Konfigurierbare Systemsoftware (KSS)

VL 1 – Einführung

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 16 – 2016-04-11

http://www4.informatik.uni-erlangen.de/Lehre/SS16/V_KSS
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
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
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.

Big is beautiful?

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 (Definition 1)

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

Granularity (Definition 2)

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 $\leftrightarrow$ GP library $\leftrightarrow$ GP function)

```c
int main() {
    printf( "Hello World\n");
}
```
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
Reality Check: Granularity

The setup:

```sh
> 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()`

```sh
> echo 'main(){printf("Hello World\n");}’ | gcc -xc -w -Os -static -o hello1
> ./hello1
Hello World
> size hello1
 text  data  bss  dec  hex  filename
508723  1928  7052  517703  7e647 hello1
```

Maybe the general-purpose `printf()` is just too powerful?
- supports many data types, formatting rules, ...
- implementation requires a complex parser for the format string

Let’s try the much more specialized `puts()`!
Experiment 2: `puts()`

```bash
> echo 'main(){puts("Hello World");}'; | gcc -xc - -Os -w -static -o hello2
> ./hello2
Hello World
> size hello2
  text  data  bss  dec  hex  filename
  508723  1928  7052 517703  7e647 hello2
```

- That didn’t help much!
- Maybe `puts()` is yet too powerful?
  - buffered IO, streams
- Let’s work directly with the OS file handle!
Experiment 3: write()

```bash
> echo 'main(){write(1, "Hello World\n", 13);}'} | gcc -xc - -Os -w -static -o hello3
> ./hello3
Hello World
```

512 KiB!

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

Experiment 4: empty program

```bash
> echo 'main(){}' | gcc -xc - -Os -w -static -o hello4
> size hello4
```

Hm...

objdump -D --reloc hello4 | grep printf | wc -l
yields still 2611 matches!

It’s the startup code!
Reality Check: Granularity (Cont’d)

Experiment 5: `write()`, no startup code

```
> echo ‘_start(){write(1, "Hello World\n", 13);_exit(0);}’ | gcc -xc - -Os -w
-static -nostartfiles -o hello5
> size hello5
   text  data  bss  dec  hex  filename
      597     0     4  601  259 hello5
> ./hello5
Segmentation fault
```

- Even a simple `write()` cannot be issued without the complete initialization.
- Last ressort: invoke the syscall directly!

Experiment 6: `SYS_write()`

```
> echo ‘_start(){syscall(4, 1, "Hello World\n", 13);_exit(0);}’ | gcc -xc - -Os -w
-static -nostartfiles -o hello6
> size hello6
   text  data  bss  dec  hex  filename
      293     0     4  297  129 hello6
> ./hello6
Hello World
```

0.5 KiB :-|
but segfault :-(

0.25 KiB :-|
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

---

**Every Program?**

“I know of no feature that is always needed. When we say that two functions are almost always used together, we should remember that “almost” is a euphemism for “not”.”

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

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?
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
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</th>
<th>SRAM</th>
<th>IO</th>
<th>Timer 8/16</th>
<th>UART</th>
<th>SPI</th>
<th>ADC</th>
<th>PWM</th>
<th>EUR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTINY13</td>
<td>1 KiB</td>
<td>64 B</td>
<td>6</td>
<td>1/-</td>
<td>-</td>
<td>-</td>
<td>1*4</td>
<td>-</td>
<td>0.86</td>
</tr>
<tr>
<td>ATTINY2313</td>
<td>2 KiB</td>
<td>128 B</td>
<td>18</td>
<td>1/1</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0.99</td>
</tr>
<tr>
<td>ATMEGA48</td>
<td>4 KiB</td>
<td>512 B</td>
<td>23</td>
<td>2/1</td>
<td>1</td>
<td>1</td>
<td>8*10</td>
<td>6</td>
<td>1.40</td>
</tr>
<tr>
<td>ATMEGA16</td>
<td>16 KiB</td>
<td>1024 B</td>
<td>32</td>
<td>2/1</td>
<td>1</td>
<td>1</td>
<td>8*10</td>
<td>4</td>
<td>2.05</td>
</tr>
<tr>
<td>ATMEGA32</td>
<td>32 KiB</td>
<td>2048 B</td>
<td>32</td>
<td>2/1</td>
<td>1</td>
<td>1</td>
<td>8*10</td>
<td>4</td>
<td>3.65</td>
</tr>
<tr>
<td>ATMEGA64</td>
<td>64 KiB</td>
<td>4096 B</td>
<td>53</td>
<td>2/2</td>
<td>2</td>
<td>1</td>
<td>8*10</td>
<td>8</td>
<td>5.70</td>
</tr>
<tr>
<td>ATMEGA128</td>
<td>128 KiB</td>
<td>4096 B</td>
<td>53</td>
<td>2/2</td>
<td>2</td>
<td>1</td>
<td>8*10</td>
<td>8</td>
<td>7.35</td>
</tr>
<tr>
<td>ATMEGA256</td>
<td>256 KiB</td>
<td>8192 B</td>
<td>86</td>
<td>2/2</td>
<td>4</td>
<td>1</td>
<td>16*10</td>
<td>16</td>
<td>8.99</td>
</tr>
</tbody>
</table>

Prices and features of ATmega variants (excerpt, Reichelt April 2015)

**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

(a) Types of operating systems

- Commercial: 36.30%
- Open Source (commercially distributed): 11.50%
- Open Source (developed internally): 15.10%
- Proprietary: 28.70%
- No: 8.40%

(b) Why no operating system?

- Too complicated: 7%
- Too expensive: 10%
- Resource concerns: 30%
- Do not need one: 85%

Operating systems (not) employed in embedded-system projects in 2006 [12]
The Role of the Operating System

(a) Types of operating systems 
(n = 1200)

- commercial: 36.30%
- no: 28.70%
- proprietary (developed internally): 15.10%
- open source (commercially distributed): 11.50%
- open source

(b) Why no operating system? 
(Multiple answers possible)

- too complicated: 7%
- too expensive: 10%
- resource concerns: 30%
- do not need one: 85%

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]
The Role of the Operating System

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

Rest spreads over **hundreds of different** operating systems!

..., C{51, 166, 251}, CiAO, CMX RTOS, Contiki, C-Smart/Raven, eCos, eRTOS, Embos, Ercos, Euros Plus, FreeRTOS, Hi Ross, Hynet-OS, LynxOS, MicroX/OS-II, Nucleus, OS-9, OSE, OSEK {Flex, Turbo, Plus}, OSEKtime, Precise/MQX, Precise/RTCS, proOSEK, pSOS, PURE, PXROS, QNX, Realos, RTMOSxx, Real Time Architect, RTA, RTX{51, 166, 251}, RTXC, Softune, SSXS RTOS, ThreadX, TinyOS, Tresos, VRTX, VxWorks, ...

〜 The “glibc approach” (one size fits all) **does not work**!
Between a Rock and a Hard Place...

**functional and nonfunctional requirements**

**Application**

**System Software**

**Hardware**

**functional and nonfunctional properties**

- tasks
- sockets
- file system
- ... event latency
- safety
- ...

- ISA
- IRQ handling
- MMU / MPU
- ...
- cache size
- coherence
- IRQ latency
- ...
Between a Rock and a Hard Place...

- 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 requirements

- functional and nonfunctional properties

- tasks
  - sockets
  - file system
- ... event latency safety ...
- ISA
  - IRQ handling
  - MMU / MPU ...
  - cache size coherence
  - IRQ latency ...

© dl KSS (VL 1 | SS 16) 1 Motivation | 1.3 The Domain of Embedded Systems
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**!
Between a Rock and a Hard Place...

- 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
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
297 \leftrightarrow 517703 \text{ Bytes!} \quad \textbf{Why?}

On Linux/glibc, a simple “Hello World” application takes \textit{1750 times} more memory than necessary!

\begin{itemize}
  \item Reason: software structure
    \begin{itemize}
      \item Trade-off between \textit{reuse} \leftrightarrow \textit{coupling}
        \begin{itemize}
          \item by extensive internal reuse, glibc has become an all-or-nothing blob
        \end{itemize}
    \end{itemize}
  \item Reason: software interface
    \begin{itemize}
      \item C standard defines \textit{printf()} as a swiss army knife \cite[§7.19.6]{3}
        \begin{itemize}
          \item \textit{printf()} has become a “god method” \cite{1}
        \end{itemize}
    \end{itemize}
  \item Reason: language and tool chain
    \begin{itemize}
      \item Compiler/linker work on the granularity of symbols or even object files
        \begin{itemize}
          \item dead code is not effectively eliminated
        \end{itemize}
    \end{itemize}
\end{itemize}
What to do?

297 ←→ 517703 Bytes!

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

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.
Individually Developed Software Product

System Developer

Specific Problem

\[ f_1, f_2, \ldots \]

intended properties

Variant

System User

Specific Solution

\[ A \rightarrow B \rightarrow C \rightarrow D \]

actual implementation

intentional side

extensional side

© dl KSS (VL 1 | SS 16) 1 Motivation | 1.4 About KSS
Software Product Derived from Reusable Assets

Solution Space

Architect / Developer

Aspect

Class

Architecture and Implementation

System User

Variant

System Developer

Specific Problem

Specific Solution

intentional side

extensional side

© dl  KSS (VL 1 | SS 16)  1 Motivation | 1.4 About KSS
Configurable Software – Software Product Line

- Problem Space
  - Domain Expert
  - Features and Dependencies
    - $f_1$
    - $f_2$
    - $f_3$
    - $f_4$
    - $f_5$
    - $f_6$
    - $f_7$

- Solution Space
  - Architect / Developer
  - Aspect
  - Class
  - Architecture and Implementation

- System User
  - Configuration
    - $f_1$
    - $f_2$
    - ... (intended properties)

- Specific Problem

- Specific Solution

- KSS
  - intentional side
  - extensional side

- Tools
  - Principles
  - Methods
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
   - Objectives
   - Einordnung
   - Semesterplanung
1.6 References
Learning Objectives

- **Improve** your understanding of the design and development of low-level system software
  - Starting point: “Betriebssysteme”
  - 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
  - Aspect-oriented and generative programming in C/C++

- **Apply** these techniques in the context of current operating-system research projects
  - CiAO, SLOTH, VAMOS, DanceOS
  - Get prepared for a master thesis or project in the field!
Konfigurierbare Systemsoftware

Software engineering

Variability
Product Lines
Aspects
Generators
eCos [6]
CiAO [5]
Sloth [2]
Linux [11]

Operating systems
KSS – Einordnung

(Klassifizierung Bachelor/Master)

Systemprogrammierung
10 ECTS

3

Systemsoftwaretechnik

4

EZZ 5–7,5

5

BS 5–7,5

EZS2 P: 10

6

BST 5

PASST P: 10

SST 7,5

KSS 2,5

VS 5–7,5

© dl KSS (VL 1 | SS 16) 1 Motivation | 1.5 KSS – Organization 1-26
Modul Systemsoftwaretechnik (SST) 7.5 ECTS

1 Vorlesung Betriebssystemtechnik (BST) 2.5
   - Mo 14–16
   - 12–14 Vorlesungstermine

2 Übungen zu Betriebssystemtechnik (BST-Ü) 2.5
   - Mi 10–12
   - 10 Übungstermine

3 Vorlesung und Übung Konfigurierbare Systemsoftware (KSS) 2.5
   - Mo 12–14
   - 7 Vorlesungstermine, 1–2 Übungsaufgaben
   - 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

Christian Dietrich
Valentin Rothberg
<table>
<thead>
<tr>
<th>KW</th>
<th>Mo</th>
<th>Di</th>
<th>Mi</th>
<th>Do</th>
<th>Fr</th>
<th>Themen</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>20.04</td>
<td>21.04</td>
<td>22.04</td>
<td>23.04</td>
<td>24.04</td>
<td>BST VL2: Systemaufruf</td>
</tr>
<tr>
<td>18</td>
<td>27.04</td>
<td>28.04</td>
<td>29.04</td>
<td>30.04</td>
<td>01.05</td>
<td>KSS VL2: Software Families and Software Product Lines</td>
</tr>
<tr>
<td>19</td>
<td>04.05</td>
<td>05.05</td>
<td>06.05</td>
<td>07.05</td>
<td>08.05</td>
<td>KSS VL4: Aspect-Aware Design, CIAO BST VL4: Hierarchien</td>
</tr>
<tr>
<td>20</td>
<td>11.05</td>
<td>12.05</td>
<td>13.05</td>
<td>14.05</td>
<td>15.05</td>
<td>KSS VL5: Variability in the Large, VAMOS BST VL5: Generative Programming, Sloth</td>
</tr>
<tr>
<td>21</td>
<td>18.05</td>
<td>19.05</td>
<td>20.05</td>
<td>21.05</td>
<td>22.05</td>
<td>KSS VL7: Conclusion, Summary BST VL6: Adressraummodelle</td>
</tr>
<tr>
<td>22</td>
<td>25.05</td>
<td>26.05</td>
<td>27.05</td>
<td>28.05</td>
<td>29.05</td>
<td>BST VL7: Sprachbasierung</td>
</tr>
<tr>
<td>23</td>
<td>01.06</td>
<td>02.06</td>
<td>03.06</td>
<td>04.06</td>
<td>05.06</td>
<td>BST VL8: Interprozesskommunikation</td>
</tr>
<tr>
<td>24</td>
<td>08.06</td>
<td>09.06</td>
<td>10.06</td>
<td>11.06</td>
<td>12.06</td>
<td>BST VL9: Kommunikationsabstraktionen</td>
</tr>
<tr>
<td>25</td>
<td>15.06</td>
<td>16.06</td>
<td>17.06</td>
<td>18.06</td>
<td>19.06</td>
<td>BST VL10: Mitbenutzung</td>
</tr>
<tr>
<td>26</td>
<td>22.06</td>
<td>23.06</td>
<td>24.06</td>
<td>25.06</td>
<td>26.06</td>
<td>BST VL11: Bindelader</td>
</tr>
<tr>
<td>27</td>
<td>29.06</td>
<td>30.06</td>
<td>01.07</td>
<td>02.07</td>
<td>03.07</td>
<td>BST VL12: Nachlese</td>
</tr>
<tr>
<td>28</td>
<td>06.07</td>
<td>07.07</td>
<td>08.07</td>
<td>09.07</td>
<td>10.07</td>
<td>Abgabe BST A3</td>
</tr>
<tr>
<td>29</td>
<td>13.07</td>
<td>14.07</td>
<td>15.07</td>
<td>16.07</td>
<td>17.07</td>
<td>Abgabe BST A3</td>
</tr>
</tbody>
</table>

Abgabe BST A1: Himmelf. KSS VL4, BST VL4, TÜ BST A1
Abgabe BST A2: Abgabe BST A2
Abgabe BST A3: Pfingsten/Berg
Abgabe BST A2: Anstich
Abgabe BST A3: Pfingsten/Berg
Abgabe BST A2: Pfingsten/Berg

http://www4.informatik.uni-erlangen.de/Lehre/SS16/V_KSS
Referenzen


