Betriebssystemtechnik

Übung 1 - Privilegienisolation und Systemaufrufe

Daniel Danner
Christian Dietrich
Gabor Drescher

April 21, 2015

Ziel dieser Übung

Auftrennung in Benutzer- und Systemmodus

- Bisher ist OO/MPStuBS im Kernelmodus gelaufen
- Jeder Maschinenbefehl konnte jederzeit im Code auftreten
- Unterschied: Verschiedene Ausführungsrechte
  - Privilegierte Befehle: `hlt`, `cli`, `lgdt`, `lidt`, `iret`, etc.
  - Kontrollregister: `cr0` – `cr3`

Implementierung auf x86

- Es gibt 4 Ringe: 0 (System) – (3) Benutzer
- Ringe implizieren keinen Speicherschutz!
- Ring 3: Privilegierte Befehle und Register sind sicher

Was ist zu tun?

Schutzringe

- Konfiguration des Prozessors
- Wechsel in den Systemmodus

Was ist zu tun?

Schutzringe

- Konfiguration des Prozessors
- Wechsel in den Systemmodus

Was ist zu tun?

Systemaufrufe

- Erlauben von Traps aus Ring 3
- Übertragung von Argumenten ins System
- OOStuBS anpassen

Schutzringe

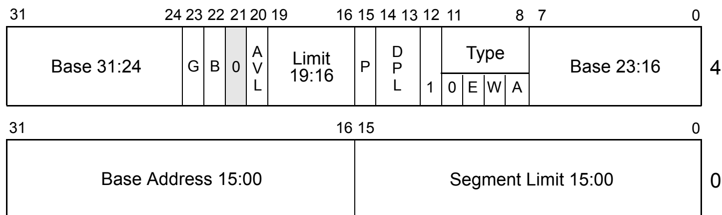
- Konfiguration des Prozessors
- Wechsel in den Systemmodus

Was ist zu tun?

Systemaufrufe

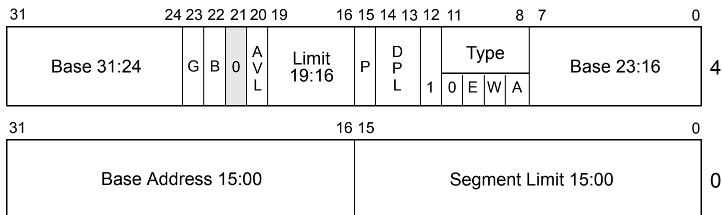
- Erlauben von Traps aus Ring 3
- Übertragung von Argumenten ins System
- OOStuBS anpassen

Code und Datensegment für den Benutzer



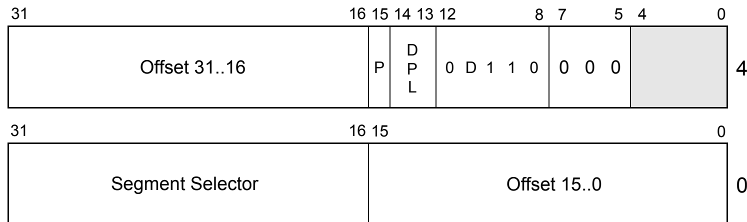
- Segmente sind tief in x86 (32-Bit) verwurzelt!
- Segmentregister (SS,CS,DS) referenzieren die GDT (Offset in die GDT)
  - Zugriffsrechte sind sowohl im Segmentregister als auch im GDT Eintrag
  - `mov ds, 0x8`
  - `mov [fs:eax], 42`

Code und Datensegment für den Benutzer



- Segmente sind tief in x86 (32-Bit) verwurzelt!
- Segmentregister (SS,CS,DS) referenzieren die GDT (Offset in die GDT)
  - Zugriffsrechte sind sowohl im Segmentregister als auch im GDT Eintrag
  - `mov ds, 0x8`
  - `mov [fs:eax], 42`
  - Dekodiere: `FF FF 00 00 | 00 9A CF 00`

Ring 3 -> Ring 0 (Systemcall)



- Man kann nicht jeden Interrupt in Software auslösen (`int 0x80`)!
  - Nicht jeder sollte den Timer ticken lassen können!
  - Im IDT Eintrag das Descriptor Protection Level (DPL) anpassen.
- Mit `int 0x..` können wir ein IRQ Gate anspringen
  - Segment Selector gibt neues Code Segment an
  - Was passiert aber nach der Unterbrechungsanforderung?

Ablauf eines Systemaufrufes

- Im Userspace

Ablauf eines Systemaufrufes

- Im Userspace
  - Argumente in Register und `int 0x..`
- Hardware

Ablauf eines Systemaufrufes

- Im Userspace
  - Argumente in Register und `int 0x..`
- Hardware
  - IP setzen (aus IDT Eintrag)
  - Ring wechseln (aus IDT Eintrag)
  - Stack auf Kernelstack wechseln
  - Zustand auf Stack sichern
- Im Kernel

Ablauf eines Systemaufrufes

- Im Userspace
  - Argumente in Register und `int 0x..`
- Hardware
  - IP setzen (aus IDT Eintrag)
  - Ring wechseln (aus IDT Eintrag)
  - Stack auf Kernelstack wechseln
  - Zustand auf Stack sichern
- Im Kernel
  - Register sichern
  - Systemaufruf ausführen
  - Zurückkehren (`iret`)

Was fehlt?

Ablauf eines Systemaufrufes

- Im Userspace
  - Argumente in Register und `int 0x..`
- Hardware
  - IP setzen (aus IDT Eintrag)
  - Ring wechseln (aus IDT Eintrag)
  - Stack auf Kernelstack wechseln
  - Zustand auf Stack sichern
- Im Kernel
  - Register sichern
  - Systemaufruf ausführen
  - Zurückkehren (`iret`)

Was fehlt?

- Wo kommt der neue Stackpointer her?
  - Task State Segment (TSS)
  - Weitere arkane Datenstruktur!

Task State Segment

<table>
<tr><td colspan="2">EAX</td><td>40</td></tr>
<tr><td colspan="2">EFLAGS</td><td>36</td></tr>
<tr><td colspan="2">EIP</td><td>32</td></tr>
<tr><td colspan="2">CR3 (PDBR)</td><td>28</td></tr>
<tr><td>Reserved</td><td>SS2</td><td>24</td></tr>
<tr><td colspan="2">ESP2</td><td>20</td></tr>
<tr><td>Reserved</td><td>SS1</td><td>16</td></tr>
<tr><td colspan="2">ESP1</td><td>12</td></tr>
<tr><td>Reserved</td><td>SS0</td><td>8</td></tr>
<tr><td colspan="2">ESP0</td><td>4</td></tr>
<tr><td>Reserved</td><td>Previous Task Link</td><td>0</td></tr>
</table>

- Intel unterstüzt Hardware Tasks
- Im GDT wird ein TSS-Deskriptor angelegt, der das TSS referenziert
- Wird mit `ltr <GDT-Offset>` geladen
- Wir haben ein TSS; jeder Task hat seinen eigenen Kernelstack!

Im Userland: System Call

Im Userland: System Call

```
1  int syscall_arg3(int syscall, int a0, int a1, int a2) {
2      int retval;
3      asm volatile("int 0x42;"
4                   // Output List
5                   : "=a"(retval)
6                   // Input List
7                   : "a" (syscall), // eax
8                     "b" (a1),      // ebx
9                     "c" (a2),      // ecx
10                    "d" (a3)       // edx
11                  // Clobber List
12                  : "memory");
13     return retval;
14 }
```

Im Kernelspace: System Call

Im Kernelspace: System Call

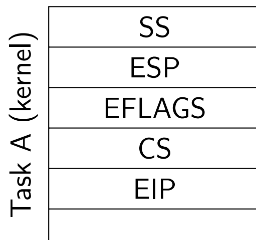
- Eintragen des Syscall-Handlers in `startup.asm`
  - Mit NASM Macro anstatt `guardian()` aufrufen
- Syscall-Handler (in ASM)
  - schiebt Register auf den Stack
  - ruft Syscall-Handler (in C) auf
  - `iret`
- Syscall-Handler (in C) is grosse if-else if-else Kaskade

Im Kernel: Anpassungen an OOStuBS?

Initialer Wechsel

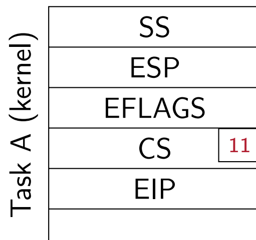
Initialer Wechsel

- Ähnlich zum toc_settle(). Einen Rückkehrstack faken



Initialer Wechsel

- Ähnlich zum toc_settle(). Einen Rückkehrstack faken



Exkurs: Linux 0.01 – GDT Aufbau

GDT

–
Kernel: Code
Kernel: Data
TMP
Task 0: TSS
Task 0: LDT
...

Kernel: Code
base=0, size=8M

Kernel: Data
base=0, size=8M

TSS Segment

LDT
...

Task0: LDT

Task 0: Code
Task 0: Data

Task 0: Code
base=ID · 64M, size=64M

Task 0: Data
base=ID · 64M, size=64M

FS Register

Exkurs: Linux 0.01 – kernel/sched.c

```
58  /*
59   *  'schedule()' is the scheduler function. This is GOOD CODE! There
60   * probably won't be any reason to change this, as it should work well
61   * in all circumstances (ie gives IO-bound processes good response etc).
62   * The one thing you might take a look at is the signal-handler code here.
63   *
64   *   NOTE!! Task 0 is the 'idle' task, which gets called when no other
65   * tasks can run. It can not be killed, and it cannot sleep. The 'state'
66   * information in task[0] is never used.
67   */
68  void schedule(void)
69  {
70          int i,next,c;
71          [...]
104         switch_to(next);
105 }
```

Exkurs: Linux 0.01 – include/linux/sched.h

```
162 /*
163  * switch_to(n) should switch tasks to task nr n, first
164  * checking that n isn't the current task, in which case it does nothing.
165  * This also clears the TS-flag if the task we switched to has used
166  * tha math co-processor latest.
167  */
168 #define switch_to(n) {\
169 struct {long a,b;} __tmp; \
170 __asm__("cmpl %%ecx,_current\n\t" \
171         "je 1f\n\t" \
172         "xchgl %%ecx,_current\n\t" \
173         "movw %%dx,%1\n\t" \
174         "ljmp %0\n\t" \
175         "cmpl %%ecx,%2\n\t" \
176         "jne 1f\n\t" \
177         "clts\n" \
178         "1:" \
179         ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
180         "m" (last_task_used_math),"d" _TSS(n),"c" ((long) task[n])); \
181 }
```