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
15.03.2016

- 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 20 Blättern: 1 Deckblatt, 16 Aufgabenblättern mit insgesamt 3 Aufgaben und 3 Blättern Man-Pages.
  The examination consists of 20 pages: 1 cover sheet, 16 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	18	22	20	60
Erreichte Punkte				
Note				

Aufgabe 1: C Grundlagen

Assignment 1: C Basics

a) Wie werden die folgenden Operationen in der Programmiersprache C geschrieben? Nehmen Sie an, dass die folgenden Definitionen gelten: `int a, b, *c;` **2 pt**

How are the following operations expressed in the C programming language? Assume the following definitions: `int a, b, *c;`

bitwise not of `a`	
bitwise exclusive or of `a` and `b`	
address of value pointed to by `c`	
value pointed to by `c`	

b) Beschreiben Sie, wie sich die Länge einer Zeichenkette (`char*`) ohne die Verwendung von Hilfsfunktionen wie `strlen()` bestimmen lässt. **1 pt**

Explain how the length of a string (`char*`) can be determined without using any helper methods such as `strlen()`.

c) Betrachten Sie das C Programm aus Listing 1. In welchen Speichersegmenten liegen die Daten der jeweiligen Symbole bei der Ausführung? **2 pt**

Consider the C program in Listing 1. In which memory segments is the data of the following symbols stored during execution?

```
const char *str = "Hello World!";
int a = 0;

int* add(int p)
{
    int b = p + a;
    return &b;
}
```

Listing 1: Basic C Program

	Text	Stack	Heap	BSS	Data	RO-Data
`str`	☐	☐	☐	☐	☐	☐
`a`	☐	☐	☐	☐	☐	☐
`p`	☐	☐	☐	☐	☐	☐
`add`	☐	☐	☐	☐	☐	☐

d) Wieso kann ein Aufrufer der Funktion `add()` aus Listing 1 den Rückgabewert nicht gefahrlos dereferenzieren? **1 pt**

Why is it not safe for a caller of the function `add()` from Listing 1 to dereference the return value?

e) Erläutern Sie den Unterschied zwischen `unsigned int` und `uint32_t`. **1 pt**

Explain the difference between `unsigned int` and `uint32_t`.

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

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

```
uint32_t addr = (uint32_t)((uint64_t*)0xffffffff + 0x01);
```

g) Erklären Sie, warum es in C wichtig ist, lokale Variablen vor der ersten Verwendung explizit zu initialisieren. **2 pt**

Explain why it is important in C to explicitly initialize local variables before their first use.

h) Erklären Sie, warum es auf Systemen mit wenig RAM bei der Verwendung von Rekursion zu Problemen kommen kann. **1 pt**

Explain why the use of recursion can lead to problems on systems with little RAM.

i) Erklären Sie, was unter einem Void-Pointer (`void*`) zu verstehen ist. **1 pt**

Explain the meaning of a void-pointer (`void*`).

j) C unterscheidet beim Datentyp `char` zwischen *signed* und *unsigned*. Ist diese Unterscheidung sinnvoll? Begründen Sie Ihre Antwort. **1 pt**

For the data type `char`, C distinguishes between signed *and* unsigned*. Does this distinction make sense? Explain your answer.*

k) Welche der folgenden Aussagen sind richtig?
(falsches Kreuz: -1P, kein Kreuz: 0P, korrektes Kreuz: 1P) **4 pt**

Which of the following statements are correct?
(incorrectly marked: 1P, not marked: 0P, correctly marked: 1P)

korrekt/ *correct*	inkorrekt/ *incorrect*	
☐	☐	`(a >> 2) == (a / 2)`
☐	☐	`sizeof(int) == sizeof(unsigned int)`
☐	☐	Ein C-`struct` kann ein Array als Feld enthalten. *A C `struct` can contain an array as a member.*
☐	☐	`free(NULL)` führt zum Absturz des Programms. *`free(NULL)` leads to a program crash.*

Total: 18.0pt

Aufgabe 2: Druckerserver

Assignment 2: Printer Spooler

Schreiben Sie einen Druckerserver (*spooler*) und einen Client, der Aufträge an den Server schickt.

Client und Server nutzen POSIX Message Queues und temporäre Dateien, um Druckaufträge und die zu druckenden Daten auszutauschen.

Beim Starten initialisiert der Druckerserver zunächst eine POSIX Message Queue und wartet dann auf eingehende Nachrichten. Um einen Druckauftrag abzusetzen, speichert der Client die zu druckenden Daten in eine temporäre Datei und sendet dann den Namen der temporären Datei über die Message Queue an den Druckerserver. Sobald der Druckerserver eine Nachricht empfängt, liest er den Druckauftrag aus der temporären Datei und schreibt die Daten in die Character-Device Datei `/dev/lp0` des Druckers.

Binden Sie in allen Teilaufgaben die notwendigen C-Header ein und lassen Sie niemals vom Betriebssystem angeforderte Objekte (z.B. Dateideskriptoren) ungenutzt zurück (*leak*).

Write a printer spooler and a client for sending jobs to the spooler.

The client and spooler make use of POSIX message queues and temporary files to exchange print requests and the print jobs' data, respectively.

When starting up, the printer spooler first initializes a POSIX message queue and then waits for incoming messages. For launching a print job, a client stores the job's data in a temporary file and then sends the file name of the temporary file for printing to the spooler via the message queue. Once the spooler receives a request, it reads the print job from the temporary file and writes it to the printer's character device file `/dev/lp0`*.*

In all assignments, include necessary C headers and never leak allocated operating system objects (e.g., file descriptors).

a) Wie trägt ein *Spooler* (so wie der Druckerserver in dieser Aufgabe) zur Vermeidung von Deadlocks *(deadlock prevention)* bei? **2 pt**

How does a spooler *(e.g., for a printer, as in this assignment) contribute to deadlock prevention?*

b) Vervollständigen Sie die Funktion `init_message_queue()` für den Server, die eine POSIX Message Queue mit dem Namen `\exam-print-spooler` anlegt und öffnet.

- Die maximale Nachrichtengröße der Message Queue soll `MAX_PATH` betragen.
- Die Puffergröße der Message Queue soll acht Nachrichten betragen.
- Die Funktion gibt ein Handle auf die neu erzeugte Message Queue zurück.
- Nehmen Sie an, dass keine Fehler auftreten.
- Beachten Sie, dass der Druckerserver und die Clients unter verschiedenen Benutzer IDs laufen können.

4 pt

Complete the function `init_message_queue()` for the printer spooler, which creates and opens a POSIX message queue called `\exam-print-spooler`.

- *The maximum size of messages to be accepted into the queue should be `MAX_PATH`.*
- *The queue should store up to eight messages in its buffer.*
- *The function returns a handle to the newly created message queue.*
- *Assume that no errors occur.*
- *Keep in mind that printer spooler and clients might run with different user ids.*

```
#define MAX_PATH 256
mqd_t init_message_queue(void) {
}
```

c) Wie können Sie beeinflussen, ob sich die Sende- und Empfangsoperationen einer POSIX Message Queue synchron oder asynchron verhalten?

Kann sich dieses Verhalten zwischen Sender und Empfänger einer Nachricht unterscheiden? Begründen Sie Ihre Antwort kurz. **2 pt**

How do you control if send and receive operations on a POSIX message queue should be synchronous or asynchronous?

Can this behavior be different between sender and receiver of a message? Briefly explain your answer.

d) Sollte der Druckerserver im gegebenen Szenario synchrone oder asynchrone Empfangsoperationen nutzen? Begründen Sie Ihre Antwort. **1 pt**

In the given scenario, should the printer spooler use synchronous or asynchronous receive operations? Explain your answer.

e) Vervollständigen Sie die Funktion `send_print_job()`, mit der ein Client einen Druckauftrag an den Server schicken an.

- Die Funktion `store_data_to_tmp()` speichert die zu druckenden Daten in einer temporären Datei und liefert den Pfad zu dieser Datei zurück. Der Speicher des Pfads muss nicht freigegeben werden.
- Senden Sie den Pfadnamen der temporären Datei (sonst nichts) als Druckauftrag über die Message Queue `\exam-print-spooler` an den Druckerserver.
- Die Priorität der Nachricht in der Message Queue soll `0` betragen.
- Die Funktion gibt bei Erfolg `0` zurück, `-1` andernfalls.

3 pt

Complete the function `send_print_job()`, which allows a client to send a print job to the spooler.

- *The function `store_data_to_tmp()` stores the print job's data in a temporary file and returns the path name of that file. The path name does not need to be freed.*
- *Send the path name of the temporary file (nothing else) as a request to the printer spooler via the message queue `\exam-print-spooler`.*
- *The priority of the message in the queue should be `0`.*
- *The function returns `0` on success, `-1` otherwise.*

```
char* store_data_to_tmp(char *jobname, void *jobdata, size_t joblength);
#define MAX_PATH 256
int send_print_job(char *jobname, void *jobdata, size_t joblength) {
    char *path = store_data_to_tmp(jobname, jobdata, joblength);
}
```

f) Vervollständigen Sie die `main()`-Funktion des Druckerservers, die zunächst die Initialisierungsfunktion aus Teilaufgabe b) aufruft und dann in einer Endlosschleife eingehende Druckaufträge verarbeitet.

- Die Funktion empfängt Druckaufträge über die Message Queue und sendet sie über die Funktion `print_job()` an den Drucker.
- Nehmen Sie an, dass keine Fehler auftreten können.

3 pt

Write the spooler's `main()`-function, which first calls the initialization function from assignment b) and then enters an infinite loop servicing client requests.

- *The function receives print jobs from the message queue and sends them to the printer by calling `print_job()`.*
- *Assume that no errors occur.*

```
int print_job(char *jobpath);
int main(int argc, char *argv[]) {
}
```

g) Vervollständigen Sie die Funktion `print_job()`, die einen Druckauftrag aus einer temporären Datei liest und ihn an den Drucker schickt, indem sie die Daten in die Character-Device Datei `/dev/lp0` schreibt.

- Behandeln Sie den Fall, dass die temporäre Datei des Druckauftrags oder die Gerätedatei des Druckers nicht geöffnet werden können.
- Nehmen Sie an, dass keine Fehler auftreten können, wenn Sie Dateien lesen oder schreiben.
- Beachten Sie, dass die Systemaufrufe `read()` und `write()` zurückkehren können, obwohl sie weniger Bytes gelesen bzw. geschrieben haben, als angefordert.
- Die Funktion gibt bei Erfolg `0` zurück, `-1` andernfalls.

7 pt

Complete the function `print_job()`, which reads a print job from a temporary file and sends it to the printer by writing it to the printer's character device file `/dev/lp0`.

- *Handle the case that the temporary file or the device file cannot be opened.*
- *Assume that no errors can occur when reading or writing to files.*
- *Keep in mind that the `read()` and `write()` system calls may return with fewer bytes read/written than requested.*
- *The function returns `0` on success, `-1` otherwise.*

```
int print_job(char *jobpath) {
```

}

Total: 22.0pt

Aufgabe 3: Speicherverwaltung

Assignment 3: Memory Management

Betrachten Sie ein System mit 31 bit virtuellen und 45 bit physischen Adressen. Die Seitengröße beträgt 8 KiB. Zur Übersetzung kommt ein durch Software gefüllter TLB zum Einsatz.

Ein virtueller Adressraum (VAS) wird im Betriebssystem mit der `vas_t`-Struktur beschrieben. Die `vma_t`-Struktur stellt einen gültigen Speicherbereich (VMA) innerhalb eines Adressraums dar. Alle VMAs eines Adressraums sind in einer unsortierten, einfach-verketteten Liste abgelegt, wobei ein `vma_t`-Zeiger von `NULL` das Listenende bzw. eine leere Liste markiert.

Nutzen Sie zur Lösung der Teilaufgaben die folgenden Typen, Konstanten und Funktionen:

Consider a system with 31 bit virtual and 45 bit physical addresses. The page size is 8 KiB. The system uses a software-filled TLB for address translation.

The operating system describes a virtual address space (VAS) with the `vas_t` structure. The `vma_t` structure represents a valid virtual memory area (VMA) within an address space. All VMAs of an address space are stored in an unsorted, singly-linked list, where a `vma_t` pointer of `NULL` indicates the list's end, or an empty list respectively.

For solving the questions, use the following types, constants, and functions:

```
typedef struct _vma_t {
    struct _vma_t *next; // Pointer to next VMA in list
    uint32_t start;      // Number of first virtual page in VMA
    uint32_t end;        // Number of last virtual page in VMA
} vma_t;

typedef struct {
    vma_t *vmas;         // Unsorted list of VMAs in address space
    page_dir_t pdir;     // Page directory
} vas_t;

// Constant to return or set for invalid page frame numbers (PFNs)
#define INVALID_PFN ((uint32_t)(-1))

// Allocates a number of bytes from kernel virtual memory
// (not zero-initialized). Returns a pointer to the newly allocated
// memory on success, NULL otherwise.
void* kmalloc(size_t bytes);

// Allocates a free physical page frame (zero-initialized).
// Returns the page frame number (PFN) on success,
// INVALID_PFN otherwise.
uint32_t allocFrame(void);

// Frees a physical page frame previously allocated by allocFrame().
// Calling freeFrame() with INVALID_PFN is undefined behavior.
void freeFrame(uint32_t pfn);

// Flushes the software-filled TLB, thereby invalidating all TLB entries.
void flushTlb(void);
```

a) Definieren Sie die folgenden Strukturen für eine zweistufige Seitentabelle zur Übersetzung von virtuellen in physische Adressen in dem gegebenen System. Beginnen Sie mit der Berechnung der Anzahl von Einträgen pro Tabellenstufe.

- Tabellen der zweiten Ebene sollen nur bei Bedarf alloziert werden.

4 pt

Define the following structures for a two-level page table for the translation of virtual to physical addresses in the given system. Start with calculating the number of entries per page table level.

- *Second-level page tables should be allocated on demand only.*

Entries per page table level:

```
typedef struct {
.....
.....
.....
} pte_t; // 2nd-level page table entry (PTE)
typedef struct {
.....
.....
.....
} page_table_t; // 2nd-level page table
typedef struct {
.....
.....
.....
} page_dir_t; // 1st-level page table (page directory)
```

b) Vervollständigen Sie die Funktion `allocVas()`, die einen neuen Adressraum (`vas_t`) alloziert, initialisiert und zurückgibt.

- Der Adressraum enhält keine gültigen Speicherbereiche.
- Der durch die Seitentabellenhierarchie belegte Platz soll möglichst klein sein.
- Die Funktion gibt im Fehlerfall `NULL` zurück.

2 pt

Complete the function `allocVas()`, which allocates, initializes, and returns a new address space (`vas_t`).

- *The address space does not contain any valid virtual memory areas.*
- *The memory required by the page table hierarchy should be minimal.*
- *The function returns `NULL` on error.*

```
vas_t* allocVas(void) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
```

c) Vervollständigen Sie die Funktion `addVma()`, die den gegebenen virtuellen Speicherbereich `vma` in die Liste der gültigen Speicherbereiche des Adressraums `vas` einfügt.

- Die Operation schlägt fehl, wenn der gegebene Speicherbereich sich mit einer bereits eingetragenen Region überschneidet.
- Die Funktion gibt im Fehlerfall `-1` zurück, ansonsten `0`.

3 pt

Complete the function `addVma()`, which adds the specified virtual memory area (`vma_t`) to the list of valid VMAs in the address space `vas`.

- *The operation fails if the supplied area overlaps with an existing area.*
- *The function returns `-1` on error, `0` otherwise.*

```
int addVma(vas_t *vas, vma_t *vma) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
```

```
}
```

d) Vervollständigen Sie die Funktion `getPte()`, die einen Pointer auf den Seitentabelleneintrag (PTE) zu der virtuellen Adresse `vaddr` liefert.

- Die Funktion liefert `NULL`, wenn keine Seitentabelle für die Adresse alloziert ist.
- Gehen Sie davon aus, dass Bit 32 in `vaddr` stets 0 ist.

3 pt

Complete the function `getPte()`, which returns a pointer to the page table entry (PTE) for the virtual address `vaddr`.

- *The function returns `NULL` if no page table exists for the virtual address.*
- *Assume that bit 32 of `vaddr` is always 0.*

```
pte_t* getPte(vas_t *vas, uint32_t vaddr) {

}
```

e) Vervollständigen Sie die Funktion `invalidateVma()`, die alle Abbildungen von virtuellen auf physische Seiten des Speicherbereichs `vma` aufhebt und betroffene physische Seiten freigibt.

- Sie können die Funktion `getPte()` verwenden.
- Dynamisch allozierte Seitentabellen müssen nicht freigegeben werden.
- Beachten Sie, dass nicht alle Seitentabellen alloziert sein und nicht alle PTEs gültige Abbildungen auf physische Seiten enthalten müssen.

3 pt

Complete the function `invalidateVma()`, which invalidates all mappings of virtual to physical pages for the memory area `vma`, and frees the physical pages.

- *You can use the function `getPte()`.*
- *Dynamically allocated page tables do not need to be released.*
- *Keep in mind that not all page tables must be allocated and that not all PTEs must contain valid mappings to physical pages.*

```
void invalidateVma(vas_t *vas, vma_t *vma) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
.....
```

f) Vervollständigen Sie die Funktion `pagefault()`, die bei einem Seitenfehler aufgerufen wird und für die zugegriffene virtuelle Seite eine gültige Abbildung auf eine leere physische Seite herstellt.

- Die Adresse der virtuellen Seite ist in einen Index `idx1` in die erste Ebene und einen Index `idx2` in die zweite Ebene der Seitentabelle zerlegt worden.
- Der Zugriff auf eine Adresse außerhalb eines gültigen Speicherbereichs (VMA) ist ein Fehlerfall.
- Die Funktion liefert die Nummer der physischen Seite (PFN) bei Erfolg und `INVALID_PFN` im Fehlerfall.

5 pt

Complete the function `pagefault()`*, which is called on a page fault and establishes a mapping from the accessed virtual page to an empty physical page.*

- *The address of the virtual page has been split into an index* `idx1` *into the first level and an index* `idx2` *into the second level of the page table.*
- *Accessing an address that is not within a valid memory area (VMA) is an error.*
- *The function returns the number of the physical page (PFN) on success,* `INVALID_PFN` *on error.*

```
uint32_t pagefault(vas_t *vas, uint32_t idx1, uint32_t idx2) {
}
```

Total: 20.0pt

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).

EIO An I/O error occurred.

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.

NAME

mq_close − close a message queue descriptor

SYNOPSIS

#include <mqueue.h>

int mq_close(mqd_t *mqdes***);**

Link with *−lrt*.

DESCRIPTION

mq_close() closes the message queue descriptor *mqdes*.

If the calling process has attached a notification request to this message queue via *mqdes*, then this request is removed, and another process can now attach a notification request.

RETURN VALUE

On success **mq_close**() returns 0; on error, −1 is returned, with *errno* set to indicate the error.

ERRORS

EBADF
The descriptor specified in *mqdes* is invalid.

NAME

mq_getattr, mq_setattr − get/set message queue attributes

SYNOPSIS

#include <mqueue.h>

int mq_getattr(mqd_t *mqdes***, struct mq_attr *****attr***);**

int mq_setattr(mqd_t *mqdes***, const struct mq_attr *****newattr***,**
struct mq_attr *****oldattr*);**

DESCRIPTION

mq_getattr() and **mq_setattr**() respectively retrieve and modify attributes of the message queue referred to by the descriptor *mqdes*.

mq_getattr() returns an *mq_attr* structure in the buffer pointed by *attr*. This structure is defined as:

```
struct mq_attr {
   long mq_flags;      /* Flags: 0 or O_NONBLOCK */
   long mq_maxmsg;     /* Max. # of messages on queue */
   long mq_msgsize;    /* Max. message size (bytes) */
   long mq_curmsgs;    /* # of messages currently in queue */
};
```

The *mq_flags* field contains flags associated with the open message queue description. This field is initialized when the queue is created by **mq_open**(3). The only flag that can appear in this field is **O_NONBLOCK**.

The *mq_maxmsg* and *mq_msgsize* fields are set when the message queue is created by **mq_open**(3). The *mq_maxmsg* field is an upper limit on the number of messages that may be placed on the queue using **mq_send**(3). The *mq_msgsize* field is an upper limit on the size of messages that may be placed on the queue. Both of these fields must have a value greater than zero.

The *mq_curmsgs* field returns the number of messages currently held in the queue.

mq_setattr() sets message queue attributes using information supplied in the *mq_attr* structure pointed to by *newattr*. The only attribute that can be modified is the setting of the **O_NONBLOCK** flag in *mq_flags*. The other fields in *newattr* are ignored. If the *oldattr* field is not NULL, then the buffer that it points to is used to return an *mq_attr* structure that contains the same information that is returned by **mq_getattr**().

RETURN VALUE

On success **mq_getattr**() and **mq_setattr**() return 0; on error, −1 is returned, with *errno* set to indicate the error.

NAME

mq_open − **open a message queue**

SYNOPSIS

```
#include <fcntl.h>          /* For O_* constants */"
#include <sys/stat.h>       /* For mode constants */"
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
        struct mq_attr *attr);
```

DESCRIPTION

mq_open() creates a new POSIX message queue or opens an existing queue. The queue is identified by *name*.

The *oflag* argument specifies flags that control the operation of the call. (Definitions of the flags values can be obtained by including *<fcntl.h>*.) Exactly one of the following must be specified in *oflag*:

O_RDONLY
Open the queue to receive messages only.

O_WRONLY
Open the queue to send messages only.

O_RDWR
Open the queue to both send and receive messages.

Zero or more of the following flags can additionally be *OR*ed in *oflag*:

O_NONBLOCK
Open the queue in nonblocking mode. In circumstances where **mq_receive**(3) and **mq_send**(3) would normally block, these functions instead fail with the error **EAGAIN**.

O_CREAT
Create the message queue if it does not exist. The owner (user ID) of the message queue is set to the effective user ID of the calling process. The group ownership (group ID) is set to the effective group ID of the calling process.

If **O_CREAT** is specified in *oflag*, then two additional arguments must be supplied. The *mode* argument specifies the permissions to be placed on the new queue, as for **open**(2). (Symbolic definitions for the permissions bits can be obtained by including *<sys/stat.h>*.)

The *attr* argument specifies attributes for the queue. See **mq_getattr**(3) for details. If *attr* is NULL, then the queue is created with implementation-defined default attributes.

RETURN VALUE

On success, **mq_open**() returns a message queue descriptor for use by other message queue functions. On error, **mq_open**() returns *(mqd_t) −1*, with *errno* set to indicate the error.

NAME

mq_send − **send a message to a message queue**

SYNOPSIS

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr,
        size_t msg_len, unsigned int msg_prio);
```

DESCRIPTION

mq_send() adds the message pointed to by *msg_ptr* to the message queue referred to by the descriptor *mqdes*. The *msg_len* argument specifies the length of the message pointed to by *msg_ptr*; this length must be less than or equal to the queue's *mq_msgsize* attribute. Zero-length messages are allowed.

The *msg_prio* argument is a nonnegative integer that specifies the priority of this message. Messages are placed on the queue in decreasing order of priority, with newer messages of the same priority being placed after older messages with the same priority.

If the message queue is already full (i.e., the number of messages on the queue equals the queue's *mq_maxmsg* attribute), then, by default, **mq_send**() blocks until sufficient space becomes available to allow the message to be queued, or until the call is interrupted by a signal handler. If the **O_NONBLOCK** flag is enabled for the message queue description, then the call instead fails immediately with the error **EAGAIN**.

RETURN VALUE

On success, **mq_send**() returns zero; on error, −1 is returned, with *errno* set to indicate the error.

NAME

mq_receive − **receive a message from a message queue**

SYNOPSIS

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
        size_t msg_len, unsigned int *msg_prio);
```

DESCRIPTION

mq_receive() removes the oldest message with the highest priority from the message queue referred to by the descriptor *mqdes*, and places it in the buffer pointed to by *msg_ptr*. The *msg_len* argument specifies the size of the buffer pointed to by *msg_ptr*; this must be greater than or equal to the *mq_msgsize* attribute of the queue (see **mq_getattr**(3)). If *msg_prio* is not NULL, then the buffer to which it points is used to return the priority associated with the received message.

If the queue is empty, then, by default, **mq_receive**() blocks until a message becomes available, or the call is interrupted by a signal handler. If the **O_NONBLOCK** flag is enabled for the message queue description, then the call instead fails immediately with the error **EAGAIN**.

RETURN VALUE

On success, **mq_receive**() returns the number of bytes in the received message; on error, −1 is returned, with *errno* set to indicate the error.

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

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

O_CREAT

If the file does not exist, it will be created.

mode specifies the permissions to use in case a new file is created.

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

RETURN VALUE

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

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 one of the errors below 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.