Technische Universität Berlin
Fachgebiet
Distributed and Operating Systems

Sommersemester 2025

Aufgabenblatt 1

zu – Systemprogrammierung –

Prof. Dr. Odej Kao

Aufgabe 1.1: Vorbereitung (Tafelübung)

Die Hausaufgaben und auch einige Übungsaufgaben im Tutorium sollen in der Programmiersprache C bearbeitet werden. Damit die Programme auf Ihren Computern genauso funktionieren wie auf den Computern der Prüfer ist es sinnvoll, dass alle das gleiche Betriebssystem nutzen. Wir nutzen daher die Linux-Distribution Ubuntu, die Sie kostenlos nutzen können. Richten Sie sich rechtzeitig vor der ersten Hausaufgabenabgabe Ihre Entwicklugsumgebung ein und machen Sie sich mit dieser vertraut. Folgende Software wird dazu benötigt:

- OS: Ubuntu. Dieses können Sie auf verschiedene Weisen nutzen
  - Unter Windows per WSL[1]
  - Auf Rechnern im PC-Pool in der Uni (Zugriff auch per SSH möglich)[2]
  - Auf eigener Hardware[3] oder in virtueller Maschine[4]
- Texteditor (z.B. VS Code, vim, sublime, emacs, helix,...) oder IDE (z.B. CLion). Wir empfehlen die Nutzung eines Editors gegenüber der Nutzung von CLion, da wir Ihnen dabei einen besseren Support liefern können. Es ist im zeitlichen Rahmen von Tutorien nicht immer möglich auf spezifische Probleme im Umgang mit CLion einzugehen.
- Compiler: clang
- Weitere Programme: gdb (Debugger), valgrind (Memorychecker), make (build-Tool)

Unter Ubuntu können Sie diese Tools mit dem Paketmanager `apt` im Terminal mit einem Befehl installieren: `sudo apt install clang gdb valgrind make`. Sie können sich auch das Videotutorial[5] ansehen, in dem all diese Schritte erklärt werden.

[1] `https://learn.microsoft.com/de-de/windows/wsl/install`
[2] `https://wiki.freitagsrunde.org/SSH`
[3] `https://wiki.ubuntuusers.de/Installation/`
[4] `https://kofler.info/ubuntu-22-04-in-virtualbox-7-unter-windows/`
[5] `https://youtube.com/playlist?list=PL5FRnzOULdYfzyeoaKFJmfeHJvwIwqK0a`

Aufgabe 1.2: Übungsaufgaben für C (Tafelübung)

a) Legen Sie mit dynamischem Speicher eine Variable an. Geben Sie die Speicheradresse der Variable sowohl im üblichen Hex-Format, als auch als Dezimalzahl aus.

b) Deklarieren Sie auf normalem Weg eine Variable. Geben Sie wieder die Speicheradresse im Hex-Format und als Dezimalzahl aus.

c) Erklären Sie, inwiefern und warum sich die Adressen unterscheiden. Sind die Adressen bei jeder Programmausführung gleich?

d) Betrachten Sie den folgenden Quellcode:

```
#include <stdio.h>

void upgrade(int meinezahl) {
    meinezahl = 18;
}

int main() {
    int meinezahl = 5;
    upgrade(meinezahl);
    printf("%d\n", meinezahl);
}
```

Welche Zahl wird ausgegeben? Warum?

e) Betrachten Sie nun folgenden Quellcode:

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

void upgrade(int *meinezahl) {
    *meinezahl = 18;
}

int main() {
    int *meinezahl = malloc(sizeof(int));
    *meinezahl = 5;
    upgrade(meinezahl);
    printf("%d\n", *meinezahl);
}
```

Welche Zahl wird jetzt ausgegeben? Warum?

f) Betrachten Sie nun folgenden Quellcode:

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

void upgrade(int *meinezahl) {
    meinezahl = 18;
}

int main() {
    int *meinezahl = malloc(sizeof(int));
    *meinezahl = 5;
    upgrade(meinezahl);
    printf("%d\n", *meinezahl);
}
```

Welche Zahl wird jetzt ausgegeben? Warum?

g) Betrachten Sie folgenden Quellcode:

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

typedef struct mystruct {
    int zahl1;
    int zahl2;
    int zahl3;
} mystruct;

int main() {


    mystruct test1;
    mystruct *test2 = malloc(sizeof(mystruct));

    int t1 = sizeof(mystruct);
    int t2 = sizeof(test1);
    int t3 = sizeof(test2);

    printf("t1: %d\nt2: %d\nt3: %d\n", t1, t2, t3);
}
```

Welchen Wert nehmen t1, t2 und t3 an? Warum? Zu welchen Fehlern kann das führen?

h) Sie schreiben folgenden Code. Ihr Programm stürzt jedoch ab. Was ist passiert?

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *meinezahl;
    meinezahl = 60412;

    printf("%d\n", *meinezahl);
}
```

i) Betrachten Sie folgenden Code. Welche Werte nehmen die vier Zahlen an?

```c
#include <stdio.h>
#include <stdlib.h>

int main() {

    int my_array[] = {3, 17, 42, 4, -9};
    int zahl1 = my_array[1];
    int zahl2 = *my_array;
    int zahl3 = *(my_array + 2);
    int zahl4 = *my_array + 2;

    printf("Zahl 1: %d\n Zahl 2: %d\n Zahl 3: %d\n Zahl 4: %d\n",
        zahl1, zahl2, zahl3, zahl4);
}
```

Lösung:

Alle Quellcodes findet Ihr auch im Verzeichnis `resources/code`. Ihr könnt diese einfach mit `make run_x` kompilieren und ausführen. Also beispielsweise `make run_a` um Aufgabe 1.1.a auszuführen.

a) Quellcode:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *meinezahl = malloc(sizeof(int));
    printf("%p\n", meinezahl);

    unsigned long adresse = (unsigned long) meinezahl;
    printf("%lu\n", adresse);
}
```

b) Quellcode:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int meinezahl;
    printf("%p\n", &meinezahl);

    unsigned long adresse = (unsigned long) &meinezahl;
    printf("%lu\n", adresse);
}
```

c) Stack vs. Heap, verschiedene Speicherbereiche. Adressen sind immer anders.

d) 5, weil call-by-value

e) 18, weil call-by-reference

f) Wieder 5, obwohl call-by-reference. Es wird der Pointer manipuliert, nicht sein Inhalt geändert, da das Sternchen fehlt. Tatsächlich kommt es hier auch auf den genutzten Compiler an. *clang* gibt einen Error aus und kompiliert das Programm nicht, während *gcc* nur eine Warnung ausgibt. (geteste mit *clang* Version 15.0.7 und *gcc* Version 12.2.1)

g) t1 ist 12, da sizeof() die Größe des Datentyps (=das struct mystruct) zurückgibt. t2 ist auch 12, weil sizeof() auf einer Variablen mit dem Datentyp mystruct aufgerufen wird. t3 ist aber 8 (oder 4 auf 32 Bit Systemen), da test2 ein Pointer ist, und die Größe des Datentyps „Pointer“ bestimmt wird. Mit Sternchen wäre es aber wieder 12.

h) Segmentation Fault. Pointer wurde manipuliert, anstatt dass eine Zahl in den Speicher geschrieben wurde. Es folgt ein illegaler Speicherzugriff. Beachte auch hier den Unterschied zwischen verschiedenen Compilern (siehe oben).

i) zahl1: 17, da in C Indizes mit 0 beginnen.
zahl2: 3, da der Pointer auf den Array dem Eintrag an Index 0 entspricht.
zahl3: 42, da wir den Pointer um 2 int-Größen verschieben.
zahl4: 5, da wir diesmal auf den Wert, der am Pointer liegt, 2 aufaddieren.

Aufgabe 1.3: Parallelisierung I (Tafelübung)

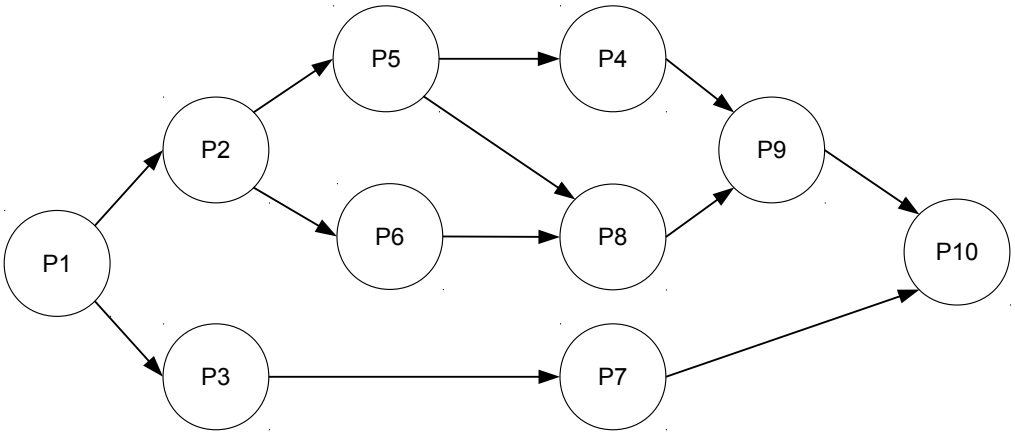


Abbildung 1: Abhängigkeitsgraph

Wie unterscheidet sich die Herangehensweise von parbegin/parend und fork/join? Gegeben sei oben stehender Abhängigkeitsgraph. Setzen Sie diesen mithilfe der aus der Vorlesung bekannten Befehle fork/join und parbegin/parend in Pseudocode um.

Gehen Sie dann wieder den umgekehrten Weg und zeichnen Sie den Abhängigkeitsgraphen aus dem Pseudocode. Haben sich zusätzliche Abhängigkeiten gebildet?

Lösung:

- **parbegin/parend** einfache Blockstruktur zur Darstellung von Parallelität. parend wartet auf die Beendigung *aller* vorherigen Prozesse.
- **fork/join** fork erstellt dynamisch einen neuen Child Prozess (der parallel ausgeführt werden kann). Mit join kann man genauer steuern, wann und auf welche Prozesse gewartet werden soll. Ermöglicht komplexere Abhängigkeiten.

```
parbegin/parend:            fork/join:
----------------            -----------

p1;                         p1
parbegin                    fork a
    begin                   p3
        p2;                 p7
        parbegin            join a
            p5;             p10
            p6;
        parend              a:
                            p2
        parbegin            fork p6
```

```
            p4;             p5
            p8;             fork p4
        parend              join p6
        p9;                 p8
    end                     join p4
    begin                   p9
        p3;                 end
        p7;
    end
parend
p10;
```

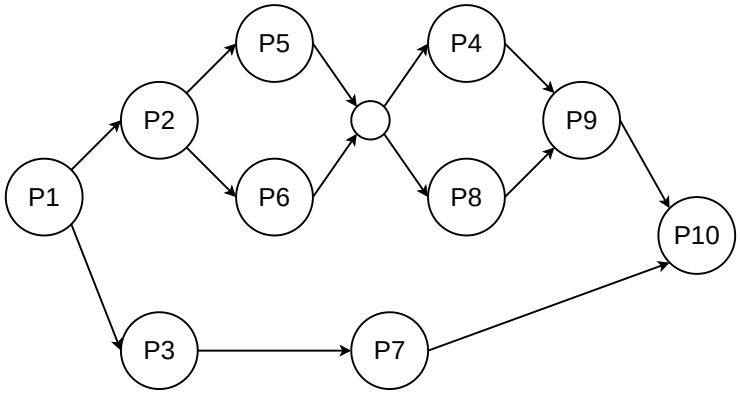


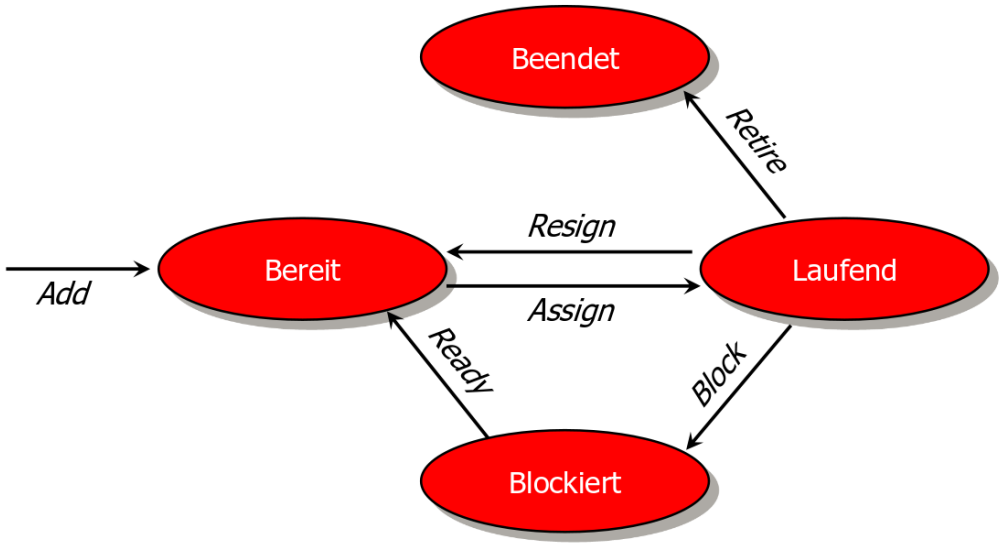
Abbildung 2: Abhängigkeitsgraph nach Rücktransformation von parbegin/parend

- Es hat sich nach der Rücktransformation tatsächlich eine neue Abhängigkeit ergeben. P4 muss nun auch auf P6 warten, was im Originalgraph nicht der Fall war.
- Bei der Rücktransformation von fork/join bleibt der Graph genau so erhalten.

Aufgabe 1.4: Prozessmanagement (Tafelübung)

Benennen Sie die möglichen Zustände eines Prozesses und Skizzieren Sie die Übergange.

Lösung:



- **Laufend** Prozess besitzt physikalischen Prozessor und wird ausgeführt
- **Bereit** Prozess hat alle BM und wartet auf Zuteilung eines Prozessors
- **Blockiert** Prozess wartet auf Erfüllung einer Bedingung
- **Beendet** Prozess hat alle Berechnungen beendet und gibt BM zurück

Übergänge:

- **Add** Ein neu erzeugter Prozess wird in die Klasse Bereit aufgenommen
- **Assign** Infolge des Kontextwechsels wird der Prozessor zugeteilt
- **Block** Aufruf einer blockierenden E/A-Operation oder Synchronisation bewirkt, dass der Prozessor entzogen wird
- **Ready** Nach Beendigung der blockierenden Operation wartet der Prozess auf erneute Zuteilung des Prozessors
- **Resign** Einem laufenden Prozess wird der Prozessor – aufgrund eines Timer-Interrupts, z.B. Zeitscheibe abgelaufen – entzogen, oder er gibt den Prozessor freiwillig ab
- **Retire** Der laufende Prozess terminiert und gibt alle Ressourcen wieder frei

Aufgabe 1.5: Prozessormodi (Selbststudium)

a) Welche Prozessormodi existieren? Nennen Sie zwei privilegierte Modi und einen nicht privilegierten Modus.

b) Worin unterscheiden sich nicht privilegierte und privilegierte Modi (Rechte und Anwendung)? Nennen Sie 3 Unterschiede.

c) Wie wird zwischen privilegierten und nicht privilegierten Modus gewechselt? Nennen Sie drei Beispiele zum Wechseln in einen privilegierten Modus und ein Beispiel zum Wechseln in den unprivilegierten Modus.

Lösung:

a)
- Prozessormodi: *user*, *system* und *supervisor mode*. Andere Modi auch möglich (*interrupt*, *abort*, *undefined mode*).

b)
- privilegiert: Voller Instruktionssatz ausführbar, nicht privilegiert: Eingeschränkter Instruktionssatz
- privilegiert; Alle Register nutzbar, nicht privilegiert: Einige Register gesperrt
- privilegiert: i.d.R. für das Betriebssystem, nicht privilegiert: i.d.R für Benutzerprogramme (z.B. Webbrowser)

c)
- unprivilegiert $>$ privilegiert: Interrupts, Fehler (Zugriffsfehler, mathematischer Fehler), Syscall (svc)
- privilegiert $>$ unprivilegiert: jederzeit möglich (z.B. durch Setzen von Prozessorstatusbits)

Aufgabe 1.6: Interrupts (Selbststudium)

a) Was ist ein Hardware-Interrupt? Wie reagiert das Betriebssystem und der Prozessor auf einen Hardware-Interrupt?

b) Wie erkennt der Prozessor, dass ein Hardware-Interrupt vorliegt?

c) Nennen Sie zwei Beispiele, wie Hardware-Interrupts entstehen können.

d) Wie unterscheiden sich sequentielle Unterbrechungsbehandlung und geschachtelte Unterbrechungsbehandlung? Was passiert, wenn ein Interrupt eintritt, während noch ein vorheriger Interrupt behandelt wird?

Lösung:

a)
- Signal, welches den Prozessor/das Betriebssystem über Ereignisse informiert. Erfordert schnelle Reaktion des Systems. Kontrollfluss wird unterbrochen, wechseln zum Interrupthandler.

b) • Der Prozessor liest die Interruptleitung (es können mehrere Leitungen vorhanden sein). Liegt Spannung an, so ist ein Interrupt zu behandeln.

c) • Viele Beispiele möglich: Interrupt zum Ende einer Übertragung. Andere Beispiele: PS2 Keyboard, Timer-Interrupt, etc.

d) • sequentiell: Abarbeitung der Interrupts nach Auftrittsreihenfolge. Neuer Interrupt muss immer auf die Abarbeitung des momentanen Interrupts warten

• geschachtelte: Interrupt-Handler können von neuen Interrupts unterbrochen werden. Der noch laufende Interrupt-Handler wird von dem neuen Interrupt unterbrochen

Aufgabe 1.7: Parallelisierung II (Selbststudium)

Gegeben ist das folgende nicht-parallele C-Programm. Die Funktionen jobA, jobB, ..., jobK wurden zuvor im Programm implementiert und enthalten länger laufende Berechnungen.

```
int main(void)
{
    int a, b, c, d, e, f, g, h, i, j, k;
    a = jobA();
    b = jobB(a);
    c = jobC(a);
    d = jobD(c);
    e = jobE();
    f = jobF(c, e);
    g = jobG(d, f);
    h = jobH();
    i = jobI(g);
    j = jobJ(b, g);
    k = jobK(i, j);

    return k;
}
```

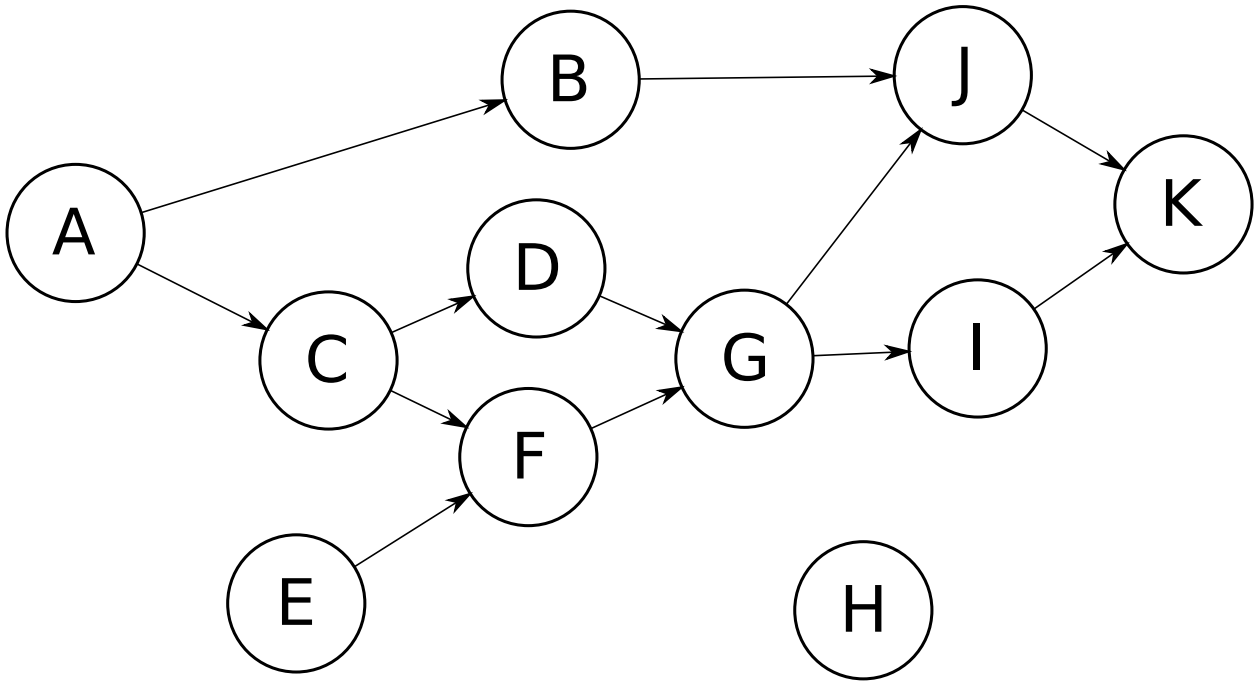
a) Zeichnen Sie einen Prozessabhängigkeitsgraphen, der die Abhängigkeiten der einzelnen Jobs darstellt. Jede aufgerufene Funktion soll dabei einem Task bzw. Prozess/Thread entsprechen.

b) Schreiben Sie basierend auf dem Graphen ein Programm in Pseudocode mit fork/join, das die Jobs möglichst effizient abarbeitet.

c) Schreiben Sie nun ebenfalls basierend auf demselben Graphen ein Programm in Pseudocode mit parbegin/parend, das die Jobs möglichst effizient abarbeitet. Untersuchen Sie Ihren Pseudocode anschließend auf womöglich neu hinzugefügte oder nicht mehr vorhandene Abhängigkeiten zwischen einzelnen Jobs.

Lösung:

a)



b) Eine Möglichkeit:

```
fork jobE
fork jobH
jobA
fork jobB
jobC
fork jobD
join jobE
jobF
join jobD
jobG
fork jobI
join jobB
jobJ
join jobI
jobK
join jobH
```

c) Es gibt mehrere Möglichkeiten dieses Programm zu schreiben, jedoch ist keine davon eine optimale Lösung. Beispielweise die Kante von jobB zu jobJ im Abhängigkeitsgraphen kann nur dargestellt werden, wenn noch eine Kante von jobB zu jobI hinzukommt.

Aufgabe 1.8: Shell (Selbststudium)

Die folgende Aufgabe ist als Vertiefungsaufgabe gedacht und somit **nicht** bewertet. Sie dient als weitere Übung zum Umgang mit C und behandelt insbesondere, begleitend zur Vorlesung, das Thema der Nebenläufigkeit.
Ziel dieser Aufgabe ist es, eine einfache Shell zu implementieren. Hierfür gibt es eine Vorlage auf der ISIS Seite. Zu bearbeiten ist die Datei `shell.c`. Die Aufgabe gliedert sich in folgende Bestandteile:

a) Ausführen von Programmen:

Das Parsen der Eingabe ist bereits vorhanden. Es soll lediglich das Ausführen eines Programmes realisiert werden. Wie bei den meisten Shells üblich, soll die Shell normalerweise auf das ausgeführte Programm warten oder es im Hintergrund laufen lassen, falls als letztes Argument ein alleinstehendes `&` übergeben wird. Dies soll in der Funktion `void execute_cmd(char * args[], int bg)` implementiert werden.
Dazu wird in der Shell ein Kindprozess geforkt und die Eingabe entsprechend im Kindprozess ausgeführt. Der Elternprozess soll nun entweder auf den Kindprozess warten oder weiterlaufen.
Hierbei kann sich an folgendem Pseudocode orientiert werden:

```
fork();
if(/*this is the child process*/){
  exec(...);
} else {
  if(/* execute in foreground */){
    waitpid(...);
  }

  /* ... */
}
```

Zudem soll Ihre Shell dem Nutzer eine Rückmeldung geben, falls das auszuführende Programm nicht gestartet werden konnte (Tippfehler o.Ä.).

Nach dem Beenden dieser Aufgabe sollte Ihre Shell bereits einfache Kommandos ausführen können. Testen Sie Ihre Shell zum Beispiel indem Sie eine Datei mit `touch test.txt` anlegen. Diese kann dann auch mit `ls` aufgelistet werden und mit `rm test.txt` gelöscht werden können.

b) Überblick über laufende Prozesse:

Die Shell soll nun einen Überblick über alle laufenden Prozesse behalten. Verwenden Sie hierfür das Array `pid_t children[]`, dessen Größe in `int children_amount` festegehalten ist. In diesem Array sollen alle pids der im Hintergrund laufenden Kindprozesse registriert werden. Eine `0` steht für einen freien Eintrag, da diese nicht als pid vorkommen wird.

Updaten Sie Ihre Funktion `execute_cmd(...)` sodass die pid eines gestarteten Prozesses nun in dem Array `children` festgehalten wird.

Implementieren Sie zudem die Funktion `int update_children()`. Diese Funktion soll einerseits die Anzahl der momentan laufenden Prozesse zurückgeben und zudem die Einträge in `children[]` von bereits beendeten Prozessen zurücksetzten.

Zum Testen kann der Wert `PIDS_SIZE` in `shell.h`, welcher die Anzahl der erlaubten Hintergrundprozesse festlegt, gerne verändert werden.

Als länger laufender Prozess kann zum Beispiel der Befehl `ping` genutzt werden, der einen anderen Computer im Netzwerk kontaktiert und misst nach welcher Zeit eine Antwort von diesem Computer kommt. Mit dem Befehel `ping tu.berlin -c 5 -i 2` werden die Server der TU 5-mal (count, -c) im Abstand von 2 Sekunden (interval, -i) angefragt. Die Zahlen und der angefragte Server können natürlich auch verändert werden.

c) Beenden der Shell:

Wenn in der Shell `exit` eingetippt wird, so wird die Funktion `void shell_exit()` ausgeführt, welche die Shell beendet. Erweitern Sie die Funktion so, dass die Shell sich erst dann beendet, wenn alle im Hintergrund laufenden Prozesse, die von der Shell aus gestartet wurden, beendet sind.

Der Nutzer soll eine Rückmeldung bekommen, auf wie viele Prozesse noch gewartet werden muss.

Wichtige Funktionen:

Folgend werden die wichtigsten Funktionen aufgelistet, welche zur Bearbeitung der Aufgabe notwendig sind. Für detailliertere Informationen zu den Funktionen können die `Manpages` benutzt werden.

a) `pid_t fork()`

Die Funktion fork gibt als Rückgabewert den pid des Kindprozesses zurück.

Weiterer Informationen: `man 2 fork`.

b) `pid_t waitpid(pid_t pid, int *wstatus, int options)`

`pid_t pid`	Der pid des Kindprozess.
`int *wstatus`	Pointer auf einen Integer in dem der Status des Kindprozess gespeichert werden kann.
`int options`	Definiert das Verhalten der Funktion.

Da es für `waitpid()` mehrere Optionen zur Verwendung gibt sind hier die beiden Möglichkeiten, die zur Bearbeitung der Aufgabe ausreichend sind, aufgelistet.

(a) `waitpid( <pid> , NULL , 0 )`
Mit `0` als Option wird die Ausführung des Prozesses welcher die Funktion aufgerufen hat so lange angehalten bis der Prozess mit dem angegebenen pid beendet wurde. Zurückgegeben wird der pid selber oder `-1` falls ein Fehler aufgetreten ist.

(b) `waitpid( <pid> , NULL , WNOHANG)`
Mit `WNOHANG` als Option wird die Ausführung des Prozesses nicht unterbrochen. Wenn der Prozess mit dem angegebenen pid noch aktiv ist, wird `0` zurückgegeben. Ansonsten wird der pid selber zurückgegeben oder `-1` falls ein Fehler aufgetreten ist.

Weitere Informationen: `man 2 wait`

c) `int execvp(const char *file, char *const argv[])`

`const char *file`	Name der auszuführenden Datei
`char *const argv[]`	Argumente

Diese Funktion ersetzt den laufenden Prozess durch einen neuen. Dadurch springt diese Funktion auch nur zurück, falls ein Fehler aufgetreten ist.

Weitere Informationen: `man 3 execvp`