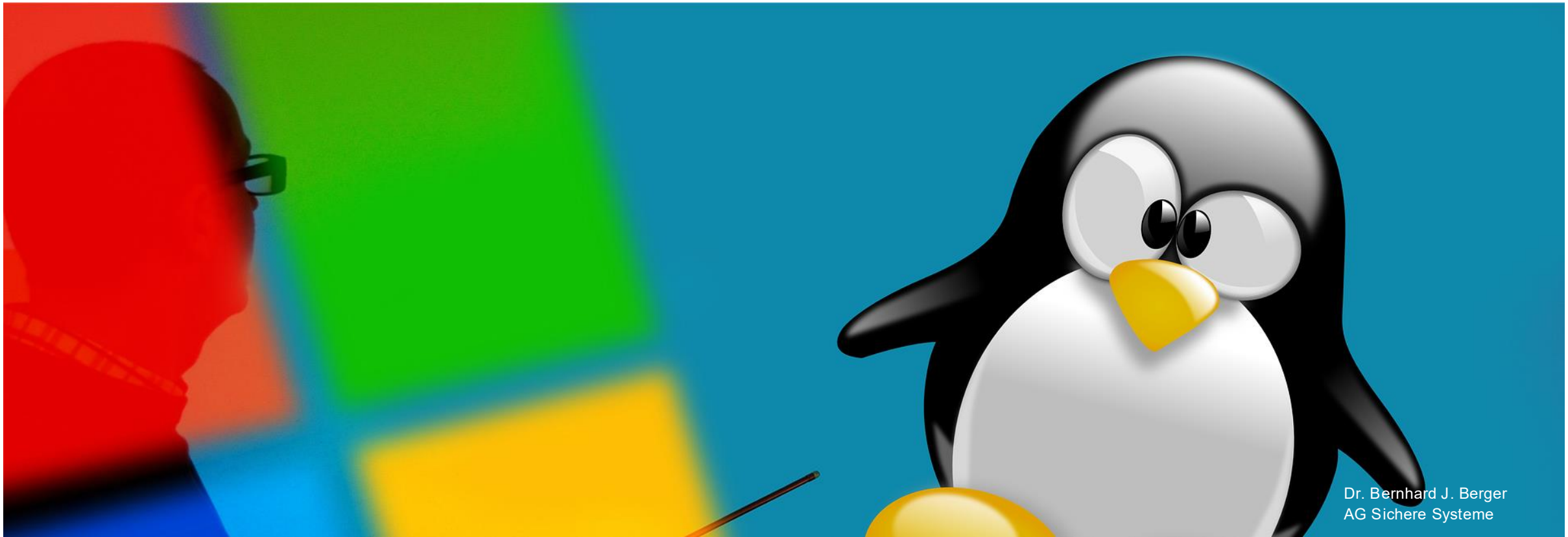
Universität Bremen

Betriebssysteme

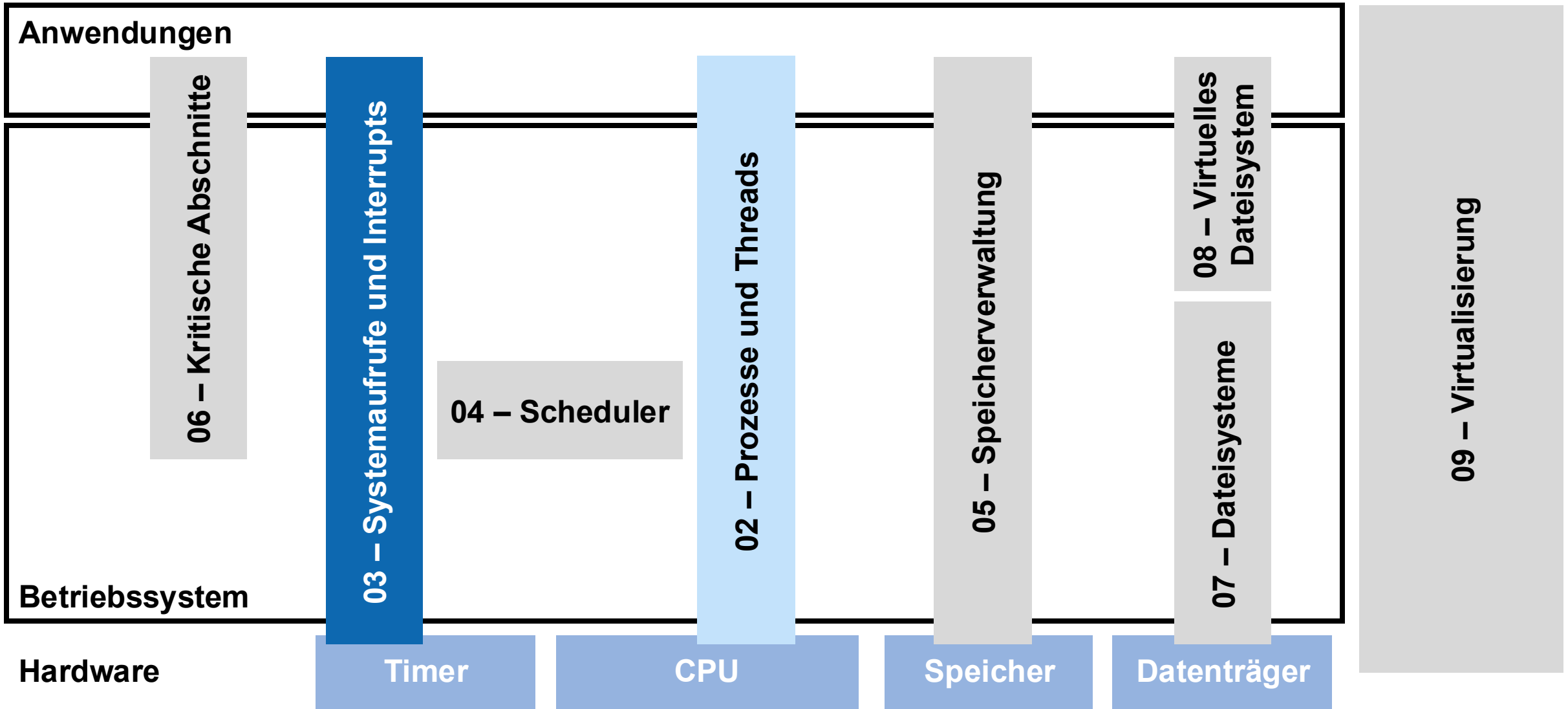
Systemaufrufe



Dr. Bernhard J. Berger
AG Sichere Systeme

Universität Bremen

Themenübersicht



Universität Bremen

Themenübersicht

- IDT
- Kernel-ISR
- SoftIRQ + Tasklets
- Signale
- SIGALRM

Unterbrechungen

- Prozessorunterstützung
- Ablauf
- Interrupts
- System Call-Tabelle

Systemaufrufe

- Interrupt 0x80
- syscall/sysret
- RISC-V

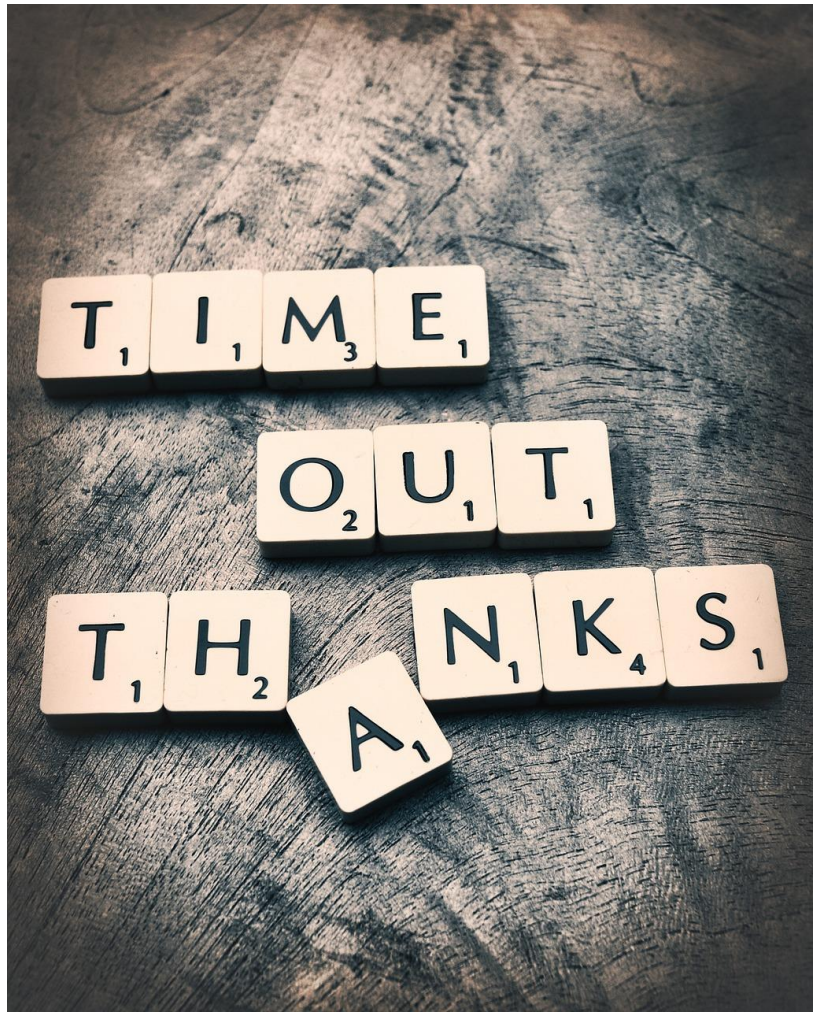
Umsetzung

Erinnerung: Interrupts

- Möglichkeit die CPU zu unterbrechen
- Mögliche Quellen
  - Hardware
  - Software
- Ablauf
  - Interruptleitung ist gesetzt

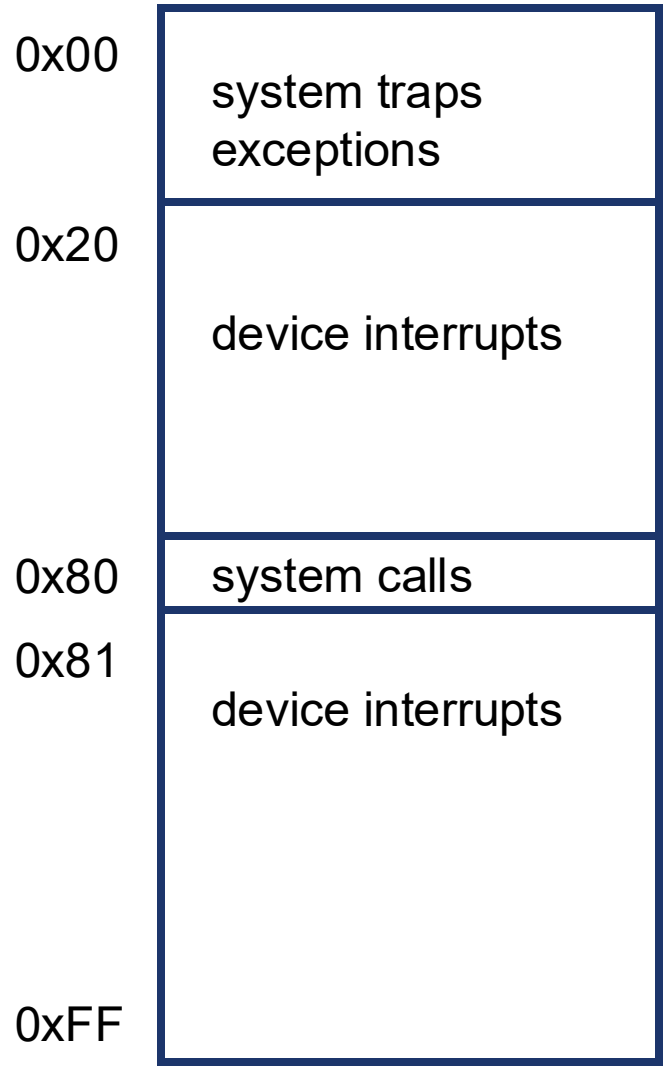
    (interrupt request, IRQ)
  - CPU bestimmt den Interrupt-Handler

    (interrupt service routine, ISR)
  - ISR wird abgearbeitet



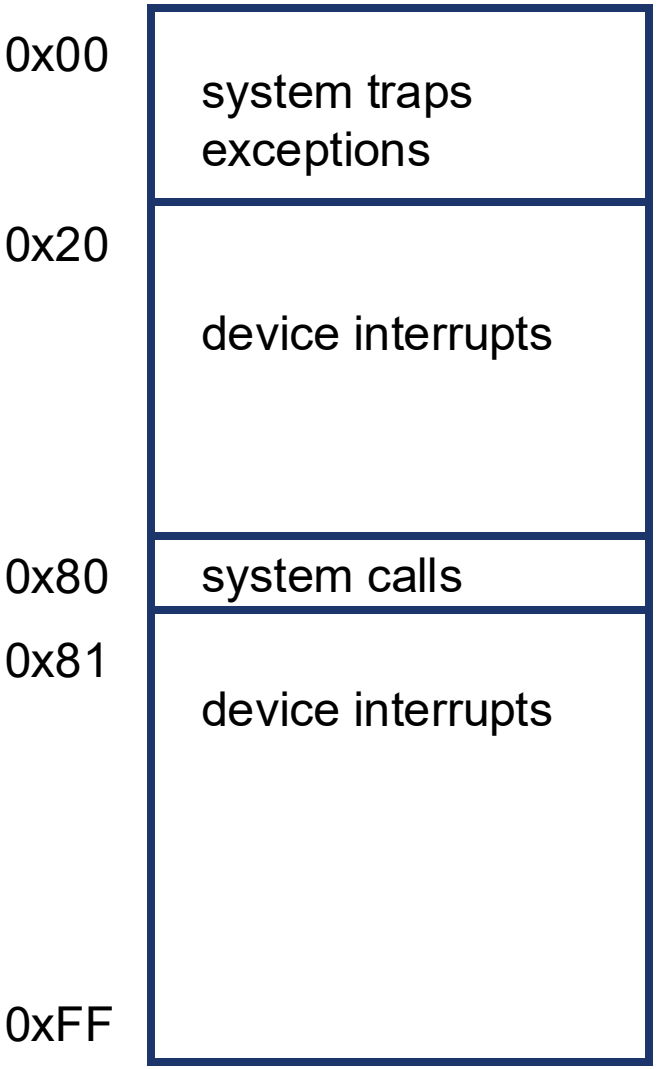
Interrupt Descriptor Table (IDT)

- Befindet sich in
  - arch/x86/include/{asm/irq_vectors.h,idt.c, traps.c}
- Architekturspezifisch
  - x86 besitzt eine IDT
  - x86_64 besitzt eine IDT pro CPU
- Interupt-Bereiche
  - 0x00 – 0x1F CPU-Interrupts (Division durch 0, Seitenfehler)
  - 0x20 – 0x7F Device-Interrupts (Timer, Tastatur, …)
  - 0x80 – Systemaufrufe
  - 0x81 – 0xFF Weitere Device-Interrupts oder Software-Interrupts
- APIC (Advanced Programmable Interrupt Controller) erlaubt
  - Zuordnung von Interrupts
  - Prioritäten von Interrupts
  - Inter-CPU Interrupts
- System-Timer
  - Regelmäßiger Interrupt (z.B. für Scheduling)
  - PIT, HPET, APIC



Interrupt Descriptor Table (IDT)

- Kernel erstellt Array mit ISR-Informationen
  `struct idt_entry idt[256];`
- Einträge können mit `set_idt_entry(uint8_t interrupt, uint32_t isr)` gesetzt werden
- IDT wird mit Assemblerbefehl `lidt` in das `idtr`-Register geladen



Kernel-Module mit ISR

```c
#include <linux/module.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#define IRQ_NUMBER 42 // Beispiel IRQ


MODULE_LICENSE("GPL");
MODULE_AUTHOR("Bernhard J. Berger");
MODULE_DESCRIPTION("Nur ein ISR-Beispiel");

// ISR
static irqreturn_t isr_42(int irq, void *dev_id) {
  printk(KERN_INFO "Interrupt %d empfangen!\n", irq);
  return IRQ_HANDLED;
}
```

```c
// Initialisieren
static int __init my_isr_init(void) {
  int result = request_irq(IRQ_NUMBER, isr_42,
                IRQF_SHARED, "isr_42", (void *)(isr_42));
  if (result) {
    printk(KERN_ERR "Fehler beim Registrieren\n");
    return result;
  }

  printk(KERN_INFO "Interrupt-Handler registriert\n");
  return 0;
}

// Aufräumen
static void __exit my_isr_exit(void) {
  free_irq(IRQ_NUMBER, (void *)(isr_42));
  printk(KERN_INFO "Interrupt-Handler entfernt\n");
}

module_init(my_isr_init);
module_exit(my_isr_exit);
```

SoftIRQs und Tasklets

- Software IRQs für Software-Interrupts
- Hierfür eine gesonderte Struktur
  - kernel/softirq.c

```
/* Rewritten. Old one was good in 2.2, but in 2.3 it was immoral. --ANK (990903) */
static struct softirq_action softirq_vec[NR_SOFTIRQS] __cacheline_aligned_in_smp;
```

- Interrupts sollen möglichst kurz sein
- Tasklets erlauben asynchrone Verarbeitung

```
void tasklet_function(unsigned long data) {
  printk(KERN_INFO "Tasklet ausgeführt mit Parameter: %lu\n", data);
}

DECLARE_TASKLET(tasklet, tasklet_function, 1234);

// in init
tasklet_schedule(&tasklet);

// in exit
tasklet_kill(&tasklet);
```

Interrupts vs. Signale

- Interrupts nur im Kernel behandelbar
  - `request_irq`
  - `free_irq`
- Signale sind User-Level-Konzept
  - `signal`
  - Signale haben ein Standard-Handler
  - Handler können nur überschrieben werden
- Einige Signale entsprechen Interrupts
  - SIGINT entspricht dem Keyboard bei Strg+C
  - SIGSEGV
  - SIGBUS
  - SIGALRM

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void signal_handler(int signo) {
  if (signo == SIGUSR1) {
    printf("Signal SIGUSR1 empfangen\n");
  }
}

int main() {
  if (signal(SIGUSR1, signal_handler) == SIG_ERR) {
    printf("Fehler beim Registrieren des Handlers\n");
    return 1;
  }

  printf("Warte auf Signale...\n");
  while (1) {
    pause(); // Warten auf Signal
  }

  return 0;
}
```

SIGALRM

- Vergleichbar mit System-Timer
- Regelmäßiges Signal
- Eignet sich für regelmäßige Arbeiten
- Signal-Handler wird bei Ablauf des Timers aufgerufen
- Ermöglicht Kontextwechsel bei ucontext
  - Fragt aktiven Kontext ab
  - Erzeugt Kontext für Scheduler
  - Wechselt Kontext
  - Schedulter wählt nächsten Kontext aus.

```c
#include <sys/time.h>
#include <signal.h>
#include <stdio.h>

void handle_alarm(int signo) {
  if(signo == SIGALRM) printf("SIGALRM empfangen!\n");
}

int main() {
  struct itimerval timer;
  signal(SIGALRM, handle_alarm);

  // Initiales Intervall
  timer.it_value.tv_sec = 0;
  timer.it_value.tv_usec = 500;

  // Wiederholungsintervall
  timer.it_interval.tv_sec = 0;
  timer.it_interval.tv_usec = 500;

  // Timer aktivieren
  setitimer(ITIMER_REAL, &timer, NULL);

  while (1) {}
  return 0;
}
```

Universität Bremen

Themenübersicht

- IDT
- Kernel-ISR
- SoftIRQ + Tasklets
- Signale
- SIGALRM

Unterbrechungen

- Prozessorunterstützung
- Ablauf
- Interrupts
- System Call-Tabelle

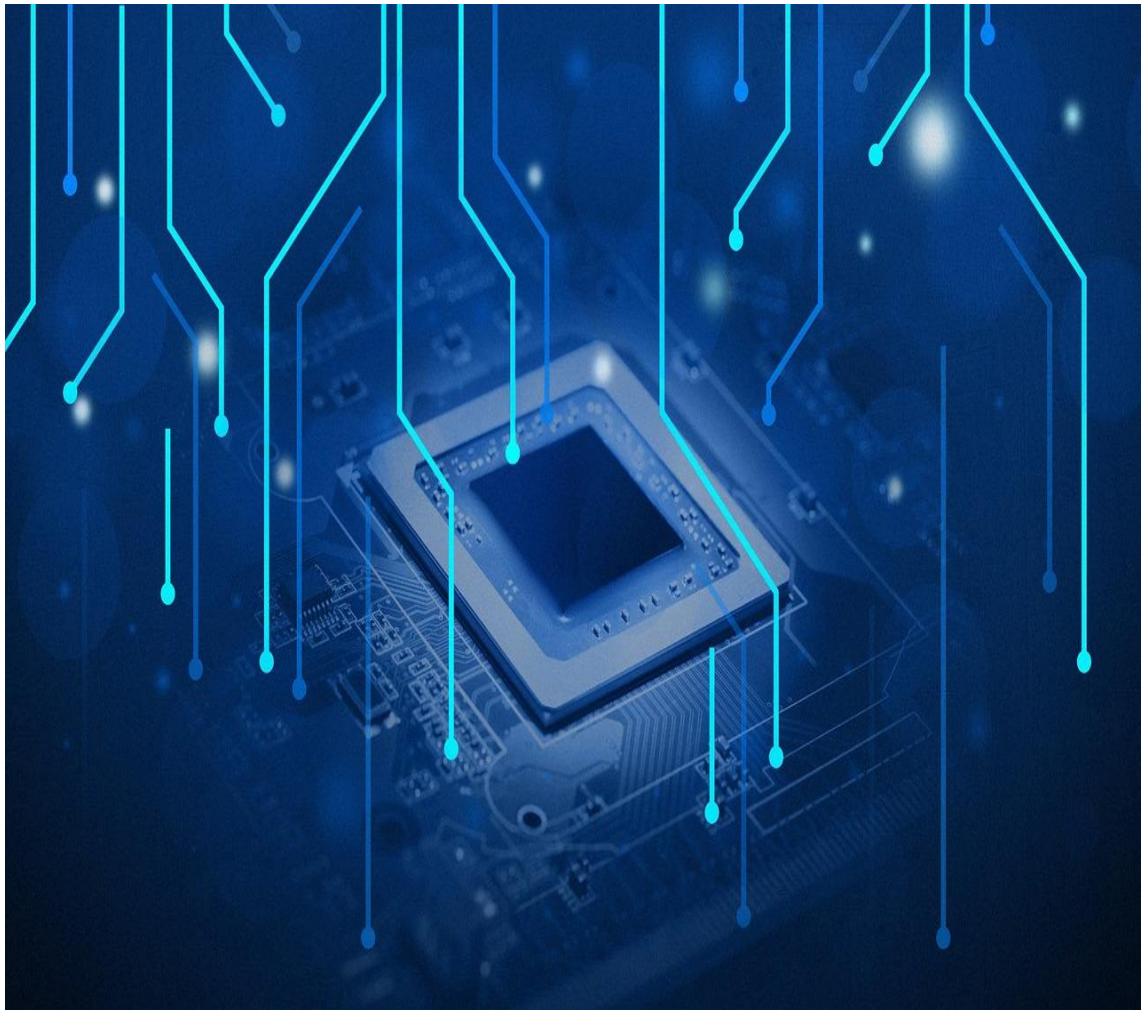
Systemaufrufe

- Interrupt 0x80
- syscall/sysret
- RISC-V

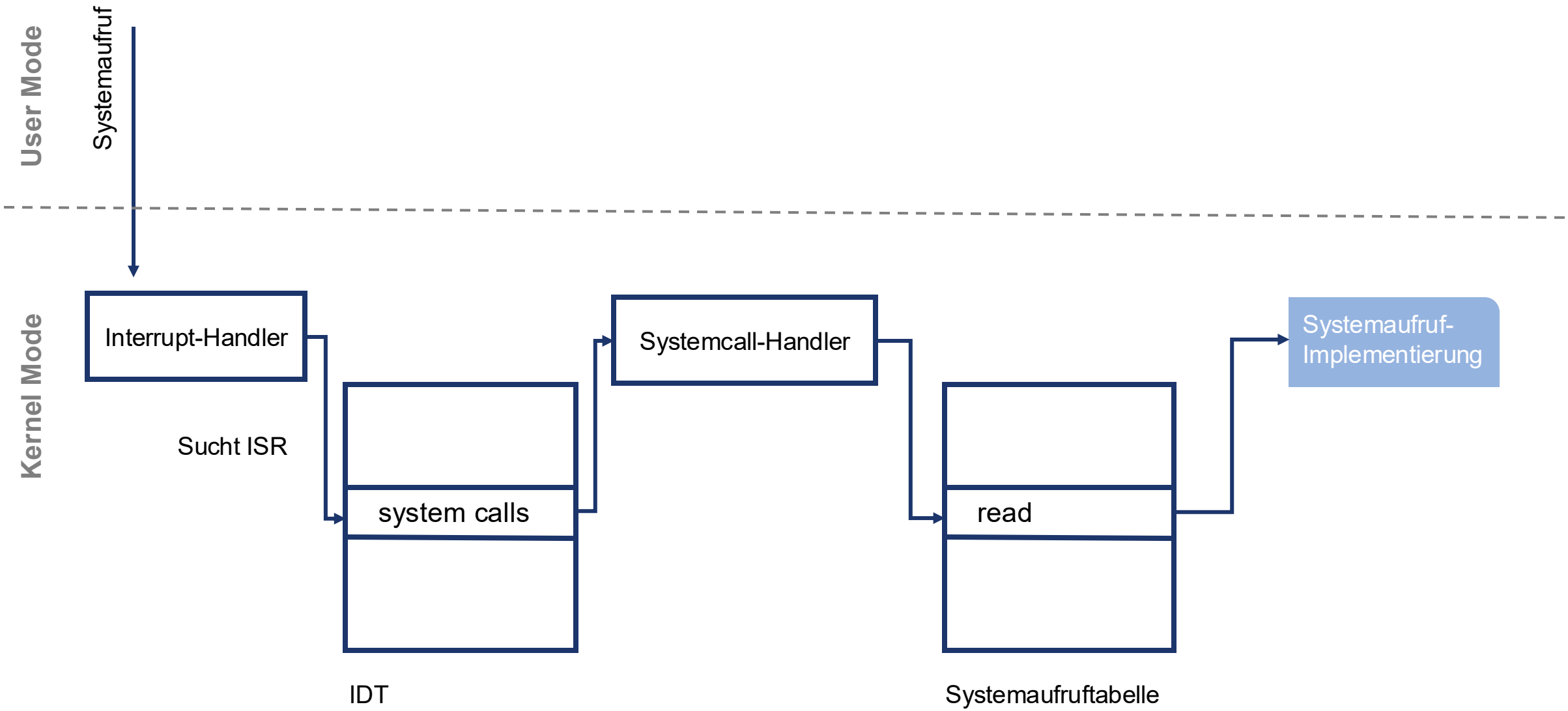
Umsetzung

Prozessorunterstützung

- Verschiedene CPU-Modi
  - Kernel Mode
  - User Mode
- Ringarchitektur (x86 Architektur)
  - Zugriffskontrollmodell
  - Vier Ringe mit gestaffelten Berechtigungen
  - Ring 0 führt Kernel aus
  - Ring 3 niedrigste Berechtigung für Anwendungen
  - Linux nutzt Ring 1 und 2 nicht
- Berechtigungen
  - Bestimmen ausführbare Befehle
  - Schränkt Zugriff auf Ressourcen/Register ein
- Ein Systemaufruf erzwingt einen Ringwechsel
  - Basierend auf Interrupts



Konzeptioneller Ablauf



System Call-Tabelle

- Linux sammelt eine Liste aller Systemaufrufe
  - Abbildung ID auf Funktionszeiger
  - /usr/include/asm*/unistd.h
  - arch/*/include/asm*/unistd*.h
- Systemaufruf besteht aus zwei Teilen
  - Funktion im User-Prozess
  - Funktion im Kernel
- User-Prozess-Funktion
  - Stellt Parameter für Kernel bereit
  - Initiiert Wechsel des Kontext

```
#define __NR_exit 93
__SYSCALL(__NR_exit, sys_exit)

#define __NR_exit_group 94
__SYSCALL(__NR_exit_group, sys_exit_group)

typedef long (*sys_call_ptr_t)(const struct pt_regs *);
extern const sys_call_ptr_t sys_call_table[];
```

Universität Bremen

Themenübersicht

- IDT
- Kernel-ISR
- SoftIRQ + Tasklets
- Signale
- SIGALRM

Interrupts

- Prozessorunterstützung
- Ablauf
- Interrupts
- System Call-Tabelle

Systemaufrufe

- Interrupt 0x80
- syscall/sysret
- RISC-V

Umsetzung

Interrupt 0x80 (x86 – 32 Bit)

User Space

- Anwendung ruft Systemaufruf auf
- Systemaufruf wird vorbereitet:
  - ID in Register speichern
  - Parameter in Registern abgelegen
- Interrupt 0x80 wird ausgelöst (`int 0x80`)

Kernel Space

- ISR für 0x80 wird ausgeführt
- Parameter Cleanup
- Aufruf über System-Call Tabelle
- ASM-Befehl `iret` zum Verlassen
- Ring 0 wird verlassen

User

- Systemaufruf gibt Ergebnis zurück
- Programm läuft weiter

```c
int main() {
  unsigned int syscall_nr = 1;
  int exit_status = 42;

  asm (
   // Kopiere lokale Var syscall_nr in Reg. eax
   "movl %0, %%eax\n"

   // Kopiere lokale Var exit_status in Reg. ebx
   "movl %1, %%ebx\n"

   // Befehl int mit der Konstante 0x80 ausführen
   "int $0x80"
  : // Befehl liefert keinen Rückgabewert
  : // Spezifikation der Eingabeparameter (für den Compiler)
   "m" (syscall_nr), "m" (exit_status)
  : // Information, welche Register modifiziert werden
   "eax", "ebx");
}
```

Fast System Calls – syscall/sysret (x86 – 64 Bit)

Kernel

- syscall_init registriert Callback (Link)
  `wrmsrl(MSR_LSTAR, system_call);`

User Space

- Anwendung ruft Systemaufruf auf
- Systemaufruf wird vorbereitet:
  - ID in Register `%rax` speichern
  - Parameter in Registern abgelegen
  - Maximal sechs Parameter
  - Keine Stackparameter
- `syscall` wird ausgeführt

Kernel

- Aufruf über registrierte Tabelle
- ASM-Befehl `sysret` zum Verlassen
- Ergebnis in Register `%rax`

User

- Systemaufruf gibt Ergebnis zurück
- Programm läuft weiter

```c
int main() {
  unsigned int syscall_nr = 93;
  int exit_status = 42;

  asm (
   // Kopiere lokale Var syscall_nr in Reg. rax
   "movq %0, %%rax\n"

   // Kopiere lokale Var exit_status in Reg. rdi
   "movq %1, %%rdi\n"

   // syscall-Vefehl ausführen
   ”syscall"
  : // Befehl liefert keinen Rückgabewert
  : // Spezifikation der Eingabeparameter (für den Compiler)
   "m" (syscall_nr), "m" (exit_status)
  : // Information, welche Register modifiziert werden
   ”rax", ”rdi");
}
```

System Calls – Exkurs RISC-V

- RISC-V Sicherheitsmodell
  - Modulares Ringkonzept
  - Machine-, Supervisor- und User-Mode
  - Unterteilung von U/S-Mode für Virtualisierung
- Hardware-Instruktionen
  - `ecall` – Aufruf eines Systemaufrufs
  - `sret` – Rückkehr aus dem Supervisor-Mode
  - `mret` – Rückkehr aus dem Machine-Mode
- Systemaufruftabelle über Register
    - `mvec` – Tabelle für M-Mode
    - `stvec` – Tabelle für S-Mode
- Systemaufruf setzt
    - Systemaufrufnummer in `a7`
    - Parameter in Register `a0` bis `a5`
    - `ecall` wechselt den Mode

Universität Bremen

Themenübersicht

- IDT
- Kernel-ISR
- SoftIRQ + Tasklets
- Signale
- SIGALRM

Unterbrechungen

- Prozessorunterstützung
- Ablauf
- Interrupts
- System Call-Tabelle

Systemaufrufe

- Interrupt 0x80
- syscall/sysret
- RISC-V

Umsetzung