Einstieg in die Betriebssystementwicklung

Grundlagen für das eigene OS

Tobias Stumpf
Matthias Keller
Fernando Arroniz

Chemnitzer Linux Tage

18. März 2012

Agenda

1. Einleitung
2. Entwicklungsumgebung
3. Hardwareinitialisierung die Erste
4. Ausgabe erzeugen
5. Aufsetzen von GDT und IDT
6. Interrupt Controller
7. Timer
8. Debuggen
9. Speicherverwaltung
10. Weitere Elemente
11. Literatur

Was erwartet uns?

Ziel

- Bau eines einfachen OS für die x86-Architektur (32-bit)
- monolitisches Kerneldesign
- einfaches Design des Kernels
- Kernel soll später leicht zu erweitern sein

Ablauf

- Abwechslung zwischen Theorie und Praxis
- zu Beginn kleine Theorieblöcke gefolgt von je einem praktischen Teil
- am Ende folgt ein größerer Theorieblock mit einem Ausblick

Hinweis

- Workshop kann nur einen groben Einstieg geben
- wer sein eigenes OS entwickeln möchte, muss bereit sein Handbücher zu lesen

Entwicklungsumgebung

Programmiersprachen

- C (soweit möglich)
- x86-Assembler für hardwarenahe Funktionen

Tools

- gcc, gnu as und gnu ld
- make
- Editor nach Wahl
- Qemu

JamesM's kernel development tutorials

- Workshop basiert auf dem Kernel Tutorial von James Molloy
- das Tutorial kann als Vertiefung / Ergänzung zu diesem Workshop herangezogen werden

Unterschiede

- anstelle von NAS (Netwide Assembler) wird GNU AS benutzt (andere Syntax)
- wir nutzen Qemu anstelle von Bochs

Makefile

Variablen

SOURCES Auflistung aller Module (C- u. AS-Files) die wir benutzen wollen (Module werden in der Form Dateiname**.o** angegeben)

CFLAGS Parameter für den C-Compiler

ASFLAGS Parameter für den AS-Compiler

LDFLAG Parameter für den Linker

Parameter

all führt zuerst kernel, dann floppy aus

kernel erzeugt Kernel Image

floppy Erzeugt eine Boot Diskette

clean aufräumen

Makefile

Besonderheiten für 64-bit Entwicklungsrechner

CCFLAGS -m32

ASFLAGS -m32

LDFLAGS -melfi386

Durch diese Parameter wird 32-bit Code anstelle von 64-bit Code erzeugt.

Makefile

```
 1 CC=gcc
 2 AS=gcc
 3
 4 SOURCES=boot.o main.o
 5
 6 CCFLGAS=-nostdlib -nostdinc -ffreestanding \
 7     -fno-builtin -fno-stack-protector -O0 -W -m32 \
 8     $(COMPONENTS)
 9 LDFLAGS=-Tlink.ld -melf_i386
10 ASFLAGS=-m32
11
12 all: kernel floppy
13
14 kernel: $(SOURCES) link
15
16 clean:
17   -rm *.o kernel
18
19 link:
20   ld $(LDFLAGS) -o kernel $(SOURCES)
21
22 floppy:
23   sudo losetup /dev/loop0 ../floppy.img
24   sudo mount /dev/loop0 /mnt
25   sudo cp kernel /mnt/kernel
26   sudo umount /dev/loop0
27   sudo losetup -d /dev/loop0
28
```

Makefile

```
29 %.o: %.c
30     $(CC) $(CCFLGAS) -c $<
31
32 %.o: %.S
33     $(AS) $(ASFLAGS) -c $<
```

Compilerflags

-nostdlib verhindert das Einbinden der standard C-Libraries

-nostdinc verhindert das Einbinden der standard Headerfiles

-ffreestanding schaltet hostspezifische GCC Optimierungen ab

Bsp. 1: printf("string") wird durch puts("string") ersetzt

Bsp. 2: main-Funktion stellt nicht zwingend den Programmstart dar

-fno-bultin benutzt keine speziellen auf Speicher und Laufzeit optimierten Funktionen

-fno-stack-protector schaltet die Stack Overflow Protection ab

andernfalls ordnet der GCC die Variablen gegebenenfalls auf dem Stack um

Linker Script

3 Codesegmente

- **.text** ausführbarer Code
- **.data** vorinitialisierte statische Daten
- **.bss** uninitialisierte statische Daten

Linker Script

```
1  ENTRY(start)
2  SECTIONS
3  {
4    start_addr = 0x00000000;
5    .text 0x100000:
6    {
7      code = .; _code = .; __code = .;
8      *(.text)
9      . = ALIGN(4096);
10   }
11   .data :
12   {
13      data = .; _data = .; __data = .;
14      *(.data)
15      *(.rodata)
16      . = ALIGN(4096);
17   }
18   .bss :
19   {
20     bss = .; _bss = .; __bss = .;
21     *(.bss)
22     . = ALIGN(4096);
23   }
24   end = .; _end = .; __end = .;
25 }
```

Qemu

- zum Testen des Kernels wird Qemu verwendet
- Qemu emuliert einen vollständigen PC
- wir nutzen ein “virtuelles” Floppy-Disk-Laufwerk zum Booten
- GDB-Schnittstelle zum einfachen Debuggen

Aufrufparameter

qemu -cpu 486 -fda floppy.img –serial stdio

Qemu Monitor

Wechsel zum Monitor mit: [Strg]+[Alt]+[2]
$ info registers // Gibt die CPU Register aus

Hardwareinitialisierung

Zielsetzung

- CPU soll 32-bit Code ausführen
- zum Test soll 0xdeadbeaf ins Register **eax** geschrieben werden
- Verständnis über das Zusammenspiel von C- und Assemblercode

Bootvorgang beim x86

- x86 historisch gewachsene Architektur die rückwärtskompatibel ist
- alle CPUs starten im Real Mode (CPU führt nur 16-bit Instruktionen aus)
- danach kann die CPU stufenweise in den 32-bit oder 64-bit Modus gefahren werden
- wir nutzen den GRUB-Bootloader mit Multiboot
- GRUB erledigt für uns einen Teil des Hardwaresetups
- wir können direkt 32-bit Code ausführen

Multiboot

- Standard für das Booten eines Betriebssystems
- führt den ersten Schritt der Hardwareinitialisierung aus
- vor Aufruf des Betriebssystems wird die CPU nicht zurück in den Real Mode gesetzt
- Multiboot bietet die Möglichkeit Kernelparameter zu übergeben
- Minimierung des nötigen Bootcodes ($<$20 Zeilen)

GCC Aufrufkonvention

- Parameter werden über den Stack übergeben
- Parameter werden von rechts nach links auf den Stack gelegt
- der Stackpointer zeigt auf die Rücksprungadresse
- der Rückgabewert liegt im Register **eax**
- die Register **eax, ecx, edx** können innerhalb der aufgerufenen Funktion frei genutzt werden

Aufgabe 1

- machen Sie sich mit der Entwicklungsumgebung vertraut
- schreiben Sie eine Mainfunktion, welche 0xdeadbeaf ins Register **eax** schreibt
- verwenden Sie als Dateiname main.c und erweitern Sie das Makefile diesbezüglich
- benutzen Sie nur C-Code (kein Assemblercode innerhalb der Mainfunktion)
- nutzen Sie den Qemu Monitor um den Erfolg Ihrer Arbeit zu prüfen
- **Zusatzaufgabe:** achten Sie auf die richtige Deklaration von main

Ausgabe erzeugen

Zielsetzung

- Ein “Hello Linux Tage!” über COM1
- Schreiben und Lesen von den I/O-Ports
- Benutzen der UART-Schnittstelle

Ausführliche Beschreibung

http://en.wikibooks.org/wiki/Serial_Programming/8250_UART_Programming

Eigene Datentypen

- Abstraktion von der Hardware
- die Länge von einigen Datentypen hängt in C von der eingesetzten Hardware ab
- Definition eigener Datentypen macht das OS universeller

Datentypen

```
1 typedef unsigned char    uint8;
2 typedef unsigned short   uint16;
3 typedef unsigned int     uint32;
4
5 typedef char    sint8;
6 typedef short   sint16;
7 typedef int     sint32;
```

I/O-Ports

- Zugriff auf externe Geräte
- wird vor allem bei älteren Systemen benutzt
- jedes Gerät ist einem festen I/O-Portbereich zugeordnet
- ein Port ist 1, 2 oder 4-Byte groß
- 16-bit Portadresse
- neue Geräte verwenden häufig I/O-Speicher anstelle von I/O-Ports

Lese-/Schreibzugriffe von/auf I/O-Ports

Lesen: inb, inw, inl
Schreiben: outb, outw, outl

I/O-Ports

Beispiel: von 32-bit breitem Port einlesen

```
1 void outl(uint16 port, uint32 value) {
2   asm volatile ("outl %1, %0" : : "dN" (port), "a" (value) );
3 }
```

Beispiel: in ein 32-bit breites Register schreiben

```
1 uint32 inl(uint16 port) {
2   uint32 ret;
3   asm volatile("inl %1, %0" : "=a" (ret) : "dN" (port) );
4   return ret;
5 }
```

Hardwarezugriff

- Steuerung der Hardware erfolgt über Registerzugriffe
- Register sind entweder als I/O-Port ansprechbar oder memorymapped
- Ausnahmen bilden spezielle CPU Register
- die Länge eines Registers ist im allgemeinen ein Vielfaches von 8-bit
- die CPU-Register einer 32-bit Maschine sind meist 32-bit
- Registerinhalte können als Datenwort aufgefasst werden
- jedes Bit eines Registers hat eine bestimmte Bedeutung

UART8250

- ursprüngliche Chipvariante für die serielle Kommunikation
- Nachfolgerversionen wie z.B. der UART16650 sind Erweiterungen von diesem Chip und Rückwärtskompatibel
- 8 Hardwareregister wovon einige mehrfach belegt sind
- die PC Spezifikation sieht bis zu 4 UART-Schnittstellen vor

UART8250

serielle Schnittstellen eines standard PCs

COM Port	IRQ	Base Port I/O address
1	4	0x3F8
2	3	0x2F8
3	4	0x3E8
4	3	0x2E8

UART8250

Register

Offset	DLAB	I/O Access	Abbrv.	Register Name
+0	0	Write	THR	Transmitter Holding Buffer
+0	0	Read	RBR	Receiver Buffer
+0	1	Read/Write	DLL	Divisor Latch Low Byte
+1	0	Read/Write	IER	Interrupt Enable Register
+1	1	Read/Write	DLH	Divisor Latch High Byte
+2	x	Read	IIR	Interrupt Identification Register
+2	x	Write	FCR	FIFO Control Register
+3	x	Read/Write	LCR	Line Control Register
+4	x	Read/Write	MCR	Modem Control Register
+5	x	Read	LSR	Line Status Register
+6	x	Read	MSR	Modem Status Register
+7	x	Read/Write	SR	Scratch Register

UART8250

Interrupt Enable Register

7	6	5	4	3	2	1	0
undifined		LPM	SM	MSI	RLS	THRE	RDA

Registerbeschreibung

RDA Enable Received Data Available Interrupt

THRE Transmitter Holding Register Empty Interrupt

RLS Receiver Line Status Interrupt

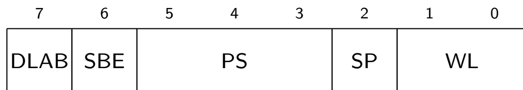
MSI Modem Status Interrupt

SM Enables Sleep Mode (16750)

LPM Enables Low Power Mode (16750)

UART8250

Line Control Register



Registerbeschreibung

WL Word Length (2: 8-Bit)

SP Stop Bit (0: One Stop Bit)

PS Parity Select (1: Odd Parity)

SBE Set Brake Enable

DLAB Divisor Latch Access Bit

Aufgabe 2

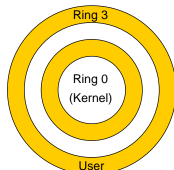
- Schreiben Sie eine Funktion inb um ein Byte von einem I/O-Port einzulesen
- Schreiben Sie eine Funktion outb um ein Byte in das Register eines I/O-Ports zu schreiben
- Schreiben Sie eine Funktion uart8250_putc um ein Zeichen über die serielle Schnittstelle auszugeben
- Nutzen Sie Ihre selbst geschriebene Funktion um von der Mainfunktion einen Text auszugeben
- **Zusatzaufgabe:** Schreiben Sie eine Funktion uart8250_puts um einen String auszugeben

Aufsetzen von GDT und IDT

Zielsetzung

- Kennenlernen der verschiedenen Speichermodelle des X86
- Ausführungsmodi
- Aufsetzen der Global Descriptor Table
- Interrupt- und Exeptionbehandlung
- Aufsetzen der Interrupt Descriptor Table
- Auslösen eines Interrupts

Ausführungsmodi (Ring-Level)



- 4 Ringe (Betriebsmodi) von 0 - 3
- innerster Ring ist Kernel Modus
- äußerster Ring ist User Modus
- Kernel Modus erlaubt maximalen Hardwarezugriff
- User Modus schränkt HW-Zugriff ein (z.B. Interrupts können nicht blockiert werden oder Einschränkung des Speicherzugriffes)
- die meisten Betriebssysteme nutzen nur Kernel und User Modus
- Ring 1 und 2 wird u.a. von Microkernelsystemen für die Treiberimplementation genutzt

Speicherzugriffsmodelle

- Zugriffsschutz
- eine Anwendung soll den Speicher einer anderen weder gewollt noch ungewollt verändern
- zwei grundsätzlich verschiedene Ansätze:
  - Segmentierung
  - Paging

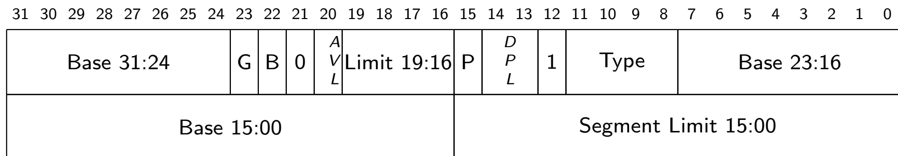
Speicherzugriffsmodelle

Segmentierung

- teilt den Speicher in mehrere Segmente
- linearer Address Space
- beginnt bei 0 und endet bei der Größe des Segments
- Zugriff nur innerhalb eines Segmentes möglich
- Segmentierung ist fest verankert in der x86-Architektur
- ohne ein minimales Setup kommen wir nicht aus
- 64-bit Systeme benötigen für bestimmte Instruktionen einen flachen Adressraum von 0x0000 0000 - 0xffff ffff
- wir verwenden ebenfalls einen flachen Adressraum

Segment Descriptor

Register



Siehe *Intel x86 Software Developer's Manual, Volume 3A, Kap. 4* für weitere Details.

Abbildung in C durch eine Stuktur

```
1 typedef struct gdt_entry_str {
2   uint16 limit_low;
3   uint16 base_low;
4   uint8  base_middle;
5   uint8  access;
6   uint8  granularity;
7   uint8  base_high;
8 } __attribute__((packed)) gdt_entry_t;
```

Global Descriptor Table (GDT)

Tabelleneinträge

- min. 5 Einträge
- 1. Eintrag ist null und wird von der CPU genutzt
- 2 Kerneleinträge (Code- und Datensegment)
- 2 Einträge für den User Level (ebenfalls Code- und Datensegment)

Aktivieren der neuen GDT

- Assemblercode nötig
- Pointer zur GDT ins Register **GDTR** laden
- Offset zum Kerneldatensegment wird in folgende Register geschrieben: **ds, es, fs, gs, ss**
- Codesegment wird über die **ljmp** Instruktion aktiviert

Global Descriptor Table (GDT)

GDT Pointer Struktur

```
1 typedef struct gdt_ptr_str {
2   uint16 limit;
3   uint32 base;
4 } __attribute__((packed)) gdt_ptr_t;
```

Werte in Register schreiben

```
1 extern void gdt_flush(gdt_ptr_t *);
```

```
 1 gdt_flush:
 2   mov    4(%esp),%eax;
 3   lgdtl  (%eax);
 4
 5   mov $0x10, %ax
 6   mov %ax, %ds;
 7   mov %ax, %es;
 8   mov %ax, %fs;
 9   mov %ax, %gs;
10   mov %ax, %ss;
11   ljmp $0x08, $1f;
12 1:
13   ret;
```

Interrupt Descriptor Table (IDT)

- die IDT wird analog zur GDT aufgebaut
- gibt es für einen IRQ keinen Eintrag in der IDT führt dies zu einer Panic und den Reset der CPU
- 256 Einträge nötig (für jeden möglichen Interrupt)
- nicht benötigte Einträge werden mit 0 initialisiert
- alle Interruptroutinen sollen im Kernelmodus abgearbeitet werden
- wir nutzen 32-bit große Segmente
- jeder Eintrag beinhaltet einen Funktionspointer zur zugehörigen Interrupt Service Routine (ISR)

Siehe *Intel x86 Software Developer's Manual, Volume 3A, Kap. 5* für weitere Details.

Interrupt Service Routine

- für jeden IRQ wird eine eigene ISR benötigt, da beim Aufruf der ISR die Interrupt Nr. nicht übergeben wird
- Zweiteilen der ISR
  - 1. Teil sichert Registerwerte, setzt die entsprechenden Datensegmente und sichert Interrupt Nr. (Assemblercode)
  - 2. Teil ist in C implementiert und ruft den entsprechenden Handler auf (C-Code)
- durch diese Teilung ist der Code für alle IRQ gleich und Handlerfunktion können zur Laufzeit einfach hinzugefügt werden
- Interrupt 8, Interrupt 10-14 und Interrupt 17 liefern zusätzlich einen Fehlercode

Aufgabe 3

- erstellen Sie eine C-Struktur, welche die IDT Register wiedergibt
- erstellen Sie eine Struktur, welche die Basisadresse und das Limit der IDT beinhaltet
- Setzen Sie die IDT auf
- beachten Sie, dass Einträge ohne gültige ISR 0 sein sollen
- Sie können die Funktion *memset* nutzen
- Lösen Sie einen IRQ durch den Assemblerbefehl *int $0x3* aus
- **Zusatz:** Schreiben Sie eine Handlerfunktion für den Interrupt 4, welche den Fehlercode sowie den Instruktion Pointer ausgibt.
  Testen Sie Ihre Handlerfunktion mit Hilfe der **int** Anweisung
  Verwenden Sie nun Interrupt 8 anstelle von Interrupt 4
  Was stellen Sie fest?

Interrupt Controller

Zielsetzung

- Grundlagen Interrupts
- Kennenlernen des PIC8259 Bausteins
- Erweitern der ISR für externe Interrupts
- Erweitern des UART8250 Treibers für den Empfang von einzelnen Zeichen

Hardware (externe) / Software Interrupts

Hardware Interrupts

- liegen am INTR PIN der CPU an
- können maskiert werden (IF Flag im Register **EFLAGS**)

Software Interrupts

- durch **int**-Anweisung innerhalb der Software ausgelöst
- Verhalten stimmt nicht mit einem HW Interrupt überein
- Bsp.1: es wird kein Error Code auf den Stack gelegt
- Bsp.2: Beim NMI-Interrupt wird der Software Handler aufgerufen, nicht aber der CPU interne Handler

Allgemein

- Jeder Interrupt hat eine Priorität
- Interrupts werden entsprechend ihrer Priorität abgearbeitet

Interne / Externe HW Interrupts

interne Interrupts (Architecture related)

- IRQ 0 - 31 sind für CPU Exceptions reserviert
- Bsp.: Divide Error, Overflow, General Protection, Page Fault

externe Interrupts (angeschlossene Hardware)

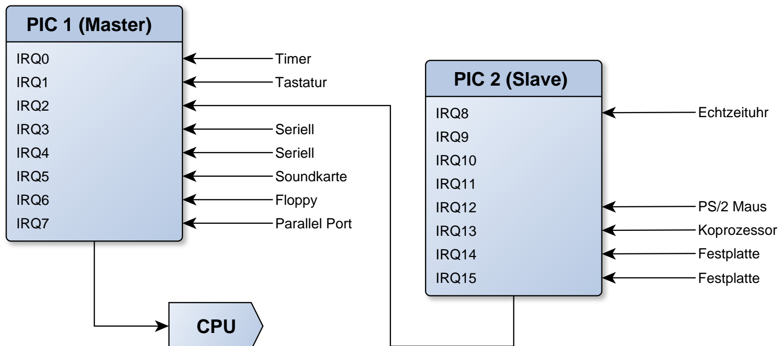
- IRQ 32 - 255 können hierfür frei gewählt werden
- es wird ein Interrupt Controller wie der PIC8259 benötigt
- neuere CPUs verfügen zusätzlich über einen APIC der neben weiteren Funktionen vor allem mehr Interruptleitungen im Vergleich zum PIC hat
- Bsp.: Timer, UART, Tastatur

PIC8259

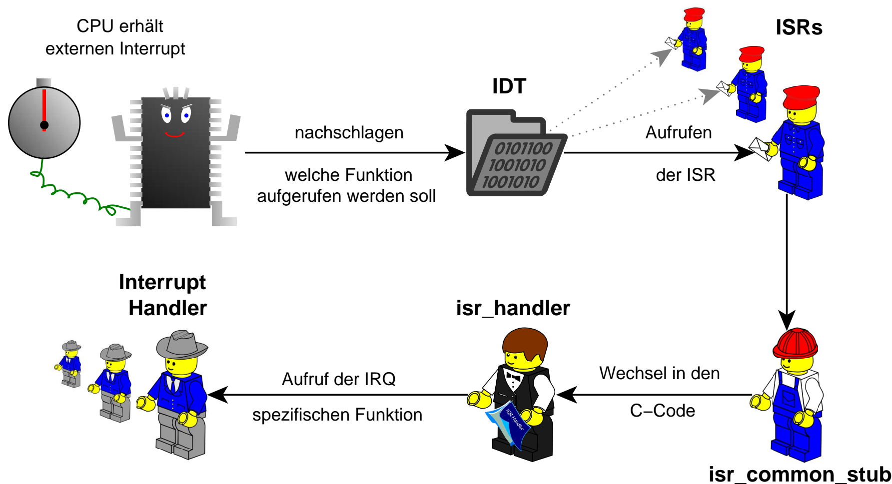
- Einfacher Interrupt Controller mit 8 Leitungen
- durch die Kaskatierung von 2 PICs stehen dem x86 15 IRQ Leitungen zur Verfügung
- nach einem Reboot besteht folgendes Mapping
  - PIC 1: IRQ 0-7 liefert Interrupt Nr. 0x08 - 0x0f an CPU
  - PIC 2: IRQ 8-15 liefert Interrupt Nr. 0x70 - 0x77 an CPU
- Konflikt mit den architekturspezifischen Interrupts
- Remap der Interrupt Nr. nötig
- nachdem ein IRQ ausgelöst wurde muss die Abarbeitung dem Controller quittiert werden

PIC8259

IRQ Belegung beim x86



IRQ Abarbeitung



Aufgabe 4

- schreiben Sie einen für alle externen IRQs gültigen Handler.
  - dieser soll den Empfang an den PIC quitieren
  - den Interrupt Handler aufrufen
  - wurde kein Interrupt Handler registriert ist nichts zu tun
  - alternativ kann zum Testen die IRQ Nummer ausgegeben werden
- ergänzen Sie die Datei aisr.S um einen irq_common_stub, welcher Ihren Interrupt Handler aufruft
- machen Sie sich mit dem erweiterten Setup der IDT vertraut
- vervollständigen Sie die Funktion init_idt
- schreiben Sie eine Funktion enable_irq
- **Zusatzaufgabe:** erweitern Sie den UART Treiber, so dass dieser einzelne Zeichen von der Console einließt. (Nutzen Sie die Funktionen uart8250_receive())

Timer (PIT8253)

Zielsetzung

- Kennenlernen des PIT8253
- Setup des Timers für ein späteres Multitasking

PIT8253

- programmierbarer Zähler
- 3 unabhängige Zähler mit einer Eingangsfrequenz von bis zu 2MHz
  - **Counter 1** Erzeugen eines Clock Signals
  - **Counter 2** wird für den DRAM benutzt
  - **Counter 3** Soundgenerierung
- früher: eigenständiger Chip
- heute: fest in der South Bridge integriert
- neuere CPUs bieten weitere Timer
- der Counter 0 ist mit dem Interrupteingang 0 des PIC verbunden

Für weitere Details siehe:

- `http://www.intel.com/design/archives/periphrl/docs/7203.htm?wapkw=8254`
- `http://www.sharpmz.org/mz-700/8253ovview.htm`

PIT8253

I/O Port Base Addr.

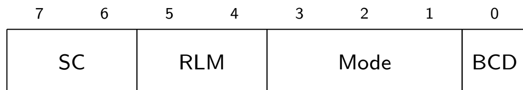
- 0x40
- 4 8-bit Register

Register

Offset	Register Name
0	Counter 0
1	Counter 1
2	Counter 2
3	Control

PIT8253

Control Register



SC Select Counter

00	Counter 0	01	Counter 1
10	Counter 2	11	ungültiger Wert

RLM Read / Load Mode

00	Zählerwert zwischenspeichern	01	R/W nur Least-Significant Byte
10	R/W nur Most-Significant Byte	11	R/W erst Least- dann Most-Significant Byte

Mode Ausführungsmodus

000	Interrupt on Terminal Count	001	Programable One-Shot
x10	Rate Generator	x11	Square Wave Generator
100	Software Triggered Strobe	101	Hardware Triggered Strobe

BCD Zählmodus 0 binär 1 dezimal

PIT8253

Counter Register

- beim Schreiben ins Register Counter 0 - 3 wir der Startwert des jeweiligen Zählers gesetzt
- ein Lesezugriff gibt den aktuellen Zählerstand zurück
- vor dem Lesen sollte der Zählerstand in einem internen Register des PIT zwischengespeichert werden **RLM = 0**, sonst kann ein ungültiger Wert ausgelesen werden

Berechnen des Counterwerts

$$count_value = \frac{InputFrequenze}{OutputFrequenze}$$

$$InputFrequenze = 1.193182 MHz$$

Aufgabe 5

- Schreiben Sie eine Funktion pit8253_init zum Initialisieren des PIT Controllers
  - nutzen Sie den Counter 0
  - am Ausgang soll ein Rechtecksignal anliegen
  - die Ausgangsfrequenz des Timers soll der Initfunktion als Parameter übergeben werden
- Schreiben Sie einen Interrupthandler für den Timer, welcher ausgibt, wie oft bereits ein Timer Interrupt ausgelöst worden ist

Zusatzaufgaben

- Welche Aussage können Sie treffen, wenn Ihnen die Anzahl Ticks, so wie Frequenz des Timer bekannt sind?
- Experimentieren Sie mit verschiedenen Frequenzen
- Was stellen sie bei kleinen Frequenzen 1 - 100 Hz fest?

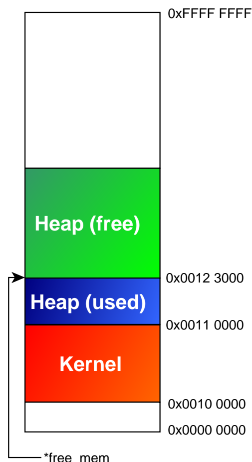
Speicherverwaltung und Debuggen

Zielsetzung

- Aufsetzen einer minimalen Speicherverwaltung
- Kennenlernen der Debugfunktionalität des Qemu

Speicherverwaltung

- im weiteren Verlauf benötigen wir eine Möglichkeit Speicher dynamisch zu allozieren
- Speicherbereich wird als kontinuierliches Array betrachtet
- Code- und Datenbereich des Kernels ist bereits vergeben
- restlicher Speicher kann der Reihe nach alloziert werden
- keine Speicherfreigabe
- dieser Ansatz wird häufig in Echtzeitsystemen angewandt



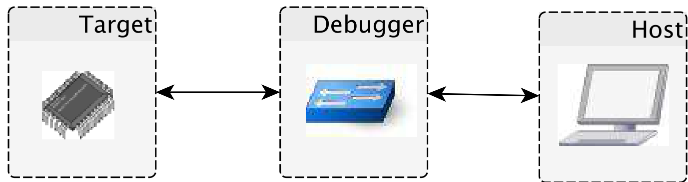
Speicherverwaltung

Anforderungen

- Möglichkeit Speicher Page Aligned anzufordern
- physikalische Adresse wird benötigt
  (wird beim Einführen eines virtuellen Adressraums benötigt)

Debuggen

- irgendwann wird sich der erste Programmierfehler einschleichen, der nicht ganz einfach zu finden ist
- neben dem Debuggen mit printf gibt es andere effektive Methoden
- Qemu stellt z.B. einen GDB Server bereit
- Qemu erlaubt das schrittweise Ausführen einzelner Instruktionen
- zusätzlich können die Daten in Registern und Speicher beobachtet werden
- mit dem DDD steht eine graphische Oberfläche zur Verfügung



Debuggen

Compiler Optionen

-g fügt dem auszuführenden Code Debuginformationen hinzu

-O0 durch das Abschalten von Optimierungen ist der Code leichter zu Debuggen (keine unerwarteten Sprünge beim Ausführen)

Qemu Aufrufparameter

-gdb tcp::1234 Nutz tcp als Übertragungsprotokol zwischen dem GDB-Server und dem GDB

-S hält den virtuellen PC sofort nach dem Einschalten an

GDB

- $ ddd kernel
- target remote :1234 (Verbindung zum Server herstellen)
- Breakpoint auf den Beginn der Mainfunktion setzen
- continue (Bootvorgang wird ausgeführt, Ausführung stoppt bei der main-Funktion)

Aufgabe 6

- machen Sie sich mit dem gegebenen Programmcode vertraut
- Schauen Sie sich nicht die Datei kheap.c an!
- die Funktion kmalloc sollte den nächsten freien Speicherbereich zurückgeben
- die Funktion kmalloc_a gibt den Beginn der nächsten freien Page zurück
- nutzen Sie den Debugger, um nachzuverfolgen, warum alle zurückgegebenen Speicheradressen Page Aligned sind
- beheben Sie den gefundenen Fehler

Speicherverwaltung

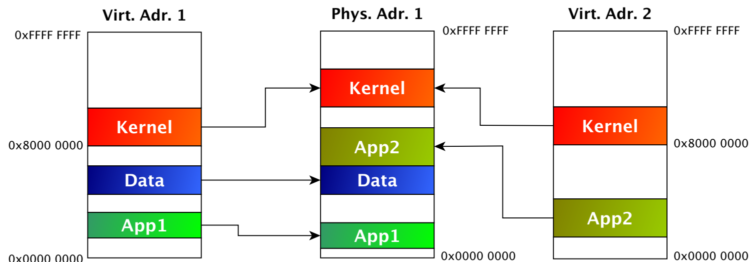
Zielsetzung

- Kennenlernen des Konzepts eines virtuellen Adressraums
- Einführung in die Pagingfunktionalität des x86
- Aufsetzen eines virtuellen Adressraums
- Bitfelder und Bitmaps
- Eigenen Page Fault Handler entwerfen

Paging

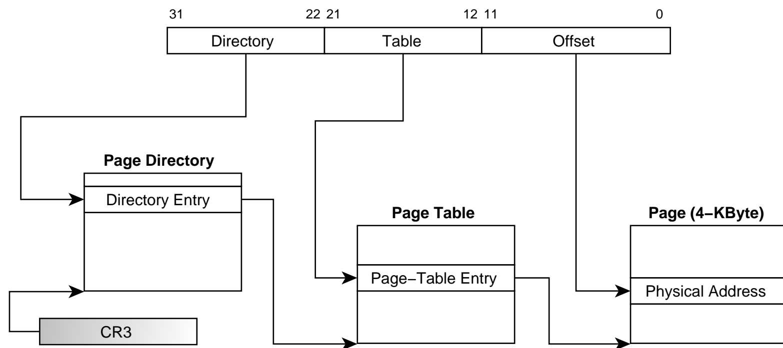
- Abstraktion vom physikalischen Adressraum
- bietet zwei wichtige Funktionen:
  - virtueller Adressraum
  - Speicherzugriffsschutz
- Umsetzung von virtueller nach physikalischer Adresse wird durch zusätzliche Hardware (MMU) realisiert
- Betriebssystem stellt Übersetzungstabellen bereit
- die Anwendung sieht nur die virtuellen Adressen
- der physikalische und somit auch der virtuelle Adressraum wird in Seiten (Pages) aufgeteilt
- eine typische Seitengrößen ist 4KByte

virtueller Adressraum



virtueller Adressraum

Adressübersetzung



- Page Directory und Page Table müssen Page Aligned sein

virtueller Adressraum

Eigenschaften

- kein vollständiges Mapping zwischen physikalischen und virtuellen Adressen (bzw. umgekehrt)
- nicht kontinuierlich
- häufig wird der virtuelle Adressraum aufgeteilt, z.B.
  - 0x0000 0000 - 0x7fff ffff : anwendungsbezogen
  - 0x8000 0000 - 0xffff ffff : Betriebssystem
- durch die Aufteilung ist das Kernelmapping für alle virtuellen Adressräume gleich
- der Zugriff auf eine Seite im Speicher kann eingeschränkt werden
  - Lesen / Schreiben
  - Kernel / User

Paging x86

Page Table Entry (4-KByte Page)

Bit Feld

```
1  typedef struct page_str {
2    uint32 present       :  1;
3    uint32 rw            :  1;
4    uint32 user          :  1;
5    uint32 write_through :  1;
6    uint32 disalbe_cache :  1;
7    uint32 accessed      :  1;
8    uint32 dirty         :  1;
9    uint32 attr_intdex   :  1;
10   uint32 global        :  1;
11   uint32 internal      :  3;
12   uint32 frame         : 20;
13 } page_t;
```

Page Fault

Page Fault tritt auf wenn:

- Page nicht gemapped ist
- Schreiben auf eine read-only Page
- Zugriff auf eine Kernel Page vom User Mode aus
- Page Fault löst Interrupt 14 aus und liefert Error Code

Error Code

Bit 0:	1 -	Page mapped	Bit 2:	1 -	User Mode
	0 -	Page not mapped		0 -	Kernel Mode
Bit 1:	1 -	Write	Bit 3:	1 -	Reserved bit was overwritten
	0 -	Read	Bit 4:	1 -	Instruction Fetch

Page Fault Addr.

- Adresse, welche den Page Fault verursacht hat, wird in CR2 geschrieben

Bitmaps (Bitsets)

Einsatzzweck

- Verwaltung von Informationen (meist Aussagen, die mit einem Bit ausgedrückt werden können)
- Ziel ist das Einsparen von Speicherplatz

Funktionen

- Set
- Unset
- Test

7							0
0	1	1	0	1	0	1	1
1	1	0	1	0	1	1	0
0	0	1	1	0	1	0	0
1	0	1	1	1	0	0	1

Aufgabe 7

- Schreiben Sie Ihren eigenen Page Fault Handler
- geben Sie folgende Informationen aus
  - Pointer zur aktuellen Page Table
  - Fehlercode
  - Programmcounter
  - Adresse, welche den Page Fault ausgelöst hat
- Nutzen Sie das Makro PANIC aus dem Headerfile common.h um die CPU anzuhalten
- **Zusatzaufgabe:** Decodieren Sie den Fehlercode
  erweitern Sie den Page Fault Handler, so dass der Fehlercode in einer leicht lesbaren Form ausgegeben wird

Heap

- erweiterte Speicherverwaltung
- bis jetzt: Speicher anfordern aber keine Freigabe
- Funktion *free* um Speicher wieder freizugeben
- nötig sobald Programme gestartet und wieder beendet werden sollen
- oder wenn Programme dynamisch Speicher anfordern und wieder freigeben
- ohne *free* steht dem System irgendwann kein Speicher mehr zur Verfügung
- Systeme mit statischem Setup (Programme werden nur während der Initialisierungsphase gestartet, kein dynamisches allozieren von Speicher) kommen ohne *free* aus
- unsere bereits implementierte Speicherverwaltung ist eine minimale Implementation eines Heaps

Heap

- James Molly bietet eine einfache Implementation in seinem Tutorial
- wer sein OS später in einer produktiven Umgebung einsetzen möchte, sollte weitere Literatur zu Rate ziehen
- die Art und Weise, wie Speicher alloziert und wieder freigeben wird, beeinflusst die Leistung eines Systems
- durch das Anfordern und Freigeben von Speicher kann es zu einer Speicherfragmentierung kommen

Multitasking

- unser Betriebssystem nutzt nur eine physikalische CPU, auch wenn das System mehrere Kerne zur Verfügung stellt
- ohne Multitasking kann nur eine Applikation ausgeführt werden
- durch Multitasking können mehrere Anwendungen scheinbar parallel ausgeführt werden
- das Betriebssystem teilt jeder Anwendung die CPU für eine bestimmte Zeitspanne zu
- Möglichkeiten die CPU aufzuteilen
  - Round Robin: die CPU-Zeit wird reihum auf alle Anwendung verteilt
  - Prioritätsbasiert: jede Anwendung bekommt eine bestimmte Priorität, es wird immer die Anwendung mit der höchsten Priorität ausgeführt

Multitasking

Taskwechsel

- Programmausführung wird vom Betriebssystem unterbrochen
- Programmstatus (Registerwerte) wird vom Betriebssystem gesichert
- Status der nächsten Anwendung wird wieder hergestellt
- nächste Anwendung wird ausgeführt

Multitasking

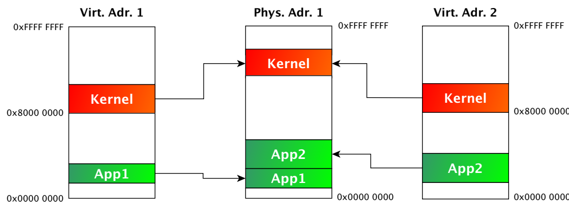
Scheduler

- ist für den Taskwechsel zuständig
- Preemptiv
  - Tasks können vom Scheduler unterbrochen werden
  - Scheduler wird aufgerufen wenn ein Ereignis eingetreten ist (z.B. Timerinterrupt oder anderes externes Ereignis)
- Non-Preemptiv
  - jede Anwendung gibt die CPU selbst frei (z.B. wenn alle Aufgaben erledigt sind oder zu festgelegten Zeitpunkten innerhalb des Programmablaufs)

Multitasking

Virtueller Adressraum

- bis jetzt war ein 1:1 Mapping möglich
- beim Erstellen eines neuen Threads (Anwendung) müssen Teile des Adressraums 1:1 übernommen werden, andere Teile müssen ausgetauscht werden
- der Kernelteil ist in allen virtuellen Adressräumem gleich



User Mode

- eine fehlerhafte Anwendung soll andere Anwendungen weder stören noch das System zum Erliegen bringen
- Einschränkungen beim HW Zugriff nötig
- kein Zugriff auf die Daten und den Code anderer Anwendung
- keine Änderungen am Setup der CPU

User Mode

Besonderheiten beim x86

- kein direkter Wechsel durch einen Befehl möglich
- Umweg über IRET nötig
- Ausführungsmodus wird in den 2 unteren Bits der Segmentregister codiert
- Wechsel von Kernel in User Mode ist kritisch, deshalb sollten alle Interrupts ausgeschaltet werden
- ein enable_irq() ist im User Mode nicht möglich
- damit eine Anwendung im User Mode unterbrochen werden kann, muss der Wert des EFLAGS Register, welches beim IRET geladen wird, entsprechend modifizert werden

System Calls

- Austausch zwischen User und Kernel Mode
- Anwendung muss für bestimmte Operationen den Kernel um Hilfe bitten
- der Kernel stellt jeder Anwendung ein System Call Interface mit bestimmten Funktionen bereit
- System Calls werden traditionell über Software Interrupts abgebildet
- Anzahl der Parameter eines System Calls sind durch das Interface beschränkt
- aus Sicherheitsgründen sollte jede Anwendung einen eigenen Kernel Stack haben

System Calls

x86 Besonderheiten

- die x86 Architektur bietet eine Alternative zur Variante mit Software Interrupts
- Task State Segment (TSS)
- TSS bietet nur minimale Laufzeitvorteile, weswegen meist die oben genannte Variante verwendet wird
- dennoch ist ein minimales TSS Setup nötig
- die Descriptor Tabellen müssen diesbezüglich erweitert werden

Siehe JamesM's kernel development tutorials für den Wechsel in den User Mode und das Bereitstellen von System Calls

Alles Zusammen

- Kernel initialisiert die Hardware und führt danach einen Wechsel in den User Mode aus
- 1. Anwendung z.B. eine Shell wird ausgeführt
- Möchte der User eine neue Anwendung starten, wird ein System Call aufgerufen
- es erfolgt ein Wechsel in den Kernel Mode
- es wird ein neuer virtueller Adressraum für die neue Anwendung aufgesetzt
- danach erfolgt der Wechsel vom Kernel in den User Mode und die neue Anwendung wird ausgeführt
- beim nächsten Timer Interrupt erfolgt der Wechsel in den Kernel, welcher der aktuellen Anwendung die CPU entzieht und diese an die 1. Anwendung (Shell) übergibt
- hat der User keine Eingabe in die Shell gemacht, ruft diese erneut einen System Call auf, um die CPU gleich wieder freizugeben

Ausblick

- VESA-Treiber für die Textausgabe am Monitor schreiben
- Tastaturtreiber entwickeln
- serielle Konsole durch Monitor und Tastatur ersetzten
- serielle Schnittstelle verbessern, neue Chipvarianten unterstützen, automatisches Erkennen der Chipvariante
- eigenen Heap entwickeln
- eigenes kleines Dateisystem schreiben

Literatur

- Modern Operating Systems, Andrew S. Tanenbaum
- Microprocessors PC Hardware and Interfacing, N. Mathivanan
- Linux Device Drivers, Corbet, Rubini und Kroah-Hartman
- Intel x86 Handbücher

Literatur

- JamesM's kernel development tutorials
  http://www.jamesmolloy.co.uk/tutorial_html
- 8250 UART Programming
  http://en.wikibooks.org/wiki/Serial_Programming/8250_UART_Programming
- 8259 PIC
  http://wiki.osdev.org/PIC
- 8253 PIT
  http://www.brokenthorn.com/Resources/OSDevPit.html
  http://www.intel.com/design/archives/periphrl/docs/7203.htm?wapkw=8254
- Wiki OSDev.org
  http://wiki.osdev.org/