Technische Informatik 2 | WS 2016/17 | C02
Tutor: Marc/Bingbin

Rene Engel
Dennis Jacob
Jan Schoneberg

Übungsblatt 4

1	2	3	$\sum$

Lösungsvorschlag
Abgabe: 28.11.2016

Aufgabe 1

Implementierung

```
1  #include <stddef.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string>
5  #include <sys/wait.h>
6  #include <unistd.h>
7  #include <iostream>
8  #include <signal.h>
9
10 #include <sys/types.h>
11 #include <sys/stat.h>
12 #include <cstring>
13 #include "dirent.h"
14 #include <vector>
15
16 #include "parser.h"
17
18 using namespace std;
19
20 bool bg = false;
21
22 inline bool is_in_dir(const char* p_dir, char* file_name)
23 {
24     DIR *dir;
25     struct dirent *ent;
26     if ((dir = opendir (p_dir)) != NULL) {
27         /* print all the files and directories within directory */
28         while ((ent = readdir (dir)) != NULL) {
29             if(std::string(ent->d_name,strlen(ent->d_name)) == std::string(
                    file_name,strlen(file_name)))
30             {
31                 return true;
32             }
33         }
34         closedir (dir);
35     }
36     return false;
37 }
38
39 inline int dirExists(const char *path)
40 {
41     struct stat info;
42     if(stat( path, &info ) != 0)
43         return 0;
44     else if(info.st_mode & S_IFDIR)
```

```
45          return 1;
46      else
47          return 0;
48  }
49
50  inline std::vector<std::string> split_str(std::string p_str, char regex)
51  {
52      std::vector<std::string> tmp;
53      std::string var;
54      for(int i = 0; i < p_str.length(); ++i)
55      {
56          if(p_str.at(i) == regex)
57          {
58              tmp.push_back(var);
59              var.clear();
60          }
61          else
62          {
63              var += p_str[i];
64          }
65      }
66      return tmp;
67  }
68
69  void signalHandler( int signum ) {
70     if(wait(0) < 0 && bg ) {
71       perror ("signalHandler: invalid PID of chlid process");
72     }
73     return;
74  }
75
76  int main(){
77      for (;;){
78      struct command cmd = read_command_line();
79
80      // Exits ti2sh
81      if(std::string(cmd.argv[0],strlen(cmd.argv[0])) == std::string("exit"))
82      {
83          std::cout << "Bye!" << std::endl;
84          exit(0);
85      }
86
87      // Change dir
88      if(std::string(cmd.argv[0],strlen(cmd.argv[0])) == std::string("cd"))
89      {
90          if(dirExists(cmd.argv[1]))
91          {
92              chdir(cmd.argv[1]);
93              char cwd[1024];
94              if (getcwd(cwd, sizeof(cwd)) != NULL)
95                  std::cout << cwd << std::endl;
96              continue;
97          }
98          else if(cmd.argv[1] == NULL)
99          {
100             char* home = getenv("HOME");
101             if(home != NULL && dirExists(home))
102             {
103                 chdir(home);
104                 char cwd[1024];
105                 if (getcwd(cwd, sizeof(cwd)) != NULL)
106                     std::cout << cwd << std::endl;
```

```
107                 continue;
108             }else
109             {
110                 std::cerr << "Home \"" << home << "\" does not exist!" << std::
                        endl;
111             }
112         }
113         else
114         {
115             std::cerr << "Directory: " << cmd.argv[1] << " does not exists!" <<
                    std::endl;
116             continue;
117         }
118     }
119
120     cout << "command: " << cmd.argv[0]
121          << ", background: " << (cmd.background ? "ja" : "nein") << endl;
122
123     /* Hier muesste die Ausfuehrung stehen */
124     if(cmd.background == 1) {
125         bg = true;
126     } else {
127       bg = false;
128     }
129
130     std::string bin = "/usr/bin/";
131     char* env_path = getenv("PATH");
132     bool command_found = false;
133     std::vector<std::string> paths = split_str(std::string(env_path,strlen(
            env_path)), ':');
134     for(int i = 0; i < paths.size(); i++)
135     {
136         if(is_in_dir(paths.at(i).c_str(),cmd.argv[0]))
137         {
138             command_found = true;
139             bin = paths.at(i) + "/";
140             break;
141         }
142     }
143     if(!command_found)
144     {
145         std::cerr << "Command not found!" << std::endl;
146         continue;
147     }
148     // ======================
149     std::string command = cmd.argv[0];
150     std::string pathString = bin + command;
151     const char* path = pathString.c_str();
152
153     //nach https://docs.oracle.com/cd/E19455-01/806-4750/signals-7/index.html
154     signal(SIGCHLD, signalHandler);
155     switch (fork()) {
156     case -1:
157         perror ("main: fork");
158         exit (0);
159     case 0:
160         return execv(path,cmd.argv);
161         break;
162     default:
163         //child process runs not in background
164         //parent process waits until child process finishes
165         if(cmd.background == 0) {
```

```
166                 if(wait(0) < 0) {
167                     perror ("main: invalid PID of chlid process");
168                 }
169             }
170              break;
171             }
172         }
173         return 0;
174 }
```

Für die Lösung der Aufgabe 1 wurden zuerst mehrere Hilfsfunktionen definiert. *is_in_dir* stellte fest, ob ein gegebener Name als Datei oder Ordner in dem gegebenen Pfad vorhanden ist. Dies vereinfacht das durchsuchen der in der **PATH**-Variable gegebenen Pfade. Die Methode *dirExists* prüft, ob der gegebene Pfad existiert und ein Ordner ist. Dies vereinfacht das Behandeln von Pfadwechseln. Die Methode *split_str* liefert eine Liste von *std::string*s zurück, die durch das Trennen des gegebenen Strings am *Regex* entsteht. *signalHandler* ist ein Callback für eingehende Signale.
In der *main*-Methode wird zuerst das nächste Kommando geholt und danach geprüft, ob es eine Anweisung für die Shell direkt ist. Eine solche Anweisung könnte *exit* oder *cd ¡dir¿* sein. Bei einem *exit* wird die Shell beendet. Die Anweisung cd benötigt eine etwas aufwendigere Behandlung. Zuerst wird geprüft, ob das gegebene Argument ein existierender Ordner ist. Ist dies der Fall, wechselt die Shell zu dem Ordner und gibt den jetzt aktuellen Pfad aus. Wird kein Argument gegeben, so holt sich die Shell die Umgebungsvariable für den *home*-Ordner, prüft ob er existiert und wechselt zu diesem. Ist keine solche Variable gegeben oder Fehlerhaft, wird eine entsprechende Fehlermeldung ausgegeben. Sollte der Parameter ein nicht existierender Ordner sein, wird auch dies ausgegeben. Danach wird einmal der übergebene Befehl mit der Information, ob dieser im Hintergrund ausgeführt wird, ausgegeben und eine Variable entsprechend gesetzt. Als nächstes wird geprüft, ob die, in der Umgebungsvariable **PATH** definierten Pfade, das entsprechende Programm für das eingegebene Kommando beinhalten. Dafür wird zuerst die Variable *bin* temporär auf den Pfad */usr/bin/* gesetzt. Die Umgebungsvariable wird dann in *env_path* zwischen gespeichert und danach mit *split_str* aufgeteilt. Dann iteriert eine Schleife über jeden Eintrag in der Pathvariable und sucht den Befehl. Wird der Befehl gefunden, so wird die Variable *command_found* auf *true* gesetzt und die Schleife beendet. Existiert das Programm nicht, so wird eine Fehlermeldung ausgegeben. Danach wird aus dem Pfad und dem Namen des Programms ein *char*-String erzeugt. Jetzt wird die Callback-Methode *signalHandler* für das Signal *SIGCHLD* registriert. Mit dem Systemaufruf *fork* wird ein neuer Kindprozess erzeugt. Sollte dies nicht funktioniert haben, so wird eine Fehlermeldung ausgegebenen. War der *fork*-Aufruf erfolgreich, wird der Befehl mit den restlichen Parametern ausgeführt. Wird der Kindprozess im Vordergrund ausgeführt, so wartet der Elternprozess auf dessen Beendigung. Der Signalhandler wird nicht aktiv (durch entsprechende if Bedingung) das das Kind bereits in Elternprozess „eingesammelt“ wurde. Wird der Kindprozess im Hintergrund ausgeführt wird nach dessen Terminierung das Signal *SIGCHLD* durch den Signalhandler bearbeitet und der Kindprozess *eingesammelt*

Test

Der Test der TI2 Shell erfolgte im wesentlichen manuell. Dazu wurden in der Shell (Testrechner x02) verschiedene Befehle (u.a. cd, exit, ls, echo) eingeben und die Ausgaben beobachtet. Diese waren wie erwartet, z.B. bei *echo Test* die Ausgabe *Test*. Da die Einleseroutine vorgegeben war, haben wir diese nicht in unsere Tests einbezogen. Außerdem muss die Ein-/Ausgabeumlenkung nicht getestet werden. Es bleibt also nur noch folgendes zu testen:
1. Fehlerfälle
2. starten der Prozesse im Hintergrund
zu 1.
Der Test erfolgte ebenso hauptsächlich manuell. Es wird im Fehlerfall jeweils eine entsprechende Fehlermeldung ausgegeben. Folgende Fehlerfälle haben wir identifiziert und geprüft ob entsprechende Fehlermeldungen ausgegeben werden:
ungültige Befehlen(z.b. unvollständige Befehle oder der Aufruf von Programmen, die nicht in PATH liegen), die Fehlermeldung „Command not found “
cd ohne weiteren Parameter, wenn kein gültiger Pfad in *HOME* liegt, die Meldung „Home does not exist! “
cd mit ungültigem Pfad, die Meldung „Directory does not exist! “
zu 2.
Um zu prüfen ob ein Prozess im Hintergrund gestartet wird haben wir ein kleines Skript geschrieben. Dort wird ein Prozess benötigt, der auf die Konsole schreibt. Da der *Echo* Befehl auf den x-Rechnern in */bin* liegt und dieses Verzeichnis (zumindest auf dem x02 Rechner) nicht in *PATH* war haben wird dieses Verzeichnis zunächst *PATH* hinzugefügt. Auch das Verzeichniss */usr/bin* wird *PATH* hinzugefügt, falls dies noch nicht geschehen ist. Auch dort liegen einige Programme und auf einigen Unix ähnlichen Betriebssystemen, wie openSuse, sogar die meisten Programme.

```
1  bin="/usr/bin"
2  binX="/bin"
3  PATH2=$PATH
4  paths=$(echo $PATH2 | tr ":" "\n")
5  containsbin=0
6  containsbin2=0
7
8  for p in $paths
9  do
10 if [ "$p" = "$bin" ]
11 then
12 containsbin=1
13 fi
14 done
15
16 if [ "$containsbin" -eq 1 ]
17 then
18 PATH=$PATH:/usr/bin
19 export PATH
20 fi
21
22 for q in $paths
23 do
24 if [ "$q" = "$binX" ]
25 then
26 containsbin2=1
27 fi
28 done
29
30 if [ "$containsbin2" -eq 1 ]
```

```
31 then
32 PATH=$PATH:/bin
33 export PATH
34 fi
```

Danach beginnen die eigentlichen Testfälle.

```
35
36 failCounter=0
37 testCounter=0
38
39 echo "Test gestartet" > TestResults.txt
40 echo "-----------------------------------" >> TestResults.txt
41
42 ((testCounter++))
43 echo "echo Test1 &" > Befehle
44 echo "sleep 1 " >> Befehle
45 echo "echo Test2" >> Befehle
46
47 ./ti2sh < Befehle > loc
48
49 l1=$(sed -n '3p' loc | tr -d '\n')
50 l2=$(sed -n '5p' loc | tr -d '\n')
51
52 echo "$l1 is in line 3. $l1 should be in line 3" >> TestResults.txt
53 echo "$l2 is in line 5. $l2 should be in line 5" >> TestResults.txt
54
55 if [ "$l1" != "Test1" -o "$l2" != "Test2" ]
56 then
57 ((failCounter++))
58 echo "Testfall Fehgeschlagen" >> TestResults.txt
59 fi
60 echo "-----------------------------------" >>TestResults.txt
61
62 ((testCounter++))
63 echo "echo Test1" > Befehle
64 echo "sleep 1 " >> Befehle
65 echo "echo Test2" >> Befehle
66
67 ./ti2sh < Befehle > loc
68
69 l1=$(sed -n '2p' loc | tr -d '\n')
70 l2=$(sed -n '5p' loc | tr -d '\n')
71
72 echo "$l1 is in line 2. $l1 should be in line 2" >> TestResults.txt
73 echo "$l2 is in line 5. $l2 should be in line 5" >> TestResults.txt
74
75 if [ "$l1" != "Test1" -o "$l2" != "Test2" ]
76 then
77 ((failCounter++))
78 echo "Testfall Fehgeschlagen" >> TestResults.txt
79 fi
80 echo "-----------------------------------" >> TestResults.txt
81
82 ((testCounter++))
83 echo "ech Test1" > Befehle
84
85 ./ti2sh < Befehle > loc
86
87 l1=$(sed -n '2p' loc | tr -d '\n')
88 echo "$l1" >> loc2
89 l2=$(tail -n -1 loc2 | cut -d$ -f2)
```

```
90 rm loc2
91
92 echo "line 2 is$l2. line 2 should be$l2" >> TestResults.txt
93
94 if [ "$l2" != " (bye)" ]
95 then
96 ((failCounter++))
97 echo "Testfall Fehgeschlagen" >> TestResults.txt
98 fi
99 echo "------------------------------------" >> TestResults.txt
100
101
102 echo "Test beendet" >> TestResults.txt
103 echo "$testCounter Testfälle. $failCounter Testfälle fehgeschlagen" >> TestResults
    .txt
104
105 rm loc
106 rm Befehle
```

Zunächst wird die TI2 Shell so aufgerufen, dass der Befehl *echo* im Hintergrund läuft und dann die Befehle *sleep* und nochmal *echo*. Da der erste *echo* Befehl im Hintergrund aufgerufen wurde und danach ohne Verzögerung sleep ausgeführt wird, erscheint die Ausgabe vom ersten *echo* erst nach beiden! *comand...* Ausgaben in Zeile 3.

Im nächsten Testfall wird der erste *echo* Befehl im Vordergrund gestartet und erst nachdem dieser die Ausgabe getätigt hat, wird mit sleep weiter gemacht. Deshalb befindet sich die Ausgabe in Zeile 2.
Dieses wird im Testskript automatisch ausgewertet und sowohl formatiert als auch mit sinnvollen Meldungen in die Datei *TestResults.txt* geschrieben.

```
1  Test gestartet
2  ------------------------------------
3  Test1 is in line 3. Test1 should be in line 3
4  Test2 is in line 5. Test2 should be in line 5
5  ------------------------------------
6  Test1 is in line 2. Test1 should be in line 2
7  Test2 is in line 5. Test2 should be in line 5
8  ------------------------------------
9  line 2 is (bye). line 2 should be (bye)
10 ------------------------------------
11 Test beendet
12 3 Testfälle. 0 Testfälle fehgeschlagen
```

Der letzte Testfall prüft zusätzlich zu den manuellen Tests eine Fehlerhafte Eingabe. Die TI2 Shell führt keinen Befehl aus, sondern es erscheint die Ausgabe „Command not Found“.

Aufgabe 2

Bei der folgenden Aufgabe wird stets in KiB gerechnet, da die Rechnung in Byte äquivalent aber umständlicher ist. Lediglich der k Wert ist anders, was folgende Beispielrechnung belegt.

$$10KiB = 10240Byte$$
$$log_2(10240) \approx 13,3$$
$$2^{14}Byte = 16384Byte = 16KiB = 2^4KiB$$

Weiterhin werden die Anforderungen mit Buchstaben eindeutig nach folgender Tabelle durchnummeriert.

Anforderung	Speicher (KiB)	Block (KiB)
A	10	16
B	12	16
C	3	4
D	16	16
E	1	4
F	20	32
G	2	4

In den Speicherblöcken stehen sowohl der Buchstabe der Anforderung als auch die Blockgröße in KiB. Ein kleines f bedeutet, dass der Block frei ist. Über den Speicherblöcken steht die Startadresse der jeweiligen Blöcken.

Aufgabe 2a)

Im folgenden ist der Speicher für die ersten Anforderungen dargestellt.

Anforderung	0	16	32	48
A	f64			
	f32		f32	
	f16	f16	f32	
	A16	f16	f32	
B	A16	B16	f32	
C	A16	B16	f16	f16

Im folgenden ändert sich der Speicherbereich von Adresse 0 bis 31 nicht mehr, da dieser vollständig gefüllt ist und kein Block wieder frei gegeben wird. Deshalb wird der restliche Bereich (32 bis 63) näher betrachtet.

Anforderung	32	36	40	48
C	f16			f16
	f8		f8	f16
	f4	f4	f8	f16
	C4	f4	f8	f16
D	C4	f4	f8	D16
E	C4	E4	f8	D16

Da es keinen ausreichend großen freien Block gibt, um die Anforderung F zu erfüllen, muss mehr Speicher von Betriebssystem geholt werden (brk() System Call). Der zusätzliche Bereich ist 32 KiB groß (Startadresse 64) und wird im folgenden betrachtet.
Dieser Bereich ist nun belegt und es wird wieder der Bereich von Adresse 32 bis 63 betrachtet.

Aufgabe 2b)

Jetzt werden die Blöcke G, D und C nacheinander freigegeben.

Anforderung	64
brk(F)	f32
F	F32

Anforderung	32	36	40	44	48
G	C4	E4	f8		D16
	C4	E4	f4	f4	D16
	C4	E4	G4	f4	D16

Anforderung	32	36	40	44	48
	C4	E4	G4	f4	D16
free(G)	C4	E4	f4	f4	D16
verschmelzen	C4	E4	f8		D16
free(D)	C4	E4	f8		f16
free(C)	f4	E4	f8		f16

Die Speicheranforderung von 21 KiB (32 KiB) kann nicht erfüllt werden, ohne zusätzlichen Speicher vom Betriebssystem zu holen, da es keinen ausreichend großen Block mehr gibt und keine Verschmelzung mehr möglich ist.

Aufgabe 3

> Stellt die Änderungen der Prozesshierarchie über die angegebene Zeit grafisch dar wie im Tutorium gezeigt.

Erläuterung der Befehle bzw. Prozesse

0. `bash`: startet einen **bash**-Prozess innerhalb der **Shell**. Danach läuft ein **bash**-Prozess in dem **Shell**-Fenster über dem **Shell**-Prozess.
1. `ls`: Zeigt den Inhalt d.h. Ordner und Verzeichnisse des aktuellen Verzeichnisses an.
2. `xterm &`: Öffnet neue **Shell** in seperatem Fenster des X-Window-Systems.
3. `cc baz.c &`: Kompiliert die Datei *baz.c* zu einer *a.out*-Datei.
4. `make`: Rekompiliert, wenn eine *make*-Datei existiert.
5. `ls -lR / &`: Zeigt den Inhalt d.h. Ordner und Verzeichnisse, sowie aller Unterverzeichnisse des gesamten Systems in detaillierter Form an. Durch & wird der Prozess im Hintergrund ausgeführt. Dabei kann die **Shell** des **xterm**-Fensters allerdings trotzdem nicht gentutzt werden, weil die Ausgabe von `ls` immer noch in der **Shell** erfolgt.
6. `kill -STOP 12345`: Das Programm `kill` sendet mit der Option `STOP` ein **SIGSTOP**-Signal an den Prozess des **xterm**-Fensters (**process id:** 12345). Es wird zwar eine Kopie der Signalanordungen (*Signal dispositions*) des Vaterprozesses, welche bestimmen wie sich ein Prozess verhält, wenn es ein Signal geliefert bekommt, an Kindsprozesse die mit `fork`

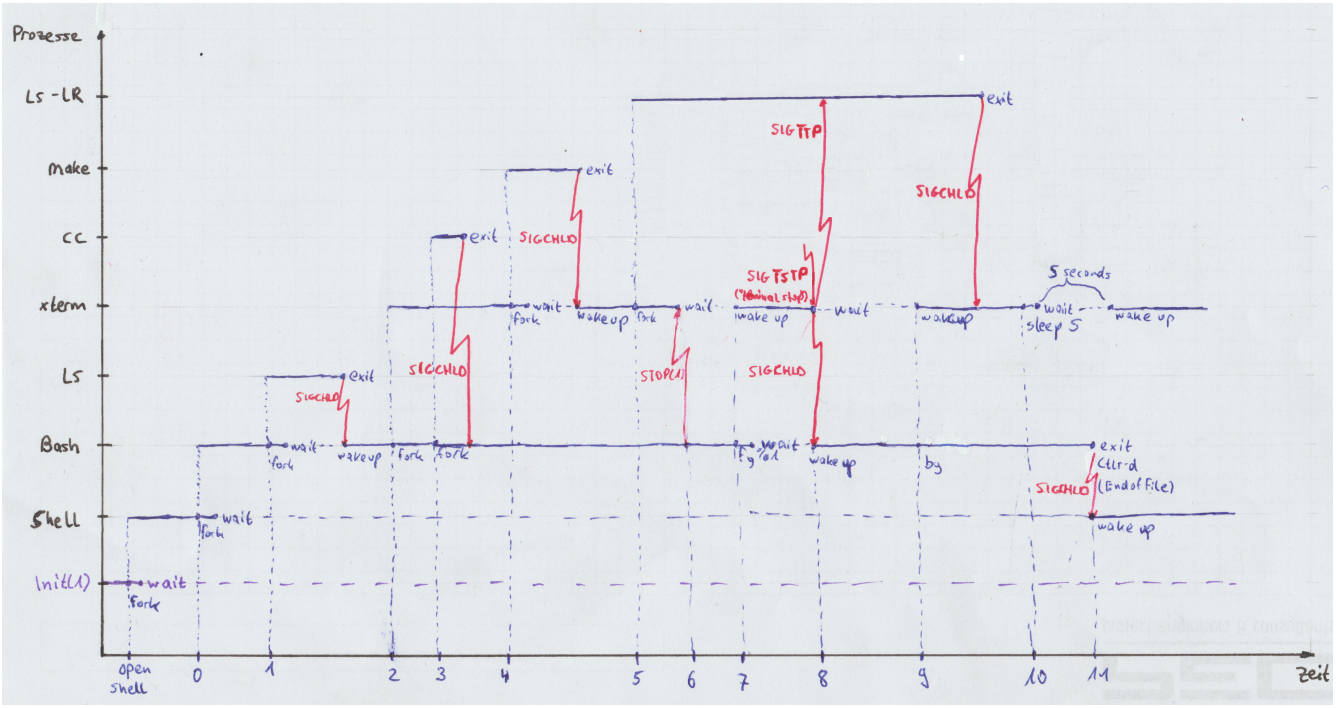

Abbildung 1: Prozessdiagramm

erstellt wurden vererbt, allerdings wird das Signal **SIGSTOP** nicht von Kindsprozessen behandelt. Deshalb wird das **SIGSTOP**-Signal, welches als Kopie gesendet wird nicht von dem Prozess `ls -lR / &` behandelt. Allerdings wird die Ausgabe d.h. der `write`-Systemaufruf gestoppt. Dieser wird erst fortgesetzt, wenn der **xterm**-Prozess fortsetzt.

7. `fg %1`: Startet den ersten Prozess (bzw. **job**) der **Shell-Job**-Tabelle im Vordergrund. Dies ist in diesem Fall der **xterm**-Prozess.

8. `Ctrl-z`: sendet ein **SIGTSTP**-Signal (*terminal stop*) an den **xterm**-Prozess der zu Zeitpunkt des Aufrufs im Vordergrund läuft. Auch hiervon wird der **ls -lR / &**-Prozess nicht beeinträchtigt.

9. `bg`: Der **xterm**-Prozess wird im Hintergrund fortgesetzt.

10. `sleep 5`: Der **xterm**-Prozess pausiert für 5 Sekunden.

11. `Ctrl-d`: Ist die Eingabe der Shell bzw. hier des **Bash**-Prozesses leer löst die Tastenkombination eine *End-Of-File*-Bedingung aus. Die Tastenkombination ist auch der Standardwert für das *End-Of-File*-Spezialkontrollzeichen (`\04`). Der Terminaltreiber leitet beim drücken der Tastenkombination die gesamte Zeile der aktuellen Eingabe weiter, sodass `read()` ausgeführt wird. Ist die aktuelle Zeile leer gibt `read()` 0 zurück. Dies signifiziert der Applikation in diesem Fall dem **Bash**-Prozess, dass die *End-Of-File*-Bedingung zutrifft und keine Daten mehr gelesen werden können. Der **Bash**-Prozess beendet sich selbst.

Wann entstehen neue Prozesse, wann terminieren sie wieder?

Neue Prozesse entstehen in der Aufgabe bei den Shell-Kommandos 0 bis 5. Alle weiteren Kommandos und Tastenkombinationen erzeugen keine neuen Prozesse.

Die Prozesse der Befehle `ls`, `cc`, `make` und `ls -lR / &` terminieren automatisch nach beendetem Programmablauf. Der **Bash**-Prozess wird durch das Zutreffen einer *End-Of-File*-Bedingung beendet.

> Wann wird welchem Prozess welches Signal gesendet (und von wem)?

Das erste Signal wird in dem Shell-Kommando 6 gesendet. `kill -STOP 12345` sendet ein **SIGSTOP**-Signal an den **xterm**-Prozess. Ein **SIGTSTP**-Signal wird durch drücken der Tastenkombination `Ctrl-z` gesendet.

> Zu welchem Prozess gehört am Ende das in Schritt 2 gestartete Terminal?

Bei Termination eine Vaterprozesses wird der Kindsprozess an den **Init(1)**-Prozess vererbt. In diesem Fall terminiert der Vaterprozess **Bash** vor dem Kindsprozess **xterm**. **xterm** wird an **Init(1)** vererbt.