Hobby-Betriebssysteme unter Linux entwickeln

Hans-Georg Eßer
Univ. Erlangen-Nürnberg
h.g.esser@cs.fau.de

Linux-Infotag 2013
Linux User Group Augsburg
23.03.2013

Was? Und warum? (1/2)

- Zahlreiche Projekte, die kleine Hobby-Systeme entwickeln
- Ansatz ist i.d.R. nicht, eine Alternative zu professionellen Systemen (Linux etc.) zu schaffen
- sondern: Grundlagen von BS verstehen und selbst umsetzen (→ Spaßfaktor)
- Mitarbeit an etablierten Systemen erfordert sehr umfangreiche Einarbeitung

Was? Und warum? (2/2)

- Nötige Komponenten einer Eigenentwicklung
  - Booten, Wechsel in Protected Mode
  - Speicherverwaltung (Paging)
  - Interrupt-Behandlung (Tastatur, Disk-I/O, Timer)
  - Prozess- und Thread-Verwaltung
  - Prozess- und Thread-Scheduling, Context Switch
  - System Calls
  - Dateisystem, Floppy-/Platten-/Ramdisk-Treiber
  - Netzwerk (nicht zwingend nötig)

Linux als Entwicklungsplattform

- Alle nötige Software frei verfügbar
  - Compiler, Assembler, Linker
  - Debugger
  - virtuelle Maschinen / PC-Emulation (mit „Anschluss" an Debugger)
  - Editor oder andere Entwicklungsumgebung
  - Header-Dateien von Linux (fürs schnelle Nachschlagen von Typdeklarationen)

Beispielprojekt: ULIX-i386

- ULIX (Literate Unix)
- kleines Unix-ähnliches OS für i386
- work in progress
- bisher: Interrupts, Paging, Prozesse, Round-Robin-Scheduler, Dateisystem, System Calls, Anfänge einer *libc*
- Implementation und Dokumentation mit „Literate Programming“ (D. E. Knuth)

Motivation zu Ulix

- Felix Freilings Vorlesung „Betriebssysteme“ an der Uni Mannheim
- Lehrbuch „Betriebssysteme“, das Theorie und Implementierung nebeneinander zeigt
- Tanenbaums Minix-Buch: vorne Theorie, hinten Code
- Literate Programming erlaubt Integration

→ meine Doktorarbeit an der FAU Erlangen-Nürnberg

Obligatorischer Screenshot, Ulix 0.06

```
Ulix-i386 0.06                                                              -
Paging activated (CR0, CR3 loaded).
Physical RAM (64 MB) mapped to 0xD0000000-0xD3FFFFFF.
FDC: fda is 1.44M, fdb is 1.44M
Modul aktiviert.
Setting Status Line.
initial_stack = 0xc01e5ec4
Starting Shell. Type exit to quit.
DEBUG. start_from_disk, kstack    = 0xbffff000
DEBUG. start_from_disk, tss.esp0 = 0xc0000000
ULIX fork
Ulix Usermode Shell. Commands: exit, ps, fork, ls, head
Press [Shift+Esc] to launch kernel mode shell (reboot to get back here)
esser@ulix:/$ ps
PID PPID STATE CMD
  1    0 READY idle
  2    1 READY sh
esser@ulix:/$ ls
makefs          9464    1
makefs.c        2275   20
sh              4096   25
Makefile          42   33
esser@ulix:/$ A_

Ulix-i386                                                          00:00:32
```

Benutzte Tools

- Entwicklungsumgebung:
  Debian-VM in VirtualBox (→ ändert sich nicht),
  unter Linux und OS X
- Compiler/Assembler: gcc, nasm
- Virtuelle Maschine: qemu, Bochs
- Debugger: qemu + gdb; Bochs (hat integrierten Debugger)
- Zum Booten: GRUB (v1)
- Boot-Diskette: mtools (FAT)

Programmiersprachen

- Hauptteil des Codes: C
- Kleine Teile: Assembler (meist inline im C-Code)

```
[esser@dev:Code]$ wc -l ulix.c ulix.h Apps/C/ulixlib.c
Apps/C/ulixlib.h Apps/C/testprog.c start.asm
    7353 ulix.c
     112 ulix.h
     208 Apps/C/ulixlib.c
      77 Apps/C/ulixlib.h
     131 Apps/C/testprog.c
     677 start.asm
    8558 total
[esser@dev:Code]$ ls -l ulix.bin
-rwxr-xr-x  1 esser  users  179290 11 Mär 18:26 ulix.bin
```

(keine Kommentare im generierten C-/Assembler-Code → Literate Programming)

Zweimal Assembler-Syntax (1)

Zwei Standards für x86-Assembler:

- nasm verwendet Intel-Syntax

  `mov eax, esp` bedeutet: eax := esp

- gcc-Inline-Assembler nutzt AT&T-Syntax

  `movl %esp, %eax` bedeutet: eax := esp

- man kann gcc aber auf Intel-Syntax umstellen

Zweimal Assembler-Syntax (2)

- Intel-Syntax im gcc-Inline-Assembler

```
asm ("                                  \
   .intel_syntax noprefix;   \
   starta: mov eax, 0x1001;  \
           mov ebx, 'A';     \
           int 0x80;         \
   .att_syntax;              \
");
```

- Eigener Prä-Prozessor erlaubt solche Anweisungen:

```
asm {
   starta: mov eax, 0x1001   // comment
           mov ebx, 'A'      // more comment
           int 0x80
}
```

Drei Code-Beispiele

Beschreibung der bisher implementierten Features würde ganzen Tag füllen, darum nur ein paar Beispiele:

- Booten und Umschalten in Protected Mode
- Paging
- System Calls

Booten (1)

- BIOS startet Bootloader-Code, z. B. Grub
- Grub lädt Kernel und aktiviert ihn
- System läuft zunächst im Real Mode (16 Bit), muss in Protected Mode wechseln (32 Bit)
- Für Wechsel in Protected Mode:
  - GDT (Global Descriptor Table) vorbereiten
  - Segmentregister laden (cs, ds, …)
  - `jmp cs:address` (far jump), aktiviert PM

Booten (2)

- Grobe Speicheraufteilung:
  - Kernel an Adresse 0xc000.0000 kompiliert
  - Global Descriptor Table (GDT) definiert Segmente mit Offset 0x4000.0000 (→ auf alle Adressen wird dieser Wert addiert)
  - Kernel nutzt dann effektiv die (phys.) Adressen ab 0 (denn 0xc000.0000 + 0x4000.0000 = 0)
  - Wenn Protected Mode und Segmente aktiv sind, kann man später Paging aktivieren
  - Das ist der „Higher Half Trick“, http://www.osdever.net/tutorials/pdf/memory1.pdf

Booten (3)

```
[section .setup]
start:
    lgdt [gdt]
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax

    ; jump to
    ; protected mode
    jmp 0x08:prot

[section .text]
prot:
    ; now in
    ; protected mode
    ...
```

```
gdt:
    ; size of the GDT
    dw gdt_end - gdt_start - 1
    ; linear address of GDT
    dd gdt_start

gdt_start:
    dd 0, 0      ; null gate
    ; code selector 0x08: base 0x40000000,
    ; limit 0xFFFFFFFF, type 0x9A,
    ; granularity 0xCF
    db 0xFF, 0xFF, 0, 0, 0, 10011010b,
       11001111b, 0x40
    ; data selector 0x10: base 0x40000000,
    ; limit 0xFFFFFFFF, type 0x92,
    ; granularity 0xCF
    db 0xFF, 0xFF, 0, 0, 0, 10010010b,
       11001111b, 0x40
gdt_end:
```

Booten (4)

- Kernel auf Boot-Diskette kopieren

```
mcopy -o -i ulixboot.img ulix.bin ::
```

- Dabei ist ulixboot.img ein FAT-Floppy-Image mit Grub-Konfiguration in menu.lst:

```
timeout 5

title ULIX-i386  (c) 2008-2013 F. Freiling & H.-G. Esser
root (fd0)
kernel /ulix.bin
```

- Booten im qemu:

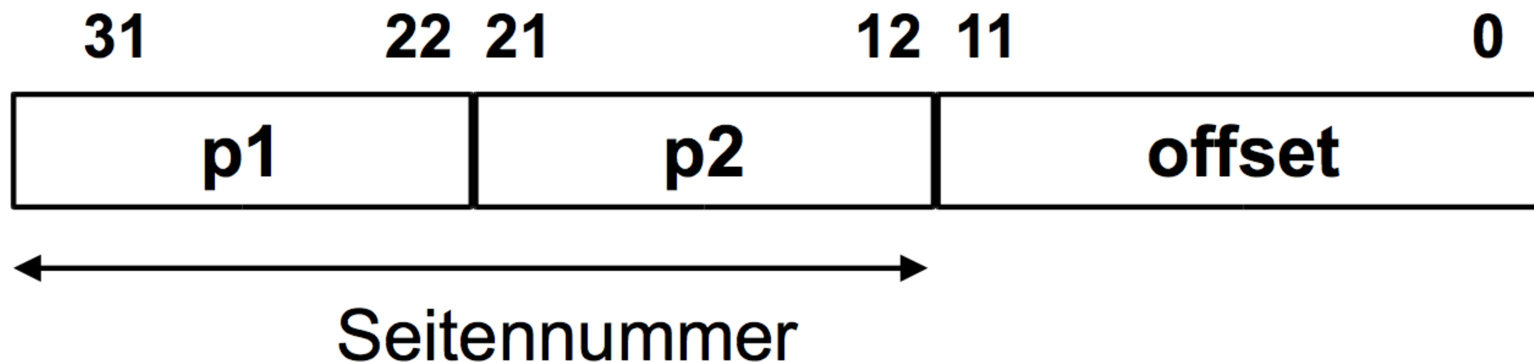
```
qemu -m 64 -fda ulixboot.img -fdb ulixdata.img \
  -d cpu_reset -s -serial mon:stdio
```

Paging (1)

- Nächster Schritt: Paging (virtuellen Speicher) aktivieren
- Problem: Nach dem Einschalten von Paging sind die phys. Adressen nicht mehr verfügbar → was ist die nächste Instruktion?
- Lösung: Seitentabelle vorbereiten, die Teil des phys. Speicher 1:1 auf virt. Speicher mappt („identity mapping“)

Paging (2)

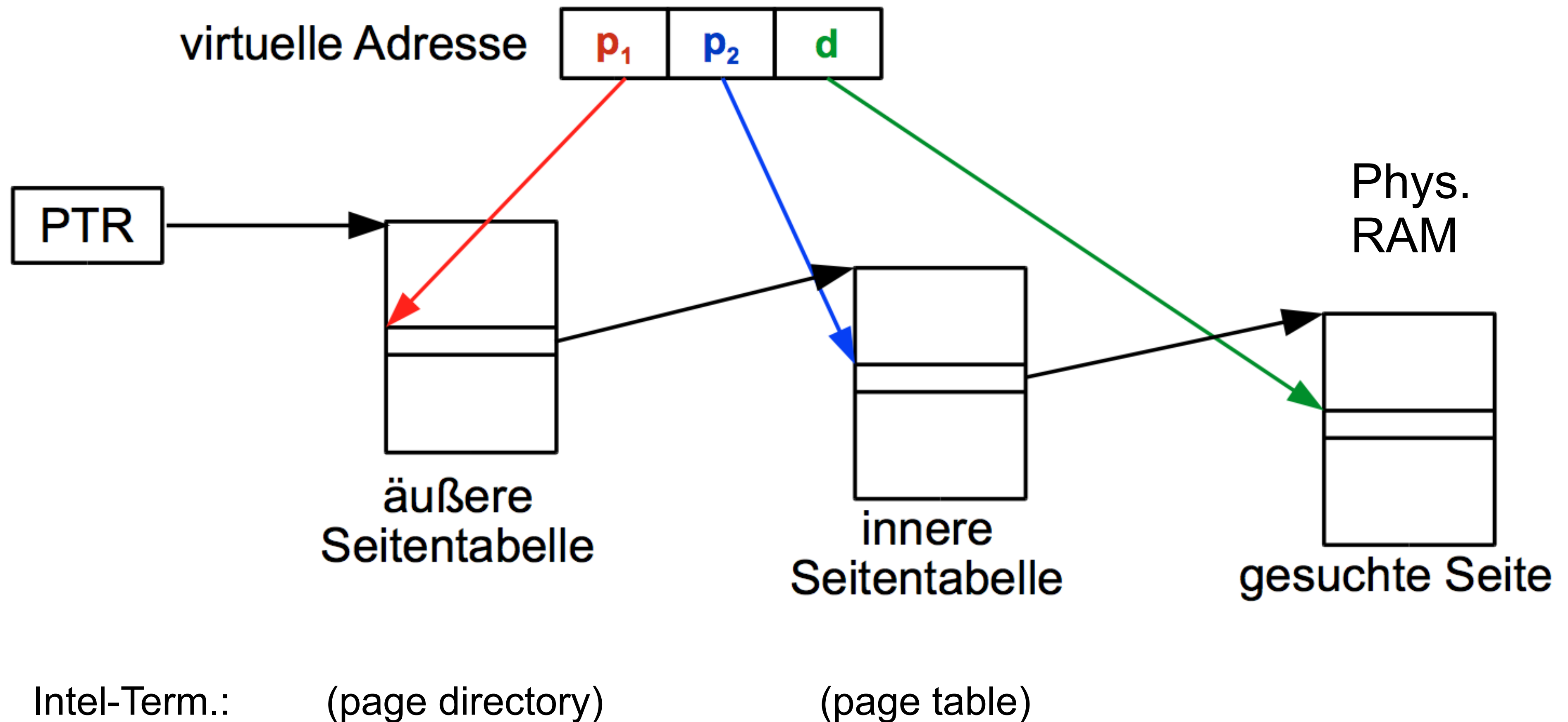
- Theorie: Intel-Paging ist zweistufig;
  32 Bit lange Adresse zerlegt in
  - 10 Bit (31..22), Index in Page Directory
  - 10 Bit (21..12), Index in Page Table
  - 12 Bit (11..0), Offset, innerhalb einer 4 KB großen Seite



Paging (3)

- Grundlage: Seitentabelle besteht aus
  - Page Directory (zeigt auf bis zu 1024 Page Tables)
  - Page Table (zeigt auf bis zu 1024 Page Frames)
  - Jeder Frame ist 4 KB groß, insgesamt also 1024 x 1024 x 4 KB (= 4 GB) verwaltet
- Vokabular
  - Page Table Descriptor: ein Eintrag im Page Directory, der auf die phys. Adresse einer Page Table zeigt
  - Page Descriptor: ein Eintrag in einer Page Table, der auf die phys. Adresse eines Page Frames zeigt

Paging (4)



Paging (5)

```
< create identity mapping >  :=
  for (int i=0; i<1024; i++) {
    //  map first 1024 pages (4 MByte)
    < identity map page i in kernel page table >
  };


< identity map page i in kernel page table >  :=
  fill_page_desc (
    &(kernel_pt->pds[i]),   //  address of i-th entry
    true,                   //  present: yes
    true,                   //  writeable: yes
    true,                   //  user accessible: yes
    false,                  //  dirty: no
    i*4096                  //  physical address: start of i-th frame
  );
```

Paging (6)

```
typedef struct {
  uint present          : 1;  //  0       Seite vorhanden?
  uint writeable        : 1;  //  1       beschreibbar?
  uint user_accessible  : 1;  //  2       Zugriff im User Mode ok?
  uint pwt              : 1;  //  3
  uint pcd              : 1;  //  4
  uint accessed         : 1;  //  5
  uint dirty            : 1;  //  6
  uint zeroes           : 2;  //  8.. 7
  uint unused_bits      : 3;  // 11.. 9
  uint frame_addr       : 20; // 31..12   phys. Frame-Nummer
} page_desc;


void fill_page_desc (page_desc *pd, uint present, uint
  writeable, uint user_accessible, uint dirty, uint frame_addr)
{
  // zero out page descriptor
  memset (pd, 0, sizeof(pd));

  // enter the argument values in the right elements
  pd->present         = present;
  pd->writeable       = writeable;
  pd->user_accessible = user_accessible;
  pd->dirty           = dirty;
  pd->frame_addr      = frame_addr >> 12;    // right shift, 12 bits
};
```

Paging (7)

- Für jeden Prozess eigene Seitentabelle

Adressbereich	Inhalt
FFFF.FFFF D000.0000	Physikalischer Speicher in virtuellen Speicher gemappt (Zugriff nur im Kernel-Mode)
CFFF.FFFF … C000.0000	Kernel-Code und -Daten (für Prozess nur beim Wechsel in Kernel-Mode sichtbar)
BFFF.FFFF … 0000.0000	Prozess-Code und -Daten (für Prozess immer sichtbar) – Programm wird an virt. Adresse 0 geladen – wichtig fürs Kompilieren von Ulix-Programmen

Paging (8)

- Zugriff auf phys. Speicher durch Mapping-Trick
- Makros PEEK und POKE (kennt noch jemand Homecomputer?)

```
typedef unsigned char uchar;
#define PHYSICAL(x) ((x)+0xd0000000)

#define PEEK(addr)       (*(uchar *)(addr))
#define POKE(addr, b)    (*(uchar *)(addr) = (b))

#define PEEKPH(addr)     (*(uchar *)(PHYSICAL(addr)))
#define POKEPH(addr, b) (*(uchar *)(PHYSICAL(addr)) = (b))
```

System Calls (1)

- ähnliches System-Call-Interface wie bei Linux
- über `int 0x80`
- Beispiel für Aufruf in *libc*-Implementierung:

```
inline int syscall4 (int eax, int ebx, int ecx, int edx) {
  int result;
  asm ( "int $0x80" : "=a" (result) :
        "a" (eax), "b" (ebx), "c" (ecx), "d" (edx) );
  return result ;
}

#define __NR_read 3
int read (int fd, void *buf, size_t nbyte) {
  return syscall4 (__NR_read, fd, (unsigned int) buf, nbyte);
};
```

System Calls (2)

- Im Ulix-Kernel
  - Interrupt-Handler für IRQ 0x80
  - prüft EAX-Register (enthält Syscall-Nummer)
  - springt (über Syscall-Handler-Tabelle) in Sycall-Handler für diese Syscall-Nummer

```
#define MAX_SYSCALLS 0x8000
void *syscall_table[MAX_SYSCALLS];
```

  - Eintragen neuer Syscalls:

```
void insert_syscall (int syscallno, void* syscall_handler) {
  if (syscallno < MAX_SYSCALLS)
    syscall_table[syscallno] = syscall_handler;
  return;
};
```

System Calls (3)

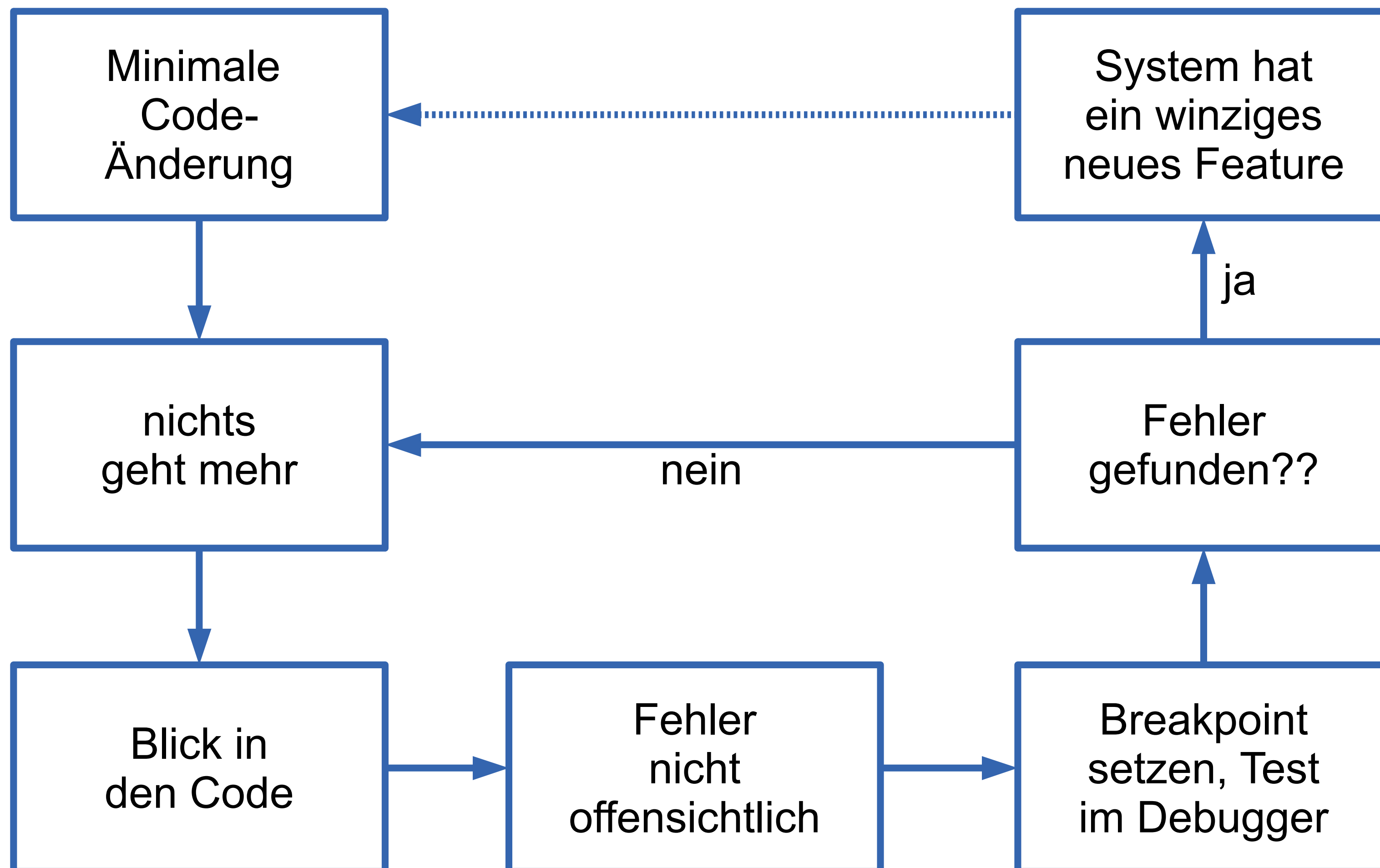
- Beispiel für `read()` System Call
  - bei Initialisierung des Systems:

```
#define __NR_read  3
insert_syscall (__NR_read, syscall_read);
```

  - Syscall Handler:

```
void syscall_read (struct regs *r) {
  // erwartet: ebx: fd, ecx: *buf, edx: nbytes
  int fd = r->ebx;
  char* buf = (char*) r->ecx;
  int nbytes = r->edx;
  r->eax = simplefs_read (fd, buf, nbytes);
};
```

Praxis der OS-Entwicklung (1)



Praxis der OS-Entwicklung (2)

- Debuggen mit qemu und gdb:
- qemu akzeptiert an Port 1234 eine gdb-Verbind.

```
(gdb) target remote localhost:1234
(gdb) cont
Continuing.
^C
Program received signal SIGINT, Interrupt.
0xc010727d in ?? ()
(gdb) info registers
eax            0xc010727d    -1072663939
ecx            0x300        768
edx            0x20         32
ebx            0x0      0
esp            0xbfffffac    0xbfffffac
ebp            0xc01e5e68    0xc01e5e68
esi            0x163000     1454080
edi            0xc01e5ec4    -1071751484
eip            0xc010727d    0xc010727d
eflags         0x46         70
cs             0x8      8
ss             0x10         16
ds             0x10         16
es             0x10         16
fs             0x10         16
gs             0x10         16
```

Praxis der OS-Entwicklung (3)

- Schöner mit Bochs und dem eingebauten grafischen Debugger
- Der Bochs-Debugger interpretiert die (funktionslose) Anweisung xchg bx,bx als „Magic Breakpoint“
- zeigt neben Registern und Speicherinhalten auch Seitentabellen übersichtlich an

Praxis der OS-Entwicklung (4)

Bochs x86 emulator, http://bochs.sourceforge.net/

```
Ulix-i386 0.06
Paging activated (CR0, CR3 loaded).
Physical RAM (64 MB) mapped to 0xD0000000-0xD3FFFFFF.
FDC: fda is 1.44M, fdb is 1.44M
FDC: can't send byte to controller
exit
FDC: can't recalibrate
FDC: seek error on drive
Modul aktiviert.
Setting Status Line.
initial_stack = 0xc01e5e
Starting Shell. Type exi
DEBUG. start_from_disk,
DEBUG. start_from_disk,
ULIX fork
```

7. bochs

```
/usr/cross/bin/i586-elf-ld: warning: cannot find entry symbol main; defaulting to 0000000
Bochs x86 Emulator 2.4.5
Build from CVS snapshot, on April 25, 2010
00000000000i[     ] reading configuration from bochsrc.txt
00000000000i[     ] Stopping on magic break points
```

Bochs Enhanced Debugger

Commands View Options Help

Continue [c] | Step [s] | Step N [s ###] | Refresh | Break [^C]

Reg Name	Hex Value	Decimal
eax	00000033	51
ebx	c011b380	3222385536
ecx	00000000	0
edx	000000e9	233
esi	00000030	48
edi	00000002	2
ebp	bffff68	3221225320
esp	bffffed0	3221225168
eip	c0106ddb	3222302171
eflags	00000002	

L.Address	Bytes	Mnemonic
c0106ddb	(5) ...	mov eax, dword ptr ds:0xc016
c0106de0	(3) ...	lea edx, dword ptr ds:[eax+eax*2]
c0106de3	(3) ...	lea edx, dword ptr ds:[eax+edx*4]
c0106de6	(3) ...	lea eax, dword ptr ds:[eax+edx*4]
c0106de9	(4) ...	mov eax, dword ptr ds:[ebx+eax*4+
c0106ded	(1) 50	push eax
c0106dee	(1) ...	ret
c0106def	(3) ...	add esp, 0x00000010
c0106df2	(1) FA	cli
c0106df3	(3) ...	lea esp, dword ptr ss:[ebp-12]
c0106df6	(1) ...	pop ebx
c0106df7	(1) 5E	pop esi

L.Address	is mapped to P.Address
0x00000000 - 0x00007FFF	0x0000000000412000 - 0x0000000
0xBFFFF000 - 0xBFFFFFFF	0x0000000000410000 - 0x0000000
0xC0000000 - 0xC03FDFFF	0x0000000000000000 - 0x0000000
0xC03FE000 - 0xC03FFFFF	0x0000000000400000 - 0x0000000
0xC0400000 - 0xC0400FFF	0x000000000040B000 - 0x0000000
0xC0401000 - 0xC0403FFF	0x000000000040D000 - 0x0000000
0xD0000000 - 0xD3FFFFFF	0x0000000000000000 - 0x0000000

(0) Magic breakpoint

Break | CPU: PMode (32) (PG) | t= 71770226 | IOPL=0 id vip vif ac vm rf nt of df if tf sf zf af pf cf

Ulix-i386

CTRL + 3rd button enables mouse

Mitarbeit

Teile der Ulix-Implementation als Bachelor-Arbeiten an Studenten abgegeben

- virtuelles Dateisystem, RAM-Disk
  (Liviu Beraru, FH Nürnberg; fertig)
- ELF-Programm-Loader
  (Frank Kohlmann, FH Nürnberg; fertig)
- Scheduler
  (Markus Felsner, FOM München; in Arbeit)
- … (hat noch jemand Lust?)

Ressourcen

Webseiten für OS-Devel-Einsteiger

- OS Development Wiki: http://wiki.osdev.org
- Bran's Tutorial: http://www.osdever.net/tutorials/view/brans-kernel-development-tutorial
- JamesM's Tutorial: http://www.jamesmolloy.co.uk/tutorial_html/
- BrokenThorn Tutorial: http://www.brokenthorn.com/Resources/OSDevIndex.html

Sonstige Literatur

- Minix-Buch: Tanenbaum/Woodhull, Operating Systems – Design and Implementation, 3rd ed., 2006

ULIX

- Webseite: http://www.ulixos.org/