Übung zu Betriebssysteme

Interrupts und Traps auf x86

Wintersemester 2020/21

Bernhard Heinloth & Christian Eichler

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

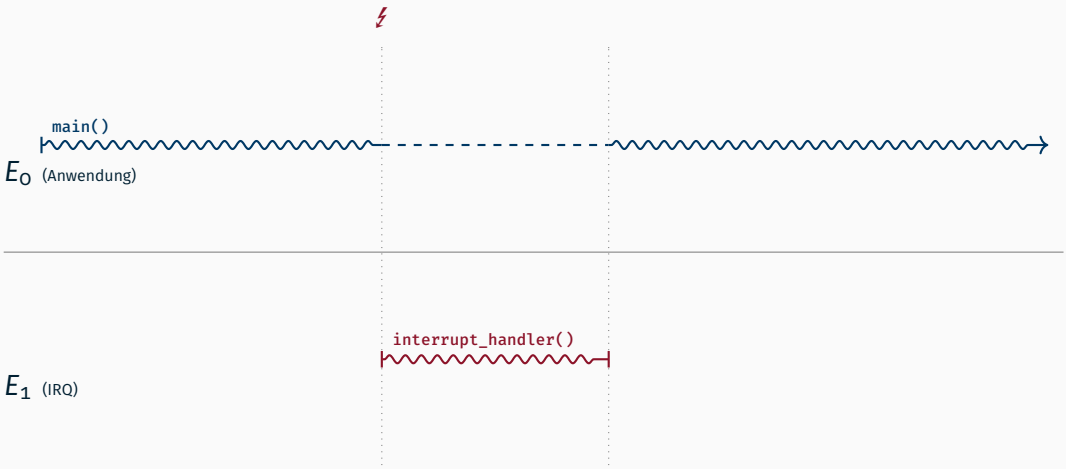
Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT


CPU

Unterbrechung (Anwendungssicht)



Minimaler zu sichernder Zustand?

- CPU sichert automatisch

rip	Instruktionszeiger/Rücksprungadresse
rflags	Status/Condition Codes
cs	Aktuelles Code Segment

- Wiederherstellung des ursprünglichen Prozessorzustandes durch Befehl `iretq`

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler
interrupt_entry:
  ;; Behandle IRQ
  ;; in Hochsprache
  call interrupt_handler
  iretq
```

```
// C++

void interrupt_handler()
{
    // Magie.
}
```

```
heinloth:~/oostubs$ make
LD          .build/system64
.build/interrupt/handler.asm.o: in function `interrupt_entry':
interrupt/handler.asm:(.text+0x16): undefined reference to `
    interrupt_handler'
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler
interrupt_entry:
  ;; Behandle IRQ
  ;; in Hochsprache
  call interrupt_handler
  iretq
```

```
// C++ (mit C Linkage)
extern "C"
void interrupt_handler()
{
    // Magie.
}
```

```
heinloth:~/oostubs$ objdump -d .build/interrupt/handler.o
Disassembly of section .text:

0000000000000000 <interrupt_handler>:
   0:  f3 c3                    repz ret
```

Was ist mit den restlichen Registern?

- Müssen durch den Interrupthandler selbst gesichert werden.
  - entweder im Assembler-Teil oder
  - der Compiler generiert bereits entsprechend Code
- Kontextsicherung beim Aufruf von Funktionen
  1. Aufrufende Funktion sichert alle Register, die sie braucht
  2. Aufgerufene Funktion sichert alle Register, die sie verändert
  3. Ein Teil der Register wird vom Aufrufer, ein anderer Teil vom Aufgerufenen gesichert
- In der Praxis wird Variante 3 verwendet
  - Aufteilung ist grundsätzlich compilerspezifisch
  - Um Interoperabilität auf Binärcodeebene sicher zu stellen gibt es jedoch Konventionen (bei x64 zwei: *Microsoft* und *System V*)

  → Aufrufkonvention ist Teil der *Application Binary Interface* (ABI)

Aufteilung der Register in zwei Gruppen

- Flüchtige Register (*scratch registers*)
  - Compiler geht davon aus, dass Unterprogramm den Inhalt verändert
  - Aufrufer muss Inhalt gegebenenfalls sichern
  - Beim x64 (nach *System V* ABI) sind `rax`, `rcx`, `rdx`, `rsi`, `rdi` und `r8` – `r11` als flüchtig definiert
- Nicht-flüchtige Register (*non-scratch registers*)
  - Compiler geht davon aus, dass der Inhalt durch Unterprogramm nicht verändert wird
  - Aufgerufene Funktion muss Inhalt gegebenenfalls sichern
  - Beim x64 sind alle sonstigen Register als nicht-flüchtig definiert: `rbx`, `rbp`, `rsp` und `r12` – `r15`

Unterbrechungsbehandlungen müssen auch flüchtige Register sichern!

Unterbrechungsbehandlung (Kontextsicherung)

```
;; Assembler
interrupt_entry:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11


  call interrupt_handler
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

Parameterübergabe

- auf Stack
  - bei x86 war dies gemäß der *C declaration* der Standard
- in Register
  - Vorteil: schneller
  - Problem: Anzahl der Register begrenzt
- kombiniert
  - die ersten Parameter via Register, danach bei Bedarf Stack
  - auch nach x64 *System V* ABI:
    - Parameter zuerst in Register `rdi`, `rsi`, `rdx` `rcx`, `r8` und `r9`
    - im Userspace danach auch in die Register `xmm0` – `xmm7`
    - Rest auf Stack (letzter Parameter wird als erstes gepushed)

Unterbrechungsbehandlung (Kontext)

```
;; Assembler
interrupt_entry:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp
  call interrupt_handler
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

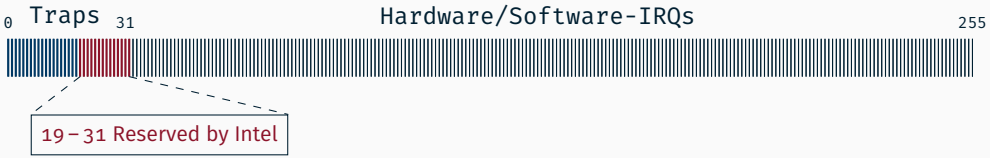
```
// C++
struct Context {
    uint64_t r11;
    // ...
    uint64_t rcx;
    uint64_t rax;
    uint64_t rip;
    uint64_t cs;
    uint64_t rflags;
} __attribute__((packed));

extern "C"
void interrupt_handler(Context* c)
{
    // Magie.
}
```


0-255
CPU

Interruptvektoren (x86/x64)

0	**Division-by-Zero**
1	Debug Exception
2	Non-Maskable Interrupt(NMI)
3	**Breakpoint (INT 3)**
4	Overflow Exception
5	Bound Exception
6	**Invalid Opcode**
7	FPU not Available
8	Double Fault
10	Invalid TSS
11	Segment not Present
12	Stack Exception
13	**General Protection Fault**
14	Page Fault
16	Floating-Point Error
17	Alignment Check
18	Machine Check

Interruptvektoren (x86/x64)



Interruptvektoren (x86/x64)

32 – 255 Einträge für IRQs

- Softwareauslösung mit `int <vec#>`
- Hardwareauslösung durch externe Geräte

Interruptvektoren (x86/x64)

Kann durch Prozessorbefehle maskiert werden

- `cli` (**cl**ear **i**nterrupt flag) Interruptleitung sperren
- `sti` (**s**e**t** **i**nterrupt flag) Interruptleitung freigeben

Unterbrechungsbehandlung

```
;; Assembler
interrupt_entry:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp


  call interrupt_handler
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++
extern "C"
void interrupt_handler(
    Context* c,
    uint32_t vector
  )
{
    // Magie:
    switch (vector){
        case KBD:
            kbd.magic();
            break;
        case TMR:
            tmr.magic();
            break;
    }
}
```

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler
interrupt_entry_6:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp
  ;; Vektornummer
  mov rsi, 6
  call interrupt_handler
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++
extern "C"
void interrupt_handler(
    Context* c,
    uint32_t vector
  )
{
    // Magie:
    switch (vector){
        case KBD:
            kbd.magic();
            break;
        case TMR:
            tmr.magic();
            break;
    }
}
```

Unterbrechungsbehandlung

```
%macro IRQ 1
align 8
interrupt_entry_%1:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp
  ;; Vektornummer
  mov rsi, %1
  call interrupt_handler
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
%endmacro
```

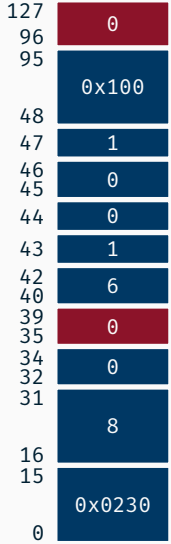
```
heinloth:~/oostubs$ make
ASM   interrupt/handler.asm
CXX   interrupt/handler.cc
LD    .build/system64
```

Speicheradressen beim **Binden**:

```
100 70f0 <interrupt_handler>
         ...
100 0230 <interrupt_entry_6>
100 0200 <interrupt_entry_5>
100 01d0 <interrupt_entry_4>
100 01a0 <interrupt_entry_3>
100 0170 <interrupt_entry_2>
100 0140 <interrupt_entry_1>
100 0110 <interrupt_entry_0>
```

Woher weiß die CPU wo die entsprechende Unterbrechungsbehandlung liegt?

Interrupt Deskriptor (Beispiel)



Unused – muss `0` sein

Offset (high): *oberer* Teil der Einsprungsadresse für die Interruptbehandlung (z.B. `interrupt_entry_6`)

Present: Eintrag aktiv (`1`) oder inaktiv (`0`)

Descriptor Privilege Level

Storage Segment: `0` für Interrupt und Traps

Mode: 16-bit (`0`) oder 32/64-bit (`1`)

Type: Task (`5`), Interrupt (`6`) oder Trap (`7`)?

Unused – muss `0` sein

Interrupt Stack Table

Selector: Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)

Offset (low): *unterer* Teil der Einsprungsadresse für die Interruptbehandlung

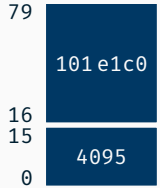
für `100 0230 <interrupt_entry_6>` → `0x100 8e00 0008 0230`

Interrupt Deskriptor Tabelle (IDT)

```
100 30e0 <interrupt_entry_255>
         ...
100 0230 <interrupt_entry_6>
100 0200 <interrupt_entry_5>
100 01d0 <interrupt_entry_4>
100 01a0 <interrupt_entry_3>
100 0140 <interrupt_entry_2>
100 0130 <interrupt_entry_1>
100 0110 <interrupt_entry_0>
```

Interrupt Deskriptor Tabelle (IDT)

IDT-Register `idtr`



Basis: Startadresse der IDT

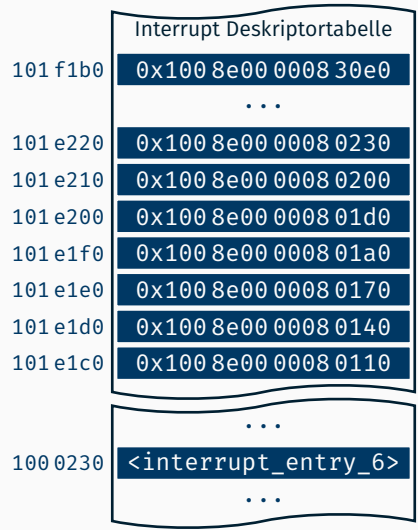
Limit: Bytes *Einträge * 16 - 1*

Instruktionen:

`lidt` in Register laden

`sidt` aus Register lesen

Speicher



Interrupt Deskriptor Tabelle (IDT)