Karlsruher Institut für Technologie

Betriebssysteme
(Operating Systems)

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

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

Nachklausur
06.09.2024

- 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 27 Blättern: Einem Deckblatt, 19 Aufgabenblättern mit insgesamt 3 Aufgaben sowie 7 Seiten Man-Pages.

  The examination consists of 27 pages: One cover sheet, 19 sheets containing 3 assignments, and 7 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	1	2	3	Total
Max. Punkte	20	20	20	60
Erreichte Punkte				

Aufgabe 1: Virtualisierung

Assignment 1: Virtualization

a) Nennen Sie drei Dinge, die im Sinne der Vorlesung vom Betriebssystem virtualisiert werden, und erklären Sie kurz die Virtualisierung. **3 pt**

Name three things that are virtualized within the meaning of the lecture, and shortly explain the virtualization.

1:

2:

3:

b) Erklären Sie die Rolle des Loaders beim Starten eines Prozesses. **2 pt**

Explain the role of the loader when starting a process.

c) In dieser Aufgabe analysieren wir einen Scheduling-Algorithmus. Bevor mit der Analyse begonnen wird, implementieren Sie zwei Hilfsfunktion für die Verwaltung einer einfach verketteten Liste, welche dann im Algorithmus verwendet werden.

In dieser Aufgabe dürfen Sie davon ausgehen, dass alle Bibliotheksfunktionen erfolgreich ausgeführt werden, dass die Datenstruktur in einem korrekten Zustand war und dass alle Header, die Sie benötigen könnten, bereits inkludiert sind. Beachten Sie, dass alle Pointer korrekt gesetzt werden, sowie dass alle nicht mehr benötigten Ressourcen am Ende wieder freigegeben werden.

In this exercise, we analyse a scheduling algorithm. Before you start analysing, you are to implement two helper functions for the management of a singly linked list, which are used in the algorithm.

You can assume that all library functions execute successfully, that the data structure was in a correct state, and that all headers you might need are already included. Take care that all pointers are set correctly, and that all resources are cleaned up at the end.

Zunächst soll die Funktion `enqueue_back` implementiert werden, die ein `int` am Ende einer `singly_linked_list` einfügt. Beachten Sie, dass alle Pointer korrekt gesetzt werden, sodass das Element sowohl über die Liste als auch über den `end`-Pointer erreicht werden kann. **2 pt**

Eine leere Liste erkennen Sie daran, dass sowohl `start`- als auch `end`-Pointer auf `NULL` gesetzt sind.

First, implement the function `enqueue_back`, which inserts an `int` at the end of a `singly_linked_list`. Take care that all pointers are set correctly, so that the element can both be accessed via the list as well as the `end` pointer.

You can identify an empty list by it having both `start` and `end` pointer set to `NULL`.

```
#include <stdlib.h>
#include <stdbool.h>

struct sll_entry {
    // NULL if no more entries
    struct sll_entry *next;
    int payload;
};
struct singly_linked_list {
    struct sll_entry *start;
    struct sll_entry *end;
};

typedef struct sll_entry sll_entry;
typedef struct singly_linked_list singly_linked_list;

void enqueue_back(singly_linked_list *list, int entry) {

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

}
.....
```

Als weitere Hilfsfunktion soll `dequeue_front` implementiert werden, die das erste `int`-Objekt am Anfang der `singly_linked_list` wieder entfernt. Beachten Sie die Hinweise aus der letzten Teilaufgabe sowie der Aufgabenbeschreibung. **2 pt**

Additionally, implement the helper function `dequeue_front`, which removes the first `int` object at the beginning of a `singly_linked_list`. Take note of the hints in the previous subquestion and the introductory text.

```
int dequeue_front(singly_linked_list *list) {

}
```

Betrachten Sie den folgenden Scheduling-Algorithmus. Geben Sie die ersten zehn Scheduling-Entscheidungen für die unten vorgegebene Initialisierung an. Nehmen Sie an, dass anfangs ein Prozess mit ID `00` läuft. **1.5 pt**

Consider the following scheduling algorithm. State the first ten scheduling decisions for the initialization given below. Assume that a process with id `00` is initially running.

```
#define NUM_ENTRIES 3
static int index = 0;
static singly_linked_list scheduler_lists[NUM_ENTRIES] = {0};
int schedule_next_task(int last) {
    enqueue_back(scheduler_lists + index, last);
    index += 1;
    for (int i = 0; i < NUM_ENTRIES; ++i) {
        if (scheduler_lists[(index + i) % NUM_ENTRIES].start != 0) {
            index = (index + i) % NUM_ENTRIES;
            return dequeue_front(scheduler_lists + index);
        }
    }
    return 0;
}
```

Initialisierung / *Initialization*:

```
enqueue_back(scheduler_lists + 0, 01);
enqueue_back(scheduler_lists + 0, 02);
enqueue_back(scheduler_lists + 0, 03);
enqueue_back(scheduler_lists + 0, 04);
enqueue_back(scheduler_lists + 1, 10);
enqueue_back(scheduler_lists + 1, 11);
enqueue_back(scheduler_lists + 1, 12);
enqueue_back(scheduler_lists + 2, 20);
enqueue_back(scheduler_lists + 2, 21);
```

Scheduling Cycle	0	1	2	3	4	5	6	7	8	9	10
Process ID	00										

d) Beschreiben Sie das M-to-N-Modell bei Threads, und geben Sie einen Nachteil gegenüber Kernel-Level-Threads an. **1.5 pt**

Describe the M-to-N model for threads, and name one disadvantage as compared to kernel level threads.

e) Geben Sie außer Swapping zwei Fälle an, in denen Page Faults auftreten. **1 pt**

Except swapping, name two cases in which page faults occur.

Beschreiben Sie den Ablauf im Betriebsystem, welcher bei einem Speicherzugriff auf eine ausgelagerte Seite (*swapping*) durchlaufen wird. Nehmen Sie hierfür zunächst an, dass genügend freier physischer Speicher vorhanden ist. Schreiben Sie entweder einen Text oder zeichnen eine Grafik auf einer der Rückseiten der Klausur. Markieren Sie klar, wo und was Ihre Antwort ist! **2 pt**

Describe the procedure in the operating system for a memory access to a swapped out page, For now, assume that enough physical memory is available. Either write a text or draw a figure on one of the backsides of this exam. Clearly mark your answer and where you put it!

Welche zusätzlichen Schritte muss das Betriebssystem durchführen, wenn kein nicht genug physischer Speicher zur Verfügung steht? Beschreiben Sie oder erweitern Sie ihre Grafik aus der vorherigen Teilaufgabe. Markieren Sie in diesem Fall ihre Erweiterungen der Grafik klar als solche. **2 pt**

Which additional steps does the operating system need to execute if not enough physical memory is available? Describe or extend your figure from the previous exercise. In this case clearly mark your extensions in the figure.

f) Erklären Sie den Unterschied zwischen synchroner und asynchroner IO, und beschreiben Sie, wie man das eine durch das jeweils andere abbilden kann. **3 pt**

Describe the difference between synchronous and asynchronous IO, and describe how you can emulate one with the respective other.

Total: 20.0pt

Aufgabe 2: Nebenläufigkeit

Assignment 2: Concurrency

a) Erläutern Sie die zwei Phasen einer *Two-Phase Lock* (`futex`). **2 pt**

Explain the two phases of a Two-Phase Lock *(`futex`).*

Phase 1:

Phase 2:

b) Zeichen Sie einen *Resource Allocation Graph*, der ein Deadlock darstellt. **2 pt**

Draw a Resource Allocation Graph *that represents a deadlock.*

c) Nennen Sie die vier notwendigen Bedingungen für Deadlocks. **2 pt**

Name the four necessary conditions for deadlocks.

```
1  #include <sched.h>
2
3  struct account {
4      int id;
5      int balance;
6  };
7
8  void update_balance(struct account *acc, int amount) {
9      acc->balance += amount;
10 }
11
12 void do_transaction(struct account *from, struct account *to, int amount) {
13     update_balance(from, -amount);
14     update_balance(to, amount);
15     sched_yield();
16 }
```

d) Betrachten Sie das obige Programm. Nehmen Sie an, dass für jede Transaktion jeweils ein neuer Thread gestartet und in diesem `do_transaction()` ausgeführt wird. Das Programm funktioniert auf einem Single-Core-Prozessor einwandfrei. Es wird neue Hardware gekauft und das System hat nun einen Multi-Core-Prozessor, das Betriebssystem allerdings bleibt das gleiche. Plötzlich funktioniert das Programm nicht mehr wie erwartet. Was könnte der Grund dafür sein? Was sagt das über das verwendete Betriebssystem aus? Nennen Sie außerdem die konkrete Zeilennummer, in der das Problem auftritt. **3 pt**

Consider the program above. Assume that for each transaction a new thread is started and `do_transaction()` is executed in this thread. The program is working fine on a single-core processor. New hardware is bought and now the system has a multi-core processor, but the operating system remains the same. Suddenly, the program is not working as expected anymore. What could be the reason for this? What does this say about the operating system in use? Also, name the specific line number where the problem occurs.

e) Modifizieren Sie den Code so, dass er auch auf einem Multi-Core-Prozessor korrekt funktioniert. Falls Sie dafür eine oder mehrere Zeile(n) zwingend entfernen müssen, streichen Sie diese durch. Wenn nötig, fügen Sie Code zur Initialiserung in `initialize_account()` hinzu. **3 pt**

Modify the code so that it works correctly on a multi-core processor. If you have to remove one or more line(s) for this, strike them through. If necessary, add initialization code in `initialize_account()`.

```
#include <sched.h>
.....
.....
struct account {
    int id;
    int balance;
.....
.....
};

void update_balance(struct account *acc, int amount) {
.....
.....
    acc->balance += amount;
.....
.....
}

void do_transaction(struct account *from, struct account *to, int amount) {
.....
.....
    update_balance(from, -amount);
.....
.....
    update_balance(to, amount);
.....
.....
    sched_yield();
.....
.....
}

void initialize_account(struct account *acc) {
.....
.....
.....
}
```

f) In einem weiteren Hardwareupgrade erhält das System einen Prozessor, der *transaktionalen Speicher* unterstützt. Hierbei können innerhalb einer Transaktion mehrere Speicherzugriffe atomar ausgeführt werden. Erst beim *Commit* werden die innerhalb der Transaktion modifizierten Speicherbereiche aktualisiert. Sollte ein anderer Prozessorkern die Speicherbereiche gleichzeitig modifizieren, so schlägt der *Commit* fehl und die Transaktion wird abgebrochen und muss wiederholt werden. **2 pt**

Schreiben Sie `do_transaction()` so um, dass es transaktionalen Speicher verwendet unter Zuhilfenahme der bereitgestellten Funktionen. Nutzen Sie nur so viele Transaktionen wie zwingend nötig.

In another hardware upgrade, the system receives a processor that supports transactional memory*. With this, multiple memory accesses can be executed atomically within a transaction. Only when* committing*, the memory areas modified within the transaction are updated. If another processor core modifies the memory areas at the same time, the* commit *fails and the transaction is aborted and must be repeated.*

Rewrite `do_transaction()` to use transactional memory using the provided functions. Use only as many transactions as absolutely necessary.

```
// Starts a new transaction
// Returns 1 when successful, 0 when a transaction is already running
int begin();
// Commits a running transaction
// Returns 1 when the transaction was successful, 0 otherwise
int commit();

void update_balance(struct account *acc, int amount) {
    acc->balance += amount;
}

void do_transaction(struct account *from, struct account *to, int amount) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
```

Jemand in Ihrem Team schlägt ein alternatives Programmdesign vor, wo nicht ein Thread pro Transaktion gestartet wird, sondern für jedes Konto (`struct account`) wird zum Programmstart jeweils ein einzelner eigener Thread initialisiert. Die Threads arbeiten jeweils die anstehenden Kontostandsaktualisierungen für ihr jeweiliges Konto ab. Die entsprechenden Aktualisierungsaufgaben werden dazu in einem Puffer abgelegt, der eine fixe Anzahl an Einträgen aufnehmen kann. Eine beliebige Zahl anderer Threads fügt Aufgaben in diesen Puffer ein.

Someone in your team suggests an alternative program design where not one thread is started per transaction, but at program start, a single thread is initialized for each account (`struct account`). The threads each process the pending account balance updates for their respective account. The corresponding update tasks are stored in a buffer that can hold a fixed number of entries. Any number of other threads add tasks to this buffer.

g) Bei Verwendung des Puffers ergibt sich ein klassisches Synchronisationsproblem. Benennen Sie dieses. **1 pt**

When using the buffer, a classic synchronization problem arises. Name it.

h) Diskutieren Sie die Eignung der folgenden drei Synchronisationsprimitive für die Lösung des Problems: *Mutex*, *Condition Variable*, *Semaphore*. Legen Sie dar, welches am besten geeignet ist und warum, und erläutern Sie jeweils einen Nachteil der anderen beiden. **3 pt**

Discuss the suitability of the following three synchronization primitives for solving the problem: mutex, condition variable, semaphore. *Explain which one is best suited and why, and explain the disadvantages or problems for each of the other two.*

i) Benennen und erklären Sie ein weiteres klassisches Synchronisationsproblem. **2 pt**

Name and explain another classic synchronization problem.

Total: 20.0pt

Aufgabe 3: Persistenz

Assignment 3: Persistence

In dieser Aufgabe implementieren Sie eine vereinfachte Variante vom Kommandozeilenwerkzeug `du`, um für ein gegebenes Verzeichnis den auf dem Hintergrundspeicher allozierten Platz für alle enthaltenen Dateien rekursiv zu ermitteln. Für die Teilaufgaben a) bis d) sollen Sie annehmen, dass für jeden vorkommenden *inode* exakt ein Verzeichniseintrag existiert und dass alle Bibliotheksaufrufe bzw. Systemaufrufe erfolgreich sind, sofern keine ungültigen Argumente verwendet werden. **Es muss keine Fehlerbehandlung implementiert werden**. Weiter können Sie annehmen, dass alle notwendige Header bereits inkludiert sind.

In this exercise, you implement a simple version of the command line utility `du` for calculating the allocated disk space of all files in a given directory recursively. For exercise a) to d), you should assume that for each inode there is exactly one directory entry, and that all library calls and system calls complete successfully if you do not use invalid arguments. ***You do not have to implement any error handling****. Further, you may assume that all necessary header files are already included.*

a) Implementieren Sie die Funktion `get_alloced_size()`, welche die Menge an allozierten Platz in Bytes auf dem Hintergrundspeicher für die im Dateideskriptor `fd` übergebene Datei ermittelt. Der Dateideskriptor darf nicht in `get_alloced_size()` geschlossen werden. Sie dürfen annehmen, dass `fd` ein gültiger Dateideskriptor auf eine reguläre Datei ist. **2 pt**

Hinweis: der allozierte Platz entspricht nicht zwangsläufig der Dateigröße.

Implement the function `get_alloced_size()` that returns the allocated disk space in bytes for the file passed in the file descriptor `fd`. You must not close the file descriptor in `get_alloced_size()`. You may assume that `fd` is a valid file descriptor on a regular file.

Hint: the allocated space does not necessarily equal the file size.

```
size_t get_alloced_size(int fd) {
.....

.....

.....

.....

.....

.....

.....
}
```

b) Weshalb könnte die Dateigröße von der Menge an alloziertem Platz auf dem Hintergrundspeicher abweichen? Nennen Sie einen möglichen Grund. **1 pt**

Why could the file size differ from the allocated disk space? Give one possible reason.

c) Implementieren Sie die Funktion `is_special_name()`, welche ermittelt, ob es sich beim Dateinamen `name` um einen der beiden Verzeichniseinträge `"."` oder `".."` handelt. Für die beiden Namen `"."` und `".."` soll 1 zurückgegeben werden, ansonsten 0. `name` ist ein nullterminierter String. **1 pt**

Implement the `is_special_name()` function that determines if the file name `name` matches on of the directory entries `"."` or `".."`. For the names `"."` and `".."`, return 1, otherwise 0. `name` is a null-terminated string.

```
int is_special_name(const char *name) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
```

d) Vervollständigen Sie die Funktion `walk_dir()`, welche die Summe des allozierten Platzes auf dem Hintergrundspeicher über alle Dateien im von `dir_fd` beschriebenen Verzeichnis rekursiv ermittelt. Geben Sie den ermittelten Platzverbrauch in Bytes zurück. Sie dürfen annehmen, dass `dir_fd` ein gültiger Dateideskriptor auf ein Verzeichnis ist. **4.5 pt**

- Verwenden Sie für die Ermittlung des Platzverbrauchs einer Datei die in a) implementierte `get_alloced_size()`.
- Überspringen Sie besondere Verzeichnisse (`"."` und `".."`). Verwenden Sie hierfür `is_special_name()` aus Teilaufgabe b).
- Geben Sie alle in dieser Funktion allozierten Ressourcen (z.B. Dateideskriptoren) vor dem Verlassen der Funktion frei. `dir_fd` muss vom Aufrufer freigegeben werden.

Hinweise:

- Sie können `openat()` verwenden, um einen von `dir_fd` ausgehenden relativen Pfad zu öffnen.
- `closedir()` gibt den zugrundeliegenden Dateideskriptor frei.
- Das Öffnen eines Verzeichnisses für schreibenden Zugriff ist verboten.

Complete the function `walk_dir()` that calculates the sum of allocated disk space over all files contained in the directory described by `dir_fd` recursively. Return the calculated disk usage in bytes. You may assume that `dir_fd` is a valid file descriptor on a directory.

- *For calculating the disk usage of a file, use `get_alloced_size()` implemented in exercise a).*
- *Skip special directories (`"."` and `".."`) using `is_special_name()` implemented in exercise b).*
- *Free all resources (e.g., file descriptors) allocated in this function before returning. `dir_fd` must be freed by the caller.*

Hints:

- *You can use `openat()` for opening a path relative to `dir_fd`.*
- *`closedir()` closes the underlying file descriptor.*
- *Opening directories for writing is forbidden.*

```
size_t get_alloced_size(int fd);
int is_special_name(const char *name);

size_t walk_dir(int dir_fd) {
    size_t size = 0;
    DIR *dirp = fdopendir(dup(dir_fd)); // open dir stream
.....

.....

.....
    while (                                                    ) {
.....

.....

.....

.....
        switch (                                               ) {
.....
        case DT_DIR:
.....

.....

.....

.....

.....

.....

.....

.....
            break;
.....
        case DT_REG:
.....

.....

.....

.....

.....
            break;
.....
        default: break; // ignore other file types
.....
        }
.....

.....

.....
    }

    closedir(dirp);
    return size;
}
```

e) Implementieren Sie die Funktion `calc_disk_usage()`, welche die Anzahl an allozierten Bytes mittels der in c) implementierten `walk_dir()` für das durch `path` beschriebene Verzeichnis ermittelt. Sie dürfen annehmen, dass `path` ein gültiges Verzeichnis beschreibt. Geben Sie alle in dieser Funktion allozierten Ressourcen vor dem Verlassen frei. **2 pt**

Implement the function `calc_disk_usage()` that calculates the number of allocated bytes for the directory described by `path` using `walk_dir()` from exercise c). You may assume that `path` describes a valid directory. Free all resources allocated in this function before returning.

```
size_t get_alloced_size(int fd);
int is_special_name(const char *name);
size_t walk_dir(int dir_fd);

size_t calc_disk_usage(const char *path) {
.....

.....

.....

.....

.....

.....

.....
}
```

f) Beschreiben Sie den Unterschied zwischen symbolischen Links und *hard links*. Geben Sie zudem an, auf welcher Ebene des Betriebssystems symbolische Links aufgelöst werden. **1.5 pt**

Describe the difference between symbolic links and hard links. Further, name the layer of the operating system that resolves symbolic links.

Symbolic link resolved by:

g) Nehmen Sie an, dass die Menge an *hard links* pro *inode* nicht auf eins beschränkt ist und dass ausschließlich reguläre Dateien und Verzeichnisse existieren. Ist es möglich, dass `calc_disk_usage()` bzw. `walk_dir()` nicht den korrekten Platzverbrauch berechnen? Begründen Sie Ihre Antwort. **1 pt**

Assume that the number of hard links per inode is not limited to one and that only regular files and directories exist. Is it possible that `calc_disk_usage()` *or* `walk_dir()` *do not calculate the correct disk usage? Justify your answer.*

h) Nun sollen auch symbolische Links bei der Ermittlung des verbrauchten Hintergrundspeicherplatzes unterstützt werden. Wieso reicht es für eine korrekte Implementation nicht aus, den symbolischen Link aufzulösen und wie eine reguläre Datei oder ein Verzeichnis zu behandeln? Beschreiben Sie ein mögliches Problem. **1 pt**

Now, we want to also support symbolic links when calculating the disk usage. Why does it not suffice for a correct implementation to simply resolve a symbolic link and handle it like a regular file or a directory? Describe one possible problem.

i) Nennen Sie den Zweck, welchen das *Flash Translation Layer (FTL)* einer SSD erfüllt. **0.5 pt**

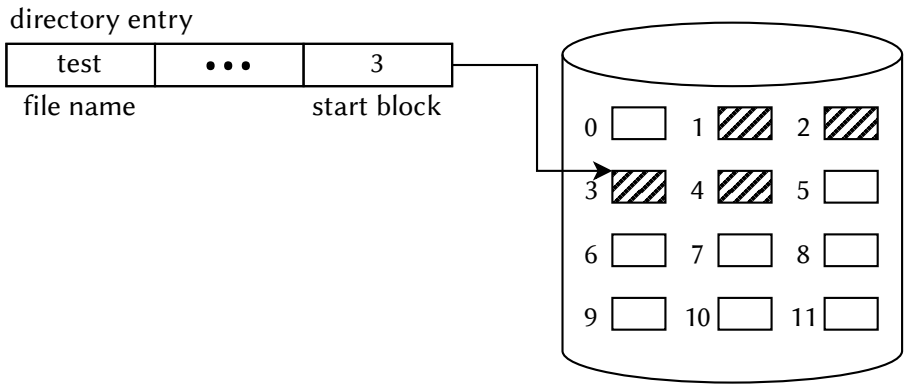
Give the purpose that the flash translation layer (FTL) of a SSD fulfills.

j) Beschreiben Sie das Problem, welches beim Überschreiben von Daten auf einer HDD mit *Shingled Magnetic Recording (SMR)* auftreten kann. **0.5 pt**

Describe the problem that may occur when overwriting data on an HDD with shingled magnetic recording (SMR).

k) Geben Sie an, welche Dateiallokationsstrategie in der folgenden Abbildung dargestellt wird. Geben Sie zudem an, wie viele Zugriffe auf den Hintergrundspeicher notwendig sind, um den vierten Block der Datei "test" (Startblock 3) zu lesen. Nehmen Sie hierfür an, dass durch Caching im Hauptspeicher keine weiteren Zugriffe für die Dateitabelle (*file table*) anfallen. **1 pt**

Give the name of the file allocation strategy depicted in the following figure. Further, give the number of disk accesses that are required for reading the fourth block of the file "test" (start block 3). For this, you shall assume that all accesses on the file table do not require additional disk accesses due to caching in system memory.



file table

block	next
0	⊥
1	2
2	4
3	1
4	⊥
5	⊥
6	⊥
7	⊥
8	⊥
9	⊥
10	⊥
11	⊥

File Allocation Strategy:

Number of Disk Accesses:

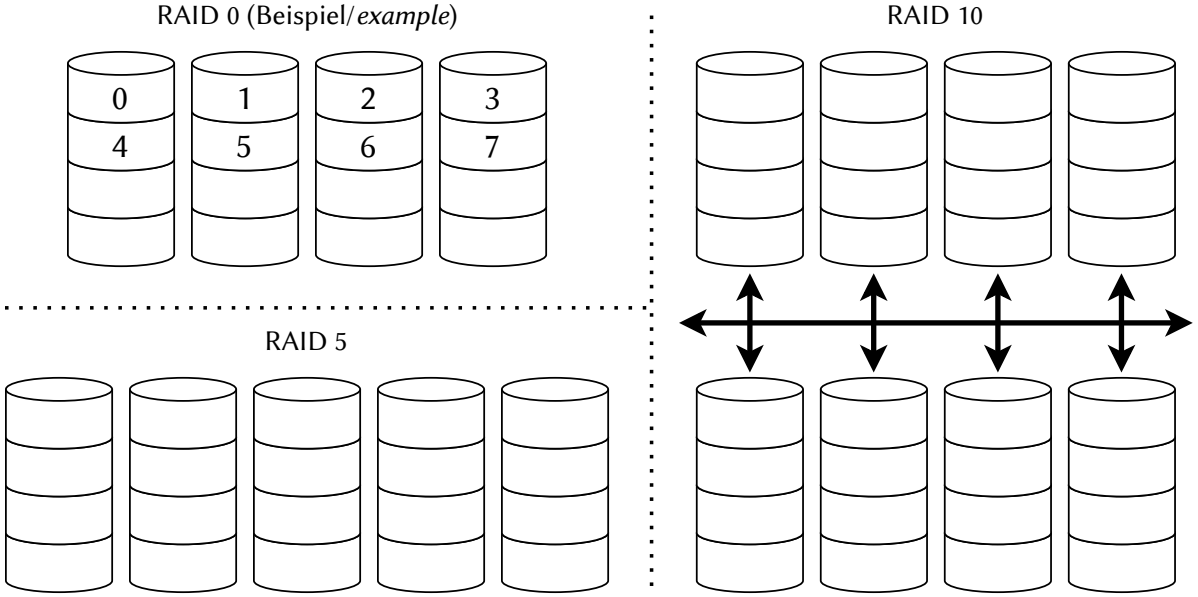
Geben Sie einen Vor- und einen Nachteil der oben gezeigten Dateiallokationsstrategie an. **1 pt**

Give one advantage and one disadvantage of the file allocation strategy shown above.

(+): **(-):**

l) In der folgenden Abbildung finden Sie die Verteilung von den ersten acht Datenblöcken in einem RAID 0-Verbund über vier Hintergrundspeicher. Vervollständigen Sie die Verteilung für die ersten acht Datenblöcke für RAID 5 und RAID 10. Sollten Paritätsblöcke in einem der RAID-Level verwendet werden, sollen die Paritätsblöcke in dieser Aufgabe über vier Datenblöcke gebildet werden. Ein Paritätsblock über die Blöcke n bis m soll als P(n - m) angegeben werden. Gehen Sie beim Verteilen der Blöcke erst von links nach rechts, dann von oben nach unten vor. RAID 0 dient hierfür als Beispiel. **2 pt**

The following figure shows the distribution of the first eight data blocks in a RAID 0 array with four disks. Complete the distribution of the first eight data blocks for RAID 5 and RAID 10. If a RAID level requires parity blocks, assume that parity blocks are formed over four data blocks. Use P(n - m) as notation for a parity block over the blocks n to m. When distributing the blocks, first go from left to right, then from top to bottom. RAID 0 serves as example for this.



Erklären Sie in wenigen Worten, in welchen RAID-Konfigurationen ein RAID 10- einem RAID 01-Verbund vorzuziehen ist und warum das der Fall ist. **1 pt**

Explain in few words in which RAID configuration a RAID 10 setup should be preferred over a RAID 01 setup and why this is the case.

Total: 20.0pt

NAME

open, openat − open and possibly create a file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <fcntl.h>

int open(const char ****pathname*, int** *flags***, ...**
/* **mode_t** *mode* */ **);**

int openat(int *dirfd***, const char ****pathname***, int** *flags***, ...**
/* **mode_t** *mode* */ **);**

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 an index to an entry in the process's table of open file descriptors. The file descriptor 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. For further details on open file descriptions, see NOTES.

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 creation flags and file status flags can be bitwise ORed in *flags*. The *file creation flags* are **O_DIRECTORY** and **O_NOFOLLOW**. The *file status flags* are all of the remaining flags listed below. The distinction between these two groups of flags is that the file creation flags affect the semantics of the open operation itself, while the file status flags affect the semantics of subsequent I/O operations. The file status flags can be retrieved and (in some cases) modified; see **fcntl**(2) for details.

The full list of file creation flags and file status flags is as follows:

O_DIRECTORY
If *pathname* is not a directory, cause the open to fail. This flag was added in Linux 2.1.126, to avoid denial-of-service problems if **opendir**(3) is called on a FIFO or tape device.

O_NOFOLLOW
If the trailing component (i.e., basename) of *pathname* is a symbolic link, then the open fails, with the error **ELOOP**. Symbolic links in earlier components of the pathname will still be followed.

O_PATH (since Linux 2.6.39)
Obtain a file descriptor that can be used for two purposes: to indicate a location in the filesystem tree and to perform operations that act purely at the file descriptor level. The file itself is not opened, and other file operations (e.g., **read**(2), **write**(2), **fchmod**(2), **fchown**(2), **fgetxattr**(2), **ioctl**(2), **mmap**(2)) fail with the error **EBADF**.

The following operations *can* be performed on the resulting file descriptor:

- **close**(2).
- **fchdir**(2), if the file descriptor refers to a directory (since Linux 3.5).
- **fstat**(2) (since Linux 3.6).
- Duplicating the file descriptor (**dup**(2), **fcntl**(2) **F_DUPFD**, etc.).
- Passing the file descriptor as the *dirfd* argument of **openat**() and the other "*at()" system calls. This includes **linkat**(2) with **AT_EMPTY_PATH** (or via procfs using **AT_SYMLINK_FOLLOW**) even if the file is not a directory.

When **O_PATH** is specified in *flags*, flag bits other than **O_DIRECTORY** and **O_NOFOLLOW** are ignored.

Opening a file or directory with the **O_PATH** flag requires no permissions on the object itself (but does require execute permission on the directories in the path prefix). Depending on the subsequent operation, a check for suitable file permissions may be performed (e.g., **fchdir**(2) requires execute permission on the directory referred to by its file descriptor argument). By contrast, obtaining a reference to a filesystem object by opening it with the **O_RDONLY** flag requires that the caller have read permission on the object, even when the subsequent operation (e.g., **fchdir**(2), **fstat**(2)) does not require read permission on the object.

openat()
The **openat**() system call operates in exactly the same way as **open**(), except for the differences described here.

The *dirfd* argument is used in conjunction with the *pathname* argument as follows:

- If the pathname given in *pathname* is absolute, then *dirfd* is ignored.
- If the pathname given in *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like **open**()).
- If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **open**() for a relative pathname). In this case, *dirfd* must be a directory that was opened for reading (**O_RDONLY**) or using the **O_PATH** flag.

If the pathname given in *pathname* is relative, and *dirfd* is not a valid file descriptor, an error (**EBADF**) results. (Specifying an invalid file descriptor number in *dirfd* can be used as a means to ensure that *pathname* is absolute.)

RETURN VALUE

On success, **open**() and **openat**() return the new file descriptor (a nonnegative integer). On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EBADF
(**openat**()) *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EISDIR
pathname refers to a directory and the access requested involved writing (that is, **O_WRONLY** or **O_RDWR** is set).

ELOOP
pathname was a symbolic link, and *flags* specified **O_NOFOLLOW** but not **O_PATH**.

ENOTDIR
A component used as a directory in *pathname* is not, in fact, a directory, or **O_DIRECTORY** was specified and *pathname* was not a directory.

ENOTDIR
(**openat**()) *pathname* is a relative pathname and *dirfd* is a file descriptor referring to a file other than a directory.

NOTES

Open file descriptions
The term open file description is the one used by POSIX to refer to the entries in the system-wide table of open files. In other contexts, this object is variously also called an "open file object", a "file handle", an "open file table entry", or—in kernel-developer parlance—a *struct file*.

When a file descriptor is duplicated (using **dup**(2) or similar), the duplicate refers to the same open file description as the original file descriptor, and the two file descriptors consequently share the file offset and file status flags. Such sharing can also occur between processes: a child process created via **fork**(2) inherits duplicates of its parent's file descriptors, and those duplicates refer to the same open file descriptions.

Each **open**() of a file creates a new open file description; thus, there may be multiple open file descriptions corresponding to a file inode.

NAME

opendir, fdopendir − open a directory

closedir − close a directory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

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

DIR *opendir(const char *name);
DIR *fdopendir(int fd);

int closedir(DIR *dirp);
```

DESCRIPTION

The **opendir**() function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

The **fdopendir**() function is like **opendir**(), but returns a directory stream for the directory referred to by the open file descriptor *fd*. After a successful call to **fdopendir**(), *fd* is used internally by the implementation, and should not otherwise be used by the application.

The **closedir**() function closes the directory stream associated with *dirp*. A successful call to **closedir**() also closes the underlying file descriptor associated with *dirp*. The directory stream descriptor *dirp* is not available after this call.

RETURN VALUE

The **opendir**() and **fdopendir**() functions return a pointer to the directory stream. On error, NULL is returned, and *errno* is set to indicate the error.

The **closedir**() function returns 0 on success. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
: *fd* is not a valid file descriptor opened for reading.

ENOENT
: Directory does not exist, or *name* is an empty string.

ENOTDIR
: *name* is not a directory.

EBADF
: Invalid directory stream descriptor *dirp*.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
opendir(), **fdopendir**(), **closedir**()	Thread safety	MT-Safe

NOTES

Filename entries can be read from a directory stream using **readdir**(3).

The **opendir**() function sets the close-on-exec flag for the file descriptor underlying the *DIR **. The **fdopendir**() function leaves the setting of the close-on-exec flag unchanged for the file descriptor, *fd*. POSIX.1-200x leaves it unspecified whether a successful call to **fdopendir**() will set the close-on-exec flag for the file descriptor, *fd*.

SEE ALSO

open(2), **readdir**(3)

NAME

readdir − read a directory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

DESCRIPTION

The **readdir**() function returns a pointer to a *dirent* structure representing the next directory entry in the directory stream pointed to by *dirp*. It returns NULL on reaching the end of the directory stream or if an error occurred.

In the glibc implementation, the *dirent* structure is defined as follows:

```
struct dirent {
    ino_t          d_ino;       /* Inode number */
    unsigned char  d_type;      /* Type of file */
    char           d_name[256]; /* Null-terminated filename */
};
```

The fields of the *dirent* structure are as follows:

d_ino
: This is the inode number of the file.

d_type
: This field contains a value indicating the file type, making it possible to avoid the expense of calling **lstat**(2) if further actions depend on the type of the file. *d_type*:

  DT_BLK This is a block device.

  DT_CHR This is a character device.

  DT_DIR This is a directory.

  DT_FIFO This is a named pipe (FIFO).

  DT_LNK This is a symbolic link.

  DT_REG This is a regular file.

  DT_SOCK This is a UNIX domain socket.

  DT_UNKNOWN The file type could not be determined.

  All applications must properly handle a return of **DT_UNKNOWN**.

d_name
: This field contains the null terminated filename.

The data returned by **readdir**() may be overwritten by subsequent calls to **readdir**() for the same directory stream.

RETURN VALUE

On success, **readdir**() returns a pointer to a *dirent* structure. (This structure may be statically allocated; do not attempt to **free**(3) it.)

If the end of the directory stream is reached, NULL is returned and *errno* is not changed. If an error occurs, NULL is returned and *errno* is set to indicate the error. To distinguish end of stream from an error, set *errno* to zero before calling **readdir**() and then check the value of *errno* if NULL is returned.

ERRORS

EBADF Invalid directory stream descriptor *dirp*.

NOTES

A directory stream is opened using **opendir**(3).

The order in which filenames are read by successive calls to **readdir**() depends on the filesystem implementation; it is unlikely that the names will be sorted in any fashion.

SEE ALSO

closedir(3), **opendir**(3)

NAME

stat, fstat – get file status

SYNOPSIS

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

int stat(const char *pathname, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

DESCRIPTION

These functions return information about a file, in the buffer pointed to by *buf*. No permissions are required on the file itself, but—in the case of **stat**()—execute (search) permission is required on all of the directories in *pathname* that lead to the file.

stat() retrieves information about the file pointed to by *pathname*.

fstat() is identical to **stat**(), except that the file about which information is to be retrieved is specified by the file descriptor *fd*.

All of these system calls return a *stat* structure, which contains the following fields:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* file type and mode */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */

    time_t st_atime;  /* Time of last access */
    time_t st_mtime;  /* Time of last modification */
    time_t st_ctime;  /* Time of last status change */
};
```

ERRORS

EACCES
: Search permission is denied for one of the directories in the path prefix of *pathname*. (See also **path_resolution**(7).)

EFAULT
: Bad address.

ELOOP
: Too many symbolic links encountered while traversing the path.

ENAMETOOLONG
: *pathname* is too long.

ENOENT
: A component of *pathname* does not exist or is a dangling symbolic link.

ENOENT
: *pathname* is an empty string and **AT_EMPTY_PATH** was not specified in *flags*.

ENOTDIR
: A component of the path prefix of *pathname* is not a directory.

SEE ALSO

inode(7)

NAME

inode – file inode information

DESCRIPTION

Each file has an inode containing metadata about the file. An application can retrieve this metadata using **stat**(2) (or related calls), which returns a *stat structure*

The following is a list of the information typically found in, or associated with, the file inode, with the names of the corresponding structure fields returned by **stat**(2):

File type and mode
: *stat.st_mode*

  See the discussion of file type and mode, below.

The file type and mode

The *stat.st_mode* field contains the file type and mode.

POSIX refers to the *stat.st_mode* bits corresponding to the mask **S_IFMT** (see below) as the *file type*, the 12 bits corresponding to the mask 07777 as the *file mode bits* and the least significant 9 bits (0777) as the *file permission bits*.

The following mask values are defined for the file type:

S_IFMT 0170000 bit mask for the file type bit field

S_IFSOCK 0140000 socket
S_IFLNK 0120000 symbolic link
S_IFREG 0100000 regular file
S_IFBLK 0060000 block device
S_IFDIR 0040000 directory
S_IFCHR 0020000 character device
S_IFIFO 0010000 FIFO

Thus, to test for a regular file (for example), one could write:

```
stat(pathname, &sb);
if ((sb.st_mode & S_IFMT) == S_IFREG) {
    /* Handle regular file */
}
```

SEE ALSO

stat(2)

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

dup, dup2 – duplicate a file descriptor

SYNOPSIS

#include <unistd.h>

int dup(int *oldfd***);**
int dup2(int *oldfd***, int** *newfd***);**

DESCRIPTION

The **dup**() system call creates a copy of the file descriptor *oldfd*, using the lowest-numbered unused file descriptor for the new descriptor.

After a successful return, the old and new file descriptors may be used interchangeably. They refer to the same open file description (see **open**(2)) and thus share file offset and file status flags; for example, if the file offset is modified by using **lseek**(2) on one of the file descriptors, the offset is also changed for the other.

The two file descriptors do not share file descriptor flags (the close-on-exec flag). The close-on-exec flag (**FD_CLOEXEC**; see **fcntl**(2)) for the duplicate descriptor is off.

dup2()

The **dup2**() system call performs the same task as **dup**(), but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in *newfd*. If the file descriptor *newfd* was previously open, it is silently closed before being reused.

RETURN VALUE

On success, these system calls return the new file descriptor. On error, −1 is returned, and *errno* is set appropriately.

NAME

stdin, stdout, stderr – standard I/O streams

SYNOPSIS

#include <stdio.h>

extern FILE ***stdin*;**
extern FILE ***stdout*;**
extern FILE ***stderr*;**

DESCRIPTION

Under normal circumstances every UNIX program has three streams opened for it when it starts up, one for input, one for output, and one for printing diagnostic or error messages. These are typically attached to the user's terminal (see **tty**(4)) but might instead refer to files or other devices, depending on what the parent process chose to set up. (See also the "Redirection" section of **sh**(1).)

Each of the corresponding symbols is a **stdio**(3) macro of type pointer to *FILE*, and can be used with functions like **fprintf**(3) or **fread**(3).

Since *FILE*s are a buffering wrapper around UNIX file descriptors, the same underlying files may also be accessed using the raw UNIX file interface, that is, the functions like **read**(2) and **lseek**(2).

On program startup, the integer file descriptors associated with the streams *stdin*, *stdout*, and *stderr* are 0, 1, and 2, respectively. The preprocessor symbols **STDIN_FILENO**, **STDOUT_FILENO**, and **STDERR_FILENO** are defined with these values in *<unistd.h>*. (Applying **freopen**(3) to one of these streams can change the file descriptor number associated with the stream.)

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

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_destroy, pthread_mutex_init — destroy and initialize a mutex

SYNOPSIS

#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *_mutex_);
int pthread_mutex_init(pthread_mutex_t *restrict _mutex_,
const pthread_mutexattr_t *restrict _attr_);

DESCRIPTION

The *pthread_mutex_destroy*() function shall destroy the mutex object referenced by *mutex*; the mutex object becomes, in effect, uninitialized. An implementation may cause *pthread_mutex_destroy*() to set the object referenced by *mutex* to an invalid value.

A destroyed mutex object can be reinitialized using *pthread_mutex_init*(); the results of otherwise referencing the object after it has been destroyed are undefined.

The *pthread_mutex_init*() function shall initialize the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Only *mutex* itself may be used for performing synchronization. The result of referring to copies of *mutex* in calls to *pthread_mutex_lock*(), *pthread_mutex_trylock*(), *pthread_mutex_unlock*(), and *pthread_mutex_destroy*() is undefined.

RETURN VALUE

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

ERRORS

The *pthread_mutex_init*() function shall fail if:

ENOMEM
Insufficient memory exists to initialize the mutex.

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

sched_yield – yield the processor

SYNOPSIS

```
#include <sched.h>

int sched_yield(void);
```

DESCRIPTION

The *sched_yield*() function shall force the running thread to relinquish the processor until it again becomes the head of its thread list. It takes no arguments.

RETURN VALUE

The *sched_yield*() function shall return 0 if it completes successfully; otherwise, it shall return a value of -1 and set *errno* to indicate the error.

ERRORS

No errors are defined.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-2001, *<sched.h>*

COPYRIGHT

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

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

strstr – locate a substring

SYNOPSIS

```
#include <string.h>

char *strstr(const char *haystack, const char *needle);
```

DESCRIPTION

The **strstr**() function finds the first occurrence of the substring *needle* in the string *haystack*. The terminating null bytes ('\0') are not compared.

The **strcasestr**() function is like **strstr**(), but ignores the case of both arguments.

RETURN VALUE

These functions return a pointer to the beginning of the located substring, or NULL if the substring is not found.