Technische Universität Dortmund

Klausur „Betriebssysteme“

	erreichbare Punkte	erhaltene Punkte						
Aufgabe 1	12	**a**	**b**	**c**	**d**	**e**	**f**	
Aufgabe 2	12	**a**	**b**					
Aufgabe 3	11							
Aufgabe 4	10	**a**	**b**	**c**				
Summe (A1 bis A4)	45							
Summe (Programmierung)	21.5							

(Matrikel-Nr.)

(Vorname) (Name)

Durch meine Unterschrift bestätige ich

- den Empfang der vollständigen Klausur (18 Klausurseiten sowie 7 Handbuchseiten),
- die Kenntnisnahme der Hinweise auf Seite 2.

Dortmund, 01.07.2026 ----------

(Unterschrift)

Hinweise

Bitte lesen Sie die Hinweise **aufmerksam**. Unterschreiben Sie die Erklärung auf der ersten Seite.

- Maximal sind **45 Punkte** möglich. 50% der Punkte reichen zum Bestehen. Die Bearbeitungszeit beträgt **45 Minuten**.
- Antworten Sie auf **Deutsch** oder **Englisch**.
- Es gelten die Begriffe, Algorithmen und Varianten aus Vorlesung, Übungen und Tutorien.
- Abgesehen von den Handbuchseiten am Ende des Dokumentes sind keine Hilfsmittel erlaubt. Alle Aufgaben sind selbstständig und ohne fremde Hilfe zu bearbeiten.
- **Tipp:** Bleiben Sie ruhig. Wenn Sie bei einer Aufgabe nicht weiterkommen, bearbeiten Sie erst eine andere. Teillösungen geben Teilpunkte. Viel Erfolg!

Aufgabe 1: Ankreuzfragen (12 Punkte)

Bei den Einfachauswahlfragen in dieser Aufgabe ist jeweils nur **eine** richtige Antwort eindeutig anzukreuzen. Auf die richtige Antwort gibt es die angegebene Punktzahl.

a) **Verwaiste Prozesse:** Was geschieht mit einem verwaisten Prozess in einem UNIX-System? 2 Punkte

- ☐ Er wird automatisch beendet.
- ☐ Der `init`-Prozess übernimmt ihn.
- ☐ Sein Elternprozess muss auf ihn warten.
- ☐ Er wird beim nächsten Aufruf von `fork()` wiederverwendet.

b) **Prozesszustände:** Welche Aussage zu Prozesszuständen auf einem Mono-Prozessorsystem stimmt? 2 Punkte

- ☐ Bei einem Interrupt wechselt der laufende Prozess für die Interrupt-Behandlung in den Zustand *blockiert* (blocked).
- ☐ Mittelfristige Einplanung kann einen laufenden Prozess in *schwebend laufend* (suspended running) versetzen.
- ☐ Ein Seitenfehler (page fault) kann den laufenden Prozess blockieren (blocked), bis die Seite geladen ist.
- ☐ Bei kooperativem Scheduling kann ein Prozess nie direkt von *laufend* (running) nach *bereit* (ready) wechseln.

c) **Ablaufplanung (Scheduling):** Welche Aussage zum Scheduling stimmt? 2 Punkte

- ☐ First-Come-First-Served (FCFS) sorgt dafür, dass kurze Prozesse immer vor langen Prozessen fertig werden.
- ☐ Prioritätsorientiertes Scheduling kann Prozesse mit niedriger Priorität verhungern lassen (Starvation).
- ☐ Präemptives Scheduling unterbricht einen Prozess nie, solange er noch Rechenzeit benötigt.
- ☐ Round-Robin-Scheduling bevorzugt E/A-intensive Prozesse.

d) **Programme:** Welche Aussage zum Unterschied zwischen *Programmen* und *Prozessen* stimmt?

2 Punkte

- ☐ Ein Programm hat immer eigene Register und einen eigenen Programmzähler.
- ☐ Der Binder (Linker) erzeugt aus Objekt-Dateien einen Prozess.
- ☐ Ein Programm kann immer nur von einem Prozess ausgeführt werden.
- ☐ Ein Prozess ist ein Programm in Ausführung. Ein Prozess kann nacheinander verschiedene Programme ausführen.

e) **Virtueller Speicher:** Was passiert auf einem x86-Linux-System, wenn ein C-Programm über einen ungültigen Zeiger auf Speicher zugreift?

2 Punkte

- ☐ Beim Laden wird die ungültige Adresse erkannt. Der Zugriff wird durch einen Sprung auf eine Abbruchfunktion ersetzt, die das Programm mit „Segmentation fault“ beendet.
- ☐ Der Fehler kann unerkannt bleiben und macht das Programm unsicher.
- ☐ Der Übersetzer erkennt die Stelle und erzeugt Code, der beim Zugriff einen Fehler auslöst.
- ☐ Die MMU erkennt die ungültige Adresse bei der Adressumsetzung und löst einen Trap aus.

f) **Speicher:** Welche Aussage zu *Demand-Paging* stimmt?

2 Punkte

- ☐ Demand-Paging braucht segmentierte Speicherverwaltung.
- ☐ Demand-Paging setzt voraus, dass ein Prozess nie mehr Seiten anspricht als physische Seitenrahmen vorhanden sind.
- ☐ Demand-Paging kommt ohne Hardware-Unterstützung aus; eine MMU ist nicht nötig.
- ☐ Demand-Paging lädt eine Seite erst bei Zugriff in den Hauptspeicher. Nicht genutzte Seiten können ausgelagert werden.

Aufgabe 2: Prozesse und Scheduling (12 Punkte)

a) UNIX-Systemaufrufe

4 Punkte

Welche Ausgabe erzeugt das folgende C-Programm? Schreiben Sie Zeilenumbrüche als **\n**
Hinweis: Header-Dateien und Teile der Fehlerbehandlung fehlen. Nehmen Sie einen fehlerfreien Ablauf an. Jede Ausgabe wird sofort sichtbar.

```
1   int x = 1;
2   void next() {
3       printf("%d\n", x++);
4   }
5   int main() {
6       next();
7       pid_t pid = fork();
8       if (pid > 0) {
9           wait(NULL);
10          next();
11          int x = 0;
12          next();
13      } else if (pid == 0) {
14          next();
15      } else {
16          next();
17          printf("Ohje\n");
18      }
19      return 0;
20  }
```

Ausgabe:

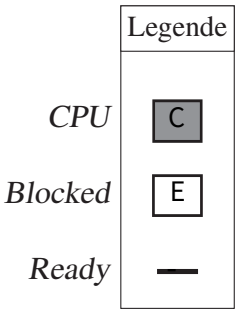
b) Scheduling-Verfahren

8 Punkte

Gegeben sind drei zyklische Prozesse P1, P2 und P3. Sie kommen zu den Zeiten in der Tabelle an. Jeder Zyklus besteht aus einem vollständigen CPU-Stoß und danach einem vollständigen E/A-Stoß. Danach ist der Prozess sofort wieder bereit und startet denselben Zyklus erneut.

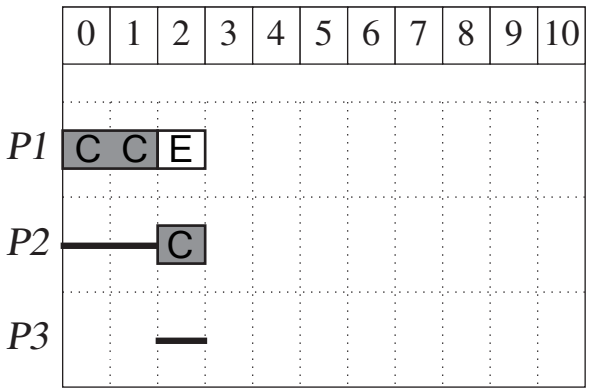
Prozess	Ankunftszeit	CPU-Zeit	E/A-Zeit
P1	0	2	2
P2	0	1	4
P3	2	3	1

Verwenden Sie in jeder Teilaufgabe das dort genannte Scheduling-Verfahren. Die ersten drei Zeiteinheiten sind vorgegeben. Ergänzen Sie das Gantt-Diagramm für P1, P2 und P3 mit den Prozesszuständen entsprechend der Legende.



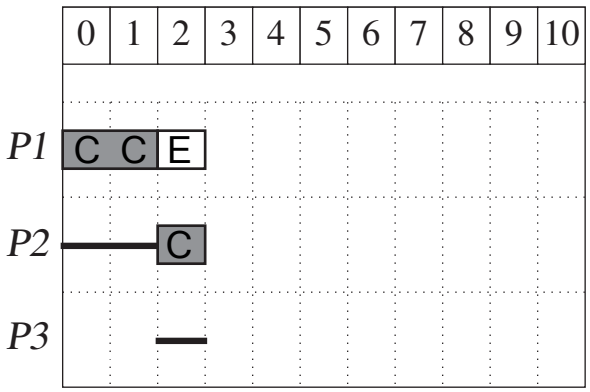
Hinweis: Es gibt **eine CPU**; **E/A-Vorgänge laufen parallel**. Prozesswechsel kosten keine Zeit. War- teschlangen werden nach dem geforderten Scheduling-Verfahren behandelt. Bei gleichzeitigen Ereignissen mit undefinierter Auswahl: Wählen Sie eine Reihenfolge und nutzen Sie sie in der Teilaufgabe konsistent. Bei sonstigem Gleichstand gilt P1 vor P2 vor P3.

I) First Come First Serve (2 Punkte)



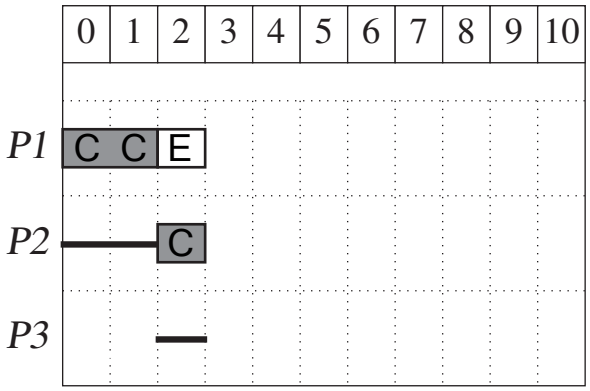
II) Round Robin (2 Punkte)

Zeitscheibe: 2 Zeiteinheiten.

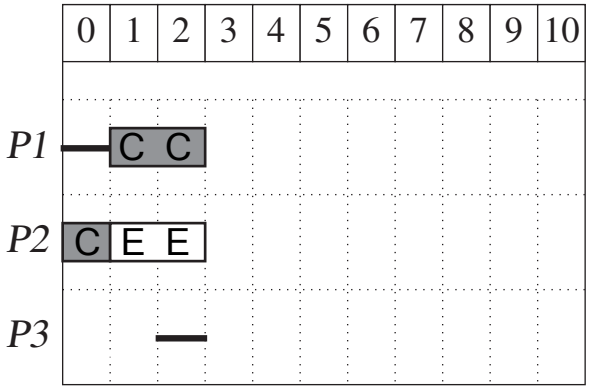


III) Virtual Round Robin (2 Punkte)

Zeitscheibe: 2 Zeiteinheiten.



IV) Shortest-Process-Next-Strategie (2 Punkte)



Aufgabe 3: Synchronisation (11 Punkte)

Gegeben sind ein *Produzent* und ein *Konsument*. Beide laufen zyklisch und sind gleichzeitig bereit. Sie tauschen Daten über den gemeinsamen Puffer `buffer` mit Größe `BUF_SIZE` aus. Nutzen Sie drei Semaphore, um den Puffer zu schützen und Überlauf sowie Unterlauf zu verhindern.

Aufgabe:

- Nutzen und initialisieren Sie alle drei vorgegebenen Semaphoren: freie Plätze, belegte Plätze und Mutex für den kritischen Abschnitt.
- Ergänzen Sie im Programm die nötigen Semaphor-Operationen (Pseudocode, z.B. `P(<Semaphor>)/V(<Semaphor>)`).

```
/* Globale Semaphore */
semaphore S1 =          ;
semaphore S2 =   ;
semaphore S3 = 1;
```

```
/* Globale Daten */
#define BUF_SIZE 4
int buffer[BUF_SIZE];
int in  = 0; /* Einfügeplatz */
int out = 0; /* Entnahmeplatz */
```

```
/* Produzent */

    ;

    ;

buffer[in] = getItem();

in = (in + 1) % BUF_SIZE;

    ;

    ;
```

```
/* Konsument */

    ;

    ;

processItem(buffer[out]);

out = (out + 1) % BUF_SIZE;

    ;

    ;
```

Aufgabe 4: Speicherverwaltung (10 Punkte)

a) Programmsegmente und Variablen

4 Punkte

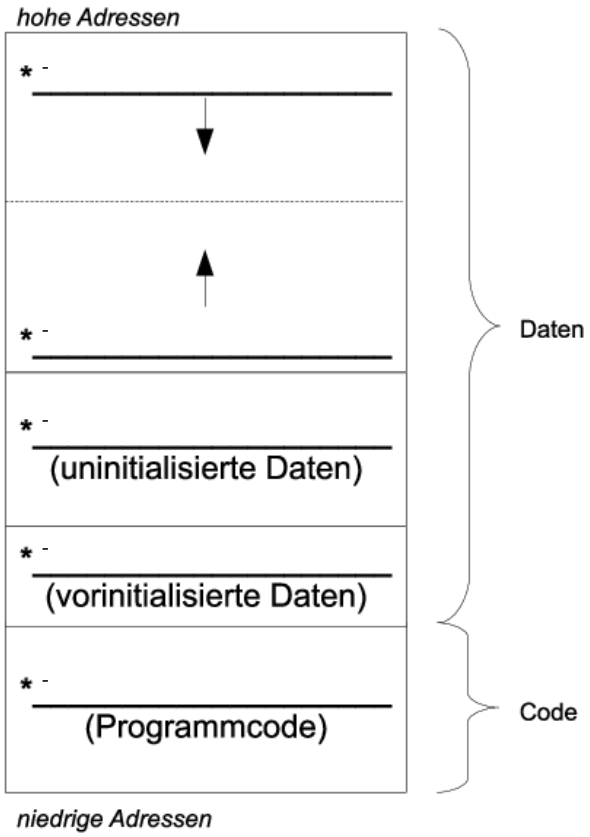
Das folgende C-Programm wird auf einem x86-System übersetzt und gebunden. Ergänzen Sie im Speicherlayout die fehlenden Segmentnamen (*). Ordnen Sie außerdem die Variablen **x**, **a**, **b**, **z** und die Funktionen **add** und **main** den passenden Segmenten zu.

Hinweis: Gehen Sie von einer konzeptionellen Darstellung ohne Compileroptimierungen aus.

```
int x;

void add(int a, int b) {
  int z = 7;
  x = a + b + z;
}

int main(void) {
  x = 1;
  add(x, 5);
  return x;
}
```



Zuordnung:

a : -

b : -

x : -

z : -

add : -

main : -

b) Platzierungsstrategien

4 Punkte

Gegeben ist ein Speicher mit 32 MiB (**1 Feld = 2 MiB**). Jede Teilaufgabe enthält genau eine Speicheranforderung. Die erste Tabellenzeile zeigt die Belegung davor. **Füllen Sie die leere Zeile mit der Belegung danach**. Tragen Sie den tatsächlich reservierten Bereich ein und beachten Sie die angegebene Platzierungsstrategie. Bei Fit-Verfahren wird im gewählten freien Bereich an der kleinsten Adresse platziert. Kann eine Anforderung nicht erfüllt werden, kreuzen Sie das Attribut 'Fehler' an und übernehmen Sie die gegebene Belegung.

- **Buddy-Verfahren (1):**
  - Prozess C besitzt den markierten Bereich bereits und fordert zusätzlich 10 MiB an. Der alte Bereich von C bleibt unverändert: Fehler: ☐

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
		A	A	B	B										C

- **Buddy-Verfahren (2):**
  - Prozess D fordert 7 MiB an: Fehler: ☐

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
		A	A	B	B										C

- **Worst Fit-Verfahren:**
  - Prozess E fordert 2 MiB an: Fehler: ☐

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
		A	A	B	B										C

- **Best Fit-Verfahren:**
  - Prozess F fordert 3 MiB an: Fehler: ☐

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
		A	A	B	B										C

c) Segmentbasierte Adressberechnung

2 Punkte

Berechnen Sie für die logischen Adressen $02\ 040000_{16}$ und $03\ 0F0100_{16}$ jeweils die physikalische Adresse mit segmentbasierter Adressabbildung. Die höchstwertigen 8 Bit der logischen Adresse wählen den Eintrag in der Segmenttabelle. Falls der Zugriff eine Zugriffsverletzung auslöst, kennzeichnen Sie dies dies durch XXXXXXXX.

Segmenttabelle:

	Startadresse	Länge
00_{16}	08 150000	000EFF
01_{16}	02 040000	01DDDD
02_{16}	F0 0D0000	500000
03_{16}	03 0F0100	0000FF
...		
10_{16}	99 C00000	AFFFFF

logische Adresse: 02 040000

→ physikalische Adresse:

logische Adresse: 03 0F0100

→ physikalische Adresse:

Leerseite

Programmieraufgabe: Dateiauflistung mit Größe (21.5 Punkte)

Vervollständigen Sie das Programm `list_sizes`, welches alle regulären Dateien in einem Ver zeichnis nach ihrer Größe sortiert ausgibt.

Aufgabenstellung:

Sie müssen die folgenden Funktionen implementieren:

a) **`list_files()`**, welche ein Array aus Einträgen (`FileInfo`) mit allen regulären Dateien eines Verzeichnisses und deren Größe erstellt.

b) **`main()`**, welche `list_files()` mit einem als Argument übergebenen Pfad aufruft und die Einträge anschließend sortiert auf der Standardausgabe ausgibt.

Details zur Aufgabenstellung erhalten Sie an den passenden Stellen.

Hilfsfunktionen:

Folgende Hilfsfunktionen stehen Ihnen zur Verfügung (Details siehe nächste Seite):

- `count_dirents()`: Ermittelt die Anzahl an Verzeichniseinträgen.
- `concat_paths()`: Konkateniert ein Pfadpräfix mit einem Suffix zu einem vollständigen Pfad. (*Achtung:* Alloziert Speicher).
- `fill_fileinfo()`: Füllt die `FileInfo`-Struktur für eine reguläre Datei.
- `compare_file_sizes()`: Vergleichsfunktion für `FileInfo`-Datensätze für die Nutzung in `qsort()`.
- `error()`: Gibt eine Fehlermeldung aus und bricht das Programm ab (Fehlerfall).

Relevante Systemaufrufe:

- Beigefügte Handbuchseiten: `opendir`, `readdir` und `closedir`, `stat`, `inode`, `qsort`

Wichtige Hinweise:

- *Belegte Ressourcen* (d.h. geöffnete Datein/Verzeichnisse und allozierter Speicher) müssen im Normalfall *freigegeben* werden. Ausnahme: Fehlerfall!
- Achten Sie in Ihrer Implementierung auf *korrekte Fehlerbehandlung* aller Systemaufrufe und Hilfsfunktionen. Nutzen Sie `error()`, um Schreibarbeit zu sparen (Achtung: ggf. auf `errno` achten).
- Einfache syntaktische Fehler (z.B. vergessene Strichpunkte) führen nicht zu Punktabzug, es geht um die semantische Umsetzung.

Vorgegebene Funktionen:

```c
// Gehen Sie davon aus, dass alle notwendigen Header inkludiert sind
// Kurzschreibweise Fehlerbehandlung (Programmabbruch)
void error(char* msg) {perror(msg); exit(errno);}

// Konkateniere Prä- und Suffix zu einem Pfad.
// Rückgabewert muss später mit free freigegeben werden.
// Rückgabe 0: Fehler aufgetreten und errno gesetzt
// Sonst: Rückgabewert ist ein gültiger char*
static char* concat_paths(char* path, char* filename) {
    /* Implementierungsdetails nicht relevant */
}

// Bestimmt die Anzahl an Verzeichniseinträgen.
// Zum Schluss wird die Position im Strom auf den Anfang gesetzt.
// Rückgabe >= 0: Anzahl an verzeichniseinträgen
// Rückgabe -1: Fehler aufgetreten und errno gesetzt
static int count_dirents(DIR* d) {
    /* Implementierungsdetails nicht relevant */
}

// Struktur zum Speichern von Dateinamen und Größe
typedef struct FileInfo {
    char* path; // muss ggf. seperat freigegeben werden
    off_t size;
} FileInfo;

// Speichert den Dateipfad und die Größe in ein FileInfo-Struct.
int fill_fileinfo(struct FileInfo* fi, char* filepath, struct stat* sb) {
    fi->path = filepath;
    fi->size = sb->st_size;
    return 0;
}

// Vergleicht Dateieinträge.
// Rückgabe: neg. wenn a < b, pos. wenn a > b, 0 wenn gleich.
int compare_file_sizes(const void *a, const void *b) {
    FileInfo *fileA = (FileInfo *)a;
    FileInfo *fileB = (FileInfo *)b;
    return fileA->size - fileB->size;
}
```

Die Funktion **`list_files()`** erstellt ein Array mit Einträgen für alle regulären Dateien in einem Verzeichnis. Gehen sie dabei wie folgt vor:

Schritt 1: Diese Funktion öffnet das angegebene Verzeichnis (`path`), bestimmt die Anzahl an Einträge (nutzen Sie `count_dirents()`) darin und alloziert ein entsprechend großes Feld (Array) von `FileInfo`-Einträgen (`malloc` oder `calloc`).

a) `list_files`-Funktion (15 Punkte)

```
FileInfo* list_files(char* path, int* fileCount) {
    DIR* dir;
    struct dirent* entry;
    int num_dirents = 0;
    struct stat sb;
    *fileCount = 0;
```

Schritt 2: Lesen Sie nun alle Einträge des geöffneten Verzeichnisses, erzeugen Sie jeweils den vollen Pfad (`concat_paths()`) und identifiziert mithilfe von `stat()` die regulären Dateien.

Schritt 3: Befüllen Sie fortlaufend das `FileInfo`-Arrary. Nutzen Sie hierfür `fill_FileInfo()`.

Schritt 4: Geben Sie das Array von `FileInfo`-Strukturen als Rückgabewert zurück.

```
}
```

Diese **`main()`** Funktion erwartet einen Pfad auf ein Verzeichnis als Eingabeparameter. Geben sie die Dateien im Ziel nach Größe sortiert aus und gehen Sie dabei wie folgt vor:

Schritt 1: Diese Funktion verarbeitet die Kommandozeilenargumente. Stellen Sie sicher, dass genau ein Argument (der Pfad zum Verzeichnis) übergeben wird.

Schritt 2: Rufen Sie `list_files()` auf, um ein Array von `FileInfo`-Strukturen zu erhalten

Schritt 3: Sortieren Sie das Array mithilfe von `qsort()` und der Vergleichsfunktion `compare_file_sizes()`

Schritt 4: Geben Sie die Dateipfade zusammen mit ihrer Größe in Bytes über die Konsole aus.

Schritt 5: Geben Sie alle allozierten Zeichenketten und Arrays wieder frei.

b) `main`-Funktion und Vergleichsfunktion (siehe nächste Seite) (6.5 Punkte)

```
int main(int argc, char* argv[]) {

    return 0;
}
```

NAME

inode − file inode information

DESCRIPTION

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

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

Inode number
: *stat.st_ino*; *statx.stx_ino*

  Each file in a filesystem has a unique inode number. Inode numbers are guaranteed to be unique only within a filesystem (i.e., the same inode numbers may be used by different filesystems, which is the reason that hard links may not cross filesystem boundaries). This field contains the file's inode number.

File type and mode
: *stat.st_mode*; *statx.stx_mode*

  See the discussion of file type and mode, below.

Link count
: *stat.st_nlink*; *statx.stx_nlink*

  This field contains the number of hard links to the file. Additional links to an existing file are created using **link**(2).

File size
: *stat.st_size*; *statx.stx_size*

  This 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.

Number of blocks allocated to the file
: *stat.st_blocks*; *statx.stx_blocks*

  This 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 POSIX.1 standard notes that the unit for the *st_blocks* member of the *stat* structure is not defined by the standard. On many implementations it is 512 bytes; on a few systems, a different unit is used, such as 1024. Furthermore, the unit may differ on a per-filesystem basis.

The file type and mode

The *stat.st_mode* field (for **statx**(2), the *statx.stx_mode* field) contains the file type and mode.

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

The following mask values are defined for the file type:

S_IFMT	0170000	bit mask for the file type bit field
S_IFSOCK	0140000	socket
S_IFLNK	0120000	symbolic link
S_IFREG	0100000	regular file
S_IFBLK	0060000	block device
S_IFDIR	0040000	directory
S_IFCHR	0020000	character device
S_IFIFO	0010000	FIFO

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

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

Because tests of the above form are common, additional macros are defined by POSIX to allow the test of the file type in *st_mode* to be written more concisely:

S_ISREG(m) is it a regular file?

S_ISDIR(m) directory?

S_ISCHR(m) character device?

S_ISBLK(m) block device?

S_ISFIFO(m) FIFO (named pipe)?

S_ISLNK(m) symbolic link? (Not in POSIX.1-1996.)

S_ISSOCK(m) socket? (Not in POSIX.1-1996.)

The preceding code snippet could thus be rewritten as:

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

SEE ALSO

stat(1), **stat**(2), **statx**(2), **symlink**(7)

NAME

calloc, malloc, free, realloc − Allocate and free dynamic memory

SYNOPSIS

#include <stdlib.h>

void *calloc(size_t *nmemb***, size_t** *size***);**
void *malloc(size_t *size***);**
void free(void **ptr***);*
void *realloc(void **ptr***, **size_t** *size***);**

DESCRIPTION

calloc() allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero.

malloc() allocates *size* bytes and returns a pointer to the allocated memory. The memory is not cleared.

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

realloc() changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If *ptr* is **NULL**, the call is equivalent to **malloc(size)**; if size is equal to zero, the call is equivalent to **free(***ptr***)**. Unless *ptr* is **NULL**, it must have been returned by an earlier call to **malloc**(), **calloc()** or **realloc()**.

RETURN VALUE

For **calloc()** and **malloc()**, the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or **NULL** if the request fails.

free() returns no value.

realloc() returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from *ptr*, or **NULL** if the request fails. If *size* was equal to 0, either NULL or a pointer suitable to be passed to *free*() is returned. If **realloc()** fails the original block is left untouched - it is not freed or moved.

CONFORMING TO

ANSI-C

SEE ALSO

brk(2), **posix_memalign**(3)

NAME

opendir – open a directory / readdir – read a directory

SYNOPSIS

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

DIR *opendir(const char *_name_);
int closedir(DIR *_dirp_);
struct dirent *readdir(DIR *_dir_);

DESCRIPTION opendir

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

RETURN VALUE

The **opendir**() function returns a pointer to the directory stream. On error, NULL is returned, and *errno* is set appropriately.

DESCRIPTION closedir

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

RETURN VALUE

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

DESCRIPTION readdir

The **readdir()** function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by *dir*. It returns NULL on reaching the end-of-file or if an error occurred. It is safe to use **readdir()** inside threads if the pointers passed as *dir* are created by distinct calls to **opendir()**.

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

The *dirent* structure is defined as follows:

```
struct dirent {
    long        d_ino;      /* inode number */
    char        d_name[256]; /* filename */
};
```

RETURN VALUE

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

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

ERRORS

EACCES
Permission denied.

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

ENOTDIR
name is not a directory.

NAME

qsort − sorts an array

SYNOPSIS

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
      int(*compar)(const void *, const void *));
```

DESCRIPTION

The **qsort**() function sorts an array with *nmemb* elements of size *size*. The *base* argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

RETURN VALUE

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

EXAMPLES

For one example of use, see the example under **bsearch**(3).

Another example is the following program, which sorts the strings given in its command-line arguments:

```
#include <stdio.h> #include <stdlib.h> #include <string.h>  static int cmpstringp(const void *p1, const void
*p2) {
   /* The actual arguments to this function are "pointers to
      pointers to char", but strcmp(3) arguments are "pointers
      to char", hence the following cast plus dereference. */
   return strcmp(*(const char **) p1, *(const char **) p2); }  int main(int argc, char *argv[]) {
   if (argc < 2) {
      fprintf(stderr, "Usage: %s <string>...\n", argv[0]);
      exit(EXIT_FAILURE);
   }
   qsort(&argv[1], argc − 1, sizeof(char *), cmpstringp);
   for (size_t j = 1; j < argc; j++)
      puts(argv[j]);
   exit(EXIT_SUCCESS); }
```

SEE ALSO

sort(1), **alphasort**(3), **strcmp**(3), **versionsort**(3)

ATTRIBUTES

Multithreading (see pthreads(7))

The **qsort**() function is thread-safe if the comparison function *compar* does not access any global variables.

NAME

stat, fstat, lstat − get file status

SYNOPSIS

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

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

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

lstat(): _BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

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

stat() stats the file pointed to by *path* and fills in *buf*.

lstat() is identical to **stat**(), except that if *path* is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

fstat() is identical to **stat**(), except that the file to be stat-ed 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;    /* protection */
    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 file system I/O */
    blkcnt_t  st_blocks;  /* number of blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

The *st_dev* field describes the device on which this file resides.

The *st_rdev* field describes the device that this file (inode) represents.

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 symlink is the length of the pathname it contains, without a trailing 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 file system I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

Not all of the Linux file systems implement all of the time fields. Some file system types allow mounting in such a way that file accesses do not cause an update of the *st_atime* field. (See "noatime" in **mount**(8).)

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

The field *st_mtime* is changed by file modifications, for example, by **mknod**(2), **truncate**(2), **utime**(2) and **write**(2) (of more than zero bytes). Moreover, *st_mtime* of a directory is changed by the creation or deletion of files in that directory. The *st_mtime* field is *not* changed for changes in owner, group, hard link count, or mode.

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

The following POSIX macros are defined to check the file type using the *st_mode* field:

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?
S_ISCHR(m)	character device?
S_ISBLK(m)	block device?
S_ISFIFO(m)	FIFO (named pipe)?
S_ISLNK(m)	symbolic link? (Not in POSIX.1-1996.)
S_ISSOCK(m)	socket? (Not in POSIX.1-1996.)

RETURN VALUE

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

ERRORS

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

EBADF
fd is bad.

EFAULT
Bad address.

ELOOP
Too many symbolic links encountered while traversing the path.

ENAMETOOLONG
File name too long.

ENOENT
A component of the path *path* does not exist, or the path is an empty string.

ENOMEM
Out of memory (i.e., kernel memory).

ENOTDIR
A component of the path is not a directory.

SEE ALSO

access(2), **chmod**(2), **chown**(2), **fstatat**(2), **readlink**(2), **utime**(2), **capabilities**(7), **symlink**(7)