Karlsruher Institut für Technologie

Betriebssysteme
(Operating Systems)

Prof. Dr. Frank Bellosa
Peter Maucher, M.Sc.
Lukas Werling, M.Sc.

Nachname/ *Last name*	Vorname/ *First name*	Matrikelnr./ *Matriculation no*

Hauptklausur
01.03.2023

- Bitte tragen Sie zuerst auf dem Deckblatt Ihren Namen, Ihren Vornamen und Ihre Matrikelnummer ein. Tragen Sie dann auf den anderen Blättern (auch auf Konzeptblättern) Ihre Matrikelnummer ein.

  Please fill in your last name, your first name, and your matriculation number on this page and fill in your matriculation number on all other (including draft) pages.

- Die Prüfung besteht aus 29 Blättern: Einem Deckblatt, 25 Aufgabenblättern mit insgesamt 5 Theorieaufgaben und 3 Programmieraufgaben sowie 3 Seiten Man-Pages.

  The examination consists of 29 pages: One cover sheet, 25 sheets containing 5 theory assignments as well as 3 programming assignments, and 3 sheets with man pages.

- Es sind keinerlei Hilfsmittel erlaubt!

  No additional material is allowed!

- Die Prüfung ist nicht bestanden, wenn Sie versuchen, aktiv oder passiv zu betrügen.

  You fail the examination if you try to cheat actively or passively.

- Sie können auch die Rückseite der Aufgabenblätter für Ihre Antworten verwenden. Wenn Sie zusätzliches Konzeptpapier benötigen, verständigen Sie bitte die Klausuraufsicht.

  You can use the back side of the assignment sheets for your answers. If you need additional draft paper, please notify one of the supervisors.

- Bitte machen Sie eindeutig klar, was Ihre endgültige Lösung zu den jeweiligen Teilaufgaben ist. Teilaufgaben mit mehreren widersprüchlichen Lösungen werden mit 0 Punkten bewertet.

  Make sure to clearly mark your final solution to each question. Questions with multiple, contradicting answers are void (0 points).

- Programmieraufgaben sind gemäß der Vorlesung in C zu lösen.

  Programming assignments have to be solved in C.

Die folgende Tabelle wird von uns ausgefüllt!
The following table is completed by us!

Aufgabe	T1	T2	T3	T4	T5	P1	P2	P3	Total
Max. Punkte	9	9	9	9	9	15	15	15	90
Erreichte Punkte									

Aufgabe T1: Grundlagen

Assignment T1: Basics

a) Beschreiben Sie den Unterschied zwischen Programm und Prozess. **1 pt**

Describe the difference between program and process.

b) Nennen Sie zwei Arten privilegierter CPU-Instruktionen und erklären Sie jeweils, warum diese Instruktionen privilegiert sein müssen. **2 pt**

Name two types of privileged CPU instructions. For each type, explain why these instructions must be privileged.

c) Warum führt Programmed I/O üblicherweise zu hoher Prozessorauslastung, wenn das Betriebssystem schnell auf Eingaben von Geräten reagieren soll? **1 pt**

Why does programmed I/O cause high CPU utilization if the OS should quickly react to device input?

d) Erklären Sie den Unterschied zwischen freiwilligen und unfreiwilligen Aufrufen des Betriebssystems und nennen Sie je ein Beispiel. **2 pt**

Explain the difference between voluntary and involuntary calls of the operating system and name one example each.

e) Beim Kopieren einer großen Datei von einer SSD auf eine Festplatte liegt die Kopiergeschwindigkeit für einige Zeit deutlich über der maximalen Schreibgeschwindigkeit der Festplatte und fällt dann abrupt auf das Maximum ab. Wie erklären Sie diese Beobachtung? **2 pt**

While copying a large file from an SSD to an HDD the copy speed greatly exceeds the HDD's maximum write speed and then suddenly drops to the maximum. How do you explain this?

f) Nennen Sie die zwei grundsätzlichen Kommunikationsmodelle der Interprozesskommunikation. Hinweis: Gefragt sind nicht die verschiedenen Designparameter, sondern die grundlegenden Modelle. **1 pt**

Enumerate the two fundamental models of interprocess communication (IPC). Note: We do not ask for the various design parameters, but for the fundamental models.

Total: 9.0pt

Aufgabe T2: Prozesse und Threads

Assignment T2: Processes and Threads

a) Erläutern Sie den Unterschied zwischen Nebenläufigkeit und Parallelismus. **2 pt**

Explain the difference between concurrency and parallelism.

Kreuzen Sie die Konzepte an, die zur Implementation von Multiprogrammierung verwendet werden können. **0.5 pt**

Tick the concepts that may be used to implement multiprogramming.

Nebenläufigkeit/ *Concurrency*	Parallelismus/ *Parallelism*
☐	☐

b) Nennen Sie ein Beispiel für einen Daemon auf einem typischen Computersystem. **1 pt**

Name an example for a daemon on a typical computer system.

c) Auf POSIX-Systemen können neue Prozesse mittels `fork()` und `exec()` gestartet werden. Derselbe Vorgang kann auch mittels `posix_spawn()` in einem einzelnen Systemaufruf durchgeführt werden. Nennen Sie zwei Vorteile gegenüber der Verwendung von `fork()` und `exec()`. **2 pt**

New processes can be started on POSIX systems by using `fork()` and `exec()`. The same result may also be achieved with a single system call by using `posix_spawn()` instead. Name two advantages of using `posix_spawn()` over the combination of `fork()` and `exec()`.

d) Wann wird für `malloc()` ein Systemaufruf ausgeführt, wann nicht? **1 pt**

In which cases is it required to perform a system call for a `malloc()` operation? When is it not required?

Wie heißt der Systemaufruf, der hierfür auf POSIX-Systemen verwendet wird? **0.5 pt**

What is the name of the system call that is used on POSIX systems for that purpose?

e) Typische Multithreading-Bibliotheken wie `pthreads` verwenden ausschließlich Kernel-Level-Threads, um Parallelismus zu ermöglichen. In einigen jüngeren Programmiersprachen (bspw. Go) ist es gängig, stattdessen auf hybrides Multithreading gemäß dem M-zu-N-Modell zu setzen. Erklären Sie, wie ein derartiger Ansatz die Performance von Anwendungsprogrammen verbessern kann. **2 pt**

Typical multithreading libraries such as `pthreads` exlusively use kernel-level threads to enable parallelism. In several recent programming languages (e.g., Go) it is common to employ hybrid multithreading according to the M-to-N model instead. Explain how such an approach may improve the performance of user applications.

Total: 9.0pt

Aufgabe T3: Speicher

Assignment T3: Memory

a) Nennen Sie neben mehrstufigen Seitentabellen zwei weitere Übersetzungsverfahren von virtuellen auf physische Adressen und jeweils einen Anwendungsfall wofür dieses Verfahren gut geeignet ist. **2 pt**

Other than multi-level page tables, name two other mechanisms that translate virtual to physical addresses, and for each name a use case the mechanism is well-catered for.

b) Nennen Sie vier Informationen, die pro Page in in der Seitentabelle gespeichert werden. **2 pt**

State four pieces of information stored in the page table per page.

c) Nennen Sie zwei Gründe für einen Page Fault in einem Prozess, nach welchen der Prozess weiterlaufen kann. **1 pt**

Name two reasons for page faults which do not lead to termination of the process.

d) Beschreiben Sie das Phänomen Thrashing. **2 pt**

Describe the phenomenon thrashing.

e) Gegeben sei ein System, das virtuelle Adressen mittels einer dreistufigen Seitentabelle übersetzt. Die Seitengröße beträgt 4 KiB. Das System ist außerdem mit einem TLB ausgestattet, der 1024 Einträge fasst. Ein Cache ist nicht vorhanden. **2 pt**

Auf diesem System wird ein Prozess ausgeführt, der innerhalb eines zusammenhängenden 3 MiB-Speicherbereichs zufällige Anfragen macht. Als Sie den Speicherbereich auf 5 MiB vergrößern, messen Sie bei einigen Speicherzugriffen größere Latenzen. Woran liegt das? Bitte geben Sie die Rechnung an.

Consider a system which translates virtual addresses with a three-level page table. The page size is 4 KiB. Additionally, the system includes a TLB. There are no CPU caches.

The system runs a process which randomly accesses memory in a 3 MiB area of contiguous memory. After you increased the memory area to 5 MiB, some memory accesses show higher latency. What is the reason? Please note down your calculations.

Total: 9.0pt

Aufgabe T4: Koordination und Kommunikation von Prozessen

Assignment T4: Process Coordination and Communication

a) Nennen Sie die vier notwendigen Bedingungen für einen Deadlock. **2 pt**

Name the four necessary deadlock conditions.

b) Datenbanksysteme ermöglichen es Anwendungen, große Datenmengen strukturiert zu speichern und abzufragen. Prozesse kommunizieren mit einem Datenbanksystem mittels sogenannter *Transaktionen*. Innerhalb einer Transaktion können beliebig viele Datensätze abgerufen und modifiziert werden. Beim Abrufen wird ein Datensatz automatisch gesperrt, sodass parallel ablaufende Transaktionen warten müssen, bevor sie auf den Datensatz zugreifen können. Erst beim Beenden einer Transaktion (*Commit*) werden alle Locks wieder freigegeben. Im Folgenden wird eine vereinfachte Pseudo-Abfragesprache verwendet, um den Ablauf von Transaktionen anzugeben. Diese Befehle stehen zur Verfügung:

- `START TRANSACTION` initiiert eine neue Transaktion.
- `SELECT x` ruft den Datensatz `x` aus der Datenbank ab und sperrt ihn.
- `UPDATE x SET y` aktualisiert den Datensatz `x` in der Datenbank auf den Wert `y` (ohne ihn dabei zu entsperren).
- `COMMIT` beendet eine Transaktion und gibt alle gehaltenen Locks frei.

Database systems enable applications to store and retrieve large amounts of data in a structured manner. Processes can communicate with a database system using transactions*. Within a transaction, the client may read and modify an arbitrary amount of datasets. When a dataset is read, it is locked automatically. In turn, other transactions running at the same time will need to wait before they can access the dataset. All locks are freed as soon as a transaction is completed (*commit*). In this task, we will use a simplified query language to specify transaction sequences. The following commands are available:*

- *`START TRANSACTION` initiates a new transaction.*
- *`SELECT x` retrieves the dataset with the ID `x` from the database and locks it.*
- *`UPDATE x SET y` updates the dataset `x` in the database and sets its value to `y`. Note that `UPDATE` does not unlock datasets.*
- *`COMMIT` finishes a transaction and frees all locks held by it.*

Zwei Prozesse A und B führen die folgenden Transaktionen aus.

Two processes A and B execute the following transactions.

Process A

```
1  START TRANSACTION
2  SELECT 7
3  UPDATE 7 SET 3
4  COMMIT
5
6  START TRANSACTION
7  SELECT 4
8  SELECT 10
9  UPDATE 10 SET 1
10 SELECT 5
11 SELECT 7
12 UPDATE 7 SET 1337
13 COMMIT
```

Process B

```
1  START TRANSACTION
2  SELECT 8
3  SELECT 7
4  SELECT 5
5  UPDATE 7 SET 1338
6  COMMIT
```

Im dargestellten Szenario kann bei paralleler Ausführung ein *Deadlock* zwischen den Prozessen A und B auftreten. Zeichnen Sie basierend auf dem Szenario einen *Resource Allocation Graph* (RAG), der den Zustand zum Zeitpunkt eines aufgetretenen *Deadlock* illustriert. Achten Sie darauf, dass alle gehaltenen Ressourcen eingezeichnet sind. **3.5 pt**

The scenario above may cause a deadlock *between the processes A and B when they are executed in parallel. Draw a* resource allocation graph *(RAG) based on the scenario that illustrates the locking state when a deadlock has occurred. Make sure to draw all held resources.*

c) Beschreiben Sie präzise, wie Sie **eine** der Transaktionen so verändern würden, dass kein Deadlock mehr auftreten kann. Entfernen Sie dabei keine Operationen und verändern Sie nicht die Semantik einzelner Operationen. Markieren Sie eindeutig, welche der Transaktionen Sie verändern. **1 pt**

*Accurately describe how you would modify **one** of the transactions to prevent deadlocks from occurring. Do not remove any operations and do not modify the semantics of individual operations. Clearly indicate the transaction that you modify.*

Prozess A, erste Transaktion/ *Process A, first transaction*	Prozess A, zweite Transaktion/ *Process A, second transaction*	Prozess B/ *Process B*
☐	☐	☐

d) Welche der vier notwendigen Bedingungen für einen Deadlock haben Sie durch das Verändern der Transaktion in der vorherigen Teilaufgabe gebrochen? **1 pt**

Which of the four necessary deadlock conditions did you break by modifying the transaction in the previous question?

e) Das Datenbanksystem führt automatisch einen RAG und nutzt diesen, um Deadlocks zu erkennen. Im Falle eines aufgetretenen Deadlocks wird eine der beteiligten Transaktionen abgebrochen und die vorgenommenen Änderungen rückgängig gemacht. Die andere(n) Transaktion(en) dürfen dann weiterlaufen. Welche der Transaktionen sollte im obigen Szenario sinnvollerweise abgebrochen werden? **0.5 pt**

The database system automatically maintains an RAG to detect deadlocks. In case a deadlock has occurred, it aborts one of the involved transactions and rolls all its changes back. The other transactions may then continue to execute. Which of the transactions should reasonably be aborted in the given scenario?

Prozess A, erste Transaktion/ *Process A, first transaction*	Prozess A, zweite Transaktion/ *Process A, second transaction*	Prozess B/ *Process B*
☐	☐	☐

Warum? **1 pt**

Why?

Total: 9.0pt

Aufgabe T5: I/O, Hintergrundspeicher und Dateisysteme

Assignment T5: I/O, Secondary Storage, and File Systems

a) Anwendungen nutzen häufig Code ähnlich dem unten gegebenen, um ihre Dateien zu speichern.

Applications often use code similar to that in the snippet below for saving files.

```
1 /* open new temporary file */
2 fd = open("config.tmp", O_WRONLY | O_CREAT);
3 /* write new file contents */
4 write(fd, data, data_size);
5 /* close the temporary file */
6 close(fd);
7 /* replace the original file */
8 rename("config.tmp", "config.txt");
```

Das System stürzt nach dem Abschließen des Speichervorgangs ab. Nach einem Neustart stellen Sie fest, dass die Datei `config.txt` existiert, aber leer ist (d.h. Länge 0 hat) und somit die geschriebenen Daten verloren gingen. Wie kann es dazu kommen? **1.5 pt**

The system crashes after completing the save operation. After a restart, you notice that the file `config.txt` exists, but is empty (i.e., has length 0), thus losing the written data. How can this happen?

Beheben Sie das oben beschriebene Problem durch Einfügen einer einzelnen Codezeile. **1 pt**

Fix the problem described above by inserting a single line of code.

Code:

Insert after line:

Wie hilft *Journaling* dabei, das beschriebene Problem zu verhindern? **1 pt**

How does journaling *help prevent the described problem?*

b) Das untenstehende Shellskript wird auf einem Linux-System ausgeführt. Welchen Dateiinhalt haben die einzelnen Dateien danach? Füllen Sie die Tabelle aus. Schreiben Sie „X“, falls auf eine Datei nicht zugegriffen werden kann. **2.5 pt**

The shell script below is run on a Linux system. What content does each file have afterwards? Fill in the table. Write "X" if a file cannot be accessed.

```
# -n: no newline
echo -n A > a
echo -n B > b
ln a c
ln c d
# -s: symlink
ln -s b e
# >>: append to file
echo -n 1 >> c
# mv source dest
mv b c
echo -n 2 >> d
echo -n 3 >> a
```

File	Contents
a	
b	
c	
d	
e	

c) Welche drei Benutzerklassen kennt das traditionelle Modell der UNIX-Zugriffskontrolle? **1 pt**

Which three classes of users are present in the traditional UNIX access control model?

d) Was ist der wichtigste Vorteil einer File Allocation Table (FAT) gegenüber Chained Allocation? **1 pt**

What is the most important advantage of a file allocation table (FAT) compared to Chained Allocation?

e) Welche Datenstrukturen durchläuft das Betriebssystem, um einen Dateideskriptor zu einer Inode aufzulösen? **1 pt**

What data structures does the operating system traverse to resolve a file descriptor to an inode?

Total: 9.0pt

Aufgabe P1: C-Grundlagen

Assignment P1: C Basics

a) In dem untenstehenden Code haben sich 5 Fehler eingeschlichen. Markieren Sie die fehlerhaften Zeilen mit einem X und korrigieren Sie den Code. Gehen Sie von einem 64-Bit-System aus. **5 pt**

There are 5 errors in the code below. Mark the incorrect lines with an X and correct the code. Assume a 64-bit system.

```
struct meta;
typedef struct meta meta;

size_t size(meta *buffer);
size_t capacity(meta *buffer);
```

```
struct meta {
    size_t element_size;
    void *begin;
    void *end;
    void *capacity;
};
```

```
☐ meta *allocate_meta(size_t element_size) {
☐     meta *pointer = malloc(sizeof(meta));
☐     if (!pointer) return 0;
☐     pointer = (meta) {element_size, 0, 0, 0};
☐     return pointer;
☐ }

☐ bool append(meta *buffer, void *data) {
☐     if(!data) return false;
☐     size_t buff_size = size(&buffer);
☐     size_t buff_bytes =
☐         buff_size * buffer.element_size;
☐     size_t buff_cap = capacity(buffer);
☐     if (buff_size >= buff_cap) {
☐         size_t new_bytes =
☐             (buff_size * 2 + 1) * buffer->element_size;
☐         void *new_begin = malloc(new_bytes);
☐         if (!new_begin) return false;
☐         void *new_end = (char *)new_begin + buff_bytes;
☐         void *new_capacity =
☐             (char *)new_begin + new_bytes;
☐         memcpy(new_begin, buffer->begin, buff_bytes);
☐         buffer->begin = new_begin;
☐         buffer->end = new_end;
☐         buffer->capacity = new_capacity;
☐     }
☐     memcpy(buffer->end, data, buffer->element_size);
☐     buffer->end =
☐         (char *)buffer->end + buffer->element_size;
☐     return true;
☐ }
```

```
void *get(meta *buffer, size_t where) {
    (char *)buffer->begin +
        where * buffer->element_size;
}
size_t size(meta *buffer) {
    return ((char *)buffer->end - (char *)buffer->begin) /
        buffer->element_size;
}
size_t capacity(meta *buffer) {
    return ((char *)buffer->capacity - (char *)buffer->begin) /
        buffer->element_size;
}
```

Welche Datenstruktur wird hier implementiert? **0.5 pt**

Which data structure is implemented here?

Schreiben Sie eine Funktion `free_meta()`, die ein `meta`-Objekt samt Daten freigibt. **1.5 pt**

Write a function `free_meta()`, which frees a `meta` object including its data.

```
void free_meta(meta *buffer) {

}
```

b) Was ist der Wert der Ausdrücke nach Ausführung des Programmschnipsels? **2 pt**

What is the value of the expressions after execution of the given code snippet?

```
uint32_t a[] =
  { 0x00112233, 0x44556677,
    0x8899aabb, 0xccddeeff,
    0xff00ff00, 0x0123abcd };

uint16_t b = a[2] - a[1];
uint32_t c = b >> 2;
// <-- Evaluate expression here
```

Expression	Value
`a[2] ^ a[2]`	
`a[4] & a[5]`	
`b || ~b`	
`c`	

c) Nennen Sie die 3 Verwendungen des Zeichens $\star$ in C. **1.5 pt**

List the three usages of the symbol $\star$ in C.

d) Eine ABI kann verschiedene Orte spezifizieren, um Parameter beim Funktionsaufruf zwischenzuspeichern. Nennen Sie zwei davon. **1 pt**

Where can the ABI specify to store arguments during a function call? Name two options.

e) Warum ist dieser Code gefährlich, und wie könnte man das beheben? **1.5 pt**

Why is this code dangerous, and how to fix it?

```
int *ptr = some_pointer_returning_function();
int storage = (int)ptr;
// some time later
int *new_pointer = (int*)storage;
int value = *new_pointer;
```

f) Welche Zahlenwerte haben die folgenden `enum`-Werte? **1 pt**

Which integer values are represented by the following `enum` values?

```
enum JUST_A_NAME {
  FOO,
  BAR,
  BAZ = 15,
  ALICE,
  BOB,
};
```

Expression	Value
BAR	
ALICE	

g) Beschreiben Sie, wie sich die Länge einer Zeichenkette (`char*`) ohne die Verwendung von Hilfsfunktionen wie `strlen()` bestimmen lässt. **1 pt**

Explain how the length of a string (`char*`) can be determined without using any helper methods such as `strlen()`.

Total: 15.0pt

Aufgabe P2: Datenbankserver

Assignment P2: Database Server

In dieser Aufgabe sollen Sie einen einfachen Datenbankserver implementieren. Der Server nimmt Verbindungen von Prozessen entgegen und erzeugt einen neuen Thread für jede Verbindung. In diesem Thread werden die auf der Verbindung eingehenden Befehle abgearbeitet. Die Kommunikation läuft in sogenannten *Transaktionen* ab. Innerhalb einer Transaktion können beliebig viele Datensätze abgerufen und modifiziert werden. Beim Abrufen wird ein Datensatz automatisch gesperrt, sodass parallel ablaufende Transaktionen warten müssen, bevor sie auf den Datensatz zugreifen können. Erst beim Beenden einer Transaktion (*Commit*) werden alle Locks wieder freigegeben. Während echte Datenbanksysteme komplexe Tabellenstrukturen abbilden können, nehmen wir in dieser Aufgabe an, dass genau 32 einfache Integer-Werte gespeichert werden. Die unterstützten Befehle sind dieselben wie in Aufgabe T4:

- `START TRANSACTION` initiiert eine neue Transaktion.
- `SELECT x` ruft den Datensatz `x` aus der Datenbank ab und sperrt ihn.
- `UPDATE x SET y` aktualisiert den Datensatz `x` in der Datenbank auf den Wert `y` (ohne ihn dabei zu entsperren).
- `COMMIT` beendet eine Transaktion und gibt alle gehaltenen Locks frei.

- Geben Sie vom Betriebssystem angeforderte Ressourcen (z. B. Speicher) explizit zurück.
- Binden Sie die in den Teilaufgaben notwendigen C-Header in dem gekennzeichneten Bereich ein.
- Sofern nicht anderweitig bestimmt, gehen Sie davon aus, dass (1) bei Systemaufrufen und Speicherallokationen keine Fehler auftreten, (2) Systemaufrufe nicht durch Signale unterbrochen werden und (3) Funktionsparameter valide sind.

In this assignment, you will implement a simple database server. The server accepts incoming connections from other processes and creates a new thread for every connection. This thread handles the commands sent via the associated connection. The server communicates with the clients via so-called transactions*. Within a transaction, the client may retrieve and modify an arbitrary amount of datasets. When a dataset is retrieved it is locked automatically, so that other transactions running in parallel will have to wait before accessing the dataset. Only when a transaction is finished (*committed*), all locks are freed again. While real-world database systems may store complex table structures, we assume for this task that exactly 32 simple integer values may be stored. The supported commands are the same as in assignment T4:*

- `START TRANSACTION` *initiates a new transaction.*
- `SELECT x` *retrieves the dataset* `x` *from the database and locks it.*
- `UPDATE x SET y` *updates the dataset* `x` *to the value* `y` *in the database (without unlocking it).*
- `COMMIT` *terminates a transaction and frees all held locks.*

- *Explicitly return allocated operating system resources (e.g., memory).*

- *Include necessary C headers in the marked area.*
- *Unless stated otherwise, assume that (1) system calls and memory allocations do not fail, (2) system calls are not interrupted by signals, and (3) function arguments are valid.*

```
/* include statements for the required C headers */
#include <stdbool.h>
#include <pthread.h>

#define NUM_VALUES 32

struct database {
    int values[NUM_VALUES];
    pthread_mutex_t locks[NUM_VALUES];
}
struct database db;

struct conn {
    pthread_t thread;
    int fd;
    bool in_trans;
    bool locked[NUM_VALUES];
}
```

a) Wenn eine neue Verbindung eingeht, sollen deren Befehle auf einem eigenständigen Thread abgearbeitet werden. Eine Verbindung wird durch einen Dateideskriptor (*file descriptor*, `fd`) repräsentiert. Allozieren und initialisieren Sie hierzu zunächst eine Struktur für die Verbindungsdaten (`struct conn`) und übergeben Sie diese dann als Argument an den neu gestarteten Thread. Der Thread soll die Funktion `conn_thread()` ausführen. Geben Sie die Verbindungsdaten-Struktur zurück. **2.5 pt**

When a new connection is established, a new thread shall be spawned to handle its commands. A connection is represented by a file descriptor (`fd`). Start by allocating and initializing a structure for the data associated with the connection (`struct conn`) and then pass it as an argument to the newly created thread. The thread shall execute the function `conn_thread()`. Return the connection data structure.

```
void *conn_thread(void *arg);

struct conn *on_new_connection(int fd) {

}
```

Für jeden Befehl gibt es eine Funktion, welche für dessen Abarbeitung im Rahmen einer Verbindung zuständig ist. Implementieren Sie diese Funktionen analog der im Einleitungstext gegebenen Befehlsbeschreibung. Beachten Sie dabei folgendes:

- Nutzen Sie die vordefinierten Strukturen. Sie müssen keine eigenen Strukturen definieren.
- Die einzige Instanz der Datenbank ist in der Variable `db` vorgegeben.
- Gehen Sie davon aus, dass alle Mutexe und die Verbindungsstruktur korrekt initialisiert sind.
- Geben Sie in allen Funktionen `true` zurück, wenn der Befehl erfolgreich ausgeführt wurde und `false`, wenn ein Fehler aufgetreten ist.
- Prüfen Sie die im Befehl gegebenen Argumente (bspw. `sel_pos`) auf Plausibilität.
- `UPDATE`-Befehle dürfen nur auf zuvor gesperrten Datensätzen ausgeführt werden.
- Sämtliche Befehle außer `START TRANSACTION` dürfen nur innerhalb von Transaktionen ausgeführt werden. `START TRANSACTION` innerhalb einer bereits laufenden Transaktion ist nicht erlaubt.
- `SELECT` darf innerhalb einer Transaktion mehrfach für den gleichen Datensatz ausgeführt werden. In diesem Fall darf der Datensatz nicht erneut gesperrt werden.
- `SELECT` soll den gelesenen Wert über den Dateideskriptor der Verbindung ausgeben. Nach dem Zahlenwert soll zusätzlich ein Zeilenumbruch (`\n`) ausgegeben werden. **Hinweis**: Ziehen Sie die Nutzung einer der Varianten von `printf()` hierfür in Betracht.

A function is defined for each command that is called for its execution during a connection. Implement the corresponding functions based on the definitions given in this task's introductory text. Observe the following restrictions:

- *Make use of the predefined C structures. You do not need to define your own structures.*
- *The only instance of the database is already defined in the variable `db`.*
- *Assume that all mutexes and the connection data structure are correctly initialized.*
- *Return `true` when the command was successful and `false` when an error has occurred.*
- *Check all command arguments (e.g., `sel_pos`) for plausibility.*
- *`UPDATE` commands may only be executed on datasets that were previously locked.*
- *All commands except `START TRANSACTION` may only be executed within a transaction. `START TRANSACTION` within an already running transaction is not allowed.*
- *`SELECT` may be executed on the same dataset multiple times within a single transaction. Do not lock the dataset again in that case.*
- *`SELECT` shall output the retrieved value via the connection's file descriptor. Additionally print a line break (`\n`) after the value. **Hint**: Consider using one of the variants of `printf()`.*

b) Implementieren Sie `do_start_transaction()` für den Befehl `START TRANSACTION`. **1.5 pt**

Implement `do_start_transaction()` for the `START TRANSACTION` command.

```
bool do_start_transaction(struct conn *data) {

}
```

c) Implementieren Sie `do_commit()` für den Befehl `COMMIT`. **3.5 pt**

Implement `do_commit()` for the `COMMIT` command.

```
bool do_commit(struct conn *data) {

}
```

d) Implementieren Sie `do_update()` für den Befehl `UPDATE`. **2.5 pt**

Implement `do_update()` for the `UPDATE` command.

```
bool do_update(struct conn *data, unsigned int upd_pos,
               int upd_value) {
}
```

e) Implementieren Sie `do_select()` für den Befehl `SELECT`. **5 pt**

Implement `do_select()` for the `SELECT` command.

```
bool do_select(struct conn *data, unsigned int sel_pos) {
}
```

Total: 15.0pt

Aufgabe P3: Log-strukturierter Speicher

Assignment P3: Log-Structured Storage

In einem Log-strukturierten Speichersystem werden neue Einträge immer an das Ende des Logs geschrieben. In dieser Aufgabe setzen Sie ein einfaches Log-strukturiertes Speichersystem um.

Jeder Eintrag besteht aus einem Header (`struct record`) und einer variablen Menge an Daten. Die Länge der Daten ist immer ein Vielfaches der Blockgröße `BS`. Jeder Eintrag besitzt eine eindeutige ID, die für gelöschte Einträge auf `ID_INV` gesetzt wird. An der aktuellen Einfügeposition `tail_off` steht immer ein Tail-Block, ein ungültiger Eintrag mit der Länge des verbleibenden freien Speichers.

- Nutzen Sie die Funktionen `write_block()` und `read_block()`, die jeweils einen einzelnen Block vom Hintergrundspeicher schreiben oder lesen.
- Die Funktion `copy_blocks()` kopiert Blöcke sequentiell innerhalb des Hintergrundspeichers.

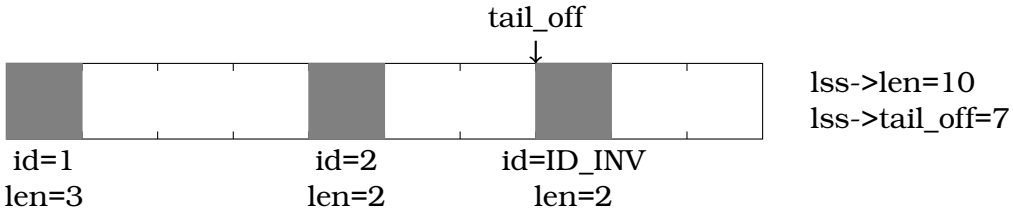
In a log-structured storage system, new entries are always written to the end of the log. In this assignment, you will realize a simple log-structured storage system.

Every entry has a header (`struct record`) and a variable amount of data. The data length is always a multiple of the block size `BS`. Every entry has a unique id which is set to `ID_INV` for deleted entries. There is always a tail block at the current insertion position `tail_off`. A tail block is an invalid entry with length equal to the remaining free space.

- *Use the functions `read_block()` and `write_block()` which read or write a single block from the background storage.*
- *The function `copy_blocks()` copies blocks sequentially within the background storage.*

```
#define BS 512                    /* block size */
#define ID_INV ((uint64_t) -1)    /* id for invalid records */

struct lss {
  uint64_t len;       /* total length of the storage area in blocks */
  uint64_t tail_off;  /* offset of the tail in blocks */
  /* fields for block storage access omitted */
};

struct record {
  uint64_t id;   /* user-defined id, or ID_INV */
  uint64_t len;  /* length of data in blocks after the record header */
  uint32_t type; /* user-defined type */

  char _padding[                                                    ];
};

void write_block(struct lss *lss, uint64_t off, void *mem);
void read_block(struct lss *lss, uint64_t off, void *mem);
void copy_blocks(struct lss *lss, uint64_t target, uint64_t source,
                 uint64_t len);
```

Beispiel mit zwei Einträgen: / *Example with two entries:*



a) Vervollständigen Sie die Deklaration von `_padding`, sodass `struct record` eine Gesamtgröße von `BS` Byte hat. **1 pt**

Complete the declaration of `_padding` so that `struct record` has a total size of `BS` bytes.

b) Vervollständigen Sie die Funktion `ensure_space()`, die sicherstellt, dass mindestens `len` freie Blöcke am Ende des Logs frei sind. **2.5 pt**

- Rufen Sie die Funktion `compact_log()` auf, falls nicht ausreichend viele Blöcke frei sind.
- Falls anschließend immer noch nicht genügend Blöcke frei sind, gibt die Funktion `-ENOSPC` zurück, ansonsten 0.

Complete the function `ensure_space()` which ensures that at least `len` free blocks are available at the tail of the log.

- *Call the function `compact_log()` if there are not enough free blocks.*
- *If the amount of free blocks is still not sufficient, the function returns `-ENOSPC`, otherwise 0.*

```
void compact_log(struct lss *lss);

int ensure_space(struct lss *lss, uint64_t len) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
.....
```

c) Vervollständigen Sie die Funktion `write_tail()`, die wie oben beschrieben einen Tail-Block an das Ende des Logs schreibt. **1.5 pt**

- Sie müssen den alten Tail-Block nicht anfassen.
- Aktualisieren Sie schließlich `tail_off`.

Complete the function `write_tail()`, which writes a tail block at the end of the log, as specified above.

- *You do not need to touch the old tail block.*
- *Finally, update `tail_off`.*

```
void write_block(struct lss *lss, uint64_t off, void *mem);

void write_tail(struct lss *lss, uint64_t off) {
.....

.....

.....

.....

.....

.....

.....

.....

.....

.....
}
.....
```

d) Vervollständigen Sie die Funktion `append_record()`, die einen Eintrag an den Log anhängt. **4.5 pt**

- Stellen Sie zunächst durch einen Aufruf von `ensure_space()` sicher, dass genügend Speicherplatz für den `record`, die Daten und den Tail-Block vorhanden ist. Geben Sie auftretende Fehler zurück, ansonsten 0.
- Schreiben Sie dann die Daten sowie einen neuen Tail-Block.
- Schreiben Sie schließlich den `record`.

Complete the function `append_record()`, which appends an entry to the log.

- *First, by calling `ensure_space()`, make sure that there is enough space left for the `record`, the data, and the tail block. Return any errors that may occur and otherwise 0.*
- *Then, write the data and the new tail block.*
- *Finally, write the `record`.*

```
int ensure_space(struct lss *lss, uint64_t len);
void write_tail(struct lss *lss, uint64_t off);
void write_block(struct lss *lss, uint64_t off, void *mem);

int append_record(struct lss *lss, struct record *rec, void *data) {
  uint64_t off = lss->tail_off;
}
```

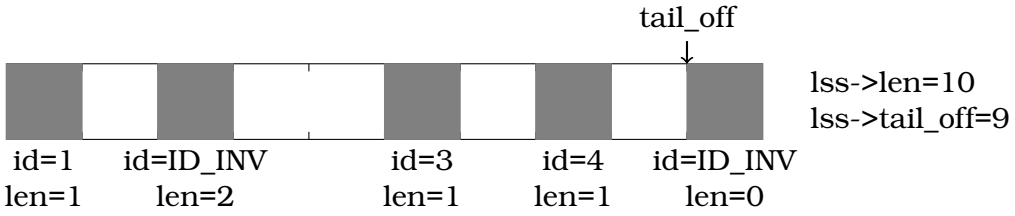
e) Vervollständigen Sie die Funktion `compact_log()`, die ungültige Einträge aus dem Log entfernt und folgende gültige Blöcke verschiebt, um Lücken zu füllen. **4 pt**

- Durchlaufen Sie den Log von dem `record` an Offset 0 aus.
- Aktualisieren Sie zum Schluss den Tail-Block.

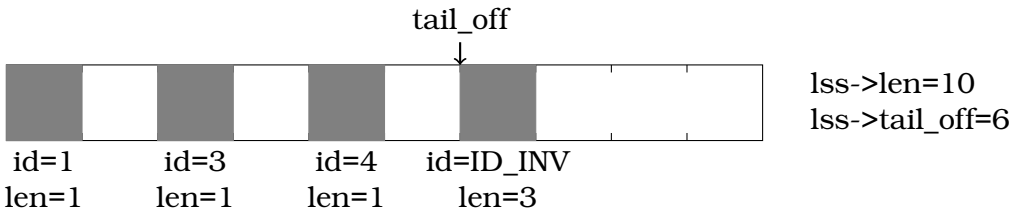
Complete the function `compact_log()`, which removes invalid entries from the log and which moves the following valid blocks to fill any gaps.

- *Walk the log starting from the `record` at offset 0.*
- *Finally, update the tail block.*

Before:



After:




```
void write_tail(struct lss *lss, uint64_t off);
void write_block(struct lss *lss, uint64_t off, void *mem);
void read_block(struct lss *lss, uint64_t off, void *mem);
void copy_blocks(struct lss *lss, uint64_t target, uint64_t source,
                 uint64_t len);

void compact_log(struct lss *lss) {
  struct record rec;
```

```
}
```

f) Warum können Daten verloren gehen, wenn das System während der Ausführung von `compact_log()` abstürzt? Skizzieren Sie eine problematische Situation. **1.5 pt**

Why may there be data loss if the system crashes during execution of `compact_log()`*? Draw a problematic situation.*

Note: Only answers where data is irrevocably lost by overwriting are accepted. **Total: 15.0pt**

NAME

malloc, free – allocate and free dynamic memory

SYNOPSIS

#include <stdlib.h>

void *malloc(size_t *size***);**
void free(void **ptr***);**

DESCRIPTION

The **malloc**() function allocates *size* bytes and returns a pointer to the allocated memory. *The memory is not initialized*. If *size* is 0, then **malloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

The **free**() function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**(). Otherwise, or if *free(ptr)* has already been called before, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

RETURN VALUE

The **malloc**() function return a pointer to the allocated memory, which is suitably aligned for any built-in type. On error, these functions return NULL. NULL may also be returned by a successful call to **malloc**() with a *size* of zero.

The **free**() function returns no value.

ERRORS

malloc() can fail with the following error:

ENOMEM
Out of memory. Possibly, the application hit the **RLIMIT_AS** or **RLIMIT_DATA** limit described in **getrlimit**(2).

NOTES

By default, Linux follows an optimistic memory allocation strategy. This means that when **malloc**() returns non-NULL there is no guarantee that the memory really is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer. For more information, see the description of */proc/sys/vm/overcommit_memory* and */proc/sys/vm/oom_adj* in **proc**(5), and the Linux kernel source file *Documentation/vm/overcommit-accounting*.

Normally, **malloc**() allocates memory from the heap, and adjusts the size of the heap as required, using **sbrk**(2). When allocating blocks of memory larger than **MMAP_THRESHOLD** bytes, the glibc **malloc**() implementation allocates the memory as a private anonymous mapping using **mmap**(2). **MMAP_THRESHOLD** is 128 kB by default, but is adjustable using **mallopt**(3). Prior to Linux 4.7 allocations performed using **mmap**(2) were unaffected by the **RLIMIT_DATA** resource limit; since Linux 4.7, this limit is also enforced for allocations performed using **mmap**(2).

To avoid corruption in multithreaded applications, mutexes are used internally to protect the memory-management data structures employed by these functions. In a multithreaded application in which threads simultaneously allocate and free memory, there could be contention for these mutexes. To scalably handle memory allocation in multithreaded applications, glibc creates additional *memory allocation arenas* if mutex contention is detected. Each arena is a large region of memory that is internally allocated by the system (using **brk**(2) or **mmap**(2)), and managed with its own mutexes.

SUSv2 requires **malloc**() to set *errno* to **ENOMEM** upon failure. Glibc assumes that this is done (and the glibc versions of these routines do this); if you use a private malloc implementation that does not set *errno*, then certain library routines may fail without having a reason in *errno*.

Crashes in **malloc**() or **free**() are almost always related to heap corruption, such as overflowing an allocated chunk or freeing the same pointer twice.

The **malloc**() implementation is tunable via environment variables; see **mallopt**(3) for details.

NAME

memset – fill memory with a constant byte

SYNOPSIS

#include <string.h>

void *memset(void **s*, **int *c*, **size_t** *n*);**

DESCRIPTION

The **memset**() function fills the first *n* bytes of the memory area pointed to by *s* with the constant byte *c*.

RETURN VALUE

The **memset**() function returns a pointer to the memory area *s*.

NAME

memcpy – copy memory area

SYNOPSIS

#include <string.h>

void *memcpy(void **dest*, **const void **src*, **size_t** *n*);**

DESCRIPTION

The **memcpy**() function copies *n* bytes from memory area *src* to memory area *dest*. The memory areas must not overlap. Use **memmove**(3) if the memory areas do overlap.

RETURN VALUE

The **memcpy**() function returns a pointer to *dest*.

NOTES

Failure to observe the requirement that the memory areas do not overlap has been the source of significant bugs. (POSIX and the C standards are explicit that employing **memcpy**() with overlapping areas produces undefined behavior.) Most notably, in glibc 2.13 a performance optimization of **memcpy**() on some platforms (including x86-64) included changing the order in which bytes were copied from *src* to *dest*.

NAME

strlen – calculate the length of a string

SYNOPSIS

#include <string.h>

size_t strlen(const char **s*);

DESCRIPTION

The **strlen**() function calculates the length of the string pointed to by *s*, excluding the terminating null byte ('\0').

RETURN VALUE

The **strlen**() function returns the number of characters in the string pointed to by *s*.

NAME

printf, fprintf, dprintf, sprintf – formatted output conversion

SYNOPSIS

```
#include <stdio.h>

int printf(const char *restrict format, ...);
int fprintf(FILE *restrict stream,
        const char *restrict format, ...);
int dprintf(int fd,
        const char *restrict format, ...);
int sprintf(char *restrict str,
        const char *restrict format, ...);
```

DESCRIPTION

The functions in the **printf**() family produce output according to a *format*. The function **printf**() writes output to *stdout*, the standard output stream; **fprintf**() writes output to the given output *stream*; **sprintf**() writes to the character string *str*.

The function **dprintf**() is the same as **fprintf**() except that it outputs to a file descriptor, *fd*, instead of to a **stdio**(3) stream.

RETURN VALUE

Upon successful return, these functions return the number of characters printed (excluding the null byte used to end output to strings).

If an output error is encountered, a negative value is returned.

NAME

pthread_create – create a new thread

SYNOPSIS

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
        void *(*start_routine) (void *), void *arg);
```

DESCRIPTION

The **pthread_create**() function starts a new thread in the calling process. The new thread starts execution by invoking *start_routine*(); *arg* is passed as the sole argument of *start_routine*().

The new thread terminates in one of the following ways:

* It calls **pthread_exit**(3), specifying an exit status value that is available to another thread in the same process that calls **pthread_join**(3).
* It returns from *start_routine*(). This is equivalent to calling **pthread_exit**(3) with the value supplied in the *return* statement.
* It is canceled (see **pthread_cancel**(3)).
* Any of the threads in the process calls **exit**(3), or the main thread performs a return from *main*(). This causes the termination of all threads in the process.

The *attr* argument points to a *pthread_attr_t* structure whose contents are used at thread creation time to determine attributes for the new thread; this structure is initialized using **pthread_attr_init**(3) and related functions. If *attr* is NULL, then the thread is created with default attributes.

Before returning, a successful call to **pthread_create**() stores the ID of the new thread in the buffer pointed to by *thread*; this identifier is used to refer to the thread in subsequent calls to other pthreads functions.

RETURN VALUE

On success, **pthread_create**() returns 0; on error, it returns an error number, and the contents of **thread* are undefined.

ERRORS

EAGAIN
: Insufficient resources to create another thread.

EINVAL
: Invalid settings in *attr*.

EPERM
: No permission to set the scheduling policy and parameters specified in *attr*.

NOTES

A thread may either be *joinable* or *detached*. If a thread is joinable, then another thread can call **pthread_join**(3) to wait for the thread to terminate and fetch its exit status. Only when a terminated joinable thread has been joined are the last of its resources released back to the system. When a detached thread terminates, its resources are automatically released back to the system: it is not possible to join with the thread in order to obtain its exit status. Making a thread detached is useful for some types of daemon threads whose exit status the application does not need to care about. By default, a new thread is created in a joinable state, unless *attr* was set to create the thread in a detached state (using **pthread_attr_setdetachstate**(3)).

NAME

pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock – lock and unlock a mutex

SYNOPSIS

#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t ***mutex*****);**
int pthread_mutex_trylock(pthread_mutex_t ***mutex*****);**
int pthread_mutex_unlock(pthread_mutex_t ***mutex*****);**

DESCRIPTION

The mutex object referenced by *mutex* shall be locked by calling *pthread_mutex_lock*(). If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner.

The *pthread_mutex_trylock*() function shall be equivalent to *pthread_mutex_lock*(), except that if the mutex object referenced by *mutex* is currently locked (by any thread, including the current thread), the call shall return immediately.

The *pthread_mutex_unlock*() function shall release the mutex object referenced by *mutex*. If there are threads blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock*() is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread shall resume waiting for the mutex as if it was not interrupted.

RETURN VALUE

If successful, the *pthread_mutex_lock*() and *pthread_mutex_unlock*() functions shall return zero; otherwise, an error number shall be returned to indicate the error.

The *pthread_mutex_trylock*() function shall return zero if a lock on the mutex object referenced by *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

COPYRIGHT

NAME

write – write to a file descriptor

SYNOPSIS

#include <unistd.h>

ssize_t write(int *fd*, **const void** **buf*, **size_t** *count*);

DESCRIPTION

write() writes up to *count* bytes from the buffer pointed *buf* to the file referred to by the file descriptor *fd*.

The number of bytes written may be less than *count* if, for example, there is insufficient space on the underlying physical medium, or the **RLIMIT_FSIZE** resource limit is encountered (see **setrlimit**(2)), or the call was interrupted by a signal handler after having written less than *count* bytes.

For a seekable file (i.e., one to which **lseek**(2) may be applied, for example, a regular file) writing takes place at the current file offset, and the file offset is incremented by the number of bytes actually written. If the file was **open**(2)ed with **O_APPEND**, the file offset is first set to the end of the file before writing. The adjustment of the file offset and the write operation are performed as an atomic step.

RETURN VALUE

On success, the number of bytes written is returned (zero indicates nothing was written). On error, –1 is returned, and *errno* is set appropriately.

If *count* is zero and *fd* refers to a regular file, then **write**() may return a failure status if one of the errors below is detected. If no errors are detected, 0 will be returned without causing any other effect. If *count* is zero and *fd* refers to a file other than a regular file, the results are not specified.

ERRORS

EAGAIN
The file descriptor *fd* refers to a file other than a socket and has been marked nonblocking (**O_NONBLOCK**), and the write would block.

EBADF
fd is not a valid file descriptor or is not open for writing.

EFAULT
buf is outside your accessible address space.

EINTR
The call was interrupted by a signal before any data was written; see **signal**(7).

EINVAL
fd is attached to an object which is unsuitable for writing; or the file was opened with the **O_DIRECT** flag, and either the address specified in *buf*, the value specified in *count*, or the current file offset is not suitably aligned.

EIO A low-level I/O error occurred while modifying the inode.

ENOSPC
The device containing the file referred to by *fd* has no room for the data.

NOTES

If a **write**() is interrupted by a signal handler before any bytes are written, then the call fails with the error **EINTR**; if it is interrupted after at least one byte has been written, the call succeeds, and returns the number of bytes written.