

Aufgabe 1: Ankreuzfragen (22 Punkte)

1) Einfachauswahlfragen (18 Punkte)

Bei den Einfachauswahlfragen in dieser Aufgabe ist jeweils nur **eine** richtige Antwort eindeutig anzukreuzen. Auf die richtige Antwort gibt es die angegebene Punktzahl.

Wollen Sie eine Antwort korrigieren, streichen Sie bitte die falsche Antwort mit drei waagrechten Strichen durch (~~☒~~) und kreuzen die richtige an.

Lesen Sie die Frage genau, bevor Sie antworten.

a) Ausnahmesituationen bei einer Programmausführung werden in die beiden Kategorien Trap und Interrupt unterteilt. Welche der folgenden Aussagen sind zutreffend?

2 Punkte

- Ein Systemaufruf im Anwendungsprogramm ist der Kategorie Interrupt zuzuordnen.
- Ein durch einen Interrupt unterbrochenes Programm darf je nach der Interruptursache entweder abgebrochen oder fortgesetzt werden.
- Ein Trap signalisiert einen schwerwiegenden Fehler und führt deshalb immer zur Beendigung des unterbrochenen Programms.
- Die Ausführung einer Ganzzahl-Rechenoperation (z. B. Addition, Division) kann zu einem Trap führen.

b) Gegeben sei der nachfolgende C-Quelltext:

```
char a[] = "path";
char b[] = "to";
char c[] = "file";
char buf[n];
sprintf(buf, "%s/%s/%s", a, b, c);
```

2 Punkte

Welchen Wert muss n mindestens haben, damit die zusammengesetzte Zeichenkette ohne Überlauf in den Puffer `buf` geschrieben werden kann?

- 12
- 15
- 13
- 14

c) Ein Prozess wird in den Zustand *bereit* überführt. Welche Aussage passt zu diesem Vorgang?

2 Punkte

- Der Prozess hat auf Daten von der Festplatte gewartet und die Daten stehen nun zur Verfügung.
- Ein anderer Prozess blockiert sich an einem Semaphor.
- Der Prozess hat einen Seitenfehler für eine Seite, die aber noch im Hauptspeicher vorhanden ist.
- Der Prozess wartet auf eine Tastatureingabe.

d) Man unterscheidet die Begriffe Programm und Prozess. Welche der folgenden Aussagen zu diesem Thema ist richtig?

2 Punkte

- Mit Hilfe von Threads kann ein Prozess mehrere Programme gleichzeitig ausführen.
- Der UNIX-Systemaufruf `fork(2)` lädt eine Programmdatei in einen neu erzeugten Prozess.
- Der Prozess ist der statische Teil (Rechte, Speicher, etc.), das Programm der aktive Teil (Programmzähler, Register, Stack).
- Ein Programm kann durch mehrere Prozesse gleichzeitig ausgeführt werden.

e) User-Level- und Kernel-Level-Threads unterscheiden sich in verschiedenen Eigenschaften. Welche Kombination ist richtig?

2 Punkte

- Bei User-Level-Threads können anwendungsabhängig Schedulingstrategien eingesetzt werden; blockierende Systemaufrufe von Kernel-Level-Threads blockieren keine anderen Threads.
- Bei Kernel-Level-Threads ist die Schedulingstrategie meist vorgegeben; User-Level-Threads können Multiprozessoren ausnutzen.
- Kernel-Level-Threads werden sehr effizient umgeschaltet; User-Level-Threads blockieren sich bei blockierenden Systemaufrufen gegenseitig.
- Bei Kernel-Level-Threads können anwendungsabhängig Schedulingstrategien eingesetzt werden; Kernel-Level-Threads können Multiprozessoren nicht ausnutzen.

f) Was muss bei der Entwicklung von Software unbedingt beachtet werden, damit Programme nicht Opfer eines Hacker-Angriffs werden?

2 Punkte

- Der Einsatz der Bibliotheksfunktion `system(3)` muss verboten werden, da diese Funktion nicht sicher ausgeführt werden kann.
- Bei Verwendung der Programmiersprache C muss darauf geachtet werden, dass die Stringoperationen `strcpy()` und `strcat()` nur eingesetzt werden, wenn die Länge des zu übertragenden Strings noch in das Ziel-Array passt.
- Der Quellcode darf nicht herausgegeben werden, damit Schwachstellen nicht entdeckt werden können.
- Es dürfen keine vorgefertigten Bibliotheksfunktionen verwendet werden, weil deren Implementierung als nicht vertrauenswürdig eingestuft werden muss.

g) Welche Aussage ist in einem Monoprozessor-Betriebssystem richtig?

2 Punkte

- Es befindet sich zu einem Zeitpunkt maximal ein Prozess im Zustand *laufend*.
- Ein Prozess im Zustand *blockiert* muss warten, bis der laufende Prozess den Prozessor abgibt und kann dann in den Zustand *laufend* überführt werden.
- Ist zu einem Zeitpunkt kein Prozess im Zustand *bereit*, so ist auch kein Prozess im Zustand *laufend*.
- In den Zustand *blockiert* gelangen Prozesse nur aus dem Zustand *bereit*.

h) Welche Aussage zu Zeigern ist richtig?

2 Punkte

- Die Übergabesemantik für Zeiger als Funktionsparameter ist call-by-reference.
- Ein Zeiger kann zur Manipulation von schreibgeschützten Datenbereichen verwendet werden.
- Der Übersetzer erkennt bei der Verwendung eines ungültigen Zeigers die problematische Code-Stelle und generiert Code, der zur Laufzeit die Meldung „Segmentation fault“ ausgibt.
- Zeiger können verwendet werden, um in C eine call-by-reference Übergabesemantik nachzubilden.

i) In einem UNIX-Dateisystem gibt es symbolische Verweise (symbolic links) und feste Verweise (hard links) Welche der folgenden Aussagen ist richtig?

2 Punkte

- Ein hard link kann nicht auf Dateien, sondern nur auf Verzeichnisse verweisen.
- Auf ein Verzeichnis verweist immer genau ein symbolic link.
- Die Anzahl der hard links, die auf ein Verzeichnis verweisen, hängt von der Anzahl seiner Unterverzeichnisse ab.
- Ein symbolic link kann nicht auf Dateien in anderen Dateisystemen verweisen.

2) Mehrfachauswahlfragen (4 Punkte)

Bei den Mehrfachauswahlfragen in dieser Aufgabe sind jeweils m Aussagen angegeben, davon sind n ($0 \leq n \leq m$) Aussagen richtig. Kreuzen Sie alle richtigen Aussagen an. Jede korrekte Antwort in einer Teilaufgabe gibt einen Punkt, jede falsche Antwort einen Minuspunkt. Eine Teilaufgabe wird minimal mit 0 Punkten gewertet, d. h. falsche Antworten wirken sich nicht auf andere Teilaufgaben aus.

Wollen Sie eine falsch angekreuzte Antwort korrigieren, streichen Sie bitte das Kreuz mit drei waagrechten Strichen durch (~~⊗~~).

Lesen Sie die Frage genau, bevor Sie antworten.

a) Gegeben sei folgendes Programmfragment:

4 Punkte

```
extern int a;

int *f1 (int * const y, int x) {
    static int b = 0x20072021;
    auto int c;
    (*y)++;
    char *d = (char *) malloc(x);
    memcpy(d, y, x);
    int (*e)(int) = (int (*)(int)) 0x00002b04;
    b = e(c);
    return &b;
}
```

Welche der folgenden Aussagen zum obigen Programmfragment sind richtig?

- Die Anweisung `(*y)++` führt zu einem Übersetzerfehler, da ein konstanter Wert manipuliert würde.
- Andere Funktionen können mittels des „::“-Operator auf `b` zugreifen (z.B. `f1::b`).
- Der Funktionsaufruf über den Zeiger `e` schlägt immer fehl, da er niemals auf gültige Maschinenbefehle zeigen kann.
- Im Erfolgsfall zeigt `d` in den Heap.
- Obwohl `b` eine lokale Variable ist, darf ihre Adresse von anderen Funktionen genutzt werden.
- `c` ist mit dem Wert `0` initialisiert.
- Die Variable `c` liegt im Stacksegment.
- Der initiale Wert von `a` ist nicht ersichtlich.

Sie dürfen diese Seite zur besseren Übersicht bei der Programmierung heraustrennen!

Aufgabe 2: bau (45 Punkte)

Schreiben Sie das Programm `bau` (**build and update**), welches eine vereinfachte Variante von `make` darstellt. Die Bauanweisungen werden dabei über die Standardeingabe übertragen. Jede Zeile enthält eine Bauanweisung, die durch Leerzeichen getrennt aus dem *Ziel*, dessen *Abhängigkeiten* und dem *Befehl* zum Erstellen besteht. Das *Ziel* ist ein Dateipfad. Ihm folgt die Anzahl an *Abhängigkeiten* sowie deren Namen. Die restlichen Bestandteile der Zeile werden als *Befehl* interpretiert. Der *Befehl* wird immer dann ausgeführt, wenn die Zielfile noch nicht existiert, oder eine der *Abhängigkeiten* jünger ist als das *Ziel* selbst.

Exemplarischer Aufbau von Bauanweisungen:

```

bau 1 bau.o      gcc -o bau bau.o
bau.o 2 bau.c bau.h gcc -o bau.o -c bau.c
  Ziel  Abhängigkeiten      Befehl

```

Implementieren Sie dazu folgende Funktionen:

`char **tokenize(char *line)`

Zerlegt `line` in die durch Leerzeichen getrennten Bestandteile und gibt diese in einem dynamisch allokierten, NULL-terminierten Array zurück.

`void parse(char **tokens, struct recipe *out)`

Nimmt die Elemente einer Bauanweisungszeile aus `tokens` und speichert diese passend in `outs` Strukturmitglieder. `*out` wurde durch den Aufrufer allokiert. Alle Ressourcen von `tokens`, die nicht mehr benötigt werden, müssen freigegeben werden. Ungültige Eingaben führen zum Programmabbruch.

`void build(char *cmd[])`

Führt den im NULL-terminierten Array `cmd` spezifizierten Befehl in einem eigenen Prozess aus und wartet auf dessen Terminierung. Ein Rückgabewert des Kindprozess ungleich 0 soll als schwerwiegender Fehler interpretiert werden (`die()`).

`time_t check(char *target, struct recipe* recipe, size_t n)`

Existiert das *Ziel* `target` und ist ein Verzeichnis, so wird der Rückgabewert von `check_dir` zurückgegeben. Andernfalls ist diese Funktion dafür verantwortlich, `target` im Bedarfsfall zu erstellen oder zu erneuern. Dazu wird die passende Bauanleitung aus dem Array `recipe` der Größe `n` gesucht. Wird keine gefunden, so soll die letzte Modifikationszeit von `target` zurückgegeben werden oder das Programm mit einer Fehlermeldung abbrechen, falls `target` nicht existiert. Andernfalls muss durch Aufrufe an `check` bestimmt werden, ob eine der *Abhängigkeiten* jünger ist als `target` selbst. Ist dies der Fall oder `target` existiert noch nicht, so muss der entsprechende *Befehl* ausgeführt werden, so dass dieses aktualisiert oder erzeugt wird. Rückgabewert ist letztlich der Zeitstempel der letzten Modifikation von `target`.

`time_t check_dir(char *target, struct recipe * recipe, size_t n)`

Durchsucht das Verzeichnis `target` und ermittelt durch `check` den jüngsten Eintrag der nicht mit `'.'` beginnt und gibt dessen Zeitstempel zurück. Ist kein Eintrag vorhanden so wird 0 zurückgegeben.

`int main(int argc, char *argv[])`

Liest mit der vorgegebenen Funktion `read_recipes()` (siehe nächste Seite) alle Bauanweisungen ein. Alle an `bau` übergebenen Parameter werden als *Ziel* interpretiert und gebaut. Vor dem ordnungsgemäßen Beenden müssen belegte Ressourcen wieder freigegeben werden.

Hinweise:

- Sie dürfen annehmen, dass Dateinamen keine Leerzeichen enthalten
- Zeitstempel sind Integer, die Sekunden seit 1. Januar 1970 zählen. Je höher der Wert, desto jünger die Datei
- Der Zeitstempel der letzten Modifikation einer Datei steht in `st_mtim.tv_sec` (`stat(3)`)
- Beachten Sie die auf der nächsten Seite beschriebenen Hilfsfunktionen und Makros
- Für schwerwiegende Fehler darf der Prozess nach Ausgabe einer Fehlermeldung terminieren
- Vorausdeklarationen der Funktionen sind nicht nötig.

Sie dürfen diese Seite zur besseren Übersicht bei der Programmierung heraustrennen!

```

#include <dirent.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

```

```

struct recipe {
    char *target; // name of the target
    char **deps; // NULL-terminated array
    char **cmd; // NULL-terminated array
};

```

```

static inline time_t max(time_t a, time_t b) { return a > b ? a : b; }

```

```

_Noreturn static void die(const char *msg) { perror(msg); exit(EXIT_FAILURE); }
#define fdie(format, ...) \
    do {fprintf(stderr, format, __VA_ARGS__); exit(EXIT_FAILURE);} while(0)

```

```

// converts a string 's' into an unsigned integer of type size_t
// arg s: the string which will be parsed
// arg out: location where the parsed integer will be stored
// returns: 0 on success, otherwise -1
static int str_to_size_t(char *s, size_t *out);

```

```

// arg path: path to a file
// returns: modification timestamp of file 'path'
// error: calls die() on error
static time_t get_mtime_or_fail(char *path);

```

```

// reads lines from stdin and converts them to 'struct recipe'
// arg out: dynamically allocated array of 'struct recipe',
//          stored at location '*out'
// arg tokenize: pointer to function used for split read lines
// arg parse: pointer to function used for parsing lines
// returns: the number of entries stored in '*out'

```

```

static size_t read_recipes(struct recipe **out,
    char **(*tokenize)(char *),
    void (*parse)(char **tokens, struct recipe *out));

```

```

// frees all strings of a NULL-terminated string array and the array itself
static void free_array(char **a);

```

char **tokenize(char *line) {

}

void parse(char **tokens, struct recipe *out) {

T:



// Abhängigkeiten

// Befehl

// Ressourcenfreigabe

}

void build(char *cmd[]) {

// Ausführung starten



P:



// Auf Beendigung warten

}

B:

time_t check_dir(char *target, struct recipe *recipe, size_t n) {

// Verzeichnis auslesen

}

V:

time_t check(char *target, struct recipe * recipe, size_t n) {

// Existenz prüfen

// Bauanweisung finden

// Abhängigkeit prüfen

// Falls nötig: Ziel bauen

}

int main(int argc, char *argv[]) {

// Bauanweisungen einlesen

// Ziele prüfen

// Ressourcen freigeben

}

C:

M:

Aufgabe 3: Echtzeitbetrieb (9 Punkte)

1) Erklären Sie die Besonderheiten des *Echtzeitbetriebs* in einigen Stichpunkten. (3 Punkte)

2) Erklären Sie die Verhaltensunterschiede eines Echtzeitbetriebssystems, das eine Terminverletzung bei festen (*firm*) bzw. harten (*hard*) Terminvorgaben erkannt hat. Gehen Sie hierbei darauf ein, welcher Teil des Systems die Behandlung der Terminüberschreitung vornimmt. (6 Punkte)

