The Operating System – A Swiss Army Knife?

Commodity operating systems provide a rich set of features to be prepared for all kinds of applications and contingencies:

- Malicious or erroneous applications
  - preemptive scheduling, address space separation, disk quotas
- Multi-user operation
  - authentication, access validation and auditing
- Multi-threaded and interacting applications
  - Threads, semaphores, pipes, sockets
- Many/large concurrently running applications
  - virtual memory, swapping, working sets

One size fits all? ➔ Variability

“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 be easily identified; a common criticism of many systems is that in attempting to be all things to all men they wind up being totally satisfactory to no one."

Some applications may require only a subset of services or features that other applications need. These ‘less demanding’ applications should not be forced to pay for the resources consumed by unneeded features.

Parnas 1979: “Designing Software for Ease of Extension and Contraction” [8]

### Variability and Granularity

**Variability**

Variability of system software is the property that denotes the range of functional requirements that can be fulfilled by it.

**Granularity**

Granularity of system software is the property that denotes the resolution of which requirements can be fulfilled by it, in the sense that requirements are fulfilled but not overfulfilled.

- Can general purpose (GP) systems fulfill these demands?
- Reality check – a small study with `printf()` from glibc:
  (Analogy: GP operating system \(
  \rightarrow \)
  GP library \(
  \rightarrow \)
  GP function)

```c
int main()
{
    printf("Hello World\n");
}
```

### Reality Check: Granularity

**The setup:**

```bash
> uname -a
Linux faui48a 2.6.32-5-amd64 #1 SMP Mon Oct 3 05:45:56 UTC 2011 x86_64 GNU/Linux
> gcc -dumpversion
4.4.5
```

**Experiment 1: `printf()`**

```bash
> echo 'main(){printf("Hello World\n");}' | gcc -xc - -w -Os -static -o hello1
> ./hello1
Hello World
> size hello1
```

- Maybe the general-purpose `printf()` is just to powerful?
  - supports many data types, formatting rules, ...
  - implementation requires a complex parser for the format string
- Let’s try the much more specialized `puts()`!
Reality Check: Granularity (Cont'd)

Experiment 2: puts()
> echo 'main(){puts("Hello World");}' | gcc -xc -Os -w -static -o hello2
> ./hello2
Hello World
> size hello2
<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>508723</td>
<td>1928</td>
<td>7852</td>
<td>517703</td>
<td>7e647 hello2</td>
<td></td>
</tr>
</tbody>
</table>

That didn’t help much!
Maybe puts() is yet too poweful?
- buffered IO, streams
- Let’s work directly with the OS file handle!

Experiment 3: write()
> echo 'main(){write(1, "Hello World\n", 13);}' | gcc -xc -Os -w -static -o hello3
> ./hello3
Hello World
> size hello3
<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>508138</td>
<td>1928</td>
<td>7052</td>
<td>517118</td>
<td>7e3fe hello3</td>
<td></td>
</tr>
</tbody>
</table>

517703 compared to 517118 – a net saving of 585 bytes (0.1%) :-(

Experiment 4: empty program
> echo 'main(){}' | gcc -xc -Os -w -static -o hello4
> size hello4
<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>508074</td>
<td>1928</td>
<td>7052</td>
<td>517054</td>
<td>7e3be hello4</td>
<td></td>
</tr>
</tbody>
</table>

objdump -D --reloc hello4 | grep printf | wc -l
yields still 2611 matches!
- It’s the startup code!

Experiment 5: write(), no startup code
> echo '_start(){write(1, "Hello World\n", 13);_exit(0);}' | gcc -xc -Os -w -static -nostartfiles -o hello5
> ./hello5
Segmentation fault

0.5 KiB :-|
but segfault :-(

Experiment 6: SYS_write()
> echo '_start(){syscall(4, 1, "Hello World\n", 13);_exit(0);}' | gcc -xc -Os -w -static -nostartfiles -o hello6
> ./hello6
Hello World
> size hello6
<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>293</td>
<td>0</td>
<td>4</td>
<td>297</td>
<td>129 hello6</td>
<td></td>
</tr>
</tbody>
</table>

297 ←→ 517703 Bytes!
On Linux/glibc, a simple “Hello World” application takes 1750 times more memory than necessary!

However, is this a problem?
- The glibc has been designed for a “standard case”
  - Large, multithreaded, IO-intensive UNIX application
  - Assumption: every program uses malloc(), printf(),...

- Variability has been traded for Granularity

Variability has been traded for Granularity

Parnas 1979: “Designing Software for Ease of Extension and Contraction” [8]
Reality Check: Lessons Learned

297 ←→ 517703 Bytes!
On Linux/glibc, a simple "Hello World" application takes 1750 times more memory than necessary!

However, is this a problem?
- The glibc has been designed for a "standard case" - Large, multithreaded, IO-intensive UNIX application
- Assumption: every program uses malloc(), printf(), ...
- Variability has been traded for Granularity

Assumption: The GP operating system will compensate for it...
- Virtual memory → memory is not an issue (but is that a reason to waste it?)
- Shared libraries → memory is actually shared between processes (unless we relocate the symbols, e.g., for address-space randomization...)

What about other domains?

A Different Domain: Embedded Systems

CPU/DSP sales in 2002 [13]

The ATmega μC Family (8-Bit)

<table>
<thead>
<tr>
<th>Type</th>
<th>Flash (KB)</th>
<th>SRAM (B)</th>
<th>IO</th>
<th>Timer 8/16</th>
<th>UART</th>
<th>Timer</th>
<th>ATC</th>
<th>AD</th>
<th>Price (£)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTINY11</td>
<td>1</td>
<td>6</td>
<td>1-/-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.31</td>
</tr>
<tr>
<td>ATTINY13</td>
<td>1</td>
<td>6 B</td>
<td>1-/-</td>
<td>-</td>
<td>-</td>
<td>4*10</td>
<td></td>
<td>0.66</td>
<td></td>
</tr>
<tr>
<td>ATTINY2313</td>
<td>2</td>
<td>128 B</td>
<td>1/1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1.06</td>
<td></td>
</tr>
<tr>
<td>ATMega4820</td>
<td>4</td>
<td>512 B</td>
<td>2/1</td>
<td>2</td>
<td>1</td>
<td>6*10</td>
<td></td>
<td>1.26</td>
<td></td>
</tr>
<tr>
<td>ATMega8515</td>
<td>8</td>
<td>512 B</td>
<td>35</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>2.04</td>
<td></td>
</tr>
<tr>
<td>ATMega8535</td>
<td>8</td>
<td>512 B</td>
<td>32</td>
<td>2/1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>2.67</td>
<td></td>
</tr>
<tr>
<td>ATMega169</td>
<td>16</td>
<td>1024 B</td>
<td>54</td>
<td>1</td>
<td>1</td>
<td>8*10</td>
<td></td>
<td>4.03</td>
<td></td>
</tr>
<tr>
<td>ATMega64</td>
<td>64</td>
<td>4096 B</td>
<td>53</td>
<td>2/2</td>
<td>2</td>
<td>8*10</td>
<td></td>
<td>5.60</td>
<td></td>
</tr>
<tr>
<td>ATMega128</td>
<td>128</td>
<td>4096 B</td>
<td>53</td>
<td>2/2</td>
<td>2</td>
<td>8*10</td>
<td></td>
<td>7.91</td>
<td></td>
</tr>
</tbody>
</table>

Bulk prices and features of ATmega variants (excerpt, DigiKey 2006)

Limited Resources
- Flash is limited, RAM is extremely limited
- A few bytes can have a massive impact on per-unit costs
- The "glibc approach" is doomed to fail!
The Role of the Operating System

Operating systems (not) employed in embedded-system projects in 2006 [12]

> 40% of all projects use “in house” OS functionality!

Wide-spread fear of the resource overhead of GP operating systems
- OS functionality is developed “side-by-side” with the applications
- This leads to very high “hidden” development costs [14]

Operating systems (not) employed in embedded-system projects in 2006 [12]

Between a Rock and a Hard Place...

The “glibc approach” (one size fits all) does not work!

functional and nonfunctional requirements

Hardware

ISA

tasks

sockets

file system

file system

event latency

safety

MMU / MPU

cache size

coherence

IRQ latency
High variety of functional and nonfunctional application requirements
High variety of hardware platforms
High per-unit cost pressure

System software has to be tailored for each concrete application

functional and nonfunctional properties

Between a Rock and a Hard Place...

System Software
- functional and nonfunctional requirements
- tasks
- sockets
- file system
- ... event latency
- ... safety
- ISA
- IRQ handling
- MMU / MPU
- ... cache size
- coherence
- IRQ latency
- ...

Functional and nonfunctional properties

High variety of functional and nonfunctional application requirements
High variety of hardware platforms
High per-unit cost pressure

System software has to be tailored for each concrete application

Between a Rock and a Hard Place...

Customizing / Tailoring

Customizing or tailoring is the activity of modifying existing system software in order to fulfill the requirements of some particular application.

This calls for granularity and variability!

Agenda

1.1 Commodity Operating Systems Today
1.2 Reality Check: Granularity
1.3 The Domain of Embedded Systems
1.4 About KSS
1.5 KSS — Organization
1.6 References
What to do?

297 ←→ 517703 Bytes! Why?
On Linux/glibc, a simple "Hello World" application takes 1750 times more memory than necessary!

- Reason: software structure
  - Trade-off between reuse ⇐⇒ coupling
    ~ by extensive internal reuse, glibc has become an all-or-nothing blob
- Reason: software interface
  - C standard defines printf() as a swiss army knife [3, §7.19.6]
    ~ printf() has become a "god method" [1]
- Reason: language and tool chain
  - Compiler/linker work on the granularity of symbols or even object files
    ~ dead code is not effectively eliminated

Individually Developed Software Product

Software Product Derived from Reusable Assets

~ Konfigurierbare Systemsoftware – KSS
Throughout the software development cycle, variability and granularity have to be considered as primary design goals from the very beginning!
In KSS you will learn about principles, methods, and tools to achieve this.
Learning Objectives

- **Improve** your understanding of the design and development of low-level system software
  - Starting point: “Betriebssysteme” [BS]
  - Focus: Static configuration and tailoring

- **Expand** your knowledge by new software engineering methods and language techniques for configurable system software
  - Software families and software product lines [7]
  - Aspect-oriented and generative programming in C/C++ [10]

- **Apply** these techniques in the context of current operating-system research projects
  - CiAO, SLOTH, VAMOS, DanceOS [2, 5, 9, 11]
  - Get prepared for a master thesis or project in the field!
Topics
Konfigurierbare Systemsoftware
Software engineering
Variability
Product Lines
Aspects
Generators
eCos [6]
CiAO [5]
Sloth [2]
Linux [11]
Operating systems

KSS – Einordnung
(Bachelor/Master)

Organisation: Systemsoftwaretechnik (SST)
- Modul Systemsoftwaretechnik (SST) 7.5 ECTS
  ① Vorlesung Betriebssystemtechnik (BST) 2.5
    - Mo 12 – 14
    - 12–14 Vorlesungstermine
  ② Übungen zu Betriebssystemtechnik (BST-Ü) 2.5
    - Di 10 – 12
    - 12–14 Übungstermine/Rechnerübungen
  ③ Vorlesung und Übung Konfigurierbare Systemsoftware (KSS) 2.5
    - Do 14 – 16 (Vorlesung)
    - 7 Vorlesungstermine, 1 Übungsaufgabe, 1 Projekt
    - Übung integriert in BST-Übung / Rechnerübung

  ~ KSS kann nur zusammen mit BST belegt werden!
  - Es gibt keine 2.5 ECTS Module...
  - Wenn Bedarf besteht, wird KSS auf 5 ECTS erweitert

Organisation: Beteiligte
- Vorlesung
  Daniel Lohmann

- Übung
  Daniel Danner
  Gabor Drescher

- Projekt
  Daniel Danner
  Martin Hoffmann
  Jens Schedel
  ?
Semesterplanung

Referenzen (Cont’d)


