O
O S E
E

Modularization and Hierarchy

Operating-System Engineering

Hierarchy in System-Software Design

- unfortunately there is no unique meaning of "hierarchy" in systems design:
  - **module hierarchy** 4
  - **functional hierarchy** 7
  - **uses hierarchy** 14
- these kinds of hierarchy are quite different in representation and semantic
  - they will be investigated by discussing a common **case study** 2
- which kind of hierarchy to choose depends on what needs to be expressed

Case Study — A Memory-Management Subsystem

- given is a subset of functionally dedicated operating-system building blocks
  - the building blocks represent "coarse-grain structured" system functions [1]
  - the system exhibits three different threads of concurrency:
    * two processes (one application thread, one system thread)
    * one interrupt (clock)
  - the task is to design module, functional, and uses hierarchy from these parts
  - the building-block subset is extended in the course of *stepwise refinement*
- name and intention of a building block are specified by the *domain lexicon*

[1] Note, this is for the ease of understanding. Goal is not to fully design a memory-management subsystem—and, thus, to get lost in a lot of details—but to design a somewhat realistic system structure for comparison purposes.

Domain Lexicon — 1. Printing

garbage collector searches for allocated but unused memory, reclaims the corresponding segments and frees the reclaimed pieces.

memory manager maintains the free list, allocates memory upon request and relates the allocated segments to processes.

resource manager performs blocking synchronization based on semaphores.

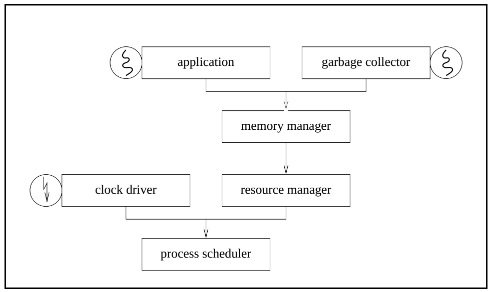
process scheduler maintains the run list and suspends, preempts, and schedules processes upon request and dispatches them to the CPU.

clock driver supports the implementation of CPU protection (i.e., preemption).

Module Hierarchy

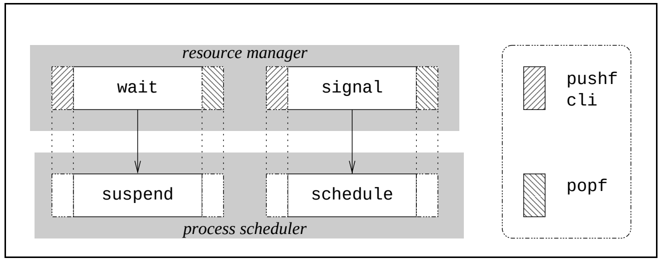
- the arrangement documents the *call relation* between the building blocks
  - calls coded in the programs involved largely define the global structure
  - system functions are technically represented by (a set of) procedures
- the hierarchy is built from programming-language structuring concepts
  - procedures and functions, i.e., procedures free of side effects
  - modules encapsulating procedures and/or data sets
- the implementation may consist of functions not represented by these concepts

Module Hierarchy — Call Relation



- arcs/arrows show the call relation
  - *garbage collector* is scheduled
    - as is *application*
  - *clock driver* is an interrupt
- *memory manager* is critical
  - overlapped by two threads
  - secured by *resource manager*
- *resource manager* and *process scheduler* must be secured against interrupts

Module Hierarchy — Refinement



The critical sections of `suspend()` and `schedule()` are implicitly secured because both functions are called from within the already secured critical sections of `wait()` and `signal()`. Also note that, in this scenario, `pushf` could be deleted and `popf` could be replaced by `sti`. Any way, neither of these solutions is a good practice. Why?

- the interrupt-synchronization function is important, yet not documented
- the design may lack functions which appear to be present in the implementation

Functional Hierarchy

- the arrangement documents the *functional relation* between the building blocks
  - specified is the logical relationship, not the physical (i.e., real) one
  - a memory footprint may exhibit no structuring measures at all
  - structures may be visible only in the design document or source code
- the design abstracts from the function's actual implementation
  - functions may be represented as processes, modules, procedures, or macros
  - from the functions' point of view, any representation is as good [2]
  - in the same design, the representations may become a configuration matter
- the implementation does not show a function which is not shown in the design

Functional Hierarchy — Habermann et al. [1]

In a functional hierarchy where functions may actually be macros, a sequence of functions calls may result in a single machine instruction (or possibly none at all) when the system is compiled.

It is **the** system **design** which **is hierarchical**, not its implementation.

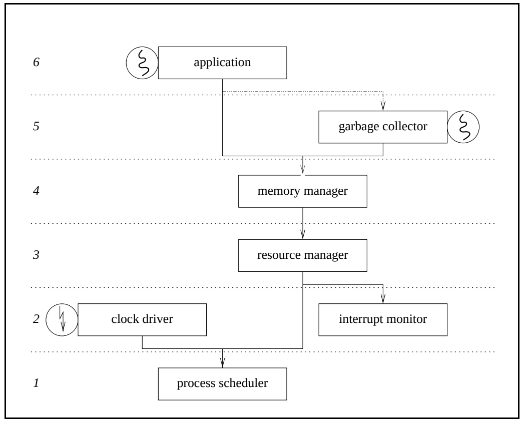
Functional Hierarchy — Virtual Machines

- the concept is to partition the system design into levels
  - without implying anything per se about the interactions between the levels
- the hierarchical structuring is based upon functions [1]:

  > The levels L_0, L_1, . . . L_n are ordered such that functions defined in level L_i are also known to L_{i+1} (and, at the discretion of L_{i+1}, to L_{i+2}, etc.). L_0 corresponds to the hardware instructions of the target machine. Each level, in fact, is regarded as providing new "hardware" to the next higher level.

- each level is comprised of a set of functions whose names are statitcally known

Functional Hierarchy — Levels of Abstraction



- arcs relate employed functions
  - *application* employs *garbage collector* indirectly (IPC)
- functions are assigned to levels
  - providing operations with statically known names
- functions are "passed through"
  - those of level L_i are also known/visible at L_{i+1}

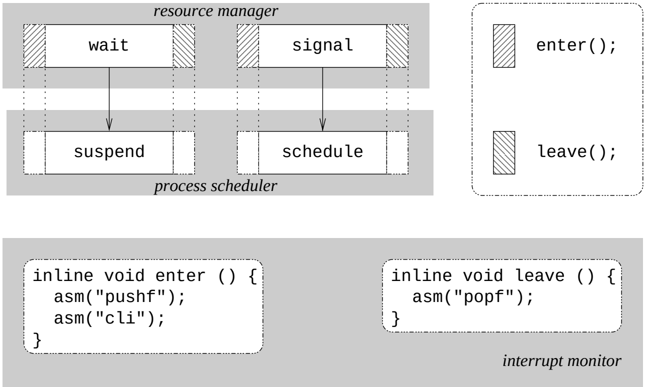
Domain Lexicon — 2. Printing

interrupt monitor (or **clock monitor**) takes care of (clock) interrupt synchronization. The function at this level may be implemented as follows:

1. hard synchronization by basing on privileged CPU instructions such as to physically disable/enable all, or a selected subset of, hardware interrupts.
2. soft synchronization by distinguishing between unmaskable hardware interrupts and maskable software interrupts. For example, software interrupts can be masked by raising a lock variable and delayed, for the duration of the critical section, by putting them on a queue.

Both, *clock driver* and (*interrupt*) *monitor* share the particular design decision on how synchronization at this level of abstraction takes place.

Functional Hierarchy — Refinement



Functional Hierarchy — Module vs. Level

Information **modules are comprised of** some data structures (possibly) and **a set of functions which share knowledge of a particular design decision** (reflected, for example, in the details of the data structures).

A level is a set of function names which are implemented via functions in lower levels.

There exists no necessary relationship between the two concepts. This not only allows the division of a single level into several distinct modules, but in addition allows for the selective spanning of several levels by a single module! (→ p. 19) [1]

Uses Hierarchy

- the arrangement documents the *functional dependency* of the building blocks
  - it specifies dependencies in a way allowing one to reason about correctness
- "*uses*" means "*to be dependent on the availability of a correct implementation*"

$$A \textit{ uses } B \text{ if} \begin{cases} B\text{'s correct execution is mandatory to fullfil } A\text{'s task} \\ \text{the correctness of } A \text{ depends on the correctness of } B \end{cases}$$

- that B is *used* by A is obtained from A's implementation <u>and</u> specification

Uses Hierarchy — "Uses" vs. "Call"

- a (procedure) call must not necessarily be an instance of a uses relation
  - e.g., when (according to A's specification) B is called conditionally by A
  - A may execute correctly although B's implementation may be incorrect
- a uses relation may be given even in the absence of an explicit call relation
  - A *uses* B implicitly if, e.g., B handles asynchronous program interruptions
  - A may execute incorrectly although A's specification lacks any call to B
- that calls are not automatically instances of *uses* must not be a rare case

Uses Hierarchy — Acyclic Graph

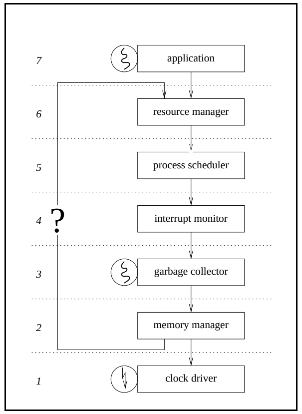
- level L_0
  - is made of programs which don't *use* any further programs
- level L_i, $i > 0$
  - is made of programs which *use* at least one program of level L_{i-1}
  - excludes the *use* of all programs above level L_{i-1}

Uses Hierarchy — Criteria of Uses

A ***uses*** B (recommended) if . . .

- A becomes more simpler and elementary through the *use* of B
  - B has been designed (originally) to support only A
- the structural complexity of B (when *used* by A) is not increased → p. 19
  - since the (direct/indirect) *use* of A by B is excluded
- there exists another subset already containing B but not A
  - B exists, is already *used* and will be re-*used* by A
- there exists no other subset already containing A but not B
  - otherwise, the specification of A tends to be inconsistent

Uses Hierarchy — Uses Relation



- a layering of functions to reflect dependency issues:

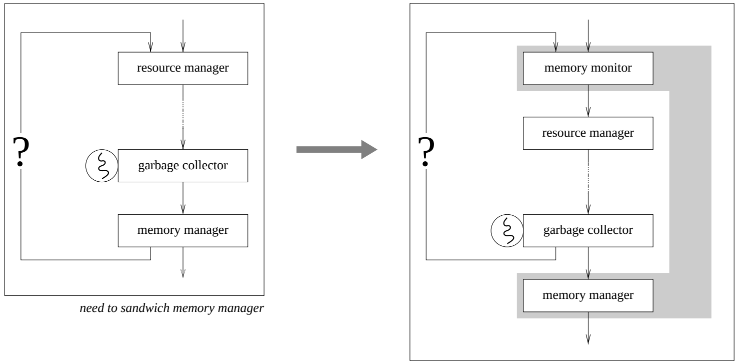
  L_4 the *interrupt monitor* must ensure the integrity of higher-level critical sections
  L_3 the *garbage collector* must ensure to never reclaim allocated "active" memory
  L_2 the *memory manager* must ensure to never allocate "active" memory to processes
  L_1 the *clock driver* must ensure to never change the processor state of interrupted programs

- L_6 and L_2 are conflicting, they *use* each other **!**

Uses Hierarchy — Sandwiching

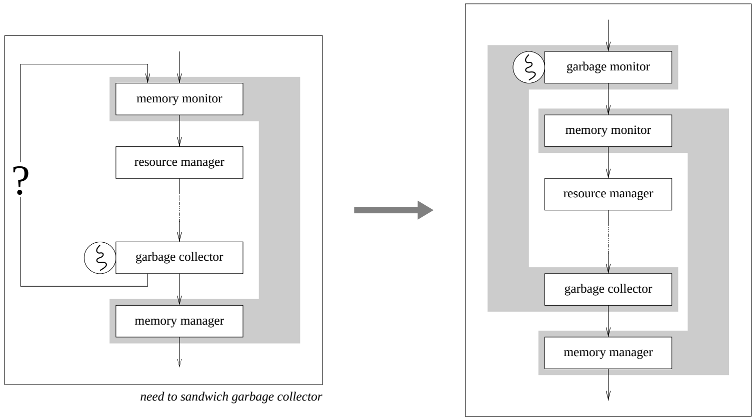
- sometimes programs may benefit from each other, causing a *cyclic uses relation*
  - e.g., *resource manager* depends on *memory manager* and vice versa
- this *uses* conflict is resolved by splitting one program up into two slices
  - if A and B *mutually use* each other, e.g. B is split up into B_1 and B_2
  - additionally, A is changed to *use* B_2 and B_1 is set up to *use* A
  - A becomes the "spread" of a sandwich with B_1 and B_2 as the "bread"
- the technique may be applied recursively and, thus, "fine-tunes" modularization

Uses Hierarchy — Memory-Manager Slicing



need to sandwich memory manager

Uses Hierarchy — Garbage-Collector Slicing



need to sandwich garbage collector

Uses Hierarchy — Modularization

- sandwiching is typical for modules not being levels, and vice versa
  - a module's functions share knowledge about design decisions → p. 13

    $\left.\begin{array}{l}\textit{memory monitor \& memory manager}\\ \textit{garbage monitor \& garbage collector}\end{array}\right\}$ likewise build a module

  - the module's functions may be assigned to different levels → p. 24
    - ∗ *memory management* spans levels L_2 and L_7
    - ∗ *garbage collection* spans levels L_3 and L_8
  - several modules may be assigned the same level of abstraction → L_2, p. 10
- sandwiching increases the structural complexity of sliced programs

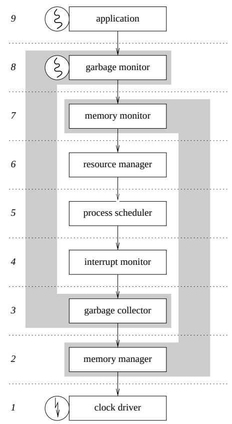
Domain Lexicon — 3. Printing

memory monitor separates the (non-functional) synchronization aspect from the (functional) memory management aspect by adding "synchronization brackets" (i.e., pairs of `wait()`/`signal()` by *using* the *resource manager*) to the unsynchronized *memory manager* functions.

garbage monitor separates the (non-functional) synchronization aspect from the (functional) garbage collection aspect by depending on the synchronization measures of the *memory monitor* to ensure integrity of the critical *garbage collector* section(s).

Uses Hierarchy

Refinement



Summary

- the *module hierarchy* shows dependencies with respect to calls
  - the design tends to be incomplete concerning the functions provided
- the *functional hierarchy* shows dependencies with respect to functions
  - the design abstracts from any implementation decision
  - the design specifies a hierarchy of virtual machines building a system
- the *uses hierarchy* shows dependencies with respect to correctness
  - the design abstracts from the real (functional) system structure
  - the design specifies on how to reason about system integrity

Bibliography

[1] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[2] H. C. Lauer and R. M. Needham. On the Duality of Operating System Structures. *ACM Operating Systems Review*, 13(2):3–19, Apr. 1979.

[3] D. L. Parnas. Some Hypotheses About the “Uses” Hierarchy for Operating Systems. Technical Report BS I 75/2, TH Darmstadt, 1975.