BS-Entwicklung mit Literate Programming

Foliensatz 8

Hans-Georg Eßer
TH Nürnberg

v1.0, 25.11.2013

System Calls

- Vorbereitung für Prozesse
- Im User Mode kein Zugriff auf Kernel-Funktionen
- Syscalls: Interface zu Kernel-Funktionen
- Prozesse können Syscalls über kontrolliertes Interface aufrufen
- Syscall-Handler können Berechtigung prüfen
- Beispiel (Linux): `write()`

Chunk: ⟨example for system calls in linux⟩ (1)

⟨example for system calls in linux⟩≡

```
_start:                         ; tell linker entry point
      mov edx,len               ; message length
      mov ecx,msg               ; message to write
      mov ebx,1                 ; file descriptor (stdout)
      mov eax,4                 ; system call number (sys_write)
      int 0x80                  ; software interrupt 0x80
      mov eax,1                 ; system call number (sys_exit)
      int 0x80                  ; software interrupt 0x80

section .data
msg   db 'Hello, world!',0xa    ; the string to be printed
len   equ $ - msg               ; length of the string
```

- mit `nasm -f elf prog.asm; ld prog.o -o prog` übersetzen
- ruft `write`-Syscall über klassisches Interface auf
- Syscall-Nummer (4) in EAX, Argumente in EBX bis EDX

Chunk: ⟨constants⟩ (1)

- ULIX: Syscalls auch über `int 0x80`
- brauchen Interrupt Handler für `0x80`
- größte Syscall-Nummer: 0x7FFF

⟨constants⟩≡

```
  #define MAX_SYSCALLS 0x8000          // max sysca↩
…ll number: 0x7fff
```

Chunk: ⟨global variables⟩ (1)

- Erzeuge Syscall-Handler-Tabelle
- jeder Eintrag speichert eine Adresse (`void *`)

⟨global variables⟩≡

```
  void *syscall_table[MAX_SYSCALLS];
```

Chunk: ⟨function prototypes⟩ (1)

- Für einzelne Syscalls separaten Handler schreiben
- z. B. `syscall_write (context *r)`
- dann deren Adresse eintragen

⟨function prototypes⟩≡

```
  void install_syscall_handler (int syscallno, voi↩
…d *syscall_handler);
```

Chunk: ⟨function implementations⟩ (1)

⟨*function implementations*⟩≡

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

Chunk: ⟨syscall entry example⟩ (1)

- für `write()` System Call:
- `__NR_write = 4`
- `void sys_write (context *r);`

⟨*syscall entry example*⟩≡

```
install_syscall_handler (__NR_write, sys_write);
```

Chunk: ⟨function implementations⟩ (2)

- generischer Syscall-Handler
- vergleichbar `irq_handler()` und `fault_handler()`
- sucht in Tabelle (`eax`: Syscall-Nummer)
- ruft Handler-Funktion auf

⟨*function implementations*⟩+≡

```
  void syscall_handler (context_t *r) {
    void (*handler) (context_t*);   // handler is ↩
…a function pointer
    int number = r->eax;
    handler = syscall_table[number];
    if (handler != 0) {
      handler (r);
    } else {
      printf ("Unknown syscall no. eax=0x%x; ebx=0↩
…x%x. eip=0x%x, esp=0x%x. "
              "Continuing.\n", r->eax, r->ebx, r->↩
…eip, r->esp);
    };
    return;
  }
```

Chunk: ⟨function prototypes⟩ (2)

- später: für jeden konkreten Syscall einen Prototyp definieren, z. B.
- `void syscall_write (context *r);`

⟨*function prototypes*⟩+≡
⟨*syscall prototypes*⟩

Chunk: ⟨function implementations⟩ (3)

- und dann implementieren:
- `void syscall_write (context *r) { int fd = r->ebx; ... }`

⟨*function implementations*⟩+≡
⟨*syscall functions*⟩

Interrupt-Handler in start.asm

- Aufruf von `int 0x80` bewirkt Software Interrupt
- Behandlung von Software Interrupts wie bei Hardware Interrupts
- also Interrupt-Handler für IRQ 0x80
- Code sieht wie bei anderen Interrupt-Handlern aus
- aber: `call syscall_handler` (statt `irq_handler`)

Chunk: ⟨start.asm⟩ (1)

⟨start.asm⟩≡

```
[section .text]
        extern syscall_handler
        global isr128

isr128: push byte 0       ; put 128 on the stack so it looks the same
        ; push byte 128   ; as it does after a hardware interrupt
        push byte -128    ; (getting rid of nasm error for signed byte)
        ⟨push registers onto the stack⟩
        call syscall_handler
        ⟨pop registers from the stack⟩
        add  esp, 8       ; undo the two "push byte" commands from the start
        iret
```

- Register auf Stack: General Purpose Registers (EAX, ECX, EDX, EBX, old ESP, EBP, ESI, EDI), DS, ES, FS, GS plus ESP (= Zeiger auf Kontext)

Syscall ausführen

- Zur Vereinfachung: generische Funktionen `syscall1()`, ..., `syscall4()`
- für Syscalls mit 0 bis 3 Argumenten (jeweils plus Syscall-Nummer)
- Prozedere:
  - Syscall-Nummer nach EAX
  - 1./2./3. Argument nach EBX/ECX/EDX
  - `int 0x80`
  - Rückgabewert aus EAX lesen

Chunk: ⟨standard functions for making system calls⟩ (1)

⟨standard functions for making system calls⟩≡

```
inline int syscall1 (int eax) {
  int result;
  asm ( "int $0x80" : "=a" (result) : "a" (eax) );
  return result ;
}

inline int syscall2 (int eax, int ebx) {
  int result;
  asm ( "int $0x80" : "=a" (result) : "a" (eax), "b" (ebx) );
  return result ;
}

inline int syscall3 (int eax, int ebx, int ecx) {
  int result;
  asm ( "int $0x80" : "=a" (result) : "a" (eax), "b" (ebx), "c" (ecx) );
  return result ;
}

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 ;
}
```

Chunk: ⟨example: write() prototype⟩ (1)

- Beispiel `write()`: Lib-Funktion nimmt 3 Argumente
- dafür können wir `syscall4()` verwenden

⟨example: write() prototype⟩≡

```
int write (int fd, const void *buf, int nbyte);
```

Chunk: ⟨example: write() implementation⟩ (1)

⟨example: write() implementation⟩≡

```
int write (int fd, const void *buf, int nbyte) {
  return syscall4 (__NR_write, fd, (int)buf, nbyte);
}
```

- So kann im Prinzip der größte Teil der User-Level-Bibliothek (`ulixlib.c`) aussehen
- Anwendungen binden dann `ulixlib.h` ein und werden gegen die Bibliothek gelinkt
- (keine dynamischen Binaries in ULIX)

Chunk: ⟨linux system calls⟩ (1)

- Für "Linux-Kompatibilität":
- Gleiche Syscall-Nummern verwenden
- Quelle: 32-Bit-Ubuntu 11.10, /usr/include/i386-linux-gnu/asm/unistd_32.h

⟨linux system calls⟩≡

```
#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
#define __NR_close               6
#define __NR_waitpid             7
#define __NR_creat               8
#define __NR_link                9
#define __NR_unlink             10
#define __NR_execve             11
#define __NR_chdir              12
#define __NR_time               13
#define __NR_mknod              14
#define __NR_chmod              15
#define __NR_lchown             16
#define __NR_break              17
#define __NR_lseek              19
#define __NR_getpid             20
#define __NR_mount              21
#define __NR_umount             22
#define __NR_alarm              27
#define __NR_utime              30
#define __NR_access             33
#define __NR_nice               34
#define __NR_ftime              35
#define __NR_sync               36
#define __NR_kill               37
#define __NR_rename             38
#define __NR_mkdir              39
#define __NR_rmdir              40
#define __NR_dup                41
#define __NR_pipe               42
#define __NR_times              43
#define __NR_brk                45
#define __NR_signal             48
#define __NR_lock               53
#define __NR_ulimit             58
#define __NR_umask              60
#define __NR_chroot             61
#define __NR_dup2               63
#define __NR_getppid            64
#define __NR_sigaction          67
#define __NR_sigsuspend         72
#define __NR_sigpending         73
#define __NR_symlink            83
#define __NR_readlink           85
#define __NR_readdir            89
#define __NR_mmap               90
#define __NR_munmap             91
#define __NR_truncate           92
#define __NR_ftruncate          93
#define __NR_fchmod             94
#define __NR_fchown             95
#define __NR_getpriority        96
#define __NR_setpriority        97
#define __NR_stat              106
```

```
#define __NR_lstat              107
#define __NR_fstat              108
#define __NR_wait4              114
#define __NR_sigreturn          119
#define __NR_uname              122
#define __NR_sigprocmask        126
#define __NR_fchdir             133
#define __NR_getdents           141
#define __NR_nanosleep          162
#define __NR_mremap             163
#define __NR_chown              182
#define __NR_getcwd             183
#define __NR_lchown32           198
#define __NR_getuid32           199
#define __NR_getgid32           200
#define __NR_geteuid32          201
#define __NR_getegid32          202
#define __NR_setreuid32         203
#define __NR_setregid32         204
#define __NR_getgroups32        205
#define __NR_setgroups32        206
#define __NR_fchown32           207
#define __NR_setresuid32        208
#define __NR_getresuid32        209
#define __NR_setresgid32        210
#define __NR_getresgid32        211
#define __NR_chown32            212
#define __NR_setuid32           213
#define __NR_setgid32           214
#define __NR_setfsuid32         215
#define __NR_setfsgid32         216
#define __NR_waitid             284
#define __NR_openat             295
#define __NR_tee                315
#define __NR_dup3               330
#define __NR_pipe2              331
```

Chunk: ⟨constants⟩ (2)

- Syscall-Nummern zu Konstanten hinzufügen

⟨*constants*⟩+≡
  ⟨*linux system calls*⟩
  ⟨*ulix system calls*⟩