4 Quest

4.1 Aufgabe

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

int main(int argc, char *argv[])
{
  FILE *dz;
  int i;

  dz = fopen("test.txt", "w");
  if (dz==NULL)
  {
    printf("Die Datei test.txt konnte nicht geoeffnet werden!\n");
    exit(1);
  }
  else
  {
    switch(fork())
    {
      case -1:
        printf("fork-Fehler!\n");
        exit(1);
      case 0:
        for (i=0; i<10; i++)
        {
          fprintf(dz, "Kindprozess: %2d \n", i);
          fflush(NULL);
          sleep(1);
        }
        exit(0);
      default:
        for (i=0; i<10; i++)
        {
          fprintf(dz, "Elternprozess: %2d \n", i);
          fflush(NULL);
          sleep(1);
        }
        fclose(dz);
    }
    exit(0);
  }
}
```

Dieses Programm führt zu dem folgenden Inhalt der Datei:

```
Kindprozess:  0
Elternprozess:  0
Kindprozess:  1
Elternprozess:  1
Kindprozess:  2
Elternprozess:  2
Kindprozess:  3
Elternprozess:  3
Kindprozess:  4
Elternprozess:  4
Kindprozess:  5
Elternprozess:  5
Kindprozess:  6
Elternprozess:  6
Kindprozess:  7
Elternprozess:  7
Kindprozess:  8
Elternprozess:  8
Kindprozess:  9
Elternprozess:  9
```

4.2 Aufgabe

Je nach Implementation kann es sein, dass sich `fork()` und `vfork()` nicht unterscheiden. Ursprünglich unterschieden sich die beiden Befehle doch im Folgenden:
`fork()` erstellt eine komplette Kopie des Adressraums vom Elternprozess. Nutzt man stattdessen `vfork()`, so teilt sich der Kindprozess den Adressraum mit dem Elternprozess bis zu einem `exec`-Befehl. Dabei wird bis zu diesem Befehl der Elternprozess blockiert um die Konsistenz zu wahren.

4.3 Aufgabe

Der Elternprozess P-0 erzeugt vier Prozesse P-0.1, P-0.2, P-0.3, P-0.4. Dann erzeugt P-0.1 drei weitere Prozesse P-0.1.2, P-0.1.3 und P-0.1.4. Sukzessive werden dadurch insgesamt 16 Prozesse erzeugt:

<table>
<tr><td colspan="8">P-0</td></tr>
<tr><td colspan="5">P-0.1</td><td colspan="2">P-0.2</td><td>P-0.3</td><td>P-0.4</td></tr>
<tr><td colspan="2">P-0.1.2</td><td colspan="2">P-0.1.3</td><td>P-0.1.4</td><td>P-0.2.3</td><td>P-0.2.4</td><td>P-0.3.4</td><td></td></tr>
<tr><td>P-0.1.2.3</td><td>P-0.1.2.4</td><td colspan="2">P-0.1.3.4</td><td></td><td>P-0.2.3.4</td><td colspan="3"></td></tr>
<tr><td>P-0.1.2.3.4</td><td colspan="7"></td></tr>
</table>

4.4 Aufgabe

Dieses Programm erzeugt einen Kindprozess, der sich selbst beendet:

```
else if (pid == 0)
  exit(0);
```

Das Betriebssystem lässt den Prozess jedoch weiterhin in der internen Datentabelle eingetragen bis der Elternprozess den `exit`-Wert des Kindprozesses liest oder der Elternprozess beendet wird. Dadurch entsteht

ein sogenanntes “Zombie” - ein Prozess, der war beendet wurde, jedoch noch weiterhin in der Prozesstabelle geführt wird.
Dies erkennt man auch an der Ausschrift `<defunct>` von `ps`, etwa:

```
  PID TTY          TIME CMD
 2871 pts/0    00:00:00 bash
 3115 pts/0    00:00:00 myapp
 3116 pts/0    00:00:00 myapp <defunct>
 3117 pts/0    00:00:00 ps
```

Beim Ausführen des Befehls `ps x` statt `ps` würde unter `STAT` das Zombie durch ein `Z` markiert werden.

4.5 Aufgabe

Bibliothek:	`pthread.h`
Funktionen:	`pthread_create, pthread_join`
Compiler-Flag:	`-lpthread`