at the same time on a number of processors. It is more difficult to write operating systems that run on multiprocessors than on uniprocessors. Sometimes, to simplify the design, one dedicated processor will execute the OS and one dedicated processor will handle all I/O. When more than some fairly small number of processors access a single shared memory, contention for memory access becomes too great. The effect can be alleviated by providing the processors with hardware controlled caches in which recently accessed memory locations are held.

1.3.4 Multicomputer multiprocessors

Interconnected computers
assumption: each processor accesses only its own local memory directly. The processors may be placed on a shared bus. They communicate with each other through the control and data paths of the bus. A hierarchy of buses may be used to increase the number of processors in the system. To exploit massive concurrency we may wish to use many tens of processors. In this case their connectivity is a major design decision.
A general problem with multicomputers is the complexity of the software systems required to manage them and the time to load software prior to execution. At present, such topologies tend to be used for software that will run indefinitely once loaded, such as in embedded control systems.

1.3.5 Dataflow (data-driven) architectures

An instruction may execute as soon as its data operands are ready. Any number of instructions can execute in parallel, depending on the number of processors available. It is possible to arrange for the operations to be at most two arguments. The first argument to be ready for an operation will detect that its pair is not yet available and will wait in memory provided for the purpose. The second will detect that the first is ready, by an associative match on the memory, and they will then go off together to a processor for execution. The result of the operation will become an argument for another operation, and so on.
The concurrency made available by the dataflow approach is potentially very great and at a very fine granularity: at the machine instruction level.
A problem with a data-driven approach is that unnecessary processing may be carried out for paths in a program that may never be followed.

1.3.6 Architectures for functional languages

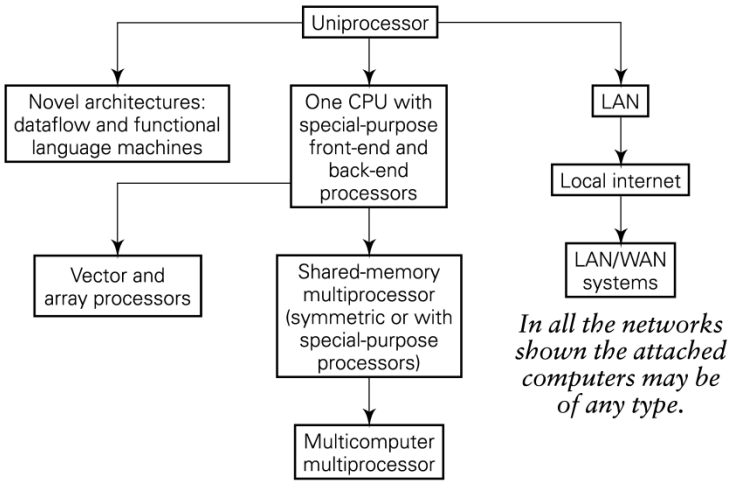
A pure functional language does not have destructive assignment statements. When a function is applied to its arguments it always produces the same result since there can be no side-effects.
It is argued therefore that functional languages are inherently concurrent. Any number of function applications can be carried out in parallel. There is greater potential for controlling the concurrency than in the dataflow approach if we can defer a function application until we are sure that the result is needed: so called ‘lazy evaluation’. Both data-driven and demand-driven architectures are still in the research domain.

1.3.7 Network-based systems

A local area network (LAN) offers full connectivity for some number of computer systems. The basic philosophy of network communications software is to regard the network as a shared resource which must be managed. Certain attached computers exist solely to support communication, for example, when two LANs are connected by a gateway computer. LANs are widely used as a basis for program development environments. The LAN is the

interconnection medium for a distributed system.
When computers of LANs need to be connected over long distances, wide area networks (WANs) are used.

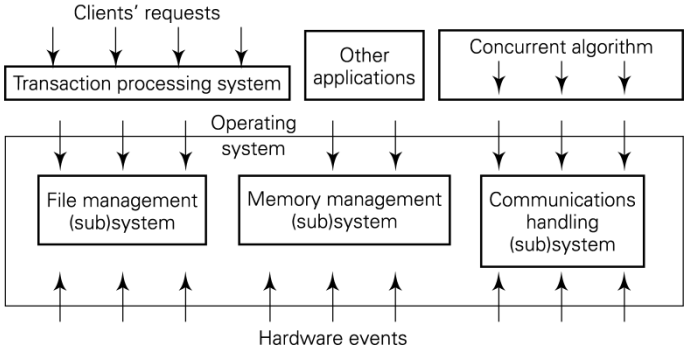
1.3.8 Summary of hardware bases for concurrent systems



The figure summarizes the topologies that might be used as the hardware basis for a concurrent system. The arrows indicate the direction of increasing concurrency achieved through different paths of development.
The central path takes a single uniprocessor as starting point and attempts to introduce an increasing amount of processing power. Special-purpose processors can handle devices, but a major design change comes when several processors are used to execute programs residing in a single memory. Above a certain number of processors accessing a given memory, a topology comprising interconnected computers is necessary. There is a variety of possible interconnection structures. Vector an array processors may be seen as special-purpose multiprocessor machines.
An alternative to the complexity of the interconnection structures of multi-computer multiprocessor systems is a network. A LAN medium may be used to achieve full connectivity of a certain number of computers, but communication is achieved through software rather than hardware protocols. As LAN performance increases, networked computers can be used for distributed computations as well as the more conventional program development environments, provided that the operating system and communications software impose an acceptable and bounded overhead.

1.4 defining a concurrent system

Different activities are in the process of being executed at the same time, that is, concurrently. Concurrency in the application must be supported by the system that runs it.
The *figure* shows some of the concurrent (sub)systems that have been mentioned. In each case a number of activities may be in progress simultaneously within the system, either because components of a concurrent algorithm have been started off in parallel or because a number of clients may simultaneously make demands on the system (as shown).



The concurrent system may be an OS, a subsystem within an OS or an application-level service running above an OS. In all cases, the concurrent system shown may be distributed across a number of computers rather than centralized in a single computer.
Separate activities may be working together to solve a problem; they may be independent but running above common system software and needing to share common system resources for which they must compete.
The activities may need to interact with the external environment of the computer system and may need to meet timing requirements when doing so; they may need to use the main memory only of a single computer, or use main memory only but of a number of computers in a distributed system; they may access or record data in persistent memory, in this case they use both main memory and persistent memory of a single computer / computers in a distributed system.

1.5 systems implementation requirements

- support for separate activities: monitoring & control activities, running user programs, handling devices,…
- support for the management of separate activities: create, run, stop, kill activities, indicate relative priorities of activities
- support for related activities to work together (part 2): activities generating part of a solution (cooperative), device handlers delivering data to user programs, transactions on behalf of many customers (competitive)
- ability to meet timing constraints: real-time constraints vs. reasonable response time, synchronize, jitter-free audio & video
- support for composite tasks (part 3): a single task may have several components, executed concurrently (with other tasks); the system may fail before all components are completed

1.6 Security, protection and fault tolerance in system design

The need for this is orthogonal to the development of concurrent systems.
We can express a general requirement that it should be possible for policies to be expressed and enforced relating to who may have access to computers, networks and information.
Huge topic, cfr. also other chapters.