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?

"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."

The Operating System – A Swiss Army Knife?

Big is beautiful?  

→ Granularity

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 (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 ←→ GP library ←→ GP function)

```
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:

```
> 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()

```
> echo 'main(){printf("Hello World\n");}' | gcc -x c -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()!
Reality Check: Granularity (Cont'd)

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  7952  517703  2e647 hello2
```

That didn’t help much!
- Maybe puts() is yet too powerful?
- buffered I/O, 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
> size hello3
  text  data  bss  dec  hex  filename
  508138  1928  7952  517118  7e3fe hello3
```

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
  text  data  bss  dec  hex  filename
  508074  1928  7952  517054  7e3be hello4
```

```bash
objdump -D --reloc hello4 | grep printf | wc -l
```

yields still 2611 matches!
- It’s the startup code!

Experiment 5: write(), no startup code

```bash
> 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    0  259 hello5
> ./hello5
Segmentation fault
```

0.5 KiB :-|
- but segfault :-(

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

Experiment 6: SYS_write()

```bash
> 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
```

0.25 KiB :-)

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

“...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?

A Different Domain: Embedded Systems

CPU/DSP sales in 2002

<table>
<thead>
<tr>
<th>Type</th>
<th>Flash</th>
<th>SRAM</th>
<th>I/O</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>1/1</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 cost
  - 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]

> Rest spreads over hundreds of different operating systems!

- C (C1, 106, 231), Ciao, CMX RTOS, Contiki, C-Smart/Flash, eCos, RTOS, Emboss, Evers, Euca, Fran, FreeRTOS, H: Riso, Hypervisor, LynxOS, Minix/OS-II; Nuclea, OS-9, OSE, OSEK (Foss, Turbo, Flax), OSEK/TFx, Precise/MQX, Precise/RTCS, proOSE, pSOS, PURE, PARCOS, QNX, Realtek, RTAI/RTKII, Real Time Architect, RTX (RTX1, 106, 231), RTXC, Software, SSX5 RTOS, ThreadOS, TinyOS, Tmote, VRTX, VxWorks, ...

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

Between a Rock and a Hard Place...

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

System Software

- functional
- and nonfunctional
- requirements

Hardware

- functional
- and nonfunctional
- properties

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

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

Reason: software structure
- Trade-off between reuse $\iff$ coupling
  $\sim$ by extensive internal reuse, glibc has become an all-or-nothing blob

Reason: software interface
- C standard defines printf() as a swiss army knife $[3, \S7.19.6]$
  $\sim$ printf() has become a "god method" $[1]$

Reason: language and tool chain
- Compiler/linker work on the granularity of symbols or even object files
  $\sim$ dead code is not effectively eliminated

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

Software Product Derived from Reusable Assets
**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  
   - 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!
Organisation: Systemsoftwaretechnik (SST)

- Modul Systemsoftwaretechnik (SST)  
  7.5 ECTS
- Vorlesung Betriebssystemtechnik (BST)  
  2.5
  - Mo 14–16
  - 12–14 Vorlesungstermine
- Übungen zu Betriebssystemtechnik (BST-U)  
  2.5
  - Mi 10–12
  - 10 Übungstermine
- 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
Referenzen (Cont'd)


