5. Fallbeispiel UNIX

Michael Schöttner

Betriebssysteme und Systemprogrammierung

5.1 Vorschau

- Prozesse
- Trennung User-Space / Kernel-Space
- Dateien: Strukturen und Rechte
- Signale und Pipes
- Komponenten eines UNIX-Systems

5.2 Prozesse

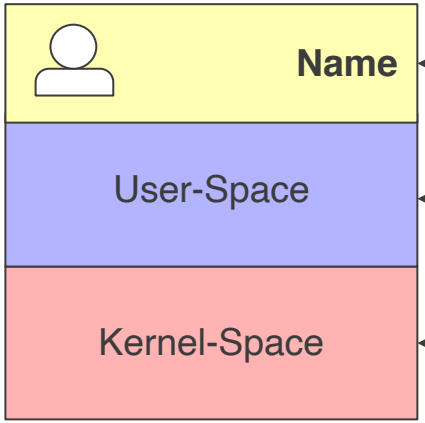
- UNIX-Prozess
  - ein Programm (in Ausführung)
  - ein Thread (Aktivitätsträger, früher: ein Thread, heute: viele Threads)
  - ein Adressraum → Schutz zwischen Prozessen
  - „Besitzer" der Betriebsmittel (Speicher, Dateien, ...) eines Programms
  - Läuft unter einem Benutzerkonto (Benutzer-ID, Gruppen-ID) → Berechtigungen

- Viele Prozesse pro Rechner
  - Vordergrund-Prozesse → direkte Interaktion
  - Hintergrund-Systemprozesse (engl. daemons)

5.3 Dateien

- Alle Informationen sind über die Datei-Schnittstelle zugänglich
  - Normale Dateien auf der Festplatte
  - Spezielle Dateien haben keine Datenblöcke auf der Festplatte
    - Geräte-Dateien in `/dev` bzw. `/sys`
    - Systeminformationen in `/proc`
- Datei-Funktionen:
  - Öffnen: `fd = open(name, flags, mode)`
  - Lesen: `bytes = read(fd, buf, size)`
  - Schreiben: `bytes = write(fd, buf, size)`
  - Schließen: `close(fd)`
  - Löschen: `unlink(pathname)`
  - Dateien werden im Betriebssystem über einen File-Descriptor `fd` identifiziert

5.4 Darstellung von Prozessen



Zuordnung zu einem Benutzerkonto,
sowie einem Programmnamen

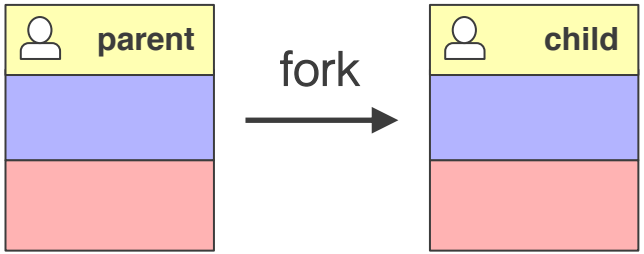
Code, Daten des Prozesses

Dem Prozess zugeordnete Kern-Strukturen.
Hierauf hat nur das Betriebssystem Zugriff.

Betriebssystemcode + Treiber + Daten

5.5 Erzeugung von Prozessen

- Prozesskopie des laufenden Prozesses mit **`fork`**:
  - Eltern-Kind Beziehung
  - mit neuer PID




```
pid_t p;
...
p = fork();
if ( p == (pid_t)0 ) {
   /* child */
   ...
} else if( p != (pid_t)-1 ) {
   /* parent */
   ...
} else {
   /* error */
   ...
}
```

Weitere Kernaufrufe

- `execve`: Aufrufer durch ein neues Programm ersetzen
  - `int execve(char *path, char *const argv[]);`
  - Aufruf kehrt nicht mehr zurück
  - rufender Prozess wird ersetzt
  - PID wird behalten

- `exit`: Prozess terminieren
  - `void exit(int status);`
  - Überträgt Status zum Elternprozess
    - `EXIT_SUCCESS` (bei POSIX hast dieses Makro den Wert 0)
    - `EXIT_FAILURE` (bei POSIX, 1)
    - Siehe auch, https://www.gnu.org/software/libc/manual/html_node/Exit-Status.html

Weitere Kernaufrufe (2)

- `waitpid`: warten auf Terminierung von Kind-Prozess
  - `pid_t waitpid(pid_t pid, int *status, int options);`
    - `pid`: Kind, auf dessen Terminierung gewartet wird (-1 = irgendein Kind)
    - `status`: exit-Code des Kindes
    - `options`: siehe man-Pages (0=blockierend warten)

Beispiel zu waitpid

```c
#include <unistd.h>      /* fork       */
#include <stdio.h>
#include <stdlib.h>      /* exit codes */
#include <sys/wait.h>    /* waitpid    */

int main () {
   pid_t pid;
   int status;

   pid = fork ();
   if ( pid == (pid_t)0 ) {
      printf ("Child: PID: %d, sleeping.\n", getpid());
      sleep(5);
      exit(EXIT_SUCCESS);
   }
   else {
      printf ("Parent: PID: %d, waiting for child ...\n", getpid());
      waitpid(pid, &status, 0);
      printf("Parent: done.\n");
   }
   return EXIT_SUCCESS;
}
```

Beispiel: Zombie

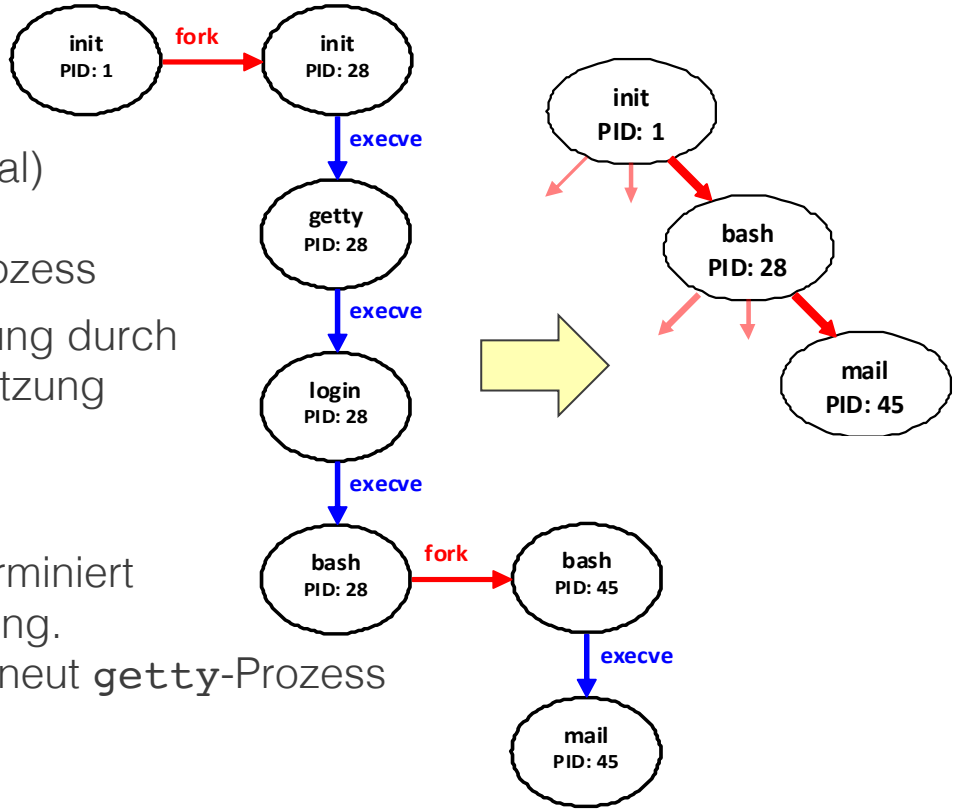
- Ein terminierter Prozess dessen Exit-Code nicht abgeholt wurde
- Dieser wird für den Eltern-Prozess auf-bewahrt
- Terminiert der Eltern-Prozess vor dem Kind, so wird das Kind dem `init`-Prozess zugeordnet → Re-Parenting

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main () {
   pid_t pid;
   int status;

   pid = fork ();
   if ( pid == (pid_t)0 ) {
      printf ("Child: PID: %d, done.\n", getpid());
      exit(EXIT_SUCCESS);
   }
   else {
      printf ("Parent: PID: %d, sleeping ...\n", getpid());
      sleep(60);
      printf("Parent: done.\n");
   }
   return EXIT_SUCCESS;
}
```

Zombie anzeigen mit:
`ps -aux`

Prozesshierarchie

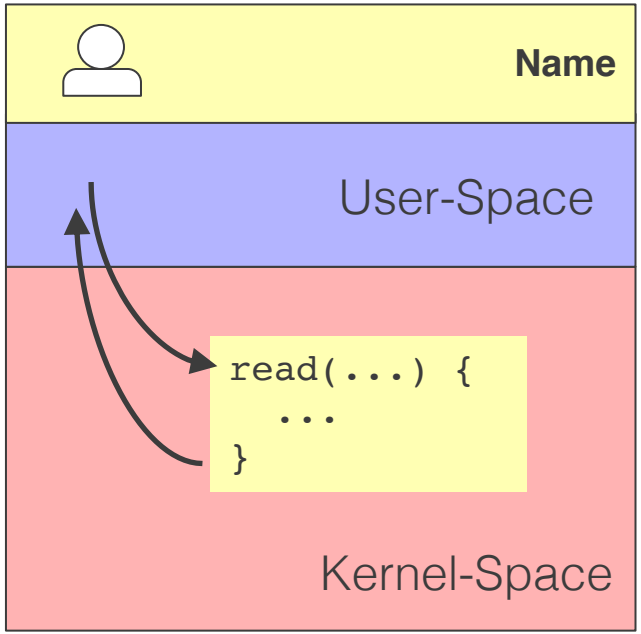
- Beispiel:
  - `init` erzeugt Terminals
    → `getty`-Prozess (~ get terminal)
    liest Benutzernamen ein und
    ersetzt sich durch `login`-Prozess
  - Nach erfolgreicher Authentisierung durch den `login`-Prozess erfolgt Ersetzung durch `bash`-Prozess (Shell)
  - ...
  - Bem.: Bei falschem Passwort terminiert `login`-Prozess mit Fehlermeldung.
    `init`-Prozess erzeugt damm erneut `getty`-Prozess
  - Ausgabe mit: `pstree PID`



5.6 Kernaufruf

- Beispiel: Lesen von einer Datei `status = read (fd, buf, anzahl);`

- Prozessor läuft je nachdem im User-Mode oder Kernel-Mode
  - Kernel-Mode:
    - Zugriff auf alle Daten (auch User-Space)
    - Alle Befehle des Prozessors erlaubt
    - Insbesondere für Schutz und Adressraumwechsel
  - User-Mode:
    - Kein Zugriff auf Kernel-Space
    - Keine privilegierten Befehle



Kernaufruf im Detail

User-Space

```
read(...) {

   /* Parameteraufbereitung */
   ...
   call = read;
   INT 0X80 // trap (alt)
```

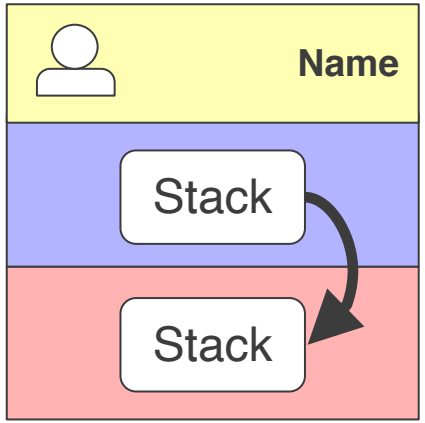
```
   /* weiter geht‘s */
}
```

Kernel-Space

```
/* TRAP-Entry */
switch (call) {
   case read:
      ...
   case write:
      ...
}
iret /* return from trap */
```

Stacks beim Kernaufruf

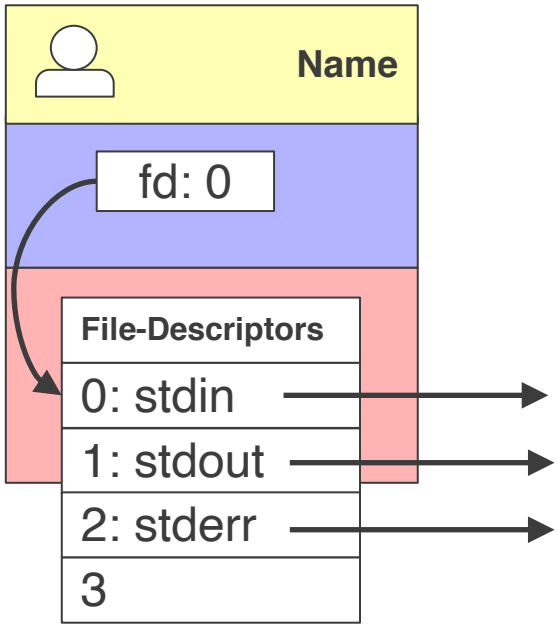
- Bei Systemaufrufen (engl. system calls) wird
  - auf den Kernel-Stack geschaltet
  - der Kern-Modus eingeschaltet
    → dadurch wird der Kern-Adressraum sichtbar
  - an eine feste Einsprungstelle (per Trap) gesprungen und von dort kontrolliert verzweigt



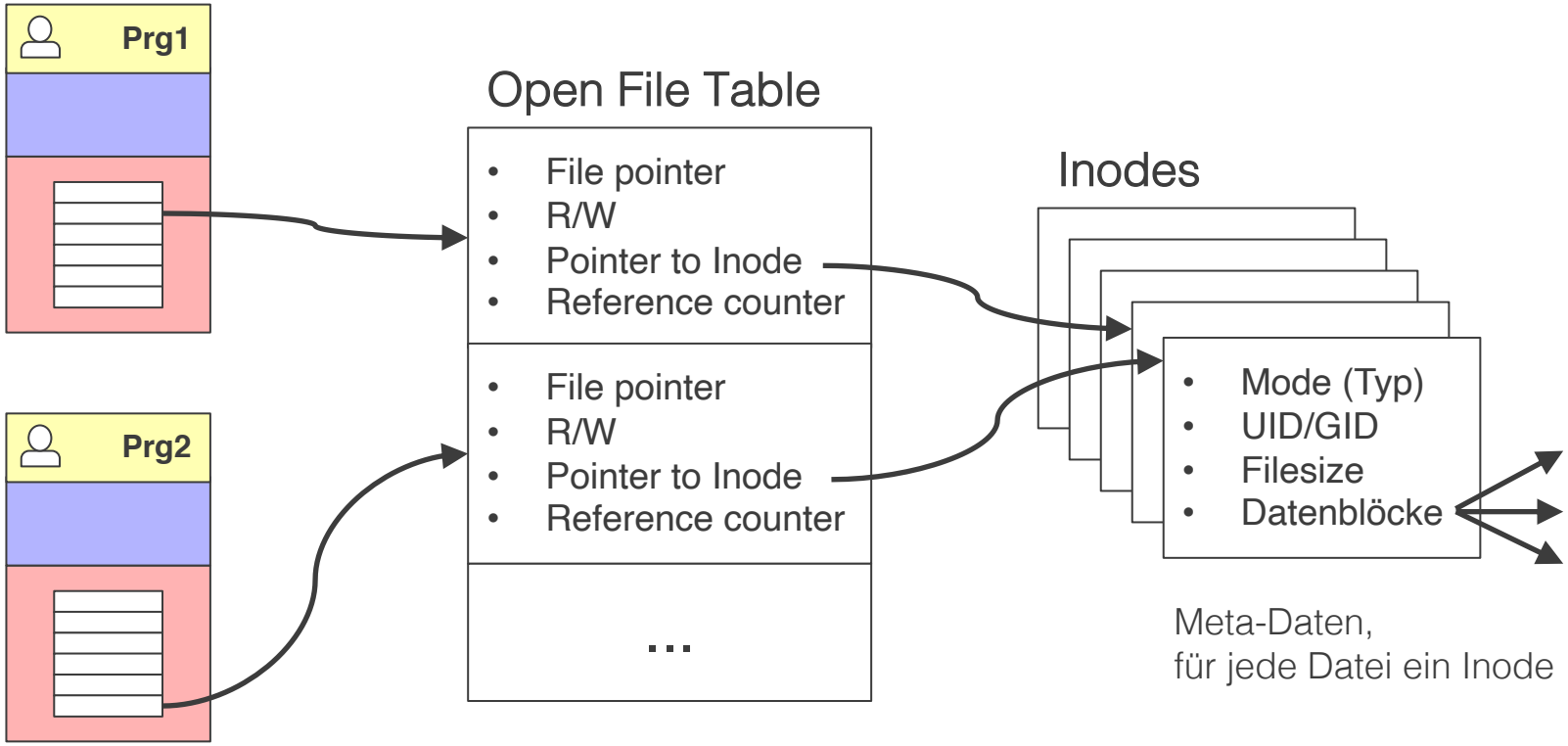
Datei-Deskriptor

- File descriptor (fd) ist im User-Space ein `unsigned int`
- fd wird als Index in die Datei-Tabelle des Prozesses verwendet
  → User-Space hat somit nur indirekten Zugriff

- Jeder Prozess hat mind. drei Datei-Deskriptoren
  - Standard-Input; stdin
  - Standard-Output: stdout:
  - Standard-Error: stderr
  - Default-Stream: Text-Terminal von dem Programm gestartet wurde, außer der Stream wird umgelenkt

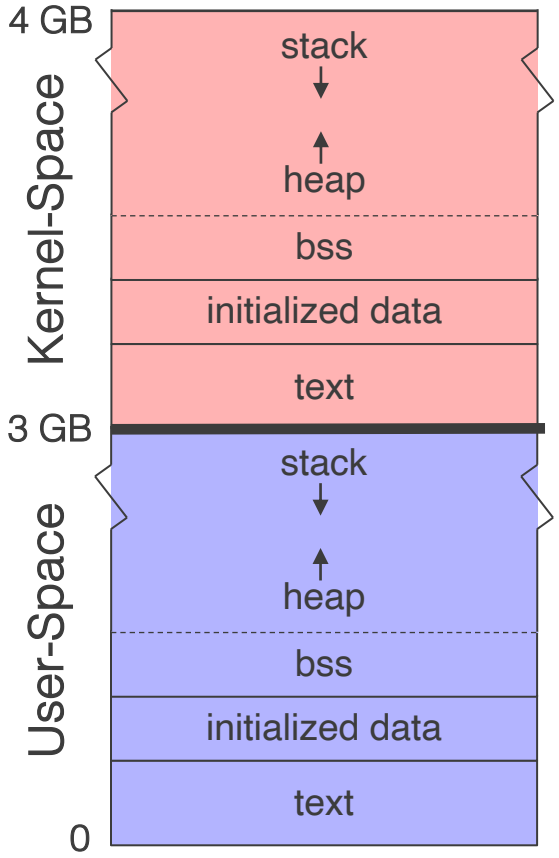


Kern-Strukturen für Dateien



5.7 Speichermodell (alt, 32 Bit)

- Kernel-Space:
  - Gemeinsam für alle Prozessen
  - Durch Hardware geschützt
  - An hohen Adressen,
    z.B. ab 3 GB bei 32 Bit Systemen
  - Betriebssystem, privilegierte Befehle, ...
- User-Space:
  - Getrennt zwischen Prozessen
    → Trennung regelt Betriebssystem
  - Schutz vor unabsichtlichen und
    bösartigen Zugriffen
  - Code, Stack, Daten



5.7 Systemstart

- Bootlader lädt Kernel-Image (steht an definierter Stelle im Verzeichnisbaum)
- Kernel startet dann den ersten Prozess `init` (PID = 1)
- Nachdem die wichtigsten Grundfunktionen initialisiert wurden durchläuft der Startvorgang verschiedene Systemzustände (engl. runlevel)
  - Jedem runlevel sind bestimmte System-Dienste zugeordnet
  - Diese werden beim Booten als Prozesse, in wohldefinierter Reihenfolge, innerhalb des Betriebssystems gestartet.
  - Skripte für den Start von Systemprozessen befinden sich in `/etc/init.d`

Runlevel	Zustand
0	Shutdown
S	Singleuser
1	Multiuser ohne Netzwerk
2	Multiuser mit Netzwerk
3	Multiuser mit Netzwerk und GUI
6	Reboot

5.8 Inter-Prozesskommunikation: Signale

- Signale: kurze wichtige Meldungen über asynchrone Ereignisse
- Generiert von Kern- oder Benutzerprozessen
- Können ignoriert oder verarbeitet werden
- Führen meist zur Terminierung

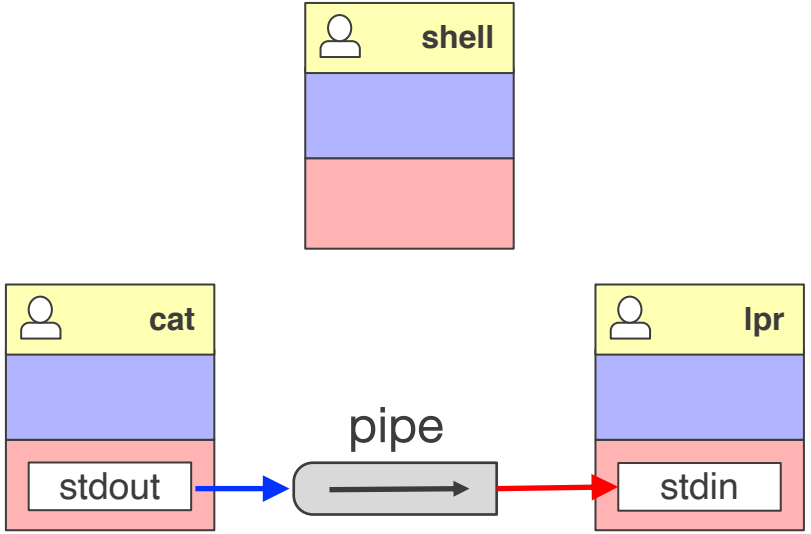
- Signale senden mit `kill –SIG PID`
  - SIG = Signalname bzw. ID
  - PID = ProzessID (an wenn geht das Signal)

UNIX Signale (Auszug)

Signal	Ursache
SIGABRT	Sent to abort process and force a core dump
SIGILL	The process has executed an illegal machine instruction
SIGINT	The user has hit the DEL key to interrupt the process
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written on a pipe with no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

Pipes und Filterketten

- Programme lesen von **`stdin`** und schreiben nach **`stdout`**
- Kein Unterschied, ob lesen/schreiben von/in Datei oder über pipe zu einem anderen Prozess.
  - `>` stdout umlenken
  - `<` stdin umlenken
  - `|` Verknüpfung **`stdout -> stdin`**
- Beispiele:
  - `ls > file`
  - `cat a | lpr`
  - `cat a | sort | lpr`



Anonyme Pipe zwischen Eltern- und Kind-Prozess

```
#include <stdio.h>
#include <unistd.h>           /* different standard constants              */

int main() {
   char   data[80];
   int    rb;                 /* nr of read bytes                          */
   int    pipe_ends[2];       /* handles: read=0; write=1                  */


   pipe(pipe_ends);           /* create anonymous pipe                     */

   if (fork()==0) {           /* child process                             */
      close(pipe_ends[1]);    /* close write end -> process wants to read  */
      rb = read(pipe_ends[0], data,79);
      data[rb]='\0';          /* terminate string                          */
      printf ("%s\n",data);
      close(pipe_ends[0]);    /* done, close read end                      */
   }
   else {                     /* parent process                            */
      close(pipe_ends[0]);    /* close read end -> process wants to writen */
      write(pipe_ends[1],"hello",5);
      close(pipe_ends[1]);    /* done, close write end                     */
   }
}
```

Hörsaal-Aufgabe

- Modifizieren Sie das vorhergehende Pipe-Beispiel
- Das Programm soll eine Zahl als Argument vom Terminal übergeben bekommen. (Zahl muss nicht unbedingt geprüft werden).
- Der Kind-Prozess soll die Quersumme dieser Zahl berechnen und mithilfe der Pipe an den Eltern-Prozess schicken und dann terminieren.
- Der Elternprozess soll das Ergebnis der Berechnung ausgeben und terminieren.
- Beispiel: Zahl = 1234 -> Quersumme = 1+2+3+4 = 10

- Tipp: Das Ergebnis kann man einfach in einen String konvertieren, der dann über die Pipe zurückgeschickt wird → `man sprintf`

5.9 Rechte: Benutzer

- In UNIX werden Benutzer intern dargestellt durch eine User-ID (UID)
  - Speicherort: `/etc/passwd`
  - Passwort separat in `/etc/shadow`
  - UID-Aufteilung (abhängig vom System):
    0: root, 1 - 99: system user, Ab 1000: non-privileged users
  - Format: `Name:Passwort:User-ID:Group-ID:Kommentar:Verzeichnis:Shell`

`/etc/passwd`

```
root:x:0:0:Björn:/root:/bin/bash
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:100:sync:/bin:/bin/sync
mail:x:8:8:mail:/var/mail:/bin/sh
christiane:x:1000:100:Christiane:home/users/christiane:/bin/bash
johannes:x:1001:100:Johannes:/home/users/johannes:/bin/bash
...
```

Gruppen

- Benutzer gehören zu einer oder mehreren Gruppen
- Gruppen werden intern durch eine Group-ID (GID) repräsentiert
  - Speicherort: `/etc/group`
  - Passwort separat in `/etc/gshadow`
  - Format: `Gruppenname:Passwort:Gruppennummer:Mitglied1,...`

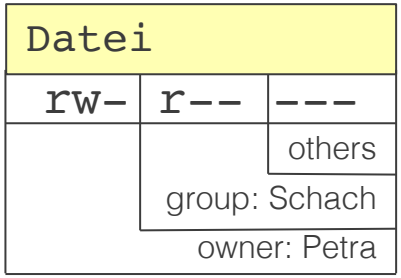
`/etc/group`

```
offline:x:102:ulli,iwer,veritaz
www:!:105:chris,bjoern,iwer,veritaz,hen,robert,anatol
ftp:x:106:chris,iwer
...
```

(`!` bei `www` nur zur Hervorhebung; jede andere Zeichenfolge auch erlaubt)

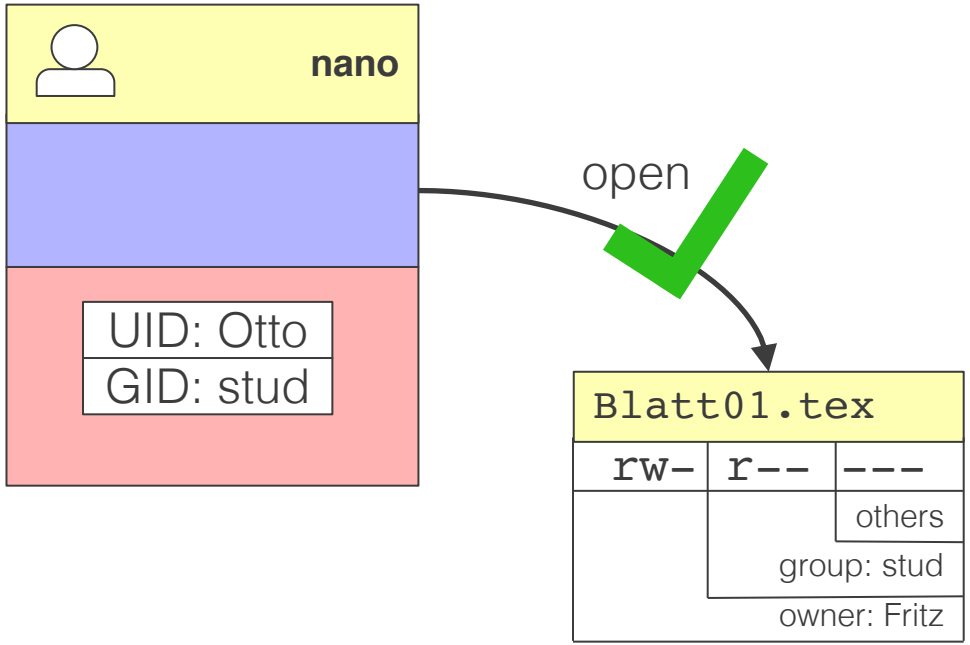
Dateien

- Zugriffsrechte zu Dateien festgelegt in Bezug auf Benutzer
- Jede Datei hat Attribute für Besitzer (steht im Inode)
  - owner: UID und group: GID
- Rechte: r(read), w(write), x(eXecute) → 3 Bit
- Rechte an einer Datei werden festgelegt in Bezug auf
  - owner, group, others (= Rest der Welt)
  - Insgesamt 9 Bit (3 • 3 Bit)
- Rechte an einem Verzeichnis
  - r: Inhalt darf aufgelistet werden
  - x: mit cd darf in das Verz. gewechselt werden
  - x+w: neue Dateien dürfen im Verzeichnis angelegt werden



Prozesse und Dateien

- Die Prozess-Attribute UID und GID bestimmen beim Zugriff auf Dateien die Rechte eines Prozesses.
- Jeder Prozess hat UID und GID
  - Übernommen vom Elternprozess



5.10 Komponenten eines UNIX-Kerns

- Trennung: Kernel-/User-Space/-Mode

- Dispatcher: leitet Systemaufrufe an ihr Ziel
- Scheduler: entscheidet welcher Prozess die CPU als nächstes bekommt
- IPC: Inter-Prozesskommunikation
- Puffer-Cache:
  - puffert Daten für schnelleren Zugriff
  - Aber nur für Block-Geräte
  - Z.B. Festplatte, DVD etc.