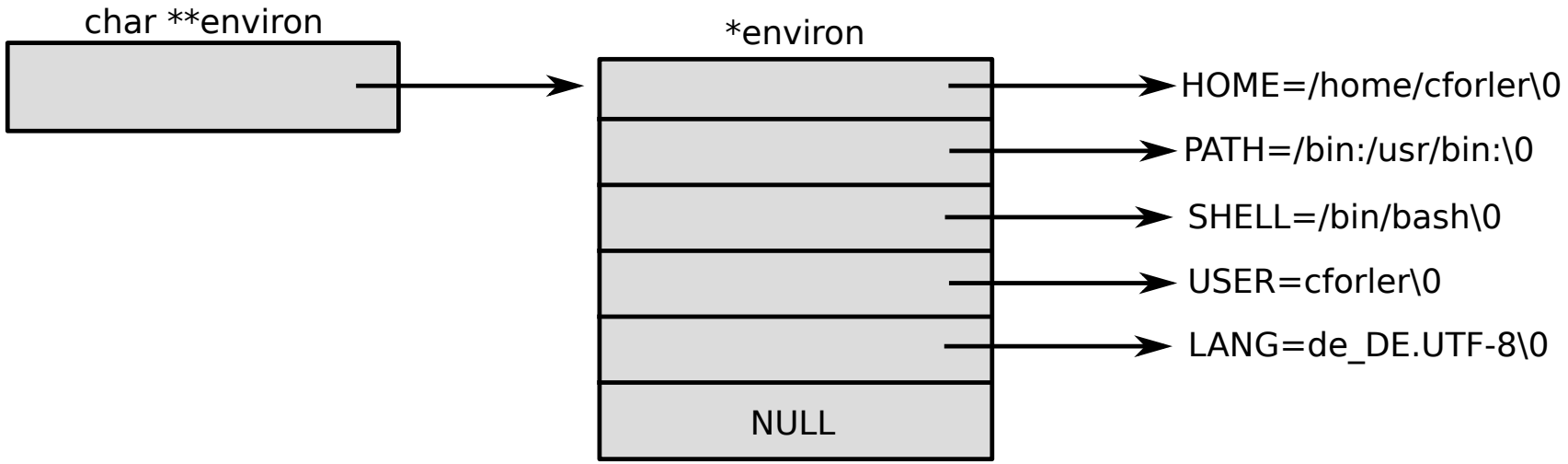
Kapitel 2: Der Unix-Prozess

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?

2.2: Prozessumgebung



- Unix-Prozesse verfügen über eine Umgebung (engl. *environment*)
- Die Environment `extern char **environ;` entspricht einer Liste von Strings im Format `<IDENTIFIER>=<VALUE>`
- Alternativ: Zugriff über einen dritten `main`-Parameter

Environment: Beispiel

```
1  #include<stdio.h>
2  extern char **environ;
3
4  int main () {
5    while(*environ) puts(*environ++);
6    return 0;
7  }
```

```
1  #include<stdio.h>
2
3  int main (int args, char *argv[], char *envp[]) {
4    puts("Arguments");
5    if(args>0) while(*argv) puts(*argv++);
6
7    puts("\nEnvironment");
8    while(*envp) puts(*envp++);
9    return 0;
10 }
```

Werte einzelner Umgebungsvariablen erfragen

```
1  #include <stdlib.h>
2
3  char *getenv(const char *name);
```

- **`getenv()`** sucht in **`environ`** die Umgebungsvariable `name` und gibt einen Zeiger auf dessen Wert zurück.
- **`getenv()`** gibt **`NULL`** zurück falls es keine Übereinstimmung gibt.
- Achtung: Umgebungsvariablen können vom Benutzer gesetzt werden.

Beispiel: Werte von Umgebungsvariablen erfragen

```
1  #include <string.h>
2  #include <ctype.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(int args, char *argv[]) {
7    for(int i=1; i <args; i++) {
8      for(size_t j=0; j < strlen(argv[i]); j++)
9        argv[i][j] = toupper(argv[i][j]);
10
11     char *s = getenv(argv[i]);
12     if(s) printf("%s=%s\n", argv[i], s);
13   }
14   return EXIT_SUCCESS;
15 }
```

Hinzufügen und Löschen von Umgebungsvariablen

```
1  #include <stdlib.h>
2
3  int setenv(const char *name, const char *value,
4             int overwrite);
5
6  int unsetenv(const char *name);
```

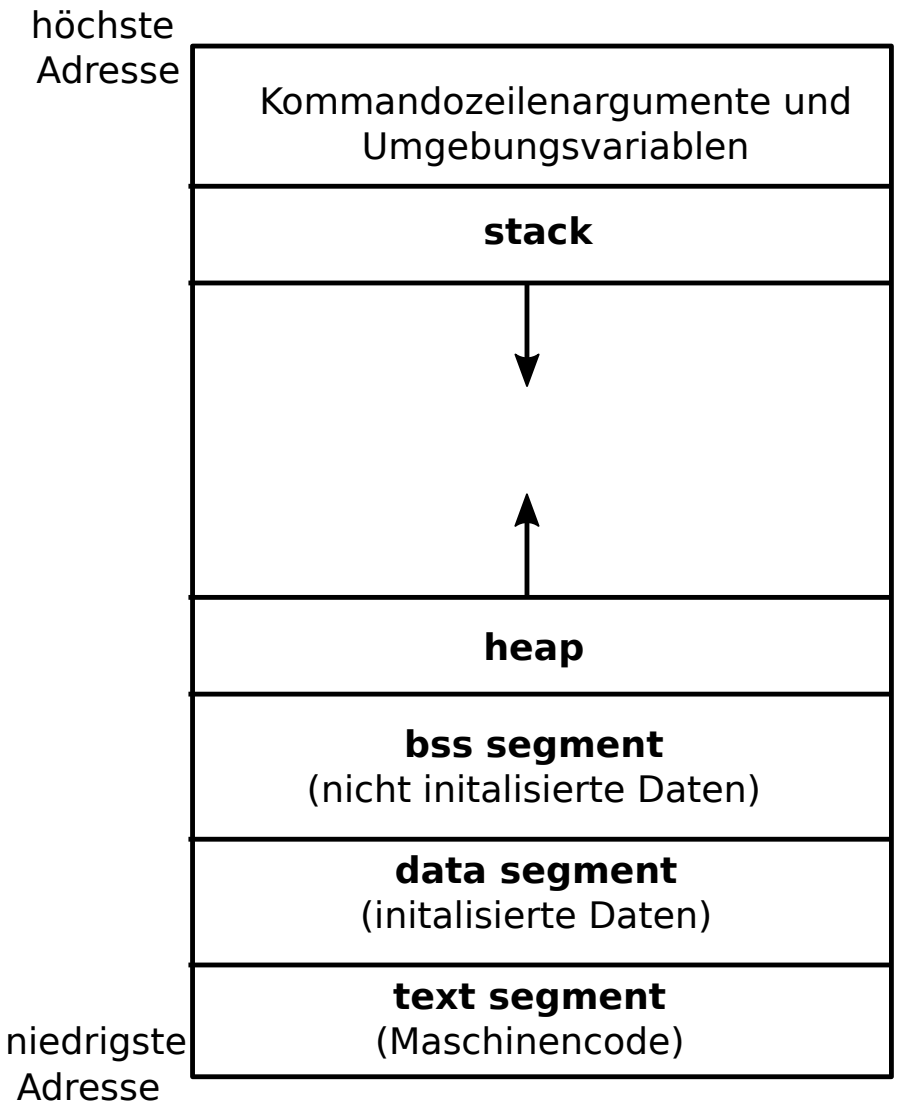
- **`setenv()`** fügt eine Umgebungsvariable **`name`** hinzu oder ändert ihren Wert.
- Falls **`overwrite`** ungleich 0 ist, werden vorhandene Werte überschrieben.
- **`unsetenv()`** entfernt einen Eintrag.
- Beide Funktionen liefern im Erfolgsfall 0 und im Fehlerfall einen Wert ungleich 0 zurück.

Beispiel für setenv und unsetenv

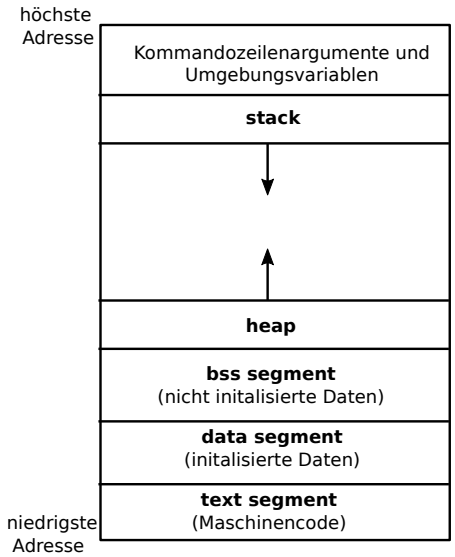
```
1  #include <string.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main() {
6    setenv("FOO","42",0);
7    setenv("USER","root",0);
8    puts(getenv("USER"));
9
10   setenv("FOO","23",1);
11   puts(getenv("FOO"));
12
13   unsetenv("FOO");
14   puts(getenv("FOO"));
15
16   return EXIT_SUCCESS;
17 }
```

Frage: Was ist die Ausgabe des Programms?

2.3: Unix-Prozesse im Hauptspeicher

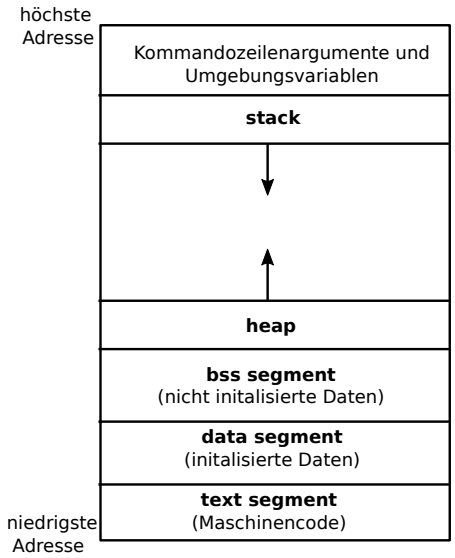


Der Stack



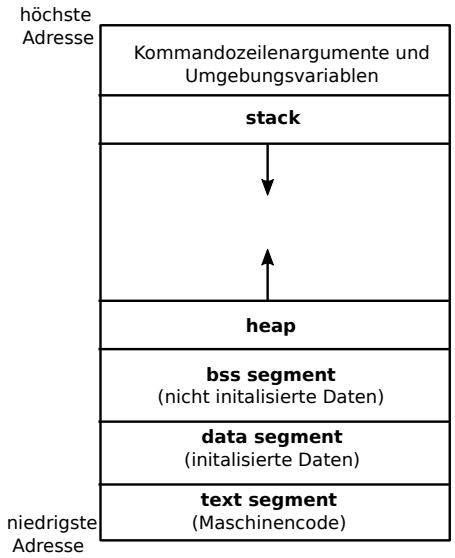
- Bei einem Funktionsaufruf werden die Rücksprungadresse und Parameter auf den `stack` gelegt.
- Im Anschluß legt die aufgerufene Funktion ihre lokalen Variablen auf den `stack`.
- Zu Begin liegen die lokalen Variablen von **`main()`** auf dem `stack`.

Der Heap



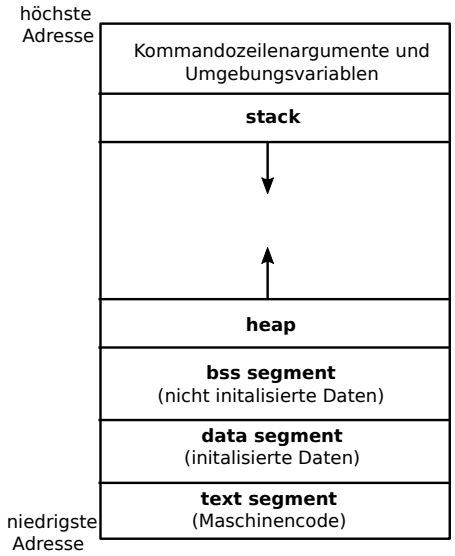
Fordert ein Prozess während seines Ablaufs dynamischen Speicher – z. B. mittels **`malloc()`** – an, so wird dieser aus dem `heap`-Bereich zugeteilt.

bss segment



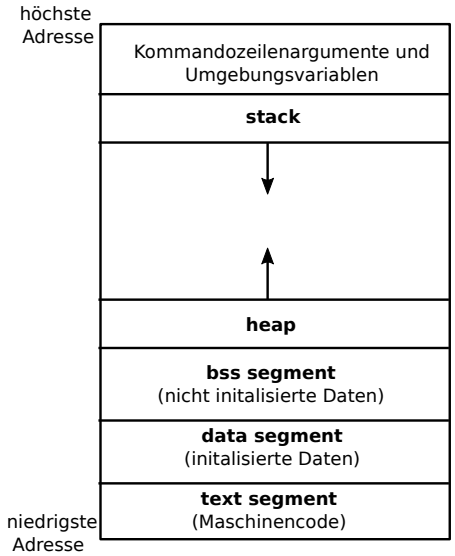
- Der Name `bss` stammt von dem Assembler-Operator **`bss`** (*block started by symbol*).
- Daten dieses Prozesses werden vom Kernel mit 0 initalisiert.
- In dem Segment befinden sich all statischen und globale Variablen, die deklariert aber nicht initalisiert wurden (Beispiel: `static int foobar;`).

data segment



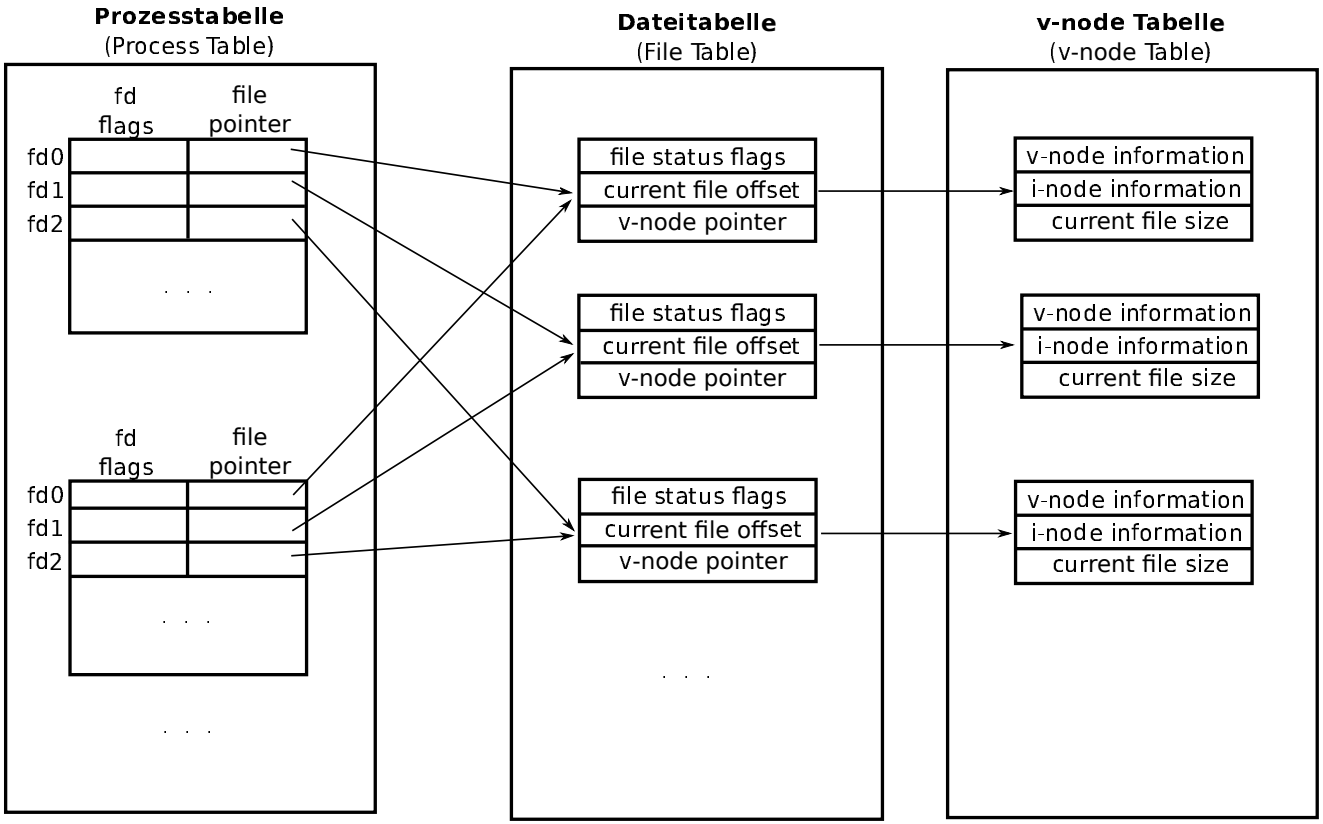
- Das `data segment` enthält alle (initalisierten) statischen und globlen Variablen (Beispiel: `int foo = 23;`).
- Variablen sind global falls sie außerhalb einer Funktion deklariert werden (Beispiel: `static int bar = 42;`).

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*).

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 }
```

Warten auf Beendigung eines Prozesses

```
1 #include <sys/wait.h>
2
3 pid_t wait(int *wstatus);
```

- Bei Erfolg wird die PID des Kindprozesses zurückgegben, ansonsten -1.
- Sofortige Rückkehr wenn...
  - ...ein Kindprozess sich bereits beendet hat und der Kernel auf Abholung des Beendigungstatus dieses *Zombieprozesses* wartet.
  - ...w enn kein Kindprozess exisitert.
- Blockiert bis sich ein Kindprozess beendet.
- `wstatus` kann `NULL` sein falls kein Interesse am Beendigungstatus besteht.

Beendigungsstatus

Der Beendigungstatus kann durch Makros, welche in `<sys/wait.h>` definiert sind, dekodiert werden.

Makro	Beschreibung
`WIFEXITED(wstatus)`	`TRUE`: Prozess hat sich normal beendigt.
`WEXITSTATUS(wstatus`	Exit-Status des Kindprozesses.
`WIFSIGNALED(wstatus)`	`TRUE`: Beendigung durch ein Signal.
`WTERMSIG(wstatus)`	Gibt das Beendigungssignal zurück.

Beispiel: Wait

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#define CHILD          0

int main() {
  int p = fork();
  if(p == CHILD) {
    printf("PID:  %u\n", getpid());
    sleep(10);
    exit(23);
  } else if(p > CHILD) {
    int s;
    wait(&s);
    if(WIFEXITED(s))
      printf("Exit-Status: %d\n", WEXITSTATUS(s));
    if(WIFSIGNALED(s))
      printf("Kill Signal: %d\n", WTERMSIG(s));
    if(WIFSTOPPED(s))
      printf("Stop Signal: %d\n", WTERMSIG(s));
  } else { return EXIT_FAILURE; }
  return EXIT_SUCCESS;
}
```

Auf einen bestimmten Kindprozess warten

```
1 #include <sys/wait.h>
2
3 pid_t waitpid(pid_t pid, int *wstatus, int opt);
```

- Verhält sich ähnlich wie **`wait()`**
- Wartet auf den Kindprozess mit der PID `pid`
- Wird `pid` auf `-1` gesetzt, wird auf einen beliebigen Kindprozess gewartet.
- Falls `opt` auf `WNOHANG` gesetzt wird, blockiert der Aufruf nicht.
- In der Regel wird `opt` auf `0` gesetzt.

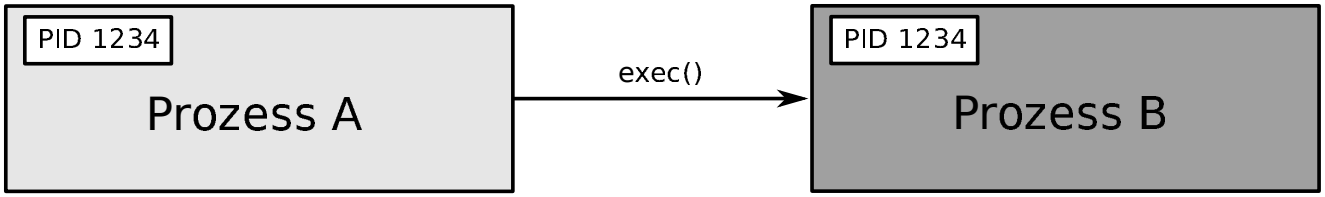
Wettrennen I

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <time.h>
#include <string.h>

#define CHILD          0

static void rennen(int i) {
  int meter   = 0;
  int len     = (i+1)*15-5;
  char *space = malloc(len+1);

  memset(space, '␣',len);
  space[len]='\0';

  srand(time(NULL)+getpid()+i);
  while(meter<50) {
    sleep(rand()%3+1);
    meter += 5;
    printf("%s%d\n", space, meter);
  }
  free(space);
}
```

Wettrennen II

```
27  int main() {
28    pid_t pid[3], pid_ende[3];
29
30    printf("%17s%15s%15s\n",
31      "Laeufer␣1", "Laeufer␣2", "Laeufer␣3");
32    puts("-----------------------------------------");
33
34    if(!(pid[0] = fork())) rennen(0);
35    else if(!(pid[1] = fork())) rennen(1);
36    else if(!(pid[2] = fork())) rennen(2);
37    else {
38      for(int i=0;i<3; i++) pid_ende[i] = wait(NULL);
39
40      puts("Zieleinlauf");
41       for(int i=0;i<3; i++)
42         for(int j=0;j<3; j++)
43           if(pid_ende[i]==pid[j])
44             printf("␣␣Laeufer␣%d\n",j+1);
45    }
46    return EXIT_SUCCESS;
47  }
```

2.6: Die exec()-Funktionsfamilie (1/2)



- Die Syscalls **`execve()`** und **`execveat()`** ersetzen einen Prozess durch einen anderen.
- Es findet eine Prozess-**Verkettung** statt
- Der neue Prozess erbt die PID
- Ein Prozess kann ein neues Programm starten indem er **`fork()`** aufruft und der Kindprozess dann mittels eines **`exec()`**-Aufrufs das Programm startet.
- Die Shell geht auch so vor.
- Ohne **`fork()`** wird der Elternprozess einfach ersetzt.

Die exec()-Funktionsfamilie (2/2)

```
1  #include <unistd.h>
2
3  int execl(const char *path, const char *arg, ...,  NULL);
4  int execlp(const char *file, const char *arg, ..., NULL);
5  int execle(const char *path, const char *arg, ..., NULL,
6    char * const envp[] */);
7
8  int execv  (const char *path, char *const argv[]);
9  int execvp (const char *file, char *const argv[]);
10 int execvpe(const char *file, char *const argv[],
11   char *const envp[]);
```

- Im Fehlerfall wird `-1` zurückgegeben.
- Führt Binaries und Skripte aus
  (nur bei Shebang und angegebenem Interpreter)

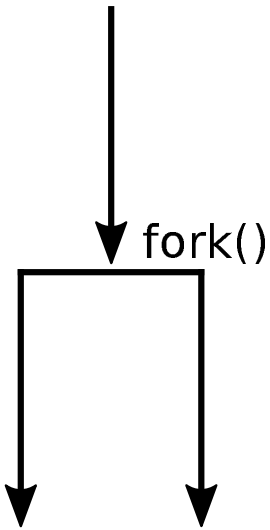
exec()-Demo

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(int args, char *argv[]) {
5    if(args>1)
6    {
7        printf("PID/PPID: %u / %u\n", getpid(), getppid());
8        execv(argv[1], argv+1);
9        printf("Hallo\n");
10       perror(argv[1]);
11   }
12   else
13     fputs("Usage: exec <cmd> [args...]\n" ,stderr);
14   return -1;
15 }
```

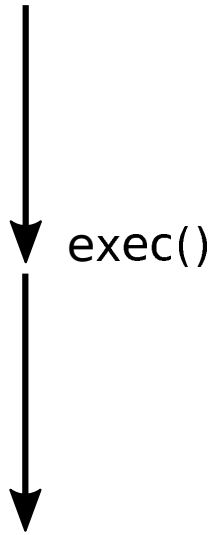
```
./exec ./getpid; echo $?
PID/PPID: 6840 / 6227
PID:  6840
PPID: 6227
0
```

Prozesserzeugungsstrategien

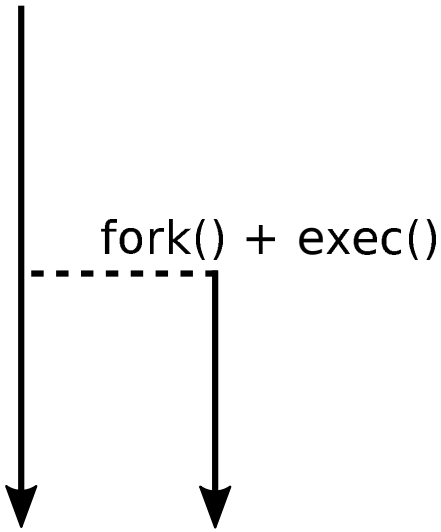
Prozessvergabelung
(forking)



Prozessverkettung
(chaining)



Prozesserzeugung
(creation)



Beispiel: Prozesserzeugung

```
1  #include <readline/readline.h>
2  #include <sys/wait.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6
7  #define CHILD    0
8  #define FAILURE -1
9
10 int main() {
11   int status;
12   while(1) {
13     char *command = readline("cli:␣");
14
15     if((!strcmp("exit",command))||(!strcmp("quit",command))) break;
16
17     if(!strcmp("status",command)) {
18       printf("%d\n",WEXITSTATUS(status));
19       continue;
20     }
21
22     switch(fork()) {
23     case CHILD: execl("/bin/sh", "sh", "-c", command, NULL); break;
24
25     case FAILURE: perror("fork()"); exit(EXIT_FAILURE); break;
26
27     default: /* Parent */ wait(&status);
28     }
29   }
30 }
```

Erzeugung eines Prozesses in Windows

```
1  #ifdef WINDOWS8 | WINDOWS_SERVER_2012
2    #include <WinBase.h>
3  #else
4  #include <Processthreadsapi.h>
5  #endif
6
7  BOOL WINAPI CreateProcess(
8   _In_opt_    LPCTSTR                lpApplicationName,
9   _Inout_opt_ LPTSTR                 lpCommandLine,
10  _In_opt_    LPSECURITY_ATTRIBUTES  lpProcessAttributes,
11  _In_opt_    LPSECURITY_ATTRIBUTES  lpThreadAttributes,
12  _In_        BOOL                   bInheritHandles,
13  _In_        DWORD                  dwCreationFlags,
14  _In_opt_    LPVOID                 lpEnvironment,
15  _In_opt_    LPCTSTR                lpCurrentDirectory,
16  _In_        LPSTARTUPINFO          lpStartupInfo,
17  _Out_       LPPROCESS_INFORMATION  lpProcessInformation
18 );
```

Quelle: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms682425%28v=vs.85%29.aspx`

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.
- . . . den Umgang mit `wait()` und `waitpid()` beherrschen.
- . . . den Umgang mit den `exec()`-Funktionen beherrschen.
- . . . wissen was ein Zombieprozess ist.