Betriebssystem-Entwicklung

2. Einstieg in die BS-Entwicklung

Michael Schöttner

Vorschau

Heinrich Heine
Universität Düsseldorf

- Einordnung
- Übersetzen und Linken
- Boot-Vorgang
- Debugging
- Zusammenfassung

BS-Entwicklung (oft ein harter Kampf)

Heinrich Heine
Universität Düsseldorf

- Erste Schritte: Wie bringt man sein System auf die Zielhardware?
  - Übersetzung
  - Bootvorgang
- Testen und Debugging: was tun, wenn das System nicht reagiert?
  - „printf“ debugging
  - Emulatoren
  - Debugger
  - Remote-Debugger
  - Hardwareunterstützung

Vorschau

Heinrich Heine
Universität Düsseldorf

- Einordnung
- Übersetzen und Linken
- Boot-Vorgang
- Debugging
- Zusammenfassung

Übersetzung – Hello, World?

Heinrich Heine
Universität Düsseldorf

```
#include <iostream>


int main () {
   std::cout << "Hello, World" << std::endl;
}
```

```
$ g++ -o hello hello.cc
```

- Annahme:
  - das Entwicklungssystem läuft unter Linux/x86
  - das Zielsystem ist ebenfalls ein PC
- Läuft dieses Programm auch auf der „nackten“ Hardware?
- Kann man Betriebssysteme überhaupt in einer Hochsprache entwickeln?

Übersetzung – Probleme und Lösungen

Heinrich Heine
Universität Düsseldorf

- Kein dynamischer Binder/Lader vorhanden → alle nötigen Bibliotheken statisch einbinden
- libstdc++ und libc benutzen Linux-Systemaufrufe (insbesondere write)
  - Die normalen C/C++ Laufzeitbibliotheken können nicht benutzt werden
  - Andere Bibliotheken haben wir auch nicht
- Generierte Adressen beziehen sich auf virtuellen Speicher!
  - Die Standardeinstellungen des Binders können nicht benutzt werden.
  - Man benötigt eine eigene Binderkonfiguration.
- Hochsprachencode stellt Anforderungen
  - Registerbelegung, Adressabbildung, Laufzeitumgebung, Stack, ...
  - Ein eigener Startup-Code muss die Ausführung des Hochsprachencodes vorbereiten

Vorschau

Heinrich Heine
Universität Düsseldorf

- Einordnung
- Übersetzen und Linken
- Boot-Vorgang
- Debugging
- Zusammenfassung

Bootstrapping

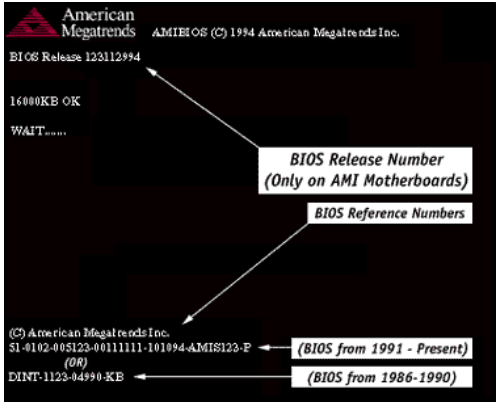
- Bootstrapping: (englisches Wort für Stiefelschlaufe) bezeichnet einen Vorgang bei dem ein einfaches System ein komplexeres System startet.

- Der Name des Verfahrens kommt von der Münchhausen-Methode.
- Diese basiert auf der Geschichte, in der sich Münchhausen am eigenen Schopf aus dem Sumpf gezogen haben will.

BIOS: Boot-Vorgang

- BIOS = Basic Input Output System
  - Firmware auf dem Mainboard, für die Hardware-Initialisierung und Zugriff auf Geräte
  - Stellt viele Funktionen per Software-Interrupt bereit; alles im Real-Mode
  - Gespeichert in einem Chip
    - früher: ROM, später: EEPROM, heute: NAND flash memory
  - Der Begriff BIOS wurde im CP/M Betriebssystem 1975 eingeführt

- Führt zunächst den POST = „Power On Self Test“ durch
  - CPU prüfen → built-in Self-Test der CPU
  - Initialisierung des Chipsatzes & Hauptspeichers
  - Initialisierung der Main-Board-Komponenten



BIOS: Real Mode

- Aktivierter Modus nach dem Einschalten / Reset
- 16 Bit Register, Segmente mit 64 KB
- Nur physikalische 20 Bit Adressen
- Effektive Adresse = (Segment << 4) + Offset

Intel 8086

cs	ax (ah | al)	si	ip
ds	bx (bh | bl)	di	flags
es	cx (ch | cl)	bp	
ss	dx (dh | dl)	sp	

16 Bit Segment	1	2	3	4	
16 Bit Offset		5	6	7	8
20 Bit Adresse	1	7	9	B	8

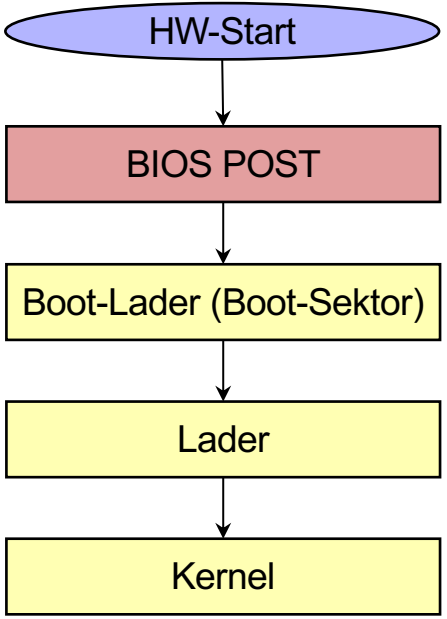
BIOS: Bootstrapping

Heinrich Heine
Universität Düsseldorf

- Ermitteln des ersten / nächsten Bootable Device
  - Test auf die Signatur `0x55AA` im Sektor 0 = Master Boot Record (MBR)
  - Bei Signatur- oder Lese-Fehler → nächstes Device
- Falls OK, lädt BIOS den ersten Sektor ab bestimmte Adresse und springt den Code an
  - Erster Lader muss also in einen Sektor (512 Byte) passen → Assembler
  - Falls Aufruf zurückkehrt → nächstes Device
- Der Boot-Lader im Boot-Sektor lädt mithilfe des BIOS den eigentlichen Lader
  - Evt. Zeigt dieser ein Auswahlmenü an, falls mehrere Betriebssysteme installiert sind
  - Anschließend wird ein Kernel-Image geladen und vom Lader angesprungen

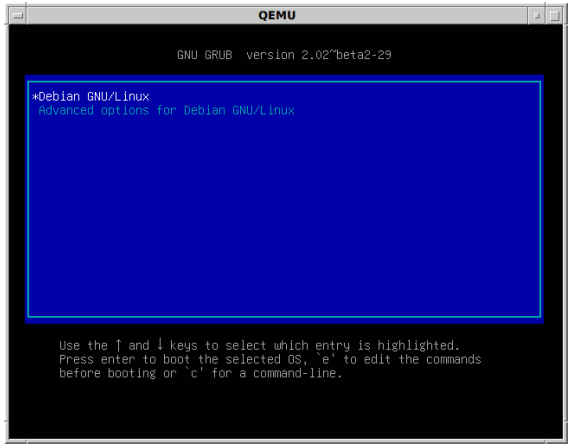
BIOS: Ablauf Bootvorgang

- Kernel hat eine Einstiegsfunktion, geschrieben in Assembler
  - Schaltet in den Protected Mode (32 Bit)
  - Und evt. danach in den Long-Mode (64 Bit)
  - Initialisiert die Interrupt-Tabelle und –Controller
  - Und springt danach eine Funktion in einer Hochsprache an, meist eine C Funktion



Boot-Vorgang mit GRUB

- GRUB = GNU GRand Unified Bootloader
  - Bietet ein Boot-Menü sowie verschiedene Boot-Parameter
  - Ist eine Referenzimplementierung des Multiboot-Standards
    - https://www.gnu.org/software/grub/manual/multiboot/multiboot.html
  - Kernel-Image ist eine ELF-Datei und hat am Anfang einen Multiboot-Record
  - Grub schaltet in den Protected Mode (32 Bit) und lädt dann die in `grub.cfg` definierte ELF-Datei und springt diese an.
  - Sofern im Multiboot-Record angegeben wird auch in einen VGA-Grafikmodus geschaltet und es gibt auch noch jede Menge weitere Infos
- GRUB verwenden wir für hhuTOS



Boot-Vorgang mit UEFI

Heinrich Heine
Universität Düsseldorf

- Unified Extensible Firmware Interface (UEFI)

- Software Interface zwischen Betriebssystem und Hardware
  - 64-Bit Code statt 16-Bit Real-Mode des BIOS
  - Device Drivers für Pre-boot Environment (auch Netzwerkstack)
  - Bootmanager, Disk Support: Unterstützung der GUID Partition Tabelle,
  - Dateisystem Support (FAT32), textuelle und graphische Konsole
  - Erweiterungen: können von persistenten Speichern geladen und installiert werden
  - Unterstützung von Pre-Boot Applikationen

- Weitere Infos: http://software.intel.com/en-us/articles/about-uefi/

Vorschau

Heinrich Heine
Universität Düsseldorf

- Einordnung
- Übersetzen und Linken
- Boot-Vorgang
- Debugging
- Zusammenfassung

`printf`-Debugging

Heinrich Heine
Universität Düsseldorf

- Gar nicht so einfach, da es `printf()` ohne Bibliotheken nicht gibt!

- `printf()` ändert auch das zeitliche Verhalten
  - mit `printf()` tritt der Fehler plötzlich nicht mehr / anders auf
  - das gilt gerade auch bei der Betriebssystementwicklung
  - häufig führen Fehler zu einem Reset,
  - wodurch der Bildschirminhalt sofort gelöscht wird

- „Strohhalme“
  - serielle Schnittstelle
  - PC-Lautsprecher
  - blinkende LED

Emulatoren

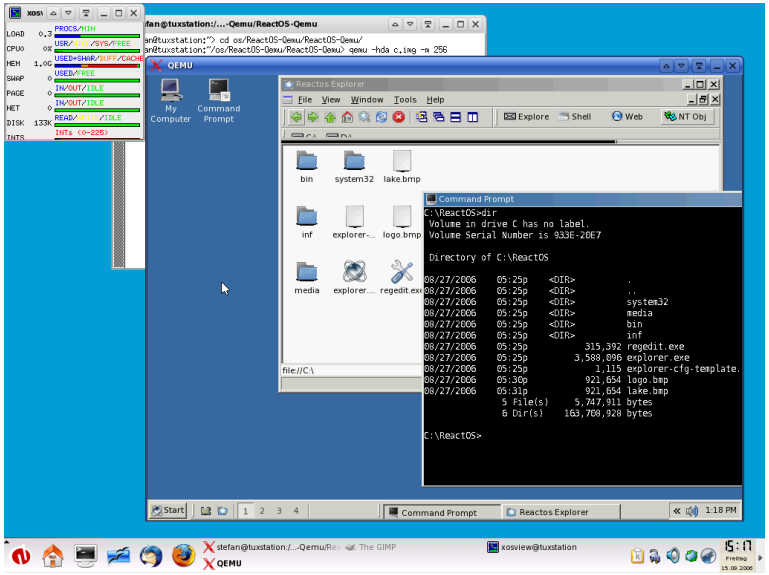
Heinrich Heine
Universität Düsseldorf

- Emulieren in Software reale Hardware
  - Meist kann ein Debugger verwendet werden,
  - Um das Gast-Betriebssystem im Emulator zu debuggen
  - Der Debugger beeinflusst aber wieder das System, aber dennoch sehr hilfreich
- Vorsicht: am Ende muss das System auf realer Hardware laufen!
  - in Details können sich Emulator und reale Hardware unterscheiden!
- "virtuelle Maschinen" und "Emulatoren" sind nicht gleichbedeutend
  - zum Beispiel in VMware wird kein x86 Prozessor emuliert, sondern ein vorhandener Prozessor führt Maschinencode in der VM direkt aus

Beispiel: Qemu

Heinrich Heine Universität Düsseldorf

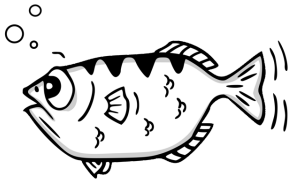
- Emuliert verschiedene Prozessoren, z.B. x86 (auch mehrere Kerne)
- Emuliert einen kompletten PC
  - Speicher, Geräte (selbst Sound- und Netzwerkkarte)
- Implementiert in C
- Entwicklungsunterstützung
  - Debugger-Unterstützung (GDB-Stub)
  - Dump bei einem Absturz
- Verwenden wir in den Übungen



Bilder: https://en.wikipedia.org/wiki/QEMU

Beispiel: gdb

- gdb = GNU Debugger (wichtig für die Übungen)



- Nochmals „Hello, World“

```
#include <iostream>

int main () {
   std::cout << "Hello, World" << std::endl;
}
```

- Wichtig: Erzeugen von Debug-Informationen → Compilieren mit Option `-g`

```
$ g++ -g -o hello hello.cc
```

Beispiel: gdb

```
betriebssysteme@bs:~$ gdb hello
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
...
(gdb) break main
Breakpoint 1 at 0x400861: file hello.cc, line 4.
(gdb) run
Starting program: /home/student/hello

Breakpoint 1, main () at hello.cc:4
4    std::cout << "Hello";
(gdb) next
5    std::cout << ", World";
(gdb) next
6    std::cout << std::endl;
(gdb) continue
Continuing.
Hello, World
[Inferior 1 (process 2745) exited normally]
(gdb) quit
```

Setzen eines Haltepunktes (engl. breakpoint)

Start des Programms

Ablauf in Einzelschritten

Fortsetzung des Programms

Debugging-Unterstützung bei x86

Heinrich Heine
Universität Düsseldorf

- INT3 Instruktion löst "breakpoint interrupt" aus (ein TRAP)
  - Wird gezielt durch den User-Level Debugger (z.B. gdb) im Code platziert
  - Der TRAP-Handler im Kernel leitet den Kontrollfluss in den Debugger

- Durch Setzen des Trap Flags (TF) im Statusregister (EFLAGS) wird nach jeder Instruktion ein "debug interrupt“, INT 1, ausgelöst
  - kann für die Implementierung des Einzelschrittmodus im Debugger genutzt werden

- mit Hilfe der Debug Register DR0-DR7 (ab i386) können bis zu vier Haltepunkte überwacht werden, ohne den Code manipulieren zu müssen
  - erheblicher Vorteil bei Code im ROM/FLASH oder nicht-schreibbaren Speichersegmenten
  - Verwendet von Kernel-Level Debuggern, z.B. kdb

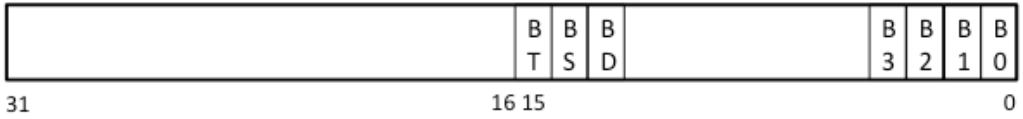
Debug-Register bei x86

Heinrich Heine
Universität Düsseldorf

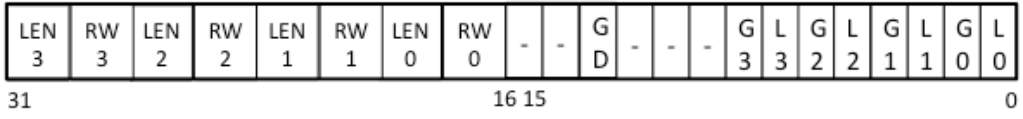
Breakpoint Register

Breakpoint 0: lineare Adresse
Breakpoint 1: lineare Adresse
Breakpoint 2: lineare Adresse
Breakpoint 3: lineare Adresse
reserviert
reserviert

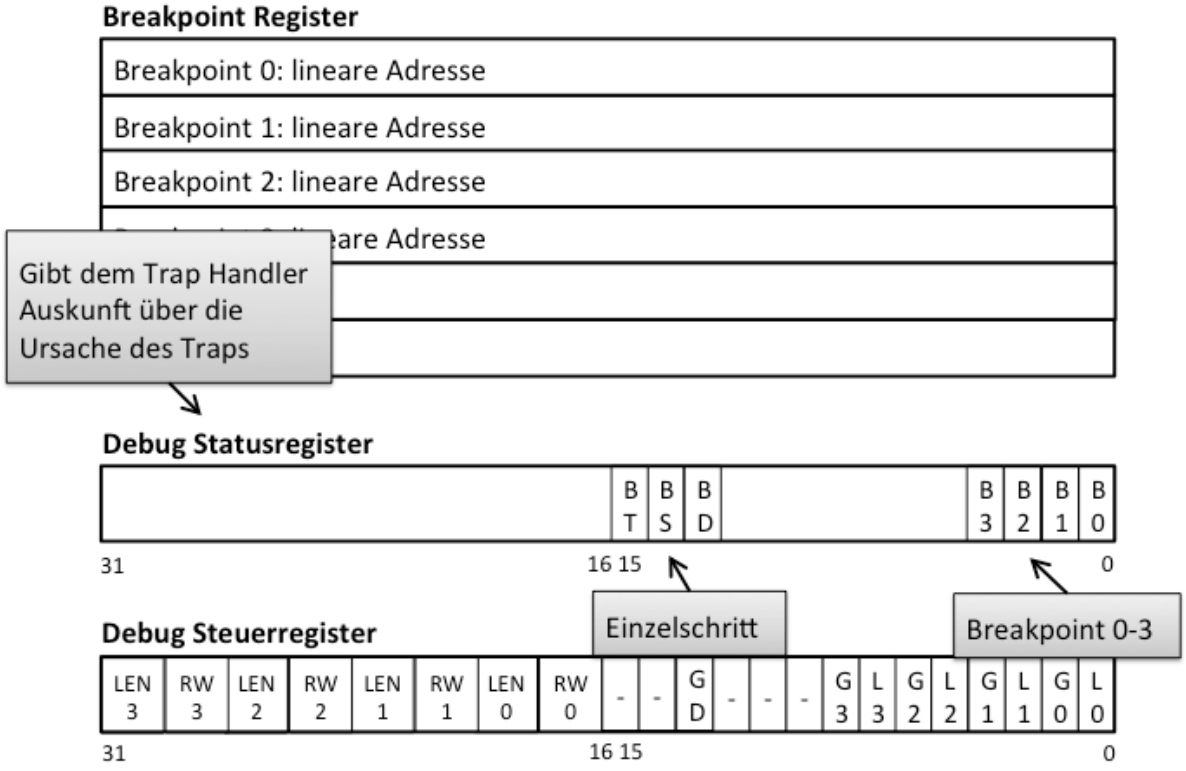
Debug Statusregister



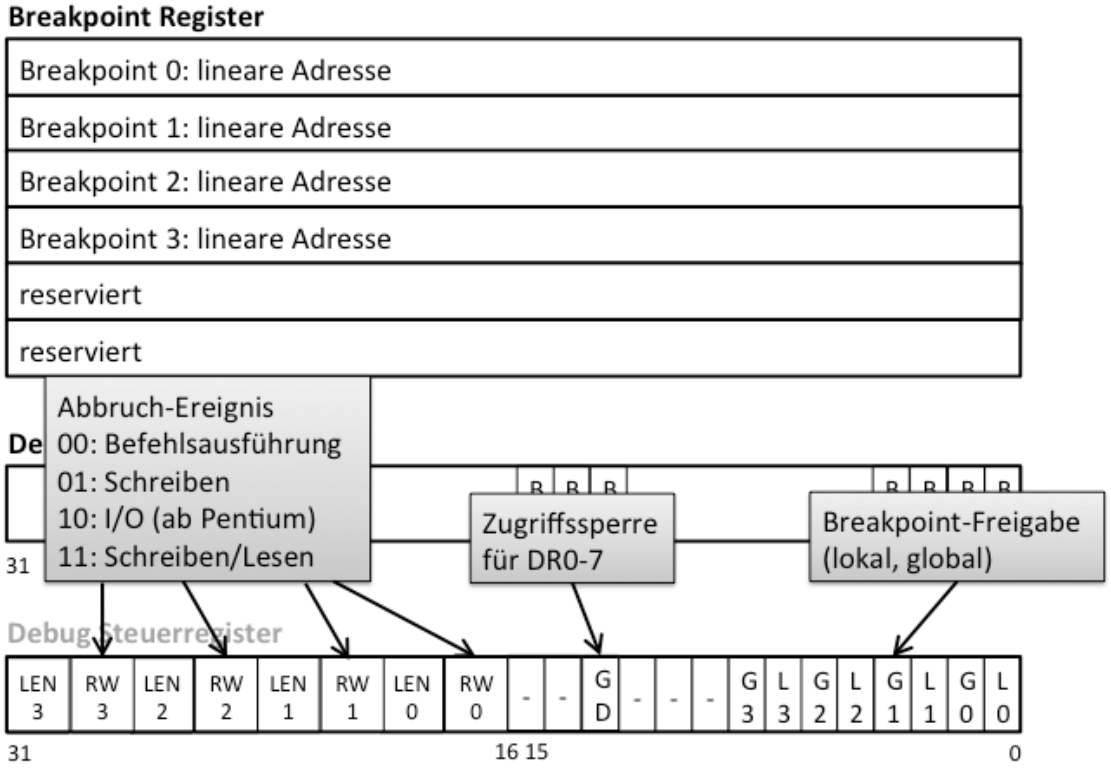
Debug Steuerregister



Debug-Register bei x86



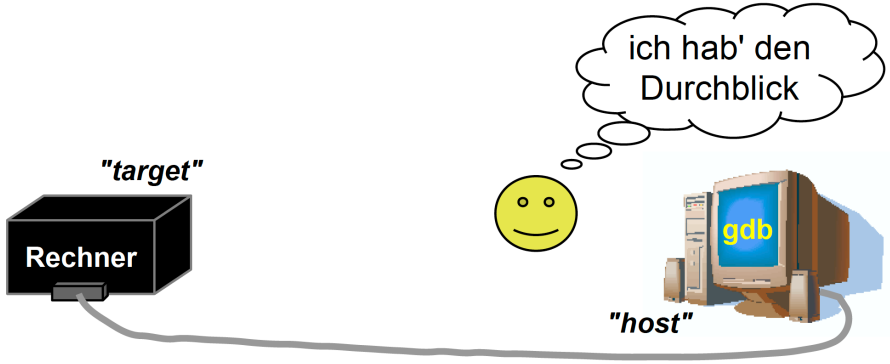
Debug-Register bei x86



Remote-Debugging

- Bietet die Möglichkeit Programme auf Plattformen zu debuggen, die (noch) kein interaktives Arbeiten erlauben
  - setzt eine Kommunikationsverbindung voraus (seriell, Ethernet, ...)
  - erfordert einen Gerätetreiber
  - der Zielrechner kann auch ein Emulator sein (z.B. Qemu)

- Die Debugging-Komponente auf dem Zielsystem (stub) sollte möglichst einfach sein



Remote-Debugging

Heinrich Heine
Universität Düsseldorf

- Das Kommunikationsprotokoll ("GDB Remote Serial Protocol" - RSP)
  - Spiegelt die Anforderungen an den gdb *stub* wieder
  - Basiert auf der Übertragung von ASCII Zeichenketten
  - Nachrichtenformat: `$<Kommando oder Antwort>#<Prüfsumme>`
  - Nachrichten werden unmittelbar mit `+` (OK) oder `-` (Fehler) beantwortet

- Beispiele:
  - `$g#67` → Lesen aller Registerinhalte
    - Antwort: `+ $123456789abcdef0...#...` → Reg. 1 ist `0x12345678`, Reg. 2 ist `0x9ab...`
  - `$G123456789abcdef0...#...` → Setze Registerinhalte
    - Antwort: `+ $OK#9a` → hat funktioniert
  - `$m4015bc,2#5a` → Lese 2 Bytes ab Adresse `0x4015bc`
    - Antwort: `+ $2f86#06` → Wert ist `0x2f86`

Remote-Debugging mit Qemu

Heinrich Heine
Universität Düsseldorf

```
student@student-pc:~/hhuTOSc/Aufgabe5$ make qemu-gdb &
[1] 5401
student@student-pc:~/hhuTOSc/Aufgabe5$ qemu-system-x86_64 -cdrom ./build/system.iso -k en-us -s -S -soundhw pcspk -vga s
td
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]

student@student-pc:~/hhuTOSc/Aufgabe5$ make gdb
gdb -x /tmp/gdbcommands.1000 ./build/system
GNU gdb (Ubuntu 8.3-0ubuntu1) 8.3
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./build/system...
Breakpoint 1 at 0x2024c4: main. (2 locations)
0x000000000000fff0 in ?? ()

Breakpoint 1, main () at main.cc:82
82      int main() {
(gdb) break aufgabe05()
Breakpoint 2 at 0x202bd1: file main.cc, line 69.
(gdb) c
Continuing.

Breakpoint 2, aufgabe05 () at main.cc:69
69          PreemptiveThreadDemo *demoApp = new PreemptiveThreadDemo();
(gdb)
```

QEMU [Paused]

Machine View

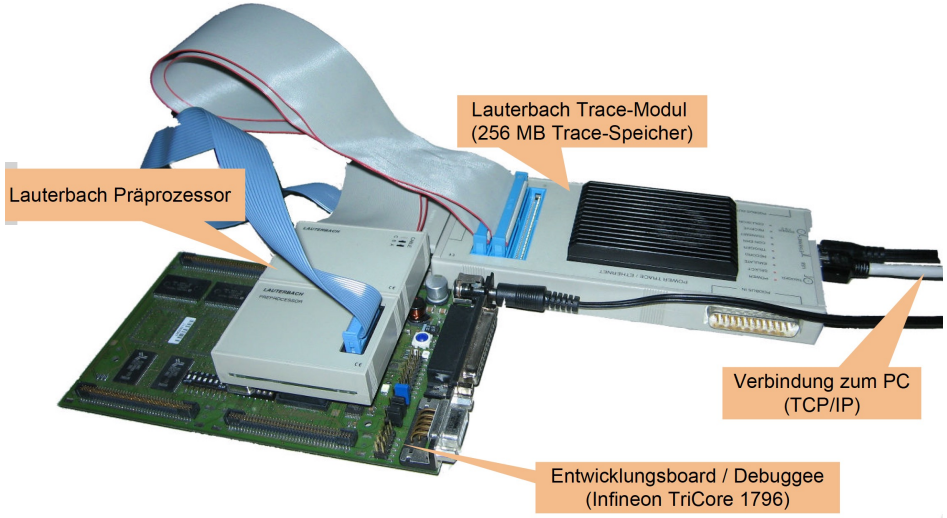
```
hhuTOS 0.5
==========

Unterstuetzte Funktionen:
   - Bildschirmausgaben
   - Sound ueber den PC-Lautsprecher
   - Einfache Heap-Verwaltung
   - Tastatureingaben
   - Interrupts
   - Koroutinen
   - Scheduler
   - preemptive Threads
```

Remote-Debugging mit Qemu

Heinrich Heine
Universität Düsseldorf

- Viele Prozessorhersteller bieten Hardwareunterstützung für Debugging auf ihren Chips (OCDS – On Chip Debug System)
- I.d.R. einfaches serielles Protokoll zwischen Debugging-Einheit und externem Debugger (Pins sparen!)



Vorschau

- Einordnung
- Übersetzen und Linken
- Boot-Vorgang
- Debugging
- Zusammenfassung

Zusammenfassung

Heinrich Heine
Universität Düsseldorf

- Betriebssystementwicklung unterscheidet sich deutlich von Applikationsentwicklung
  - Bibliotheken fehlen
  - Die „nackte“ Hardware bildet die Grundlage
- Die ersten Schritte sind oft die schwersten
  - Übersetzung
  - Bootvorgang
  - Systeminitialisierung
- Komfortable Fehlersuche erfordert eine Infrastruktur
  - Gerätetreiber für `printf`-Debugging
  - STUB und Verbindung/Treiber für Remote Debugging
  - Hardware Debugging-Unterstützung