Aufgabe 1: Ankreuzfragen (30 Punkte)

1) Einfachauswahlfragen (22 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 der folgenden Aussagen zu statischem bzw. dynamischem Binden ist richtig? 2 Punkte

- ☐ Bei dynamischem Binden müssen zum Übersetzungszeitpunkt alle Adressbezüge vollständig aufgelöst werden.
- ☐ Bei dynamischem Binden können Fehlerkorrekturen in Bibliotheken leichter übernommen werden, da nur die Bibliothek selbst neu erzeugt werden muss. Programme, die die Bibliothek verwenden, müssen nicht neu kompiliert und gebunden werden.
- ☐ Bei statischem Binden werden durch den Compiler alle Adressbezüge vollständig aufgelöst.
- ☐ Beim statischen Binden werden alle Adressen zum Ladezeitpunkt aufgelöst

b) Ein Betriebssystem setzt logische Adressräume auf der Basis von Segmentierung ein. Welche Aussage ist richtig? 2 Punkte

- ☐ Segmente können verschiedene Längen haben. Die Einhaltung der Längenbegrenzung wird vom C-Compiler überprüft.
- ☐ Die Segmentierung schränkt den logischen Adressraum derart ein, dass nur auf gültige Speicheradressen erfolgreich zugegriffen werden kann.
- ☐ Adressraumschutz durch Segmentierung erfordert keine Hardwareunterstützung.
- ☐ Die Bindung von Programm- an Arbeitsspeicheradressen erfolgt zur Ladezeit des Programms.

c) Wie funktioniert Adressraumschutz durch Eingrenzung? 2 Punkte

- ☐ Beim Laden eines Programms prüft das Betriebssystem, ob alle Speicherzugriffe im gültigen Bereich liegen.
- ☐ Begrenzungsregister legen einen Adressbereich im logischen Adressraum fest, auf den alle Speicherzugriffe beschränkt werden.
- ☐ Zur Laufzeit eines Programms ist durch Begrenzungsregister festgelegt, auf welchen Bereich des physikalischen Speichers das Programm zugreifen darf.
- ☐ Der Compiler grenzt beim Erzeugen des Codes die Adresszugriffe auf einen bestimmten Bereich ein.

d) Welche der folgenden Aussagen zum Thema Adressräume ist richtig? 2 Punkte

- ☐ Die maximale Größe des virtuellen Adressraums kann unabhängig von der verwendeten Hardware frei gewählt werden.
- ☐ Virtuelle Adressräume sind Voraussetzung für die Realisierung logischer Adressräume.
- ☐ Der physikalische Adressraum ist durch die gegebene Hardwarekonfiguration definiert.
- ☐ Der virtuelle Adressraum eines Prozesses kann nie größer sein als der physikalisch vorhandene Arbeitsspeicher.

e) Welche der folgenden Aussagen zum Thema Prozesszustände ist richtig? 2 Punkte

- ☐ Das Auftreten eines Seitenfehlers kann dazu führen, dass der aktuell laufende Prozess in den Zustand beendet überführt wird.
- ☐ Der Planer (scheduler) kann einen Prozess in den Zustand „blockiert“ überführen, indem er einen anderen Prozess einlastet.
- ☐ In einem Vierkernsystem können sich maximal vier Prozesse gleichzeitig im Zustand „bereit“ befinden.
- ☐ In einem Achtkernsystem gibt es stets genau acht laufende Benutzerprozesse.

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

```
#define ADD(x, y) x + y
#define SUB(x, y) x - y
```

Was ist das Ergebnis des folgenden Ausdrucks? `SUB(3, ADD(1, 4)) * 2`

- ☐ -7
- ☐ 10
- ☐ -4
- ☐ 12

g) Welche der folgenden Aussagen zum Thema Synchronisation sind richtig? 2 Punkte

- ☐ Für nicht-blockierende Synchronisationsverfahren ist spezielle Unterstützung durch das Betriebssystem notwendig.
- ☐ Der Einsatz von nicht-blockierenden Synchronisationsmechanismen kann zu Verklemmungen (Deadlocks) führen.
- ☐ Das Sperren von Interrupts kann von Benutzerprogrammen ohne weiteres zur Synchronisation auf Multiprozessor-Systemen eingesetzt werden.
- ☐ Der Einsatz von nicht-blockierenden Synchronisationsmechanismen kann nicht zu Verklemmungen (Deadlocks) führen.

h) Man unterscheidet zwischen Traps und Interrupts. Welche Aussage ist richtig?

2 Punkte

- ☐ Bei der mehrfachen Ausführung eines unveränderten Programmes mit den selben Eingabedaten treten Interrupts immer an den selben Stellen auf.
- ☐ Traps stehen immer in ursächlichem Zusammenhang zu der Ausführung eines Maschinenbefehls.
- ☐ Ein Trap wird immer durch das Programm behandelt, welches den Trap ausgelöst hat, Interrupts werden hingegen immer durch das Betriebssystem behandelt.
- ☐ Traps können nicht durch Speicherzugriffe ausgelöst werden.

i) Welche Aussage zu Zeigern ist richtig?

2 Punkte

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

j) Welche Aussage zum Thema Systemaufrufe ist richtig?

2 Punkte

- ☐ Nach der Bearbeitung eines beliebigen Systemaufrufes ist es für das Betriebssystem nicht mehr möglich, zu dem Programm zu wechseln, welches den Systemaufruf abgesetzt hat.
- ☐ Durch einen Systemaufruf wechselt das Betriebssystem von der Systemebene auf die Benutzerebene, um unprivilegierte Operationen ausführen zu können.
- ☐ Mit Hilfe von Systemaufrufen kann ein Benutzerprogramm privilegierte Operationen durch das Betriebssystem ausführen lassen, die es im normalen Ablauf nicht selbst ausführen dürfte.
- ☐ Benutzerprogramme dürfen keine Systemaufrufe absetzen, diese sind dem Betriebssystem vorbehalten.

k) Welche der folgenden Aussagen über UNIX-Dateisysteme ist richtig?

2 Punkte

- ☐ Hard-links auf Dateien können nur innerhalb des Dateisystems angelegt werden, in dem auch die Datei selbst liegt.
- ☐ Wenn der letzte symbolic link, der auf eine Datei verweist, gelöscht wird, wird auch der zugehörige Dateikopf (inode) gelöscht.
- ☐ Auf eine Datei in einem Dateisystem verweisen immer mindestens zwei hard-links.
- ☐ In einem Verzeichnis darf es mehrere Einträge mit dem selben Namen geben, falls diese Einträge auf unterschiedliche Dateiköpfe (inodes) verweisen.

2) Mehrfachauswahlfragen (8 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

```
static int a = 81034;
int main(int argc, char *argv[]) {
  static int b;
  int c;
  int (*d)(int, char **) = main;
  long *e = malloc(800);
  argc++;
  // ...
}
```

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

- ○ Die Adresse `e` liegt auf dem Stack.
- ○ Die Anweisung `argc++` ändert den Wert von `argc` und beeinflusst somit den Aufrufer.
- ○ `e` zeigt auf ein Array, in dem Platz für 800 Ganzzahlen vom Typ **`long`** ist.
- ○ Die in `e` nach der Zuweisung enthaltene Speicheradresse kann problemlos verwendet werden.
- ○ Das Ergebnis des Aufrufs der Funktion `main` wird in `d` gespeichert.
- ○ `c` ist uninitialisiert und enthält einen undefinierten Wert.
- ○ `c` verliert beim Rücksprung aus `main` seine Gültigkeit.
- ○ `b` ist mit 0 initialisiert und liegt im BSS-Segment.

b) Welche der folgenden Aussagen zum Thema Einplanungsverfahren sind richtig?

4 Punkte

- ○ First-Come-First-Served ist nur bei lang laufenden Aufträgen sinnvoll einsetzbar.
- ○ Shortest-Process-Next (SPN) ist nur theoretisch interessant, weil die Länge der nächsten Rechenphase (CPU-Stoß) in der Praxis nicht abgeschätzt werden kann.
- ○ Zur Realisierung von verdrängenden Einplanungsverfahren wird Hardwareunterstützung durch eine MMU benötigt.
- ○ Round-Robin benachteiligt E/A-intensive Prozesse zu Gunsten von rechenintensiven Prozessen
- ○ Prioritäten-basierte Verfahren sind auch für interaktiven Betrieb gut geeignet.
- ○ Virtual-Round-Robin benachteiligt E/A-intensive Prozesse zu Gunsten von rechenintensiven Prozessen.
- ○ Shortest-Process-Next (SPN) basiert auf einer Heuristik, die die erwartete Länge des nächsten Rechenstoßes der Prozesse verherzusagt.
- ○ Strategien die eine Multilevel-Feedback-Queue (MLFQ) verwenden, sind eine Erweiterung von Round-Robin um höhere Prioritätsebenen.

Aufgabe 2: dir - Directory Information Retriever (60 Punkte)

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

Schreiben Sie ein Programm `dir`, das auf dem TCP/IPv6-Port 2025 (`LISTEN_PORT`) einen Dienst anbietet, über den ein Benutzer Dateien vom Server herunterladen und Verzeichnisse des Servers auflisten kann. Die Abarbeitung parallel eintreffender Anfragen soll von einem Arbeiter-Thread-Pool aus initial 3 (`DEFAULT_THREADS`) Arbeiter-Threads übernommen werden; die Threads werden über einen entsprechend synchronisierten Ringpuffer mit Verbindungen versorgt. Der verwendete Ringpuffer soll hierbei maximal 64 (`BB_SIZE`) Einträge speichern können.

Ein Client sendet nach erfolgreicher Verbindung eine Zeile, die den angeforderten Pfad enthält. Zur Vereinfachung dürfen Sie davon ausgehen, dass eine Zeile aus maximal 256 (`MAX_LINE`) Zeichen besteht. Nach Einlesen der Zeile sendet der Server die angeforderte Datei oder das angeforderte Verzeichnis an den Client.

Das Programm soll folgendermaßen strukturiert sein:

- Das Hauptprogramm initialisiert zunächst alle benötigten Datenstrukturen, startet die benötigte Anzahl an Arbeiter-Threads und nimmt auf einem Socket Verbindungen an. Eine erfolgreich angenommene Verbindung soll zur weiteren Verarbeitung in den Ringpuffer eingefügt werden. Nutzen Sie hierzu den aus der Übung bekannten Ringbufer `jbuffer`.
- Funktion `void* thread_worker(void *arg)`:
  Hauptfunktion der Arbeiter-Threads. Entnimmt in einer Endlosschleife dem Ringpuffer eine Verbindung, und ruft – falls es sich um "normale" Dateideskriptoren handelt (s.u.) – die Funktion `handle_connection` zur weiteren Verarbeitung auf. Achten Sie darauf, dass während der Beantwortung von Clientanfragen aufgetretene Fehler (bspw. nicht vorhandene Dateien oder fehlende Zugriffsrechte) nicht zur Terminierung des Servers führen dürfen.
- Funktion `void handle_connection(int clientSock)`:
  Liest den Pfad (= eine Zeile) vom Client und sendet den Inhalt an den Client.
  - Verzeichnis: Rekursives Auflisten aller Dateien und Unterverzeichnisse via `dump_dir`
  - Reguläre Datei: Dateiinhalt an Client senden
  - Sonst: Anfrage ignorieren

  Zur Vereinfachung dürfen Sie davon ausgehen, dass kein Client Zeilen länger als `MAX_LINE` Zeichen sendet.
- Funktion `void dump_dir(FILE *fh, char *path)`:
  Schreibt den Inhalt des übergebenen Verzeichnisses (und aller Unterverzeichnisse) auf den übergebenen Dateizeiger `fh`.

Mithilfe der Signale `SIGUSR1` (bzw. `SIGUSR2`) soll die Anzahl der laufenden Arbeiter-Threads um einen Thread erhöht respektive verringert werden. Hierzu fügt der jeweilige Signalhandler die Werte `BIRTH` (`POISON`) in den Ringpuffer ein; das Erzeugen (Terminieren) von Threads soll in der Funktion `thread_worker` durchgeführt werden. Zur Vereinfachung dürfen Sie den Fall "kein Arbeiter-Thread läuft mehr" ignorieren.

Achten Sie bei Ihrer Implementierung auf korrekte und sinnvolle Fehlerbehandlung. Insbesondere dürfen Fehler während der Abhandlung von Clientanfragen nicht zum Abbruch des Programms führen. Behandeln Sie `SIGPIPE` zur Vereinfachung nicht.

```
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <signal.h>
#include <pthread.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>

// Prototypen des Ringpuffers
BNDBUF* bbCreate(size_t size);
void bbPut(BNDBUF *bb, int value);
int bbGet(BNDBUF *bb);

// Konstanten, Hilfsfunktionen
#define MAX_LINE 256
#define BB_SIZE 64
#define LISTEN_PORT 2017
#define DEFAULT_THREADS 3
#define POISON (-1)
#define BIRTH  (-2)

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

// Funktionsdeklarationen, globale Variablen usw.

```
// Signalhandler
```

```
// Funktion main()
```

```
    // Threads starten
```

```
    // Signalbehandlung aufsetzen
```

```
    // Auf eingehende Verbindungen warten
```

```
// Ende Funktion main()
```

M:

```
// Funktion dump_dir()
```

```
// Ende Funktion dump_dir()
```

D:

```
// Funktion handle_connection()
```

```
// Ende Funktion handle_connection()
```

H:

```
// Funktion thread_worker()
```

```
// Ende Funktion thread_worker
```

T:

Aufgabe 3: Fehler und Ausnahmen (12 Punkte)

Gegeben ist folgende C-Funktion, die Programmierfehler enthält.

```
int *new_array(size_t size, int value) {
    int *array = calloc(size, sizeof(*array));
    for (size_t i = 0; i <= size; i++)
        array[i] = value;
    return array;
}
```

1) Welcher der Fehler wird -wenn er auftritt- zuverlässig von der Hardware erkannt? (1 Punkt)

- calloc gibt 0 zurück => ungültiger Speicherzugriff

2) Welche Hardwarekomponente ist dafür verantwortlich und wie wird die Ausnahme dem Betriebssystem signalisiert? (2 Punkte)

- MMU, Trap

3) Aunahmesituationen lassen sich in die Kategorien *Trap* und *Interrupt* einteilen. (6 Punkte)

a) Beschreiben Sie, wodurch Ausnahmesituationen der einzelnen Kategorien entstehen. (2 Punkte)

- Trap: Durch das laufende Programm
- Int.: externe Hardware

b) Geben Sie je ein Beispiel pro Kategorie. (2 Punkte)

- Trap: Systemaufruf
- Int.: Daten von Festplatte kommen an, Taste auf Tastatur wird gedrückt, Netzwerkpaket kommt an, Timer-Interrupt

c) Nennen Sie zwei Eigenschaften, in denen sich die Kategorien unterscheiden. (2 Punkte)

- T: det., synch.
- Int.: nichtdet., asynch.

4) Ein weiterer Fehler kann nicht so zuverlässig durch die Hardware entdeckt werden. Nennen Sie die fehlerhafte Stelle und begründen Sie mit einem Beispiel weshalb der Fehler von der Hardware unerkannt bleiben kann. (3 Punkte)

- out-of-bounds-Zugriff in Zeile 4
- Speicher liegt trotzdem im Bereich des Prozesses
- Beispiel: Addresse liegt im internen Verschnitt

Aufgabe 4: Koordinierung (18 Punkte)

1) Zur Koordinierung von nebenläufigen Vorgängen, die auf gemeinsame Betriebsmittel zugreifen, unterscheidet man zwischen einseitiger und mehrseitiger Synchronisation. Wie unterscheidet sich einseitige von mehrseitiger Synchronisation? (2 Punkte)

- einseitige Synchronisation: Ein Vorgang schließt einen anderen aus
- mehrseitige Synchronisation: Vorgänge schließen sich gegenseitig aus

2) Betriebsmittel lassen sich in zwei Kategorien einteilen. Nennen und beschreiben Sie diese und geben Sie je zwei Beispiele. (6 Punkte)

- wiederverwendbar:
Werden belegt/freigegeben
Beispiele: Prozessor, Speicher, kritischer Abschnitt
- konsumierbar:
Werden erstellt/zerstört
Beispiele: Signale, Nachrichten, Strom (?)

3) Erläutern Sie das Konzept Semaphor. Welche Operationen sind auf Semaphoren definiert und was tun diese Operationen? (5 Punkte)

synchronisierter Zähler

- up/V: Inkrementieren um 1, Eventuell wartende Threads freigeben
- down/P: Dekrementieren, wenn Zähler > 0, sonst warten

4) Skizzieren Sie in Programmiersprachen-ähnlicher Form, wie mit Hilfe von zählenden Semaphoren das folgende Szenario korrekt synchronisiert werden kann: Zu jedem Zeitpunkt müssen so viele Threads wie möglich, maximal jedoch 4, die Funktion `threadFunc` ausführen. Dabei soll die Ergebnisausgabe in der jeweils ausgeführten Funktion `doWork` zusätzlich serialisiert werden. Ihnen stehen dabei folgende Semaphor-Funktionen zur Verfügung: (5 Punkte)

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

Beachten Sie, dass nicht unbedingt alle freien Zeilen für eine korrekte Lösung nötig sind. Kennzeichnen Sie durch `/`, wenn Ihre Lösung in einer freie Zeile keine Operation benötigt.

Hauptthread:

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

 s = semCreate(1);
-----------------------------------------
 p = semCreate(4);
-----------------------------------------
  while(1) {

    P(p);
-----------------------------------------
  startWorkerThread(threadFunc);


-----------------------------------------
  }


-----------------------------------------
}
```

Arbeiterthread:

```
void threadFunc(void) {


-----------------------------------------
  doWork();

    V(p);
-----------------------------------------
}

void doWork(void) {


-----------------------------------------
  int result = doCalculations();

   P(s);
-----------------------------------------
  printf("Result: %d\n", res);

   V(s);
-----------------------------------------
}
```