1. Aufgabe
    a. Es wird ein Fork erstellt und gewisse Daten werden ausgegeben. Zum Beispiel die Process ID des aktuellen und des Elternprozesses und die i Variable.
    b. `i vor fork: 5`

       `... wir sind die Eltern 5376 mit i=4 und Kind 5377,`
       `unsere Eltern sind 5266`

       `... ich bin das Kind 5377 mit i=6, meine Eltern sind 5376`

       `. . . . . und wer bin ich ?`

       `. . . . . und wer bin ich ?`
2. Aufgabe
    b. `i vor fork: 5`

       `... wir sind die Eltern 5447 mit i=4 und Kind 5448,`
       `unsere Eltern sind 5266`

       `... ich bin das Kind 5448 mit i=6, meine Eltern sind 5447`

       `. . . . . und wer bin ich ?`

       Da nicht mehr das gleiche Programm läuft, verbleibt das zweite "und wer bin ich?". Der Kindprozess wurde komplett mit dem neuen Programm erstetzt.
    c. Da das Programm jetzt nicht gefunden werden kann, wird das Image nicht ersetzt und das bestehende Programm wird im Fork fortgesetzt.
    d. `perror` ergänzt die Ausgabe um einen sinnvollen Fehlertext.
3. Aufgabe
    a.
```
Parent
|-> fork1
  |-> fork1.1
    |-> fork1.1.1
      |-> fork1.1.1.1
    |-> fork1.1.2
  |-> fork1.2
    |-> fork1.2.1
  |-> fork1.3
|-> fork2
  |-> fork2.1
    |-> fork2.1.1
  |-> fork2.2
|-> fork3
  |-> fork3.1
|-> fork4
```
    b.
```
┬─ProcA3.e─┬─ProcA3.e─┬─ProcA3.e──ProcA3.e
│          │          └─ProcA3.e
│          ├─ProcA3.e──ProcA3.e
│          └─ProcA3.e
├─ProcA3.e─┬─ProcA3.e──ProcA3.e
│          └─ProcA3.e
├─ProcA3.e──ProcA3.e
```

```
└─ProcA3.e
```

4. Aufgabe
    a. Da nur 1 Thread verwendet wird, wird oft ge-Kontext-Switched und Mother und Child wechseln sich jeweils ab.
5. Aufgabe
    a. Der Parent Prozess wird vor dem Child terminieren und wird vom Bash Prozess geerbt (falls in der Shell gestartet).
    b. Man erkennt klar, dass das ursprüngliche Programm schon terminiert ist, da die Bash bereits die Prompt wieder anzeigt und gleichzeitig weitere Daten in stdout geschrieben werden.
    c. Der Prozess bleibt der Parentprozess des Kindes.
6. Aufgabe
    c. Die Child-Prozesse terminieren bevor für sie vom main Prozess "gewaited" wird, weshalb sie zu Zombies (defunct) werden.
    d.
        i. Denn Child-Prozess ignorieren und sich selbst terminieren, Child wird zum Waisen
        ii. Denn Child-Prozess beenden
7. Aufgabe
    a. Es werden mehrere Child-Prozesse gespawnt, die dann verschiedene Aufgaben erfüllen (je nach Argument).
    b. Programm wird beendet da ein Segmentation Fault auftritt.
    Es wird das Core signal geschickt.
    Der Absturz passiert auf Zeile 35:
    ```
    case 1: *a     = i;             // force segmentation error
    ```
    c. Default Handler für Signal 30 wird ausgeführt (Power failure restart).
    d. Der Kindprozess wird abgebrochen und es wird ein Core-Dump gemacht.
    e.
    ```
    wait(&status);
    if (WIFEXITED(status))
        printf("Child exits with status    %d\n", WEXITSTATUS(status));
    if (WIFSIGNALED(status)) {
        printf("Child exits on signal      %d\n", WTERMSIG(status));
        printf("Child exits with core dump %d\n", WCOREDUMP(status));
    }
    ```
8. Aufgabe
    a. Das Programm wird folgendes ausgeben:
    Hallo, I am on the way to fork now, ...lo ok: I am the child ok: I am the parent clear ? clear ?
    Tatsächliche Ausgabe:
    ```
    Hallo, I am on the way to fork now, ......look: I am the parent

    clear ?

    Hallo, I am on the way to fork now, ......look: I am the child

    clear ?
    ```
    b. Erwartete Ausgabe:

Kinderarray ccccccccc ccccccccc ccccccccc ccccccccc ccccccccc ccccccccc ccccccccc ccccccccc
Elternarray ppppppppp ppppppppp ppppppppp ppppppppp ppppppppp ppppppppp ppppppppp ppppppppp
Effektive Ausgabe:

```
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -

Kinderarray

- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
c c c c c c c c c
c c c c c c c c c
c c c c c c c c c
c c c c c c c c c

Elternarray

p p p p p p p p p
p p p p p p p p p
p p p p p p p p p
p p p p p p p p p
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
```

c. Erwartet: Mischung von Fritzlis und Mamis

```
Mami    1
Mami    2
Mami    3
Mami    4
Fritzli 1
Mami    5
Fritzli 2
Mami    6
Fritzli 3
Mami    7
Mami    8
Mami    9
Mami    10
Fritzli 4
Fritzli 5
Fritzli 6
Fritzli 7
Fritzli 8
Fritzli 9
```

```
Mami    11
Mami    12
Mami    13
Fritzli 10
Fritzli 11
Mami    14
Fritzli 12
Fritzli 13
Fritzli 14
Fritzli 15
Mami    15
```

9. Aufgabe
    a. Erwartet: Im Gegensatz zu dem Prozess wird die obere Hälfte und die untere Hälfte im gleichen Array gefüllt.
    pppppppp pppppppp pppppppp pppppppp cccccccc cccccccc cccccccc cccccccc
    Effektiv:

```
Array vor Threads

- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -

p p p p p p p p
p p p p p p p p
p p p p p p p p
p p p p p p p p
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -

p p p p p p p p
p p p p p p p p
p p p p p p p p
p p p p p p p p
c c c c c c c c
c c c c c c c c
c c c c c c c c
c c c c c c c c


... nach Threads
p p p p p p p p
p p p p p p p p
p p p p p p p p
p p p p p p p p
c c c c c c c c
c c c c c c c c
c c c c c c c c
```

c c c c c c c c

b. Die beiden Threads beanspruchen praktisch 100% der CPU und terminieren nie.
   Wenn die Threads allerdings nicht gejoint werden, terminiert das Programm am Ende der Main-Funktion.