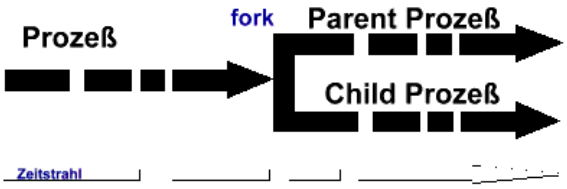
Inhaltsverzeichnis

1. Posix: process – fork,getpid,wait ... 1
1.1. fork() - erzeuge einen Kind-Prozess ... 1
1.1.1. Beispiel: demo_fork.c ... 1
1.1.2. Beispiel: demo_punktstrich.c ... 3
1.1.3. Aufgabe: fork_selfalarm.c ... 3
1.2. execlp() - Prozesse überlagern ... 4
1.2.1. Beispiel: demo_execlp.c ... 4
1.3. wait() - warten auf Prozesse ... 4
1.3.1. Beispiel: demo_wait.c ... 5
1.3.2. Aufgabe: fork_fileserver.c ... 6

1. Posix: process – fork,getpid,wait

1.1. fork() - erzeuge einen Kind-Prozess



- Vater- und Kindprozess haben einen **getrennten Adressraum**.
- Nach einem fork() wird ein sogenannter Child-Prozess erzeugt, der einen eigenen (echte Kopie des Parent-Prozesses) Speicher besitzt.
- Auch der IP (Instruction Pointer) wird kopiert. Deshalb ist eine Verzweigung nach dem fork() wichtig. (s.u.)

  Anmerkung:
  Bei **Threads** i(pthread) st dies nicht der Fall. Hier **teilen sich die Threads den Adressraum**. Jeder Thread besitzt allerdings einen eigenen Stack und eigene Statusinformationen. Dazu aber weiter unten genaueres.

1.1.1. Beispiel: demo_fork.c

Bringen Sie das folgende Programm zum Laufen:

```
// a.hofmann
// demo_fork.c

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

void childs_code();
void parents_code();
```

```
int main() {
     int i;
     int pid= getpid();

     printf("\nJust one process till now: PID= %i", pid);
     printf("\nCalling fork() in 5 sec ...\n");

     for(i=1; i<=5; i++){
          printf("%i sec, \n", i);
          sleep(1);
     }

     switch(fork()){
          case  -1: exit(1); // error
          case   0: childs_code(); break;
          default: parents_code(); break;
     }
     printf("\n\n");
     return 0;
}

void childs_code(){
     int pid= getpid();
     int i;

     for (i=1; i<=10; i++){
          printf("\n                    CHILD: PID= %i i=%i\n", pid, i);
          sleep(1);
     }

     exit(0);
}

void parents_code(){
     int pid= getpid();
     int i=1;

     for (i=1; i<=10; i++){
          printf("\nPARENT: PID= %i i=%i\n", pid, i);
          sleep(1);
     }
}

// gcc demo_fork.c -o demo_fork.exe; ./demo_fork.exe
```

☑ Anmerkung:
Erst bei schreibendem Zugriff auf Variablen, werden diese kopiert.
D.h. parent und child haben dann eigene Variablen

☑ Anmerkung:
oft wird statt des switch ein if...else if … else verwendet;
...
int pid= fork();
if (pid==0) childs_code();
else if(pid>0) parents_code();
else exit(1);

1.1.2. Beispiel: demo_punktstrich.c

```
//demo_punktstrich.c

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

// child und parent verwenden diese Funktion
void arbeite(char*);

int main() {
    int i = 0;
    int status;

    // -------------------------------
    // Prozess erzeugen
    // -------------------------------
    status= fork();


    // -------------------------------
    // CHILD
    // -------------------------------
    if (status == 0) {
       arbeite(".");
    }

    // -------------------------------
    // PARENT
    // -------------------------------
    else if (status > 0)
       arbeite("-");

    else
       printf("\nERROR: Kein Child erzeugt!!\n");

    return 0;
}

void arbeite(char *s) {
    while(1) {
        printf("%s", s);
    }
}

// gcc punktstrich.c -o punktstrich.exe; ./punktstrich.exe
```

1.1.3. Aufgabe: fork_selfalarm.c

Schreiben Sie das obige Programm demo_selfalarm.c derart um, dass das Verhalten gleich bleibt, aber nur folgendermaßen aus der Shell aufgerufen werden muss.

./fork_selfalarm.exe 1 0

1.2. execlp() - Prozesse überlagern

Wir wollen nun die sogenannte Prozessüberlagerung kennenlernen. Ein Prozess kann sich selbst in einen anderen Prozess "verwandeln".

1.2.1. Beispiel: demo_execlp.c

Bringen Sie das folgende Programm zum Laufen:

```
// Prozessüberlagerung mit execlp(), ....
// demo_execlp.c

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

int main(){
    int rv;

    printf("Demo: execlp(): Ausfuehrung von sort:\n");
    printf("   Bitte geben sie einzelne Textzeilen ein.\n");
    printf("   Diese werden nach <Ctrl-D> sortiert ausgegeben\n");

    rv = execlp("/usr/bin/sort", "sort", NULL);

    /* Fehlerfall!!: Dürfte eigentlich nicht mehr ausgeführt werden */
    printf("Fehler bei execlp");
    exit(1);
}

// gcc -o demo_execlp.exe demo_execlp.c; ./demo_execlp.exe
```

Hinweis:
rv = execlp("/bin/ls", "ls", "-l", NULL);

1.3. wait() - warten auf Prozesse

Oft bzw. meist ist es praktisch/notwendig, dass der Vaterprozess auf den Kindprozess wartet. Dabei spricht man von einer Art Synchronisation.

Die Funktion **wait**() kann dazu verwendet werden. Man kann dadurch auch den Rückgabewert (return bzw. exit()) des Kindprozesses erhalten. Also eine einfache Art der Kommunikation erreichen.

Hinweis:

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

3 Formen sind möglich:

1. `pid_of_child= wait (&status);`
   `//status= der vom child mittels exit() zurückgeg. Wert`

2. `pid_of_child= wait ((int*)0);`
   `//status ist für parent nicht interessant`

3. `pid_t waitpid(pid_t pid, int *status, int options);`

   ```
   man waitpid
   … The waitpid() system call suspends execution of the calling process
   until a child specified by pid argument has changed state. By default,
   waitpid() waits only for terminated children, but this behavior is mod-
   ifiable via the options argument, as described below...
   ```

Anmerkung:

Wenn der Parentprozess nicht auf den Childprozess warten soll, aber kein Zombie erzeugt werden soll, verwendet man:

```
waitpid(-1, NULL, WNOHANG);
```

In diesem Fall werden die Zombies vom init-Prozess verwaltet, sobald der Parent beendet.

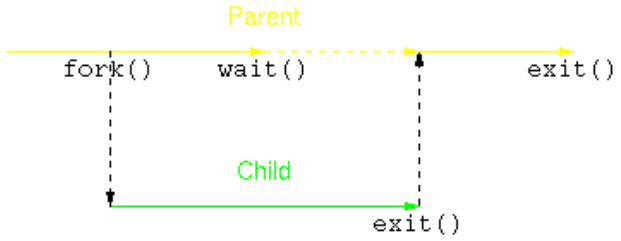
1.3.1. Beispiel: demo_wait.c

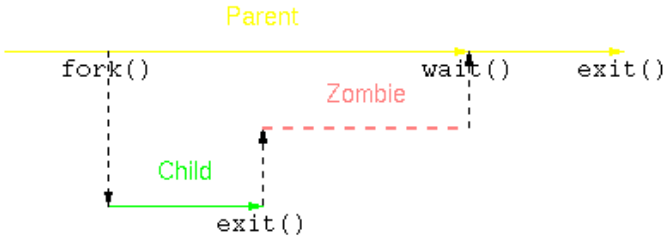
Der laufende Prozess teilt sich durch **fork** auf in Parent und Child, Parent bleibt unverändert. Child lädt durch **execlp** ein anderes Programm und verwandelt sich in dieses.

Von da an laufen beide Prozesse konkurrent zueinander ab und verrichten verschiedene Aufgaben. Durch **wait** kann Parent sich mit dem Child **synchronisieren** in dem Sinne, dass er wartet, bis dieser fertig ist. Durch exit() kann das Child zusätzlich eine Erfolgsmeldung an den Parent-Prozess geben.

```
// demo_wait.c
// 2 unabh. prozesse starten: fork-> execlp | wait

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

int main() {
   int pid;
   printf("Demo: fork(), execlp(), wait(): Ausfuehrung von sort\n");
   printf("   Bitte geben sie einzelne Textzeilen ein.\n");
   printf("   Diese werden nach <Ctrl-D> sortiert ausgegeben\n");

   pid = fork();

   /* Parent-Prozeß ----------------------  */
   if (pid > 0) {
      int status;

      wait(&status); /* wartet, bis Child fertig */

      printf("PARENT-Process: CHILD is ready & returned: %i\n", status);
      exit(0);
   }

   /* Child-Process ---------------------- */
   if (pid == 0) {
```

```
      execlp("/usr/bin/sort", "sort", NULL);

      /* Fehlerfall */
      printf("Fehler bei execlp");
      exit(1);
   }

}
// gcc -o demo_wait.exe demo_wait.c ; ./demo_wait.exe
```





Anmerkung:

Wenn der Parentprozess nicht auf den Childprozess warten soll, aber kein Zombie erzeugt werden soll, verwendet man:

```
waitpid(-1, NULL, WNOHANG);
```

In diesem Fall werden die Zombies vom init-Prozess verwaltet, sobald der Parent beendet.

1.3.2. Aufgabe: fork_fileserver.c

Schreiben Sie das Programm **t_fileserver.c** derart um, dass das Filehandling durch einen Child-Prozess realisiert wird. Dadurch kann der Server bereits den nächsten Request eines Client-Programmes **t_fileclient.c** annehmen. Es können demnach mehrere Clients gleichzeitig bedient werden.

Hinweis:
Um keine sog. Zombie-Prozesse zu erzeugen, vergessen Sie für den Parent-Prozess nicht auf die Funtkion **waitpid(-1,NULL,WNOHANG).**