



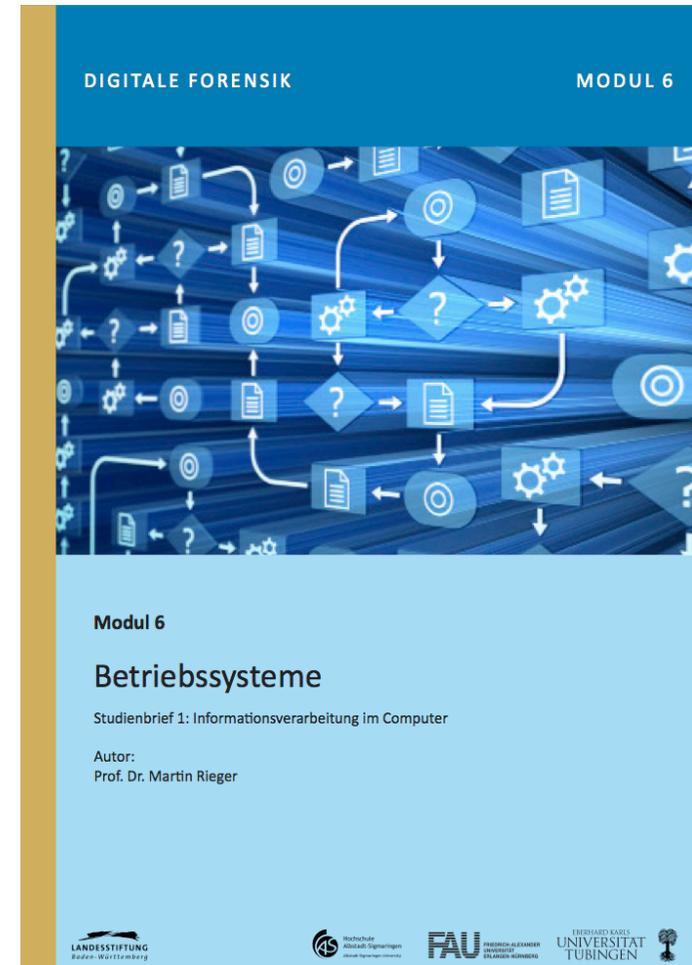
Hochschule
Albstadt-Sigmaringen
Albstadt-Sigmaringen University

Betriebssysteme

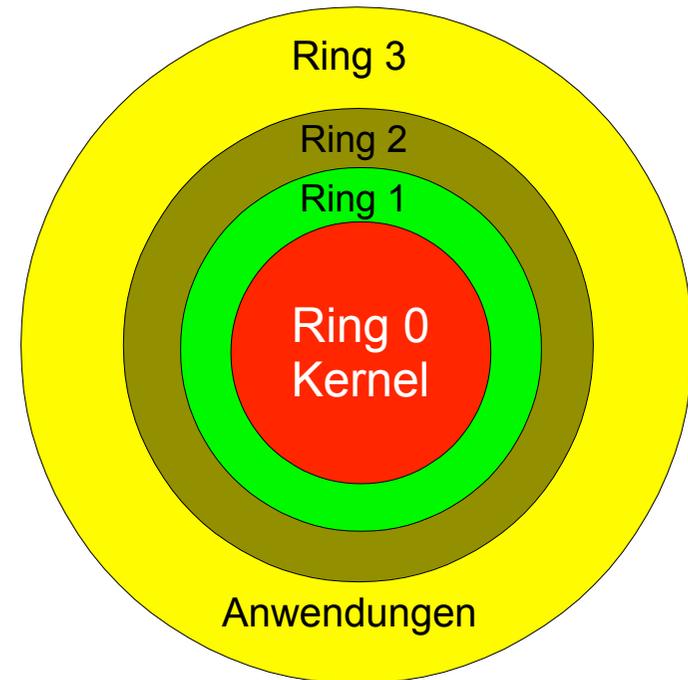
SS 2013

Hans-Georg Eßer
Dipl.-Math., Dipl.-Inform.

SB 4 (14.03.2013)
Systemprogrammierung Linux



- Prozessor kennt verschiedene Schutzstufen
 - Ring 0: Kernel Mode / Supervisor Mode
 - voller Zugriff auf alle Ressourcen
 - alle CPU-Instruktionen erlaubt
 - hier läuft das Betriebssystem
 - Ring 3: User Mode
 - eingeschränkter Zugriff auf Ressourcen
 - „privilegierte“ Instruktion verboten
 - hier laufen Anwendungen
 - Ringe 1, 2: nicht benutzt



- Anwendung kann nicht direkt auf Hardware zugreifen
 - keine Plattenzugriffe
 - keine I/O (USB, Firewire, seriell etc.)
 - kein Zugriff auf Bildschirmspeicher
 - Tastatur / Maus
 - physikalisches RAM (aber: virtueller Speicher)

- Anwendung muss Dienste des Betriebssystems nutzen
 - kontrollierter Übergang von Ring 3 → Ring 0 über ein „Gate“
 - realisiert über System Calls / Software Interrupts
 - kein direkter Sprung in BS-Funktion (`call os_print`), sondern
 - Verwendung von Software-Interrupt (`int`):

```
mov eax, OS_PRINT  
int 0x80
```
 - danach Rücksprung in Ring 3 (`iret`)

- BS-Funktionen prüfen beim Aufruf, ob Anwender berechtigt ist; können Ausführung verweigern
- während System Call läuft: veränderte Sicht auf Speicher (Zugriff auf Prozessspeicher und auf Kernel-Speicher)
- nach System Call: Rücksprung in User-Mode, Programm erhält Rückgabewert

Einfaches Beispiel (1)

- Ziel: Ausgabe von „A“ im Textmodus (80x25) in linker oberer Ecke
- technisch:
 - Bildschirmspeicher: 80 x 25 x 2 Bytes ab 0xB8000
 - erste zwei Bytes für Position links oben zuständig (ASCII-Code und Farbe)
 - Aufgabe: `char *addr=0xB8000; *addr='A';`
 - Problem: Anwendung nutzt virtuellen Speicher, Adresse 0xB8000 nicht erreichbar

- Lösungsansatz:

Betriebssystem hat Funktion `write_screen`:

```
int write_screen (short spalte, zeile, char c) {  
    int addr = 0xB8000 + 2 * spalte + 160 * zeile;  
    char *ptr = (char*) addr;  
    *ptr = c;  
    return 0;  
}
```

- Anwendung müsste

```
write_screen (0, 0, 'A');
```

aufrufen – wie „kommt sie da ran“?

- Betriebssystem installiert System-Call-Handler für verschiedene Dienste, z. B. `write_screen`:

```
#define SYSCALL_WRITE_SCREEN 101

int syscall_handler_0x80 (int eax, ...) {
    switch (eax) {
        case SYSCALL_WRITE_SCREEN:
            // call write_screen (syscall 101)
            // ebx: column, ecx: row, edx: char
            write_screen (ebx, ecx, edx);
            break;
        case SYSCALL_...:
            ...
    };
    asm (iret);
};
```

- Programm lädt passende Werte in Register (Linux):

```
asm (  
    mov eax, 101    // syscall no.  
    mov ebx, 0     // column  
    mov ecx, 0     // row  
    mov edx, 'A'   // char  
    int 0x80       // software int. 0x80  
);
```

- Alternative Implementierung über Stack (z. B. FreeBSD):

```
asm (  
    mov eax, 101    // syscall no.  
    push 'A'       // char  
    push 0         // row  
    push 0         // column  
    int 0x80       // software int. 0x80  
);
```

Echtes Beispiel: Text ausgeben

```
section .text
    global _start      ;fuer den Linker (ld)

_start:                ;fuer Linker (wo gehts los)

    mov     edx,len    ;Nachrichtenlaenge
    mov     ecx,msg    ;Adresse der Nachricht
    mov     ebx,1      ;file descriptor (1=stdout)
    mov     eax,4      ;Syscall-Nr. (sys_write)
    int     0x80       ;Syscall ausfuehren

    mov     eax,1      ;Syscall-Nr. (sys_exit)
    int     0x80       ;Syscall ausfuehren

section .data

msg     db     'Hallo Welt!',0xa    ;Text
len     equ   $ - msg              ;Laenge
```

Quelle des Listings: <http://asm.sourceforge.net/intro/hello.html> (übersetzt) - Syntax für Assembler nasm geeignet

Welche Syscalls gibt es?

In `/usr/include/asm/unistd_32.h`
oder `/usr/include/asm/unistd_64.h`
für 64 Bit:

```
# grep -c __NR unistd_32.h  
335
```

(335 System Calls)

```
#define __NR_restart_syscall 0  
#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_oldstat 18  
#define __NR_lseek 19  
#define __NR_getpid 20  
...
```

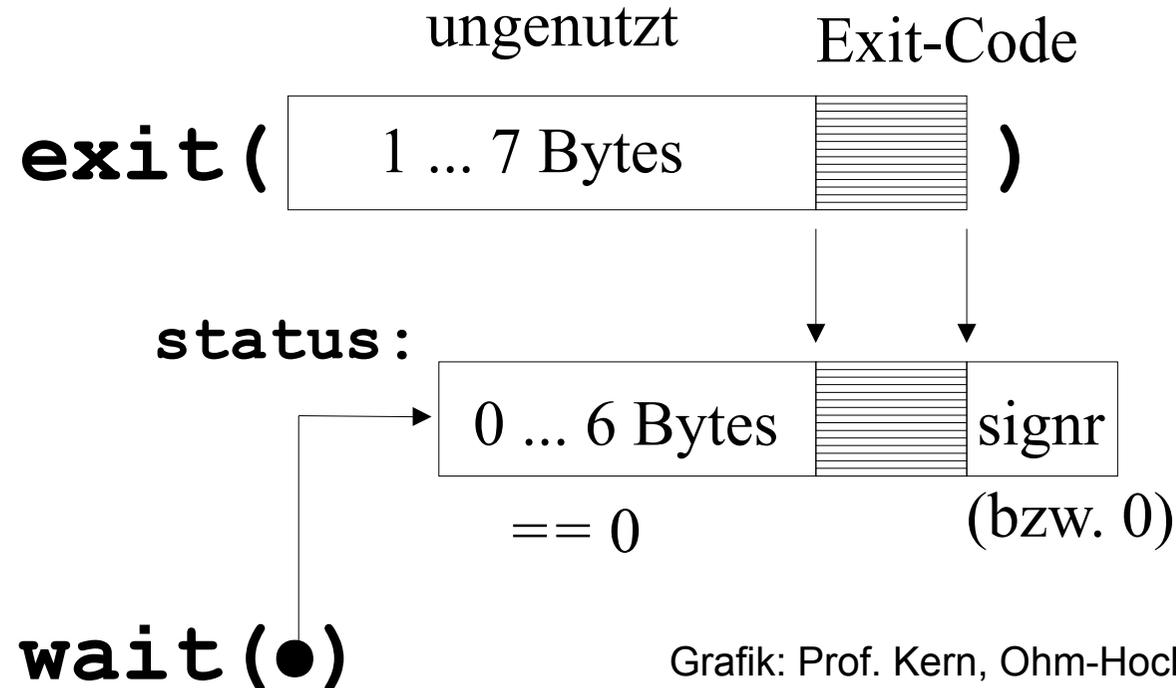


- `fork`: erzeugt (fast) identischen Sohn-Prozess
- `exec`: lädt anderes Programm in aktuellen Prozess
- `wait`: wartet auf Ende eines Sohn-Prozesses
- `open`: Datei öffnen (Spezialfall: `creat`)
- `read`: aus Datei lesen
- `write`: in Datei schreiben
- `close`: Datei schließen

- Für jeden Syscall (z. B. `fork`) gibt es gleichnamige Funktion (z. B. `fork()`).
- Die Funktion führt den eigentlichen Syscall aus
 - lädt Register
 - führt `int 0x80` aus
 - wertet Rückgabewert aus

Schon gesehen: fork(), exec(), wait()

Zu exit() und wait():



- auslesen des `exit`-Werts: über `(status >> 8)` oder mit Makro `WEXITSTATUS`
- Signalnummer `!= 0`, falls erzwungener Abbruch durch Signal

Zusammenhang exit / wait

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

void forktest (int retval) {
    int pid, status;
    pid = fork ();
    if (pid==0) {
        // child:
        if (retval >= 0) {
            // normal termination
            printf ("Child: exiting with exit code %d\n", retval);
            exit (retval);
        } else {
            // abnormal termination
            printf ("Child: exiting abnormally\n");
            abort ();
        }
    };
} else {
    // parent:
    printf ("Parent: Waiting for child to terminate\n");
    wait (&status);
    if (WIFEXITED(status)) {
        printf ("Parent: Child exited with exit code %d\n",
                WEXITSTATUS(status));
    } else {
        printf ("Parent: Child exited abnormally.\n");
    }
};
return;
}

int main () {
    forktest (0);
    forktest (3);
    forktest (-1); // terminate abnormally
    return 0;
}
```

```
esser@ubu64:forkwait$ gcc forkwait.c
esser@ubu64:forkwait$ ./a.out
Parent: Waiting for child to terminate
Child: exiting with exit code 0
Parent: Child exited with exit code 0
Parent: Waiting for child to terminate
Child: exiting with exit code 3
Parent: Child exited with exit code 3
Parent: Waiting for child to terminate
Child: exiting abnormally
Parent: Child exited abnormally.
```

- `wait`: wartet auf beliebigen Prozess
- `waitpid`: wartet auf Prozess mit angegebener PID (oder auf Prozess, der zur Prozessgruppe PGID gehört, wenn Argument `-PGID` übergeben wird)
- `wait = waitpid (-1)`
- `wait` und `waitpid` erhalten auch Status des (beendeten) Sohnprozesses (\rightarrow E-18)

- Neue Datei erzeugen: `creat ()`

```
#include <sys/types.h>
#include <sys/stat.h>
...
```

```
char filename[]="datei.txt";
int fd = creat ((char*)&filename, S_IRUSR | S_IWUSR);
```

- Datei öffnen: `open ()`

```
#include <fcntl.h>
...
```

```
char filename[]="datei.txt";
int fd = open ((char*)&filename, O_RDONLY);
```

- Optionen beim Öffnen (`O_RDONLY` etc.) stehen in `/usr/include/asm-generic/fcntl.h`
 - `O_RDONLY`: nur lesen
 - `O_WRONLY`: nur schreiben
 - `O_RDWR`: lesen/schreiben, ...
- Attribute beim Erzeugen (`S_IRUSR` etc.) stehen in `/usr/include/sys/stat.h`
 - `S_IRUSR`: Leserechte für Besitzer
 - `S_IWGRP`: Schreibrechte für Gruppe, ...

- Zugriffsrechte (mode) bei
 - `creat(f, mode)` bzw.
 - `open(f, O_CREAT, mode)`werden durch `umask` beeinflusst:
- tatsächliche Rechte: `mode & ~umask`
- `umask` setzen mit `umask(maske)`
- Beispiel: →



```
// umask-test.c
#include <stdlib.h>
int main () {
    creat ("test1.rwx", 0777); // max. Rechte: rwxrwxrwx
    creat ("test1.-wx", 0333); // Rechte: -wx-wx-wx
    creat ("test1.r-x", 0555); // Rechte: r-xr-xr-x
    umask (0);
    creat ("test2.rwx", 0777); // max. Rechte: rwxrwxrwx
    creat ("test2.-wx", 0333); // Rechte: -wx-wx-wx
    creat ("test2.r-x", 0555); // Rechte: r-xr-xr-x
    system ("stat -c '%a %A %n' test?.???");
};
```

```
root@ubu64:~# umask
0022
root@ubu64:~# ./umask-test
755 -rwxr-xr-x test1.rwx
555 -r-xr-xr-x test1.r-x
311 --wx--x--x test1.-wx
777 -rwxrwxrwx test2.rwx
555 -r-xr-xr-x test2.r-x
333 --wx-wx-wx test2.-wx
```

- Lesen: `read ()`

```
read (fd, &buffer, count);
```

(liest `count` Bytes aus der Datei und schreibt sie in den Puffer; Rückgabewert: Anzahl der gelesenen Bytes)

- Schreiben: `write ()`

```
write (fd, &buffer, count);
```

(schreibt `count` Bytes aus dem Puffer in die Datei; Rückgabewert: Anzahl der geschriebenen Bytes)

- Bisher: Dateien sequenziell lesen oder schreiben
- `lseek ()` erlaubt Positionierung des Schreib-/Lese-Zeigers
- drei Varianten:
 - `lseek (fd, offset, SEEK_SET)`: absolut
 - `lseek (fd, offset, SEEK_CUR)`: relativ
 - `lseek (fd, offset, SEEK_END)`:
Dateiende + `offset` (meist: `offset = 0`)
- Rückgabewert: neuer Offset



- Datei schließen: `close ()`

```
close (fd);
```

- System Calls können fehlschlagen
 - immer den Rückgabewert des Syscalls überprüfen
 - Manpages erklären, woran man Fehler erkennt
- Beispiele:
 - `fork ()`: Prozess kann nicht erzeugt werden, Rückgabewert -1
 - `open ()`: Datei kann nicht geöffnet werden, Rückgabewert -1, genauere Fehlerbeschreibung in Variable `errno`



Fehlerbehandlung (2)

- Variable `errno`: `#include <errno.h>`
- Standard-Fehler-Codes in `/usr/include/asm-generic/errno-base.h`
- Für Anzeige des Fehlers gibt es Funktion `perror ()`.
- Beispiel: Datei öffnen



Fehlerbehandlung (3)

```
/* open1.c, Hans-Georg Esser, Systemprogrammierung */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main () {
    int fd = open ("/etc/dontexist", O_RDONLY);

    if (fd == -1) {
        // Fehler
        int err = errno;
        printf ("Fehler bei open(), errno = %d, ", err);

        switch (errno) {
            case ENOENT: printf ("No such file or directory\n"); break;
            case EACCES: printf ("Permission denied\n"); break;
            default: printf ("\n");
        };

        exit (-1); // Programm mit Fehlercode verlassen
    }

    close (fd);
};
```

mit `perror()`:

```
/* open2.c, Hans-Georg Esser, Systemprogrammierung */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main () {

    int fd = open ("/etc/dontexist", O_RDONLY);

    if (fd == -1) {
        // Fehler
        perror ("open2");
        exit (-1); // Programm mit Fehlercode verlassen
    }

    close (fd);
};
```

- Ausgabe open1.c:

```
$ ./open1  
Fehler bei open(), errno = 2, No such file or directory
```

- Ausgabe open2.c:

```
$ ./open2  
open2: No such file or directory
```



Synchronisation

- gemeinsamen Semaphor einrichten:
- Prozess A:
 - `semid = ftok ("/tmp/abcdef", 1234);`
 - `semget (semid, anzahl, IPC_CREAT)`
 - `semctl` (initialisieren)
- Prozess B:
 - `semid = ftok (/tmp/abcdef, 1234);`
 - `semget (semid, anzahl, 0)`
 - dann verwenden

- Semaphor erzeugen:

```
main () {  
    key_t k;  
    int semid, i;  
  
    union semun {  
        int val; struct semi_ds *buf;  
        unsigned short *array; struct seminfo *__buf;  
    } sunion;  
  
    creat ("/tmp/mykey", 0);           // Datei erzeugen  
    k = ftok ("/tmp/mykey", 1234);    // Daraus Key generieren  
    printf ("k = %ld \n", (long)k);  
  
    semid = semget (k, 1, IPC_CREAT); // 1 Semaphor; erzeugen  
    printf ("semid = %d \n", semid);  
  
    sunion.val = 1;                   // Sem. 1 auf 1 init.  
    semctl (semid, 0, SETVAL, sunion);  
  
    printf ("Semaphor auf 1 initialisiert.\n");  
}
```

- Operation `down()` bzw. `wait()` - ohne Schlafen:

```
main () {
    // Variablen wie gehabt ...

    k = ftok ("/tmp/mykey", 1234);           // Key aus Datei generieren
    printf ("k = %ld \n", (long)k);

    semid = semget (k, 1, 0);               // 1 Semaphor; VERWENDEN
    printf ("semid = %d \n", semid);

    struct sembuf sbuf;
    sbuf.sem_op = -1;                       // -1: wait() bzw down()
    sbuf.sem_num = 0;
    sbuf.sem_flg = IPC_NOWAIT;

    printf ("Versuche 'down'... ");
    if (semop(semid, &sbuf, 1) != 0)
        printf ("Anfordern hat nicht geklappt\n");
    else
        printf ("Anfordern erfolgreich\n");
}
```

- Operation `down()` bzw. `wait()` - mit Schlafen:

```
main () {
    // Variablen wie gehabt ...

    k = ftok ("/tmp/mykey", 1234);           // Key aus Datei generieren
    printf ("k = %ld \n", (long)k);

    semid = semget (k, 1, 0);               // 1 Semaphor; VERWENDEN
    printf ("semid = %d \n", semid);

    struct sembuf sbuf;
    sbuf.sem_op = -1;                       // -1: wait() bzw down()
    sbuf.sem_num = 0;
    // sbuf.sem_flg = IPC_NOWAIT;         // keine Flags

    printf ("Versuche 'down'... ");
    semop(semid, &sbuf, 1);                 // schläft eventuell
    printf ("Anfordern erfolgreich\n");
}
```

- Operation `up()` bzw. `signal()`:

```
main () {  
    // Variablen wie gehabt ...  
  
    k = ftok ("/tmp/mykey", 1234);           // Key aus Datei generieren  
    printf ("k = %ld \n", (long)k);  
  
    semid = semget (k, 1, 0);                // 1 Semaphor; VERWENDEN  
    printf ("semid = %d \n", semid);  
  
    struct sembuf sbuf;  
    sbuf.sem_op = 1;                        // +1: signal() bzw. up()  
    sbuf.sem_num = 0;  
    semop(semid, &sbuf, 1);  
  
    printf ("Fuehre 'up' durch.\n");  
}
```

- Signal-Handler mit `sigaction` (neu) oder `signal` (alt) eintragen:

```
struct sigaction sa1;  
memset (&sa1, 0, sizeof(sa1));  
sa1.sa_handler = sigChildHandler;  
sa1.sa_flags   = SA_NOCLDSTOP;  
sigaction(SIGCHLD, &sa1, NULL);
```

```
signal(SIGCHLD, sigChildHandler);
```

- Alternativen zur Angabe der Handler-Funktion
 - `SIG_DFL`: Standard-Verhalten (Abbruch)
 - `SIG_IGN`: Signal ignorieren