O
O S E
E

Introduction

Operating-System Engineering

Trivial Pursuit in Computer Science

Q: *"What is an elephant?"*

A: *"A mouse with an operating system."*

The Operating-System Design Dilemma

Clearly, the operating system design must be strongly influenced by the type of use for which the machine is intended. Unfortunately it is often the case with ‘general purpose machines’ that the type of use cannot easily be identified; **a common criticism of many systems is that, in attempting to be all things to all individuals, they end up being totally satisfactory to no-one**. [2]

General Purpose System

- being prepared on all eventualities — „Eier legende Wollmilchsau“

$$\text{e.g., enforcing} \left\{ \begin{array}{l} \textit{scheduling} \\ \textit{protection} \\ \textit{security} \end{array} \right\} \text{in a single-} \left\{ \begin{array}{l} \text{process} \\ \text{program} \\ \text{user} \end{array} \right\} \text{environment}$$

- optimized towards the most probable and common “standard” use case
  - at the cost of all the cases that deviate from the artificially defined norm
- no (system) function is free of charge — not even a “sleeping beauty”

General Purpose System $\Longleftrightarrow$ General Purpose Function

. . . System provides general <u>services</u> for a broad range of applications

- shows up with a rich set of system functions
  – trying to cover all of the demands stated by the various user programs
- aims at providing users with the best possible compromise
  – there can't be only one optimal solution to a number of various problems
- follows some sort of *black-box model* by its (mostly) fixed system interface

. . . Function same as above, but services become <u>attributes</u> of a single function

General Purpose Function — `printf(3)`

```
wosch@hawaii 37> uname -snrm
Linux hawaii.cs.uni-magdeburg.de 2.2.14 i686
wosch@hawaii 38> echo 'main(){printf("Hello world!\n");}' > hello.c
wosch@hawaii 39> gcc -O6 -c hello.c; gcc -static -o hello hello.o
wosch@hawaii 40> hello
Hello world!
wosch@hawaii 41> ls -l hello*
-rwxr-xr-x   1 wosch    ivs        932099 Dec  1 11:08 hello*
-rw-r--r--   1 wosch    ivs            33 Dec  1 11:06 hello.c
-rw-r--r--   1 wosch    ivs           908 Dec  1 11:07 hello.o
wosch@hawaii 42> size hello hello.o
   text    data     bss     dec     hex filename
 195562    5064    3340  203966   31cbe hello
     29       0       0      29      1d hello.o
```

General Purpose Function(contd.) — No Isolated Phenomenon

```
skiffcluster5:~$ uname -snrm
Linux skiffcluster5.handhelds.org 2.4.0-test1-ac7-rmk1-crl2 armv4l
skiffcluster5:~$ echo 'main(){printf("Hello world!\n");}' > hello.c
skiffcluster5:~$ gcc -O6 -c hello.c; gcc -static -o hello hello.o
skiffcluster5:~$ ./hello
Hello world!
skiffcluster5:~$ ls -l hello*
-rwxr-xr-x    1 guest    users      988494 Mar  6 03:13 hello
-rw-r--r--    1 guest    users          34 Mar  6 03:13 hello.c
-rw-r--r--    1 guest    users         816 Mar  6 03:13 hello.o
skiffcluster5:~$ size hello hello.o
   text    data     bss     dec     hex filename
 214538    4208    3252  221998   3632e hello
     42       0       0      42      2a hello.o
```

```
wosch@statler 37> uname -snrm
Linux statler 2.2.15pre3 ppc
wosch@statler 38> echo 'main(){printf("Hello world!\n");}' > hello.c
wosch@statler 39> gcc -O6 -c hello.c; gcc -static -o hello hello.o
wosch@statler 40> hello
Hello world!
wosch@statler 41> ls -l hello*
-rwxr-xr-x   1 wosch    ivs       1051922 Mar  6 11:32 hello*
-rw-r--r--   1 wosch    ivs            33 Mar  6 11:32 hello.c
-rw-r--r--   1 wosch    ivs           864 Mar  6 11:32 hello.o
wosch@statler 42> size hello hello.o
   text    data     bss     dec     hex filename
 236908    4804    3740  245452   3becc hello
     60       0       0      60      3c hello.o
```

```
wosch@cabrera 37> uname -snrm
Linux cabrera 2.2.17-4 alpha
wosch@cabrera 38> echo 'main(){printf("Hello world!\n");}' > hello.c
wosch@cabrera 39> gcc -O6 -c hello.c; gcc -static -o hello hello.o
wosch@cabrera 40> hello
Hello world!
wosch@cabrera 41> ls -l hello*
-rwxrwxr-x    1 wosch    ivs       2673622 Mar  6 11:43 hello*
-rw-rw-r--    1 wosch    ivs            34 Mar  6 11:43 hello.c
-rw-rw-r--    1 wosch    ivs          1888 Mar  6 11:43 hello.o
wosch@cabrera 42> size hello hello.o
   text    data     bss     dec     hex filename
 427176   19634    7088  453898   6ed0a hello
     62       0       0      62      3e hello.o
```

```
wosch@niihau 37> uname -snrm
SunOS niihau 5.7 sun4u
wosch@niihau 38> echo 'main(){printf("Hello world!\n");}' > hello.c
wosch@niihau 39> gcc -O6 -c hello.c; gcc -static -o hello hello.o
wosch@niihau 40> hello
Hello world!
wosch@niihau 41> ls -l hello*
-rwxr-xr-x   1 wosch    ivs        299608 Mar  6  2001 hello*
-rw-r--r--   1 wosch    ivs            34 Mar  6  2001 hello.c
-rw-r--r--   1 wosch    ivs           860 Mar  6  2001 hello.o
wosch@niihau 42> size hello hello.o
   text    data     bss     dec     hex filename
 173560    6741    3072  183373   2cc4d hello
     38       0       0      38      26 hello.o
```


General Purpose Function(contd.) — Memory Footprints

Program	Size (in Bytes)					
	Linux				Solaris	Windows
	i686	arm	ppc	alpha	sparc	i586
`hello`	203 966	221 998	245 452	453 898	183 373	30 935
`hello.o`	29	42	60	62	38	≈29
%	*0.014*	*0.018*	*0.024*	*0.013*	*0.02*	≈0.094

"General Purpose" Considered Harmful?

- it depends — the interface alone is not always the cause of all evil
  - much more crucial tends to be the function's internal software structure
    - degree of modularization, modul interdependencies, uses relation, . . .
  - black-boxing aims at hiding exactly these internals from the user
    - much in the same way as an *abstract data type* (ADT) [1]
  - the desired features may be present internally, but they remain hidden

- one might expect `puts(3)` to be the "streamlined" `printf(3)` alternative

Special Purpose Function — `puts(3)`

```
wosch@hawaii 37> uname -snrm
Linux hawaii.cs.uni-magdeburg.de 2.2.14 i686
wosch@hawaii 38> echo 'main(){puts("Hello world!");}' > hello.c
wosch@hawaii 39> gcc -O6 -c hello.c; gcc -static -o hello hello.o
wosch@hawaii 40> hello
Hello world!
wosch@hawaii 41> ls -l hello*
-rwxr-xr-x   1 wosch    ivs        932715 Mar  7 16:16 hello*
-rw-r--r--   1 wosch    ivs            30 Mar  7 16:16 hello.c
-rw-r--r--   1 wosch    ivs           908 Mar  7 16:16 hello.o
wosch@hawaii 42> size hello hello.o
   text    data     bss     dec     hex filename
 195759    5076    3360  204195   31da3 hello
     28       0       0      28      1c hello.o
```

Where the Shoe Pinches

Where the Shoe Pinches(contd.) System Architecture

- `printf(3)` supports formatted I/O in many respects:
  1. assortment of plain data types and sizes +
     - `int`, `unsigned`, `float`, `char`, `char*`; `short`, `long`, `double`
  2. various kinds of numbering schemes +
     - dual, octal, decimal, hexa-decimal
  3. different formats +
     - left/right aligned, user-defined field widths
- not every application exploits all these features
  - "`Hello World!`": a left-aligned character string (`char*`)
- but nonetheless, every application is charged with all these features −

Where the Shoe Pinches(contd.) — Object Modules

- a single reference to `printf(3)` entails a number of follow-up references
  - to functions called unconditionally
    - ∗ e.g., output of the assembled character buffer using `write(2)` +
  - to functions called conditionally
    - ∗ e.g., output of character '-' when displaying a negative `int` +
    - ∗ e.g., output of an `unsigned` after having parsed format '%u' +
    - ⋮
    - ∗ e.g., the option for an `int` although '%i' gets never parsed −
  - similar holds for variables, constants, and other addressable units
- "monolithic source modules" of that kind result in overloaded object modules

Where the Shoe Pinches(contd.)

foo.cc — example of a source-module structure similar to `printf(3)`

```
#define LINE_SIZE 64

char line[LINE_SIZE];
int  slot;

void resetline () {
    for (int i = 0; i < LINE_SIZE; i++)
        line[i] = '\0';
    slot = 0;
}

void flushline () {
    write(1, line, slot);
}

void writeline (char c) {
    line[slot++] = c;
}
```

```
void putcharacter (char c) {
    if (slot == LINE_SIZE) {
        flushline();
        resetline();
    }
    writeline(c);
}

void putstring (char* line) {
    char c;
    while ((c = *line++)) putcharacter(c);
}

void putunsigned (unsigned value) {
    if (value / 10) putunsigned(value / 10);
    putcharacter('0' + (value % 10));
}

void putnumber (int value) {
    if (value < 0) {
        putcharacter('-');
        value = -value;
    }
    putunsigned(value);
}
```

Where the Shoe Pinches(contd.) foo.{cc,o}

```
wosch@hawaii 40> g++ -O6 -fno-rtti -fno-exceptions -fno-inline -c foo.cc
wosch@hawaii 41> nm -v foo.o
         U write
00000000 t gcc2_compiled.
00000000 B line
00000000 T resetline__Fv
00000024 T flushline__Fv
0000003c T writeline__Fc
00000040 B slot
00000058 T putchararacter__Fc
00000080 T putstring__FPc
000000a8 T putunsigned__FUi
000000ec T putnumber__Fi
wosch@hawaii 42> size foo.o
   text    data     bss     dec     hex filename
    270       0      68     338     152 foo.o
```

Resolution of symbol putstring__FPc causes not only the (static) binding of function putstring(), and of all other objects/functions (or object modules) directly or indirectly referenced by that source, but also of all unreferenced objects/functions (e.g. putnumber() and putunsigned()) contained in the same object module—and this holds recursively.

Where the Shoe Pinches(contd.) Binder

- the usual case is to consider an entire object module as binding unit
  - not the actually referenced functions/objects of that particular module
  - the consequence is a larger memory footprint → p. 14
- static binding can benefit from the output generated by a compiler
  - many compilers leave a `.size` assembler pseudo-instruction → p. 16
  - many assemblers and/or linkers don't make capital out of this
- dynamic binding seems to be the solution to these problems, if any
  - but at which other costs? — even this (nice) feature is not for free!

Where the Shoe Pinches (contd.)

- the output of `gcc ... -S foo.cc` ?
  - `.size` leaves the actual object size
  - . . . to be kept in the symbol table
  - . . . to be used by the linker
  - . . . to extract `putstring()`
- not every compiler does like this
  - not every assembler notices `.size`
  - not every linker binds selectively
- tools that do not work always properly

```
    .align 4
    .globl putstring__FPc
    .type putstring__FPc,@function
putstring__FPc:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    jmp .L21
    .p2align 4,,7
.L18:
    movsbl %al,%eax
    pushl %eax
    call putchararacter__Fc
    addl $4,%esp
.L21:
    movb (%ebx),%al
    incl %ebx
    testb %al,%al
    jne .L18
    movl -4(%ebp),%ebx
    leave
    ret
.Lfe5:
    .size putstring__FPc,.Lfe5-putstring__FPc
```


Where the Shoe Pinches (contd.) Compiler

- a compile-time option such as "`-fdismember`" would be nice to have
  - to cut a single (monolithic) source module in pieces of compilation units
    * each of which being the source-code representation of a binding unit
  - to create object modules which, likewise, export a single reference only
    * thus to outwit the binder and link only the truly referenced parts
  - to archive the (possibly many) "slim" object modules in a library
- an extensive local data and control flow analysis alone is not enough
  - functions unreachable inside a source module may be reachable from outside
- a global analysis is not always feasible — and wouldn't solve our problem

Where the Shoe Pinches(contd.) — Programming Language

- a problem lies in the overloaded and monolithic interface of `printf(3)`
  - an actual parameter decides upon which of the many operations to perform
  - a format-string interpreter fetches instructions and reads their operands
  - references to functions implementing the instructions are hard encoded
- moreover, `printf(3)` is not a programming-language but a library concept
  - thus, static format-string analysis becomes not the compiler's task
  - consequently, unused functions cannot be eliminated at compile-time
- an integrated programming system looks nice — but is not everybodies darling

Where the Shoe Pinches(contd.) — `<iostream.h>`

```
#include <iostream.h>

void main () {
    cout << "Hello world!" << endl;
}
```

- `ostream` provides specialized operations
  - by overloading operator "<<"
- users see a function set to choose from
- "`ostream& operator << (const char *s)`" and "`endl`" is all one needs
  - other operators are not used, not referenced and, hence, need not be linked
  - the question is whether or not `ostream` makes up a single compilation unit
- at a first glance object-orientation or C++ is really the right way to go

Where the Shoe Pinches(contd.) class ostream

```
wosch@hawaii 37> uname -snrm
Linux hawaii.cs.uni-magdeburg.de 2.2.14 i686
wosch@hawaii 38> vi hello.cc
wosch@hawaii 39> g++ -O6 -c hello.cc; g++ -static -o hello hello.o
wosch@hawaii 40> hello
Hello world!
wosch@hawaii 41> ls -l hello*
-rwxr-xr-x   1 wosch    ivs       1403951 Mar  9 16:13 hello*
-rw-r--r--   1 wosch    ivs            73 Mar  9 16:09 hello.cc
-rw-r--r--   1 wosch    ivs          1248 Mar  9 16:13 hello.o
wosch@hawaii 42> size hello hello.o
   text    data     bss     dec     hex filename
 300921   19564    3812  324297   4f2c9 hello
     44      52       0      96      60 hello.o
```


An Omnipresent Problem

- the `printf(3)` and `iostream.h` examples are no exceptions
  - existing software-development tools leave much to be desired
  - nonetheless is "user-friendly" software exceedingly required
- a highly modular and application-oriented software structure is needed
  - beginning at the "drawing-board" where the software design takes place
  - ending on the spot where the software implementation is been carried out
- certain software-engineering principles must be applied (more) consequently

That's the State of the Art

- operating systems provide nice features for optimized storage management

  virtual memory enables the execution of uncomplete programs, i.e. the programs must not necessarily be entirely present in main memory to be executed. A program's memory footprint varies with the size of the process's working set (of pages).

  shared libraries work similar. In addition, their use leads to a significant reduction of the disk-memory space occupied by the executable programs.

- but these are only good for higher-level (system) software abstractions
  - what's about the virtual-memory or shared-library system itself **?**
  - what's about device driver, process management, scheduling etc. **?**

- lower-level (system) software cannot always make a profit from it **!**

An Operating-System Problem?

- no . . .
  - it's a general problem concerning all kinds of software :-(

- . . . but operating systems are highly sensitive to software deficiencies
  - they generally are of functionally high <u>complexity</u>
  - they come up with a <u>durability</u> that lies in the order of decades
  - they are often expected to show a broad <u>applicability</u>

- operating systems are key technology — in the past, present, and future

Preventive Measures

- the design and development of an operating system as a program family
  - establishment of a many-layered functional hierarchy
  - creation of a manufacture of "standardized" (reusable) units
  - construction of a rich set of small and simple modules
- a careful use of object orientation in the implementation process
  - composition of abstract data types by means of classes
  - use of inheritance to promote the functional enrichment of the system
  - thriftiness in the employment of late-binding
- *understand application programs as final operating-system specializations*

Application-Oriented Operating Systems

Some users may require only a subset of services or features that other users need. These **'less demanding' users** may demand that they are not be forced to pay for the resources consumed by the unneeded features. [3]

Bibliography

[1] B. H. Liskov and S. Zilles. Programming with Abstract Data Types. *SIGPLAN Notices*, 9(4), 1974.

[2] A. M. Lister and R. D. Eager. *Fundamentals of Operating Systems*. The Macmillan Press Ltd., fifth edition, 1993. ISBN 0-333-59848-2.

[3] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, 1979.