Übung zu Betriebssysteme

Aufgabe 2 - Interrupts

Daniel Danner

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität
Erlangen Nürnberg

23. Oktober 2014

Agenda: IRQ-Behandlung in OO/MPStuBS

- Interrupts und Traps beim x86 Prozessor
  - Die Interrupt-Deskriptor-Tabelle (IDT)
  - Der Aufbau der IDT
  - Traps und Hardware-IRQs
- Externe Interrupts mit dem APIC
  - Aufbau eines modernen PC-Systems mit Local APIC und IO-APIC
  - Programmierung des IO-APIC
  - Verteilung von Interrupts in Multiprozessorsystemen
- Interruptbehandlung in OOStuBS/MPStuBS
  - Aufbau und Ablauf
  - Kontextsicherung

Intel 80386: Interrupt Deskriptor Tabelle (IDT)

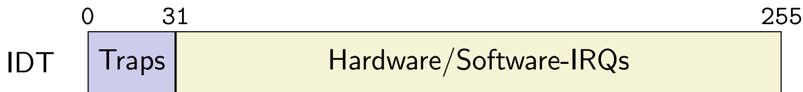
- Maximal 256 Einträge
  - Basisadresse und Größe(Limit) in IDTR
- 8 Byte pro Eintrag (Gate)
- Verschiedene Gate-Typen:
  - Task-Gate (Hardwaretasks)
  - Trap-Gate (Ausnahmebehandlung)
  - Interrupt-Gate (Ausnahmebehandlung + `cli`)

```
Speicher
IRQ-Handler

Interrupt-
Deskriptortabelle
Deskriptor 0xff
...
Deskriptor 0x03
Deskriptor 0x02
Deskriptor 0x01
Deskriptor 0x00

IDTR
INT 2
Limit
Basis
```

Intel 80386: Interruptvektoren in der IDT



Number	Description
0	Divide-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
9	Coprocessor Segment Overrun
10	Invalid TSS
11	Segment not Present
12	Stack Exception
13	General Protection Fault
14	Page Fault
15	Reserved
16	Floating-Point Error
17	Alignment Check
18	Machine Check
19-31	Reserved by Intel

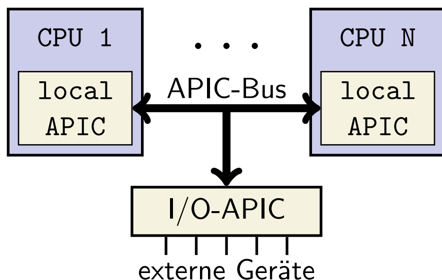
- Einträge 0-31 Traps (fest)
- Traps/Exceptions (Intel 'Lingo')
  - Synchron zum Kontrollfluss
  - Beispiele:
    Division durch Null
    Seitenfehler
    Schutzfehler
    Breakpoint
    ...
- Einträge 32-255 für IRQs
- Auslösen durch
  - Software: (`int <vec#>`)
  - Hardware: Externe Geräte
    APIC löst zugeordnete Unterbrechung aus

Hardware-IRQs bei x86-CPUs

- Traditionell nur eine INT-Leitung (`INT`) an der CPU
  - `INT` kann durch Prozessorbefehle maskiert werden
    - `cli` (Clear Interrupt Enable) Interruptleitung sperren
    - `sti` (Set Interrupt Enable) Interruptleitung freigeben
  - Anschluss von mehreren externen Geräten durch externen Controller
    - Traditionell ist dies der Programmable Interrupt Controller (`PIC 8259A`)
    - Bietet u.a. keine Unterstützung für Mehrprozessorsysteme

- APIC-Architektur bei „modernen“ PCs
  - Bietet mehr Anschlussmöglichkeiten für externe Geräte (24 vs. 15)
  - Freie Konfigurierbarkeit des Interruptvektors pro Gerät möglich
  - Wird für Multiprozessorsysteme auf jedenfall benötigt (MPStuBS!)
  - Funktioniert natürlich auch in einem Einprozessorsystem (OOStuBS)

Aufbau der APIC-Architektur

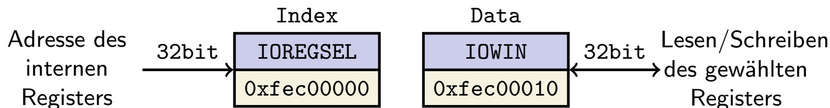


Aufteilung in lokalen APIC und I/O APIC:

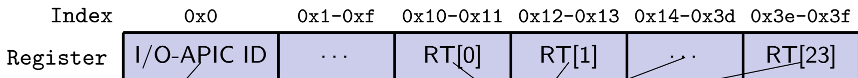
- I/O-APIC dient zum Anschluss der Geräte
  - Zuweisung von beliebigen Vektornummern
  - Aktivieren und Deaktivieren von einzelnen Interruptquellen
  - Zuweisung von Zielprozessoren für einzelnen Interrupts in MP-Systemen
- Interrupts werden zu Nachrichten auf dem APIC-Bus
- Local APIC
  - Verbindet eine CPU mit dem APIC-Bus
  - Liest Nachrichten vom APIC-Bus und unterbricht die CPU
  - Muss Interrupts explizit quittieren (ACK)

Programmierung des Intel I/O-APIC

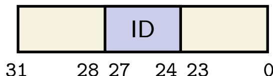
- Zugriff auf die internen Register über memory-mapped Ein-/Ausgabe
  - Jedoch keine direkte Abbildung von internen Registern auf Adressen
  - „Umweg“über ein Index- und Datenregister



- Interne Register des I/O-APICs



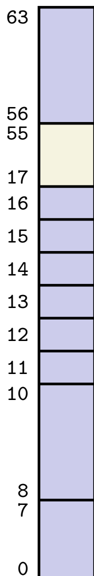
ID auf dem APIC-Bus
Aufbau:



Redirection Table

- Ein Eintrag für jede Interruptquelle
- Konfiguration dadurch pro Interruptquelle
- Zwei interne Register pro Eintrag (64bit)

Aufbau eines Redirection Table Eintrags



Destination Field: Zieladresse des IRQs

- APIC ID der Ziel-CPU falls Dest. Mode == Physical
- Gruppe von Ziel-CPUs falls Dest. Mode == Logical

reserviert

Interrupt-Mask: Interrupt aktiv (0) oder inaktiv (1)
Trigger Mode: Flanken-(0) oder Pegel-(1)steuerung
Remote IRR: Art der erhaltenen Bestätigung
Interrupt Polarity: Active High (0) bzw. Active Low (1)
Delivery Status: Interrupt Nachricht noch unterwegs?
Destination Mode: Physical (0) oder Logical (1) Mode
Delivery Mode: Modus der Nachrichtenzustellung

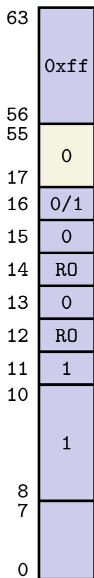
- 0: Fixed - Signal allen Zielprozessoren zustellen
- 1: Lowest Priority - CPU mit niedrigster Priorität ist Ziel
- ...

Interrupt Vektor: Nummer in der Vektortabelle (32-255)

Programmierung des APIC Systems

- Wo ist welches Gerät angeschlossen?
  - Das kann evtl. unterschiedlich sein von Rechner zu Rechner!
  - Steht in der Systemkonfiguration, heutzutage typischerweise ACPI
  - Klasse `APICSystem` stellt die relevanten Teile diese Informationen bereit:
    - Methode `APICSystem::getIOAPICSlot` liefert für jedes Gerät den Index in die Redirection Table
    - Methode `APICSystem::getIOAPICID` liefert die ID des IOAPICs
- Adressierung der APIC Nachrichten in OOStuBS/MPStuBS
  - Zusammenspiel mehrerer Faktoren:
    - Destination Mode, Destination Field und Delivery Mode im I/O-APIC
    - Prozessor Priorität in den Local APICs der einzelnen CPUs
  - Ziel: Gleichverteilung der Interrupts auf alle CPUs
    - Priorität der Prozessoren im Local APIC fest auf `0` einstellen
    - Im I/O-APIC `Lowest Priority` als Delivery Mode verwenden
    - Verwendung des `Logical Destination Mode`; bis zu 8 CPUs adressierbar
    - Destination Field auf `0xff`: Potentiell jede CPU kann Empfänger sein

Aufbau eines Redirection Table Eintrags



Destination Field: Zieladresse des IRQs

- APIC ID der Ziel-CPU falls Dest. Mode == Physical
- Gruppe von Ziel-CPUs falls Dest. Mode == Logical

reserviert

Interrupt-Mask: Interrupt aktiv (0) oder inaktiv (1)
Trigger Mode: Flanken-(0) oder Pegel-(1)steuerung
Remote IRR: Art der erhaltenen Bestätigung
Interrupt Polarity: Active High (0) bzw. Active Low (1)
Delivery Status: Interrupt Nachricht noch unterwegs?
Destination Mode: Physical (0) oder Logical (1) Mode
Delivery Mode: Modus der Nachrichtenzustellung

- 0: Fixed - Signal allen Zielprozessoren zustellen
- 1: Lowest Priority - CPU mit niedrigster Priorität ist Ziel
- ...

Interrupt Vektor: Nummer in der Vektortabelle (32-255)

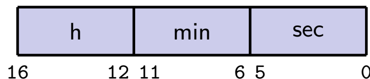
Einstellung in OOStuBS/MPStuBS (RO: Read Only)

Einschub: Ansprechen von einzelnen Bits

- Mittels Bitmasken und logischen Bitoperationen
  (Das solltet ihr kennen oder zumindest schon gesehen haben)
- Weniger verbreitet sind Bitfelder in C/C++:

```
struct time {
    unsigned int sec : 6,
                 min : 6,
                   h : 5;
};
int foo(struct time* t) {
    t->sec = 42;
    return t->h;
}
```

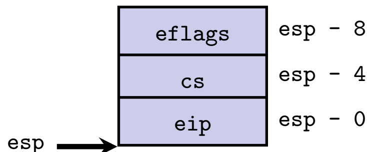
Layout mit GCC auf x86:



- Anordnung der Member jedoch compilerspezifisch
- Dies gilt jedoch für Systemsoftware (in gewissen Maßen) sowieso!

Zustandssicherung beim Auftreten von Interrupts

- Minimaler zu sichernder Zustand?
- CPU sichert automatisch:
  - Condition Codes (`eflags`)
  - Aktuelles Code Segment (`cs`)
  - Programmzeiger/Rücksprungadresse (`eip`)



- Wiederherstellung des ursprünglichen Prozessorzustandes
  - Spezieller Befehl (`iret`) stellt den gesicherten Zustand wieder her.

Zustandssicherung beim Auftreten von Interrupts (2)

- `eflags`, `cs` und `eip` wurden bereits von der CPU gesichert
- 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
  - Lösung 1: *Aufrufende* Funktion sichert alle Register, die sie braucht
  - Lösung 2: *Aufgerufene* Funktion sichert alle Register, die sie verändert
  - Lösung 3: Ein Teil der Register wird vom Aufrufer, ein anderer Teil vom Aufgerufenen gesichert

Kontextsicherung in Hochsprachen

- In der Praxis wird Lösung 3 verwendet
  - Aufteilung ist grundsätzlich compilerspezifisch
  - CPU-Hersteller definiert jedoch Konventionen, damit Interoperabilität auf Binärcodeebene sichergestellt ist

- Aufteilung der Register in 2 Subsets
  - Flüchtige Register („scratch registers“)
    - Compiler geht davon aus, dass Unterprogramm den Inhalt verändert
    - *Aufrufer* muss Inhalt gegebenenfalls sichern
    - Beim x86 sind **eax**, **ecx**, **edx** und **eflags** als flüchtig definiert
  - Nicht-flüchtige Register („non-scratch registers“)
    - Compiler geht davon aus, dass der Inhalt nicht verändert wird
    - *Aufgerufene* Funktion muss Inhalt gegebenenfalls sichern
    - Beim x86 sind alle sonstigen Register als nicht-flüchtig definiert: **ebx**, **esi**, **edi**, **ebp** und **esp**

Interrupt-Handler müssen auch flüchtige Register sichern!

Interrupthandler in OOStuBS/MPStuBS

- Behandlung startet in der Funktion `guardian(int)`
  - bekommt die IRQ-Nummer als Parameter:

```
void guardian(unsigned int vector) {
    ...//IRQ-Handler (Gate) aktivieren
}
```

  - Interrupts sind während der Abarbeitung gesperrt
    - können mit `sti` manuell wieder zugelassen werden
    - werden automatisch wieder zugelassen, wenn `guardian(int)` terminiert

- Die eigentlichen (spezifischen) IRQ-Handler
  - sind Instanzen der Klasse *Gate*
  - werden an- und abgemeldet über die Klasse *Plugbox*

Interrupthandler in OOStuBS/MPStuBS

Key

LAPIC

CPU

Keyboard_Controller

IOAPIC

Plugbox

Gate

Keyboard

Panic