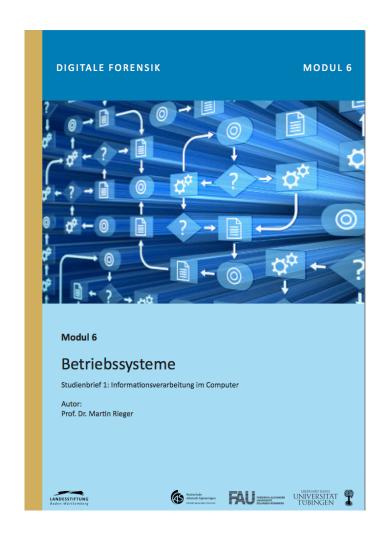


## Betriebssysteme

SS 2013

Hans-Georg Eßer Dipl.-Math., Dipl.-Inform.

SB 3 (14.03.2013) Speicherverwaltung Ein-/Ausgabegeräte und Schnittstellen



#### Studienbrief 3

- Speicherverwaltung
- Speicher-Forensik (Linux)
- Ein-/Ausgabe: Interrupts
- Ein-/Ausgabe: I/O Ports, DMA, Floppy Controller

## 3.5 Virtueller Speicher

#### **Motivation**

- Speicher zu knapp für große Programme
  - → Overlay-Programmierung
- Programmteile dynamisch nachladen, wenn sie benötigt werden
- Programmierer muss sich um Aufteilung in Overlays kümmern



### **Overlay-Programmierung**

### Turbo Pascal, um 1985-90:

```
program grossesprojekt;
overlay procedure kundendaten;
overlay procedure lagerbestand;
{ Hauptprogramm }
begin
 while input <> "exit" do begin
    case input of
      "kunden": kundendaten;
      "lager": lagerbestand;
    end;
  end;
end.
```





### Lösung des Problems

- Virtueller Speicher, der das gesamte Programm aufnehmen kann
- Programm sieht Speicherbereich, der ihm zur Verfügung gestellt wurde – wie viel wirklich vorhanden ist, spielt (für das Programm) keine Rolle

### **Speicherschutz**

- Speicherbereiche der Prozesse sollen durch das Betriebssystem von anderen Prozessen abgeschirmt werden
  - Frühe CPUs: Segmentierung des Speichers
  - Moderne CPUs: Virtueller Speicher (Paging); normale Prozesse dürfen Paging nicht deaktivieren
  - Ganz alte CPUs: Kein Schutz, darum auch keine stabilen / sicheren BS möglich



## Virtuelle Speicherverwaltung (Paging)

- Aufteilung des Adressraums in Seiten (pages) fester Größe und des Hauptspeichers in Seitenrahmen (page frames) gleicher Größe.
  - Typische Seitengrößen: 512 Byte bis 8192 Byte (immer Zweierpotenz).
- Der lineare, zusammenhängende Adressraum eines Prozesses ("virtueller" Adressraum) wird auf beliebige, nicht zusammenhängende Seitenrahmen abgebildet.
- BS verwaltet eine einzige Liste freier Seitenrahmen



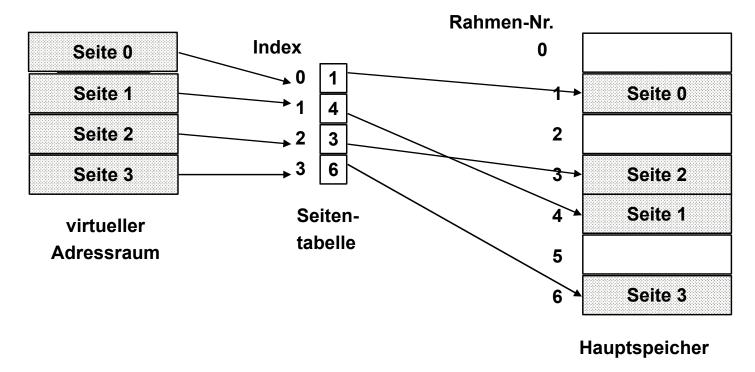
## Virtuelle Speicherverwaltung (Paging)

- Die Berechnung der physikalischen Speicheradresse aus der vom Programm angegebenen virtuellen Adresse
  - geschieht zur Laufzeit des Programms,
  - ist transparent f
     ür das Programm,
  - muss von der Hardware unterstützt werden.
- Vorteile der virtuellen Speicherverwaltung:
  - Einfache Zuteilung von Hauptspeicher.
  - Kein Aufwand für den Programmierer.



### Virtueller Adressraum (1)

 Paging stellt den Zusammenhang zwischen Programmadresse und physikalischer Hauptspeicheradresse erst zur Laufzeit mit Hilfe der Seitentabellen her.



### Virtueller Adressraum (2)

- Die vom Programm verwendeten Adressen werden deshalb auch virtuelle Adressen genannt.
- Der virtuelle Adressraum eines Programms ist der lineare, zusammenhängende Adressraum, der dem Programm zur Verfügung steht.

## Adressübersetzung beim Paging (1)

- Die Programmadresse wird in zwei Teile aufgeteilt:
  - eine Seitennummer
  - eine relative Adresse (offset) in der Seite

Beispiel: 32-bit-Adresse bei einer Seitengröße von 4096 (=2<sup>12</sup>) Byte:

31 12 11 0 Seitennummer offset

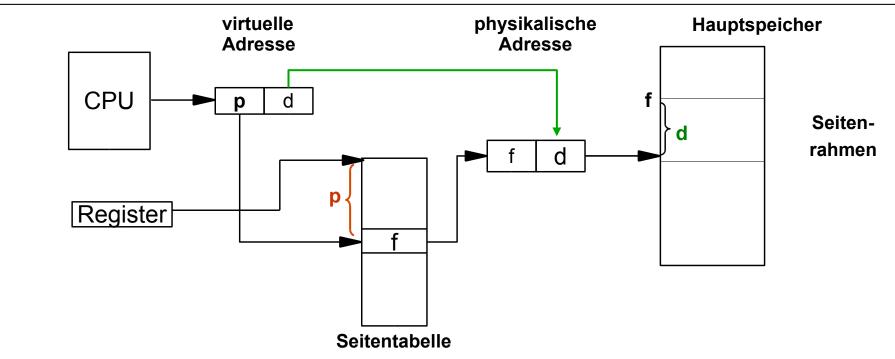


## Adressübersetzung beim Paging (2)

- Für jeden Prozess gibt es eine Seitentabelle (page table). Diese enthält für jede Prozessseite
  - eine Angabe, ob die Seite im Speicher ist,
  - die Nummer des Seitenrahmens im Hauptspeicher, der die Seite enthält.
- Das Page Table Register (PTR) enthält die Anfangsadresse der Seitentabelle für den aktuellen Prozess.
- Die Seitennummer wird als Index in die Seitentabelle verwendet.



## Adressübersetzung beim Paging (3)



- Für jeden Hauptspeicherzugriff wird ein zusätzlicher Hauptspeicherzugriff auf die Seitentabelle benötigt. Dies muss durch Caches in der Hardware beschleunigt werden.
- Seite nicht im Speicher → spezielle Exception, einen sog. page fault (Seitenfehler) auslösen.

# Virtueller Speicher allgemein (1)

- Mehr Prozesse können effektiv im Speicher gehalten werden
  - → bessere Systemauslastung
- Ein Prozess kann viel mehr Speicher anfordern als physikalisch verfügbar

# Virtueller Speicher allgemein (2)

- allgemeiner Vorgang:
  - Nur Teile des Prozesses befinden sich im physikalischen Speicher
  - falls Zugriff auf eine Adresse, die ausgelagert ist:
    - BS setzt den Prozess auf blockiert
    - BS setzt eine Disk-I/O-Leseanfrage ab
    - Nach Laden der fehlenden Seite wird ein I/O-Interrupt erzeugt
    - das BS setzt Prozess zuletzt wieder in den Bereit-(Ready-) Zustand



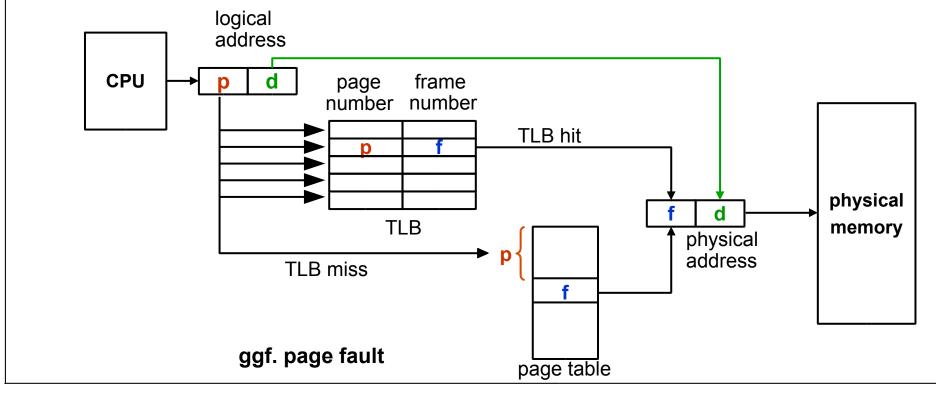
## Virtueller Speicher allgemein (3)

- Thrashing: Prozessor verbringt die meiste Zeit mit Ein- und Auslagern von Prozessteilen statt mit der Ausführung von Prozessanweisungen
- Lokalitätsprinzip:
  - Zugriffe auf Daten und Programmcode häufig lokal gruppiert;
    - → Annahme gerechtfertigt, dass nur wenige Prozessstücke während einer kurzen zeitlichen Periode gleichzeitig vorgehalten werden müssen



## Translation Look-Aside Buffer (1)

- Translation Look-Aside Buffer (TLB): schneller Hardware-Cache für zuletzt benutzte Seitentabelleneinträge
- Assoziativ-Speicher: bei Übersetzung einer Adresse wird deren Seitennummer gleichzeitig mit allen Einträgen des TLB verglichen.

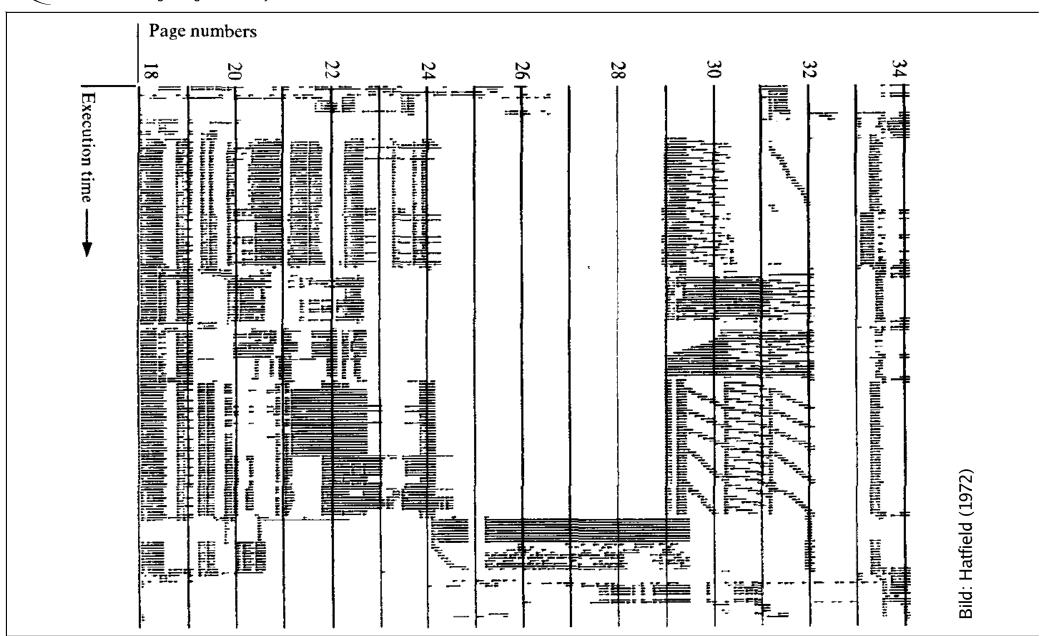


## Translation Look-Aside Buffer (2)

- Treffer im TLB → Speicherzugriff auf Seitentabelle unnötig
- Fehltreffer → Zugriff auf die Seitentabelle;
   alten Eintrag im TLB durch neuen ersetzen
- Trefferquote (hit ratio) beeinflusst die durchschnittliche Zeit einer Adressübersetzung.
- Lokalitätsprinzip: Programme greifen meist auf benachbarte Adressen zu → auch bei kleinen TLBs hohe Trefferquoten (typisch: 80-98%).

14.03.2013

### Lokalitätsprinzip





## Translation Look-Aside Buffer (3)

- Inhalt des TLB ist prozess-spezifisch!
   Zwei Möglichkeiten:
  - Jeder Eintrag enthält ein "valid bit". Bei Prozesswechsel (Context Switch) ganzen TLB invalidieren.
  - Jeder Eintrag im TLB enthält Prozessidentifikation (PID), die mit der PID des zugreifenden Prozesses verglichen wird.
- <u>Beispiele</u> für TLB-Größen:
  - Intel 80486: 32 Einträge.
  - Pentium-4, PowerPC-604: 128 Einträge für jeweils Code und Daten.



## Translation Look-Aside Buffer (4)

### Was macht hier eigentlich das Betriebssystem?

- Page-Table-Register laden
- Im Falle eines Page Fault: Fehlende Seite aus dem Swap holen und Seitentabelle aktualisieren
- Evtl. vorher: Seitenverdrängung welche Seite aus dem Hauptspeicher entfernen? (→ später)

#### Alles andere: Hardware

- Zugriff auf TLB und ggf. auf Seitentabelle
- Wenn Seite im Speicher: Berechnung der phys. Adresse
- Inhalt aus Cache oder ggf. aus Hauptspeicher holen



## **Mehrstufiges Paging (1)**

Die Seitentabelle kann sehr groß werden.

Beispiel:

- 32-Bit-Adressen,
- 4 KByte Seitengröße,
- 4 Byte pro Eintrag

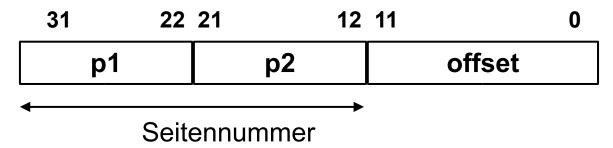
Seitentabelle:

>1 Million Einträge,

4 MByte Größe (pro Prozess!)

## **Mehrstufiges Paging (2)**

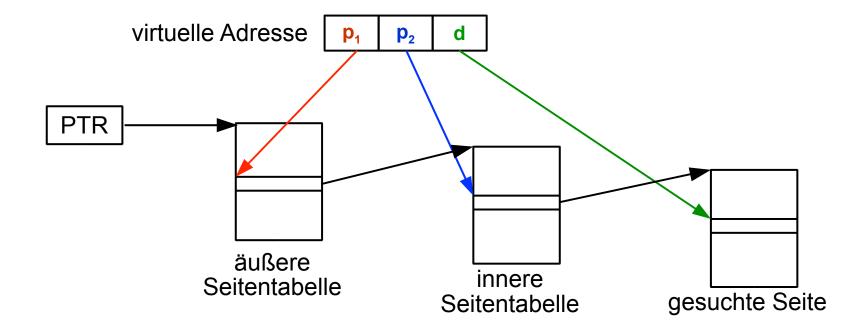
- Zweistufiges Paging:
  - Seitennummer noch einmal unterteilen, z. B.:



- p<sub>1</sub>: Index in äußere Seitentabelle, deren Einträge jeweils auf eine innere Seitentabelle zeigen
- p<sub>2</sub>: Index in eine der inneren Seitentabellen, deren Einträge auf Seitenrahmen im Speicher zeigen
- Die inneren Seitentabellen müssen nicht alle speicherresident sein
- Analog dreistufiges Paging etc. implementieren

## **Mehrstufiges Paging (3)**

### Adressübersetzung bei zweistufigem Paging:

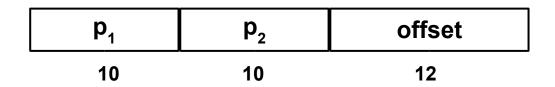




## **Mehrstufiges Paging (4)**

Größe der Seitentabellen:

Beispiel:



- Die äußere Seitentabelle hat 1024 Einträge, die auf (potentiell) 1024 innere Seitentabellen zeigen, die wiederum je 1024 Einträge enthalten.
- Bei einer Länge von 4 Byte pro Seitentabelleneintrag ist also jede Seitentabelle genau eine 4-KByte-Seite groß.
- Es werden nur so viele innere Seitentabellen verwendet, wie nötig.

## **Mehrstufiges Paging (5)**

- Jede Adressübersetzung benötigt noch mehr Speicherzugriffe, deshalb ist der Einsatz von TLBs noch wichtiger.
- Als Schlüssel für den TLB werden alle Teile der Seitennummer zusammen verwendet (p<sub>1</sub>,p<sub>2</sub>,...).

## Aufgabenbeispiel (1)

Paging mit folgenden Parametern:

- 32-Bit-Adressbus
- 32 KB Seitengröße
- 64 MB RAM
- 1-stufiges Paging

Zu berechnen:

- a) maximale Anzahl der adressierbaren virtuellen Seiten
- b) Größe der erforderlichen Seitentabelle (in KB)

a) 32 KB (Seitengröße) =  $2^5 \times 2^{10}$  Byte =  $2^{15}$  Byte d.h.: Offset ist 15 Bit lang

17 Bit	15 Bit
Seitennummer	Offset

Also gibt es 2<sup>17</sup> virtuelle Seiten

- b) Zur Seitentabelle: In 64 MB RAM passen 64 M / 32 K = 2 K = 2048 (2<sup>11</sup>) Seitenrahmen Ein Eintrag in der Seitentabelle benötigt darum 11 Bit, in der Praxis 2 Byte.
  - → Platzbedarf:
     #(virt. Seiten) x Größe(Eintrag)
     = 2<sup>17</sup> x 2 Byte = 2<sup>18</sup> Byte = 256 KB



## Aufgabenbeispiel (2)

#### Paging mit folgenden Parametern:

- 32-Bit-Adressbus
- 16 KB Seitengröße
- 2 GB RAM
- 3-stufiges Paging

#### Zu berechnen:

- a) maximale Anzahl der adressierbaren virtuellen Seiten
- b) Größe der Seitentabelle(n)
- c) Anzahl der Tabellen
- a) 16 KB (Seitengröße) = 2<sup>4</sup> x 2<sup>10</sup> Byte = 2<sup>14</sup> Byte, d.h.: Offset ist 14 Bit lang

6 Bit 6 Bit 14 Bit
Seitennummer Offset

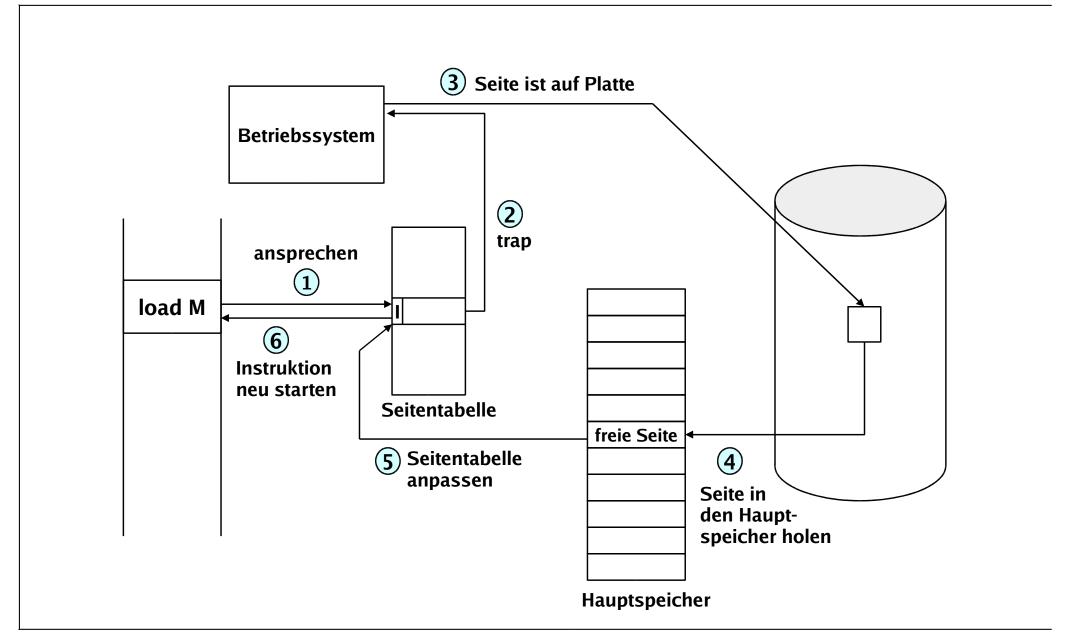
Also gibt es 2<sup>18</sup> virtuelle Seiten

- b) Zur Seitentabelle:
  In 2 GB RAM passen 2 G / 16 K
  = 128 K = 2<sup>17</sup> Seitenrahmen
  Ein Eintrag in der Seitentabelle benötigt
  darum 17 Bit, in der Praxis 4 Byte.
  - → Platzbedarf einer Tabelle:
     #(Einträge) x Größe(Eintrag)
     = 2<sup>6</sup> x 4 Byte = 2<sup>8</sup> Byte = 256 Byte

Es gibt 1 äußere, 2<sup>6</sup> mittlere und 2<sup>12</sup> innere Seitentabellen



### Page-Fault-Behandlung



## Speicher-Forensik (Linux)

- Speicherabbild (unter Linux) z. B. über /dev/fmem erstellen
  - benötigt Kernel-Modul fmem
  - Die Datei /dev/kmem (→ Skript) gibt es in aktuellen Linux-Kernel-Versionen nicht mehr; /dev/mem greift nur auf I/O-Speicher zu.
- Problem: Speicherabbild analysieren
  - physischer Speicher! BS verwendet virt. Adressen
  - ausgelagerte Seiten fehlen (waren auf Platte)
- Aktuelle Forschung: RamParser (2010)

## Speicher-Forensik (Linux)

- Umsetzen virtueller in physische Adressen kein großes Problem: Muss Seitentabelle finden und die MMU-Funktion simulieren.
- Aber: Wo stehen die diversen Kernel-Datenstrukturen?
   Bei jeder Kernel-Version und für jede CPU-Architektur verschieden.
- RamParser [1] analysiert Kernel-Code und leitet daraus die Positionen im Speicher ab.

[1] Andrew Case, Lodovico Marziale, Golden G. Richard: *Dynamic Recreation of kernel data structures for live forensics*, http://www.dfrws.org/2010/proceedings/2010-304.pdf

## Speicher-Forensik (Linux)

- RamParser kann u. a. die Funktionen von ps und netstat nachbilden
- unterstützt diverse Linux-2.6.x-Kernel für x86, x86\_64 und PPC64
- verwendet Datei System.map des zu untersuchenden Systems (Namen und Adressen aller im Kernel definierten Symbole)
- Es gab schon vor RamParser ähnliche Tools für Linux, aber immer nur für eine bestimmte Linux-Version und Architektur



## 3.15 Interrupts (Softwarearchitektur für I/O)

#### Übersicht

- Motivation: Interrupts vs. Polling
- Interrupt-Handler

### **Motivation (1)**

- Festplattenzugriff ca. um Faktor 1.000.000 langsamer als Ausführen einer CPU-Anweisung
- Naiver Ansatz für Plattenzugriff:

```
naiv () {
  rechne (500 ZE);
  sende_anfrage_an (disk);
  antwort = false;
  while (! antwort ) {
    /* diese Schleife rechnet 1.000.000 ZE lang */
    antwort = test_ob_fertig (disk);
  }
  rechne (500 ZE);
  return 0;
}
```

### **Motivation (2)**

- Naiver Ansatz heißt "Pollen": in Dauerschleife ständig wiederholte Geräteabfrage
- Pollen verbraucht sehr viel Rechenzeit:

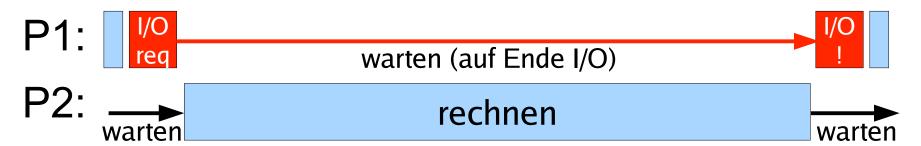
#### I/O-Gerät pollen

- Besser wäre es, in der Wartezeit etwas anderes zu tun
- Auch bei Parallelbearbeitung mehrerer Prozesse: Polling immer noch ungünstig



## Motivation (3)

 Idee: Prozess, der I/O-Anfrage gestartet hat, solange schlafen legen, bis die Anfrage bearbeitet ist – in der Zwischenzeit was anderes tun



- Woher weiß das System,
  - wann die Anfrage bearbeitet ist, also
  - wann der Prozess weiterarbeiten kann?

# **Motivation (4)**

- Lösung: Interrupts bestimmte Ereignisse können den "normalen" Ablauf unterbrechen
- Nach jeder ausgeführten CPU-Anweisung prüfen, ob es einen Interrupt gibt

## Interrupt-Klassen

- I/O (Eingabe/Ausgabe, asynchr. Interrupts)
   Meldung vom I/O-Controller: "Aktion ist abgeschlossen"
- Timer
- Hardware-Fehler
   Stromausfall, RAM-Paritätsfehler
- Software-Interrupts (Exceptions, Traps, synchrone Interrupts) Falscher Speicherzugriff, Division durch 0, unbekannte CPU-Instruktion, ...



### Interrupts: Vor- und Nachteile

#### **Vorteile**

#### Effizienz

I/O-Zugriff sehr langsam → sehr lange Wartezeiten, wenn Prozesse warten, bis I/O abgeschlossen ist

### Programmierlogik

Nicht immer wieder Gerätestatus abfragen (Polling), sondern abwarten, bis passender Interrupt kommt

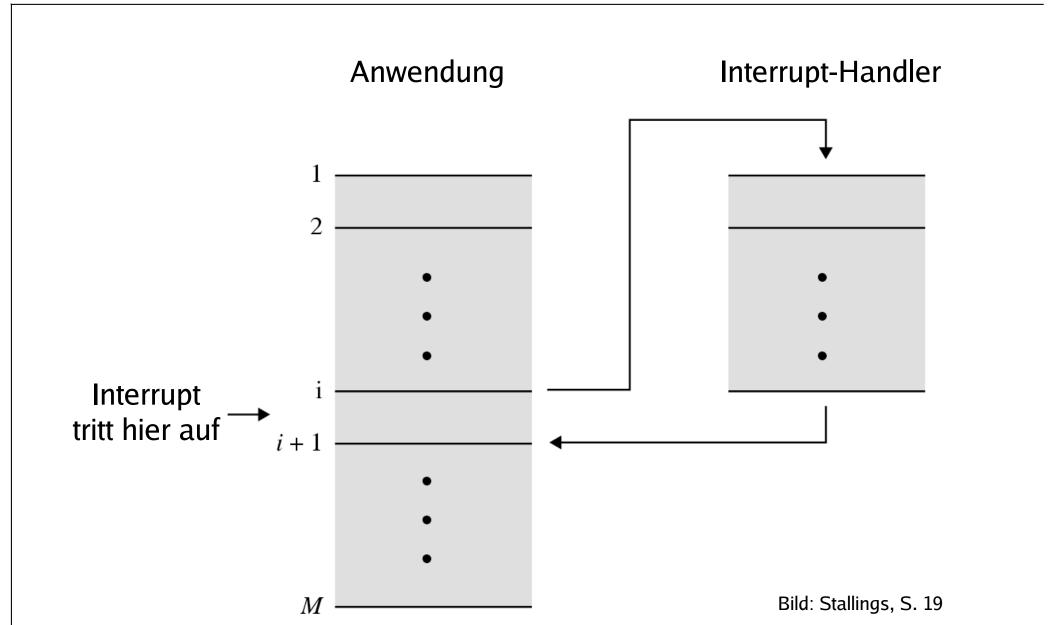
#### **Nachteile**

#### Mehraufwand

Kommunikation mit Hardware wird komplexer, Instruction Cycle erhält zusätzlichen Schritt



# Interrupt-Bearbeitung (1)





# Interrupt-Bearbeitung (2)

#### Grundsätzlich

- Interrupt tritt auf
- Laufender Prozess wird (nach aktuellem Befehl) unterbrochen, BS übernimmt Kontrolle
- BS speichert Daten des Prozesses (wie bei Prozesswechsel → Scheduler)
- BS ruft Interrupt-Handler auf
- Danach: Scheduler wählt Prozess aus, der weiterarbeiten darf (z. B. den unterbrochenen)



# Interrupt-Bearbeitung (3)

### Was tun bei Mehrfach-Interrupts?

### Drei Möglichkeiten

- Während Abarbeitung eines Interrupts alle weiteren ausschließen (DI, disable interrupts)
  - → Interrupt-Warteschlange
- Während Abarbeitung andere Interrupts zulassen
- Interrupt-Prioritäten: Nur Interrupts mit höherer Priorität unterbrechen solche mit niedrigerer

# Eingabe & Ausgabe

- Kommunikation mit Hardware
- I/O-Ports mit Registern:
  - Status-Register
  - Control-Register
  - Daten-Register (lesen, schreiben)
- Zugriff über Prozessorbefehle in, out

### **Disketten Controller**

### Beispiel: Disketten-Controller

```
enum FloppyRegisters
                                    = 0x3F0, // read-only
   STATUS REGISTER A
   STATUS REGISTER B
                                    = 0x3F1, // read-only
  DIGITAL OUTPUT REGISTER
                                    = 0x3F2
   TAPE DRIVE REGISTER
                                    = 0x3F3
  MAIN STATUS REGISTER
                                    = 0x3F4, // read-only
  DATARATE SELECT REGISTER
                                    = 0x3F4, // write-only
  DATA FIFO
                                    = 0x3F5
  DIGITAL INPUT REGISTER
                                   = 0x3F7, // read-only
                                   = 0x3F7 // write-only
   CONFIGURATION CONTROL REGISTER
};
```



### **Disketten Controller**

Aufbau des Main Status Register:

MRQ	DIO	NDMA	BUSY	ACTD	ACTC	АСТВ	ACTA
7	6	5	4	3	2	1	0

#### MRQ (Main Request)

1 = Data Register bereit

0 = nicht bereit

#### DIO (Data Input / Output)

1 = Controller → CPU

 $0 = CPU \rightarrow Controller$ 

#### NDMA (Non-DMA Mode)

1 = Controller nicht im DMA-Modus

0 = Controller im DMA-Modus

#### **BUSY**

1 = Instruktion wird gerade ausgeführt

0 = keine aktive Instruktion

#### ACTA-ACTD

(Laufwerk A, B, C, D Seek)

1 = aktiv

0 = nicht aktiv

Quelle: http://viralpatel.net/taj/tutorial/programming\_fdc.php

### Kommunikation mit Controller

- Allgemeine Vorgehensweise:
  - mit IN Statusregister auslesen und interpretieren
  - falls bereit: mit OUT zunächst Datenregister belegen und dann Befehl an Gerät schicken
  - entweder auf Interrupt vom Controller warten oder mit Polling Statusregister wiederholt auslesen (IN)
  - mit IN Datenregister auslesen

- Klassische Übertragung von Daten zwischen Speicher und Gerät: byte-/wort-weise, über einzelne IN-/OUT-Befehle
- bei großen Datenmengen langsam, umständlich
- Alternative: DMA (Direct Memory Access)
- Kommando an Controller enthält RAM-Adresse
  - Controller liest selbständig Daten aus dem RAM
  - oder schreibt selbständig in das RAM
  - Transfer benötigt keine CPU-Zeit, Interrupt nach Fertigstellung

- DMA arbeitet immer mit physikalischen Adressen (nicht mit virtuellen Adressen)
- denn: nur CPU/MMU "verstehen" virtuelle Adressen
- zwei DMA-Arten im PC:
  - ISA-DMA (im Wesentlichen: Floppy)
  - PCI-Busmastering-DMA (z. B. Festplatten)

## **Keyboard-Controller**

- Keyboard-Controller erzeugt bei jedem Tastendruck Interrupt 1
- In BS: Interrupt-Handler für IRQ 1 installieren
- Register IDT mit Adresse der Interrupt-Handler-Tabelle laden
- Handler liest I/O-Port 0x60
   byte = inb (0x60);
- Rückgabewert ist ein Scancode, nach Konvertieren in ASCII-Zeichen Eintragen in Keyboard-Buffer

## **Keyboard-Interrupt-Handler**

```
char system_kbd[BUFLEN];  /* globaler Puffer */
int system_kbd_pos = 0;  /* aktuelle Position */
void keyboard handler (struct regs *r) {
  /* Scancode auslesen ... */
  scancode = inb (0x60);
  c = convert_to char (scancode);
  /* ... und \overline{i}n \overline{P}uffer schreiben */
  system kbd[system kbd pos] = c;
  system kbd pos = (system kbd pos + 1) % BUFLEN;
  system kbd count++;
};
void keyboard install () {
  /* Handler in Int.-Handler-Tabelle eintragen */
  irq install handler(1, keyboard handler);
void irq install handler (int irq,
                            void (*handler)(struct regs *r)) {
  irq routines[irq] = handler;
```

# **Keyboard ohne Interrupts**

- Keine Interrupts? → Tastatur pollen
- Port 0x64: Keyboard-Status-Register (ro)

### Flag IBF = Input Buffer Full

Quelle: http://www.computer-engineering.org/ps2keyboard/