Karlsruher Institut für Technologie

Betriebssysteme
(Operating Systems)

Prof. Dr. Frank Bellosa
Dr. Lukas Werling
Peter Maucher, M.Sc.
Daniel Habicht, M.Sc.
Yussuf Khalil, M.Sc.

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

Hauptklausur

17.03.2026

- Bitte tragen Sie zuerst auf dem Deckblatt Ihren Nachnamen, Ihren Vornamen und Ihre Matrikelnummer ein. Tragen Sie dann auf allen 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 23 Blättern: Einem Deckblatt, 17 Aufgabenblättern mit insgesamt 3 Aufgaben sowie 5 Seiten Man-Pages.

  The examination consists of 23 pages: One cover sheet, 17 sheets containing 3 assignments, and 5 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 also 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 according to the lecture.

Die folgende Tabelle wird von uns ausgefüllt!

The following table is completed by us!

Aufgabe	1	2	3	Total
Max. Punkte	20	20	20	60
Erreichte Punkte				

Matrikelnummer/*Matriculation number*

Aufgabe 1: Virtualisierung (20 P)

Assignment 1: Virtualization (20 P)

a) Auf Unix-Systemen werden neue Prozesse traditionell mit einer Kombination aus `fork()` und `exec()` gestartet.

On Unix systems, new processes are traditionally started using a combination of `fork()` and `exec()`.

i) Nennen Sie den Mechanismus, der beim Aufruf von `fork()` das Kopieren aller Speicherseiten erspart. **0.5 P**

Name the mechanism that avoids copying all memory pages when calling `fork()`.

ii) Durch die Benutzung von `fork()` in einem Prozess mit hoher Speichernutzung entstehen Performanceprobleme, da alle Seitentabellen kopiert und angepasst werden müssen. Berechnen Sie für einen Prozess mit einem dicht besetzten, 32 GiB großen Adressraum die Größe der Seitentabellen. Gehen Sie von einer vierstufigen Adressübersetzung mit 4 KiB großen Seiten und 8 Byte pro Eintrag der Seitentabelle aus. Geben Sie Ihre Ergebnisse pro Stufe in KiB bzw. MiB an. **3 P**

Using `fork()` in a process with high memory usage causes performance problems because all page tables must be copied and adjusted. Calculate the size of the page tables for a process with a densely populated 32 GiB address space. Assume a four-level address translation with 4 KiB pages and 8 byte per page table entry. Provide your results per level in KiB or MiB.

Level 1:

Level 2:

Level 3:

Level 4:

iii) Das oben genannte Performanceproblem kann durch einen alternativen Systemaufruf gelöst werden, der direkt ein gegebenes Programm in einem neuen Prozess startet. Welchen Nachteil bietet ein solcher Systemaufruf im Vergleich zu getrenntem `fork()/exec()`? **1 P**

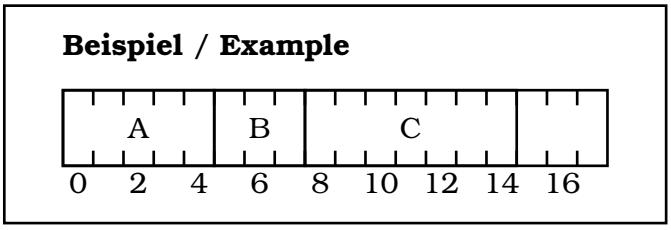
The aforementioned performance issue can be solved by an alternative system call that directly starts a given program in a new process. What disadvantage does such a system call have compared to separate `fork()/exec()`?

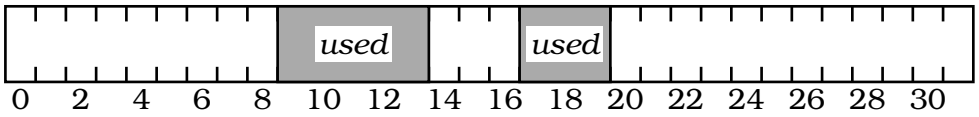
b) **Dynamic Memory Allocation**

i) Fügen Sie die folgenden Speicherallokationen nacheinander entsprechend der „Worst Fit"-Strategie in den gegebenen Speicherbereich ein. Markieren Sie die Speicherbereiche wie im Beispiel mit ihrem jeweiligen Namen. **1.5 P**

Insert the following memory allocations one after another according to the "Worst Fit" strategy into the given memory area. Mark the memory areas with their respective names as in the example.

Name	**Size**
A	5
B	3
C	7





ii) Eine weitere Allokation der Größe 7 kann nicht mehr erfüllt werden, obwohl genügend freie Speicherzellen vorhanden wären. Welche Art von Fragmentierung liegt hier vor? **0.5 P**

Another allocation of size 7 can no longer be satisfied, even though enough free memory cells are available. Which type of fragmentation is present here?

_____ Fragmentierung / *Fragmentation*

iii) Vervollständigen Sie den untenstehenden Code, sodass die Funktion `find_worst_fit()` einen passenden freien Speicherplatz nach der „Worst Fit"-Strategie findet. **3.5 P**

- Die Funktion `bitmap_get()` gibt zurück, ob das `idx`-te Bit der Bitmap gesetzt ist. Gehen Sie davon aus, dass `bitmap` ein ausreichend großes Array ist.
- Die Schleife in `find_worst_fit()` bestimmt bereits die Größe der zusammenhängenden freien Speicherbereiche. Bestimmen Sie auf dieser Basis den passenden Speicherbereich gemäß „Worst Fit".

- `find_worst_fit()` soll den Startindex des gefundenen Speicherbereichs zurückgeben, oder −1 falls kein passender Speicherbereich für die angeforderte Größe `req_size` vorhanden ist.

Complete the code below so that the function `find_worst_fit()` finds a suitable free memory block using the "Worst Fit" strategy.

- *The function `bitmap_get()` returns whether the `idx`-th bit of the bitmap is set. Assume that `bitmap` is a sufficiently large array.*
- *The loop in `find_worst_fit()` already determines the size of contiguous free memory areas. Based on this, determine the suitable memory block according to "Worst Fit".*
- *`find_worst_fit()` should return the start index of the found memory block, or −1 if no suitable memory block for the requested size `req_size` exists.*

```
bool bitmap_get(uint8_t *bitmap, int idx) {

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

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

--------------------------------------------------------------------------------
}
int find_worst_fit(uint8_t *free_map, int len, int req_size) {
  // Declare local variables here.

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

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

--------------------------------------------------------------------------------
  // Loop over individual bits.
  int i = 0;
  while (i < len) {
    // Find length of consecutive set bits (free memory).
    int start = i;
    while (i < len && bitmap_get(free_map, i++)) {}
    int size = i - start - 1;
    // Evaluate worst fit criteria.

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

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

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

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

--------------------------------------------------------------------------------
  }
  // Return location of free memory block, or -1 if none are large enough.

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

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

--------------------------------------------------------------------------------
```

```
}
```

c) Der Adressraum der meisten Prozesse folgt einem typischen Layout mit verschiedenen Segmenten.

The address space of most processes follows a typical layout with various segments.

i) Füllen Sie die leeren Felder der folgenden Tabelle aus. **2 P**

Fill in the empty fields of the following table.

Segment	**Beschreibung / Description**
OS	Adressen, wo der Kernel gemappt ist *Addresses where the kernel is mapped*
	Lokale Variablen, Parameter, Rücksprungadressen *Local variables, parameters, return addresses*
	Dynamisch allozierte Daten (`malloc()`) *Dynamically allocated data (`malloc()`)*
BSS	
Data	
Text	Programm- und Maschinencode *Program, machine code*

ii) In welche der Segmente zeigen die folgenden Zeiger typischerweise? Gehen Sie von der cdecl-ABI aus. **1.5 P**

Into which of the segments do the following pointers typically point? Assume the cdecl ABI.

Zeiger / Pointer	**Segment**
Instruction Pointer `eip`	
Frame Pointer `ebp`	
Program Break Pointer (BRK)	

d) Welchen Vorteil bietet *lazy binding* beim Laden dynamischer Bibliotheken? Wie wird es umgesetzt? **1.5 P**

What advantage does lazy binding *offer when loading dynamic libraries? How is it implemented?*

Advantage: _____

_____

Implementation:

e) Führen Sie die folgenden Scheduling-Algorithmen aus und berechnen Sie die durchschnittliche Turnaround-Time des Systems. Zu jedem Zeitschritt kann dabei eine Scheduling-Entscheidung getroffen werden. Neue Prozesse werden vorne in Warteschlangen eingefügt. Sie haben die folgenden Prozesse gegeben: **5 P**

Execute the following scheduling algorithms and calculate the average turnaround time of the system. At each time step a new scheduling decision can be made. New processes are added at the front of waiting queues. The following processes are given:

ID	Arrival Time	Runtime
P1	$t = 0.5$	3
P2	$t = 1.5$	1
P3	$t = 3.5$	4
P4	$t = 5.5$	2

$t =$	1	2	3	4	5	6	7	8	9	10
PSJF										
RR										

Average Turnaround Times

PSJF:

RR:

Platz für Skizzen / *Sketch space*

1	2	3	4	5	6	7	8	9	10

1	2	3	4	5	6	7	8	9	10

Total: 20 P

Aufgabe 2: Nebenläufigkeit (20 P)

Assignment 2: Concurrency (20 P)

a) Erklären Sie den Unterschied zwischen Nebenläufigkeit und Parallelität. Nennen Sie je eine Situation, in der Nebenläufigkeit bzw. Parallelität die Performanz des Systems erhöht. **2 P**

Explain the difference between concurrency and parallelism. Name one situation each where concurrency or parallelism increases system performance.

Unterschied / *Difference:*

Situation für Nebenläufigkeit / *Situation for concurrency:*

Situation für Parallelität / *Situation for parallelism:*

b) Nennen Sie zwei Gründe, warum parallele Programme nicht von *Sequential Consistency* ausgehen können. **1 P**

Name two reasons why parallel programs cannot assume Sequential Consistency*.*

c) Erklären Sie den Begriff *Mailbox* im Kontext von Message Passing. **1 P**

Explain the term mailbox *in the context of message passing.*

d) Im Folgenden implementieren Sie einen Server, der über eine Pipe Anfragen zum Starten von Prozessen entgegennimmt.
- Gehen Sie davon aus, dass alle benötigten Header bereits inkludiert sind.
- Sie müssen keine Fehlerbehandlung implementieren.

In the following, you will implement a server that receives requests to start processes via a pipe.
- *Assume that all required headers are already included.*
- *You do not need to implement error handling.*

i) Implementieren Sie die Funktion `init_program_starter()`. Sie erstellt eine Pipe und startet mittels `fork()` den Serverprozess. Im Elternprozess soll das Schreibende der Pipe zurückgegeben werden und im Kindprozess die Funktion `handle_requests()` mit dem Leseende der Pipe aufgerufen werden. Achten Sie darauf, die nicht benötigten Enden der Pipe zu schließen. **3 P**

Implement the function `init_program_starter()`. It creates a pipe and starts the server process using `fork()`. In the parent process, the write end of the pipe shall be returned, and in the child process, the function `handle_requests()` shall be called with the read end of the pipe. Make sure to close the unneeded ends of the pipe.

```
void handle_requests(int pipefd);

int init_program_starter() {

-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
  return -1;
}
```

ii) Vervollständigen Sie die Funktion `run_program()`, die eine Anfrage zum Start eines Programms in die Pipe schreibt. Gehen Sie dabei wie folgt vor: **4.5 P**

- Bestimmen Sie zunächst die Gesamtlänge der übergebenen Argumente. Das Array `argv` enthält eine beliebige Anzahl null-terminierter Strings und ist selbst mit `NULL` terminiert (vgl. Beispiel in `main()` auf der nächsten Seite).
- Kopieren Sie dann die Argumente inklusive ihrer terminierenden null-Bytes in den neu allozierten Puffer `buf`.
- Schreiben Sie schließlich den Puffer `buf` in die Pipe.

Complete the function `run_program()`, which writes a request to start a program into the pipe. Proceed as follows:

- *First, determine the total length of the provided arguments. The array `argv` contains an arbitrary number of null-terminated strings and is itself terminated with a `NULL` (see example in `main()` on the following page).*
- *Then, copy the arguments including their terminating null bytes into the newly allocated buffer `buf`.*
- *Finally, write the buffer `buf` into the pipe.*

```
void run_program(int pipefd, char **argv) {
  // Determine combined string length of arguments.
  size_t len = 0;

-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
  // Allocate buffer to write into the pipe.
  char *buf = malloc(len);
  // Copy arguments to buf.

-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
  // Write buf into the pipe.

-------------------------------------------------------------------------
-------------------------------------------------------------------------
  free(buf);
}
```

```c
/* Example usage */
int main() {
  int pipefd = init_program_starter();
  // Writes to the pipe: echo\0Hello World!\0
  run_program(pipefd, (char*[]) { "echo", "Hello World!", NULL });
  return 0;
}
```

iii) Vervollständigen Sie schließlich die Funktion `handle_requests()`, die Anfragen aus der Pipe liest und bearbeitet. **2 P**

- Gehen Sie davon aus, dass jede Anfrage in `buf` passt und mit genau einem Systemaufruf gelesen wird.
- Wenn beim Lesen aus der Pipe EOF auftritt, soll sich der Server mit Status 0 beenden.
- Stellen Sie sicher, dass keine unnötigen Dateideskriptoren an den Kindprozess weitergegeben werden.

Finally, complete the function `handle_requests()`, which reads and processes requests from the pipe.

- *Assume that each request fits into `buf` and is read with exactly one system call.*
- *If EOF occurs when reading from the pipe, the server shall exit with status 0.*
- *Ensure that no unnecessary file descriptors are passed to the child process.*

```c
void handle_requests(int pipefd) {
  char buf[1024];
  ssize_t len;
  while (1) {
    // Read from pipe, set len, exit on EOF.

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

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

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

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

-------------------------------------------------------------------------
    // Split the arguments back into an argv array.
    char **argv = split_args(buf, len);
    if (fork() == 0) {
      // Ensure that no unnecessary file descriptors are passed to the child.

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

-------------------------------------------------------------------------
      execvp(argv[0], argv);
    }
    free(argv);
  }
}
```

e) Ein einfacher Spinlock kann mit einer Test-And-Set-Operation wie folgt implementiert werden.

A simple spinlock can be implemented with a test-and-set operation as follows.

```
void lock(lock_t *lock) {
  while (TestAndSet(&lock->flag, 1) == 1);
}
```

i) Warum kann ein solcher Spinlock in Kombination mit User-Level-Threads (ULT) nicht sinnvoll eingesetzt werden? Erläutern Sie. **1 P**

Why can such a spinlock not be used meaningfully in combination with user-level threads (ULT)? Explain.

ii) Welche der Anforderungen an eine gültige Lösung des Problems kritischer Abschnitte ist für den gegebenen Spinlock auch in Kombination mit Kernel-Level-Threads (KLT) nicht erfüllt? Begründen Sie. **1.5 P**

Which of the requirements for a valid solution to the critical section problem is not fulfilled by the given spinlock even in combination with kernel-level threads (KLT)? Justify.

iii) Implementieren Sie eine Funktion `unlock()`, die zur gegebenen `lock()`-Funktion passt. **1 P**

Implement an `unlock()` function matching the given `lock()` function.

```
void unlock(lock_t *lock) {

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

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

f) Nennen Sie zwei notwendige Bedingungen für das Auftreten einer Verklemmung. Beschreiben Sie für beide Bedingungen, wie sie im Rahmen von *Deadlock Prevention* negiert werden könnten. **3 P**

Name two necessary conditions for the occurrence of a deadlock. For both conditions, describe how they could be negated in the context of deadlock prevention.

Condition 1:

Condition 2:

Total: 20 P

Aufgabe 3: Persistenz (20 P)

Assignment 3: Persistence (20 P)

a) Füllen Sie die Lücken im untenstehenden Text aus. **2.5 P**

Fill in the blanks in the text below.

SSDs kommen im Gegensatz zu Festplatten ohne bewegliche Teile aus und benötigen keine _____-Operationen, die bei Festplatten den Schreib-/Lesekopf neu positionieren. Alle Flash-Seiten der SSD können mit ähnlicher Geschwindigkeit gelesen werden, wodurch die SSD _____ Lesezugriffe deutlich schneller bearbeiten kann als eine Festplatte. Einzelne Seiten können jedoch nicht einfach überschrieben werden. Stattdessen muss ein ganzer _____ zunächst gelöscht werden. Das Betriebssystem kann dazu mit dem _____ -Kommando signalisieren, dass Daten nicht mehr benötigt werden. Die notwendige Indirektion zwischen logischen Adressen und dem Flash-Speicher wird im _____ umgesetzt.

Unlike hard disk drives, SSDs have no moving parts and do not require _____ operations, which reposition the read/write head in HDDs. All flash pages of the SSD can be read at a similar speed, allowing the SSD to handle _____ read accesses significantly faster than a hard drive. However, individual pages cannot simply be overwritten. Instead, an entire _____ must first be erased. The operating system can use the _____ command to signal that data is no longer needed. The necessary indirection between logical addresses and the flash memory is implemented in the _____.

b) Geben Sie je ein Beispiel für einen absoluten und einen relativen Pfad an. Welche zusätzliche Information wird zum Auflösen eines relativen Pfads benötigt? Wo wird diese Information üblicherweise gespeichert? **1.5 P**

Give one example each of an absolute and a relative path. What additional information is needed to resolve a relative path? Where is this information usually stored?

Absolute path: _____

Relative path: _____

Resolve relative path with _____

stored in the _____

c) Eine tar-Datei fasst mehrere Dateien in einem Archiv zusammen. Vor dem Inhalt jeder Datei steht ein 512 Byte großer Header, dessen Felder in der Tabelle unten beschrieben sind.

Eine tar-Datei startet mit dem Header zur ersten Datei. Nach dem Dateiinhalt stehen zwischen 0 und 511 Bytes Padding, sodass der nächste Header wieder auf 512 Byte ausgerichtet starten kann. Am Ende der tar-Datei stehen zwei mit 0-Bytes gefüllte Header.

A tar file combines multiple files into a single archive. Before the content of each file, there is a 512-byte header whose fields are described in the table below.

A tar file starts with the header for the first file. After the file content, there are between 0 and 511 bytes of padding so that the next header can start aligned to 512 bytes again. At the end of the tar file, there are two headers filled with 0 bytes.

tar Header (512 bytes)

Field	**Offset**	**Size**	**Format**
name	0	100	0-terminated string
mode	100	8	octal number (ASCII)
…			
size	124	12	octal number (ASCII)
…			
prefix	345	155	0-terminated string

i) Neben dem Einsatz als Archivdatei werden Daten im tar-Format traditionell direkt auf Magnetband geschrieben. Welches Dateiallokationsverfahren setzt tar um? Warum ist dieses Allokationsverfahren für Magnetbänder am besten geeignet? **1.5 P**

In addition to being used as an archive file, data in tar format is traditionally written directly to magnetic tape. Which file allocation scheme does tar implement? Why is this allocation scheme best suited for magnetic tapes?

File allocation scheme: _____

Advantage for tape: _____

_____

_____

_____

ii) Das Feld *mode* beschreibt die Zugriffsberechtigung auf die Datei als Oktalzahl (z. B. `0640`). Welche drei Subjekte kennt das UNIX-Berechtigungsmodell, das hier zur Anwendung kommt? **1 P**

The mode *field describes the file access permissions as an octal number (e.g., `0640`). Which three subjects does the UNIX permission model, which is applied here, recognize?*

_____

iii) Vervollständigen Sie das untenstehende C-Programm, das eine tar-Datei einliest und die Namen und Größen der darin enthaltenen Dateien ausgibt. Nehmen Sie an, dass benötigte C-Header bereits inkludiert sind, alle Systemaufrufe erfolgreich sind und die tar-Datei wohlgeformt ist. **7.5 P**

Gehen Sie wie folgt vor:

1. Öffnen Sie die als ersten Kommandozeilenparameter übergebene Datei zum Lesen. Falls das Programm mit einer falschen Anzahl an Parametern gestartet wird, soll es sich mit Status 1 beenden.
2. Lesen Sie in der Schleife einen tar-Header ein (ein einzelner `read()`-Aufruf genügt). Falls der gelesene Header komplett mit 0-Bytes gefüllt ist, ist das Ende der Datei erreicht und das Programm soll sich mit Status 0 beenden. Sie können diesen Fall anhand eines leeren `name`-Feldes erkennen.
3. Lesen Sie die im Feld `size` als Oktalzahl (ASCII) enthaltene Dateigröße (in Bytes) aus. Nutzen Sie hierfür die Funktion `strtol()`. Beachten Sie, dass `size` nicht zwingend 0-terminiert ist. Kopieren Sie deshalb dessen Inhalt zunächst in einen passenden 0-terminierten Puffer.
4. Falls das Feld `prefix` nicht leer (d. h. mit 0-Bytes gefüllt) ist, ist der vollständige Dateiname „prefix/name“. Vervollständigen Sie den Code entsprechend.
5. Springen Sie schließlich durch Aufruf von `lseek()` zum Start des nächsten Headers. Beachten Sie, dass die Header immer an Blöcken von 512 Byte ausgerichtet sind und somit ggf. nicht direkt an den Inhalt der vorherigen Datei anschließen.

Hinweis: Nutzen Sie die vordefinierten Variablen, um einzelne Schritte ggf. unabhängig zu lösen.

Complete the C program below that reads a tar file and outputs the names and sizes of the files contained in it. Assume that the required C headers are already included, all system calls are successful, and the tar file is well-formed.

Proceed as follows:

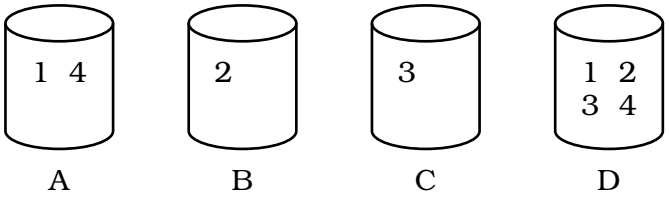
1. *Open the file passed as the first command line parameter for reading. If the program is started with an incorrect number of parameters, it should exit with status 1.*
2. *In the loop, read a tar header (a single call to `read()` is sufficient). If the read header is completely filled with 0 bytes, the end of the file has been reached and the program should exit with status 0. You can recognize this case by an empty `name` field.*
3. *Read the file size (in bytes) contained in the `size` field as an octal number (ASCII). Use the function `strtol()` for this. Note that `size` is not necessarily 0-terminated. Therefore, copy its content into a suitable 0-terminated buffer first.*
4. *If the `prefix` field is not empty (i.e., filled with 0 bytes), the full file name is "prefix/name". Complete the code accordingly.*
5. *Finally, jump to the start of the next header by calling `lseek()`. Note that the headers are always aligned to blocks of 512 bytes and therefore may not directly follow the content of the previous file.*

***Hint**: Use the predefined variables to solve individual steps independently if necessary.*

```
int main(int argc, char **argv) {
  int fd;
  // (1) Open the file given as command-line parameter for reading.

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

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

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

--------------------------------------------------------------------------------
  while (1) {
    char header[512];
    // (2) Read a header and detect the end of the file.

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

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

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

--------------------------------------------------------------------------------
    long size;
    // (3) Copy size field to 0-terminated buffer, parse with strtol().

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

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

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

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

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

--------------------------------------------------------------------------------
    // (4) Fill in the fields. Handle non-empty prefix.

    char *name = ------------------------------------------------------

    char *prefix = ----------------------------------------------------

    if (-------------------------------------------------------------------)
     printf("%s/%s %ld\n", prefix, name, size);
   else
     printf("%s %ld\n", name, size);

   // (5) Seek to the start of the next header.

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

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

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

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

d) Vier sequenzielle Blöcke (1, 2, 3, 4) werden auf ein RAID-System bestehend aus vier Festplatten geschrieben. Für jeden Block wird dabei wie unten dargestellt auf eine Untermenge der Festplatten zugegriffen.

Four sequential blocks (1, 2, 3, 4) are written to a RAID system consisting of four hard drives. For each block, a subset of the drives is accessed as shown below.



i) Um welches RAID-Level handelt es sich? Begründen Sie. **1 P**

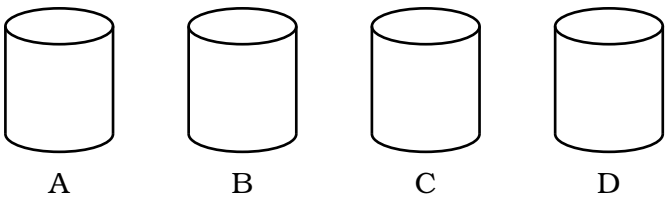
Which RAID level is this? Justify your answer.

ii) Welche der dargestellten Festplattenzugriffe sind lesend, welche sind schreibend? Begründen Sie. Gehen Sie davon aus, dass keine Caches zum Einsatz kommen. **1.5 P**

Which of the depicted drive accesses are reads and which are writes? Justify your answer. Assume no caches are used.

iii) Notieren Sie im Diagramm unten, auf welche Festplatten des RAID-Systems für Leseoperationen auf die vier Blöcke (1, 2, 3, 4) zugegriffen wird. **1 P**

In the diagram below, note which hard drives of the RAID system are accessed for read operations on the four blocks (1, 2, 3, 4).



e) Nehmen Sie an, dass das folgende Shell-Script in einem leeren Verzeichnis ausgeführt wird. Beschreiben Sie die Dateiinhalte nach der Ausführung des Scripts, indem Sie Tabelle 2 ausfüllen. Wenn nicht auf eine Datei zugegriffen werden kann, schreiben sie X in den entsprechenden Tabelleneintrag. Tabelle 1 dient als Beispiel und zeigt die Dateiinhalte, nachdem das Script bis zu Zeile 6 ausgeführt wurde. **2.5 P**

Hinweis: der Befehl `rm` löscht die angegebene Datei ohne dabei Links zu folgen.

Assume that the following shell script is executed in an empty directory. Describe the file contents after running the script by filling out table 2. If a file cannot be accessed, write X into the corresponding table entry. Table 1 serves as an example and shows the file contents after executing the script up to line 6.

Hint*: the `rm` command deletes the given file without following links.*

```
1   # >>: append to file
2   # -n: no newline
3   echo -n A >> a
4   echo -n A >> a
5   echo -n B >> b
6   # ln [TARGET] [LINK_NAME]
7   # -s: symbolic link
8   ln b e
9   ln -s a c
10  ln -s e d
11  echo -n 1 >> c
12  echo -n 2 >> e
13  echo -n 3 >> d
14  rm b
```

a	**b**	**c**	**d**	**e**
AA	B	X	X	X

Table 1: Zustand nach Zeile 5 / *state after line 5*

a	**b**	**c**	**d**	**e**

Table 2: Zustand nach Skript / *state after script*

Total: 20 P

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).

NAME

execv, execvp - execute a file

SYNOPSIS

#include <unistd.h>

int execv(const char ****pathname*, char *const** *argv***[]);**
int execvp(const char ****file*, char *const** *argv***[]);**

DESCRIPTION

The **exec**() family of functions replaces the current process image with a new process image. The functions described in this manual page are layered on top of **execve**(2). (See the manual page for **execve**(2) for further details about the replacement of the current process image.)

The initial argument for these functions is the name of a file that is to be executed.

The functions can be grouped based on the letters following the "exec" prefix.

v - execv(), execvp()

The *char *const argv[]* argument is an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a null pointer.

p - execvp()

This function duplicates the actions of the shell in searching for an executable file if the specified filename does not contain a slash (/) character. The file is sought in the colon-separated list of directory pathnames specified in the **PATH** environment variable.

If the specified filename includes a slash character, then **PATH** is ignored, and the file at the specified pathname is executed.

All other **exec**() functions (which do not include 'p' in the suffix) take as their first argument a (relative or absolute) pathname that identifies the program to be executed.

RETURN VALUE

The **exec**() functions return only if an error has occurred. The return value is -1, and *errno* is set to indicate the error.

NAME

pipe - create pipe

SYNOPSIS

#include <unistd.h>

int pipe(int *pipefd***[2]);**

DESCRIPTION

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array *pipefd* is used to return two file descriptors referring to the ends of the pipe. *pipefd[0]* refers to the read end of the pipe. *pipefd[1]* refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. For further details, see **pipe**(7).

RETURN VALUE

On success, zero is returned. On error, -1 is returned, *errno* is set appropriately, and *pipefd* is left unchanged.

NAME

fork - create a child process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION

fork() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

The child process and the parent process run in separate memory spaces. At the time of **fork**() both memory spaces have the same content. Memory writes, file mappings (**mmap**(2)), and unmappings (**munmap**(2)) performed by one of the processes do not affect the other.

The child process is an exact duplicate of the parent process except for the following points:

* The child has its own unique process ID, and this PID does not match the ID of any existing process group (**setpgid**(2)) or session.
* The child's parent process ID is the same as the parent's process ID.

Note the following further points:

* The child process is created with a single thread--the one that called **fork**(). The entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects; the use of **pthread_atfork**(3) may be helpful for dealing with problems that this can cause.
* The child inherits copies of the parent's set of open file descriptors. Each file descriptor in the child refers to the same open file description (see **open**(2)) as the corresponding file descriptor in the parent. This means that the two file descriptors share open file status flags, file offset, and signal-driven I/O attributes (see the description of **F_SETOWN** and **F_SETSIG** in **fcntl**(2)).

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and *errno* is set appropriately.

NOTES

Under Linux, **fork**() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

NAME

open - open a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
```

DESCRIPTION

The **open**() system call opens the file specified by *pathname*.

The return value of **open**() is a file descriptor, a small, nonnegative integer that is used in subsequent system calls (**read**(2), **write**(2), **lseek**(2), **fcntl**(2), etc.) to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

A call to **open**() creates a new *open file description*, an entry in the system-wide table of open files. The open file description records the file offset and the file status flags (see below). A file descriptor is a reference to an open file description; this reference is unaffected if *pathname* is subsequently removed or modified to refer to a different file.

The argument *flags* must include one of the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

In addition, zero or more file status flags can be bitwise-*or*'d in *flags*. The file status flags can be retrieved and (in some cases) modified; see **fcntl**(2) for details.

The abridged list of file status flags is as follows:

O_APPEND

The file is opened in append mode. Before each **write**(2), the file offset is positioned at the end of the file, as if with **lseek**(2). The modification of the file offset and the write operation are performed as a single atomic step.

O_APPEND may lead to corrupted files on NFS filesystems if more than one process appends data to a file at once. This is because NFS does not support appending to a file, so the client kernel has to simulate it, which can't be done without a race condition.

O_TRUNC

If the file already exists and is a regular file and the access mode allows writing (i.e., is **O_RDWR** or **O_WRONLY**) it will be truncated to length 0. If the file is a FIFO or terminal device file, the **O_TRUNC** flag is ignored. Otherwise, the effect of **O_TRUNC** is unspecified.

RETURN VALUE

open() returns the new file descriptor, or -1 if an error occurred (in which case, *errno* is set appropriately).

NAME

close - close a file descriptor

SYNOPSIS

#include <unistd.h>

int close(int *fd***);**

DESCRIPTION

close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see **fcntl**(2)) held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

If *fd* is the last file descriptor referring to the underlying open file description (see **open**(2)), the resources associated with the open file description are freed; if the file descriptor was the last reference to a file which has been removed using **unlink**(2), the file is deleted.

RETURN VALUE

close() returns zero on success. On error, -1 is returned, and *errno* is set appropriately.

NAME

exit - cause normal process termination

SYNOPSIS

#include <stdlib.h>

void exit(int *status***);**

DESCRIPTION

The **exit**() function causes normal process termination and the value of *status & 0377* is returned to the parent (see **wait**(2)).

All open **stdio**(3) streams are flushed and closed. Files created by **tmpfile**(3) are removed.

The C standard specifies two constants, **EXIT_SUCCESS** and **EXIT_FAILURE**, that may be passed to **exit**() to indicate successful or unsuccessful termination, respectively.

RETURN VALUE

The **exit**() function does not return.

NOTES

The use of **EXIT_SUCCESS** and **EXIT_FAILURE** is slightly more portable (to non-UNIX environments) than the use of 0 and some nonzero value like 1 or -1. In particular, VMS uses a different convention.

After **exit**(), the exit status must be transmitted to the parent process. There are two cases:

- If the parent was waiting on the child, it is notified of the exit status and the child dies immediately.
- Otherwise, the child becomes a "zombie" process: most of the process resources are recycled, but a slot containing minimal information about the child process (termination status, resource usage statistics) is retained in process table. This allows the parent to subsequently use **waitpid**(2) (or similar) to learn the termination status of the child; at that point the zombie process slot is released.

NAME

read - read from a file descriptor

SYNOPSIS

#include <unistd.h>

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

DESCRIPTION

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

On files that support seeking, the read operation commences at the current file offset, and the file offset is incremented by the number of bytes read. If the current file offset is at or past the end of file, no bytes are read, and **read**() returns zero.

If *count* is zero, **read**() *may* detect the errors described below. In the absence of any errors, or if **read**() does not check for errors, a **read**() with a *count* of 0 returns zero and has no other effects.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because **read**() was interrupted by a signal. On error, -1 is returned, and *errno* is set appropriately. In this case, it is left unspecified whether the file position (if any) changes.

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 an error 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.

NAME

lseek -- move the read/write file offset

SYNOPSIS

#include <unistd.h>

off_t lseek(int *fildes*, off_t *offset*, int *whence*);

DESCRIPTION

The *lseek*() function shall set the file offset for the open file description associated with the file descriptor *fildes*, as follows:

* If *whence* is SEEK_SET, the file offset shall be set to *offset* bytes.
* If *whence* is SEEK_CUR, the file offset shall be set to its current location plus *offset*.
* If *whence* is SEEK_END, the file offset shall be set to the size of the file plus *offset*.

The symbolic constants SEEK_SET, SEEK_CUR, and SEEK_END are defined in <*unistd.h*>.

The behavior of *lseek*() on devices which are incapable of seeking is implementation-defined. The value of the file offset associated with such a device is undefined.

The *lseek*() function shall allow the file offset to be set beyond the end of the existing data in the file. If data is later written at this point, subsequent reads of data in the gap shall return bytes with the value 0 until data is actually written into the gap.

The *lseek*() function shall not, by itself, extend the size of a file.

If *fildes* refers to a shared memory object, the result of the *lseek*() function is unspecified.

If *fildes* refers to a typed memory object, the result of the *lseek*() function is unspecified.

RETURN VALUE

Upon successful completion, the resulting offset, as measured in bytes from the beginning of the file, shall be returned. Otherwise, -1 shall be returned, *errno* shall be set to indicate the error, and the file offset shall remain unchanged.

ERRORS

The *lseek*() function shall fail if:

EBADF

The *fildes* argument is not an open file descriptor.

EINVAL

The *whence* argument is not a proper value, or the resulting file offset would be negative for a regular file, block special file, or directory.

EOVERFLOW

The resulting file offset would be a value which cannot be represented correctly in an object of type **off_t**.

ESPIPE

The *fildes* argument is associated with a pipe, FIFO, or socket.

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

stpcpy, strcpy, strncpy - copy a string

SYNOPSIS

#include <string.h>

char *stpcpy(char *dest, const char *src);
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);

DESCRIPTION

The **stpcpy**() and **strcpy**() functions copy the string pointed to by *src*, including the terminating null byte ('\0'), to the buffer pointed to by *dest*. The strings may not overlap, and the destination string *dest* must be large enough to receive the copy. *Beware of buffer overruns!*

The **strncpy**() function is similar, except that at most *n* bytes of *src* are copied. **Warning**: If there is no null byte among the first *n* bytes of *src*, the string placed in *dest* will not be null-terminated.

If the length of *src* is less than *n*, **strncpy**() writes additional null bytes to *dest* to ensure that a total of *n* bytes are written.

RETURN VALUE

stpcpy() returns a pointer to the terminating null byte of the copied string.

The **strcpy**() and **strncpy**() functions return a pointer to the destination string *dest*.

NAME

strcmp, strncmp - compare two strings

SYNOPSIS

#include <string.h>

int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);

DESCRIPTION

The **strcmp**() function compares the two strings *s1* and *s2*. It returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

The **strncmp**() function is similar, except it compares only the first (at most) *n* bytes of *s1* and *s2*.

RETURN VALUE

The **strcmp**() and **strncmp**() functions return an integer less than, equal to, or greater than zero if *s1* (or the first *n* bytes thereof) is found, respectively, to be less than, to match, or be greater than *s2*.

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

strtol, strtoll - convert a string to a long integer

SYNOPSIS

#include <stdlib.h>

long strtol(const char ****nptr*, char **** endptr***, int** *base***);**
long long strtoll(const char ****nptr*, char **** endptr***, int** *base***);**

DESCRIPTION

The **strtol**() function converts the initial part of the string in *nptr* to a long integer value according to the given *base*, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by **isspace**(3)) followed by a single optional '+' or '-' sign. If *base* is zero or 16, the string may then include a "0x" or "0X" prefix, and the number will be read in base 16; otherwise, a zero *base* is taken as 10 (decimal) unless the next character is '0', in which case it is taken as 8 (octal).

The remainder of the string is converted to a *long* value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter 'A' in either uppercase or lowercase represents 10, 'B' represents 11, and so forth, with 'Z' representing 35.)

If *endptr* is not NULL, and the *base* is supported, **strtol**() stores the address of the first invalid character in **endptr*. If there were no digits at all, **strtol**() stores the original value of *nptr* in **endptr* (and returns 0). In particular, if **nptr* is not '\0' but ***endptr* is '\0' on return, the entire string is valid.

The **strtoll**() function works just like the **strtol**() function but returns a *long long* integer value.

RETURN VALUE

The **strtol**() function returns the result of the conversion, unless the value would underflow or overflow. If an underflow occurs, **strtol**() returns **LONG_MIN**. If an overflow occurs, **strtol**() returns **LONG_MAX**. In both cases, *errno* is set to **ERANGE**. Precisely the same holds for **strtoll**() (with **LLONG_MIN** and **LLONG_MAX** instead of **LONG_MIN** and **LONG_MAX**).

ERRORS

This function does not modify *errno* on success.

EINVAL
(not in C99) The given *base* contains an unsupported value.

ERANGE
The resulting value was out of range.

The implementation may also set *errno* to **EINVAL** in case no conversion was performed (no digits seen, and 0 returned).

STANDARDS

C11, POSIX.1-2008.