

systemcall

getrlimit(2)

```
#include <sys/resource.h>
```

```
int getrlimit(int resource, struct rlimit *rlp);
```

```
int setrlimit(int resource, const struct rlimit *rlp);
```

#### RETURN VALUES

Upon successful completion, `getrlimit()` and `setrlimit()` return 0. Otherwise, these functions return -1 and set `errno` to indicate the error.

#### DESCRIPTION

Limits on the consumption of a variety of system resources by a process and each process it creates may be obtained with the `getrlimit()` and set with `setrlimit()` functions.

Each call to either `getrlimit()` or `setrlimit()` identifies a specific resource to be operated upon as well as a resource limit. A resource limit is a pair of values: one specifying the current (soft) limit, the other a maximum (hard) limit. Soft limits may be changed by a process to any value that is less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or equal to the soft limit. Only a process with an effective user ID of super-user can raise a hard limit. Both hard and soft limits can be changed in a single call to `setrlimit()` subject to the constraints described above. Limits may have an "infinite" value of `RLIM_INFINITY`. The `_r_l_p` argument is a pointer to `struct rlimit` that includes the following members:

```
struct rlimit {  
    .  
    .
```

```

    rlim_t  rlim_cur; /* current (soft) limit */
    rlim_t  rlim_max; /* hard limit */
    .
    .
}

```

The type `rlim_t` is an arithmetic data type to which objects of type `int`, `size_t`, and `off_t` can be cast without loss of information.

The possible resources, their descriptions, and the actions taken when the current limit is exceeded are summarized as follows:

#### RLIMIT\_CORE

The maximum size of a core file in bytes that may be created by a process. A limit of 0 will prevent the creation of a core file. The writing of a core file will terminate at this size.

#### RLIMIT\_CPU

The maximum amount of CPU time in seconds used by a process. This is a soft limit only. The `SIGXCPU` signal is sent to the process. If the process is holding or ignoring `SIGXCPU`, the behavior is scheduling class defined.

#### RLIMIT\_DATA

The maximum size of a process's heap in bytes. The `brk(2)` function will fail with `errno` set to `ENOMEM`.

#### RLIMIT\_FSIZE

The maximum size of a file in bytes that may be created by a process. A limit of 0 will prevent the creation of a file. The `SIGXFSZ` signal is sent to the process. If the process is holding or ignoring `SIGXFSZ`, continued attempts to increase the size of a file beyond the limit will fail with `errno` set to `EFBIG`.

#### RLIMIT\_NOFILE

One more than the maximum value that the system may assign to a newly created descriptor. This limit constrains the number of file descriptors

that a process may create.

#### RLIMIT\_STACK

The maximum size of a process's stack in bytes. The system will not automatically grow the stack beyond this limit.

Within a process, `setrlimit()` will increase the limit on the size of your stack, but will not move current memory segments to allow for that growth. To guarantee that the process stack can grow to the limit, the limit must be altered prior to the execution of the process in which the new stack size is to be used.

Within a multithreaded process, `setrlimit()` has no impact on the stack size limit for the calling thread if the calling thread is not the main thread. A call to `setrlimit()` for `RLIMIT_STACK` impacts only the main thread's stack, and should be made only from the main thread, if at all.

The `SIGSEGV` signal is sent to the process. If the process is holding or ignoring `SIGSEGV`, or is catching `SIGSEGV` and has not made arrangements to use an alternate stack (see `sigaltstack(2)`), the disposition of `SIGSEGV` will be set to `SIG_DFL` before it is sent.

#### RLIMIT\_VMEM

The maximum size of a process's mapped address space in bytes. If this limit is exceeded, the `brk(2)` and `mmap(2)` functions will fail with `errno` set to `ENOMEM`. In addition, the automatic stack growth will fail with the effects outlined above.

#### RLIMIT\_AS

This is the maximum size of a process's total available memory, in bytes. If this limit is exceeded, the `brk(2)`, `malloc(3C)`, `mmap(2)` and `sbrk(2)` functions will fail with `errno` set to `ENOMEM`. In addition, the automatic stack growth will fail with the effects outlined above.

Because limit information is stored in the per-process

information, the shell builtin `ulimit` command must directly execute this system call if it is to affect all future processes created by the shell.

The value of the current limit of the following resources affect these implementation defined parameters:

Limit	Implementation Defined Constant
<code>RLIMIT_FSIZE</code>	<code>FCHR_MAX</code>
<code>RLIMIT_NOFILE</code>	<code>OPEN_MAX</code>

When using the `getrlimit()` function, if a resource limit can be represented correctly in an object of type `rlim_t`, then its representation is returned; otherwise, if the value of the resource limit is equal to that of the corresponding saved hard limit, the value returned is `RLIM_SAVED_MAX`; otherwise the value returned is `RLIM_SAVED_CUR`.

When using the `setrlimit()` function, if the requested new limit is `RLIM_INFINITY`, the new limit will be "no limit"; otherwise if the requested new limit is `RLIM_SAVED_MAX`, the new limit will be the corresponding saved hard limit; otherwise, if the requested new limit is `RLIM_SAVED_CUR`, the new limit will be the corresponding saved soft limit; otherwise, the new limit will be the requested value. In addition, if the corresponding saved limit can be represented correctly in an object of type `rlim_t`, then it will be overwritten with the new limit.

The result of setting a limit to `RLIM_SAVED_MAX` or `RLIM_SAVED_CUR` is unspecified unless a previous call to `getrlimit()` returned that value as the soft or hard limit for the corresponding resource limit.

A limit whose value is greater than `RLIM_INFINITY` is permitted.

The `exec` family of functions also cause resource limits to be saved. See `exec(2)`.

## ERRORS

The `getrlimit()` and `setrlimit()` functions will fail if:

**EFAULT** The `_r_l_p` argument points to an illegal address.

**EINVAL** An invalid `_r_e_s_o_u_r_c_e` was specified; or in a `setrlimit()` call, the new `rlim_cur` exceeds the new

rlim\_max.

**EPERM** The limit specified to `setrlimit()` would have raised the maximum limit value, and the effective user of the calling process is not super-user.

The `setrlimit()` function may fail if:

**EINVAL** The limit specified cannot be lowered because current usage is already higher than the limit.

C Library Functions

`getrusage(3C)`

## NAME

`getrusage` - get information about resource utilization

## SYNOPSIS

```
#include <sys/resource.h>
#include <sys/time.h>
```

```
int getrusage(int who, struct rusage *r_usage);
```

## DESCRIPTION

The `getrusage()` function provides measures of the resources used by the current process or its terminated and waited-for child processes. If the value of the `who` argument is `RUSAGE_SELF`, information is returned about resources used by the current process. If the value of the `who` argument is `RUSAGE_CHILDREN`, information is returned about resources used by the terminated and waited-for children of the current process. If the child is never waited for (for instance, if the parent has `SA_NOCLDWAIT` set or sets `SIGCHLD` to `SIG_IGN`), the resource information for the child process is discarded and not included in the resource information provided by `getrusage()`.

The `r_usage` argument is a pointer to an object of type `struct rusage` in which the returned information is stored. The members of `rusage` are as follows:

```
struct timeval ru_utime; /* user time used, defined in <sys/time.h> */
```

```

struct timeval ru_stime; /* system time used, defined in <sys/time.h> */
long          ru_maxrss; /* maximum resident set size */
long          ru_idrss; /* integral resident set size */
long          ru_minflt; /* page faults not requiring physical I/O */
long          ru_majflt; /* page faults requiring physical I/O */
long          ru_nswap; /* swaps */
long          ru_inblock; /* block input operations */
long          ru_oublock; /* block output operations */
long          ru_msgsnd; /* messages sent */
long          ru_msgrcv; /* messages received */
long          ru_nsignals; /* signals received */
long          ru_nvcsw; /* voluntary context switches */
long          ru_nivcsw; /* involuntary context switches */

```

The structure members are interpreted as follows:

`ru_utime` The total amount of time spent executing in user mode. Time is given in seconds and microseconds.

`ru_stime` The total amount of time spent executing in system mode. Time is given in seconds and microseconds.

`ru_maxrss` The maximum resident set size. Size is given in pages (the size of a page, in bytes, is given by the `getpagesize(3C)` function).

`ru_idrss` An "integral" value indicating the amount of memory in use by a process while the process is running. This value is the sum of the resident set sizes of the process running when a clock tick occurs. The value is given in pages times clock ticks. It does not take sharing into account. See the NOTES section of this page.

`ru_minflt` The number of page faults serviced which did not require any physical I/O activity. See the NOTES section of this page.

`ru_majflt` The number of page faults serviced which required physical I/O activity. This could include page ahead operations by the kernel. See the NOTES section of this page.

`ru_nswap` The number of times a process was swapped out of main memory.

ru\_inblock

The number of times the file system had to perform input in servicing a read(2) request.

ru\_oublock

The number of times the file system had to perform output in servicing a write(2) request.

ru\_msgsnd The number of messages sent over sockets.

ru\_msgrcv The number of messages received from sockets.

ru\_nsignals

The number of signals delivered.

ru\_nvcsw The number of times a context switch resulted due to a process voluntarily giving up the processor before its time slice was completed (usually to await availability of a resource).

ru\_nivcsw The number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

## RETURN VALUES

Upon successful completion, `getrusage()` returns 0. Otherwise, -1 is returned and `errno` is set to indicate the error.

## ERRORS

The `getrusage()` function will fail if:

**EFAULT** The address specified by the `_r_u_s_a_g_e` argument is not in a valid portion of the process' address space.

**EINVAL** The `who` parameter is not a valid value.

System Calls

`time(2)`

NAME

time - get time

## SYNOPSIS

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

```
time_t time(time_t *tloc);
```

## DESCRIPTION

The `time()` function returns the value of time in seconds since 00:00:00 UTC, January 1, 1970.

If `tloc` is non-zero, the return value is also stored in the location to which `tloc` points. If `tloc` points to an illegal address, `time()` fails and its actions are undefined.

## RETURN VALUES

Upon successful completion, `time()` returns the value of time. Otherwise, `(time_t)-1` is returned and `errno` is set to indicate the error.

## SEE ALSO

`stime(2)`, `ctime(3C)`, `attributes(5)`

C Library Functions

`ctime(3C)`

## SYNOPSIS

```
#include <time.h>
```

```
char *ctime(const time_t *clock);
```

## DESCRIPTION

The `ctime()` function converts the time pointed to by `_c_l_o_c_k`, representing the time in seconds since the Epoch (00:00:00 UTC, January 1, 1970), to local time in the form of a 26-character string as shown below. Time zone and daylight savings corrections are made before string generation. The fields are constant width:

```
Fri Sep 13 00:00:00 1986\n\0
```

Declarations of all the functions and externals, and the `tm` structure, are in the `<time.h>` header. The members of the `tm` structure are:

```
int  tm_sec; /* seconds after the minute - [0, 61] */
        /* for leap seconds */
int  tm_min; /* minutes after the hour - [0, 59] */
int  tm_hour; /* hour since midnight - [0, 23] */
int  tm_mday; /* day of the month - [1, 31] */
int  tm_mon; /* months since January - [0, 11] */
int  tm_year; /* years since 1900 */
int  tm_wday; /* days since Sunday - [0, 6] */
int  tm_yday; /* days since January 1 - [0, 365] */
int  tm_isdst; /* flag for alternate daylight savings time */
```

```
Example: time_t      t;
        char        *p;
        time(&t);
        p=ctime(&t);
        printf("%s\n", p);
```

C Library Functions

`sleep(3C)`

#### SYNOPSIS

```
unsigned sleep (unsigned time)
```

#### DESCRIPTION

The `sleep` utility will suspend execution for at least the integral number of seconds specified by the `time` operand.

#### OPERANDS

The following operands are supported:

`_ t_ i_ m_ e` A non-negative decimal integer specifying the

number of seconds for which to suspend execution.

#### EXIT STATUS

The following exit values are returned:

- 0 The execution was successfully suspended for at least `time` seconds, or a `SIGALRM` signal was received (see `NOTES`).
- >0 An error has occurred.

C Library Functions

`usleep(3C)`

#### SYNOPSIS

```
unsigned    usleep (unsigned    time)
```

#### DESCRIPTION

The `sleep` utility will suspend execution for at least the integral number of microseconds specified by the `time` operand.

#### OPERANDS

The following operands are supported:

- `_ t_ i_ m_ e` A non-negative decimal integer specifying the number of microseconds for which to suspend execution.

C Library Functions

`gethostname(3C)`

#### NAME

`gethostname`, `sethostname` - get or set name of current host

#### SYNOPSIS

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

```
int gethostname(char *name, int namelen);
```

```
int sethostname(char *name, int namelen);
```

## DESCRIPTION

The `gethostname()` function returns the standard host name for the current processor, as previously set by `sethostname()`. The `_n_a_m_e_l_e_n` argument specifies the size of the array pointed to by `_n_a_m_e`. The returned name is null-terminated unless insufficient space is provided.

The `sethostname()` function sets the name of the host machine to be `_n_a_m_e`, which has length `_n_a_m_e_l_e_n`. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

Host names are limited to `MAXHOSTNAMELEN` characters, currently 256, defined in the `<netdb.h>` header.

## RETURN VALUES

Upon successful completion, `gethostname()` and `sethostname()` return 0. Otherwise, they return -1 and set `errno` to indicate the error.

## ERRORS

The `gethostname()` and `sethostname()` functions will fail if:

**EFAULT** The `_n_a_m_e` or `_n_a_m_e_l_e_n` argument gave an invalid address.

The `sethostname()` function will fail if:

**EPERM** The caller was not the super-user.

## SEE ALSO

`sysinfo(2)`, `uname(2)`, `gethostid(3C)`

C Library Functions

`getlogin(3C)`

NAME

getlogin, getlogin\_r - get login name

## SYNOPSIS

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

```
char *getlogin(void);
```

## DESCRIPTION

The `getlogin()` function returns a pointer to the login name as found in `/var/adm/utmp`. It may be used in conjunction with `getpwnam(3C)` to locate the correct password file entry when the same user ID is shared by several login names.

If `getlogin()` is called within a process that is not attached to a terminal, it returns a null pointer. The correct procedure for determining the login name is to call `cuserid(3S)`, or to call `getlogin()` and if it fails to call `getpwuid(3C)`.

You don't have to `malloc()` to any string for this function.

## RETURN VALUES

Upon successful completion, `getlogin()` returns a pointer to the login name or a null pointer if the user's login name cannot be found. Otherwise it returns a null pointer and sets `errno` to indicate the error.

## ERRORS

The `getlogin()` function may fail if:

**EMFILE** There are `OPEN_MAX` file descriptors currently open in the calling process.

**ENFILE** The maximum allowable number of files is currently open in the system.

**ENXIO** The calling process has no controlling terminal.

The `getlogin_r()` function will fail if:

**ERANGE** The size of the buffer is smaller than the result

to be returned.

EINVAL And entry for the current user was not found in the /var/adm/utmp file.

## USAGE

The return value may point to static data whose content is overwritten by each call.

Three names associated with the current process can be determined: `getpwuid(geteuid())` returns the name associated with the effective user ID of the process; `getlogin()` returns the name associated with the current login activity; and `getpwuid(getuid())` returns the name associated with the real user ID of the process.

## Standard I/O Functions

## fflush(3S)

### NAME

`fflush` - flush a stream

### SYNOPSIS

```
#include <stdio.h>
```

```
int fflush(FILE *stream);
```

### DESCRIPTION

If `_s_t_r_e_a_m` points to an output stream or an update stream in which the most recent operation was not input, `fflush()` causes any unwritten data for that stream to be written to the file, and the `st_ctime` and `st_mtime` fields of the underlying file are marked for update.

If `_s_t_r_e_a_m` is a null pointer, `fflush()` performs this flushing action on all streams for which the behavior is defined above.

### RETURN VALUES

Upon successful completion, `fflush()` returns 0. Otherwise, it returns EOF and sets `errno` to indicate the error.

## ERRORS

The `fflush()` function will fail if:

**EAGAIN** The `O_NONBLOCK` flag is set for the file descriptor underlying `_s_t_r_e_a_m` and the process would be delayed in the write operation.

**EBADF** The file descriptor underlying `_s_t_r_e_a_m` is not valid.

**EFBIG** An attempt was made to write a file that exceeds the maximum file size or the process's file size limit; or the file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

**EINTR** The `fflush()` function was interrupted by a signal.

**EIO** The process is a member of a background process group attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking `SIGTTOU`, and the process group of the process is orphaned.

**ENOSPC** There was no free space remaining on the device containing the file.

**EPIPE** An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A `SIGPIPE` signal will also be sent to the process.

The `fflush()` function may fail if:

**ENXIO** A request was made of a non-existent device, or the request was beyond the limits of the device.

## NAME

gettimeofday, settimeofday - get or set the date and time

## SYNOPSIS

```
#include <sys/time.h>
```

```
int gettimeofday(struct timeval *tp, void *);
```

```
int settimeofday(struct timeval *tp, void *);
```

## DESCRIPTION

The `gettimeofday()` function gets and the `settimeofday()` function sets the system's notion of the current time. The current time is expressed in elapsed seconds and microseconds since 00:00 Universal Coordinated Time, January 1, 1970. The resolution of the system clock is hardware dependent; the time may be updated continuously or in clock ticks.

The `_t_p` argument points to a `timeval` structure, which includes the following members:

```
long tv_sec; /* seconds since Jan. 1, 1970 */  
long tv_usec; /* and microseconds */
```

If `_t_p` is a null pointer, the current time information is not returned or set.

The TZ environment variable holds time zone information. See [TIMEZONE\(4\)](#).

The second argument to `gettimeofday()` and `settimeofday()` should be a pointer to `_N_U_L_L`.

Only the super-user may set the time of day.

## RETURN VALUES

Upon successful completion, 0 is returned. Otherwise, -1 is returned and `errno` is set to indicate the error.

## ERRORS

The `gettimeofday()` function will fail if:

**EINVAL** The structure pointed to by `_t_p` specifies an invalid time.

**EPERM** A user other than the privileged user attempted to set the time or time zone.

Additionally, the `gettimeofday()` function will fail for 32-bit interfaces if:

**E\_OVERFLOW** The system time has progressed beyond 2038, thus the size of the `tv_sec` member of the `timeval` structure pointed to by `_t_p` is insufficient to hold the current time in seconds.

#### USAGE

If the `tv_usec` member of `_t_p` is  $> 500000$ , `settimeofday()` rounds the seconds upward. If the time needs to be set with better than one second accuracy, call `settimeofday()` for the seconds and then `adjtime(2)` for finer accuracy.

#### SEE ALSO

`adjtime(2)`, `ctime(3C)`, `TIMEZONE(4)`, `attributes(5)`

#### NAME

`getenv`, `putenv`, `setenv`, `unsetenv` - environment variable functions

#### SYNOPSIS

```
#include <stdlib.h>
```

```
char *
```

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

```
int
```

```
setenv(const char *name, const char *value, int overwrite);
```

```
int
```

```
putenv(const char *string);
```

```
void
```

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

## DESCRIPTION

These functions set, unset, and fetch environment variables from the host environment list. For compatibility with differing environment conventions, the given arguments name and value may be appended and prepended, respectively, with an equal sign "=".

The `getenv()` function obtains the current value of the environment variable, name. If the variable name is not in the current environment, a null pointer is returned.

The `setenv()` function inserts or resets the environment variable name in the current environment list. If the variable name does not exist in the list, it is inserted with the given value. If the variable does exist, the argument overwrite is tested; if overwrite is zero, the variable is not reset, otherwise it is reset to the given value.

The `putenv()` function takes an argument of the form name=value and is equivalent to:

```
setenv(name, value, 1);
```

The `unsetenv()` function deletes all instances of the variable name pointed to by name from the list.

## RETURN VALUES

The functions `setenv()` and `putenv()` return zero if successful; otherwise the global variable `errno` is set to indicate the error and -1 is returned.

If `getenv()` is successful, the string returned should be considered read-only.

## ERRORS

[ENOMEM] The function `setenv()` or `putenv()` failed because they were unable to allocate memory for the environment.

## SEE ALSO

`csh(1)`, `sh(1)`, `execve(2)`, `environ(7)`

## STANDARDS

The `getenv()` function conforms to ANSI X3.159-1989 ("ANSI C").

## HISTORY

The functions `setenv()` and `unsetenv()` appeared in Version 7 AT&T UNIX.

The `putenv()` function appeared in 4.3BSD-Reno.

C Library Functions

`putenv(3C)`

## NAME

`putenv` - change or add value to environment

## SYNOPSIS

```
#include <stdlib.h>
```

```
int putenv(char *string);
```

## DESCRIPTION

The `putenv()` function makes the value of the environment variable `_n_a_m_e` equal to `_v_a_l_u_e` by altering an existing variable or creating a new one. In either case, the string pointed to by `_s_t_r_i_n_g` becomes part of the environment, so altering the string will change the environment.

The `_s_t_r_i_n_g` argument points to a string of the form `_n_a_m_e=_v_a_l_u_e`. The space used by `_s_t_r_i_n_g` is no longer used once a new string-defining `_n_a_m_e` is passed to `putenv()`.

The `putenv()` function uses `malloc(3C)` to enlarge the environment.

After `putenv()` is called, environment variables are not in alphabetical order.

## RETURN VALUES

The `putenv()` functions returns a non-zero value if it was unable to obtain enough space using `malloc(3C)` for an expanded environment. Otherwise, 0 is returned.

## ERRORS

The `putenv()` function may fail if:

**ENOMEM** Insufficient memory was available.

## USAGE

The `putenv()` function can be safely called from multithreaded programs. Caution must be exercised when using

this function and `getenv(3C)` in multithreaded programs. These functions examine and modify the environment list, which is shared by all threads in a program. The system prevents the list from being accessed simultaneously by two different threads.

It does not, however, prevent two threads from successively accessing the environment list using `putenv()` or `getenv()`.

#### SEE ALSO

`exec(2)`, `getenv(3C)`, `malloc(3C)`, `attributes(5)`, `environ(5)`

#### WARNINGS

The `_s_t_r_ i_n_g` argument should not be an automatic variable. It should be declared static if it is declared within a function because it cannot be automatically declared. A potential error is to call `putenv()` with a pointer to an automatic variable as the argument and to then exit the calling function while `_s_t_r_ i_n_g` is still part of the environment.

#### C Library Functions

`getenv(3C)`

#### NAME

`getenv` - return value for environment name

#### SYNOPSIS

```
#include <stdlib.h>
```

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

#### DESCRIPTION

The `getenv()` function searches the environment list (see `environ(5)`) for a string of the form `_n_a_m_e=_v_a_l_u_e` and, if the string is present, returns a pointer to the `_v_a_l_u_e` in the current environment.

#### RETURN VALUES

If successful, `getenv()` returns a pointer to the `_v_a_l_u_e` in the current environment; otherwise, it returns a null pointer.

#### USAGE

The `getenv()` function can be safely called from a multithreaded application. Care must be exercised when using both `getenv()` and `putenv(3C)` in a multithreaded application. These functions examine and modify the environment list, which is shared by all threads in an application. The system prevents the list from being accessed simultaneously by two different threads. It does not, however, prevent two threads from successively accessing the environment list using `getenv()` or `putenv(3C)`.