Technische Universität Dresden

Faculty of Computer Science Institute of Systems Architecture, Operating Systems Group

OPERATING-SYSTEM CONSTRUCTION

Exercise 3: Traps, startup.asm, Task #3

https://tud.de/inf/os/studium/vorlesungen/betriebssystembau

TILL SMEJKAL (slides by HORST SCHIRMEIER)

Overview

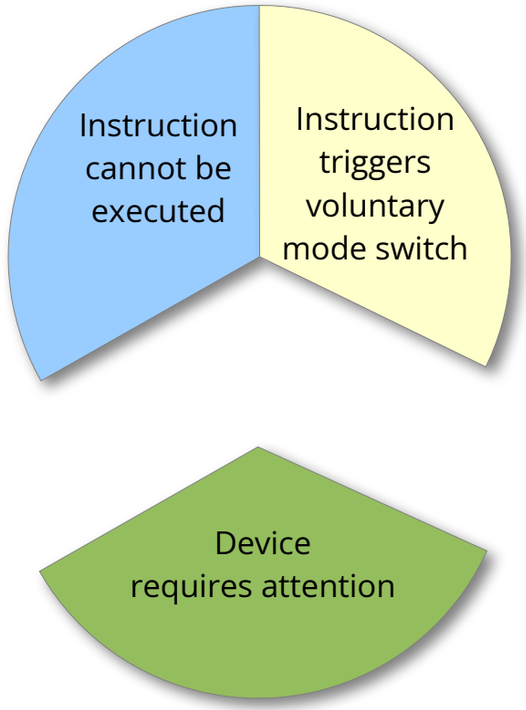
- Traps
- Lab Task #2
  - What does startup.asm do?
  - What should happen in parts c) and d)?
  - Repetition: **References** in C++
    - Plugbox and Gate
- Lab Task #3
  - Introduction
  - Repetition: **Pointers** in C++
    - The (weird?) Queue Class

Overview

- **Traps**
- Lab Task #2
  - What does startup.asm do?
  - What should happen in parts c) and d)?
  - Repetition: **References** in C++
    - Plugbox and Gate
- Lab Task #3
  - Introduction
  - Repetition: **Pointers** in C++
    - The (weird?) Queue Class

Interrupts and Traps

- **“Trap”**
  - Triggered (directly or indirectly) by an **instruction**
  - **Synchronous** to program execution
  - (Usually) predictable + reproducible
  - **Restart or abort** the triggering activity
- **“Interrupt”**
  - Triggered by **hardware** event
  - **Asynchronous** to program execution
  - Not predictable, not reproducible
  - Usually **resume** the interrupted activity



Traps – Overview

- Triggered by the CPU if it determines a **problem** executing the current instruction
  - Division by 0
  - Bus error (program generates invalid address)
  - OS must do something (e.g. page fault)
  - Invalid instruction
- … or if the executed instruction is ***supposed*** to cause a trap
  - `int 0x3`
  - `int 0x80`
  - `syscall` / `sysenter`

x86 IDT: Structure

Number	Description
0	Divide-by-zero
1	Debug exception
2	Non-Maskable Interrupt (NMI)
3	Breakpoint (INT 3)
4	Overflow
5	Bound exception
6	Invalid Opcode
7	FPU not available
8	Double Fault
9	Coprocessor Segment Overrun
10	Invalid TSS
11	Segment not present
12	Stack exception
13	General Protection
14	Page fault
15	Reserved
16	Floating-point error
17	Alignment Check
18	Machine Check
19-31	Reserved By Intel

- Entries 0–31 for traps (fixed)
- Trap = Exception that occurs synchronously to control flow
  - Division by 0
  - Page fault
  - Breakpoint
  - …
- Entries 32–255 for IRQs (configurable)
  - Software (**`INT`** <number>)
  - Hardware (CPU's INT pin to HIGH, #number on data bus)

Traps – Examples

- **Division by zero**

```
1  int main() {
2     int i, j;
3
4     i = 3;
5     j = 0;
6
7     return i / j;
8  }
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $3, -4(%rbp)
        movl    $0, -8(%rbp)
        movl    -4(%rbp), %eax
        cltd
        idivl   -8(%rbp)
        popq    %rbp
        ret
```

```
$ gdb ./divnull
(gdb) run
Program received signal SIGFPE,
Arithmetic exception.
0x0804836b in main () at divnull.c:7
7             return i/j;
```

Traps – Examples

- **Page Fault**

```
1  int main() {
2     int *p;
3
4     p = (int *)0;
5
6     return *p;
7  }
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    $0, -8(%rbp)
        movq    -8(%rbp), %rax
        movl    (%rax), %eax
        popq    %rbp
        ret
```

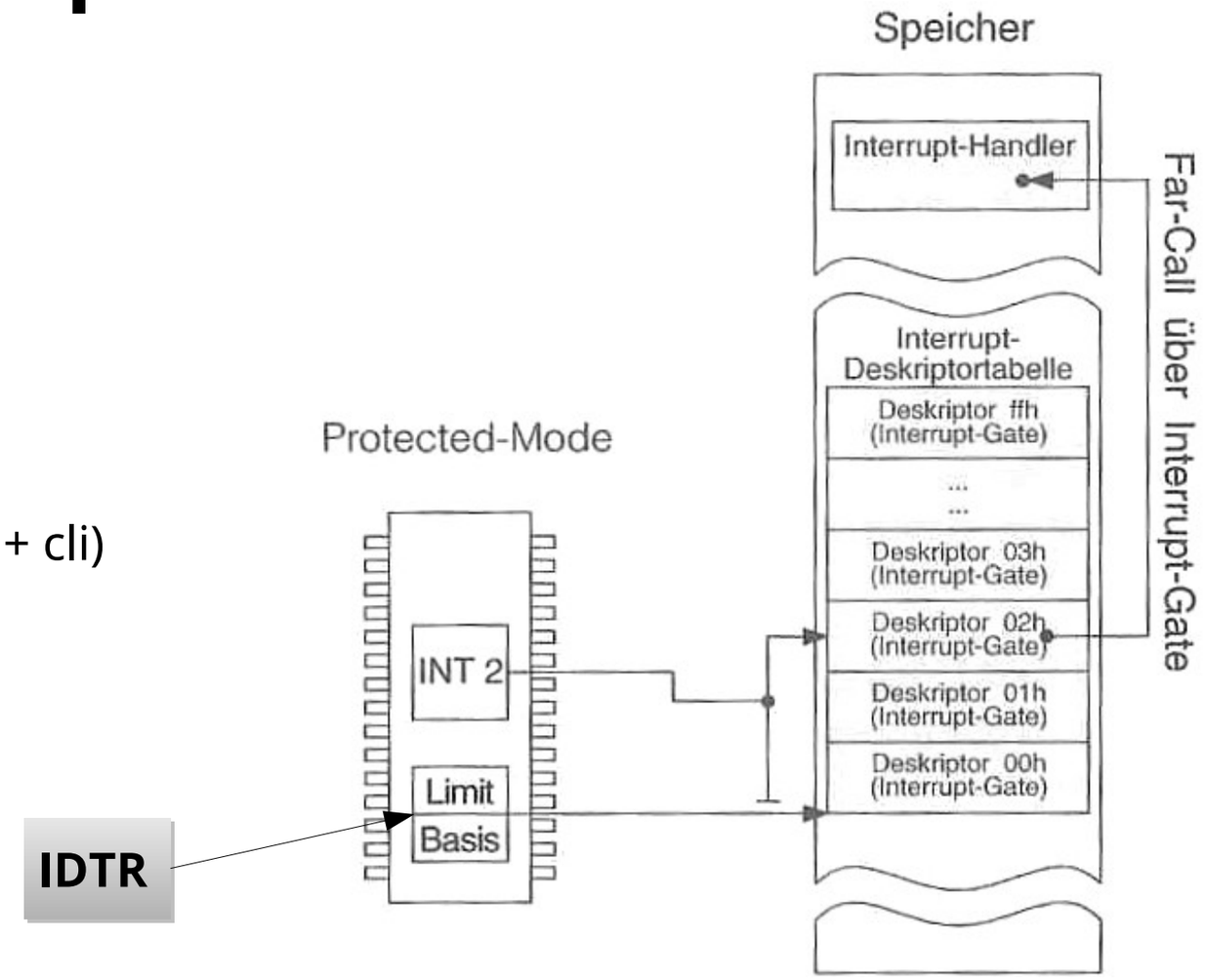
```
$ gdb ./segfault
(gdb) run
Program received signal SIGSEGV,
Segmentation fault.
0x0804835f in main () at segfault.c:6
6             return *p;
```

Overview

- Traps
- **Lab Task #2**
  - What does startup.asm do?
  - What should happen in parts c) and d)?
  - Repetition: **References** in C++
    - Plugbox and Gate
- Lab Task #3
  - Introduction
  - Repetition: **Pointers** in C++
    - The (weird?) Queue Class

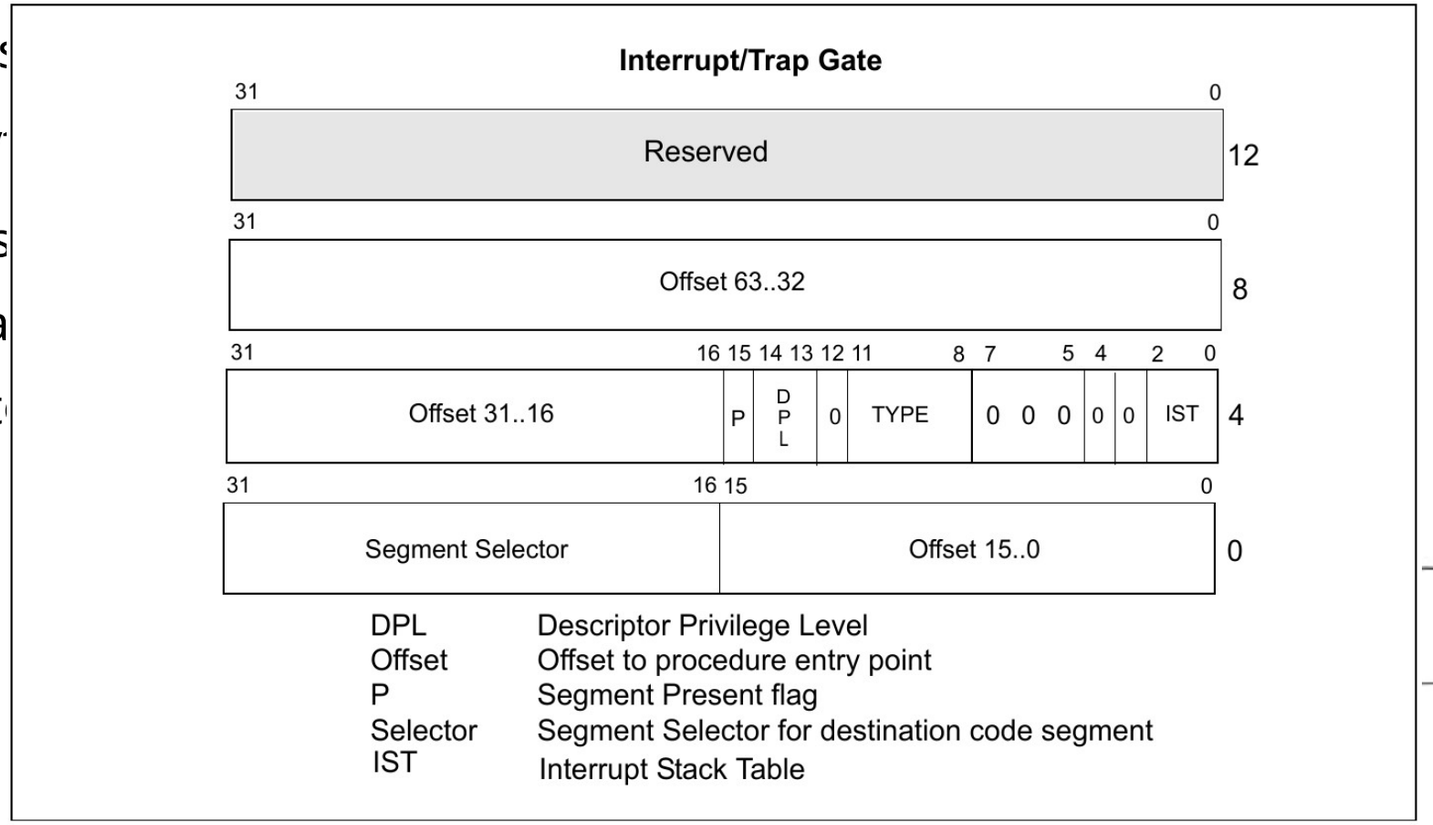
x86-64 Interrupt Descriptor Table

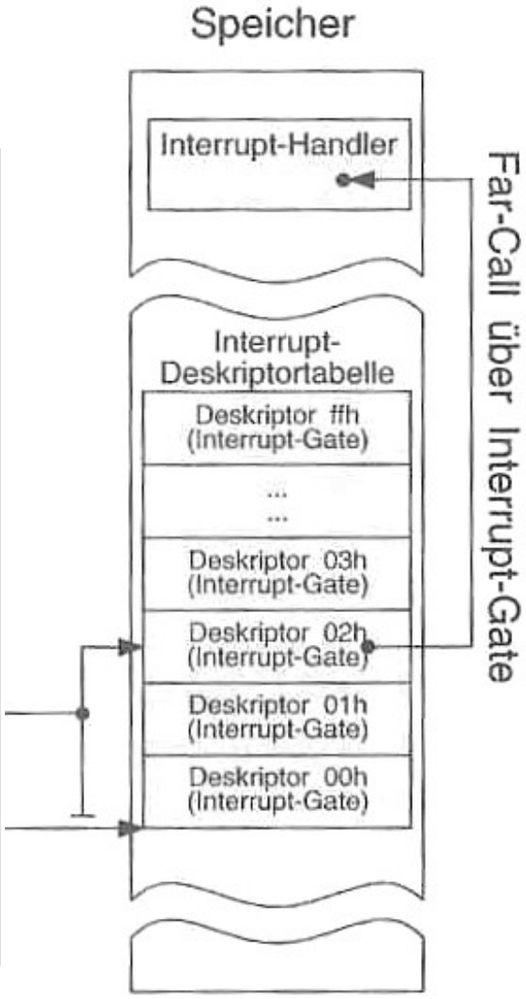
- max. 256 entries
  - Base address and size in IDTR
- 16 bytes per entry (“gate”)
  - Task gate (Hardware tasks)
  - Trap gate (Exception handler)
  - Interrupt gate (Exception handler + cli)



x86-64 Interrupt Descriptor Table

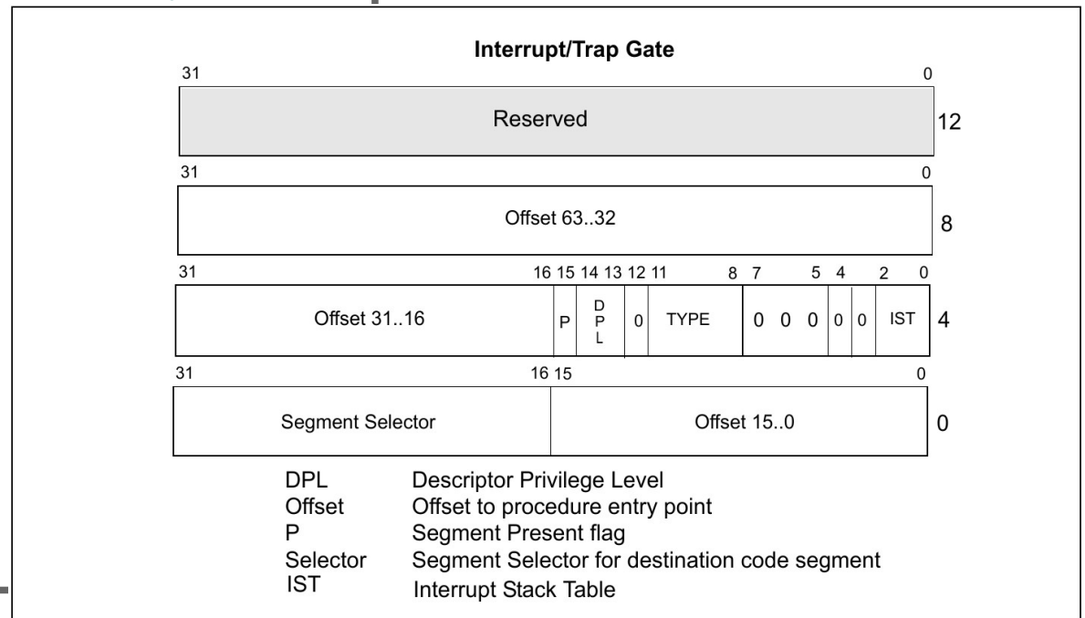
- max. 256 entries
  - Bas
- 16 by
  - Tas
  - Tra
  - Int





x86-64 Interrupt Descriptor Table

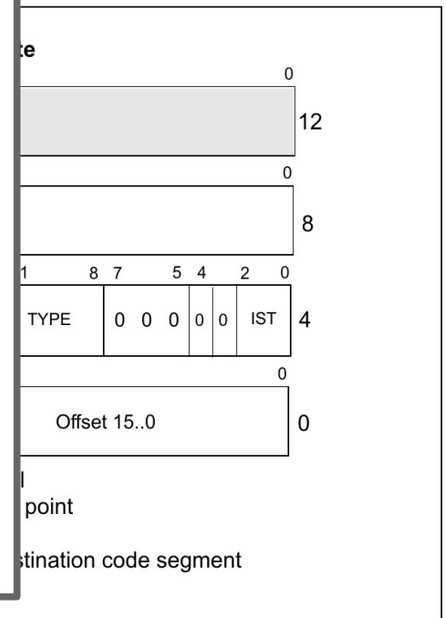
```
[SECTION .data]
; [...]
; Interrupt descriptor table with 256 entries

idt:
%macro idt_entry 1
  dw  (wrapper_%1 - wrapper_0) & 0xffff                         ; offset
  dw  0x0000 | 0x8 * 2  ; 64-bit code segment selector
  dw  0x8e00            ; present + 64-bit interrupt gate
  dw  ((wrapper_%1 - wrapper_0) & 0xffff0000) >> 16             ; offset
  dd  ((wrapper_%1 - wrapper_0) & 0xffffffff00000000) >> 32 ; offset
  dd  0x00000000 ; reserved
%endmacro

%assign i 0
%rep 256
idt_entry i
%assign i i+1
%endrep

idt_descr:
        dw      256*8-1         ; 256 entries
        dq      idt
```



x86-64 Interrupt Descriptor Table

```
[SECTION .data]
; [...]
; Interrupt descripto

idt:
%macro idt_entry 1
  dw  (wrapper_%1 - w
  dw  0x0000 | 0x8 *
  dw  0x8e00
  dw  ((wrapper_%1 -
  dd  ((wrapper_%1 -
  dd  0x00000000 ; re
%endmacro

%assign i 0
%rep 256
idt_entry i
%assign i i+1
%endrep

idt_descr:
        dw      256*8
        dq      idt
```

```
; Relocating of IDT entries and setting IDTR

setup_idt:
  mov    rax, wrapper_0

  ; bits 0..15 -> ax, 16..31 -> bx, 32..63 -> edx
  mov    rbx, rax
  mov    rdx, rax
  shr    rdx, 32
  shr    rbx, 16

  mov    r10, idt    ; pointer to the actual interrupt gate
  mov    rcx, 255    ; counter
.loop:
  add    [r10+0], ax
  adc    [r10+6], bx
  adc    [r10+8], edx
  add    r10, 16
  dec    rcx
  jge    .loop

  lidt   [idt_descr]
  ret
```



State Save

- Every CPU has internal state
  - represented as register contents
- x86-64 (we're ignoring FPU + vector extensions here):

RAX	RSI	R8	R12	CS	FS
RBX	RDI	R9	R13	DS	GS
RCX	RBP	R10	R14	ES	SS
RDX	RSP	R11	R15	RFLAGS	

Total save (stack): 22 registers == 176 Bytes

Example (Linux arch/x86/entry/calling.h)

- Implemented as an assembly macro (used in many places)
  - analogously POP_REGS

```
.macro PUSH_AND_CLEAR_REGS rdx=%rdx rax=%rax save_ret=0
        pushq   %rdi            /* pt_regs->di */
        pushq   %rsi            /* pt_regs->si */
        pushq   \rdx            /* pt_regs->dx */
        xorl    %edx, %edx      /* nospec   dx */
        pushq   %rcx            /* pt_regs->cx */
        xorl    %ecx, %ecx      /* nospec   cx */
        pushq   \rax            /* pt_regs->ax */
        pushq   %r8             /* pt_regs->r8 */
        xorl    %r8d, %r8d      /* nospec   r8 */
        pushq   %r9             /* pt_regs->r9 */
        xorl    %r9d, %r9d      /* nospec   r9 */
        pushq   %r10            /* pt_regs->r10 */
        xorl    %r10d, %r10d    /* nospec   r10 */
        pushq   %r11            /* pt_regs->r11 */
        xorl    %r11d, %r11d    /* nospec   r11*/
        pushq   %rbx            /* pt_regs->rbx */
        xorl    %ebx, %ebx      /* nospec   rbx*/
        pushq   %rbp            /* pt_regs->rbp */
        xorl    %ebp, %ebp      /* nospec   rbp*/
        pushq   %r12            /* pt_regs->r12 */
        xorl    %r12d, %r12d    /* nospec   r12*/
        pushq   %r13            /* pt_regs->r13 */
        xorl    %r13d, %r13d    /* nospec   r13*/
        pushq   %r14            /* pt_regs->r14 */
        xorl    %r14d, %r14d    /* nospec   r14*/
        pushq   %r15            /* pt_regs->r15 */
        xorl    %r15d, %r15d    /* nospec   r15*/
        UNWIND_HINT_REGS
.endm
```

Example (Linux arch/x86/kernel/entry_64.S)

- Used e.g. in interrupt handlers:

```
ENTRY(error_entry)
        UNWIND_HINT_FUNC
        cld
        PUSH_AND_CLEAR_REGS save_ret=1
        ENCODE_FRAME_POINTER 8

        /* ... */

        CALL_enter_from_user_mode
        ret
```

Context Save: Who Does What?

- Context save **in interrupt handlers**:
  - **ss**, **rsp**, **rflags**, **cs**, and **rip** are automatically saved by the CPU
  - All other registers must be saved by the IRQ handler:
    - either in the wrapper function (assembler)
    - or the compiler already generates code for this

- Context save when **calling functions**

  (forget interrupt handling for a second!):
  - **Solution 1**: *Caller* saves all registers it still needs later
  - **Solution 2**: *Callee* saves all registers it modifies
  - **Solution 3**: A subset of the registers is *caller-saved*, the others are *callee-saved*

High-Level Language Context Save

- In practice, **solution 3** is used
  - Generally, this depends on the compiler – but CPU manufacturers define a standard to guarantee **interoperability on the binary level**.
- Partitioning into **two subsets** of registers
  - **Volatile registers** (*caller-saved*, *scratch* registers)
    - Compiler assumes the called function **will modify contents**
    - Caller must save (but only if it still needs contents)
    - x86-64: rax, rdi, rsi, rdx, rcx, r8–r11, all FPU/SSE/AVX/… registers
  - **Non-volatile registers** (*callee-saved*, non-scratch registers)
    - Compiler assumes the called function **will not modify contents**
    - Callee must save (but only if it modifies)
    - x86-64: rbx, rsp, rbp, r12–15

Interrupt handlers must also save volatile registers!

State Save in the Wrapper

```
; template for header for each interrupt-handling routine
%macro wrapper 1
wrapper_%1:
  push    rbp
  mov     rbp, rsp
  push    rax
  mov     al, %1
  jmp     wrapper_body
%endmacro

; common handler body
wrapper_body:
  cld                     ; GCC expects the direction flag to be 0
  push    rcx             ; save volatile registers
  push    rdx
  ; … rdi, rsi, r8, r9, r10 …
  push    r11
  and     rax, 0xff       ; generated wrapper only gives us 8 bits, mask the rest
  mov     rdi, rax        ; pass interrupt number as the first parameter
  call    guardian
  pop     r11             ; restore volatile registers
  ; … r10, r9, r8, rsi, rdi …
  pop     rdx
  pop     rcx
  pop     rax             ; ... also those from the header
  pop     rbp
  iretq                   ; done
```

PIC Initialization – Part 1

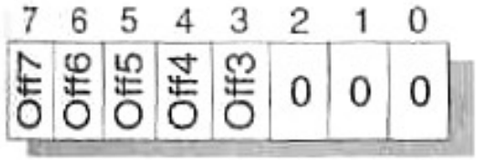
ICW1

LTIM: 0=Flankentriggerug 1=Pegeltriggerung
SNGL: 0=kaskadierte PICs 1=nur Master
IC4: 0=kein ICW4 1=ICW4 notwendig

OOStuBS setting:

Master	Slave
00010001	00010001

ICW2

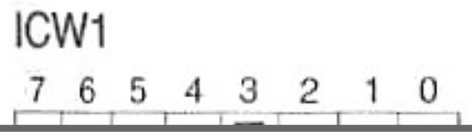


Off7..Off3: programmierbarer Offset des Interrupt-Vektors

OOStuBS setting:

Master	Slave
00100000	00101000
=32	=40

PIC Initialization – Part 1




```
; Reprogram the PICs (programmable interrupt controllers) to have
; all 15 hardware interrupts in sequence in the IDT.

reprogram_pics:
        mov     al,0x11   ; ICW1: 8086 mode with ICW4
        out     0x20,al
        call    delay
        out     0xa0,al
        call    delay
        mov     al,0x20   ; ICW2 master: IRQ # offset (32)
        out     0x21,al
        call    delay
        mov     al,0x28   ; ICW2 slave: IRQ # offset (40)
        out     0xa1,al
        call    delay
```

OOStuBS setting:

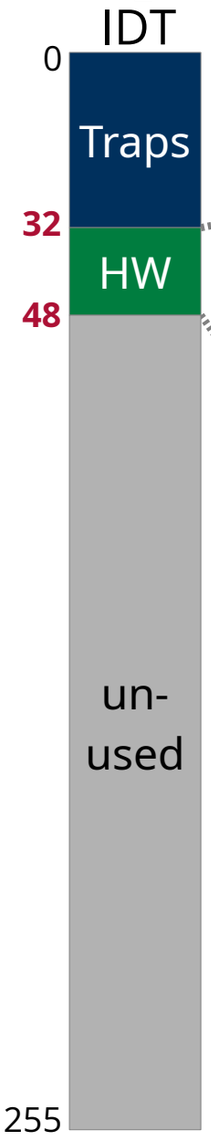
Master	Slave
0001	00010001

OOStuBS setting:

ster	Slave
0000	00101000
=32	=40

Mapping of HW IRQs (OOStuBS)

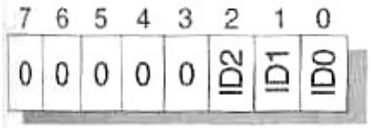
Standard AT IRQ mapping



IRQ	Description
0	Programmable Interrupt Timer (PIT)
1	Keyboard
2	(PIC Cascade)
3	COM2
4	COM1
5	LPT2
6	Floppy-Disk Drive
7	LPT1 / spurious interrupt
8	CMOS Real-Time Clock
9	
10	
11	
12	PS/2 Mouse
13	FPU / Coprocessor / Inter-Processor
14	Primary ATA HDD
15	Secondary ATA HDD

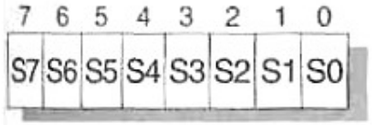
PIC Initialization – Part 2

ICW3 (Slave)



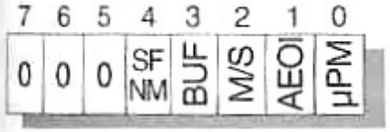
ID2..ID0: Identifizierungsnummer des Slave-PIC

ICW3 (Master)



S7..S0: 0=zugehörige IR-Leitung ist mit Peripheriegerät verbunden oder frei
1=zugehörige IR-Leitung ist mit Slave-PIC verbunden

ICW4



SFNM:	0=kein Special-Fully-Nested-Modus	1=Special-Fully-Nested-Modus
BUF:	0=kein gepufferter Modus	1=gepufferter Modus
M/S:	0=Slave-PIC	1=Master-PIC
AEOI:	0=manueller EOI	1=automatischer EOI
µPM:	0=Betrieb im MCS-80/85-Modus	1=Betrieb im 8086/88-Modus

OOStuBS setting:

Master	Slave
00000100	0000010

IRQ 2 → Slave **ID 2**

OOStuBS setting:

Master	Slave
00000011	00000011

PIC Initialization – Part 2

ICW3 (Slave)

7	6	5	4	3	2	1	0
0	0	0	0	0	ID2	ID1	ID0

```
...
mov     al, 0x04    ; ICW3 master: slaves with IRQs
out     0x21, al
call    delay
mov     al, 0x02    ; ICW3 slave: connected to master's IRQ2
out     0xa1, al
call    delay
mov     al, 0x03    ; ICW4: 8086 mode and automatic EOI
out     0x21, al
call    delay
out     0xa1, al
call    delay
...
```

OOStuBS setting:

Master	Slave
00100	0000010
→ Slave	ID 2

OOStuBS setting:

ster	Slave
00011	00000011

SFNM:	0=kein Special-Fully-Nested-Modus	1=Special-Fully-Nested-Modus
BUF:	0=kein gepufferter Modus	1=gepufferter Modus
M/S:	0=Slave-PIC	1=Master-PIC
AEOI:	0=manueller EOI	1=automatischer EOI
µPM:	0=Betrieb im MCS-80/85-Modus	1=Betrieb im 8086/88-Modus

PIC Programming

OCW1

7	6	5	4	3	2	1	0
M7	M6	M5	M4	M3	M2	M1	M0

M7..M0: 0=zugehörige IRQ-Leitung ist nicht maskiert
1=zugehörige IRQ-Leitung ist maskiert

Interrupt mask (IMR)

- read and write via Port 0x21 / 0xa1

OCW2

7	6	5	4	3	2	1	0
R	SL	EOI	0	0	L2	L1	L0

000: im AEOI-Modus rotieren
001: nicht-spezifischer EOI-Befehl
010: kein Vorgang (NOP)
011: spezifischer EOI-Befehl (mit L2..L0)
100: im AEOI-Set-Modus rotieren
101: bei nicht-spezifischem EOI-Befehl rotieren
110: Prioritätsbefehl setzn
111: bei spezifischem EOI-Befehl rotieren

OCW3

7	6	5	4	3	2	1	0
0	ESMM	SMM	0	1	P	RR	RIS

ESMM, SMM: 00=kein Vorgang (NOP) 01=kein Vorgang (NOP)
10=spez. Maske löschen 11=spez. Maske setzen
RR, RIS: 00=kein Vorgang (NOP) 01=kein Vorgang (NOP)
10=IRR lesen 11=ISR lesen
P: Polling: 0=kein Polling 1=Polling-Modus

PIC Programming

OCW1

7	6	5	4	3	2	1	0
M7	M6	M5	M4	M3	M2	M1	M0

M7..M0: 0=zugehörige IRQ-Leitung ist nicht maskiert
1=zugehörige IRQ-Leitung ist maskiert

Interrupt mask (IMR)

- read and write via Port 0x21 / 0xa1

OCW2

7	6	5	4	3	2	1	0
R	SL	EOI	0	0	L2	L1	L0

000: im AEOI-Modus rotieren
001: nicht-spezifischer EOI-Bef
010: kein Vorgang (NOP)
011: spezifischer EOI-Befehl (m
100: im AEOI-Set-Modus rotier
101: bei nicht-spezifischem EO
110: Prioritätsbefehl setzn
111: bei spezifischem EOI-Befe

```
...
mov     al, 0xff    ; Mask/disable hardware interrupts
out     0xa1, al    ; in the PICs. Only interrupt #2, which
call    delay       ; serves for cascading both PICs, is
mov     al, 0xfb    ; allowed.
out     0x21, al

ret
```

OCW3

7	6	5	4	3	2	1	0
0	ESMM	SMM	0	1	P	RR	RIS

ESMM, SMM: 00=kein Vorgang (NOP) 01=kein Vorgang (NOP)
10=spez. Maske löschen 11=spez. Maske setzen
RR, RIS: 00=kein Vorgang (NOP) 01=kein Vorgang (NOP)
10=IRR lesen 11=ISR lesen
P: Polling: 0=kein Polling 1=Polling-Modus

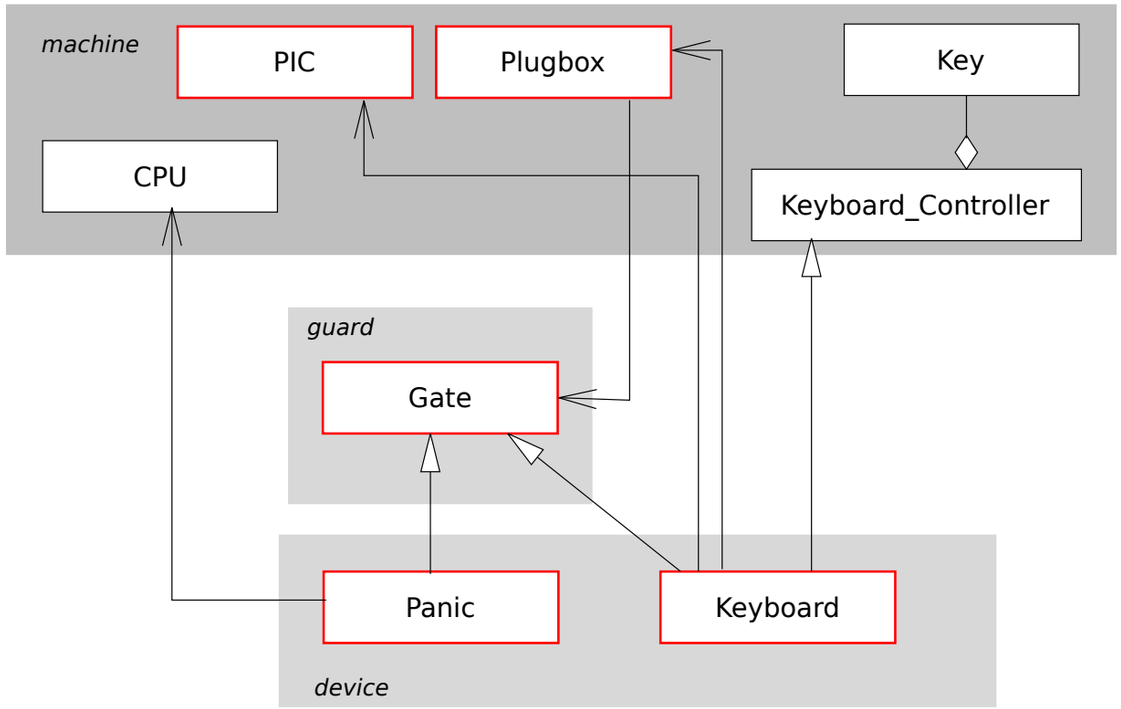
Overview

- Traps
- Lab Task #2
  - What does startup.asm do?
  - **What should happen in parts c) and d)?**
  - Repetition: **References** in C++
    - Plugbox and Gate
- Lab Task #3
  - Introduction
  - Repetition: **Pointers** in C++
    - The (weird?) Queue Class

Overview

- Traps
- Lab Task #2
  - What does startup.asm do?
  - What should happen in parts c) and d)?
  - **Repetition: References in C++**
    - Plugbox and Gate
- Lab Task #3
  - Introduction
  - Repetition: **Pointers** in C++
    - The (weird?) Queue Class

Interrupt Handler in OOStuBS




```
// ASSIGN: Plug in a handler routine, provided in the form of a Gate obj.
void assign(unsigned int slot, Gate& gate);

// REPORT: Retrieve the Gate object for the specified slot.
Gate& report (unsigned int slot);
```

How Do C++ References Work?

- Semantically: **Aliases** for objects
- Technically: Initialized, immutable **pointers**
  - Upon **initialization** of a reference, the compiler automatically **takes the address** of the initializing object.
  - When using a reference **in an expression**, automatically the referenced object is used.

```
int v1;
int &ref = v1;
int v2 = ref;

int &f(int &refarg) {
  refarg = 42;
  return refarg;
}

int v3 = f(v2);
```

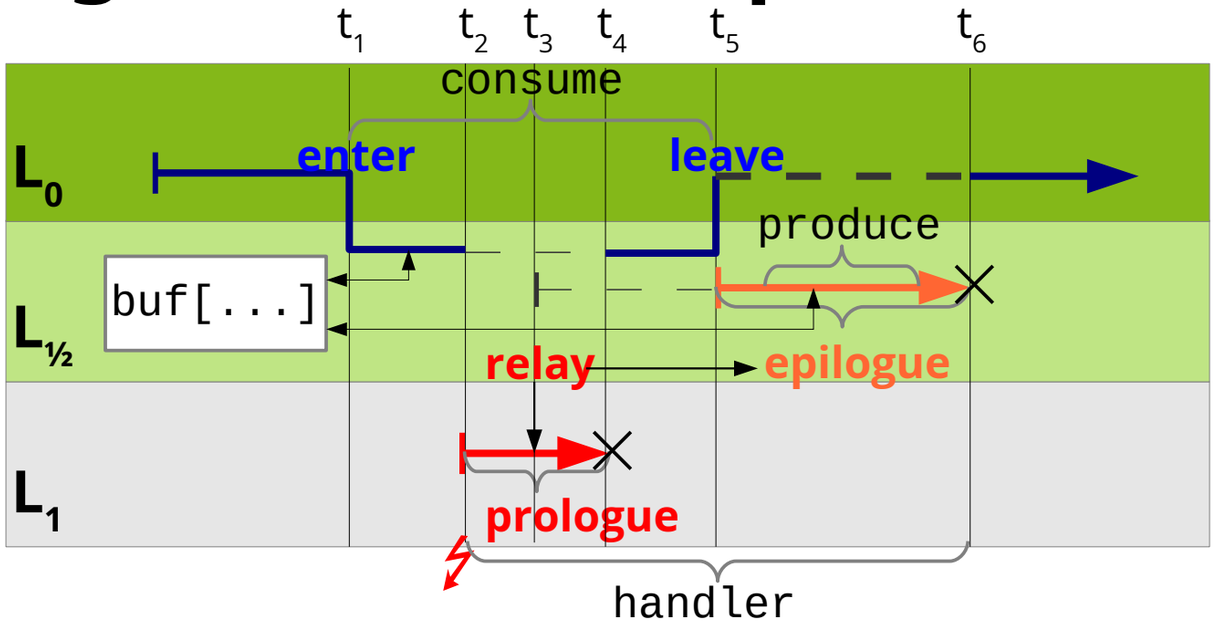
```
int v1;
int *ref = &v1;
int v2 = *ref;

int *f(int *refarg) {
  *refarg = 42;
  return &*refarg;
}

int v3 = *f(&v2);
```

technically equivalent

Overview

- Traps
- Lab Task #2
  - What does startup.asm do?
  - What should happen in parts c) and d)?
  - Repetition: **References** in C++
    - Plugbox and Gate
- **Lab Task #3**
  - Introduction
  - Repetition: **Pointers** in C++
    - The (weird?) Queue Class

Pro/Epilogue Model – Sequence Example

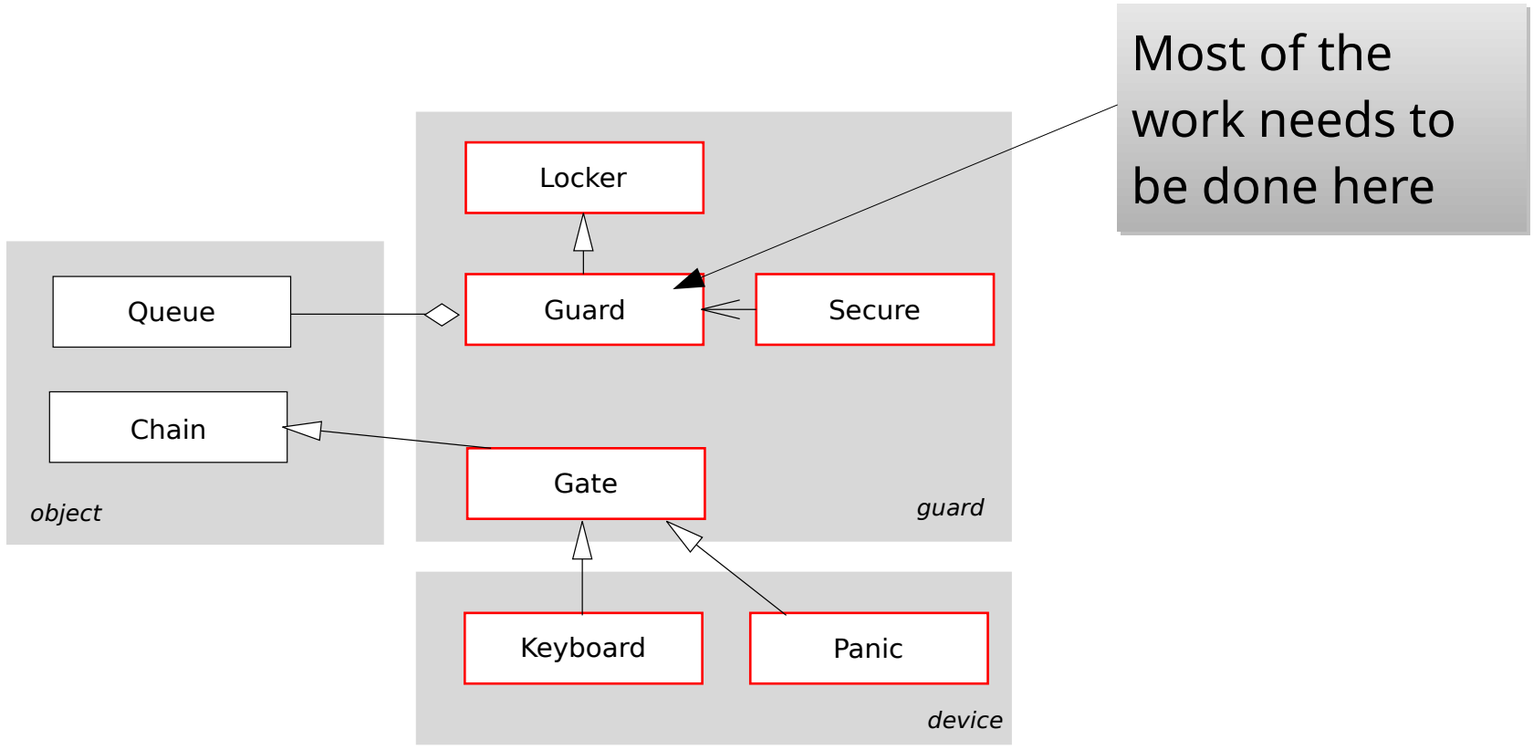


L_1 interrupts are never disabled.

Interrupt-handler activation latency is minimal.

1 Application control flow enters epilogue level $L_{½}$ (**enter**).
2 Interrupt is signaled on level L_1, execute prologue.
3 Prologue requests epilogue for delayed execution (**relay**).
4 Prologue terminates, interrupted $L_{½}$ control flow (application) continues.
5 Application control flow leaves epilogue level $L_{½}$ (**leave**), process meanwhile accumulated epilogues.
6 Epilogue terminates, application control flow continues on L_0.

Lab Task #3: Pro/Epilogue Model



Tricky Pointers: Queue in Task #3

- Queue elements inherit from class Chain
  - Thereby, they inherit a pointer to the next element
- A Queue object contains
  - a pointer to the first element
  - **a pointer to a pointer called ‘tail’?!**

```
class Chain {
public:
      Chain* next;
};
```

```
class Queue {
      Chain* head;
      Chain** tail;
public:
      Queue () { head = 0; tail = &head; }
      void enqueue (Chain* item);
      Chain* dequeue ();
      void remove (Chain*);
};
```

Tricky Pointers: Queue in Task #3

- ‘tail’ is a pointer to the ‘next’ pointer in the last element
  - This simplifies enqueueing!