Aufgabe 1: Ankreuzfragen (13 Punkte)

a) Einfachauswahlfragen (6 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.

I) Welche Aussage über `exec()` ist richtig? 2 Punkte

- ○ `exec()` erzeugt einen neuen Kind-Prozess und startet darin das angegebene Programm.
- ○ Dem Vater-Prozess wird die Prozess-ID des Kind-Prozesses zurückgeliefert.
- ○ Das im aktuellen Prozess laufende Programm wird durch das angegebene Programm ersetzt.
- ○ Der an `exec()` übergebene Funktionszeiger wird durch einen neuen Thread im aktuellen Prozess ausgeführt.

II) Ein Prozess wird vom Zustand `blockiert` in den Zustand `bereit` überführt. Welche Aussage passt zu diesem Vorgang? 2 Punkte

- ○ Ein anderer Prozess wurde vom Betriebssystem verdrängt und der erstgenannte Prozess wird nun auf der CPU eingelastet.
- ○ Der Prozess wird wegen eines ungültigen Speicherzugriffs (Segmentation Fault) beendet.
- ○ Es ist kein direkter Übergang von blockiert nach bereit möglich.
- ○ Der Prozess hat auf Daten von der Festplatte gewartet, die nun verfügbar sind.

III) Welche Aussagen zu hierarchischen Dateisystemen ist richtig? 2 Punkte

- ○ Der Nachteil von hierarchischen Namensräumen besteht darin, dass das Dateisystem spezielle Funktionen zum Auflösen von Namenskonflikten implementieren muss
- ○ Hierarchische Dateisysteme sind aufgrund ihrer flachen Namensräume besonders einfach zu implementieren und werden daher insbesondere für Mehrbenutzersysteme verwendet.
- ○ Hierarchische Dateisysteme ermöglichen es, dass gleiche Namen in verschiedenen Kontexten vorkommen dürfen.
- ○ Verweise auf Dateien übergeordneter Ebenen sind nicht Zulässig, da ansonsten Zyklen entstehen könnten.

b) Mehrfachauswahlfragen (7 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.

I) Welche der genannten Aufgaben gehören zwingend zum Betriebssystemkern? 2 Punkte

- ☐ Multiplexing der Hardware
- ☐ Gerätetreiber
- ☐ Isolation zwischen Prozessen
- ☐ Benutzeroberfläche (z.Ḃ. Shell)

II) Welche der genannten Aussagen zum Thema Semaphoren sind richtig? 2 Punkte

- ☐ Die `up()` Funktion eines Semaphors erhöht den Zähler der Semaphore falls möglich und wartet sonst bis dies möglich ist.
- ☐ Die `down()` Operation kann nur von dem Kontrollfluss aufgerufen weden, der sich gerade im kritischen Abschnitt befindet.
- ☐ Ein Semaphor kann zum Signalisieren von Ereignissen verwendet werden.
- ☐ Ein Semaphor kann verwendet werden, um gegenseitigen Ausschluss zu implementieren.

III) Betrachten Sie folgendes Programmfragment. Welche der Aussagen trifft zu? 3 Punkte

```
static int a = 20180801;
int foo(int x) {
  char b[] = "Hello ";
  static int c;
  int (*d)(int *) = foo;
  int *e = malloc(800*sizeof(int));
  strcat(&b, "Paul");
  ....
}
```

- ☐ Auf `a` kann von anderen Modulen aus zugegriffen werden.
- ☐ `b` verliert beim Rücksprung aus `foo` seine Gültigkeit.
- ☐ `e` liegt auf dem Stack.
- ☐ `strcat` erzeugt einen Pufferüberlauf.
- ☐ Das Ergebnis des Aufrufs von `foo` wird in `d` gespeichert.
- ☐ `c` ist uninitialisert und enthält einen zufälligen Wert

Aufgabe 2: Programmieraufgabe – minidu (6 Punkte)

Schreiben Sie ein Programm `minidu` (**mini**mum **d**isk **u**sage), welches aus den über die Kommandozeile übergebenen Dateien diejenige herausucht, die die geringste Größe hat. Zu dieser Datei soll der Name der Datei gefolgt von der Größe ausgegeben werden.

Sollte die Information zu einer Datei nicht verfügbar sein, so wird eine sinnvolle Fehlermeldung ausgegeben und dann fortgefahren.

Bei Ausführung ohne Angabe von Dateien wird ein Nutzungshinweis ausgegeben.

Das Ermitteln der Größe einer Datei soll in der Funktion `int getsize(const char * filename);` implementiert werden. Diese Funktion liefert als Ergebnis die Größe einer Datei. Im Fehlerfall wird `-1` zurückgegeben und die Fehlerursache in der globalen Variable `errno` hinterlegt.

Beispiel:

```
bjoe.fiedler@gbs:~ $ ./minidu test file1 temp dir1 file2
dir1: Permission denied
file1: 5
```

Beachten Sie die beigefügten Man-Pages. Diese geben Hinweise auf die Verwendung von Bibliotheksfunktionen und deren Verhalten. Achten Sie weiterhin darauf, dass Sie eventuelle Fehlersituationen der Bibliotheksfunktionen im Sinne der Aufgabenstellung behandeln.

Verwenden Sie zum Bearbeiten dieser Aufgabe die Vorgabe auf der folgenden Seite. Ergänzen Sie jewils die fehlenden Teile des Programmtextes, sodass ein Programmablauf im Sinne der Aufgabenstellung entsteht.


```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <limits.h>

int getsize(const char * filename) {
   struct stat info;
   int status = 
   if (status == -1) {
      return 
   }
   return 
}
int main(int argc, char* argv[]) {
   if (argc <= ) {
      printf("Usage: %s FILE...n", argv[0]);
      exit(EXIT_SUCCESS);
   }
   int min_size = 
   char * min_file = 
   for (int i= ++i) {
      int size = 
      if (size == ) {
         perror(argv[i]);
      } else {
         if (size < min_size) {
         min_size = size;
         min_file = argv[i];
         }
      }
   }
   if (min_file) {
      printf("%s: %d\n", min_file, min_size);
   } else {
      printf("could not obtain any information.\n");
   }

   return 0;
}
```

Aufgabe 3: Programmieraufgabe – slush (13.5 Punkte)

Schreiben Sie ein Programm `slush` (**s**tudents **l**earn **u**sing **sh**ells), welches ein Kommandozeileninterpreter ist. Die `slush` soll von der Standardeingabe zeilenweise Befehle entgegennehmen und diese dann ausführen. Hierzu muss eventuell ein Kindprozess erzeugt werden.

Die bereitgestellte Funktion **`int`** `getNextCommand(cmd_t * cmd)` (nicht selber implementieren!) übernimmt das Einlesen der Befehlszeilen. Die Rückgabe signalisiert, ob das Ende der Eingabe(0) erreicht wurde oder eine Zeile erfolgreich gelesen wurde(1). Die übergebene Struktur enthält nach dem Aufruf die gelesene Zeile in `line` und die einzelnen Teile dieser aufgetrennt (mittels `strtok()`) in `argv`. Der Eintrag `invalid` signalisiert, ob die gelesene Zeile valide für die Ausführung ist oder übersprungen werden soll. Die Funktion allokiert Speicher, den Sie an geeigneter Stelle mit `freeCommand(cmd_t *cmd)` wieder freigeben müssen.

Die Befehle werden immer im Hintergrund gestartet. Dies heißt, dass die Shell **nicht** auf das Terminieren des Kindprozesses wartet bevor sie die nächste Zeile liest. Die Shell gibt jeweils beim Starten eines Kindes eine Zeile mit der Befehlszeile und der Prozess ID aus.

Bei dem Lesen einer neuen Zeile sollen alle eventuell terminierte Kindprozesse eingesammelt werden. Dies geschieht in der Funktion `bury()`. Für jeden "begrabenen" Kindprozess wird eine Zeile mit der Prozess ID und dem Exitstatus ausgegeben. Fehler hier aufgerufener Funktionen werden ignoriert. Alle Kinder beenden sich normal (nicht durch Signale o.ä.).

Beachten Sie die beigefügten Man-Pages. Diese geben Hinweise auf die Verwendung von Bibliotheksfunktionen und deren Verhalten. Achten Sie weiterhin darauf, dass Sie eventuelle Fehlersituationen der Bibliotheksfunktionen im Sinne der Aufgabenstellung behandeln.

Beispiel:

Eingabe:

```
echo 1
bash -c non_existing_command
```

Ausgabe:

```
Started [echo 1] pid=44
Started [bash -c non_existing_command] pid=43
Exited pid=44 exitstatus=0
Exited pid=43 exitstatus=127
```

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

typedef struct cmd {
  int invalid;                  // 0=valid, 1=skip this line
  char * line;                  // the full line as read
  size_t line_length;           // size of the mem behind line
  char ** argv;                 // the parsed arguments (after strtok)
} cmd_t;
```



```
int getNextCommand(cmd_t *cmd); //read a line, store in cmd
void freeCommand(cmd_t *cmd);   //free allocated memory
void bury(void);

int main(int argc, char* argv[]) {
  cmd_t cur_cmd;
  while (getNextCommand(&cur_cmd)) {
  }
```

Diese Seite darf herausgetrennt werden.

```
void bury(void) { //Kinder einsammeln
  pid_t child;
  int status;
```

S:

Aufgabe 4: Textaufgabe (12.5 Punkte)

Die folgenden Beschreibungen sollen kurz und prägnant erfolgen (Stichworte, kurze Sätze).

a) Welches Problem kann hier auftreten? Wo tritt das Problem auf? Wie kann man dies vermeiden? 3 Punkte

```
volatile unsigned long finished = 0;

//thread 1                    //thread 2
do {                          do {
  waitForJob();                 waitForJob();
  doJob();                      doJob();
  ++finished;                   ++finished;
} while(1);                   } while(1);
```


b) Worin besteht der Unterschied zwischen einem Symlink und einem Hardlink auf Ebene des Dateisystems? 2 Punkte

c) Beschreiben Sie die Prozesszustände bei der Einplanung von Prozessen sowie die Ereignisse, die jeweils zu expliziten und impliziten Zustandsübergängen führen (Skizze mit kurzer Erläuterung/Benennung der Zustände und Übergänge). 7.5 Punkte