1 Prozesse und Scheduling (8 Punkte)

a) UNIX-Systemaufrufe (4 Punkte) Geben Sie die Ausgabe des folgenden C-Programms an.

Hinweis: Das Einbinden der Header-Dateien sowie ein Teil der Fehlerbehandlung wurden ausgelassen. Gehen Sie von einem fehlerfreien Ablauf aus. Das Symbol ␣ steht für ein Leerzeichen und soll nicht in die Antwortfelder eingetragen werden!

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int x = 1;
7
8  void foo() { printf("%d␣", ++x); x += 5; }
9
10 int main() {
11     foo();
12     fflush(NULL);
13     pid_t pid = fork();
14     if (pid > 0) {
15     wait(NULL);
16     foo();
17     int x = 0;
18     foo();
19     } else if (pid == 0) {
20         foo();
21     } else {
22         foo();
23         perror("Fehler!\n");
24         exit(-1);
25     }
26     return 0;
27 }
```

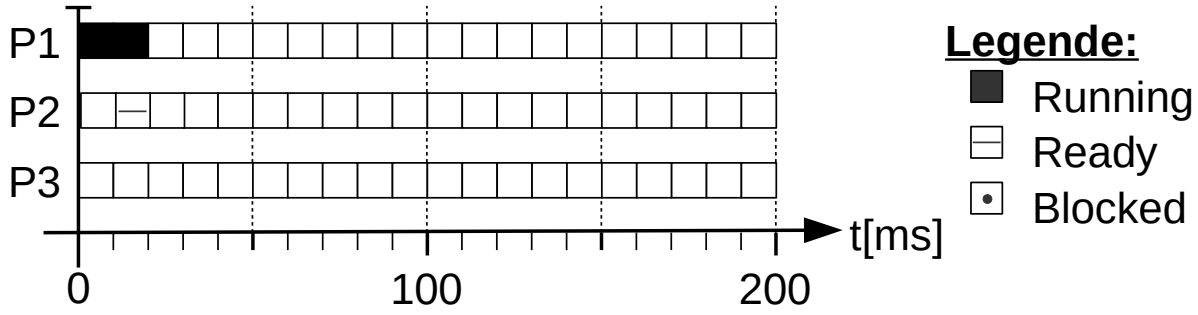
Ausgabe: | | | | |

b) Prozess-Scheduling (VRR) (4 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.

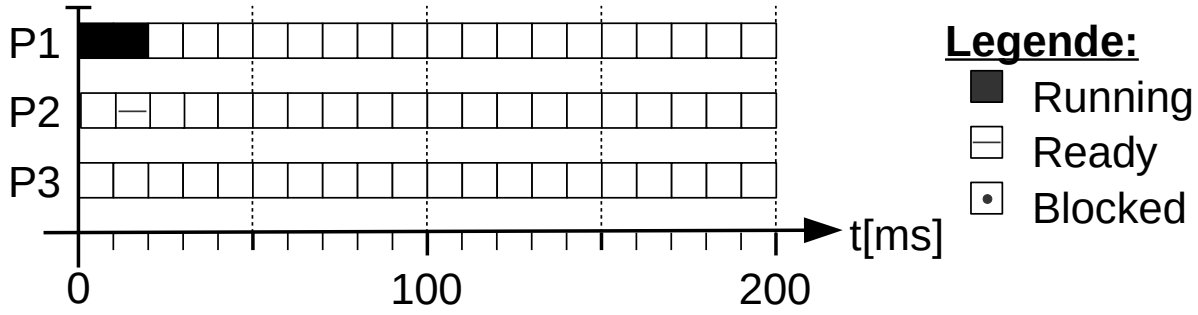
Prozess	Ankunftszeit	CPU-Zeit	E/A-Zeit
P1	0	20	20
P2	10	40	20
P3	30	40	30

Zeichnen Sie in das folgende Gantt-Diagramm ein, wie die drei Prozesse P1, P2 und P3 abgearbeitet werden, wenn das Scheduling nach der VRR-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 200ms anzugeben.

Hinweis: Die ersten zwei Zeiteinheiten sind bereits fertig ausgefüllt.



(Reserve)



2 Synchronisation und Verklemmungen (9 Punkte)

a) Synchronisierungsmuster (6 Punkte) Geben Sie für die nachfolgenden Interaktionen mit Semaphoren die entsprechenden Semaphor-Operationen (`P()` und `V()`) an und initialisieren Sie die Semaphore geeignet.

Gegenseitiger Ausschluss. Der kritische Abschnitt darf nicht mehrfach betreten werden.

```
/* gemeinsamer Speicher */
Semaphore S1 = [     ];
```

```
/* Prozess 1 */

// Kritischer Abschnitt
```

```
/* Prozess 2 */

// Kritischer Abschnitt
```

Betriebsmittelorientierte Synchronisierung. Das betroffene Betriebsmittel ist *fünf* mal vorhanden.

```
/* gemeinsamer Speicher */
Semaphore S2 = [     ];
```

```
/* Prozess 1 */

// Betriebsmittel verwenden
```

```
/* Prozess 2 */

// Betriebsmittel verwenden
```

Einseitige Synchronisierung mit gegenseitigem Ausschluss. Es darf nur dann ein Element aus der Warteschlange entnommen werden, wenn diese nicht leer ist. Zudem dürfen dequeue und enqueue nicht gleichzeitig ausgeführt werden. Gehen Sie davon aus, dass die Warteschlange Platz für beliebig viele Elemente bietet und vermeiden Sie Verklemmungen. Zu Beginn ist die Warteschlange leer.

```
/* gemeinsamer Speicher */
Semaphore S3 = [     ];
Semaphore S4 = [     ];
```

```
/* Prozess 1 */

item = dequeue(); // entnehmen
```

```
/* Prozess 2 */

enqueue(item); // zur Warteschlange hinzufügen
```

b) Verklemmungsbedingungen (3 Punkte) Nennen Sie die englischen Fachbegriffe der drei notwendigen Bedingungen, die eine Verklemmung ermöglichen.

1. _____

2. _____

3. _____

3 Speicherverwaltung und Virtueller Speicher (6 Punkte)

a) Freispeicherbelegung (3 Punkte)

In einem System mit Seitenadressierung und der Seitenersetzungsstrategie LRU (Least Recently Used) tätigt ein Prozess Seitenzugriffe entsprechend der unten gegebenen Referenzfolge.

Tragen Sie in die Tabelle jeweils die Nummer der Seite ein, die zum gegebenen Zeitpunkt in der jeweiligen Kachel eingelagert ist.

Registertabelle mit den eingelagerten Seiten

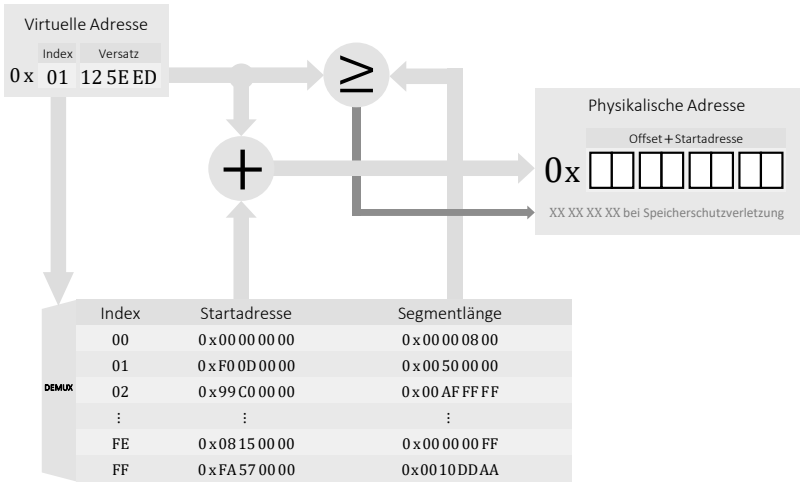
Referenzfolge:	1	5	3	3	6	5	7	5	5	3	5	3	7	2
Kachel 0:	**1**	1	1	1										
Kachel 1:	X	**5**	5	5										
Kachel 2:	X	X	**3**	**3**										
Kachel 3:	X	X	X	X										

Zugriffstabelle mit den Kontrollzuständen (Zahl der Zyklen seit des letzten Aufrufs)

Kachel 0:	**0**	1	2	3										
Kachel 1:	INT_MAX	**0**	1	2										
Kachel 2:	INT_MAX	INT_MAX	**0**	**0**										
Kachel 3:	INT_MAX	INT_MAX	INT_MAX	INT_MAX										

b) Segmentbasierte Addressberechnung (3 Punkte)

Geben Sie für die logische Adresse `0x01125EED` die gemäß der unten gegebenen Segmenttabelle die physikalische Adresse an. Falls die Anfrage eine Speicherschutzverletzung auslöst, tragen Sie `XXXXXXXX` in die Antwortfelder ein.



4 E/A und Dateisysteme (7 Punkte)

a) E/A-Scheduling (3 Punkte) Gegeben sei ein Plattenspeicher mit 16 Spuren. Der E/A-Scheduler bekommt immer wieder Aufträge für eine bestimmte Spur. Die Leseaufträge in L_0 sind dem E/A-Scheduler bereits bekannt. Nach drei bearbeiteten Aufträgen erhält er die Aufträge in L_3. Nach drei weiteren (d.h. nach insgesamt sechs) bearbeiteten Aufträgen erhält er die Aufträge in L_6. Zu Beginn befindet sich der Schreib-/Lesekopf über Spur 0.

$L_0 = \{6, 5, 3, 9\},\ L_3 = \{1, 8, 2\},\ L_6 = \{7, 5\}$

Tragen Sie hier die Reihenfolge der gelesenen Spuren für einen E/A-Scheduler ein, der nach der **Fahrstuhlstrategie** (**Aufzugalgorithmus** oder auch **SCAN**) arbeitet.

b) Geräteklassen (4 Punkte) Nennen Sie die aus der Vorlesung bekannten Geräteklassen von E/A-Geräten, und erläutern Sie kurz den Unterschied zwischen ihnen.

5 Programmieraufgabe (15 Punkte)

Implementieren Sie das Programm `multicat`, welches den Inhalt von einer oder mehreren Text-Dateien auf der Konsole ausgibt:

Verwendung: `multicat [file1] [file2] ...`

(Beispiel: `multicat foo.txt bar.txt`)

Funktionsweise:

- Die Ausführung startet nur bei korrekter Argumentenliste. Gehen Sie davon aus, dass es bei den übergebenen Argumenten um Textdateien handelt.
- `multicat` erzeugt für jeden Aufrufparameter einen eigenen Prozess, welcher den Inhalt der jeweiligen Datei auf der Standardausgabe ausgibt. Die Ausgaben dürfen sich nicht überlappen. Der Elternprozess wartet auf seine Kinder mittels `waitpid(2)`. Diese Funktionalität ist im wesentlichen in der Funktion `main` zu ergänzen.
- Jede Datei wird lesend geöffnet und dann mittels `fgetc(3)` zeichenweise eingelesen und auf der Konsole ausgegeben. Zusätzlich wird am Ende jeder erfolgreichen Ausgabe der Dateiname, die Anzahl der Leerzeichen sowie die Dateigröße in Byte (z.B. ‘‘`[file], spaces: [spaces], size: [size]\n`‘‘) ausgegeben und die Datei wieder ordnungsgemäß geschlossen. Fehler werden dem Elternprozess über den Exit-Status des Kindprozesses angezeigt. Diese Funktionalität ist im wesentlichen in der Funktion `parseFile` zu ergänzen. Nutzen Sie gerne die bereits vordeklarierten lokalen Variablen.

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

Rückgabewert von `parseFile(...)` und Exit-Status der Kindprozesse:

-2 Es ist ein Fehler in der Ausgabe der Datei aufgetreten

0 Die Ausgabe der Datei war erfolgreich

Rückgabewerte von `multicat`:

-2 Es ist ein Fehler in der Prozesshandhabung aufgetreten

0 Die Ausführung war erfolgreich (unabhängig von der Ausgabe der einzelnen Dateien)

Relevante Manual-Seiten: fopen, fgetc, fork, waitpid

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

u_int8_t parseFile(const char* path) {

    u_int32_t length = 0
    u_int32_t spaces = 0;
    char val;
    FILE* file;









    return 0;
}
```

```c
// Fortsetzung

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

    int child_exitstatus;









    return 0;
}
```

NAME

fgetc, fgets, getc, getchar, ungetc – input of characters and strings

LIBRARY

Standard C library (*libc*, *-lc*)

SYNOPSIS

```
#include <stdio.h>

int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);

char *fgets(char s[restrict .size], int size, FILE *restrict stream);

int ungetc(int c, FILE *stream);
```

DESCRIPTION

fgetc() reads the next character from *stream* and returns it as an *unsigned char* cast to an *int*, or **EOF** on end of file or error.

getc() is equivalent to **fgetc**() except that it may be implemented as a macro which evaluates *stream* more than once.

getchar() is equivalent to **getc**(*stdin*).

fgets() reads in at most one less than *size* characters from *stream* and stores them into the buffer pointed to by *s*. Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.

ungetc() pushes *c* back to *stream*, cast to *unsigned char*, where it is available for subsequent read operations. Pushed-back characters will be returned in reverse order; only one pushback is guaranteed.

Calls to the functions described here can be mixed with each other and with calls to other input functions from the *stdio* library for the same input stream.

For nonlocking counterparts, see **unlocked_stdio**(3).

RETURN VALUE

fgetc(), **getc**(), and **getchar**() return the character read as an *unsigned char* cast to an *int* or **EOF** on end of file or error.

fgets() returns *s* on success, and NULL on error or when end of file occurs while no characters have been read.

ungetc() returns *c* on success, or **EOF** on error.

ATTRIBUTES

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

Interface	Attribute	Value
fgetc(), **fgets**(), **getc**(), **getchar**(), **ungetc**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89.

NOTES

It is not advisable to mix calls to input functions from the *stdio* library with low-level calls to **read**(2) for the file descriptor associated with the input stream; the results will be undefined and very probably not what you want.

SEE ALSO

read(2), **write**(2), **ferror**(3), **fgetwc**(3), **fgetws**(3), **fopen**(3), **fread**(3), **fseek**(3), **getline**(3), **gets**(3), **getwchar**(3), **puts**(3), **scanf**(3), **ungetwc**(3), **unlocked_stdio**(3), **feature_test_macros**(7)

NAME

fopen, fdopen, fileno – stream open functions

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fildes, const char *mode);
int fileno(FILE *stream);
int fclose(FILE *stream);
```

DESCRIPTION

The **fopen** function opens the file whose name is the string pointed to by *path* and associates a stream with it.

The argument *mode* points to a string beginning with one of the following sequences (Additional characters may follow these sequences.):

r Open text file for reading. The stream is positioned at the beginning of the file.

r+ Open for reading and writing. The stream is positioned at the beginning of the file.

w Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

w+ Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

a Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

a+ Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

The **fdopen** function associates a stream with the existing file descriptor, *fildes*. The *mode* of the stream (one of the values "r", "r+", "w", "w+", "a", "a+") must be compatible with the mode of the file descriptor. The file position indicator of the new stream is set to that belonging to *fildes*, and the error and end-of-file indicators are cleared. Modes "w" or "w+" do not cause truncation of the file. The file descriptor is not dup'ed, and will be closed when the stream created by **fdopen** is closed. The result of applying **fdopen** to a shared memory object is undefined.

The function **fileno**() examines the argument *stream* and returns its integer descriptor.

The **fclose**() function flushes the stream pointed to by *stream* (writing any buffered output data using **fflush**(3)) and closes the underlying file descriptor.

RETURN VALUE

Upon successful completion **fopen**, **fdopen** and **freopen** return a **FILE** pointer. Otherwise, **NULL** is returned and the global variable *errno* is set to indicate the error. Upon successful completion of **fclose**, 0 is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error.

ERRORS

EINVAL

The *mode* provided to **fopen**, **fdopen**, or **freopen** was invalid.

EBADF

The file descriptor underlying *stream* passed to **fclose** is not valid.

The **fopen**, **fdopen** and **freopen** functions may also fail and set *errno* for any of the errors specified for the routine **malloc**(3).

The **fopen** function may also fail and set *errno* for any of the errors specified for the routine **open**(2).

The **fdopen** function may also fail and set *errno* for any of the errors specified for the routine **fcntl**(2).

NAME

fork – create a child process

SYNOPSIS

#include <unistd.h>

pid_t fork(void);

DESCRIPTION

fork() creates a new process by duplicating the calling process. The new process, referred to as the *child*, is an exact duplicate of the calling process, referred to as the *parent*, except for the following points:

* The child has its own unique process ID, and this PID does not match the ID of any existing process group (**setpgid**(2)).
* The child's parent process ID is the same as the parent's process ID.
* The child does not inherit its parent's memory locks (**mlock**(2), **mlockall**(2)).
* Process resource utilizations (**getrusage**(2)) and CPU time counters (**times**(2)) are reset to zero in the child.
* The child's set of pending signals is initially empty (**sigpending**(2)).
* The child does not inherit semaphore adjustments from its parent (**semop**(2)).
* The child does not inherit record locks from its parent (**fcntl**(2)).
* The child does not inherit timers from its parent (**setitimer**(2), **alarm**(2), **timer_create**(2)).
* The child does not inherit outstanding asynchronous I/O operations from its parent (**aio_read**(3), **aio_write**(3)), nor does it inherit any asynchronous I/O contexts from its parent (see **io_setup**(2)).

The process attributes in the preceding list are all specified in POSIX.1-2001. The parent and child also differ with respect to the following Linux-specific process attributes:

* The child does not inherit directory change notifications (dnotify) from its parent (see the description of **F_NOTIFY** in **fcntl**(2)).
* The **prctl**(2) **PR_SET_PDEATHSIG** setting is reset so that the child does not receive a signal when its parent terminates.
* Memory mappings that have been marked with the **madvise**(2) **MADV_DONTFORK** flag are not inherited across a **fork**().
* The termination signal of the child is always **SIGCHLD** (see **clone**(2)).

Note the following further points:

* The child process is created with a single thread — the one that called **fork**(). The entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects; the use of **pthread_atfork**(3) may be helpful for dealing with problems that this can cause.
* The child inherits copies of the parent's set of open file descriptors. Each file descriptor in the child refers to the same open file description (see **open**(2)) as the corresponding file descriptor in the parent. This means that the two descriptors share open file status flags, current file offset, and signal-driven I/O attributes (see the description of **F_SETOWN** and **F_SETSIG** in **fcntl**(2)).
* The child inherits copies of the parent's set of open message queue descriptors (see **mq_overview**(7)). Each descriptor in the child refers to the same open message queue description as the corresponding descriptor in the parent. This means that the two descriptors share the same flags (*mq_flags*).
* The child inherits copies of the parent's set of open directory streams (see **opendir**(3)). POSIX.1-2001 says that the corresponding directory streams in the parent and child *may* share the directory stream positioning; on Linux/glibc they do not.

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.

ERRORS

EAGAIN
fork() cannot allocate sufficient memory to copy the parent's page tables and allocate a task structure for the child.

EAGAIN
It was not possible to create a new process because the caller's **RLIMIT_NPROC** resource limit was encountered. To exceed this limit, the process must have either the **CAP_SYS_ADMIN** or the **CAP_SYS_RESOURCE** capability.

ENOMEM
fork() failed to allocate the necessary kernel structures because memory is tight.

CONFORMING TO

SVr4, 4.3BSD, POSIX.1-2001.

NOTES

Under Linux, **fork**() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

Since version 2.3.3, rather than invoking the kernel's **fork**() system call, the glibc **fork**() wrapper that is provided as part of the NPTL threading implementation invokes **clone**(2) with flags that provide the same effect as the traditional system call. The glibc wrapper invokes any fork handlers that have been established using **pthread_atfork**(3).

EXAMPLE

See **pipe**(2) and **wait**(2).

SEE ALSO

clone(2), **execve**(2), **setrlimit**(2), **unshare**(2), **vfork**(2), **wait**(2), **daemon**(3), **capabilities**(7), **credentials**(7)

COLOPHON

This page is part of release 3.27 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.

NAME

waitpid – wait for child process to change state

SYNOPSIS

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

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

DESCRIPTION

waitpid( ) suspends the calling process until one of its children changes state; if a child process changed state prior to the call to **waitpid( )**, return is immediate. *pid* specifies a set of child processes for which status is requested.

If *pid* is equal to **(pid_t)−1**, status is requested for any child process.

If *pid* is greater than **(pid_t)0**, it specifies the process ID of the child process for which status is requested.

If *pid* is equal to **(pid_t)0** status is requested for any child process whose process group ID is equal to that of the calling process.

If *pid* is less than **(pid_t)−1**, status is requested for any child process whose process group ID is equal to the absolute value of *pid*.

If **waitpid( )** returns because the status of a child process is available, then that status may be evaluated with the macros defined by **wstat**(5)**.** If the calling process had specified a non-zero value of *stat_loc*, the status of the child process will be stored in the location pointed to by *stat_loc*.

The *options* argument is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in the header **<sys/wait.h>**:

WCONTINUED The status of any continued child process specified by *pid*, whose status has not been reported since it continued, is also reported to the calling process.

WNOHANG **waitpid( )** will not suspend execution of the calling process if status is not immediately available for one of the child processes specified by *pid*.

WNOWAIT Keep the process whose status is returned in *stat_loc* in a waitable state. The process may be waited for again with identical results.

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.

WIFSIGNALED(*wstatus*)
returns true if the child process was terminated by a signal.

WTERMSIG(*wstatus*)
returns the number of the signal that caused the child process to terminate. This macro should be employed only if **WIFSIGNALED** returned true.

RETURN VALUES

If **waitpid( )** returns because the status of a child process is available, this function returns a value equal to the process ID of the child process for which status is reported. If **waitpid( )** returns due to the delivery of a signal to the calling process, **−1** is returned and **errno** is set to **EINTR**. If this function was invoked with

WNOHANG set in *options*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, **0** is returned. Otherwise, **−1** is returned, and **errno** is set to indicate the error.

ERRORS

waitpid( ) will fail if one or more of the following is true:

ECHILD The process or process group specified by *pid* does not exist or is not a child of the calling process or can never be in the states specified by *options*.

EINTR **waitpid( )** was interrupted due to the receipt of a signal sent by the calling process.

EINVAL An invalid value was specified for *options*.

SEE ALSO

exec(2), **exit**(2), **fork**(2), **sigaction**(2)