Prof. Dr. Frank Bellosa
Lukas Werling, M.Sc.
Dr. Marc Rittinghaus

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

Hauptklausur

Programmieren

02.03.2020

- 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 20 Blättern: 1 Deckblatt, 14 Aufgabenblättern mit insgesamt 3 Aufgaben und 5 Blättern Man-Pages.

  The examination consists of 20 pages: 1 cover sheet, 14 sheets containing 3 assignments, and 5 sheets for man pages.

- Es sind keinerlei Hilfsmittel erlaubt!

  No additional material is allowed.

- Die Prüfung ist nicht bestanden, wenn Sie aktiv oder passiv 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 task 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 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	15	15	15	45
Erreichte Punkte				
Note				

Aufgabe 1: C Grundlagen

Assignment 1: C Basics

a) Was gibt der unten stehende Code bei der Ausführung der Funktion `print_test()` aus? **1 pt**

What does the code below print when running the function `print_test()`?

```
struct test {
    int a, b, c;
};

void print_test(void) {
    struct test t = {1};
    printf("%d/%d/%d", t.a, t.b, t.c);
}
```

b) Geben Sie für alle Felder des unten stehenden `struct mystruct` jeweils die Größe des Feldes und die Größe des Paddings *nach* dem Feld in Byte an. Schreiben Sie "0", falls kein Padding eingefügt wird. Gehen Sie von einem 64-Bit-System aus. **3 pt**

For each field of the `struct mystruct` below, give the field's size and the size of the padding after *the field in Bytes. Write "0" if the compiler does not insert any padding. Assume a 64-bit system.*

Code	Field size [Byte]	Padding size [Byte]
`struct mystruct {`	—	—
`char a;`		
`uint32_t b;`		
`int16_t c;`		
`int64_t d;`		
`};`	—	—

c) Definieren Sie ein C-Makro `ARRAY_SIZE`, das die Anzahl der Elemente eines statisch allozierten Arrays berechnet. **1 pt**

Define a C macro `ARRAY_SIZE` that prints the number of elements in a statically allocated array.

Examples:

```
int array[13];
char str[100];
assert(ARRAY_SIZE(array) == 13);
assert(ARRAY_SIZE(str) == 100);
assert(~(~ARRAY_SIZE(array)) == 13);
```

Begründen Sie, warum das Makro `ARRAY_SIZE` nicht in der unten stehenden Funktion `sum()` eingesetzt werden kann. Welchen Wert gibt `ARRAY_SIZE` in der Funktion `sum()` auf einem 64-Bit-System tatsächlich zurück? Wie kann das Problem behoben werden? **1.5 pt**

Give a reason why the macro `ARRAY_SIZE` does not work in the function `sum()` given below. Which value does `ARRAY_SIZE` actually return in the function `sum()` on a 64-bit system? How can this issue be solved?

```
int32_t sum(int32_t *array) {
    int32_t result = 0;
    for (size_t i = 0; i < ARRAY_SIZE(array); i++)
        result += array[i];
    return result;
}

int main() {
    int32_t array[15];
    /* ... */
    int s = sum(array);
    /* ... */
}
```

d) Welches Problem tritt bei der Verwendung der Funktion `init_config()` auf? **1 pt**

Which problem occurs when using the function `init_config()`?

```
struct config {
    int verbose;
    int jobs;
    int dry_run;
};

struct config *init_config(int n) {
    struct config c = {
        .verbose = 0,
        .jobs = n,
        .dry_run = 0
    };
    return &c;
}
```

Schreiben Sie eine alternative Funktion `init_config2()`, die die gleiche Funktionalität wie `init_config()` bietet, aber das oben genannte Problem nicht hat. **1.5 pt**

Write an alternative function `init_config2()` that offers the same functionality as `init_config()` but does not have the problem above.

```
struct config *init_config2(int n) {
........................................................................................................................
........................................................................................................................
........................................................................................................................
........................................................................................................................
........................................................................................................................
........................................................................................................................
........................................................................................................................
........................................................................................................................
}
........................................................................................................................
```

e) Die `memset()`-Funktion der C-Standardbibliothek überschreibt einen Speicherbereich mit einem beliebigen Byte-Wert. In dieser Aufgabe sollen Sie eine Funktion `pattern_memset()` implementieren, die einen Speicherbereich mit einem 8 Bytes langem Muster wie im Beispiel unten überschreiben soll.

The `memset()` function in the C standard library overwrites a memory area with an arbitrary byte value. In this assignment, you will write a function `pattern_memset()` that overwrites a memory area with an 8 bytes long pattern as in the example below.

```
uint64_t pat = 0x123456789ABCDEF0ull;
char buf[14];
pattern_memset(buf, &pat, sizeof(buf));
```

In-memory view of `pat` (little endian):

Byte	0	1	2	3	4	5	6	7
pat	F0	DE	BC	9A	78	56	34	12

Result `buf`:

Byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13
buf	F0	DE	BC	9A	78	56	34	12	F0	DE	BC	9A	78	56

Implementieren Sie die Funktion `rotr()`, die eine gegebene 64-Bit-Ganzzahl um ein Byte (= 8 Bits) nach rechts rotiert. **1 pt**

Implement the function `rotr()` that rotates a 64-bit integer by one byte (= 8 bits) to the right.

```
uint64_t pat = 0x123456789ABCDEF0ull;
assert(rotr(pat) == 0xF0123456789ABCDEull);
```

In-memory view (little endian):

Byte	0	1	2	3	4	5	6	7
`pat`	F0	DE	BC	9A	78	56	34	12
`rotr(pat)`	DE	BC	9A	78	56	34	12	F0

```
uint64_t rotr(uint64_t v) {
...........................................................................

...........................................................................

...........................................................................

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

Implementieren Sie die Funktion `pattern_memset()`, die das Muster in `*pat` Byte für Byte in den Puffer `s` der Länge `n` schreibt. Nach dem Aufruf soll `*pat` passend rotiert sein, sodass ein weiterer Aufruf das Muster fortsetzt. **2 pt**

Implement the function `pattern_memset()` that writes the pattern in `*pat` byte by byte to the buffer `s` of length `n`. After the call, `*pat` shall be rotated appropriately so that a subsequent call continues the pattern.

```
uint64_t pat = 0x123456789ABCDEF0ull, pat1 = pat;
char buf[5];
pattern_memset(buf, &pat, 2);
uint64_t pat2 = pat;
assert(pat2 == 0xDEF0123456789ABCull);
pattern_memset(buf + 2, &pat, 3);
uint64_t pat3 = pat;
assert(pat3 == 0x789ABCDEF0123456ull);
```

In-memory view (little endian):

Byte	0	1	2	3	4	5	6	7
`pat1`	F0	DE	BC	9A	78	56	34	12
`pat2`	BC	9A	78	56	34	12	F0	DE
`pat3`	56	34	12	F0	DE	BC	9A	78
`buf`	F0	DE	BC	9A	78			

```
void pattern_memset(void *s, uint64_t *pat, size_t n) {
...........................................................................

...........................................................................

...........................................................................

...........................................................................

...........................................................................

...........................................................................

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

3 pt

Implementieren Sie die Funktion `pattern_memset_64()`, die das Muster in `*pat` in den Puffer `s` der Länge `n` schreibt und dabei die Zahl der Speicherzugriffe wie folgt reduziert:

- Wenn die Adresse von `s` an 8-Byte ausgerichtet ist (`(addr & 0x7) == 0`), soll `s` mit so vielen 8-Byte-Zugriffen wie möglich befüllt werden.
- Verwenden Sie `pattern_memset()` um restliche Bytes zu schreiben oder wenn die Adresse nicht ausgerichtet ist.
- Nach dem Aufruf soll `*pat` passend rotiert sein, sodass ein weiterer Aufruf das Muster fortsetzt.

Implement the function `pattern_memset_64()` that writes the pattern in `*pat` to the buffer `s` of length `n`. The function shall reduce the number of memory accesses as follows:

- *If the address of `s` is 8-byte aligned (`(addr & 0x7) == 0`), `s` has to be filled with as much 8-byte writes as possible.*
- *Use `pattern_memset()` to write remaining bytes or if the address is not aligned.*
- *After the call, `*pat` shall be rotated appropriately so that a subsequent call continues the pattern.*

Example 1: `s = (void *)0x1008, n = 10`

Address	0x1008								0x1010	
s	F0	DE	BC	9A	78	56	34	12	F0	DE
Access size	8								1	1

Example 2: `s = (void *)0x1006, n = 10`

Address	0x1006		0x1008							
s	F0	DE	BC	9A	78	56	34	12	F0	DE
Access size	1	1	1	1	1	1	1	1	1	1

```
void pattern_memset_64(void *s, uint64_t *pat, size_t n) {













}
```

Total: 15.0pt

Aufgabe 2: Log-Rotation

Assignment 2: Log Rotation

Sie sollen Funktionen schreiben, die eine Log-Datei in Textform verwalten. Damit das Log hierbei nicht beliebig viel Speicherplatz verbraucht, soll Ihr Code eine Rotation des Logs implementieren: Sobald die Log-Datei eine bestimmte Größe erreicht hat, soll sie an einen anderen Ort verschoben werden, wobei gegebenenfalls eine andere dort liegende alte Log-Datei überschrieben wird. Weitere Log-Einträge sollen in Folge in eine neue Datei am ursprünglichen Pfad geschrieben werden.

- Binden Sie die in den Teilaufgaben notwendigen C-Header in dem gekennzeichneten Bereich ein.
- Sie müssen in dieser Aufgabe keine Fehlerbehandlung implementieren.
- Geben Sie jegliche in ihrem Code angeforderten Resourcen wieder frei. Lediglich der aktuelle Dateideskriptor `log_fd` darf bei Programmende noch geöffnet sein.

You have to write functions to manage a log file in text form. To prevent the log from using an arbitrary amount of storage space, your code shall implement a rotation of the log: As soon as the log file reaches a certain size, the file shall be moved to another place, potentially overwriting another old log file in that place. New log entries then shall be written into a new file at the original path.

- *Include all required C headers in the marked area.*
- *You do not have to implement error handling.*
- *Free all resources allocated in your code. Only the current file descriptor `log_fd` is allowed to remain open at the end of the program.*

```
/* include statements for the required C headers */
.................................................................
.................................................................
.................................................................
.................................................................
.................................................................
.................................................................
.................................................................
.................................................................
.................................................................

#define PATH "my_app.log"            /* log file path */
#define ROTATED_PATH "my_app.log.2"  /* path to which the log file is moved */
#define MAX_SIZE 1000000             /* maximum log size in bytes */

int log_fd;          /* file descriptor of the log file */
uint64_t log_size;   /* current size of the log file in bytes */

/* mutex for global variables, initialized and ready to be used */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

a) Die Variable `log_size` hat den Typ `uint64_t`. Welchen Header müssen Sie für diesen Typ inkludieren? **0.5 pt**

The variable `log_size` is of the type `uint64_t`. Which header do you need to include for this type?

---

b) Vervollständigen Sie die Funktion `log_init()`, die zu Beginn des Programmes aufgerufen wird und die Log-Datei öffnet, ihre aktuelle Größe bestimmt und den Dateideskriptor sowie die Größe in den globalen Variablen speichert. **6.5 pt**

- Falls die Datei noch nicht existiert, soll sie erstellt werden.
- Auf die Log-Datei sollen der Eigentümer lesend und schreibend und alle anderen Benutzer nur lesend zugreifen können.
- Folgende `write()`-Aufrufe sollen die Daten ans Ende der Datei anhängen.

Complete the function `log_init()` which is called at the start of the program and which opens the log file, determines its current size, and stores the file descriptor as well as the size in the global variables.

- *If the file does not exist yet, it shall be created.*
- *The owner shall access the file for reading and writing, whereas other users shall only be able to read the file.*
- *Following `write()` calls shall append the data to the end of the file.*

```
void log_init(void) {
.......................................................................
.......................................................................
.......................................................................
.......................................................................
.......................................................................
.......................................................................
.......................................................................
}
.......................................................................
```

c) Vervollständigen Sie die Funktion `log_write()`, die den übergebenen Text in das Log schreibt und durch den Aufruf von `rotate_log()` eine Rotation des Logs auslöst, wenn durch den Schreibvorgang die durch `MAX_SIZE` festgelegten Größe erreicht oder überschritten wurde. **5.5 pt**

- Die Funktion soll von mehreren Threads parallel aufgerufen werden können. Verwenden Sie den globalen Mutex `mutex`, um Operationen wo nötig zu synchronisieren.
- Sie können davon ausgehen, dass bei einem Schreibvorgang die dem Systemaufruf übergebenen Daten immer vollständig geschrieben werden.

Complete the function `log_write()` which writes the passed string into the log and which triggers a rotation of the log by calling `rotate_log()` if the size specified by `MAX_SIZE` was reached or exceeded by the write operation.

- *It shall be possible to call the function from multiple threads in parallel. Use the global mutex `mutex` to synchronize operations where needed.*
- *You can assume that during a write operation the data passed to the system call is always written completely.*

```
void rotate_log(void);
void log_write(const char *text) {
...........................................................
...........................................................
...........................................................
...........................................................
...........................................................
...........................................................
...........................................................
...........................................................
...........................................................
...........................................................
}
...........................................................
```

d) Vervollständigen Sie die Funktion `rotate_log()`, die die Inhalte des aktuellen Logs an den Pfad `ROTATED_LOG` verschiebt und unter Nutzung der Funktion `log_init()` eine neue Log-Datei am ursprünglichen Pfad anlegt. **2.5 pt**

Complete the function `rotate_log()` which moves the contents of the current log to the path `ROTATED_LOG` and which creates a new log file at the original path by using the function `log_init()`.

```
void log_init(void);
void rotate_log(void) {
...........................................................
...........................................................
...........................................................
...........................................................
...........................................................
...........................................................
}
...........................................................
```

Total: 15.0pt

Aufgabe 3: Dateisystemimplementierung

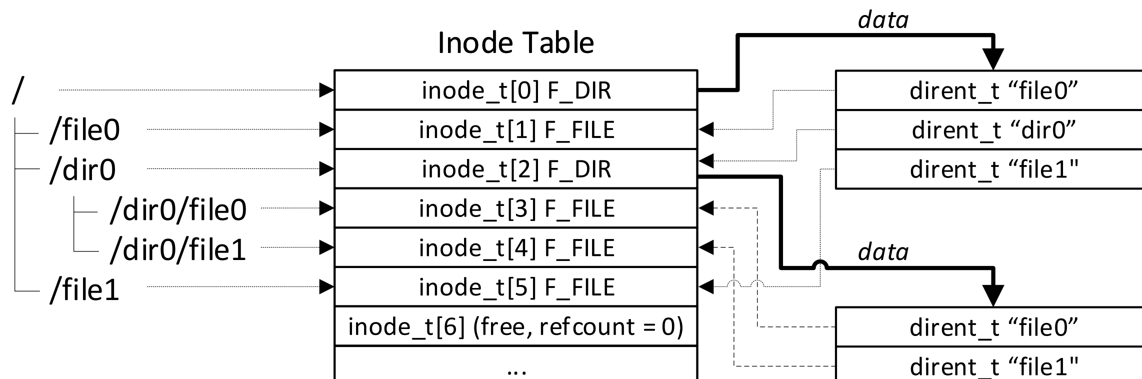
Assignment 3: File System Implementation

Im Folgenden sollen verschiedene Funktionen eines einfachen Dateisystems implementiert werden. Dateien und Ordner werden als Inodes (`inode_t`) repräsentiert, die in einer statischen Inode-Tabelle gespeichert werden. Ordner unterscheiden sich von regulären Dateien, indem der entsprechender Typ (`F_DIR`) im Inode hinterlegt ist und die referenzierten Daten aus einer Liste von Verzeichniseinträgen (`dirent_t`) bestehen.

- Sofern nicht anders genannt, gehen Sie davon aus, dass alle Parameter valide Werte enthalten und das Dateisystem nicht beschädigt ist.

In the following you have to implement various functions of a simple file system. Files and directories are represented by inodes (`inode_t`), which are stored in a static inode table. Directories differ from regular files in that the corresponding type (`F_DIR`) is assigned to the inode and the referenced data is a list of directory entries (`dirent_t`).

- *Unless otherwise specified, assume that all parameters contain valid values and that the file system is not damaged.*




```c
typedef enum {
    F_FILE = 0,         /* the inode is a regular file */
    F_DIR  = 1          /* the inode is a directory */
} file_type;

typedef struct {
    uint32_t refcount;  /* number of references to inode, 0 = free */
    off_t length;       /* length of the file in bytes */

    file_type type;     /* type of file */

    /* ... */
} inode_t;

#define ROOT_DIR 0      /* inode number of root directory */
#define MAX_INODES 64
typedef struct {
    inode_t inodes[MAX_INODES];  /* static inode table */
} filesystem;
```

```c
#define MAX_NAME 15      /* max length of file/directory name */
typedef struct {
    int inode;           /* inode number referenced by directory entry */

    char name[MAX_NAME + 1];      /* MAX_NAME chars + zero byte */
} dirent_t;
```

a) Vervollständigen Sie die Funktion `fs_alloc_inode()`, die einen freien Inode aus der Inode-Tabelle alloziert und dessen Nummer zurückgibt. **3 pt**

- Setzen Sie den Inode-Referenzzähler auf 1, die Länge auf 0 und konfigurieren Sie den Dateityp.
- Geben Sie -1 zurück, falls kein freier Inode existiert.

Complete the function `fs_alloc_inode()`, which allocates a free inode from the inode table and returns its number.

- *Set the inode's reference counter to 1, the length to 0, and configure the file type.*
- *Return -1 if no free inode exists.*

```c
int fs_alloc_inode(filesystem* fs, file_type type) {
................................................................................................
................................................................................................
................................................................................................
................................................................................................
................................................................................................
................................................................................................
................................................................................................
................................................................................................
................................................................................................
................................................................................................
................................................................................................
................................................................................................
................................................................................................
................................................................................................
................................................................................................
................................................................................................
}
................................................................................................
```

b) Vervollständigen Sie die Funktion `fs_alloc_dirent()`, die einen `dirent_t`-Eintrag an ein Verzeichnis anfügt. **2 pt**

- `dir` ist die Inode-Nummer des Verzeichnisses, `inode` gibt die Inode-Nummer und `name` den Namen des neuen Eintrags an. Gehen Sie davon aus, dass dieser nicht länger als `MAX_NAME` ist und der Referenzzähler der Inode bereits angepasst ist.
- Nutzen Sie die Funktion `fs_pwrite()`, um Daten zu schreiben.

Complete the function `fs_alloc_dirent()`, which appends a `dirent_t` entry to a directory.

- *`dir` is the inode number of the directory, `inode` supplies the inode number and `name` specifies the name of the new entry. Assume that the name is not longer than `MAX_NAME` and that the reference counter of the inode has already been adjusted.*
- *Use the function `fs_pwrite()` to write data.*

```
/* writes count bytes of buf into the data area of the inode using the
   given offset (in bytes). Updates file length. Never fails. */
void fs_pwrite(filesystem* fs, int inode, void* buf, size_t count,
    off_t offset);

void fs_alloc_dirent(filesystem* fs, int dir, int inode, const char* name) {
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
}
.....................................................................................
```

c) Vervollständigen Sie die Funktion `fs_find()`, die in einem Verzeichnis nach einem passenden Eintrag zur gegeben Pfadkomponente sucht und deren Inode-Nummer zurückgibt. **4.5 pt**

- `dir` ist die Inode-Nummer des Verzeichnisses. Die Pfadkomponente umfasst die ersten `len` Zeichen des Strings `path`. Gehen Sie davon aus, dass `len` ≤ `MAX_NAME` ist.
- Nutzen Sie die Funktion `fs_pread()`, um Daten zu lesen.
- Geben Sie `-ENOENT` zurück, wenn kein passender Eintrag existiert.

Complete the function `fs_find()`, which searches in a directory for a matching directory entry to the given path component and returns its inode number.

- *`dir` is the inode number of the directory. The path component consists of the first `len` characters of the `path` string. Assume `len` ≤ `MAX_NAME`.*
- *Use the function `fs_pread()` to read data.*
- *Return `-ENOENT` if no matching entry exists.*

```
/* reads count bytes from inode's data area into buf using the given
   offset (in bytes). Returns the number of bytes read. Returns 0, when
   reading beyond the end of the data area; otherwise never fails. */
size_t fs_pread(filesystem* fs, int inode, void* buf, size_t count,
    off_t offset);

int fs_find(filesystem* fs, int dir, const char* path, size_t len) {
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
}
.....................................................................
```

d) Vervollständigen Sie die Funktion `fs_open()`, die den absoluten Pfad `path` traversiert und die entsprechende Inode-Nummer der bezeichneten Datei zurückgibt. **5.5 pt**

- `inode` ist zu Beginn die Inode-Nummer des Wurzelverzeichnisses. Sie können `fs_open()` rekursiv aufrufen.
- Benutzen Sie die Funktion `split_path()`, um die nächste Pfadkomponente zu erhalten. Gehen Sie davon aus, dass diese nicht länger als `MAX_NAME` ist. Der Pfad hat die Form `/dir/file"` mit einer beliebigen Anzahl von Unterverzeichnissen.
- Sie können die Funktion `fs_find()` nutzen.
- Fehlt eine Pfadkomponente, geben Sie `-ENOENT` zurück.
- Enthält `flags` das `O_DIR` flag, öffnet die Funktion ausschließlich Verzeichnisse und gibt für Dateien `-ENOTDIR` zurück. Im umgekehrten Fall (d. h. `O_DIR` ist nicht gesetzt und das Ziel ist ein Verzeichnis) geben Sie `-EISDIR` zurück.

Complete the function `fs_open()`, which traverses the absolute path `path` and returns the inode number of the corresponding file.

- *Initially, `inode` is the inode number of the root directory. You may call `fs_open()` recursively.*
- *Use the function `split_path()` to get the next path component. Assume that each component is not longer than `MAX_NAME`. The path has the form `"/dir/file"` with an arbitrary number of subdirectories.*
- *You may use the function `fs_find()`.*
- *If a path component is missing, return `-ENOENT`.*
- *If `flags` contains the `O_DIR` flag, the function only opens directories and returns `-ENOTDIR` for files. In the opposite case (i.e., `O_DIR` is not set and `path` is a directory), return `-EISDIR`.*

```c
/* returns the beginning of the next path component within the same path
   string. len returns the length of the path component in characters.
   The function skips leading path separators ('/'). Examples:
     "/dir/file" => "dir/file", len=3
     "file"      => "file", len=4
     ""          => "", len=0 */
const char* split_path(const char* path, size_t* len);

typedef enum {
    O_NONE   = 0,         /* open file entry, -ENOENT if not existing */
    O_DIR    = 1          /* open directory instead of file */
} open_flags;

/* returns the inode number of entry with name [path,path+len] in
   directory dir, or -ENOENT if no such entry exists. */
int fs_find(filesystem* fs, int dir, const char* path, size_t len);
```

```c
int fs_open(filesystem* fs, int inode, const char* path, open_flags flags) {
    inode_t* node = &fs->inodes[inode];
}
```

Total: 15.0pt

NAME

malloc, free − allocate and free dynamic memory

SYNOPSIS

#include <stdlib.h>

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

DESCRIPTION

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

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

RETURN VALUE

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

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

ERRORS

malloc() can fail with the following error:

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

NOTES

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

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

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

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

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

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

NAME

open − open and possibly create a file

SYNOPSIS

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

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

DESCRIPTION

The **open**() system call opens the file specified by *pathname*. If the specified file does not exist, it may optionally (if **O_CREAT** is specified in *flags*) be created by **open**().

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. 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-*or*'d in *flags*. The *file creation flags* are **O_CLOEXEC**, **O_CREAT**, **O_DIRECTORY**, **O_EXCL**, **O_NOCTTY**, **O_NOFOLLOW**, **O_TMPFILE**, and **O_TRUNC**. 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 abridged list of file creation flags and 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_CLOEXEC (since Linux 2.6.23)
Enable the close-on-exec flag for the new file descriptor. Specifying this flag permits a program to avoid additional **fcntl**(2) **F_SETFD** operations to set the **FD_CLOEXEC** flag.

Note that the use of this flag is essential in some multithreaded programs, because using a separate **fcntl**(2) **F_SETFD** operation to set the **FD_CLOEXEC** flag does not suffice to avoid race conditions where one thread opens a file descriptor and attempts to set its close-on-exec flag using **fcntl**(2) at the same time as another thread does a **fork**(2) plus **execve**(2). Depending on the order of execution, the race may lead to the file descriptor returned by **open**() being unintentionally leaked to the program executed by the child process created by **fork**(2). (This kind of race is in principle possible for any system call that creates a file descriptor whose close-on-exec flag should be set, and various other Linux system calls provide an equivalent of the **O_CLOEXEC** flag to deal with this problem.)

O_CREAT
If *pathname* does not exist, create it as a regular file.

The owner (user ID) of the new file is set to the effective user ID of the process.

The *mode* argument specifies the file mode bits be applied when a new file is created. This argument must be supplied when **O_CREAT** or **O_TMPFILE** is specified in *flags*; if neither **O_CREAT** nor **O_TMPFILE** is specified, then *mode* is ignored. The effective mode is modified by the process's *umask* in the usual way: in the absence of a default ACL, the mode of the created file is *(mode & ~umask)*. Note that this mode applies only to future accesses of the newly created file; the **open**() call that creates a read-only file may well return a read/write file descriptor.

The following symbolic constants are provided for *mode*:

S_IRWXU
00700 user (file owner) has read, write, and execute permission

S_IRUSR
00400 user has read permission

S_IWUSR
00200 user has write permission

S_IXUSR
00100 user has execute permission

S_IRWXG
00070 group has read, write, and execute permission

S_IRGRP
00040 group has read permission

S_IWGRP
00020 group has write permission

S_IXGRP
00010 group has execute permission

S_IRWXO
00007 others have read, write, and execute permission

S_IROTH
00004 others have read permission

S_IWOTH
00002 others have write permission

S_IXOTH
00001 others have execute permission

According to POSIX, the effect when other bits are set in *mode* is unspecified.

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

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

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf − formatted output conversion

SYNOPSIS

#include <stdio.h>

int printf(const char ****format*, ...);**
int fprintf(FILE ****stream*, const char ****format***, ...);**

DESCRIPTION

The functions in the **printf**() family produce output according to a *format* as described below. The function **printf**() writes output to *stdout*, the standard output stream; **fprintf**() writes output to the given output *stream*.

These functions write the output under the control of a *format* string that specifies how subsequent arguments are converted for output.

Return value

Upon successful return, these functions return the number of characters printed (not including the trailing '\0' used to end output to strings).

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

Format of the format string

The format string is a character string, beginning and ending in its initial shift state, if any. The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character **%**, and ends with a *conversion specifier*.

The arguments must correspond properly (after type promotion) with the conversion specifier. By default, the arguments are used in the order given, where each conversion specifier asks for the next argument (and it is an error if insufficiently many arguments are given).

The conversion specifier

A character that specifies the type of conversion to be applied. The conversion specifiers and their meanings are:

d, **i** The *int* argument is converted to signed decimal notation.

c The *int* argument is converted to an *unsigned char*, and the resulting character is written.

s The *const char ** argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating null byte ('\0').

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

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1, 2003 Edition, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 6, Copyright (C) 2001-2003 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at http://www.opengroup.org/unix/online.html .

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

rename − change the name or location of a file

SYNOPSIS

#include <stdio.h>

int rename(const char ***_oldpath_, const char ***_newpath_**);**

DESCRIPTION

rename() renames a file, moving it between directories if required. Any other hard links to the file (as created using **link**(2)) are unaffected. Open file descriptors for _oldpath_ are also unaffected.

If _newpath_ already exists, it will be atomically replaced, so that there is no point at which another process attempting to access _newpath_ will find it missing. However, there will probably be a window in which both _oldpath_ and _newpath_ refer to the file being renamed.

If _oldpath_ and _newpath_ are existing hard links referring to the same file, then **rename**() does nothing, and returns a success status.

If _newpath_ exists but the operation fails for some reason, **rename**() guarantees to leave an instance of _newpath_ in place.

oldpath can specify a directory. In this case, _newpath_ must either not exist, or it must specify an empty directory.

If _oldpath_ refers to a symbolic link, the link is renamed; if _newpath_ refers to a symbolic link, the link will be overwritten.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and _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

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 */
};
```

The field _st_atime_ is changed by file accesses, for example, by **execve**(2), **mknod**(2), **pipe**(2), **utime**(2) and **read**(2) (of more than zero bytes). Other routines, like **mmap**(2), may or may not update _st_atime_.

The field _st_mtime_ is changed by file modifications, for example, by **mknod**(2), **truncate**(2), **utime**(2) and **write**(2) (of more than zero bytes). Moreover, _st_mtime_ of a directory is changed by the creation or deletion of files in that directory. The _st_mtime_ field is _not_ changed for changes in owner, group, hard link count, or mode.

The field _st_ctime_ is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and _errno_ is set appropriately.

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

strcpy, strncpy – copy a string

SYNOPSIS

#include <string.h>

char *strcpy(char ***_dest_, const char ***_src_**);**
char *strncpy(char ***_dest_, const char ***_src_**, size_t** _n_**);**

DESCRIPTION

The **strcpy**() function copies 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

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_.