Karlsruher Institut für Technologie

Betriebssysteme
(Operating Systems)

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

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

Hauptklausur
29.02.2024

- 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 22 Blättern: Einem Deckblatt, 15 Aufgabenblättern mit insgesamt 3 Aufgaben sowie 6 Seiten Man-Pages.

  The examination consists of 22 pages: One cover sheet, 15 sheets containing 3 assignments, and 6 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				

Matrikelnummer/*Matriculation number*

Aufgabe 1: Virtualisierung

Assignment 1: Virtualization

a) Erklären Sie Programm, Prozess und Adressraum. Nennen Sie insgesamt vier Dinge, die nur entweder im Programm oder im Prozess vorkommen, aber nicht in beidem. **3.5 pt**

Explain program, process, and address space. Name a total of four things that are only part of either program or process, but not both.

Explanation:

Only Program: **Only Process:**

b) Was ist ein Thread und woraus besteht er? Nennen Sie zwei Vorteile von Multithreading gegenüber Multiprogramming, und zwei Vorteile von Multiprogramming. **3 pt**

What is a thread and what is it made of? Give two advantages of multithreading over multiprogramming, and two for multiprogramming.

(+) Multithreading: **(+) Multiprogramming:**

c) Der Betriebssystemkernel läuft nicht dauerhaft. Wie kann er trotzdem Hintergrundaufgaben und Aufräumaufgaben abarbeiten? Nennen Sie zwei Möglichkeiten. **1 pt**

The operating system kernel is not running constantly. How can it still work on bookkeeping and background tasks? Name two possibilities.

d) Erklären Sie das Konzept Calling Convention. **1 pt**

Explain the concept of a calling convention.

e) Was ist der Unterschied zwischen Base- und Limitregistern und Segmentation? **1 pt**

What is the difference between base and limit registers and segmentation?

f) Erklären Sie die Funktionsweise des Buddy-Allokators. Nennen Sie einen Nachteil dieser Technik, und eine weitere Allokationsstrategie die diesen Nachteil nicht hat. **3 pt**

Explain how the buddy allocator works. Name one disadvantage of this technique, and another allocation strategy which does not have that disadvantage.

g) Was ist der Vorteil eines read-only Mappings einer Datei, wenn man swappen muss? **1 pt**

What is the advantage of a file mapped read-only when you need to swap?

h) Schreiben Sie den `C`-Aufruf des `mmap()`-Systemaufrufs, um 2 MiB an les- und schreibbarem, privatem Speicher in ihren Prozess zu mappen. **2 pt**

Write down the `C` call to the `mmap()` system call to map 2 MiB *of readable and writable, private memory into your process.*

i) In dieser Aufgabe betrachten Sie ein System mit 32 bit Addressbreite, 4 KiB Seiten und einer zweistufigen Seitentabelle. Beschreiben Sie zunächst, wie die Adressübersetzung funktioniert, und geben Sie die Anzahl der Bits für die erste und zweite Stufe der Seitentabelle sowie den Offset innerhalb der Seite an. **1.5 pt**

In this exercise you will consider a system with 32 bit *addresses,* 4 KiB *pages and a two-level page table. First, describe how the address translation works, and give the number of bits for the first and second level of the page table as well as the offset into the page.*

Explanation:

Bits First Level: **Bits Second Level:** **Bits Offset:**

Zu dem oben beschriebenen System mit 32 bit Adressen und 4 KiB Seitengröße finden Sie folgendes Speicherabbild inlusive CR3-Register mit dem Einstiegspunkt in die Seitentabelle. Füllen Sie anhand dessen die Tabelle aus, und zeichnen Sie Pfeile zwischen den Speicheradressen, anhand derer Sie ihre Speicherübersetzung durchführen. **3 pt**

For the system described above with 32 bit *addresses and* 4 KiB *page size you find the following memory dump including the CR3 register which holds the entry point into the page table. Use it to fill out the table below, and draw arrows along the memory addresses which you use for memory translation.*

CR3=0x80001000

phy. addr:	0x80001040	0x...044	0x...048	0x...04C	0x...050
	0x80002000	0x80004000	0x80006000	0x80008000	0x8000A000

phy. addr:	0x8000A310	0x...314	0x...318	0x...31C	0x...320
	0x800A2000	0x800A4000	0x800A6000	0x800A8000	0x800AA000

Virtual Address	First Level	Second Level	Offset	Physical Address
	0x014	0x0C8	0xFEE	

Platz für Korrekturen:

Space for corrections:

CR3=0x80001000

phy. addr:	0x80001040	0x...044	0x...048	0x...04C	0x...050
	0x80002000	0x80004000	0x80006000	0x80008000	0x8000A000

phy. addr:	0x8000A310	0x...314	0x...318	0x...31C	0x...320
	0x800A2000	0x800A4000	0x800A6000	0x800A8000	0x800AA000

Virtual Address	First Level	Second Level	Offset	Physical Address
	0x014	0x0C8	0xFEE	

Total: 20.0pt

Aufgabe 2: Nebenläufigkeit

Assignment 2: Concurrency

Apple-Betriebssysteme (bspw. macOS) verfügen über ein Parallelisierungsframework namens *Grand Central Dispatch* (GCD). Mittels GCD können Anwendungsentwickler nebenläufige Aufgaben definieren und diese parallel ausführen lassen, ohne sich Gedanken über die Verwaltung von Threads machen zu müssen. Die zentrale Idee von GCD ist das Konzept von *dispatch queues*. Ein:e Entwickler:in fügt Aufgaben zu einer *dispatch queue* hinzu, welche dann von GCD in FIFO-Reihenfolge (first-in, first-out) abgearbeitet werden. Wir betrachten zunächst den einfachsten Fall, sogenannte *serial dispatch queues*. Diese führen jeweils nur eine Aufgabe gleichzeitig aus.

Die Funktion
`void dispatch(queue *q, void (*func)(void *arg), void *arg)`
fügt eine Aufgabe `func` mit dem Argument `arg` in die Warteschlange `q` ein. Diese wird dann asynchron von GCD abgearbeitet. `dispatch()` ist garantiert frei von *race conditions*.

`account_q` sei eine gültige, definierte *serial dispatch queue*. `my_account` sei eine gültige Instanz von `account`. Betrachten Sie folgende Codeabschnitte:

Apple operating systems (e.g., macOS) feature a parallelization framework called Grand Central Dispatch *(GCD). Using GCD, application developers can define concurrent tasks and execute them in parallel, without having to worry about managing threads. The central idea of GCD is the concept of* dispatch queues*. A developer adds tasks to a* dispatch queue*, which then get processed by GCD in FIFO order (first-in, first-out). We first consider the simplest case, the so-called* serial dispatch queues*. This type of queue executes only one task at a time.*

The function
`void dispatch(queue *q, void (*func)(void *arg), void *arg)`
adds a task `func` *with the argument* `arg` *to the queue* `q`*. This task is then processed asynchronously by GCD.* `dispatch()` *is guaranteed to be free of* race conditions*.*

Let `account_q` *be a valid, defined* serial dispatch queue*. Let* `my_account` *be a valid instance of* `account`*. Consider the following code snippets:*

```
1  typedef struct account {
2      int id;
3      int balance;
4  } account;
5  typedef struct task {
6      account *acc;
7      int amount;
8  } task;
9
10 void upd_balance(void *arg) {
11     task *t = (task *)arg;
12     t->acc->balance += t->amount;
13 }
```

Thread 1

```
20 task *t = malloc(sizeof(task));
21 t->account = my_account;
22 t->amount = 100;
23 dispatch(account_q, upd_balance, t);
```

Thread 2

```
30 task *t = malloc(sizeof(task));
31 t->account = my_account;
32 t->amount = -200;
33 dispatch(account_q, upd_balance, t);
```

a) In dem Codeabschnitt ist ein kritischer Abschnitt zu sehen. Geben Sie die Zeilennummer(n) an, in denen sich der kritische Abschnitt befindet. **1 pt**

The code snippet contains a critical section. Give the line number(s) where the critical section is located.

b) Die beiden Threads werden parallel ausgeführt. Beurteilen Sie für jedes der drei Kriterien (ohne Performance) zur Lösung des Problems kritischer Abschnitte, ob es erfüllt ist und begründen Sie Ihre Antworten. **6 pt**

The two threads are executed in parallel. Evaluate for each of the three criteria (ignoring performance) for solving the critical section problem whether it is fulfilled and justify your answers.

Kriterium / *Criterion*: _____ Erfüllt / *Fulfilled*: ☐ Ja ☐ Nein
Begründung / *Justification*:

Kriterium / *Criterion*: _____ Erfüllt / *Fulfilled*: ☐ Ja ☐ Nein
Begründung / *Justification*:

Kriterium / *Criterion*: _____ Erfüllt / *Fulfilled*: ☐ Ja ☐ Nein
Begründung / *Justification*:

c) Angenommen, es gibt insgesamt 1000 Accounts (Instanzen von `account`). Wie viele `upd_balance`-Aufgaben könnten theoretisch insgesamt parallel ausgeführt werden? Wie müssten dazu die *dispatch queues* erstellt werden? **2 pt**

Assume there are a total of 1000 accounts (instances of `account`). How many `upd_balance` tasks could theoretically be executed in parallel? How would the dispatch queues *have to be created for this?*

d) Thread 1 und 2 allokieren jeweils Instanzen von `task`. Geben Sie **präzise** (mit Position und Code des/der Funktionsaufruf(s/e)) an, wie der allokierte Speicher freigegeben werden sollte. **1.5 pt**

Threads 1 and 2 each allocate instances of `task`*. Give a* ***precise*** *(with position and code of the function call(s)) description of how the allocated memory should be deallocated.*

Im Folgenden betrachten wir *concurrent dispatch queues*. Diese führen mehrere Aufgaben gleichzeitig aus, allerdings werden die Aufgaben weiterhin in FIFO-Reihenfolge ausgewählt. **Nehmen Sie für die folgenden Teilaufgaben an, dass System- und Bibliotheksaufrufe nicht fehlschagen und übergebene Funktionsparameter valide sind. Inkludieren Sie notwendige C-Header im gekennzeichneten Bereich.**

Next, we consider concurrent dispatch queues*. These execute multiple tasks simultaneously, but tasks are still selected in FIFO order.* ***For all following subquestions, assume that system and library calls do not fail and all passed function parameters are valid. Include necessary C headers in the marked area.***

e) GCD entscheidet dynamisch, wie viele Threads für die Ausführung einer *concurrent dispatch queue* verwendet werden. Implementieren Sie die Funktion `update_threads()`, die periodisch aufgerufen wird und entscheidet, ob ein neuer Thread gestartet wird. **3.5 pt**

- Starten Sie einen Thread, wenn die Anzahl wartender Aufgaben in der *dispatch queue* mind. doppelt so hoch ist wie beim letzten Threadstart.
- Aktualisieren Sie `last_task_count` und `thread_count` entsprechend.
- Neu gestartete Threads führen `dispatch_thread()` aus und erhalten die *dispatch queue* als Argument.
- Nutzen Sie `ringbuf_count()`, um die Anzahl wartender Aufgaben in der *dispatch queue* zu ermitteln.

GCD dynamically decides how many threads are used for executing a concurrent dispatch queue*. Implement the function* `update_threads()`*, which is called periodically and decides whether a new thread should be started.*

- *Start a new thread if the number of pending tasks in the* dispatch queue *is at least twice as high as when the last thread was started.*
- *Update* `last_task_count` *and* `thread_count` *accordingly.*
- *Newly started threads execute* `dispatch_thread()` *and receive the* dispatch queue *as an argument.*
- *Use* `ringbuf_count()` *to get the number of pending tasks in the* dispatch queue*.*

```c
/* include statements for the required C headers */
#include "ringbuf.h"
```

```c
typedef struct queue {
    ringbuf tasks;
    int last_task_count; // initialized to 0
    int thread_count;
} queue;

int ringbuf_count(ringbuf *rb);
void *dispatch_thread(void *q);
void update_threads(queue *q) {
```

```c
}
```

f) In dieser vereinfachten Version unterstützt GCD drei verschiedene Prioritätsstufen für *dispatch queues*. Diese sollen durch entsprechende Anpassung der Threadprioritäten reflektiert werden. Implementieren Sie die Funktion `update_priority()`. **6 pt**

- Ermitteln Sie den gültigen numerischen Bereich für die Prioritäten.
- `LOW` soll auf die niedrigste numerische Priorität abgebildet werden, `HIGH` auf die höchste und `MEDIUM` auf die Mitte zwischen den numerischen Prioritätsgrenzen.
- Aktualisieren Sie die Priorität des übergebenen Threads.
- Behalten Sie die derzeit für den Thread gesetzte Scheduling-Policy bei.

Hinweis: Überlegen Sie sich, welche der Funktionen aus den angehängten Man-Pages hier nützlich sein könnten.

In this simplified version, GCD supports three different priority levels for dispatch queues*. These should be reflected by adjusting the thread priorities accordingly. Implement the function* `update_priority()`*.*

- *Determine the valid numerical range for priorities.*
- `LOW` *should be mapped to the lowest numerical priority,* `HIGH` *to the highest, and* `MEDIUM` *to the middle between the numerical priority limits.*
- *Update the priority of the passed thread.*
- *Do not change the scheduling policy currently set for the thread.*

***Note**: Consider which of the functions from the attached man pages could be useful for this task.*

```c
typedef enum prio {
    LOW,
    MEDIUM,
    HIGH
} prio;

void update_priority(pthread_t thread, prio p) {
}
```

Total: 20.0pt

Aufgabe 3: Persistenz

Assignment 3: Persistence

Ringpuffer werden häufig zur Kommunikation mit Geräten verwendet. Der folgende Code definiert einen solchen Ringpuffer.

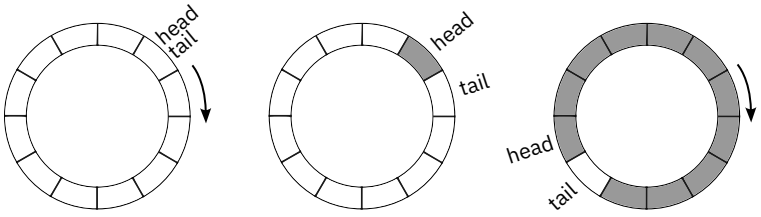
- `head` und `tail` sind Indices in `entries`, die am Ende des Arrays wieder zum Anfang springen.
- Elemente werden am `tail` eingefügt und vom `head` entfernt.
- Der Ringpuffer ist leer, falls `head` und `tail` gleich sind.
- Der Ringpuffer ist voll, falls `tail` genau ein Eintrag hinter `head` ist.

Ring buffers are often used for communication with devices. The following code defines such a ring buffer.

- *`head` and `tail` are indexes in `entries` that wrap around to the beginning at the end of the array.*
- *Entries are inserted at the `tail` and removed from the `head`.*
- *The ring buffer is empty if `head` equals `tail`.*
- *The ring buffer is full if `tail` is exactly one entry behind `head`.*

```
typedef char entry[512];
#define RB_SIZE 64

typedef struct ringbuf {
   entry entries[RB_SIZE];
   int head, tail;
} ringbuf;
```



empty ring buffer | ring buffer with one entry | full ring buffer

a) Geben Sie ein Beispiel für ein Gerät, das üblicherweise über solche Ringpuffer kommuniziert. **0.5 pt**

Give an example of a device that usually communicates with such ring buffers.

b) Nehmen Sie an, das Gerät nutzt DMA, um auf den Ringpuffer zuzugreifen. Wie kann das Betriebssystem signalisieren, dass neue Einträge vorhanden sind? **1 pt**

Assume that the device uses DMA to access the ring buffer. How can the operating system signal that there are new entries?

c) Nennen Sie zwei Möglichkeiten, wie das Betriebssystem erfahren kann, dass ein Gerät neue Einträge in den Ringpuffer geschrieben hat. Erklären Sie, welche Möglichkeit besser ist für ein Gerät mit konstant hohem Durchsatz. **2 pt**

Name two ways for the operating system to learn about new entries from a device in the ring buffer. Explain which way is better for a device with constant high throughput.

d) Implementieren Sie die Funktionen `ringbuf_empty()` und `ringbuf_full()`, die zurückgeben, ob der Ringpuffer leer bzw. voll ist. **1.5 pt**

Implement the functions `ringbuf_empty()` and `ringbuf_full()` that return whether the ring buffer is empty or full.

```
bool ringbuf_empty(ringbuf *rb) {
.....

.....
}
.....

bool ringbuf_full(ringbuf *rb) {
.....

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

e) Implementieren Sie die Funktion `ringbuf_push()`, die einen Eintrag in den Ringpuffer einfügt. Dazu kopiert sie zunächst den Eintrag und passt dann `tail` an. Die Funktion gibt -1 zurück, falls der Eintrag nicht eingefügt werden konnte, sonst 0. **2.5 pt**

Implement the function `ringbuf_push()` that inserts an entry into the ring buffer. It copies the entry first and then adjusts `tail`. The function returns -1 if the entry could not be inserted, otherwise 0.

```
bool ringbuf_full(ringbuf *rb);
int ringbuf_push(ringbuf *rb, entry *e) {
.....

.....

.....

.....

.....

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

f) Implementieren Sie die Funktion `ringbuf_pop()`, die einen Eintrag aus dem Ringpuffer entfernt. Allokieren Sie Speicher für den Eintrag, den Sie schließlich zurückgeben. Aktualisieren Sie nach dem Kopieren des Eintrags `head`. Falls ein Fehler auftritt, geben Sie `NULL` zurück. **3.5 pt**

Implement the function `ringbuf_pop()` that removes an entry from the ring buffer. Allocate memory for the entry that will be returned. After copying the entry, update `head`. Return `NULL` if an error occurs.

```
bool ringbuf_empty(ringbuf *rb);
entry *ringbuf_pop(ringbuf *rb) {
.....

.....

.....

.....

.....

.....

.....

.....

.....

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

g) Nennen Sie drei aus der Vorlesung bekannte Verfahren zur Dateiallokation. Ordnen Sie die Verfahren in aufsteigender Reihenfolge bzgl. der Performance von wahlfreien Zugriffen. **1.5 pt**

Name three strategies for file allocation introduced in the lecture. Order them by increasing performance of random accesses.

h) Gegeben sei das folgende Programm. Gehen Sie davon aus, dass alle Systemaufrufe erfolgreich sind.

Have a look at the following program. Assume that all system calls are successful.

```
int main(int argc, char **argv) {
   int i, n, fd;
   char buf[100];
   for (i = 1; i < argc; i++) {
      fd = open(argv[i], O_RDONLY);
      while ((n = read(fd, buf, sizeof(buf))))
         write(STDOUT_FILENO, buf, n);
      close(fd);
   }
   return 0;
}
```

Welche Funktion hat das Programm? **1 pt**

What function does the program perform?

Das Programm wird mit einer 150 Byte großen Datei als Argument aufgerufen. Wie viele `read`-Systemaufrufe führt das Programm aus? Füllen Sie in der Tabelle unten den Rückgabewert sowie den Offset in der Datei nach jedem `read`-Aufruf aus. Gehen Sie davon aus, dass jeder Aufruf so viel wie möglich liest. **2 pt**

The program is called with a file of size 150 bytes as argument. How many `read` system calls does the program perform? In the table below, fill in the return value and the offset in the file after each call to `read`. Assume that every call reads as much as possible.

Anzahl `read`-Aufrufe:
Number of `read` calls:

syscall	return value	offset
`open`	3	0
`read`		

Wo wird der Offset durch das Betriebssystem gespeichert? **0.5 pt**

Where does the operating system store the offset?

i) Die Berechtigung `x` hat unterschiedliche Bedeutungen abhängig von dem in der Inode kodierten Dateityp. Nennen Sie zwei Dateitypen und erklären Sie jeweils die zugehörige Bedeutung von `x`. **2 pt**

The access right `x` has different meanings depending on the file type coded in the inode. Name two file types and explain the respective meaning of `x`.

j) Gegeben sei ein Dateisystem, das Dateien mittels Inodes umsetzt. Nehmen Sie eine Blockgröße von 4 KiB und 8 Byte große Zeiger auf Blöcke an. Eine Inode beinhaltet 10 Zeiger auf direkte Blöcke, und je einen Zeiger zu einem einfach, doppelt, sowie dreifach indirekten Block. Berechnen Sie die Menge an Dateispeicherplatz, der mit den jeweiligen Komponenten adressiert werden kann. Vereinfachen Sie Ihre Ergebnisse zu den gegebenen Einheiten. **2 pt**

Consider a file system that uses inodes to represent files. Assume that disk blocks are 4 KiB in size and a pointer to a disk block is 8 bytes long. An inode contains 10 pointers to direct blocks, and one pointer to a single, double, and triple indirect block, respectively. Calculate the amount of file data that is addressable with each of these components. Simplify your results to the given units.

Component	Amount of file data
10 direct	KiB
1 single indirect	MiB
1 double indirect	GiB
1 triple indirect	GiB

Total: 20.0pt

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 (which must be greater than 0).

If *addr* is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping. If *addr* is not NULL, then the kernel takes it as a hint about where to place the mapping; If another mapping already exists there, the kernel picks a new address that may or may not depend on the hint. The address of the new mapping is returned as the result of the call.

The contents of a file mapping (as opposed to an anonymous mapping; see **MAP_ANONYMOUS** below), 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)*.

The *prot* argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file). It is either **PROT_NONE** or the bitwise OR of one or more of the following flags:

PROT_EXEC Pages may be executed.

PROT_READ Pages may be read.

PROT_WRITE Pages may be written.

PROT_NONE Pages may not be accessed.

The flags argument

The *flags* argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file. This behavior is determined by including exactly one of the following values in *flags*:

MAP_SHARED
Share this mapping. Updates to the mapping are visible to other processes mapping the same region, and (in the case of file-backed mappings) are carried through to the underlying file. (To precisely control when updates are carried through to the underlying file requires the use of **msync**(2).)

MAP_PRIVATE
Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the **mmap**() call are visible in the mapped region.

In addition, zero or more of the following values can be ORed in *flags*:

MAP_ANONYMOUS
The mapping is not backed by any file; its contents are initialized to zero. The *fd* argument is ignored; however, some implementations require *fd* to be −1 if **MAP_ANONYMOUS** (or **MAP_ANON**) is specified, and portable applications should ensure this. The *offset* argument should be zero. Support for **MAP_ANONYMOUS** in conjunction with **MAP_SHARED** was added in Linux 2.4.

MAP_FIXED
Don't interpret *addr* as a hint: place the mapping at exactly that address. *addr* must be suitably aligned: for most architectures a multiple of the page size is sufficient; however, some architectures may impose additional restrictions. If the memory region specified by *addr* and *length* overlaps pages of any existing mapping(s), then the overlapped part of the existing mapping(s) will be discarded. If the specified address cannot be used, **mmap**() will fail.

Software that aspires to be portable should use the **MAP_FIXED** flag with care, keeping in mind that the exact layout of a process's memory mappings is allowed to change significantly between Linux versions, C library versions, and operating system releases.

MAP_GROWSDOWN
This flag is used for stacks. It indicates to the kernel virtual memory system that the mapping should extend downward in memory. The return address is one page lower than the memory area that is actually created in the process's virtual address space. Touching an address in the "guard" page below the mapping will cause the mapping to grow by a page. This growth can be repeated until the mapping grows to within a page of the high end of the next lower mapping, at which point touching the "guard" page will result in a **SIGSEGV** signal.

MAP_NORESERVE
Do not reserve swap space for this mapping. When swap space is reserved, one has the guarantee that it is possible to modify the mapping. When swap space is not reserved one might get **SIGSEGV** upon a write if no physical memory is available. See also the discussion of the file */proc/sys/vm/overcommit_memory* in **proc**(5). Before Linux 2.6, this flag had effect only for private writable mappings.

MAP_POPULATE (since Linux 2.5.46)
Populate (prefault) page tables for a mapping. For a file mapping, this causes read-ahead on the file. This will help to reduce blocking on page faults later. The **mmap**() call doesn't fail if the mapping cannot be populated. Support for **MAP_POPULATE** in conjunction with private mappings was added in Linux 2.6.23.

MAP_STACK (since Linux 2.6.27)
Allocate the mapping at an address suitable for a process or thread stack.

MAP_UNINITIALIZED (since Linux 2.6.33)
Don't clear anonymous pages. This flag is intended to improve performance on embedded devices. This flag is honored only if the kernel was configured with the **CONFIG_MMAP_ALLOW_UNINITIALIZED** option. Because of the security implications, that option is normally enabled only on embedded devices (i.e., devices where one has complete control of the contents of user memory).

Of the above flags, only **MAP_FIXED** is specified in POSIX.1-2001 and POSIX.1-2008. However, most systems also support **MAP_ANONYMOUS** (or its synonym **MAP_ANON**).

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 error.

ERRORS

EACCES
A file descriptor refers to a non-regular file. Or a file mapping was requested, but *fd* is not open for reading.

EBADF
fd is not a valid file descriptor (and **MAP_ANONYMOUS** was not set).

EINVAL
We don't like *addr*, *length*, or *offset* (e.g., they are too large, or not aligned on a page boundary).

ENOMEM
No memory is available.

Use of a mapped region can result in these signals:

SIGSEGV
Attempted write into a region mapped as read-only.

SIGBUS
Attempted access to a page of the buffer that lies beyond the end of the mapped file. For an explanation of the treatment of the bytes in the page that corresponds to the end of a mapped file that is not a multiple of the page size, see NOTES.

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

sched_get_priority_max, sched_get_priority_min − get static priority range

SYNOPSIS

```
#include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

DESCRIPTION

sched_get_priority_max() returns the maximum priority value that can be used with the scheduling algorithm identified by *policy*. **sched_get_priority_min**() returns the minimum priority value that can be used with the scheduling algorithm identified by *policy*. Supported *policy* values are **SCHED_FIFO**, **SCHED_RR**, **SCHED_OTHER**, **SCHED_BATCH**, **SCHED_IDLE**, and **SCHED_DEADLINE**. Further details about these policies can be found in **sched**(7).

Processes with numerically higher priority values are scheduled before processes with numerically lower priority values. Thus, the value returned by **sched_get_priority_max**() will be greater than the value returned by **sched_get_priority_min**().

The range of scheduling priorities may vary on other POSIX systems, thus it is a good idea for portable applications to use a virtual priority range and map it to the interval given by **sched_get_priority_max**() and **sched_get_priority_min**()

POSIX systems on which **sched_get_priority_max**() and **sched_get_priority_min**() are available define **_POSIX_PRIORITY_SCHEDULING** in <*unistd.h*>.

RETURN VALUE

On success, **sched_get_priority_max**() and **sched_get_priority_min**() return the maximum/minimum priority value for the named scheduling policy. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
: The argument *policy* does not identify a defined scheduling policy.

NAME

get_nprocs − get number of processors

SYNOPSIS

```
#include <sys/sysinfo.h>

int get_nprocs(void);
```

DESCRIPTION

The function **get_nprocs**() returns the number of processors currently available in the system.

RETURN VALUE

As given in DESCRIPTION.

NAME

pthread_setschedparam, pthread_getschedparam − set/get scheduling policy and parameters of a thread

SYNOPSIS

```
#include <pthread.h>

int pthread_setschedparam(pthread_t thread, int policy,
                          const struct sched_param *param);
int pthread_getschedparam(pthread_t thread, int * policy,
                          struct sched_param * param);
```

DESCRIPTION

The **pthread_setschedparam**() function sets the scheduling policy and parameters of the thread *thread*.

policy specifies the new scheduling policy for *thread*. The supported values for *policy*, and their semantics, are described in **sched**(7).

The structure pointed to by *param* specifies the new scheduling parameters for *thread*. Scheduling parameters are maintained in the following structure:

```
struct sched_param {
    int sched_priority;     /* Scheduling priority */
};
```

As can be seen, only one scheduling parameter is supported. For details of the permitted ranges for scheduling priorities in each scheduling policy, see **sched**(7).

The **pthread_getschedparam**() function returns the scheduling policy and parameters of the thread *thread*, in the buffers pointed to by *policy* and *param*, respectively. The returned priority value is that set by the most recent **pthread_setschedparam**(), **pthread_setschedprio**(3), or **pthread_create**(3) call that affected *thread*. The returned priority does not reflect any temporary priority adjustments as a result of calls to any priority inheritance or priority ceiling functions (see, for example, **pthread_mutexattr_setprioceiling**(3) and **pthread_mutexattr_setprotocol**(3)).

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number. If **pthread_setschedparam**() fails, the scheduling policy and parameters of *thread* are not changed.

ERRORS

Both of these functions can fail with the following error:

ESRCH
: No thread with the ID *thread* could be found.

pthread_setschedparam() may additionally fail with the following errors:

EINVAL
: *policy* is not a recognized policy, or *param* does not make sense for the *policy*.

EPERM
: The caller does not have appropriate privileges to set the specified scheduling policy and parameters.

POSIX.1 also documents an **ENOTSUP** ("attempt was made to set the policy or scheduling parameters to an unsupported value") error for **pthread_setschedparam**().

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

memset – fill memory with a constant byte

SYNOPSIS

#include <string.h>

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

DESCRIPTION

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

RETURN VALUE

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

NAME

memcpy – copy memory area

SYNOPSIS

#include <string.h>

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

DESCRIPTION

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

RETURN VALUE

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

NOTES

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

NAME

strlen – calculate the length of a string

SYNOPSIS

#include <string.h>

size_t strlen(const char ****s*);**

DESCRIPTION

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

RETURN VALUE

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

NAME

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

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.