

```

Sep 19 14:20:18 amd64 sshd[20494]: Accepted rsa for esser from ::ffff:87.234.201.207 port 61557
Sep 19 14:27:41 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 01:00:01 amd64 /usr/sbin/cron[29278]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 20 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 02:00:01 amd64 /usr/sbin/cron[17053]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 20 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 12:46:44 amd64 sshd[5516]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62004
Sep 20 12:48:41 amd64 sshd[6609]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62105
Sep 20 12:54:44 amd64 sshd[6094]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62514
Sep 20 15:27:35 amd64 sshd[9077]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64242
Sep 20 15:27:35 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:37:11 amd64 sshd[1071]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63375
Sep 20 16:37:11 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 01:00:01 amd64 /usr/sbin/cron[17055]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 21 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 02:00:01 amd64 /usr/sbin/cron[17878]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 21 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 17:43:26 amd64 sshd[1088]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63397
Sep 21 17:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 17:53:29 amd64 sshd[9191]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64931
Sep 21 18:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 19:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 01:00:01 amd64 /usr/sbin/cron[4674]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 22 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 02:00:01 amd64 /usr/sbin/cron[5495]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 22 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 20:23:21 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 01:00:01 amd64 /usr/sbin/cron[24739]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 23 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 02:00:01 amd64 /usr/sbin/cron[25535]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 23 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 18:04:05 amd64 sshd[6341]: Accepted publickey for esser from ::ffff:192.168.1.5 port 59771 ssh2
Sep 23 18:04:05 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 18:04:34 amd64 sshd[6606]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62093
Sep 24 01:00:01 amd64 /usr/sbin/cron[12421]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 24 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 02:00:01 amd64 /usr/sbin/cron[13253]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 24 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 11:15:48 amd64 sshd[2098]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64456
Sep 24 11:15:48 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 13:49:08 amd64 sshd[23197]: Accepted rsa for esser from ::ffff:87.234.201.207 port 61330
Sep 24 13:49:08 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 15:42:07 amd64 kernel: snd_seq_midl event: unsupported module, tainting kernel.
Sep 24 15:42:07 amd64 kernel: snd_seq_oss: unsupported module, tainting kernel.
Sep 24 20:25:31 amd64 sshd[29399]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62566
Sep 24 20:25:31 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 01:00:02 amd64 /usr/sbin/cron[662]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 25 01:00:02 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 02:00:01 amd64 /usr/sbin/cron[1484]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 25 02:00:02 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 10:59:25 amd64 sshd[889]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64183
Sep 25 10:59:25 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 10:59:47 amd64 sshd[8921]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64253
Sep 25 11:30:02 amd64 sshd[9372]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62029
Sep 25 11:59:25 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 14:05:37 amd64 sshd[11554]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62822
Sep 25 14:05:37 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 14:06:10 amd64 sshd[11861]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62951
Sep 25 14:07:17 amd64 sshd[11608]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63392
Sep 25 14:08:23 amd64 sshd[11630]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63709
Sep 25 15:25:33 amd64 sshd[12930]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62778

```

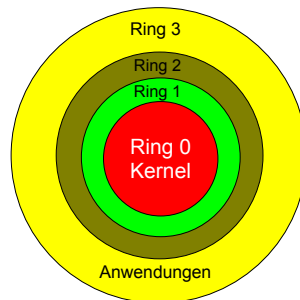
3. System Calls

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 (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 (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 (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 (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 (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  
);
```

Exkurs Intel x86 Assembler (2)

- Befehlsausführung: linear
- Sprungbefehle
- Schleifen sehen in Assembler immer so aus (Pseudosyntax):

```
<< Variablen für Schleife initialisieren >>  
start:  
    << Schleifenrumpf >>  
    << Test Abbruchbedingung >>  
    if (! Test) jump to start;  
  
// hinter der Schleife
```

Exkurs Intel x86 Assembler (1)

- Nur das wichtigste zu Assembler ...
- C-Compiler übersetzt C-Programme in Assemblersprache, Assembler übersetzt diese in Maschinensprache (ein Binary)
- 32-bittige CPU: Adressen sind 32 Bit breit, Register auch: EAX, EBX, ECX, ...; C-Typ int genau passend
- Spezialregister: EIP, EFLAGS, ... siehe http://de.wikipedia.org/wiki/Intel_80386#Register

Exkurs Intel x86 Assembler (3)

- Assembler-Code angucken:
gcc -S -masm=intel
- Beispiel:

```
int main () {  
    register int i,j,k;  
    i = 0x1234;  
    j = 0x5678;  
    if (i<j) {  
        k = j - i;  
    } else {  
        k = i - j;  
    };  
    return k;  
}
```



```
main:  
    push    ebp  
    mov     ebp, esp  
    push   esi  
    push   ebx  
    mov     esi, 0x1234  
    mov     ebx, 0x5678  
    cmp     esi, ebx  
    jge    .L2  
    sub     ebx, esi  
    jmp    .L3  
.L2:  
    mov     eax, esi  
    sub     eax, ebx  
    mov     ebx, eax  
.L3:  
    mov     eax, ebx  
    pop     ebx  
    pop     esi  
    pop     ebp  
    ret
```

Exkurs Intel x86 Assembler (4)

```

Ausgabe von gcc -S ...
Disassemblieren der erzeugten Object-Datei (a.out)
mit Tool udc1i, http://udis86.sourceforge.net/

main:
    push    ebp                00000394 55          push ebp
    mov     ebp, esp          00000395 89e5        mov ebp, esp
    push   esi                00000397 56          push esi
    push   ebx                00000398 53          push ebx
    mov     esi, 0x1234       00000399 be34120000    mov esi, 0x1234
    mov     ebx, 0x5678       0000039e bb78560000    mov ebx, 0x5678
    cmp     esi, ebx          000003a3 39de        cmp esi, ebx
    jge    .L2                000003a5 7d04        jge 0x3ab
    sub    ebx, esi           000003a7 29f3        sub ebx, esi
    jmp    .L3                000003a9 eb06        jmp 0x3b1
.L2:
    mov     eax, esi          000003ab 89f0        mov eax, esi
    sub    eax, ebx           000003ad 29d8        sub eax, ebx
    mov     ebx, eax          000003af 89c3        mov ebx, eax
.L3:
    mov     eax, ebx          000003b1 89d8        mov eax, ebx
    pop    ebx                000003b3 5b          pop ebx
    pop    esi                000003b4 5e          pop esi
    pop    ebp                000003b5 5d          pop ebp
    ret                       000003b6 c3          ret
  
```

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
...
  
```

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

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. `exec`) gibt es gleichnamige Funktion (z. B. `exec()`).
- Die Funktion führt den eigentlichen Syscall aus
 - setzt Register
 - führt `int 0x80` aus
 - wertet Rückgabewert aus

Prozesse erzeugen (1/7)

- 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

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 (2/7)

- 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 */
}
```

Prozesse erzeugen (3/7)

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 */
    }
}
```

Prozesse erzeugen (5/7)

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
- mehr zu `errno/perror`: gleich...

Prozesse erzeugen (4/7)

Wirklich mehrere Prozesse:

Nach `fork ()` zwei Prozesse in der Prozessliste

```
> pstree | grep simple
... -bash---simplefork---simplefork
```

```
> ps w | grep simple
25684 pts/16 S+      0:00 ./simplefork
25685 pts/16 S+      0:00 ./simplefork
```

Prozesse erzeugen (6/7)

Abbruch aller Kind-Prozesse

Zwei Szenarien:

1. Shell wird mit `exit` verlassen
→ Kind-Prozesse laufen weiter.
2. Shell wird gewaltsam geschlossen
(`kill`, Fenster schließen etc.)
→ Kind-Prozesse werden auch beendet.

Prozesse erzeugen (7/7)

```
[ In 2. Fenster ] > nedit &

> pstree | grep nedit
|   |_-xterm---bash---nedit
> ps auxw | grep nedit
esser  24676  1.0  0.8  8248  4336 pts/4    S   15:13   0:00 nedit
> cat /proc/24676/status | grep PPid
PPid:   24659
> ps auxw|grep 24659
esser  24659  0.0  0.3  4424  1936 pts/4    Ss+ 15:12   0:00 bash

[ In 2. Fenster ] > exit

> cat /proc/24676/status | grep PPid
PPid:   1
```

Dokumentation

- **wait:** man wait
 - 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 geben auch Status des (beendeten) Sohnprozesses zurück (→ später)

Dokumentation

- **fork:** man 2 fork
- **exec:** man exec
 - execl: absoluter Programmpfad, Argumente separat, mit Nullzeiger abgeschlossen
 - execlp: Programmname, Argumente separat, mit Nullzeiger abgeschlossen
 - execl: wie execl, zusätzlich Environment nach Argumentliste
 - execv: wie execl, aber Argumente als Array
 - execvp: wie execlp, aber Argumente als Array

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 (4)

- Datei schließen: `close ()`
`close (fd);`

Dateizugriffe (3)

- 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)

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 (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 (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 (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
```