NAME

`ialloc` – Allocates storage for a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/file.h>
#include <unistd.h>
long ialloc (int fildes, long nb, int flag, int part, long *avl);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `ialloc` system call provides the means to preallocate storage for a file with certain user-specified constraints. These include mandatory contiguous storage and the partition of the file system in which to begin the search. The new space is allocated at the end of the file.

The `ialloc` system call accepts the following arguments:

fildes Specifies a file descriptor. It is obtained from a `creat`(2), `dup`(2), `fcntl`(2), or `open`(2) system call.

nb Specifies the number of bytes to allocate.

flag Controls allocation. The following are valid values for *flag*:

- `IA_CONT` Allocates contiguous storage only; if unavailable, returns error.
- `IA_PART` Allocates partition specified by *part*. If *cbits* was specified at file creation time (see `open`(2)), allocates space on the partitions specified by that argument.
- `IA_BEST` If all the blocks cannot be allocated as specified, allocates as much as possible.
- `IA_RAVL` If allocation is successful, stores number of bytes actually allocated at *avl*. If `IA_CONT` is set and allocation is unsuccessful, stores maximum number of bytes that could have been allocated at *avl*.

part Specifies the partition in which allocation is attempted.

avl Points to where the number of bytes actually allocated is stored, if `IA_RAVL` is specified.

NOTES

The process must be granted write permission to the file via the security label. That is, the active security label of the process must be equal the security label of the file.

A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_MAC_WRITE	The process is granted write permission to the file via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted write permission to the file via the security label.

RETURN VALUES

If ialloc completes successfully, *nb* is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The ialloc system call fails if one of the following error conditions occurs:

Error Code	Description
EBADF	The *fildes* argument is not a valid file descriptor open for writing, or *fildes* is not a regular file in the native file system (NC1FS or SFS).
EBADF	The security label of the process does not equal the security label of the file, and the process does not have appropriate privilege.
EFAULT	*avl* points outside the program address space.
EFBIG	An attempt was made to allocate a file that exceeds the file size limit or the maximum file size of the process. See ulimit(2).
EINVAL	*flag* value not defined.
ENOSPC	During the allocation, no free space was found in the file system.
EQACT	A file or inode quota limit was reached for the current account ID.
EQGRP	A file or inode quota limit was reached for the current group ID.
EQUSR	A file or inode quota limit was reached for the current user ID.

FORTRAN EXTENSIONS

The ialloc system call can be called from Fortran as a function:

```
INTEGER fildes, nb, flag, part, avl, IALLOC, I
I = IALLOC (fildes, nb, flag, part, avl)
```

Alternatively, ialloc can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
INTEGER fildes, nb, flag, part, avl
CALL IALLOC (fildes, nb, flag, part, avl)
```

The Fortran program must not specify both the subroutine call and the function reference to `ialloc` from the same procedure.

EXAMPLES

The following examples illustrate different uses of the `ialloc` system call.

Example 1: This `ialloc` request attempts to preallocate 10 data blocks (4096 bytes each) to the newly created file `test_file`. If the requested amount of contiguous space is unavailable, the request fails.

```
#define BLK_SZ  4096

int fd;

fd = open("test_file", O_WRONLY | O_CREAT, 0644);

if (ialloc(fd, 10*BLK_SZ, IA_CONT, 0, (long *) 0) == -1) {
     perror("ialloc failed to allocate 10 blocks contiguously");
     exit(1);
}
```

Example 2: This `ialloc` request attempts to preallocate 100,000 bytes to the newly created file `datafile` in the third partition of the file system in which the file resides. If insufficient space exists to allocate the file in this partition, the allocation is attempted in other partitions of the file system. (File system partitions are numbered 0 –*n*.) A contiguous allocation is not required since the `IA_CONT` flag is not specified.

```
int fd;

fd = open("datafile", O_WRONLY | O_CREAT, 0600);

if (ialloc(fd, 100000, IA_PART, 2, (long *) 0) == -1) {
     perror("ialloc failed for file datafile");
     exit(1);
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `ialloc` system call

SEE ALSO

`creat`(2), `dup`(2), `fcntl`(2), `open`(2), `ulimit`(2)

NAME

`ioctl` – Controls device

SYNOPSIS

```
#include <sys/ioctl.h>
int ioctl (int fildes, int request, int arg);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `ioctl` system call performs a variety of functions on character special files (devices). It accepts the following arguments:

fildes Specifies a file descriptor of a special file. It is obtained from an `accept`(2), `dup`(2), `fcntl`(2), `open`(2), `socket`(2), or `socketpair`(2) system call

request Specifies a command to be issued to the device driver.

arg Specifies an argument to the *request* command passed to the device driver.

The descriptions of various devices in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014, discuss how `ioctl` applies to them.

NOTES

Only a process with appropriate privilege can control a restricted device.

To retrieve information about certain devices, the active security label of the process must be greater than or equal to the security label of the device file.

To set information about certain devices, the active security label of the process must equal the security label of the device file.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_IO`	The process is allowed to control a restricted device.
`PRIV_MAC_READ`	The process is allowed to retrieve information about certain devices regardless of the security label of the device file.
`PRIV_MAC_WRITE`	The process is allowed to set information about certain devices regardless of the security label of the device file.

If the PRIV_SU configuration option is enabled, the super user is allowed to override the security label restrictions.

If the PRIV_SU configuration option is enabled, the super user is allowed to control a restricted device.

RETURN VALUES

If ioctl completes successfully, it returns an integer value that depends on the device control function; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The ioctl system call fails if one of the following error conditions occurs:

Error Code	Description
EBADF	The *fildes* argument is not a valid open file descriptor.
EBADF	The process does not meet security label requirements and does not have appropriate privilege.
EINVAL	The *request* or *arg* argument is not valid.
ENOTTY	The *fildes* argument is not associated with a character special device.
EPERM	The process does not have appropriate privilege to control a restricted device.

For information about specific devices, see the appropriate entry in section 4 in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR−2014.

FORTRAN EXTENSIONS

The ioctl system call can be called from Fortran as a function:

```
INTEGER fildes, request, arg, IOCTL, I
I = IOCTL (fildes, request, arg)
```

Alternatively, ioctl can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
INTEGER fildes, request, arg
CALL IOCTL (fildes, request, arg)
```

The Fortran program must not specify both the subroutine call and the function reference to ioctl from the same procedure.

EXAMPLES

The following examples illustrate how to use the `ioctl` system call to control two (terminals and CPUs) of the many character devices that `ioctl` can control.

Example 1: Terminals typically operate in line mode, meaning that a process reading data from the terminal (like a shell program) does not receive any data until the user enters a line terminator (usually a CR character).

This program disables that characteristic such that after the user enters only 2 characters, the reading process receives the 2 characters.

For additional information on this topic and other capabilities, refer to `termio`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014.

```
#include <termio.h>

main()
{
     struct termio termo, termn;

     if (ioctl(1, TCGETA, &termo) == -1) {
          perror("ioctl (TCGETA) failed getting terminal parameters");
          exit(1);
     }

     termn = termo;            /* copy old terminal parameters to new */
     termn.c_lflag &= ~ICANON; /* disable canonical terminal mode */
     termn.c_cc[VMIN] = 2;     /* characters sent after 2 typed */
     termn.c_cc[VTIME] = 100;  /* specify 10 second delay between keystrokes */

     if (ioctl(1, TCSETA, &termn) == -1) {
          perror("ioctl (TCSETA) failed setting new terminal parameters");
          exit(1);
     }

     /* After the ioctl request changed the terminal parameters,
        the user interface changed.  The terminal is no longer in
        line mode (canonical).  After 2 keystrokes with or without
        a CR character, the 2 characters are delivered to the reading
        process. */

     /* Before the program terminates, the terminal's parameters are restored
        to their original state. */

     if (ioctl(1, TCSETA, &termo) == -1) {
          perror("ioctl (TCSETA) failed resetting terminal parameters");
          exit(1);
     }
}
```

Example 2: An `ioctl` system call can control a CPU in a variety of ways. This `ioctl` request causes the CPU to interrupt the currently running program with a `SIGALRM` signal at regularly scheduled intervals (measured in milliseconds). The program catches each signal at the defined intervals and continues.

For additional information on this topic and other capabilities, refer to `cpu`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014.

```
#include <fcntl.h>
#include <sys/cpu.h>
#include <signal.h>
#include <time.h>

long before, after;

main()
{
     int fd;
     struct cpudev cpudev;
     void catch(int signo);

     (void) signal(SIGALRM, catch);

     fd = open("/dev/cpu/any", O_RDONLY);

     cpudev.word = 10000;  /* set interval to 10 seconds (10,000 mill) */
     if(ioctl(fd, CPU_SETTMR, &cpudev) == -1) {
          perror("ioctl failed");
          exit(1);
     }

     before = rtclock();   /* read the real time clock */

     for(;;) {             /* loop indefinitely waiting for SIGALRMs
                              every 10 sec */
     }                     /* kill with <ctrl><C> */
}

void catch(int signo)
{
     float time;

     (void) signal(signo, catch);

     after = rtclock();                                  /* read the real
                                                            time clock */
     time = (float) (after - before) / (float) CLK_TCK;  /* compute seconds */
     printf("Caught signal #%d after %f seconds\n", signo, time);
}
```

FILES

`/usr/include/sys/ioctl.h`	Contains C prototype for the `ioctl` system call

SEE ALSO

`accept`(2), `dup`(2), `fcntl`(2), `open`(2), `socket`(2), `socketpair`(2)

`cpu`(4), `termio`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`jacct` – Enables or disables job accounting

SYNOPSIS

```
#include <unistd.h>
int jacct (char *path);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `jacct` system call enables or disables job accounting for the calling job (see `setjob`(2)). If job accounting is enabled, an accounting record is written on a job accounting file for each of the job's processes that terminates. If daemon accounting is enabled, daemon accounting records are also written to this file. An `exit`(2) call or a signal can cause termination. Any process member of a job may use the `jacct` call.

The `jacct` system call accepts the following argument:

path Points to a path name that contains the job accounting file. `acct`(5) and `/usr/include/acct/dacct.h` describe the types of records found in this file.

Job accounting is enabled if *path* is nonzero and no errors occur during the system call. It is disabled if *path* is 0 and no errors occur during the system call.

If job accounting is already enabled and *path* differs from the job accounting file currently in use, the job accounting file will be switched to *path* without the loss of any accounting information.

NOTES

To be granted write permission to the file, the active security label of the process must equal the security label of the file.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path prefix via the permission bits and access control list.
`PRIV_MAC_READ`	The process is granted search permission to a component of the path prefix via the security label.
`PRIV_MAC_READ`	The process is granted read permission to the file via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted read permission to the file via the security label.

RETURN VALUES

If jacct completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The jacct system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A component of the *path* prefix denies write permission.
EACCES	The file specified by *path* is not an ordinary file.
EACCES	The write permission is denied for the specified job accounting file.
EFAULT	The *path* argument points to an illegal address.
EINVAL	The calling process is not a member of a job.
EISDIR	The specified file is a directory.
ENOENT	One or more components of the job accounting file path name do not exist.
ENOTDIR	A component of the *path* prefix is not a directory.
EROFS	The specified file resides on a read-only file system.

FILES

/usr/include/unistd.h	Contains C prototype for the jacct system call

SEE ALSO

acct(2), dacct(2), exit(2), setjob(2), signal(2)

acct(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

join, fjoin – Joins files

SYNOPSIS

```
#include <unistd.h>
int join (char *path1, char *path2);
int fjoin (int fildes1, int fildes2);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The join system call concatenates the data blocks of one file to another. The join system call accepts the following arguments:

path1 Points to the file to which you want to add data blocks.

path2 Points to the file from which you want to take data blocks away.

Data blocks are not copied from the file referenced by *path2* to that referenced by *path1*; rather the address descriptors in the inode for the second file are appended to the address descriptors in the inode for the first.

Any allocated, but unused data blocks at the end of the file identified by *path1* are deallocated prior to the addition of data blocks from address descriptors in the inode for the file identified by *path2*. The length of the file identified by *path2* is truncated to 0 by the system call.

The fjoin system call performs the same operation as join with the specified files. The fjoin system call accepts the following arguments:

fildes1 Specifies the file descriptor of the file to which you want to add data blocks.

fildes2 Specifies the file descriptor of the file from which you want to take data blocks away.

RETURN VALUES

If join or fjoin completes successfully, 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The join system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A component of the *path1* or *path2* prefix denies search permission.
EACCES	Write permission is denied for the file specified by *path1* or *path2*.

EAGAIN	Mandatory file and record locking is set, and there are outstanding record locks on one of the files (see chmod(2)).
EFAULT	The *path1* or *path2* argument points outside the allocated process address space.
EINVAL	*path1* and *path2* identify the same file.
EINVAL	Files reside on different file systems.
EINVAL	*path1* or *path2* does not identify a regular file.
EINVAL	The length of file identified by *path1* is not an even multiple of 4096 bytes, not an even multiple of the physical I/O unit size of the device on which the file system resides, nor an even multiple of the file system allocation unit size of the partition on which the file resides.
ENOENT	The file identified by *path1* or *path2* does not exist.
ENOTDIR	A component of the *path1* or *path2* prefix is not a directory.

The fjoin system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	Write permission is denied for the file specified by *fildes1* or *fildes2*.
EAGAIN	Mandatory file and record locking is set, and there are outstanding record locks on one of the files (see chmod(2)).
EBADF	The calling process does not have MAC read access to the file to which the file descriptor refers.
EINVAL	*fildes1* and *fildes2* identify the same file.
EINVAL	Files reside on different file systems.
EINVAL	*fildes1* or *fildes2* does not identify a regular file.
EINVAL	The length of file identified by *fildes1* is not an even multiple of 4096 bytes, not an even multiple of the physical I/O unit size of the device on which the file system resides, or an even multiple of the file system allocation unit size of the partition on which the file resides.

FILES

/usr/include/unistd.h	Contains C prototype for the join and fjoin system calls

SEE ALSO

chmod(2)

mkfs(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

kill, killm, _lwp_kill, _lwp_killm – Sends a signal to a process or a group of processes

SYNOPSIS

All Cray Research systems:

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

int kill (pid_t pid, int sig);

#include <sys/category.h>
#include <signal.h>

int killm (int category, int id, int sig);
```

Cray PVP systems:

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

int _lwp_kill (pid_t pid, int sig);

#include <sys/category.h>
#include <signal.h>

int _lwp_killm (int category, int id, int sig);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to kill)

DESCRIPTION

The kill system call sends a signal to a process or a group of processes. The kill and _lwp_kill system calls accept the following arguments:

pid Specifies the process or group of processes to which the signal is to be sent.

sig Specifies the signal that is to be sent. Specify either 0 or one of the values for the *sig* argument for the signal(2) system call. If *sig* is 0 (the null signal), error checking is performed, but no signal is sent. You can use this to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the sending process has appropriate privilege. The caller can send a SIGCONT signal to any process within its session, regardless of the process owner.

The processes with a process ID of 0 and a process ID of 1 are special processes (see intro(2)), and they are referred to as proc0 and proc1, respectively, in the following conditions:

- If *pid* is greater than 0, *sig* is sent to the process that has a process ID equal to *pid*; *pid* can equal 1.
- If *pid* equals 0, *sig* is sent to all processes, excluding proc0 and proc1, whose process group ID is equal to the process group ID of the sender.
- If *pid* equals −1 and the sending process has appropriate privilege, *sig* is sent to all processes excluding proc0 and proc1.
- If *pid* equals −1 and the sending process does not have appropriate privilege, *sig* is sent to all processes, excluding proc0 and proc1, whose real user ID is equal to the effective user ID of the sender.
- If *pid* is negative but not −1, *sig* is sent to all processes whose process group ID is equal to the absolute value of *pid*.

The killm system call sends a signal to a process or a group of processes. The killm and _lwp_killm system calls accept the following arguments:

category Specifies C_PROC, C_PGRP, C_ALL, C_UID, or C_JOB. A category of C_ALL is available only when the sending process has the appropriate privilege.

id Specifies the *pid, pgrp, jid,* or *uid* corresponding to the *category*. An *id* of 0 means all processes in the current *category*.

sig Identifies the signal to be sent. See signal(2) for *sig* values.

Currently the _lwp_kill and _lwp_killm interfaces are synonyms for kill and killm, respectively. In a future release, this will change (see the NOTES section).

If the target of a kill or killm system call is an MPP application, each processing element (PE) in the application receives the signal.

NOTES

For multitasked applications, the kill and killm system calls treat the entire multitasking group as the target of a signal. More specifically, the system decides which member of the multitasking group receives the signal and the *pid* argument is not considered significant in this choice (although it may introduce some bias in the selection).

In contrast, the _lwp_kill and _lwp_killm system calls do consider the *pid* argument as an explicit specifier of the receiver of a signal. However, use of these calls is discouraged since they may disappear in future releases of the UNICOS operating system.

Signals are not allowed to cross security label boundaries unless the sending process has privilege to override the system mandatory access control (MAC) policy. If an unprivileged process attempts to send a signal to another process that has a different security label, an ESRCH error status is returned.

The active security label of the process must equal the security label of every affected process.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_KILL	The process is considered the owner of all affected processes.
PRIV_MAC_WRITE	The active security label of the process is considered to equal the security label of all affected processes.

If the PRIV_SU configuration option is enabled, the super user is considered the owner of all affected processes. If the PRIV_SU configuration option is enabled, the super user overrides all security label restrictions.

RETURN VALUES

If kill, killm, _lwp_killm, or _lwp_killm completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The kill or killm system call fails and no signal is sent if one of the following error conditions occurs:

Error Code	Description
EINVAL	The *sig* argument is not a valid signal number.
EPERM	The *pid* argument is 1 (proc1), and *sig* is either SIGKILL or SIGSTOP.
EPERM	The process does not have appropriate privilege, and its real or effective user ID does not match the real or effective user ID of the receiving process.
ESRCH	The active security label of the process does not equal those of a receiving process, and the process does not have appropriate privilege.
ESRCH	No process can be found corresponding to that specified by *pid*.

FORTRAN EXTENSIONS

The kill system call can be called from Fortran as a function:

```
INTEGER pid, sig, KILL, I
I = KILL (pid, sig)
```

Alternatively, kill can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable:

```
INTEGER pid, sig
CALL KILL (pid, sig)
```

The Fortran program must not specify both the subroutine call and the function reference to `kill` from the same procedure.

EXAMPLES

The following examples illustrate the use of the `kill` system call and `killm`, the Cray Research extension. Each example entails sending a `SIGUSR1` signal to the parent of the calling process:

Example 1: The `kill` request sends a `SIGUSR1` signal to the parent process:

```
int ppid;

ppid = getppid();
if (kill(ppid, SIGUSR1) == -1) {
     perror("kill failed sending SIGUSR1 to parent");
     exit(1);
}
```

Example 2: The `killm` request sends a `SIGUSR1` signal to the parent process:

```
int ppid;

ppid = getppid();
if (killm(C_PROC, ppid, SIGUSR1) == -1) {
     perror("killm failed sending SIGUSR1 to parent");
     exit(1);
}
```

SEE ALSO

`getpid`(2), `intro`(2), `setpgrp`(2), `signal`(2)

`kill`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`limit` – Sets resource limits

SYNOPSIS

```
#include <sys/category.h>
#include <sys/resource.h>
long limit (int category, int id, int resource, long newlimit);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `limit` system call establishes limits on resource usage or returns information on resource limits for a process or job. It accepts the following arguments:

category Selects the resource category. The following are valid resource categories:

- `C_JOB` Job limit.
- `C_JOBPROCS` Process limit of all processes associated with job *jid*.
- `C_PROC` Process limit.

id Specifies the *pid* or *jid* corresponding to the *category*. A *pid* of 0 means the current process, and a *jid* of 0 means the current job.

resource Selects the resource. The following are valid resources:

- `L_CORE` Maximum core file limit (clicks). A *newlimit* value less than the process size will result in a truncated core file consisting of the user common structure and the user area. A value of `NO_CORE_FILES` will disable the creation of core files altogether. This limit is supported only for the `C_PROC` category.
- `L_CPROC` Maximum number of processes that can exist concurrently within a job. This limit is supported only for the `C_JOB` category.
- `L_CPU` Maximum CPU time per category (clocks). If a process exceeds the process limit, `SIGCPULIM` is sent. The parent shell recognizes the death of the process and sends an error message to standard error. If the job limit is exceeded, `SIGCPULIM` is sent to all processes in the job, which includes the parent shell. Processes may register to catch this signal and continue, but `SIGKILL` is sent a few seconds later. (See the CAUTIONS section.)

L_FD	Maximum number of open files that children of this process will have when created. This limit is supported only for the C_PROC category. If the new limit is less than the value of OPEN_MAX (64), the limit will be set to OPEN_MAX and no error will be returned. The specified limit must be less than that set by the L_FDM resource or the system-imposed open file maximum value of K_OPEN_MAX.
L_FDM	Maximum limit on the L_FD resource setting (the minimum limit is set by OPEN_MAX). Changing the L_FDM resource does not affect the open file maximum of any processes. Rather, it affects the open file maximum of any future child processes by limiting the maximum L_FD resource specification. This limit is supported only for the C_PROC category. If the new limit is less than the specified process' current open file maximum, limit will fail with an EINVAL error status. If the new limit is greater than the system imposed K_OPEN_MAX open file limit, limit will set the limit to K_OPEN_MAX.
L_FSBLK	Maximum number of file system blocks (clicks) that can be used per category. If a process tries to exceed established process or job limits, an EDISKLIM error is returned.
L_MEM	Maximum memory size per category (clicks). If a process tries to exceed established process or job limits, the brk(2) or sbrk(2) system call fails and returns the ENOMEM error.
L_MPPB	(Deferred implementation) Maximum number of Cray MPP synchronization barriers. This limit is supported only for the C_JOB category.
L_MPPE	Maximum number of Cray MPP processing elements (PEs). If this limit is set to 0, the Cray MPP systems cannot be used by the job. This limit is supported only for the C_JOB category.
L_MPPT	Maximum number of wall clock seconds that the job can have the Cray MPP systems assigned to it. If this limit is set to 0, the Cray MPP systems cannot be used by the job. This limit is supported for all categories.
L_SDS	Maximum number of secondary data blocks per category. It is enforced by the system on an ssbreak(2) call. Because the minimum allocation unit for secondary data segments (SDS) may be greater than 1 block, the limit set may be exceeded by a fraction of the minimum allocation unit.
L_SOCKBF	Maximum total socket buffer (sockbuf) space per session. The per session sockbuf space is the sum of the sockbuf space reserved by all of the sockets used by the session. The limit is in clicks (4096 bytes per click). This limit is enforced on the accept(2), setsockopt(2), and socket(2) calls.
L_TAPE	Maximum number of tape devices from tape group 0 for the job. This is a synonym for L_TAPE0 and will be removed in a future release. It is enforced by the tape daemon. This limit is supported only for the C_JOB category.

	L_TAPE*n*	Maximum number of tape devices from tape group *n* (which can be 0 through 7) for the job. It is enforced by the tape daemon. This limit is supported only for the C_JOB category.

newlimit Specifies a new limit. *newlimit* is one of the following:

–1	Limit unchanged; current limit value is returned.
≥0	New limit value. For all limits except tape group limits, L_SDS, L_MPPB, L_MPPE, and L_MPPT, a value of 0 means no limit.
–2	For L_CORE only. This special value disables the creation of core files.

Any process can make a limit more restrictive, but only a process with appropriate privilege can make a limit less restrictive. Limits are inherited by child processes.

NOTES

The following mandatory access control (MAC) read and MAC write checks are performed based on the *category* parameter:

Parameter	**Description of check**
C_PROC	Against the specified process
C_JOBPROCS	Against each process in the job
C_JOB	Against the job leader

That is, the active security label of the calling process must equal the security label of each process where access is being verified.

To set process resource information, the active security label of the calling process must equal the security label of every affected process.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
PRIV_MAC_READ	The calling process is allowed to override the restriction that its active security label must be greater than or equal to the security label of every affected process.
PRIV_MAC_WRITE	The calling process is allowed to set resource information regardless of the security label of the target process.
PRIV_POWNER	The process is considered the owner of every affected process.
PRIV_RESOURCE	The calling process is allowed to increase the value of a limit.

If the PRIV_SU configuration option is enabled, the super user is considered the owner of every affected process and is allowed to increase the value of a limit. If the PRIV_SU configuration option is enabled, the super user is allowed to override security label restrictions.

CAUTIONS

The CPU time limit does not apply when running as root.

RETURN VALUES

If `limit` completes successfully, the previous value of `limit` is returned for categories `C_PROC` and `C_JOB`, and 0 is returned for `C_JOBPROCS`; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `limit` system call fails if one of the following error conditions occurs:

Error Code	Description
EBADF	The process does not meet security label requirements and does not have appropriate privilege.
EINVAL	One of the arguments contains an invalid value.
EPERM	The user ID of the requesting process is not that of a super user, and its real or effective user ID does not match the real or effective user ID of the affected processes.
EPERM	An attempt was made to increase the value of a limit, and the user ID of the requesting process is not that of a super user.
EPERM	An attempt was made to change a limit on a recovered job, recovered process, or a system process; this is not allowed.
ESRCH	No process can be found that matches the *category* and *id* requests.

FORTRAN EXTENSIONS

The `limit` system call can be called from Fortran as a function:

```
INTEGER category, id, resource, newlimit, LIMIT, I
I = LIMIT (category, id, resource, newlimit)
```

Alternatively, `limit` can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
INTEGER category, id, resource, newlimit
CALL LIMIT (category, id, resource, newlimit)
```

The Fortran program must not specify both the subroutine call and the function reference to `limit` from the same procedure.

EXAMPLES

The following example shows how to use the `limit` system call to return the current maximum CPU time and memory size limits for the calling process. Neither `limit` request changes the CPU time or memory size limit because of the argument value −1 specified.

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

main()
{
     long time, mem;

     time = limit(C_PROC, 0, L_CPU, -1);
     printf("The current CPU time limit is %ld clock ticks or %ld sec\n",
             time, time/CLK_TCK);        /* CLK_TCK defined in time.h> */

     mem = limit(C_PROC, 0, L_MEM, -1);
     printf("The current memory limit is %ld blocks, %ld words, or %.1f Mw\n",
             mem, mem * 512, (mem * 512)/1048576.);
}
```

SEE ALSO

`accept`(2), `brk`(2), `setsockopt`(2), `signal`(2), `socket`(2), `ssbreak`(2), `ulimit`(2)

`limit`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR−2011

NAME

limits – Returns or sets limits structure for fair-share scheduler

SYNOPSIS

```
#include <sys/types.h>
#include <sys/lnode.h>
#include <sys/param.h>
#include <sys/iosw.h>
#include <sys/signal.h>
#include <sys/dir.h>
#include <sys/perm.h>
#include <sys/retlim.h>
#include <sys/share.h>

int limits (struct lnode *address, int function);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The limits system call manipulates a kernel limits structure according to the value of *function*. limits accepts the following arguments:

address Points to an lnode or an array of lnodes except where indicated.

function Specifies a function. *function* may be one of the following:

- L_MYLIM — Gets user's own limits structure. Returns the number of processes currently attached to the node.
- L_OTHLIM — Gets limits associated with *uid* in lnode. The lnode to which *address* points must contain the correct user ID. Returns the number of processes currently attached to the node.
- L_ALLLIM — Returns the number of active lnodes, along with all active user limits structures.
- L_SETLIM — Connects to a new limits structure. Initializes a new limits structure with the passed lnode, and attaches the calling process to it. All children of that process inherit the new structure. Only a process with appropriate privilege can specify this function.
- L_NEWLIM — Same as L_SETLIM, but attaches parent processes rather than calling processes. Only a process with appropriate privilege can specify this function.

L_DEADLIM	Waits for dead limits belonging to a child process. The *address* should point to a `retlim` structure, which is defined in the `sys/retlim.h` file. This function performs a `wait`(2) system call, then returns a structure containing both the limits and process zombie structures. The value returned is the number of processes still attached to the lnode.
L_CHNGLIM	Changes `limits` fields in existing limits. The lnode to which *address* points must contain the correct user ID. Only a process with appropriate privilege can specify this function. NOTE: This function updates the CPU quota-used field. To synchronize the information in active lnodes with the user database (UDB), execute the `shrsync`(8) command with the `-q` option.
L_DEADGROUP	Picks up a dead limits structure. This function searches for a dead limits structure, removes it from the list of active limits, and returns `lnode`. Only a process with appropriate privilege can specify this function.
L_MSGSON	Enables system messages to a user.
L_MSGSOFF	Disables system messages to a user; default is enabled.
L_MSGSSTAT	Returns the status of system messages; if system messages are enabled, a nonzero number is returned.
L_MYKN	Gets a user's own `kern_lnode` structure. The *address* should point to a `kern_lnode` structure, which is defined in the `sys/lnode.h` file. Returns the number of processes currently attached to the node.
L_OTHKN	Gets the structure associated with *uid*. The *address* should point to a `kern_lnode` structure, which is defined in the `sys/lnode.h` file. The `kern_lnode` to which *address* points returns the number of processes currently attached to the node.
L_ALLKN	Returns the number of active lnodes, along with all active kernel structures. The *address* should point to a `kern_lnode` structure, which is defined in the `sys/lnode.h` file.
L_SETIDLE	Sets `limits` fields for idle lnode, which is initialized at system boot time; otherwise, it acts as `L_SETLIM` does. Only a process with appropriate privilege can specify this function.

Any other *function* is illegal and returns an error of `EINVAL`. Unless otherwise specified, the call returns the number of limits structures returned.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_RESOURCE	The process is allowed to specify the L_SETLIM, L_CHNGLIM, L_DEADGROUP, and L_SETIDLE functions.

If the PRIV_SU configuratino option is enabled, the super user or a process with the PERMBITS_RESLIM permbit is allowed to specify the L_SETLIM, L_CHNGLIM, L_DEADGROUP, and L_SETIDLE functions.

The L_GETCOSTS and L_SETCOSTS functions have been removed in the UNICOS 9.0 release. Their functionality has been replaced by the GET_COSTS and SET_COSTS actions, respectively, of the policy(2) system call.

RETURN VALUES

If limits completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The limits system call fails if one of the following error conditions occurs:

Error Code	Description
EINVAL	An illegal argument was passed to the system call.
EPERM	For functions L_SETLIM, L_CHNGLIM, L_DEADGROUP, and L_SETIDLE, this error indicates that the process does not have appropriate privilege.
EPROCLIM	The user already has the maximum number of processes active (valid for function L_SETLIM).
ESRCH	For functions L_DEADGROUP, L_OTHKN, L_OTHLIM, and L_CHNGLIM, this error indicates that the desired limits structure does not exist. For function L_SETLIM, this error indicates that this lnode's group has not been set up.
ETOOMANYU	No space is left in the kernel limits table (valid for function L_SETLIM).

SEE ALSO

policy(2), wait(2)

share(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

shrsync(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

UNICOS Resource Administration, Cray Research publication SG–2302

NAME

`link` – Creates a link to a file

SYNOPSIS

```
#include <unistd.h>
int link (const char *path1, const char *path2);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `link` system call creates a new link (directory entry) for the existing file. Its accepts the following arguments:

path1 Points to a path name identifying an existing file.

path2 Points to a path name identifying the directory entry to be created.

NOTES

The process must be granted search permission to every component of each path prefix via the permission bits and access control list. The process must be granted search permission to every component of each path prefix via the security label.

The process must be granted write permission to the parent directory of *path2* via the permission bits and access control list. The process must be granted write permission to the parent directory of *path2* via the security label.

If *path1* is a directory, the process must have appropriate privilege to create a link.

If `FSETID_RESTRICT` is enabled, only processes with appropriate privileges can create a link to set-user-ID or set-group-ID files.

A process with the effective privileges shown are granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of each path prefix via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted write permission to the parent directory of *path2* via the permission bits and access control list.

PRIV_FSETID	If FSETID_RESTRICT is enabled, the process can create a link to the set-user-ID or set-group-ID file.
PRIV_LINK_DIR	The process can create a link to a directory.
PRIV_MAC_READ	The process is granted search permission to every component of each path prefix via the security label.
PRIV_MAC_WRITE	The process is granted write permission to the parent directory of *path2* via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of each path prefix and is granted write permission to the parent directory of *path2*. The super user can create a link to a directory. The super user or a process with the suidgid permission can override the restriction enabled by the FSETID_RESTRICT system configuration option.

RETURN VALUES

If link completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The link system call fails and no link is created if one of the following error conditions occurs:

Error Code	Description
EACCES	A component of either path prefix denies search permission.
EACCES	The requested link requires writing in a directory with a mode that denies write permission.
EEXIST	The link specified by *path2* exists.
EFAULT	The *path* argument points outside the allocated address space of the process.
EMANDV	The security label of the file does not allow linking.
EMANDV	If the FSETID_RESTRICT and PRIV_SU configuration options are enabled, the process does not have appropriate privileges to link to a set-user-ID or set-group-ID file.
EMLINK	The maximum number of links to a file (LINK_MAX) would be exceeded.
ENOENT	A component of either path prefix does not exist.
ENOENT	The file specified by *path1* does not exist.
ENOENT	The *path2* argument points to a null path name.
ENOTDIR	A component of either path prefix is not a directory.
EPERM	The file specified by *path1* is a directory, and the effective user ID is not that of a super user.

EQACT	A file or inode quota limit was reached for the current account ID.
EQGRP	A file or inode quota limit was reached for the current group ID.
EQUSR	A file or inode quota limit was reached for the current user ID.
EROFS	The requested link requires writing in a directory on a read-only file system.
EXDEV	The link specified by *path2* and the file named by *path1* are on different logical devices (file systems).

FORTRAN EXTENSIONS

The `link` system call can be called from Fortran as a function (on all systems except Cray MPP systems and CRAY T90 series systems):

```
CHARACTER path1*M, path2*N
INTEGER LINK, I
I = LINK (path1, path2)
```

Alternatively, `link` can be called from Fortran as a subroutine (on all systems except Cray MPP systems and CRAY T90 series systems). In this case, the return value of the system call is unavailable.

```
CHARACTER path1*M, path2*N
CALL LINK (path1, path2)
```

The Fortran program must not specify both the subroutine call and the function reference to `link` from the same procedure. *path1* and *path2* may also be integer variables. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte. The `PXFLINK`(3F) subroutine provides similar functionality and is available on all Cray Research systems.

EXAMPLES

This example shows how to use the `link` system call to create a link to another user's file. The following `link` request creates a link (called `joe_file`) to user `joe`'s file, `datafile`.

The request causes a new directory entry named `joe_file` to be created and the inode for `joe`'s `datafile` to reflect this additional link.

If user `joe` later removes (by using `rm`(1) or `unlink`(2)) `datafile`, the file is not actually removed from the file system since another user now has a link to it. The file cannot be removed until the last link to the file is removed.

```
if (link("../joe/datafile", "joe_file") == -1) {
     perror("link failed creating new link to joe's datafile");
     exit(1);
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `link` system call

SEE ALSO

`unlink`(2)

`rm`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`PXFLINK`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

`link`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`listen` – Listens for connections on a socket

SYNOPSIS

```
int listen (int s, int backlog);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

To accept connections, a socket must first be created with `socket`(2). A backlog for incoming connections is then specified with `listen`, and the connections are accepted with `accept`(2). The `listen` call applies only to sockets of type `SOCK_STREAM`.

Connection requests for an address go into the connection queue for the socket bound to the specified address. The `accept`(2) system call removes connection requests from the queue.

The `listen` system call accepts the following arguments:

s Specifies the descriptor for a socket.

backlog Defines the maximum length to which the queue of pending connections can grow. If a connection request arrives with the queue full, the client might receive an error with an indication of `ECONNREFUSED`, or if the underlying protocol supports retransmission, the request might be ignored so that retries can succeed.

The kernel has a limit for the maximum length of the queue of pending connections. If the *backlog* parameter exceeds that limit, the kernel limit is used instead. However, the kernel also allows some number of connections beyond the queue limit to be accepted, to allow for transient connections that never get established fully.

Note: The maximum limit is defined by `SOMAXCONN` in the `sys/socket.h` file. Currently, the maximum for queued pending connections is (*backlog* * 3)/2 +1.

NOTES

If the `SOCKET_MAC` option is enabled, the active security label of the process must be greater than or equal to the security label of the socket. Note that `SOCKET_MAC` is part of TCP/IP configurable feature variables list in `uts/cf/Nmakefile`. For more information, see the `connect`(2) man page.

A process with the effective privilege shown is granted the following ability:

Privilege	Description
`PRIV_MAC_READ`	The process is allowed to override the security label restrictions when the `SOCKET_MAC` option is enabled.

If the PRIV_SU configuration option is enabled, the super user is allowed to override security label restrictions when the SOCKET_MAC option is enabled.

RETURN VALUES

If listen completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The listen system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	If the SOCKET_MAC option is enabled, the process does not meet the security label requirements and does not have appropriate privilege.
EBADF	The *s* descriptor is invalid.
ENOTSOCK	The *s* descriptor is not a socket.
EOPNOTSUPP	The socket is not of a type that supports the operation listen (for example, the socket is of type SOCK_DGRAM).

BUGS

Currently, the backlog is limited (silently) to five pending connection requests by SOMAXCONN (SOMAXCONN = 5 is defined in the sys/socket.h include file). When the number entered in the *backlog* argument is higher than 5, no error message is issued.

EXAMPLES

This server program shows how to use the listen system call in context with other TCP/IP calls. (Some system calls in this example are not supported on Cray MPP systems.) The program simply creates a TCP/IP socket, waits for a client process from some host to attempt a connection, accepts the connection, and forks a child process to provide the service to the client.

The original (parent) server loops back to look for additional connection attempts while the temporary (child) server reads a string of data sent by the client process.

```
/*  Server side of client-server socket example. For client side,
    see socket(2).
    Syntax: server portnumber & */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>

main(int argc, char *argv[])
{
     int s, ns;
     struct sockaddr_in src;  /* source socket address */
     int len=sizeof(src);
     char buf[256];

     /* create port */
     src.sin_family = AF_INET;
     src.sin_port = atoi(argv[1]);
     src.sin_addr.s_addr = 0;

     if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
          perror("server, unable to open socket");
          exit(1);
     }

     while (bind(s, (struct sockaddr *) &src, sizeof(src)) < 0) {
          printf("Server waiting on bind...\n");
          sleep(1);
     }

     listen(s, 5);

     while (1) {
          ns = accept(s, (struct sockaddr *) &src, &len);
          if (ns < 0) {
               perror("server, accept failed");
               exit(1);
          }

          if (fork() == 0) {
               /* in child server */
               close(s);       /* child will use socket ns, parent uses s */
```

```
                read(ns, &buf, sizeof(buf));
                printf("Server read: %s\n", buf);
                close(ns);
                exit(0);
            }
            close(ns);           /* close socket used by child */
        }
    }
```

FILES

`/usr/include/sys/socket.h`	Header file for sockets

SEE ALSO

`accept`(2), `connect`(2), `socket`(2)

NAME

`listio` – Initiates a list of I/O requests

SYNOPSIS

```
#include <sys/types.h>
#include <sys/iosw.h>
#include <sys/listio.h>

int listio (int cmd, struct listreq *list, int nent);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `listio` system call provides a means to initiate a list of distinct I/O requests and, optionally, waits for all of them to complete. Each I/O request in the list provides for maximum control over the desired I/O characteristics. The `listio` system call accepts the following arguments:

cmd Specifies a command. The following are valid *cmd* commands:

- `LC_START` Initiates the I/O requests and returns control as soon as possible.
- `LC_WAIT` Initiates the I/O requests and returns when all requests have completed.

list Points to an array of `listreq` structures. Each array includes the following members:

```
int        li_opcode;    Operation code for the request:
                         LO_READ for read and LO_WRITE
                         for write.
unsigned   li_drvr:32;   Driver dependent.
unsigned   li_flags:32;  Special request flags:  LF_LSEEK
                         to set initial file offset to li_offset.
long       li_offset;    Initial file byte offset, if LF_LSEEK
                         is set in flags.
int        li_fildes;    File descriptor (obtained from a
                         creat(2), dup(2), fcntl(2),
                         open(2), or pipe(2) system call).
char       *li_buf;      Pointer to an I/O data buffer
                         in memory.
unsigned   li_nbyte;     Number of bytes to read or write
                         for each stride.
struct iosw *li_status;  Pointer to an I/O status word
                         where the kernel will put completion
                         status for this request.  See reada(2).
int         li_signo;    Signal number of signal to send
                         to the process when the request
                         completes.  If this field is 0,
                         no signal is sent (see signal(2)
                         for a list of signal numbers).
int        li_nstride;   Number of strides; defaults to 1.
                         (On Cray MPP systems, li_nstride must
                         be 0 or 1.)
long       li_filstride; File stride in bytes;
                         default is for contiguous data flow
                         to/from the file.  (On Cray MPP systems,
                         li_filstride must be 0.)
long       li_memstride; Memory stride in bytes; default is
                         for contiguous data flow
                         to/from the memory buffer.  (On Cray MPP systems,
                         li_memstride must be 0.)
```

nent Specifies the number of requests in the list to process.

When reading or writing an *n*-dimensional array on a disk, the desired data I/O occurs at regular intervals, but it may not be contiguous. The last three variables in the `listreq` structure can be used to specify a compound request, causing multiple sections of data to be transferred. The distance from the start of one section of data on disk to the start of the next section is called the *file stride*. There is an analogous stride through memory.

When a particular request completes, the associated status word is filled in, and if `li_signo` was nonzero, the signal corresponding to the number is sent to the process. In the `iosw` structure, the status word `sw_flag` is always set upon completion; `sw_error` may contain a system call error number; and `sw_count` contains the number of bytes actually moved. For a successful compound request, `sw_count` would be `li_nstride * li_nbyte`.

The following are three ways of handling I/O completions:

- When registering for a given signal by using the `sigctl`(2) system call, the process may specify 0 rather than a handler function. After initiating one or more I/O requests with that signal number and doing any other work available, the process checks for I/O completions and, finding none, goes to sleep using the `pause`(2) system call. When the next I/O completes, the process is awakened. If an I/O completes after the process checks it but before it actually goes to sleep, the `pause`(2) system call will return immediately.
- A process may register a signal handler for a given signal and specify that signal on the `listio` request. When the I/O completes, the handler will be called and the process may service the completion. This method is interrupt or event driven. See `reada`(2) for more information about this approach. Because the kernel and library must save the process' context before calling the signal handler and restore it again after the signal handler, there is some additional overhead in this approach. The process' context is A, S, and V registers and some local memory.
- A process may specify 0, rather than a signal number, on the I/O request. In this case, the process must arrange to be awakened by some other event, perhaps a timer, or through polling the status word. The `recall`(2) (on all Cray Research system) and `recalla`(2) (on Cray PVP systems) system calls may be used to wait for ending status.

If one or more of the I/O requests are ill-formed and cannot be started, an `LC_WAIT` type command will return immediately.

NOTES

To perform a write operation, the process must be granted write permission via the security label. That is, the active security label of the process must be equal to the security label of the file.

To perform a read operation, the process must be granted read permission via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_MAC_READ`	The process is granted read permission to the file via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted read and write permission to the file via the security label.

RETURN VALUES

If `listio` completes successfully, a nonnegative integer is returned, indicating the number of requests that were successfully started. Otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `listio` system call fails if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	The request list is not fully contained within the process address space.
`EINTR`	A signal was caught while waiting for all I/O requests to complete during an `LC_WAIT` command.
`EINVAL`	The *cmd* argument is not a valid command.

A particular request fails if one of the following error conditions occurs:

Error Code	Description
`EAGAIN`	Mandatory file and record locking was set, `O_NDELAY` was set, and there was a blocking record lock.
`EBADF`	The `li_fildes` value is not an open file descriptor.
`EBADF`	The process is not granted read or write permission to the file via the security label and does not have appropriate privilege.
`EDEADLK`	The request was going to go to sleep and cause a deadlock situation to occur.
`EFBIG`	An attempt was made to write a file that exceeds the process' file size limit or the maximum file size. See `ulimit`(2).
`EINTR`	An I/O request to a slow device, such as tty, was interrupted.
`EINTR`	A signal was caught waiting for an I/O quota or blocking record lock.
`EINVAL`	The `listreq` entry contains an invalid argument.
`ENOLCK`	The system record lock table was full, so the request could not go to sleep until the blocking record lock was removed.
`ENOSPC`	During a write to an ordinary file, no free space was found in the file system.
`EQACT`	A file or inode quota limit was reached for the current account ID.
`EQGRP`	A file or inode quota limit was reached for the current group ID.
`EQUSR`	A file or inode quota limit was reached for the current user ID.
`ESPIPE`	An attempt was made to specify a byte offset on a pipe or a FIFO special file (named pipe).

EXAMPLES

This example shows how to use the `listio` system call to initiate a list of input requests. The following `listio` request reads every tenth block (that is, blocks 1, 11, 21, 31, and so on) from file `datafile`. When a total of 10 data blocks is transferred, a `SIGUSR1` signal is sent to show that the transfer has been completed.

The request is initiated asynchronously (LC_START), and the data is stored in the buffer buf contiguously. Because input is performed asynchronously, the program can complete other work in parallel with the request.

```
#include <fcntl.h>
#include <sys/types.h>
#include <signal.h>
#include <sys/iosw.h>
#include <sys/listio.h>

#define BLK_SIZ 4096

struct blk {
     char    blk_data[BLK_SIZ];
};

main()
{
     struct listreq request;
     struct iosw reqstat;
     struct blk buf[10];
     int fd;

     sigctl(SCTL_REG, SIGUSR1, 0);       /* notify process on SIGUSR1 but */
                                         /* don't execute any handler    */

     if ((fd = open("datafile", O_RDONLY)) == -1) {
          perror("open (datafile) failed");
          exit(1);
     }

     /* Set up the I/O request */
     request.li_opcode = LO_READ;        /* request read */
     request.li_fildes = fd;             /* file descriptor */
     request.li_buf = (char *) buf;      /* store input data here */
     request.li_nbyte = BLK_SIZ;         /* each stride = 1 block */
     request.li_status = &reqstat;       /* status for this request */
     request.li_signo = SIGUSR1;         /* send signal upon completion */
     request.li_nstride = 10;            /* read 10 strides */
     request.li_filstride = 10 * BLK_SIZ;/* file stride = 10 blocks */

     sigoff();                           /* defer signal reception so SIGUSR1 */
                                         /* not received before pause() below */
     if (listio(LC_START, &request, 1) != 1) {
          perror("listio failed");
```

```
            exit(1);
        }

        /* other work can be performed here while listio request completes */

        pause();                            /* automatically calls sigon() */

        printf("Number of bytes read = %d\n\n", reqstat.sw_count);
    }
```

SEE ALSO

lseek(2), pause(2), reada(2), recall(2), recalla(2), sigctl(2), signal(2), ulimit(2), write(2)

NAME

lseek – Moves read/write file pointer

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek (int fildes, off_t offset, int whence);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The lseek system call sets the read/write file pointer. It accepts the following arguments:

fildes Specifies a file descriptor. It is returned from a creat(2), dup(2), fcntl(2), or open(2) system call.

offset Specifies the number of bytes associated with the pointer's new location.

whence Specifies a value to indicate the pointer's location. The following are valid *whence* values.

0 or SEEK_SET	Pointer is set to *offset* bytes.
1 or SEEK_CUR	Pointer is set to its current location plus *offset*.
2 or SEEK_END	Pointer is set to the file size plus *offset*.

Some devices such as terminals are incapable of seeking (for example, a user cannot reposition the current offset in a file to an arbitrary position). The value of the file pointer associated with such a device is undefined.

RETURN VALUES

If lseek completes successfully, it returns a nonnegative integer indicating the file pointer value as measured in bytes from the beginning of the file; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The lseek system call fails and the file pointer remains unchanged if one of the following error conditions occurs:

Error Code	Description
EBADF	The *fildes* argument is not an open file descriptor.

EINVAL	The resulting file pointer would be negative.
EINVAL and SIGSYS signal	The *whence* argument is not 0, 1, or 2.
ESPIPE	The *fildes* argument is associated with a FIFO special file (named pipe).

FORTRAN EXTENSIONS

The lseek system call can be called from Fortran as a function:

```
INTEGER fildes, offset, whence, LSEEK, I
I = LSEEK (fildes, offset, whence)
```

Alternatively, lseek can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
INTEGER fildes, offset, whence
CALL LSEEK (fildes, offset, whence)
```

The Fortran program must not specify both the subroutine call and the function reference to lseek from the same procedure.

EXAMPLES

The following examples show different applications of the lseek system call. For each lseek example, file datafile is opened using the following:

```
int fd;
fd = open("datafile", O_RDWR);
```

Example 1: The following lseek request positions the current file pointer for datafile to the fourth data block (4096 bytes each) of the file:

```
lseek(fd, (long) 3 * 4096, 0);
```

Example 2: This lseek request returns the process's current offset into datafile:

```
int ret;
ret = lseek(fd, 0L, 1);
```

Example 3: This lseek request positions the current offset at the end of datafile. Therefore, the next write operation to the file will append to the file:

```
lseek(fd, 0L, 2);
```

Example 4: The following `lseek` request updates a record on `datafile`:

```
struct record recd;

read(fd, &recd, sizeof(struct record));       /* read record */
/* update record in user memory */
lseek(fd, (long) -sizeof(struct record), 1);  /* backup file offset */
write(fd, &recd, sizeof(struct record));      /* write updated record */
```

Example 5: This `lseek` request returns the current size of `datafile`. However, the current offset into the file is now positioned at the end-of-file (EOF):

```
int ret;
ret = lseek(fd, 0L, 2);
```

Example 6: This `lseek` request positions the current offset into `datafile` 100 bytes beyond the end of the file. Therefore, the next write operation to `datafile` will cause a 100-byte, 0-filled gap to be created in the file with the output data written at the current offset position.

If a `read`(2) request is issued to `datafile` while the current offset points beyond the end of the file, the request just returns an EOF condition.

```
lseek(fd, 100L, 2);
```

Example 7: This `lseek` request always fails. A file's current file pointer cannot be positioned before the beginning of a file:

```
lseek(fd, -100L, 0);
```

FILES

`/usr/include/sys/types.h`	Contains types required by ANSI X3J11
`/usr/include/unistd.h`	Contains C prototype for the `lseek` system call

SEE ALSO

`creat`(2), `dup`(2), `fcntl`(2), `open`(2), `read`(2)

NAME

lsetattr – Sets metadata for a file

SYNOPSIS

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

int lsetattr (char *fname, struct vattr *vap, int asize);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The lsetattr system call makes the generality of the VOP_SETATTR() macro defined in the sys/vnode.h file available to user space. A single kernel call can set all the user-accessible metadata associated with a file. This information includes the time stamp, account ID, owner ID, and permission bits.

The lsetattr system call accepts the following arguments:

fname Points to the file name.

vap Points to an attribute structure containing the desired metadata changes.

asize Contains the size of the structure pointed to by the *vap* argument. *asize* provides robustness across minor changes in the attribute structure definition; it will not cause an error return.

The lsetattr call does not follow symbolic links.

NOTES

Since some attribute change requests are validated above the vnode switch and since lsetattr goes directly to the vnode switch, the initial implementation is restricted. A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_ADMIN	The process is allowed to set metadata for a file.

If the PRIV_SU configuration option is enabled, the super user is allowed to set metadata for a file.

RETURN VALUES

If lsetattr completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error. For an explanation of the error, enter an explain(1) command or see the error code list on the intro(2) man page.

These calls are referenced on this man page instead of listing the errors in an ERRORS section because `lsetattr` can behave like one or more system calls depending upon the arguments specified and can cause several different errors.

EXAMPLES

The following examples illustrate different uses of the `lsetattr` system call.

Example 1: The following code fragment shows how a single `lsetattr` system call can perform the function of combined `chown`(2) and `chmod`(2) system calls.

```
vap->va_uid = 1005;
vap->va_mode = 0644;
vap->va_mask = AT_UID | AT_MODE;
ex = lsetattr( "file_name", vap, sizeof(*vap));
if (ex < 0) fprintf(stderr, "example failed\n");
```

Example 2: This example shows how to set the sitebits on a symbolic link.

```
vap->va_sitebits = SITE_BITS_VALUE;
vap->va_mask = AT_SITEBITS;
ex = lsetattr( "link_name", vap, sizeof(*vap));
if (ex < 0) fprintf(stderr, "second example also failed\n");
```

Example 3: A final example illustrates how to set the account ID.

```
vap->va_acid = 42;
vap->va_mask = AT_ACID;
ex = lsetattr( "nuther_file", vap, sizeof(*vap));
if (ex < 0) fprintf(stderr, "third example busted, too\n");
```

FILES

`/usr/include/types.h`	Contains types required by ANSI X3J11
`/usr/include/vnode.h`	Contains vnode, attribute structure, and bit definitions for `va_mask`.

SEE ALSO

`chmod`(2), `chown`(2), `intro`(2)

`explain`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`mkdir` – Makes a directory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir (const char *path, mode_t mode);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `mkdir` system call creates a new directory. It accepts the following arguments:

path Names the new directory.

mode Provides the mode of the new directory. The protection part of the *mode* argument is modified by the process' mode mask (see `umask`(2)).

The owner ID of the new directory is set to the process' real user ID. The group ID of the new directory is set to the group ID of the parent directory. The newly created directory is empty, with the exception of entries for "." and "..".

NOTES

The new directory is assigned the active security label of the process.

The active security label of the process must fall within the security label range of the file system in which the directory is being created.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

The process must be granted write permission to the parent directory via the security label.

To be granted write permission to the parent directory, the active security label of the process must equal the security label of the parent directory.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.

PRIV_DAC_OVERRIDE	The process is granted write permission to the parent directory via the permission bits and access control list.
PRIV_MAC_READ	The calling process is granted search permission to every component of the path prefix via the security label.
PRIV_MAC_WRITE	The process is granted write permission to the parent directory via the security label.

If the PRIV_SU configuration option is enabled, to every component of the path prefix. If the PRIV_SU configuration option is enabled, the super user is granted write permission to the parent directory.

RETURN VALUES

If mkdir completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The mkdir system call fails and no directory is created if one of the following error conditions occurs:

Error Code	Description
EACCES	Either a component of the path prefix denies search permission or write permission is denied on the parent directory of the directory to be created.
EACCES	Parent directory label is not equal to the active security label of the process.
EEXIST	The specified file already exists.
EFAULT	The *path* argument points outside the allocated address space of the process.
EFLNEQ	Attempt was made to create a directory outside the bounds of the file system.
EFLNEQ	The active security label of the process falls outside the range of the file system.
EIO	An I/O error has occurred during access of the file system.
EMLINK	The maximum number of links to the parent directory would be exceeded.
ENOENT	A component of the path prefix does not exist.
ENOENT	The path is longer than the maximum allowed.
ENOTDIR	A component of the path prefix is not a directory.
EQACT	A file or inode quota limit was reached for the current account ID.
EQGRP	A file or inode quota limit was reached for the current group ID.
EQUSR	A file or inode quota limit was reached for the current user ID.
EROFS	The path prefix resides on a read-only file system.

SEE ALSO

umask(2)

NAME

mknod, mkfifo – Makes a directory or a special or regular file

SYNOPSIS

```
#include <unistd.h>
int mknod (char *path, int mode, int dev, long p0, long p1,...,p7);
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *path, mode_t mode);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to mkfifo)

DESCRIPTION

The mknod system call creates a file or directory. It accepts the following arguments:

path Names the new file or directory.

mode Specifies the mode of the new file. (Symbolic names for these constants exist in <sys/stat.h>). The following are valid values for *mode*.

0170000 File type, as follows:

- 0010000 FIFO special file (named pipe)
- 0020000 Character special file
- 0040000 Directory
- 0060000 Block special file
- 0100000 Regular file
- 0120000 Offline file without data
- 0110000 Offline file with data

0004000 Set user ID on execution

0002000 Set group ID on execution

0000777 Access permissions, as follows:

- 0000400 Read by owner
- 0000200 Write by owner

0000100	Execute (search on directory) by owner
0000070	Read, write, or execute (search) by group
0000007	Read, write, or execute (search) by others

dev Specifies a device.

If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device.

If *mode* does not indicate a block special or character special device, *dev* is ignored.

p0,p1,. . .,p7 Specifies device-specific parameter words. These words give the operating system more information about the device's configuration.

The file's owner ID is set to the effective user ID of the process. The file's group ID is set to the group ID of the parent directory.

Values of *mode* other than the preceding are undefined and should not be used. The low-order 9 bits of *mode* are modified by the file mode creation mask of the process; all bits set in the mask are cleared. See `umask`(2).

Only a process with appropriate privilege can use this system call.

The file handle for an offline file created by the `mknod` system call has zero elements.

The `mkfifo` routine creates a new FIFO special file named by the path name to which *path* points. The file permission bits of the new FIFO are initialized from *mode*. The low-order 9 bits of *mode* are modified by the file mode creation mask of the process; all bits set in the mask are cleared. See `umask`(2). `mkfifo` sets the file's owner ID and group ID, following the same rules that `mknod` uses.

NOTES

The active security label of the calling process must fall within the security label range of the file system on which the new node will reside.

If the `FSETID_RESTRICT` option is enabled, only a process with appropriate privilege can create set-user-ID or set-group-ID files.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

To be granted write permission to the parent directory, the active security label of the process must equal the security label of the directory.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_ADMIN`	The process is allowed to use this system call.
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.

PRIV_DAC_OVERRIDE	The process is granted write permission to the parent directory via the permission bits and access control list.
PRIV_FSETID	When the FSETID_RESTRICT option is enabled, the process is allowed to create set-user-ID or set-group-ID files.
PRIV_MAC_READ	The process is granted search permission to every component of the path prefix via the security label.
PRIV_MAC_WRITE	The process is granted write permission to the parent directory via the security label.
PRIV_RESTART	The process is allowed to create a restart file.

If the PRIV_SU configuration option is enabled, the super user or a process with the PERMBITS_MKNOD permbit is allowed to use this system call. The super user is granted search permission to every component of the path prefix and is granted write permission to the parent directory. The super user is allowed to create restart files. If the PRIV_SU and FSETID_RESTRICT configuration options are enabled, the super user is allowed to create set-user-ID and set-group-ID files.

RETURN VALUES

If mknod or mkfifo completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The mknod or mkfifo system call fails and the new file is not created if one of the following error conditions occurs:

Error Code	Description
EACCES	Search permission is denied for a component of the path prefix.
EACCES	Write permission is denied to the parent directory.
EEXIST	The specified file exists.
EFAULT	The *path* argument points outside the allocated process address space.
EFLNEQ	The active security label of the calling process does not fall within the range of the file system on which the new file or directory will reside.
EINVAL	The call contains an argument that is not valid. For example, an attempt was made to create a file or directory on a symbolic link.
ENOENT	A component of the path prefix does not exist.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The process does not have appropriate privilege to use this system call.
EROFS	The directory in which the file is to be created is located on a read-only file system.

FORTRAN EXTENSIONS

The mknod system call can be called from Fortran as a function:

```
CHARACTER*n path
INTEGER  mode, dev, MKNOD, I
I = MKNOD (path, mode, dev)
```

Alternatively, mknod can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
CHARACTER*n path
INTEGER mode, dev
CALL MKNOD (path, mode, dev)
```

The Fortran program must not specify both the subroutine call and the function reference to mknod from the same procedure. On all Cray PVP systems except CRAY T90 systems, *path* may also be an integer variable. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte.

EXAMPLES

The following examples illustrate use of the mkfifo and mknod system calls.

Example 1: This mkfifo request creates a named pipe (that is, FIFO special file), called name_pipe with the specified permissions:

```
if (mkfifo("name_pipe", 0640) == -1) {
     perror("mkfifo failed");
     exit(1);
}
```

Example 2: This example shows one of the applications for the mknod system call, the creation of a named pipe (that is, FIFO special file). This mknod request creates a named pipe called name_pipe with the specified permissions.

```
if (mknod("name_pipe", 010640) == -1) {
     perror("mknod failed");
     exit(1);
}
```

FILES

/usr/include/sys/stat.h	Contains ANSI C prototype for the mkfifo system call
/usr/include/sys/types.h	Contains data type definitions and definition for mode_t
/usr/include/unistd.h	Contains C prototype for the mknod system call

SEE ALSO

`chmod`(2), `creat`(2), `exec`(2), `umask`(2)

`mkdir`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`fs`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

mount – Mounts a file system

SYNOPSIS

```
#include <sys/mount.h>
int mount (char *spec, char *fsname, char *dir, char *options, int flags,
int fstyp, char *data, int datalen);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mount system call requests that a removable file system contained on a block or character special file be mounted on a directory as specified by the following arguments:

spec Points to the block or character special file's path name. The *spec* argument may be a block special file.

fsname Points to the file system full path name.

dir Points to the path name of the directory on which *spec* is to be mounted. On successful completion, references to the *dir* file refer to the root directory on the mounted file system. The full path must be specified.

options Points to the file system mount options. This is a comma-separated list of words from the -o option on the mount(8) command.

flags Identifies the flag whose low-order bit controls write permission on the mounted file system. If the bit is 1, writing is forbidden; otherwise, writing is permitted according to individual file accessibility. The *flags* argument can contain three values, as defined in /usr/include/sys/mount.h:

- MS_RDONLY Read only.
- MS_FSS The *fstyp* argument has meaning.
- MS_DATA The *data* and *datalen* arguments have meaning. This flag is used by the NFS file system type.

fstyp Specifies the file system type being mounted (see sysfs(2)). This field is interpreted only if the MS_FSS flag is set.

data Points to file-system-specific mount information of size *datalen*. The *data* and *datalen* arguments are interpreted only if MS_DATA is set in *flags*.

datalen Specifies size of mount information, as described in explanation of *data* pointer.

Only an appropriately privileged process can use this system call.

NOTES

If the MLS_OBJ_RANGES configuration is enabled, the minimum and maximum security levels (with the exception of the syslow and/or syshigh security levels) and the authorized compartments of a file system must fall within the authorized ranges of the UNICOS system, otherwise the mount request fails.

File systems that have not been explicitly assigned a security label range (by using the labelit(8) or mkfs(8) commands) are considered to have the security label range [level 0: none, level 0: none].

The mount point label of the file system must be less than or equal to the lowest label assigned to the file system.

When the MLS_OBJ_RANGES option is set to SECURE, the security label range of the file system must fall within the security label range of the UNICOS system.

To mount a file system with DEV_ENFORCE_ON set to ON, the device must be off or the minimum and maximum security level of the file system must be within the minimum and maximum security levels authorized for the device. Also, the authorized security compartments for the file system must be equal to or a subset of the authorized compartments for the device, and the device must be labeled as multilevel.

A process is granted search permission to a component of the path prefix only if the active security label of the process is greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_ADMIN	The process is allowed to use this system call.
PRIV_DAC_OVERRIDE	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
PRIV_MAC_READ	The process is granted search permission to every component of the path prefix via the security label.

If the PRIV_SU configuration option is enabled, a super user is allowed to use this system call and is granted search permission to every component of the path prefix.

RETURN VALUES

If mount completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The mount system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	Search permission is denied for a component of the path prefix.

EBUSY	The device associated with *spec* is currently mounted.
EBUSY	The *dir* argument is currently mounted, is someone's current working directory, or is otherwise busy.
EFAULT	The *spec* or *dir* argument points outside the allocated process address space.
EINVAL	The *fstyp* argument is not valid.
ENODEV	The file system super-block device number does not match the block special device.
ENOENT	Any of the specified files does not exist.
ENOEXEC	The super block or dynamic block of the file system is corrupt.
ENOTDIR	A component of a path prefix is not a directory.
ENOTDIR	The *dir* argument is not a directory.
ENXIO	The device associated with *spec* does not exist.
EPERM	The process does not have appropriate privilege to use this system call.
ESYSLV	The upper security level of the file system is greater than the upper security level of the UNICOS system and is not the `syshigh` security label.
ESYSLV	The lower security level of the file system is less than the lower security level of the UNICOS system and is not the `syslow` security label.
ESYSLV	A user with an active security level (nonzero) tried to mount a non-UNICOS file system. This file system is treated as unclassified with lower and upper security levels equal to 0.
ESYSLV	The `MLS_OBJ_RANGES` option is enabled, and the authorized compartments of the file system are not a subset of the authorized compartments for the UNICOS system.
ESYSLV	The `DEV_ENFORCE_ON` option is enabled and the file system label is not within the device label range.
ESYSLV	The `DEV_ENFORCE_ON` option is enabled and the device is on, but not labeled as multilevel.

FILES

`/usr/include/sys/mount.h`	Mount structure

SEE ALSO

`sysfs`(2), `umount`(2)

`libudb`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`labelit`(8), `mkfs`(8), `mount`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`msgctl` – Provides message control operations

SYNOPSIS

```
#include <sys/msg.h>
int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

XPG4

DESCRIPTION

The `msgctl` system call provides a variety of message control operations. It accepts the following arguments:

msqid Specifies a message queue identifier.

cmd Specifies a message control operation. The following are valid *cmd* values.

`IPC_STAT` Places the current value of each member of the `msqid_ds` data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in the `sys/msg.h` include file (see `msg`(5)). The calling process must have read permission to the message queue associated with *msqid*.

`IPC_SET` Sets the value of the following members of the `msqid_ds` data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*.

```
msg_perm.uid
msg_perm.gid
msg_perm.mode     /* only access permission bits */
msg_qbytes
```

`IPC_SET` can be executed only by a process that has an effective user ID equal to the value of `msg_perm.cuid` or `msg_perm.uid` in the `msqid_ds` data structure associated with *msqid*. Only a process with the appropriate privilege can raise the value of `msg_qbytes`.

`IPC_RMID` Removes the message queue identifier specified by *msqid* from the system and destroys the message queue and `msqid_ds` data structure associated with it. `IPC_RMID` can be executed only by a process that has an effective user ID equal to the value of `msg_perm.cuid` or `msg_perm.uid` in the `msqid_ds` data structure associated with *msqid*.

IPC_SETACL (Secure systems only) Sets the access control list (ACL) on the message queue specified by *msqid*. The ipc_perm structure within the msqid_ds structure pointed to by *buf* contains a pointer, ipc_acl, to an acl_rec structure with the required ACL entries, and a count of those entries, ipc_aclcount. If an ACL exists for the message queue, it is replaced by the one provided with this call. If ipc_aclcount is 0, any existing ACL is removed. The calling process must be the owner of the message queue specified by *msqid*.

IPC_GETACL (Secure systems only) Retrieves the access control list (ACL) for the message queue specified by *msqid*. The ipc_perm structure within the msqid_ds structure pointed to by *buf* contains a pointer, ipc_acl, to an acl_rec structure where the ACL entries are to be returned. The count of entries to be returned is specified in the ipc_aclcount field. If there are more than ipc_aclcount entries, only the first ipc_aclcount entry is returned. If there are fewer than ipc_aclcount entries, all entries are returned. The return value indicates the number of entries returned. If there is no ACL, the return value is 0. The calling process must have read permission to the message queue specified by *msqid*.

IPC_SETLABEL (Secure systems only) Sets the security label on the message queue specified by *msqid*. The ipc_perm structure within the msqid_ds structure pointed to by *buf* contains a security level, ipc_slevel, and a compartment set, ipc_scomps, to be set in the security label on the message queue. Only a process with the appropriate privilege can set the security label of a message queue.

buf Points to a structure.

NOTES

A process is granted read permission to a message queue only if the active security label of the process is greater than or equal to the security label of the message queue, and the process is granted read access by the message queue access control list (ACL) (if one is assigned). This applies to the IPC_STAT and IPC_GETACL operations.

The IPC_SET, IPC_RMID, and IPC_SETACL operations require that the active security label of the process is equal to the security label of the message queue.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_MAC_READ	The process is considered to meet the security label requirements for being granted read permission to a message queue.
PRIV_MAC_WRITE	The process is considered to meet the security label requirements for performing an IPC_SET, IPC_RMID, or IPC_SETACL operation.

PRIV_DAC_OVERRIDE	The process is considered to meet the permission mode and ACL requirements for being granted read permission to a message queue.
PRIV_FOWNER	The process is considered to meet the message queue ownership requirements for the IPC_SET, IPC_RMID, and IPC_SETACL operations. For the IPC_SET operation, the process is also permitted to raise the value of msg_qbytes.
PRIV_MAC_UPGRADE	The process is allowed to raise the security label of a message queue.
PRIV_MAC_DOWNGRADE	The process is allowed to lower the security label of a message queue.

If the PRIV_SU configuration option is enabled, the super user is granted the same abilities as all effective privileges shown previously. The super user is considered the owner of a message queue, and is granted read permission to that message queue.

RETURN VALUES

If msgctl completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The msgctl system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	The *cmd* argument is IPC_STAT, and the calling process does not have read permission (see msg(5)).
EACCES	The *cmd* argument is IPC_GETACL, and the calling process does not have read permission.
EFAULT	The *buf* argument points to an illegal address.
EFAULT	The *cmd* argument is IPC_SETACL or IPC_GETACL and the ipc_acl field in *buf* points to an illegal address.
EINVAL	The *msqid* argument is not a valid message queue identifier.
EINVAL	The *cmd* argument is not a valid command.
EINVAL	The *cmd* argument is IPC_SET, and msg_perm.uid or msg_perm.gid is not valid.
EINVAL	The *cmd* argument is IPC_SETACL, and one of the following is true: • The ipc_aclcount field in *buf* is 0, but there is no ACL associated with *msqid*. • The ipc_aclcount field in *buf* is less than 0 or greater than 256. • The ACL supplied failed validation.

ENOMEM	The *cmd* argument is IPC_SETACL, and no memory is available to store the ACL. The command should be retried at a later time.
EPERM	The *cmd* argument is IPC_RMID or IPC_SET, and the effective user ID of the calling process is not equal to the value of msg_perm.cuid or msg_perm.uid in the msgid_ds data structure associated with *msqid*; the calling process does not have the appropriate privilege.
EPERM	The *cmd* argument is IPC_SET, an attempt is being made to increase to the value of msg_qbytes, and the calling process does not have the appropriate privilege.
EPERM	The *cmd* argument is IPC_SETLABEL, and the calling process does not have the appropriate privilege.
EPERM	The *cmd* argument is IPC_SETACL, and the calling process does not meet ownership requirements and does not have the appropriate privilege.

FILES

/usr/include/sys/msg.h	Contains message-related data structures and macros

SEE ALSO

msgget(2), msgrcv(2), msgsnd(2)

ipc(5), msg(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

ipc(7) Online only

NAME

msgget – Accesses the message queue

SYNOPSIS

```
#include <sys/msg.h>
int msgget (key_t key, int msgflg);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

XPG4

DESCRIPTION

The msgget system call returns the message queue identifier. It accepts the following arguments:

key Specifies the message queue.

msgflg Specifies a flag value.

A message queue identifier and associated message queue and data structure (see msg(5)) are created for *key* if one of the following is true:

- *key* is IPC_PRIVATE.
- *key* does not already have a message queue identifier associated with it, and the value of *msgflg*&IPC_CREAT is not 0.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

- msg_perm.cuid, msg_perm.uid, msg_perm.cgid, and msg_perm.gid are set to the effective user ID and effective group ID, respectively, of the calling process.
- The access permission bits of msg_perm.mode are set to the access permission bits of *msgflg*.
- msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime are set to 0.
- msg_ctime is set to the current time.
- msg_qbytes is set to the system limit.

NOTES

If the calling process has the ipc_persist permission bit, the message queue is created as a persistent queue. Persistent message queues will not be removed from the system unless a msgctl(2) system call with the command IPC_RMID, or an ipcrm(1) command, is performed on the queue.

If the calling process does not have this permission bit, the message queue is linked into a list of nonpersistent queues belonging to the session of which the process is a member. When the last process of the session terminates, all the message queues linked to the session are removed from the system.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_RESOURCE	The process is considered to have the ipc_persist permission bit.

If the PRIV_SU configuration option is enabled, the super user is granted the same abilities as all effective privileges shown above. The super user is considered to have the ipc_persist permission bit.

RETURN VALUES

If msgget completes successfully, a nonnegative integer, namely a message queue identifier, is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The msgget system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A message queue identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *msgflg* would not be granted (see ipc(7)).
EEXIST	A message queue identifier exists for *key* but the values of *msgflg*&IPC_CREAT and *msgflg*&IPC_EXCL are both nonzero.
ENOENT	A message queue identifier does not exist for *key*, and the value of *msgflg*&IPC_CREAT is 0.
ENOSPC	A message queue identifier is to be created, but the system-imposed limit on the maximum number of allowed message queue identifiers system-wide would be exceeded.

FILES

/usr/include/sys/msg.h	Contains message-related data structures and macros

SEE ALSO

msgctl(2), msgrcv(2), msgsnd(2)

ipcrm(1), ipcs(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

stdipc(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

ipc(5), msg(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

ipc(7) Online only

NAME

msgrcv – Reads a message from a message queue

SYNOPSIS

```
#include <sys/msg.h>
int msgrcv (int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

XPG4

DESCRIPTION

The msgrcv system call reads a message from the queue associated with the message queue identifier and places it in the user-defined buffer. It accepts the following arguments:

msqid Specifies a message queue identifier.

msgp Points to a user-defined buffer.

This user-defined buffer must contain a message type field (of type long int) followed by the data portion for the message text. The following structure is defined in the include file sys/msg.h (see msg(5)):

```
struct msgbuf {
      long int    msgtype;    /* message type */
      char        msgtext[1]; /* message text */
}
```

The structure member msgtype is the received message's type, as specified by the sending process. The structure member msgtext is the text of the message.

msgsz Specifies the size, in bytes, of msgtext. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and the value of *msgflg*&MSG_NOERROR is not 0. The truncated part of the message is lost; no indication of the truncation is given to the calling process.

msgtyp Specifies the type of message requested. The following are valid types.

- If *msgtyp* is 0, the first message on the queue is received.
- If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.
- If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

msgflg Specifies a flag value.
If a message of the desired type is not on the queue, *msgflg* identifies one of the following actions:

- If the value of *msgflg*&IPC_NOWAIT is not 0, the calling process returns immediately with a return value of −1 and sets errno to ENOMSG.
- If the value of *msgflg*&IPC_NOWAIT is 0, the calling process suspends execution until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue identifier *msqid* is removed from the system. When this occurs, errno is set to EIDRM, and a value of −1 is returned.
  - The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes execution in the manner prescribed in sigaction(2).

Upon successful completion, the data structure associated with *msqid* (see msg(5)) is changed as follows:

- msg_qnum is decremented by 1.
- msg_lrpid is set to the process ID of the calling process.
- msg_rtime is set to the current time.

NOTES

A process is granted read permission to a message queue only if the active security label of the process is greater than or equal to the security label of the message queue, and the process is granted read access by the message queue access control list (ACL) (if one is assigned).

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_MAC_READ	The process is considered to meet the security label requirements for being granted read permission to a message queue.
PRIV_DAC_OVERRIDE	The process is considered to meet the permission mode and ACL requirements for being granted read permission to a message queue.

If the PRIV_SU configuration option is enabled, the super user is granted the same abilities as all effective privileges shown above. The super user is granted read permission to a message queue.

RETURN VALUES

If msgrcv completes successfully, a value that indicates the number of bytes actually placed in the *msgtext* buffer is returned. Otherwise, a value of −1 is returned, and errno is set to indicate the error; no message is received.

ERRORS

The msgrcv system call fails and receives no message if one of the following error conditions occurs:

Error Code	Description
E2BIG	The value of *msgtext* is greater than *msgsz*, and *msgflg*&MSG_NOERROR is equal to 0.
EACCES	Operation permission is denied to the calling process (see msg(5)).
EFAULT	*msgp* points to an illegal address.
EIDRM	The message queue identifier *msqid* is removed from the system.
EINTR	The msgrcv system call was interrupted by a signal.
EINVAL	*msqid* is not a valid message queue identifier.
EINVAL	*msgsz* is less than 0.
ENOMSG	The queue does not contain a message of the desired type, and *msgtyp*&IPC_NOWAIT is not 0.

FILES

/usr/include/sys/msg.h	Contains message-related data structures and macros

SEE ALSO

msgctl(2), msgget(2), msgsnd(2), sigaction(2)

ipc(5), msg(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

ipc(7) Online only

NAME

msgsnd – Sends a message to a message queue

SYNOPSIS

```
#include <sys/msg.h>
int msgsnd (int msqid, const void *msgp, size_t msgsz, int msgflg);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

XPG4

DESCRIPTION

The msgsnd system call sends a message to the queue associated with the message queue identifier. It accepts the following arguments:

msqid Specifies a message queue identifier.

msgp Points to a user-defined buffer. It must contain a message type field (of type long int) followed by a data portion for the message text. The following structure is defined in the include file sys/msg.h (see msg(5)):

```
struct msgbuf {
      long msgtype;    /* message type */
      char msgtext[];  /* message text */
}
```

The structure member msgtype is a positive integer that can be used by the receiving process for message selection.

msgsz Specifies the length of msgtext in the number bytes. *msgsz* can range from 0 to a system-imposed maximum.

msgflg Specifies the action to be taken if the message cannot be immediately processed. *msgflg* specifies the action to be taken if one or more of the following are true and thus prevents the message from being immediately processed:

- The number of bytes already on the queue is equal to msg_qbytes (see msg(5)).
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

The following actions are available:

- If *msgflg*&IPC_NOWAIT is not 0, the message is not sent and the calling process returns immediately.

- If *msgflg*&IPC_NOWAIT is 0, the calling process suspends execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists. In this case, the message is sent.
  - The message queue identifier *msqid* is removed from the system (see msgctl(2)). When this occurs, errno is set to EIDRM, and a value of −1 is returned.
  - The calling process receives a signal that is to be caught. In this case, the message is not sent and the calling process resumes execution in the manner prescribed in the sigaction(2) system call.

Upon successful completion, the data structure associated with *msqid* (see msg(5)) is changed as follows:

- msg_qnum is incremented by 1.
- msg_lspid is set to the process ID of the calling process.
- msg_stime is set to the current time.

NOTES

A process is granted write permission to a message queue only if the active security label of the process is equal to the security label of the message queue, and the process is granted write access by the message queue access control list (ACL) (if one is assigned).

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_MAC_WRITE	The process is considered to meet the security label requirements for being granted write permission to a message queue.
PRIV_DAC_OVERRIDE	The process is considered to meet the permission mode and ACL requirements for being granted write permission to a message queue.

If the PRIV_SU configuration option is enabled, the super user is granted the same abilities as all effective privileges shown above. The super user is granted write permission to a message queue.

RETURN VALUES

If msgsnd completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error, and no message is sent.

ERRORS

The msgsnd system call fails and sends no message if one of the following error conditions occurs:

Error Code	Description
EACCES	Operation permission is denied to the calling process (see ipc(7)).

EAGAIN	The message cannot be sent for one of the reasons cited in the DESCRIPTION section, and *msgflg*&IPC_NOWAIT is not 0.
EFAULT	*msgp* points to an illegal address.
EIDRM	The message queue identifier *msqid* is removed from the system.
EINTR	The msgsnd system call was interrupted by a signal.
EINVAL	The value of *msqid* is not a valid message queue identifier.
EINVAL	The value of *msgtype* is less than 1.
EINVAL	The value of *msgsz* is less than 0 or greater than the system-imposed limit.

FILES

/usr/include/sys/msg.h	Contains message-related data structures and macros

SEE ALSO

msgctl(2), msgget(2), msgrcv(2), sigaction(2)

ipc(5), msg(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

ipc(7) Online only

NAME

mtimes – Provides multitasking execution overlap profile

SYNOPSIS

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

struct mtms *mtimes (struct mtms *buf);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The mtimes system call lets users have a structure in user memory continually updated with multitasking execution overlap information. (This is the same information that appears in accounting records for multitasking programs.) From this mtms structure, users can determine how much execution time the multitasking program has accumulated in an interval.

The mtimes system call accepts the following argument:

buf Specifies the address of the structure to receive the data. If the address is 0, the structure will no longer be updated.

The mtms structure contains the following members:

```
time_t   mtms_update;                /* Time of last update */
short    mtms_conn;                  /* # of cpus presently connected */
time_t   mtms_mutime[NCPU];          /* Multitask cpu utilization */
```

Update of the structure stops if one of the following occurs:

- The process shrinks, placing the structure outside the program's address space.
- An exec(2) system call is executed.

NOTES

The times in the mtms structure refer to the period since the multitasking group began execution, not to the period since the invocation of the mtimes call.

Monitoring the mtms structure at the user level is somewhat tricky because the operating system running in another CPU may be updating it at the same time.

RETURN VALUES

The `mtimes` system call returns the address of the `mtms` structure. If the value returned is 0, the feature is disabled; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `mtimes` system call fails if the following error condition occurs:

Error Code	Description
`EFAULT`	The *buf* argument points outside the program's address space.

SEE ALSO

`exec`(2)

`MTIMESX`(3F), `MTTIMES`(3F) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`SECOND`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

`newarraysess` – Starts a new array session

SYNOPSIS

```
#include <unistd.h>
int newarraysess (void);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The `newarraysess` system call creates a new array session and moves the current process from its original array session to the new one. The parents, children, and siblings of the current process are not affected by this move and remain in their original array sessions.

The system generates a handle for the new array session. Normally, the new handle is guaranteed to be unique on the current system only, although some systems may be able to assign global array session handles that are unique across an entire array of systems by setting the `asmachid` system variable. Otherwise, the range of values that the system may assign for array session handles is defined by the system variables `minash` and `maxash`. If necessary, the `setash`(2) system call can be used to override the default handle after the array session has been created.

Ordinarily, a new array session should be started whenever the conceptual equivalent of a login is performed. This includes programs that do conventional logins (for example, `login`(1) or `telnet`(1B)), as well as programs that are essentially logging in to do work on behalf of another user, such as `cron`(8) or batch queueing systems.

RETURN VALUES

If `newarraysess` completes successfully, a value of 0 is returned; otherwise, a value of `-1` is returned, and `errno` is set to indicate the error.

ERRORS

The `newarraysess` system call fails if the following condition occurs:

Error Code	Description
`ENOMEM`	The system is unable to allocate memory or other resources for the new array session.

SEE ALSO

`setash`(2)

`login`(1), `telnet`(1B) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`array_services`(7), `array_sessions`(7),

`cron`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

nice, nicem – Changes priority of processes

SYNOPSIS

```
#include <unistd.h>
int nice (int incr);
#include <sys/category.h>
#include <unistd.h>
int nicem (int category, int id, int incr);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to nice)

DESCRIPTION

The nice system call adds a specified value to the nice value of the calling process. A process' nice value is a positive number for which a greater value results in lower CPU priority. Only an appropriately privileged process can specify a negative *incr*.

Only an appropriately privileged process can set the nice value for a process that it does not own.

The system imposes a maximum nice value of 39 and a minimum nice value of 0. When values above or below these limits are requested, the nice value will be set to the corresponding limit.

The nicem system call changes the nice value of a process or group of processes as specified by the following arguments:

category Specifies a category. The following are valid values for *category*: C_PROC, C_PGRP, C_JOB, or C_UID.

id Specifies the *pid*, *pgrp*, *jid*, or *uid* corresponding to *category*. A *pid* of 0 means the current process, a *pgrp* of 0 means the current process group, a *jid* of 0 means the current job, and a *uid* of 0 means the current user.

incr Specifies the value to be added to the nice value of the calling process.

NOTES

The active security label of the process must be greater than or equal to the security label of every affected process.

To set a nice value, the active security label of the process must equal the security label of every affected process.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_MAC_READ	The active security label of the process is considered to be greater than or equal to the security label of all affected processes.
PRIV_MAC_WRITE	The active security label of the process is considered to equal the security label of all affected processes.
PRIV_POWNER	The process is considered the owner of all affected processes.
PRIV_RESOURCE	The process is allowed to specify a negative value for *incr*.

If the PRIV_SU configuration option is enabled, and is allowed to specify a negative value for *incr*. If the PRIV_SU configuration option is enabled, the super user overrides all security label restrictions.

RETURN VALUES

When nice completes successfully, it returns the new nice value minus 20 (–20 <= return <= 19), and errno is never set. When nicem completes successfully, it returns the new nice value unchanged (0 <= return <= 39). Otherwise a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The nice or nicem system call fails if one of the following error conditions occurs:

Error Code	Description
EINVAL	One of the arguments contains an invalid value.
EPERM	The process specified a negative value for *incr* and does not have appropriate privilege.
EPERM	The process does not own all affected processes and does not have appropriate privilege.
ESRCH	No process can be found to match the *category* and *id* requests.

FORTRAN EXTENSIONS

The nice system call can be called from Fortran as a function:

```
INTEGER incr, NICE, I
I = NICE (incr)
```

The nicem system call can be called from Fortran as a function:

```
INTEGER category, id, incr, NICEM, I
I = NICEM (category, id, incr)
```

EXAMPLES

This example illustrates how the nice and nicem system calls can change the nice value of a process and thereby affect the CPU priority of the process. The following nice and nicem requests return the current nice value of the calling process and increase the nice value to lower the CPU priority of the process. The current nice value returned by nice is offset by a factor of –20.

Only users with super-user status can lower the nice value of a process and thereby raise the CPU priority.

```
#include <sys/category.h>
#include <unistd.h>

main()
{
     int i;

     printf("Current nice value from nice(2) = %d\n", nice(0));
     printf("Current nice value from nicem(2) = %d\n", nicem(C_PROC, 0, 0));

     if ((i = nicem(C_PROC, 0, 5)) == -1)
          perror("nicem (+incr) failed");
     else
          printf("New nice value = %d\n", i);

     if ((i = nicem(C_PROC, 0, -5)) == -1)     /* for non-superusers - */
          perror("nicem (-incr) failed");      /* always fails */
     else
          printf("New nice value = %d\n", i);
}
```

FILES

/usr/include/unistd.h	Contains C prototype for the nice and nicem system calls

SEE ALSO

exec(2)

nice(1), renice(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

passwd(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`nsecctl` – Accesses or manipulates network security information

SYNOPSIS

```
#include <net/nsecctl.h>
#include <net/al.h>
int nsecctl (int op, caddr_t entry, struct al_addr *addr_list, int num);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `nsecctl` system call manipulates and accesses network security information stored in the kernel.

It accepts the following arguments:

op Specifies the operation to be performed. A version number is built into the upper halfword of the operation. The following operations are valid:

`NALM_ADD`	Adds one or more network access list (NAL) entries.
`NALM_DELETE`	Deletes a NAL entry.
`NALM_RESOLVE`	Gets a NAL entry.
`WALM_ADD`	Adds one or more workstation access list (WAL) entries.
`WALM_DELETE`	Deletes a WAL entry.
`WALM_RESOLVE`	Gets a WAL entry.
`WALM_CHECK`	Checks the access for a given WAL entry.
`MAPM_ADD`	Adds a map.
`MAPM_DELETE`	Deletes a map.

entry Specifies the address of a structure, the type of which depends on *op*. For NAL operations, this is a pointer to a `struct nalentry`. For WAL operations, this is a pointer to a `struct walentry`. For map operations, this is a pointer to a `struct ipso_map`.

On calls in which a `walentry` or `nalentry` is expected back, the resultant information is placed at this address.

addr_list Specifies a pointer to an array of `al_addr` structures. These specify either the host or network addresses described by the NAL or WAL entry. This argument is ignored for map operations.

For NAL and WAL operations, this array is copied to the calling process upon completion of the operation. If an error occurs in an add or delete operation for a NAL or a WAL entry, the error number appears in the upper halfword of the `al_flags` field for that element of the `al_addr` list. More than one element of the list can result in an error. Elements of the list for which an error does not appear have been processed (added or deleted) successfully, except in cases where `errno` is set to `EACCES`, `ENOBUFS`, or `EINVAL`.

num Specifies the number of entries in the `al_addr` list. This argument is ignored for map operations.

NOTES

The calling process must have `PRIV_ADMIN` effective privilege.

RETURN VALUES

If `nsecctl` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

If all elements of NAL and WAL multiple address requests are processed successfully, a value of 0 is returned. If an error occurs for any element, a value of -1 is returned, and `errno` is set to indicate the last error encountered.

ERRORS

The `nsecctl` system call fails if one of the following error conditions occurs:

Error code	Description
`EACCES`	The calling process did not have `PRIV_ADMIN` effective privilege. No list elements were processed.
`EEXIST`	An attempt was made to add an entry that already exists.
`EINVAL`	The argument specified was not valid, or the calling process was not built with the correct version of `nsecctl.h`. No list elements were processed.
`ENOBUFS`	No buffer space is available. No list elements were processed.
`ESRCH`	An attempt was made to delete or to resolve an entry that does not exist.

For an `nsecctl` call involving multiple addresses, each element that results in an error is processed if possible, or else has the error number in the upper halfword of the `al_flags` field for that element of the `al_addr` list.

FILES

`usr/include/net/al.h`	Header file for address list `al_addr` structure
`/usr/include/net/map.h`	Header file for `ipso_map` structure
`/usr/include/net/nal.h`	Header file for `nalentry` structure
`/usr/include/net/nsecctl.h`	Header file for `nsecctl` requests
`/usr/include/net/wal.h`	Header file for `walentry` structure

SEE ALSO

`spnet`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

open – Opens a file for reading or writing

SYNOPSIS

```
#include <fcntl.h>
int open (const char *path, int oflag [,mode_t mode] [,long cbits]
[,int cblks]);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The open system call opens a file descriptor and sets the file status flags according to the value of the following arguments:

path Points to a path name of a file.

oflag Specifies status flags. These are constructed by flags from the following list. Only one of the first three flags may be used.

- O_RDONLY — If set, opens file for reading only.
- O_WRONLY — If set, opens file for writing only. To append a file, see the O_APPEND flag.
- O_RDWR — If set, opens file for reading and writing.
- O_RAW — If set, reads or writes whole sectors of data into user space, bypassing system buffers. Usually, the system does automatic read-ahead and write-behind to improve performance. Use of O_RAW does not imply O_LDRAW.
- O_LDRAW — When used in conjunction with O_RAW, I/O bypasses logical device cache as well as system buffer cache. Use of O_LDRAW does not imply O_RAW.
- O_NDELAY — If set, affects subsequent reads and writes (see read(2) and write(2)).

  When opening a FIFO special file (named pipe) with the O_RDONLY or O_WRONLY flag set, the O_NDELAY flag results in the following actions:

  - If O_NDELAY is set, an open for reading-only returns without delay. An open for writing-only returns an error if no process currently has the file open for reading.

- If O_NDELAY is clear, an open for reading-only blocks until a process opens the file for writing. An open for writing-only blocks until a process opens the file for reading.

When opening a file associated with a communication line, the O_RDONLY flag results in the following actions:

- If O_NDELAY is set, the open returns without waiting for a carrier.
- If O_NDELAY is clear, the open blocks until a carrier is present.

O_NONBLOCK With pipes, functions like O_NDELAY, but eliminates the ambiguity of 0 bytes transferred, which means both end-of-file and no data available.

When the O_NDELAY flag is used while opening a pipe and a read from the pipe is performed, if the read returns a 0 value, the user's program cannot determine whether this situation means an end-of-file condition or whether there is no data currently in the pipe to read since a return value of 0 can indicate either condition.

When the O_NONBLOCK flag is used while opening a pipe and a read from the pipe is performed, if an end-of-file condition is encountered, the read returns a value of 0. If there is simply no data in the pipe to read, the read returns a value of −1.

When the O_NONBLOCK flag is used while opening a migrated file (file type IFOFL or IFOFD), the open does not block waiting for the file to be recalled from offline media. The open will return a value of −1 and set errno to EDMNBLK while the recall of the file from the media is in progress. Subsequent open calls with this flag will continue to return with this status until the file is fully recalled, at which point the open proceeds normally.

O_NOCTTY If used while opening a terminal device, open does not cause the terminal device to become the controlling terminal for the process.

O_BIG Allows a user to specify that a file is big when it is created, rather than wait for the file to grow large enough for the system to categorize it as big.

The BIGFILE parameter (defined in the header file sys/param.h) specifies the size in bytes past which a file is considered big.

File space, when allocated, comes from the secondary partition area (see mkfs(8)), if such an area is defined.

O_APPEND If set, the file pointer is set to the end of the file before each write. In addition, the file must be open for writing; see O_RDWR and O_WRONLY.

O_CREAT	This option requires a third argument, *mode*, which is of type mode_t. If the file specified by *path* exists, this flag has no effect, except as noted under O_EXCL in the following. Otherwise, the file is created; the file's user ID is set to the effective user ID of the process, the file's group ID is set to the group ID of the directory in which the file is created, and the low-order 12 bits of the file mode are set to the value of *mode*, modified as follows (see creat(2)): all bits set in the process' file mode creation mask are cleared (see umask(2)).
O_TRUNC	If the file exists, its length is truncated to 0, and the mode and owner are unchanged.
O_EXCL	If O_EXCL and O_CREAT are set, open fails if the file exists.
O_PLACE	If the file specified by *path* exists, this flag has no effect; otherwise, the cbits and cblks parameters, if passed, are used to establish file system partition residency and the number of blocks allocated in each partition. Multiple set bits indicate that the file is to be striped on the specified partitions.
O_RESTART	If O_RESTART and O_CREAT and the process has appropriate privilege, the file is created as a restart file (see restart(2) and chkpnt(2)).
O_SSD	If O_SSD is set, all subsequent I/O is done from the users' secondary data segment (SDS) memory instead of main memory. The I/O is done with the backdoor channel on the IOS if it is configured, or else it uses the sidedoor (which is a backdoor software emulator) (Cray PVP systems except for CRAY EL series, CRAY J90 series, and CRAY T90 series). Addresses passed on the I/O requests are interpreted as word addresses relative to the beginning of the process' SDS field length (see ssbreak(2)). The addresses must be on a 512-word boundary. The count passed is still a byte count, but it must be a multiple of 4096 bytes. This flag is not supported across NFS-mounted file systems. If the O_SSD flag is used to open a file accessed via NFS, the EINVAL error code is returned.
O_SYNC	When a regular file is opened, this flag affects subsequent writes. If set, each write(2) waits for both the file data and file status to be physically updated.

`O_T3D` This flag controls memory usage when the high-speed (HISP) channel connects the I/O subsystem (IOS) and CRAY T3D system. If the `O_T3D` flag is set, all subsequent I/O is performed from the user's CRAY T3D memory instead of secondary data segments (SDS) or the main memory of the Cray host computer system. The I/O is done via backdoor channel on the IOS only. Sidedoor (which is a backdoor software emulator) is not supported.

Addresses passed on the I/O requests are interpreted as CRAY T3D memory addresses.

The count passed is still a byte count but must be a multiple of disk sector size of the referenced device.

`O_WELLFORMED` Used with `O_RAW` to force read and write requests that are not well-formed to fail. If ill-formed I/O requests are not specified with `O_RAW`, they are buffered without any notification to the user.

`O_SFS_DEFER_TM`

This flag is valid only for shared file systems (SFSs). It can reduce file system overhead by declaring to the UNICOS kernel that updates to file inode time stamps may be done less frequently than they otherwise would. If it is not critical to the application that the time stamps returned by the `stat` or `fstat` system calls be highly accurate, then setting this flag on very active files is advisable. The number of inode updates to media is reduced, which implies a reduction in overhead and a corresponding increase in performance.

`O_SFSXOP` This flag grants exclusive open lock on an SFS file system.
On return from the `open` the user process is guaranteed that no other process in the SFS cluster has this file open. While that process owns the open lock, the process may execute an `unlink`(2) system call on the file, thus causing all other pending `open` calls for this file to fail with an `ENOENT` error. If a process tries to open a file without the `O_SFSXOP` flag when the file is already open by another process with an exclusive open lock, the resulting behavior is determined by the presence or absence of either the `O_NDELAY` or the `O_NONBLOCK` flags. If either is set on the attempted open, the open will fail with `EAGAIN`. If neither flag is set, the process will sleep until the file has been closed by all other processes on all machines. If the same process opens a file with the exclusive open flag, and then attempts a subsequent non-exclusive open, the second open attempt will fail with `EDEADLK`.

mode Sets the bit pattern denoting the file's access permission. The following are valid patterns.

04000 Sets user ID on execution.

020#0 Sets group ID on execution if # is 7, 5, 3, or 1. Enables mandatory file or record locking if the file is an ordinary file and # is 6, 4, 2, or 0.

00400 Reads by owner.

00200 Writes by owner.

00100 Executes (or searches if a directory) by owner.

00070 Reads, writes, and executes (searches) by group.

00007 Reads, writes, and executes (searches) by others.

cbits Specifies the bits of the *cbits* argument correspond (starting with 2^0 or the rightmost bit in the argument) in the file system. Multiple set bits indicate that the file is to be striped on the specified partitions. O_PLACE and O_CREAT must be set if *cbits* is passed as an argument to open( ).

If the file system uses primary and secondary partitions, you should not specify any *cbits* bits for the primary partitions. The file system uses the primary partitions to hold file system metadata (that is, inodes and directories) and the data for small files; it uses the secondary partitions to hold data for large files. If *cbits* bits are specified for primary partitions, those partitions may fill up and thus prevent the file system from creating any new files.

The primary partitions are always the first partitions in a file system. For example, if a file system has 2 primary partitions and 8 secondary partitions, partitions 0 and 1 are primary, and partitions 2 through 9 are secondary.

In a C program, you can use the statfs(2) system call to identify the primary and secondary partitions. The df(1) command with the -p option can be used to identify the primary and secondary partitions.

cblks Specifies the number of 512-word blocks allocated in each partition (as specified in *cbits* per stripe). This number is rounded up to the nearest multiple of the sector size. O_PLACE and O_CREAT must be set if *cblks* is passed as an argument to open().

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across exec(2) system calls. See fcntl(2).

No process may have more than OPEN_MAX file descriptors open simultaneously.

To ensure that data is written to disk immediately, either the O_SYNC flag or both the O_RAW and the O_LDRAW flags should be used.

NOTES

Only a process with appropriate privilege can create a restart file.

Only a process with appropriate privilege can open restricted devices.

If the file has an access control list (ACL), users who are not the file owner have permissions checked with respect to the access control list. Users who are not affected by entries in the access control list have permissions checked with respect to the other mode bits.

Permission to the file is also based on a comparison between the active security label of the process and the security label of the file. These comparisons are summarized as follows:

- For read or execute permission, the active security label of the process must dominate the security label of the file.
- For write permission, the active security label of the process must equal the security label of the file.
- For read permission to pipes, the active security label of the process must be equal to the security label of the file if the SECURE_PIPE configuration option is enabled; otherwise, the active security label of the process must dominate the security label of the file.

If the file is a labeled device, the active security label of the process must fall within the security label range of the device.

Only appropriately authorized users are granted permission to files that are in the OFF state or that are multilevel.

The process must be granted search permission to every component of the path prefix via the permission bits and access control list. The process must be granted search permission to every component of the path prefix via the security label.

If FSETID_RESTRICT is enabled, only processes with appropriate privileges can be granted write permission to set-user-ID or set-group-ID files.

To create a file, the calling process must have write permission to the file's parent directory via the permission bits and access control list. To create a file, the calling process must have write permission to the file's parent directory via the security label.

To create a file, the process must specify a file system that has been labeled as secure. The active security label of the process must fall with the security label range of the file system.

A process with the effective privileges shown are granted the following abilities:

Privilege	Description
PRIV_DAC_OVERRIDE	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
PRIV_DAC_OVERRIDE	The process is granted read, execute, or write permission to the file via the permission bits and access control list.
PRIV_FSETID	If FSETID_RESTRICT is enabled, the process is granted write permission to the set-user-ID or set-group-ID file or is allowed to create a set-user-ID or set-group-ID file.
PRIV_IO	The process is allowed to open restricted devices.
PRIV_MAC_READ	The process is granted search permission to every component of the path prefix via the security label.
PRIV_MAC_READ	The process is granted read or execute permission to the file via the security label.

PRIV_MAC_WRITE	The process is granted write permission to the file via the security label. For file creation, the process is granted write permission to the file's parent directory via the security label.
PRIV_RESTART	The process is allowed to create a restart file.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted read, execute, or write permission to the file. The super user is allowed to create a restart file. The super user or a process with the suidgid permission can override the restriction enabled by the FSETID_RESTRICT system configuration option. The super user is allowed to open restricted devices.

RETURN VALUES

Upon successful completion of open, the file descriptor is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The open system call fails to open the specified file if one of the following error conditions occurs:

Error Code	Description
EACCES	A component of the path prefix denies search permission.
EACCES	The *oflag* permission is denied for the specified file.
EACCES	The file is in the OFF state and the calling process does not have appropriate privilege.
EACCES	The file is a multilevel file and the calling process does not have appropriate privilege.
EACCES	The security label of the file does not allow the requested access.
EACCES	If the FSETID_RESTRICT and PRIV_SU configuration options are enabled, the process does not have appropriate privilege to gain write permission to the set-user-ID or set-group-ID file.
EACCES	The tape device file does not reside in directory /dev/tape, is a diagnostic device, or the device is not configured down.
EAGAIN	The file exists, mandatory file and record locking is set, and there are outstanding record locks on the file (see chmod(2)).
EBUSY	Device is already open.
EDMNBLK	O_NONBLOCK is set and the specified file is migrated offline.
EDMOFF	The specified file is offline, and the data migration facility is not configured in the system.
EEXIST	O_CREAT and O_EXCL are set, and the specified file exists.
EFAULT	The *path* argument points outside the allocated process address space.

EFLNEQ	The active security label of the process falls outside the security label range of the specified file system.
EFSNOTEXCL	The caller is attempting to open a file with O_EXCL, and other processes have the file open.
EINTR	A signal was caught during the open system call.
EINTR	The open system call was interrupted.
EINVAL	O_APPEND was specified in *oflag*, and the file to be opened is a process in the */proc* file system.
EINVAL	O_SSD was used to open a file accessed via NFS.
EISDIR	The specified file is a directory, and *oflag* is write or read and write.
EMANDV	The security label range of a device does not allow the requested access.
EMFILE	OPEN_MAX file descriptors are currently open.
ENOENT	O_CREAT is not set, and the specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The specified file is a character special or block special file, and the device associated with this special file does not exist.
ENXIO	O_NDELAY is set, the specified file is a FIFO special file, O_WRONLY is set, and no process has the file open for reading.
EOFFLIN	The specified file is offline, and automatic file retrieval is disabled.
EOFLNDD	The specified file is offline, and the data management daemon is not currently executing.
EOFLNNR	The specified file is offline, and it is currently unretrievable.
EPERM	O_CREAT and O_RESTART are set, and the effective user ID of the caller is not super user.
EPERM	The caller does not have appropriate privilege to open a restricted device.
EQACT	A file or inode quota limit was reached for the current account ID.
EQGRP	A file or inode quota limit was reached for the current group ID.
EQUSR	A file or inode quota limit was reached for the current user ID.
EROFS	The specified file resides on a read-only file system, and *oflag* is write or read and write.
ESYSLV	The specified file system has not been labeled as a secure file system.
ETPDCNF	The tape subsystem has not been configured.
ETPD_BAD_REQT	There is no path to the device.

ETXTBSY The text file is busy.

FORTRAN EXTENSIONS

The open system call can be called from Fortran as a function (on all systems except Cray MPP systems and CRAY T90 series systems):

```
CHARACTER*n path
INTEGER oflag, mode, OPEN, I
I = OPEN (path, oflag, mode)
```

The *path* argument may also be an integer variable. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte. The PXFOPEN(3F) subroutine provides similar functionality and is available on all Cray Research systems.

EXAMPLES

The following examples illustrate different uses of the open system call.

Example 1: This file, specified by its full path name, is opened for reading only and uses the default buffered I/O. That is, data passes through the system buffer cache because the O_RAW flag has not been specified.

```
int fd
fd = open("/usr/include/fcntl.h", O_RDONLY);
```

Example 2: This open request creates a file named newfile in the current directory (relative path name supplied). The file is given permissions of 0644 adjusted by the current user file-creation mode mask value. All write operations to the file are appended onto the end of the file.

If newfile already exists in the current directory, that file is opened for writing only.

```
int fd;
fd = open("newfile", O_WRONLY | O_CREAT | O_APPEND, 0644);
```

Example 3: The file, whose path name is found at the address specified by the pointer fptr, is created if it does not currently exist. If the file already exists, it is opened with its contents truncated.

All write operations to the file bypass the system buffer cache (raw mode) and write directly to the device.

```
int fd;
char *fptr;
fd = open(fptr, O_WRONLY | O_TRUNC | O_CREAT | O_RAW, 0640);
```

Example 4: This open request creates newfile for writing only.

The exclusive create feature (O_EXCL) indicates that if a file named newfile already exists in the current directory, the open request fails.

```
int fd;
fd = open("newfile", O_WRONLY | O_CREAT | O_EXCL, 0644);
```

Example 5: The following open request with the O_SYNC flag forces each succeeding write operation (for datafile, the file being opened) to wait until the output data has reached the physical device. By default, output data is staged in the system's buffer cache (buffered I/O) and does not reach the device immediately (called delayed I/O).

```
int fd;
fd = open("datafile", O_WRONLY | O_CREAT | O_SYNC, 0600);
```

Using the O_SYNC feature impairs I/O performance.

Example 6: The O_SSD flag on an open request enables data residing in a user's secondary data segments (SDS) area on the SDS to be written directly to the user's data file. For the following open request, succeeding write operations transfer data from the user's SDS area (rather than from the user's process memory) to the user's file sdsdata. The write(2) operation transfers 10 blocks (40,960 bytes) of data, starting at block position 1 (relative word location 512) of the user's SDS area, to the file sdsdata.

```
int fd;
fd = open("sdsdata", O_WRONLY | O_CREAT | O_SSD, 0644);
write(fd, 512, 40960);
```

SEE ALSO

chkpnt(2), chmod(2), close(2), creat(2), dup(2), exec(2), fcntl(2), ialloc(2), lseek(2), mknod(2), read(2), restart(2), setdevs(2), ssbreak(2), statfs(2), umask(2), unlink(2), write(2)

umask(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

PXFOPEN(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

mkfs(8), spdev(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`openi` – Opens a file by using the inode number

SYNOPSIS

```
int openi (long dev, long ino, long gen, long uflag);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `openi` system call presents the user with a flat view of all native UNICOS file systems currently mounted. Rather than use the directory tree structure to search through directories for a file, `openi` provides access by inode number.

The `openi` system call accepts the following arguments:

dev Specifies the device number as built by the `makedev` macro that is defined outside of the kernel.

ino Specifies an inode number for the file as reported by the `ls –i` command.

gen Specifies the generation number of the inode. This provides a unique identification for a specific file. The generation number changes when an inode is reused. To print the inode generation values, use the `fck`(1) command with the `-i` and `-l` options.

uflag Specifies the open flags. These are bit values of the form `O_`*name* that are defined in the `fcntl.h` file.

NOTES

Only a process with appropriate privilege can use this system call.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to use this system call.

A process with the `PRIV_MAC_READ` and `PRIV_DAC_OVERRIDE` effective privileges are allowed to use this system call. See the effective privilege discussion in the NOTES section of the `open`(2) man page for additional privilege requirements. The `open`(2) search access discussions do not apply to this system call.

RETURN VALUES

If `openi` completes successfully, a nonnegative integer is returned which may be used in further I/O operations. Otherwise, `openi` returns a negative value, and `errno` is set to indicate the error.

ERRORS

The `openi` system call fails to open the specified file if one of the error conditions listed on the `open`(2) man page occurs.

EXAMPLES

The following code fragment shows how `openi` is used in a program that examines restart files. The `rv_fs` field is a file system identifier. The `rv_fid` field contains an *ino gen* pair.

```
fd = openi( usr_makedev(krn_major(rvp->rv_fs.val[0]),
                        krn_minor(rvp->rv_fs.val[0])),
            rvp->rv_fid.fid_data[0],
            rvp->rv_fid.fid_data[1], 0);
```

FILES

`/usr/include/fcntl.h`	Contains symbol descriptions for the `open`(2) system call
`/usr/include/sys/sysmacros.h`	Contains a description of the `makedev` macro

SEE ALSO

`open`(2)

`fck`(1), `ls`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`fsck`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

General UNICOS System Administration, Cray Research publication SG–2301

NAME

pathconf, fpathconf – Determines value of file or directory limit

SYNOPSIS

```
#include <unistd.h>
long pathconf (const char *path, int name);
long fpathconf (int fildes, int name);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The pathconf and fpathconf system calls provide a method for an application to determine the current value of a configurable limit or option (variable) that is associated with a file or directory.

The pathconf and fpathconf system calls accept the following arguments:

path Points to the path name of a file or directory.

name Represents the variable to be queried relative to that file or directory.

fildes Specifies an open file descriptor.

The values for *name* are listed below with a brief description of the value each returns.

_PC_LINK_MAX	Returns the maximum number of links allowed per file.
_PC_MAX_CANON	Returns the maximum number of bytes that can be read from a terminal device in canonical mode. The behavior is undefined if *path* or *fildes* does not refer to a terminal file.
_PC_MAX_INPUT	Returns the maximum number of bytes that can be read from a terminal device in raw mode. The behavior is undefined if *path* or *fildes* does not refer to a terminal file.
_PC_NAME_MAX	Returns the maximum number of characters in a file name relative to the file system containing the file specified as the first argument. If *path* or *fildes* refers to a directory, the value returned applies to the file names within the directory. The behavior is undefined if *path* or *fildes* does not refer to a directory.

_PC_PATH_MAX	Returns the maximum number of characters in a path name relative to the file system containing the file specified as the first argument. The behavior is undefined if *path* or *fildes* does not refer to a directory. If *path* or *fildes* refers to a directory, the value returned is the maximum length of a relative path name when the specified directory is the working directory.
_PC_PIPE_BUF	Returns the maximum number of bytes that can be written to a pipe to be assured that the write is atomic. If the number of bytes written to a pipe is less than the value returned by the _PC_PATH_MAX request, then the writing process is assured that the data written is not interleaved with data from another process' write to the same pipe. If *path* refers to a FIFO, or *fildes* refers to a pipe or FIFO, the value returned applies to the referenced object itself. If *path* or *fildes* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *path* or *fildes* refer to any other type of file, the behavior is undefined.
_PC_CHOWN_RESTRICTED	Returns value 1 if chown() is a restricted operation; returns –1 if chown() is unrestricted. The system can be configured such that only users granted permission may use chown(). If *path* or *fildes* refers to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.
_PC_NO_TRUNC	Returns value 1 if path name components longer than NAME_MAX generate an error. Returns value –1 if path name components longer than NAME_MAX do not generate errors (that is, path name components longer than NAME_MAX are truncated to NAME_MAX characters without causing an error condition). If *path* or *fildes* refers to a directory, the value returned applies to the file names within the directory. The behavior is undefined if *path* or *fildes* does not refer to a directory.
_PC_VDISABLE	Returns the disable character for terminal devices. Terminal special control characters defined in termios.h can be disabled using this character value. The behavior is undefined if *path* or *fildes* does not refer to a terminal file.

NOTES

The process must be granted read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

To be granted search permission to a component of the path prefix (for the `pathconf` system call), the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path prefix via the permission bits and access control list. (Only for `pathconf` system call.)
`PRIV_MAC_READ`	The process is granted search permission to a component of the path prefix via the security label. (Only for `pathconf` system call.)
`PRIV_MAC_READ`	The process is granted read permission to the file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to use this system call and is granted search permission to every component of the path prefix. If the `PRIV_SU` configuration option is enabled, the super user is granted read permission to the file via the security label.

RETURN VALUES

If *name* is an invalid value, the `pathconf` and `fpathconf` system calls return a value of –1.

If the variable corresponding to *name* has no limit for the path or file descriptor, `pathconf` and `fpathconf` return a value of –1 without changing `errno`.

If *path* determines the value of *name* and *name* is not associated with the file specified by *path*, or if the process did not have the appropriate privileges to query the file specified by *path*, or if *path* does not exist, `pathconf` returns a value of –1.

If *fildes* determines the value of *name* and *name* is not associated with the file specified by *fildes*, or if *fildes* is an invalid file descriptor, `fpathconf` returns a value of –1. Otherwise, `pathconf` and `fpathconf` return the current variable value for the file or directory without changing `errno`.

ERRORS

The `pathconf` or `fpathconf` system call returns a value of –1 and sets `errno` to the corresponding value if one of the following conditions occurs:

Error Code	Description
`EINVAL`	The value of the name is not valid.
`EPERM`	The process does not have appropriate privilege to use this system call.

The pathconf system call returns a value of –1 and sets errno to the corresponding value if one of the following conditions occurs:

Error Code	Description
EACCES	Search permission is denied for a component of the path prefix.
EACCES	The process is denied read permission to the file via the security label.
EINVAL	The variable name is not associated with the specified file.
ENAMETOOLONG	The length of the *path* argument exceeds PATH_MAX, or a path name component is longer that NAME_MAX when _POSIX_NO_TRUNC is in effect.
ENOENT	The specified file does not exist or the *path* argument points to an empty string.
ENOTDIR	A component of the path prefix is not a directory.

The fpathconf system call returns a value of –1 and sets errno to the corresponding value if one of the following conditions occurs:

Error Code	Description
EBADF	The *fildes* argument is not a valid file descriptor.
EBADF	The process is denied read permission to the file via the security label.
EINVAL	The variable name is not associated with the specified file.

EXAMPLES

This example illustrates various applications of the pathconf system call for files and directories residing in the user's home (HOME) directory.

```
#include <unistd.h>

main()
{
    char path[100], *ptr;
    long name_max;

    if (ptr = getenv("HOME")) {
         strcpy(path, ptr);
    }
    else {
         fprintf(stderr, "getenv failed to locate HOME!\n");
         exit(1);
    }

    printf("Configurable parameters for files within %s:\n", path);
    printf("    maximum # of links = %ld\n", pathconf(path, _PC_LINK_MAX));
    name_max = pathconf(path, _PC_NAME_MAX);
    printf("    maximum # of chars in a filename = %ld\n", name_max);
    printf("    maximum # of chars in a path name = %ld\n",
                                                  pathconf(path, _PC_PATH_MAX));
    printf("    maximum # of bytes for atomic writes to a pipe = %ld\n",
                                                  pathconf(path, _PC_PIPE_BUF));
    if (pathconf(path, _PC_CHOWN_RESTRICTED) == -1) {
         printf("    chown is unrestricted\n");
    }
    else {
         printf("    chown is restricted\n");
    }
    if (pathconf(path, _PC_NO_TRUNC) == -1) {
         printf("    path name components longer than %d char's ", name_max);
         printf("do not generate errors;\n");
         printf("        (that is, the system will use only the first ");
         printf("%d characters)\n\n", name_max);
    }
    else {
         printf("    path name components longer than %d char's ", name_max);
         printf("generate errors\n\n");
    }

    printf("Configurable parameters for terminals:\n");
    printf("    maximum # bytes for canonial reads = %ld\n", fpathconf(0, _PC_M AX_CANON));
    printf("    maximum # bytes for raw reads = %ld\n", fpathconf(0, _PC_MAX_IN PUT));
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `pathconf` and `fpathconf` system calls

SEE ALSO

`sysconf`(2)

`getconf`(1), `sysconf`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

pause – Suspends process until signal

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The pause system call suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process. The pause system call causes an implicit sigon (see sigoff(3C)).

On Cray MPP systems, the pause system call suspends the process only for the PE on which it is called. It has no effect on any other PE of the application.

RETURN VALUES

If the signal causes termination of the calling process, pause does not return.

If the signal is caught by the calling process and control is returned from the signal-catching function (see signal(2)), the calling process resumes execution from the point of suspension, a value of −1 is returned, and errno is set to EINTR.

FORTRAN EXTENSIONS

The pause system call can be called from Fortran as a function:

```
INTEGER PAUSE, I
I = PAUSE ( )
```

Alternatively, pause can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable:

```
CALL PAUSE ( )
```

EXAMPLES

This example shows how to use the pause system call to make a process wait for a specific signal.

The pause system call suspends a process until any signal is received.

In this example, the use of the sigsetmask(2) system call with pause enables the request to delay the program until receipt of a SIGUSR1 signal.

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

int omask, nmask;

main()
{
     void catch(int signo);

     signal(SIGUSR1, catch);

     /* Process performs work here, but after finishing work and before
        proceeding, it needs to wait for a SIGUSR1 signal to be sent
        from another process. */

     nmask = ~sigmask(SIGUSR1);/* enable all bits in mask except SIGUSR1 */
     omask = sigsetmask(nmask);/* hold all signals except SIGUSR1 */
     pause();                  /* wait for SIGUSR1 signal only */

     /* Work continues here after waiting and catching SIGUSR1 signal */
}

void catch(int signo)
{
     sigsetmask(omask);        /* restore signal hold mask after signal
                                  is received */
}
```

FILES

/usr/include/unistd.h	Contains C prototype for the pause system call

SEE ALSO

alarm(2), kill(2), signal(2), sigsetmask(2), wait(2)

sigoff(3C) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

pipe – Creates an interprocess channel

SYNOPSIS

```
#include <unistd.h>
int pipe (int fildes[2]);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The pipe system call creates an I/O mechanism called a pipe and returns two file descriptors. It accepts the following argument:

fildes Specifies the file descriptors returned. These are *fildes*[0] and *fildes*[1]; *fildes*[0] is opened for reading, and *fildes*[1] is opened for writing.

A maximum of MAXPIPE data is buffered by the pipe before the writing process is blocked. MAXPIPE, defined in the /usr/src/uts/c1/cf/config.h header file, specifies the number of data blocks (4096 bytes each) reserved in the pipe's kernel buffer space. A read on file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a FIFO special file basis.

NOTES

The active security label of the process must fall within the security label range of the root file system.

RETURN VALUES

If pipe completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The pipe system call fails if one of the following error conditions occurs:

Error Code	Description
EMFILE	OPEN_MAX –1 or more file descriptors are currently open.
ENFILE	The system file table is full.
ENOSPC	During a write(2) to an ordinary file, the free space left on the device was exhausted.

EFLNEQ The active security label of the process does not fall within the security label range of the root file system.

FORTRAN EXTENSIONS

The `pipe` system call can be called from Fortran as a function:

```
INTEGER fildes(2), PIPE, I
I = PIPE (fildes)
```

EXAMPLES

This example shows how to use the `pipe` system call to create a system pipe. (Some system calls in the example are not supported on Cray MPP systems.) Because a system pipe can transfer data only between related processes, such as a parent and child or siblings, this example shows the essential elements of parent and child processes needed for data transfer by a system pipe.

```
/* This is the parent side of the system pipe example. It delivers data to
   the child process using a system pipe. */
#include <stdio.h>
#include <unistd.h>

main()
{
     int fd[2];
     char rfd[10], wfd[10];

     if (pipe(fd) == -1) {          /* create system (unnamed) pipe */
          perror("creating system pipe failed");
          exit(1);
     }

     if (fork() == 0) {             /* create child process */
          sprintf(rfd, "%d", fd[0]); /* convert pipe's file descriptors - */
          sprintf(wfd, "%d", fd[1]); /* to strings to pass as arguments */
          execl("child_prog", "child_prog", rfd, wfd, 0);
          perror("execl failed");
          exit(1);
     }

     close(fd[0]);                  /* parent closes its read access to
                                       pipe since the  parent will write
                                       to the pipe */

     /* In this part of program, the parent writes to fd[1] to deliver data
         to the pipe. */

     close(fd[1]);                  /* parent closes its write access to pipe -
```

```
                                        required for the child to detect an EOF
                                        condition */
     wait((int *)0);                 /* parent waits for child to complete */
}

/* This is the child side of the system pipe example. It receives data
   from the parent process on a system pipe. */


main(int argc, char *argv[])
{
     int rfd, wfd;

     rfd = atoi(argv[1]);            /* since pipe's file descriptors were
     wfd = atoi(argv[2]);            /* passed as arguments, the string
                                        arguments are converted back to
                                        integers */


     close(wfd);                     /* child closes its write access
                                        to pipe since the child will read
                                        from the pipe - also required for child
                                        to detect an EOF condition */

     /* In this part of program, the child reads from rfd (fd[0]) to receive
         data from the pipe. */

     close(rfd);                     /* child closes its read access
                                        to the pipe and then exits */
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `pipe` system call
`/usr/src/uts/c1/cf/config.h`	Contains `MAXPIPE` definition

SEE ALSO

`read`(2), `write`(2)

`ksh`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

plock – Locks process in memory

SYNOPSIS

```
#include <sys/lock.h>
int plock (int op);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The plock system call allows the calling process to lock itself in memory. Locked processes are immune to all routine swapping. The plock system call also allows these segments to be unlocked. Only a process with appropriate privilege can use this system call.

The plock system call accepts the following argument:

op Specifies a locking option. The following are valid *op* values:

DATLOCK	Locks data in memory (data lock).
DLYSHUFFLE	Locks process in memory, movable (process lock); does not force process to low-memory address immediately.
NOSHUFFLE	Locks process in memory, not movable (process lock).
PROCLOCK	Locks process in memory, movable (process lock).
TXTLOCK	Locks text in memory (text lock).
UNLOCK	Removes locks.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_RESOURCE	The process is allowed to use this system call.

If the PRIV_SU configuration option is enabled, the super user or a process with the PERMBITS_PLOCK permbit is allowed to use this system call.

RETURN VALUES

If plock completes successfully, a value of 0 is returned to the calling process; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The `plock` system call fails if one of the following error conditions occurs:

Error Code	Description
EINVAL	The *op* argument is equal to PROCLOCK, and a process lock already exists on the calling process.
EINVAL	The *op* argument is equal to NOSHUFFLE, and a process lock already exists on the calling process.
EINVAL	The *op* argument is equal to TXTLOCK, and a text lock or a process lock already exists on the calling process.
EINVAL	The *op* argument is equal to DATLOCK, and a data lock or a process lock already exists on the calling process.
EINVAL	The *op* argument is equal to UNLOCK, and no type of lock exists on the calling process.
EINVAL	The *op* argument is equal to TXTLOCK or DATLOCK, and the program is not compiled with split code and data.
EINVAL	The *op* argument is equal to DLYSHUFFLE, and a process lock already exists on the calling process.
EPERM	The process does not have appropriate privilege to use this system call.

FORTRAN EXTENSIONS

The `plock` system call can be called from Fortran as a function:

```
INTEGER op, PLOCK, I
I = PLOCK (op)
```

SEE ALSO

exec(2), exit(2), fork(2)

NAME

policy – Returns or sets information on the CPU allocation policy

SYNOPSIS

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

int policy (int function, void *address, int action);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The policy system call allows site selection of CPU allocation policy (deferred implementation) without requiring changes to the priority adjustment mechanism in the UNICOS kernel. This system call is also used by shrdaemon(8) to access the sh_consts structure in the kernel.

The policy system call accepts the following arguments:

function Specifies the CPU allocation policy to be affected by the *action*. The following policies are available:

- FAIR_SHARE — Specifies the standard (default) CPU allocation policy for the fair-share scheduler.
- BANK_POINTS — (Deferred implementation) Specifies the *bank points* CPU allocation policy, which provides a "bank account" of resources that is depleted through use.

address Specifies the location of the policy-definition structure. For example, the sh_consts structure, defined in the include file sys/share.h, is used for the standard fair-share information.

action Specifies the action to be performed. The following values are available:

- GET_COSTS — Retrieves the contents of the system sh_consts structure.
- SET_COSTS — Sets the contents of the system sh_consts table. This action acts on only those parameters that can be changed at the user level; for example, the counts and maximum value fields are not updated. Only a process with appropriate privilege can specify this function.
- MOD_MXUSG — Sets the maximum usage value field (sc_mxcusage) of the shconst structure. Only a process with appropriate privilege can specify this function. This action is not valid with the BANK_POINTS function.

NOTES

The implementation of the BANK_POINTS function is deferred. The following actions are currently available (the example address &shconsts assumes a declaration of the form struct sh_consts shconsts):

- policy(FAIR_SHARE, &shconsts, GET_COSTS)
- policy(FAIR_SHARE, &shconsts, SET_COSTS)
- policy(FAIR_SHARE, &shconsts, MOD_MXUSG)

A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_RESOURCE	The process is allowed to specify the SET_COSTS and MOD_MXUSG functions.

If the PRIV_SU configuration option is enabled, the super user or a process with the PERMBITS_SYSPARAM permbit is allowed to specify the SET_COSTS and MOD_MXUSG functions.

RETURN VALUES

If policy completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The policy system call fails if one of the following error conditions occurs:

Error Code	Description
EFAULT	The *address* of the structure points to invalid data.
EINVAL	Either the *function* or *action* argument is invalid. This error is returned if the deferred BANK_POINTS function is specified.
EPERM	The process does not have appropriate privileges for the SET_COSTS and MOD_MXUSG actions.

SEE ALSO

share(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

shradmin(8), shrdaemon(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

UNICOS Resource Administration, Cray Research publication SG–2302

NAME

`profil` – Generates an execution time profile

SYNOPSIS

```
#include <unistd.h>
void profil (long *buf, int bufsiz, int offset, int scale, int rate);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `profil` system call generates an execution time profile of the user's program. It accepts the following arguments:

buf Points to a memory area.

bufsiz Specifies the length (in bytes) of the memory area.

offset Specifies the number subtracted from the user's program counter (PC) every interval specified by *rate*.

scale Specifies the multiplier of the difference resulting from subtracting *offset* from the user's PC. It is interpreted as an unsigned, fixed-point fraction with binary point at the left. If the resulting number corresponds to a word inside *buf*, that word is incremented. The octal number 037777777777 gives a one-to-one mapping of PCs to words in *buf*; the octal number 017777777777 maps each pair of instruction words together; the octal number 02 maps all instructions onto the first word in *buf* producing a noninterrupting core clock.

rate Specifies the rate in microseconds at which the program is sampled. Default is 1/100 second if *rate* is 0 or is not specified (Cray PVP systems).

The `profil` system call is inoperative under the following conditions:

- An update in *buf* would cause a memory fault.
- The *bufsiz* argument is 0.
- The *scale* argument is 0 or 1.
- The `exec`(2) system call is executed.

The `profil` system call remains operative in both a child process and parent process after processing a `fork`(2) system call.

RETURN VALUES

None

FILES

`/usr/include/unistd.h`	Contains C prototype for the `profil` system call

SEE ALSO

`exec`(2), `fork`(2)

NAME

ptrace – Traces processes

SYNOPSIS

```
#include <unistd.h>
long ptrace (int request, int pid, long addr, long data);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The ptrace system call provides a means by which a parent process may control the execution of a child process. Its primary use is for the implementation of breakpoint debugging (see adb(1)). The child process behaves normally until it encounters a signal (see signal(2) for the list), at which time it enters a stopped state and its parent process is notified through wait(2). When the child process is in the stopped state, its parent process can examine and modify its core image, using ptrace. Also, the parent process can cause the child process either to terminate or to continue, with the possibility of ignoring the signal that caused it to stop.

The ptrace arguments are as follows:

request Identifies the action to be taken by ptrace. The following are valid values for *request*:

- `0` This request must be issued by the child process if it is to be traced by its parent process. It turns on the child process' trace flag, stipulating that the child process should be left in a stopped state on receipt of a signal rather than the state specified by *func*; see signal(2). The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. If the parent process does not expect to trace the child process, peculiar results occur.

  The remainder of the requests can be used only by the parent process. The child process must be in a stopped state before these requests are made.

- `1, 2` With these requests, the word at location *addr* in the address space of the child process is returned to the parent process. Requests 1 and 2 produce equal results. The *data* argument is ignored.

- `3` With this request, the word at location *addr* in the child process' USER area in the system's address space (see the sys/user.h file) is returned to the parent process. Addresses in this area range from 0 to *sizeof* (structure user). The *data* argument is ignored. If *addr* is outside the USER area, this request fails.

- `4, 5` With these requests, the value given by the *data* argument is written into the address space of the child process at location *addr*. Using either request 4 or 5 causes equal results. On successful completion, the value written into the address space of the child process is returned to the parent process.

`6` With this request, you can write a set of limited fields in the child process' USER area. The *data* argument gives the value to be written, and *addr* is the location of the entry. You can write the following entries:

- The general registers (that is, A, S, and V registers)
- The vector mask (VM) and vector length (VL) special registers
- B and T registers

`7` This request causes the child process to resume execution. If the *data* argument is 0, all pending signals, including the one that caused the child process to stop, are canceled before it resumes execution. If the *data* argument is a valid signal number, the child process resumes execution as if it had incurred that signal, and any other pending signals are canceled. The *addr* argument must be 1. If the *data* argument is 0, the child process resumes execution where it entered the stopped state. On successful completion, the value of *data* is returned to the parent process.

`8` This request causes the child process to terminate with the same consequences as those of `exit`(2).

`10` With this request, the word at location *addr* in the child process' UCOMM area in the system's address space (see `sys/user.h`) is returned to the parent process. Addresses in this area range from 0 to *sizeof* (structure `ucomm`). The *data* argument is ignored. If *addr* is outside the UCOMM area, this request fails.

`11` With this request, you can write a few entries in the child process' UCOMM area. The *data* argument gives the value to be written, and *addr* is the location of the entry. You can write the following entries:

- The semaphores
- Shared B registers
- Shared T registers

pid Specifies the process ID of the child process when *request* is 1 through 8. The *pid* argument is ignored when *request* equals 0.

addr Specifies a location that varies in meaning depending on the value of *request*.

data Specifies information supplied to or received from the target process; this information varies in meaning depending on the value of *request*.

To prevent fraud, `ptrace` inhibits the set user ID facility on subsequent `exec`(2) calls. If a traced process calls `exec`(2), it stops before executing the first instruction of the new image showing the `SIGTRAP` signal.

NOTES

On a UNICOS system using privilege assignment lists (PALs), a privileged process should not have itself traced. The security policy could be circumvented if a privileged process is traced by a nonprivileged parent process. This is also true if the PRIV_SU configuration option is enabled, although the phrase "privileged process" means a process owned by root on a PRIV_SU system.

To retrieve information about a child process, the active security label of the process must be greater than or equal to the security label of the child.

To set information about or modify the state of a child process, the active security label of the process must equal the security label of the child.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_MAC_READ	The process is allowed to retrieve information about a child process regardless of the security label of the child.
PRIV_MAC_WRITE	The process is allowed to set information about or modify the state of a child process regardless of the security label of the child.

If the PRIV_SU configuration option is enabled, the super user is allowed to override the security label restrictions.

CAUTIONS

Requests to read can return a valid data value of –1, which can be confused with an error return value. errno is cleared by the library interface; therefore, if ptrace returns –1 and errno is nonzero, an error has occurred.

RETURN VALUES

If ptrace completes successfully, any requested value is returned; otherwise, a value of –1 is returned to the parent process, and the errno of the parent process is set to indicate the error.

ERRORS

The ptrace system call fails if one of the following conditions occurs:

Error Code	Description
EIO	The *request* argument is an illegal number, or when *request* is 7, *data* is not 0 or a valid signal number.
ESRCH	The *pid* argument identifies a child process that does not exist or that has not executed a ptrace with request 0.
ESRCH	The process does not meet security label requirements and does not have appropriate privilege.

FORTRAN EXTENSIONS

The ptrace system call can be called from Fortran as a function:

```
INTEGER request, pid, addr, data, PTRACE, I
I = PTRACE (request, pid, addr, data)
```

FILES

/usr/include/unistd.h	Contains C prototype for the ptrace system call

SEE ALSO

exec(2), exit(2), signal(2), wait(2)

adb(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

proc(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

ptyrecon – Manages pty reconnection

SYNOPSIS

```
#include <sys/ptyrecon.h>
int ptyrecon (int cmd, struct reconcntl *reconp);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The ptyrecon system call enables or disables pty disconnection, and reconnects, searches, or hangs up disconnected sessions.

If disconnection is enabled, a session does not disappear on a pty master side close operation, but remains disconnected for a specified amount of time. Users can later search, reconnect to, or hang up disconnected sessions.

The ptyrecon system call accepts the following arguments:

cmd Indicates the operation to be performed.

RECON_ENABLE Enables reconnection on a pty. After closing the connection (by using the telnet close command or equivalent) the terminal remains in a disconnected state for the time indicated in the *distimeo* field of the reconcntl structure. If this value is 0, the default DISTIMEO is used.

RECON_DISABLE Disables reconnection that was enabled with RECON_ENABLE.

RECON_HANGUP Hangs up a terminal that is in a disconnected state.

RECON_SEARCH Fills the reconcntl structure with the data corresponding to the first disconnected pty greater than or equal to the pty number that was passed to the kernel. If the reconcntl flag RECON_ANYUSER is set, it returns the following message:

```
Disconnected session owned by any user (superuser only).
```

Otherwise, it returns the following message:

```
Sessions owned by the caller
```

RECON_CONNECT Connects the current terminal to a disconnected session in another pty, and terminates the current session.

reconp Points to a two-way communication structure. All of the operations act on the pty specified in the *pty* field of the `reconcntl` structure. If this field is set to `RECON_CURRPTY`, the operation acts on the current (controlling) pty. Following is the `reconcntl` structure, followed by a list of the possible operations:

```
struct reconcntl {
        int     pty;            /* pty number */
        int     uid;            /* session owner uid */
        pid_t   pid;            /* process-id of session leader */
        time_t  distime;        /* seconds disconnected */
        time_t  distimeo;       /* max disconnect time (sec) */
        int     flags;
};
```

RETURN VALUES

If `ptyrecon` completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `ptyrecon` system call fails if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	The *reconp* parameter is bad.
`EINVAL`	The *cmd* parameter is bad.
`ENOENT`	No more search entries exist.
`ENOTTY`	No controlling tty exists.
`ENXIO`	The pty number is bad.
`EPERM`	You are not allowed to perform the operation.

SEE ALSO

`ptyrecon`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR−2011

NAME

`quotactl` – Manipulates file system quotas

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/quota.h>

int quotactl (char *spec, int request, char *arg);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `quotactl` system call manipulates disk quotas. The arguments are as follows:

spec Points to the name of the file system. The pointer may be either the device node name or the directory on which the device is mounted.

request Specifies type of request. Types are defined in **Request Types**.

arg Specifies address of a request-specific data structure described with each *request*. All structures are defined in the `sys/quota.h` file.

You can look at your own quota data and header information; however, only an appropriately privileged process can view all the records and change information. The requests that get and set information can be used only on file systems that have been mounted and activated with one of the `Q_ON_`*xxxx* requests. The following are valid values for *request*.

Request Types

The following types of requests are supported:

`Q_ON_COUNT` Appropriately privileged process only. Turns on quotas for the file system specified in *spec* but maintains only counts. *arg* points to the full name of the quota file to associate with this file system.

`Q_ON_INFORM` Appropriately privileged process only. Turns on quotas for the file system specified in *spec*, maintain counts, and issues warning and quota limit messages, but does not enforce quota limits. *arg* points to the full name of the quota file to associate with this file system.

`Q_ON_ENFORCE` Appropriately privileged process only. Turns on quotas for the file system specified in *spec*, maintains counts, issues warning and quota limit messages, and enforces quota limits. This is considered the normal mode of quota operation; the other modes are for evaluation or test use. *arg* points to the full name of the quota file to associate with this file system.

Q_GETQUOTA Returns the quota information for the ID specified in the qf_entry.id field from the file system specified in *spec*. If the request is not from an appropriately privileged process, only information related to the caller's *uid*, authorized *gid*s, and the currently active *acid*s can be obtained. *qf_magic* must be set to QF_MAGIC as defined in the sys/quota.h file. *arg* points to a q_request structure.

Q_SETQUOTA Appropriately privileged process only. Changes selected quota information for the ID specified in the qf_entry.id field on the file system specified in *spec*. Selection is through the acct, group, and user flag fields in the q_request structure. *qf_magic* must be set to QF_MAGIC as defined in the sys/quota.h file. *arg* points to a q_request structure.

For the Q_GETQUOTA and Q_SETQUOTA requests, *arg* points to a q_request structure defined in sys/quota.h:

```
struct q_request   {
    long     qf_magic;   magic number
    int      acct;       QFL_xxx account flags
    int      group;      QFL_xxx group flags
    int      user;       QFL_xxx user flags
    struct   qf_entry
             qf_entry;   quota record
};
```

The fields of struct qf_entry that contain valid information are indicated through the acct, group, and user fields. Only the values in the flagged fields are defined. The caller must set the correct acct, group, and user flags when making a Q_SETQUOTA request; the kernel sets the flags when returning information from a Q_GETQUOTA request. The following table shows the values that are returned and their associated meanings:

QFL_F1	(1<< 0)	Evaluator's field 1
QFL_F2	(1<< 1)	Evaluator's field 2
QFL_F3	(1<< 2)	Evaluator's field 3
QFL_F4	(1<< 3)	Evaluator's field 4
QFL_F5	(1<< 4)	Evaluator's field 5
QFL_FQ	(1<< 5)	File quota
QFL_FR	(1<< 6)	Soft file quota (runquota)
QFL_FT	(1<< 7)	File warning time
QFL_FU	(1<< 8)	File usage
QFL_FW	(1<< 9)	File warning
QFL_IQ	(1<< 10)	Inode quota
QFL_IU	(1<< 11)	Inode usage
QFL_IW	(1<< 12)	Inode warning

Q_GETHEADER Returns the header information for the file system specified in the *spec* argument. *qf_magic* must be set to QF_MAGIC. *arg* points to a qf_header structure.

Q_SETHEADER Appropriately privileged process only. Changes the header information for the file system specified in *spec*. All the header information, except for the qf_name field (which cannot be altered through this interface), is changed by this request; therefore, it is recommended that you use a Q_GETHEADER request to preset the qf_header structure to preserve information you do not want to change. *arg* points to a qf_header structure.

For the Q_GETHEADER and Q_SETHEADER requests, *arg* points to the following structure:

```
struct   qf_header  {
    long   qf_magic;           quota version identification
    struct q_header
           acct_h,             account header
           group_h,            group header
           user_h;             user header
    time_t qf_min_dm;          minimum data migration threshold
    uint   qflvl : 8,          Q_ON_DEFAULT enable level
           qf_eval : 8,        evaluator selector
           qf_spare : 48;      reserved
    long   hef1,               field 1 reserved for evaluator's use
           hef2;               field 2 reserved for evaluator's use
    uint   qf_qfnamesize;      size of qf_name[]
    uint   qf_hashents;        length of the hash table in entries
    off_t  qf_hashtaboffs;     offset of the hash table
    char   qf_name[PATH_MAX+1]; name of the quota file
};
```

The header contains all of the default values used for the quota control file used by the specified file system.

See the sys/quota.h file for a description of the fields.

Q_FSINFO Returns file system specific information. File size, quota enforcement level, and other information specific to a one quota control file is returned. *arg* points to a q_fsinfo structure.

```
struct  q_fsinfo  {
    int    count;    count of file systems using this quota file
    in     errors;   count of errors on this quota file
    long   level;    quota enforcement level
    long   size;     size of the quota file in bytes
};
```

Q_GENINFO Returns generic quota enforcement information in a q_geninfo structure pointed to by *arg*.

```
struct q_geninfo   {
    long   q_types;     mask of configured quota types
    long   q_nquota;    total number of quota entries
    long   q_curact;    current number of active quota entries
    long   q_maxact;    maximum number of active quota entries
    long   q_put;       qput calls
    long   q_updat;     qupdat calls
    long   q_get;       qget calls
    long   q_read;      qread calls
    long   q_cache;     count of inode cache flushes
    long   q_wait;      count of number of quota_wait sleeps
    long   q_readch;    count of hash chain reads
};
```

The configured quota types mask (`q_types`) has a value for each configured combination of ID classes, as shown in the following list:

Value	ID classes
1	User ID
2	Group ID
3	User ID and group ID
4	Account ID
5	User ID and account ID
6	Group ID and account ID
7	User ID, group ID, and account ID

`Q_ON_DEFAULT` Appropriately privileged process only. Turns on quotas for the file system specified in *spec*. The enforcement mode is the mode stored in the `qf_lvl` field of `struct qf_header` of the specified quota file. *arg* points to the name of the quota file to associate with this file system.

NOTES

To be granted read permission to the quota file, the caller's active security label must be greater than or equal to the security label of the quota file.

To be granted write permission to the quota file, the caller's active security label must equal the security label of the quota file.

To be granted `searcdh` permission to a component of the *spec* path prefix, the caller's active security label must be greater than or equal to the security label of the component.

A process with the effective privilege shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the *spec* path prefix via the permission bits and access control list.

PRIV_DAC_OVERRIDE	The process is granted read and write permission to the quota file via the permission bits and access control list.
PRIV_MAC_READ	The process is granted search permission to every component of the *spec* path prefix via the security label.
	The process is granted read permission to the quota file via the security label.
PRIV_MAC_WRITE	The process is granted write permission to the quota file via the security label.
PRIV_RESOURCE	The process is allowed to specify `quotactl` requests that are restricted to processes with appropriate privilege.

If the PRIV_SU configuration option is enabled, the super user is allowed to override all file protections and is allowed to specify `quotactl` requests that are restricted to processes with appropriate privilege.

RETURN VALUES

If `quotactl` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `quotactl` system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	The caller does not have read and/or write permission to the quota file.
EACCES	Search permission is denied for a component of the *spec* path prefix.
EACCES	The magic number in the quota file does not match the one used by the kernel.
EBUSY	The quota feature software is already running on file system *spec*.
EFAULT	The *arg* argument points outside the allocated process address space.
EINVAL	The *request* argument specified was not valid.
EIO	The kernel could not read and/or update the quota entry specified by argument *arg*.
ENODEV	The *spec* argument is not the root of a file system.
ENOENT	The quota feature software is not running on file system *spec*.
EPERM	The process does not have appropriate privilege to perform the requested action.
EPERM	The *spec* argument specifies a nonnative file system.

EXAMPLES

The following program illustrates two features of the `quotactl` system call: `Q_GETQUOTA` and `Q_GETHEADER`. The program is invoked specifying one argument, the path name of the file system for which quota information is to be retrieved.

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/quota.h>
#include <time.h>
#include <stdio.h>

main(int argc, char *argv[])
{
     struct q_request request;
     struct qf_header header;
     int id;

     if (argc < 2) {
          fprintf(stderr, "File system path name not supplied as argument\n");
          exit(0);
     }

     id = getuid();
     request.qf_entry.id = id;
     request.qf_magic = QF_MAGIC;

     if (quotactl(argv[1], Q_GETQUOTA, &request) == -1) {
          perror("quotactl (Q_GETQUOTA) failed");
          exit(1);
     }
     printf("file system name = %s\n", argv[1]);
     printf("account flags = %o\n", request.acct);
     printf("group flags = %o\n", request.group);
     printf("user flags = %o\n\n", request.user);
     if (request.user) {
          printf("Quotas for user %d are as follows:\n\n", id);
          if (request.qf_entry.user_q.f_quota == QFV_DEFAULT) {
               printf("The default file quota is in effect - see below\n");
          }
          else {
               printf("File quota = %ld blocks\n",
                                             request.qf_entry.user_q.f_quota);
          }
          printf("File usage = %ld blocks\n", request.qf_entry.user_q.f_use);
          if (request.qf_entry.user_q.f_warn == QFV_DEFAULT) {
               printf("The default file quota warning window is in effect ");
               printf("- see below\n");
          }
          else {
               printf("File quota warning window = %d blocks\n",
                                              request.qf_entry.user_q.f_warn);
          }
          if (request.qf_entry.user_q.i_quota == QFV_DEFAULT) {
```

```
                    printf("The default inode quota is in effect - see below\n");
               }
               else {
                    printf("Inode quota = %d\n", request.qf_entry.user_q.i_quota);
               }
               printf("Inode usage = %d\n", request.qf_entry.user_q.i_use);
               if (request.qf_entry.user_q.i_warn == QFV_DEFAULT) {
                    printf("The default inode quota warning window is in effect ");
                    printf("- see below\n");
               }
               else {
                    printf("Inode quota warning window = %d\n",
                                                  request.qf_entry.user_q.i_warn);
               }
               if (request.qf_entry.user_q.f_wtime != 0) {
                    printf("Time when file warning was reached = %s\n",
                                       ctime(&request.qf_entry.user_q.f_wtime));
               }
          }

          header.qf_magic = QF_MAGIC;
          if (quotactl(argv[1], Q_GETHEADER, &header) == -1) {
               perror("quotactl (Q_GETHEADER) failed");
               exit(2);
          }
          printf("\nDefault quotas for file system %s:\n\n", argv[1]);
          printf("File quota = %ld blocks\n", header.user_h.def_fq);
          printf("File warning window = %d blocks\n", header.user_h.warn_fq);
          printf("File quota warning fraction = %3.1f\n", header.user_h.wf_fq);
          printf("Inode quota = %d\n", header.user_h.def_iq);
          printf("Inode warning window = %d\n", header.user_h.warn_iq);
          printf("Inode quota warning fraction = %3.1f\n", header.user_h.wf_iq);
     }
```

SEE ALSO

mount(2)

NAME

`read` – Reads from file

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
ssize_t read (int fildes, void *buf, size_t nbyte);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `read` system call tries to read a specified number of bytes from a file into a specified buffer. It accepts the following arguments:

fildes Specifies a file descriptor. It is obtained from an `accept`(2), `creat`(2), `dup`(2), `fcntl`(2), `open`(2), `pipe`(2), `socket`(2), or `socketpair`(2) system call

buf Points to the buffer into which the data is to be read.

nbyte Specifies the number of bytes to be read.

On devices capable of seeking, the `read` starts at a position in the file given by the file pointer associated with *fildes*. On return from `read`, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

On successful completion, `read` returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line (see `ioctl`(2) and `termio`(4)), or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

When you try to read from an empty pipe (or FIFO special file), the following occurs:

- If `O_NDELAY` is set, the read returns a 0.
- If `O_NONBLOCK` is set, the read returns a –1.
- If `O_NDELAY` and `O_NONBLOCK` are both clear, the read blocks until data is written to the file or the file is no longer open for writing.

When you try to read a file associated with a tty that has no data currently available, the following occurs:

- If O_NDELAY is set, the read returns a 0.
- If O_NONBLOCK is set, the read returns a –1.
- If O_NDELAY and O_NONBLOCK are both clear, the read blocks until data becomes available.

When you try to read from a regular file that has mandatory file and record locking set (see chmod(2)), and a blocking write lock exists on the segment of the file to be read (that is, it is owned by another process):

- If either O_NDELAY or O_NONBLOCK is set, the read returns –1 and sets errno to EAGAIN.
- If O_NDELAY and O_NONBLOCK are both clear, the read sleeps until the blocking record lock is removed.

NOTES

The process must be granted read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_MAC_READ	The process is granted read permission to the file via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted read permission to the file via the security label.

RETURN VALUES

If read completes successfully, a nonnegative integer is returned, indicating the number of bytes actually read; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The read system call fails if one of the following error conditions occurs:

Error Code	Description
EAGAIN	Mandatory file and record locking was set, O_NDELAY was set, and there was a blocking record lock.
EBADF	The *fildes* argument is not a valid file descriptor open for reading.
EBADF	The active security label is not greater than or equal to the security label of the file, and the process does not have appropriate privilege.
EDEADLK	The read was going to go to sleep and cause a deadlock situation to occur.
EFAULT	The *buf* argument points outside the allocated address space.
EINTR	A signal was caught during the read system call.

EINVAL	The call contains an argument that is not valid such as the dismounting of a nonmounted device, the mention of an undefined signal in `signal`(2) or `kill`(2), or the reading or writing of a file for which `lseek`(2) has generated a negative pointer. This error is also set by the math functions described in the (3) entries.
ENOLCK	The system record lock table was full; so the read could not go to sleep until the blocking record lock was removed.
ENXIO	During a read or write on a special file, a subdevice that does not exist or is beyond the limits of the device was referenced.

EXAMPLES

The following examples illustrate different uses of the `read` system call.

Example 1: This example shows a simple `read` request that reads 100 bytes sequentially from a regular data file on each execution of the `while` loop. A value of 0 returned by `read` indicates an EOF condition has been reached.

```
int fd, cnt;
char buf[100];

if ((fd = open("datafile", O_RDONLY)) == -1) {
     perror("Opening file datafile failed");
     exit(1);
}

while ((cnt = read(fd, buf, 100)) != 0) { /* read returning 0 means EOF */
     /* process data (cnt bytes) in buf here */
}

printf("EOF reached on file datafile.\n");
```

Example 2: This example shows how to use a `read` system call with an open pipe. Since the pipe is opened with the `O_NONBLOCK` flag set, `read` requests are not delayed when no data is available in the pipe to be read. (Typically, an empty pipe causes a `read` request to delay (block) until data arrives in the pipe.)

A value of 0 returned by `read` indicates an EOF condition has been reached; while a value of −1 returned means that no data is currently residing in the pipe to read.

```
int pfd, cnt, nbyte;
char pdata[256];

if ((pfd = open("named_pipe", O_RDONLY | O_NONBLOCK)) == -1) {
     perror("Opening named pipe named_pipe failed");
     exit(1);
}

if ((cnt = read(pfd, pdata, nbyte)) > 0) {
    /* process data (cnt bytes) from pipe in pdata here */
}
else {
     if (cnt == 0) {         /* read returning 0 means EOF */
          printf("EOF reached on pipe named_pipe.\n");
     }
     else {                  /* read returning -1 means no data now */
          /* no data currently available in pipe named_pipe -
             perform some other work and try again later */
     }
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `read` system call

SEE ALSO

`accept`(2), `chmod`(2), `creat`(2), `dup`(2), `fcntl`(2), `ioctl`(2), `kill`(2), `lseek`(2), `open`(2), `pipe`(2), `signal`(2), `socket`(2), `socketpair`(2)

`termio`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

reada – Performs asynchronous read from a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/iosw.h>
#include <signal.h>

int reada (int fildes, char *buf, unsigned nbyte, struct iosw *status,
int signo);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The reada system call tries to read a specified number of bytes from a file into a specified buffer (see the read(2) man page). The system call returns directly, even when the data cannot be delivered until later.

The first three arguments of the reada system call are the same as the read(2) system call. The last two arguments enable you to notify the process when the request has completed.

The reada system call accepts the following arguments:

fildes Specifies a file descriptor. It is obtained from an accept(2), creat(2), dup(2), fcntl(2), open(2), pipe(2), socket(2), or socketpair(2) system call.

buf Points to the buffer into which the data is to be read.

nbyte Specifies the number of bytes to be read.

status Points to a iosw structure. This structure is defined in the usr/include/sys/iosw.h file. The *status* word has the following structure:

```
struct iosw   {
    uint   sw_flag     :1,
           sw_error    :31,
           sw_count    :32;
};
```

signo Specifies the signal that should be sent to indicate that the I/O transfer is complete. For a list of signals, see the signal(2) man page.

When a request completes, the status word is filled in, and if signo was nonzero, that signal is sent to the process. The sw_flag is always set on completion, sw_error may contain a system call error number (see the intro(2) man page), and sw_count contains the number of bytes actually moved. If a process issues reada on a slow device, such as a tty, and must be moved in memory to satisfy a brk(2) request, the reada will fail with EINTR.

When an attempt to read from a regular file with mandatory file and record locking set is made (see the chmod(2) man page), and a blocking write lock exists on the segment of the file to be read (that is, it is owned by another process), the following occurs:

- If either O_NDELAY or O_NONBLOCK is set, the read returns a −1, and sets errno to EAGAIN.
- If O_NDELAY and O_NONBLOCK are both clear, the read sleeps until the blocking record lock is removed.

There is a limit to the number of outstanding asynchronous I/O requests that a process may have active. If a process exceeds this limit, it is not rescheduled until one or more of the requests have completed.

The file position for reading or writing is always the file position at the time of the reada or writea(2) system call. The file's position is incremented at that time by *nbyte* bytes. In this way, reada, writea(2), and lseek(2) system calls can be interspersed, and the file position is incremented naturally.

To use asynchronous I/O effectively, several rules must be followed:

- All outstanding I/O requests must have their own status words.
- One or more signal numbers may be used for I/O completions, but each signal must have its own handling routine. Several outstanding requests may share a signal handling routine.
- When an I/O completion handler is entered, the status words under its control should be scanned for completed I/Os.
- As the status words are processed, they must be set to 0.
- At the end of I/O completion handling, the status words are rescanned for newly completed I/Os. If any are found, the signal handler loops back and processes the new completions; otherwise, the handler returns.

NOTES

The process must be granted read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_MAC_READ	The process is granted read permission to the file via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted read permission to the file via the security label.

RETURN VALUES

If reada completes successfully, a nonnegative integer is returned, indicating the number of bytes remaining to be read; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The reada system call fails if one of the following error conditions occurs:

Error Code	Description
EAGAIN	Mandatory file and record locking was set, O_NDELAY was set, and there was a blocking record lock.
EBADF	The *fildes* argument is not a valid file descriptor open for reading.
EBADF	The active security label of the process is not greater than or equal to the security label of the file, and the process does not have appropriate privilege.
EDEADLK	The read was going to go to sleep and cause a deadlock situation to occur.
EFAULT	The *buf* or *status* argument is not fully contained within the process address space.
EINTR	The process caught a signal during the reada system call.
EINVAL	The *signo* argument is not a valid signal number and not 0.
ENOLCK	The system record lock table was full, so the read could not go to sleep until the blocking record lock was removed.

FORTRAN EXTENSIONS

The reada system call can be called from Fortran as a function:

```
INTEGER fildes, buf(n), nbyte, istat, signo, READA, I
I = READA (fildes, buf, nbyte, istat, signo)
```

EXAMPLES

The following examples illustrate different ways of using the reada system call so that a read operation completes in parallel with other work in a user's process. Simpler solutions appear in the last two examples, which make use of additional calls.

Example 1: In this program, the reada request specifies delivery of a SIGUSR1 signal on completion of the request.

The program uses the pause(2) system call to wait for the completion of the asynchronous read operation (that is, reception of the SIGUSR1 signal). The sigoff library routine provides assurance that the SIGUSR1 signal is not received before reaching the pause(2) request.

```
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/iosw.h>

struct iosw rdstat;

main()
{
     char buf[1000];
     int fd;
     void rdhdlr(int signo);

     signal(SIGUSR1, rdhdlr);

     if ((fd = open("datafile", O_RDONLY)) == -1) {
          perror("open (datafile) failed");
          exit(1);
     }

     sigoff();      /* delay signal reception until pause() is reached */
     reada(fd, buf, 1000, &rdstat, SIGUSR1);  /* SIGUSR1 sent when
                                                 read completes */

     /* perform other work here in parallel with I/O completion */

     pause();       /* wait for read to complete - pause() calls sigon() */

     /* input data from reada now available in buffer buf */
}

void rdhdlr(int signo)
{
     signal(signo, rdhdlr);
     printf("reada read %d bytes\n", rdstat.sw_count);
     rdstat.sw_flag = 0;
}
```

Example 2: (Some system calls in the example are not supported on Cray MPP systems.) Unlike the program in example 1, this program uses the recalla(2) system call to wait for completion of the asynchronous input operation. The user's program is informed of the completion by reception of the SIGUSR1 signal. While recalla(2) can wait for completion of multiple asynchronous I/O requests from multiple files, it only waits for one read operation in this example.

```
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/iosw.h>
#include <sys/param.h>

struct iosw rdstat;

main()
{
     char buf[1000];
     int fd;
     long mask[RECALL_SIZEOF];
     void rdhdlr(int signo);

     signal(SIGUSR1, rdhdlr);

     if ((fd = open("datafile", O_RDONLY)) == -1) {
          perror("open (datafile) failed");
          exit(1);
     }

     RECALL_SET(mask, fd);                    /* set bit for fd in mask */

     reada(fd, buf, 1000, &rdstat, SIGUSR1);  /* SIGUSR1 sent when
                                                 read completes */

     /* perform other work here in parallel with I/O completion */

     recalla(mask);                           /* wait for read to complete */

     /* input data from reada now available in buffer buf */
}

void rdhdlr(int signo)
{
     signal(signo, rdhdlr);
     printf("reada read %d bytes\n", rdstat.sw_count);
     rdstat.sw_flag = 0;
}
```

Example 3: Unlike the programs in examples 1 and 2, this program does not have an I/O completion signal specified on the `reada` request. The program uses the `recall`(2) system call to wait for completion of the asynchronous read operation. While `recall`(2) can wait for completion of multiple asynchronous I/O requests from multiple files or even the same file, it only waits for one read operation in this example.

```
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/iosw.h>

main()
{
     char buf[1000];
     int fd;
     struct iosw rdstat[1], *statlist[1];

     if ((fd = open("datafile", O_RDONLY)) == -1) {
          perror("open (datafile) failed");
          exit(1);
     }

     reada(fd, buf, 1000, &rdstat[0], 0); /* no signal sent when
                                             read completes */
     statlist[0] = &rdstat[0];

     /* perform other work here in parallel with I/O completion */

     recall(fd, 1, statlist);              /* wait for read to complete */

     printf("reada read %d bytes\n", rdstat[0].sw_count);
     rdstat[0].sw_flag = 0;

     /* input data from reada now available in buffer buf */
}
```

SEE ALSO

`brk`(2), `chmod`(2), `intro`(2), `lseek`(2), `pause`(2), `read`(2), `recalla`(2), `recall`(2), `writea`(2)

`sigoff`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

`readlink` – Reads value of a symbolic link

SYNOPSIS

```
#include <unistd.h>
int readlink (char *path, char *buf, int bufsiz);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `readlink` system call places the contents of the symbolic link in a buffer of specified size. The contents of the link are not null terminated when returned.

The `readlink` system call accepts the following arguments:

path Specifies the contents of the symbolic link. That is, the path name of the file being referred.

buf Points to the buffer that contains the symbolic link.

bufsiz Specifies the buffer size in characters.

NOTES

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

The process must have read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to the component via the permission bits and access control list.
`PRIV_MAC_READ`	The calling process is granted read permission to the file via the security label.
`PRIV_MAC_READ`	The process is granted search permission to the component via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix. If the `PRIV_SU` configuration option is enabled, the super user is granted read permission to the file via the security label.

RETURN VALUES

If `readlink` completes successfully, the count of characters placed in the buffer is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `readlink` system call fails and the buffer is unchanged if one of the following error conditions occurs:

Error Code	Description
EACCES	Search permission is denied for a component of the path prefix of *path*.
EACCES	The process is not granted read permission to the file via the security label and does not have appropriate privilege.
EFAULT	The *path* or *buf* argument extends outside the process allocated address space.
EINVAL	The specified file is not a symbolic link.
EINVAL	The *path* argument contained a byte with the high-order bit set.
EIO	An I/O error occurred during a read from or write to the file system.
EMLINK	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.
ENOENT	The specified file does not exist.

EXAMPLES

This example shows how to use the `readlink` system call to retrieve the path name of a file for which a symbolic link is targeting.

First, a `symlink`(2) system call is used to create a symbolic link. (Two arguments must be supplied to this program; the first is the path name of an existing file, which is the target of the symbolic link, and the second is the new link.) The `readlink` request then returns the path name of the target of the new link (that is, the value of the first argument, `argv[1]`).

```
#include <unistd.h>

main(int argc, char *argv[])
{
     char buf[1024];

     if (symlink(argv[1], argv[2]) == -1) { /* argv[1] -> existing file */
          perror("symlink failed");         /* argv[2] -> new link      */
     }

     readlink(argv[2], buf, 1024);          /* buf should contain argv[1]
                                               string */

     printf("Symbolic link contains => %s\n", buf);
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `readlink` system call

SEE ALSO

`lstat`(2), `stat`(2), `symlink`(2)

NAME

recall, recalls – Waits for I/O completions

SYNOPSIS

```
#include <sys/types.h>
#include <sys/iosw.h>
int recall (int fildes, int cnt, struct iosw **list);
int recalls (int cnt, struct iosw **list);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The recall system call provides a means to wait for any of a specified set of asynchronous I/O requests to complete. Each element in *list*, if it is not null, points to an I/O completion status word. When the completion bit is set in any of the specified status words, the system call returns. Any null entries in *list* are ignored. The arguments are as follows:

fildes Specifies a file descriptor. It is obtained from a creat(2), dup(2), fcntl(2), open(2), or pipe(2) system call or socket descriptor obtained from a call to the socket(2) system call.

cnt Specifies the number of elements in *list*.

list Points to an array of pointers to asynchronous I/O status structures of type struct iosw.

When calling recall, all status structures in *list* must correspond to I/O requests made for file descriptor *fildes*. The recalls system call has the same effect as recall, but the restriction that all status structures in *list* correspond to I/O requests for just one file descriptor is relaxed. Users are encouraged to use recall because the *fildes* argument permits the procstat(1) command to gather more complete statistics.

RETURN VALUES

If recall or recalls completes successfully, the value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The recall or recalls system call fails if one of the following error conditions occurs:

Error Code	Description
EFAULT	The request *list* is not fully contained within the process address space.
EINTR	A signal was caught during a wait for an I/O completion.

EINVAL The *cnt* argument is not a valid size. Implementation-defined limits exist on the maximum size of this list.

EXAMPLES

The following example shows how to use the `recall` system call to wait for completion of an asynchronous read operation so that the operation is performed in parallel with other work in a user's process.

In this program, the `reada`(2) request does not include an I/O completion signal. The `recall` request waits for the read operation to complete. Although `recall` can wait for completion of multiple asynchronous I/O requests from multiple files or even the same file, it waits for only one read operation in this example.

```
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/iosw.h>

main()
{
     char buf[1000];
     int fildes;
     struct iosw rdstat[1], *statlist[1];

     if ((fildes = open("datafile", O_RDONLY)) == -1) {
          perror("open (datafile) failed");
          exit(1);
     }

     reada(fildes, buf, 1000, &rdstat[0], 0); /* no signal sent when
                                                 read completes */
     statlist[0] = &rdstat[0];

     /* perform other work here in parallel with I/O completion */

     recall(fildes, 1, statlist);         /* wait for read to complete */

     printf("reada read %d bytes\n", rdstat[0].sw_count);
     rdstat[0].sw_flag = 0;

     /* input data from reada now available in buffer buf */
}
```

SEE ALSO

`creat`(2), `dup`(2), `fcntl`(2), `listio`(2), `open`(2), `pipe`(2), `reada`(2), `recalla`(2), `socket`(2), `writea`(2)

`procstat`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

recalla – Waits for I/O completion(s)

SYNOPSIS

```
#include <sys/param.h>
#include <sys/types.h>
#include <sys/iosw.h>

int recalla (long mask[RECALL_SIZEOF]);

RECALL_INIT (mask);
RECALL_CLR (mask, fd);
RECALL_SET (mask, fd);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The recalla system call waits for the completion of one or more I/O requests on files specified by *mask* that were previously initiated by a reada(2) or writea(2) system call.

It accepts the following argument:

mask Specifies a left-justified bit array in which each bit corresponds to a file descriptor. The bit array is an array of size RECALL_SIZEOF where each array element is of type long.

If any of the files to which *mask* points are not busy, control is returned immediately. If all of the files to which *mask* points are busy, the process is blocked until at least one of the file's I/O requests completes.

The caller must check the status word associated with the files in the mask to ensure completion.

The following macros are defined in the sys/iosw.h file. In these macros, *mask* specifies the *mask* argument used in the recalla call, and *fd* specifies a file descriptor.

RECALL_INIT (*mask*) Clears all bits in *mask*.

RECALL_CLR (*mask, fd*) Clears bit corresponding to *fd* in *mask*.

RECALL_SET (*mask, fd*) Sets bit corresponding to *fd* in *mask*.

RETURN VALUES

If recalla completes successfully, the value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The `recalla` system call fails if one of the following error conditions occurs:

Error Code	Description
EFAULT	The request *mask* is not fully contained within the process address space.
EINTR	A signal was caught during a wait for an I/O completion.

EXAMPLES

The following example shows how to use the `recalla` system call to wait for completion of an asynchronous read operation so that the operation is performed in parallel with other work in a user's process.

In this program, the `reada`(2) request specifies delivery of a `SIGUSR1` signal on completion of the request. The `recalla` system call waits for the completion of the read operation. The user's program is informed of the completion by the reception of the `SIGUSR1` signal. While `recalla` can wait for completion of multiple asynchronous I/O requests from multiple files, it only waits for one read request in this example.

```
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/iosw.h>
#include <sys/param.h>

struct iosw rdstat;

main()
{
     char buf[1000];
     int fd;
     long mask[RECALL_SIZEOF];
     void rdhdlr(int signo);

     signal(SIGUSR1, rdhdlr);

     if ((fd = open("datafile", O_RDONLY)) == -1) {
          perror("open (datafile) failed");
          exit(1);
     }

     RECALL_SET(mask, fd);                    /* set bit for fd in mask */

     reada(fd, buf, 1000, &rdstat, SIGUSR1);  /* SIGUSR1 sent when
                                                 read completes */

     /* perform other work here in parallel with I/O completion */

     recalla(mask);                          /* wait for read to complete */
```

```
        /* input data from reada now available in buffer buf */
}

void rdhdlr(int signo)
{
        signal(signo, rdhdlr);
        printf("reada read %d bytes\n", rdstat.sw_count);
        rdstat.sw_flag = 0;
}
```

SEE ALSO

listio(2), open(2), reada(2), recall(2), writea(2)

NAME

recv, recvfrom, recvmsg – Receives a message from a socket

SYNOPSIS

All Cray Research systems:

```
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/socket.h>

int recv (int s, char *buf, int len, int flags);

int recvfrom (int s, char *buf, int len, int flags,
struct sockaddr *from, int *fromlen);
```

Cray PVP systems:

```
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/socket.h>

int recvmsg (int s, struct msghdr *buf, int flags);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The recv, recvfrom, and recvmsg system calls receive a message (*buf*) from a socket.

You can use a recv call only on a connected socket. You can use recvfrom or recvmsg on either a connected or unconnected socket. The recvmsg system call uses the same msghdr structure as the sendmsg(2) system call to minimize the number of directly supplied arguments. For more information, see the connect(2) and send(2) man pages.

The recv, recvfrom, and recvmsg system calls accept the following arguments:

s Specifies the descriptor of the socket from which messages are received. Descriptor is returned when a socket is created by the socket(2) or socketpair(2) system call; socketpair(2) uniquely identifies the socket's access path.

buf Points to the address of a buffer into which received messages are placed.

len Specifies the length of the buffer pointed to by the *buf* argument.

flags Allows the caller to control the reception of messages. The argument value is formed by performing an OR operation on one or more of the following values:

- MSG_OOB Process out-of-band data.
- MSG_PEEK Peek at incoming message without removing message from the socket.

MSG_DONTROUTE	Send without using routing tables.
MSG_EOR	Data sent completes record.
MSG_TRUNC	Data discarded before delivery.
MSG_CTRUNC	Control data lost before delivery.
MSG_WAITALL	Wait for full request or error.

The recvfrom and recvmsg system calls can be used to receive data on a socket whether or not it is in a connected state. The recvfrom system call accepts the following additional arguments allowing the caller to specify where the sender's address should be recorded:

from Specifies the address of a sockaddr structure that the operating system will use to record the address of the message sender.

fromlen Specifies the address of an integer that the operating system uses to record the length of the sender's address, which is recorded in *from*.

If no messages are available at the socket, the receive call waits for a message to arrive; however, if the socket is nonblocking (set by using an ioctl(2) system call with FIONBIO (see socket(2)), a value of −1 is returned, and the external variable errno is set to EWOULDBLOCK. Use the select(2) call to determine when data arrives.

The recvmsg system call uses the same msghdr structure which is defined in the sys/socket.h file. This structure has the following form:

```
struct msghdr {
        caddr_t    msg_name;            /* optional address */
        u_int      msg_namelen;         /* size of address */
        struct     iovec *msg_iov;      /* scatter/gather array */
        u_int      msg_iovlen;          /* # elements in msg_iov */
        caddr_t    msg_control;         /* ancillary data, see below */
        u_int      msg_controllen;      /* ancillary data buffer len */
        int        msg_flags;           /* flags on received message */
};
```

In this structure, msg_name and msg_namelen describe the source address for recvmsg or the destination address for sendmsg(2). If no names are desired or required, msg_name is given as a null pointer. The msg_name and msg_namelen arguments operate similarly to the recvfrom *from* and *fromlen* arguments, and the sendto(2) *to* and *tolen* arguments.

The msg_iov and msg_iovlen arguments describe an array of buffer descriptors. The msg_iov argument points to an array of structure iovec, which is defined as follows:

```
struct iovec   {
    caddr_t    iov_base;
    int        iov_len;
};
```

Each iovec entry specifies the base address and length of an area in memory from which data must be read or to which data must be written.

The *iov_len* argument specifies the number of structure iovec entries in the array to which msg_iov points. recvmsg and sendmsg(2) always process the entire iov_len bytes of one iovec structure before proceeding to the next.

The msg_control argument, which has length msg_controllen, is a buffer for other protocol control-related messages or other miscellaneous ancillary data. The messages are of the following form:

```
struct cmsghdr {
        u_int   cmsg_len;       /* data byte count, including hdr */
        int     cmsg_level;     /* originating protocol */
        int     cmsg_type;      /* protocol-specific type */
/* followed by
        u_char  cmsg_data[]; */
};
```

For recvmsg, the msg_controllen argument is a value-result parameter, which is initialized to the size of the msg_control argument, and it displays the number of bytes of control information returned.

Open file descriptors are now passed as ancillary data for AF_UNIX domain sockets (for example, cmsg_level is SOL_SOCKET and cmsg_type is SCM_RIGHTS). This feature is disabled on systems that are configured to support nonzero security labels.

The msg_flags argument is set on return using a method that includes some of the same values that are specified for the *flags* parameter to a recv system call. The returned value MSG_EOR indicates end-of-record, MSG_TRUNC indicates that some trailing datagram data was discarded, and MSG_CTRUNC indicates that some control data was discarded because of lack of space. MSG_OOB is returned to indicate that expedited data was received.

NOTES

If the SOCKET_MAC option is enabled, the active security label of the process must be greater than or equal to the security label of the socket. SOCKET_MAC is part of TCP/IP configurable feature variables list in uts/cf/Nmakefile. For more information, see the connect(2) man page.

A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_MAC_READ	When the SOCKET_MAC is enabled, the process may override the security label restrictions.

If the PRIV_SU configuration option is enabled, the super user may override security label restrictions when the SOCKET_MAC option is enabled.

If *s* is an Internet-domain socket and the `recvfrom` or `recvmsg` system call is used, the `sin_addr` and `sin_port` fields of the sending socket name identify only the socket at the other end of the connection, not the remote process or remote user. Additional knowledge is required to interpret those fields. For example, if the `sin_addr` field designates another UNICOS system, a `sin_port` value of less than 1024 indicates a connection with trusted software (for example, `rlogin`(1B)), which may include additional identity information in its protocol data stream. If it is necessary to identify the actual user associated with the socket, the communicating peers must agree in advance on a method, such as the sender placing its `sin_port` value in a protected file accessed through NFS (or other means) by the receiver.

Because no sender name information can be obtained from a UNIX-domain socket, the other end of the connection cannot be identified except to the extent that additional authentication techniques are used. Although no identity-based access controls restrict use of `connect`(2) or `sendto`(2) for a UNIX-domain socket, such a socket can be created in a directory to which execute (search) access is restricted. This limits the ability of other processes to connect to the socket. Alternatively, the listening process could place a random value or secret password in a protected file and require the value or password be included in all messages it accepts; this ensures that only users with access to that file can send valid messages. For both Internet-domain and UNIX-domain, this authentication requires explicit action on the part of the receiver.

RETURN VALUES

If `recv`, `recvmsg`, or `recvfrom` completes successfully, the number of characters received is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `recv`, `recvfrom`, or `recvmsg` system call fails if one of the following conditions occurs:

Error Code	Description
EACCES	Permission is denied (because of a security violation).
EACCES	If the `SOCKET_MAC` option is enabled, the process does not meet the security label requirements and does not have appropriate privilege.
EBADF	Descriptor *s* is not valid.
EFAULT	Data is specified to be received into a nonexistent or protected part of the process address space.
EINTR	Receive operation is interrupted by delivery of a signal before any data is available for the receive.
EMSGSIZE	The `msg_iovlen` field is greater than or equal to the `MSG_MAXIOVLEN` parameter (defined in the `sys/socket.h` file).
EINVAL	No out-of-band data is available when the `MSG_OOB` flag is specified.
ENOTCONN	Socket is not connected.
ENOTSOCK	Descriptor *s* is not a socket.

EWOULDBLOCK	Socket is marked nonblocking, and the receive operation would block.

FILES

/etc/config/spnet.conf	Network access list file
/usr/include/sys/socket.h	Contains definitions related to sockets, types, address families, and options
/usr/include/sys/types.h	Contains types required by ANSI X3J11
/usr/include/sys/uio.h	Contains user I/O structures and definitions

SEE ALSO

accept(2), connect(2), ioctl(2), select(2), send(2), socket(2), socketpair(2)

rlogin(1B) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

UNICOS File Formats and Special Files Reference Manual, Cray Research publication SR–2014

UNICOS Networking Facilities Administrator's Guide, Cray Research publication SG–2304

NAME

`rename` – Changes the name of a file

SYNOPSIS

```
#include <stdio.h>
int rename (const char *old, const char *new);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `rename` system call changes the name of a file.

The `rename` system call accepts the following arguments:

old Points to the path name of the file to be renamed.

new Points to the new path name of the file.

If the *old* argument and the *new* argument both refer to links to the same existing file, the `rename` system call returns successfully and performs no other action.

If the *old* argument points to the path name of a file that is not a directory, the *new* argument does not point to the path name of a directory. If the link specified by the *new* argument exists, it is removed and *old* is renamed *new*. In this case, a link named *new* exists throughout the renaming operation and refers either to the file referred to by *new* or *old* before the operation began. Write access permission is required for both the directory that contains *old* and the directory that contains *new*.

If the *old* argument points to the path name of a directory, the *new* argument points to the path name of a file that is a directory. If the directory specified by the *new* argument exists, it shall be removed and *old* renamed *new*. In this case, a link named *new* exists throughout the renaming operation and refers either to the file referred to by *new* or *old* before the operation began. If *new* specifies an existing directory, it must be an empty directory.

The *new* path name does not contain a path prefix that names *old*. Write access permission is required for the directory that contains *old* and the directory that contains *new*. If the *old* argument points to the path name of a directory, write access permission is required for the directory named by *old*, and, if it exists, the directory named by *new*.

If the link specified by the *new* argument exists and the file's link count becomes 0 when it is removed and no process has the file open, the space occupied by the file is freed and the file is no longer accessible. If one or more processes have the file open when the last link is removed, the link is removed before `rename` returns, but the removal of the file contents is postponed until all references to the file are closed.

Upon successful completion, `rename` marks for update the `st_ctime` and `st_mtime` fields of the parent directory of each file.

NOTES

Under UNICOS, `rename` is implemented as a system call, but the `rename`(3C) function is also defined to be a part of the ANSI Standard C library. For this reason, this documentation appears both here and in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080.

The process must be granted search permission to every component of each path prefix via the permission bits and access control list. The process must be granted search permission to every component of each path prefix via the security label.

The process must be granted write permission to each path's parent directory via the permission bits and access control list. The process must be granted write permission to each path's parent directory via the security label.

If *old* is a directory, and *new*'s parent directory is not the same as *old*'s parent directory, the process must be granted write permission to *old* via the permission bits and access control list. If *old* is a directory and *new*'s parent directory is not the same as *old*'s parent directory, the process must be granted write permission to *old* via the security label.

If *new* already exists, the process must be granted write permission to *new* via the permission bits and access control list. If *new* already exists, the process must be granted write permission to *new* via the security label.

If `FSETID_RESTRICT` is enabled, the set-user-ID and set-group-ID bits are cleared if the file is renamed across file systems, unless the process has appropriate privilege.

A process with the effective privileges shown are granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of each path prefix via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted write permission to each path's parent directory *old*, and *new* (if it already exists) via the permission bits and access control list.
`PRIV_FSETID`	If `FSETID_RESTRICT` is enabled, set-user-ID and set-group-ID bits are not cleared access file systems.
`PRIV_MAC_READ`	The process is granted search permission to every component of each path prefix via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to each path's parent directory *old*, and *new* (if it already exists) via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of each path prefix and is granted write permission to each path's parent directory. The super user is granted write permission to *old* and *new* (if it exists). The super user or a process with the suidgid permission can override the restriction enabled by the FSETID_RESTRICT system configuration option.

RETURN VALUES

If rename completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The rename system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A component of either path prefix denies search permission; or one of the directories containing *old* or *new* denies write permissions; or, write permission is required and is denied for a directory pointed to by the *old* or *new* argument.
EBUSY	The directory named by *old* or *new* cannot be renamed because it is being used by the system or another process and the implementation considers this to be an error.
EEXIST or ENOTEMPTY	The link named by *new* is a directory containing entries other than dot and dot-dot.
EINVAL	The *new* directory path name contains a path prefix that names the *old* directory.
EISDIR	The *new* argument points to a directory and the *old* argument points to a file that is not a directory.
ENAMETOOLONG	The length of the *old* or *new* argument exceeds {PATH_MAX}, or a path name component is longer than {NAME_MAX} when {_POSIX_NO_TRUNC} is in effect.
ENOENT	The link named by the *old* argument does not exist or either *old* or *new* points to an empty string.
ENOSPC	The directory that should contain *new* cannot be extended.
ENOTDIR	A component of either path prefix is not a directory; or the *old* argument names a directory and the *new* argument names a nondirectory file.
EROFS	The requested operation requires writing in a directory on a read-only file system.
EXDEV	The links named by *new* and *old* are on different file systems and the implementation does not support links between file systems.

FILES

`/usr/include/sys/stdlib.h`	Contains C prototype for the `rename` system call

SEE ALSO

`link`(2), `rmdir`(2), `unlink`(2)

`rename`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

resch – Reschedules a process

SYNOPSIS

```
#include <unistd.h>
void resch (void);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The resch system call causes a process to be rescheduled by logically placing it at the end of the queue of processes that can run.

If the process is a thread-process with its wakeup flag set to 0, it is suspended until the wakeup flag is set to a nonzero value. This is used by the microtasking library to activate and deactivate processes.

RETURN VALUES

None

FILES

/usr/include/unistd.h	Contains C prototype for the resch system call

SEE ALSO

fork(2), thread(2)

NAME

`restart` – Restarts a process, multitask group, or job

SYNOPSIS

```
#include <sys/restart.h>
int restart (char *path, long flags);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `restart` system call validates, loads, and restarts the process, multitask group, or job defined in the specified restart file as created by the `chkpnt`(2) system call.

It accepts the following arguments:

path Specifies path name of the restart file containing the process, multitask group, or job (or interactive session) to be recovered.

flags Identifies optional `restart` actions. The flags present in this field are OR'ed together to define the optional actions to be performed by `restart`. The optional actions are as follows:

`RESTART_FORCE` Forces recovery even when one or more of the files referred to by processes saved in the restart file have been changed.

`RESTART_PAG` Allows the restarted processes to inherit the Distributed Computing Environment (DCE) credentials from the process that calls `restart` rather than using the credentials stored in the restart file.

`RESTART_PTRACE` Restarts and traces the process (see `ptrace`(2)). If the `RESTART_PTRACE` flag is set, the restart file must describe a process or multitask group, not a job. If the restart is successful, the restarted process is recovered as though it had executed a `ptrace`(2) system call. When restarting a multitask group, it is as though the eldest process of the multitask group executed the `ptrace`(2) system call.

`RESTART_SUSPEND` Restarts all the recovered processes in a suspended state (see `suspend`(2) and `resume`(2)).

`RESTART_WAIT` Makes the restarted process, if it is interactive, the foreground process.

The eldest process of each multitask group receives a `SIGRECOVERY` signal on restoration (see `signal`(2)). By default, the `SIGRECOVERY` signal is ignored, but processes expecting to be checkpointed and restored successively can elect to catch this signal; thereby, they can perform any special actions needed for their proper recovery.

Processes with open pipes can be checkpointed and restarted if their pipe connections do not go outside the job or multitask group being checkpointed. For a process with open pipes to have been checkpointed, all of its pipe connections must have terminated with processes also included in the restart file.

Processes with open files that reside on NFS file systems can be checkpointed and restarted. To restart a process with open NFS files, the NFS file systems on which the files reside have to be mounted, unless the NFS file systems are managed by the automounter. In this case, the automounter will try to remount the file systems automatically.

Processes with open files that reside on Distributed File System (DFS) file systems can be checkpointed and restarted. The following conditions must exist in order for a user to restart a process with an open DFS file.

- The DFS client must be running on the local host.
- The DFS server must be running on the host where the file resides.

Access to DFS files is controlled by a user's DCE credentials as opposed to user identification (UID) and group identification (GID). DFS credentials consist of Kerberos tickets stored in a special file. When a process is checkpointed, a reference to these credentials is stored in the restart file. The credentials must be present and valid when `restart` is performed. If the credentials are no longer present or have expired, accesses to DFS files that are performed after the `restart` call will appear to be from the UID –2.

The `RESTART_PAG` flag allows the reference to the original credentials to be replaced by a reference to a new set of credentials. Using this flag allows processes that access DFS files to be restarted after their original credentials have been deleted or have expired.

Processes with unlinked files can be checkpointed and restarted if the total of the size of all unlinked files in use by the target process set is within the size limit established by the system administrator. See the system variable `MAX_UNLINKED_BYTES` in the `/usr/src/uts/c1/cf/config.h` file to see the site local definition.

On systems with SSD solid-state storage devices, processes that are using secondary data segments (SDS) can be checkpointed and restarted if sufficient disk space is available and can contain an image of the process' SDS area within the restart file. An `ENOSDS` error may occur at restart time if the SDS area available at that time is less than what was in use at checkpoint time. The `ENOSDS` error means that `restart` must be retried at a later time when sufficient SDS space is available.

Processes using online tape files cannot be checkpointed or restarted.

NOTES

The following restrictions apply to processes and jobs (including interactive sessions) that are to be restarted:

- All files that a process was using when it was checkpointed must be present when the process is restarted. These files include all open files, any shared-text executables that the process was using (such as shells), and the present working directory. In the restart file, each of these files is identified by its inode number and the minor number of the file system. If either changes, the `restart` system call fails, and the call returns an `EFILERM` error. For example, if a file system is restored by `/etc/restore`, any process that was using files on that file system and was checkpointed before the restore, will fail to restart. After the restore, each file on the file system has a different inode number than it did when the process was checkpointed.

- Only a process with appropriate privilege may checkpoint or restart another user's job or process.
- Processes using online tapes cannot be checkpointed or restarted.
- Processes using shared memory segments (CRAY T90 series systems only) cannot be checkpointed or restarted.
- Whenever any process is recovered from a restart file, all its multitask sibling processes are also recovered. Thus, when `restart` is invoked to perform recovery from a process restart file (a restart file that does not define an entire job), it is still possible for several processes to be recovered, because all the multitask siblings of the original target process must also be restored.
- All processes recovered by `restart` retain their original attributes, such as process ID (PID), parent process ID (PPID), process group ID (PGRP), and job ID (JID). The only possible exceptions to this rule concern the process attributes of PGRP and PPID of the oldest restarted process.
- The exception to the *pgrp* conservation rule occurs only when one multitask group is recovered from a process restart file. If the *pgrp* of the recovered multitask group is found to be nonzero and not equal to the *pid* of a process in the group, the *pgrp* of the recovered group is set to the *pgrp* of the caller.
- If a user attempts to copy a restart file, the `restart` system call fails.
- If a user attempts to move a restart file to a different file system, the `restart` system call fails.
- If an interactive session is checkpointed and later recovered with a `restart` system call, each process that is part of the session that performed the `restart` system call is sent a `SIGHUP` signal to indicate that the connection hung up. The call assigns the pseudo tty of the calling session to the restarted session.

A process with the effective privilege shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
`PRIV_FOWNER`	The process is considered the owner of the specified file.
`PRIV_MAC_READ`	The process is granted search permission to every component of the path prefix via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the restart file via the security label.
`PRIV_POWNER`	The calling process is considered the owner of the session being restarted.

If the `PRIV_SU` configuration option is enabled, the super user is considered the owner of the restart file and is granted access to the restart file. The super user is considered the owner of the session being restarted.

RETURN VALUES

If `restart` completes successfully, the PID (JID) of the recovered process (job) is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `restart` system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A component of the restart file path prefix denies search permission.
EACCES	The restart file is not owned by the caller, and the caller does not have appropriate privilege.
EACCES	The caller's active security label did not dominate that of the restart file.
EAGAIN	The system-imposed limit on the total number of processes in the system (NPROC) would be exceeded by the recovery of all processes from the restart file.
EAGAIN	The system-imposed limit on the total number of processes in the system allocated to one user (CHILD_MAX) would be exceeded by the recovery of all processes from the restart file.
EAGAIN	The restart file contains recovery information for an entire job, and the maximum number of jobs allowed to exist in the system (NJOB) at any one time already exist.
EAGAIN	The multitask group or process defined in the restart file could not be recovered, because a job I/O quota would be exceeded.
EBUSY	One or more of the processes to be recovered from the restart file has a process ID that is already allocated to an existing process in the system.
EBUSY	The restart file contains recovery information for an entire job, and the job ID of the job to be recovered is already allocated to an existing job in the system.
EDEADLK	The reapplication of record lock(s) owned by the process(es) to be restarted would result in a deadlock situation.
EFAULT	The *path* argument points outside the allocated process address space.
EFILECH	One or more files referenced in the restart file have been changed, and the RESTART_FORCE flag was not set.
EFILECH	One or more files referenced in the restart file have changed either user or group ownership. This situation cannot be overridden by the RESTART_FORCE flag.
EFILERM	One or more files referenced in the restart file are no longer present.
EFILERM	One or more files referenced in the restart file reside on an NFS file system that is not mounted.
EFILERM	A DFS file cannot be located. Either the file has been removed, or the DFS server holding it is either down or unreachable.
EINVAL	The `restart` system call was invoked with the RESTART_PTRACE flag, and the restart file described a job rather than a process or multitask group.
ENFILE	The system inode or file table is full.

ENODEV	The DFS client is currently not running and needs to be started before `restart` can proceed.
ENOENT	The specified restart file does not exist.
ENOEXEC	The restart file path name does not refer to a valid restart file.
ENOLCK	One or more of the processes to be restored owned record locks at checkpoint time (see `fcntl`(2) and `lockf`(3C)), and not enough record locks are available to complete recovery.
ENOMEM	There is not enough main memory or swap space to complete the recovery.
ENOSDS	Insufficient SDS space is available to complete the recovery.
ENOSPC	Insufficient free file space exists to re-create unnamed pipes previously in use by one or more of the processes to be recovered.
ENOSPC	Insufficient free file space exists to re-create unlinked regular files previously in use by one or more of the processes to be recovered.
ENOTDIR	A component of the restart file path prefix is not a directory.
ENOTTY	One or more of the processes to be recovered had a controlling tty, and the caller has no controlling tty.
EQACT	A file or inode quota limit was reached for the current account ID.
EQGRP	A file or inode quota limit was reached for the current group ID.
EQUSR	A file or inode quota limit was reached for the current user ID.
ERFLOCK	Record lock(s) owned by the process(es) to be restarted could not be reapplied because record lock(s) owned by currently existing process(es) have one or more of the target file regions already locked.

EXAMPLES

The following example shows how to use the `restart` system call to recover a checkpointed process or job. The path name of the checkpoint/restart file, `argv[1]`, is supplied as the only argument to this program.

(This system call is not used frequently by users because the `restart`(1) command provides similar functionality.)

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

     if ((id = restart(argv[1], 0)) == -1) {
          perror("restart failed");
          exit(1);
     }

     printf("pid (jid) of recovered process (job) = %d\n\n", id);
     system("ps");  /* view recovered processes in ps(1) display */
}
```

FILES

`/usr/include/sys/restart.h`	Contains the optional `restart` actions
`/usr/src/uts/c1/cf/config.h`	Contains the system variable `MAX_UNLINKED_BYTES`

SEE ALSO

`chkpnt`(2), `chmod`(2), `chown`(2), `creat`(2), `fcntl`(2), `open`(2), `ptrace`(2), `resume`(2), `signal`(2), `suspend`(2), `_tfork`(2)

`chkpnt_util`(1), `restart`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`lockf`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

`rmdir` – Removes a directory

SYNOPSIS

```
#include <unistd.h>
int rmdir (const char *path);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `rmdir` system call removes the directory specified by a path name. The directory must not have any entries other than "." and "..".

The `rmdir` system call accepts the following argument:

path Points to the path name of the directory.

To remove a directory that has the "sticky" bit set, the process must be the owner of that directory.

NOTES

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

To be granted write permission to the parent directory, the active security label of the process must equal the security label of the parent directory.

The process must be granted write permission to the directory being removed via the security label. That is, the active security label of the process must equal the security label of the directory being removed.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted write permission to the parent directory via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
`PRIV_FOWNER`	The process is considered the owner of a directory that has the "sticky" bit set.
`PRIV_MAC_READ`	The calling process is granted search permission to every component of the path prefix via the security label.

PRIV_MAC_WRITE	The process is granted write permission to the parent directory via the security label.
PRIV_MAC_WRITE	The process is granted write permission to the directory being removed via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix. If the PRIV_SU configuration option is enabled, the super user is granted write permission to the parent directory and to the directory being removed. The super user is considered the owner of a directory that has the "sticky" bit set.

RETURN VALUES

If rmdir completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The specified directory is removed unless one of the following error conditions occurs:

Error Code	Description
EACCES	Search permission is denied for a component of the path prefix.
EACCES	Write permission is denied on the directory containing the directory to be removed.
EACCES	The process is not granted write permission to the directory being removed via the security label, and the process does not have appropriate privilege.
EACCES	Directory label does not dominate the active security label of the process.
EACCES	Parent directory label does not equal the active security label of the process.
EBUSY	The directory to be removed is the mount point for a mounted file system.
EEXIST	The directory contains entries other than those for "." and "..".
EFAULT	The *path* argument points outside the process' allocated address space.
EINVAL	The current directory may not be removed.
EINVAL	The "." entry of a directory may not be removed.
EIO	An I/O error occurred during the access of the file system.
EMLINK	The directory has been linked. Use unlink(2) to remove the directory.
ENOENT	The specified directory does not exist.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The directory has the "sticky" bit set and the process is not the owner.
EROFS	The directory entry to be removed is part of a read-only file system.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `rmdir` system call

SEE ALSO

`mkdir`(2), `unlink`(2)

`mkdir`(1), `rm`(1), `rmdir`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`rmfacl` – Removes an access control list from a file

SYNOPSIS

```
#include <sys/acl.h>
int rmfacl (char *fname);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `rmfacl` system call removes the access control list (ACL) from a file. A `rmfacl` request is allowed only for a process with an active `secadm` category, a process executing on behalf of the file owner, or a process with appropriate privilege. If the process is not a member of the owning group of the file, the set-group-ID mode bit of the file is cleared unless the process has appropriate privilege. If the `FSETID_RESTRICT` system configuration parameter is enabled, the set-user-ID and set-group-ID mode bits of a file are cleared unless the process has appropriate privilege.

The `rmfacl` system call accepts the following argument:

fname Specifies the file from which the ACL is removed.

NOTES

Errors are recorded in the security log if discretionary access logging is enabled.

The functionality provided by this system call is also provided by the `setfacl`(2) system call.

The process must have write permission to the file via the security label. That is, the active security label of the process must equal the security label of the file.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to the component via the permission bits and ACL.
`PRIV_FOWNER`	The process is considered the file's owner.
`PRIV_FSETID`	The process is allowed to set an ACL on a file whose mode includes the set-user-ID or set-group-ID mode bit.
`PRIV_MAC_READ`	The process is granted search permission to the component via the security label.

PRIV_MAC_WRITE The process is granted write permission to a component of the path prefix via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted write permission to the file via the security label. The super user is considered the file owner and is allowed to set an ACL on a file whose mode includes the set-user-ID or set-group-ID mode bit.

RETURN VALUES

If rmfacl completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The rmfacl system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A component of the path prefix denies search permission.
EFAULT	The *fname* argument points outside the process address space.
EMANDV	The process does not have write permission to the file via the security label and does not have appropriate privilege.
ENAMETOOLONG	The supplied file name is too long.
ENOACL	The specified file does not have an ACL or its ACL is corrupted.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.
EOWNV	The process is not the file owner and does not have appropriate privilege.

FILES

/usr/include/sys/acl.h Contains C prototype for the rmfacl system call

SEE ALSO

getfacl(2), setfacl(2)

spacl(1), spclr(1), spset(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

acl(5), slog(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

schedv – Sets memory scheduling parameters

SYNOPSIS

```
#include <sys/schedv.h>
int schedv (int svar, struct schedvar *svartab);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The schedv system call gets and sets the system memory scheduling (schedvar) structure. It accepts the following arguments:

svar Specifies the command type. *svar* can be one of the following:

SVAR_GET Transfers system schedvar table to the specified *svartab* address.

SVAR_SET Transfers the schedvar table specified by *svartab* to the system schedvar table. The caller must fill in the sv_magic and sv_size fields in the schedvar structure with the SV_MAGIC constant and the size of the schedvar structure. Only a process with appropriate privilege can specify this command type.

svartab Points to the schedvar structure.

The schedvar structure includes the following members:

```
int     sv_memhog;         /* Size of a "big" process in clicks     */
time_t  sv_cpuhog;         /* Utime ticks used by CPU-bound proc.   */
int     sv_hog_max_mem;    /* Max clicks allotted to "hog" procs    */
float   sv_fit_boost;      /* Best fit boost given to in-core proc  */
                           /* if bigger than proc coming in         */
int     sv_thrash_inter;   /* Thrash interval in seconds            */
int     sv_thrash_blks;    /* Thrash blocks per interval            */
float   sv_mfactor_in;     /* Memory size factor - loaded procs     */
float   sv_mfactor_out;    /* Memory size factor - swapped procs    */
float   sv_tfactor_in;     /* Time factor - loaded procs            */
float   sv_tfactor_out;    /* Time factor - swapped procs           */
                           /* tfactor's are multiplied against time */
                           /* of residence.                         */
float   sv_pfactor_in;     /* Priority factor - loaded procs        */
float   sv_pfactor_out;    /* Priority factor - swapped procs       */
float   sv_nfactor_in;     /* Nice factor - loaded procs            */
float   sv_nfactor_out;    /* Nice factor - swapped procs           */
int     sv_max_outage;     /* Maximum time in seconds for which a   */
                           /* swapped process will be passed over   */
```

```
                                /* during memory-tight situations.       */
int     sv_flags;               /* Behavior modification flags           */
                                /* Used for things such as interactive   */
                                /* preferred, etc. See defines in        */
                          /* schedv.h for list of flags           */
time_t  sv_packtime;            /* The time, in clocks, between attempts */
                                /* slide processes down to pack memory.  */
float   sv_kfactor_in;          /* 0'th polynomial term - incore procs   */
float   sv_kfactor_out;         /* 0'th polynomial term - swapped procs  */
float   sv_gfactor0_in;         /* Guaranteed residence factor 0         */
                                /*                   - loaded procs      */
float   sv_gfactor0_out;        /*                   - swapped procs     */
float   sv_gfactor1_in;         /* Guaranteed residence factor 1         */
                                /*                   - loaded procs      */
float   sv_gfactor1_out;        /*                   - swapped procs     */
float   sv_ufactor_in;          /* University of Texas priority factor   */
float   sv_ufactor_out;         /* For interactive processes -           */
                                /*   Time since last interaction         */
                                /* For non-interactive processes -       */
                                /*   Time remaining                      */
                                /*     (cpu time limit - cpu time used)  */
int     sv_cpufactor;           /* # of running processes in-core to try */
                                /* for. Default is 8 + (2 * ncpu)        */
int     sv_bigproc;             /* If non-zero, processes above this     */
                                /* size in clicks won't be swapped unless*/
                                /* they're expanding or suspended.       */
word    sv_magic;               /* Magic number to indicate valid struct */
int     sv_size;                /* Size of schedvar structure.  This is  */
                                /* Used by the kernel to verify that the */
                                /* calling nschedv is in sync with the   */
                                /* kernel.                               */
int     sv_maxruns;             /* Maximum # of sched runs per second.   */
int     sv_smallproc;           /* Small interactive process click size  */
int     sv_itime;               /* Interaction time                      */
```

Values of 0 disable all scheduling variables except for sv_max_outage.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_ADMIN	The process is allowed to use the SVAR_SET command.

If the PRIV_SU configuration option is enabled, the super user or a process with the PERMBITS_SYSPARAM permbit is allowed to use the SVAR_SET command.

RETURN VALUES

If schedv completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The schedv system call fails if one of the following error conditions occurs:

Error Code	Description
EFAULT	The user field length did not contain the schedvar structure to which *svartab* points.
EINVAL	A value in either the sv_magic or sv_size field did not match the value expected by the kernel.
EPERM	The SVAR_SET command was used and the process did not have appropriate privilege.

SEE ALSO

limit(2)

limit(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

nschedv(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

secstat, lsecstat, fsecstat – Gets file security attributes

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secstat.h>
int secstat (char *path, struct secstat *buf);
int lsecstat (char *path, struct secstat *buf);
int fsecstat (int fildes, struct secstat *buf);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The secstat system call obtains the security attributes of a file; fsecstat system call obtains the same information for an open file. The lsecstat system call is similar to secstat except when the specified file is referenced by the link. In this case, lsecstat returns information about the link, and secstat returns information about the file referenced by the link.

The secstat and lsecstat system calls accept the following arguments:

path Specifies the file from which the security attributes are obtained.

buf Points to a secstat structure in which the information is returned.

The fsecstat system call accepts the following arguments:

fildes Specifies the file descriptor that identifies the file from which the security attributes are obtained.

buf Points to a secstat structure in which the information is returned.

A secstat structure includes the following members:

```
int      st_slevel;        /* File security level */
long     st_compart;       /* File compartments */
long     st_acldsk;        /* Access control disk address */
int      st_secflg;        /* Security flag */
int      st_intcls;        /* Class (not used) */
int      st_intcat;        /* Categories (not used) */
int      st_minlvl;        /* Device minimum security level */
int      st_maxlvl;        /* Device maximum security level */
long     st_valcmp;        /* Device authorized compartments */
```

NOTES

Any process may obtain the security attributes of a file labeled with a wildcard security label.

On a nondevice file, the `st_minlvl`, `st_maxlvl`, and `st_valcmp` fields of the `secstat` structure are equal to the minimum security level, maximum security level, and valid compartments, respectively, of the file system on which the file resides.

Only a process with appropriate privilege can retrieve the actual state of the file trap flags (`trapr` and `trapw`) in the `st_secflg` field.

The `secstat`, `lsecstat`, and `fsecstat` requests are not recorded in the security log.

The process must have read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component. (`secstat`/`lsecstat` systems call only.)

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_ADMIN`	The process is allowed to retrieve the trap state of the file.
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path prefix via the permission bits and access control list. (`secstat`/`lsecstat` system calls only.)
`PRIV_MAC_READ`	The process is granted search permission to a component of the path prefix via the security label. (`secstat`/`lsecstat` system calls only.)
`PRIV_MAC_READ`	The process is granted read permission to the file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix (`secstat`/`lsecstat` system calls only) and is granted read permission to the file via the security label. The super user is allowed to retrieve the trap state of the file.

RETURN VALUES

If `secstat`, `lsecstat`, or `fsecstat` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `secstat` or `lsecstat` system call fails if one of the following error conditions occurs:

Error Code	Description
`EACCES`	A component of the path prefix denies search permission.
`EACCES`	The *buf* argument points outside the process address space.

EACCES	The process is not granted read permission to the file via the security label, and the process does not have appropriate privilege.
EFAULT	The *path* argument points outside the process address space.
ENAMETOOLONG	The supplied file name is too long.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.
ESYSLV	The caller does not have an authorized `secadm` or `sysadm` category and is not a trusted process.

The `fsecstat` system call fails if one of the following error conditions occurs:

Error Code	**Description**
EBADF	The *fildes* argument is not a valid open file descriptor.
EBADF	The process is not granted read permission to the file via the security label, and the process does not have appropriate privilege.
EFAULT	The *buf* argument points outside the process address space.
EINVAL	The specified file is a socket.
ESYSLV	The caller does not have a authorized `secadm` or `sysadm` category, is not a trusted process, and is not authorized to execute at the security label of the file.

BUGS

When `secstat`, `lsecstat`, or `fsecstat` is used to obtain the labeling information for an inactive pty device special file, the labeling reported reflects the label and label range of the calling process. If the active label of the calling process is outside the label range of the calling process, the label and range returned reflects this. Since this is an illegal combination, any attempt to recreate a pty device node with these attributes fail. Because the pty device is automatically relabeled the next time it is used, this failure does not leave the pty device in an incorrectly labeled state even if it appears to do so.

FILES

`/usr/include/sys/secstat.h`	Defines secstat structure
`/usr/include/sys/types.h`	Contains types required by ANSI X3J11

SEE ALSO

`getfacl`(2), `setdevs`(2), `setfacl`(2), `setfcmp`(2), `setfflg`(2), `setflvl`(2)

`spset`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

General UNICOS System Administration, Cray Research publication SG–2301

NAME

select – Examines synchronous I/O multiplexing

SYNOPSIS

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

int select (int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);

FD_SET (fd, &fdset);
FD_CLR (fd, &fdset);
FD_ISSET (fd, &fdset);
FD_ZERO (&fdset);
int fd;
fd_set fdset;
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The select system call examines I/O descriptor sets to determine whether the files associated with the specified file descriptors are ready for reading, are ready for writing, or have an exceptional condition pending.

An I/O descriptor set is an array of long integers where each bit in the array corresponds to a file descriptor defined for the process. The leftmost bit in the first array element corresponds to file descriptor 0, and so on. The number of bits allocated in each descriptor set is defined by the parameter FD_SETSIZE in header file <sys/types.h> which corresponds to the maximum number of files a process can have open concurrently.

The select system call accepts the following arguments:

nfds Specifies the number of file descriptors that are to be checked in the descriptor sets pointed to by *readfds*, *writefds*, and *exceptfds*. The bits from 0 through *nfds* -1 in the descriptor sets are examined.

readfds Points to an I/O descriptor set used to specify which file descriptors are to be examined to determine if the associated files are ready for reading. If no file descriptors are to be examined for reading, specify a null pointer.

writefds Points to an I/O descriptor set used to specify which file descriptors are to be examined to determine if the associated files are ready for writing. If no file descriptors are to be examined for writing, specify a null pointer.

exceptfds Points to an I/O descriptor set used to specify which file descriptors are to be examined to determine if the associated files have any exceptional conditions pending. If no file descriptors are to be examined for exceptional conditions, specify a null pointer.

timeout Points to a `timeval` structure which specifies the maximum interval to wait for the examination to complete. If *timeout* is a null pointer, `select` is blocked indefinitely. To cause a poll, the *timeout* argument must be nonzero, pointing to a `timeval` structure containing 0 values.

The `select` system call returns, in place, descriptor sets of the file descriptors that are ready. The value returned by `select` is the total number of file descriptors which are ready.

The following macros are provided for manipulating I/O descriptor sets. In these descriptions, *fd* stands for file descriptor, and *fdset* refers to file descriptor sets.

`FD_CLR (`*fd*`, &`*fdset*`)` Removes *fd* from *fdset*.

`FD_ISSET (`*fd*`, &`*fdset*`)` Returns nonzero if *fd* is a member of *fdset*; otherwise, returns 0.

`FD_SET (`*fd*`, &`*fdset*`)` Includes a particular in *fd* in *&fdset*.

`FD_ZERO (`*&fdset*`)` Sets all bits in *fdset* to 0.

The behavior of these macros is undefined if a descriptor value is less than 0 or greater than or equal to `FD_SETSIZE`, which is normally equal to the maximum number of files a process can have open concurrently. A program may give `FD_SETSIZE` a larger value by defining it before the inclusion of header file `<sys/types.h>`.

NOTES

The active security label of the calling process must be greater than or equal to the security label of each file. If this condition is not met for a given file descriptor, events on that file descriptor are ignored.

A process with the effective privilege shown is granted the following ability:

Privilege	Description
`PRIV_MAC_READ`	The calling process is allowed to override the security label restrictions.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to override security label restrictions.

RETURN VALUES

If `select` completes successfully, it returns the number of descriptors contained in the descriptor sets. If the time limit expires, `select` returns 0.

Because failure to meet the security label requirements causes a file descriptor to be ignored, this failure alone does not result in an error.

If an error occurs, a value of –1 is returned, the descriptor sets remain unmodified (even in the case of an interrupted call), and `errno` is set to indicate the error.

ERRORS

An error return from the `select` system call indicates one of the following conditions:

Error Code	Description
EBADF	One of the descriptor sets specified a descriptor that was not valid.
EFAULT	The argument address is not valid.
EINTR	A signal was delivered before any of the selected events occurred, or the time limit expired.
EINVAL	The specified time limit is not valid; one of its components is negative or too large, or *nfds* is less than or equal to 0.

BUGS

The current implementation works only for the master side of a pty, a tty, a pipe, and sockets.

The `select` system call should probably return the time remaining from the original time-out, if any, by modifying the time value in place. This may be implemented in future versions of the system. Therefore, do not assume that the time-out value will be unmodified by `select`.

EXAMPLES

The following example shows how to use the `select` system call:

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/time.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

/*
 *
 *
 *      Returns:
 *              0, if read would block
 *              1, if read would not block
 */
chkread(fd)
int fd;                                 /* File descriptor */
{
        int nfound;                     /* Number of ready descriptors */
        fd_set readfds;                 /* Read file descriptors bit mask */
        struct timeval timeout;


        FD_ZERO(&readfds); FD_SET(fd, &readfds);

        timeout.tv_sec = 0;             /* Cause select to return immediately */
        timeout.tv_usec = 0;

        while ((nfound = select(FD_SETSIZE, &readfds, 0, 0, &timeout)) == -1) {
               if (errno == EINTR)
                     continue;         /* Ignore interrupts */

                fprintf(stderr, "select() failed, errno = %d\n", errno);
                exit(1);
        }
        return(nfound);
 }
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `select` system call

SEE ALSO

`read`(2), `write`(2)

NAME

`semctl` – Provides semaphore control operations

SYNOPSIS

```
#include <sys/sem.h>
int semctl (int semid, int semnum, int cmd, union semun arg...);
union semun {
   int val;
   struct semid_ds *buf;
   ushort *array;
};
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

XPG4

DESCRIPTION

The `semctl` system call provides a variety of semaphore control operations as specified by *cmd*. It accepts the following arguments:

semid Specifies a semaphore identifier associated with a set of semaphores.

semnum Identifies the semaphore in the *semid* group.

cmd Specifies a semaphore control operation. The following are valid *cmd* values.

The following semaphore control operations are executed for the semaphore specified by *semid* and *semnum*. The level of permission required for each operation is specified with each command (see `sem`(5) `ipc`(7)).

`GETVAL` Returns the value of `semval` (see `sem`(5)). This command requires read permission.

`SETVAL` Sets the value of `semval` to *arg.*`val`. When this command is successfully executed, the `semadj` value corresponding to the specified semaphore in all processes is cleared. This command requires alter permission.

`GETPID` Returns the value of `sempid`. This command requires read permission.

`GETNCNT` Returns the value of `semncnt`. This command requires read permission.

`GETZCNT` Returns the value of `semzcnt`. This command requires read permission.

The following values for *cmd* operate on each `semval` in the set of semaphores (see `sem`(5)):

GETALL Places semvals into the array (of type unsigned short) pointed to by *arg.*array. This command requires read permission.

SETALL Sets semvals according to the array (of type unsigned short) pointed to by *arg.*array. When this *cmd* is successfully executed, the semadj values corresponding to each specified semaphore in all processes are cleared. This command requires alter permission.

The following values for *cmd* are also available (see ipc(5)):

IPC_STAT Places the current value of each member of the semid_ds data structure associated with *semid* into the semid_ds structure pointed to by *arg.*buf. The contents of this structure are defined in sem(5). This command requires read permission.

IPC_SET Sets the value of the members of the semid_ds data structure associated with *semid* to the corresponding value found in the semid_ds structure pointed to by *arg.*buf. See the following:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode     /* only access permission bits */
```

The mode bits specified in ipc(7) are copied into the corresponding bits of the sem_perm.mode associated with *semid*. The values of any other bits are unaltered.

The IPC_SET command can be executed only by a process that has an effective user ID equal to the value of sem_perm.cuid or sem_perm.uid in the semid_ds data structure associated with *semid*.

IPC_RMID Removes the semaphore identifier specified by *semid* from the system and destroys the set of semaphores and semid_ds data structure associated with it. This command can be executed only by a process that has an effective user ID equal to the value of sem_perm.cuid or sem_perm.uid in the semid_ds data structure associated with *semid*.

IPC_SETACL Sets the access control list (ACL) on the semaphore set specified by *semid*. The ipc_perm structure within the semid_ds structure pointed to by *buf* contains a pointer, ipc_acl, to an acl_rec structure with the required ACL entries, and a count of those entries, ipc_aclcount. If an ACL exists for the semaphore set, it is replaced by the one provided with this call. If ipc_aclcount is 0, any existing ACL is removed. The calling process must be the owner of the semaphore set specified by *semid*.

IPC_GETACL Retrieves the access control list (ACL) for the semaphore set specified by *semid*. The ipc_perm structure within the semid_ds structure pointed to by *buf* contains a pointer, ipc_acl, to an acl_rec structure where the ACL entries are to be returned. The count of entries to be returned is specified in the ipc_aclcount field. If there are more than ipc_aclcount entries, only the first ipc_aclcount is returned. If there are fewer than ipc_aclcount entries, all entries are returned. The return value indicates the number of entries returned. If there is no ACL, the return value is 0. The calling process must have read permission to the semaphore set specified by *semid*.

IPC_SETLABEL
Sets the security label on the semaphore set specified by *semid*. The ipc_perm structure within the semid_ds structure pointed to by *buf* contains a security level, ipc_slevel, and a compartment set, ipc_scomps, to be set in the security label on the semaphore set. Only a process with the appropriate privilege can perform this operation.

arg Specifies an optional structure used by the *cmd* argument.

NOTES

A process is granted read permission to a semaphore set only if the active security label of the process is greater than or equal to the security label of the semaphore set, and the process is granted read access by the semaphore set ACL (if one is assigned). This applies to the IPC_STAT and IPC_GETACL operations.

The IPC_SET, IPC_RMID, and IPC_SETACL operations require that the active security label of the process is equal to the security label of the semaphore set.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_MAC_READ	The process is considered to meet the security label requirements for being granted read permission to a semaphore set.
PRIV_MAC_WRITE	The process is considered to meet the security label requirements for performing an IPC_SET, IPC_RMID, or IPC_SETACL operation.
PRIV_DAC_OVERRIDE	The process is considered to meet the permission mode and ACL requirements for being granted read permission to a semaphore set.
PRIV_FOWNER	The process is considered to meet the semaphore set ownership requirements for the IPC_SET, IPC_RMID, and IPC_SETACL operations.

If the PRIV_SU configuration option is enabled, the super user is granted the same abilities as all effective privileges shown above.

The super user is considered the owner of a semaphore set, and is granted read permission to that semaphore set.

RETURN VALUES

Upon successful completion, the value returned by `semctl` depends on *cmd*, as follows:

`GETVAL`	Value of *semval*
`GETPID`	Value of *sempid*
`GETNCNT`	Value of *semncnt*
`GETZCNT`	Value of *semzcnt*
All others	Value of 0

Otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `semctl` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	Operation permission is denied to the calling process (see `sem`(5)).
`EACCES`	The *cmd* argument is `IPC_GETACL`, and the calling process does not have read permission.
`EFAULT`	*arg.*`buf` points to an illegal address.
`EFAULT`	The *cmd* argument is `IPC_SETACL` or `IPC_GETACL`, and the *ipc_acl* field in *buf* points to an illegal address.
`EINVAL`	The *semid* argument is not a valid semaphore identifier.
`EINVAL`	The *semnum* argument is less than 0 or greater than (`sem_nsems` −1).
`EINVAL`	The *cmd* argument is not a valid command.
`EINVAL`	The *cmd* argument is `IPC_SET`, and `sem_perm.uid` or `sem_perm.gid` is not valid.

EINVAL	The *cmd* argument is IPC_SETACL, and one of the following is true: • The ipc_aclcount field in *buf* is 0, but there is no ACL associated with *msqid*. • The ipc_aclcount field in *buf* is less than 0 or greater than 256. • The ACL supplied failed validation.
ENOMEM	The *cmd* argument is IPC_SETACL, and no memory was available to store the ACL. The command should be retried at a later time.
EPERM	The *cmd* argument is equal to IPC_RMID or IPC_SET, and the effective user ID of the calling process is not equal to the value of sem_perm.cuid or sem_perm.uid in the semid_ds data structure associated with *semid*, and the calling process does not have appropriate privilege.
EPERM	The *cmd* argument is IPC_SETLABEL, and the calling process does not have appropriate privilege.
EPERM	The *cmd* argument is IPC_SETACL, and the calling process does not meet ownership requirements and does not have appropriate privilege.
ERANGE	The *cmd* argument is SETVAL or SETALL, and the value to which semval is to be set is greater than the system-imposed maximum.

FILES

/usr/include/sys/sem.h	Contains semaphore-related data structures and macros

SEE ALSO

semget(2), semop(2)

ipcs(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

ipc(5), sem(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

ipc(7) Online only

NAME

semget – Provides access to semaphore identifiers

SYNOPSIS

```
#include <sys/sem.h>
int semget (key_t key, int nsems, int semflg);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

XPG4

DESCRIPTION

The semget system call returns the semaphore identifier associated with *key*. It accepts the following arguments:

key Specifies the semaphore.

nsems Specifies the number of semaphores to allocate for the *key*.

semflg Specifies a flag value.

A semaphore identifier, with its associated semid_ds data structure and set containing *nsems* semaphores (see sem(5)), is created for *key* if one of the following is true:

- *key* is equal to IPC_PRIVATE.
- *key* does not already have a semaphore identifier associated with it, and *semflg*&IPC_CREAT is not 0.

Upon creation, the semid_ds data structure associated with the new semaphore identifier is initialized as follows:

- sem_perm.cuid, sem_perm.uid, sem_perm.cgid, and sem_perm.gid are set to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of sem_perm.mode are set to the low-order 9 bits of *semflg*.
- sem_nsems is set to the value of *nsems*.
- sem_otime is set to 0, and sem_ctime is set to the current time.
- The data structure associated with each semaphore in the set is not initialized. The SETVAL or SETALL command of the semctl(2) system call can be used to initialize each semaphore.

NOTES

If the calling process has the ipc_persist permission bit, the semaphore set will be created as a persistent set. Persistent semaphore sets will not be removed from the system unless a semctl(2) system call with the command IPC_RMID or an ipcrm(1) command is performed on the set.

If the calling process does not have this permission bit, the semaphore set will be linked into a list of nonpersistent sets belonging to the session of which the process is a member. When the last process of the session terminates, all the semaphore sets linked to the session will be removed from the system.

A process with the effective privilege shown is granted the following abilities:

Privilege	Description
PRIV_RESOURCE	The process is considered to have the ipc_persist permission bit.

If the PRIV_SU configuration option is enabled, the super user is granted the same abilities as all effective privileges shown above. The super user is considered to have the ipc_persist permission bit.

RETURN VALUES

If semget completes successfully, a nonnegative integer, namely a semaphore identifier, is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The semget system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A semaphore identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *semflg* would not be granted (see ipc(7)).
EEXIST	A semaphore identifier exists for *key* but both *semflg*&IPC_CREAT and *semflg*&IPC_EXCL are not 0.
EINVAL	The value of *nsems* is either less than or equal to 0 or greater than the system-imposed limit.
EINVAL	A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems*, and *nsems* is not equal to 0.
ENOENT	A semaphore identifier does not exist for *key*, and *semflg*&IPC_CREAT is 0.
ENOSPC	A semaphore identifier is to be created, but the system-imposed limit on the maximum number of allowed semaphore identifiers system-wide would be exceeded.

FILES

`/usr/include/sys/sem.h`	Contains semaphore-related data structures and macros

SEE ALSO

`semctl`(2), `semop`(2)

`ipcrm`(1), `ipcs`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`stdipc`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`ipc`(5), `sem`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`ipc`(7) Online only

NAME

semop – Provides general semaphore operations

SYNOPSIS

```
#include <sys/sem.h>
int semop (int semid, struct sembuf *sops, size_t nsops);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

XPG4

DESCRIPTION

The semop system call is used to perform an array of semaphore operations atomically on the set of semaphores associated with the semaphore identifier.

The semop system call accepts the following arguments:

semid Specifies a semaphore identifier associated with a set of semaphores.

sops Points to the array of semaphore-operation structures.

nsops Specifies the number of such structures in the array. Each sembuf structure includes the following members:

```
short sem_num;    /* semaphore number */
short sem_op;     /* semaphore operation */
short sem_flg;    /* operation flags */
```

Each semaphore operation specified by sem_op is performed on the corresponding semaphore specified by *semid* and sem_num. See the sem(5) man page for information on the available types of permissions. The variable sem_op specifies one of three semaphore operations, as follows:

1. If sem_op is a negative integer and the calling process has alter permission, one of the following actions occurs:
   - If semval (see sem(5)) is greater than or equal to the absolute value of sem_op, the absolute value of sem_op is subtracted from semval. Also, if *sem_flg*&SEM_UNDO is not 0, the absolute value of sem_op is added to the calling process' semadj value for the specified semaphore (see exit(2)).
   - If semval is less than the absolute value of sem_op and *sem_flg*&IPC_NOWAIT is not 0, semop returns immediately.

- If semval is less than the absolute value of sem_op and *sem_flg*&IPC_NOWAIT is 0, semop increments the semncnt associated with the specified semaphore and suspends execution of the calling process until one of the following conditions occurs:
  - semval becomes greater than or equal to the absolute value of sem_op. When this occurs, the value of semncnt associated with the specified semaphore is decremented, the absolute value of sem_op is subtracted from semval, and, if *sem_flg*&SEM_UNDO is not 0, the absolute value of sem_op is added to the calling process' semadj value for the specified semaphore.
  - The *semid* for which the calling process is awaiting action is removed from the system (see semctl(2)). When this occurs, errno is set equal to EIDRM, and a value of −1 is returned.
  - The calling process receives a signal that is to be caught. When this occurs, the value of semncnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in the sigaction(2) system call.

2. If sem_op is a positive integer and the calling process has alter permission, the value of sem_op is added to semval, and, if *sem_flg*&SEM_UNDO is true, the value of sem_op is subtracted from the calling process's semadj value for the specified semaphore.
3. If sem_op is 0 and the calling process has read permission, one of the following actions occurs:
   - If semval is 0, semop returns immediately.
   - If semval is not equal to 0 and *sem_flg*&IPC_NOWAIT is not 0, semop returns immediately.
   - If semval is not equal to 0 and *sem_flg*&IPC_NOWAIT is also 0, semop increments the semzcnt associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:
     - semval becomes 0, at which time the value of semzcnt associated with the specified semaphore is decremented.
     - The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, errno is set equal to EIDRM, and a value of −1 is returned.
     - The calling process receives a signal that is to be caught. When this occurs, the value of semzcnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in the sigaction(2) system call.

Upon successful completion, the value of sempid for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

NOTES

A process is granted read permission to a semaphore set only if the active security label of the process is greater than or equal to the security label of the semaphore set, and the process is granted read access by the semaphore set access control list (ACL) (if one is assigned).

A process is granted write permission to a semaphore set only if the active security label of the process is equal to the security label of the semaphore set, and the process is granted write access by the semaphore set ACL (if one is assigned).

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_MAC_READ	The process is considered to meet the security label requirements for being granted read permission to a semaphore set.
PRIV_MAC_WRITE	The process is considered to meet the security label requirements for being granted write permission to a semaphore set.
PRIV_DAC_OVERRIDE	The process is considered to meet the permission mode and ACL requirements for being granted read and write permission to a semaphore set.

If the PRIV_SU configuration option is enabled, the super user is granted the same abilities as all effective privileges shown above. The super user is granted read and write permission to a semaphore set.

RETURN VALUES

If semop completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The semop system call fails if one or more of the following are true for any of the semaphore operations specified by *sops*:

Error Code	Description
E2BIG	The *nsops* argument is greater than the system-imposed maximum.
EACCES	Operation permission is denied to the calling process (see ipc(7)).
EAGAIN	The operation would result in suspension of the calling process, but *sem_flg*&IPC_NOWAIT is not 0.
EFAULT	The *sops* argument points to an illegal address.
EFBIG	The *sem_num* field is less than 0 or greater than or equal to the number of semaphores in the set associated with *semid*.
EIDRM	The semaphore identifier *semid* was removed from the system.
EINTR	The semop system call was interrupted by a signal.
EINVAL	The *semid* argument is not a valid semaphore identifier.
EINVAL	The number of individual semaphores for which the calling process requests a SEM_UNDO would exceed the limit.

ENOSPC	The limit on the number of individual processes requesting a SEM_UNDO would be exceeded.
ERANGE	An operation would cause a semval value to overflow the system-imposed limit.
ERANGE	An operation would cause a semadj value to overflow the system-imposed limit.

FILES

/usr/include/sys/sem.h	Contains semaphore-related data structures and macros

SEE ALSO

exec(2), exit(2), fork(2), semctl(2), semget(2), sigaction(2)

ipcs(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

ipc(5), sem(5), types(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

ipc(7) Online only

NAME

send, sendmsg, sendto – Sends a message from a socket

SYNOPSIS

All Cray Research systems:

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

int send (int s, char *buf, int len, int flags);

int sendto (int s, char *buf, int len, int flags, struct sockaddr *to,
int tolen);
```

Cray PVP systems:

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

int sendmsg (int s, struct msghdr *buf, int flags);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The send, sendmsg, and sendto system calls transmit a message (*buf*) to another socket.

You can use a send call only on a connected socket. You can use sendto or sendmsg on either a connected or unconnected socket. The sendmsg system call uses the same msghdr structure as the recvmsg(2) system call to minimize the number of directly supplied arguments. For more information, see the connect(2) and recv(2) man pages.

The send, sendmsg, and sendto system calls accept the following arguments:

s Specifies the descriptor for a socket.

buf Points to the address of the message to be sent. See recv(2) for a description of the msghdr structure.

len Specifies the number of bytes to be sent. If the message is too long to pass atomically through the underlying protocol, the error message EMSGSIZE is returned, and the message is not transmitted.

flags Specifies optional flags that control transmission of the message. The following are valid values for *flag*:

MSG_OOB Specifies that the message should be sent out-of-band on sockets that support such a notion. Out-of-band messages correspond to the TCP notion of urgent data.

MSG_DONTROUTE Specifies that the message be sent without using local routing tables. Allows the caller to take control of routing (for example, in network debugging software).

to Points to a sockaddr structure that must be filled with the destination address.

tolen Specifies the length of the destination address, specified by *to*.

The send system call is unreliable. A return value of −1 indicates some locally detected errors, but you cannot determine whether the message was received. The send call only queues data for transmission. When send returns 0, it indicates that the message was put in the queue. If the message is not received, error messages are unavailable.

If no message space is available at the socket to hold the message to be transmitted, send usually waits for space to become available, unless the socket was placed in the nonblocking I/O mode by an ioctl(2) request of FIONBIO. You can use the select(2) call to determine when it is possible to send more data.

NOTES

These system calls can be subjected to additional security rules. The two sockets being connected each have security attributes that are inherited from their associated processes. These attributes must be equal if the SOCKET_MAC option is enabled. In addition, the network and remote host have security-attribute ranges, which are specified in the network access list (NAL) portion of the spnet.conf configuration file and administered with the spnet(8) command.

If the SOCKET_MAC option is not enabled, the security attributes of the socket are not required to be equal, but the security range of the process, which is specified in the UDB for the user, must include the minimum label for the remote host as specified in the NAL. Note that SOCKET_MAC is part of TCP/IP configurable feature variables list in uts/cf/Nmakefile.

A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_MAC_WRITE	The process is allowed to override the security label restrictions when the SOCKET_MAC option is enabled.

If the PRIV_SU configuration option is enabled, the super user is allowed to override security label restrictions when the SOCKET_MAC option is enabled.

RETURN VALUES

If send, sendmsg, or sendto completes successfully, the number of characters sent is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The send, sendmsg, or sendto system call fails if one of the following conditions occurs:

Error Code	Description
EACCES	Permission is denied (because of a security violation).
EACCES	If the SOCKET_MAC option is enabled, the process does not meet the security label requirements and does not have appropriate privilege.
EBADF	Descriptor *s* is not valid.
EFAULT	Invalid user space address is specified for a parameter.
EMSGSIZE	Socket requires the message to be sent atomically, and the size of the message to be sent makes this impossible.
EMSGSIZE	The msg_iovlen field is greater than or equal to the MSG_MAXIOVLEN parameter (defined in the sys/socket.h file).
ENOBUFS	System cannot allocate an internal buffer. The operation can succeed when buffers become available.
ENOBUFS	Output queue for a network interface is full. This generally indicates that the interface has stopped sending, but it can indicate transient congestion.
ENOTSOCK	Descriptor *s* is not a socket.
EWOULDBLOCK	Socket is marked nonblocking, and the requested operation would block.

FILES

/etc/config/spnet.conf	Network access list file
/usr/adm/sl/slogfile	Receives security log records
/usr/include/sys/socket.h	Contains definitions related to sockets, types, address families, and options
/usr/include/sys/types.h	Contains types required by ANSI X3J11

SEE ALSO

ioctl(2), recv(2), select(2), socket(2)

slog(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

spnet(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

UNICOS Networking Facilities Administrator's Guide, Cray Research publication SG–2304

NAME

setash – Sets an array session handle

SYNOPSIS

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

#int setash (ash_t ash);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The setash system call changes the handle for the array session containing the current process to the specified value. The current process must have super-user privileges to invoke the setash system call.

Ordinarily, a handle that is unique within the current system is assigned to an array session when the array session is created with the newarraysess(2) system call. The setash system call can override this default handle, perhaps for assigning a handle that is unique across an entire array or for synchronizing handles with an array session on another system.

The setash system call accepts the following argument:

ash Represents the array session handle that is to be assigned to the current array session. The handle specified by *ash* must be a positive value, must not be in use on the current system, and must not be in the range of values that UNICOS uses for default array session handles. The range of default handles is defined by the system variables minash and maxash.

RETURN VALUES

If setash completes successfully, a value of 0 is returned; otherwise, a value of -1 is returned, and errno is set to indicate the error.

ERRORS

The setash system call fails if one of the following conditions occurs:

Error Code	Description
EINVAL	*ash* is negative, in use by another array session on this system, or in the range of values reserved by the system for default array session handles.
EPERM	The current process does not have super-user privileges.

SEE ALSO

getash(2), newarraysess(2)

array_services(7), array_sessions(7)

NAME

setdevs – Sets file security label and security flag attributes

SYNOPSIS

```
#include <sys/secdev.h>
int setdevs (char *dname, struct secdev *sdev);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setdevs system call sets the minimum, maximum, and active security labels, and the security flags of a file to the values contained in the secdev structure.

The setdevs system call accepts the following arguments:

dname Specifies the file for which the security labels and flags are set.

sdev Points to the secdev structure which contains the security values to be set.

A secdev structure includes the following members:

```
int     dv_minlvl;      /* minimum security level */
int     dv_maxlvl;      /* maximum security level */
int     dv_actlvl;      /* active security level */
long    dv_valcmp;      /* authorized compartments */
long    dv_actcmp;      /* active compartments */
int     dv_intcls;      /* active integrity class (not used) */
long    dv_intcat;      /* active integrity categories (not used) */
int     dv_devflg;      /* device security flags */
int     dv_devprv;      /* device privileges */
```

NOTES

Only an appropriately privileged process can set the minimum or maximum security label of a device special file. Any process can attempt to set the minimum or maximum security label of a file that is not a device special file. If the file is not a device special file, then the supplied minimum and maximum security labels are ignored, and the file's minimum and maximum security labels are set to the minimum and maximum security labels of the file system on which the file resides.

Only an appropriately privileged process can downgrade the active security label of the specified file. If the specified file is an empty directory, then any process can upgrade the active security label of the file. Otherwise, only an appropriately privileged process can upgrade the active security label of the file.

If the specified file is a device special file, and the supplied security flags include the `mldev` flag, then the supplied active security label is ignored, and the active security label of the file is set to the supplied maximum security label.

If the supplied file is a device special file, and the supplied security flags do not include the `secdv` flag, then the `secdv` and `state` flags are turned off for the specified file.

This system call changes only the state of the `state`, `secdv`, `mldev`, and `entry` security flags. No other security flags are changed.

If the file is not a device special file, attempts to enable `state` are ignored. If the file is a device special file, attempts to enable `state` without also enabling `secdv` are ignored. Disabling `secdv` automatically causes `state` to be disabled.

Attempts to change the security label or flags on a public device are ignored. Attempts to change the flags on a pseudo tty are ignored.

A user is allowed to upgrade the label on his/her directory if the directory is empty and the user has write access to the directory.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_ADMIN`	The process is allowed to set the minimum and maximum security label of the file.
`PRIV_ADMIN`	The process is allowed to set the security flags of the file.
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path prefix via the permission bits and access control list.
`PRIV_MAC_DOWNGRADE`	The process is allowed to downgrade the active security label of the file.
`PRIV_MAC_READ`	The process is granted search permission to a component of the path prefix via the security label.
`PRIV_MAC_UPGRADE`	The process is allowed to upgrade the active security label of the file.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix. The super user is allowed to set the minimum, maximum, and active security label of the file, and the security flags of the file.

RETURN VALUES

If `setdevs` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setdevs` system call fails if one of the following error conditions occurs:

Error Code	Description
`EACCES`	A component of the path prefix denies search permission.
`ECOMPV`	The specified compartments are not valid within the compartments of the system.
`ECOMPV`	If the `MLS_OBJ_RANGES` configuration option is enabled, and the specified compartments are not valid compartments for the file.
`EFAULT`	Error occurred in reading `setdev` structure.
`EINVAL`	The security label parameters are not valid.
`EINVFS`	The file system on which the file exists is a pre-UNICOS 6.0 file system.
`ENAMETOOLONG`	The *dname* argument is longer than allowed by `PATH_MAX`.
`ENOENT`	The specified file does not exist.
`ENOTDIR`	A component of the path prefix is not a directory.
`EPERM`	The process does not have permission to upgrade an empty directory.
`ESECADM`	The process does not have appropriate privilege to use this system call.
`ESYSLV`	If the `MLS_OBJ_RANGES` configuration option is enabled, and the specified levels are not within the security level range of the system.

SEE ALSO

`secstat`(2), `setfacl`(2), `setfcmp`(2), `setfflg`(2)

`slog`(4), `slrec`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`spdev`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

General UNICOS System Administration, Cray Research publication SG–2301

NAME

setfacl – Sets access control list for file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/acl.h>
int setfacl (char *fname, struct acl *aclents, int count);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setfacl system call sets the access control list (ACL) of a file to the value specified in an array. A file's ACL controls, in part, access to the file. Only the file owner or a process with appropriate privilege can set an ACL on a file. If the process is not a member of the owning group of the file, the set-group-ID mode bit of the file is cleared unless the process has appropriate privilege. If the FSETID_RESTRICT system configuration parameter is enabled, the set-user-ID and set-group-ID mode bits of a file are cleared unless the process has appropriate privilege.

The setfacl system call accepts the following arguments:

fname Specifies the file for which the ACL is set.

aclents Specifies an array of ACL entries.

count Indicates the number of entries in the array; cannot exceed 256.

NOTES

A setfacl request replaces any previously existing ACL on the file.

If the system is configured with discretionary access violation logging enabled, all errors are recorded in the security log (except errors of type ENOENT, ENOTDIR, and ENAMETOOLONG).

The process must have write permission to the file via the security label. That is, the active security label of the process must be equal to the security label of the file.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_DAC_OVERRIDE	The process is granted search permission to the component via the permission bits and ACL.
PRIV_FOWNER	The process is considered the owner of the file.

PRIV_FSETID	The file's set-user-ID or set-group-ID mode bit are not cleared.
PRIV_MAC_READ	The process is granted search permission to the component via the security label.
PRIV_MAC_WRITE	The process is granted write permission to a component of the path prefix via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted write permission to the file via the security label. The super user is considered the file owner and is allowed to set an ACL on a file whose mode includes the set-user-ID or set-group-ID mode bit. If the caller is the super user, the file's set-user-ID and set-group-ID mode bits are not cleared.

RETURN VALUES

If setfacl completes successfully, the number of elements in the file's ACL is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The setfacl system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A component of the path prefix denies search permission.
EFAULT	The *fname* argument points outside the process address space.
EFAULT	The *aclents* argument points outside the process address space.
EINVAL	The *count* argument is less than 0. If *count* exceeds 256, its value is truncated to 256.
EINVAL	The specified file resides on a nonnative file system.
EMANDV	The process does not have write permission to the file via the security label.
ENAMETOOLONG	The specified file name is too long.
ENOACL	The file's previously assigned ACL, if one exists, is corrupted.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.
EOWNV	The process is not the file owner and does not have appropriate privilege.

EXAMPLES

This example shows how to use the setfacl system call to create ACL entries for a file.

First, a getfacl(2) system call displays (on stdout) the current ACL entries for the file, argv[1]. After this display, the program allows the user to add entries to the ACL. A setfacl request then creates a new ACL for the designated file.

ACLSIZE, defined in the sys/acl.h file, specifies the maximum number of entries that can exist in an ACL.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/acl.h>
#include <pwd.h>

main(int argc, char *argv[])
{
     struct acl buf[ACLSIZE];
     struct passwd *pwptr;
     char lid[15];
     int no, i;

     if ((no = getfacl(argv[1], buf, ACLSIZE)) == -1) {
          perror("getfacl failed");
          exit(1);
     }

     printf("Access control list for %s currently contains", argv[1]);
     printf(" the following users:\n\n");
     printf("ID         Login ID   Name");
     printf("                        Permissions\n\n");

     for (i = 0; i < no; i++) {
          pwptr = getpwuid(buf[i].ac_usid);
          printf("%-5d      %-10s %-25s   %c%c%c\n",
                  buf[i].ac_usid, pwptr->pw_name, pwptr->pw_gecos,
                  buf[i].ac_mode & 04 ? 'r' : ' ',
                  buf[i].ac_mode & 02 ? 'w' : ' ',
                  buf[i].ac_mode & 01 ? 'x' : ' ');
     }

     /* Add entries to access control list. */
     printf("\nWhich entries are to be added (q to quit)?\n\n");
     while(1) {
          printf("User's login ID --> ");
          gets(lid);
          if (strcmp(lid, "q") == 0) break;
          if ((pwptr = getpwnam(lid)) == NULL) {
               fprintf(stderr, "Invalid login ID!\n");
               continue;
          }
          buf[no].ac_usid = pwptr->pw_uid;
          buf[no].ac_grid = pwptr->pw_gid;
```

```
                buf[no].ac_flag = FLAG_UIDGID;
                buf[no].ac_mode = 04;  /* allow read permission only */
                buf[no].ac_sort = 0;
                buf[no].ac_same = 0;
                no++;                /* increment number of ACL entries */
           }

           if (setfacl(argv[1], buf, no) == -1) {
                perror("setfacl failed");
                exit(1);
           }
      }
```

SEE ALSO

getfacl(2), rmfacl(2), secstat(2), setdevs(2), setfcmp(2), setfflg(2), setflvl(2)

spacl(1), spclr(1), spset(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

slog(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

setfcmp – Sets file compartments

SYNOPSIS

```
#include <unistd.h>
int setfcmp (char *fname, long fcmp);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setfcmp system call sets the compartments of a file to the value specified by the compartment bit mask. A file's compartments control, in part, access to the file on a UNICOS system. Only a process with appropriate privilege can use this system call.

The setfcmp system call accepts the following arguments:

fname Specifies the file for which the compartments are set.

fcmp Specifies the compartment bit mask which determines the value of the compartments to be set.

NOTES

All setfcmp requests are recorded in the security log, indicating success or failure (except errors of type ENOENT, ENOTDIR, and ENAMETOOLONG).

A user is allowed to upgrade the label on his or her directory if the directory is empty and the user has write access to the directory.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_DAC_OVERRIDE	The process is granted search permission to a component of the path prefix via the permission bits and access control list.
PRIV_MAC_DOWNGRADE	The process as allowed to use this system call to set the file's compartments to a value that does not include every compartment that is currently associated with the file.
PRIV_MAC_READ	The process is granted search permission to a component of the path prefix via the security label.

PRIV_MAC_UPGRADE	The process is allowed to use this system call to set the file's compartments to a value that includes at least every compartment that is currently associated with the file.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix and is allowed to upgrade or downgrade the file's compartments.

RETURN VALUES

If setfcmp completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The setfcmp system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A component of the path prefix denies search permission.
ECOMPV	The requested compartments are not authorized for the file system in which the file resides.
EFAULT	The *fname* argument points outside the process address space.
EINVAL	The specified file resides on a nonnative file system.
ENAMETOOLONG	The specified file name is too long.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The process does not have permission to upgrade an empty directory.
ESECADM	The process does not have appropriate privilege to use this system call.

The security administrator's security level range must include the security level of the file, and the security administrator's authorized compartments must dominate the specified compartments of the file.

FILES

/usr/include/unistd.h	Contains C prototype for the setfcmp system call

SEE ALSO

`secstat`(2), `setdevs`(2), `setfacl`(2), `setfflg`(2), `setflvl`(2)

`spset`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`slog`(4), `slrec`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

setfflg – Sets file security flags

SYNOPSIS

```
#include <unistd.h>
#include <sys/tfm.h>
int setfflg (char *fname, long flags);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setfflg system call sets the security flags of a file to the value specified by flag bit mask. A file's security flags indicate whether a file requires special handling. Only a process with appropriate privilege can use this system call.

The setfflg system call accepts the following arguments:

fname Specifies the file for which the flags are set.

flags Specifies the flag bit mask which determines the value of the flags to be set.

NOTES

Only the ml_symlink, exec, trapr, trapw, mldev, and entry flags can be set using the setfflg system call. Requests to set other flags are ignored. The secdv flag can be cleared but not set by setfflg. The exec flag is obsolete on UNICOS 9.1 and later systems.

If the specified file is a device special file, and the supplied security flags include the mldev flag, the active security label of the file is sent to the maximum security label.

The ml_symlink flag is the only flag that can be placed on a symbolic link. Attempts to set ml_symlink and additional flags on a symbolic link in a single request results in an error. Attempts to set the mld_symlink flag on anything other than a symbolic link results in an error. Attempts to change the flags on a public device or a pseudo ttyp are ignored.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_ADMIN	The process is allowed to use this system call.
PRIV_DAC_OVERRIDE	The process is granted search permission to a component of the path prefix via the permission bits and access control list.

PRIV_MAC_READ	The process is granted search permission to a component of the path prefix via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix and is allowed to use this system call.

RETURN VALUES

If setfflg completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The setfflg system call fails if one of following error conditions occurs:

Error Code	Description
EACCES	A component of the path prefix denies search permission.
EFAULT	The *fname* argument points outside the process address space.
EINVAL	The specified file resides on a nonnative file system.
EINVFS	The file system on which the file exists is from a release previous to the UNICOS 6.0 release.
ENAMETOOLONG	The specified file name is too long.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.
ESECADM	The process does not have appropriate privilege to use this system call.
ESECFLGV	Requested security flags are not authorized for the UNICOS system.
ESECFLGV	Requested security flags are not allowed for the object type.

FILES

/usr/include/sys/tfm.h	
/usr/include/unistd.h	Contains C prototype for the setfflg system call

SEE ALSO

secstat(2), setdevs(2), setfacl(2), setfcmp(2), setflvl(2)

spset(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

slog(4), slrec(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`setflvl` – Sets security level of a file

SYNOPSIS

```
#include <unistd.h>
int setflvl (char *fname, int flvl);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setflvl` system call sets the security level of file to a specified value. A file's security level controls, in part, access to the file on a UNICOS system. Only a process with appropriate privilege can use this system call.

The `setflvl` system call accepts the following arguments:

fname Specifies the file for which the level is set.

flvl Specifies the security level.

NOTES

All `setflvl` requests are recorded in the security log, indicating success or failure (except errors of type `ENOENT`, `ENOTDIR`, and `ENAMETOOLONG`).

A user is allowed to upgrade the label on his or her directory if the directory is empty and the user has write access to the directory.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path prefix via the permission bits and access control list.
`PRIV_MAC_DOWNGRADE`	The process is allowed to use this system call to set the security level of the file to a value that is less than or equal to the current security level of the file.
`PRIV_MAC_READ`	The process is granted search permission to a component of the path prefix via the security label.
`PRIV_MAC_UPGRADE`	The process is allowed to use this system call to set the security level of the file to a value that is greater than or equal to the current security level of the file.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix and is allowed to upgrade or downgrade the security level of the file.

RETURN VALUES

If setflvl completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The setflvl system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A component of the path prefix denies search permission.
EFAULT	The *fname* argument points outside the process address space.
EINVAL	The specified file resides on a nonnative file system.
ELEVELV	The requested security level is not authorized for the file system on which the file resides.
ENAMETOOLONG	The specified file name is too long.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The process does not have permission to upgrade an empty directory.
ESECADM	The process does not have appropriate privilege to use this system call.

FILES

/usr/include/unistd.h	Contains C prototype for the setflvl system call

SEE ALSO

secstat(2), setdevs(2), setfacl(2), setfcmp(2), setfflg(2)

spset(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

slog(4), slrec(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`setjob` – Sets job ID

SYNOPSIS

```
int setjob (int uid, int sig);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setjob` system call creates a new job by assigning a new job ID and job table entry to the calling process. If successful, the calling process becomes the first process in the job.

The `setjob` system call accepts the following arguments:

uid Specifies the real user ID of the job owner.

sig Specifies the signal to be sent to the job parent when the last process in the job exits. The job parent is defined as the parent of the calling process. If *sig* is 0, no signal is sent on job termination.

A job is a set of one or more processes. Jobs may have resource limits that are enforced by the system. The job ID and all resource limits are inherited by child processes. Only a process with appropriate privilege can use this system call.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	Description
`PRIV_RESOURCE`	The process is allowed to use this system call.

If the `PRIV_SU` configuration option is enabled, the super user or a process with the `PERMBITS_ID` permbit is allowed to use this system call.

RETURN VALUES

If `setjob` completes successfully, the job ID is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setjob` system call fails if one of the following error conditions occurs:

Error Code	Description
`EAGAIN`	The job table is full.
`EINVAL`	Invalid *uid* or *sig* argument.

EPERM	The process does not have appropriate privilege to use this system call.

SEE ALSO

fork(2), getjtab(2), killm(2), limit(2), nicem(2), signal(2), suspend(2), waitjob(2)

NAME

setlim – Sets user-controllable resource limits

SYNOPSIS

```
#include <sys/category.h>
#include <sys/resource.h>
int setlim (int id, struct resclim *rptr);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setlim system call establishes resource limit values. It accepts the following arguments:

id Specifies the PID, SID, or UID that corresponds to the resclim field resc_category. A 0 indicates the current PID, SID, or UID. Only a process with appropriate privilege can set resource limits of another user, process, or session.

rptr Points to the resclim structure. It includes the following members (for a complete list, see /usr/include/sys/resource.h):

```
struct resclim {
  int  resc_resource;          /* One of:  L_CPU */
  int  resc_category;          /* One of:  C_PROC, C_SESS, C_UID, C_SESSPROCS */
  int  resc_type;              /* One of:  L_T_ABSOLUTE, L_T_HARD, L_T_SOFT */
  int  resc_action;            /* One of:  L_A_TERMINATE, L_A_CHECKPOINT */
  int  resc_used;              /* Current amount of resource used */
  int  resc_value[R_NLIMTYPES];/* Current resource limit value for each of: */
                               /* L_T_ABSOLUTE, L_T_HARD, L_T_SOFT */
};
```

To set a limit value, all resclim fields must be set to either a value or a null. To set a value to be unlimited, use CPUUNLIM. To set a value to be null, use NULL.

The following describes each of the fields in the resclim structure and their acceptable values.

Field	Description
resc_resource	Represents the resource for which a limit is to be established. Currently, only central processing unit (CPU) resources are supported; therefore, the value of resc_resource must be L_CPU.
resc_category	Identifies which category of resource is to be set. The resc_category determines if the *id* argument is a PID, SID, or UID. Acceptable values are: C_PROC, C_SESS, C_UID, and C_SESSPROCS. The resc_category of C_SESSPROCS requires a SID. A short description follows:

Value	Description
C_PROC	Sets process limits
C_SESS	Sets session limits
C_UID	Sets user limits
C_SESSPROCS	Sets default process limits for the session

resc_type
: Identifies the type of limit to be set. Acceptable values are: L_T_ABSOLUTE, L_T_HARD, and L_T_SOFT. Only a process with appropriate privilege can set L_T_ABSOLUTE limits.

resc_action
: Determines, when a hard limit is reached, whether the process is checkpointed before termination. Acceptable values are: NULL, L_A_TERMINATE or L_A_CHECKPOINT. When the resc_action field is set to L_A_TERMINATE or L_A_CHECKPOINT, the resc_type must be L_T_HARD.

resc_used
: Is not used with the setlim system call. The acceptable value is NULL.

resc_value[R_NLIMTYPES]
: Is an array of three words that contain the absolute, hard, and soft limit values. To set the absolute limit, field resc_value[L_T_ABSOLUTE] must be set. Only a process with appropriate privilege can set absolute limits. To set hard limits, the field resc_type must be set to L_T_HARD and a value must be placed in resc_value[L_T_HARD]. To set soft limits, field resc_type must be set to L_T_SOFT and a value must be placed in resc_value[L_T_SOFT]. The values in resc_value[R_NLIMTYPES] for resc_resource L_CPU must be in clocks. Only one of the following can be set with each setlim system call: resc_value[L_T_ABSOLUTE], resc_value[L_T_HARD], or resc_value[L_T_SOFT].

NOTES

The following mandatory access control (MAC) write check is performed based on the resc_category parameter:

Parameter	Description of check
C_PROC	Against the specified process
C_SESS	Against the session leader
C_SESSPROCS	Against each process in the session
C_UID	No MAC check performed

That is, the active security label of the calling process must be equal to the security label of each process where MAC write access is being verified.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_MAC_WRITE	The calling process is granted write permission to every affected process via the security label.
PRIV_POWNER	The calling process is allowed to set the resource limits of another user, process, or session.
PRIV_RESOURCE	The calling process is allowed to set absolute limits.

If the PRIV_SU configuration option is enabled, the super user is allowed to set the resource limits of another user, process, or session. The super user is allowed to set absolute limits. If the PRIV_SU configuration option is enabled, the super user is granted write permission to every affected process via the security label.

RETURN VALUES

If setlim completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The setlim system call fails and no resource limits are set if one of the following conditions occurs:

Error Code	Description
EFAULT	The address specified for *rptr* is not valid.
EINVAL	One of the arguments contains a value that is not valid.
EPERM	The calling process does not have appropriate privilege to set absolute limits.
EPERM	The calling process does not have appropriate privilege to set resource limits of another user, process, or session.
EPERM	An attempt is made to change a limit on a system process; this is not allowed.
ESRCH	No processes are found that match the request.

SEE ALSO

getlim(2)

nlimit(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

nlimit(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NLIMIT(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

setpal – Sets the privilege assignment list (PAL) and privilege sets of a file

SYNOPSIS

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

int setpal (char *path, pal_t *buf, int bufsize);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setpal system call sets the privilege assignment list (PAL) and privilege sets of a file using the information in the buffer. The calling process must have PRIV_SETFPRIV in its effective privilege set, and must either own the file or have PRIV_FOWNER in its effective privilege set. The calling process must have MAC write access to the file or have PRIV_MAC_WRITE in its effective privilege set. The caller can change the state of privileges in the file's allowed, forced, or set-effective privilege sets only when those privileges are also in the caller's permitted privilege set.

If the PRIV_SU configuration option is enabled, then any process with effective user ID 0 meets all the requirements specified in the previous paragraph.

The setpal system call accepts the following arguments:

path Specifies the file for which the PAL and privilege sets are set.

buf Points to the buffer that contains the PAL and privilege set information.

bufsize Indicates the size of the buffer in bytes.

RETURN VALUES

If setpal completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The setpal system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A component of the *path* prefix denies search permission.
EACCES	The caller is denied MAC write access to the file.
EFAULT	The *buf* or *path* argument points outside the address space of the process.
EINVAL	The *bufsize* argument specifies an invalid value.

EINVAL	The contents of the supplied PAL is invalid.
ENAMETOOLONG	The supplied file name is too long.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the *path* prefix is not a directory.
EPERM	The process is not the file owner and does not have appropriate privilege.
EROFS	The affected file system is a read-only file system.
ESECADM	The process does not have appropriate privilege to use this system call.

SEE ALSO

fgetpal(2), fsetpal(2), getpal(2)