NAME

setpgid – Sets process-group-ID for job control

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
int setpgid (pid_t pid, pid_t pgid);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The setpgid system call is used to join either an existing process group or create a new process group within the session of the calling process. The process-group-ID of a session leader does not change.

The setpgid system call accepts the following arguments:

pid Specifies the existing process ID.

pgid Specifies the new process ID.

On successful completion, the process-group-ID of the process with a process ID that matches *pid* is set to *pgid*. As a special case, if *pid* is 0, the process ID of the calling process is used; if *pgid* is 0, the process ID of the process indicated by *pid* is used.

RETURN VALUES

If setpgid 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 setpgid system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	The value of *pid* matches the process ID of a child process of the calling process and the child process has successfully executed one of the exec(2) functions.
EINVAL	The value of *pgid* is less than 0 or is not a value supported by the implementation.

EPERM	The process indicated by *pid* is a session leader. The value of *pid* is valid but matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process. The value of *pgid* does not match the process ID of the process indicated by *pid* and no process with a process group ID exists that matches the value of *pgid* in the same session as the calling process.
ESRCH	The value of *pid* does not match the ID of the calling process or of a child of the calling process.

FILES

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

SEE ALSO

exec(2), getpgrp(2), setsid(2), tcgetpgrp(2), tcsetpgrp(2)

NAME

`setpgrp` – Sets process-group ID

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setpgrp` system call sets the process-group ID of the calling process to the process ID of the calling process and returns the new process-group ID.

RETURN VALUES

The `setpgrp` system call returns the value of the new process-group ID.

FORTRAN EXTENSIONS

The `setpgrp` system call may be called from Fortran as a function:

```
INTEGER SETPGRP, I
I = SETPGRP ( )
```

EXAMPLES

This example shows how to use the `setpgrp` system call to establish a new process group. (Some system calls in the example are not supported on Cray MPP systems.) The group includes the calling process as well as any of its descendents (in this case, three child processes). As a result of the `setpgrp` request, the new process group ID (PGID) is the process ID (PID) of the calling process.

Typically, a user's processes terminate when the user logs off because all of the user's processes are usually included in the process group of the user's shell process. In contrast, if this program is initiated as a background process and the interactive user logs off from UNICOS, the process and its descendents will not terminate but continue to execute.

```
#include <unistd.h>

main()
{
     int res;

     setpgrp();    /* establish new process group here */

     res = fork();
     if (res == 0) {
          execl("child1", "child1", 0);
          perror("execl for child1 failed");
          exit(1);
     }

     res = fork();
     if (res == 0) {
          execl("child2", "child2", 0);
          perror("execl for child2 failed");
          exit(1);
     }

     res = fork();
     if (res == 0) {
          execl("child3", "child3", 0);
          perror("execl for child3 failed");
          exit(1);
     }

     /* parent program performs its work here */

}
```

FILES

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

SEE ALSO

`exec`(2), `fork`(2), `getpid`(2), `intro`(2), `kill`(2), `signal`(2)

NAME

setportbm, getportbm – Sets or gets the kernel memory port bit map

SYNOPSIS

```
#include <sys/types.h>
#include <sys/sysmacros.h>
int setportbm (unsigned long *bitmap);
int getportbm (unsigned long *bitmap);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setportbm system call copies *bitmap* into the kernel memory port bit map, which reflects the well-known reserved port numbers defined in the /etc/services file.

The getportbm system call gets a copy of the port bit map in the kernel memory.

The setportbm and getportbm system calls accept the following argument:

bitmap Points to the bit map to copy into or from the kernel memory. *bitmap* is an array of unsigned long integers. Its declaration should always be as follows:

```
u_long      bitmap[PORTBITMAX];
```

NOTES

Never use the setportbm and getportbm system calls directly. Only the rsvportbm(8) administrator command should set the kernel memory port bit map, and only the bindresvport(3C) and rresvport(3C) library routines should access the port bit map.

Only a super user or a process with PRIV_ADMIN on a least privilege system can use the setportbm system call.

RETURN VALUES

If setportbm or getportbm 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 setportbm system call fails if one of the following error conditions occurs:

Error Code	Description
EFAULT	Cannot copy the bit map into the kernel memory.

EINVAL	The pointer to the port bit map (*bitmap*) is NULL.
EPERM	The user is not super user.

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

Error Code	Description
EFAULT	Cannot get the bit map from the kernel memory.
EINVAL	The pointer to the port bit map (*bitmap*) is NULL.

EXAMPLES

The following example shows how to use the setportbm and getportbm system calls:

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

main()
{
      u_long      bitmap[PORTBITMAX];

      setportbm(&bitmap[0]);
      getportbm(&bitmap[0]);
}
```

FILES

/etc/services	Contains a list of port numbers

SEE ALSO

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

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

NAME

`setppriv` – Sets the privilege state of the calling process

SYNOPSIS

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

int setppriv (priv_proc_t *buf, int bufsize);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setppriv` system call sets the privilege state of the calling process to the state contained in the buffer. This call returns an error if an attempt is made to modify the state of any privilege that is not permitted for the process. This system call does not set the value of the process privilege text.

The `setppriv` system call accepts the following arguments:

buf Specifies the privilege state to be set to the calling process.

bufsize Specifies the size of the buffer in bytes.

RETURN VALUES

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

If the return value is –1, the privilege state of the calling process is not affected.

ERRORS

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

Error Code	Description
`EFAULT`	The *buf* argument points outside the address space of the process.
`EPERM`	The caller attempted to modify the state of a privilege that did not exist in its permitted privilege set.

SEE ALSO

`getppriv`(2)

NAME

setregid, setegid, setrgid – Sets real or effective group ID

SYNOPSIS

All Cray Research systems:

```
#include <unistd.h>

int setregid (int rgid, int egid);
```

Cray PVP systems:

```
#include <unistd.h>

int setegid (int egid);

int setrgid (int rgid);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setregid` system call sets the real and effective group IDs of the current process to the argument values *rgid* and *egid*, respectively. It accepts the following arguments:

rgid Specifies the real group ID.

egid Specifies the effective group ID.

If *rgid* is –1, the real group ID is not changed; if *egid* is –1, the effective group ID is not changed. The `setegid` call sets the effective group ID of the current process; `setegid(`*egid*`)` is equivalent to the following:

```
setregid(-1, egid)
```

The `setrgid` call sets the real group ID of the current process; `setrgid(`*rgid*`)` is equivalent to the following:

```
setregid(rgid, -1)
```

Processes with appropriate privilege can set their real and effective group IDs to any value. All other processes can change only their effective group ID to their real group ID or their real group ID to their effective group ID.

NOTES

These calls are provided for compatibility reasons; they aid in the porting of code from other systems. Future releases may not support them.

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

Privilege	Description
PRIV_SETGID	The process may set its real and effective group IDs to any specified value.

If the PRIV_SU configuration option is enabled, the super user may set its real and effective group IDs to any specified value.

RETURN VALUES

If the setregid, setegid, or setrgid calls complete successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

If the following condition occurs, the setregid, setegid, or setrgid system call fails.

Error Code	Description
EPERM	The process does not have appropriate privilege to set its real and effective group IDs to the specified values.

EXAMPLES

The setegid request is generally used in *setgid programs*. A setgid program is one that has had its setgid permission bit (octal 2000) set by the chmod(1) command.

When a user executes a setgid program belonging to another group, the effective group ID and saved group ID of the process is set to the group ID of the group owning the program. It is the process's effective group ID that is checked when access to a file is attempted.

Therefore, a user executing another user's setgid program would be allowed to open files belonging to the other user's group for which the user possibly would not be given access permission by the normal access permission bits. While a process's effective group ID is changed to that of another user's group, UNICOS thinks the process belongs to that other group.

The following program has had its setgid permission bit (octal 2000) set by the chmod(1) command. This program shows a common usage of the setegid request.

```
#include <unistd.h>

main()
{
     int gid, egid;

     gid = getgid();
     egid = getegid();

     printf("real group ID of process (before setegid()) is %d\n", gid);
     printf("effective group ID of process (before setegid()) is %d\n", egid);

     /* Open any files that have restricted access here.  That is, this
        program (assuming it can be executed by any user) needs to open files
        belonging to the same group as the owner of this program but those
        files have no general access permission for any other user.  Assuming
        this program is a setgid program, these open(2) requests are permitted
        since the effective group ID of this process has been changed to the
        group ID of the owner of the program. */

     setegid(getgid());  /* for security reasons, set effective group ID to
                            value of real group ID */

     printf("real group ID of process (after setegid()) is %d\n", getgid());
     printf("effective group ID of process (after setegid()) is %d\n",
                                                                  getegid());
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `setregid`, `setegid`, and `setrgid` system calls

SEE ALSO

`getgid`(2), `setgid`(2), `setreuid`(2), `setuid`(2)

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

NAME

setreuid, seteuid, setruid – Sets real or effective user ID

SYNOPSIS

```
#include <unistd.h>
int setreuid (int ruid, int euid);
int seteuid (int euid);
int setruid (int ruid);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `setreuid` system call sets the real and effective user IDs of the current process according to the argument values *ruid* and *euid*, respectively. It accepts the following arguments:

ruid Specifies the real user ID.

euid Specifies the effective user ID.

If *ruid* or *euid* is –1, the real or effective user ID remains unchanged. The `seteuid` call sets the effective user ID of the current process; `seteuid(`*euid*`)` is equivalent to the following:

```
setreuid(-1, euid)
```

The `setruid` call sets the real user ID of the current process; `setruid(`*ruid*`)` is equivalent to the following:

```
setreuid(ruid, -1)
```

Processes with appropriate privilege can set their real and effective user IDs to any value. Any other process is restricted to changing only its effective user ID to either its real user ID or saved user ID.

NOTES

These calls are provided for compatibility reasons; they aid in the porting of code from other systems. Future releases might not support these calls; therefore, use `setuid`(2), which will continue to be supported.

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

Privilege	Description
PRIV_SETUID	The process may set its real and effective user IDs to any specified value.

If the PRIV_SU configuration option is enabled, the super user may set its real and effective group IDs to any specified value.

RETURN VALUES

If the setreuid, seteuid, or setruid call completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

If the following error condition occurs, the setreuid, seteuid, or setruid system call fails.

Error Code	Description
EPERM	The process does not have appropriate privilege to set it real and effective user IDs to the specified values.

BUGS

If NFS block io daemons are running (biod for asynchronous write operations) and the write request is handled by a biod, the write() will appear to succeed. The biod will get an error back, but will be unable to return the error to the user, because it was an asynchronous operation. The server is left with an empty file, and the error is listed in the error return following the close().

EXAMPLES

The seteuid request is generally used in *setuid programs*. A setuid program is one that has had its setuid permission bit (octal 4000) set by the chmod(1) command.

When a user executes a setuid program belonging to another user, the effective ID and saved ID of the process is set to the ID of the user owning the program. It is the process's effective ID that is checked when access to a file is attempted.

Therefore, a user executing another user's setuid program would be allowed to open files belonging to the other user for which the user possibly would not be given access permission by the normal access permission bits. While a process's effective ID is changed to that of another user, UNICOS thinks the process belongs to that other user.

The following program has had its setuid permission bit (octal 4000) set by the chmod(1) command. This program shows common usages of the seteuid request.

```
#include <unistd.h>

main()
{
     int uid, euid;

     uid = getuid();
     euid = geteuid();

     printf("real ID of process (before setuid()) is %d\n", uid);
     printf("effective ID of process (before setuid()) is %d\n", euid);

     /* Open any files that have restricted access here.  That is, this
        program (assuming it can be executed by any user) needs to open files
        belonging to the same user as the owner of this program but those
        files have no general access permission for any other user.  Assuming
        this program is a setuid program, these open(2) requests are permitted
        since the effective Id of this process has been changed to that of the
        owner of the program. */

     seteuid(getuid());  /* for security reasons, set effective ID to value
                            of real ID */

     printf("real ID of process (after setuid()) is %d\n", getuid());
     printf("effective ID of process (after setuid()) is %d\n", geteuid());

     seteuid(euid);      /* set effective ID back to the effective ID the
                            process originally had since another restricted
                            file needs to be opened now */

     /* open the restricted file here */

     seteuid(getuid());   /* for security reasons, set effective ID to
                             value of real ID - will automatically occur
                             when process dies */
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `setreuid`, `seteuid`, and `setruid` system calls

SEE ALSO

`getuid`(2), `setregid`(2), `setuid`(2)

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

NAME

setsid – Creates session and sets process group ID

SYNOPSIS

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

pid_t setsid (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

If the calling process is not a process group leader, the setsid system call creates a new session. The calling process is the session leader of this new session, the process group leader of a new process group, and has no controlling terminal. The process group ID of the calling process is set equal to the process ID of the calling process. The calling process is the only process in the new process group and the only process in the new session.

RETURN VALUES

If setsid completes successfully, it returns the process group ID of the calling process; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The setsid system call fails if the following condition occurs:

Error Code	Description
EPERM	The calling process is already a process group leader, or the process ID of the calling process equals the process group ID of a different process.

FILES

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

SEE ALSO

exec(2), exit(2), fork(2), getpid(2), kill(2), setpgid(2), sigaction(2)

tty(4) *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

setsysv – Sets minimum and maximum level range, authorized compartments, and security auditing options

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysv.h>

int setsysv (struct sysv *buf, int bufsize);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setsysv system call sets the authorized compartments, and minimum and maximum security level range for the UNICOS system.

The setsysv system call accepts the following arguments:

buf Points to a sysv structure in which the security values are stored.

bufsize Specifies the size of the sysv structure in bytes.

The sysv structure includes the following members:

```
short   sy_minlvl;              /* minimum security level */
short   sy_maxlvl;              /* maximum security level */
long    sy_valcmp;              /* authorized compartments */
```

The setsysv system call can be used by a properly privileged process to change the selection of the security audit options. To change the options, the options and the sy_audit_chng flag are set in the sysv structure, which is passed via the *buf* argument.

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_ADMIN	The process is allowed to use this system call.

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

The setsysv system call sets the security boundary conditions (the minimum and maximum security levels, and authorized compartments) for execution within the system.

The setsysv system call does not force termination of tasks initiated at the original system security levels; therefore, the system can still have processes outside of the new level range and authorized compartments.

When the MLS_OBJ_RANGES configuration option is enabled, a check is made to ensure that the new minimum and maximum levels and authorized compartments do not conflict with any of the mounted file system labels. Therefore, it is most effective to use setsysv at system startup, before the file systems are mounted. The file systems of other companies are treated as if they have a security label of a maximum and minimum security level of 0, and no authorized security compartments.

When the setsysv system call is used to change the security auditing options, the new option values are saved into the kernel low memory tables (lowmem.c)

All setsysv requests are recorded in the security log, indicating success or failure.

RETURN VALUES

If setsysv 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 setsysv system call fails if one of the following error conditions occurs:

Error Code	Description
ECOMPV	If the MLS_OBJ_RANGES configuration option is enabled, and the requested authorized compartments are not within the authorized UNICOS system set.
ECOMPV	The requested authorized compartments conflict with those of a mounted file system.
EFAULT	The *buf* argument points outside the process address space.
EINVAL	The *bufsize* argument is less than the size of the sysv structure. If *bufsize* is greater than the size of the sysv structure, *bufsize* is bounded silently by the actual size.
EINVAL	The requested minimum security level is greater than the requested maximum security level.
ESECADM	The process does not have appropriate privilege to use this system call.
ESYSLV	The requested minimum and maximum security level range falls outside the allowable UNICOS system range.
ESYSLV	If the MLS_OBJ_RANGES configuration option is enabled, and the requested minimum and maximum security level range conflicts with that of a mounted file system.

FILES

`/usr/include/sys/param.h`	Defines configuration files
`/usr/include/sys/sysv.h`	Defines structure for system security values
`/usr/include/sys/types.h`	Contains types required by ANSI X3J11

SEE ALSO

`getsysv`(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

setucat – Sets active categories of a process

SYNOPSIS

```
#include <unistd.h>
int setucat (long cat);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setucat system call sets the active categories of the process to the value specified by the category bit mask. The category bit mask is the union of bit values corresponding to each category to be activated. The requested categories must be authorized for the process. A process with appropriate privilege can set its active categories to any value within the authorized category range of the system.

The setucat system call accepts the following argument:

cat Specifies the value of the category bit mask, which is used to set the active categories of the process.

NOTES

All setucat requests are recorded in the security log, indicating success or failure.

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

Privilege	Description
PRIV_MAC_RELABEL_SUBJECT	The process is allowed to set its active categories to any value within the authorized category range of the system.

If the PRIV_SU configuration option is enabled, the super user is allowed to set its active categories to any value within the authorized category range of the system.

RETURN VALUES

If setucat 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 setucat system call fails if one of the following error conditions occurs:

Error Code	Description
EINTCATV	The requested categories are not authorized for use on the UNICOS system.

EINTCATV	The requested categories are not a subset of the caller's authorized categories, and the process does not have appropriate privilege.

FILES

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

SEE ALSO

getusrv(2), setucmp(2), setulvl(2), setusrv(2)

setucat(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

setucmp – Sets active compartments of the process

SYNOPSIS

```
#include <unistd.h>
int setucmp (long cmp);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setucmp system call sets the active compartments of the process to the value specified by the compartment bit mask. The compartment bit mask is the union of bit values corresponding to each compartment to be activated. The *cmp* argument must include all compartments that were active for the process prior to this call.

Each compartment specified by *cmp* must be authorized for the process. A process with appropriate privilege can set its active compartments to any value within the authorized compartment range of the system.

The setucmp system call accepts the following argument:

cmp Specifies the value of the compartment bit mask, which is used to set the active compartments of the process.

NOTES

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

Privilege	Description
PRIV_MAC_RELABEL_SUBJECT	The process is allowed to set its active compartments to any value within the authorized compartment range of the system.
PRIV_MAC_RELABEL_SUBJECT	The process is not restricted to the login shell process.
PRIV_MAC_RELABEL_SUBJECT	The process environment may contain additional background processes.
PRIV_MAC_RELABEL_SUBJECT	The process is allowed to override security compartment access violations with open files.

If the PRIV_SU configuration option is enabled, the super user is allowed to set its active compartments to any value within the authorized compartment range of the system. The super user is not restricted to the login shell process. The super user environment may contain additional background processes. The super user is allowed to override security compartment access violations with open files.

Because of standard I/O buffering, data may be lost when a subject's security label is changed. This occurs if the subject does not have MAC access to the file when the buffer is flushed.

RETURN VALUES

If `setucmp` 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 `setucmp` system call fails if one of the following error conditions occurs:

Error Code	Description
EMANDV	The requested compartments are not authorized for use on the UNICOS system.
EMANDV	The requested compartments are not a subset of the caller's authorized compartments, and the process does not have appropriate privilege.
EMANDV	Activating the requested compartments creates an access violation with existing open files (open character special files owned by the caller are a special case), and the process does not have appropriate privilege.
EMANDV	The request is not issued from the login shell process, and the process does not have appropriate privilege.
EMANDV	There was more than one multitask group in the job (there are background processes), and the process does not have appropriate privilege.
EMANDV	The requested compartment set does not include all compartments that were active prior to this call, and the process does not have appropriate privilege.

FILES

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

SEE ALSO

`getusrv`(2), `setucat`(2), `setulvl`(2), `setusrv`(2)

`setucmp`(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

`setuid`, `setgid` – Sets user or group IDs

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
int setuid (uid_t uid);
int setgid (gid_t gid);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `setuid` system call sets the real user ID, effective user ID, and saved user ID of the calling process; `setgid` sets the real group ID, effective group ID, and saved group ID of the calling process. The `setuid` and `setgid` system calls accept the following arguments:

uid Specifies the real user ID, effective user ID, and saved user ID.

gid Specifies the real group ID, effective group ID, and saved group ID.

The following conditions determine the setting of an ID. They are checked in the order given, and the first condition that is true is the one that applies:

- If the process has appropriate privilege, the real, effective, and saved IDs are all set to *uid* (or *gid*).
- If *uid* is equal to either the real user ID or the saved user ID, the effective user ID is set to *uid*.
- If *gid* is equal to either the real group ID or the saved group ID, the effective group ID is set to *gid*.

NOTES

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

Privilege	Description
`PRIV_SETGID`	The process may set the real group ID, effective group ID, and saved group ID.
`PRIV_SETUID`	The process may set the real user ID, effective user ID, and saved user ID.

If the `PRIV_SU` configuration option is enabled, the super user may set the real, effective, and saved IDs.

RETURN VALUES

If `setuid` or `setgid` 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 `setuid` or `setgid` system call fails if one of the following error conditions occurs:

Error Code	Description
`EINVAL`	The *uid* is out of range.
`EPERM`	The real user or group ID of the calling process is not equal to *uid* or *gid*, and the process does not have appropriate privileges.

FORTRAN EXTENSIONS

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

```
INTEGER uid, SETUID, I
I = SETUID (uid)
```

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

```
INTEGER gid, SETGID, I
I = SETGID (gid)
```

BUGS

If a shell script is made set *uid* or set *gid* and starts with "#!" and the name of the shell to execute the shell script, `exec`(2) in the kernel should execute the shell with the specified effective *gid* or effective *gid*. Instead, `exec`(2) checks the shell for set *uid* and set *gid*, even though the set *uid* and set *gid* of the shell script should take precedence.

EXAMPLES

The `setuid` request is generally used in *setuid programs*. A setuid program is one that has had its setuid permission bit (octal 4000) set by the `chmod`(1) command.

When a user executes a setuid program belonging to another user, the effective ID and saved ID of the process is set to the ID of the user owning the program. It is the process's effective ID that is checked when access to a file is attempted.

Therefore, a user executing another user's setuid program would be allowed to open files belonging to the other user for which the user possibly would not be given access permission by the normal access permission bits. While a process's effective ID is changed to that of another user, UNICOS thinks the process belongs to that other user.

The following program has had its setuid permission bit (octal 4000) set by the chmod(1) command. This program shows common usages of the setuid request. It behaves differently if the owner is a privileged user.

```
#include <unistd.h>

main()
{
     int uid, euid;

     uid = getuid();
     euid = geteuid();

     printf("real ID of process (before setuid()) is %d\n", uid);
     printf("effective ID of process (before setuid()) is %d\n", euid);

     /* Open any files that have restricted access here.  That is, this
        program (assuming it can be executed by any user) needs to open files
        belonging to the same user as the owner of this program but those
        files have no general access permission for any other user.  Assuming
        this program is a setuid program, these open(2) requests are permitted
        since the effective ID of this process has been changed to that of the
        owner of the program. */

     setuid(getuid());  /* for security reasons, set effective ID to value
                           of real ID */

     printf("real ID of process (after setuid()) is %d\n", getuid());
     printf("effective ID of process (after setuid()) is %d\n", geteuid());

     setuid(euid);      /* set effective ID back to the effective ID the
                           process originally had since another restricted
                           file needs to be opened now */

     /* open the restricted file here */

     setuid(getuid());   /* for security reasons, set effective ID to
                            value of real ID - will automatically occur
                            when process dies; call fails if program is
                            owned by a privileged user */
}
```

FILES

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

SEE ALSO

`exec`(2), `getuid`(2), `intro`(2), `setregid`(2), `setreuid`(2)

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

NAME

setulvl – Sets the active security level of the process

SYNOPSIS

```
#include <unistd.h>
int setulvl (int level);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setulvl system call raises the active security level of the calling process. A process with appropriate privilege can raise or lower its active security level to syslow, syshigh, or to any value within the security level range of the system.

The setulvl system call accepts the following argument:

level Specifies the value of the active security level of the calling process. This argument must fall within the authorized security level range of the process.

NOTES

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

Privilege	Description
PRIV_MAC_RELABEL_SUBJECT	The process is allowed to raise or lower its active security level to syshigh, syslow, or to any value within the security level range of the system.
PRIV_MAC_RELABEL_SUBJECT	The process is not restricted to the login shell process.
PRIV_MAC_RELABEL_SUBJECT	The process environment may contain additional background processes.
PRIV_MAC_RELABEL_SUBJECT	The process is allowed to override security level access violations with open files.

If the PRIV_SU configuration option is enabled, the super user is allowed to raise or lower its active security level to syslow, syshigh, or to any value within the security level range of the system. The super user is not restricted to the login shell process. The super user environment may contain additional background processes. The super user is allowed to override security level access violations with open files.

Because of standard I/O buffering, data may be lost when a subject's security label is changed. This occurs if the subject does not have MAC access to the file when the buffer is flushed.

RETURN VALUES

If `setulvl` 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 `setulvl` system call fails if one of the following error conditions occurs:

Error Code	Description
EMANDV	The requested level is not authorized for use on the UNICOS system.
EMANDV	The requested level is not within the caller's authorized security level range, and the process does not have appropriate privilege.
EMANDV	The requested level is less than the current active security level of the process, and the process does not have appropriate privilege.
EMANDV	Changing to the requested level creates an access violation with existing open files (open character special files owned by the caller are a special case), and the process does not have appropriate privilege.
EMANDV	The request is not issued from the login shell process, and the process does not have appropriate privilege.
EMANDV	There was more than one multitask group in job (there are background processes), and the process does not have appropriate privilege.

FILES

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

SEE ALSO

`getusrv`(2), `setucat`(2), `setucmp`(2), `setusrv`(2)

`setulvl`(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

setusrv – Sets security validation attributes of the process

SYNOPSIS

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

int setusrv (struct usrv *buf);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setusrv system call sets security validation attributes for a process.

The setusrv system call accepts the following argument:

buf Points to a usrv structure in which the attribute values are stored.

A usrv structure includes the following members:

```
short   sv_minlvl;      /* minimum security level */
short   sv_maxlvl;      /* maximum security level */
long    sv_valcmp;      /* authorized compartments */
long    sv_savcmp;      /* TFM_EXEC command saved compartments (not used)*/
long    sv_actcmp;      /* active compartments */
short   sv_permit;      /* permissions */
short   sv_actlvl;      /* active security level */
short   sv_savlvl;      /* TFM_EXEC saved security level (not used) */
short   sv_intcls;      /* active integrity class (not used) */
short   sv_maxcls;      /* maximum integrity class (not used) */
long    sv_intcat;      /* active categories */
long    sv_valcat;      /* authorized categories */
struct  {               /* saved integrity parameters over TFM_EXEC
                           (not used) */
        int     actcls  :32;    /* integrity class before TFM_EXEC
                                   (not used) */
        int     actcat  :32;    /* active category before TFM_EXEC
                                   (not used) */
} sv_savint;
int     sv_audit_off    :1;     /* audit on/off flag */
int     sv_audit_chng   :1;     /* audit change flag */
```

A process can use this system call to expand or constrict its active and authorized security attributes. Any process can constrict its authorized security attributes (minimum and maximum security level range, authorized compartments, authorized categories, and so on). Only an appropriately privileged process can expand its authorized security attributes or modify its active security attributes.

A process can enable or disable kernel auditing of its activities by setting the `sv_audit_chg` flag and setting/clearing the `sv_audit_off` flag. Any process can enable kernel auditing for itself. Only an appropriately privileged process can disable kernel auditing of its activities.

NOTES

The login program sets the active security level to the user's default security level with a `setulvl`(2) system call immediately after the `setusrv` call.

All `setusrv` requests are recorded in the security log, indicating success or failure.

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

Privilege	Description
`PRIV_ADMIN`	The process is allowed to change the state of the `usrtrap` permission.
`PRIV_AUDIT_CONTROL`	The process is allowed to disable kernel auditing of its activities.
`PRIV_MAC_RELABEL_SUBJECT`	The process is allowed to expand its authorized security attributes and to set its active security attributes.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to expand its authorized security attributes and to set its active security attributes. A trusted process is allowed to change the state of the `usrtrap` permission. The super user is allowed to disable kernel auditing of its activities.

Because of standard I/O buffering, data may be lost when a subject's security label is changed. This occurs if the subject does not have MAC access to the file when the buffer is flushed.

RETURN VALUES

If `setusrv` 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 `setusrv` system call fails if one of the following error conditions occurs:

Error Code	Description
`ECOMPV`	The requested active compartments are not authorized for the process.
`EFAULT`	The *buf* argument points outside the process address.
`EINTCATV`	The requested authorized categories include the `archive` category.
`EINTCATV`	The requested active categories are not valid for the process.

EINTCLSV	The requested maximum class is not equal to or greater than the authorized minimum class.
EINTCLSV	The requested active class is not within the minimum and maximum classes for this process.
ESYSLV	The requested minimum level is greater than the requested maximum level.
ESYSLV	The requested minimum and maximum level range is not included in the UNICOS system minimum and maximum level range.
ESYSLV	The requested active level is not within the minimum and maximum levels for this process.

Additionally, when called by a process without appropriate privilege, `setusrv` fails if one of the following error conditions occurs:

Error Code	**Description**
ECOMPV	Attempt was made to expand the authorized compartments.
ECOMPV	Attempt was made to change active compartments.
EINTCATV	Attempt was made to expand authorized categories.
EINTCATV	Attempt was made to change active categories.
EINVAL	Attempt was made to expand permissions.
ESYSLV	Attempt was made to expand the authorized security level range.
ESYSLV	Attempt was made to change active security level.

If the requested minimum and maximum security levels are outside those authorized for the UNICOS system, they are set within the bounds of the system.

If the requested valid compartments, categories, or permissions are outside those authorized for the UNICOS system, they are set within the bounds of the system.

If the calling process does not have `suidgid` permission, the file creation mask of the process is set to disallow creation of setuid or setgid files.

If the calling process has no permissions, or has only user permissions, the process is assigned only the user permissions from the requested set. If the calling process has at least one nonuser permission, `setusrv` sets the process' permissions to the requested value.

When called by a process without appropriate privilege, `setusrv` sets the security labels of open character special files (ttys) to the process' active security label.

SEE ALSO

`setulvl`(2), `getusrv`(2)

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

NAME

shmat – Attaches shared memory segment

SYNOPSIS

```
#include <sys/shm.h>
void *shmat (int shmid, void *shmaddr, int shmflg);
```

IMPLEMENTATION

All Cray Research systems. The interface is supported on all platforms, but invocation will return an ENOSYS error for all systems except the CRAY T90 series.

STANDARDS

XPG4

DESCRIPTION

The shmat system call attaches the shared memory segment associated with the shared memory identifier. It accepts the following arguments:

shmid Specifies a shared memory segment.

shmaddr Specifies the address of the shared memory segment.

shmflg Specifies a flag value.

The segment is attached to the address specified by one of the following criteria:

- If *shmaddr* is a null pointer, the segment is attached at the first available address as selected by the system.
- If *shmaddr* is not a null pointer and *shmflg*&SHM_RND is not 0, the segment is attached at the address given by *shmaddr* – (*shmaddr* modulus SHMLBA).
- If *shmaddr* is not a null pointer and *shmflg*&SHM_RND is 0, the segment is attached at the address given by *shmaddr*. *shmaddr* must be aligned (on a MEMKLIK boundary).

The segment is attached for reading if *shmflg*&SHM_RDONLY is not 0 and the calling process has read permission. Otherwise, if *shmflg*&SHM_RDONLY is 0 and the process has read and write permission, the segment is attached for reading and writing.

NOTES

If the user has persistence permission, shared memory segments will remain in the system. If the user does not have persistence permission, and does not explicitly remove segments created, these segments are removed from the system when the session terminates or after the final detach, if attached by processes from another session.

The user must explicitly remove shared memory segments after the last reference to them has been removed.

The alignment requirement, which varies in different machines, is determined by the mapping size of the memory system. (To remain XPG4 compliant, SHMLBA is expressed as a byte value on UNICOS systems. This allows it to be used in expressions passed into shmget(2) to specify a size.)

Processes that have attached shared memory segments cannot be checkpointed or restarted; a checkpoint operation fails with error ESHMA.

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

A process is granted write permission to a shared memory segment only if the active security label of the process is equal to the security label of the shared memory segment, and the process is granted write access by the shared memory segment 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 shared memory segment.
PRIV_MAC_WRITE	The process is considered to meet the security label requirements for being granted write permission to a shared memory segment.
PRIV_DAC_OVERRIDE	The process is considered to meet the permission mode and ACL requirements for being granted read and write permission to a shared memory segment.

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 shared memory segment.

RETURN VALUES

If shmdt completes successfully, the value of the shm_nattch field in the data structure associated with the shared memory ID of the attached shared memory segment is incremented and a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The shmat system call fails and does not attach the shared memory segment if one of the following error conditions occurs:

Error Code	Description
EACCES	Operation permission is denied to the calling process (see ipc(7)).
EINVAL	The *shmid* argument is not a valid shared memory identifier.
EINVAL	The *shmaddr* argument is not equal to 0, and the value of *shmaddr* – (*shmaddr* modulus SHMLBA) is an illegal address for attaching shared memory.
EINVAL	The *shmaddr* argument is not equal to 0, *shmflg*&SHM_RND is equal to 0, and the value of *shmaddr* is an illegal address for attaching shared memory.
EMFILE	The number of shared memory segments attached to the calling process would exceed the system-imposed limit.
ENOMEM	The available data space is not large enough to accommodate the shared memory segment.
ENOSYS	Shared memory operations are permitted only on the CRAY T90 series.

FILES

/usr/include/sys/shm.h	Contains shared memory data structures and macros

SEE ALSO

exec(2), exit(2), fork(2), shmctl(2), shmdt(2), shmget(2)

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

ipc(7) Online only

NAME

shmctl – Provides shared memory control operations

SYNOPSIS

```
#include <sys/shm.h>
int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

IMPLEMENTATION

All Cray Research systems. The interface is supported on all platforms, but invocation will return an ENOSYS error for all systems except the CRAY T90 series.

STANDARDS

XPG4

DESCRIPTION

The shmctl system call provides a variety of shared memory control operations. It accepts the following arguments:

shmid Specifies the shared memory identifier.

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

IPC_STAT Places the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in the include file sys/shm.h (see shm(5)). This command requires read permission.

IPC_SET Sets the value of members of the shmid_ds data structure associated with *shmid*. It sets the value of the following members to the corresponding value found in the structure pointed to by *buf*:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode     /* low-order 9 bits */
```

The IPC_SET command can be executed only by a process that has an effective user ID equal to the value of shm_perm.cuid or shm_perm.uid in the data structure associated with *shmid*.

IPC_RMID	Removes the shared memory identifier specified by *shmid* from the system and destroys the shared memory segment and shmid_ds data structure associated with *shmid*. The IPC_RMID command can be executed only by a process that has an effective user ID equal to the value of shm_perm.cuid or shm_perm.uid in the data structure associated with *shmid*.
IPC_SETACL	Sets the access control list (ACL) on the shared memory identifiers specified by *shmid*. The ipc_perm structure within the shmid_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 shared memory identifier, 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 shared memory identifiers specified by *shmid*.
IPC_GETACL	Retrieves the access control list (ACL) for the shared memory identifier specified by *shmid*. The ipc_perm structure within the shmid_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 entries are returned. If there are less 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 shared memory identifiers specified by *shmid*.
IPC_SETLABEL	Sets the security label on the shared memory identifier specified by *shmid*. The ipc_perm structure within the shmid_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 shared memory identifier. If the shared memory segment is currently attached by any processes, the security label is not altered; a value of –1 is returned and errno is set to EAGAIN. Only a process with the appropriate privilege can perform this operation.
SHM_DCACHE	Disables scalar caching of this segment for this process.
SHM_ECACHE	Enables scalar caching of this segment for this process.
SHM_ICACHE	Invalidates the scalar cache of each CPU currently running a process with the specified segment attached and cached.
SHM_LOCK	Locks the shared memory segment specified by *shmid* in memory. This command can be executed only by a process with the appropriate privilege.
SHM_UNLOCK	Unlocks the shared memory segment specified by *shmid*. This command can be executed only by a process with the appropriate privilege.

buf Points to a structure.

NOTES

If the user has persistence permission, shared memory segments will remain in the system. If the user does not have persistence permission, and does not explicitly remove segments created, these segments are removed from the system when the session terminates or after the final detach, if attached by processes from another session.

The user must explicitly remove shared memory segments after the last reference to them has been removed.

If the kernel list of processes caching each segment becomes corrupted, all processes with that segment attached will be sent the SIGSMCE signal. The default action is termination.

A process is granted read permission to a shared memory identifier only if the active security label of the process is greater than or equal to the security label of the shared memory identifier, and the process is granted read access by the shared memory identifier 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 shared memory identifier.

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 shared memory identifier.
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 shared memory identifier.
PRIV_FOWNER	The process is considered to meet the shared memory identifier ownership requirements for the IPC_SET, IPC_RMID, and IPC_SETACL operations. The process is also permitted to lock and unlock a shared memory segment.

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 shared memory identifier, and is granted read permission to that shared memory identifier. The super-user is also permitted to lock and unlock a shared memory segment.

RETURN VALUES

If `shmctl` 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 `shmctl` system call fails if one of the following error conditions occurs:

Error Code	Description
`EACCES`	The *cmd* argument is equal to `IPC_STAT` and the calling process does not have read permission (see `shm`(5)).
`EACCES`	The *cmd* argument is `IPC_GETACL` and the calling process does not have read permission.
`EAGAIN`	The *cmd* argument is `IPC_SETLABEL` and the shared memory segment is currently attached by one or more processes.
`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 *shmid* argument is not a valid shared memory identifier.
`EINVAL`	The *cmd* argument is not a valid command.
`EINVAL`	The *cmd* argument is `IPC_SET`, and `shm_perm.uid` or `shm_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 *shmid*. • The `ipc_aclcount` field in *buf* is less than 0 or greater than 256. • The ACL supplied failed validation.
`ENOMEM`	The *cmd* argument is equal to `SHM_LOCK` and there is not enough memory.
`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.
`ENOSYS`	Shared memory operations are permitted only on the CRAY T90 series.
`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 process with the appropriate permissions or to the value of `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with *shmid*, and the 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 SHM_LOCK or SHM_UNLOCK, 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/shm.h	Contains shared memory data structures and macros

SEE ALSO

exec(2), exit(2), fork(2), shmat(2), shmget(2), shmdt(2)

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

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

ipc(7) Online only

NAME

shmdt – Detaches shared memory segment

SYNOPSIS

```
#include <sys/shm.h>
int shmdt (void *shmaddr);
```

IMPLEMENTATION

All Cray Research systems. The interface is supported on all platforms, but invocation will return an ENOSYS error for all systems except the CRAY T90 series.

STANDARDS

XPG4

DESCRIPTION

The shmdt system call detaches the shared memory segment from the calling process's address space. It accepts the following argument:

shmaddr Specifies the address of the shared memory segment.

NOTES

If the user has persistence permission, shared memory segments will remain in the system. If the user does not have persistence permission, and does not explicitly remove segments created, these segments are removed from the system when the session terminates or after the final detach, if attached by processes from another session.

The alignment requirement, which varies on different machines, is determined by the mapping size of the memory system.

RETURN VALUES

If shmdt completes successfully, the value of the shm_nattch field in the data structure associated with the shared memory ID of the attached shared memory segment is decremented and a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The shmdt system call fails and does not detach the shared memory segment if one of the following error conditions occurs:

Error Code	Description
EINVAL	The *shmaddr* argument is not the data segment start address of a shared memory segment.

EINVAL	There are outstanding asynchronous I/O operations.
ENOSYS	Shared memory operations are permitted only on the CRAY T90 series.

FILES

/usr/include/sys/shm.h	Contains shared memory data structures and macros

SEE ALSO

exec(2), exit(2), fork(2), shmat(2), shmctl(2), shmget(2)

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

ipc(7) Online only

NAME

shmget – Accesses shared memory identifier

SYNOPSIS

```
#include <sys/shm.h>
int shmget (key_t key, size_t size, int shmflg);
```

IMPLEMENTATION

All Cray Research systems. The interface is supported on all platforms, but invocation will return an ENOSYS error for all systems except the CRAY T90 series.

STANDARDS

XPG4

DESCRIPTION

The shmget system call returns the shared memory identifier associated with *key*. It accepts the following arguments:

key Specifies the shared memory segment.

size Specifies the shared memory segment size in bytes.

shmflg Specifies a flag value.

A shared memory identifier, associated data structure, and shared memory segment of at least *size* bytes (see shm(5)) are created for *key* if one of the following is true:

- *key* is equal to IPC_PRIVATE.
- *key* does not already have a shared memory identifier associated with it, and *shmflg*&IPC_CREAT is not 0.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- shm_perm.cuid, shm_perm.uid, shm_perm.cgid, and shm_perm.gid are set to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of shm_perm.mode are set to the low-order 9 bits of *shmflg*. shm_segsz is set to the value of *size*.
- shm_lpid, shm_nattch, shm_atime, and shm_dtime are set to 0.
- shm_ctime is set to the current time.

NOTES

If the calling process has the ipc_persist permission bit, then the shared memory identifier will be created as a persistent ID. Persistent shared memory identifiers will not be removed from the system unless a shmctl(2) system call with the command IPC_RMID or an ipcrm(1) command is performed on the ID.

If the calling process does not have this permission bit, then the shared memory identifier will be linked into a list of nonpersistent IDs belonging to the session of which the process is a member. When the last process of the session terminates, all the shared memory identifiers linked to the session will be 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 in the preceding list.

The super user is considered to have the ipc_persist permission bit.

RETURN VALUES

If shmget 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 shmget system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A shared memory identifier exists for *key* but operation permission as specified by the low-order 9 bits of *shmflg* would not be granted (see ipc(7)).
EEXIST	A shared memory identifier exists for *key* but both *shmflg*&IPC_CREAT and *shmflg*&IPC_EXCL are not 0.
EINVAL	The *size* argument is less than the system-imposed minimum or greater than the system-imposed maximum.
EINVAL	A shared memory identifier exists for *key*, but the size of the segment associated with it is less than *size* and *size* is not equal to 0.
EMEMLIM	The request would exceed the limits for the session associated with the calling process.
ENOENT	A shared memory identifier does not exist for *key* and *shmflg*&IPC_CREAT is 0.
ENOMEM	A shared memory identifier and associated shared memory segment are to be created, but the amount of available memory is not sufficient to fill the request.
ENOSPC	A shared memory identifier is to be created, but the system-imposed limit on the maximum number of allowed shared memory identifiers system-wide would be exceeded.

ENOSYS	Shared memory operations are permitted only on the CRAY T90 series.

FILES

/usr/include/sys/shm.h	Contains shared memory data structures and macros

SEE ALSO

shmat(2), shmctl(2), shmdt(2)

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

ipc(7) Online only

NAME

shutdown – Shuts down part of a full-duplex connection

SYNOPSIS

```
int shutdown (int s, int how);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The shutdown system call shuts down all or part of a full-duplex connection on the specified socket. It accepts the following arguments:

s Specifies the descriptor for the socket.

how Specifies whether further sends and receives are allowed. The following are valid *how* values:

- 0 Further receives are disallowed.
- 1 Further sends are disallowed.
- 2 Further sends and receives are disallowed.

Unlike the close(2) system call, shutdown can shut down a socket one direction at a time (send or receive). The close(2) system call frees up kernel resources and the socket descriptor, but shutdown does not.

NOTES

If some protocols (such as tcp(4P)) do a shutdown before a close(2), the normal termination of a connection is modified.

If the SOCKET_MAC option is enabled, the active security label of the process must equal the security label of the socket. 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 level and compartment restrictions when the SOCKET_MAC option is enabled.

RETURN VALUES

If `shutdown` 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 `shutdown` 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 not valid.
`EINVAL`	An invalid value was specified for *how*.
`ENOTSOCK`	The *s* descriptor is not a socket.

SEE ALSO

`close`(2), `connect`(2), `socket`(2)

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

NAME

sigaction, sigvec – Examines or changes action associated with a signal

SYNOPSIS

```
#include <signal.h>

int sigaction (int sig, const struct sigaction *act,
struct sigaction *oact);

int sigvec (int sig, struct sigvec *vec, struct sigvec *ovec);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to sigaction)

DESCRIPTION

The sigaction system call allows the calling process to examine or specify (or both) the action to be associated with a specific signal.

The sigaction system call accepts the following arguments:

sig Specifies the signal. See signal(2) for *sig* values.

act or *vec* Specifies the action to be taken when the signal is delivered.

oact or *ovec* Returns the previous signal action.

On Cray MPP systems, the sigaction system call examines or changes the signal action only for the PE on which it is called. It has no effect on any other PE of the application.

The sigaction structure, which describes an action to be taken, is defined in the signal.h header file, and contains the following members:

```
struct sigaction  {
    void (*sa_handler) ();   /* SIG_DFL, SIG_IGN, or pointer to a function */
    sigset_t sa_mask;        /* added to signal mask when in handler */
    int sa_flags;            /* flags to affect behavior of signal */
}
```

If the argument *act* is not null, it points to a structure specifying the action to be associated with the specified signal. If the argument *oact* is not null, the action previously associated with the signal is stored in the location pointed to by the argument *oact*. If *act* is null, signal handling is unchanged by this call; thus the call can be used to inquire about the current handling of a given signal.

The sa_flags field specifies a set of flags used to modify the behavior of the specified signal. It is formed by OR'ing together any of the following values (defined in signal.h):

SA_NOCLDSTOP	If set and if *sig* equals SIGCHLD, *sig* is not sent to the calling process when its children change state due to job control.
SA_RESETHAND	If set, the action associated with *sig* is reset to SIG_DFL on entry to the signal handler (except for the SIGILL, SIGTRAP, and SIGPWR signals).
SA_CLEARMASK	If set, *sig* is cleared from the calling process' signal mask on registration.
SA_CLEARPEND	If set, *sig* is cleared from the set of pending signals on registration.
SA_NODEFER	If set, *sig* is not added to the calling process' signal mask when entering the signal handler.
SA_NOCLDWAIT	If set, children of the calling process do not create zombie processes when they terminate.
SA_WAKEUP	If set, the process is just awakened when *sig* is received and does not enter a signal handler.
SA_REGMTASK	If set, signal registration is performed for all the tasks in a multitasking group; starting in UNICOS 8.0 this is the default behavior. To get the previous behavior, see the SA_REGLWP flag.
SA_REGLWP	If set, signal registration is performed for the current process; this was the default behavior before UNICOS 8.0. However, it is not recommended that applications depend on this behavior since it may not be supported in later releases.

When a signal is caught by a signal-catching function installed by sigaction, a new signal mask is calculated and installed for the duration of the signal-catching function (or until the signal mask is changed explicitly by another system call). This mask is formed by taking the union of the current signal mask and the value of sa_mask for the signal being delivered, and then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested or until one of the exec(2) functions is called.

If sigaction fails, a new signal handler is not installed.

The sigvec system call is provided for 4.3 BSD compatibility. Since the semantics of sigvec are equivalent to those of sigaction (and the sigvec structure has similar members to the sigaction structure), this system call is implemented by calling sigaction with the same arguments as sigvec.

The sigvec structure has the following members:

```
struct sigvec {
    void (*sv_handler) ();   /* signal handler */
    int sv_mask;             /* added to signal mask when in handler */
    int sv_flags;            /* use SA_* flags in sigaction(2) */
}
```

RETURN VALUES

If `sigaction` or `sigvec` 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 `sigaction` or `sigvec` system call fails if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	A `sigaction` (*act* or *oact*) or `sigvec` (*vec* or *ovec*) argument points to an invalid address.
`EINVAL`	The *sig* argument is an illegal signal number, `SIGKILL`, or `SIGSTOP`.

EXAMPLES

This example shows how to use the `sigaction` system call to prepare for the receipt of a signal. In the following program, the `sigaction` request is anticipating receipt of `SIGINT`.

```
#include <signal.h>

main()
{
    void catch(int signo);
    struct sigaction act, oact;

    act.sa_handler = catch;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    sigaction(SIGINT, &act, &oact);

    printf("\nPrevious disposition for signal SIGINT (#%d) = %o",
                                                    SIGINT, oact.sa_handler);
    if (oact.sa_handler == SIG_DFL)
          printf(" (Default)\n");
    else if (oact.sa_handler == SIG_IGN)
          printf(" (Ignored)\n");
    else
          printf("\n");

    /* The process performs its work here fully prepared if a SIGINT
       signal should be delivered to this process - if SIGINT signal
       is sent, process is interrupted and control passes to routine "catch". */
}

void catch(int signo)
{
     /* Code to process a SIGINT signal resides here - function returns to
        the point of interruption when complete. */
}
```

SEE ALSO

`exec`(2), `signal`(2), `sigpending`(2), `sigprocmask`(2), `sigsuspend`(2)

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

NAME

`sigctl` – Provides generalized signal control

SYNOPSIS

```
#include <signal.h>
int sigctl (int action, int sig, void (*func) (int));
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `sigctl` system call, like `signal`(2), allows the calling process to specify what to do upon receipt of a signal.

The `sigctl` system call accepts the following arguments:

action Specifies the action to be taken when the signal is received.

The simplest, and most common, use of `sigctl` is to set *sig* to the desired signal number and set *action* to one of the three bits:

`SCTL_DEF` Takes a system-defined default action.

`SCTL_IGN` Ignores the signal.

`SCTL_REG` Registers to catch the signal.

In this case, *func* contains the address of the signal-catching function or 0. If *func* is set to 0, the process is awakened when the signal occurs, but no signal-catching function is called.

Previously, the following actions provided additional control over the action taken.

```
SCTL_KIL
SCTL_DMP
SCTL_STOP
SCTL_CONT
```

This control is no longer supported; see the NOTES section for more information. The use of these actions is equivalent to specifying `SCTL_DEF`.

sig Specifies a signal. See `signal`(2) for *sig* values.

func Specifies the address of the signal handler if the action is `SCTL_REG`.

The `sigctl` system call provides additional functionality and control beyond that offered by `signal`(2). The two primary differences with signal-catching in `sigctl` are the following:

- Normally, *func* does not revert to `SIG_DFL`; therefore, the process does not need to re-register the signal-catching function.

- Further signal catching is postponed when the signal-catching function is entered (see `sigon`(3C)).

NOTES

With the introduction of the `sigaction`(2) system call in UNICOS 6.0, the `sigctl` system call has become obsolete. While the `sigaction`(2) interface does not provide a superset of the functionality of `sigctl`, the additional functionality that `sigctl` provides is no longer considered necessary. Because of this change, both the UNICOS MAX and UNICOS versions of `sigctl` are written in terms of `sigaction`(2).

The specific additional functionality provided by `sigctl` is the ability to choose an arbitrary action for any signal. For example, set `SIGINT` to terminate with a core dump or `SIGUSR1` to stop the process. In contrast, `sigaction`(2) only allows the user to ignore, catch with a signal handler, or choose a system-defined "default" action for each signal. For example, `SIGINT` always terminates the process by default, and `SIGABRT` always terminates the process and causes a core dump by default.

When written in terms of `sigaction`(2), calls to `sigctl` with `SCTL_KIL`, `SCTL_DMP`, `SCTL_STOP`, or `SCTL_CONT` as the action are mapped to a `sigaction`(2) call with the handler set to `SIG_DFL`. The other characteristics of `sigctl` are handled as before.

Some additional complexity is involved in returning from the signal-catching function to the point at which the process was interrupted. The C library manages this complexity so that users do not need to understand it. To return to the point of interruption, the operating system must be called to restore the last few registers. Nesting is not limited. To return to the previous environment, a special *action* bit, `SCTL_RET`, is used.

To provide the functionality of `signal` efficiently, where signal-catching usually reverts to killing the process or killing with core dump, there is one additional complexity: When registering a signal-catching function, the process may specify a second bit besides `SCTL_REG`. Before entering the signal-catching function, the status for that signal will be set to one of the following signals as requested, and further signal catching is not postponed.

```
SCTL_IGN
SCTL_KIL
SCTL_STOP
SCTL_CONT
SCTL_DMP
```

RETURN VALUES

If `sigctl` completes successfully, it returns the previous *action* for the specified signal *sig*; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

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

Error Code	Description
`EINVAL`	The *sig* argument is an invalid signal number, including `SIGKILL` or `SIGSTOP`. Also, only `SIGCONT` may be set with the *action* `SCTL_CONT`; `SIGCONT` cannot be registered with `SCTL_KIL`, `SCTL_DMP`, or `SCTL_STOP`.

FORTRAN EXTENSIONS

The `sigctl` system call may be called from Fortran through `fsigctl`(3F).

EXAMPLES

The following example shows how to use the `sigctl` system call to prepare for the receipt of a signal. In this program, the `sigctl` request is anticipating receipt of `SIGINT`.

```
#include <signal.h>

main()
{
     void catch(int signo);

     sigctl(SCTL_REG, SIGINT, catch);

     /* The process performs its work here fully prepared
        if a SIGINT signal should be delivered to this process -
        if SIGINT signal is sent, process is interrupted
        and control passes to routine "catch". */
}

void catch(int signo)
{
     /* Code to process a SIGINT signal resides here -
        function returns to the point of interruption
        when complete. */
}
```

SEE ALSO

`kill`(2), `pause`(2), `ptrace`(2), `sigaction`(2), `signal`(2), `wait`(2)

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

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

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

NAME

signal, bsdsignal, sigset, sigignore – Changes action associated with a signal

SYNOPSIS

```
#include <signal.h>
void (*signal (int sig, void (*func) (int))) (int);
void (*bsdsignal (int sig, void (*func) (int))) (int);
void (*sigset (int sig, void (*func) (int))) (int);
int sigignore (int sig);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to signal)

DESCRIPTION

The signal, bsdsignal, sigset, and sigignore system calls allow the calling process to choose the action to be associated with the receipt of a specific signal. All of these calls are implemented in terms of the sigaction(2) system call. The *sig* argument specifies the signal, and the *func* argument specifies the choice (sigignore has no *func* argument; it implicitly ignores the specified signal).

Valid arguments for the signal, bsdsignal, sigset, and sigignore system calls are as follows:

sig Specifies the signal. It can be assigned any one of the signals available on the operating system except SIGKILL or SIGSTOP (which cannot be caught or ignored): These are listed in the following table:

Signal	Number	Default	Description
SIGHUP	1	Exit	Hangup
SIGINT	2	Exit	Interrupt
SIGQUIT	3	Core	Quit
SIGILL	4	Core	Illegal instruction
SIGTRAP	5	Core	Trace trap
SIGABRT	6	Core	Abort
SIGERR	7	Core	Error exit
SIGFPE	8	Core	Floating-point exception
SIGKILL	9	Exit	Kill (cannot be caught or ignored)
SIGPRE	10	Core	Program range error
SIGORE	11	Core	Operand range error

Signal	Number	Default	Description
SIGSYS	12	Core	Bad argument to system call
SIGPIPE	13	Exit	Write on a pipe with no one to read it
SIGALRM	14	Exit	Alarm clock
SIGTERM	15	Exit	Software termination signal from kill
SIGIO	16	Ignore	Input/output possible signal
SIGURG	17	Ignore	Urgent condition on I/O channel
SIGCLD	18	Ignore	Death of a child process
SIGPWR	19	Ignore	Power failure
SIGBUFIO	22	Exit	Reserved for CRI-library use on Cray MPP systems
SIGRECOVERY	23	Ignore	Recovery signal (advisory)
SIGUME	24	Core	Uncorrectable memory error
SIGDLK	25	Core	True deadlock detected (Cray PVP systems)
SIGCPULIM	26	Exit	CPU time limit exceeded (see limit(2))
SIGSHUTDN	27	Ignore	System shutdown imminent (advisory)
SIGSTOP	28	Stop	Sendable stop signal not from a tty (cannot be caught or ignored)
SIGTSTP	29	Stop	Stop signal from a tty
SIGCONT	30	Ignore	Continue a stopped process
SIGTTIN	31	Stop	To reader's process group on background tty read
SIGTTOU	32	Stop	Like SIGTTIN for output, if selected
SIGWINCH	33	Ignore	Window size changes
SIGRPE	34	Exit	Cray PVP register parity error
SIGWRBKPT	35	Core	Write breakpoint (CRAY C90 series only)
SIGNOBDM	36	Core	Cray PVP binary enabled bidirectional memory (cannot be caught or ignored)
SIGAMI	37	Core	CRAY T90 address multiply interrupt
SIGSMCE	38	Exit	Shared memory caching error
SIGINFO	48	Ignore	Information signal (see getinfo(2))
SIGUSR1	49	Exit	User-defined signal 1
SIGUSR2	50	Exit	User-defined signal 2

The following alternative definitions are also available:

```
SIGIOT     6
SIGHWE     6
SIGEMT     7
SIGBUS    10
SIGSEGV   11
SIGCHLD   18
```

Signals 49 through 64 are available for users.

func Specifies the action associated with the signal.

`SIG_DFL` The default actions are outlined in the default column of the signal table. The defaults are as follows:

Exit Upon receipt of the *sig* signal, the receiving process is terminated with all of the consequences outlined in `exit`(2).

Core Upon receipt of the *sig* signal, the receiving process is terminated. A core image is made in the current working directory of the receiving process if the following conditions are met: first, the effective user ID and the real user ID of the receiving process are equal, and second, a file named `core` (or, if extended core file naming is turned on, `core.`*pid*) can be written or created.

Two forms of a core image can be created. The system attempts to create a restart file of the process (see `restart`(1)). If this fails, the system creates a core image that cannot be restarted. Both forms describe the state of the process at the point the signal was received, but the restart file allows the user to continue execution under the control of a debugger.

Stop Upon receipt of the *sig* signal, the receiving process is stopped.

Ignore Upon receipt of the *sig* signal, the receiving process ignores it. This default is identical to the action specified by `SIG_IGN`.

`SIG_IGN` The *sig* signal is ignored.

`SIG_HOLD` (`sigset` only) The specified signal is added to the calling process' signal mask.

function address Upon receipt of the *sig* signal, the receiving process executes the signal-catching function pointed to by *func*. The signal number *sig* is passed as the only argument to the signal-catching function.

Upon return from the signal-catching function, the receiving process resumes execution at the point at which it was interrupted.

When a signal that is to be caught occurs during certain system calls (for example, a `read`(2) or `write`(2) system call on a terminal or pipe), the signal-catching function is executed, and then the interrupted system call returns a −1 to the calling process with `errno` set to `EINTR`.

The `SIGKILL`, `SIGSTOP`, and `SIGNOBDM` signals cannot be caught or ignored; also, these signals cannot be blocked by using `sigset` with the `SIG_HOLD` action.

Whenever a process receives a `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, or `SIGTTOU` signal, regardless of the action associated with it, any pending `SIGCONT` signal is discarded.

Whenever a process receives a SIGCONT signal, regardless of the action associated with it, any pending SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal is discarded. In addition, if the process was stopped, it is continued.

NOTES

The signal system call is compatible with the ANSI C standard, and also follows UNIX System V, Release 3.0 semantics. The bsdsignal system call is compatible with the 4.3 BSD signal system call, and is renamed to avoid conflicts with the ANSI routine. The sigset and sigignore system calls are provided for System V3 compatibility; their use is discouraged as they do not belong to any particular standard.

Differences in the semantics of these system calls are described in the following list:

System Call	Description
signal	When a signal is received (and the action is to execute a signal handler), the action for that signal is reset to SIG_DFL before entering the handler (except for the SIGILL, SIGTRAP, and SIGPWR signals). Also, when a process registers for a signal, all pending signals of that type are cleared.
bsdsignal, sigset	When a signal is received (and the action is to execute a signal handler), the received signal is added to the process' signal mask before entering the handler.
signal, sigset, sigignore	Setting the action for SIGCLD to SIG_IGN causes any child processes of the calling process to not create zombie processes when they terminate (see exit(2)). If the parent process then does a wait(2), the wait blocks until all of the child processes terminate before returning a value of −1 with errno set to ECHILD.
sigset, sigignore	When a process registers for a signal, that signal is cleared from the calling process' signal mask.

Under UNICOS, signal is implemented as a system call, but the signal(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.

RETURN VALUES

If signal, bsdsignal, or sigset completes successfully, the previous signal action (*func*) is returned; otherwise, a value of SIG_ERR is returned (defined in header file signal.h), and errno is set to indicate the error.

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

ERRORS

The `signal`, `bsdsignal`, `sigset`, or `sigignore` system call fails if the following error condition occurs:

Error Code	Description
`EINVAL`	The *sig* argument is an illegal signal number, `SIGKILL`, or `SIGSTOP`.

FORTRAN EXTENSIONS

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

```
INTEGER sig, FSIGNAL, I
EXTERNAL FUNC
I = FSIGNAL(sig, FUNC)
```

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

```
INTEGER sig
EXTERNAL FUNC
CALL FSIGNAL (sig, FUNC)
```

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

EXAMPLES

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

Example 1: This `signal` request prepares for the receipt of a `SIGINT` signal. When using `signal` to catch signals, the programmer needs to remember to re-register to catch the signal in the signal-handling function, because the signal's default disposition is reinstated before entrance to the handler.

```
#include <signal.h>

main()
{
     void catch(int signo);

     signal(SIGINT, catch);

     /* The process performs its work here fully prepared
        if a SIGINT signal should be delivered to this process -
        if SIGINT signal is sent, process is interrupted and
        control passes to routine "catch". */
}

void catch(int signo)
{
     signal(SIGINT, catch);

     /* Code to process a SIGINT signal resides here -
        function returns to the point of interruption
        when complete. */
}
```

Example 2: This `signal` request in conjunction with a `wait`(2) system call causes a process to wait (delay) until all of its child processes have completed:

```
#include <signal.h>

main()
{
     int ret_val, ret_stat;

     signal(SIGCLD, SIG_IGN);

     /* Parent process forks child processes here and
        performs other work - it then wants to wait
        for all of its child processes to terminate. */

     ret_val = wait(&ret_stat);

     /* Parent process proceeds after completion
        of all child processes. */
}
```

SEE ALSO

`exit`(2), `getinfo`(2), `limit`(2), `read`(2), `restart`(2), `sigaction`(2), `sigpending`(2), `sigprocmask`(2), `sigsuspend`(2), `wait`(2), `write`(2)

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

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

NAME

`sigpending` – Stores pending signals

SYNOPSIS

```
#include <signal.h>
int sigpending (sigset_t *set);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `sigpending` system call stores the set of signals that are blocked from delivery and pending for the calling process. It accepts the following argument:

set Points to the space where the set of signals is stored. On Cray MPP systems, the `sigpending` system call stores pending signals only for the PE on which it is called. It has no effect on any other PE of the application.

RETURN VALUES

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

EXAMPLES

The following example shows how to use the `sigpending` system call in a program to determine whether there are any signals currently pending and blocked for this process. If any signals are pending, the program displays the corresponding signal numbers.

```
sigset_t pset;
long i;
int j;

if (sigpending(&pset) == -1) {
     perror("sigpending failed");
     exit(1);
}
printf("sigpending reveals the following signals are pending => ");
printf("%lo\n", pset);
if (pset != 0) {
     printf("   or signals numbered =>  ");
     for (i = 1L, j = 1; i > 0; i <<= 1, j++) {
          if (pset & i) {
               printf("%d  ", j);
          }
     }
     printf("\n");
}
```

SEE ALSO

`sigaction`(2), `signal`(2), `sigprocmask`(2), `sigsuspend`(2)

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

NAME

sigprocmask, sigblock, sigsetmask, sighold, sigrelse – Examines and changes blocked signals

SYNOPSIS

```
#include <signal.h>
int sigprocmask (int how, const sigset_t *set, sigset_t *oset);
int sigblock (int mask);
int sigsetmask (int mask);
mask = sigmask (sig);
int sighold (int sig);
int sigrelse (int sig);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to sigprocmask)

DESCRIPTION

The sigprocmask system call examines or changes (or both) the calling process's signal mask.

how Indicates the manner in which the set is changed. It consists of one of the following values, as defined in the header file signal.h:

- SIG_BLOCK — The resulting set is the union of the current set and the signal set to which *set* points.
- SIG_UNBLOCK — The resulting set is the intersection of the current set and the complement of the signal set to which *set* points.
- SIG_SETMASK — The resulting set is the signal set to which *set* points.

set Points to a set of signals that can be used to change the current signal mask. If the *set* argument is null, it does not point to a set of signals.

oset Points to the space in which the previous mask is stored. If the *oset* argument is null, it does not point to this space. If the value of *set* is null, the value of *how* is not significant and the process signal mask is unchanged by this system call; the call can be used to inquire about currently blocked signals.

mask Specifies a set of signals as a bitmask.

sig Specifies a signal. See signal(2) for *sig* values.

If any pending unblocked signals exist after a call to sigprocmask, at least one of those signals is delivered before sigprocmask returns.

It is not possible to block the SIGKILL and SIGSTOP signals; this is enforced by the system without causing an error to be indicated.

The sigblock and sigsetmask system calls are provided for 4.3 BSD compatibility, and call sigprocmask to actually change the signal mask. The sigblock system call adds the signals specified in *mask* to the calling process's signal mask. The sigsetmask system call sets the calling process's signal mask to the value of *mask*. The sigmask macro creates a signal mask for these system calls; to mask a signal *sig*, use sigmask(*sig*).

The sighold and sigrelse system calls are provided for UNIX System V, Release 3.0, compatibility; they also call sigprocmask to actually change the signal mask. The sighold system call adds the signal *sig* to the calling process's signal mask; sigrelse removes the signal *sig* from the mask.

On Cray MPP systems, the sigprocmask system call examines or changes blocked signals only for the PE on which it is called. It has no effect on any other PE of the application.

RETURN VALUES

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

The sigblock and sigsetmask system calls return the old value of the signal mask.

ERRORS

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

Error Code	Description
EFAULT	The *set* or *oset* argument points to an address that is not valid.
EINVAL	The value of *how* is not equal to one of the defined values.

The sighold and sigrelse system calls fail if the following error condition occurs:

Error Code	Description
EINVAL	The *sig* argument is an illegal signal number, SIGKILL, or SIGSTOP.

EXAMPLES

The following examples illustrate how to use the sigprocmask, sigsetmask, sigblock, and sighold system calls.

Example 1: In this program, the `sigprocmask` request blocks and unblocks signals for a process. In other words, the example shows how bits are added and removed from the process's signal hold mask.

```
#include <signal.h>

#define NULL 0

main()
{
     sigset_t set, oset;

     if (sigprocmask(NULL, NULL, &oset) == -1) {
          perror("sigprocmask failed");
          exit(1);
     }
     printf("\nInitial signal mask = %lo\n", oset);

     sigemptyset(&set);       /* clear the signal set */
     sigaddset(&set, SIGINT);
     sigaddset(&set, SIGFPE);
     sigaddset(&set, SIGUSR1);

     if (sigprocmask(SIG_BLOCK, &set, NULL) == -1) {
          perror("sigprocmask failed");
          exit(2);
     }

     /* Signals SIGINT, SIGFPE, and SIGUSR1 are now blocked
        as well as any other signals blocked prior to the
        sigprocmask request. */

     /* Later, it is needed to unblock one of those signals, SIGFPE. */

     sigdelset(&set, SIGUSR1); /* Modify mask such that SIGFPE */
     sigdelset(&set, SIGINT);  /* can be unblocked */

     if (sigprocmask(SIG_UNBLOCK, &set, NULL) == -1) {
          perror("sigprocmask failed");
          exit(3);
     }
}
```

Example 2: In this program, the `sigsetmask`, `sigblock`, and `sighold` requests manipulate a process's signal hold mask.

```
#include <signal.h>

main()
{
     int ret, mask;

     /* sigsetmask(2) is used to hold signals SIGINT and SIGQUIT -
        all other signal types are not held. */

     mask = sigmask(SIGINT) | sigmask(SIGQUIT);
     printf("initial mask = %lo\n", sigsetmask(mask));

     /* sigblock(2) is used to add signal types SIGFPE and SIGUSR2
        to signal hold mask. */

     mask = sigmask(SIGFPE) | sigmask(SIGUSR2);
     ret = sigblock(mask);
     printf("after sigsetmask, mask = %lo\n", ret);

     ret = sigsetmask(0L);        /* determine current mask */
     printf("after sigblock, mask = %lo\n", ret);
     (void) sigsetmask(ret);      /* restore mask */

     /* sighold(2) is used to add signal type SIGUSR1
        to signal hold mask. */

     (void) sighold(SIGUSR1);
     printf("after sighold, mask = %lo\n", sigsetmask(0L));
}
```

SEE ALSO

`sigaction`(2), `signal`(2), `sigpending`(2), `sigsuspend`(2)

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

NAME

sigsuspend, bsdsigpause, sigpause – Releases blocked signals and waits for interrupt

SYNOPSIS

```
#include <signal.h>
int sigsuspend (const sigset_t *sigmask);
int bsdsigpause (int mask);
int sigpause (int sig);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to sigsuspend)

DESCRIPTION

The sigsuspend system call replaces the process' signal mask with the set of signals pointed to by the *sigmask* argument and then suspends the process until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

The sigsuspend, bsdsigpause, and sigpause system calls accept the following arguments:

sigmask Specifies a set of signal as a bitmask.

mask Specifies a set of signal as a bitmask.

sig Specifies a signal. See signal(2) for *sig* values.

If the action is to terminate the process, sigsuspend does not return. If the action is to execute a signal-catching function, sigsuspend returns after the signal-catching function returns, with the signal mask restored to the setting that existed prior to the sigsuspend call. It is not possible to block the SIGKILL and SIGSTOP signals; this is enforced by the system without indicating an error.

The bsdsigpause system call is equivalent to the 4.3 BSD sigpause system call; it is renamed to avoid conflicts with the UNIX System V, Release 3.0, sigpause system call. It has the same behavior as sigsuspend.

The sigpause system call is provided for UNIX System V, Release 3.0, compatibility. It releases the signal *sig*, and suspends the process until an interrupt occurs.

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

RETURN VALUES

Since `sigsuspend`, `bsdsigpause`, and `sigpause` suspend process execution indefinitely, no successful completion return value exists; instead, a value of –1 is always returned, and `errno` is set to indicate the error.

ERRORS

The `sigsuspend`, `bsdsigpause`, and `sigpause` system calls fail if the following error condition occurs:

Error Code	Description
`EINTR`	A signal is caught by the calling process, and control is returned from the signal-catching function.

EXAMPLES

This example shows how to use the `sigsuspend` system call to wait for a signal to be delivered to a process. In particular, it shows how the `sigsuspend` request suspends the program until the process receives a specific signal (`SIGUSR1`).

```
#include <signal.h>

main()
{
     struct sigaction act;
     sigset_t set;

     act.sa_handler = catch;
     sigemptyset(&act.sa_mask);
     act.sa_flags = 0;
     sigaction(SIGUSR1, &act, NULL);

     sigfillset(&set);          /* turn on (1) all bits in set - */
     sigdelset(&set, SIGUSR1);  /* except SIGUSR1 */

     /* 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. */

     sigsuspend(&set);          /* wait for SIGUSR1 signal */

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

void catch(int signo)
{
     /* process SIGUSR1 signal here */
```

```
}
```

SEE ALSO

pause(2), sigaction(2), signal(2), sigpending(2), sigprocmask(2)

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

NAME

slgentry – Makes security log entry

SYNOPSIS

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

int slgentry (int type, word *entry);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The slgentry system call makes an entry in the /dev/slog security log. The caller defines the type of the entry to be made and passes the address of the entry. The type is decoded and the entry is cast as the proper security structure before the entry is written to the security log.

The slgentry system call accepts the following arguments:

type Defines the type of the entry to be made.

entry Specifies the address of the entry.

The slgentry system call accepts only a specific subset of valid record types. See slrec(5) for more information on these types.

Only an appropriately privileged process can use this system call.

NOTES

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

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

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

RETURN VALUES

If slgentry 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 `slgentry` system call fails if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	The process specified an *entry* where the length is not valid (that is, less than 0 or larger than the largest allowed `slgentry` record).
`EINVAL`	The process specified an *entry* in which some portion of the *entry* is outside the user's address space.
`ESECADM`	The process does not have appropriate privilege to use this system call.

FILES

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

SEE ALSO

`slog`(4), `slrec`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

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

General UNICOS System Administration, Cray Research publication SG–2301

NAME

socket – Creates an endpoint for communication

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int af, int type, int protocol);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The socket system call creates sockets. A socket is an endpoint for communications on a local or a remote host.

The socket system call accepts the following arguments:

af Specifies an address family with which addresses specified in later operations that use the socket should be interpreted.

These families are defined in include file sys/socket.h. The Internet address family (AF_INET), the UNIX address family (AF_UNIX), and the ISO address family (AF_ISO) are the only families currently recognized by UNICOS.

type Specifies the type of socket to be created. Sockets are typed according to their communications properties. Currently defined types are as follows:

SOCK_STREAM Provides sequenced, reliable, two-way, connection-based byte streams. It also provides an auxiliary out-of-band data transmission mechanism.

SOCK_DGRAM Supports datagrams, which are connectionless, unreliable messages of a fixed (typically small) maximum length.

SOCK_RAW Provides access to internal network interfaces. These sockets are available only to the super user.

protocol Specifies a protocol to be used with the socket. (See icmp(4P) for an example.) Usually, only one protocol exists to support a particular socket type, using a given address family. However, many protocols can exist; in which case, you must specify a particular protocol in this argument. The protocol number to use is particular to the communication domain in which communication occurs; see protocols(5).

Sockets of type SOCK_STREAM are full-duplex byte streams. A stream socket must be in a connected state before any data can be sent from or received on it. A connection to another socket is created with a connect(2) or accept(2) call. After the socket is connected, data can be transferred using read(2) and write(2) calls or some variant of the send(2) and recv(2) system calls. When a message is complete, a close(2) call can be performed. *Out-of-band data*, which is data not sent in sequence with other data, can also be transmitted (as described in send(2)) and received (as described in recv(2)).

The communications protocols used to implement a socket of type SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be transmitted successfully within a reasonable length of time, the connection is considered broken, and system calls using the connection return a –1 value and place the ETIMEDOUT code in the global variable errno. The protocols optionally keep sockets active by forcing transmissions every minute in the absence of other activity. If no response can be elicited on an otherwise idle connection for a extended period (for example, 5 minutes), an error is then indicated. If a process sends on a broken stream, a SIGPIPE signal is raised; this causes naive processes, which do not handle the signal, to terminate.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents specified in send(2) calls. It is also possible to receive datagrams at such a socket by using recvfrom (see the recv(2) man page).

You can use an ioctl(2) call to specify a process group to receive a SIGURG signal when out-of-band data arrives.

The socket call sets up send and receive socket buffers (sockbufs) using a protocol-specific sockbuf space limit (see netvar(8)). The socket call fails and returns an ELIMIT error if this call would cause the user's per-session sockbuf space limit to be exceeded.

Socket-level options control the operation of sockets. These options are defined in the sys/socket.h include file and in the following list. Use setsockopt(2) and getsockopt(2) (see getsockopt(2)) to set and to get options, respectively.

The ioctl(2) system call performs a variety of functions at different levels (socket, interface, and routing). The following is a list of command names for the ioctl(2) system call and a description of the function performed by each command.

Command	**Description**
Socket level:	
FIOASYNC	Sets and clears asynchronous input/output by using SIGIO.
FIONBIO	Sets and clears nonblocking input/output.
FIONREAD	Gets number of bytes available for reading from socket.
SIOCATMARK	Indicates whether any out-of-band data is waiting in the socket (0 means yes; otherwise, no).
SIOCGPGRP	Gets process group to receive SIGIO and SIGURG for this socket.
SIOCSPGRP	Sets process group to receive SIGIO and SIGURG for this socket.

Interface level:

SIOCGIFADDR	Gets network address of interface into the ifr_addr member to which the data argument points.
SIOCGIFCONF	Returns interface configuration information into the ifconf structure to which the ioctl data argument points.
SIOCGIFDSTADDR	Specifies address of the remote host on a point-to-point link in the ifr_addr member of the ifreq structure to which the data argument points.
SIOCGIFFLAGS	Returns interface flags in the ifr_flags member of the ifreq structure to which the data argument points.
SIOCSIFADDR	Sets network address of interface from the ifr_addr member of the ifreq structure to which the data argument points. Also initializes a routing table entry for the interface (must be root).
SIOCSIFDSTADDR	Sets network address of the remote node on a point-to-point link from the ifr_addr member of the ifreq structure to which the data argument (must be root) points.
SIOCSIFFLAGS	Sets interface flags from the ifr_flags member of the ifreq structure to which the data argument (must be root) points. Flag values are as follows:

```
IFF_UP          0x1     /* interface is up */
IFF_DEBUG       0x4     /* turn on debugging */
IFF_POINTOPOINT 0x10    /* interface is point-to-point link */
IFF_NOTRAILERS  0x20    /* avoid use of trailers */
IFF_NOARP       0x80    /* no address resolution protocol */
```

Network media sublevel:

HYSETROUTE	Sets HYPERchannel routing table.
HYGETROUTE	Gets HYPERchannel routing table.
HYSETTYPE	Sets interface type.
HYGETTYPE	Gets interface type.

NOTES

The socket is assigned the active security label of the process.

RETURN VALUES

If socket completes successfully, a descriptor that references the socket is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

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

Error Code	Description
`EMFILE`	Either the per-process descriptor table is full, or the system file table is full.
`ELIMIT`	The user's socket buffer space limit is exceeded.
`ENOBUFS`	Buffer space is not available. The socket cannot be created.
`EPERM`	Permission denied for operation.
`EPROTONOSUPPORT`	The specified protocol is not supported.

EXAMPLES

Because the `socket` system call is used in both client and server programs along with other networking calls, the following examples are simple client and server programs that illustrate how to use the `socket` request.

Example 1: The client program creates a TCP/IP socket and then attempts to establish a connection between the newly created socket and the socket within the server program on the designated server host. If a connection is successful, the client process sends a string of data to the server process.

```
/*  Client side of client/server socket example.
    Syntax: client hostname portnumber */

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

/*      in in.h is this socket structure
 *
 *       Socket address, internet style.
 *
 *       struct sockaddr_in {
 *          short   sin_family;
 *          u_short sin_port;
 *          struct  in_addr sin_addr;
 *          char    sin_zero[8];
 *      };
 */

#define DATA "Test message from client to server."

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

```
        int s;
        struct sockaddr_in dest;       /* destination socket address */
        struct hostent *hp;            /* host structure pointer */

    /* Convert host name into network address */
        hp = gethostbyname(argv[1]);
        bzero((char*)&dest, sizeof(sockaddr_in));
        dest.sin_family = hp->h_addrtype;  /* addr type (AF_INET) */
        bcopy(hp->h_addr_list[0], &dest.sin_addr, hp->h_length);
        dest.sin_port = atoi(argv[2]);


        /* create port */

        if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
             perror("client, cannot open socket");
             exit(1);
        }
        if (connect (s, (struct sockaddr *) &dest, sizeof(dest)) < 0) {
             close(s);
             perror("client, connect failed");
             exit(1);
        }
        write(s, DATA, sizeof(DATA));

        close(s);
        exit(0);
    }
```

Example 2: (Some system calls in this example are not supported on Cray MPP systems.) The server program creates a TCP/IP socket, waits for a client process from some host to attempt a connection, accepts the connection, and then 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.
    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/net/route.h`	Route file that contains the `rtentry` structure
`/usr/include/sys/socket.h`	Defines the address families
`/usr/include/sys/types.h`	Defines types of sockets

SEE ALSO

`accept`(2), `bind`(2), `close`(2), `connect`(2), `getsockname`(2), `getsockopt`(2), `ioctl`(2), `listen`(2), `read`(2), `recv`(2), `select`(2), `send`(2), `shutdown`(2), `socketpair`(2), `write`(2)

`icmp`(4P), `protocols`(5), `services`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

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

NAME

socketpair – Creates a pair of connected sockets

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int socketpair (int af, int type, int protocol, int sv[2]);
```

IMPLEMENTATION

All Cray Research systems

Implemented only for the UNIX address domain (AF_UNIX)

DESCRIPTION

The socketpair system call was designed to simulate the UNIX pipe mechanism with the use of sockets. The call is very similar to the socket(2) system call. With standard network operations, sockets are created individually using socket(2). To simulate pipe operations, the calling process must create both endpoints for the communication simultaneously. socketpair creates a pair of sockets in one request to the operating system.

The socketpair system call accepts the same *af*, *type*, and *protocol* arguments as socket(2) does. For descriptions of these arguments, see the socket(2) man page. In addition, socketpair accepts the following argument:

sv Specifies an array of two integers (*sv*[0] and *sv*[1]) that receive the descriptors for the new pair of sockets.

NOTES

The socket is assigned the active security label of the process.

RETURN VALUES

If socketpair 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 socketpair system call fails if one of the following error conditions occurs:

Error Code	Description
EAFNOSUPPORT	This machine does not support the specified address family.
EFAULT	The *sv* address does not specify a valid part of the process address space.
EMFILE	Too many descriptors are in use by this process.

EOPNOTSUPP	The specified protocol does not support creation of socket pairs.
EPROTONOSUPPORT	This machine does not support the specified protocol.

FILES

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

SEE ALSO

pipe(2), read(2), write(2)

NAME

ssbreak – Changes size of secondary data segment

SYNOPSIS

```
#include <unistd.h>
int ssbreak (long incr);
```

IMPLEMENTATION

All Cray Research systems except CRAY J90 series and CRAY EL series

DESCRIPTION

The ssbreak system call increases or decreases the size of the secondary data segment (SDS), which is allocated from an area of the SSD solid-state storage device reserved for this purpose at configuration time (see ssd(4)). It accepts the following argument:

incr Specifies a count of 4096-byte blocks allocated and deallocated in the SSD.

The ssbreak system call changes secondary storage size, as follows:

- If the ssbreak system call has a positive *incr* argument, the amount of secondary storage increases by a multiple of the number of blocks that is defined as the unit allocation size. The unit allocation size is the number of 4096-byte blocks that compose the smallest amount of space that a user process can allocate in the SDS; UNICOS is delivered with a unit allocation size of 128 blocks. The unit allocation size can be found as SDS_WGHT in /usr/include/sys/ssd.h. If the *incr* argument is less than or equal to the number of blocks in one unit, the ssbreak system call allocates one whole unit of secondary storage. If the *incr* is larger than the number of blocks in one unit and less than or equal to twice the number of blocks in one unit, two units of secondary storage are allocated, and so on. Because the unit for secondary storage usually is greater than one block, the amount of storage allocated can be greater than that requested.
- If the ssbreak system call has a negative *incr* argument, it deallocates secondary storage only in multiples of one unit. The *incr* argument must be larger than or equal to the number of blocks in one unit for deallocation to result.
- If the ssbreak system call has an *incr* argument of 0, the size of secondary storage remains the same.

Secondary storage allocated by the ssbreak system call is freed on exit of the program.

CAUTIONS

The ssbreak system call works only on a system with an SSD in its configuration, and a secondary data segment (SDS) area must be configured as a slice of the SSD.

Using ssbreak directly interferes with the operation of sdsalloc(3F), which manages the SDS space within a process; sdsalloc(3F) should be used instead of ssbreak. CRI strongly discourages direct use of ssbreak in user programs.

RETURN VALUES

If `ssbreak` completes successfully, the current amount of secondary data storage in blocks is returned to the caller; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

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

Error Code	Description
`ENOMEM`	The request requires more secondary storage than can be satisfied.

FORTRAN EXTENSIONS

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

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

The `ssbreak` system call should not be used in a Fortran program that accesses SDS through the `assign`(1) command or auxiliary arrays because the libraries use `sdsalloc`(3F) to control SDS allocation. Using `ssbreak` from Fortran directly conflicts with the SDS management that `sdsalloc`(3F) provides.

EXAMPLES

The following example shows how to use the `ssbreak` system call to request a SDS allocation. In this case, the programmer asks for an area of 10 blocks for the process. Because SDS segment space is allocated in units called the unit allocation size, which is typically configured at 128 blocks, this `ssbreak` request actually allocates the process 128 blocks of SDS space.

```
int size;

if ((size = ssbreak(10L)) == -1) {
     perror("sds allocation error");
     exit(1);
}
else {
     printf("The size of the sds is now %d - 4096-byte blocks\n", size);
}

/* To make use of the allocated SDS area, the program next issues
   ssread and sswrite requests. */

/* When usage of the allocated SDS area is complete, the program
   releases its SDS allocation using ssbreak with negative argument -
   process termination also releases SDS space. */

if ((size = ssbreak(-size)) == -1) {
     perror("sds deallocation error");
     exit(1);
}
else {
     printf("The size of the sds is now %d - 4096-byte blocks\n", size);
}
```

FILES

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

SEE ALSO

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

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

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

NAME

ssread, sswrite – Reads or writes to secondary data segment

SYNOPSIS

```
#include <unistd.h>
int ssread (long pds, long sds, long count);
int sswrite (long pds, long sds, long count);
```

IMPLEMENTATION

All Cray Research systems except CRAY J90 series and CRAY EL series

DESCRIPTION

The ssread system call moves data from a secondary data area reserved with the ssbreak(2) system call to a buffer. The sswrite system call moves data from a buffer to a secondary data area.

The ssread system call accepts the following arguments:

pds Specifies a word-aligned address of a buffer.

sds Specifies the secondary data area offset. This is a 4096-byte sector offset in the process' secondary data area. It is specified numerically, with 0 giving the beginning block, 1 giving the second block, and so on. It must be allocated with ssbreak(2).

count Specifies the number of 4096-byte blocks to be moved.

CAUTIONS

The ssread and sswrite system calls work only on a system with an SSD solid-state storage device in its configuration, and a secondary data segment area (SDS) must be configured as a slice of the SSD.

RETURN VALUES

If ssread or sswrite 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 ssread and sswrite system calls fail if one of the following error conditions occurs:

Error Code	Description
EFAULT	The request exceeds the boundaries of either the buffer or the secondary data area.
EIO	An error occurred during the data transfer.

FORTRAN EXTENSIONS

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

```
INTEGER pds (512*n), sds, words, SSREAD, I
I = SSREAD (pds, sds, words)
```

The third argument to the Fortran interface to `SSREAD` specifies the number of words to be read. This is different than the third argument to the system call. The `sswrite` system call can be called from Fortran as a function:

```
INTEGER pds (512*n), sds, words, SSWRITE, I
I = SSWRITE (pds, sds, words)
```

The third argument to the Fortran interface to `SSWRITE` specifies the number of words to be written. This is different than the third argument to the system call.

EXAMPLES

The following example shows how to use the `ssread` system call in conjunction with other system calls to transfer data to and from the SDS allocation for a process. In this portion of the program, the `ssbreak`(2) request asks for an SDS area of 1000 blocks, and the `sswrite` request then transfers 1000 blocks of data from the user's process memory space to the process's SDS allocation. Lastly, `ssread` reads the 1000 blocks of data back into the user's process memory from the SDS allocation.

```
int size;
char buff[4096 * 1000];

if ((size = ssbreak(1000L)) == -1) {
  perror("sds allocation error");
  exit(1);
}
else {
  printf("The size of the sds is now %d - 4096-byte blocks\n", size);
}

/* SDS write illustration */

if (sswrite( (long) buff, 0L, 1000L) == -1) {
  perror("sswrite error");
  exit(2);
}

/* SDS read illustration */

if (ssread( (long) buff, 0L, 1000L) == -1) {
  perror("ssread error");
  exit(3);
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `ssread` and `sswrite` system calls

SEE ALSO

`ssbreak`(2)

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

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

NAME

stat, lstat, fstat – Gets file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (const char *path, struct stat *buf);
int lstat (const char *path, struct stat *buf);
int fstat (int fildes, struct stat *buf);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to stat, fstat)

DESCRIPTION

The stat system call obtains information about the file specified by *path*. All directories in the path name leading to the file must be searchable, although it is not necessary to have read, write, or execute permission to the file.

The lstat system call is similar to stat, except when the specified file is a symbolic link. In this case, lstat returns information about the link, and stat returns information about the file that the link references.

The fstat system call obtains the same information as stat about a specified open file. When using fstat on a file descriptor returned from the accept(2), socket(2), or socketpair(2) system call, only the st_uid, st_gid, st_slevel, st_blksize, st_oblksize, and the type portion of st_mode fields are meaningful. All other fields will be 0.

The stat, lstat, and fstat system calls accept the following arguments:

path Points to a file's path name (stat and lstat only). Read, write, or execute permission of the specified file is not required, but all directories listed in the path name leading to the file must be searchable.

buf Points to the stat structure.

fildes Specifies a file descriptor. It is obtained from a successful accept(2), creat(2), dup(2), fcntl(2), open(2), pipe(2), socket(2), or socketpair(2) system call (fstat only).

A `stat` structure includes the following members:

```
mode_t   st_mode;      /* File mode; see mknod(2). */
ino_t    st_ino;       /* Inode number for this file */
dev_t    st_dev;       /* Device on which this file resides */
dev_t    st_rdev;      /* Device ID; this entry is defined only */
                       /* for character special or block special files. */
nlink_t  st_nlink;     /* Number of links */
uid_t    st_uid;       /* User ID of the file's owner */
gid_t    st_gid;       /* Group ID of the file's group */
int      st_acid;      /* Account id of the file */
off_t    st_size;      /* File size in bytes */
time_t   st_atime;     /* Time that file data was last accessed; */
                       /* changed by system calls creat(2), mknod(2), */
                       /* pipe(2), utime(2), and read(2). */
time_t   st_mtime;     /* Time when data was last modified; */
                       /* changed by system calls creat(2), mknod(2), */
                       /* pipe (2), utime(2), and write(2). */
time_t   st_ctime;     /* Time when file status was last changed; */
                       /* times measured in seconds since 00:00:00 */
                       /* GMT, January 1, 1970.  Changed by */
                       /* system calls chmod(2), chown(2), creat(2), */
                       /* link(2), mknod(2), pipe(2), unlink(2), utime(2), */
                       /* and write(2). */
int      st_count;     /* Reference count from inode; number of active */
                       /* file table entries */
int      st_blocks;    /* Number of 4096 byte blocks allocated to the file */
unsigned int
         st_msref:1,   /* Modification signature referenced flag */
         st_ms:31,     /* Modification signature */
         st_gen;       /* Inode generation number */
int      st_param[8];  /* Device parameter words; this entry is defined only */
                       /* for character special or block special files */
ushort   st_dm_mode;   /* Actual file mode when migrated */
long     st_dm_status; /* Migrated file status flags */
long     st_dm_mid;    /* Migrated file machine id */
long     st_dm_key;    /* Migrated file key */
unsigned int
         st_hasacl:l,  /* File has an ACL */
         st_hascomps:l;/* File has compartments */

short    st_slevel;    /* File level */
short    st_secflg;    /* Security flags (not used) */
short    st_intcls;    /* Integrity class (not used) */
short    st_intcat;    /* Integrity category (not used) */

long     st_site;      /* Site field from inode */
long     st_allocf;    /* Allocation control flags; see fcntl(2) */
```

The file type S_IFREG is returned in st_mode if an IFOFL (offline file) is encountered. The actual file type S_IFOFL is returned in st_dm_mode.

NOTES

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 (stat/lstat system calls only).

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 (stat/lstat system calls only).
PRIV_MAC_READ	The process is granted search permission to a component of the path prefix via the security label (stat/lstat 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 (stat/lstat system calls only) and is granted read permission to the file via the security label.

RETURN VALUES

If stat, lstat, or fstat 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 stat or lstat 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.
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 *buf* or *path* argument points to an address that is not valid.
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.

The `fstat` 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 to an address that is not valid.

FORTRAN EXTENSIONS

The `stat` 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 buf(m), STAT, I
I = STAT (path, buf)
```

Alternatively, `stat` 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*n path
INTEGER buf(m)
CALL STAT (path, buf)
```

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

EXAMPLES

The following examples illustrate how to use the `stat` and `lstat` system calls.

Example 1: This example shows the simplest form of the `stat` system call. The following `stat` request provides status information for a file whose path name is supplied as an argument.

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

main(int argc, char *argv[])
{
     struct stat buf;

     if (stat(argv[1], &buf) == -1) {
          perror("stat failed");
          exit(1);
     }

     /* Data from the specified file's inode now available in buf. */
}
```

Example 2: This example shows how to use the `lstat`, `readlink`(2), and `stat` requests. It uses the list of file names supplied as arguments to produce a display listing each file name along with the size of each file. If any file in the argument list is a symbolic link, the program also displays the path name of the file that is the target of the link as well as that file's size. For a definition of `S_IFLNK`, see the `sys/stat.h` file.

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

main(int argc, char *argv[])
{
     char file[50], tfile[50];
     struct stat buf;
     int i;

     for (i = 1; i < argc; i++) {
          strcpy(file, argv[i]);
          if (lstat(argv[i], &buf) == -1) {
               perror("lstat failed");
               continue;
          }
          if ((buf.st_mode & S_IFLNK) == S_IFLNK) {    /* a symbolic link? */
               readlink(argv[i], tfile, 50);
               if (stat(tfile, &buf) == -1) {
                    perror("stat failed");
                    exit(1);
               }
               strcat(file, "->");
               strcat(file, tfile);
          }
          printf("%-50s   %d\n", file, buf.st_size);
     }
}
```

SEE ALSO

`accept`(2), `chmod`(2), `chown`(2), `creat`(2), `dmofrq`(2), `dup`(2), `fcntl`(2), `link`(2), `mknod`(2), `open`(2), `pipe`(2), `readlink`(2), `socket`(2), `socketpair`(2), `time`(2), `unlink`(2)

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

NAME

statfs, fstatfs – Gets file system information

SYNOPSIS

```
#include <sys/statfs.h>
int statfs (char *path, struct statfs *buf, int len, int fstyp);
int fstatfs (int fildes, struct statfs *buf, int len, int fstyp);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `statfs` system call returns a *generic superblock* describing a file system. It can be used to acquire information about mounted and unmounted file systems, and usage is slightly different in the two cases. The `statfs` and `fstatfs` system calls accept the following arguments:

path Specifies path to a file system.

buf Points to a structure. It will be filled by the system call, as described in the following text.

len Specifies the number of bytes of information that the system should return in the structure.

The *len* argument must be no greater than `sizeof (struct statfs)`, and ordinarily it contains exactly that value; if it holds a smaller value, the system fills the structure with that number of bytes. (This allows future versions of the system to grow the structure without invalidating older binary programs.)

fstyp Specifies file system type.

fildes Specifies open file descriptor.

If the file system of interest is currently mounted, *path* must specify a file that resides on that file system. In this case, the file system type is known to the operating system, and the *fstyp* argument must be 0. For an unmounted file system, *path* must specify the block special file containing it, and *fstyp* must contain the nonzero file system type. In both cases, read, write, or execute permission of the specified file is not required, but all directories listed in the path name leading to the file must be searchable.

The `statfs` structure to which *buf* points includes the following members:

```
short   f_fstyp;      /* File system type */
long    f_bsize;      /* Block size */
long    f_frsize;     /* Fragment size */
long    f_blocks;     /* Total number of blocks */
long    f_bfree;      /* Count of free blocks */
long    f_files;      /* Total number of file nodes */
long    f_ffree;      /* Count of free file nodes */
char    f_fname[6];   /* Volume name */
char    f_fpack[6];   /* Pack name */
long    f_priparts;   /* Bitmap of primary partitions */
long    f_secparts;   /* Bitmap of secondary partitions */
long    f_npart;      /* Number of partitions (logical drives) in FS*/
long    f_bigsize;    /* Block to "bigunit" allocation crossover */
long    f_bigunit;    /* Allocation size for large files */
long    f_prinblks;   /* Total number of 512 wd blocks in primary */
long    f_prinfree;   /* Number of free 512 wd blocks in primary */
long    f_priaunit;   /* Size of primary area allocation unit */
long    f_secnblks;   /* Total number of 512 wd blocks in secondary */
long    f_secnfree;   /* Number of free 512 wd blocks in secondary */
long    f_secaunit;   /* Size of secondary area allocation unit */
```

The `fstatfs` system call is similar, except that the file specified by *path* in `statfs` is identified instead by an open file descriptor, *fildes*, obtained from a successful `creat`(2), `dup`(2), `fcntl`(2), `open`(2), or `pipe`(2) system call.

If *fstyp* indicates the network file system (NFS) file system, the fields of the structure have the following meanings:

`f_frsize`	0
`f_bsize`	Block size on a remote system
`f_blocks`	Number of blocks on remote file system
`f_bfree`	Free blocks on remote file system
`f_files`	0
`f_ffree`	0
`f_fname`	Name of host in which remote file system resides

NOTES

The `statfs` system call obsoletes the `ustat`(2) system call for most purposes.

To be granted search permission to a component of the path prefix (for the `statfs` 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 (`statfs` system call only).
`PRIV_MAC_READ`	The process is granted search permission to a component of the path prefix via the security label (`statfs` system call only).

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix.

RETURN VALUES

If `statfs` 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 `statfs` or the `fstatfs` 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.
`EBADF`	The *fildes* argument is not a valid open file descriptor.
`EBADF`	The calling process does not have MAC read access to the file to which the file descriptor refers.
`EFAULT`	The *buf* or *path* argument points to an address that is not valid.
`EINVAL`	The *fstyp* argument is an not a valid file system type; the *path* argument is not a block special file, and the *fstyp* argument is nonzero; the *len* argument is negative or is greater than `sizeof` (`struct statfs`).
`ENOENT`	The specified file does not exist.
`ENOTDIR`	A component of the path prefix is not a directory.

EXAMPLES

This example shows how to use the `statfs` and `sysfs` system calls to obtain file system information. The `statfs` request retrieves information for the file system containing the file whose name is passed as an argument. The `sysfs` system call converts the numerical file system type to a character-string format before displaying it.

```
#include <sys/types.h>
#include <sys/statfs.h>
#include <sys/fstyp.h>
#include <sys/fsid.h>

main(int argc, char *argv[])
{
     struct statfs stats;
     char buf[FSTYPSZ];

     if (statfs(argv[1], &stats, sizeof(struct statfs), 0) == -1) {
          perror("statfs error");
          exit(1);
     }

     if (sysfs(GETFSTYP, stats.f_fstyp, buf) == -1) {
          perror("sysfs (GETFSTYP) error");
          exit(1);
     }

     printf("File system type = %s\n", buf);
     printf("Block size = %d\n", stats.f_bsize);
     printf("Fragment size = %d\n", stats.f_frsize);
     printf("Total number of blocks on file system = %d\n", stats.f_blocks);
     printf("Total number of free blocks = %d\n", stats.f_bfree);
     printf("Total number of file nodes (inodes) = %d\n", stats.f_files);
     printf("Total number of free file nodes = %d\n", stats.f_ffree);
     printf("Volume name = %s\n", stats.f_fname);
     printf("Pack name = %s\n", stats.f_fpack);
     printf("Primary partition bit map = %o\n", stats.f_priparts);
     printf("Secondary partition bit map = %o\n", stats.f_secparts);
     printf("Number of partitions = %d\n", stats.f_npart);
     printf("Big file threshold = %d bytes ", stats.f_bigsize);
     printf("or %d blocks\n", stats.f_bigsize/stats.f_bsize);
     printf("Big file allocation unit size = %d bytes ", stats.f_bigunit);
     printf("or %d blocks\n", stats.f_bigunit/stats.f_bsize);
     printf("Number of blocks in primary partitions = %d\n", stats.f_prinblks);
     printf("Number of free blocks in primary partitions = %d\n",
                                                          stats.f_prinfree);
     printf("Primary partition allocation unit size = %d ", stats.f_priaunit);
     printf("bytes or %d blocks\n", stats.f_priaunit/stats.f_bsize);
     printf("Number of blocks in secondary partitions = %d\n",stats.f_secnblks);
     printf("Number of free blocks in secondary partitions = %d\n",
                                                          stats.f_secnfree);
     printf("Secondary partition allocation unit size = %d ",stats.f_secaunit);
     printf("bytes or %d blocks\n", stats.f_secaunit/stats.f_bsize);
}
```

SEE ALSO

chmod(2), chown(2), creat(2), dup(2), fcntl(2), link(2), mknod(2), open(2), pipe(2), read(2), time(2), unlink(2), ustat(2), utime(2), write(2)

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

NAME

`statvfs`, `fstatvfs` – Gets file system information

SYNOPSIS

```
#include <sys/statvfs.h>
int statvfs (const char *path, struct statvfs *buf);
int fstatvfs (int fildes, struct statvfs *buf);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The `statvfs` system call obtains information about the file specified by *path*. All directories in the path name leading to the file must be searchable, although it is not necessary to have read, write, or execute permission to the file.

The `fstatvfs` system call obtains the same information as `statvfs` about the file referenced by *fildes*

The `statvfs` and `fstatvfs` system calls accept the following arguments:

path Points to a file's path name (`statvfs` only).

buf Points to the `statvfs` structure.

fildes Specifies a file descriptor (`fstatvfs` only).

The following flags can be returned in the `f_flag` member:

ST_RDONLY read-only file system

ST_NOSUID setuid/setgid bits ignored by exec

ST_NOTRUNC does not truncate long file names

NOTES

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 (`statvfs` system calls only).

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 (`statvfs` system calls only).
PRIV_MAC_READ	The process is granted search permission to a component of the path prefix via the security label (`statvfs` 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 (`statvfs` system calls only) and is granted read permission to the file via the security label.

RETURN VALUES

If `statvfs` or `fstatvfs` 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 `statvfs` or `fstatvfs` system call fails if one of the following error conditions occurs:

Error Code	Description
EIO	An I/O error occurred while reading the file system.
EINTR	A signal was caught during execution of the function.

The `statvfs` 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.
ELOOP	Too many symbolic links were encountered in resolving the path.
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.

The `fstatvfs` system call fails if 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.

FILES

sys/statvfs.h

SEE ALSO

chmod(2), chown(2), creat(2), dup(2), exec(2), fcntl(2), link(2), mknod(2), open(2), pipe(2), read(2), time(2), unlink(2) utime(2) write(2)

NAME

stime – Sets time

SYNOPSIS

```
#include <unistd.h>
int stime (long *tp);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The stime system call sets the system time and date.

tp Points to the value of time as measured in seconds from 00:00:00 Greenwich mean time (GMT), January 1, 1970. 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_TIME	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_SYSPARAM permbit is allowed to use this system call.

RETURN VALUES

If stime 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 stime system call fails if the following error condition occurs:

Error Code	Description
EPERM	The process did not have appropriate privilege to use this system call.

FILES

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

SEE ALSO

time(2)

NAME

suspend, resume – Controls execution of processes

SYNOPSIS

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

int suspend (int category, int id);

int resume (int category, int id);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The suspend system call makes a process or group of processes ineligible to execute; the resume system call restores a process or group of processes to be eligible to execute.

The suspend and resume system calls accept the following arguments:

category Specifies one of the following categories: C_PROC, C_PGRP, or C_SESS

id Specifies the PID, PGRP, or SID corresponding to the *category*. A PID of 0 means that the current process is affected, and a PID of –1 means that all processes except the current process are affected. Similarly, a PGRP of 0 means that all processes in the current process group are affected, and a PGRP of –1 means that all processes not in the current process group are affected. A SID of 0 means that all processes in the current session are affected. System processes, such as processes 0 and 1, are never suspended.

The calling process must be the owner of the specified process or have appropriate privilege. If an affected process is not part of the calling process' session, the calling process must have appropriate privilege.

NOTES

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

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

Privilege	Description
PRIV_ADMIN	The calling process is allowed to suspend or resume a process that is not part of its session.
PRIV_MAC_WRITE	The calling process is allowed to override the security label restrictions for suspend and resume.
PRIV_POWNER	The calling process is considered the owner of the specified process.

If the PRIV_SU configuration option is enabled, the super user is considered the owner of every affected process and is allowed to suspend or resume a process that is not part of its session. If the PRIV_SU configuration option is enabled, the super user is allowed to override security label restrictions.

RETURN VALUES

If suspend or resume 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 suspend or resume system call fails if one of the following error conditions occurs:

Error Code	Description
EAGAIN	One of the processes being suspended was never in a suspendible state during the last 120 seconds. The suspend system call may be attempted again.
EINTR	An asynchronous signal (such as interrupt or quit), which you have elected to catch, occurred during a suspend system call. When execution resumed after processing the signal, the interrupted system call returned this error condition.
EINVAL	One of the arguments contains a value that is not valid.
EPERM	The calling process does not own an affected process and does not have appropriate privilege.
EPERM	The calling process is attempting to suspend or resume a process that is not part of its session and does not have appropriate privilege.
ESRCH	No process can be found that matches the *category* and *id* requests.
ESRCH	The calling process does not meet security label requirements and does not have appropriate privilege.
ESRCH	The calling process does not own any processes in the requested process group or session and does not have appropriate privilege.

FORTRAN EXTENSIONS

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

```
INTEGER category, id, SUSPEND, I
I = SUSPEND (category, id)
```

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

```
INTEGER category, id, RESUME, I
I = RESUME (category, id)
```

EXAMPLES

The following examples show how to use the `suspend` and `resume` system calls to suspend and resume program execution.

Example 1: This program suspends itself using the `suspend` request. When the process resumes, it computes the number of seconds it was in a suspended state.

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

main()
{
     time_t stime, etime;

     stime = time((long *) 0);

     if (suspend(C_PROC, 0) == -1) {
          perror("suspend failed");
          exit(1);
     }

     etime = time((long *) 0);
     printf("Program was suspended for %ld seconds\n", etime - stime);
}
```

Example 2: Using the `resume` system call, this program resumes any suspended process whose process identification number (PID) is supplied as an argument.

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

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

     sscanf(argv[1], "%d", &pid);  /* convert pid string to int */

     if (resume(C_PROC, pid) == -1) {
          perror("resume failed");
          exit(1);
     }
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `suspend` and `resume` system calls

SEE ALSO

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

NAME

`symlink` – Makes a symbolic link to a file

SYNOPSIS

```
#include <unistd.h>
int symlink (char *name1, char *name2);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `symlink` system call makes a symbolic link to a file. It accepts the following arguments:

name1 Specifies the string used in creating the symbolic link.

name2 Specifies the name of the file created.

A symbolic link *name2* is created to *name1*. Either name may be an arbitrary path name; the files do not need to be on the same file system.

NOTES

The active security label of the calling process must fall within the security label range of the file system on which *name2* will reside.

To be granted search permission to a component of the path prefix of *name2*, 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 of *name2*, the active security label of the process must be equal to the security label of the 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 *name2* path prefix via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted write permission to the parent directory of *name2* via the permission bits and access control list.
`PRIV_MAC_READ`	The process is granted search permission to every component of the *name2* path prefix via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the parent directory of *name2* 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 parent directory of *name2*.

RETURN VALUES

If symlink 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 symlink 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 of *name2*.
EACCES	Write permission is denied to the parent directory of *name2*.
EEXIST	The file referred to by *name2* already exists.
EFAULT	The *name1* or *name2* argument points outside the process allocated address space.
EFLNEQ	The active security label of the calling process does not fall within the range of the file system on which *name2* will reside.
EINVAL	The *name2* argument contains a character 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 *name2*.
ENAMETOOLONG	A component of either *name1* or *name2* exceeds 255 characters, or either *name1* or *name2* exceeds 1023 characters.
ENOENT	A component of the path prefix of *name2* does not exist.
ENOSPC	The directory in which the entry for the new symbolic link is being placed cannot be extended because of one of the following: • There is no space left in the file system to make the directory longer. Sometimes, but not always, the new directory name added by symlink requires that an additional block be allocated. • There was not enough space (one block) to write the *name2* string to disk.
ENOSPC	The new symbolic link cannot be created because no space left is on the file system that will contain the link.
ENOSPC	No free inodes exist on the file system on which the file is being created.
ENOTDIR	A component of the path prefix of *name2* is not a directory.

EQACT	The new symbolic link cannot be created for one of the following reasons: the quota of inodes on the file system on which the file is being created has been exhausted for the current account, the account's quota of disk blocks on the file system that will contain the link has been exhausted, or the directory in which the entry for the new symbolic link is being placed cannot be extended because the account's quota of disk blocks on the file system containing the directory has been exhausted.
EQGRP	The new symbolic link cannot be created for one of the following reasons: the quota of inodes on the file system on which the file is being created has been exhausted for the current group, the group's quota of disk blocks on the file system that will contain the link has been exhausted, or the directory in which the entry for the new symbolic link is being placed cannot be extended because the group's quota of disk blocks on the file system containing the directory has been exhausted.
EQUSR	The new symbolic link cannot be created for one of the following reasons: the quota of inodes on the file system on which the file is being created has been exhausted for the current user, the user's quota of disk blocks on the file system that will contain the link has been exhausted, or the directory in which the entry for the new symbolic link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
EROFS	The file *name2* would reside on a read-only file system.

EXAMPLES

This example shows how to use the `symlink` system call to create a symbolic link. The following `symlink` request makes a symbolic link to a file from information supplied as arguments. The first argument, `argv[1]`, is the path name of an existing file or directory that is the target of the new link. The second argument, `argv[2]`, is the name of the new link. The program later forces an `ls -l` display of the new link.

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

main(int argc, char *argv[])
{
     static char cmd[50] = {"ls -l "};

     if (argc < 3) {
          fprintf(stderr, "Insufficient arguments supplied!\n");
          exit(1);
     }

     if (symlink(argv[1], argv[2]) == -1) {
          perror("symlink failed");
          exit(1);
     }

     strcat(cmd, argv[2]);
     system(cmd);
}
```

FILES

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

SEE ALSO

`link`(2), `lstat`(2), `readlink`(2), `stat`(2), `unlink`(2)

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

NAME

sync – Flushes system buffers out of main memory

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The sync system call causes all information in memory that should be on disk to be flushed out of main memory, including modified inodes and delayed block I/O.

Information is flushed to the logical device cache (a second-level cache) or disk if a logical device cache is not configured. The ldsync(8) command flushes data from the logical device cache to disk.

Use sync in programs that examine a file system (for example, fsck(8) and df(1)). You must execute sync before halting or rebooting the system.

The sync system call issues the write request; it may return before all of the data is written.

RETURN VALUES

None

FORTRAN EXTENSIONS

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

```
INTEGER SYNC, I
I = SYNC ( )
```

Alternatively, sync can be called from Fortran as a subroutine, because there is no return value:

```
CALL SYNC ( )
```

FILES

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

SEE ALSO

`fsync`(2), `ioctl`(2)

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

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

General UNICOS System Administration, Cray Research publication SG–2301

NAME

sysconf – Retrieves system implementation information

SYNOPSIS

```
#include <unistd.h>
long sysconf (int name);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The sysconf system call provides a method for an application to determine the current value of a configurable system limit or option. It accepts the following argument:

name Represents the system variable to be queried.

The values for *name* specified by the POSIX P1003.1 standard are listed in the following with a brief description of the value each returns:

_SC_ARG_MAX	The maximum length of arguments in bytes for exec( ).
_SC_CHILD_MAX	The maximum number of processes allowed per user.
_SC_CLK_TCK	The number of clock ticks per second.
_SC_JOB_CONTROL	The POSIX job control option has been implemented; if true, it is 1.
_SC_NGROUPS_MAX	The multigroups size; if multigroups are not implemented, it is 0.
_SC_OPEN_MAX	The maximum number of open files.
_SC_PID_MAX	The maximum value for a process ID. (This name is no longer specified by POSIX P1003.1.)
_SC_SAVED_IDS	The exec( ) routine saves the real UID and GID of the caller for later use; if true, it is 1.
_SC_STREAM_MAX	The number of streams that one process can have open at any given time.
_SC_TZNAME_MAX	The maximum number of bytes supported for the name of a time zone.
_SC_UID_MAX	The maximum value for a user ID. (This name is no longer specified by POSIX P1003.1.)
_SC_VERSION	The version/revision of the POSIX standard used for this implementation.

The values for *name* specified by the POSIX P1003.2 standard are listed in the following with a brief description of the value each returns:

_SC_BC_BASE_MAX	The maximum *obase* value allowed by the bc(1) utility.
_SC_BC_DIM_MAX	The maximum number of elements permitted in an array by the bc(1) utility.
_SC_BC_SCALE_MAX	The maximum *scale* value allowed by the bc(1) utility.
_SC_BC_STRING_MAX	The maximum length of a string constant accepted by the bc(1) utility.
_SC_COLL_WEIGHTS_MAX	The maximum number of weights that can be assigned to an entry of the LC_COLLATE *order* keyword in the locale definition file.
_SC_EXPR_NEST_MAX	The maximum number of expressions that can be nested within parentheses by the expr(1) utility.
_SC_LINE_MAX	Unless otherwise noted, the maximum length, in bytes, of a utility's input line (either standard input or another file), when the utility is described as processing text files. The length includes room for the trailing newline character.
_SC_RE_DUP_MAX	The maximum number of repeated occurrences of a regular expression permitted when using the interval notation *m*,*n*.
_SC_2_VERSION	The C-language development facilities support the POSIX P1003.2 C-Language Bindings Option.
_SC_2_C_DEV	The system supports the POSIX P1003.2 C-Language Development Utilities Option.
_SC_2_FORT_DEV	The system supports the POSIX P1003.2 FORTRAN Development Utilities Option.
_SC_2_FORT_RUN	The system supports the POSIX P1003.2 FORTRAN Runtime Utilities Option.
_SC_2_LOCALEDEF	The system supports the POSIX P1003.2 Locale Creation Option.
_SC_2_SW_DEV	The system supports the POSIX P1003.2 Software Development Utilities Option.
_SC_2_C_BIND	The system supports the POSIX P1003.2 C-Language Bindings Option.
_SC_2_CHAR_TERM	The system supports at least one terminal type capable of all operations in the POSIX standard.
_SC_2_C_VERSION	The version of the POSIX P1003.2 interfaces used for this implementation.

The values for *name* specified by the X/Open XPG4 standard are listed in the following with a brief description of the value each returns:

_SC_PASS_MAX	The maximum size of a password.
_SC_XOPEN_VERSION	The version of the X/Open standard supported by this implementation.

_SC_XOPEN_CRYPT	The system supports the X/Open Encryption Feature Group.
_SC_XOPEN_ENH_I18N	The system supports the X/Open Enhanced Internationalization Feature Group.
_SC_XOPEN_SHM	The system supports the X/Open Shared Memory Feature Group.
_SC_LOGIN_NAME_MAX	The maximum length of a login name.
_SC_TTY_NAME_MAX	The maximum length of a tty path name.
_SC_GETGR_R_SIZE_MAX	The maximum size of data buffers used by the `getgrgid_r` and `getgrnam_r` library functions.
_SC_GETPW_R_SIZE_MAX	The maximum size of data buffers used by the `getpwgid_r` and `getpwnam_r` library functions.

The values for *name* that are unique to Cray Research are listed in the following with a brief description of the value each returns. These unique values will not change.

_SC_CRAY_AVL	Additional vector logical hardware; if present, it is 1.
_SC_CRAY_BDM	Bidirectional memory enabled; if true, it is 1.
_SC_CRAY_BMM	Bit matrix multiply unit; if present, it is 1.
_SC_CRAY_CHIPSZ	The memory chip size.
_SC_CRAY_CPCYCLE	The CPU cycle time in picoseconds.
_SC_CRAY_EMA	Extended memory addressing hardware; if present, it is 1.
_SC_CRAY_HPM	Hardware performance monitor hardware; if present, it is 1.
_SC_CRAY_IOS	The I/O subsystem type; `IOS_MODEL_E`.
_SC_CRAY_MFSUBTYPE	The mainframe subtype (see `sys/sn.h` and `sys/machd.h`).
_SC_CRAY_MFTYPE	The mainframe type (see `sys/sn.h` and `sys/machd.h` for all systems).
_SC_CRAY_NBANKS	The number of memory banks on the Cray Research mainframe.
_SC_CRAY_NBUF	Number of 512-word system I/O cache blocks.
_SC_CRAY_NCPU	The number of CPUs currently available.
_SC_CRAY_NDISK	The number of disk devices configured on the system.
_SC_CRAY_NMOUNT	The number of file-system mount points configured in the system.
_SC_CRAY_NPTY	The maximum number of pty devices configured into the currently running version of the operating system.
_SC_CRAY_NUSERS	The number of users configured.
_SC_CRAY_NVHISP	The number of VHISP channels to the SSD solid-state storage device.
_SC_CRAY_OPEN_MAX	The value of the largest open file limit supported by the kernel.

_SC_CRAY_OS_HZ	The frequency per second (usually 100) with which the operating system clock routine is called.
_SC_CRAY_RELEASE	The release level of the currently running version of the operating system. The release level is multiplied by 1000 (for example, release level 5.0 = 5000, release level 5.1 = 5100, and so on).
_SC_CRAY_SCTRACE	System call timing; if on, it is 1.
_SC_CRAY_SDS	Size of secondary data segment (SDS) memory in 512-word blocks.
_SC_CRAY_SECURE_MAC	The system supports `syshigh` and `syslow` security labels. This implies that file systems have been appropriately labeled.
_SC_CRAY_SECURE_SYS	The system has been generated with security enabled. Always returns TRUE (nonzero).
_SC_CRAY_SERIAL	The system serial number (see `sys/sn.h`).
_SC_CRAY_SSD	Size of the SSD in words.
_SC_CRAY_SYSMEM	The size of the kernel and tables, in words.
_SC_CRAY_USRMEM	The user memory available, in words.

RETURN VALUES

If *name* not a valid value, the `sysconf` system call returns a value of –1, and sets `errno` to `EINVAL`. If *name* is valid, `sysconf` returns the current variable value for the system.

EXAMPLES

This example shows how to use the `sysconf` system call to retrieve system implementation information. The following `sysconf` requests illustrate some of the different types of information available through this call. Because `sysconf` returns the mainframe type as an integer, the programmer creates a table to convert the mainframe type to a more recognizable character string.

```
#include <unistd.h>

main()
{
     /* The following table based upon the mainframe definitions
        in the sys/machd.h and sys/machcons.h header files. */
     static char *mftype[] = {"", "CRAY Y-MP", "", "CRAY C90"};

     printf("The mainframe type = %s\n",
             mftype[sysconf(_SC_CRAY_MFTYPE)]);
     printf("The current number of available cpu's = %ld\n",
             sysconf(_SC_CRAY_NCPU));
     printf("The size of the kernel and tables = %ld words\n",
             sysconf(_SC_CRAY_SYSMEM));
     printf("The amount of user memory available = %ld words\n",
             sysconf(_SC_CRAY_USRMEM));
     printf("The number of clock ticks per second = %ld\n",
             sysconf(_SC_CLK_TCK));
}
```

FILES

`/usr/include/sys/machd.h`	Contains machine-dependent information
`/usr/include/sys/sn.h`	Contains Cray Research mainframe hardware information
`/usr/include/sys/tfm.h`	Defines `TFM_UDB_6`
`/usr/include/unistd.h`	Contains C prototype for the `sysconf` system call

SEE ALSO

`pathconf`(2)

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

General UNICOS System Administration, Cray Research publication SG–2301

NAME

sysfs – Gets file system type information

SYNOPSIS

```
#include <sys/fstyp.h>
#include <sys/fsid.h>
int sysfs (int opcode, char *fsname);
int sysfs (int opcode, int fs_index, char *buf);
int sysfs (int opcode);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The sysfs system call returns information about the file system types configured in the system. The number of arguments accepted by sysfs varies and depends on the *opcode*.

opcode Specifies function to perform. The following are valid *opcode* values:

- GETFSIND — Translates *fsname*, a null-terminated file system identifier, into a file system type index.
- GETFSTYP — Translates *fs_index*, a file system type index, into a null-terminated file system identifier and writes it into the buffer to which *buf* points. This buffer must be at least of size FSTYPSZ, as defined in sys/fstyp.h.
- GETNFSTYP — Returns the total number of file system types configured in the system.

fsname Specifies file system identifier.

fs_index Specifies file system type index.

buf Points to a buffer.

RETURN VALUES

If sysfs completes successfully, it returns the file system type index if *opcode* is GETFSIND, a value of 0 if *opcode* is GETFSTYP, or the number of file system types configured if *opcode* is GETNFSTYP. Otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

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

Error Code	Description
`EFAULT`	The *buf* or *fsname* argument points to a user address that is not valid.
`EINVAL`	The *fsname* argument points to a file system identifier that is not valid; *fs_index* is 0, or not valid; *opcode* is not valid.

EXAMPLES

This example shows how to use the `sysfs` and `statfs`(2) system calls to obtain file system information. The `statfs`(2) request retrieves information for the file system containing the file whose name is passed as an argument. The `sysfs` system call converts the numerical file system type to a character-string format before displaying it. The final `sysfs` request determines the total number of file system types configured in the system.

```
#include <sys/types.h>
#include <sys/statfs.h>
#include <sys/fstyp.h>
#include <sys/fsid.h>

main(int argc, char *argv[])
{
     struct statfs stats;
     char buf[FSTYPSZ];
     int nconfig;

     if (statfs(argv[1], &stats, sizeof(struct statfs), 0) == -1) {
          perror("statfs error");
          exit(1);
     }

     if (sysfs(GETFSTYP, stats.f_fstyp, buf) == -1) {
          perror("sysfs (GETFSTYP) error");
          exit(1);
     }

     printf("File system type => %s\n", buf);
     printf("Block size = %d\n", stats.f_bsize);
     printf("Fragment size = %d\n", stats.f_frsize);
     printf("Total number of blocks on file system = %d\n", stats.f_blocks);
     printf("Total number of free blocks = %d\n", stats.f_bfree);
     printf("Total number of file nodes (inodes) = %d\n", stats.f_files);
     printf("Total number of free file nodes = %d\n", stats.f_ffree);
     printf("Volume name => %s\n", stats.f_fname);
     printf("Pack name => %s\n\n", stats.f_fpack);
```

```
        if ((nconfig = sysfs(GETNFSTYP)) == -1) {
             perror("sysfs (GETNFSTYP) error");
             exit(1);
        }
        else {
             printf("Number of file system types configured = %d\n", nconfig);
        }
    }
```

SEE ALSO

statfs(2)

NAME

`syssgi` – Provides a system interface to Silicon Graphics workstations

SYNOPSIS

```
#include <sys/syssgi.h>
ptrdiff_t syssgi (int request, ...);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The `syssgi` call is a system interface specific to Silicon Graphics workstations. It accepts the following argument:

request Represents the requested interface. The currently supported values for *request* are listed below.

The following *request* values are interfaces that implement various `libc` functions. They are all subject to change and should not be called directly by applications.

`SGI_GETASH`

`SGI_SETASH`

`SGI_GETPRID`

`SGI_GETDFLTPRID`

`SGI_SETPRID`

`SGI_GETSPINFO`

`SGI_SETSPINFO`

`SGI_NEWARRAYSESS`

The following *request* values are interfaces that implement various `libarray` functions. They are all subject to change and should not be used directly by applications.

`SGI_ENUMASHS`

`SGI_GETARSESS`

`SGI_GETASMACHID`

`SGI_PIDSINASH`

`SGI_SETASMACHID`

RETURN VALUES

If `syssgi` completes successfully, a command-dependent value (default of `0`) is returned; otherwise, a value of `-1` is returned, and `errno` is set to indicate the error.

ERRORS

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

Error Code	Description
`EFAULT`	A buffer is referenced which is not in a valid part of the calling program's address space.
`ENOMEM`	The specified buffer was not large enough to hold the entire list of process IDs returned by the `SGI_PIDSINASH` function.

NAME

tabinfo, tabread – Returns information on and reads a system table

SYNOPSIS

```
#include <sys/table.h>
int tabinfo (char *name, struct tbs *info);
int tabread (char *name, char *buf, long nbytes, long offset);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The tabinfo and tabread system calls let you read a system table without reading /dev/kmem. The tabinfo call describes table characteristics: location, header length, number of entries, and size of entry. Using the information returned by tabinfo, you can create a user buffer into which tabread will read all or part of a table.

If you have read permission on /dev/kmem, you will have unlimited access with tabinfo and tabread, regardless of the table permissions. The calls let you process a table in segments; the requirement for an arbitrarily large buffer does not exist. Using the information from tabinfo, you can calculate buffer sizes.

The tabinfo and tabread system calls accept the following arguments:

name Points to a table name (defined in sys/table.h).

info Points to the tbs structure to receive the information.

buf Points to the character to which the buffer points to receive the table.

nbytes Specifies the number of bytes to be read.

offset Specifies the number of bytes after the table base at which tabread is to start reading.

NOTES

The tabinfo and tabread system calls are similar to the nlist(3C) library routine and have some of the same functionality.

If the process does not have read permission to /dev/kmem and the table permissions restrict access, the process must belong to the appropriate table group, or the process must have appropriate privilege.

If the SECURE_MAC option is enabled, the calling process uses the tabread system call to retrieve process table information and the active security label of the process table entry is greater than the active security label of the calling process, the returned process table entry is zero-filled. A process with appropriate privilege is allowed to override this behavior.

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

Privilege	Description
PRIV_ADMIN	The process is granted access to tables whose permissions restrict access.
PRIV_DAC_OVERRIDE	The process is granted search permission to every directory component of the /dev/kmem path prefix via the permission bits and access control list.
PRIV_DAC_OVERRIDE	The process is granted read permission to /dev/kmem via the permission bits and access control lists.
PRIV_MAC_READ	The process is granted search permission to every directory component of the /dev/kmem path prefix via the security label.
PRIV_MAC_READ	The process is granted read permission to /dev/kmem via the security label.
PRIV_MAC_READ	The process is allowed to read all process table entries. That is, process table entries are not zero-filled.

If the PRIV_SU configuration option is enabled, the super user is allowed to override all tabread and tabinfo restrictions.

RETURN VALUES

If tabinfo or tabread 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 tabinfo or tabread system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	Access is not permitted.
EFAULT	The address of *info* or *buf* is illegal.
EINVAL	The *name* argument points to an undefined table name.

EXAMPLES

The following example shows how to use the tabinfo and tabread system calls to retrieve information from a system table. In this case, the entire file table from the system is read into the process's memory space.

```
#include <sys/table.h>
#include <stdlib.h>

/* The structure of type tbs defined as follows (from <sys/table.h>):

struct tbs {
        char    name[9];        - ASCII name of table entry -
        long    *addr,          - Start address of table (word *) -
                head,           - Length of table header (chars) -
                ent,            - Number of entries -
                len,            - Length of each entry (chars)-
                perm;           - Permission word -
};  */

main()
{
     struct tbs tinfo;
     char *tloc;
     long tsize;

     if (tabinfo(FILETAB, &tinfo) == -1) {
          perror("tabinfo failed");
          exit(1);
     }

     tsize = tinfo.head + (tinfo.ent * tinfo.len);
     tloc = (char *) malloc(tsize);

     if (tabread(FILETAB, tloc, tsize, 0) == -1) {
          perror("tabread failed");
          exit(1);
     }
}
```

FILES

`/usr/include/sys/table.h`	Contains user or system structure declaration

SEE ALSO

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

NAME

`target` – Retrieves or modifies machine characteristics

SYNOPSIS

```
#include <sys/target.h>
int target (int request, struct target *addr);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `target` system call provides a mechanism for compilers to determine the physical characteristics of the host system.

Users may retrieve machine characteristics for the machine on which they are running (host machine) or for the machine for which they are targeting code (target machine). Only a process with appropriate privilege can modify characteristics for the target machine.

The `target` system call accepts the arguments:

request Specifies the type of request; *request* may be one of the following:

`MC_GET_SYSTEM`	Retrieves the host machine characteristics.
`MC_GET_TARGET`	Retrieves the target machine characteristics.
`MC_SET_TARGET`	Modifies the target machine characteristics (on all systems except Cray MPP systems).

addr Specifies the address of a structure of type `target`.

NOTES

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

Privilege	Description
`PRIV_ADMIN`	The process is allowed to modify characteristics of the target machine.

If the `PRIV_SU` configuration option is enabled, the super user or a process with the `PERMBITS_SYSPARAM` permbit is allowed to modify characteristics of the target machine.

RETURN VALUES

If `target` 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 `target` system call fails if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	The `target` structure address is not within the user's bounds.
`EINVAL`	The *request* field is not valid.
`EPERM`	The process tried to use the `MC_SET_TARGET` request but did not have appropriate privilege.

EXAMPLES

The following example shows how to use the `target` system call to retrieve the characteristics of the target machine. The field containing the primary machine type name (`mc_pmt` in the `target` structure) contains character data, but the field type is defined as `long int`.

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

main()
{
     struct target data;

     if (target(MC_GET_TARGET, &data) == -1) {
          perror("target failed");
          exit(1);
     }

     printf("Primary machine type name = %s\n", &data.mc_pmt);
     printf("Number of memory banks = %ld\n", data.mc_bank);
     printf("Number of started processors = %ld\n", data.mc_ncpu);
     printf("Instruction Buffer Size (words) = %ld\n", data.mc_ibsz);
     printf("Main memory size (words) = %ld\n", data.mc_msz);
     printf("Number of clocks for a memory read = %ld\n", data.mc_mspd);
     printf("Clock period in picoseconds = %ld\n", data.mc_clk);
     printf("Number of cluster register sets = %ld\n", data.mc_ncl);
     printf("Memory bank busy time in clocks = %ld\n", data.mc_bbsy);
     printf("Number of clock ticks per second = %ld\n", data.mc_clktck);
     printf("System serial number = %ld\n", data.mc_serial);
     printf("UNICOS release level = %ld\n", data.mc_rls/1000);
}
```

SEE ALSO

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

NAME

tcgetpgrp, tcsetpgrp – Gets or sets terminal process group ID of the foreground process group

SYNOPSIS

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

pid_t tcgetpgrp (int fildes);

pid_t tcsetpgrp (int fildes, pid_t pgrp_id);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The tcgetpgrp system call returns the value of the process group ID of the foreground process group; the tcsetpgrp system call sets the foreground process group ID to *pgrp_id*.

The tcgetpgrp and tcsetpgrp system calls accept the following arguments:

fildes Specifies the controlling terminal of the calling process, and that controlling terminal must be currently associated with the session of the calling process.

pgrp_id Matches a process group ID of a process in the same session as the calling process.

NOTES

The tcgetpgrp system call is allowed from a process that is a member of a background process group; however, the information may subsequently be changed by a process that is a member of a foreground process group.

RETURN VALUES

If tcgetpgrp completes successfully, it returns the process group ID of the foreground process group associated with the terminal; otherwise, a value of −1 is returned, and errno is set to indicate the error.

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

FILES

`/usr/include/sys/types.h`	Contains types required by ANSI X3J11
`/usr/include/unistd.h`	Contains C prototype for the `tcgetpgrp` and `tcsetpgrp` system calls

NAME

_tfork – Creates a multitasking process

SYNOPSIS

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

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The _tfork system call creates a process much as fork(2) does. The main difference between fork(2) and _tfork is that a process created by _tfork shares the same memory area as the parent process. Because the calling process and the created process share the same memory area, the two processes have a sibling-sibling relationship rather than a parent-child relationship. The two processes are said to share a multitasking group. These processes have the following differences from normal processes:

- The process ID (PID) returned when the last process in the multitasking group exits is the *pid* of the first process to exist in the group. The parent *pid* of all processes in the multitasking group is the parent *pid* of the first process in the group. Only the last process in the group can be detected by the wait(2) call. Each process, except the last one to exit, does so without signaling its parent process.
- Whenever a process from a multitasking group is connected to a physical CPU, the process has a cluster. The cluster is loaded when the first process from the group is connected, and it remains loaded as long as any process in the group is connected.

The restart(2) system call and any of the exec(2) family of system calls are not allowed during multitasking. Using them results in the EINVAL error.

RETURN VALUES

If _tfork completes successfully, it returns to each process its own *pid*. If _tfork fails, a value of –1 is returned. Because the two processes share a memory area, a call to _tfork from C does not function as expected, because the stack is not copied; therefore, _tfork is most commonly called from a multitasking library.

FILES

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

SEE ALSO

exec(2), fork(2), restart(2), wait(2)

NAME

thread – Registers this process as a thread

SYNOPSIS

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

int thread (struct thread *buf);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The thread system call registers this process as a thread and requests special handling by the kernel. It accepts the following argument:

buf Specifies the address of the thread communication area.

The library uses _tfork(2) and this system call to implement microtasking. The thread structure and context structure are used for fast communication between the library and the kernel. A thread structure includes the following members:

```
long   pid;       /* Pid of this process */
long   wakeup;    /* Request by library to wakeup this proc */
long   giveup;    /* Request by kernel to give up cpu */
long   context;   /* Pointer to context save area */
```

The wakeup flag may be set by a sibling process in a multitasking group to request that the kernel wake up a sleeping sibling. The giveup flag is set by the kernel to request that the thread voluntarily give up the CPU. This is done so that the thread may get to a convenient stopping point and thereby allow the other threads to progress. If the thread does not give up the CPU promptly after giveup is set, the kernel will take the CPU.

If the process at any time sets the context pointer to refer to an area outside its address space or shrinks its address space by using sbreak(2) so that the thread structure is no longer included, the kernel revokes the thread status of the process and sends the SIGERR signal to the process.

RETURN VALUES

If thread 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 `thread` system call fails if one of the following error conditions occurs:

Error Code	Description
`EBUSY`	The process is already a thread.
`EFAULT`	The `thread` structure to which *buf* points is not fully contained in the process address space.
`EINVAL`	The `pid` value in the `thread` structure is not the correct value for this process.

SEE ALSO

`sbreak`(2), `_tfork`(2)

NAME

`time` – Gets time

SYNOPSIS

```
#include <time.h>
time_t time (time_t *tloc);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `time` system call returns the value of time in seconds since 00:00:00 Greenwich mean time (GMT), January 1, 1970. It accepts the following argument:

tloc Points to a second location where the return value is stored.

If *tloc* is 0, `time` returns the time only as the return value. If the *tloc* argument points to an address that is not valid, the actions for `time` are undefined.

NOTES

Under UNICOS, `time` is implemented as a system call, but the `time`(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.

RETURN VALUES

The `time` system call returns the value of time.

FORTRAN EXTENSIONS

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

```
INTEGER TIME, I
I = TIME ( )
```

EXAMPLES

The following example shows how to use the `time` system call to retrieve the current time from the system. It also illustrates how the value returned by `time` is converted to character-string format in two different ways using the `ctime`(3C) and `localtime`(3C) (see `ctime`(3C)) library routines.

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

main()
{
     static char *daytab[] = {"Sunday", "Monday", "Tuesday",
                 "Wednesday", "Thursday", "Friday", "Saturday"};
     time_t timval;
     struct tm *tmptr;

     time(&timval);
     printf("The time in seconds since Jan 1, 1970 is %ld\n", timval);

     printf("The date and time are %s", ctime(&timval));

     tmptr = localtime(&timval);
     printf("The reformatted date and time are %s %2d/%2d/%2d %.2d:%.2d\n",
            daytab[tmptr->tm_wday], tmptr->tm_mon + 1, tmptr->tm_mday,
            tmptr->tm_year, tmptr->tm_hour, tmptr->tm_min);
}
```

SEE ALSO

`stime`(2)

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

NAME

`times` – Gets process and child process times

SYNOPSIS

```
#include <sys/times.h>
clock_t times (struct tms *buffer);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `times` system call returns time-accounting information to the process. It accepts the following argument:

buffer Points to the `tms` structure.

A `tms` structure includes the following members:

```
clock_t   tms_utime;     CPU time used during the execution
                         of instructions in the user space
                         of the calling process
clock_t   tms_stime;     CPU time used by the system
                         on behalf of the calling process
clock_t   tms_cutime;    Sum of the "tms_utime"s and "tms_cutime"s
                         of the child processes
clock_t   tms_cstime;    Sum of the "tms_stime"s and "tms_cstime"s
                         of the child processes
```

This information comes from the calling process and each of its terminated child processes for which it has executed a `wait`(2). All times are given in system hardware clock ticks; there are `CLK_TCK` system hardware clock ticks per second. The `CLK_TCK` macro is defined in the `time.h` file.

RETURN VALUES

If `times` completes successfully, it returns the elapsed real time, in system hardware clock ticks, since an arbitrary point in the past (for example, system start-up time). This point does not change from one invocation of `times` to another. If `times` fails, a –1 is returned, and `errno` is set to indicate the error.

ERRORS

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

Error Code	Description
EFAULT	The *buffer* argument points to an illegal address.

FORTRAN EXTENSIONS

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

```
INTEGER buffer(n), TIMES, I
I = TIMES (buffer)
```

EXAMPLES

This example shows how to use the `times` system call to gather CPU usage information to time a particular section of user code:

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

main()
{
     struct tms before, after;
     clock_t utime, stime, startime, endtime;
     startime = times(&before);

     /* The section of code to be timed resides here. */

     endtime = times(&after);

     utime = after.tms_utime - before.tms_utime;
     stime = after.tms_stime - before.tms_stime;

     printf("\nCPU time used in user space = %f sec or %ld clock ticks\n",
               (float)utime/(float)CLK_TCK, utime);
     printf("CPU time used by the system = %f sec or %ld clock ticks\n",
               (float)stime/(float)CLK_TCK, stime);
     printf("Wall clock time used by process = %f sec ",
               (float)(endtime - startime)/(float)CLK_TCK);
     printf("or %ld clock ticks\n", endtime - startime);
}
```

SEE ALSO

`exec`(2), `fork`(2), `time`(2), `wait`(2)

NAME

trunc – Truncates a file

SYNOPSIS

```
#include <unistd.h>
long trunc (int fildes);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The trunc system call sets the size of the file indicated by *fildes* to the current file pointer. The process must have write permission to the file. The trunc system call accepts the following argument:

fildes Indicates the size of the file.

NOTES

In addition to changing the size of a file, the trunc system call releases file storage beyond the truncated size, including any storage preallocated to the file through the ialloc(2) system call.

A process is granted write permission to the file only if the active security label of the process is 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.

RETURN VALUES

If trunc completes successfully, the new file size is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

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

Error Code	Description
EAGAIN	Mandatory file and record locking is set (see chmod(2)), outstanding record locks exist on the file (see fcntl(2)), and O_NDELAY was set in the file flag word.
EBADF	The *fildes* argument is not a valid file descriptor open for writing.

EBADF	The active security label of the process does not equal the security label of the file, and the process does not have appropriate privilege.
EDEADLK	A deadlock situation would have occurred waiting for a blocking record lock to be removed.
EINTR	Mandatory file and record locking is set (see chmod(2)), outstanding record locks exist on the file (see fcntl(2)), and O_NDELAY is not set in the file flag word.
EINVAL	The pointer for *fildes* is beyond the end-of-file.
ENOLCK	The system record lock table was full; therefore, it was not possible to wait for a blocking record lock to be removed.

FORTRAN EXTENSIONS

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

```
INTEGER fildes, TRUNC, I
I = TRUNC (fildes)
```

EXAMPLES

This example shows how to use the trunc system call to truncate the last half of a file's contents. In this case, the request truncates file test_data so that the file is one-half of its original size.

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

main()
{
     int fd;
     long size;

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

     size = lseek(fd, 0L, 2);    /* determine size of the file */

     lseek(fd, size/2, 0);       /* seek to middle of the file */

     if (trunc(fd) == -1) {      /* truncate last half of the file */
          perror("trunc failed");
          exit(1);
     }

     close(fd);
}
```

FILES

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

SEE ALSO

`chmod`(2), `fcntl`(2), `ialloc`(2), `lseek`(2)

NAME

`ulimit` – Gets and sets user limits

SYNOPSIS

```
#include <ulimit.h>
long int ulimit (int cmd, ...);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `ulimit` system call controls process limits. It accepts the following argument:

cmd Specifies one of the values, defined in the `ulimit.h` file. These values are as follows:

`UL_GETFSIZE`	Gets the regular file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.
`UL_SETFSIZE`	Sets the file size limit of the process to the value of the second argument, taken as a `long int`. Any process can decrease this limit, but only a process with an effective user ID of the super user can increase the limit. The new file size limit is returned.
`UL_GMEMLIM`	Gets the maximum break value in bytes. On Cray PVP systems, this value is an integer number of bytes; on Cray MPP systems, it is the actual byte address of the break value. To use this value as an argument to the `brk`(2) system call, see example 2 in the EXAMPLES section.

Only an appropriately privileged process can increase a file size limit.

NOTES

The minimum allocation unit, both on disk and in memory, for all Cray Research systems is 4096 bytes. When `ulimit` is called to set the process limit, the limit is rounded to the next 4096-byte boundary. (For example, if `ulimit` is called to set the limit at 5120 bytes, it is actually set to 8192 bytes.)

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

Privilege	**Description**
`PRIV_RESOURCE`	The process is allowed to increase a file size limit.

If the PRIV_SU configuration option is enabled, the super user or a process with the PERMBITS_RESLIM permbit is allowed to increase a file size limit.

RETURN VALUES

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

ERRORS

If the following error condition occurs when the value of *cmd* is UL_SETFSIZE, the ulimit system call fails and the process limit remains unchanged.

Error Code	Description
EINVAL	An illegal argument was passed to the system call.
EPERM	A process without appropriate privilege tried to increase the file size limit.

FORTRAN EXTENSIONS

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

```
INTEGER cmd, newlimit, ULIMIT, I
I = ULIMIT (cmd, newlimit)
```

EXAMPLES

The following examples illustrate use of the ulimit system call to get and set user limits.

Example 1: This ulimit request returns the file size limit for the current process. Because the file size limit value is in 512-byte units, it is converted to the more familiar unit of 512 words.

```
#include <ulimit.h>

main()
{
     long fslim;

     fslim = ulimit(UL_GETFSIZE);
     printf("File size limit = %ld (512-byte) blocks\n", fslim);
     printf("                = %ld (512-word) blocks\n", fslim/8);
     }
```

Example 2: This `ulimit` request returns the maximum break value for this process; then the `brk` system call attempts to increase the process size to that limit.

```
#include <ulimit.h>

main()
{
     if  ((brk(((char *)0) + ulimit(UL_GMEMLIM))) == -1) {
           perror("brk failed");
           exit(1);
     }
}
```

FILES

`/usr/include/ulimit.h`	Contains C prototype for the `ulimit` system call; also contains the `UL_GETFSIZE`, `UL_SETFSIZE`, and `UL_GMEMLIM` symbols.

SEE ALSO

`brk`(2), `limit`(2), `write`(2)

NAME

umask – Sets and gets file creation mask

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask (mode_t cmask);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The umask system call sets the file creation mode mask of the process to *cmask* and returns the previous value of the mask.

The umask system call accepts the following argument:

cmask Specifies the new value of the file creation mode mask. Only the low-order 9 bits of *cmask* and the file creation mode mask are used.

RETURN VALUES

The previous value of the file mode creation mask is returned.

FORTRAN EXTENSIONS

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

```
INTEGER cmask, UMASK, I
I = UMASK (cmask)
```

EXAMPLES

This example shows how to use the umask system call to change a process's file creation mask. The following umask request changes the file creation mask of the current process to 077, and the previous file creation mask is displayed.

After the file creation mask is altered, an open request creates a file with permissions of 0755. Because the file creation mask is now 077, the permissions set for the new file are 0700.

```
main()
{
     printf("The previous file creation mask was %o\n", umask(077));

     if ((fd = open("datafile", O_CREAT | O_WRONLY, 0755)) == -1) {
          perror("open failed");
          exit(1);
     }
}
```

SEE ALSO

chmod(2), creat(2), mknod(2), open(2)

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

NAME

umount – Unmounts a file system

SYNOPSIS

```
int umount (char *file);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The umount system call accepts the following argument:

file Points to a path name.

The umount system call requests that a previously mounted file system contained on the block special device or directory identified by *file* be unmounted; *file* is a pointer to a path name. After unmounting the file system, the directory on which the file system was mounted reverts to its ordinary interpretation.

Only an appropriately privileged process can use this system call.

NOTES

Unmounting the root device causes the kernel to reread all in-core (in memory) information from that device.

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, the super user is allowed to use this system call and is granted search permission to every component of the path prefix.

RETURN VALUES

If umount 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 `umount` 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`	A file on *file* is busy.
`EFAULT`	The *file* argument points outside the allocated process address space.
`EINVAL`	The *file* argument is not mounted.
`ENAMETOOLONG`	The lenth of the *file* argument exceeds `PATH_MAX`, or a path name component exceeds `NAME_MAX` while `POSIX_NO_TRUNC` is in effect.
`ENOENT`	The specified file does not exist or the *file* argument points to an empty string.
`ENOTDIR`	A component of the path prefix is not a directory.
`EPERM`	The process does not have appropriate privilege to use this system call.

SEE ALSO

`mount`(2)

NAME

uname – Gets name of current operating system

SYNOPSIS

```
#include <sys/utsname.h>
int uname (struct utsname *name);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The uname system call stores information identifying the current operating system. It accepts the following argument:

name Points to the structure to receive the information. Each member of the structure receives a null-terminated character string.

A utsname structure includes the following members:

```
char   sysname[9];   /* Current operating system name */
char   nodename[9];  /* Name by which the system is known
                        on a communications network */
char   release[9];   /* Release of the operating system */
char   version[9];   /* Release version of the operating system */
char   machine[12];   /* Standard name identifying the hardware
                        on which the operating system is running */
```

RETURN VALUES

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

ERRORS

The uname system call fails if the following error condition occurs:

Error Code	Description
EFAULT	The *name* argument points to an address that is not valid.

FORTRAN EXTENSIONS

See UNAME(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165 (for all systems except Cray MPP systems and CRAY T90 series systems). Also see the PXFUNAME(3F) subroutine.

EXAMPLES

This example shows how to use the uname system call to retrieve the name of the operating system as well as the release and version of the operating system:

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

main()
{
     struct utsname opname;

     if (uname(&opname) == -1) {
          perror("uname failed");
          exit(1);
     }
     else {
          printf("The current operating system is %s\n", opname.sysname);
          printf("    Release %s\n", opname.release);
          printf("    Version %s\n", opname.version);
     }
}
```

SEE ALSO

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

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

NAME

unlink, unlink2 – Removes directory entry

SYNOPSIS

All Cray Research systems:

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

Cray PVP systems:

```
#include <unistd.h>
int unlink2 (const char *path);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to unlink)

DESCRIPTION

The unlink system call removes the directory entry specified by the path name to which the *path* argument points. When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

The unlink2 system call, which is a Cray Research extension, functions like the unlink system call except for the values returned.

The unlink and unlink2 system calls accept the following argument:

path Points to the path name of the directory entry to be removed.

The values returned by the unlink2 system call differ from those returned by unlink. When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. In this case, unlink2 returns a positive value that represents the number of blocks of space returned to the file system free space pool.

If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file are closed. In this case, unlink2 returns a 0 if the operation is allowed, and the actual file space is returned later.

NOTES

The unlink system call does not remove a directory from the file system, it simply unlinks the reference from the specified directory. Use of unlink on directories by privileged users can cause file system errors (unlinked inodes), which can be fixed by using the fsck(8) command. A privileged user should use rmdir(2) to remove a directory from the file system.

A process is granted write permission to the directory containing the link only if the active security label of the process is equal to the security label of the directory.

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.

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

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

Privilege	Description
PRIV_ADMIN	The process is allowed to unlink a directory.
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 file's parent directory via the permission bits and access control list.
PRIV_FOWNER	The process is allowed to specify a directory that has the "sticky" mode bit set and that the process does not own.
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 specified file and its parent directory via the security label.

If the PRIV_SU configuration option is enabled, the super user is allowed to unlink a directory. The super user is allowed to specify a directory that has the "sticky" mode bit set and that it does not own. The super user is granted search permission to every directory component of the path prefix. The super user is granted write permission to the file and its parent directory. 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 unlink completes successfully, a value of 0 is returned.

If unlink2 completes successfully and the file space has already been returned to the file system free space pool, a positive value that represents the number of blocks of space returned is returned. If the actual return of the file space to the file system free pool has been postponed because some other process still references the file, then a value of 0 is returned.

If unlink or unlink2 fail to complete successfully, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The unlink or unlink2 system call fails and the specified file remains linked 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 on the directory containing the link to be removed.
EACCES	The active security label of the process does not equal the specified security label of the file.
EBUSY	The entry to be unlinked is the mount point for a mounted file system.
EFAULT	The *path* argument points outside the allocated process address space.
ENAMETOOLONG	The *path* argument is longer than PATH_MAX characters.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The specified file is a directory, and the process does not have appropriate privilege.
EROFS	The directory entry to be unlinked is part of a read-only file system.

FORTRAN EXTENSIONS

The unlink 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 UNLINK, I
I = UNLINK (path)
```

Alternatively, unlink 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*n path
CALL UNLINK (path)
```

The Fortran program must not specify both the subroutine call and the function reference to unlink() from the same procedure. *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. The PXFUNLINK(3F) subroutine provides similar functionality and is available on all Cray Research systems.

EXAMPLES

This example shows how to use the unlink system call to implement a scratch file for use in the program. A unique name for the scratch file is derived by calling the tmpnam subroutine. The unlink request unlinks the scratch file immediately after it is opened. At this point, the file has no links and is called a *zero-linked file*.

The scratch file (possessing no links) is not removed because the program still has it open for access. The scratch file remains in existence until the program closes it, terminates without closing it, or abnormally terminates, or until the UNICOS system dies.

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

main()
{
     int fd;
     char *scratch;               /* path name to a scratch file */

     scratch = tmpnam((char *) 0); /* create unique temp file name */

     /* Create a file; open it for read & write. */

     if( (fd = open(scratch, O_RDWR | O_CREAT | O_EXCL, 0600)) == -1) {
          perror("open failed");
          exit(1);
     }

     /* Now remove links, but don't close it. */

     if (unlink(scratch) == -1) {
          perror("unlink failed");
          exit(1);
     }

     /* Program writes and reads the file here. */

     close(fd);                    /* also removes file, since # links = 0 */
}
```

FILES

/usr/include/unistd.h	Contains C prototype for the unlink and unlink2 system calls

SEE ALSO

`close`(2), `link`(2), `open`(2), `rmdir`(2)

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

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

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

NAME

upanic – Stops the system from a user process

SYNOPSIS

```
#include <sys/panic.h>
int upanic (int cmd);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The upanic system call, which is referred to as the user panic, allows the system to be stopped from a user process. This feature is useful with problems, such as bad data on an I/O read, that cannot be detected at the system level or for problems that occur only with a specific user code or activity, such as user data corruption. The upanic system call accepts the following argument:

cmd Specifies an entry. It can be one of the following:

PA_SET	Sets the user panic flag; requires appropriate privilege.
PA_RELAX	Clears the user panic flag; requires appropriate privilege.
PA_PANIC	Stops the system if the user panic flag has been set; can be called by any process. When PA_PANIC is sent but the user panic flag is not set, the call is inoperative; thus, it can be embedded in code with no side effect other than the overhead of the system call path.

Only an appropriately privileged process can set or clear the user panic flag.

NOTES

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

Privilege	Description
PRIV_ADMIN	The process is allowed to set or clear the user panic flag.

If the PRIV_SU configuration option is enabled, the super user is allowed to set or clear the user panic flag.

RETURN VALUES

When upanic 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 `upanic` system call fails if one of the following error conditions occurs:

Error Code	Description
`EINVAL`	An argument is not valid. The command is not one of the listed values.
`EPERM`	The process does not have appropriate privilege to set or clear the user panic flag.

SEE ALSO

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

NAME

ustat – Gets file system statistics

SYNOPSIS

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

int ustat (dev_t dev, struct ustat *buf);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The ustat system call returns information about a mounted file system. It accepts the following arguments:

dev Specifies a device number that identifies a device containing a mounted file system.

buf Points to a ustat structure.

The ustat structure includes the following members:

```
daddr_t   f_tfree;      /* Total free blocks */
ino_t     f_tinode;     /* Number of free inodes */
char      f_fname[6];   /* Name of the mounted file system */
char      f_fpack[6];   /* Name of the file system pack */
```

NOTES

The statfs(2) system call obsoletes some purposes of ustat, but ustat remains useful for determining whether a given device is mounted.

RETURN VALUES

If ustat 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 ustat system call fails if one of the following error conditions occurs:

Error Code	Description
EFAULT	The *buf* argument points outside the allocated process address space.
EINVAL	The *dev* argument is not the device number of a device containing a mounted file system.

FORTRAN EXTENSIONS

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

```
INTEGER dev, buf(m), USTAT, I
I = USTAT (dev, buf)
```

SEE ALSO

`stat`(2), `statfs`(2)

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

NAME

utime – Sets file access and modification times

SYNOPSIS

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

int utime (const char *path, const struct utimbuf *times);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The utime system call sets the access and modification times of a specified file. It accepts the following arguments:

path Points to a file path name.

times Specifies source of the access and modification times.

If *times* is null, the access and modification times of the file are set to the current time. A process must be the file owner or have write permission to use utime in this manner.

If *times* is not null, it is interpreted as a pointer to a utimbuf structure, and the access and modification times are set to the values contained in the designated structure. Only the file owner can use utime this way.

The utimbuf structure follows:

```
struct utimbuf   {
    time_t   actime;    /* Access time */
    time_t   modtime;   /* Modification time */
};
```

Times are measured in seconds since 00:00:00 Greenwich mean time (GMT), January 1, 1970.

The utime function also causes the time of the last file status change (st_ctime) to be updated (see stat(2)).

NOTES

A process is granted write permission to the file only if the active security label of the process is equal to the security label of the file.

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_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 file's parent directory via the permission bits and access control list.
PRIV_FOWNER	The process is considered the file owner.
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 file via the security label.

If the PRIV_SU configuration option is enabled, the super user is considered the file owner, is granted search permission to every component of the path prefix, and is granted write permission to the file.

RETURN VALUES

If utime 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 utime system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	Search permission is denied by a component of the path prefix.
EACCES	The process is not the file owner, *times* is null, write permission is denied, and the process does not have appropriate privilege.
EFAULT	The *path* argument points outside the allocated process address space.
EFAULT	The *times* argument is not null and points outside the allocated process address space.
EMANDV	The active security label of the process does not equal the security label of the file, and the process does not have appropriate privilege.
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, *times* is not null, and the process does not have appropriate privilege.
EROFS	The file system containing the file is mounted as read only.

FORTRAN EXTENSIONS

The utime 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 times, UTIME, I
I = UTIME (path, times)
```

Alternatively, utime 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*n path
INTEGER times
I = UTIME (path, times)
```

The Fortran program must not specify both the subroutine call and the function reference to utime from the same procedure. *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. The PXFUTIME(3F) subroutine provides similar functionality and is available on all Cray Research systems.

EXAMPLES

This example shows how to use the utime system call to modify the last accessed and last modified time-stamps in a file's inode.

The program first displays the current time stamps saved in the file's inode. Then, the utime request modifies the two time stamps, and they are displayed again.

```
#include <sys/types.h>
#include <utime.h>
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>
#include <unistd.h>

main()
{
     static char file[] = {"datafile"};
     struct stat buf;

     if (stat(file, &buf) == -1) {
          perror("stat failed");
          exit(1);
     }

     printf("Before utime(), %s was last accessed on %s",
                                                  file, ctime(&buf.st_atime));
     printf("Before utime(), %s was last modified on %s",
                                                  file, ctime(&buf.st_mtime));

     if (utime(file, ((struct utimbuf *) 0)) == -1) {  /* set timestamps to */
          perror("utime failed");                      /* current time      */
          exit(1);
     }

     if (stat(file, &buf) == -1) {
          perror("stat failed");
          exit(1);
     }

     printf("\nAfter utime(), %s was last accessed on %s",
                                                  file, ctime(&buf.st_atime));
     printf("After utime(), %s was last modified on %s",
                                                  file, ctime(&buf.st_mtime));
}
```

SEE ALSO

`stat`(2)

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

NAME

vfork – Creates a new process in a memory efficient way

SYNOPSIS

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

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The vfork system call can be used to create new processes without fully copying the address space of the old process. It is useful when the purpose of fork(2) would have been to create a new system context for an execv(2). The vfork system call differs from fork(2) in that the child borrows the parent's memory and thread of control until a call to execve(2) or an exit (either by a call to exit(2) or an abnormal exit). The parent process is suspended while the child is using its resources.

The vfork system call returns 0 in the child's context and (later) the process ID of the child in the parent's context.

The vfork system call can normally be used just like fork. It does not work, however, to return while running in the child's context from the procedure that called vfork since the eventual return from vfork would then return to a no longer existent stack frame. Be careful to call _exit(2) rather than exit(2) if you cannot call execve(2), because exit(2) will flush and close standard I/O channels, and mess up the parent process's standard I/O data structures. (Even when using fork(2), it is wrong to call exit(2) because buffered data would then be flushed twice.)

RETURN VALUES

If vfork completes successfully, it returns a value of 0 to the child process and returns the process ID of the child process to the parent process; otherwise, a value of –1 is returned to the parent process, no child process is created, and errno is set to indicate the error.

ERRORS

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

Error Code	Description
EAGAIN	The system-imposed limit on the total number of processes under execution in the whole system (NPROC) is exceeded.
EAGAIN	The system-imposed limit on the total number of processes under execution by one user (CHILD_MAX) is exceeded.
ENOMEM	Not enough main memory or swap space exists.

BUGS

Because UNICOS signal(2) and sigctl(2) signal registration is implemented with a library-level signal vector, any changes in signal registration by the child will be reflected in the parent process. This behavior differs from other UNIX systems supporting the vfork system call. Other changes to signal disposition (for example, SIG_IGN or SIG_DFL) will behave the same as with the fork(2) system call.

EXAMPLES

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

Example 1: The vfork request generates a new process (referred to as the child process). The child process returns from vfork and executes in the same process space as the parent process. The parent process does not return from the vfork request until the child process has executed some form of exec(2) request or an exit. At the time that the child process issues an exec(2) request, enough memory is generated for the new (child) process to execute the specified program; then the parent process returns from vfork and continues execution. The return value from vfork indicates whether execution is in the parent or child process.

```
int res;

if ((res = vfork()) == -1) {
    perror("vfork failed");
    exit(1);
}

if (res == 0) {
     /* Code here is executed in the child process until an exec or
        _exit request is made.  Parent does not return from vfork
        until child process issues one of these requests.  Since child
        process has access to parent's data fields and signal
        dispositions here until an exec or _exit request, it should
        not modify those on which the parent depends.  Child process
        must refrain from returning (e.g., falling out of the process)
        since that will cause the parent process' stack frame to be
        removed.  Parent process expects presence of the stack frame. */

}

else {
     /* Code here is executed in the parent process after the child
        process issues an exec or _exit request. */
}
```

Example 2: This example illustrates a typical usage of the vfork request. When a parent process generates a child process so that a different program can execute in the child process, the vfork request is the most efficient way to handle the task.

Typically, when the child process returns from `vfork`, it immediately performs an `exec`(2) request (in this case `execl`(2)) to generate a new process space and to load the specified program for execution into the child process. With `vfork`, the process space for the parent is not duplicated in the child process. The parent and child processes then execute different programs in parallel.

```
int res;

if ((res = vfork()) == -1) {
     perror("vfork failed");
     exit(1);
}

if (res == 0) {            /* In child process? */
    execl("childprog", "childprog", "arg1", "arg2", 0);
    perror("exec for childprog failed");
    _exit(1);
}

/* Parent process continues execution here after successful execl
   request in the child process. */
```

FILES

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

SEE ALSO

`exec`(2), `fork`(2), `sigctl`(2), `signal`(2), `wait`(2)

NAME

wait, waitpid – Waits for a child process to stop or terminate

SYNOPSIS

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

pid_t wait (int *stat_loc);

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

IMPLEMENTATION

Cray PVP systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The wait system call suspends the calling process until one of the child processes terminates or until a child process that is being traced stops because it has hit a breakpoint. If a signal is received, wait returns prematurely. If a child process has stopped or terminated before the call on wait, return is immediate.

The wait and waitpid system calls accept the following arguments:

stat_loc Returns the status of a terminated child process.

pid Specifies the child process that will have its status returned. -1 indicates that the status of any available terminated process should be returned.

options Sets optional flag for the waitpid system call. The *options* argument is constructed from the bitwise inclusive OR of 0 or more of the following flags, defined in header file sys/wait.h:

- WNOHANG — Indicates that waitpid returns immediately and does not suspend execution of the calling process if status is not available for one of the child processes specified by *pid*.
- WUNTRACED — Provides the following status if job control is supported. Reports to the requesting process the status of any child process specified by *pid* that has stopped and whose status has not yet been reported since it stopped.
- WMTWAIT — Waits for the children of any member of the multitasking group. In UNICOS 9.0 this is the default behavior for both wait and waitpid. The flag is still provided for source compatibility. To get the previous behavior, see the description of the WLWPWAIT flag.
- WLWPWAIT — Waits only for the immediate children of the calling light-weight process (LWP). This flag is not recommended for general use.

If 0, the caller will suspend until a child process stops or terminates.

If the *stat_loc* argument is not 0, 16 bits of status information are stored in the low-order 16 bits of the location to which *stat_loc* points. This status differentiates between stopped and terminated child processes. If the child process has terminated, The status identifies the cause of termination and passes useful information to the parent process. This is accomplished in the following manner:

- If the child process has stopped, the high-order 8 bits of status contain the number of the signal that caused the process to stop and the low-order 8 bits are set equal to 0177.
- If the child process has terminated because of an `exit`(2) call, the low-order 8 bits of status are 0 and the high-order 8 bits contain the low-order 8 bits of the argument that the child process passed to `exit`(2).
- If the child process has terminated because of a signal, the high-order 8 bits of status are 0 and the low-order 8 bits contain the number of the signal that caused the termination. If the low-order seventh bit (that is, bit 0200) is set, a core image will also have been produced; see `signal`(2).

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means that the initialization process inherits the child processes.

The `waitpid` system call behaves identically to `wait` if the *pid* argument has a value of −1 and the *options* argument has a value of 0; otherwise, the values of *pid* and *options* modify its behavior.

The *pid* argument specifies a set of child processes for which status is requested. The `waitpid` system call returns only the status of a child process from this set.

- If *pid* is equal to −1, status is requested for any child process; `waitpid` is then equivalent to `wait`.
- If *pid* is greater than 0, it specifies the process ID of a single child process for which status is requested.
- If *pid* is equal to 0, status is requested for any child process with a process group ID that is equal to that of the calling process.
- If *pid* is less than −1, status is requested for any child process with a process group ID that is equal to the absolute value of *pid*. The *options* argument is constructed from the bitwise inclusive OR of 0 or more of the following flags, defined in header file `sys/wait.h`:

If `wait` and `waitpid` return because the status of a child process is available, these system calls return a value equal to the process ID of the child process. In this case, if the value of the *stat_loc* argument is not `NULL`, information is stored in the location to which *stat_loc* points. If, and only if, the status returned is from a terminated child process that returned a value of 0 from `main`( ) or passed a value of 0 as the *status* argument to `_exit`(2) or `exit`(2), the value stored at the location to which *stat_loc* points is 0.

Regardless of its value, this information is interpreted using macros. These macros are defined in the `sys/wait.h` file and evaluate to integral expressions. The *stat_val* argument is the integer value to which *stat_loc* points.

`WIFEXITED` (*stat_val*) Returns a nonzero value if the child process terminated normally.

WEXITSTATUS (*stat_val*)	Determines the low-order 8 bits of the argument that the child process passed to _exit(2) or exit(2), or the value the child process returned from main( ). Use only if WIFEXITED returns a nonzero value.
WIFSIGNALED (*stat_val*)	Returns a nonzero value if the child process terminated due to the receipt of a signal that it did not catch (see the signal.h file).
WTERMSIG (*stat_val*)	Determines the number of the signal that caused the termination of the child process. Use only if WIFSIGNALED returns a nonzero value.
WIFSTOPPED (*stat_val*)	Returns a nonzero value if the child process is stopped due to a signal.
WSTOPSIG (*stat_val*)	Determines the number of the signal that caused the child process to stop. Use only if WIFSTOPPED returns a nonzero value.

If the information in the location to which *stat_loc* points is stored there by a call to waitpid that specified the WUNTRACED flag, exactly one of the WIFEXITED, WIFSIGNALED, and WIFSTOPPED macros evaluates to a nonzero value. If the information stored at the location to which *stat_loc* points is stored there by a call to waitpid that did not specify the flag or a call to wait, exactly one of the WIFEXITED and WIFSIGNALED macros evaluates to a nonzero value.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes (now orphaned) are assigned a new parent process ID. The parent process of orphaned child processes is the init process (*pid* = 1).

NOTES

In UNICOS 9.0, the default behavior of both wait and waitpid acts as though the WMTWAIT flag was set. The WLWPWAIT flag provides the previous default behavior. However, it is not expected that this will be useful because using waitpid with a specified process ID should provide the necessary control for child process management.

The idea of a parent process has changed in UNICOS 9.0. Previously, the parent was the entity (previously termed a *process*, now a *light-weight process*) that created the child by using the fork(2) system call. Now, the parent process is the entire multitasking group in which the former parent process was a member. This change is part of the more general change that moves from a multitasking model that supports multiple processes in a multitasking group to a model that supports a single process. This change is described more fully in the getpid(2) man page.

RETURN VALUES

If the child process stopped or terminated after the parent process's call to wait, the system call returns the child process ID. If wait is interrupted by a signal other than the death-of-a-child-process signal (SIGCLD) or if the calling process has no existing zombie-producing child processes (see the following paragraph), a value of –1 is returned, and errno is set to indicate the error.

A zombie-producing child process results when the death-of-a-child-process signal SIGCLD is set to anything other than to be ignored. If SIGCLD is set to be ignored, a call to wait returns –1, and an errno of ECHILD.

If wait or waitpid returns because the status of a child process is available, the call returns a value equal to the process ID of the child process for which status is reported. If wait or waitpid returns due to the delivery of a signal to the calling process, a value of –1 is returned and errno is set to EINTR. If the waitpid system call is invoked with WNOHANG set in *options*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The wait system call fails and its actions are undefined if the *stat_loc* argument points to an illegal address. The call fails and returns immediately if one of the following error conditions occurs:

Error Code	Description
ECHILD	The calling process has no existing unwaited-for child processes.
EINTR	Receipt of a signal other than the death-of-a-child-process signal.

The waitpid system call returns –1, and errno is set to indicate the error if one of the following error conditions occurs:

Error Code	Description
ECHILD	The process or process group specified by *pid* does not exist or is not a child of the calling process.
EINTR	The call was interrupted by a signal. The value of the location to which *stat_loc* points is undefined.
EINVAL	The value of the *options* argument is not valid.

FORTRAN EXTENSIONS

The wait system call may be called from Fortran as a function:

```
INTEGER statloc, WAIT, I
I = WAIT (statloc)
```

EXAMPLES

The following examples illustrate use of the wait and waitpid system calls. Both examples show a parent process waiting for its child process to complete.

Example 1: In this program, the `wait` request in the parent process waits for its child process to complete.

The program first creates a child process and allows the child process to perform some other work. Executing in parallel with the child process, the parent displays the process identification number (PID) of the forked child process and then waits for its completion. Once the child process completes, the parent uses a macro (that is, `WIFEXITED` or `WIFSIGNALED`) to determine the cause of the child's termination.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

main()
{
     int res, cid_ret, cid_stat;

     if ((res = fork()) == -1) {
          perror("fork failed");
          exit(1);
     }

     if (res == 0) {          /* child process */
          /* child process performs its work here */
     } else {                   /* parent process */
          printf("Child process has pid = %d\n", res);
          cid_ret = wait(&cid_stat);        /* waits for child to complete */
          if (WIFEXITED(cid_stat)) {        /* if child terminated normally */
               printf("Child process %d terminated normally with ", cid_ret);
               printf("exit status = %d.\n", WEXITSTATUS(cid_stat));
          } else {
               if (WIFSIGNALED(cid_stat)) { /* if child terminated (signal) */
                    printf("Child process %d terminated due to ", cid_ret);
                    printf("signal no. -> %d.\n", WTERMSIG(cid_stat));
               }
          }
     }
}
```

Example 2: In this program, the `waitpid` request in the parent process waits for its child process to complete.

The program first creates a child process and allows the child process to perform some other work. Executing in parallel with the child process, the parent displays the PID of the forked child process and then waits for its completion. Once the child process completes, the parent uses a macro (that is, `WIFEXITED` or `WIFSIGNALED`) to determine the cause of the child's termination.

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

main()
{
     int res, cid_ret, cid_stat;

     if ((res = fork()) == -1) {
          perror("fork failed");
          exit(1);
     }

     if (res == 0) {          /* child process */
          /* child process performs its work here */
     } else {                   /* parent process */
          printf("Child process has pid = %d\n", res);
          cid_ret = waitpid(res, &cid_stat, 0);/* waits for child to complete */
          if (WIFEXITED(cid_stat)) {        /* if child terminated normally */
               printf("Child process %d terminated normally with ", cid_ret);
               printf("exit status = %d.\n", WEXITSTATUS(cid_stat));
          } else {
               if (WIFSIGNALED(cid_stat)) { /* if child terminated (signal) */
                    printf("Child process %d terminated due to ", cid_ret);
                    printf("signal no. -> %d.\n", WTERMSIG(cid_stat));
               }
          }
     }
}
```

SEE ALSO

`exec`(2), `exit`(2), `fork`(2), `getpid`(2), `intro`(2), `pause`(2), `signal`(2)

NAME

waitjob – Gets information about a terminated child job

SYNOPSIS

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

int waitjob (struct jtab *jtab);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The waitjob system call obtains information about a terminated child job (that is, a child job in which all of the processes have exited) that has been configured to send a signal to its parent on termination. The system call is named waitjob because, like the wait(2) system call, it returns information about only an object that is considered to be a child of the calling process. Unlike wait, however, and despite its name, waitjob never blocks the caller's execution.

The waitjob system call accepts the following argument:

jtab Returns the *jtab* entry for the terminated job.

If the *jtab* argument is not 0, the jtab structure containing statistics for the terminated job is returned at that address; otherwise, no jtab structure is returned.

If the parent process of any job exits, the parent process ID of each remaining child job is set to 0, and the jobs exit silently from the system on termination.

The waitjob system call obtains information only for a terminated job that was configured to send a signal to its parent on termination. The setjob(2) system call makes it possible to create jobs that exit from the system silently. These jobs do not send a signal to their parent on termination, and waitjob provides no information about these jobs.

See getjtab(2) for a description of the jtab structure.

NOTES

Any process that does not ignore SIGCLD signals (see signal(2)) and uses waitjob must first issue a wait(2) system call, which gathers the eldest child of the job when the child exits. If wait(2) is not issued, the job will continue to exist, with the eldest process of the job existing as a zombie process; waitjob will not consider the job to be terminated.

RETURN VALUES

If `waitjob` completes successfully, the job ID of the terminated job is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

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

Error Code	Description
`EAGAIN`	The calling process is the parent of one or more jobs configured to send a signal on termination, but none of the child jobs has terminated.
`ECHILD`	The calling process does not have any child jobs configured to send a signal on termination.
`EFAULT`	The *jtab* argument points outside the allocated process address space.

SEE ALSO

`getjtab`(2), `setjob`(2), `signal`(2), `wait`(2)

NAME

`wracct` – Writes an accounting record to the kernel accounting file or to a daemon accounting file

SYNOPSIS

```
#include <acct/dacct.h>
int wracct (char *buf, int did, int jid, int nbyte);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `wracct` system call writes an accounting record to a daemon accounting file. If a user enables job accounting, the accounting record will also be written to the user's job accounting file. The `ja`(1) command can process this file.

The `wracct` system call accepts the following arguments:

buf Points to the accounting record. The size (in bytes) of this buffer is specified by *nbyte*. The accounting records are defined in `acct`(5) and in `/usr/include/acct/dacct.h`.

did Specifies the type of accounting record that will be written. These daemon identifiers are specified in `/usr/include/sys/accthdr.h`.

jid Specifies the job ID of the process for which the record is being written. This is usually the job ID contained in the accounting record to which *buf* points.

nbyte Specifies the size (in bytes) of *buf*.

The daemons and the accounting subsystem must enable the appropriate type of accounting by using the `turnacct`(8) or `turndacct`(8) command.

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_ACCT`	The process is allowed to use this system call.

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

RETURN VALUES

If `wracct` 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 `wracct` system call fails if one of the following error conditions occurs:

Error Code	Description
`EINVAL`	An argument that is not valid was passed to the system call.
`EPERM`	The process does not have appropriate privilege to use this system call.

FILES

`/usr/include/acct/dacct.h`	Defines daemon accounting files
`/usr/include/sys/accthdr.h`	Specifies daemon identifiers

SEE ALSO

`jacct`(2)

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

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

`qmgr`(8), `tpdaemon`(8), `turnacct`(8), `turndacct`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

write – Writes on a file

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
ssize_t write (int fildes, const void *buf, size_t nbyte);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The write system call writes from a buffer to a file. It accepts the following arguments:

fildes Specifies the 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 in which the data is stored.

nbyte Specifies the number of bytes to be written.

On devices capable of seeking, the writing of data proceeds from the position in the file indicated by the file pointer. On return from write, the file pointer is incremented by the number of bytes written.

On devices incapable of seeking, writing starts at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer is set to the end of the file before each write.

If the file being written is a pipe (or FIFO special file), some special semantics apply:

- If the O_NDELAY and O_NONBLOCK flags in the file flag word are both clear (the normal case), the write request will block until there is room to copy all the data into the pipe.
- If the O_NDELAY flag is set (no delay), the number of bytes to be written to the pipe is less than or equal to the value PIPE_BUF, and insufficient space exists in the pipe, write returns a value of 0 immediately (no blocking) with no data written to the pipe.
- If the O_NONBLOCK flag is set (no delay), the number of bytes to be written to the pipe is less than or equal to the value PIPE_BUF, and insufficient space exists in the pipe, write returns a value of –1 immediately (no blocking) with no data written to the pipe.

- If the O_NDELAY flag is set (no delay), the number of bytes to be written to the pipe is greater than the value PIPE_BUF, and insufficient space exists in the pipe, write copies as many bytes to the pipe as possible and returns the number of bytes written.
- If the O_NONBLOCK flag is set (no delay), the number of bytes to be written to the pipe is greater than the value PIPE_BUF, and insufficient space exists in the pipe, write copies as many bytes to the pipe as possible and returns a value of −1 to the user. (The user is not able to determine the number of bytes actually delivered to the pipe.)

The value PIPE_BUF is defined in the header limits.h and typically has a value of 512 words (4096 bytes).

For regular files, if the O_SYNC flag of the file status flags is set, the write does not return until both the file data and file status are updated physically. This function is for special applications that require extra reliability at the cost of performance. For block special files, if O_SYNC is set, the write does not return until the data is updated physically. A write to a regular file is blocked if mandatory file and record locking is set (see chmod(2)) and a record lock is owned by another process on the segment of the file to be written. If O_NDELAY and O_NONBLOCK are both clear the write sleeps until the blocking record lock is removed.

NOTES

A 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 to 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 write completes successfully, the number of bytes actually written is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

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

Error Code	Description
EAGAIN	Mandatory file and record locking was set, O_NDELAY was set, and a blocking record lock exists.
EBADF	The *fildes* argument is not a valid file descriptor open for writing.
EBADF	The active security label of the process does not equal the security label of the file, and the process does not have appropriate privilege.

EDEADLK	The write was going to go to sleep and cause a deadlock situation to occur.
EFAULT	The *buf* argument points outside the allocated process address space.
EFBIG	An attempt was made to write a file that exceeds the file size limit or the maximum file size of the process. See ulimit(2).
EINTR	A signal was caught during the write system call (see signal(2)).
ENOLCK	The system record lock table was full; therefore, the write 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.
EPIPE and SIGPIPE signals	An attempt is made to write to a pipe that is not open for reading by any process.
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.

EXAMPLES

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

Example 1: In this example, the `read`(2) and `write` system calls sequentially update the records of file `datafile`. For each iteration of the *while* loop, a record is read into user memory, updated, and then written back to `datafile`.

A value 0 returned by `read`(2) indicates an end-of-file (EOF) condition has been reached. The data read and written is staged in the system buffer cache because the `O_RAW` flag is not specified on the `open`(2) request.

```
#include <unistd.h>

main()
{
     int fd, cnt;
     char buf[100];

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

     while ((cnt = read(fd, buf, 100)) != 0) { /* read returning 0 means EOF */

          /* update data (cnt bytes) in buf here and then write back */

          lseek(fd, (long) cnt, 1);        /* backup to beginning of record */
          if (write(fd, buf, cnt) == -1) { /* write record back to file */
               perror("write failed");
               exit(1);
          }
     }

     printf("EOF reached on file datafile.\n");
}
```

Example 2: In this example, the `read`(2) and `write` system calls perform a simple file copy operation. The first argument to the program is the file name of the file to be copied. The second argument is the file name of the duplicate copy.

```
#include <fcntl.h>

#define BUFSIZE 4096

main(int argc, char *argv[])
{
     int ifd, ofd, noread, nowrite, cnt;
     char buf[BUFSIZE];

     if ((ifd = open(argv[1], O_RDONLY)) == -1) {
          perror("opening input file failed");
          exit(1);
     }

     if ((ofd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644)) == -1) {
          perror("opening output file failed");
          exit(1);
     }

     while ((noread = read(ifd, buf, BUFSIZE)) != 0) {
          if (noread == -1) {
               perror("read error");
               exit(1);
          }
          cnt = 0;
          do {
               if ((nowrite = write(ofd, &buf[cnt], noread - cnt)) == -1) {
                    perror("write error");
                    exit(1);
               }
               cnt += nowrite;
          } while (cnt < noread);
     }

     close(ifd); close(ofd);
}
```

FILES

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

SEE ALSO

accept(2), chmod(2), creat(2), dup(2), fcntl(2), lseek(2), open(2), pipe(2), read(2), signal(2), socket(2), socketpair(2), ulimit(2), writea(2)

NAME

writea – Performs asynchronous write on a file

SYNOPSIS

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

int writea (int fildes, char *buf, unsigned nbyte, struct iosw *status,
int signo);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The writea system call performs an asynchronous write of a specified number of bytes from a buffer to a file. The first three arguments of the writea system call are the same as the write(2) system call. The last two arguments are used for I/O completion notification as in the reada(2) system call.

The writea system call accepts the following arguments:

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

buf Points to the buffer in which the data is stored.

nbyte Specifies the number of bytes to be written.

status Points to a iosw structure. This structure is defined in the usr/include/sys/iosw.h file.

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.

A write to a regular file is blocked if mandatory file and record locking is set (see the chmod(2) man page), and a record lock is owned by another process on the segment of the file to be written. If O_NDELAY and O_NONBLOCK are both clear, the write sleeps until the blocking record lock is removed.

NOTES

A process must be granted write permission to the file via the security label. That is, the active security label of the proces must be equal to 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 writea completes successfully, the number of bytes remaining to be written is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

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

Error Code	Description
EAGAIN	Mandatory file and record locking was set, O_NDELAY was set, and a blocking record lock exists.
EBADF	The *fildes* argument is not a valid file descriptor open for writing.
EBADF	The active security label of the process does not equal the security label of the file, and the process does not have appropriate privilege.
EDEADLK	The write was going to go to sleep and cause a deadlock situation to occur.
EFAULT	The *buf* or *status* argument is not fully contained in the process address space.
EFBIG	An attempt was made to write a file that exceeds the file size limit or the maximum file size of the process. See the ulimit(2) man page.
EINTR	The process caught a signal during the writea system call (see signal(2)).
EINVAL	The *signo* argument is not a valid signal number or 0.
ENOLCK	The system record lock table was full; therefore, the write 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.
EPIPE and SIGPIPE signals	An attempt is made to write to a pipe that is not open for reading by any process.
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 writea system call can be called from Fortran as a function:

```
INTEGER fildes, buf(n), nbyte, status, signo, WRITEA, I
I = WRITEA (fildes, buf(n), nbyte, status, signo)
```

EXAMPLES

The following examples illustrate different uses of the `writea` system call. In each example, the write operation completes in parallel with other work in the user's process. Simpler solutions appear in the last two examples, which make use of additional calls.

Example 1: In this program, the `writea` request specifies the delivery of a `SIGUSR1` signal on the completion of the request.

The program uses the `pause`(2) request to wait for the completion of the asynchronous write operation (that is, reception of the `SIGUSR1` signal). The library routine `sigoff`(3C) 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 wrstat;

main()
{
     char buf[4096];
     int fd;
     void wrhdlr(int signo);

     signal(SIGUSR1, wrhdlr);

     if ((fd = open("newfile", O_WRONLY | O_CREAT | O_RAW, 0644)) == -1) {
          perror("open (newfile) failed");
          exit(1);
     }

     /* Program populates buffer buf with data here. */

     sigoff();      /* delay signal reception until pause() is reached */
     writea(fd, buf, 4096, &wrstat, SIGUSR1);  /* SIGUSR1 sent when
                                                  write completes */

     /* Perform other work here in parallel with I/O completion. */

     pause();       /* wait for write to complete - pause() calls sigon() */

     /* Output data has now vacated buffer buf due to writea. */
}

void wrhdlr(int signo)
{
```

```
     signal(signo, wrhdlr);
     printf("writea wrote %d bytes\n", wrstat.sw_count);
     wrstat.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 output operation. The user's program is informed of the completion by reception of the SIGUSR1 signal. While recalla(2) can wait for the completion of multiple asynchronous I/O requests from multiple files, it only waits for one write operation in this example.

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

struct iosw wrstat;

main()
{
     char buf[4096];
     int fd;
     long mask[RECALL_SIZEOF];
     void wrhdlr(int signo);

     signal(SIGUSR1, wrhdlr);

     if ((fd = open("newfile", O_WRONLY | O_CREAT | O_RAW, 0644)) == -1) {
          perror("open (newfile) failed");
          exit(1);
     }

     /* Program populates buffer buf with data here. */

     RECALL_SET(mask, fd);    /* set bit for fd in mask */

     writea(fd, buf, 4096, &wrstat, SIGUSR1);  /* SIGUSR1 sent when
                                                  write completes */

     /* Perform other work here in parallel with I/O completion. */

     recalla(mask);        /* wait for write to complete */

     /* Output data has now vacated buffer buf due to writea. */
}

void wrhdlr(int signo)
```

```
{
     signal(signo, wrhdlr);
     printf("writea wrote %d bytes\n", wrstat.sw_count);
     wrstat.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 `writea` request. The program uses the `recall`(2) system call to wait for the completion of the asynchronous write 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 asynchronous write operation in this example.

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

main()
{
     char buf[4096];
     int fd;
     struct iosw wrstat[1], *statlist[1];

     if ((fd = open("newfile", O_WRONLY | O_CREAT | O_RAW, 0644)) == -1) {
          perror("open (newfile) failed");
          exit(1);
     }

     /* Program populates buffer buf with data here. */

     writea(fd, buf, 4096, &wrstat[0], 0); /* no signal sent when
                                              write completes */
     statlist[0] = &wrstat[0];

     /* Perform other work here in parallel with I/O completion. */

     recall(fd, 1, statlist);              /* wait for write to complete */

     printf("writea wrote %d bytes\n", wrstat[0].sw_count);
     wrstat[0].sw_flag = 0;

     /* Output data has now vacated buffer buf due to writea. */
}
```

SEE ALSO

chmod(2), creat(2), dup(2), fcntl(2), open(2), pause(2), pipe(2), reada(2), recall(2), recalla(2), signal(2), socket(2), ulimit(2), write(2)

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