Leibniz Universität Hannover

Klausur Grundlagen der Betriebssysteme

	erreichbare Punkte	erhaltene Punkte
Aufgabe 1	14	
Aufgabe 2	11	
Aufgabe 3	18	
Aufgabe 4	6	
Aufgabe 5	30	
Aufgabe 6	11	
Summe	90	

(Name) (Vorname) (Matrikel-Nr.)

(Studiengang) (Semester)

Durch meine Unterschrift bestätige ich

- den Empfang der vollständigen Klausur (20 Seiten inklusive Deckblatt),
- den Empfang der Manualseiten (closedir, exec, fork, opendir, readdir, sem_destroy, sem_getvalue, sem_init, sem_post, sem_wait, strstr und wait),
- die Kenntnisnahme der Hinweise auf Seite 2.

Hannover, 03.03.2023

(Unterschrift)

Hinweise

Bitte lesen Sie die folgenden Informationen aufmerksam und unterschreiben Sie die Erklärung auf der ersten Seite.

- Die Bearbeitungszeit beträgt 90 Minuten.
- Es sind **keine** eigenen Hilfsmittel zugelassen.
- Die Lösung einer Aufgabe soll auf das Aufgabenblatt in den dafür vorgesehenen Raum geschrieben werden. Beachten Sie bitte, dass der freigelassene Platz großzügig bemessen ist und nicht unbedingt der erwarteten Antwortlänge entspricht. Sollte der Platz nicht ausreichen, können Sie die Rückseiten der Aufgabenblätter mitverwenden. Kennzeichnen Sie dabei die Zugehörigkeit Ihrer Lösung zu einer Aufgabe. Bei Bedarf können zusätzliche Lösungsblätter (weiß) ausgeteilt werden. Vermerken Sie vor deren Verwendung unbedingt Ihren Namen und Ihre Matrikelnummer darauf!
- Die Lösungen müssen dokumentenecht in blau oder schwarz geschrieben werden. Als falsch Erkanntes muss deutlich durchgestrichen werden. Tintenkiller und andere Korrekturstifte dürfen nicht verwendet werden. Keinen Bleistift verwenden!
- Fragen zu den Prüfungsaufgaben können grundsätzlich **nicht** beantwortet werden.
- Tragen Sie Ihren Namen und Vornamen, Ihre Matrikelnummer, Studiengang und Fachsemesterzahl auf dem Deckblatt der Klausur ein.
- Tragen Sie auf jedem Blatt Ihren Namen ein.
- Die Seiten 5, 6, 11, 12, 19 und 20 (Aufgabenstellungen der Programmieraufgaben und Manpages) dürfen herausgetrennt werden. Bitte schreiben Sie keine Lösungen auf herausgetrennte Seiten. Alle anderen Seiten dürfen **nicht** herausgetrennt werden.
- Um Unruhe und die Störung Ihrer Mitstudierenden zu vermeiden, ist die vorzeitige Abgabe der Klausur ausgeschlossen. Bleiben Sie an Ihrem Platz sitzen, bis am Ende alle Klausurunterlagen eingesammelt sind und die Aufsicht das Zeichen zum Gehen gibt.

Aufgabe 1: (14 Punkte)

In dieser Aufgabe sind jeweils m Aussagen angegeben. Davon sind n ($0 \leq n \leq m$) Aussagen richtig. Kreuzen Sie jeweils an, ob die entsprechende Aussage richtig oder falsch ist.
Jede korrekte Antwort gibt 0,5 Punkte, jede falsche Antwort 0,5 Punkte Abzug. Nicht beantwortete Aussagen gehen neutral in die Bewertung ein. Eine Teilaufgabe wird minimal mit 0 Punkten gewertet, falsche Antworten wirken sich nicht auf andere Teilaufgaben aus.

Wollen Sie eine falsch angekreuzte Antwort korrigieren, streichen Sie bitte das Kreuz mit drei waagrechten Strichen durch (☒).

Lesen Sie die Frage genau, bevor Sie antworten.

a) Bereich: Prozesszustände

2 Punkte

Richtig	Falsch	
☐	☐	Ein Prozess im Zustand „erzeugt“ kann sich durch Aufrufen des Systemaufrufs `exec()` in „bereit“ versetzen.
☐	☐	Übergang von „blockiert“ in „bereit“ bedeutet: Ein anderer Prozess wurde vom Betriebssystem verdrängt und der aktuelle Prozess wird nun auf der CPU eingelastet.
☐	☐	Ein Prozess kann sich selbst von „blockiert“ in „bereit“ überführen, wenn das Ereignis, auf das er wartet, eingetreten ist.
☐	☐	Prozesse können direkt von „blockiert“ in „laufend“ überführt werden.

b) Bewerten Sie die folgenden Aussagen zum Thema Betriebssysteme:

2 Punkte

Richtig	Falsch	
☐	☐	Mehrprogrammbetrieb ermöglicht die simultane Ausführung mehrerer Programme innerhalb eines Prozesses.
☐	☐	Betriebssystemdienste laufen zwingend im priviligierten Modus des Prozessors.
☐	☐	Ein Betriebssystem verteilt Betriebsmittel an sich bewerbende Nutzer.
☐	☐	Im Einprogrammbetrieb kann ein Mehrkernprozessor nicht ausgenutzt werden.

c) Bewerten Sie die folgenden Aussagen zu UNIX/Linux-Dateideskriptoren:

2 Punkte

Richtig	Falsch	
☐	☐	Das Verzeichnis `.` verweist auf das aktuelle Verzeichnis eines Prozesses. Das Verzeichnis `..` verweist auf dessen übergeordnetes Verzeichnis.
☐	☐	Ein Dateideskriptor ist eine prozesslokale Integerzahl, die der Prozess zum Zugriff auf eine geöffnete Datei benutzen kann.
☐	☐	Beim Aufruf von `fork()` werden zuvor geöffnete Dateideskriptoren in den Kindprozess vererbt.
☐	☐	Der Verzeichniseintrag `.` verweist auf das Verzeichnis, in dem der Eintrag selber steht. Der Verzeichniseintrag `..` verweist auf das im Dateibaum übergeordnete Verzeichnis.

d) Bereich: UNIX-Prozesse

2 Punkte

Richtig	Falsch	
☐	☐	Jedem UNIX-Prozess ist zu jeder Zeit ein aktuelles Arbeitsverzeichnis zugeordnet.
☐	☐	Das aktuelle Arbeitsverzeichnis erbt der Prozess vom aktuellen Benutzer.
☐	☐	Ein Prozess kann sich selbst vom Zustand „blockiert“ in „laufend“ überführen.
☐	☐	Mit dem Systemaufruf `chdir()` kann das aktuelle Arbeitsverzeichnis eines Prozesses durch seinen Elternprozess verändert werden.

e) Bewerten Sie die Aussagen zu folgendem C-Programmcode.

2 Punkte

```
int main(int argc, char *argv[]) {
    fork();
    fork();
    fork();
    printf("Hello");
}
```

Richtig	Falsch	
☐	☐	`fork()` ist konzeptionell ein Systemaufruf.
☐	☐	Es erscheint die Ausgabe „`HelloHelloHelloHelloHelloHelloHelloHello`“.
☐	☐	Nach der Ausführung existieren 5 Prozesse.
☐	☐	`fork()` hat drei verschiedene Gruppen von Rückgabewerten.

f) Bereich: Dateisysteme

2 Punkte

Richtig	Falsch	
☐	☐	Für jede reguläre Datei existiert mindestens ein Hardlink im selben Dateisystem.
☐	☐	Ein Hardlink kann auf andere Dateisysteme verweisen.
☐	☐	Ein Hardlink kann nur auf Verzeichnisse verweisen, nicht jedoch auf Dateien.
☐	☐	In einem UNIX-Dateisystem sind die Objekte immer in einer Baumstruktur angeordnet.

g) Man unterscheidet Traps und Interrupts (*Unterbrechungen*). Bewerten Sie die folgenden Aussagen:

2 Punkte

Richtig	Falsch	
☐	☐	Eine Instruktion hat einen Trap ausgelöst. Es ist möglich, dass der ausführende Prozess die Fehlerursache behebt und fortfährt.
☐	☐	Der Zeitgeber (Systemuhr) unterbricht die Programmbearbeitung in regelmäßigen Abständen. Die genaue Stelle der Unterbrechungen ist damit vorhersagbar. Somit sind solche Unterbrechungen in die Kategorie Trap einzuordnen.
☐	☐	Ein Interrupt wird immer unmittelbar durch eine Aktivität des aktuell laufenden Prozesses ausgelöst.
☐	☐	Der Zugriff auf eine virtuelle Adresse kann zu einem Trap führen.

Aufgabe 2: (11 Punkte)

Auf einer Party gibt es in einem Fass frisch gepressten Orangensaft. Wenn die Gastgeber den Saft nachfüllen, die Gäste ihn aber gleichzeitig abschöpfen, kommt es zu einer Konkurrenzsituation. Um diese zu vermeiden und allen einen schönen Abend zu gewährleisten, haben sich die Beteiligten darauf verständigt, die Abläufe zu koordinieren. Folgende Bedingungen sollen erfüllt werden:

- Gastgeber und Gäste dürfen nie gleichzeitig nachfüllen bzw. entnehmen.
- Es darf immer nur höchstens ein Gastgeber den Saft nachfüllen.
- Schöpft gerade ein Gast Saft ab, können beliebige weitere Gäste dazukommen und weggehen. Es gibt genug Schöpflöffel.
- Schöpft mindestens ein Gast Saft ab, so warten die Gastgeber mit Nachfüllen höflich, bis alle Gäste gegangen sind.

Ergänzen sie den C-Code so, dass das beschriebene Verhalten erreicht wird. Hierfür müssen Sie die gegebenen Semaphoren so initialisieren und verwenden, dass die Arbeitsabläufe passend synchronisiert werden. Dabei kann es sein, dass einzelnen Felder leer bleiben können. **Streichen Sie in diesem Fall das Feld durch!**

Hinweise:

- Zu Beginn laufen bereits mehrere Threads, die die Funktionen `guest()` und `host()` ausführen.
- Ergänzen Sie das folgende Codegerüst so, dass ein vollständig übersetzbares Programm entsteht.

Diese Seite darf herausgetrennt werden.

Diese Seite darf herausgetrennt werden.

NAME

sem_destroy – destroy a semaphore

SYNOPSIS

#include <semaphore.h>

int sem_destroy(sem_t ****sem*);**

DESCRIPTION

sem_destroy() destroys the semaphore at the address pointed to by *sem*.

Destroying a semaphore that other processes or threads are currently blocked on (in **sem_wait**(3)) produces undefined behavior.

Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized using **sem_init**(3).

RETURN VALUE

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

NAME

sem_getvalue – get the value of a semaphore

SYNOPSIS

#include <semaphore.h>

int sem_getvalue(sem_t ****sem*, int ****sval***);**

DESCRIPTION

sem_getvalue() places the current value of the semaphore pointed to *sem* into the integer pointed to by *sval*.

RETURN VALUE

sem_getvalue() returns 0 on success; on error, −1 is returned and *errno* is set appropriately.

NAME

sem_init – initialize an unnamed semaphore

SYNOPSIS

#include <semaphore.h>

int sem_init(sem_t ****sem*, int** *pshared*, **unsigned int** *value*);

DESCRIPTION

sem_init() initializes the unnamed semaphore at the address pointed to by *sem*. The *value* argument specifies the initial value for the semaphore.

The *pshared* argument indicates whether this semaphore is to be shared between the threads of a process (0), or between processes (1). Initializing a semaphore that has already been initialized results in undefined behavior.

RETURN VALUE

sem_init() returns 0 on success; on error, −1 is returned, and *errno* is set appropriately.

Diese Seite darf herausgetrennt werden.

NAME

sem_post – unlock a semaphore

SYNOPSIS

#include <semaphore.h>

int sem_post(sem_t ****sem*);**

DESCRIPTION

sem_post() increments (unlocks) the semaphore pointed to by *sem*. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a **sem_wait**(3) call will be woken up and proceed to lock the semaphore.

RETURN VALUE

sem_post() returns 0 on success; on error, the value of the semaphore is left unchanged, −1 is returned, and *errno* is set to indicate the error.

NAME

sem_wait, sem_timedwait – lock a semaphore

SYNOPSIS

#include <semaphore.h>

int sem_wait(sem_t ****sem*);**
int sem_trywait(sem_t ****sem*);**

DESCRIPTION

sem_wait() decrements (locks) the semaphore pointed to by *sem*. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

sem_trywait() is the same as **sem_wait**(), except that if the decrement cannot be immediately performed, then call returns an error (*errno* set to **EAGAIN**) instead of blocking.

RETURN VALUE

on success: 0; on error, the value of the semaphore is left unchanged, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINTR The call was interrupted by a signal handler
EINVAL *sem* is not a valid semaphore.
EAGAIN The operation could not be performed without blocking (sem_trywait() only).

---

11 Punkte

```
sem_t juice_mutex;
sem_t guest_mutex;
int juice_handling_guests;

void init(char *path) {
    [                                                    ];
    [                                                    ];
    [                                                              ];
}


void host(void) {
    while(1) {
        host_chill(); // unkritisches
        [                                                    ];
        host_juice_handle; // kritisch
        [                                                    ];
    }
}

void guest() {
    while(1) {
        [                                                ];
        juice_handling_guests++;
        if (juice_handling_guests == 1) {
            [                                                    ];
        }
        [                                                ];
        guest_juice_handle(); // kritisch
        [                                                ];
        juice_handling_guests--;
        if (juice_handling_guests == 0) {
            [                                                    ];
        }
        [                                                ];
        guest_chill(); // unkritisch
    }
}
```

Aufgabe 3: (18 Punkte)

a) Eine Freispeicherverwaltung gibt Speicher immer in Einheiten der nächstgrößeren Zweierpotenz der angefragten Menge heraus. Entsteht dadurch ein Problem? Wenn ja, welches? Begründen Sie stichwortartig.

2 Punkte

b) Beschreiben Sie was geschieht, wenn ein Prozess auf eine ausgelagerte Seite zugreift.

3 Punkte

c) Nehmen Sie an, dass das folgende Programmstück in einem Prozess eines UNIX-Systems ausgeführt wird:

3 Punkte

```
// ...
int *p = malloc(sizeof(int));
*p = -1;
// ...
```

I) Welcher Fehler kann dabei auftreten? (1 Punkt)

II) Der Fehler wird von einer Hardwarekomponente zuerst detektiert. Welche Komponente ist das? (1 Punkt)

III) Was wird das Betriebssystem mit dem Prozess machen, der den Fehler hervorruft? (1 Punkt)

d) Teilinterpretation: Wann wird das Betriebssystem aktiv? Nenne ein Beispiel für eine zu interpretierende Instruktion.

2 Punkte

e) Erläutern Sie das Konzept **Semaphor**. Welche Operationen sind auf Semaphore definiert und was tun diese Operationen?

6 Punkte

f) Worin unterscheiden sich Online- und Offlinescheduling? Nennen Sie je einen Vorteil.

2 Punkte

Aufgabe 4: (6 Punkte)

Gegeben sei ein Dateisystem mit indizierter Speicherung. Jeder Indexknoten enthält 7 direkte Verweise, und je einen einfach, zweifach und dreifach indirekten Verweis. Eine Adresse ist 4 Byte groß, ein Block 2 KiByte.

a) Wieviele Blöcke werden benötigt, um eine Datei der Größe 11 KiByte darzustellen? Wie werden die Datenblöcke adressiert?

2 Punkte

b) Wieviele Blöcke werden benötigt, um eine Datei der Größe 32 KiByte darzustellen? Wie werden die Datenblöcke adressiert?

2 Punkte

c) Wie groß kann eine Datei maximal in diesem Dateisystem sein?

2 Punkte

Aufgabe 5: (30 Punkte)

Auf einem Rechensystem befinden sich diverse Dateien und Verzeichnisse in einer Verzeichnisstruktur. Aus diesen sollen regulären Dateien herausgesucht werden, deren Name `good` enthält. Für jede gefundene Datei soll das Programm `horn` mit dem Dateinamen als Parameter aufgerufen werden. Alle anderen Dateien sollen ignoriert werden.

Aufrufsyntax: `wiesel DIRECTORY...`

Implementieren Sie das Programm `wiesel`, welches mit einer Liste von Verzeichnissen aufgerufen wird, die rekursiv nach Dateien durchsucht werden sollen. Zur Vereinfachung werden dem Programm `wiesel` stets absolute Pfade übergeben.

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

- Prüfen der Anzahl der Parameter
- Nutzungsausgabe, wenn die Anzahl der Parameter falsch ist
- Iterieren über die angegebenen Verzeichnisse. Für jedes Verzeichnis die Funktion `recurse(char *path)` aufrufen.

Funktion: `recurse(char *path)`

- Verzeichnisabstieg in das übergebene Verzeichnis
- Aufruf von `iterate_current_dir()`, um das betretene Verzeichnis zu durchlaufen.
- Rückkehr in das vorherige Verzeichnis

Funktion: `iterate_current_dir()`

- Iterieren über alle Einträge des aktuellen Verzeichnisses
- versteckte Dateien und Sicherungskopien (Dateiname beginnt mit `"."` oder `"~"`) sollen ignoriert werden
- reguläre Dateien an die Funktion `handle_file(char *filename)` übergeben
- in Verzeichnisse mittels `recurse(char *path)` absteigen

Funktion: `handle_file(char *filename)`

- Prüfen, ob der Dateiname `good` enthält
- Das Programm `horn` mit dem Dateinamen aufrufen
- Auf das Beenden des `horn`-Prozesses warten

Tritt ein nicht erwarteter Fehler auf, so ist das Programm mit Ausgabe einer Fehlermeldung zu beenden. Hierzu steht die Funktion `die(char *msg)` bereit.

Beispielfunktionsweise:

```
$ ls /foobar/
dir/ file  file1  file2  good  test
```

Aufruf und Standard-Out:

```
$ ./wiesel /foobar
```


Diese Seite darf herausgetrennt werden.

```c
#include <unistd.h>
#include <dirent.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/wait.h>

void die(const char *msg) {
  perror(msg);
  exit(1);
}


// Zu implementieren
void recurse(const char *path);
void handle_file(const char *filename);
void iterate_current_dir(void);
```


Diese Seite darf herausgetrennt werden.

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

M:

```
void recurse(const char *path) {
```

R:

```
void iterate_current_dir(void) {
```

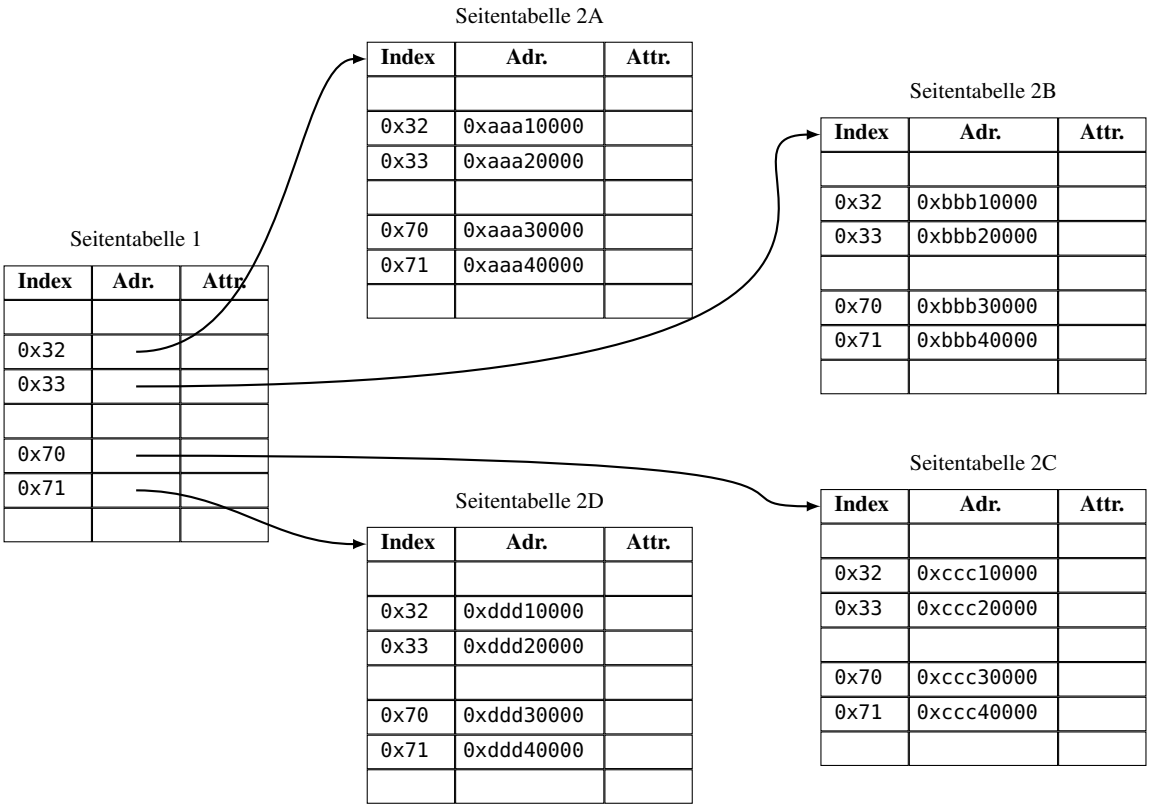
C:

```
void handle_file(const char *filename) {
```

H:

Aufgabe 6: (11 Punkte)

Auf einem byteweise adressierenden Mikrocontroller ist seitenorientierter logischer Adressraum mit zweistufiger Hierarchie implementiert. Die Adressen sind 32 Bit breit. Pro Stufe werden 8 Bit zur Indizierung verwendet. Die verbleibenden Bits werden als Offset in die Seite verwendet. Es sind außerdem 16 Bit für Attribute im Seitendeskriptor vorgesehen. Gegeben sei unten dargestellte Hierarchie zweistufiger Seitentabellen.



a) Bestimmen sie die **reale Adresse** zur logischen Adresse 0x73703233. Geben Sie hierbei die zur Bestimmung notwendigen Zwischenschritte stichpunktartig an!

4 Punkte

b) Nehmen Sie an, dass alle gültigen Seitentabelleneinträge oben abgebildet sind. Was würde in diesem Fall mit einem Prozess passieren, der schreibend auf die Adresse 0xFFFFFFFF zugreift und wieso?

2 Punkte

c) Bestimmen Sie die folgenden Größen:

3 Punkte

- Mindestanzahl Seitentabellen für einen Prozess mit ausführbarem Code und nicht ausführbaren Daten und nur lesbaren Konstanten
- Größe einer Seite
- maximale Größe des logischen Adressraums

d) Alternativ zu der gegebenen zweistufigen Implementierung könnte man auch eine einstufige Implementierung mit 16 Bit Seitennummern verwenden. Beschreiben sie einen Vor- und einen Nachteil dieser Alternative.

2 Punkte

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*.

RETURN VALUE

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

NAME

fork – create a child process

SYNOPSIS

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

pid_t fork(void);

DESCRIPTION

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

The child process is an exact duplicate of the parent process except for the following points:

* The child has its own unique process ID.
* The child's parent process ID is the same as the parent's process ID.

RETURN VALUE

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

NAME

execl, execlp, execv, execvp – execute a file

SYNOPSIS

#include <unistd.h>

int execl(const char * *pathname*, **const char *** *arg*, **... , NULL */);**
int execlp(const char * *file*, **const char *** *arg*, **... , NULL);**
int execv(const char * *pathname*, **char *const** *argv*[]);
int execvp(const char * *file*, **char *const** *argv*[]);

DESCRIPTION

The **exec**() family of functions replaces the current process image with a new process image.

The initial argument for these functions is the name of a file that is to be executed.

The functions can be grouped based on the letters following the "exec" prefix.

l - execl(), execlp()
The *const char *arg* and subsequent ellipses can be thought of as *arg0*, *arg1*, ..., *argn*. The list of arguments *must* be terminated by a null pointer.

By contrast with the 'l' functions, the 'v' functions (below) specify the command-line arguments of the executed program as a vector.

v - execv(), execvp()
The *char *const argv[]* argument is an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a null pointer.

p - execlp(), execvp()
These functions duplicate the actions of the shell in searching for an executable file if the specified filename does not contain a slash (/) character.

RETURN VALUE

The **exec**() functions return only if an error has occurred. The return value is −1, and *errno* is set to indicate the error.

NAME

opendir, fdopendir – open a directory

SYNOPSIS

DIR *opendir(const char * *name*****);**

DESCRIPTION

The **opendir**() function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream.

RETURN VALUE

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

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 pointed to by *dirp*. It returns NULL on reaching the end of the directory or if an error occurred.

The *dirent* structure is defined as follows:

```
struct dirent {
  ino_t          d_ino;      /* i-node number */
  unsigned short d_reclen;   /* Length of this record */
  unsigned char  d_type;     /* Type of file */
  char           d_name[256]; /* Null-terminated filename */ };
```

d_type This field indicates the file type:

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.

RETURN VALUE

On success, **readdir**() returns a pointer to a *dirent* structure.

If the end of the directory is reached, NULL is returned and *errno* is not changed. If an error occurs, NULL is returned and *errno* is set appropriately.

NAME

search and manipulate strings

SYNOPSIS

#include <string.h>

char *strstr(const char ****haystack*, const char ****needle***);**
size_t strlen(const char ****s* **);

DESCRIPTION

The **strstr**() function finds the first occurrence of the substring *needle* in the string *haystack*.

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

RETURN VALUE

The function **strstr()** returns a pointer to the beginning of the located substring, or NULL if the substring is not found.

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

NAME

wait, waitpid – wait for process to change state

SYNOPSIS

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

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

pid_t waitpid(pid_t *pid***, int ****wstatus***, int** *options***);**

DESCRIPTION

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below).

wait() and waitpid()

The **wait**() system call suspends execution of the calling thread until one of its children terminates. The call *wait(&wstatus)* is equivalent to:

waitpid(−1, &wstatus, 0);

The **waitpid**() system call suspends execution of the calling thread until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument, as described below.

The value of *pid* can be:

−1 meaning wait for any child process.

> 0 meaning wait for the child whose process ID is equal to the value of *pid*.

The value of *options* is an OR of zero or more of the following constants:

WNOHANG return immediately if no child has exited.

WUNTRACED
also return if a child has stopped (but not traced via **ptrace**(2)). Status for *traced* children which have stopped is provided even if this option is not specified.

WCONTINUED (since Linux 2.6.10)
also return if a stopped child has been resumed by delivery of **SIGCONT**.

If *wstatus* is not NULL, **wait**() and **waitpid**() store status information in the *int* to which it points. This integer can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it, as is done in **wait**() and **waitpid**()!):

WIFEXITED(*wstatus***)**
returns true if the child terminated normally, that is, by calling **exit**(3) or **_exit**(2), or by returning from main().

WEXITSTATUS(*wstatus***)**
returns the exit status of the child. This consists of the least significant 8 bits of the *status* argument that the child specified in a call to **exit**(3) or **_exit**(2) or as the argument for a return statement in main(). This macro should be employed only if **WIFEXITED** returned true.

RETURN VALUE

wait(): on success, returns the process ID of the terminated child; on error, −1 is returned. If no unwaited-for children exist, −1 is returned and *errno* is set to **ECHILD** ;

waitpid(): on success, returns the process ID of the child whose state has changed; if **WNOHANG** was specified and one or more child(ren) specified by *pid* exist, but have not yet changed state, then 0 is returned. On error, −1 is returned.

Each of these calls sets *errno* to an appropriate value in the case of an error.
