readline(3) - Linux man page

Name

readline - get a line from a user with editing

Synopsis

```
#include <stdio.h>
#include <readline/readline.h>
#include <readline/history.h>


char *
readline (const char *prompt);
```

Copyright

Description

readline will read a line from the terminal and return it, using **prompt** as a prompt.
If **prompt** is **NULL** or the empty string, no prompt is issued. The line returned is allocated with **malloc**(3); the caller must free it when finished. The line returned has the final newline removed, so only the text of the line remains.

readline offers editing capabilities while the user is entering the line. By default, the line editing commands are similar to those of emacs. A vi-style line editing interface is also available.

Function: void **rl_clear_history** (void)

Clear the history list by deleting all of the entries, in the same manner as the History library's clear_history() function. This differs from clear_history because it frees private data Readline saves in the history list.

Function: int **rl_on_new_line** (void)

Tell the update functions that we have moved onto a new (empty) line, usually after outputting a newline.

Function: void **rl_replace_line** (const char *text, int clear_undo) ¶

Replace the contents of rl_line_buffer with text. The point and mark are preserved, if possible. If clear_undo is non-zero, the undo list associated with the current line is cleared.

Variable: rl_voidfunc_t * **rl_redisplay_function**

If non-zero, Readline will call indirectly through this pointer to update the display with the current contents of the editing buffer. By default, it is set to rl_redisplay, the default Readline redisplay function (see Redisplay).

Function: void **add_history** (char *string)

Place string at the end of the history list. The associated data field (if any) is set to NULL.

signal(2) — Linux manual page

signal(2) System Calls Manual signal(2)

NAME top

signal - ANSI C signal handling

LIBRARY top

Standard C library (libc, -lc)

SYNOPSIS top

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

DESCRIPTION top

WARNING: the behavior of signal() varies across UNIX versions, and has also varied historically across different versions of Linux. Avoid its use: use sigaction(2) instead. See Portability below.

signal() sets the disposition of the signal signum to handler, which is either SIG_IGN, SIG_DFL, or the address of a programmer-defined function (a "signal handler").

If the signal signum is delivered to the process, then one of the following happens:

* If the disposition is set to SIG_IGN, then the signal is ignored.
* If the disposition is set to SIG_DFL, then the default action associated with the signal (see signal(7)) occurs.
* If the disposition is set to a function, then first either the disposition is reset to SIG_DFL, or the signal is blocked (see Portability below), and then handler is called with argument

signum. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from the handler.

The signals SIGKILL and SIGSTOP cannot be caught or ignored.

RETURN VALUE top

signal() returns the previous value of the signal handler. On failure, it returns SIG_ERR, and errno is set to indicate the error.

Changing Signal Dispositions: *sigaction()*

The *sigaction()* system call is an alternative to *signal()* for setting the disposition of a signal. Although *sigaction()* is somewhat more complex to use than *signal()*, in return it provides greater flexibility. In particular, *sigaction()* allows us to retrieve the disposition of a signal without changing it, and to set various attributes control- ling precisely what happens when a signal handler is invoked. Additionally, as we'll elaborate in Section 22.7, *sigaction()* is more portable than *signal()* when establish- ing a signal handler.

The *sig* argument identifies the signal whose disposition we want to retrieve or change. This argument can be any signal except SIGKILL or SIGSTOP.

The *act* argument is a pointer to a structure specifying a new disposition for the signal. If we are interested only in finding the existing disposition of the signal, then we can specify NULL for this argument. The *oldact* argument is a pointer to a structure of the same type, and is used to return information about the signal's pre- vious disposition. If we are not interested in this information, then we can specify NULL for this argument. The structures pointed to by *act* and *oldact* are of the follow- ing type:

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);
```

Returns 0 on success, or –1 on error


```
struct sigaction {
    void   (*sa_handler)(int);
    sigset_t sa_mask;
/* Address of handler */
/* Signals blocked during handler
invocation */
/* Flags controlling handler invocation */ /* Not for application use */

};

int void

  sa_flags;
(*sa_restorer)(void);
```

The *sigaction* structure is actually somewhat more complex than shown here. We consider further details in Section 21.4.

The *sa_handler* field corresponds to the *handler* argument given to *signal()*. It speci- fies the address of a signal handler, or one of the constants SIG_IGN or SIG_DFL. The *sa_mask* and *sa_flags* fields, which we discuss in a moment, are interpreted only if *sa_handler* is the address of a signal handler—that is, a value other than SIG_IGN or SIG_DFL. The remaining field, *sa_restorer*, is not intended for use in applications (and is not specified by SUSv3).

The *sa_restorer* field is used internally to ensure that on completion of a signal handler, a call is made to the special-purpose *sigreturn()* system call, which restores the process's execution context so that it can continue execution at the point where it was interrupted by the signal handler. An example of this usage can be found in the *glibc* source file sysdeps/unix/sysv/linux/i386/ sigaction.c.

The *sa_mask* field defines a set of signals that are to be blocked during invocation of the handler defined by *sa_handler*. When the signal handler is invoked, any signals in this set that are not currently part of the process signal mask are automatically added to the mask before the handler is called. These signals remain in the process signal mask until the signal handler returns, at which time they are automatically removed. The *sa_mask* field allows us to specify a set of signals that aren't permitted to interrupt execution of this handler. In addition, the signal that caused the han- dler to be invoked is automatically added to the process signal mask. This means that a signal handler won't recursively interrupt itself if a second instance of the same signal arrives while the handler is executing. Because blocked signals are not queued, if any of these signals are repeatedly generated during the execution of the handler, they are (later) delivered only once.

The *sa_flags* field is a bit mask specifying various options controlling how the signal is handled. The following bits may be ORed (|) together in this field:

SA_NOCLDSTOP
If *sig* is SIGCHLD, don't generate this signal when a child process is stopped or resumed as a consequence of receiving a signal. Refer to Section 26.3.2.

SA_NOCLDWAIT
(since Linux 2.6) If *sig* is SIGCHLD, don't transform children into zombies when they terminate. For further details, see Section 26.3.3.

SA_NODEFER

When this signal is caught, don't automatically add it to the process signal mask while the handler is executing. The name SA_NOMASK is provided as a historical synonym for SA_NODEFER, but the latter name is preferable because it is standardized in SUSv3.

SA_ONSTACK

Invoke the handler for this signal using an alternate stack installed by *sigaltstack()*. Refer to Section 21.3.

SA_RESETHAND
When this signal is caught, reset its disposition to the default (i.e., SIG_DFL) before invoking the handler. (By default, a signal handler remains estab- lished until it is explicitly disestablished by a further call to *sigaction()*.) The name SA_ONESHOT is provided as a historical synonym for SA_RESETHAND, but the latter name is preferable because it is standardized in SUSv3.

SA_RESTART

Automatically restart system calls interrupted by this signal handler. See Section 21.5.

SA_SIGINFO

Invoke the signal handler with additional arguments providing further information about the signal. We describe this flag in Section 21.4.

All of the above options are specified in SUSv3.
An example of the use of *sigaction()* is shown in Listing 21-1.

20.14 Waiting for a Signal: *pause()*

Calling *pause()* suspends execution of the process until the call is interrupted by a

signal handler (or until an unhandled signal terminates the process).

When a signal is handled, *pause()* is interrupted and always returns –1 with *errno* set to EINTR. (We say more about the EINTR error in Section 21.5.)

An example of the use of *pause()* is provided in Listing 20-2.

In Sections 22.9, 22.10, and 22.11, we look at various other ways that a pro- gram can suspend execution while waiting for a signal.

20.15 Summary

A signal is a notification that some kind of event has occurred, and may be sent to a process by the kernel, by another process, or by itself. There is a range of standard signal types, each of which has a unique number and purpose.

Signal delivery is typically asynchronous, meaning that the point at which the signal interrupts execution of the process is unpredictable. In some cases (e.g., hardware-generated signals), signals are delivered synchronously, meaning that delivery occurs predictably and reproducibly at a certain point in the execution of a program.

By default, a signal either is ignored, terminates a process (with or without a core dump), stops a running process, or restarts a stopped process. The particular default action depends on the signal type. Alternatively, a program can use *signal()* or *sigaction()* to explicitly ignore a signal or to establish a programmer-defined signal handler function that is invoked when the signal is delivered. For portability reasons, establishing a signal handler is best performed using *sigaction()*.

A process (with suitable permissions) can send a signal to another process using *kill()*. Sending the null signal (0) is a way of determining if a particular pro- cess ID is in use.

Each process has a signal mask, which is the set of signals whose delivery is cur- rently blocked. Signals can be added to and removed from the signal mask using *sigprocmask()*.

If a signal is received while it is blocked, then it remains pending until it is unblocked. Standard signals can't be queued; that is, a signal can be marked as pending (and thus later delivered) only once. A process can use the *sigpending()* sys- tem call to retrieve a signal set (a data structure used to represent multiple different signals) identifying the signals that it has pending.

#include <unistd.h> int **pause**(void);

Always returns –1 with *errno* set to EINTR

The *sigaction()* system call provides more control and flexibility than *signal()* when setting the disposition of a signal. First, we can specify a set of additional sig- nals to be blocked when a handler is invoked. In addition, various flags can be used to control the actions that occur when a signal handler is invoked. For example, there are flags that select the older unreliable signal semantics (not blocking the signal causing invocation of a handler, and having the disposition of the signal reset to its default before the handler is called).

Using *pause()*, a process can suspend execution until a signal arrives.

Further information

[Bovet & Cesati, 2005] and [Maxwell, 1999] provide background on the imple- mentation of signals in Linux. [Goodheart & Cox, 1994] details the implementa- tion of signals on System V Release 4. The GNU C library manual (available online at *http://www.gnu.org/*) contains an extensive description of signals.

Sending Signals: *kill()*

One process can send a signal to another process using the *kill()* system call, which is the analog of the *kill* shell command. (The term *kill* was chosen because the default action of most of the signals that were available on early UNIX implementa- tions was to terminate the process.)


```
#include <signal.h>
int kill(pid_t pid, int sig);
```

Returns 0 on success, or –1 on error

The *pid* argument identifies one or more processes to which the signal specified by *sig* is to be sent. Four different cases determine how *pid* is interpreted:

- If *pid* is greater than 0, the signal is sent to the process with the process ID spec- ified by *pid*.

- If *pid* equals 0, the signal is sent to every process in the same process group as the calling process, including the calling process itself. (SUSv3 states that the signal should be sent to all processes in the same process group, excluding an "unspecified set of system processes" and adds the same qualification to each of the remaining cases.)

- If *pid* is less than –1, the signal is sent to all of the processes in the process group whose ID equals the absolute value of *pid*. Sending a signal to all of the processes in a process group finds particular use in shell job control (Section 34.7).

- If *pid* equals –1, the signal is sent to every process for which the calling process has permission to send a signal, except *init* (process ID 1) and the calling pro- cess. If a privileged process makes this call, then all processes on the system will be signaled, except for these last two. For obvious reasons, signals sent in this way are sometimes called *broadcast signals*. (SUSv3 doesn't require that the call- ing process be excluded from receiving the signal; Linux follows the BSD semantics in this regard.)
  If no process matches the specified *pid*, *kill()* fails and sets *errno* to ESRCH ("No such process").
  A process needs appropriate permissions to be able send a signal to another process. The permission rules are as follows:

- ·A privileged (CAP_KILL) process may send a signal to any process.

- ·The *init* process (process ID 1), which runs with user and group of *root*, is a special case. It can be sent only signals for which it has a handler installed. This prevents the system administrator from accidentally killing *init*, which is funda- mental to the operation of the system.

- ·An unprivileged process can send a signal to another process if the real or effective user ID of the sending process matches the real user ID or saved set- user-ID of the receiving process, as shown in Figure 20-2. This rule allows users to send signals to set-user-ID programs that they have started, regardless of the current setting of the target process's effective user ID. Excluding the effective user ID of the target from the check serves a complementary purpose: it prevents one user from sending signals to another user's process that is running a set- user-ID program belonging to the user trying to send the signal. (SUSv3 mandates the rules shown in Figure 20-2, but Linux followed slightly different rules in kernel versions before 2.0, as described in the *kill(2)* manual page.)

The SIGCONT signal is treated specially. An unprivileged process may send this signal to any other process in the same session, regardless of user ID checks. This rule allows job-control shells to restart stopped jobs (process groups), even if the processes of the job have changed their user IDs (i.e., they are privi- leged processes that have used the system calls described in Section 9.7 to change their credentials).

Sending process Receiving process

indicates that if IDs match, then sender has permission to send a signal to receiver

Figure 20-2: Permissions required for an unprivileged process to send a signal

If a process doesn't have permissions to send a signal to the requested *pid*, then *kill()* fails, setting *errno* to EPERM. Where *pid* specifies a set of processes (i.e., *pid* is negative), *kill()* succeeds if at least one of them could be signaled.

We demonstrate the use of *kill()* in Listing 20-3.

20.6 Checking for the Existence of a Process

The *kill()* system call can serve another purpose. If the *sig* argument is specified as 0 (the so-called *null signal*), then no signal is sent. Instead, *kill()* merely performs error checking to see if the process can be signaled. Read another way, this means we can use the null signal to test if a process with a specific process ID exists. If sending a null signal fails

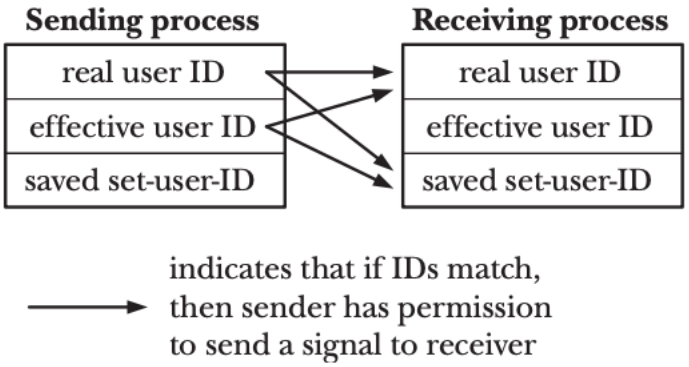
with the error ESRCH, then we know the process doesn't exist. If the call fails with the error EPERM (meaning the process exists, but we don't have permission to send a signal to it) or succeeds (meaning we do have permission to send a signal to the process), then we know that the process exists.

Verifying the existence of a particular process ID doesn't guarantee that a par- ticular program is still running. Because the kernel recycles process IDs as pro- cesses are born and die, the same process ID may, over time, refer to a different process. Furthermore, a particular process ID may exist, but be a zombie (i.e., a process that has died, but whose parent has not yet performed a *wait()* to obtain its termination status, as described in Section 26.2).

Various other techniques can also be used to check whether a particular pro- cess is running, including the following:

- ·*The wait() system calls*: These calls are described in Chapter 26. They can be employed only if the monitored process is a child of the caller.

- ·*Semaphores and exclusive file locks*: If the process that is being monitored contin- uously holds a semaphore or a file lock, then, if we can acquire the semaphore or lock, we know the process has terminated. We describe semaphores in Chapters 47 and 53 and file locks in Chapter 55.



- ·*IPC channels such as pipes and FIFOs*: We set up the monitored process so that it holds a file descriptor open for writing on the channel as long as it is alive. Meanwhile, the monitoring process holds open a read descriptor for the chan- nel, and it knows that the monitored process has terminated when the write end of the channel is closed (because it sees end-of-file). The monitoring pro- cess can determine this either by reading from its file descriptor or by monitor- ing the descriptor using one of the techniques described in Chapter 63.

- ·*The* /proc/*PID interface*: For example, if a process with the process ID 12345 exists, then the directory /proc/12345 will exist, and we can check this using a call such as *stat()*.

  All of these techniques, except the last, are unaffected by recycling of process IDs. Listing 20-3 demonstrates the use of *kill()*. This program takes two command- line arguments, a signal number and a process ID, and uses *kill()* to send the signal to the specified process. If signal 0 (the null signal) is specified, then the program
  reports on the existence of the target process.

20.7 Other Ways of Sending Signals: *raise()* and *killpg()*

Sometimes, it is useful for a process to send a signal to itself. (We see an example of

this in Section 34.7.3.) The *raise()* function performs this task.

In a single-threaded program, a call to *raise()* is equivalent to the following call to *kill()*:
kill(getpid(), sig);

On a system that supports threads, *raise(sig)* is implemented as: pthread_kill(pthread_self(), sig)

We describe the *pthread_kill()* function in Section 33.2.3, but for now it is sufficient to say that this implementation means that the signal will be delivered to the specific thread that called *raise()*. By contrast, the call *kill(getpid(), sig)* sends a signal to the calling *process*, and that signal may be delivered to any thread in the process.

The *raise()* function originates from C89. The C standards don't cover operating system details such as process IDs, but *raise()* can be specified within the C standard because it doesn't require reference to process IDs.

When a process sends itself a signal using *raise()* (or *kill()*), the signal is delivered immediately (i.e., before *raise()* returns to the caller).

Note that *raise()* returns a nonzero value (not necessarily –1) on error. The only error that can occur with *raise()* is EINVAL, because *sig* was invalid. Therefore, where we specify one of the SIGxxxx constants, we don't check the return status of this function.

#include <signal.h> int **raise**(int *sig*);

Returns 0 on success, or nonzero on error

Listing 20-3: Using the *kill()* system call

signals/t_kill.c

```
#include <signal.h>
#include "tlpi_hdr.h"
```

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

if (argc != 3 || strcmp(argv[1], "--help") == 0) usageErr("%s sig-num pid\n", argv[0]);

sig = getInt(argv[2], 0, "sig-num");

s = kill(getLong(argv[1], 0, "pid"), sig);

    if (sig != 0) {
        if (s == -1)
            errExit("kill");
} else { /* Null signal: process existence check */ if (s == 0) {

printf("Process exists and we can send it a signal\n"); } else {

if (errno == EPERM)
printf("Process exists, but we don't have "

                    "permission to send it a signal\n");
            else if (errno == ESRCH)
printf("Process does not exist\n"); else

                errExit("kill");
} }

    exit(EXIT_SUCCESS);
}
```

---

signals/t_kill.c The *killpg()* function sends a signal to all of the members of a process group.

A call to *killpg()* is equivalent to the following call to *kill()*: kill(-pgrp, sig);

If *pgrp* is specified as 0, then the signal is sent to all processes in the same process group as the caller. SUSv3 leaves this point unspecified, but most UNIX implemen- tations interpret this case in the same way as Linux.

Overview of *fork()*, *exit()*, *wait()*, and *execve()*

The principal topics of this and the next few chapters are the system calls *fork()*, *exit()*, *wait()*, and *execve()*. Each of these system calls has variants, which we'll also look at. For now, we provide an overview of these four system calls and how they are typically used together.

- ·The *fork()* system call allows one process, the parent, to create a new process, the child. This is done by making the new child process an (almost) exact dupli- cate of the parent: the child obtains copies of the parent's stack, data, heap, and text segments (Section 6.3). The term *fork* derives from the fact that we can envisage the parent process as dividing to yield two copies of itself.

- ·The *exit(status)* library function terminates a process, making all resources (memory, open file descriptors, and so on) used by the process available for subsequent reallocation by the kernel. The *status* argument is an integer that determines the termination status for the process. Using the *wait()* system call, the parent can retrieve this status.

The *exit()* library function is layered on top of the *_exit()* system call. In Chapter 25, we explain the difference between the two interfaces. In the meantime, we'll just note that, after a *fork()*, generally only one of the parent and child termi- nate by calling *exit()*; the other process should terminate using *_exit()*.

- ·The *wait(&status)* system call has two purposes. First, if a child of this process has not yet terminated by calling *exit()*, then *wait()* suspends execution of the process until one of its children has terminated. Second, the termination status of the child is returned in the status argument of *wait()*.

- ·The *execve(pathname, argv, envp)* system call loads a new program (*pathname*, with argument list *argv*, and environment list *envp*) into a process's memory. The existing program text is discarded, and the stack, data, and heap segments are freshly created for the new program. This operation is often referred to as *execing* a new program. Later, we'll see that several library functions are layered on top of *execve()*, each of which provides a useful variation in the program- ming interface. Where we don't care about these interface variations, we follow the common convention of referring to these calls generically as *exec()*, but be aware that there is no system call or library function with this name.
  Some other operating systems combine the functionality of *fork()* and *exec()* into a single operation—a so-called *spawn*—that creates a new process that then executes a specified program. By comparison, the UNIX approach is usually

simpler and more elegant. Separating these two steps makes the APIs simpler (the *fork()* system call takes *no* arguments) and allows a program a great degree of flexibility in the actions it performs between the two steps. Moreover, it is often useful to perform a *fork()* without a following *exec()*.

SUSv3 specifies the optional *posix_spawn()* function, which combines the effect of *fork()* and *exec()*. This function, and several related APIs specified by SUSv3, are implemented on Linux in *glibc*. SUSv3 specifies *posix_spawn()* to permit portable applications to be written for hardware architectures that don't pro- vide swap facilities or memory-management units (this is typical of many embedded systems). On such architectures, a traditional *fork()* is difficult or impossible to implement.

Figure 24-1 provides an overview of how *fork()*, *exit()*, *wait()*, and *execve()* are com- monly used together. (This diagram outlines the steps taken by the shell in executing a command: the shell continuously executes a loop that reads a command, performs various processing on it, and then forks a child process to exec the command.)

The use of *execve()* shown in this diagram is optional. Sometimes, it is instead useful to have the child carry on executing the same program as the parent. In either case, the execution of the child is ultimately terminated by a call to *exit()* (or by delivery of a signal), yielding a termination status that the parent can obtain via *wait()*.

The call to *wait()* is likewise optional. The parent can simply ignore its child and continue executing. However, we'll see later that the use of *wait()* is usually desirable, and is often employed within a handler for the SIGCHLD signal, which the kernel generates for a parent process when one of its children terminates. (By default, SIGCHLD is ignored, which is why we label it as being optionally delivered in the diagram.)

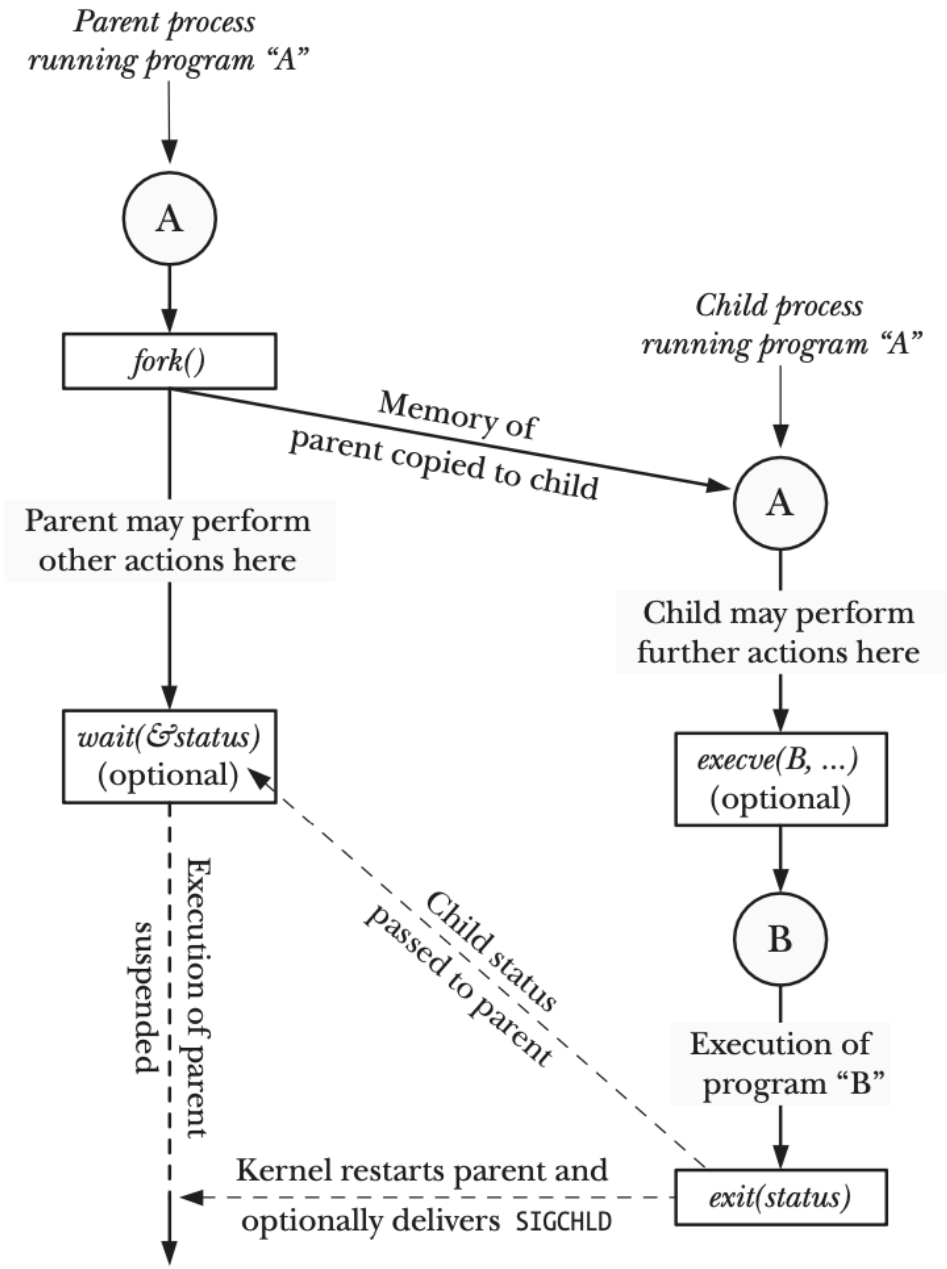


Figure 24-1: Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

24.2 Creating a New Process: *fork()*

In many applications, creating multiple processes can be a useful way of dividing up a task. For example, a network server process may listen for incoming client requests and create a new child process to handle each request; meanwhile, the server process continues to listen for further client connections. Dividing tasks up in this way often makes application design simpler. It also permits greater concur- rency (i.e., more tasks or requests can be handled simultaneously).

The *fork()* system call creates a new process, the *child*, which is an almost exact duplicate of the calling process, the *parent*.

#include <unistd.h> pid_t **fork**(void);

In parent: returns process ID of child on success, or –1 on error; in successfully created child: always returns 0

The key point to understanding *fork()* is to realize that after it has completed its work, two processes exist, and, in each process, execution continues from the point where *fork()* returns.

The two processes are executing the same program text, but they have separate copies of the stack, data, and heap segments. The child's stack, data, and heap seg- ments are initially exact duplicates of the corresponding parts the parent's memory. After the *fork()*, each process can modify the variables in its stack, data, and heap segments without affecting the other process.

Within the code of a program, we can distinguish the two processes via the value returned from *fork()*. For the parent, *fork()* returns the process ID of the newly created child. This is useful because the parent may create, and thus need to track, several children (via *wait()* or one of its relatives). For the child, *fork()* returns 0. If necessary, the child can obtain its own process ID using *getpid()*, and the process ID of its parent using *getppid()*.

If a new process can't be created, *fork()* returns –1. Possible reasons for failure are that the resource limit (RLIMIT_NPROC, described in Section 36.3) on the number of processes permitted to this (real) user ID has been exceeded or that the system- wide limit on the number of processes that can be created has been reached.

The following idiom is sometimes employed when calling *fork()*:

```
pid_t childPid;
switch (childPid = fork()) {
case -1:
   /* Handle error */
case 0:
   /* Perform actions
/* Used in parent after successful fork() to record PID of child */

/* fork() failed */
/* Child of successful fork() comes here */ specific to child */

/* Parent comes here after successful fork() */
```

It is important to realize that after a *fork()*, it is indeterminate which of the two processes is next scheduled to use the CPU. In poorly written programs, this indeter- minacy can lead to errors known as race conditions, which we describe further in Section 24.4.

Listing 24-1 demonstrates the use of *fork()*. This program creates a child that modifies the copies of global and automatic variables that it inherits during the during the *fork()*.

The use of *sleep()* (in the code executed by the parent) in this program permits the child to be scheduled for the CPU before the parent, so that the child can com- plete its work and terminate before the parent continues execution. Using *sleep()* in

```
default:
   /* Perform actions specific to parent */
}
```

this manner is not a foolproof method of guaranteeing this result; we look at a better method in Section 24.5.

When we run the program in Listing 24-1, we see the following output:

```
$ ./t_fork
PID=28557 (child) idata=333 istack=666 PID=28556 (parent) idata=111 istack=222
```

The above output demonstrates that the child process gets its own copy of the stack and data segments at the time of the *fork()*, and it is able to modify variables in these segments without affecting the parent.

Listing 24-1: Using *fork()*

procexec/t_fork.c

```
#include "tlpi_hdr.h"

static int idata = 111;
int
main(int argc, char *argv[])
{
   int istack = 222;
   pid_t childPid;
   switch (childPid = fork()) {
   case -1:
      errExit("fork");
   case 0:
      idata *= 3;
      istack *= 3;
      break;
   default:
      sleep(3);
break; }

/* Allocated in data segment */
/* Allocated in stack segment */
/* Give child a chance to execute */
/* Both parent and child come here */

printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(), (childPid == 0) ? "(child) " : "(parent)", idata,
istack);

            exit(EXIT_SUCCESS);
         }
```

procexec/t_fork.c

24.2.1 File Sharing Between Parent and Child

When a *fork()* is performed, the child receives duplicates of all of the parent's file descriptors. These duplicates are made in the manner of *dup()*, which means that corresponding descriptors in the parent and the child refer to the same open file description. As we saw in Section 5.4, the open file description contains the current

file offset (as modified by *read()*, *write()*, and *lseek()*) and the open file status flags (set by *open()* and changed by the *fcntl()* F_SETFL operation). Consequently, these attributes of an open file are shared between the parent and child. For example, if the child updates the file offset, this change is visible through the corresponding descriptor in the parent.

The fact that these attributes are shared by the parent and child after a *fork()* is demonstrated by the program in Listing 24-2. This program opens a temporary file using *mkstemp()*, and then calls *fork()* to create a child process. The child changes the file offset and open file status flags of the temporary file, and exits. The parent then retrieves the file offset and flags to verify that it can see the changes made by the child. When we run the program, we see the following:

```
$ ./fork_file_sharing
File offset before fork(): 0 O_APPEND flag before fork() is: off Child has exited
File offset in parent: 1000 O_APPEND flag in parent is: on
```

For an explanation of why we cast the return value from *lseek()* to *long long* in Listing 24-2, see Section 5.10.

Listing 24-2: Sharing of file offset and open file status flags between parent and child

procexec/fork_file_sharing.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/wait.h>
#include "tlpi_hdr.h"
int
main(int argc, char *argv[])
{
int fd, flags;
char template[] = "/tmp/testXXXXXX";

    setbuf(stdout, NULL);
    fd = mkstemp(template);
    if (fd == -1)
        errExit("mkstemp");
/* Disable buffering of stdout */
```

```
printf("File offset before fork(): %lld\n", (long long) lseek(fd, 0, SEEK_CUR));

flags = fcntl(fd, F_GETFL);
if (flags == -1)
errExit("fcntl - F_GETFL"); printf("O_APPEND flag before fork() is: %s\n",

        (flags & O_APPEND) ? "on" : "off");
switch (fork()) {
case -1:
    errExit("fork");
case 0: /* Child: change file offset and status flags */ if (lseek(fd, 1000, SEEK_SET) == -1)

    errExit("lseek");
flags = fcntl(fd, F_GETFL);
if (flags == -1)
    errExit("fcntl - F_GETFL");
flags |= O_APPEND;
if (fcntl(fd, F_SETFL, flags) == -1)
    errExit("fcntl - F_SETFL");
_exit(EXIT_SUCCESS);
/* Fetch current flags */
/* Turn O_APPEND on */
default: /* Parent: can see file changes made by child */ if (wait(NULL) == -1)

errExit("wait"); /* Wait for child exit */ printf("Child has exited\n");

printf("File offset in parent: %lld\n", (long long) lseek(fd, 0, SEEK_CUR));

        flags = fcntl(fd, F_GETFL);
        if (flags == -1)
errExit("fcntl - F_GETFL"); printf("O_APPEND flag in parent is: %s\n",

              (flags & O_APPEND) ? "on" : "off");
        exit(EXIT_SUCCESS);
} }
```

——————————————————————————————————————— procexec/fork_file_sharing.c

Sharing of open file attributes between the parent and child processes is frequently useful. For example, if the parent and child are both writing to a file, sharing the file offset ensures that the two processes don't overwrite each other's output. It does not, however, prevent the output of the two processes from being randomly intermingled. If this is not desired, then some form of process synchronization is required. For example, the parent can use the *wait()* system call to pause until the child has exited. This is what the shell does, so that it prints its prompt only after the child process executing a command has terminated (unless the user explicitly runs the command in the background by placing an ampersand character at the end of the command).

If sharing of file descriptors in this manner is not required, then an application should be designed so that, after a *fork()*, the parent and child use different file descriptors, with each process closing unused descriptors (i.e., those used by the other process)

immediately after forking. (If one of the processes performs an *exec()*, the close-on-exec flag described in Section 27.4 can also be useful.) These steps are shown in Figure 24-2.

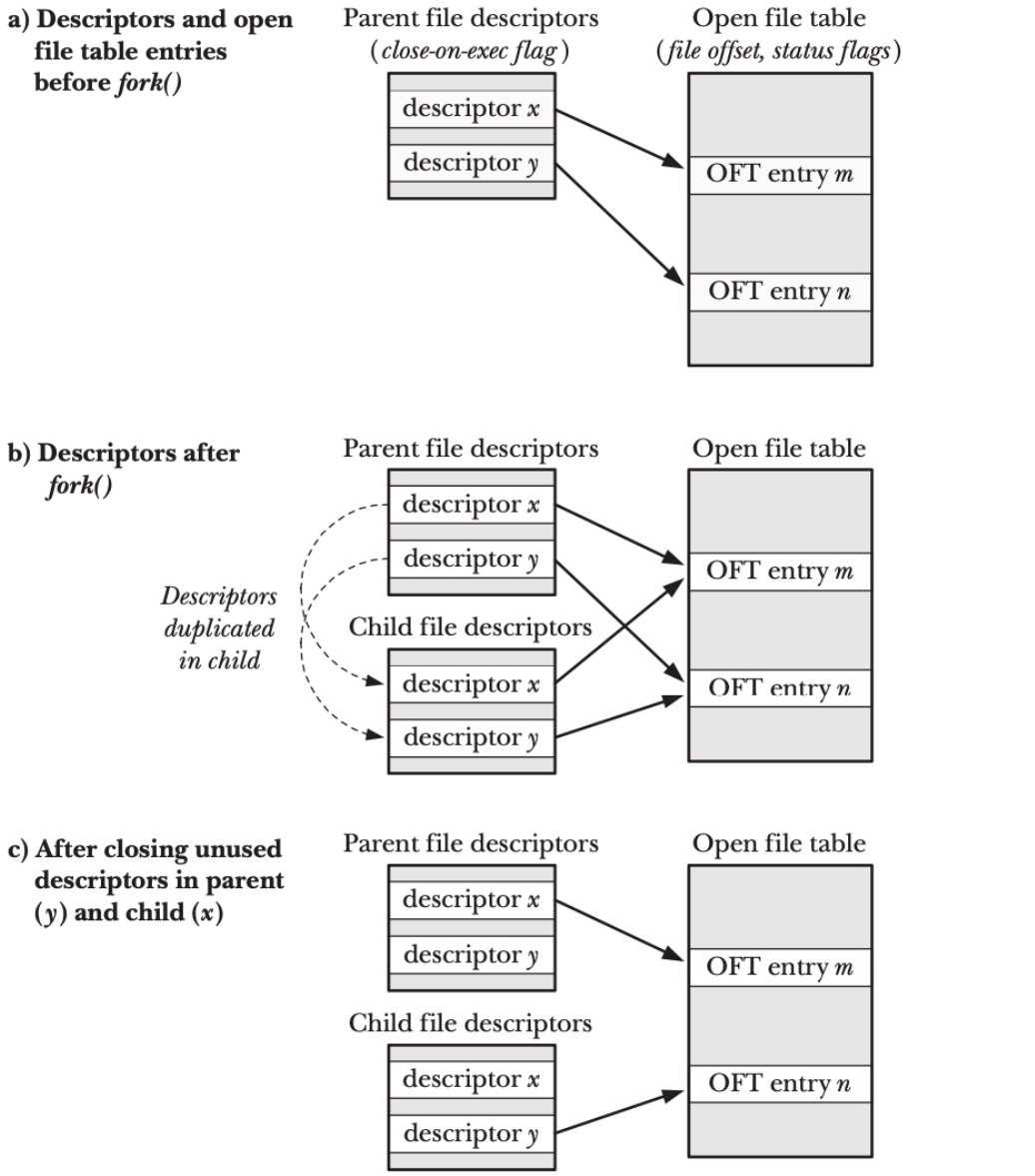


Figure 24-2: Duplication of file descriptors during *fork()*, and closing of unused descriptors

24.2.2 Memory Semantics of *fork()*

Conceptually, we can consider *fork()* as creating copies of the parent's text, data, heap, and stack segments. (Indeed, in some early UNIX implementations, such duplication was literally performed: a new process image was created by copying the parent's memory to swap space, and making that swapped-out image the child pro- cess while the parent kept its own memory.) However, actually performing a simple copy of the parent's virtual memory pages into the new child process would be wasteful for a number of reasons—one being that a *fork()* is often followed by an immediate *exec()*, which replaces the process's text with a new program and reinitializes

the process's data, heap, and stack segments. Most modern UNIX implementa- tions, including Linux, use two techniques to avoid such wasteful copying:

- ·The kernel marks the text segment of each process as read-only, so that a pro- cess can't modify its own code. This means that the parent and child can share the same text segment. The *fork()* system call creates a text segment for the child by building a set of per-process page-table entries that refer to the same virtual memory page frames already used by the parent.

- ·For the pages in the data, heap, and stack segments of the parent process, the kernel employs a technique known as *copy-on-write*. (The implementation of copy-on-write is described in [Bach, 1986] and [Bovet & Cesati, 2005].) Initially, the kernel sets things up so that the page-table entries for these segments refer to the same physical memory pages as the corresponding page-table entries in the parent, and the pages themselves are marked read-only. After the *fork()*, the kernel traps any attempts by either the parent or the child to modify one of these pages, and makes a duplicate copy of the about-to-be-modified page. This new page copy is assigned to the faulting process, and the corresponding page- table entry for the child is adjusted appropriately. From this point on, the parent and child can each modify their private copies of the page, without the changes being visible to the other process. Figure 24-3 illustrates the copy-on-write technique.

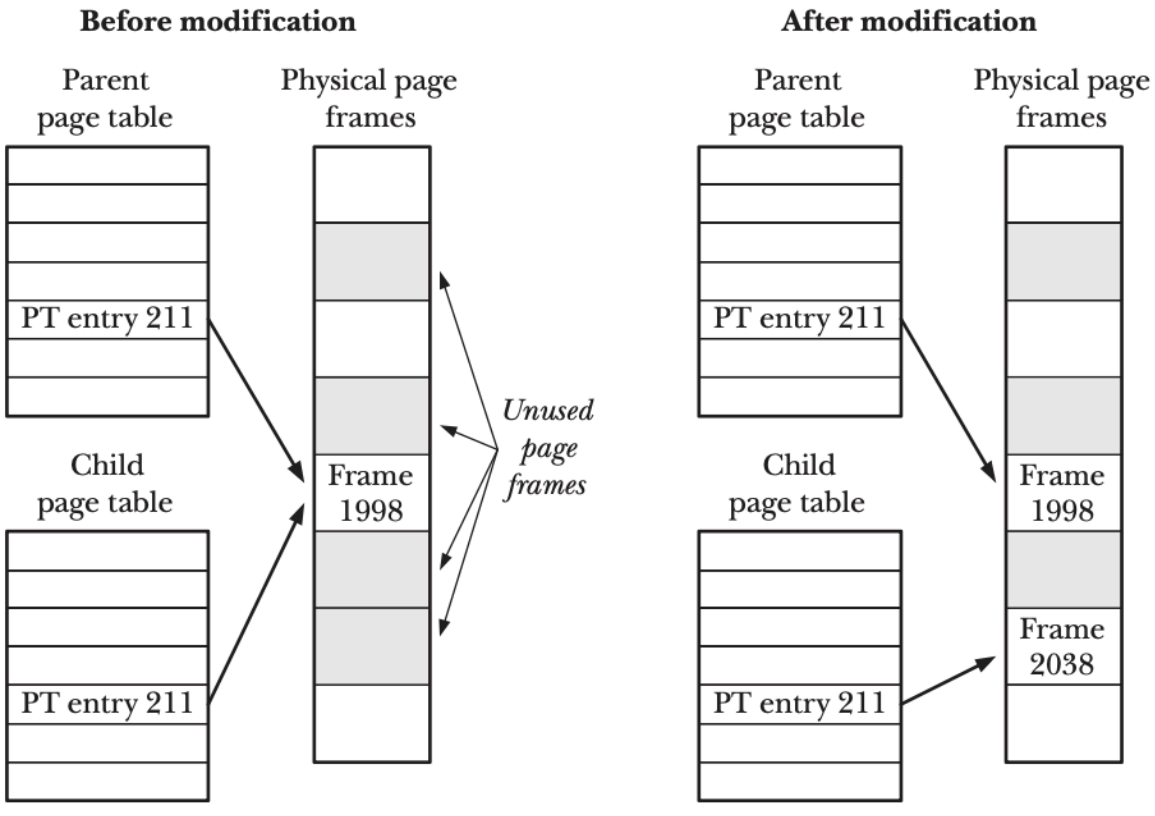


Figure 24-3: Page tables before and after modification of a shared copy-on-write page

Figure 24-3: Page tables before and after modification of a shared copy-on-write page

Controlling a process's memory footprint

We can combine the use of *fork()* and *wait()* to control the memory footprint of a process. The process's memory footprint is the range of virtual memory pages used by the process, as affected by factors such as the adjustment of the stack as functions

are called and return, calls to *exec()*, and, of particular interest to this discussion, modification of the heap as a consequence of calls to *malloc()* and *free()*.

Suppose that we bracket a call to some function, *func()*, using *fork()* and *wait()* in the manner shown in Listing 24-3. After executing this code, we know that the memory footprint of the parent is unchanged from the point before *func()* was called, since all possible changes will have occurred in the child process. This can be useful for the following reasons:

- ·If we know that *func()* causes memory leaks or excessive fragmentation of the heap, this technique eliminates the problem. (We might not otherwise be able to deal with these problems if we don't have access to the source code of *func()*.)

- ·Suppose that we have some algorithm that performs memory allocation while doing a tree analysis (for example, a game program that analyzes a range of possible moves and their responses). We could code such a program to make calls to *free()* to deallocate all of the allocated memory. However, in some cases, it is simpler to employ the technique we describe here in order to allow us to backtrack, leaving the caller (the parent) with its original memory footprint unchanged.
  In the implementation shown in Listing 24-3, the result of *func()* must be expressed in the 8 bits that *exit()* passes from the terminating child to the parent calling *wait()*. However, we could employ a file, a pipe, or some other interprocess communica- tion technique to allow *func()* to return larger results.

Listing 24-3: Calling a function without changing the process's memory footprint ———————————————————————
*from***procexec/footprint.c**

```
pid_t childPid;
int status;
childPid = fork();
if (childPid == -1)
    errExit("fork");
if (childPid == 0)
    exit(func(arg));
/* Child calls
/* uses return
func() and */
value as exit status */
determine the
```

```
/* Parent waits for child to terminate. It can result of func() by inspecting 'status'. */

if (wait(&status) == -1)
    errExit("wait");
```

getcwd(3) - Linux man page

Name

getcwd, getwd, get_current_dir_name - get current working directory

Synopsis

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);

char *getwd(char *buf);

char *get_current_dir_name(void);
```

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):
get_current_dir_name():
_GNU_SOURCE
getwd():
Since glibc 2.12:

```
_BSD_SOURCE ||
    (_XOPEN_SOURCE >= 500 ||
        _XOPEN_SOURCE && _XOPEN_SOURCE_EXTENDED) &&
    !(_POSIX_C_SOURCE >= 200809L || _XOPEN_SOURCE >= 700)
```

Before glibc 2.12:
_BSD_SOURCE || _XOPEN_SOURCE >= 500 || _XOPEN_SOURCE &&
_XOPEN_SOURCE_EXTENDED

Description

These functions return a null-terminated string containing an absolute pathname that is the current working directory of the calling process. The pathname is returned as the function result and via the argument *buf*, if present.

The **getcwd**() function copies an absolute pathname of the current working directory to the array pointed to by *buf*, which is of length *size*.

If the length of the absolute pathname of the current working directory, including the terminating null byte, exceeds *size* bytes, NULL is returned, and *errno* is set to **ERANGE**; an application should check for this error, and allocate a larger buffer if necessary.

As an extension to the POSIX.1-2001 standard, Linux (libc4, libc5, glibc) **getcwd**() allocates the buffer dynamically using **malloc**(3) if *buf* is NULL. In this case, the allocated buffer has the length *size* unless *size* is zero, when *buf* is allocated as big as necessary. The caller should **free**(3) the returned buffer.

get_current_dir_name() will **malloc**(3) an array big enough to hold the absolute pathname of the current working directory. If the environment variable **PWD** is set, and its value is correct, then that value will be returned. The caller should **free**(3) the returned buffer.

getwd() does not **malloc**(3) any memory. The *buf* argument should be a pointer to an array at least **PATH_MAX** bytes long. If the length of the absolute pathname of the current working directory, including the terminating null byte, exceeds **PATH_MAX** bytes, NULL is returned, and *errno* is set to **ENAMETOOLONG**. (Note that on some systems, **PATH_MAX** may not be a compile-time constant; furthermore, its value may depend on the file system, see **pathconf**(3).) For portability and security reasons, use of **getwd**() is deprecated.

Return Value

On success, these functions return a pointer to a string containing the pathname of the current working directory. In the case **getcwd**() and **getwd**() this is the same value as *buf*.

On failure, these functions return NULL, and *errno* is set to indicate the error. The contents of the array pointed to by *buf* are undefined on error.

chdir(3) - Linux man page

Prolog

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.
Name

chdir - change working directory
Synopsis

#include <unistd.h>

int chdir(const char ***path***);**

Description

The *chdir*() function shall cause the directory named by the pathname pointed to by the *path* argument to become the current working directory; that is, the starting point for path searches for pathnames not beginning with **'/'** .

Return Value

Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned, the current working directory shall remain unchanged, and *errno* shall be set to indicate the error.

stat(2) - Linux man page

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 || _XOPEN_SOURCE &&
_XOPEN_SOURCE_EXTENDED
|| /* Since glibc 2.10: */ _POSIX_C_SOURCE >= 200112L
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 512B 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 **major**(3) and **minor**(3) macros may be useful to decompose the device ID in this field.)
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 symbolic link is the length of the pathname it contains, without a terminating 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 and/or directory accesses do not cause an update of the *st_atime* field. (See *noatime*, *nodiratime*, and *relatime* in **mount**(8), and related information in **mount**(2).) In addition, *st_atime* is not updated if a file is opened with the **O_NOATIME**; see **open**(2).

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.)

The following flags are defined for the *st_mode* field:
The set-group-ID bit (**S_ISGID**) has several special uses. For a directory it indicates that BSD semantics is to be used for that directory: files created there inherit their group ID from the directory, not from the effective group ID of the creating process, and directories created there will also get the **S_ISGID** bit set. For a file that does not have the group execution bit (**S_IXGRP**) set, the set-group-ID bit indicates mandatory file/record locking.
The sticky bit (**S_ISVTX**) on a directory means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, and by a privileged process.

Return Value

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

opendir(3) - Linux man page

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:
_XOPEN_SOURCE >= 700 || _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 appropriately.

readdir(3) - Linux man page

Name

readdir, readdir_r - read a directory

Synopsis

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);

int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
```

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):
readdir_r():
_POSIX_C_SOURCE >= 1 || _XOPEN_SOURCE || _BSD_SOURCE || _SVID_SOURCE ||
_POSIX_SOURCE

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.

On Linux, the *dirent* structure is defined as follows:

```
struct dirent {
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* offset to the next dirent */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file; not supported
                                   by all file system types */
    char           d_name[256]; /* filename */
};
```

The only fields in the *dirent* structure that are mandated by POSIX.1 are: *d_name*[], of unspecified size, with at most **NAME_MAX** characters preceding the terminating null byte; and (as an XSI extension) *d_ino*. The other fields are unstandardized, and not present on all systems; see NOTES below for some further details.
The data returned by **readdir**() may be overwritten by subsequent calls to **readdir**() for the same directory stream.

The **readdir_r**() function is a reentrant version of **readdir**(). It reads the next directory entry from the directory stream *dirp*, and returns it in the caller-allocated buffer pointed to by *entry*. (See NOTES for information on allocating this buffer.) A pointer to the returned item is placed in **result*; if the end of the directory stream was encountered, then NULL is instead returned in **result*.

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.

The **readdir_r**() function returns 0 on success. On error, it returns a positive error number (listed under ERRORS). If the end of the directory stream is reached, **readdir_r**() returns 0, and returns NULL in **result*.

closedir(3) - Linux man page

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 appropriately.

isatty(3) - Linux man page

Name

isatty - test whether a file descriptor refers to a terminal
Synopsis

```
#include <unistd.h>
int isatty(int fd);
```
Description

The **isatty**() function tests whether *fd* is an open file descriptor referring to a terminal.
Return Value

isatty() returns 1 if *fd* is an open file descriptor referring to a terminal; otherwise 0 is returned, and *errno* is set to indicate the error.

ttyname(3) - Linux man page

Name

ttyname, ttyname_r - return name of a terminal
Synopsis

```
#include <unistd.h>
char *ttyname(int fd);int ttyname_r(int fd, char *buf, size_t buflen);
```
Description

The function **ttyname**() returns a pointer to the null-terminated pathname of the terminal device that is open on the file descriptor *fd*, or NULL on error (for example, if *fd* is not connected to a terminal). The return value may point to static data, possibly overwritten by the next call. The function **ttyname_r**() stores this pathname in the buffer *buf* of length *buflen*.
Return Value

The function **ttyname**() returns a pointer to a pathname on success. On error, NULL is returned, and *errno* is set appropriately. The function **ttyname_r**() returns 0 on success, and an error number upon error.

ttyslot(3) - Linux man page

Name

ttyslot - find the slot of the current user's terminal in some file

Synopsis

#include <unistd.h> /* on BSD-like systems, and Linux */
#include <stdlib.h> /* on System V-like systems */

int ttyslot(void);

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

ttyslot():

_BSD_SOURCE ||
_XOPEN_SOURCE && _XOPEN_SOURCE_ < 500 && _XOPEN_SOURCE_EXTENDED
Description

The legacy function **ttyslot**() returns the index of the current user's entry in some file.

Now "What file?" you ask. Well, let's first look at some history.

Ancient history

There used to be a file */etc/ttys* in UNIX V6, that was read by the **init**(8) program to find out what to do with each terminal line. Each line consisted of three characters. The first character was either '0' or '1', where '0' meant "ignore". The second character denoted the terminal: '8' stood for "/dev/tty8". The third character was an argument to **getty**(8) indicating the sequence of line speeds to try ('-' was: start trying 110 baud). Thus a typical line was "18-". A hang on some line was solved by changing the '1' to a '0', signaling init, changing back again, and signaling init again.
In UNIX V7 the format was changed: here the second character was the argument to **getty**(8) indicating the sequence of line speeds to try ('0' was: cycle through 300-1200-150-110 baud; '4' was for the on-line console DECwriter) while the rest of the line contained the name of the tty. Thus a typical line was "14console".

Later systems have more elaborate syntax. System V-like systems have */etc/inittab* instead.

Ancient history (2)

On the other hand, there is the file */etc/utmp* listing the people currently logged in. It is maintained by **login**(1). It has a fixed size, and the appropriate index in the file was determined by **login**(1) using the **ttyslot**() call to find the number of the line in */etc/ttys* (counting from 1).
The semantics of ttyslot

Thus, the function **ttyslot**() returns the index of the controlling terminal of the calling process in the file */etc/ttys*, and that is (usually) the same as the index of the entry for the current user in the file */etc/utmp*. BSD still has the */etc/ttys* file, but System V-like systems do not, and hence cannot refer to it. Thus, on such systems the documentation says that **ttyslot**() returns the current user's index in the user accounting data base.
Return Value

If successful, this function returns the slot number. On error (e.g., if none of the file descriptors 0, 1 or 2 is associated with a terminal that occurs in this data base) it returns 0 on UNIX V6 and V7 and BSD-like systems, but -1 on System V-like systems.

ioctl(2) - Linux man page

Name

ioctl - control device
Synopsis

#include <sys/ioctl.h>
int ioctl(int *d*, **int** *request*, **...);**

Description

The **ioctl**() function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with **ioctl**() requests. The argument *d* must be an open file descriptor.
The second argument is a device-dependent request code. The third argument is an untyped pointer to memory. It's traditionally **char ***argp* (from the days before **void *** was valid C), and will be so named for this discussion.

An **ioctl**() *request* has encoded in it whether the argument is an *in* parameter or *out* parameter, and the size of the argument *argp* in bytes. Macros and defines used in specifying an **ioctl**() *request* are located in the file <*sys/ioctl.h*>.

Return Value

Usually, on success zero is returned. A few **ioctl**() requests use the return value as an output parameter and return a nonnegative value on success. On error, -1 is returned, and *errno* is set appropriately.

getenv(3) - Linux man page

Name

getenv, secure_getenv - get an environment variable

Synopsis

```
#include <stdlib.h>

char *getenv(const char *name);

char *secure_getenv(const char *name);
```

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

The **getenv**() function searches the environment list to find the environment variable *name*, and returns a pointer to the corresponding *value* string.

The GNU-specific **secure_getenv**() function is just like **getenv**() except that it returns NULL in cases where "secure execution" is required. Secure execution is required if one of the following conditions was true when the program run by the calling process was loaded:

> the process's effective user ID did not match its real user ID or the process's effective group ID did not match its real group ID (typically this is the result of executing a set-user-ID or set-group-ID program);
>
> the effective capability bit was set on the executable file; or
>
> the process has a nonempty permitted capability set.

Secure execution may also required if triggered by some Linux security modules.
The **secure_getenv**() function is intended for use in general-purpose libraries to avoid vulnerabilities that could occur if set-user-ID or set-group-ID programs accidentally trusted the environment.

Return Value

The **getenv**() function returns a pointer to the value in the environment, or NULL if there is no match.

tcsetattr(3) - Linux man page

Name

tcgetattr, tcsetattr

Synopsis

```
#include <termios.h>
#include <unistd.h>

int tcgetattr(int fd, struct termios *termios_p);

int tcsetattr(int fd, int optional_actions,
              const struct termios *termios_p);
```

The termios functions describe a general terminal interface that is provided to control asynchronous communications ports.

The termios structure

Many of the functions described here have a *termios_p* argument that is a pointer to a *termios* structure. This structure contains at least the following members:

```
tcflag_t c_iflag;      /* input modes */
tcflag_t c_oflag;      /* output modes */
tcflag_t c_cflag;      /* control modes */
tcflag_t c_lflag;      /* local modes */
cc_t     c_cc[NCCS];   /* special characters */
```

The values that may be assigned to these fields are described below. In the case of the first four bit-mask fields, the definitions of some of the associated flags that may be set are only exposed if a specific feature test macro (see **feature_test_macros**(7)) is defined, as noted in brackets ("[]").
In the descriptions below, "not in POSIX" means that the value is not specified in POSIX.1-2001, and "XSI" means that the value is specified in POSIX.1-2001 as part of the XSI extension.

c_iflag flag constants:

IGNBRK
Ignore BREAK condition on input.

BRKINT

If **IGNBRK** is set, a BREAK is ignored. If it is not set but **BRKINT** is set, then a BREAK causes the input and output queues to be flushed, and if the terminal is the controlling terminal of a foreground process group, it will cause a **SIGINT** to be sent to this foreground process group. When neither **IGNBRK** nor **BRKINT** are set, a BREAK reads as a null byte ('\0'), except when **PARMRK** is set, in which case it reads as the sequence \377 \0 \0.

IGNPAR

Ignore framing errors and parity errors.

PARMRK

If **IGNPAR** is not set, prefix a character with a parity error or framing error with \377 \0. If neither **IGNPAR** nor **PARMRK** is set, read a character with a parity error or framing error as \0.

INPCK

Enable input parity checking.

ISTRIP

Strip off eighth bit.

INLCR

Translate NL to CR on input.

IGNCR

Ignore carriage return on input.

ICRNL

Translate carriage return to newline on input (unless **IGNCR** is set).

IUCLC

(not in POSIX) Map uppercase characters to lowercase on input.

IXON

Enable XON/XOFF flow control on output.

IXANY

(XSI) Typing any character will restart stopped output. (The default is to allow just the START character to restart output.)

IXOFF

Enable XON/XOFF flow control on input.

IMAXBEL
(not in POSIX) Ring bell when input queue is full. Linux does not implement this bit, and acts as if it is always set.
IUTF8 (since Linux 2.6.4)
(not in POSIX) Input is UTF8; this allows character-erase to be correctly performed in cooked mode.
c_oflag flag constants defined in POSIX.1:
OPOST
Enable implementation-defined output processing.

The remaining *c_oflag* flag constants are defined in POSIX.1-2001, unless marked otherwise.
OLCUC
(not in POSIX) Map lowercase characters to uppercase on output.

ONLCR

(XSI) Map NL to CR-NL on output.

OCRNL

Map CR to NL on output.

ONOCR

Don't output CR at column 0.

ONLRET

Don't output CR.

OFILL

Send fill characters for a delay, rather than using a timed delay.

OFDEL

(not in POSIX) Fill character is ASCII DEL (0177). If unset, fill character is ASCII NUL ('\0').
(Not implemented on Linux.)

NLDLY

Newline delay mask. Values are **NL0** and **NL1**.
[requires **_BSD_SOURCE** or **_SVID_SOURCE** or **_XOPEN_SOURCE**]

CRDLY

Carriage return delay mask. Values are **CR0**, **CR1**, **CR2**, or **CR3**.
[requires **_BSD_SOURCE** or **_SVID_SOURCE** or **_XOPEN_SOURCE**]

TABDLY

Horizontal tab delay mask. Values are **TAB0**, **TAB1**, **TAB2**, **TAB3** (or **XTABS**). A value of TAB3, that is, XTABS, expands tabs to spaces (with tab stops every eight columns).
[requires **_BSD_SOURCE** or **_SVID_SOURCE** or **_XOPEN_SOURCE**]

BSDLY

Backspace delay mask. Values are **BS0** or **BS1**. (Has never been implemented.)
[requires **_BSD_SOURCE** or **_SVID_SOURCE** or **_XOPEN_SOURCE**]

VTDLY

Vertical tab delay mask. Values are **VT0** or **VT1**.

FFDLY

Form feed delay mask. Values are **FF0** or **FF1**.
[requires **_BSD_SOURCE** or **_SVID_SOURCE** or **_XOPEN_SOURCE**]

c_cflag flag constants:
CBAUD
(not in POSIX) Baud speed mask (4+1 bits). [requires **_BSD_SOURCE** or **_SVID_SOURCE**]

CBAUDEX
(not in POSIX) Extra baud speed mask (1 bit), included in **CBAUD**.
[requires **_BSD_SOURCE** or **_SVID_SOURCE**]
(POSIX says that the baud speed is stored in the *termios* structure without specifying where precisely, and provides **cfgetispeed**() and **cfsetispeed**() for getting at it. Some systems use bits selected by **CBAUD** in *c_cflag*, other systems use separate fields, for example, *sg_ispeed* and *sg_ospeed*.)

CSIZE
Character size mask. Values are **CS5**, **CS6**, **CS7**, or **CS8**.

CSTOPB

Set two stop bits, rather than one.

CREAD

Enable receiver.

PARENB

Enable parity generation on output and parity checking for input.

PARODD

If set, then parity for input and output is odd; otherwise even parity is used.

HUPCL

Lower modem control lines after last process closes the device (hang up).

CLOCAL

Ignore modem control lines.

LOBLK

(not in POSIX) Block output from a noncurrent shell layer. For use by **shl** (shell layers). (Not implemented on Linux.)

CIBAUD

(not in POSIX) Mask for input speeds. The values for the **CIBAUD** bits are the same as the values for the **CBAUD** bits, shifted left **IBSHIFT** bits.
[requires **_BSD_SOURCE** or **_SVID_SOURCE**] (Not implemented on Linux.)

CMSPAR

(not in POSIX) Use "stick" (mark/space) parity (supported on certain serial devices): if **PARODD** is set, the parity bit is always 1; if **PARODD** is not set, then the parity bit is always 0). [requires **_BSD_SOURCE** or **_SVID_SOURCE**]

CRTSCTS
(not in POSIX) Enable RTS/CTS (hardware) flow control.
[requires **_BSD_SOURCE** or **_SVID_SOURCE**]
c_lflag flag constants:
ISIG
When any of the characters INTR, QUIT, SUSP, or DSUSP are received, generate the corresponding signal.

ICANON

Enable canonical mode (described below).

XCASE

(not in POSIX; not supported under Linux) If **ICANON** is also set, terminal is uppercase only. Input is converted to lowercase, except for characters preceded by \. On output, uppercase characters are preceded by \ and lowercase characters are converted to uppercase. [requires _BSD_SOURCE or _SVID_SOURCE or _XOPEN_SOURCE]

ECHO

Echo input characters.

ECHOE

If **ICANON** is also set, the ERASE character erases the preceding input character, and WERASE erases the preceding word.

ECHOK

If **ICANON** is also set, the KILL character erases the current line.

ECHONL

If **ICANON** is also set, echo the NL character even if ECHO is not set.

ECHOCTL
(not in POSIX) If **ECHO** is also set, terminal special characters other than TAB, NL, START, and STOP are echoed as **^X**, where X is the character with ASCII code 0x40 greater than the special character. For example, character 0x08 (BS) is echoed as **^H**.
[requires **_BSD_SOURCE** or **_SVID_SOURCE**]
ECHOPRT
(not in POSIX) If **ICANON** and **ECHO** are also set, characters are printed as they are being erased. [requires **_BSD_SOURCE** or **_SVID_SOURCE**]
ECHOKE
(not in POSIX) If **ICANON** is also set, KILL is echoed by erasing each character on the line, as specified by **ECHOE** and **ECHOPRT**. [requires **_BSD_SOURCE** or **_SVID_SOURCE**]

DEFECHO

(not in POSIX) Echo only when a process is reading. (Not implemented on Linux.)
FLUSHO
(not in POSIX; not supported under Linux) Output is being flushed. This flag is toggled by typing the DISCARD character. [requires **_BSD_SOURCE** or **_SVID_SOURCE**]

NOFLSH

Disable flushing the input and output queues when generating signals for the INT, QUIT, and SUSP characters.

TOSTOP

Send the **SIGTTOU** signal to the process group of a background process which tries to write to its controlling terminal.

PENDIN

(not in POSIX; not supported under Linux) All characters in the input queue are reprinted when the next character is read. (**bash**(1) handles typeahead this way.)
[requires **_BSD_SOURCE** or **_SVID_SOURCE**]

IEXTEN

Enable implementation-defined input processing. This flag, as well as **ICANON** must be enabled for the special characters EOL2, LNEXT, REPRINT, WERASE to be interpreted, and for the **IUCLC** flag to be effective.

The *c_cc* array defines the terminal special characters. The symbolic indices (initial values) and meaning are:
VDISCARD
(not in POSIX; not supported under Linux; 017, SI, Ctrl-O) Toggle: start/stop discarding pending output. Recognized when **IEXTEN** is set, and then not passed as input.
VDSUSP
(not in POSIX; not supported under Linux; 031, EM, Ctrl-Y) Delayed suspend character (DSUSP): send **SIGTSTP** signal when the character is read by the user program. Recognized when **IEXTEN** and **ISIG** are set, and the system supports job control, and then not passed as input.

VEOF

(004, EOT, Ctrl-D) End-of-file character (EOF). More precisely: this character causes the pending tty buffer to be sent to the waiting user program without waiting for end-of-line. If it is the first character of the line, the **read**(2) in the user program returns 0, which signifies end-of-file. Recognized when **ICANON** is set, and then not passed as input.

VEOL

(0, NUL) Additional end-of-line character (EOL). Recognized when **ICANON** is set.

VEOL2

(not in POSIX; 0, NUL) Yet another end-of-line character (EOL2). Recognized when **ICANON** is set.

VERASE

(0177, DEL, rubout, or 010, BS, Ctrl-H, or also #) Erase character (ERASE). This erases the previous not-yet-erased character, but does not erase past EOF or beginning-of-line. Recognized when **ICANON** is set, and then not passed as input.

VINTR

(003, ETX, Ctrl-C, or also 0177, DEL, rubout) Interrupt character (INTR). Send a **SIGINT** signal. Recognized when **ISIG** is set, and then not passed as input.

VKILL

(025, NAK, Ctrl-U, or Ctrl-X, or also @) Kill character (KILL). This erases the input since the last EOF or beginning-of-line. Recognized when **ICANON** is set, and then not passed as input.

VLNEXT

(not in POSIX; 026, SYN, Ctrl-V) Literal next (LNEXT). Quotes the next input character, depriving it of a possible special meaning. Recognized when **IEXTEN** is set, and then not passed as input.

VMIN

Minimum number of characters for noncanonical read (MIN).

VQUIT

(034, FS, Ctrl-\) Quit character (QUIT). Send **SIGQUIT** signal. Recognized when **ISIG** is set, and then not passed as input.

VREPRINT
(not in POSIX; 022, DC2, Ctrl-R) Reprint unread characters (REPRINT). Recognized when **ICANON** and **IEXTEN** are set, and then not passed as input.
VSTART
(021, DC1, Ctrl-Q) Start character (START). Restarts output stopped by the Stop character. Recognized when **IXON** is set, and then not passed as input.

VSTATUS
(not in POSIX; not supported under Linux; status request: 024, DC4, Ctrl-T). Status character (STATUS). Display status information at terminal, including state of foreground process and amount of CPU time it has consumed. Also sends a **SIGINFO** signal (not supported on Linux) to the foreground process group.
VSTOP
(023, DC3, Ctrl-S) Stop character (STOP). Stop output until Start character typed. Recognized when **IXON** is set, and then not passed as input.

VSUSP

(032, SUB, Ctrl-Z) Suspend character (SUSP). Send **SIGTSTP** signal. Recognized when **ISIG** is set, and then not passed as input.

VSWTCH

(not in POSIX; not supported under Linux; 0, NUL) Switch character (SWTCH). Used in System V to switch shells in *shell layers*, a predecessor to shell job control.

VTIME

Timeout in deciseconds for noncanonical read (TIME).

VWERASE
(not in POSIX; 027, ETB, Ctrl-W) Word erase (WERASE). Recognized when **ICANON** and **IEXTEN** are set, and then not passed as input.
An individual terminal special character can be disabled by setting the value of the corresponding *c_cc* element to **_POSIX_VDISABLE**.
The above symbolic subscript values are all different, except that **VTIME**, **VMIN** may have the same value as **VEOL**, **VEOF**, respectively. In noncanonical mode the special character meaning is replaced by the timeout meaning. For an explanation of **VMIN** and **VTIME**, see the description of noncanonical mode below.

Retrieving and changing terminal settings

tcgetattr() gets the parameters associated with the object referred by *fd* and stores them in the *termios* structure referenced by *termios_p*. This function may be invoked from a background process; however, the terminal attributes may be subsequently changed by a foreground process.
tcsetattr() sets the parameters associated with the terminal (unless support is required from the underlying hardware that is not available) from the *termios* structure referred to by *termios_p*. *optional_actions* specifies when the changes take effect:

TCSANOW
the change occurs immediately.
TCSADRAIN
the change occurs after all output written to *fd* has been transmitted. This function should be used when changing parameters that affect output.
TCSAFLUSH
the change occurs after all output written to the object referred by *fd* has been transmitted, and all input that has been received but not read will be discarded before the change is made.
Canonical and noncanonical mode

The setting of the **ICANON** canon flag in *c_lflag* determines whether the terminal is operating in canonical mode (**ICANON** set) or noncanonical mode (**ICANON** unset). By default, **ICANON** set.
In canonical mode:

*
Input is made available line by line. An input line is available when one of the line delimiters is typed (NL, EOL, EOL2; or EOF at the start of line). Except in the case of EOF, the line delimiter is included in the buffer returned by **read**(2).

*

Line editing is enabled (ERASE, KILL; and if the **IEXTEN** flag is set: WERASE, REPRINT, LNEXT). A **read**(2) returns at most one line of input; if the **read**(2) requested fewer bytes than are available in the current line of input, then only as many bytes as requested are read, and the remaining characters will be available for a future **read**(2).

In noncanonical mode input is available immediately (without the user having to type a line-delimiter character), no input processing is performed, and line editing is disabled. The settings of MIN (*c_cc[VMIN]*) and TIME (*c_cc[VTIME]*) determine the circumstances in which a **read**(2) completes; there are four distinct cases:
*

MIN == 0; TIME == 0: If data is available, **read**(2) returns immediately, with the lesser of the number of bytes available, or the number of bytes requested. If no data is available, **read**(2) returns 0.

*

MIN > 0; TIME == 0: **read**(2) blocks until the lesser of MIN bytes or the number of bytes requested are available, and returns the lesser of these two values.

*

MIN == 0; TIME > 0: TIME specifies the limit for a timer in tenths of a second. The timer is started when **read**(2) is called. **read**(2) returns either when at least one byte of data is available, or when the timer expires. If the timer expires without any input becoming available, **read**(2) returns 0.

*

MIN > 0; TIME > 0: TIME specifies the limit for a timer in tenths of a second. Once an initial byte of input becomes available, the timer is restarted after each further byte is received. **read**(2) returns either when the lesser of the number of bytes requested or MIN byte have been read, or when the inter-byte timeout expires. Because the timer is only started after the initial byte becomes available, at least one byte will be read.

Raw mode

cfmakeraw() sets the terminal to something like the "raw" mode of the old Version 7 terminal driver: input is available character by character, echoing is disabled, and all special processing of terminal input and output characters is disabled. The terminal attributes are set as follows:

```
termios_p->c_iflag &= ~(IGNBRK | BRKINT | PARMRK | ISTRIP
                | INLCR | IGNCR | ICRNL | IXON);
termios_p->c_oflag &= ~OPOST;
termios_p->c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);
termios_p->c_cflag &= ~(CSIZE | PARENB);
termios_p->c_cflag |= CS8;
```

Line control

tcsendbreak() transmits a continuous stream of zero-valued bits for a specific duration, if the terminal is using asynchronous serial data transmission. If *duration* is zero, it transmits zero-valued bits for at least 0.25 seconds, and not more that 0.5 seconds. If *duration* is not zero, it sends zero-valued bits for some implementation-defined length of time.
If the terminal is not using asynchronous serial data transmission, **tcsendbreak**() returns without taking any action.

tcdrain() waits until all output written to the object referred to by *fd* has been transmitted.

tcflush() discards data written to the object referred to by *fd* but not transmitted, or data received but not read, depending on the value of *queue_selector*:

TCIFLUSH
flushes data received but not read.
TCOFLUSH
flushes data written but not transmitted.
TCIOFLUSH
flushes both data received but not read, and data written but not transmitted.

tcflow() suspends transmission or reception of data on the object referred to by *fd*, depending on the value of *action*:
TCOOFF
suspends output.

TCOON

restarts suspended output.

TCIOFF

transmits a STOP character, which stops the terminal device from transmitting data to the system.

TCION

transmits a START character, which starts the terminal device transmitting data to the system.

The default on open of a terminal file is that neither its input nor its output is suspended.
Line speed

The baud rate functions are provided for getting and setting the values of the input and output baud rates in the *termios* structure. The new values do not take effect until **tcsetattr**() is successfully called.
Setting the speed to **B0** instructs the modem to "hang up". The actual bit rate corresponding to **B38400** may be altered with **setserial**(8).

The input and output baud rates are stored in the *termios* structure.

cfgetospeed() returns the output baud rate stored in the *termios* structure pointed to by *termios_p*.

cfsetospeed() sets the output baud rate stored in the *termios* structure pointed to by *termios_p* to *speed*, which must be one of these constants:

B0
B50

B75

B110

B134

B150

B200

B300

B600

B1200

B1800

B2400

B4800

B9600

B19200

B38400

B57600

B115200

B230400

The zero baud rate, **B0**, is used to terminate the connection. If B0 is specified, the modem control lines shall no longer be asserted. Normally, this will disconnect the line. **CBAUDEX** is a mask for the speeds beyond those defined in POSIX.1 (57600 and above).
Thus, **B57600** & **CBAUDEX** is nonzero.
cfgetispeed() returns the input baud rate stored in the *termios* structure.

cfsetispeed() sets the input baud rate stored in the *termios* structure to *speed*, which must be specified as one of the **Bnnn** constants listed above for **cfsetospeed**(). If the input baud rate is set to zero, the input baud rate will be equal to the output baud rate.

cfsetspeed() is a 4.4BSD extension. It takes the same arguments as **cfsetispeed**(), and sets both input and output speed.

Return Value

cfgetispeed() returns the input baud rate stored in the *termios* structure.

cfgetospeed() returns the output baud rate stored in the *termios* structure.

All other functions return:

0
on success.

-1

on failure and set *errno* to indicate the error.

Note that **tcsetattr**() returns success if *any* of the requested changes could be successfully carried out. Therefore, when making multiple changes it may be necessary to follow this call with a further call to **tcgetattr**() to check that all changes have been performed successfully.

tgetent(3) - Linux man page

Name

tgetent, **tgetflag**, **tgetnum**, **tgetstr**, **tgoto**, **tputs** - direct **curses** interface to the terminfo capability database

Synopsis

```
#include <curses.h>
#include <term.h>

extern char PC;
extern char * UP;
extern char * BC;
extern unsigned ospeed;

int tgetent(char *bp, const char *name);
int tgetflag(char *id);
int tgetnum(char *id);
char *tgetstr(char *id, char **area);
char *tgoto(const char *cap, int col, int row);
int tputs(const char *str, int affcnt, int (*putc)(int));
```

Description

These routines are included as a conversion aid for programs that use the *termcap* library. Their parameters are the same and the routines are emulated using the *terminfo* database. Thus, they can only be used to query the capabilities of entries for which a terminfo entry has been compiled.

The **tgetent** routine loads the entry for *name*. It returns 1 on success, 0 if there is no such entry, and -1 if the terminfo database could not be found. The emulation ignores the buffer pointer *bp*.

The **tgetflag** routine gets the boolean entry for *id*, or zero if it is not available.

The **tgetnum** routine gets the numeric entry for *id*, or -1 if it is not available.

The **tgetstr** routine returns the string entry for *id*, or zero if it is not available. Use **tputs** to output the returned string. The return value will also be copied to the buffer pointed to by *area*, and the *area* value will be updated to point past the null ending this value.

Only the first two characters of the **id** parameter of **tgetflag**, **tgetnum** and **tgetstr** are compared in lookups.

The **tgoto** routine instantiates the parameters into the given capability. The output from this routine is to be passed to **tputs**.

The **tputs** routine is described on the **curs_terminfo**(3X) manual page. It can retrieve capabilities by either termcap or terminfo name.

The variables **PC**, **UP** and **BC** are set by **tgetent** to the terminfo entry's data for **pad_char**, **cursor_up** and **backspace_if_not_bs**, respectively. **UP** is not used by

ncurses. **PC** is used in the **tdelay_output** function. **BC** is used in the **tgoto** emulation. The variable **ospeed** is set by ncurses in a system-specific coding to reflect the terminal speed.

Return Value

Except where explicitly noted, routines that return an integer return **ERR** upon failure and **OK** (SVr4 only specifies "an integer value other than **ERR**") upon successful completion.

Routines that return pointers return **NULL** on error.

tgetflag(3) - Linux man page

Name

tgetent, **tgetflag**, **tgetnum**, **tgetstr**, **tgoto**, **tputs** - direct **curses** interface to the terminfo capability database

Synopsis

```
#include <curses.h>
#include <term.h>

extern char PC;
extern char * UP;
extern char * BC;
extern unsigned ospeed;

int tgetent(char *bp, const char *name);
int tgetflag(char *id);
int tgetnum(char *id);
char *tgetstr(char *id, char **area);
char *tgoto(const char *cap, int col, int row);
int tputs(const char *str, int affcnt, int (*putc)(int));
```

Description

These routines are included as a conversion aid for programs that use the *termcap* library. Their parameters are the same and the routines are emulated using the *terminfo* database. Thus, they can only be used to query the capabilities of entries for which a terminfo entry has been compiled.

The **tgetent** routine loads the entry for *name*. It returns 1 on success, 0 if there is no such entry, and -1 if the terminfo database could not be found. The emulation ignores the buffer pointer *bp*.

The **tgetflag** routine gets the boolean entry for *id*, or zero if it is not available.

The **tgetnum** routine gets the numeric entry for *id*, or -1 if it is not available.

The **tgetstr** routine returns the string entry for *id*, or zero if it is not available. Use **tputs** to output the returned string. The return value will also be copied to the buffer pointed to by *area*, and the *area* value will be updated to point past the null ending this value.

Only the first two characters of the **id** parameter of **tgetflag**, **tgetnum** and **tgetstr** are compared in lookups.

The **tgoto** routine instantiates the parameters into the given capability. The output from this routine is to be passed to **tputs**.

The **tputs** routine is described on the **curs_terminfo**(3X) manual page. It can retrieve capabilities by either termcap or terminfo name.

The variables **PC**, **UP** and **BC** are set by **tgetent** to the terminfo entry's data for **pad_char**, **cursor_up** and **backspace_if_not_bs**, respectively. **UP** is not used by ncurses. **PC** is used in the **tdelay_output** function. **BC** is used in the **tgoto** emulation. The variable **ospeed** is set by ncurses in a system-specific coding to reflect the terminal speed.

Return Value

Except where explicitly noted, routines that return an integer return **ERR** upon failure and **OK** (SVr4 only specifies "an integer value other than **ERR**") upon successful completion.

Routines that return pointers return **NULL** on error.