Section 2: Xinu System Calls

The Xinu operating system kernel consists of a set of run-time procedures to implement operating system services on a microcomputer. The system supports multiple processes, I/O, synchronization based on counting semaphores, preemptive scheduling, and communications with other machines. Each page in this section describes a system routine which can be called by a user process.

Each page describes one system call, giving the number and types of arguments which must be passed to the procedure under the heading "SYNOPSIS" (by giving their declaration in C syntax). The heading "SEE ALSO" suggests the names of other system calls that may be related to the described function. For example, the "SEE ALSO" entry for system call *wait* suggests that the programmer may want to look at the page for *signal* because both routines operate on semaphores.

In general, Xinu blocks processes when requested services are not available. Unless the manual page suggests otherwise, the programmer should assume that the process requesting system services may be delayed until the request can be satisfied. For example, calling *read* may cause an arbitrary delay until data can be obtained from the device. During this delay, rescheduling will probably allow other processes to run.

Only non-blocking system functions should be called from within interrupt handling routines.

NAME

access - determine whether a file is accessible

SYNOPSIS

int access(filename, mode)
char *filename;
char *mode;

DESCRIPTION

Access examines file with name *filename* to determine if it is accessible according to the modes specified in the mode string *mode*. Valid characters in the mode string are **r** (check for read access) **w** (check for write access), **n** (check to see if a new file can be created), and **o** (check to see if file exists). If neither **r** nor **w** is specified, both are assumed.

SEE ALSO

open(2), rename(2), ckmode(3)

NAME
chdsk - login a new floppy disk in DS0

SYNOPSIS
int chdsk(device)
int device;

DESCRIPTION
Chdsk logs in a new floppy disk in *device* (usually DS0). All Xinu floppy disk I/O requires that a floppy disk (Xinu or DOS format) be present in PC drive A: at boot time.

SEE ALSO
access(2), df(4), dir(1), dsk(4), format(2), remove(2), rename(2), stat(1)

NAME

chprio - change the priority of a process

SYNOPSIS

int chprio(pid, newprio)
int pid;
int newprio;

DESCRIPTION

Chprio changes the scheduling priority of process *pid* to *newprio*. Priorities are positive integers. At any instant, the highest priority process that is ready will be running. A set of processes with equal priority is scheduled round-robin.

If the new priority is invalid, or the process id is invalid *chprio* returns SYSERR. Otherwise, it returns the old process priority. It is forbidden to change the priority of the null process, which always remains zero.

SEE ALSO

create(2), getprio(2), resume(2)

NAME

close - device independent close routine

SYNOPSIS

int close(dev)
int dev;

DESCRIPTION

Close will disconnect I/O from the device given by *dev*. It returns SYSERR if *dev* is incorrect, or is not opened for I/O. Otherwise, *close* returns OK.

Each device has a reference count that is incremented by the system call *open* and decremented by the system call *close*. Thus, if the device has been opened **n** times, the user must invoke *close* **n** times to close it.

Normally tty devices like the console do not have to be opened and closed.

SEE ALSO

control(2), getc(2), open(2), putc(2), read(2), seek(2), setdev(2), write(2)

NAME

control - device independent control routine

SYNOPSIS

int control(dev, function, arg1, arg2)
int dev;
int function;
int arg1, arg2;

DESCRIPTION

Control is the mechanism used to send control information to devices and device drivers, or to interrogate their status. (Data normally flows through GETC(2), PUTC(2), READ(2), and WRITE(2).)

Control returns SYSERR if *dev* is incorrect or if the function cannot be performed. The values returned otherwise are device dependent. For example, there is a control function for "tty" devices that returns the number of characters waiting in the input queue.

SEE ALSO

close(2), getc(2), open(2), putc(2), read(2), remove(2), rename(2), seek(2), write(2)

NAME

create - create a new process

SYNOPSIS

int create(caddr, ssize, prio, name, nargs[, argument]*)
char *caddr;
int ssize;
int prio;
char *name;
int nargs;
int argument;

DESCRIPTION

Create creates a new process that will begin execution at location *caddr*, with a stack of *ssize* bytes, initial priority *prio*, and identifying name *name*. *Caddr* should be the address of a procedure or xmain program. If the creation is successful, the (nonnegative) process id of the new process is returned to the caller. The created process is left in the suspended state; it will not begin execution until started by a *resume* command. If the arguments are incorrect, or if there are no free process slots, the value SYSERR is returned. The new process has its own stack, but shares global data with other processes according to the scope rules of C. Private global data can be shared between processes using the *pglob* pointer contained in the process' process table entry (see PGLOB(2)). If the procedure attempts to return, its process will be terminated (see KILL(2)).

The caller can pass a variable number of arguments to the created process which are accessed through formal parameters. The integer *nargs* specifies (in words) how many argument values follow. *Nargs* values from the arguments list will be passed to the created process. The type and number of such arguments is not checked; each is treated as a single machine word. The user is cautioned against passing the address of any dynamically allocated datum to a process because such objects may be deallocated from the creator's run-time stack even though the created process retains a pointer.

SEE ALSO

chprio(2), die(2), kill(2), pglob(2), resume(2)

NAME

die - commit suicide

SYNOPSIS

int die()

DESCRIPTION

Die terminates the current process, removing all record of it from the system. The following actions are performed: all standard I/O devices belonging to the current process are closed, the next-of-kin process (if any) is notified by sending it a message containing the pid of the current process, and any memory allocated with *xmalloc* or *xcalloc* is freed.

Die then calls *resched*, so it never returns to its caller.

SEE ALSO

kill(2), treceive(2), tsend(2), xmalloc(2)

BUGS

Die does not free memory allocated with *getmem*. Such memory must be deallocated with *freemem* before *die* is called. The preferred memory allocation functions for user programs are *xmalloc* and *xcalloc*.

NAME

format - initialize a floppy disk to Xinu format

SYNOPSIS

int format(device, id)
int device;
char *id;

DESCRIPTION

Format is used to format a floppy disk in drive *device* (usually DS0). *Id* is a pointer to an id string.

SEE ALSO

format(1)

NAME

freebuf - free a buffer by returning it to its buffer pool

SYNOPSIS

```
int freebuf(buf)
char *buf;
```

DESCRIPTION

Freebuf returns a previously allocated buffer to its buffer pool, making it available for other processes to use. *Freebuf* returns SYSERR if the buffer address is invalid or if the pool id has been corrupted (this version stores pool ids in the integer preceding the buffer address).

SEE ALSO

bpool(1), getbuf(2), mkpool(2), getmem(2), freemem(2)

NAME

freemem, freestk - deallocate a block of main memory

SYNOPSIS

int freemem(addr, len)
char *addr;
int len;

int freestk(addr, len)
char *addr;
int len;

DESCRIPTION

In either form, *freemem* deallocates a contiguous block of memory previously obtained with the *getmem* or *getstk* system calls GETMEM(2), and returns it to the free list. Argument *addr* specifies the address of the block being deallocated, and argument *len* specifies the length of the block in bytes. For memory allocated by *getstk* argument *addr* specifies the highest word address of the block being deallocated; for memory allocated by *getmem* argument *addr* specifies the lowest word address of the block. These definitions are consistent with the addresses returned by calls to *getstk* and *getmem*. In this version, memory is allocated in multiples of eight bytes to guarantee that sufficient space is available in each block to link it onto the free list. However, the length passed to both *getmem* and *freemem* is rounded automatically, so the user need not be aware of any extra space in the allocated block.

SEE ALSO

freebuf(2), getbuf(2), getmem(2), memstat(1)

NAME

getaddr, getiaddr - obtain the local machine's Internet (IP) address

SYNOPSIS

int getaddr(ip)
IPaddr ip;

int getiaddr(inum, ip)
int inum;
IPaddr ip;

DESCRIPTION

Getaddr obtains the local machine's primary Internet (IP) address and places it in the 4-byte array specified by argument *ip*. Calling *getaddr* may trigger a Reverse Address Resolution Protocol (RARP) broadcast to find the address. If RARP succeeds, the address is kept locally for successive lookup requests. If RARP fails, *getaddr* calls *panic* to halt processing.

Getiaddr behaves as above, except that it obtains the IP address of the network interface specified by *inum*.

SEE ALSO

getname(2), getnet(2)

NAME

getbuf - obtain a buffer from a buffer pool

SYNOPSIS

#include <mem.h>

char *getbuf(poolid)
int poolid;

DESCRIPTION

Getbuf obtains a free buffer from the pool given by argument *poolid*, and returns a pointer to the first word of the buffer. If all buffers in the specified pool are in use, the calling process will be blocked until a buffer becomes available. If the argument *poolid* does not specify a valid pool, *getbuf* returns SYSERR.

SEE ALSO

bpool(1), freebuf(2), getmem(2), freemem(2), mkpool(2)

NAME

getc - device independent character input routine

SYNOPSIS

int getc(dev)
int dev;

DESCRIPTION

Getc will read the next character from the I/O device given by *dev*. It returns SYSERR if *dev* is incorrect. A successful call may return a character (widened to an integer) or the value EOF to denote end of file, depending on the device driver.

SEE ALSO

close(2), control(2), open(2), putc(2), read(2), seek(2), write(2)

BUGS

Not all devices report the end-of-file condition.

NAME

getdev - get the device number from a character string name

SYNOPSIS

int getdev(cp)
char *cp;

DESCRIPTION

If *cp* points to a valid character string device name, *getdev* returns the device number for that device. Otherwise, *getdev* returns SYSERR.

SEE ALSO

getpdev(2), setpdev(2)

NAME

getmem, getstk - get a block of main memory

SYNOPSIS

#include <mem.h>

char *getmem(nbytes)
long nbytes;

char *getstk(nbytes)
long nbytes;

DESCRIPTION

In either form, *getmem* rounds the number of bytes, *nbytes*, to a multiple of 8 bytes, and allocates a block of *nbytes* bytes of memory for the caller. *Getmem* returns the lowest word address in the allocated block; *getstk* returns the highest word address in the allocated block. If less than *nbytes* bytes remain, the call returns SYSERR.

Getmem allocates memory starting with the end of the loaded program. *Getstk* allocates memory from the stack area downward. The routines cooperate so they never allocate overlapping regions.

User programs should use *xmalloc* or *xcalloc* to allocate memory because such memory is automatically freed on process termination.

SEE ALSO

freebuf(2), freemem(2) getbuf(2), memstat(1), xmalloc(2)

BUGS

There is no way to protect memory, so the active stack may write into regions returned by either call; allocations returned by *getstk* are more prone to disaster because they lie closest to the dynamic stack areas of other processes.

NAME

getname, getiname - get the Domain Name of this machine and place it where specified.

SYNOPSIS

int getname(nam)
char *nam;

int getiname(inum, nam)
int inum;
char *nam

DESCRIPTION

Getname obtains the Domain Name of the machine's primary network interface and stores it as a null-terminated string in the array *nam*. *Getname* uses *ip2name* to obtain the name. If the name cannot be resolved, *getname* returns SYSERR, otherwise it returns OK.

Getiname behaves as above, except that it obtains the Domain Name of the network interface specified by *inum*.

SEE ALSO

ip2name(2), name2ip(2)

BUGS

No check is done on the size of array *nam*, so overwriting will occur if the length of a name exceeds the space available in array *nam***.**

NAME

getnet - obtain the network portion of the Internet (IP) address of the local machine's network

SYNOPSIS

int getnet(ip)
IPaddr ip;

int getinet(inum, ip)
int inum;
IPaddr ip;

DESCRIPTION

Getnet obtains the network portion of the Internet (IP) address of the local machine's primary network, and stores it in the 4-byte array specified by argument *ip*. Calling *getnet* may trigger a Reverse Address Resolution Protocol (RARP) broadcast to find the address. If RARP succeeds, the address is kept locally for successive lookup requests. If RARP fails, *getnet* calls panic to halt processing.

Getinet behaves as above, except that it obtains the address of the network interface specified by *inum*.

SEE ALSO

getaddr(2), getname(2), getnet(2)

NAME

getnok - get next-of-kin of a given process

SYNOPSIS

int getnok(pid)
int pid;

DESCRIPTION

If *pid* is valid, *getnok* returns the pid of the next-of-kin process of process *pid*. Otherwise *getnok* returns SYSERR.

When a process dies, its next-of-kin process is sent a message containing the pid of the deceased process.

SEE ALSO

die(2), setnok(2)

NAME

getpdev - return device descriptor to which a process' standard I/O maps

SYNOPSIS

```
int getpdev(pid, siodev)
int pid;
int siodev;
```

DESCRIPTION

Getpdev returns the device descriptor of the device to which a process' standard I/O connects. Argument *pid* specifies a process, and argument *siodev* specifies either the standard input device, the standard output device, or the standard error device. *Getpdev* allows a process to identify the actual device being used for I/O, even if the process inherits the device from the shell.

SEE ALSO

open(2), setpdev(2)

NAME

getpid - return the process id of the currently running process

SYNOPSIS

int getpid()

DESCRIPTION

Getpid returns the process id of the currently executing process. It is necessary to be able to identify one's self in order to perform some operations (e.g., change one's scheduling priority).

NAME

getprio - return the scheduling priority of a given process

SYNOPSIS

int getprio(pid)
int pid;

DESCRIPTION

Getprio returns the scheduling priority of process pid. If *pid* is invalid, *getprio* returns SYSERR.

SEE ALSO

chprio(2)

NAME

getstdi, getstdo - return the device currently assigned as STDIN or STDOUT

SYNOPSIS

int getstdi()

int getstdo()

DESCRIPTION

Getstdi returns the device currently assigned as STDIN. *Getstdo* returns the device currently assigned as STDOUT.

SEE ALSO

getpdev(2), setpdev(2)

BUGS

There is no function which returns the device currently assigned as STDERR.

NAME

gettime - obtain the current local time in seconds past the epoch date

SYNOPSIS

int gettime(timvar)
long *timvar;

DESCRIPTION

Gettime obtains the local time measured in seconds past the epoch date, and places it in the longword pointed to by argument *timvar*. The epoch is taken to be zero seconds past Jan 1, 1970.

The correct time is usually kept by the Xinu real-time clock, but *gettime* may obtain the time from the DOS clock if the local time has not been initialized.

If *gettime* cannot obtain the current time, it returns SYSERR to the caller. Otherwise, *gettime* returns OK.

SEE ALSO

getutim(2)

BUGS

Local time computation does not take daylight savings into account. The local clock may drift, especially under heavy CPU activity or activities that require the operating system to mask interrupts for extended periods.

NAME

getutim - obtain current universal time in seconds past the epoch

SYNOPSIS

int getutim(timvar)
long *timvar;

DESCRIPTION

Getutim obtains the current time measured in seconds past the epoch date, and places it in the long word pointed to by argument *timvar*. The correct time is usually kept by the real-time clock, but *gettime* may obtain the time from the DOS clock if the local time has not been initialized.

The epoch is taken to be zero seconds past Jan 1, 1970. Universal time, formerly called Greenwich Mean Time, is the mean solar time of the meridian in Greenwich, England, and is used throughout the world as a standard for measuring time.

If *getutim* cannot obtain the current time, it returns SYSERR to the caller. Otherwise, *getutim* returns OK.

SEE ALSO

gettime(2)

BUGS

The local clock may drift, especially under heavy CPU activity or activities that require the operating system to mask interrupts for extended periods.

NAME

halt - halt Xinu and return to DOS

SYNOPSIS

halt()

DESCRIPTION

Halt aborts processing and returns to DOS. Because *halt* causes all processes and devices to cease execution, it should only be called in emergencies. Internally, *halt* calls the DOS Terminate Function (0x4c).

The correct way to terminate Xinu is by calling function *xdone*.

SEE ALSO

xdone(2)

NAME

immortal - make a process immortal

SYNOPSIS

int immortal(pid)
int pid;

DESCRIPTION

An immortal process can only commit suicide, it cannot be killed by other processes. Invoking *immortal* with a valid, non-zero *pid* will make process *pid* immortal and OK is returned. SYSERR is returned if *pid* is invalid. Invoking *immortal* with a *pid* of zero will make the current process mortal and OK is returned.

SEE ALSO

kill(2)

NAME

init - device independent initialization routine

SYNOPSIS

int init(dev, flag)
int dev;
int flag;

DESCRIPTION

If *flag* is non-zero, *init* will initialize the device given by *dev*. The actual intialization of a particular device is performed by the function specified in the *dvinit* field of the *devsw* structure for that device. That function should save state information for configurable hardware devices (such as the serial ports and network interfaces) so that device state can be restored when Xinu terminates.

If *flag* is zero, *init* will uninitialize the device given by *dev*. Uninitialization is used to return the device to the state it was in before Xinu was executed.

Init returns SYSERR if *dev* is incorrect, or if the device specific initialization routine detects errors. Otherwise, *init* returns OK.

SEE ALSO

control(2), getc(2), open(2), putc(2), read(2), seek(2), setdev(2), write(2)

BUGS

The state of a network interface is contained in both the network adapter and its foreign device driver. When uninitializing, Xinu should restart the foreign device driver to ensure that both the hardware and the driver have consistent state information.

NAME

ip2name - translate an Internet address to a host Domain Name

SYNOPSIS

int ip2name(ip, name)
IPaddr ip;
char *name;

DESCRIPTION

Ip2name accepts a 4-byte Internet (IP) address and returns the Domain Name for that host by consulting a DARPA Domain nameserver to perform the translation. Argument *ip* gives the address of a 4-byte host Internet address to be translated into a name. Argument *name* points to an area of memory in which the domain name will be written. The name is written as a null-terminated string with periods separating domain name components.

Ip2name returns SYSERR if the Internet address is invalid, if the nameserver does not respond, or if the translation fails. It returns OK otherwise.

SEE ALSO

getname(2), getaddr(2)

BUGS

There is no way to specify a long time delay, so name lookup that consults a distant nameserver may timeout due to network delays. Also, there is no way to specify a maximum name size.

NAME

kill - terminate a process

SYNOPSIS

int kill(pid)
int pid;

DESCRIPTION

The behaviour of *kill* depends on whether or not process *pid* is the current process:

If *pid* refers to the current process, that process is terminated by a call to *die* and control passes to the rescheduler.

If *pid* refers to any other valid, mortal process, the *trap function* (see TRECEIVE(2)) for that process is set to function *die*, the *phastrap* boolean in the process table entry for that process is set to TRUE and *kill* returns OK. The trap function will be executed when that process next becomes the current process.

If *pid* refers to a valid, immortal process, a TMSGKILL message is sent to that process and *kill* returns 0. Process *pid* can decide on a course of action when it receives the TMSGKILL message.

If *pid* is invalid, *kill* returns SYSERR.

SEE ALSO

create(2), die(2), immortal(2), kill(1), setdev(2), setnok(2), treceive(2), tsend(2), unsleep(2), xmalloc(2)

BUGS

Kill will not terminate a mortal process in the suspended state.

NAME

mark, unmarked - set and check initialization marks efficiently

SYNOPSIS

```
#include <mark.h>

int mark(mk)
MARKER mk;

int unmarked(mk)
MARKER mk;
```

DESCRIPTION

Mark sets *mk* to "initialized", and records its location in the system. It returns 0 if the location is already marked, OK if the marking was successful, and SYSERR if there are too many marked locations.

Unmarked checks the contents and location of *mk* to see if it has been previously marked with the *mark* procedure. It returns OK if and only if *mk* has not been marked, 0 otherwise. The key is that they work correctly after a reboot, no matter what was left in the marked locations when the system stopped.

Both *mark* and *unmarked* operate efficiently (in a few instructions) to correctly determine whether a location has been marked. They are most useful for creating self-initializing procedures when the system will be restarted. Both the value in *mk* as well as its location are used to tell if it has been marked.

Memory marking can be eliminated from Xinu by removing the definition of the symbol MEMMARK from the Configuration file. Self-initializing library routines may require manual initialization if MEMMARK is disabled.

BUGS

Mark does not verify that the location given lies in the static data area before marking it; to avoid having the system retain marks for locations on the stack after procedure exit, do not mark automatic variables.

NAME

mkdir, mkdirs - make an MSDOS directory or directory tree

SYNOPSIS

int mkdir(name)
char *name;

int mkdirs(name)
char *name;

DESCRIPTION

Mkdir creates the directory specified in string *name* on the device determined by NAMESPACE mapping. *Mkdirs* performs the same operation for a directory tree. Xinu can only create directories on MSDOS disk devices.

SEE ALSO

dos(4), nam(4), nammap(2), rmdir(2)

NAME

mkpool - create a buffer pool

SYNOPSIS

int mkpool(bufsiz, numbufs)
int bufsiz;
int numbufs;

DESCRIPTION

Mkpool creates a pool of *numbufs* buffers, each of size *bufsiz*, and returns an integer identifying the pool. If no more pools can be created, or if the arguments are incorrect, *mkpool* returns SYSERR.

SEE ALSO

bpool(1), getbuf(2), freebuf(2)

BUGS

At present there is no way to reclaim space from buffer pools once they are no longer needed. Buffer pools should only be used by background processes such as daemons, and not by transient processes started by a shell.

NAME

mount - add a prefix mapping to the namespace

SYNOPSIS

int mount(prefix, dev, replace)
char *prefix;
int dev;
char *replace;

DESCRIPTION

Mount adds a prefix mapping to the syntactic namespace, inserting it just prior to the last entry. Argument *prefix* points to a string that contains a null-terminated prefix string, argument *dev* gives the device id of the device to which the prefix maps, and argument *replace* points to a null-terminated replacement string. As a special case, *dev* can specify the value SYSERR to indicate that names matching the prefix cannot be mapped or accessed.

If the namespace table is full, or if the specified prefix or replacement strings exceed the allowed size, *mount* returns SYSERR. Otherwise it returns OK.

SEE ALSO

mount(1), nam(4), nammap(2), namrepl(2), open(2), unmount(2), unmount(1)

NAME

nammap - map a name through the syntactic namespace

SYNOPSIS

int nammap(name, newname)
char *name;
char *newname;

DESCRIPTION

Nammap uses the syntactic namespace to translate a name into a new name and returns the id of a device to which the name maps. Names are mapped iteratively until they map to a device other than the NAMESPACE.

Argument *name* points to a null-terminated string containing the name to be mapped. Argument *newname* points to a string area large enough to hold the mapped version of the name. If successful, *nammap* returns the device id of the device to which the mapping corresponds. Otherwise, it returns SYSERR.

SEE ALSO

mount(2), nam(4), namrepl(2), open(2), pwd(1), remove(2), unmount(2)

BUGS

Nammap writes the mapped name into newname without checking to make sure it fits. There is no way to distinguish errors such as string overflow from names that map to device SYSERR.

NAME

namrepl - replace a name once using the syntactic namespace

SYNOPSIS

int namrepl(name, newname)
char *name;
char *newname;

DESCRIPTION

Namrepl uses the syntactic namespace to translate a name into a new name and returns the id of a device to which the name maps. The name is translated exactly once, independent of the device to which it maps. In particular, *namrepl* will return the device id NAMESPACE without further mapping for those names that map recursively through the syntactic namespace.

Argument *name* points to a null-terminated string containing the name to be mapped, and argument *newname* points to a string area large enough to hold the mapped version of the name. If successful, *namrepl* returns the device id of the device to which the name maps. Otherwise, it returns SYSERR.

SEE ALSO

mount(2), nam(4), nammap(2), open(2), pwd(1), unmount(2)

BUGS

Namrepl writes the mapped name into *newname* without checking to make sure it fits. There is no way to distinguish errors such as string overflow from names that map to device SYSERR.

NAME

open - device independent open routine

SYNOPSIS

int open(dev, name, mode)
int dev;
char *name;
char *mode;

DESCRIPTION

Open will establish connection with the device given by *dev* using the null-terminated string *name* to name an object on that device, and null-terminated string *mode* to specify the access mode for that object. Valid access mode characters include **r** (read), **w** (write), **o** (old), and **n** (new) as specified in ACCESS(2).

Open returns SYSERR if *dev* is incorrect or cannot be opened. If successful, the value returned by *open* depends on the device. Most calls to *open* return a device descriptor that can be used in subsequent calls to *read* or *write*. For example, calling *open* on a disk device with a file name as an argument produces a descriptor by which that file can be accessed.

SEE ALSO

access(2), close(2), control(2), ckmode(3), getdev(2), mount(2), nammap(2), namrepl(2), nam(4), rename(2)

BUGS

Not all devices produce meaningful return values for *open*.

NAME

panic - abort processing due to severe error

SYNOPSIS

int panic(message)
char *message;

DESCRIPTION

Panic will print the character string message on the console, restore interrupt vectors to DOS, uninitialize devices, and abort Xinu.

A related function *_panic* is invoked when any of the following exceptions occur: divide by zero, single step, breakpoint, arithmetic overflow, unknown interrupt. Function *_panic* prints the following information on the console before aborting Xinu: an interrupt type message, machine register contents, the top few stack locations.

SEE ALSO

xdone(2)

NAME

pcount - return the number of messages currently waiting at a port

SYNOPSIS

int pcount(portid)
int portid;

DESCRIPTION

Pcount returns the message count associated with port *portid*.

A positive count **p** means that there are **p** messages available for processing. This count includes the count of messages explicitly in the port and the count of the number of processes which attempted to send messages to the queue but are blocked (because the queue is full). A negative count **p** means that there are **p** processes awaiting messages from the port. A zero count means that there are neither messages waiting nor processes waiting to consume messages.

SEE ALSO

pcreate(2), pdelete(2), preceive(2), preset(2), psend(2)

BUGS

In this version there is no way to distinguish SYSERR (which has value -1) from a legal port count.

NAME
pcreate - create a new port

SYNOPSIS
int pcreate(count)
int count;

DESCRIPTION
Pcreate creates a port with *count* locations for storing message pointers, and returns an integer identifying the port if successful. If no more ports can be allocated, or if count is non-positive, *pcreate* returns SYSERR.

Ports are manipulated with PSEND(2) and PRECEIVE(2). Receiving from a port returns a pointer to a message that was previously sent to the port.

SEE ALSO
pcount(2), pdelete(2), pinit(2), preceive(2), preset(2), psend(2)

NAME

pdelete - delete a port

SYNOPSIS

int pdelete(portid, dispose)
int portid;
int (*dispose)();

DESCRIPTION

Pdelete deallocates port *portid*. The call returns SYSERR if *portid* is illegal or is not currently allocated.

The command has several effects, depending on the state of the port at the time the call is issued. If processes are waiting for messages from portid, they are made ready and return SYSERR to their caller. If messages exist in the port, they are disposed of by procedure *dispose*. If processes are waiting to place messages in the port, they are made ready and given SYSERR indications (just as if the port never existed). *Pdelete* performs the same function of clearing messages and processes from a port as PRESET(2) except that *pdelete* also deallocates the port.

SEE ALSO

pcount(2), pcreate(2), pinit(2), preceive(2), preset(2), psend(2)

NAME

setpglob, getpglob - set or get a private global data pointer

SYNOPSIS

int setpglob(pid, memptr)
int pid;
char *memptr;

char *getpglob(pid)
int pid;

DESCRIPTION

Setpglob writes a pointer to a private global data structure *memptr* into the *pglob* field of the process table entry for process *pid*. If *pid* is the current process, the global variable *_pglob* is also set to *memptr*. If *pid* is invalid, *setpglob* returns SYSERR, otherwise OK is returned.

Getpglob returns the private global data structure pointer contained in the process table entry for process *pid*. If *pid* is invalid, SYSERR is returned. If process *pid* has no private global data structure pointer, NULLPTR is returned.

Private global data structures can be used by groups of processes for interprocess communications.

SEE ALSO

create(2)

NAME

pinit - initialize the ports table at system startup

SYNOPSIS

int pinit(maxmsgs)
int maxmsgs;

DESCRIPTION

Pinit initializes the ports mechanism by clearing the ports table and allocating memory for messages. It should be called only once (usually at system startup). Argument *maxmsgs* specifies an upper bound on the number of simultaneously outstanding messages at all ports.

SEE ALSO

pcreate(2), pdelete(2), psend(2), preceive(2)

NAME

preceive, preceivi - get a message from a port

SYNOPSIS

```
#include <ports.h>

char *preceive(portid)
int portid;

char *preceivi(portid)
int portid;
```

DESCRIPTION

Preceive retrieves the next message from the port *portid*, returning a pointer to the message if successful, or SYSERR if *portid* is invalid. The sender and receiver must agree on a convention for passing the message length.

The calling process is blocked if there are no messages available, and reawakened as soon as a message arrives. The only ways to be released from a port queue are for some other process to send a message to the port with PSEND(2) or for some other process to delete or reset the port with PDELETE(2) or PRESET(2).

Preceivi is a non-blocking version of *preceive*. It returns NULLPTR if there are no messages are available, otherwise it returns a pointer to the next message.

SEE ALSO

pcount(2), pcreate(2), pdelete(2), pinit(2), preset(2), psend(2), receive(2)

NAME

preset - reset a port

SYNOPSIS

int preset(portid, dispose)
int portid;
int (*dispose)();

DESCRIPTION

Preset flushes all messages from a port and releases all processes waiting to send or receive messages. *Preset* returns SYSERR if *portid* is not a valid port id.

Preset has several effects, depending on the state of the port at the time the call is issued. If processes are blocked waiting to receive messages from port *portid*, they are all made ready; each returns SYSERR to caller. If messages are in the port they are disposed of by passing them to function *dispose*. If process are blocked waiting to send messages they are made ready; each returns SYSERR to its caller (as though the port never existed).

The effects of *preset* are the same as PDELETE(2) followed by PCREATE(2), except that the port is not deallocated. The maximum message count remains unaltered.

BUGS

There is no way to change the maximum message count when the port is reset.

SEE ALSO

pcount(2), pcreate(2), pdelete(2), preceive(2), psend(2)

NAME

psend, psendi - send a message to a port

SYNOPSIS

int psend(portid, message)
int portid;
char *message;

DESCRIPTION

Psend adds the pointer message to the port *portid*. If successful, *psend* returns OK; it returns SYSERR if *portid* is invalid. Note that only a pointer, not the entire message, is enqueued, and that *psend* may return to the caller before the receiver has consumed the message.

If the port is full at the time of the call, the sending process will be blocked until space is available in the port for the message.

Psendi is a non-blocking version of *psend*. It returns SYSERR if the port is full or *portid* is invalid. Otherwise, it enqueues the message and returns OK.

SEE ALSO

pcount(2), pcreate(2), pdelete(2), pinit(2), preceive(2), preset(2), send(2)

NAME

putc - device independent character output routine

SYNOPSIS

int putc(dev, ch)
int dev;
char ch;

DESCRIPTION

Putc will write the character *ch* on the I/O device given by *dev*. It returns SYSERR if *dev* is incorrect, OK otherwise.

By convention, *printf* calls *putc* on device STDOUT to write formatted output. Usually, STDOUT refers to the process' output window.

SEE ALSO

close(2), control(2), getc(2), open(2), read(2), seek(2), write(2)

NAME

read - device independent input routine

SYNOPSIS

int read(dev, buffer, numchars)
int dev;
char *buffer;
int numchars;

DESCRIPTION

Read will read up to *numchars* bytes from the I/O device given by *dev*. It returns SYSERR if *dev* is incorrect. It returns the number of characters read if successful. If *numchars* is zero, *read* reads all bytes which are waiting and returns the number of bytes read.

The number of bytes actually returned depends on the device. For example, when reading from a device of type "tty", each *read* normally returns one line. For a disk, however, each *read* returns one block and the argument *numchars* is taken to be the index of the disk block desired.

SEE ALSO

close(2), control(2), getc(2), open(2), putc(2), read(1), seek(2), write(2)

NAME

receive - receive a (longword) message

SYNOPSIS

#include <kernel.h>

long receive()

DESCRIPTION

Receive returns the longword message sent to a process using SEND(2). If no messages are waiting, receive blocks until one appears.

SEE ALSO

preceive(2), recvclr(2), recvtim(2), send(2), sendf(2)

NAME

recvclr - clear incoming message buffer asynchronously

SYNOPSIS

#include <kernel.h>

long recvclr()

DESCRIPTION

A process executes *recvclr* to clear its message buffer of any waiting message in preparation for receiving messages. If a message is waiting, *recvclr* returns it to the caller. If no messages are waiting, *recvclr* returns OK.

SEE ALSO

preceive(2), receive(2), recvtim(2), send(2), sendf(2)

NAME

recvtim - receive a message with timeout

SYNOPSIS

#include <kernel.h>

long recvtim(maxwait)
int maxwait;

DESCRIPTION

Recvtim allows a process to specify a maximum time limit it is willing to wait for a message to arrive. Like RECEIVE(2), *recvtim* blocks the calling process until a message arrives from SEND(2). Argument *maxwait* gives the maximum time to wait for a message, specified in tenths of seconds.

Recvtim returns SYSERR if the argument is incorrect or if no clock is present. It returns integer TIMEOUT if the time limit expires before a message arrives. Otherwise, it returns the message.

SEE ALSO

receive(2), recvclr(2), send(2), sendf(2), sleep10(2), sleep(2), unsleep(2)

BUGS

There is no way to distinguish between messages that contain TIMEOUT or SYSERR and errors reported by *recvtim*.

NAME

remove - remove a file from the file system

SYNOPSIS

int remove(filename, key)
char *filename;
int key;

DESCRIPTION

Remove takes a file name as an argument and destroys the named file (i.e., removes it from the file system). Argument *filename* specifies the name of a file to remove, and the optional argument *key* gives a one-word protection key.

Remove uses the namespace to map the given file name to a new name, and invokes CONTROL(2) on the underlying device to destroy the file. It returns SYSERR if the name is illegal or cannot be mapped to an underlying device. It returns whatever CONTROL(2) returns otherwise.

SEE ALSO

control(2), nammap(2), nam(4)

NAME

rename - change the name of an object, usually a file

SYNOPSIS

int rename(oldname, newname)
char *oldname;
char *newname;

DESCRIPTION

Rename changes the name of an object. Argument *oldname* gives the name of an existing object and argument *newname* gives a new name for that object. *Rename* maps names through the syntactic namespace and then invokes the control function FLRENAME on the underlying device. Note that *rename* does not provide for copying of objects across device boundaries. The intent is to provide only for changing names while objects stay resident on a given device and allow users to implement copy operations separately if desired.

SEE ALSO

access(2), control(2), open(2), nam(4)

NAME

resume - resume a suspended process

SYNOPSIS

int resume(pid)
int pid;

DESCRIPTION

Resume takes process *pid* out of hibernation and allows it to resume execution. If *pid* is invalid or process *pid* is not suspended, *resume* returns SYSERR; otherwise it returns the priority at which the process resumed execution. Only suspended processes may be resumed.

SEE ALSO

chprio(2), create(2), sleep(2), suspend(2), send(2), receive(2)

NAME

rmdir, rmdirs - remove a directory or directory tree from an MSDOS file system

SYNOPSIS

int rmdir(name)
char *name;

int rmdirs(name)
char *name;

DESCRIPTION

Rmdir uses the namespace to map string *name* to a device and directory name. The directory is then removed. *Rmdirs* performs the same operation for a directory tree.

SEE ALSO

mkdir(2)

NAME

scount - return the count associated with a semaphore

SYNOPSIS

int scount(sem)
int sem;

DESCRIPTION

Scount returns the current count associated with semaphore *sem*. A count of negative **p** means that there are **p** processes waiting on the semaphore; a count of positive **p** means that at most **p** more calls to WAIT(2) can occur before a process will be blocked (assuming no intervening sends occur).

SEE ALSO

screate(2), sdelete(2), signal(2), sreset(2), wait(2)

BUGS

In this version, there is no way to distinguish SYSERR from a legal semaphore count of -1.

NAME

screate - create a new semaphore

SYNOPSIS

int screate(count)
int count;

DESCRIPTION

Screate creates a counting semaphore and initializes it to *count*. If successful, *screate* returns the integer identifier of the new semaphore. It returns SYSERR if no more semaphores can be allocated or if *count* is less than zero.

Semaphores are manipulated with WAIT(2) and SIGNAL(2) to synchronize processes and implement mutual exclusion. Waiting causes the semaphore count to be decremented; decrementing a semaphore count past zero causes a process to be blocked. Signaling a semaphore increases its count, freeing a blocked process if one is waiting.

SEE ALSO

scount(2), sdelete(2), signal(2), sreset(2), wait(2)

NAME

sdelete - delete a semaphore

SYNOPSIS

int sdelete(sem)
int sem;

DESCRIPTION

Sdelete removes semaphore *sem* from the system and returns processes that were waiting for it to the ready state. The call returns SYSERR if *sem* is not a legal semaphore; it returns OK if the deletion was successful.

SEE ALSO

scount(2), screate(2), signal(2), sreset(2), wait(2)

NAME

seek - device independent position seeking routine

SYNOPSIS

int seek(dev, position)
int dev;
long position;

DESCRIPTION

Seek will position the device given by *dev* after the *position* byte. It returns SYSERR if *dev* is incorrect, or if it is not possible to position *dev* as specified.

Seek should only be used with disk file devices.

Note that the *position* argument is declared long rather than int.

SEE ALSO

close(2), control(2), getc(2), open(2), putc(2), read(2), write(2)

NAME

send - send a (longword) message to a process

SYNOPSIS

int send(pid, msg)
int pid;
long msg;

DESCRIPTION

Send sends the longword message *msg* to the process with id *pid*. A process may have at most one outstanding message that has not been received.

Send returns SYSERR if *pid* is invalid or if the process already has a message waiting that has not been received. Otherwise, it sends the message and returns OK.

SEE ALSO

preceive(2), psend(2), receive(2), recvtim(2), recvclr(2), sendf(2)

NAME

sendf - send a (longword) message to a process, forcing delivery

SYNOPSIS

int sendf(pid, msg)
int pid;
long msg;

DESCRIPTION

Sendf sends the longword message *msg* to the process with id *pid*. A process may have at most one outstanding message that has not been received. *Sendf* returns SYSERR if process id *pid* is invalid. Otherwise, it returns OK.

SEE ALSO

preceive(2), psend(2), receive(2), recvtim(2), recvclr(2), send(2)

NAME

setnok - set next-of-kin for a specified process

SYNOPSIS

int setnok(nok, pid)
int nok;
int pid;

DESCRIPTION

Setnok sets *nok* to be the next-of-kin for process *pid* by recording *nok* in the process table entry for process *pid*. A call to *setnok* overwrites any previous information in the process table entry.

The next-of-kin for a process P is another process, Q, that the system notifies when P dies (i.e., exits). Notification consists of a message sent to Q containing only the process id, P.

Both arguments to *setnok* must be valid process ids. *Setnok* returns SYSERR to report errors in its arguments, and OK otherwise.

SEE ALSO

getnok(2), kill(2)

BUGS

There is no check to ensure that the next-of-kin remains in existence between the call to *setnok* and the termination of a process.

NAME

setpdev - set the standard input and output device ids for a process

SYNOPSIS

int setpdev(pid, siodev, dev)
int pid;
int siodev;
int dev;

DESCRIPTION

Setpdev associates standard I/O device *siodev* with *dev* in the process table entry for process *pid* so the system will automatically close the device when the process exits. It is used primarily by the shell to record the process' standard input, standard output, and standard error device ids.

SEE ALSO

close(2), getpdev(2), kill(2), sio(4)

NAME

signal, signaln - signal a semaphore

SYNOPSIS

int signal(sem)
int sem;

int signaln(sem, count)
int sem;
int count;

DESCRIPTION

In either form, *signal* signals semaphore *sem* and returns SYSERR if the semaphore does not exist, OK otherwise. The form *signal* increments the count of *sem* by 1 and frees the next process if any are waiting. The form *signaln* increments the semaphore by *count* and frees up to *count* processes if that many are waiting. Note that *signaln(sem, x)* is equivalent to executing *signal(sem)* **x** times.

SEE ALSO

scount(2), screate(2), sdelete(2), sreset(2), wait(2)

NAME

sleep, sleep10, sleept - go to sleep for a specified time

SYNOPSIS

int sleep(secs)
int secs;

int sleep10(tenths)
int tenths;

int sleept(ticks)
int ticks;

DESCRIPTION

In any form, *sleep* causes the current process to delay for a specified time and then resume. The form *sleep* expects the delay to be given in an integral number of seconds; it is most useful for longer delays. The form *sleep10* expects the delay to be given in an integral number of tenths of seconds; it is most useful for short delays. The form *sleept* expects the delay to be given in an integral number of system ticks, it is also most useful for short delays.

All forms return SYSERR if the argument is negative or if the real time clock is not enabled on the processor. Otherwise they delay for the specified time and return OK.

SEE ALSO

resume(1), recvtim(2), sleep(1), suspend(2), unsleep(2)

BUGS

The maximum sleep is 32767 seconds (about 546 minutes, or 9.1 hours). *Sleep* guarantees a lower bound on delay, but since the system may delay processing of interrupts at times, *sleep* cannot guarantee an upper bound.

NAME

sreset - reset semaphore count

SYNOPSIS

int sreset(sem, count)
int sem;
int count;

DESCRIPTION

Sreset frees processes in the queue for semaphore *sem*, and resets its count to *count*. This corresponds to the operations of *sdelete(sem)* and *sem=screate(count)*, except that it guarantees that the semaphore id *sem* does not change. *Sreset* returns SYSERR if *sem* is not a valid semaphore id. The current count in a semaphore does not affect resetting it.

SEE ALSO

scount(2), screate(2), sdelete(2), signal(2), wait(2)

NAME

suspend - suspend a process to keep it from executing

SYNOPSIS

int suspend(pid)
int pid;

DESCRIPTION

Suspend places process *pid* in a state of hibernation. If *pid* is illegal, or the process is not currently running or on the ready list, *suspend* returns SYSERR. Otherwise it returns the priority of the suspended process. A process may suspend itself, in which case the call returns the priority at which the process is resumed.

Note that hibernation differs from sleeping because a hibernating process can remain on I/O or semaphore queues. A process can put another into hibernation; a process can only put itself to sleep.

SEE ALSO

resume(2), sleep(2), send(2), receive(2)

NAME

treceive - establish a trap procedure

SYNOPSIS

int treceive(fp)
int (*fp)();

DESCRIPTION

Treceive sets the trap function for the current process to *fp*. A subsequent *tsend* to a process will cause the trap function to be executed when that process next becomes the current process.

SEE ALSO

die(2), kill(2), tsend(2)

NAME

tsend - trap another process

SYNOPSIS

int tsend(pid, arg)
int pid;
int arg;

DESCRIPTION

If *pid* is the current process, *tsend* clears *phastrap*, immediately executes the trap function (with argument *arg*) for that process, and then returns OK. Otherwise, if *pid* is valid and process *pid* has no trap set, the argument *arg* is deposited, *phastrap* is set TRUE, and *tsend* returns OK. In all other cases, *tsend* returns SYSERR and no other action is taken.

If a process has *phastrap* equal to TRUE, *resched* will clear *phastrap* and call the trap function when that process next becomes the current process.

SEE ALSO

die(2), kill(2), treceive(2)

NAME

unmount - remove an entry from the syntactic namespace mapping table

SYNOPSIS

int unmount(prefix)
char *prefix;

DESCRIPTION

Unmount searches the syntactic namespace mapping table and removes the mapping which has a prefix equal to the null-terminated string *prefix*. If no such entry exists, *unmount* returns SYSERR. Otherwise, it returns OK.

SEE ALSO

mount(1), mount(2), nam(4), nammap(2), namrepl(2), unmount(1)

NAME

unsleep - remove a sleeping process from the clock queue prematurely

SYNOPSIS

int unsleep(pid)
int pid;

DESCRIPTION

Unsleep allows one process to take another out of the sleeping state before the time limit has expired. Usually, only system routines like RECVTIM(2) and KILL(2) call *unsleep*. User-level processes can avoid using *unsleep* by arranging processes to cooperate using message passing primitives.

SEE ALSO

sleep(2), kill(2), recvtim(2)

NAME

wait - block and wait until semaphore signalled

SYNOPSIS

int wait(sem)
int sem;

DESCRIPTION

Wait decrements the count of semaphore *sem*, blocking the calling process if the count goes negative by enqueuing it in the queue for *sem*. The only ways to get free from a semaphore queue are for some other process to signal the semaphore, or for some other process to delete or reset the semaphore. *Wait* and SIGNAL(2) are the two basic synchronization primitives in the system.

Wait returns SYSERR if *sem* is invalid. Otherwise, it returns OK once freed from the queue.

SEE ALSO

scount(2), screate(2), sdelete(2), signal(2), sreset(2)

NAME

write - write a sequence of characters from a buffer

SYNOPSIS

int write(dev, buf, count)
int dev;
char *buf;
int count;

DESCRIPTION

Write writes *count* characters to the I/O device given by *dev*, from sequential locations of the buffer, *buf*. *Write* returns SYSERR if *dev* or *count* is invalid, OK for a successful write. *Write* normally returns when it is safe for the user to change the contents of the buffer. For some devices this means *write* will wait for I/O to complete before returning. On other devices, the data is copied into a kernel buffer and the write returns while it is being transferred.

SEE ALSO

close(2), control(2), getc(2), open(2), putc(2), read(2), seek(2)

BUGS

Write may not have exclusive use of the I/O device, so output from other processes may be mixed in.

NAME

xdone - print system termination message and terminate Xinu

SYNOPSIS

xdone()

DESCRIPTION

Xdone gracefully terminates Xinu by flushing all disk buffers, closing all local files, uninitializing all devices, restoring remapped interrupt vectors, printing a termination message, and executing *halt* to return to the underlying system.

SEE ALSO

dos(1), halt(2), init(2)

NAME

xfree - free memory allocated with xmalloc or xcalloc

SYNOPSIS

int xfree(addr)
char *addr;

DESCRIPTION

Xfree will free a memory block allocated with *xmalloc* or *xcalloc*. *Xfree* returns SYSERR if *addr* does not point to a previously allocated memory block, otherwise OK is returned. If the memory linkages are corrupt, *xfree* calls *panic* to abort Xinu.

SEE ALSO

xmalloc(2)

NAME

xmalloc, xcalloc - allocate memory using a malloc-like technique

SYNOPSIS

```
#include <mem.h>

char *xmalloc(nbytes)
int nbytes;

char *xcalloc(nbytes)
int nbytes;
```

DESCRIPTION

Xmalloc uses a malloc-like technique to allocate memory. Memory is allocated on paragraph boundaries and chained into a linked list for later automatic deallocation when the requesting process is terminated. *Xmalloc* returns NULLPTR if the memory request cannot be satisfied. Otherwise, *xmalloc* returns a pointer to the allocated memory.

Xcalloc behaves exactly like *xmalloc*, except that allocated memory is cleared.

SEE ALSO

kill(2), xfree(2)