Karlsruher Institut für Technologie

Betriebssysteme
(Operating Systems)

Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

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

Scheinklausur
27.03.2017

- 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 pages (including draft pages).
- Die Prüfung besteht aus 23 Blättern: 1 Deckblatt, 19 Aufgabenblättern mit insgesamt 3 Aufgaben und 3 Blättern Man-Pages.
  The examination consists of 23 pages: 1 cover sheet, 19 sheets containing 3 assignments, and 3 sheets for man pages.
- Es sind keinerlei Hilfsmittel erlaubt!
  No additional material is allowed.
- Die Prüfung gilt als nicht bestanden, wenn Sie versuchen, aktiv oder passiv zu betrügen.
  You fail the examination if you try to cheat actively or passively.
- Wenn Sie zusätzliches Konzeptpapier benötigen, verständigen Sie bitte die Klausuraufsicht.
  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	20	20	20	60
Erreichte Punkte				
Note				

Aufgabe 1: C Grundlagen

Assignment 1: C Basics

a) Eine zweidimensionale Matrix mit 64 Spalten wird auf ein Array abgebildet, indem die Zeilen der Matrix wie folgt aneinandergehängt in das Array geschrieben werden:

A two-dimensional matrix with 64 columns is to be mapped into an array. To that end, the rows of the matrix are concatenated and written into the array as follows:

row 0	row 1	row 2	row 3	···

Schreiben Sie eine Funktion, die aus den Zeilen- (`row`) und Spaltenkoordinaten (`col`) eines Matrixelements den zugehörigen Arrayindex berechnet und diesen zurückgibt. **1 pt**

Write a function that returns the corresponding array index from the row (`row`) and column coordinates (`col`) of a matrix element.

```
int get_array_index(int row, int col) {
.....
.....
.....
}
.....
```

Schreiben Sie eine Funktion, die aus dem Arrayindex `index` die Koordinaten des zugehörigen Matrixelements berechnet und diese in den Variablen `row` und `col` ablegt. **1.5 pt**

Write a function that calculates the coordinates of the corresponding matrix element from the array index `index` and puts these coordinates into the variables `row` and `col`.

```
void get_matrix_coordinates(int index, int *row, int *col) {
.....
.....
.....
.....
}
.....
```

b) Woran lässt sich in einem 32-Bit-System typischerweise erkennen, dass eine Adresse auf eine Datenstruktur im Kernel verweist? **1 pt**

On a 32 bit system, how can you typically tell if a given address references a kernel data structure?

c) Welchen Wert hat die Variable `i` nach der Ausführung? Begründen Sie Ihre Antwort. **2 pt**

What value does the variable `i` have after execution? Explain your answer.

```
int i = 1 - (uint8_t)(-1);
```

d) Was berechnet die folgende Funktion? Begründen Sie Ihre Antwort. **2 pt**

What does the following function calculate? Explain your answer.

```
unsigned int calc(unsigned int a) {
    int b = ~((1 << 12) - 1);
    return a & b;
}
```

e) Erklären Sie den Begriff *Speicherleck* ("Memory Leak"). Nennen Sie darüber hinaus eine weitere Betriebssystemressource, auf die diese Problemklasse häufig zutrifft. **1.5 pt**

Explain the term "memory leak". In addition, give another operating system resource that is often leaked.

f) Betrachten Sie die folgenden zwei alternativen Funktionssignaturen:

Consider the following two alternative function signatures:

① `void f(struct mystruct_t s);`

② `void f(struct mystruct_t *s);`

Erläutern Sie, worin sich die Signaturen semantisch unterscheiden. Gehen Sie dabei kurz auf Vor- und Nachteile ein. **2 pt**

Explain how the signatures differ semantically and briefly discuss pros and cons.

Ergänzen Sie einen Aufruf der zweiten Variante und übergeben Sie `s`. **0.5 pt**

Add a call to the second version, supplying `s`.

```
struct mystruct_t s = {0};
```

Nennen Sie einen typischen Anwendungsfall für folgende Variante: **1 pt**

Give a typical use-case for the following version:

```
void f(struct mystruct_t **s);
```

g) Können Linux-Prozesse jedes Signal, das an sie gesendet wird, selbst behandeln? Begründen Sie Ihre Antwort. **1 pt**

Are Linux processes able to handle every signal that might be sent to them? Justify your answer.

h) Kreuzen Sie für jede der folgenden Funktionen in Linux an, ob diese immer, manchmal oder nie einen Systemaufruf ausführt. **2.5 pt**

For each of the following functions in Linux, mark whether it always, sometimes, or never executes a system call.

Immer/ *Always*	Manchmal/ *Sometimes*	Nie/ *Never*	
☐	☐	☐	`close()`
☐	☐	☐	`pthread_mutex_lock()`
☐	☐	☐	`memset()`
☐	☐	☐	`malloc()`
☐	☐	☐	`mq_unlink()`

i) Warum ist es für Entwickler wichtig, zu wissen, welche Funktionen Systemaufrufe ausführen und welche nicht? Geben Sie ein Beispiel an. **2 pt**

Why do developers need to know which functions cause system calls and which do not? Give an example.

j) Welche Systemaufrufe benötigt eine Shell, um ein externes Programm im Hintergrund zu starten? Wofür werden diese Funktionen jeweils benötigt? **2 pt**

Which system calls does a shell need to start an external program in the background? What is each function needed for?

Total: 20.0pt

Aufgabe 2: Dateikompression

Assignment 2: File Compression

Schreiben Sie ein Programm zur Kompression von Dateien. Die Pfade der Eingabedateien sowie die entsprechenden Ausgabepfade für die komprimierten Daten sollen als Paare über Kommandozeilenargumente übergeben werden. Ein Aufruf des Programms könnte wie folgt aussehen:

```
compress a.txt a.txt.out b.txt b.txt.out
```

Das Programm lädt dabei die Inhalte der Dateien `a.txt` und `b.txt` in den Hauptspeicher, komprimiert diese und schreibt sie in die jeweiligen Ausgabedateien (z. B. `a.txt` → `a.txt.out`).

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

Write a program that compresses files. The paths to input files as well as the corresponding output paths for the compressed data are passed to the program as pairs of command line arguments. As an example, consider the following command line:

```
compress a.txt a.txt.out b.txt b.txt.out
```

The program loads the contents of the files `a.txt` *and* `b.txt` *into memory, compresses them, and writes them into the corresponding output files (e.g.,* `a.txt` → `a.txt.out`*).*

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

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

a) Vervollständigen Sie die Funktion `get_file_size()`, die die Größe der Datei an dem angegebenen Pfad zurückgibt. **2 pt**

Complete the function `get_file_size()`, which returns the size of the file at the specified path.

```
size_t get_file_size(const char *path) {
.....

.....

.....

.....

.....

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

b) Vervollständigen Sie die Funktion `read_file()`, die die Datei an dem gegebenen Pfad `path` öffnet und die ersten `buf_len` Bytes der Datei mithilfe des `read()` Systemaufrufs in den Puffer `buf` liest. Die Funktion gibt in jedem Fall die Anzahl tatsächlich gelesener Bytes zurück.

- Schreiben Sie nicht über das Ende des Puffers hinaus (`buf_len`).
- Implementieren Sie Fehlerbehandlung für sämtliche Systemaufrufe. Die Funktion soll auch dann `buf_len` Bytes einlesen, wenn Signale während der Ausführung der Funktion auftreten. Sonstige Fehler (inkl. Erreichen des Dateiendes) sollen zum Abbruch des Lesevorgangs führen.

6.5 pt

Complete the function `read_file()`, which opens the file at the specified path `path` and reads the first `buf_len` bytes of the file into the buffer `buf` using the `read()` system call. In any case, the function returns the actual number of bytes read.

- *Do not write beyond the end of the buffer (`buf_len`).*
- *Implement error handling for all system calls. The function shall read `buf_len` bytes, even if signals occur during the execution of the function. Other errors (incl. reaching the end of file) shall abort the read operation.*

```
size_t read_file(const char *path, char *buf, size_t buf_len) {
.....

.....

.....

.....

.....

.....

.....

.....

.....
```

```
}
```

c) Der Systemaufruf `mmap()` könnte ebenfalls für den Zugriff auf die Dateiinhalte genutzt werden. Nennen Sie einen Vor- und einen Nachteil dieses Ansatzes. Begründen Sie Ihre Antwort. **2 pt**

The system call `mmap()` could also be used to access the file contents. Give one advantage and one disadvantage of this approach. Justify your answer.

d) Vervollständigen Sie die Funktion `compress_file()`, die die Eingabedatei an Pfad `in` vollständig einliest, die Daten komprimiert und in die Ausgabedatei an Pfad `out` schreibt.

- Verwenden Sie `get_file_size()` und `read_file()`.
- Verwenden Sie die Funktion `write_compressed()`, die einen Puffer komprimiert und an den gegebenen Ausgabepfad schreibt. Existierende Inhalte der Zieldatei werden bei einem Aufruf gelöscht.
- Die Eingabedatei wird während des kompletten Vorgangs nicht modifiziert.

3 pt

Complete the function `compress_file()`, which reads the entire input file at the path `in` and writes the compressed data to the output paths `out`.

- *Use `get_file_size()` and `read_file()`.*
- *Use the function `write_compressed()`, which compresses a buffer and writes it to the given output path. A call will delete existing contents in the output file.*
- *The input file is not modified during the entire operation.*

```
size_t get_file_size(const char *path);
size_t read_file(const char *path, char *buf, size_t buf_len);
void write_compressed(const char *path, const char *buf, size_t buf_len);
void compress_file(const char *in, const char *out) {
    exit(0); /* Always terminate process at the end of compression */
}
```

e) Vervollständigen Sie die Funktion `spawn_process()`, die einen separaten Prozess erzeugt, in dem die gegebene Eingabedatei asynchron komprimiert wird.

- Verwenden Sie `compress_file()` zur Kompression.
- Die Funktion soll sicherstellen, dass nie mehr als 4 zusätzliche Prozesse gleichzeitig ausgeführt werden.
- Sie können globale Variablen in dem zu Beginn der Aufgabe 2 reservierten Platz einfügen.

3.5 pt

Complete the function `spawn_process()`, which creates a separate process that asynchronously compresses the specified input file.

- *Use `compress_file()` for compression.*
- *The function shall ensure that there are never more than 4 additional processes running at the same time.*
- *You may add global variables to the reserved space at the beginning of Assignment 2.*

```
void compress_file(const char *in, const char *out);
void spawn_process(const char *in, const char *out) {
}
```

f) Vervollständigen Sie die `main()`-Funktion des Programms, die für je zwei aufeinanderfolgende Kommandozeilenargumente `spawn_process()` aufruft.

- Das Programm soll sich erst beenden, nachdem alle zusätzlich gestarteten Prozesse terminiert sind.
- Gehen Sie von einer beliebigen Anzahl von Kommandozeilenargumenten aus. Bei ungerader Anzahl kann das letzte Argument ignoriert werden.

3 pt

Complete the program's `main()` function, which calls `spawn_process()` for each pair of two consecutive command line arguments.

- *The program shall terminate only after all additionally spawned processes have terminated.*
- *Assume an arbitrary number of command line arguments. In case of an odd number of arguments, the last one can be ignored.*

```
void spawn_process(const char *in, const char *out);

int main(int argc, char **argv) {









}
```

Total: 20.0pt

Aufgabe 3: Buddy-Speicherallokation

Assignment 3: Buddy Memory Allocation

Für die Verwaltung des physischen Speicherbereichs $[0; 1\,\text{GiB})$ kommt in einem System ein Buddy-Allokator zum Einsatz. Der Allokator ist als Binärbaum implementiert, dessen Knoten mit der `node`-Struktur beschrieben werden. Jeder Knoten repräsentiert einen Speicherbereich der Länge $2^{\texttt{order}}$, der an der Adresse `base` beginnt. `free` ist für freie Blätter 1, ansonsten 0. Kindknoten werden mit den Zeigern `l` (links) und `r` (rechts) referenziert. `p` zeigt auf den Elternknoten. `root` ist die Wurzel des Baums.

Abbildung 3.1 zeigt den Binärbaum (a) vor der ersten Allokation, (b) nach einer Allokation von 256 MiB und (c) nach einer weiteren Allokation von 512 MiB und einer anschließenden Freigabe der ersten 256 MiB. Die Blätter der Größe 2^{28} wurden wieder vereint.

A system uses a buddy allocator to manage the physical memory area $[0; 1\,\text{GiB})$. The allocator is implemented as a binary tree, whose nodes are described by the `node` structure. Each node represents a memory area of size $2^{\texttt{order}}$, which starts at address `base`. `free` is 1 for free leaves, 0 otherwise. Child nodes are referenced via `l` (left) and `r` (right). `p` points to the parent node. `root` is the root of the tree.

Figure 3.1 depicts the binary tree (a) before the first allocation, (b) after an allocation of 256 MiB, and (c) after another allocation of 512 MiB and a following free of the first 256 MiB. The leaves of size 2^{28} have been merged again.

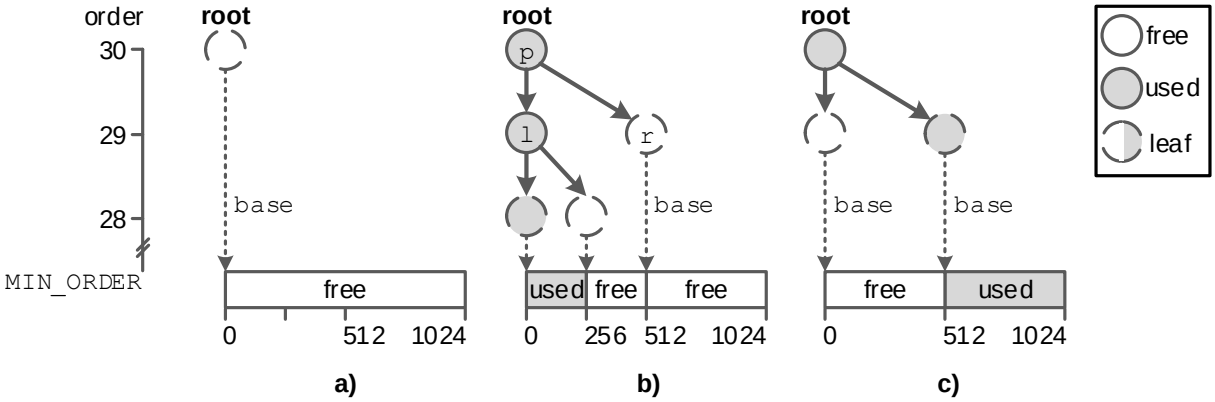

Abbildung 3.1 / *Figure 3.1*

```
typedef struct node {
    int order;          // Order of node's represented area
    void* base;         // Base address of represented area
    int free;           // = 1 for free leaves, = 0 otherwise

    struct node *p;     // Pointer to parent node, NULL for root
    struct node *l;     // Pointer to left child, NULL for leaves
    struct node *r;     // Pointer to right child, NULL for leaves
} node;

// Minimum valid allocation size is one frame (4 KiB)
#define MIN_ORDER 12

// BitScanReverse (BSR) - Returns the position of the first most
// significant bit (MSB) set (e.g., 0b00001010 -> 3). This equals log2(s).
int bsr(size_t s);
```

```
node root = {
    .order = 30,          // 2^30 = 1 GiB
    .base  = (void*)0x0, // Starts at address 0x0
    .free  = 1,           // Node is a free leaf

    .p     = NULL,        // Root has no parent
    .l     = NULL,        // No children, yet.
    .r     = NULL
};
```

a) Vervollständigen Sie die Funktion `_order()`, die eine positive Ganzzahl auf die nächste Zweierpotenz aufrundet und den Wert des Exponenten zurückgibt.
Beispiel: $6523 \rightarrow 8192 = 2^{13}$, Rückgabe: 13.

- Ist der Exponent kleiner als `MIN_ORDER`, wird `MIN_ORDER` zurückgegeben.
- Ist die Eingabe bereits eine Zweierpotenz, findet keine Aufrundung statt.
- Sie können die Funktion `bsr()` verwenden.
- Gehen Sie davon aus, dass `size_t` in jedem Fall ausreichend ist.

2.5 pt

Complete the function `_order()`, which rounds a positive integer up to the next power of two and returns the exponent. Example: $6523 \rightarrow 8192 = 2^{13}$, *Return:* 13.

- *If the exponent is less than `MIN_ORDER`, the function returns `MIN_ORDER`.*
- *If the input is already a power of two, the value is not rounded up.*
- *You can use the function `bsr()`.*
- *Assume `size_t` to be sufficient in any case.*

```
int _order(size_t s) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
.....
```

b) Vervollständigen Sie die Funktionen `_nalloc()` und `_nfree()`, die eine neue `node`-Struktur allozieren, initialisieren und zurückgeben bzw. wieder freigeben. Gehen Sie davon aus, dass dynamische Speicherallokationen nicht fehlschlagen, der Speicher jedoch nicht vorinitialisiert wurde. **2.5 pt**

Complete the functions `_nalloc()` and `_nfree()`, which allocate, initialize, and return a new `node` structure, or release it, respectively. Assume dynamic memory allocations to not fail. However, the memory is not pre-initialized.

```
node* _nalloc(int order) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
.....

void _nfree(node *n) {
.....
.....
}
.....
```

c) Vervollständigen Sie die Funktion `_isLeaf()`, die überprüft, ob ein Knoten ein Blatt ist. In dem Fall wird 1 zurückgegeben, ansonsten 0. **1 pt**

Complete the function `_isLeaf()`, which tests if a node is a leaf. In that case, it returns 1, 0 otherwise.

```
int _isLeaf(node *n) {
.....
.....
.....
.....
}
.....
```

d) Vervollständigen Sie die Funktion `_nsplit()`, die den Knoten `n` im Binärbaum in zwei gleichgroße Kindknoten aufteilt (siehe Abbildung 3.1).

- Verwenden Sie `_nalloc()` zur Allokation der Kindknoten.
- Gehen Sie davon aus, dass `MIN_ORDER` nicht unterschritten wird.
- Achten Sie darauf, dass alle Knoten korrekt verbunden und konfiguriert sind.

2.5 pt

Complete the function `_nsplit()`, which splits the node `n` in the binary tree into two equally-sized child nodes (see Figure 3.1).

- *Use `_nalloc()` for allocating the child nodes.*
- *Assume that the order won't fall below `MIN_ORDER`.*
- *Be sure to correctly connect and configure all nodes.*

```
node* _nalloc(int order); /* does not fail */
void _nsplit(node *n) {
}
```

e) Vervollständigen Sie die Funktion `_nmerge()`, die einen Knoten `n` im Binärbaum mit seinem Buddy (falls vorhanden), zusammenführt, sofern beide Knoten frei sind.

- Verwenden Sie `_nfree()` zur Freigabe von Knoten.
- Achten Sie darauf, dass gemäß dem Buddy-System die Operation rekursiv fortzusetzen ist, falls infolge weitere Zusammenführungen möglich sind.

3.5 pt

Complete the function `_nmerge()`, which merges a node `n` in the binary tree with its buddy (if existing), if both nodes are free.

- *Use `_nfree()` for freeing nodes.*
- *Be sure to continue merging according to the buddy system, if, as a result of the operation, further merges are possible.*

```
void _nfree(node *n);
```

```
void _nmerge(node *n) {
```

```
}
```

f) Vervollständigen Sie die Funktion `_balloc()`, die ausgehend von einem Knoten `n` mittels Tiefensuche im Binärbaum eine Allokation der Länge 2^{order} gemäß des Buddy-Systems durchführt. Im Erfolgsfall wird die Basisadresse `base` des allozierten Knotens zurückgegeben, `NULL` falls kein freies Blatt ausreichender Größe gefunden wird.

- Verwenden Sie `_isLeaf()` und `_nsplit()`.
- Nutzen Sie Rekursion, um die Tiefensuche zu implementieren.
- Es gilt: `order` $\in$ [`MIN_ORDER`; `root.order`].

4.5 pt

Complete the function `_balloc()`, which, starting at node `n`, uses a depth-first search in the binary tree to perform an allocation of length 2^{order} according to the buddy scheme. The function returns the base address `base` of the allocated node, or `NULL`, if no free leaf of sufficient size can be found.

- *Use `_isLeaf()` and `_nsplit()`.*
- *Employ recursion for implementing the depth-first search.*
- *Assume:* `order` $\in$ [`MIN_ORDER`; `root.order`].

```
int _isLeaf(node *n);
void _nsplit(node *n);
void* _balloc(node *n, int order) {
}
```

g) Vervollständigen Sie die Funktion `_bfree()`, die ausgehend von einem Knoten `n` im Binärbaum das passende Blatt zur Adresse `a` findet (d. h. `base == a)` und freigibt.

- Verwenden Sie `_isLeaf()` und `_nmerge()`.
- Nutzen Sie Rekursion, um die Suche im Baum zu implementieren.
- Gehen Sie davon aus, dass `a` eine gültige, allozierte Basisadresse ist und ein passenden Blatt existiert.

2.5 pt

Complete the function `_bfree()`, which, starting at node `n`, searches in the binary tree for the leaf that represents address `a` (i.e., `base == a)` and frees it.

- *Use `_isLeaf()` and `_nmerge()`.*
- *Employ recursion for implementing the search in the tree.*
- *Assume that `a` is a valid and allocated base address and that the corresponding leaf exists.*

```
int _isLeaf(node *n);
void _nmerge(node *n);
void _bfree(node *n, void *a) {




}
```

h) Nennen Sie einen Nachteil, der sich durch eine rekursive Implementierung der Baumfunktionen gegenüber einer iterativen Implementierung ergibt. Begründen Sie Ihre Antwort. **1 pt**

Give a disadvantage of a recursive implementation of the tree functions compared to an iterative implementation. Justify your answer.

Total: 20.0pt

NAME

open − open and possibly create a file or device

SYNOPSIS

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

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

DESCRIPTION

Given a *pathname* for a file, **open**() returns a file descriptor, a small, nonnegative integer for use in subsequent system calls (**read**(2), **write**(2), **lseek**(2), **fcntl**(2), etc.). The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

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_TRUNC**, and **O_TTY_INIT**. The *file status flags* are all of the remaining flags listed below.

RETURN VALUE

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

ERRORS

EACCES
The requested access to the file is not allowed, or search permission is denied for one of the directories in the path prefix of *pathname*, or the file did not exist yet and write access to the parent directory is not allowed.

EFAULT
pathname points outside your accessible address space.

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

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.

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 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.

ERRORS

EBADF
fd isn't a valid open file descriptor.

EINTR
The **close**() call was interrupted by a signal; see **signal**(7).

NOTES

Not checking the return value of **close**() is a common but nevertheless serious programming error. It is quite possible that errors on a previous **write**(2) operation are first reported at the final **close**(). Not checking the return value when closing the file may lead to silent loss of data. This can especially be observed with NFS and with disk quota. Note that the return value should only be used for diagnostics. In particular **close**() should not be retried after an **EINTR** since this may cause a reused descriptor from another thread to be closed.

NAME

mmap − map files or devices into memory

SYNOPSIS

#include <sys/mman.h>

void *mmap(void * *addr***, size_t** *length***, int** *prot***, int** *flags***, int** *fd***, off_t** *offset***);**

DESCRIPTION

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in *addr*. The *length* argument specifies the length of the mapping.

If *addr* is NULL, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new mapping.

The contents of a file mapping are initialized using *length* bytes starting at offset *offset* in the file (or other object) referred to by the file descriptor *fd*. *offset* must be a multiple of the page size as returned by *sysconf(_SC_PAGE_SIZE)*.

RETURN VALUE

On success, **mmap**() returns a pointer to the mapped area. On error, the value **MAP_FAILED** (that is, *(void *) −1*) is returned, and *errno* is set to indicate the cause of the error.

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 file offset, and the file offset is incremented by the number of bytes read. If the 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. See also NOTES.

On error, −1 is returned, and *errno* is set appropriately. In this case, it is left unspecified whether the file position (if any) changes.

ERRORS

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

EFAULT
buf is outside your accessible address space.

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

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

EIO I/O error. This will happen for example when the process is in a background process group, tries to read from its controlling terminal, and either it is ignoring or blocking **SIGTTIN** or its process group is orphaned. It may also occur when there is a low-level I/O error while reading from a disk or tape.

EISDIR
fd refers to a directory.

Other errors may occur, depending on the object connected to *fd*. POSIX allows a **read**() that is interrupted after reading some data to return −1 (with *errno* set to **EINTR**) or to return the number of bytes already read.

NOTES

The types *size_t* and *ssize_t* are, respectively, unsigned and signed integer data types specified by POSIX.1.

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 */
};
```

The *st_dev* field describes the device on which this file resides. (The **major**(3) and **minor**(3) macros may be useful to decompose the device ID in this field.)

The *st_size* field gives the size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symbolic link is the length of the pathname it contains, without a terminating null byte.

The *st_blocks* field indicates the number of blocks allocated to the file, 512-byte units. (This may be smaller than *st_size*/512 when the file has holes.)

The *st_blksize* field gives the "preferred" blocksize for efficient filesystem I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

RETURN VALUE

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

NAME

fork − create a child process

SYNOPSIS

#include <unistd.h>

pid_t fork(void);

DESCRIPTION

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

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

Note the following further points:

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

RETURN VALUE

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

NAME

wait − wait for process to change state

SYNOPSIS

#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int * *wstatus***);**

DESCRIPTION

The **wait**() system call suspends execution of the calling process until one of its children terminates. Performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below).

If a child has already changed state, then the call returns immediately. Otherwise, it blocks until either a child changes state or a signal handler interrupts the call (assuming that system calls are not automatically restarted using the **SA_RESTART** flag of **sigaction**(2)). In the remainder of this page, a child whose state has changed and which has not yet been waited upon by one of these system calls is termed *waitable*.

RETURN VALUE

wait(): on success, returns the process ID of the terminated child; on error, −1 is returned.

The call sets *errno* to an appropriate value in the case of an error.

NOTES

A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child. As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes. If a parent process terminates, then its "zombie" children (if any) are adopted by **init**(1), which automatically performs a wait to remove the zombies.