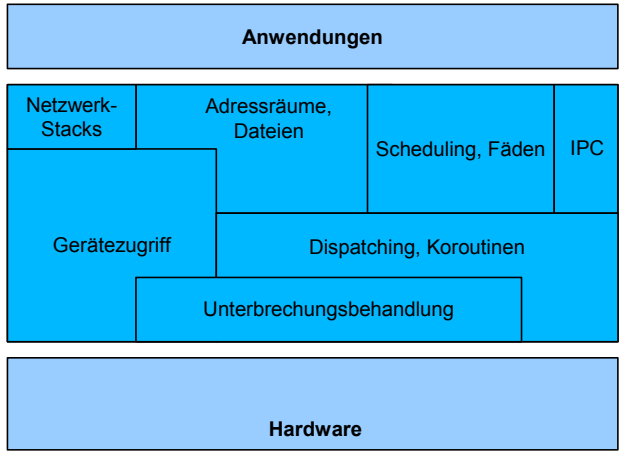
Betriebssysteme (BS)

Zusammenfassung und Ausblick

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Ausgangspunkt: Struktur eines BS



Was wir gemacht haben

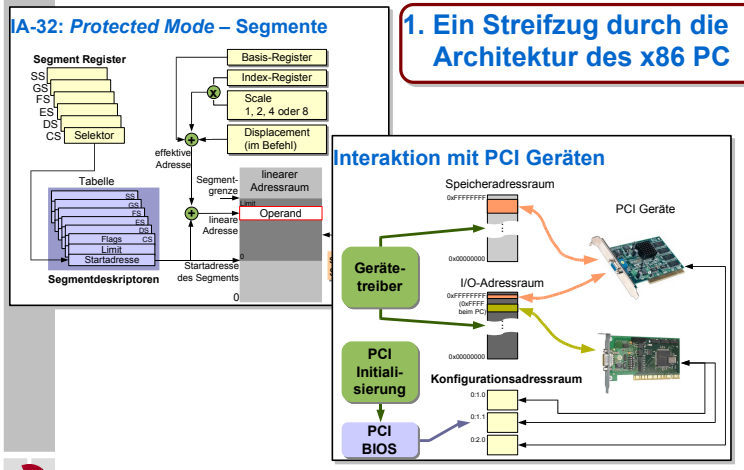
VL 1: Einführung
VL 2: Einstieg in die Betriebssystementwicklung
VL 3: Unterbrechungen - Hardware
VL 4: Unterbrechungen - Software
VL 5: Unterbrechungen - Synchronisation
VL 6: IA-32: Die 32-Bit-Intel-Architektur
VL 7: Koroutinen und Programmfäden
VL 8: Scheduling
VL 9: Betriebssystem-Architekturen
VL 10: Fadensynchronisation
VL 11: Interprozesskommunikation
VL 12: Bussysteme
VL 13: Gerätetreiber

Drei Inhaltliche Schwerpunkte

VL 1: Einführung
VL 2: Einstieg in die Betrie
VL 3: Unterbrechungen - Hardware
VL 4: Unterbrechungen - Software
VL 5: Unterbrechungen - Synchronisation
VL 6: IA-32: Die 32-Bit-Intel-Architektur
VL 7: Koroutinen und Programmfäden
VL 8: Scheduling
VL 9: Betriebssystem-Architekturen
VL 10: Fadensynchronisation
VL 11: Interprozesskommunikation
VL 12: Bussysteme
VL 13: Gerätetreiber

1. Ein Streifzug durch die Architektur des x86 PC

Drei Inhaltliche Schwerpunkte



1. Ein Streifzug durch die Architektur des x86 PC

Drei Inhaltliche Schwerpunkte

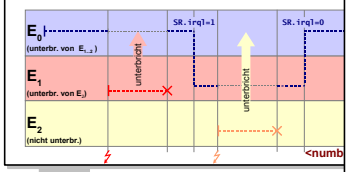
VL 1: Einführung
VL 2: Einstieg in die Betriebssy
VL 3: Unterbrechungen - Hardware
VL 4: Unterbrechungen - Software
VL 5: Unterbrechungen - Synchronisation
VL 6: IA-32: Die 32-Bit-Intel-Architektur
VL 7: Koroutinen und Programmfäden
VL 8: Scheduling
VL 9: Betriebssystem-Architekturen
VL 10: Fadensynchronisation
VL 11: Interprozesskommunikation
VL 12: Bussysteme
VL 13: Gerätetreiber

2. Kontrollflüsse und ihre Interaktionen

Drei Inhaltliche Schwerpunkte

Kontrollflussebenenmodell

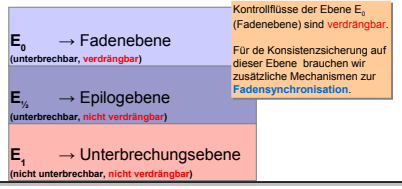
- Verallgemeinerung für mehrere Unterbrechungsebenen:
  - Kontrollflüsse auf E_i werden
    1. **jederzeit unterbrochen** durch Kontrollflüsse von E_m (für $m > l$)
    2. **nie unterbrochen** durch Kontrollflüsse von E_k (für $k \leq l$)
    3. **sequentialisiert** mit weiteren Kontrollflüssen von E_i
  - Kontrollflüsse können die Ebene **wechseln**
    - durch spezielle Operationen (hier: Modifizieren des Statusr



Erweitertes Kontrollflussebenenmodell

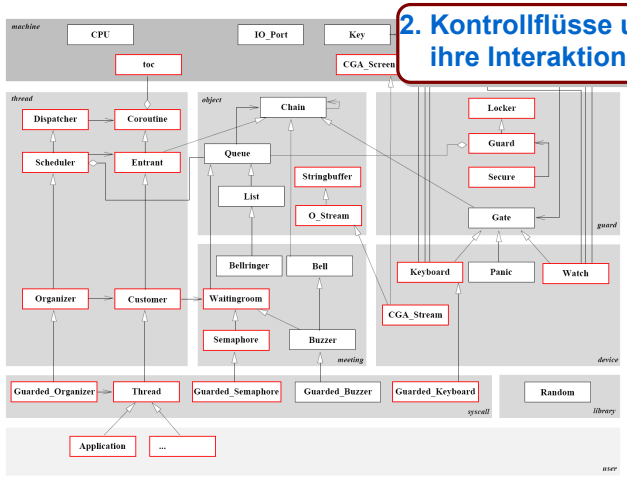
Kontrollflüsse auf Ebene E_l sind

1. **jederzeit unterbrechbar** durch Kontrollflüsse von E_m (für $m > l$)
2. **nie unterbrechbar** durch Kontrollflüsse von E_k (für $k \leq l$)
3. **jederzeit verdrängbar** durch Kontrollflüsse von E_i (für $l = 0$)



2. Kontrollflüsse und ihre Interaktionen

Drei Inhaltliche Schwerpunkte



2. Kontrollflüsse und ihre Interaktionen

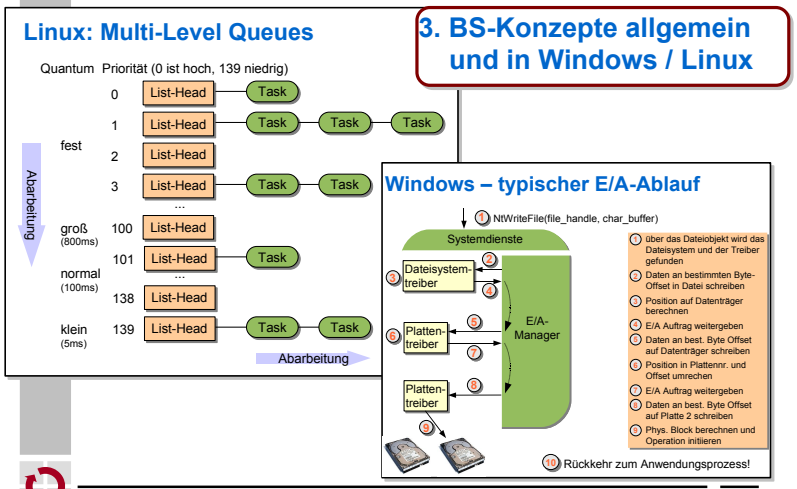
Drei Inhaltliche Schwerpunkte

3. BS-Konzepte allgemein und in Windows / Linux

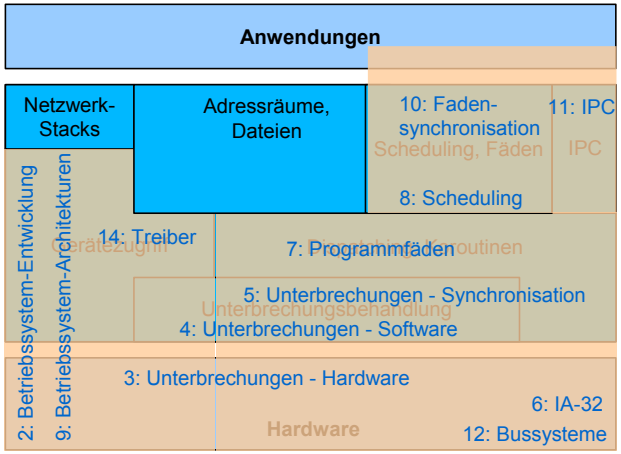
VL 1: Einführung
VL 2: Einstieg in die Betrie
VL 3: Unterbrechungen - Hardware
VL 4: Unterbrechungen - Software
VL 5: Unterbrechungen - Synchronisation
VL 6: IA-32: Die 32-Bit-Intel-Architektur
VL 7: Koroutinen und Programmfäden
VL 8: Scheduling
VL 9: Betriebssystem-Architekturen
VL 10: Fadensynchronisation
VL 11: Interprozesskommunikation
VL 12: Bussysteme
VL 13: Gerätetreiber

Drei Inhaltliche Schwerpunkte

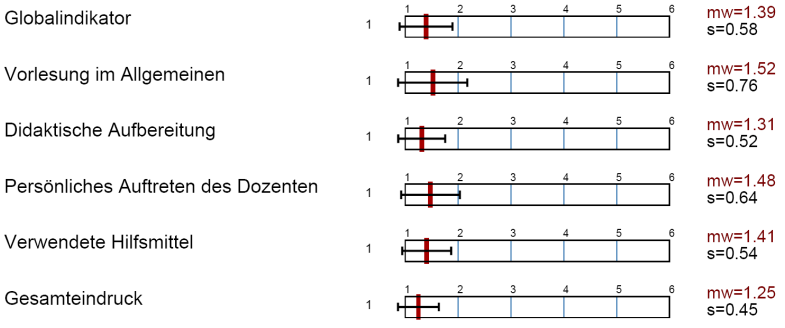
3. BS-Konzepte allgemein und in Windows / Linux



Zusammen eine ganze Menge!

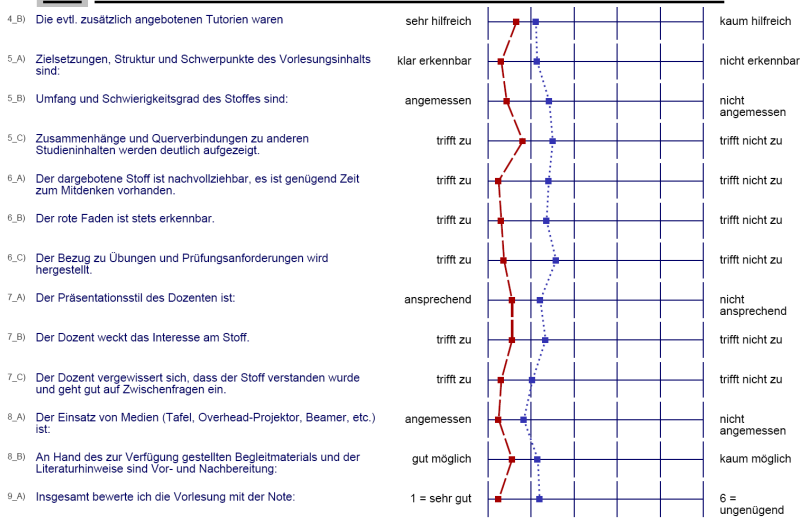


Evaluationsergebnisse: VL-BS



n = 16 aus 32 (50%)

Evaluationsergebnisse: VL-BS



Ausblick: Vorlesung 15

Kevin Klues, Stanford University

The TinyOS Operating System: Past, Present, and Future Directions

- Nächsten Mittwoch zur gewohnten Zeit, **jedoch im H4**
- **Teilnahme ist Pflicht! :-)**

Ausblick: Wie geht es danach weiter?

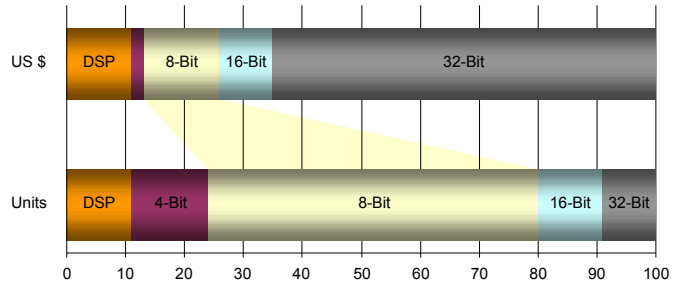
Lehrveranstaltungen des LS4 im nächsten Semester

- Verteilte Systeme
- Echtzeitsysteme 2
- **Betriebssystemtechnik (OSE)**

Eingebettete Systeme sind überall

Wo geht all das Silizium hin?

- eine Statistik aus dem Jahr 2002:



- von den etwa 8 Mrd. Prozessoren werden **mehr als 98%** im Bereich **eingebetteter Systeme** verwendet
- noch heute dominiert **8 Bit Technik**

AVR ATmega – Eine typische Hardware-Produktlinie

µ-Controllerfamilie, basierend auf 8 Bit RISC-Core

- 16 general-purpose Register
- Havard-Architektur, getrennter Programm und Datenspeicher
- On-Board IO-Pins, Timer, AD-Wandler, Bussysteme, ...

Ein kleiner Ausschnitt aus den angebotenen Varianten:

Bezeichnung	Prog.	RAM	IO	Timer 8/16	UART	I²C	AD	Preis
ATTINY11	1024		6	1/-	-	-	-	€0,31
ATTINY13	1024	64	6	1/-	-	-	4*10	€0,66
AT90S2323	2048	128	3	1/-	-	-	-	€1,72
ATMEGA8515	8192	512	35	1/1	1	-	-	€2,04
ATMEGA8535	8192	512	32	2/1	1	1	-	€2,67
ATMEGA169	16384	1024	54	2/1	1	1	8*10	€4,03
ATMEGA64	65536	4096	53	2/2	2	1	8*10	€5,60
ATMEGA128	131072	4096	53	2/2	2	1	8*10	€7,91

[Quelle: Digi-Key Produktkatalog, Sommer 2005]

AVR ATmega – Eine typische Hardware-Produktlinie

µ-Controllerfamilie, basierend auf 8 Bit RISC-Core

- 16 general-purpose Register
- Havard-Architektur, getrennter Programm und Datenspeicher
- On-Board IO-Pins, Timer, AD-Wandler, Bussysteme, ...

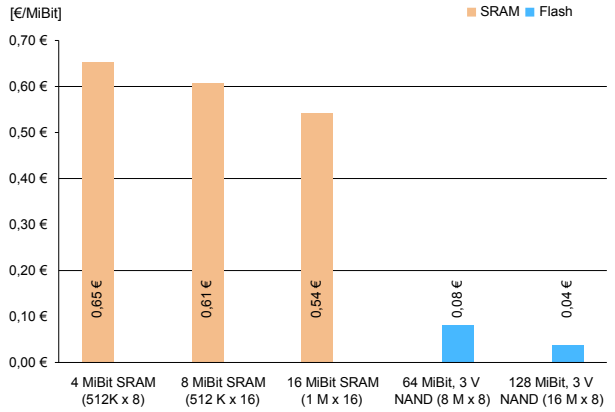
Ein kleiner Ausschnitt aus den angebotenen Varianten:

Bezeichnung	Prog.	RAM	IO	Timer 8/16	UART	I²C	AD	Preis
ATTINY11	1024		6	1/-	-	-	-	€0,31
ATTINY13	1024	64	6	1/-	-	-	4*10	€0,66
AT90S2323								€1,72
ATMEGA8515								€2,04
ATMEGA8535								€2,67
ATMEGA169								€4,03
ATMEGA64								€5,60
ATMEGA128	131072	4096	53	2/2	2	1	8*10	€7,91

Systemsoftware muss ähnlich gut skalieren!

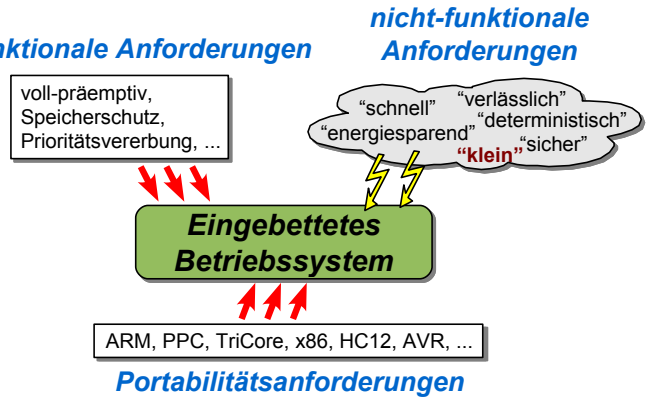
[Quelle: Digi-Key Produktkatalog, Sommer 2005]

Was macht die Hardware teuer?



- SRAM **um den Faktor 10 teurer** als Flash (ROM)

Anforderungen sind vielfältig



- Anforderungen sind hochgradig **anwendungsspezifisch**
  - Vielzweckbetriebssysteme (wie im PC Bereich) versagen hier

Eingebettete Betriebssysteme

. . . , C{51, 166, 251}, CMX RTOS, C-Smart/Raven, eCos, eRTOS, Embos, Ercos, Euros Plus, 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, VRTX, VxWorks, . . .

Markt mit > 100 Systemen
> 50% Eigenentwicklungen

- „das Rad wird neu erfunden“
  - auch die selben Fehler werden wiederholt
- oftmals bietet **ein** BS Hersteller **mehrere** Systeme an
  - mit getrennter Code-Basis
  - getrieben durch die speziellen Anforderungen seiner Kunden

Konfigurierbare Betriebssysteme

- **Ansatz**: feingranulare, anwendungsspezifische, statische Konfigurierung
  - → Ersparnis von Ressourcen
  - → Abdeckung eines breiteren Anwendungsspektrums (=Marktes!)
  - → Wiederverwendung und damit höhere Produktivität

Herausforderderungen

... beim Bau konfigurierbarer Systemsoftware

- Beherrschung der Variantenvielfalt
  - Analyse und Modellierung der Variabilität
- Minimierung der Modulabhängigkeiten, "Plug&Play"
  - Systementwurf
- Wahl geeigneter Sprachmittel für die Programmierung
  - Generizität und Wiederverwendbarkeit vs. Effizienz
- Werkzeugunterstützung
  - Technik zur Konfigurierung
- ...

Herausforderderungen

... beim Bau konfigurierbarer Systemsoftware

- Beherrschung der Variantenvielfalt
  - Analyse und Modellierung der Variabilität
- Mi…
  - S…
- W…ung
  - G…
- Werkzeugunterstützung
  - Technik zur Konfigurierung
- ...

Veranstaltung
Operating System Engineering
im Sommersemester 2008

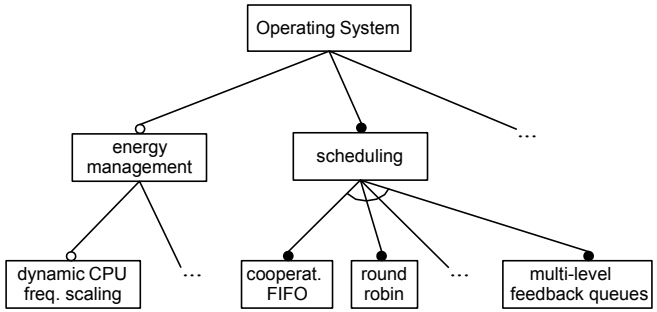
Operating System Engineering (OSE)

aka *Betriebssystemtechnik (BSE)*

- Vorlesung mit Übung (4 SWS, benoteter Schein)
- Inhaltliche Schwerpunkte: Vorlesung
  - Methoden, Techniken, Werkzeuge für die Entwicklung von konfigurierbarer Systemsoftware
  - Merkmalsmodellierung
  - Aspektorientierte Programmierung (AOP)
- Inhaltliche Schwerpunkte: Übung
  - Entwicklungsprojekt: Entwurf einer BS-Produktlinie für RCX / GBA
  - Wie bekommt man Software **klein**?
  - Wie bekommt man Software **konfigurierbar**?

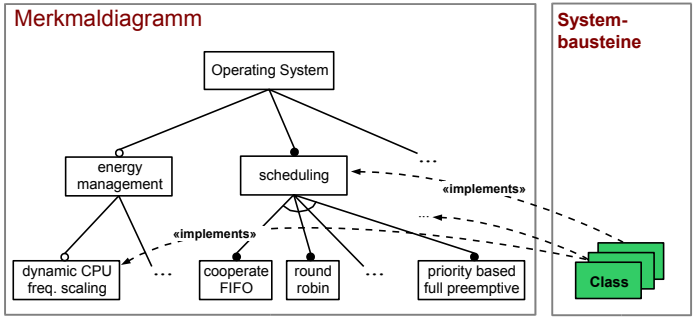
Merkmalmodelle

Merkmaldiagramm einer BS-Produktlinie



Systembausteine

Idealfall: ein Merkmal wird durch
eine **Klasse** implementiert



Problem: Querschneidende Belange

... behindern Konfigurierung und Wiederverwendung

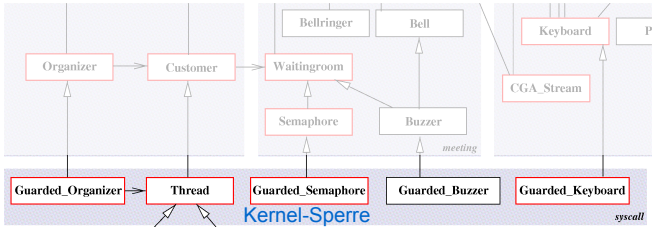
Multi-User Support

Synchronisation

Entwicklungsunterst.

Beispiel aus OO-Stubs: Kernel-Sperre

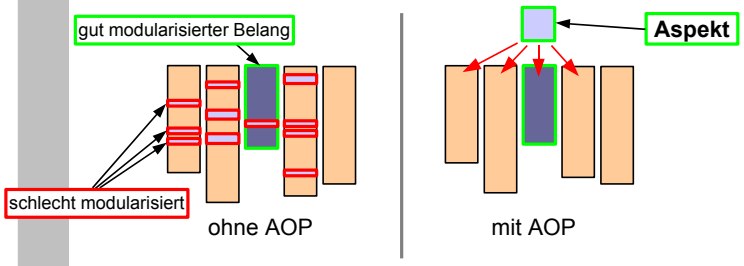
- Pro/Epilogmodell fordert:
  - *guard* sperren, bevor der Kern betreten wird
  - *guard* freigeben, nachdem der Kern verlassen wird
- Ansatz: Eigene Systemschnittstelle **durch Ableitung**



- Das ist gar nicht so doll! (Warum?)

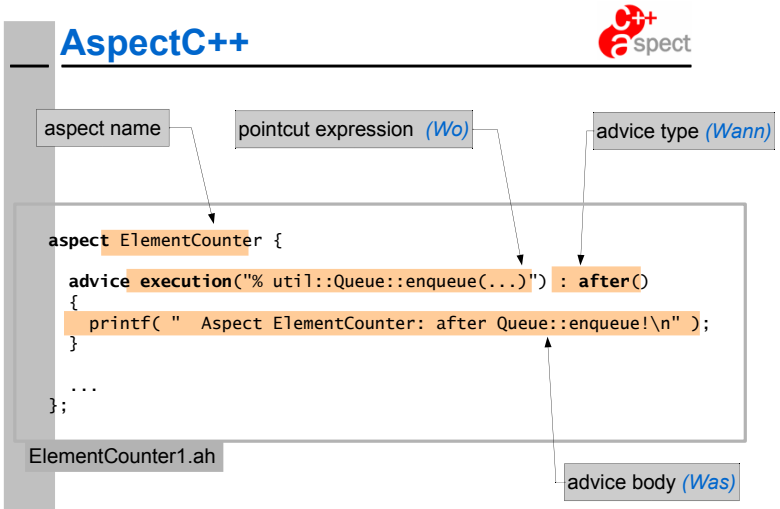
Aspect-Oriented Programming

- ...erlaubt die modulare Implementierung querschneidender Belange



- Beispiele: Kontrollflussverfolgung, Synchronisation, Sicherheit, Pufferung, Fehlerbehandlung, Überprüfung von Kontrakten, ...

AspectC++



OO-Stubs Kernel-Sperre mit AOP

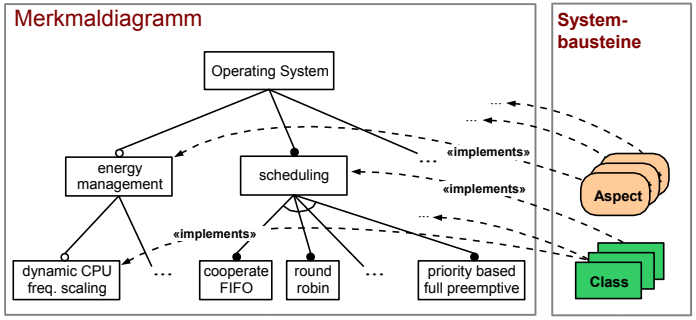
```
#include "guard/guard.h"
extern Guard guard;

aspect KernelLock {
  pointcut user() = "Application" || "Board" ...;
  pointcut kernel() = "Buzzer" || "Keyboard" ...;
  pointcut kernel_enter() = call(kernel())
                         && within(user());

   advice kernel_enter() : before() {
     guard.enter();
   }
   advice kernel_enter() : after() {
     guard.leave();
 }};
```


Aspekte als Systembausteine

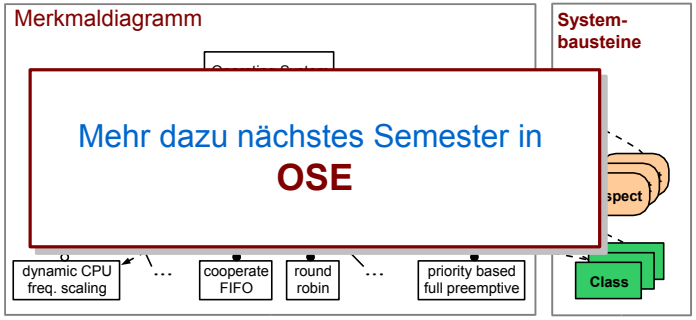
- **Querschneidende Belange** werden durch **Aspekte** implementiert



Merkmale mit querschneidender Implementierung können nun direkt auf ***ein*** Modul abgebildet werden.

Aspekte als Systembausteine

- **Querschneidende Belange** werden durch **Aspekte** implementiert



Merkmale mit querschneidender Implementierung können nun direkt auf ***ein*** Modul abgebildet werden.