Name, Vorname:	Matrikelnummer:

Klausur „Betriebssysteme“

Prof. Dr.-Ing. Peter Ulbrich

3. August 2022

Technische Universität Dortmund, Fakultät für Informatik

Hinweise:

- Die Klausur besteht aus 21 Seiten, inklusive Deck-, Reserveblatt und Handbuchseiten. Es können 60 Punkte erreicht werden; 50% der Gesamtpunktzahl reicht zum Bestehen der Klausur. Zur Bearbeitung stehen insgesamt 60 Minuten zur Verfügung.
- Als Hilfsmittel ist lediglich **ein handbeschriebenes DIN A4-Blatt** erlaubt. Dies wird nach der Klausur eingesammelt. Ansonsten dürfen **keine** weiteren Hilfsmittel verwendet werden.
- Die Antworten sind in **deutscher** oder **englischer** Sprache zu verfassen.
- Notieren Sie Ihre Lösungen direkt auf den ausgeteilten Aufgabenblättern unter Verwendung eines dokumentenechten, schwarzen oder blauen Stifts. Vor der Bearbeitung der Aufgaben **müssen** auf **allen** Blättern Ihr **Name** und Ihre **Matrikelnummer** eingetragen werden. Entfernen Sie nicht die Heftung der Blätter. Falls der vorgesehene Platz nicht ausreichen sollte, können Sie auch die Rückseiten der Blätter und die Reserveseiten am Ende verwenden. Notieren Sie in diesem Fall aber an der für die Lösung vorgesehenen Stelle einen Verweis auf die Seite. Bei den Multiple-Choice-Aufgaben können, solange nicht anders angegeben, grundsätzlich mehrere Antworten richtig und anzukreuzen sein.
- Da es unterschiedliche Sprachgebräuche sowie Algorithmen- und Konzept-Ausprägungen gibt, werden hier ausdrücklich die aus Vorlesung und Übungen bekannten Ausdrucksweisen und Ausprägungen zu Grunde gelegt.

Aufgabe	**1**	**2**	**3**	**4**	**5**	**6**	**Summe**
mögliche Punkte	8	10	10	9	8	15	60
erreichte Punkte							

Name, Vorname:	Matrikelnummer:

1 Multiple-Choice-Fragen (8 Punkte)

Bei den Multiple-Choice-Fragen in dieser Aufgabe ist jeweils nur **eine** Antwort richtig (Einfachauswahl-Fragen). Auf diese gibt es jeweils die angegebenen Punkte. *Lesen Sie die Frage genau, bevor Sie antworten!*

Prozesse und Scheduling: Welche Aussage über *Ablaufplanung* (CPU scheduling) ist richtig? (2 Punkte)

- ❑ Präemptive Ablaufplanung drängt einen Prozess möglicherweise dazu die CPU abzugeben.
- ❑ Der Konvoieffekt kann bei kooperativen Planungsverfahren wie First-Come-First-Served nicht auftreten.
- ❑ Virtual-Round-Robin ist für den interaktiven Betrieb ungeeignet.
- ❑ Round-Robin bevorzugt E/A-intensive Prozesse zu Gunsten von rechenintensiven Prozessen.

Speicher: Bei Demand-Paging kann *Seitenflattern* (Thrashing) auftreten. Welche Aussage ist richtig? (2 Punkte)

- ❑ Seitenflattern tritt auf, wenn ein Prozess zum Weiterarbeiten häufig gerade die Seiten benötigt, die durch das Betriebssystem im Rahmen einer globalen Ersetzungsstrategie gerade erst ausgelagert wurden.
- ❑ Seitenflattern tritt auf, wenn Seiten zur Defragementierung im Speicher verschoben werden.
- ❑ Bei der Ersetzungsstrategie *Second Chance* (SC) wird bei einem Zugriff auf eine Seite ein Referenzbit gesetzt. Wird die Seite längere Zeit nicht angesprochen, so wird dieses Bit gelöscht. Da dieses Bit ständig den Wert ändert, spricht man von Seitenflattern.
- ❑ Seitenflattern kann nur auftreten, wenn der dynamisch genutzte Speicher eines Prozesses größer ist, als der physikalisch vorhandene Speicher des Systems.

Synchronisierung und Verklemmungen: Welche der folgenden Aussagen zu aktivem Warten ist richtig? (2 Punkte)

- ❑ Aktives Warten vergeudet gegenüber passivem Warten immer CPU-Zeit.
- ❑ Bei verdrängenden Scheduling-Strategien verzögert aktives Warten nur den betroffenen Prozess, behindert aber nicht andere.
- ❑ Aktives Warten darf bei nicht-verdrängenden Scheduling-Strategien auf einem Einkernrechensystemen nicht verwendet werden.
- ❑ Auf Mehrkernsystemen ist aktives Warten unproblematisch und deshalb dem passiven Warten immer vorzuziehen.

Sicherheit: Welche Aussage über Systemsicherheit ist korrekt? (2 Punkte)

- ❑ Die Isolation des Adressraums eines Prozesses ist ohne eine MMU nicht möglich.
- ❑ Der Einsatz von Systemaufrufen (system calls) im Anwendungsprogramm stellt eine Verletzung der Privilegseparation dar.
- ❑ Das Betriebssystem überprüft alle Speicherzugriffe und verhindert Pufferüberläufe.
- ❑ Die Privilegseparation auf Hardwareebene (z.B. Schutzringe) gibt dem Betriebssystem die Kontrolle über die laufenden Prozesse des Systems.

Name, Vorname:	Matrikelnummer:

2 Prozesse und Scheduling (10 Punkte)

a) UNIX-Systemaufrufe (3 Punkte) Beantworten Sie zu diesem C-Programm die folgenden Fragen. Gehen Sie von einer fehlerfreien Abarbeitung aus.
Hinweis: Das Einbinden der Header-Dateien, sowie eine ordentliche Fehlerbehandlung, wurden der Einfachheit halber weggelassen.

```
1  int x = 0;
2
3  int main(void) {
4      int pid;
5
6      pid = fork();
7
8      if (pid > 0) {
9          int status;
10         wait(&status);
11     }
12     else if (pid == 0) {
13         x = 3;
14         exit(0);
15     }
16
17     x++;
18
19     printf("%d", x);
20     return 0;
21 }
```

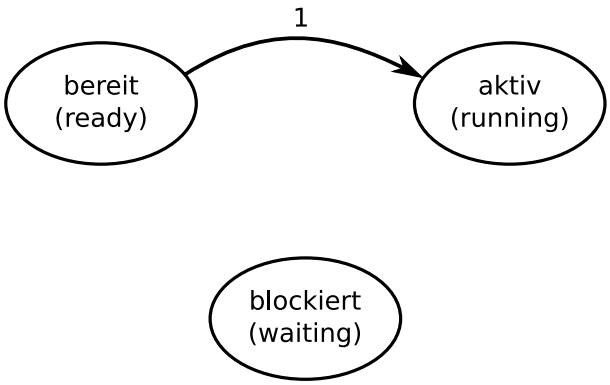
1) „fork()“ (1 Punkte) Was bewirkt der Befehl in Zeile 6?

2) Ausgabe (2 Punkte) Was ist die Ausgabe des Programms?

Ausgabe:

b) Zustandsübergänge (4 Punkte) Ergänzen Sie im untenstehenden Prozesszustandsdiagramm mit drei Zuständen die fehlenden Übergänge und beschreiben Sie jeweils kurz, durch welches Ereignis der jeweilige Zustandsübergang ausgelöst wird. Als Beispiel ist der Übergang vom Zustand „bereit“ zu „aktiv“ bereits vorgegeben:

Zustandsübergang 1 „bereit“ $\rightarrow$ „aktiv“: Der Scheduler hat einen neuen Prozess aus der Liste der bereiten Prozesse ausgewählt und diesem den Prozessor zugeteilt.



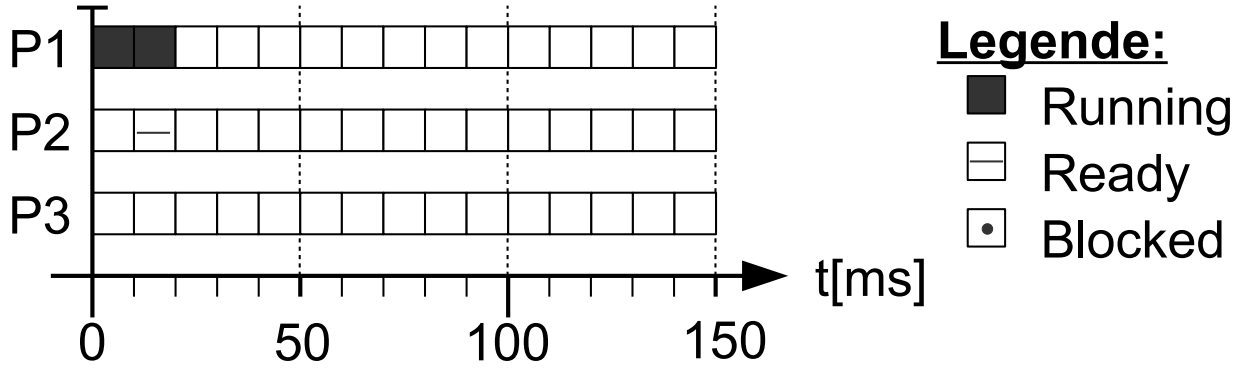
c) Prozess-Scheduling (Round Robin) (3 Punkte) Ein Betriebssystem verwaltet **drei zyklisch** arbeitende Prozesse P1, P2 und P3. Jeder Prozess führt zunächst Berechnungen auf der CPU aus. Sobald diese vollständig abgeschlossen sind, folgt ein E/A-Stoß, anschließend ist der Prozess wieder rechenbereit. Die Prozesse treffen zum Zeitpunkt der in der Tabelle angegebenen Ankunftszeit ein. In der folgenden Tabelle sind die Zeitangaben für die Ankunftszeit und Dauer von CPU- und E/A-Stößen jeweils in ms angegeben.

Prozess	Ankunftszeit	CPU-Zeit	E/A-Zeit
P1	0	40	20
P2	10	20	10
P3	20	30	20

Zeichnen Sie in das folgende Gantt-Diagramm ein, wie die drei Prozesse 1, P2 und P3 abgearbeitet werden, wenn das Scheduling nach der *Round-Robin*-Strategie mit einer Zeitscheibendauer von **30 ms** vorgenommen wird. Die Prozessumschaltzeit kann vernachlässigt werden. E/A-Vorgänge können parallel ausgeführt werden. Markieren Sie in dem folgenden Diagramm die Prozesszustände entsprechend der Legende. Es genügt, die **ersten 150 ms** anzugeben.

Hinweis: Die ersten zwei Zeiteinheiten sind bereits fertig ausgefüllt.

Name, Vorname:	Matrikelnummer:



(Ersatzdiagramme auf dem Reserveblatt: Streichen Sie ungültige Lösungen deutlich durch!)

Name, Vorname:	Matrikelnummer:

3 Synchronisierung und Verklemmungen (10 Punkte)

a) Leser/Schreiber-Problem (5 Punkte) Wie beim gegenseitigen Ausschluss, wird auch beim Leser/Schreiber-Problem ein kritischer Abschnitt geschützt. Allerdings werden die Prozesse in zwei Arten von konkurrierenden Prozessen aufgeteilt, den *Schreibern* und den *Lesern*. In der unten vorgestellten Implementierung haben sich jedoch **drei** Fehler eingeschlichen, die dafür sorgen, dass die Synchronisierung zwischen den konkurrierenden Prozessen nicht reibungslos abläuft. Markieren Sie die **drei** Fehler im Quellcode mit 1, 2, und 3 und beschreiben Sie kurz das jeweilige Problem.

```
/* gemeinsamer Speicher */
int readcount = 0;
Semaphore mutex = 1;
Semaphore wrt = 0;
```

```
/* Schreiber */


p(&wrt);


... schreibe


v(&wrt);
```

```
/* Leser */


readcount++;
p(&mutex);
if (readcount == 1)
    p(&wrt);
v(&mutex);


... lese


p(&mutex);
readcount--;
if (readcount == 0)
    v(&wrt);
```

Fehler 1: _____

Fehler 2: _____

Fehler 3: _____

Name, Vorname:	Matrikelnummer:

b) Synchronisation (5 Punkte) Die drei Funktionen des folgenden Programms werden als Fäden ausgeführt. Alle drei Fäden sind zur selben Zeit laufbereit. Sorgen Sie durch geeignete Synchronisation der Abläufe dafür, dass das Programm die ersten sechs Primzahlen in einer **absteigenden** Reihenfolge (13, 11, 7, 5, 3, 2) ausgibt. Zu diesem Zweck stehen ihnen **drei** Semaphore zur Verfügung. Diese gilt es geeignet zu initialisieren und anschließend, an den freien Stellen im Programm, die nötigen Semaphor-Operationen einzufügen.

```
Initialwerte der Semaphoren: S1: [ ] S2:[ ] S3: [ ]
```

```
f1() {

   printf("5");

   printf("3");

}
```

```
f2() {

   printf("11");

   printf("2");

}
```

```
f3() {

   printf("13");

   printf("7");

}
```

4 Speicherverwaltung und virtueller Speicher (9 Punkte)

a) (1) Platzierungsstrategien (6 Punkte) Die folgenden Tabellen zeigen die Belegung eines Speichers der Größe 32 MiB. In der ersten Tabelle sind beispielsweise 4 MiB von Prozess A belegt. Tragen Sie jeweils in die leere Zeile die *vollständige* Belegung des Speichers ein.

Hinweis: Falls eine Belegung *nicht* erfüllt werden kann, kennzeichnen Sie die betreffende Tabelle deutlich.

Wichtig: Beachten Sie, dass ein Feld in den Tabellen **zwei** MiB entspricht!

- **Buddy-Verfahren:**
  - Prozess E belegt 9 MiB:

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

- **Worst Fit-Verfahren:**
  - Prozess F belegt 2 MiB:

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

- **First Fit-Verfahren:**
  - Prozess G belegt 4 MiB:

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

Name, Vorname:	Matrikelnummer:

b) Speichersegmentierung (4 Punkte) Übersetzen Sie mit Hilfe der Speichersegmentierung die logischen Adressen *0x0200 0100* und *0x0101 FFF0* in physikalische Adressen. Die höchstwertigen 8 Bit der logischen Adresse geben die Position innerhalb der Segmenttabelle an. Löst eine Speicheranfrage eine Zugriffsverletzung aus, so machen Sie dies bitte kenntlich.

Segmenttabelle:

	Startadresse	Länge
01_{16}	B542 0000_{16}	02 0000_{16}
02_{16}	D471 0000_{16}	00 $F000_{16}$
03_{16}	8080 0000_{16}	00 $FFFF_{16}$
...		
10_{16}	4310 0000_{16}	FF $FFFF_{16}$

logische Adresse: *0x0200 0100*$_{16}$
→ physikalische Adresse: ☐

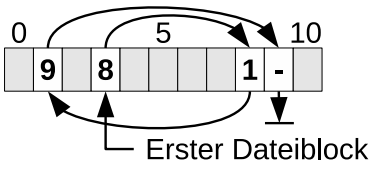
logische Adresse: *0x0101 FFF0*$_{16}$
→ physikalische Adresse: ☐

c) TLB (2 Punkte) Welche Rolle spielt der sogenannte *Translation Lookaside Buffer* (z. B. auf x86-Systemen) im Zusammenhang mit der Adressabbildung?

Name, Vorname:	Matrikelnummer:

5 E/A und Dateisysteme (8 Punkte)

a) Verkettete Speicherung außerhalb der Datenblöcke (2 Punkte) Beantworten Sie die folgenden Fragen zu dem aus der Vorlesung bekannten Schema der verketteten Speicherung von Dateien, bei dem die Verkettung in einem separaten Plattenbereich und nicht innerhalb der Datenblöcke selbst abgelegt wird (siehe nachfolgende Abbildung – markiert ist die Blockfolge 3, 8, 1, 9).



1. **Dateisystem (1 Punkt)** Welches Dateisystem nutzt, den oben dargestellten Ansatz zur verketteten Speicherung von Dateiinhalten?

   _____

2. **Vergleiche (1 Punkte)** Nennen Sie einen Vorteil dieses Schemas im Vergleich zur kontinuierlichen Speicherung:

   _____

   _____

b) I/O-Scheduling (6 Punkte) Gegeben sei ein Plattenspeicher mit 16 Spuren. Der jeweilige I/O-Scheduler bekommt immer wieder Leseaufträge für eine bestimmte Spur. Die Leseaufträge in L_1 sind dem I/O-Scheduler bereits bekannt. Nach **zwei** bearbeiteten Aufträgen erhält er die Aufträge in L_2. Nach **weiteren drei** (d.h. nach insgesamt **fünf**) bearbeiteten Aufträgen erhält er die Aufträge in L_3. Zu Beginn befindet sich der Schreib-/Lesekopf über Spur 0.

$L_1 = \{11, 3, 8, 14\}, L_2 = \{7, 1, 10\}, L_3 = \{4, 9, 6\}$

Bitte tragen Sie hier die Reihenfolge der gelesenen Spuren für einen I/O-Scheduler, der nach der **Shortest Seek Time First (SSTF)** Strategie arbeitet, ein:

Bitte tragen Sie hier die Reihenfolge der gelesenen Spuren für einen I/O-Scheduler, der nach der **Fahrstuhl (Elevator)** Strategie arbeitet, ein:

(Ersatztabellen auf dem Reserveblatt: Streichen Sie ungültige Lösungen deutlich durch!)

Name, Vorname:	Matrikelnummer:

6 Programmieraufgabe (15 Punkte)

Implementieren Sie das Programm `countfiles`, welches mit einer beliebigen Anzahl an Verzeichnissen, mindestens jedoch mindestens einem, aufgerufen wird:

Schnittstelle: `./countfiles` [Verzeichnis1] [Verzeichnis2] [Verzeichnis3] ...

(Beispiel: `./countfiles` foo/ bar/)

Funktionsweise:

a) main-Funktion (5 Punkte)

- Zunächst soll die Argumentenliste auf den korrekten Aufruf hin überprüft werden.
- Anschliessend soll ihr Programm für jedes Argument die Funktion `int count_files(char *dirName)` aufrufen.
- Für jeden Aufruf soll die Rückgabewert aufsummiert werden.
- Zum Abschluss soll die Summe zusammen mit dem Text **„Sum of all files“** ausgeben werden.

b) counf_files-Funktion (10 Punkte)

- Die Funktion `int count_files(char *dirName)` soll das Verzeichnis `dirName` öffnen, über alle Einträge iterieren und anschließend wieder schließen.
- Der übergebene Pfad `dirName` muss nicht nochmal explizit auf seine Richtigkeit oder Fehler überprüft werden. Es genügt, dies über eine passende Behandlung des Aufrufs von `opendir` zu erledigen.
- Für jeden Eintrag soll mit Hilfe der gegebenen Funktion `int dir_entry_type(char *path)` geprüft werden, ob es sich um eine Datei (Rückgabewert 1) oder ein Verzeichnis (Rückgabewert 2) handelt.
- Handelt es sich um eine Datei, so soll deren Vorkommen gezählt werden.
- Handelt es sich um ein Verzeichnis, soll nichts geschehen.
- Die Anzahl aller Dateien soll am Ende der Funktion zurückgegeben werden.
- Beachten Sie die bereitgestellten Hilfsfunktionen `char* concat_paths(const char *first, const char *second)` und `int is_dot_dir(char *entryName)`.
  Beide müssen an geeigneten Stellen verwendet werden.

Fehlerbehandlung: Beachten Sie für Ihr gesammtes Programm, dass Fehler auftreten können und entsprechend behandelt werden müssen.

Rückgabewert

0 Die Ausführung war korrekt.

1 Es gab einen Fehler bei der Ausführung.

Relevante Manual-Seiten: opendir, readdir, closedir

Hinweis: Einfache syntaktische Fehler (z.B. vergessene Strichpunkte) führen nicht zu Punktabzug, es geht um die sematische Umsetzung.

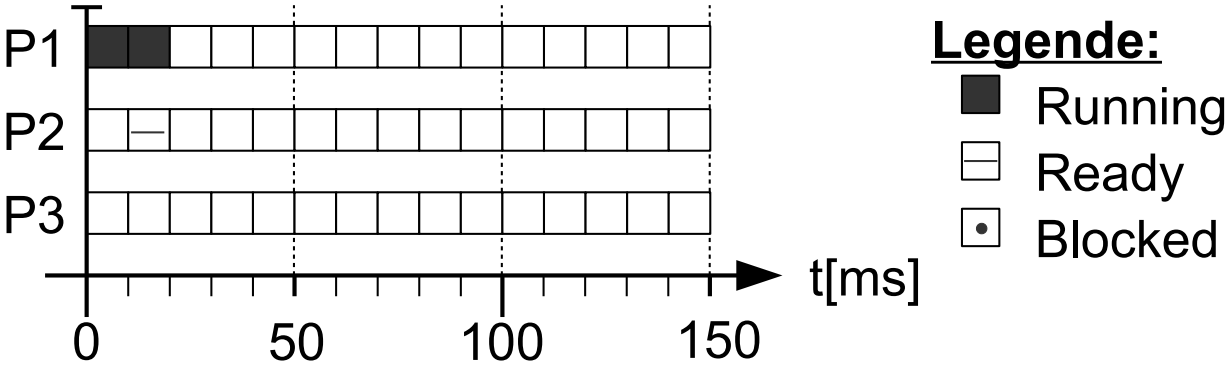
Name, Vorname:	Matrikelnummer:

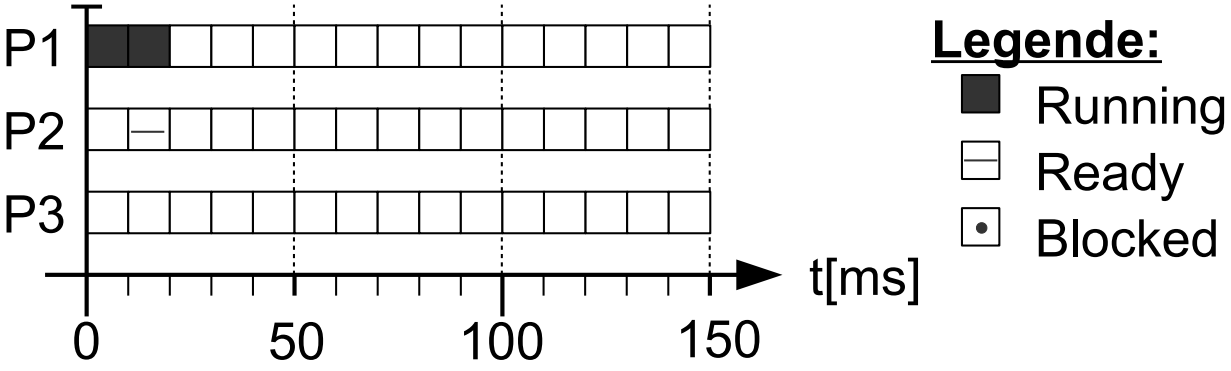
```c
#include <sys/stat.h>
#include <dirent.h>
#include <string.h>

static int count_files(const char *dirName);

/*
 * Alloziert dynamisch Speicher und
 * konkanteniert beide Parameter in einem String, inkl. Slash.
 * Es wird ein Zeiger auf den allozierten Speicher zurückgegeben.
 */
static char* concat_paths(const char *first, const char *second) {
        /*  Aus Platzgründen weggelassen */
}

/*
 * Bestimmt für den Parameter pfath, ob es eine Datei (Rückgabewert 1)
 * oder ein Verzeichnis (Rückgabewert 2) ist.
 */
static int dir_entry_type(char *path) {
        /*  Aus Platzgründen weggelassen */
}

static int is_dot_dir(char *entryName) {
        return !strncmp(entryName, ".", 1) || !strncmp(entryName, "..", 2);
}

/*
 * Ab hier den Code für die main-Funktion
 * sowie für count_files-Funktion einfügen.
 */
int main(int argc, const char *argv[]) {
```

Name, Vorname:	Matrikelnummer:

Name, Vorname:	Matrikelnummer:

Reserveblatt





Bitte tragen Sie hier die Reihenfolge der gelesenen Spuren für einen I/O-Scheduler, der nach der **Shortest Seek Time First (SSTF)** Strategie arbeitet, ein:

Bitte tragen Sie hier die Reihenfolge der gelesenen Spuren für einen I/O-Scheduler, der nach der **Fahrstuhl (Elevator)** Strategie arbeitet, ein:

Name, Vorname:	Matrikelnummer:

Reserveblatt

NAME

opendir, fdopendir − open a directory

SYNOPSIS

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

DIR *opendir(const char *name);
DIR *fdopendir(int fd);
```

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

fdopendir():
  Since glibc 2.10:
    _POSIX_C_SOURCE >= 200809L
  Before glibc 2.10:
    _GNU_SOURCE

DESCRIPTION

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.

The **fdopendir**() function is like **opendir**(), but returns a directory stream for the directory referred to by the open file descriptor *fd*. After a successful call to**fdopendir**(), *fd* is used internally by the implementation, and should not otherwise be used by the application.

RETURN VALUE

The **opendir**() and **fdopendir**() functions return a pointer to the directory stream. On error, NULL is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
Permission denied.

EBADF
fd is not a valid file descriptor opened for reading.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

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

ENOMEM
Insufficient memory to complete the operation.

ENOTDIR
name is not a directory.

VERSIONS

fdopendir() is available in glibc since version 2.4.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
opendir(), **fdopendir**()	Thread safety	MT-Safe

CONFORMING TO

opendir() is present on SVr4, 4.3BSD, and specified in POSIX.1-2001. **fdopendir**() is specified in POSIX.1-2008.

NOTES

Filename entries can be read from a directory stream using **readdir**(3).

The underlying file descriptor of the directory stream can be obtained using **dirfd**(3).

The **opendir**() function sets the close-on-exec flag for the file descriptor underlying the *DIR* *. The **fdopendir**() function leaves the setting of the close-on-exec flag unchanged for the file descriptor, *fd*. POSIX.1-200x leaves it unspecified whether a successful call to **fdopendir**() will set the close-on-exec flag for the file descriptor, *fd*.

SEE ALSO

open(2), **closedir**(3), **dirfd**(3), **readdir**(3), **rewinddir**(3), **scandir**(3), **seekdir**(3), **telldir**(3)

COLOPHON

This page is part of release 5.13 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.

NAME

readdir − read a directory

SYNOPSIS

#include <dirent.h>

struct dirent *readdir(DIR **dirp*);

DESCRIPTION

The **readdir**() function returns a pointer to a *dirent* structure representing the next directory entry in the directory stream pointed to by *dirp*. It returns NULL on reaching the end of the directory stream or if an error occurred.

In the glibc implementation, the *dirent* structure is defined as follows:

```
struct dirent {
    ino_t          d_ino;       /* Inode number */
    off_t          d_off;       /* Not an offset; see below */
    unsigned short d_reclen;    /* Length of this record */
    unsigned char  d_type;      /* Type of file; not supported
                                   by all filesystem types */
    char           d_name[256]; /* Null-terminated filename */
};
```

The only fields in the *dirent* structure that are mandated by POSIX.1 are *d_name* and *d_ino*. The other fields are unstandardized, and not present on all systems; see NOTES below for some further details.

The fields of the *dirent* structure are as follows:

d_ino This is the inode number of the file.

d_off The value returned in *d_off* is the same as would be returned by calling **telldir**(3) at the current position in the directory stream. Be aware that despite its type and name, the *d_off* field is seldom any kind of directory offset on modern filesystems. Applications should treat this field as an opaque value, making no assumptions about its contents; see also **telldir**(3).

d_reclen

This is the size (in bytes) of the returned record. This may not match the size of the structure definition shown above; see NOTES.

d_type This field contains a value indicating the file type, making it possible to avoid the expense of calling **lstat**(2) if further actions depend on the type of the file.

When a suitable feature test macro is defined (**_DEFAULT_SOURCE** on glibc versions since 2.19, or **_BSD_SOURCE** on glibc versions 2.19 and earlier), glibc defines the following macro constants for the value returned in *d_type*:

DT_BLK This is a block device.

DT_CHR This is a character device.

DT_DIR This is a directory.

DT_FIFO This is a named pipe (FIFO).

DT_LNK This is a symbolic link.

DT_REG This is a regular file.

DT_SOCK This is a UNIX domain socket.

DT_UNKNOWN

The file type could not be determined.

Currently, only some filesystems (among them: Btrfs, ext2, ext3, and ext4) have full support for returning the file type in *d_type*. All applications must properly handle a return of**DT_UNKNOWN**.

d_name
This field contains the null terminated filename. *See NOTES*.

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

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 to indicate the error. To distinguish end of stream from an error, set *errno* to zero before calling **readdir**() and then check the value of *errno* if NULL is returned.

ERRORS

EBADF
Invalid directory stream descriptor *dirp*.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
readdir()	Thread safety	MT-Unsafe race:dirstream

In the current POSIX.1 specification (POSIX.1-2008), **readdir**() is not required to be thread-safe. However, in modern implementations (including the glibc implementation), concurrent calls to **readdir**() that specify different directory streams are thread-safe. In cases where multiple threads must read from the same directory stream, using **readdir**() with external synchronization is still preferable to the use of the deprecated **readdir_r**(3) function. It is expected that a future version of POSIX.1 will require that **readdir**() be thread-safe when concurrently employed on different directory streams.

CONFORMING TO

POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD.

NOTES

A directory stream is opened using **opendir**(3).

The order in which filenames are read by successive calls to **readdir**() depends on the filesystem implementation; it is unlikely that the names will be sorted in any fashion.

Only the fields *d_name* and (as an XSI extension) *d_ino* are specified in POSIX.1. Other than Linux, the *d_type* field is available mainly only on BSD systems. The remaining fields are available on many, but not all systems. Under glibc, programs can check for the availability of the fields not defined in POSIX.1 by testing whether the macros **_DIRENT_HAVE_D_NAMLEN**, **_DIRENT_HAVE_D_RECLEN**, **_DIRENT_HAVE_D_OFF**, or **_DIRENT_HAVE_D_TYPE** are defined.

The d_name field

The *dirent* structure definition shown above is taken from the glibc headers, and shows the *d_name* field with a fixed size.

Warning: applications should avoid any dependence on the size of the *d_name* field. POSIX defines it as *char d_name[]*, a character array of unspecified size, with at most **NAME_MAX** characters preceding the terminating null byte ('\0').

POSIX.1 explicitly notes that this field should not be used as an lvalue. The standard also notes that the use of *sizeof(d_name)* is incorrect; use *strlen(d_name)* instead. (On some systems, this field is defined as *char d_name[1]*!) By implication, the use*sizeof(struct dir ent)* to capture the size of the record including the size of *d_name* is also incorrect.

Note that while the call

fpathconf(fd, _PC_NAME_MAX)

returns the value 255 for most filesystems, on some filesystems (e.g., CIFS, Windows SMB servers), the null-terminated filename that is (correctly) returned in *d_name* can actually exceed this size. In such cases, the *d_reclen* field will contain a value that exceeds the size of the glibc *dirent* structure shown above.

SEE ALSO

getdents(2), **read**(2), **closedir**(3), **dirfd**(3), **ftw**(3), **offsetof**(3), **opendir**(3), **readdir_r**(3), **rewinddir**(3), **scandir**(3), **seekdir**(3), **telldir**(3)

COLOPHON

This page is part of release 5.13 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.

NAME

closedir − close a directory

SYNOPSIS

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

int closedir(DIR ****dirp*);**

DESCRIPTION

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*dirp* is not available after this call.

RETURN VALUE

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

ERRORS

EBADF

Invalid directory stream descriptor *dirp*.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
closedir()	Thread safety	MT-Safe

CONFORMING TO

POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD.

SEE ALSO

close(2), **opendir**(3), **readdir**(3), **rewinddir**(3), **scandir**(3), **seekdir**(3), **telldir**(3)

COLOPHON

This page is part of release 5.13 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.