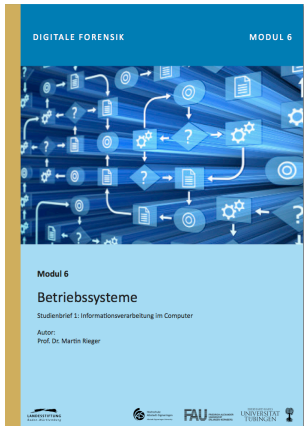
Betriebssysteme

SS 2012

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

SB 4 (13.05.2012)
Systemprogrammierung Linux



Grundlagen System Calls (1)

- 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

Grundlagen System Calls (2)

- 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)

Grundlagen System Calls (3)

- 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`)

Grundlagen System Calls (4)

- 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

Einfaches Beispiel (2)

- 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“?

Einfaches Beispiel (3)

- 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);
};
```

Einfaches Beispiel (4)

- 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
...
```


Wichtige Syscalls

- `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

Syscalls und Funktionen

- 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

Prozesse

Hierarchie

- Prozesse erzeugen einander
- Erzeuger heißt Vaterprozess (parent process), der andere Kindprozess (child process)
- Kinder sind selbständig (also: eigener Adressraum, etc.)
- Nach Prozess-Ende: Rückgabewert an Vaterprozess

Prozesse erzeugen: fork

- Neuer Prozess: `fork ()`

```
main() {
  int pid = fork();   /* Sohnprozess erzeugen */
  if (pid == 0) {
    printf("Ich bin der Sohn, meine PID ist %d.\n",
           getpid() );
  }
  else {
    printf("Ich bin der Vater, mein Sohn hat die
           PID %d.\n", pid);
  }
}
```

- erzeugt neuen Prozess
- Rückgabewert im Vater: PID des Sohnes
- Rückgabewert im Sohn: 0

Programm starten: fork / exec

- Anderes Programm starten: `fork + exec`

```
main() {
  int pid=fork();   /* Sohnprozess erzeugen */
  if (pid == 0) {
      /* Sohn startet externes Programm */
      execl( "/usr/bin/gedit", "/etc/fstab", (char *) 0 );
    }
  else {
      printf("Es sollte jetzt ein Editor starten...\n");
    }
}
```

- Andere Betriebssysteme oft nur: „spawn“

```
main() {
  WinExec("notepad.exe", SW_NORMAL);   /* Sohn erzeugen */
}
```

Auf Child-Prozess warten

Warten auf Sohn-Prozess: **`wait ()`**

```
#include <unistd.h>          /* sleep()                          */

main() {
  int pid=fork();            /* Sohnprozess erzeugen             */
  if (pid == 0) {
    sleep(2);                /* 2 sek. schlafen legen            */
    printf("Ich bin der Sohn, meine PID ist  %d\n", getpid() );
  } else {
    printf("Ich bin der Vater, mein Sohn hat die PID  %d\n", pid);
    wait();                  /* auf Sohn warten                  */
  }
}
```


Zusammenhang exit / wait

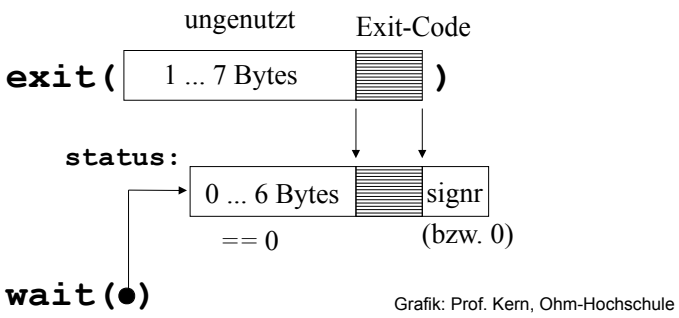
```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

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.
```


Zusammenhang exit / wait



Grafik: Prof. Kern, Ohm-Hochschule

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

Fehler bei fork / exec?

Abfrage, ob Programmstart über `fork()`, `exec()` erfolgreich war:

```
#include <errno.h>
main() {
  int pid = fork();
  int errno2;
  if (pid==0) {
    execl("/bin/xls",0);
    errno2=errno;
    perror ();
    printf("Fehlercode errno = %d\n",
           errno2);
  } else { wait(); }
}
```

```
> gcc -o fork-exec-fail fork-exec-fail.c
> ./fork-exec-fail
/bin/xls: No such file or directory
Fehlercode errno = 2
```

- `perror()`: Fehlermeldung in lesbarem Format
- `errno`: Globale Fehlervariable

`exec`-Varianten

- `execl`: absoluter Programmpfad, Argumente separat, mit Nullzeiger abgeschlossen
- `execlp`: Programmname, Argumente separat, mit Nullzeiger abgeschlossen
- `execle`: wie `execl`, zusätzlich Environment nach Argumentliste
- `execv`: wie `execl`, aber Argumente als Array
- `execvp`: wie `execlp`, aber Argumente als Array

`wait`-Varianten

- `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 (→ E-18)

Dateizugriffe (1)

- 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);
```


Dateizugriffe (2)

- 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, ...

Dateizugriffe (3)

- 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: →

Dateizugriffe (4)

```
// 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
```


Dateizugriffe (5)

- 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)

Dateizugriffe (6)

- 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

Dateizugriffe (7)

- Datei schließen: `close()`

```
close (fd);
```


Fehlerbehandlung (1)

- 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);
};
```

Fehlerbehandlung (4)

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);
};
```


Fehlerbehandlung (5)

- 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

Semaphore (1)

Ein **Semaphor** ist eine Integer- (Zähler-) Variable, die man wie folgt verwendet:

- Semaphor hat festgelegten Anfangswert N („Anzahl der verfügbaren Ressourcen“).
- Beim Anfordern eines Semaphors (P- oder **Wait**-Operation): P = (niederl.) probeer
  - Semaphor-Wert um 1 erniedrigen, falls er >0 ist,
  - Thread blockieren und in eine Warteschlange einreihen, wenn der Semaphor-Wert 0 ist.

Semaphore (2)

- Bei Freigabe eines Semaphors (V- oder **Signal**-Operation): V = (niederl.) vrijgeven
  - einen Thread aus der Warteschlange wecken, falls diese nicht leer ist,
  - Semaphor-Wert um 1 erhöhen (wenn es keinen auf den Semaphor wartenden Thread gibt)
- Code sieht dann immer so aus:

```
wait (&sem);
/* Code, der die Ressource nutzt */
signal (&sem);
```

- in vielen Büchern: `P(&sem), V(&sem)`

Mutexe (1)

- **Mutex:** boolesche Variable (true/false), die den Zugriff auf gemeinsam genutzte Daten synchronisiert
  - true: Zugang erlaubt
  - false: Zugang verboten
- **blockierend:** Ein Thread, der sich Zugang verschaffen will, während ein anderer Thread Zugang hat, blockiert → Warteschlange
- Bei Freigabe:
  - Warteschlange enthält Threads → einen wecken
  - Warteschlange leer: Mutex auf true setzen

Mutexe (2)

- **Mutex** (**mut**ual **ex**clusion) **= binärer Semaphor**, also ein Semaphor, der nur die Werte 0 / 1 annehmen kann.
- Neue Interpretation: wait → lock
  signal → unlock
- Mutexe für exklusiven Zugriff (kritische Bereiche)

Blockieren?

- Betriebssysteme können Mutexe und Semaphoren **blockierend** oder **nicht-blockierend** implementieren
- blockierend:
  wenn der Versuch, den Zähler zu erniedrigen, scheitert
  → warten
- nicht blockierend:
  wenn der Versuch scheitert
  → vielleicht etwas anderes tun

Atomare Operationen

- Bei Mutexen / Semaphoren müssen die beiden Operationen wait() und signal() **atomar** implementiert sein:

  Während der Ausführung von wait() / signal() darf kein anderer Prozess an die Reihe kommen

Linux-Semaphore

- 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

Beispiele zu Semaphoren

- Semaphor erzeugen:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>

main () {
  key_t k;
  int semid, i;

  union semun {
    int val; struct semi_ds *buf;
    unsigned short *array; struct seminof *__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");
}
```


Beispiele zu Semaphoren

- 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");
}
```

Beispiele zu Semaphoren

- 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");
}
```


Beispiele zu Semaphoren

- 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");
}
```


Signale

- 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