Technische Informatik 2 | WS 2016/17 | C-01
Tutor: Marc/Bingbin

Habermann, Paul
Köster, Joschka
Rohde, Florian

Übungsblatt 4

1	2	3	4	5	$\sum$

Lösungsvorschlag
Abgabe: 28.11.2016

Aufgabe 1 Aufgabe

Wir haben die Vorgabe um einen Debug-Modus erweitert, welcher lediglich einige Konsolenausgaben ermöglicht.

Listing 1: ti2sh.c

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <sys/wait.h>
#include <unistd.h>
#include <iostream>
#include <string.h>
#include "parser.h"

using namespace std;

int main(){
   bool debug = false;


  for (;;){
    struct command cmd = read_command_line();


    cout << "command: " << cmd.argv[0]
        << ", background: " << (cmd.background ? "ja" : "nein") << endl;



    pid_t pid = fork();
    bool foldermode = false;
```

Weiterhin verfügt unser Programm über einen `Foldermode`, welcher später im Falle der Eingabe eines absoluten Pfades aktiviert wird.

Listing 2: ti2sh.c

```
31      if (pid < 0) {
32
33         perror("!! ERROR while forking child process !!\n");
34         std::_Exit(-1);
35
36      } else {
37
38         if (pid == 0) { //I'm a child
39
40         std::string program = cmd.argv[0];
41         std::size_t found = program.find("/");
42
43         char * environment = (char*)"";
44         char * current = (char*)"";
45
46         if (found!=std::string::npos) {
47            foldermode=true;
48            } else {
49            environment = getenv("PATH");
50            current = strtok(environment, ":");
51            }
```

In dieser IF-Bedingung prüfen wir nun, da wir einen Fork erstellt haben, ob es sich bei unserem aktuell ausgeführten Programm um das Kind oder den Vater handelt, da beide ja den gleichen Code durchlaufen. Weiterhin initialisieren wir unsere Umgebundsvariablen und durchsuchen das erste Argument nach einen Slash. In diesem Fall gehen wir davon aus, dass ein absoluter Pfad eingegeben wurde, da andernfalls keiner vorhanden sein dürfte. Wir aktivieren also den `foldermode` bzw. setzen alternativ unsere Umgebungsvariable mit der Rückgabe von getenv("PATH") und lassen diese (Liste von Pfaden) am Aufspalten, indem wir den Doppelpunkt als Token für strtok verwenden.

Listing 3: ti2sh.c

```
55         while(current != NULL)
56          {
57            if (debug) cout << ">>>>>>>>>>>>>>> cDEBUG: Hello, I'm the child!
               My PID is " << pid << endl;
58
59            const char* path;
60            std::string target = cmd.argv[0];
61            std::string slash = "/";
62            if (!foldermode) {
63               std::string konkatenation = current + slash + target;
64               path = konkatenation.c_str();
65               } else {
66              path = target.c_str();
67               }
```

Hier wird der Pfad, unter dem wir nach dem Programm suchen, initialisiert. Wenn aber der `foldermode` aktiv ist, so wird der Pfad absolut auf die Eingabe gesetzt, andernfalls laufen wir in einer While-Schleife durch alle Pfade, die wir von getenv erhalten haben.

Listing 4: ti2sh.c

```
70              if(debug) cout << ">>>>>>>>>>>>>>> cDEBUG: Foldermode is " <<
                  foldermode << endl;
71
72              if(debug) cout << ">>>>>>>>>>>>>>> cDEBUG: Looking for " << path
                  << endl;
73
74              current = strtok(NULL,":");
75
76              int ret;
77
78              ret = execv(path, cmd.argv);
79              if (debug) cout << ">>>>>>>>>>>>>>> cDEBUG: ret= "<< ret << endl;
80              if (ret == -1) { //Execute succeeded
81                 if (debug) cout << " >>>>>>>>>>>>>>> cDebug: Execution failed: "
                      << path << endl;
82                continue;
83              } else {
84                 if (debug) cout << " >>>>>>>>>>>>>>> cDebug: Execution succeeded:
                      " << path << endl;
85                 break;
86              }
87             }
88           } else { // I'm the parent
89              if (debug) cout << ">>>>>>>> DEBUG: I'm the parent! Child's PID is "
                  << pid << endl;
90           if (cmd.background) {
91
92              while(wait(0) != pid) {}
93
94              }
95
96           }
97
98
99
100  }
101 }
102 }
```

Im letzten Teil des Programmes iterieren wir weiter über jeden einzelnen Pfad und versuchen, das Programm unter diesem auszuführen. Wir rufen also einfach den Namen der Eingabe auf jedem der Environment-Pfade auf, bis wir keine Fehlermeldung (Return-Wert von `-1`) mehr erhalten. In diesem Falle beenden wir die while-Schleife, da wir davon ausgehen können, das Programm gefunden zu haben. Wenn das Programm im Hintergrund ausgeführt werden soll (`cmd.background==true`), warten wir, bis uns wait(0) mitteilt, dass die PID, die den Kindprozess repräsentiert, beendet wurde.

Aufgabe 1.a Tests

Aufgabe 1.a.1 Typische Eingabesyntax eines Unix-Shellkommandos

Wir haben verschiedene, typische Shellkommandos zum Test eingegeben.

```
ti2sh$ ls -la
command: ls, background: nein
total 150
drwxr-xr-x 2 frohde stud 11 Nov 27 12:13 .
drwxr-xr-x 4 frohde stud 9 Nov 27 10:58 ..
-rw-r--r-- 1 frohde stud 15449 Nov 27 10:58 Aufgabe3.png
-rw-r--r-- 1 frohde stud 43496 Nov 27 10:58 Aufgabe3.vsdx
-rw-r--r-- 1 frohde stud 206 Nov 16 10:48 Makefile
-rw-r--r-- 1 frohde stud 160 Nov 16 10:48 parser.h
-rw-r--r-- 1 frohde stud 1029 Nov 16 10:48 r.l
-rw-r--r-- 1 frohde stud 26808 Nov 24 18:20 r.o
-rwxr-xr-x 1 frohde stud 67339 Nov 27 12:12 ti2sh
-rw-r--r-- 1 frohde stud 2844 Nov 27 12:12 ti2sh.cc
-rw-r--r-- 1 frohde stud 72176 Nov 27 12:12 ti2sh.o
```

```
ti2sh$ ls -la &
command: ls, background: ja
total 150
drwxr-xr-x 2 frohde stud 11 Nov 27 12:13 .
drwxr-xr-x 4 frohde stud 9 Nov 27 10:58 ..
-rw-r--r-- 1 frohde stud 15449 Nov 27 10:58 Aufgabe3.png
-rw-r--r-- 1 frohde stud 43496 Nov 27 10:58 Aufgabe3.vsdx
-rw-r--r-- 1 frohde stud 206 Nov 16 10:48 Makefile
-rw-r--r-- 1 frohde stud 160 Nov 16 10:48 parser.h
-rw-r--r-- 1 frohde stud 1029 Nov 16 10:48 r.l
-rw-r--r-- 1 frohde stud 26808 Nov 24 18:20 r.o
-rwxr-xr-x 1 frohde stud 67339 Nov 27 12:12 ti2sh
-rw-r--r-- 1 frohde stud 2844 Nov 27 12:12 ti2sh.cc
-rw-r--r-- 1 frohde stud 72176 Nov 27 12:12 ti2sh.o
ti2sh$
```

```
ti2sh$ whoami &
command: whoami, background: ja
frohde
ti2sh$
```

```
ti2sh$ whoami
command: whoami, background: nein
ti2sh$ frohde
```

```
ti2sh$ ping google.de
command: ping, background: nein
ti2sh$ PING google.de (172.217.20.3) 56(84) bytes of data.
64 bytes from ham02s13-in-f3.1e100.net (172.217.20.3): icmp_req=1 ttl=57 time=6.08
 ms
64 bytes from ham02s13-in-f3.1e100.net (172.217.20.3): icmp_req=2 ttl=57 time=6.07
 ms
64 bytes from ham02s13-in-f3.1e100.net (172.217.20.3): icmp_req=3 ttl=57 time=6.02
 ms
64 bytes from ham02s13-in-f3.1e100.net (172.217.20.3): icmp_req=4 ttl=57 time=6.08
 ms
64 bytes from ham02s13-in-f3.1e100.net (172.217.20.3): icmp_req=5 ttl=57 time=6.04
 ms
64 bytes from ham02s13-in-f3.1e100.net (172.217.20.3): icmp_req=6 ttl=57 time=6.07
 ms
64 bytes from ham02s13-in-f3.1e100.net (172.217.20.3): icmp_req=7 ttl=57 time=6.07
 ms
64 bytes from ham02s13-in-f3.1e100.net (172.217.20.3): icmp_req=8 ttl=57 time=6.07
 ms
64 bytes from ham02s13-in-f3.1e100.net (172.217.20.3): icmp_req=9 ttl=57 time=6.08
 ms
^C
--- google.de ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8010ms
rtt min/avg/max/mdev = 6.023/6.068/6.084/0.105 ms
```

Aufgabe 1.a.2 "Mit dem Kopf über die Tastatur"

Die Eingabe beliebiger, schwachsinniger Kommandos erzeugt keine Rückgabe, bei aktivierten Debug-Mode sieht man, dass die Ausführung fehlschlägt.

```
ti2sh$ blafasel
command: blafasel, background: nein
ti2sh$ ti2sh$ bla /fasel
command: bla, background: nein
ti2sh$ ti2sh$
```

Aufgabe 1.a.3 Ein-/Ausgabeumlenkung

...war nicht zu implementieren:

```
ti2sh$ date > bullshit
ti2sh: Ein-/Ausgabe-Umlenkung nicht implementiert!
ti2sh$ bullshit | date
ti2sh: Ein-/Ausgabe-Umlenkung nicht implementiert!
```

Aufgabe 1.a.4 Absoluter Pfad

Unser `foldermode` funktioniert... :-)

```
ti2sh$ /bin/ls -la
command: /bin/ls, background: nein
ti2sh$ total 150
drwxr-xr-x 2 frohde stud 11 Nov 27 12:13 .
drwxr-xr-x 4 frohde stud 9 Nov 27 10:58 ..
-rw-r--r-- 1 frohde stud 15449 Nov 27 10:58 Aufgabe3.png
-rw-r--r-- 1 frohde stud 43496 Nov 27 10:58 Aufgabe3.vsdx
-rw-r--r-- 1 frohde stud 206 Nov 16 10:48 Makefile
-rw-r--r-- 1 frohde stud 160 Nov 16 10:48 parser.h
-rw-r--r-- 1 frohde stud 1029 Nov 16 10:48 r.l
-rw-r--r-- 1 frohde stud 26808 Nov 24 18:20 r.o
-rwxr-xr-x 1 frohde stud 67339 Nov 27 12:12 ti2sh
-rw-r--r-- 1 frohde stud 2844 Nov 27 12:12 ti2sh.cc
-rw-r--r-- 1 frohde stud 72176 Nov 27 12:12 ti2sh.o

ti2sh$
```

Da wir aber nur das 1. Element der Eingabe auf Slashes überprüfen, sollten die Parameter eines Programmes diese enthalten dürfen:

```
ti2sh$ ls /var/tmp
command: ls, background: nein
auto_logoff.sh elvis13.ses elvis18.ses elvis22.ses elvis27.ses elvis31.ses elvis6.
  ses libcoap
elvis1.ses elvis14.ses elvis19.ses elvis23.ses elvis28.ses elvis32.ses elvis7.ses
elvis10.ses elvis15.ses elvis2.ses elvis24.ses elvis29.ses elvis33.ses elvis8.ses
elvis11.ses elvis16.ses elvis20.ses elvis25.ses elvis3.ses elvis4.ses elvis9.ses
elvis12.ses elvis17.ses elvis21.ses elvis26.ses elvis30.ses elvis5.ses kdecache-
  hen_mai
```

Aufgabe 1.a.5 Produktivität

Es ist problemlos möglich, einen `nano` zu starten und irgendeine Datei zu schreiben. Jedoch ist uns aufgefallen, dass man Ctrl-X zweimal drücken muss, um `nano` zu beenden.

```
ti2sh$ nano ti2rockt.md
command: nano, background: nein
                                  [GNU nano 2.2.6 File:ti2rockt.md]
^X
ti2sh$
```

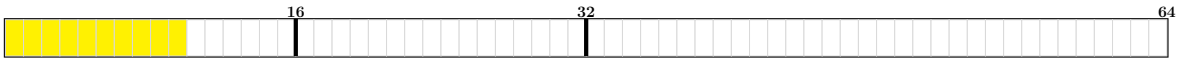
Aufgabe 2 Aufgabe

Aufgabe 2.a

Im Ausgangszustand ist der Speicher ein großer 64 KiB breiter Speicherblock.

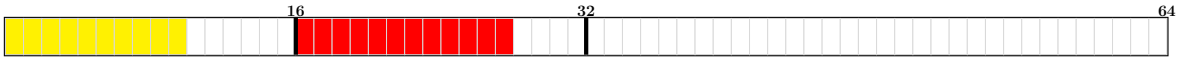
Aufgabe 2.a.1

Es folgt eine 10 KiB große Anforderung. Hierfür wird kein 64 KiB breiter Block benötigt, daher wird der Speicherblock in der Mitte geteilt, sodass zwei 32 KiB breite Blöcke entstehen. Danach wird der linke 32 KiB breite Block erneut zweigeteilt, sodass zwei 16 KiB breite Blöcke entstehen. In dem linken dieser beiden 16 KiB Blöcke wird nun das 10 KiB breite Wort abgelegt.



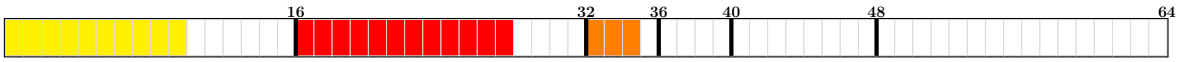
Aufgabe 2.a.2

Es folgt nun eine 12 KiB Anforderung. Diese wird in dem zweiten, 16 KiB breiten Block abgelegt.



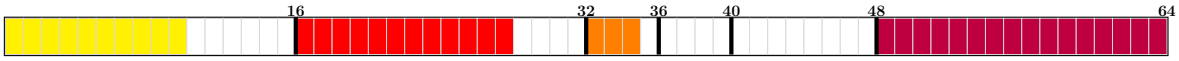
Aufgabe 2.a.3

Für die folgende 3 KiB Anforderung ist etwas mehr Aufwand notwendig. Der noch freie 32 KiB Block wird ebenfalls nach o.g. Muster unterteilt, sodass zwei 4 KiB, ein 8 KiB und ein 16 KiB Block entstehen. Das 3 KiB breite Wort wird in den linken der beiden 4 KiB breiten Blöcke geschrieben.



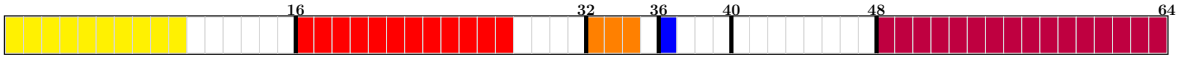
Aufgabe 2.a.4

Die folgende Anforderung von 16 KiB wird direkt in den noch freien 16 KiB Block geschrieben.



Aufgabe 2.a.5

Die nun folgende 1 KiB große Anforderung wird in den noch freien 4 KiB Block geschrieben.

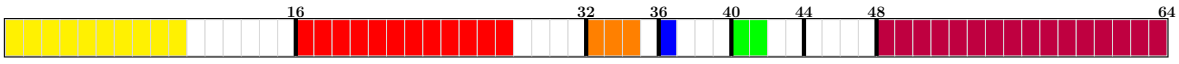


Aufgabe 2.a.6

Es passieren keine Änderungen. Die 20 KiB breite Anforderung kann in Ermangelung eines ausreichend großen Blockes nicht erfüllt werden. Die Anforderung wird somit abgelehnt.

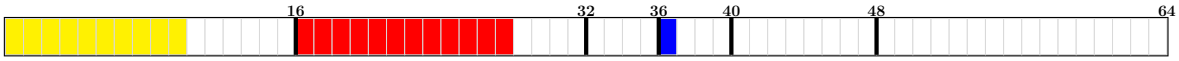
Aufgabe 2.a.7

Für die letzte Anforderung von 2 KiB muss noch einmal etwas Aufwand betrieben werden. Der noch vorhandene 8 KiB breite Block wird in zwei 4 KiB breite Blöcke unterteilt, erst danach wird das 2 KiB breite Wort in die linke der beiden neuen Blöcke geschrieben.



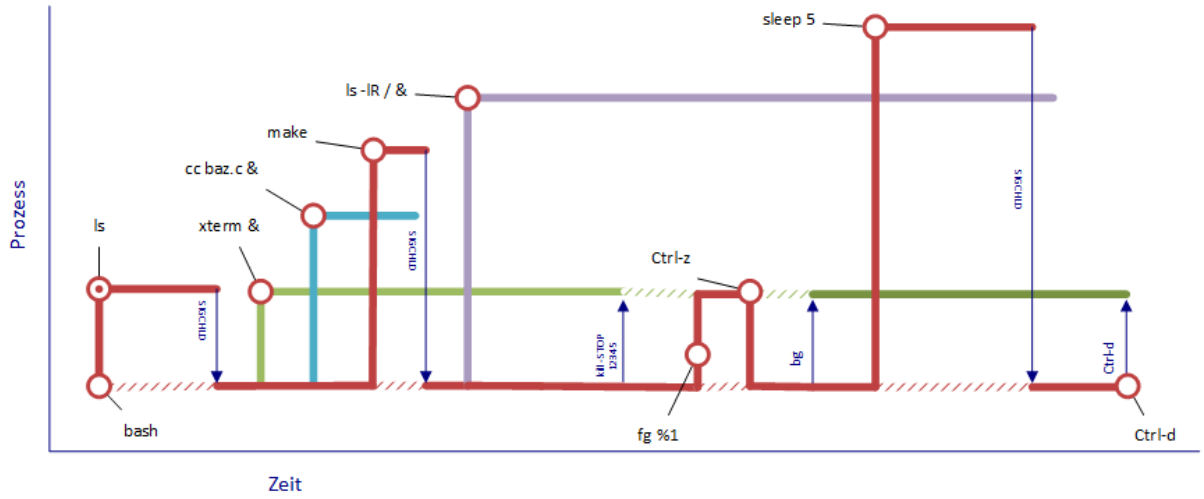
Aufgabe 2.b

Nach dem Freigeben der Blöcke mit den Worten der Breite 2 KiB, 16 KiB und 3 KiB sieht der Speicher wie folgt aus.



Man kann deutlich sehen, dass es keinen freien Speicherblock mit einer Größe von mehr als 21 KiB gibt, weshalb die Anforderung von 21 KiB nicht erfüllt werden kann.

Aufgabe 3 Aufgabe



Unsere Bash starten zuerst `ls` im Vordergrund und wartet auf dessen Terminierung. Sobald `ls` terminiert, sendet es SIGCHILD an die bash, welche dadurch wieder aktiv läuft.

Das `xterm` wird im Hintergrund gestartet und läuft parallel zur bash.
`cc baz.c &` ist ein symbolic link, der auf einen symbolic link zeigt, der auf den `gcc`-Compiler zeigt. Dieser führt also die Kompilierung im Hintergrund aus und terminiert dann.

`make` führt ein Makefile aus und beendet sich danach. `bash` erhält hier wieder ein SIGCHILD und geht somit wieder "in Betrieb".
`ls -lR / &` wird wieder im Hintergrund ausgeführt und terminiert nach der Ausgabe des Root-Verzeichnisses. Der Aufruf erfolgt hier rekursiv, wir gehen daher davon aus, dass es sehr lange dauert, bis der Prozess terminiert. Daher haben wir hier mit einer Terminierung kurz vor Ende der Laufzeit "gerechnet".

Das Signal STOP wird über `kill -STOP 12345` an unsere imaginäre Prozess-ID von `xterm` gesendet, welches daraufhin angehalten wird. `fg %1` holt es wieder in den Vordergrund und führt es fort; `Ctrl-z` ist äquivalent zum vorherigen Signal. Durch `bg` wird der Prozess, der nun im Hintergrund gestoppt ist, wieder angestossen. Der Befehl `sleep 5` lässt die bash warten, bis die 5 Sekunden um sind, und sich `sleep` mit einem SIGCHILD abmeldet.
Schließlich sendet `Ctrl-d` ein End-of-File-Signal, wodurch sich die `bash` und `xterm` beenden.