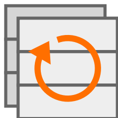


TUHH
Technische Universität Hamburg

BSB – Übung 2: Interrupts und Synchronisation

Yannick Loeck

2022-04-27

Inhalt

BSB - HÜ2

Wir wollen nicht nur auf die Tastatur warten!

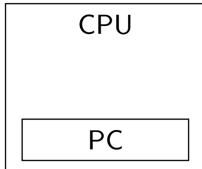
Worum geht es in der Übung heute:

- Interrupts
- Synchronisation mit Locks

Interrupts

Minimale CPU

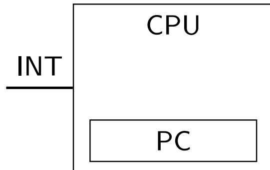
TUHH
Technische Universität Hamburg




```
while(1) {
  PC = PC + 1;
  opcode = *PC;
  execute(opcode);
}
```

- Ein Register: PC/IP
- Endlosschleife
  - PC inkrementieren
  - Instruktion lesen
  - Ausführen

Minimale CPU

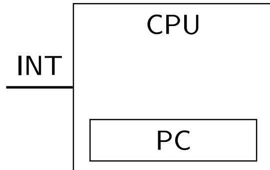


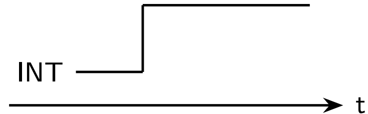

```
while(1) {
  PC = PC + 1;
  if (INT)
    PC = ISR_addr;
  opcode = *PC;
  execute(opcode);
}
```

- Schleife unterbrechen: Interrupt
- PC überschreiben
- Interrupt Service Routine

Minimale CPU

TUHH
Technische Universität Hamburg

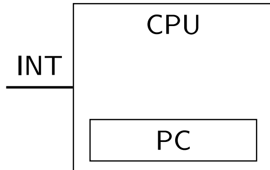


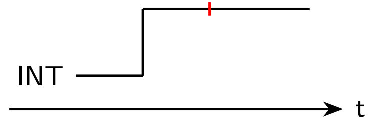



```
while (1) {
  PC = PC + 1;
  if (INT)
    PC = ISR_addr;
  opcode = *PC;
  execute(opcode);
}
```

- Schleife unterbrechen: Interrupt
- PC überschreiben
- Interrupt Service Routine

Minimale CPU

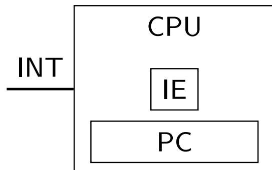


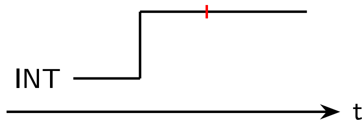



```
while(1) {
  PC = PC + 1;
  if (INT)
    PC = ISR_addr;
  opcode = *PC;
  execute(opcode);
}
```

- INT bleibt auf 1, was dann?

Minimale CPU



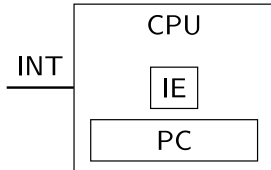


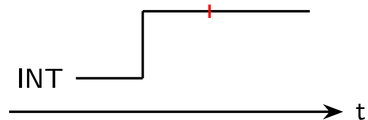

```
while(1) {
  PC = PC + 1;
  if (INT && IE)
    PC = ISR_addr;
    IE = 0;
  opcode = *PC;
  execute(opcode);
}
```

- INT bleibt auf 1, was dann?
- Interrupt Enable Flag
- Clear Interrupt `cli`
- Set Interrupt `sti`
- StuBS: `Core::Interrupt::`
  - `disable()`
  - `enable()`

Minimale CPU

TUHH Technische Universität Hamburg



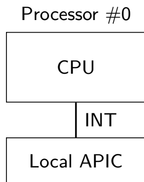



```
while(1) {
  PC = PC + 1;
  if (INT && IE)
    PC = ISR_addr;
    IE = 0;
  opcode = *PC;
  execute(opcode);
}
```

- Wohin wenn die ISR fertig ist?
- PC wurde überschrieben
- Kontext muss gesichert werden

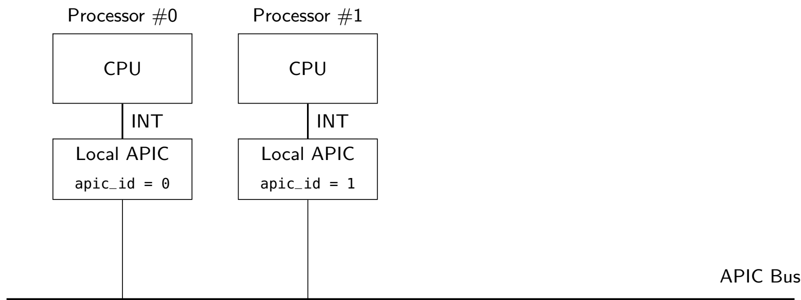
Interrupt-Architektur unter X86

TUHH
Technische Universität Hamburg



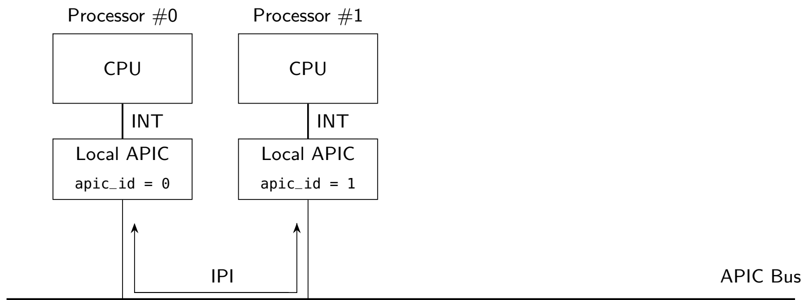
- Intel 8259 PIC (1976)
- APIC: Advanced Programmable Interrupt Controller
- Local APIC an CPU

Interrupt-Architektur unter X86



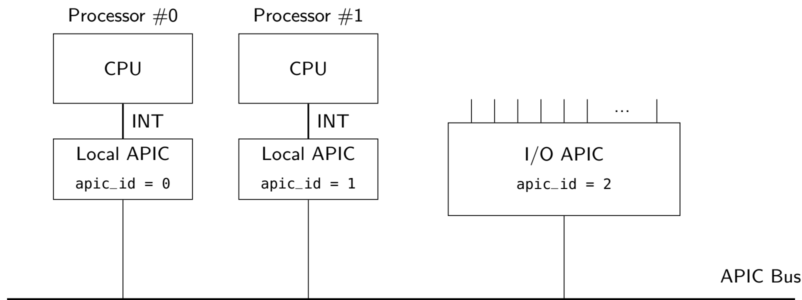
- Mehrere CPUs mit eigenem Program Counter
- LAPICs an Bus angeschlossen

Interrupt-Architektur unter X86



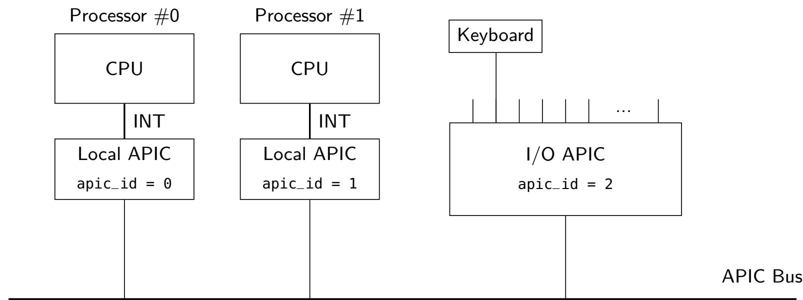
- Inter Processor Interrupts über Bus
- `apic_id` zur Identifikation

Interrupt-Architektur unter X86



- Externe Interrupts: IOAPIC
- 24 Eingänge

Interrupt-Architektur unter X86



- Geräte wie Tastatur an IOAPIC angeschlossen
- In StuBS: `APIC::getIOAPICSlot(APIC::KEYBOARD)`
- IOAPIC übersetzt Interrupt Request und schickt auf Bus

IOAPIC ↦ LAPIC

- Mapping von IOAPIC-Eingängen zu Zielen
- *Redirection Table* mit 24 Einträgen
- In Registern im IOAPIC
- 8 Byte Einträge, auf 2 32-bit Werte aufgeteilt

Redirection Table Entries	0	1	2	...	23
	0x10 0x11	0x12 0x13	0x14 0x15		

IOAPIC ↦ LAPIC

- Kein direkter Zugriff auf Einträge
- Dafür 2 Memory-Mapped Register

IOREGSEL
`0xfec00000`

IOWIN
`0xfec00010`

Redirection Table Entries

0	1	2	...	23

`0x10` `0x11` `0x12` `0x13` `0x14` `0x15`

IOAPIC ↦ LAPIC

- Für Zugriff auf Tabelleneintrag:
- `IOREGSEL` schreiben (Eintrag 1 z.B. an `0x12` und `0x13`)
- `IOWIN` lesen oder schreiben

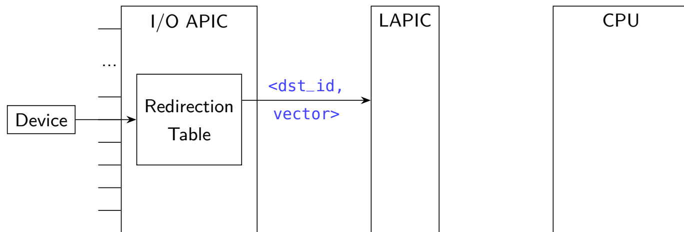
Redirection Table Entries

Bits	Feld
0-7	Vector
8-10	Delivery Mode
11	Destination Mode
12	Delivery Status
13	Polarity
14	Remote IRR
15	Trigger Mode
16	Interrupt Mask
17-55	Reserviert
56-63	Destination

- 8 bit Vektornummer: Identifiziert passenden Interrupt Handler
- 8 bit Destination: Bit-Maske für Ziel-CPUs
- 3 bit Delivery Mode: fixed / lowest priority
- 1 bit Mask: Interrupts an diesem Eingang ignorieren (vs. IE flag)

Aufgabe 2(a)

TUHH
Technische Universität Hamburg



- I/O APIC konfigurieren
  - IOAPICID richtig setzen
  - Redirection Table füllen

LAPIC $\mapsto$ CPU

- LAPIC reicht Vektornummer weiter
- CPU wählt ISR aus *Interrupt Descriptor Table*
- 256 8-Byte Einträge enthalten ISR-Funktionspointer (u.a.)
- 0-31 für Traps
- 32-255 für Hardware Interrupts
- Auch möglich: Software Interrupts (z.B. syscalls)

StuBS

Interrupts könnt ihr frei belegen.

Wiederholung - Traps vs. Interrupts

	Trap	**Interrupt**
Ursprung	von Programmausführung	von Hardware
Art	synchron	asynchron
Beispiele	General Protection Fault, Division by 0	Tastatur, Timer

An der CPU

- In StuBS bereits gegeben: `interrupt_entry_0`
- Kontext sichern
- `interrupt_handler` aufrufen
- Kontext wiederherstellen
- `iret`
- **Wichtig!** CPU sendet Ack: EOI (end of interrupt)
- LAPIC gibt das an IOAPIC zurück
- Vor EOI: Keine neuen Interrupt-Nachrichten an dem IOAPIC-Slot

Interrupt Handler und Plugbox in StuBS

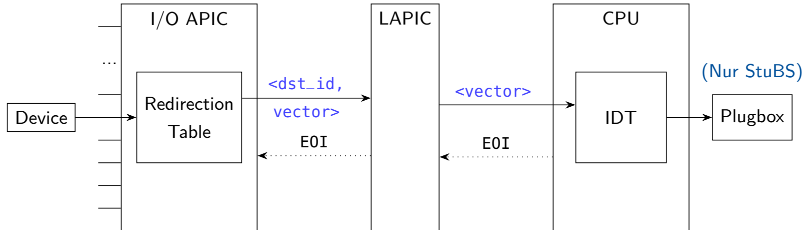
TUHH Technische Universität Hamburg

- Interrupt Handler bekommt `Vector` und `CPU::Context`
- Kontext: Zustand vor Interrupt (für Debugging)
- Eure Aufgabe: Vektortabelle (Plugbox) implementieren
- Array mit Pointern auf `Gate`-Objekte

Beispiel: Tastatur

- Vektornummer in `Core::Interrupt::Vector` definieren
- `Keyboard`-Klasse von `Gate` erben lassen
- Interrupt-Behandlungsfunktion in `trigger()` definieren
- Vor Ende: ACK mit `endOfInterrupt()`

Aufgabe 2(b)



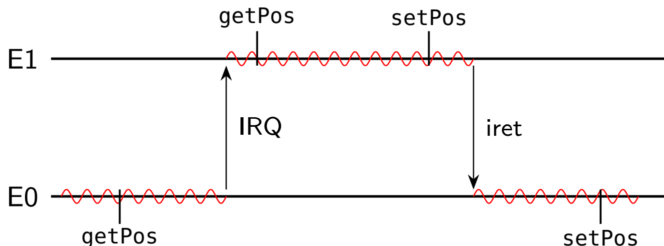
- I/O APIC konfigurieren
  - IOAPICID richtig setzen
  - Redirection Table füllen
- Plugbox erstellen: Mapping von Interrupt-Vektoren zu Gates
- Tastaturtreiber anpassen: Interrupts statt Polling

Synchronisation

Warum überhaupt Synchronisation?

TUHH Technische Universität Hamburg

- Interrupts können jederzeit ankommen: Nebenläufigkeit
- Anwendungsebene E0 und ISR-Ebene E1
- Können auf gleichen Speicher zugreifen
- Beispiel: CGA-Cursor, write-write-Konflikte



Synchronisation

- Wir wollen Serialisierbarkeit
- Operationen wie wenn nacheinander ausgeführt

- Critical Sections identifizieren
- Während dieser Interrupts ausschalten (`cli`)
- Einseitige Synchronisation

Synchronisation

TUHH
Technische Universität Hamburg

ISR

```
ISR() {
  kout << getKey();
}
```

CPU 0

```
while (1) {
  kout << i++;
}
```

Synchronisation

ISR

```
ISR() {
  kout << getKey();
}
```

← block

CPU 0

```
while (1) {
  cli();
  kout << i++;
  sti();
}
```

- In OOStuBS sind wir fertig

Synchronisation

TUHH Technische Universität Hamburg

ISR

```
ISR() {
  kout << getKey();
}
```

← block

CPU 0

```
while (1) {
  cli();
  kout << i++;
  sti();
}
```

CPU 1

```
while (1) {
  cli();
  kout << j++;
  sti();
}
```

- `cli` und `sti` sind nur CPU-lokal

Synchronisation

TUHH
Technische Universität Hamburg

ISR

```
ISR() {
  kout << getKey();
}
```

← block

CPU 0

```
while (1) {
  cli();
  kout << i++;
  sti();
}
```

CPU 1

```
while (1) {
  kout << j++;
}
```

Synchronisation

TUHH Technische Universität Hamburg

ISR

```
ISR() {
  kout << getKey();
}
```

← block

CPU 0

```
while (1) {
  cli();
  kout << i++;
  sti();
}
```

SYNC

```
int x;
```

CPU 1

```
while (1) {
  lock();
  kout << j++;
  unlock();
}
```

Synchronisation

TUHH Technische Universität Hamburg

ISR

```
ISR() {
  kout << getKey();
}
```

block

CPU 0

```
while (1) {
  cli();
  kout << i++;
  sti();
}
```

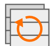
SYNC

```
int x;

lock() {
  while (x != 0) {}
  x = 1;
}
```

CPU 1

```
while (1) {
  lock();
  kout << j++;
  unlock();
}
```



Synchronisation

TUHH
Technische Universität Hamburg

ISR

```
ISR() {
  kout << getKey();
}
```

← block

CPU 0

```
while (1) {
  cli();
  kout << i++;
  sti();
}
```

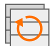
SYNC

```
int x;

lock() {
  while (x != 0) {}
  x = 1;
}
unlock () {
  x = 0;
}
```

CPU 1

```
while (1) {
  lock();
  kout << j++;
  unlock();
}
```



Synchronisation

ISR

```
ISR() {
  kout << getKey();
}
```

block

CPU 0

```
while (1) {
  cli();
  kout << i++;
  sti();
}
```

SYNC

x

```
int x;

lock() {
  while (x != 0) {}
  x = 1;
}
unlock () {
  x = 0;
}
```

CPU 1

```
while (1) {
  lock();
  kout << j++;
  unlock();
}
```

Synchronisation

TUHH
Technische Universität Hamburg

ISR

```
ISR() {
  lock();
  kout << getKey();
  unlock();
}
```

CPU 0

```
while (1) {
  cli();
  kout << i++;
  sti();
}
```

SYNC

```
int x;

lock() {
  while (x != 0) {}
  x = 1;
}
unlock () {
  x = 0;
}
```

CPU 1

```
while (1) {
  lock();
  kout << j++;
  unlock();
}
```

ISR

```
ISR() {
  lock();
  kout << getKey();
  unlock();
}
```

CPU 0

```
while (1) {
  cli();
  kout << i++;
  sti();
}
```

lock()

unlock()

SYNC

```
int x;

lock() {
  while (x != 0) {}
  x = 1;
}
unlock () {
  x = 0;
}
```

x

CPU 1

```
while (1) {
  lock();
  kout << j++;
  unlock();
}
```

Synchronisation

TUHH
Technische Universität Hamburg

ISR

```
ISR() {
  lock();
  kout << getKey();
  unlock();
}
```

CPU 0

```
while (1) {
  lock();
  cli();
  kout << i++;
  sti();
  unlock();
}
```

SYNC

```
int x;

lock() {
  while (x != 0) {}
  x = 1;
}
unlock () {
  x = 0;
}
```

CPU 1

```
while (1) {
  lock();
  kout << j++;
  unlock();
}
```

Synchronisation

TUHH Technische Universität Hamburg

ISR

```
ISR() {
  lock();
  kout << getKey();
  unlock();
}
```

CPU 0

```
while (1) {
  lock();
  cli();
  kout << i++;
  sti();
  unlock();
}
```

SYNC

x

```
int x;

lock() {
  while (x != 0) {}
  x = 1;
}
unlock () {
  x = 0;
}
```

CPU 1

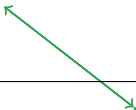
```
while (1) {
  lock();
  kout << j++;
  unlock();
}
```

Synchronisation

TUHH
Technische Universität Hamburg

ISR

```
ISR() {
  lock();
  kout << getKey();
  unlock();
}
```



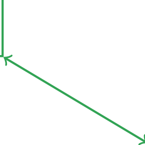
CPU 0

```
while (1) {
  cli();
  lock();
  kout << i++;
  sti();
  unlock();
}
```

SYNC

```
int x;

lock() {
  while (x != 0) {}
  x = 1;
}
unlock () {
  x = 0;
}
```



CPU 1

```
while (1) {
  lock();
  kout << j++;
  unlock();
}
```

Synchronisation

TUHH
Technische Universität Hamburg

ISR

```
ISR() {
  lock();
  kout << getKey();
  unlock();
}
```

CPU 0

```
while (1) {
  cli();
  lock();
  kout << i++;
  unlock();
  sti();
}
```

SYNC

```
int x;

lock() {
  while (x != 0) {}
  x = 1;
}
unlock () {
  x = 0;
}
```

CPU 1

```
while (1) {
  lock();
  kout << j++;
  unlock();
}
```

Synchronisation

TUHH
Technische Universität Hamburg

ISR

```
ISR() {
  lock();
  kout << getKey();
  unlock();
}
```

CPU 0

```
while (1) {
  cli();
  lock();
  kout << i++;
  unlock();
  sti();
}
```

SYNC

```
int x;

lock() {
  while (x != 0) {}
  x = 1;
}
unlock () {
  x = 0;
}
```

CPU 1

```
while (1) {
  lock();
  kout << j++;
  unlock();
}
```

Synchronisation

TUHH Technische Universität Hamburg

ISR

```
ISR() {
  lock();
  kout << getKey();
  unlock();
}
```

CPU 0

```
while (1) {
  cli();
  lock();
  kout << i++;
  unlock();
  sti();
}
```

SYNC

```
int x;

lock() {
  while (x != 0) {}
  x = 1;
}
unlock () {
  x = 0;
}
```

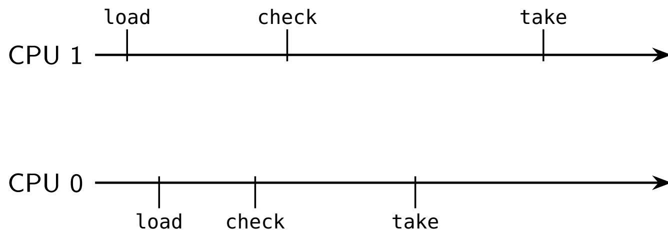
muss atomar sein!

CPU 1

```
while (1) {
  lock();
  kout << j++;
  unlock();
}
```

Atomare Operationen

TUHH
Technische Universität Hamburg

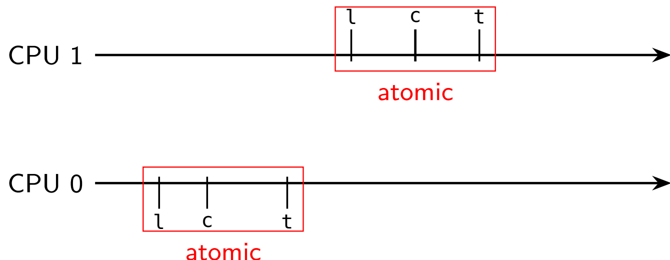


- Lesen vom Wert, Vergleich mit 0, Setzen auf 1
- Muss “zum selben Zeitpunkt” passieren
- Unmöglich in Software!

- Hardware bietet `cmpxchg`: Compare and Exchange
- Auf Hochsprachenebene: `__atomic_test_and_set()`
- `https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html`

Atomare Operationen

TUHH
Technische Universität Hamburg



- Lesen vom Wert, Vergleich mit 0, Setzen auf 1
- Muss “zum selben Zeitpunkt” passieren
- Unmöglich in Software!

- Hardware bietet `cmpxchg`: Compare and Exchange
- Auf Hochsprachenebene: `__atomic_test_and_set()`
- `https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html`

Test and Set


```
test_and_set(bool* lock) {
  bool test = *lock;
  if (test == 0) *lock = 1;
  return test;
}
```

- Liest Speicherwort
- Setzt auf 1 wenn es 0 war
- Gibt alten Wert zurück

Damit kann man ein Spinlock implementieren.

Ticketlocks

- Spinlocks sind unfair: Starvation
- Lösung: Lock das der bekommt der es zuerst probiert

- Ticket ziehen wie im Bürgeramt
- *"Ticketautomat"* inkrementiert wenn Ticket gezogen wurde
- *"Anzeige"* zeigt an wer aktuell dran ist
- Verlassen von critical section: Anzeige selbst inkrementieren
- **Wichtig:** Atomare Operationen benutzen!
  - `__atomic_fetch_add()`
  - `__atomic_load_n()`
  - `__atomic_store_n()`

Aufgabe 2

- I/O APIC konfigurieren
  - IOAPICID richtig setzen
  - Redirection Table füllen
- Plugbox erstellen: Mapping von Interrupt-Vektoren zu Gates
- Tastaturtreiber anpassen: Interrupts statt Polling
- Je nach Variante:
  - OOStuBS: Synchronisation von einem Kern und ISRs
  - MPStuBS: Synchronisation von mehreren Kernen und ISRs