Kapitel 2: Der Unix-Prozess

2: Der Unix-Prozess

Agenda

- Prozesskennung PID
- Start und Beendigung
- Der Unix-Prozess im Hauptspeicher
- Erzeugung von neuen Prozessen

Prozesskennung

- Prozess-ID (PID): Eindeutige Prozesskennung
- Die PID ist eine positive Ganzzahl
- Parent-PID (PPID): Prozess ID des Elternprozesses.
- init-Prozess (PID 1): Wird vom Kernel beim Booten generiert.
  - Initalisierung des Userland-Systems
  - Ist im Gegensatz zum Scheduler (Systemprozess) ein Benutzerprozess

```
# ps
  PID TTY          TIME CMD
 5807 pts/1    00:00:00 bash
 7204 pts/1    00:00:00 ps
# ps -q 1 -o pid=,comm=,args=
    1 systemd          /sbin/init
```

Prozessbaum

- Durch das Erzeugen immer neuer Kindprozesse entsteht ein beliebig tiefer Baum von Prozessen ($\Longrightarrow$ Prozesshierarchie)
- Das Kommando **`pstree`** gibt die laufenden Prozesse unter Linux/UNIX als Baum entsprechend ihrer Vater-/Sohn-Beziehungen aus

```
$ pstree
...
 |-gnome-terminal--+-bash-+-emacs-+-{dconf worker}
         |                |       |       |-{gdbus}
         |                |       |       `-{gmain}
         |                |       |-less
         |                |       |-2*[make---MakeSlides.sh---pdflatex]
         |                |       `-pstree
         |                |-bash
         |                |-bash---evince-+-{EvJobScheduler}
         |                |               |-{dconf worker}
         |                |               |-{gdbus}
         |                |               `-{gmain}
...
```

Erfragen der PID und PPID

```
1  #include <unistd.h>
2
3  pid_t getpid(void);
4  pid_t getppid(void);
```

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main() {
5    printf("PID:␣␣%u\n", getpid());
6    printf("PPID:␣%u\n", getppid());
7    return 0;
8  }
```

Erfragen des Namens der Prozesseigentümers

```
1  #include <unistd.h>
2
3  char *getlogin(void);
```

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <errno.h>
4
5  int main() {
6    char *owner = getlogin();
7    if(owner) printf("process␣owner:␣%s\n", owner);
8    else perror("getlogin()");
9
10   return errno;
11 }
```

2.1: Start und Beendigung eines UNIX-Prozesses

- Beim Start eines UNIX-Prozesses wird eine spezielle Startup-Routine ausgeführt.
  - Setzen von **`args`** und **`argv[]`** (Kommandozeilenparameter)
  - Setzen der Environment-Variablen (Sprache, Homedirectory, . . .)
  - Aufruf der **`main()`**-Methode
- Beendigungen eines Unix-Prozesses:
  - Normale Beendigung durch Beenden der **`main()`**-Methode oder durch das Aufrufen von **`exit()`**- bzw. **`_exit()`**-Funktion
  - Abnormale Beendigung durch ein Signal (z. B. **`SIGSEGV`**) oder durch Aufruf der **`abort()`**-Methode

Exit Status

- Jeder Prozess hat einen Exit-Status, den er bei seiner Beendigung an den aufrufenden Prozess zurückgibt.
- Wird ein Prozess ohne Rückgabewert beendet, so ist sein Exit-Status undefiniert (schlechter Programmierstil).
- Die Shell speichert den Rückgabewert in der Variable `?`;
- Rückgabewert 0: Prozess wurde erfolgreich ausgeführt

```
# date; echo $?
```

- Rückgabewert ungleich 0: Es trat ein Fehler auf.

```
# cp; echo $?
```

Beenden ohne und mit `return`-Statement

```
1  #include <stdio.h>
2
3  void main() {
4    puts("I␣have␣no␣exit␣status.␣:-(");
5  }
```

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main() {
5    puts("I␣have␣an␣exit␣status.␣:-)");
6
7    return EXIT_SUCCESS;
8  }
```

Normales Beenden ohne cleanup

```
1 #include <unistd.h>
2
3 void _exit(int status);
```

- Dieser Systemcall beendet einen Prozess.
- Datei-Deskriptoren werden geschlossen.
- Löscht alle temporären Dateien die mit **`tmpfile()`** erstellt wurden.
- Übergebener Parameter: Exit-Status
- Sendet das Signal **`SIGCHLD`** an den Elternprozess

Ende

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main() {
6    printf("Magic");
7    _exit(0);
8   }
```

Frage: Was ist die Ausgabe des Programms?

Normales Beenden mit cleanup

```
1 #include <stdlib.h>
2
3 void exit(int status);
```

- Aufruf der Exit-Handler.
- Leeren alle Datenströme (FILE).
- Aufruf von **`_exit(status)`**.

Einrichten von Exithandlern

```
1 #include <stdlib.h>
2
3 int atexit(void (*function)(void));
```

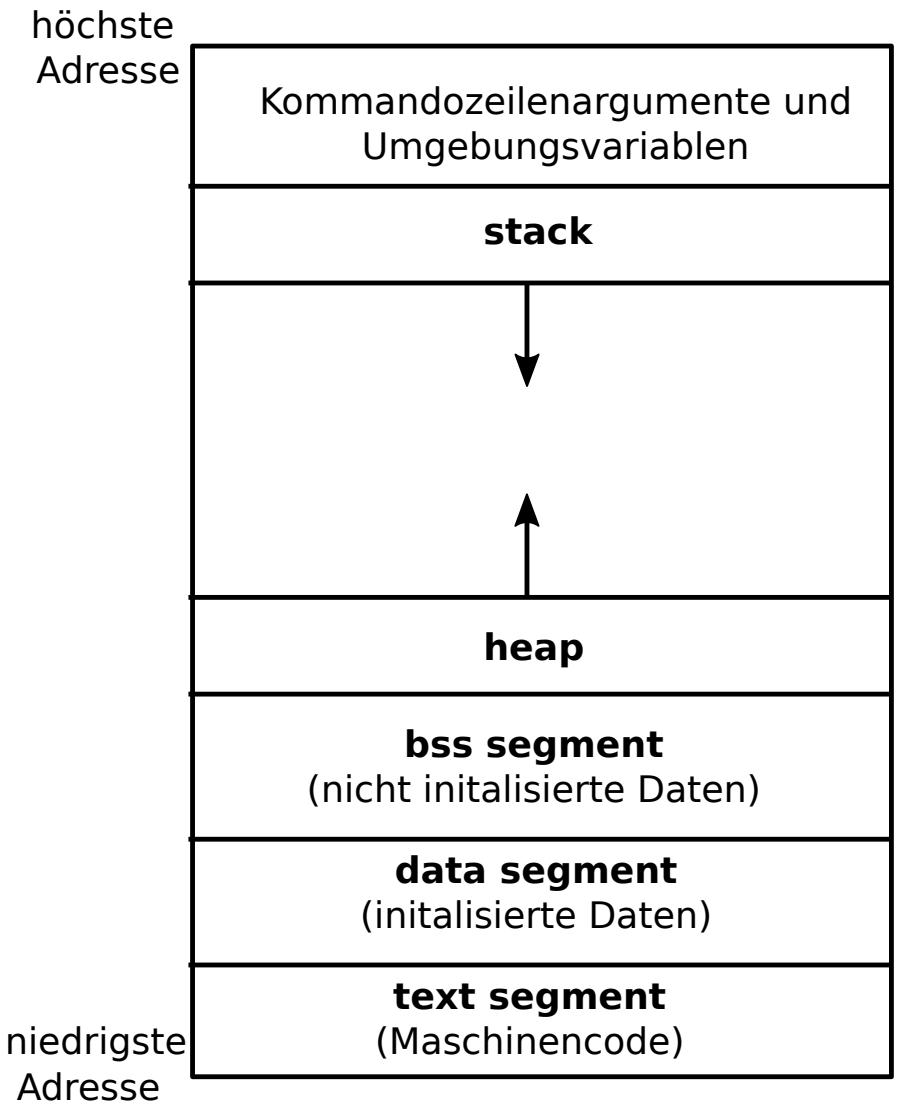
- **`atexit()`** registriert den Exithandler **`function`** welcher bei Beendigung des Programms aufgerufen wird.
- Es können mehrere Exithandler registriert werden.
- Exithandler werden im umgekehrter Reihenfolge (LIFO-Prinzip) aufgerufen.
- Exithandler werden ohne Parameter aufgerufen.
- Es können bis zu 32 Exithandler registriert werden.

Exithandler: Beispiel

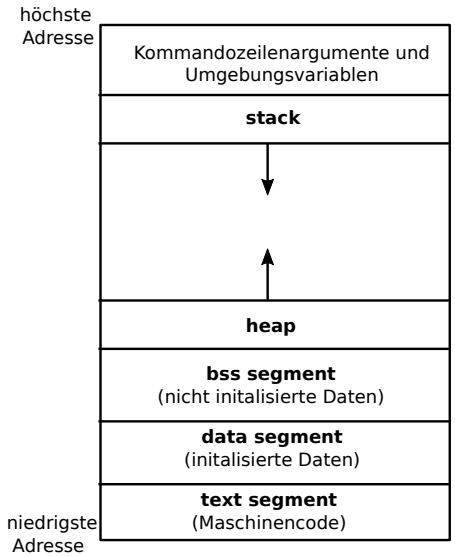
```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  void handler1() { puts("Hasta␣la␣vista."); }
6  void handler2() { _exit(EXIT_SUCCESS); }
7  void handler3() { puts("Goodbye."); }
8  void handler4() { puts("Auf␣Wiedersehn."); }
9
10 int main() {
11   atexit(handler1); atexit(handler2);
12   atexit(handler3); atexit(handler4);
13
14   return EXIT_SUCCESS;
15 }
```

Frage: Was ist die Ausgabe des Programms?

2.3: Unix-Prozesse im Hauptspeicher

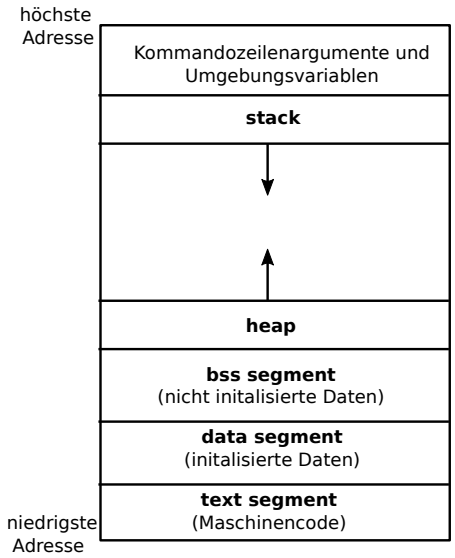


Der Stack



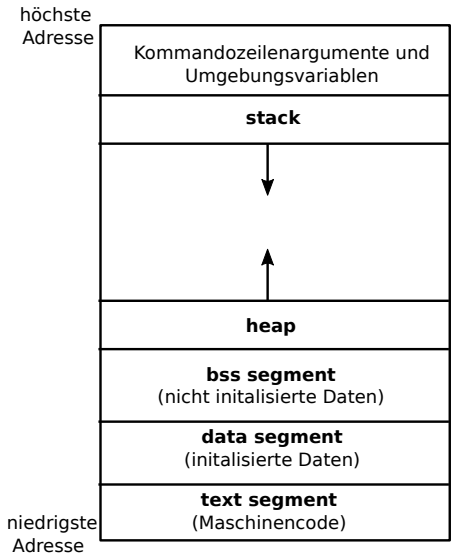
- Bei einem Funktionsaufruf werden die Rücksprungadresse und Parameter auf den `stack` gelegt.
- Im Anschluß legt die aufgerufene Funktion ihre lokalen Variablen auf den `stack`.
- Zu Begin liegen die lokalen Variablen von **`main()`** auf dem `stack`.

Der Heap



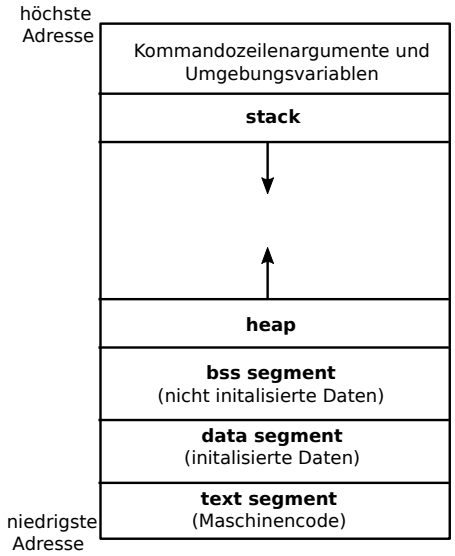
Fordert ein Prozess während seines Ablaufs dynamischen Speicher – z. B. mittels **`malloc()`** – an, so wird dieser aus dem `heap`-Bereich zugeteilt.

bss segment



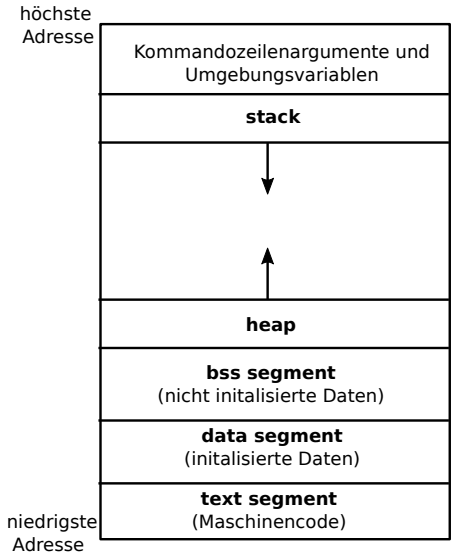
- Der Name `bss` stammt von dem Assembler-Operator **`bss`** (*block started by symbol*).
- Daten dieses Prozesses werden vom Kernel mit 0 initalisiert.
- In dem Segment befinden sich all statischen und globale Variablen, die deklariert aber nicht initalisiert wurden (Beispiel: `static int foobar;`).

data segment



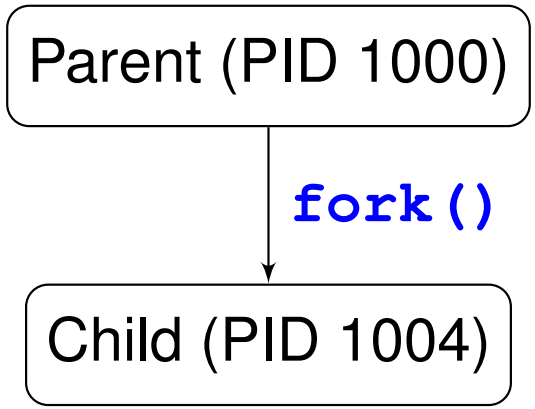
- Das `data segment` enthält alle (initalisierten) statischen und globlen Variablen (Beispiel: `int foo = 23;`).
- Variablen sind global falls sie außerhalb einer Funktion deklariert werden (Beispiel: `static int bar = 42;`).

text segment



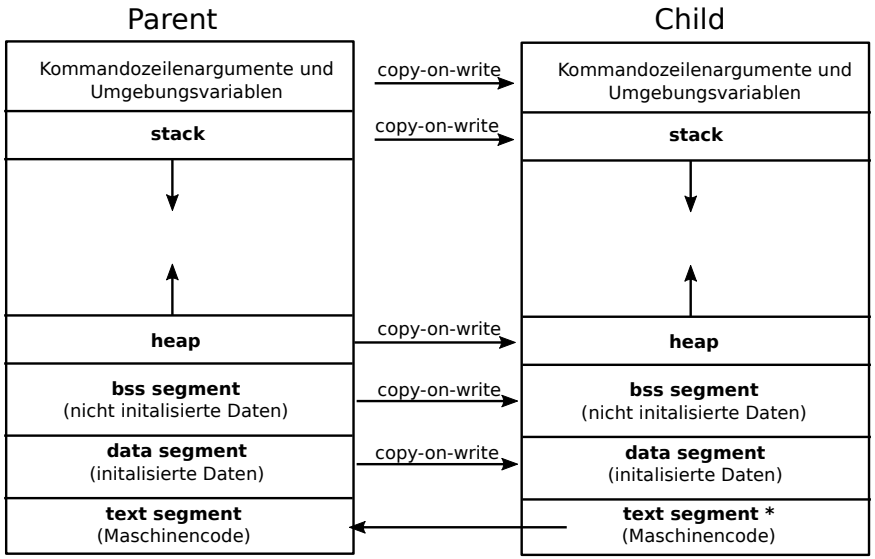
- Das `text segment` enthält den ausführbaren Maschienencode
- Es ist *sharable* und kann daher von mehreren Prozessen parallel genutzt werden.
- Wird ein Programm mehrfach ausgeführt, teilen die Prozesse ein `text segment` um Speicher zu sparen.
- Das `text Segment` ist normalerweise nur lesbar (*read and excecute only*).

2.4: Erzeugung von neuen Prozessen



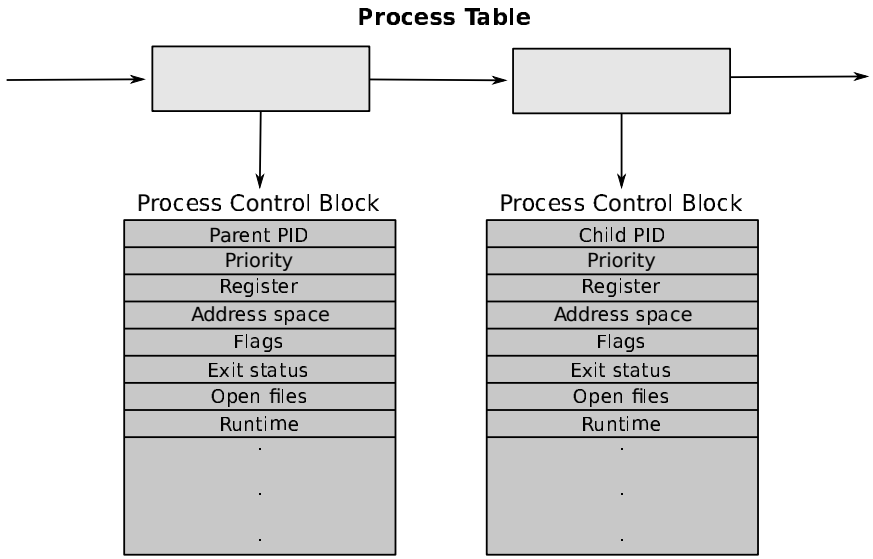
- Mittels dem Systemcall `fork()` können neue Prozesse erzeugt werden, indem der aufrufende Prozess geklont wird.
- Elternprozess: Der aufrufende Prozess (*engl. parent process*).
- Kindprozess: Der neue Prozess (*engl. child process*).
- Die Speicherbereiche von Kindprozess und Elternprozess sind – wie bei allen anderen Prozessen auch – streng voneinander.

`fork()`: Kopieren von Segmenten



- Kind- und Elternprozess teilen sich ein `.text` Segment (Programmcode).
- Speicherbereiche des Elternprozesses werden für den Kindprozess kopiert (copy-on-write) und stehen ihm nun getrennt zur Verfügung. ($\rightarrow$ Tafel)

`fork()`: Kind bekommt einen neuen PCB



- Für das Kind wird ein neuer PCB angelegt.
- Bis auf die PID wird der Inhalt des PCB des Elternprozesses übernommen.
  - Geöffnete Dateien und Sockets.
  - Registerinhalte inklusive Befehlszähler.
  - Prioritäten und Zugriffsrechte.

fork

```
#include <unistd.h>

pid_t fork(void);
```

- Bei Erfolg wird dem Elternprozess die PID des Kindprozesses zurückgegeben.
- Bei Erfolg wird dem Kinderprozess `0` zurückgegeben.
- Bei Fehlern wird dem Parent `-1` zurückgegeben und kein Kindprozess erzeugt.

Benutzung von fork (1/2)

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <errno.h>
4
5  #define CHILD  0
6  #define FAILURE -1
7
8  int main() {
9    switch (fork() ) {
10   case FAILURE: /* Fehlerbehandlung */
11     perror("Fork failed.");
12     break;
13
14   case CHILD: /* Code fuer den Kindprozess */
15     printf("Kind:  %u / %u\n", getpid(), getppid() );
16     break;
17
18   default: /* Code fuer den Elternprozess */
19     printf("Vater: %u / %u\n", getpid(), getppid() );
20     break;
21   }
22   return errno;
23 }
```

Benutzung von fork (2/2)

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <errno.h>
4
5  #define CHILD    0
6
7  int main() {
8    int p = fork();
9    if(p == CHILD) { /* Code fuer den Kindprozess */
10     printf("Kind:␣␣%u␣/␣%u\n", getpid(), getppid() );
11   }
12
13   else if(p > CHILD) { /* Code fuer den Elternprozess */
14     printf("Vater:␣%u␣/␣%u\n", getpid(), getppid() );
15   }
16
17   else { /* Fehlerbehandlung */
18     perror("Fork␣failed.");
19   }
20
21   return errno;
22 }
```

Forkdemo

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define CHILD   0
6  #define FAILURE -1
7  int global  = 100;
8
9  int main() {
10   int p, local = 1;
11
12   switch ( ( p=fork() ) ) {
13     case FAILURE: exit(EXIT_FAILURE);
14     case CHILD: local+=1;  global+=1; break;
15     default: break;
16   }
17
18   printf("%s:␣", (p==0) ? "Child␣" : "Parent");
19   printf("global=%d,␣local=%d\n", global, local);
20   return EXIT_SUCCESS;
21 }
```

Triple Fork

```
#include <unistd.h>

int main() {
  fork();
  fork();
  fork();
}
```

Frage: Wie viele Prozesse werden durch das Programm erzeugt?

Forkbomb

```
1  #include <unistd.h>
2  #include <stdlib.h>
3
4  volatile int foo;
5
6  int main() {
7    while(1) {fork(); foo= foo*foo; }
8    return EXIT_SUCCESS;
9  }
```

Eine Forkbomb, auch Rabbit genannt, ist ein Programm, dessen einziger Zweck es ist, rekursiv Kopien seiner selbst zu starten, alle verfügbaren Systemressourcen zu verbrauchen und so das System zu blockieren. Unter Unix geschieht das im einfachsten Fall mit dem Aufruf des Systemcalls fork in einer Endlosschleife.

Quelle: `https://de.wikipedia.org/wiki/Forkbomb`

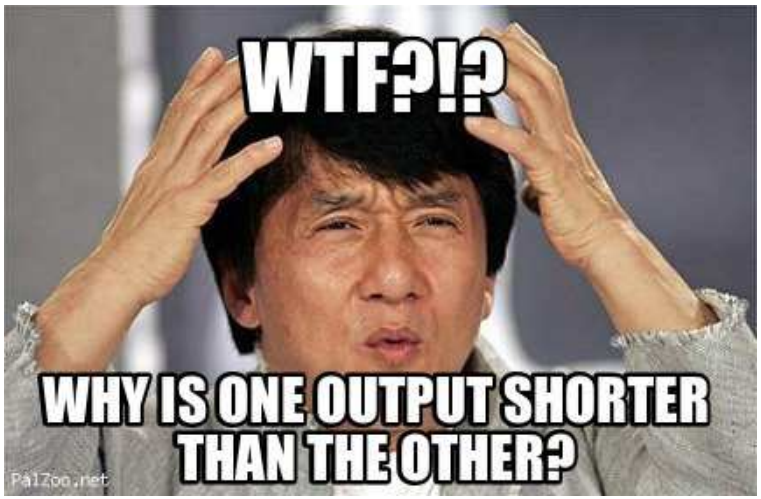
Beispiel: Enkelkind

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define FAILURE  -1
6
7  void output(int var) {
8    printf("(%u␣/␣%u)", getpid(), getppid());
9    printf("var␣=␣%d\n", var);
10 }
11
12 int main() {
13   int var = 0;
14   int p = fork();
15   if(p == FAILURE) exit(EXIT_FAILURE);
16   output(++var);
17
18   p = fork();
19   if(p == FAILURE) exit(EXIT_FAILURE);
20   output(++var);
21
22    return EXIT_SUCCESS;
23 }
```

Spaß mit C – WTF!?!

```
$ ./grandchild
(12673 / 5824) var = 1
(12674 / 12673) var = 1
(12673 / 5824) var = 2
(12675 / 12673) var = 2
(12674 / 12673) var = 2
(12676 / 12674) var = 2
```

```
$ ./grandchild > tmp; cat tmp
(12692 / 5824) var = 1
(12692 / 5824) var = 2
(12692 / 5824) var = 1
(12694 / 12692) var = 2
(12693 / 12692) var = 1
(12693 / 12692) var = 2
(12693 / 12692) var = 1
(12695 / 12693) var = 2
```



Pufferung

- Bei der Standardausgabe (`stdout`) auf ein **Terminal** eingestellt, findet eine **Zeilenpufferung** statt.
- Bedingt durch das `'\n'`-Zeichen wird der Zeilenpuffer geleert.
- Durch den Aufruf von `fork()` wird ein **leerer** Zeilenpuffer an das Kind weitergereicht.

- Ist die Standardausgabe (`stdout`) **nicht** auf ein Terminal eingestellt, findet eine **Vollpufferung** statt.
- Durch den Aufruf von `fork()` wird ein **gefüllter** Puffer an das Kind weitergereicht.

Beispiel: Enkelkind die Zweite (1/2)

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define FAILURE  -1
6
7  void output(int var) {
8    printf("(%u␣/␣%u)", getpid(), getppid());
9    printf("var␣=␣%d\n", var);
10   fflush(stdout);
11 }
12
13 int main() {
14   int var = 0;
15   int p = fork();
16   if(p == FAILURE) exit(EXIT_FAILURE);
17   output(++var);
18
19   p = fork();
20   if(p == FAILURE) exit(EXIT_FAILURE);
21   output(++var);
22   return EXIT_SUCCESS;
23 }
```

Beispiel: Enkelkind die Zweite (2/2)

```
$ ./grandchild2
(13524 / 5824) var = 1
(13524 / 5824) var = 2
(13525 / 13524) var = 1
(13526 / 13524) var = 2
(13525 / 13524) var = 2
(13527 / 13525) var = 2
```

```
$ ./grandchild2 >tmp;cat tmp
(13534 / 5824) var = 1
(13535 / 13534) var = 1
(13534 / 5824) var = 2
(13536 / 13534) var = 2
(13535 / 13534) var = 2
(13537 / 13535) var = 2
```

Frage: Warum unterscheidet sich die Reihenfolge der Ausgabe?

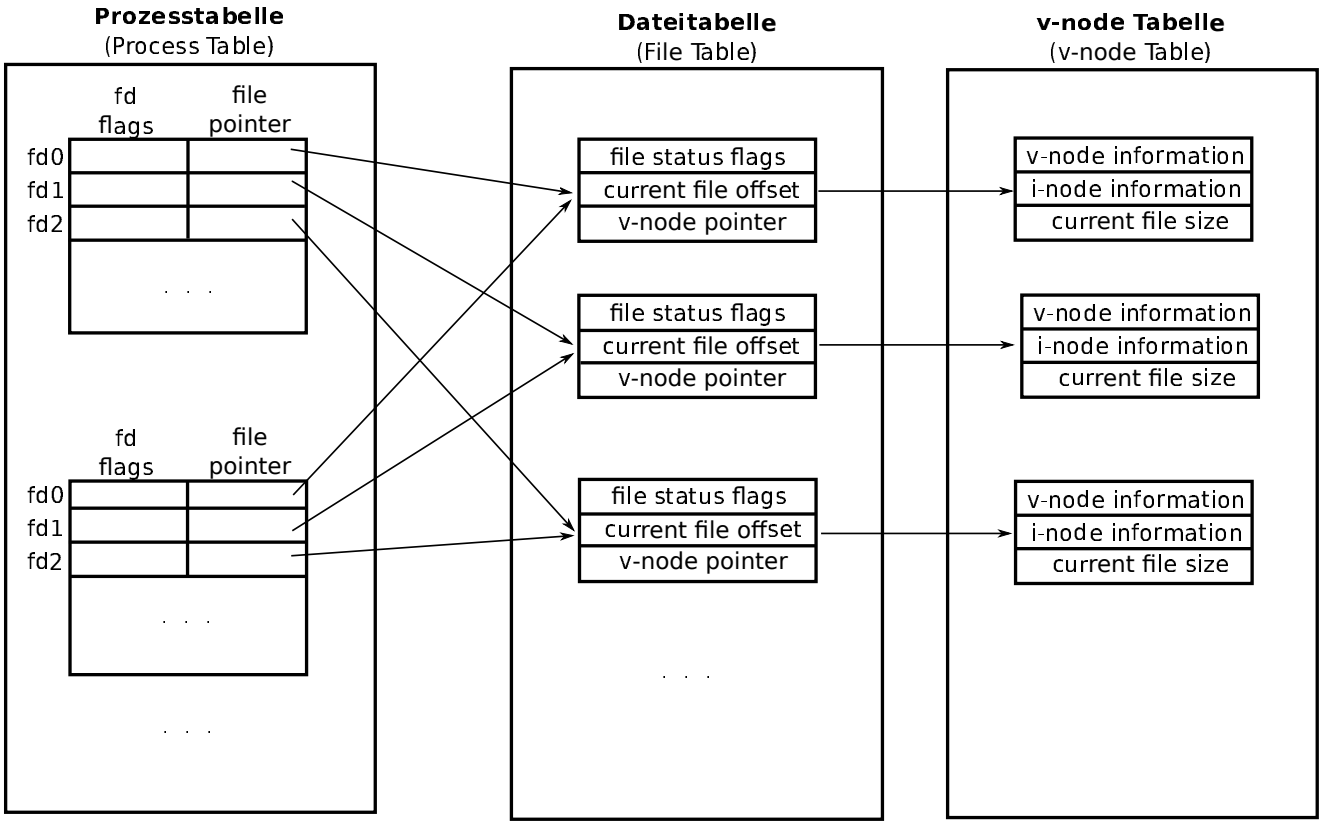
Beispiel: Unabhängiges Arbeiten (1/2)

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define CHILD          0
6  #define THRESHOLD 10000
7
8  int main() {
9    int p = fork();
10   if(p == CHILD) {
11     for(int i=0; i < THRESHOLD; i++)
12       printf("Kind: %d\n", i);
13   }
14
15   else if(p > CHILD) {
16     for(int i=0; i < THRESHOLD; i++)
17       printf("Vater: %d\n", i);
18   }
19
20   else return EXIT_FAILURE;
21   return EXIT_SUCCESS;
22 }
```

Beispiel: Unabhängiges Arbeiten (2/2)

```
$ ./forkdemo2
...
Vater: 380
Vater: 381
VateKind: 0
Kind: 1
Kind: 2
...
Kind: 419
Kind: r: 382
Vater: 383
...
Vater: 753
Vater: 7420
Kind: 421
...
Kind: 829
Ki54
Vater: 755
...
Vater: 1115
Vater:nd: 830
Kind: 831
...
Kind: 1217
 1116
Vater: 1117
...
Vater: 1456
Vater: 145Kind: 1218
Kind: 1219
...
```

Vererbung von Dateideskriptoren



- `fork()` vererbt die Dateideskriptoren an den Kindprozess.
- Umleitungen von Ausgaben werden an Kinder weitervererbt.
- Das Vererben entspricht dem Duplizieren der Filedeskriptoren mittels `dup()`.

Gleichzeitiges Schreiben in eine Datei

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <errno.h>
4  #define CHILD          0
5  #define THRESHOLD 10000
6
7  int work(char *msg, FILE *fp) {
8    int sum = 23;
9    for(int i=0;   i < THRESHOLD; i++) {
10     for(int j=0; j < THRESHOLD; j++) sum*=j+j;
11     fprintf(fp, "%s %d\n", msg, i);
12     fflush(fp);
13   }
14   return sum;
15 }
16
17 int main() {
18   FILE *fp = fopen("zzz.dat","w");
19   int p = fork();
20   if(p == CHILD) work("Kind :",fp);
21   else if(p > CHILD) work("Vater:",fp);
22   return errno;
23 }
```

2.5: Der Reaper wartet auf Zombies

Quelle: `http://pawnstarsthegame.wikia.com/wiki/The_Reaper`

Künstler: John Doppler

Der Zombieprozess

- Der Prozesskontrollblock (PCB) eines Prozess wird erst aus der Prozessliste entfernt (`engl. reaped`) nachdem der Elternprozess dessen Beendigungstatus erfragt hat.
- PCBs von terminierten (Kinder-)Prozessen werden unter UNIX `Zombies` bzw. `Zombie-Prozesse` gennannt.
- Ein Zombie wird nach dem Abfragen seines Beendigungstatus, vom Elternprozess, entfernt.
- Kinderprozesse deren Elternprozesse terminieren, werden als verwaister Prozess bezeichnet.
- **Frage:** Was ist ein verwaister Zombieprozess?

Adoption

- **Frage:** Warum sind verwaiste Prozesse ein Problem?
- Verwaiste Prozesse werden vom Reaper-Prozess (PID 1) adoptiert und entfernt. Dadurch werden nicht mehr benötigte PCBs wieder deallokiert.
- Ein Prozess kann zu einem Subreaper werden. Damit adoptiert er – und nicht mehr der Reaper-Prozess – alle Nachkommen seiner Kinderprozesse.
- Durch `prctl(PR_SET_CHILD_SUBREAPER, 1, 0, 0, 0)` wird beim aufrufenden Prozess das Attribut `SUBREAPER` gesetzt.

Beispiel: Zombie

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/wait.h>
5
6  #define CHILD          0
7
8  int main() {
9    int p = fork();
10   if(p == CHILD) {
11     printf("PID:␣␣%u\n", getpid());
12     exit(1);
13   }
14   else if(p > CHILD) {
15     sleep(30);
16   }
17   else return EXIT_FAILURE;
18   return EXIT_SUCCESS;
19 }
```

Zusammenfassung

Nach diesem Kapitel sollten Sie . . .

- . . . den Rückgabewert eines Prozesses abfragen können.
- . . . in der Lage sein einen Exithandler einzurichten.
- . . . den Umgang mit `fork()` beherrschen.
- . . . wissen dass Dateideskriptoren vererbt werden.
- . . . wissen was ein Zombieprozess ist.