Aufgabe 1: Ankreuzfragen (24 Punkte)

1) Einfachauswahlfragen (14 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) Welche Aussage zum Thema Betriebsarten ist richtig? 2 Punkte

- ☐ Echtzeitsysteme findet man hauptsächlich auf großen Serversystemen, die eine enorme Menge an Anfragen zu bearbeiten haben.
- ☐ Beim Stapelbetrieb können keine globalen Variablen existieren, weil alle Daten im Stapel-Segment (Stack) abgelegt sind.
- ☐ Mehrprogrammbetrieb ermöglicht die simultane Ausführung mehrerer Programme innerhalb desselben Prozesses.
- ☐ Mehrzugangsbetrieb ist nur in Verbindung mit CPU- und Speicherschutzmechanismen sinnvoll realisierbar.

b) Welche Aussage zu Zeigern ist richtig? 2 Punkte

- ☐ Zeiger vom Typ `void*` existieren in C nicht, da solche „Zeiger auf Nichts“ keinen sinnvollen Einsatzzweck hätten.
- ☐ Zeiger können verwendet werden, um in C eine call-by-reference Übergabesemantik nachzubilden.
- ☐ Die Übergabesemantik für Zeiger als Funktionsparameter ist call-by-reference.
- ☐ Ein Zeiger kann zur Manipulation von schreibgeschützten Datenbereichen verwendet werden.

c) Welche Aussage über `exec(3)` ist richtig? 2 Punkte

- ☐ Dem Eltern-Prozess wird die Prozess-ID des Kind-Prozesses zurückgeliefert.
- ☐ Der an `exec(3)` übergebene Funktionszeiger wird durch einen neuen Thread im aktuellen Prozess ausgeführt.
- ☐ Das im aktuellen Prozess laufende Programm wird durch das angegebene Programm ersetzt.
- ☐ `exec(3)` erzeugt einen neuen Kind-Prozess und startet darin das angegebene Programm.

d) Welche Aussage zum Thema Systemaufrufe ist richtig? 2 Punkte

- ☐ Benutzerprogramme dürfen keine Systemaufrufe absetzen, diese sind dem Betriebssystem vorbehalten.
- ☐ Durch einen Systemaufruf wechselt das Betriebssystem in den Adressraum von Maschinenprogrammen auf der Benutzerebene.
- ☐ Parameter werden nach einer festen Konvention an das System übergeben.
- ☐ Nach dem Umschalten in den privilegierten Prozessormodus wird eine vom Benutzer festgelegte Befehlsfolge ausgeführt.

e) Ein Prozess wird in den Zustand *bereit* überführt. Welche Aussage passt zu diesem Vorgang? 2 Punkte

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

f) Man unterscheidet die Begriffe Programm und Prozess. Welche der folgenden Aussagen zu diesem Thema ist richtig? 2 Punkte

- ☐ Der Binder erzeugt aus einer oder mehreren Objekt-Dateien einen 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.
- ☐ Mit Hilfe von Threads kann ein Prozess mehrere Programme gleichzeitig ausführen.

g) Gegeben seien die folgenden Präprozessor-Makros: 2 Punkte

```
#define SUB(a, b) a - b
#define MUL(a, b) a * b
```

Was ist das Ergebnis des folgenden Ausdrucks? `4 * MUL ( SUB(3,5), 2)`

- ☐ 2
- ☐ -16
- ☐ 16
- ☐ -2

2) Mehrfachauswahlfragen (10 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: 5 Punkte

```
static int a = 20190730;
int foo(int x) {
    static int b;
    b += 1;
    int c;
    int (*d)(const int) = foo;
    int *e = malloc(800);
    ++x;
    free(e);
    return b;
}
```

Welche der folgenden Aussagen zu den Variablen im Programm sind richtig?

- ○ Die Anweisung `++x` ändert den Wert von `x` und beeinflusst somit den Aufrufer.
- ○ `e` liegt auf dem Stack.
- ○ Bei jedem Aufruf von `foo` wird `e` derselbe Zeiger-Wert zugewiesen.
- ○ `c` verliert beim Rücksprung aus `foo` seine Gültigkeit.
- ○ `b` zählt wie oft die Funktion `foo` aufgerufen wird.
- ○ `b` ist mit `0` initialisiert und liegt im BSS-Segment.
- ○ Auf `a` kann von anderen Modulen aus zugegriffen werden.
- ○ `e` zeigt auf ein Array, in dem Platz für 800 Ganzzahlen vom Typ `int` ist.
- ○ Das Ergebnis des Aufrufs der Funktion `foo` wird in `d` gespeichert.
- ○ `c` ist uninitialisiert und enthält einen zufälligen Wert.

b) Welche der folgenden Aussagen zu UNIX-Dateisystemen sind richtig? 5 Punkte

- ○ Innerhalb eines UNIX-Dateisystembaumes können die Inhalte mehrerer Festplatten eingebunden sein.
- ○ Innerhalb eines Verzeichnisses können mehrere Verweise auf den selben Inode existieren, sofern diese unterschiedliche Namen haben.
- ○ Die Anzahl der festen Verknüpfungen (*hard links*), die auf ein Verzeichnis verweisen, hängt von der Anzahl seiner Unterverzeichnisse ab.
- ○ Ein Inode enthält u.a. den Namen der entsprechenden Datei.
- ○ Obwohl eine Datei gelöscht wurde, kann es symbolische Verknüpfungen (*symbolic links*) geben, die noch auf sie verweisen.
- ○ Auf eine Datei in einem Dateisystem verweisen immer mindestens zwei feste Verknüpfungen (*hard links*).
- ○ Zur Anzeige des Inhaltes einer Datei ist es notwendig, das Leserecht auf dem übergeordneten Verzeichnis zu besitzen.
- ○ In einem Verzeichnis darf es keinen Eintrag geben, der auf das Verzeichnis selbst verweist.
- ○ Beim Anlegen einer Datei wird die maximale Größe festgelegt. Wird sie bei einer Schreiboperation überschritten, wird ein Fehler gemeldet.
- ○ Beim lesenden Zugriff auf eine Datei über eine symbolische Verknüpfung (*symbolic link*) kann ein Prozess den Fehler *Permission denied* erhalten, obwohl er das Leserecht auf die symblische Verknüpfung besitzt.

Aufgabe 2: `dau` (45 Punkte)

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

Schreiben Sie ein mehrfädiges Programm `dau` (**d**isk **a**llocation per **u**ser), das den Gesamtspeicherplatzverbrauch pro Benutzer in frei wählbaren Verzeichnissen ermittelt und eine Übersicht erstellt. Der Aufruf an `dau` und die resultierende Ausgabe sehen aus wie folgt:

```
dust@faui48sp: ./dau /proj/i4sp1/pub /proj/i4sp1/sys
rudi (21608): 175073 Bytes
dust (21610): 123034 Bytes
```

Dabei soll für jeden als Argument an `dau` übergebenen Verzeichnispfad ein eigener Thread gestartet werden, der den Speicherverbrauch aller darin enthaltenen **regulären Dateien** ermittelt und dem Besitzer zuordnet. Benutzer werden durch die `uid`, einer Ganzzahl vom Typ `uid_t`, identifiziert. Gehen Sie davon aus, dass die `uid` vom Betriebssystem fortlaufend für jeden neu angelegten Benutzer vergeben wird. Dabei sollen jedoch Dateien mit einer `uid` < 1000 ignoriert werden. Die Anzahl der verfügbaren Benutzer ist nicht bekannt. Die `uid` ist Teil der `struct stat`.

Der Hauptthread startet die Arbeiterthreads und wartet passiv bis diese terminieren. Die Arbeiter aggregieren den Speicherverbrauch in einer globalen Variable vom Typ `struct agg`. Dabei zeigt das Strukturmitglied `usage` auf ein Array mit aggregiertem Speicherverbrauch je Benutzer, während `count` die Anzahl der Einträge im Array und damit die höchste gefundene `uid` speichert. Nach dem Aufsammeln der Arbeiter wird vom Hauptthread eine Rangfolge erstellt, in der die Benutzer **absteigend** nach ihrem Speicherverbrauch sortiert werden. Dafür sollen die in `struct agg` gesammelten Informationen in je ein `struct summary` pro Benutzer kopiert werden. Zur Sortierung der `struct summary` soll auf eine eigene Vergleichsfunktion zurückgegriffen werden.

Zum Abschluss wird die Zusammenfassung des Speicherverbrauchs für alle Nutzer mit einem Verbrauch > 0 Byte auf `stdout` ausgegeben. Zur Ausgabe der Ergebnisse soll die **vorgegebene** Funktion `print_user_usage()` genutzt werden. Diese ermittelt zu jeder `uid` den passenden Benutzernamen und tätigt die Ausgabe auf `stdout`. Sie kann nicht fehlschlagen.

Die Arbeiterthreads, zu implementieren als **`void *dir_iter(void *path)`**, steigen rekursiv die Verzeichnishierarchie hinab und addieren für jede relevante **reguläre Datei** den Speicherverbrauch auf den Gesamtverbrauch des Besitzers. Verzeichnisse, auf die nicht zugegriffen werden kann, sollen nach Ausgabe einer Fehlermeldung ignoriert werden. Zur Aggregation des Speicherverbrauchs soll die Funktion `add_usage()` genutzt werden.

`void add_usage(uid_t uid, off_t bytes)` erwartet eine Benutzer-Id als `uid_t` und den aufzuaddierenden Speicherverbrauch vom Typ `off_t` als Argument. Auf dem Datentyp `off_t` kann normale Ganzzahl-Arithmetik angewendet werden. Dabei arbeitet die Funktion auf einer globalen Variable vom Typ `struct agg`. Für bisher unbekannte `uid`s muss das Array `usage` ggf. vergrößert werden. Dabei entstehende Lücken müssen richtig initialisiert werden. Bedenken Sie mögliche Probleme, die durch nebenläufige Zugriff entstehen.

Zur Synchronisierung soll die aus der Vorlesung und Übung bekannte Semaphor-Schnittstelle verwendet werden. Diese ist auf nächste Seite kurz beschrieben.

Hinweis: Symbolischen Verknüpfungen soll **nicht** gefolgt werden.

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

```c
#include <dirent.h>
#include <errno.h>
#include <limits.h>
#include <pthread.h>
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

static void die(const char *msg) {
  perror(msg);
  exit(EXIT_FAILURE);
}

static void print_user_usage(uid_t uid, off_t bytes) {
  // Implementierung vorgegeben
}

#define MIN_UID 1000


// Erstellt eine neue Semaphor mit dem initialen Wert "initVal".
// Bei Erfolg ist der Rückgabewert ein Zeiger auf eine Semaphor.
// Andernfalls ist der Rückgabewert NULL und die errno wird
// entsprechend gesetzt.
SEM *semCreate(int initVal);

// Gibt alle von der Semaphor beanspruchten Ressourcen frei
void semDestroy(SEM *sem);

void P(SEM *sem);
void V(SEM *sem);

// Aggregieren des Speicherverbrauchs
struct agg {
  off_t        *usage;
  unsigned int  count;
};

// Speicherverbrauch pro einzelnem Nutzer zum Sortieren
struct summary {
  uid_t         uid;
  off_t         bytes;
};
```

```
// Deklaration globaler Variablen
// Vorausdeklaration von Funktionen ist nicht notwendig

// Hauptfunktion (main)
int main(int argc, char **argv) {
  // Initialisierung von Variablen

  // Threads starten

  // Auf Threads warten
```



```
  // Ausgabe vorbereiten und tätigen

  return EXIT_SUCCESS;
}
// Hilfsfunktion für die Sortierung
```

M:

```
// Funktion zum rekursiven Absteigen in Verzeichnishierarchien

  // Einträge des aktuellen Verzeichnis lesen
```



```
    // Auslesen der Dateiinformationen
    // Für Verzeichniseinträge ggf. rekursiv absteigen

  // Freigabe von Systemressourcen
```

F:

```
// Funktion um Speicherverbrauch pro Nutzer zu aggregieren
```

H:

Aufgabe 3: Synchronisation (9 Punkte)

1) Erläutern Sie das Konzept Semaphor. Welche Operationen sind auf Semaphoren definiert und was tun diese Operationen? (6 Punkte)

2) Skizzieren Sie in Programmiersprachen-ähnlicher Form, wie mit Hilfe eines zählenden Semaphors das folgende Szenario korrekt synchronisiert werden kann: Zu jedem Zeitpunkt müssen so viele Threads wie möglich, maximal jedoch 4, die Funktion `threadFunc` ausführen. Beachten Sie, dass es nicht notwendig ist alle freien Zeilen des untenstehenden Codefragments auszufüllen. Ihnen stehen dabei folgende Semaphor-Funktionen zur Verfügung: (3 Punkte)

- `SEM * semCreate(int);`
- `void P(SEM *);`
- `void V(SEM *);`

Hauptthread:

```
static SEM *s;
int main(void){

  while(1) {

  startWorkerThread(threadFunc);

  }

}
```

Arbeiterthread:

```
void threadFunc(void) {

  doWork();

}
```

Aufgabe 4: Prozesszustände (12 Punkte)

1) Beschreiben Sie die Prozesszustände bei der Einplanung von Prozessen sowie die Ereignisse, die jeweils zu Zustandsübergängen führen (Skizze mit kurzer Erläuterung der Zustände und Übergänge). (7 Punkte)

2) Sie haben in der Vorlesung zwei Arten von Programmunterbrechungen kennengelernt. Nennen Sie diese, beschreiben Sie mindestens zwei Eigenschaften, in denen sie sich voneinander unterscheiden und nennen Sie jeweils ein Beispiel. (5 Punkte)