Aufgabe 1: (20 Punkte)

Bei den Multiple-Choice-Fragen ist jeweils nur **eine** richtige Antwort anzukreuzen. Falsche Beantwortung führt bei der einzelnen Frage zu Null Punkten. Lesen Sie die Frage genau, bevor Sie antworten.

Wollen Sie eine Multiple-Choice-Antwort korrigieren, kreisen sie bitte die falsche Antwort ein und kreuzen die richtige an. Eine Frage, bei der nicht eindeutig die eine richtige Lösung angekreuzt ist, kann nicht gewertet werden.

a) In **jedem** Verzeichnis eines UNIX-UFS-Dateisystem gibt es Verzeichnisse mit bestimmten vorgegebenen Namen. Welche der Aussagen ist richtig? (2 Punkt)

- ❏ Die Datei „`.login`“ verweist auf ein Login-Skript.
- ❏ Das Verzeichnis „`.`“ verweist auf das Verzeichnis, in dem der Eintrag steht; das Verzeichnis „`..`“ auf das im Pfad übergeordnete Verzeichnis.
- ❏ Das Verzeichnis „`..`“ verweist auf das Verzeichnis, in dem der Eintrag steht. Das Verzeichnis „`.`“ wird ignoriert.
- ❏ Das Verzeichnis „`.`“ verweist auf das aktuelle Verzeichnis des Prozesses. Das Verzeichnis „`..`“ verweist auf dessen übergeordnetes Verzeichnis.

b) In einem UNIX-UFS-Dateisystem gibt es symbolische Namen/Verweise (Symbolic Links) und feste Links (Hard Links) auf Dateien. Welche Aussage ist richtig. (2 Punkte)

- ❏ Ein Symbolic Link kann nur auf Dateien nicht jedoch auf Verzeichnisse verweisen.
- ❏ Ein Hard Link kann nur auf Verzeichnisse nicht jedoch auf Dateien verweisen.
- ❏ Hard Links können nur vom Systemadministrator angelegt werden.
- ❏ Symbolic Links können existieren, obwohl die verwiesene Datei oder das verwiesene Verzeichnis bereits gelöscht wurde.

c) Welche Antwort stellt **kein** Attribut einer Datei eines UNIX-UFS-Dateisystems dar? (1 Punkt)

- ❏ Anzahl der Symbolischen Verweise (Symbolic Links)
- ❏ Anzahl der Links (Hard Links)
- ❏ Gruppenzugehörigkeit
- ❏ Zugriffsrechte

d) In einem UNIX- oder Linux-System gibt es Spezialdateien zur Repräsentation von Geräten. Welche Aussage ist richtig? (1 Punkte)

- ❏ Spezialdateien von Festplatten können nicht von einem Prozess angesprochen werden.
- ❏ Die Spezialdateien ermöglichen es den Anwendungsprozessen, Geräte wie Datein mit `open()` und `close()` zu öffnen und zu schließen.
- ❏ Spezialdateien müssen immer in dem Verzeichnis „`/dev`“ stehen.
- ❏ Spezialdateien dürfen nur von Prozessen mit Administrator-(Root-)Rechten geöffnet werden.

e) Sie müssen einen kritischen Abschnitt implementieren. Welches der aufgeführten Koordinierungsmittel ist dazu am Besten geeignet? (1 Punkt)

- ❏ Up-Down-Systeme
- ❏ Algorithmus von Peterson
- ❏ binärer Semaphor
- ❏ PV-Chunk-Semaphor

f) Welcher UNIX-Systemaufruf wird bei der Verwendung von Sockets auf keinen Fall gebraucht? (1 Punkt)

- ❏ `close()`
- ❏ `bind()`
- ❏ `listen()`
- ❏ `open()`

g) Für welchen Zweck wird der Systemaufruf `listen()` benutzt? (3 Punkte)

- ❏ Damit das Betriebssystem überhaupt Systemaufrufe annimmt, muss es erst mit `listen()` in einen Modus des „Zuhörens“ gebracht werden.
- ❏ Der Aufruf von `listen()` wartet solange an einem Socket, bis eine einkommende Verbindungsanfrage vorliegt.
- ❏ Mit `listen()` wird ein Socket für die Verbindungsannnahme vorbereitet. Ein Parameter gibt an, wieviele Verbindungsanfragen vor deren Annahme gepuffert werden können.
- ❏ Mit `listen()` wird ein Socket für die Verbindungsannahme vorbereitet. Ein Parameter gibt an, wieviele laufende Verbindungen maximal möglich sind.

h) Welche Aussage ist bezüglich der Festplattentreiber eines UNIX-Betriebssystems richtig? (2 Punkte)

- ❏ Festplattentreiber werden innerhalb des UNIX-Betriebssystems über eine einheitliche Schnittstelle angesprochen.
- ❏ Jede einzelne Festplatte benötigt ihren eigenen Treiber.
- ❏ Der Festplattentreiber übernimmt Betriebssystemaufgaben wie z.B. die Implementierung eines Dateisystems.
- ❏ Ein Festplattentreiber in UNIX wird in einem eigenen Prozess realisiert.

i) Was muss ein(e) Software-Entwickler(in) unbedingt beachten, damit seine/ihre Programme nicht Opfer eines Hacker-Angriffs werden? (3 Punkte)

- ❏ Bei Verwendung der Programmiersprache C muss darauf geachtet werden, dass die Funktion `gets()` nur eingesetzt wird, wenn die Länge des einzulesenden Strings in das Ziel-Array passt.
- ❏ Es dürfen keine vorgefertigten Bibliotheksfunktionen verwendet werden, weil deren Implementierung als nicht vertrauenswürdig eingestuft werden muss.
- ❏ Der Quellcode darf nicht herausgegeben werden, damit Schwachstellen nicht entdeckt werden können.
- ❏ Die Funktionen `strcpy()` und `strcat()` dürfen nicht verwendet werden.

j) Welches Ziel verfolgt eine Prozess-Auswahlstrategie (Scheduling-Strategie) eines modernen Betriebssystems? (2 Punkte)

- ❏ Ein Prozess wird in seiner Priorität abgesenkt, falls er viele Ein-, Ausgabeoperationen durchführt.
- ❏ Die Rechenzeit wird gleichmäßig auf alle Prozesse verteilt.
- ❏ Ein Prozess mit einer langen Rechenphase wird in seiner Priorität angehoben, wenn keine Ein-, Ausgabeoperationen zu erwarten sind.
- ❏ Ein Prozess wird in seiner Priorität abgesenkt, falls er an einer langen Rechenphase arbeitet, und wieder angehoben, falls er Ein-, Ausgabeoperationen durchführt oder zu lange im Zustand *bereit* verbracht hat.

k) Sie kennen den Translation-Look-Aside-Buffer (TLB). Welche Aussage ist richtig? (2 Punkte)

- ❏ Der TLB verkürzt die Zugriffszeit auf den physikalischen Speicher, da ein Teil der möglichen Speicherabbildungen in einem sehr schnellen Pufferspeicher vorgehalten wird.
- ❏ Der TLB puffert Daten bei der Ein-,Ausgabebehandlung und beschleunigt diese damit.
- ❏ Der TLB ist eine schnelle Umsetzeinheit der MMU, die physikalische in logische Adressen umsetzt.
- ❏ Wird eine Speicherabbildung im TLB nicht gefunden, wird der auf den Speicher zugreifende Prozess mit einer Schutzraumverletzung (Segmentation Fault) abgebrochen.

Aufgabe 2: (40 Punkte)

Schreiben Sie ein Programm **runtree**, welches rekursiv Verzeichnisse traversiert und für jede gefundene reguläre Datei ein Kommando ausführt. Dem **runtree**-Programm werden als Argumente das Startverzeichnis und das auszuführende Kommando übergeben:

runtree *Verzeichnis Kommando*

Für das Bearbeiten des Kommandos soll jeweils ein neuer Prozess gestartet werden, dem der jeweilige Dateiname als Argument übergeben wird. Die maximale Anzahl dieser Prozesse soll auf drei beschränkt werden. Zu diesem Zweck wird vom Programm ein Zähler verwaltet. Wird die maximale Anzahl erreicht wartet der **runtree**-Prozess solange bis einer seiner Sohn-Prozesse sich beendet. Um die beendeten Sohn-Prozesse möglichst schnell aufzuräumen, muss das Programm zusätzlich noch auf das Signal SIGCHLD reagieren.

Auf den folgenden Seiten finden Sie ein Gerüst für das beschriebene Programm. In den Kommentaren sind die Aufgaben der einzelnen, zu ergänzenden Programmteile beschrieben. Zu ergänzen sind die Funktionen:

- main()
- traverse_dir()
  Diese Funktion bearbeitet ein Verzeichnis und soll sich für Unterverzeichnisse rekursiv aufrufen.
- run_command()
  Die Funktion erzeugt einen Sohn-Prozess, der das Kommando für eine Datei ausführt.
- sigchld_handler()
  Diese Funktion wird als SIGCHLD-Handler registriert.

Alle Programmteile, an denen Ergänzungen vorgenommen werden müssen, sind wie dieser Absatz am Seitenrand markiert. Ergänzungen sind eventuell auch innerhalb vorgegebener Zeilen nötig.

Vorgegeben sind Funktionen zum Installieren des Signal-Handlers, zum Ein- und Abschalten der Signalbehandlung sowie zum Warten auf das Terminieren eines Sohn-Prozesses.


```
/*
 * runtree <dir> <command>
 */

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

/* Deklaration von Funktionen, die bei der Programmierung von
 * main benötigt werden und die im Rahmen dieser
 * Aufgabe (zusätzlich zu main) zu programmieren sind
 */

/* traversiert rekursiv das jeweilige Verzeichnis */
void traverse_dir(char* dir_name, char* command);

/* startet einen Sohnprozess */
void run_command(char* command, char* file_name);

/* Signalhandler für SIGCHLD */
void sigchld_handler(int sig);

/* Deklaration von Funktionen, die bei der Programmierung
 * benötigt werden und die im Rahmen dieser Aufgabe bereits
 * vorgegeben sind.
 */

/* Installiert den SIGCHLD-Signalhandler */
void install_signalhandler();

/* Blockiert das Signal SIGCHLD */
void block_sigchld();

/* Deblockiert das Signal SIGCHLD */
void unblock_sigchld();

/* Wartet auf einen einzelnen Sohn-Prozess */
int wait_for_child();
```

```
/* Globale Variablen
 */

/* maximale Anzahl der Sohn-Prozesse */
const int max_children=3;

/* aktuelle Anzahl von Sohn-Prozessen */
int number_of_children=0;

/*
 * Funktion main
 */

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

      /* Ueberpruefen der korrekten Anzahl von Argumenten*/
      if (argc!=     ) {
        fprintf(stderr,"usage: %s <dir> <command>",argv[0]);
        exit(EXIT_FAILURE);
      }

      /*
       * Funktionsaufruf zum Installieren vom Signal-Handler
       */

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

      /*
       * Funktionsaufruf zum Traversieren der Verzeichnisse
       */

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

      /* warten auf noch laufende Sohn-Prozesse */
      block_sigchld();
      while (number_of_children>0 && (wait_for_child()>0)) ;

      return 0;
}
```



```
/*
 * Die Funktion traverse_dir öffnet das jeweilige Verzeichnis
 * und iteriert anschliessend über die Verzeichniseinträge.
 * Ist der Verzeichniseintrag selber wieder ein Verzeichnis
 * so ruft sich die Funktion rekursiv auf. Bei Verzeichnis-
 * einträgen von regulären Dateien wird die Funktion
 * run_command aufgerufen. (Achtung: Die Funktion geht über
 * mehrere Seiten)
 */

void traverse_dir(char* dir_name,char* command) {
      struct dirent* entry;
      struct stat fstatus;
      DIR*   dir;
      char name_buf[MAX_PATH];


      /* Verzeichnis öffnen */

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

      /* Die Funktion soll hier beginnen über alle Verzeichnis-
       * einträge zu iterieren. Verzeichniseinträge, welche mit
       * einem Punkt beginnen, sollen übersprungen werden. */

      while
      ------------------------------------------------------------------
      ------------------------------------------------------------------
      ------------------------------------------------------------------
      ------------------------------------------------------------------
      ------------------------------------------------------------------
```

```
        /* Aus jedem Verzeichniseintrag wird zusammen mit
         * dem aktuellen Verzeichnisnamen ein kompletter
         * Path-Name gebildet. Path-Namen, welche länger
         * als MAX_PATH sind, werden übersprungen. */

        /* Mit dem kompleten Path-Namen ist der Dateistatus
         * abzufragen. Handelt es sich bei dem Verzeichniseintrag
         * um ein Verzeichnis so wird die Funktion traverse_dir
         * rekursiv aufgerufen, bei einer regulären Datei wird
         * run_command aufgerufen, symbolische Links werden
         * nicht verfolgt. */

        } /* Hier endet die Iteration */
        closedir(dir);
}
```



```
/*
 * Die Funktion run_command erzeugt einen neuen Sohn-Prozess
 * und startet in diesem das Kommando mit dem Dateinamen als
 * erstem Argument. (Die Funktion geht über zwei Seiten)
 */

void run_command(char* command, char* file_name) {
        pid_t pid;
        char* arguments[3];

        /*
         * Ausfüllen des Argumenten-Vektors
         */
        arguments[0] =
        arguments[1] =
        arguments[2] =

        /* SIGCHLD-Signal blockieren, damit Prozess-Zähler
         * konsistent bleibt */
        block_sigchld();

        /* Ist die maximale Anzahl von Sohn-Prozessen erreicht
         * dann soll an dieser Steller auf das Ende eines Sohn-
         * Prozesses gewartet werden */
```

```
    /*
     * An dieser Stelle soll nun ein neuer Sohn-Prozess
     * erzeugt werden. Dieser führt das Kommando aus.
     * Der Vater-Prozess erhöht anschliessend seinen
     * Kind-Prozesszähler. */

    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....

    /* Vor dem Verlassen der Funktion muss das Signal SIGCHLD
     * wieder deblockiert werden */
    unblock_sigchld();

}
```



```
/*
 * Signalhandler für SIGCHLD installieren.
 */
void install_signalhandler() {
      struct sigaction action;

      memset(&action, 0, sizeof(action));
      sigemptyset (&action.sa_mask);
      action.sa_flags = 0;
      action.sa_handler = sigchld_handler;

      if (sigaction (SIGCHLD, &action, NULL) == -1) {
        perror("Fehler beim Installieren des Signalhandlers.");
        exit(EXIT_FAILURE);
      }


}
/*
 * Signalhandler für SIGCHLD.
 * In diesem Signalhandler sollen alle zu diesem Zeitpunkt
 * beendeten Sohn-Prozesse aufgeräumt werden.
 */
void sigchld_handler(int sig) {
      pid_t ret;
      int errno_bak = errno;

      .....
      .....
      .....
      .....
      .....
      .....
      .....
      .....

      errno = errno_bak;
}
```

```
/*
 * Signal SIGCHLD blockieren.
 */
void block_sigchld() {
       sigset_t set;
       sigemptyset(&set);
       sigaddset(&set, SIGCHLD);
       sigprocmask(SIG_BLOCK, &set, NULL);

}
/*
 * Signal SIGCHLD deblockieren.
 */
void unblock_sigchld() {
      sigset_t set;
      sigemptyset(&set);
      sigaddset(&set, SIGCHLD);
      sigprocmask(SIG_UNBLOCK, &set, NULL);

}
/*
 * Die Funktion wait_for_child wartet auf das Beenden eines
 * einzelnen Sohn-Prozesses. (Diese Funktion ist nicht für
 * den Signalhandler geeignet!)
 */
int wait_for_child() {
      pid_t pid;


      while (pid=wait(NULL)) {
        if (pid<0) {
          if (errno==EINTR) continue;
          perror("");
          return -1;
        }
        number_of_children--;
        return pid;
      }
      return 0;
}
```