INTRODUCTION TO SYSTEM CALLS

Section II of this manual lists all the entries into the system. In most cases two calling sequences are specified, one of which is usable from assembly language, and the other from C. Most of these calls have an error return. From assembly language an erroneous call is always indicated by turning on the c-bit of the condition codes. The presence of an error is most easily tested by the instructions *bes* and *bec* (''branch on error set (or clear)''). These are synonyms for the *bcs* and *bcc* instructions.

From C, an error condition is indicated by an otherwise impossible returned value. Almost always this is −1; the individual sections specify the details.

In both cases an error number is also available. In assembly language, this number is returned in r0 on erroneous calls. From C, the external variable *errno* is set to the error number. *Errno* is not cleared on successful calls, so it should be tested only after an error has occurred. There is a table of messages associated with each error, and a routine for printing the message. See *perror (III)*.

The possible error numbers are not recited with each writeup in section II, since many errors are possible for most of the calls. Here is a list of the error numbers, their names inside the system (for the benefit of system-readers), and the messages available using *perror*. A short explanation is also provided.

0      –                (unused)

1      EPERM          Not owner and not super-user
       Typically this error indicates an attempt to modify a file in some way forbidden except to its owner. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2      ENOENT         No such file or directory
       This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3      ESRCH          No such process
       The process whose number was given to *signal* does not exist, or is already dead.

4      EINTR          Interrupted system call
       An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

5      EIO            I/O error
       Some physical I/O error occurred during a *read* or *write*. This error may in some cases occur on a call following the one to which it actually applies.

6      ENXIO          No such device or address
       I/O on a special file refers to a subdevice which does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not dialled in or no disk pack is loaded on a drive.

7      E2BIG          Arg list too long
       An argument list longer than 512 bytes (counting the null at the end of each argument) is presented to *exec*.

8      ENOEXEC        Exec format error
       A request is made to execute a file which, although it has the appropriate permissions, does not start with one of the magic numbers 407 or 410.

9       EBADF          Bad file number
        Either a file descriptor refers to no open file, or a read (resp. write) request is made to a file which
        is open only for writing (resp. reading).

10      ECHILD         No children
        *Wait* and the process has no living or unwaited-for children.

11      EAGAIN         No more processes
        In a *fork,* the system's process table is full and no more processes can for the moment be created.

12      ENOMEM         Not enough core
        During an *exec* or *break,* a program asks for more core than the system is able to supply.  This is
        not a temporary condition; the maximum core size is a system parameter.  The error may also oc-
        cur if the arrangement of text, data, and stack segments is such as to require more than the existing
        8 segmentation registers.

13      EACCES         Permission denied
        An attempt was made to access a file in a way forbidden by the protection system.

14      –              (unused)

15      ENOTBLK        Block device required
        A plain file was mentioned where a block device was required, e.g. in *mount.*

16      EBUSY          Mount device busy
        An attempt to mount a device that was already mounted or an attempt was made to dismount a de-
        vice on which there is an open file or some process's current directory.

17      EEXIST         File exists
        An existing file was mentioned in an inappropriate context, e.g. *link.*

18      EXDEV          Cross-device link
        A link to a file on another device was attempted.

19      ENODEV         No such device
        An attempt was made to apply an inappropriate system call to a device; e.g. read a write-only de-
        vice.

20      ENOTDIR        Not a directory
        A non-directory was specified where a directory is required, for example in a path name or as an
        argument to *chdir.*

21      EISDIR         Is a directory
        An attempt to write on a directory.

22      EINVAL         Invalid argument
        Some invalid argument: currently, dismounting a non-mounted device, mentioning an unknown
        signal in *signal,* and giving an unknown request in *stty* to the TIU special file.

23      ENFILE         File table overflow
        The system's table of open files is full, and temporarily no more *opens* can be accepted.

24      EMFILE         Too many open files
        Only 15 files can be open per process.

25      ENOTTY         Not a typewriter
        The file mentioned in *stty* or *gtty* is not a typewriter or one of the other devices to which these

calls apply.

26 ETXTBSY  Text file busy
An attempt to execute a pure-procedure program which is currently open for writing (or reading!).
Also an attempt to open for writing a pure-procedure program that is being executed.

27 EFBIG  File too large
An attempt to make a file larger than the maximum of 32768 blocks.

28 ENOSPC  No space left on device
During a *write* to an ordinary file, there is no free space left on the device.

29 ESPIPE  Seek on pipe
A *seek* was issued to a pipe.  This error should also be issued for other non-seekable devices.

30 EROFS  Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK  Too many links
An attempt to make more than 127 links to a file.

32 EPIPE  Write on broken pipe
A write on a pipe for which there is no process to read the data.  This condition normally generates
a signal; the error is returned if the signal is ignored.

-

**NAME**

      break, brk, sbrk − change core allocation

**SYNOPSIS**

      (break = 17.)

      **sys break; addr**

      **char \*brk(addr)**

      **char \*sbrk(incr)**

**DESCRIPTION**

      *Break* sets the system's idea of the lowest location not used by the program (called the break) to *addr* (rounded up to the next multiple of 64 bytes).  Locations not less than *addr* and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

      From C, *brk* will set the break to *addr.*  The old break is returned.

      In the alternate entry *sbrk, incr* more bytes are added to the program's data space and a pointer to the start of the new area is returned.

      When a program begins execution via *exec* the break is set at the highest location defined by the program and data storage areas.  Ordinarily, therefore, only programs with growing data areas need to use *break.*

**SEE ALSO**

      exec (II), alloc (III), end (III)

**DIAGNOSTICS**

      The c-bit is set if the program requests more memory than the system limit or if more than 8 segmentation registers would be required to implement the break.  From C, −1 is returned for these errors.

**BUGS**

      Setting the break in the range 0177700 to 0177777 is the same as setting it to zero.

**NAME**

        chdir − change working directory

**SYNOPSIS**

        (chdir = 12.)

        **sys chdir; dirname**

        **chdir(dirname)**
        **char \*dirname;**

**DESCRIPTION**

        *Dirname* is the address of the pathname of a directory, terminated by a null byte. *Chdir* causes
        this directory to become the current working directory.

**SEE ALSO**

        chdir (I)

**DIAGNOSTICS**

        The error bit (c-bit) is set if the given name is not that of a directory or is not readable. From C,
        a −1 returned value indicates an error, 0 indicates success.

-

**NAME**

  sort − sort or merge files

**SYNOPSIS**

  **sort** [ −**abdnrt***x* ] [ +*pos*  [ −*pos* ] ] . . .  [ −**mo** ] [ name ] . . .

**DESCRIPTION**

  *Sort* sorts all the named files together and writes the result on the standard output.  The name '−' means the standard input.  The standard input is also used if no input file names are given.  Thus *sort* may be used as a filter.

  The default sort key is an entire line.  Default ordering is lexicographic in ASCII collating sequence, except that lower-case letters are considered the same as the corresponding upper-case letters.  Non-ASCII bytes are ignored.  The ordering is affected by the flags −**abdnrt**, one or more of which may appear:

  **a**  Do not map lower case letters.

  **b**  Leading blanks (spaces and tabs) are not included in fields.

  **d**  'Dictionary' order: only letters, digits and blanks are significant in ASCII comparisons.

  **n**  An initial numeric string, consisting of optional minus sign, digits and optionally included decimal point, is sorted by arithmetic value.

  **r**  Reverse the sense of comparisons.

  **t***x*  Tab character between fields is *x.*

  Selected parts of the line, specified by +*pos* and −*pos*, may be used as sort keys.  *Pos* has the form *m.n,* where *m* specifies a number of fields to skip, and *n* a number of characters to skip further into the next field.  A missing is taken to be 0.  +*pos* denotes the beginning of the key; −*pos* denotes the first position after the key (end of line by default).  The ordering rule may be overridden for a particular key by appending one or more of the flags **abdnr** to +*pos*.

  When no tab character has been specified, a field consists of nonblanks and any preceding blanks.  Under the −**b** flag, leading blanks are excluded from a field.  When a tab character has been specified, a field is a string ending with a tab character.

  When keys are specified, later keys are compared only when all earlier ones compare equal.  Lines that compare equal are ordered with all bytes significant.

  These flag arguments are also understood:

  −**m** Merge only, the input files are already sorted.

  −**o** The next argument is the name of an output file to use instead of the standard output.  This file may be the same as one of the inputs, except under the merge flag −**m**.

**FILES**

  /usr/tmp/stm???

**NAME**

chown − change owner and group of a file

**SYNOPSIS**

(chmod = 16.)

**sys chown; name; owner**

**chown(name, owner)**
**char \*name;**

**DESCRIPTION**

The file whose name is given by the null-terminated string pointed to by *name* has its owner and group changed to the low and high bytes of *owner* respectively. Only the super-user may execute this call, because if users were able to give files away, they could defeat the (nonexistent) file-space accounting procedures.

**SEE ALSO**

chown (VIII), chgrp (VIII), passwd (V)

**DIAGNOSTICS**

The error bit (c-bit) is set on illegal owner changes. From C a −1 returned value indicates error, 0 indicates success.

**NAME**

close − close a file

**SYNOPSIS**

(close = 6.)
(file descriptor in r0)
**sys close**

**close(fildes)**

**DESCRIPTION**

Given a file descriptor such as returned from an *open, creat,* or *pipe* call, *close* closes the associated file.  A close of all files is automatic on *exit,* but since processes are limited to 15 simultaneously open files, *close* is necessary for programs which deal with many files.

**SEE ALSO**

creat (II), open (II), pipe (II)

**DIAGNOSTICS**

The error bit (c-bit) is set for an unknown file descriptor.  From C a −1 indicates an error, 0 indicates success.

**NAME**

        creat − create a new file

**SYNOPSIS**

        (creat = 8.)
        **sys creat; name; mode**
        (file descriptor in r0)

        **creat(name, mode)**
        **char \*name;**

**DESCRIPTION**

        *Creat* creates a new file or prepares to rewrite an existing file called *name,* given as the address of a null-terminated string.  If the file did not exist, it is given mode *mode.*  See *chmod* (II) for the construction of the *mode* argument.

        If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length.

        The file is also opened for writing, and its file descriptor is returned (in r0).

        The *mode* given is arbitrary; it need not allow writing.  This feature is used by programs which deal with temporary files of fixed names.  The creation is done with a mode that forbids writing.  Then if a second instance of the program attempts a *creat,* an error is returned and the program knows that the name is unusable for the moment.

**SEE ALSO**

        write (II), close (II), stat (II)

**DIAGNOSTICS**

        The error bit (c-bit) may be set if: a needed directory is not searchable; the file does not exist and the directory in which it is to be created is not writable; the file does exist and is unwritable; the file is a directory; there are already too many files open.

        From C, a −1 return indicates an error.

**NAME**

csw − read console switches

**SYNOPSIS**

(csw = 38.; not in assembler)
**sys        csw**

**getcsw( )**

**DESCRIPTION**

The setting of the console switches is returned (in r0).

**NAME**

dup − duplicate an open file descriptor

**SYNOPSIS**

(dup = 41.; not in assembler)
(file descriptor in r0)
**sys dup**

**dup(fildes)**
**int fildes;**

**DESCRIPTION**

Given a file descriptor returned from an *open, pipe,* or *creat* call, *dup* will allocate another file descriptor synonymous with the original.  The new file descriptor is returned in r0.

*Dup* is used more to reassign the value of file descriptors than to genuinely duplicate a file descriptor.  Since the algorithm to allocate file descriptors returns the lowest available value, combinations of *dup* and *close* can be used to manipulate file descriptors in a general way.  This is handy for manipulating standard input and/or standard output.

**SEE ALSO**

creat (II), open (II), close (II), pipe (II)

**DIAGNOSTICS**

The error bit (c-bit) is set if: the given file descriptor is invalid; there are already too many open files.  From C, a −1 returned value indicates an error.

-

**NAME**

exec, execl, execv − execute a file

**SYNOPSIS**

(exec = 11.)

**sys exec; name; args**

**...**

**name: <...\0>**

**...**

**args: arg0; arg1; ...; 0**

**arg0: <...\0>**

**arg1: <...\0>**

  **...**

**execl(name, arg0, arg1, ..., argn, 0)**

**char \*name, \*arg0, \*arg1, ..., \*argn;**

**execv(name, argv)**

**char \*name;**

**char \*argv[ ];**

**DESCRIPTION**

*Exec* overlays the calling process with the named file, then transfers to the beginning of the core image of the file. There can be no return from the file; the calling core image is lost.

Files remain open across *exec* calls. Ignored signals remain ignored across *exec,* but signals that are caught are reset to their default values.

Each user has a *real* user ID and group ID and an *effective* user ID and group ID. The real ID identifies the person using the system; the effective ID determines his access privileges. *Exec* changes the effective user and group ID to the owner of the executed file if the file has the ''set-user-ID'' or ''set-group-ID'' modes. The real user ID is not affected.

The form of this call differs somewhat depending on whether it is called from assembly language or C; see below for the C version.

The first argument to *exec* is a pointer to the name of the file to be executed. The second is the address of a null-terminated list of pointers to arguments to be passed to the file. Conventionally, the first argument is the name of the file. Each pointer addresses a string terminated by a null byte.

Once the called file starts execution, the arguments are available as follows. The stack pointer points to a word containing the number of arguments. Just above this number is a list of pointers to the argument strings. The arguments are placed as high as possible in core.

```
  sp→   nargs
        arg0
        ...
        argn
        −1
  arg0:  <arg0\0>
         ...
  argn:  <argn\0>
```

From C, two interfaces are available. *execl* is useful when a known file with known arguments is being called; the arguments to *execl* are the character strings constituting the file and the arguments; as in the basic call, the first argument is conventionally the same as the file name (or its last component). A 0 argument must end the argument list.

The *execv* version is useful when the number of arguments is unknown in advance; the arguments to *execv* are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

When a C program is executed, it is called as follows:

```
main(argc, argv)
int argc;
char **argv;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

*Argv* is not directly usable in another *execv,* since *argv[argc]* is −1 and not 0.

**SEE ALSO**

fork (II)

**DIAGNOSTICS**

If the file cannot be found, if it is not executable, if it does not have a valid header (407, 410, or 411 octal as first word), if maximum memory is exceeded, or if the arguments require more than 512 bytes a return from *exec* constitutes the diagnostic; the error bit (c-bit) is set. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed. From C the returned value is −1.

**BUGS**

Only 512 characters of arguments are allowed.

**NAME**

exit − terminate process

**SYNOPSIS**

(exit = 1.)
(status in r0)
**sys exit**

**exit(status)**
**int status;**

**DESCRIPTION**

*Exit* is the normal means of terminating a process. *Exit* closes all the process's files and notifies the parent process if it is executing a *wait.* The low byte of r0 (resp. the argument to *exit*) is available as status to the parent process.

This call can never return.

**SEE ALSO**

wait (II)

**DIAGNOSTICS**

None.

**NAME**

      fork  −  spawn new process

**SYNOPSIS**

      (fork = 2.)
      **sys fork**
      (new process return)
      (old process return)

      **fork( )**

**DESCRIPTION**

      *Fork* is the only way new processes are created.  The new process's core image is a copy of that of the caller of *fork.*  The only distinction is the return location and the fact that r0 in the old (parent) process contains the process ID of the new (child) process.  This process ID is used by *wait.*

      The two returning processes share all open files that existed before the call.  In particular, this is the way that standard input and output files are passed and also how pipes are set up.

      From C, the child process receives a 0 return, and the parent receives a non-zero number which is the process ID of the child; a return of −1 indicates inability to create a new process.

**SEE ALSO**

      wait (II), exec (II)

**DIAGNOSTICS**

      The error bit (c-bit) is set in the old process if a new process could not be created because of lack of process space.  From C, a return of −1 (not just negative) indicates an error.

**NAME**

        fstat  −  get status of open file

**SYNOPSIS**

        (fstat = 28.)
        (file descriptor in r0)
        **sys fstat; buf**

        **fstat(fildes, buf)**
        **struct inode *buf;**

**DESCRIPTION**

        This call is identical to *stat,* except that it operates on open files instead of files given by name.
        It is most often used to get the status of the standard input and output files, whose names are un-
        known.

**SEE ALSO**

        stat (II)

**DIAGNOSTICS**

        The error bit (c-bit) is set if the file descriptor is unknown; from C, a −1 return indicates an error,
        0 indicates success.

**NAME**

getgid  −  get group identifications

**SYNOPSIS**

(getgid = 47.; not in assembler)
**sys getgid**

**getgid**( )

**DESCRIPTION**

*Getgid* returns a word (in r0), the low byte of which contains the real group ID of the current process.  The high byte contains the effective group ID of the current process.  The real group ID identifies the group of the person who is logged in, in contradistinction to the effective group ID, which determines his access permission at the moment.  It is thus useful to programs which operate using the ''set group ID'' mode, to find out who invoked them.

**SEE ALSO**

setgid (II)

**DIAGNOSTICS**

−

**NAME**

getpid – get process identification

**SYNOPSIS**

(getpid = 20.; not in assembler)
**sys getpid**
(pid in r0)

**getpid( )**

**DESCRIPTION**

*Getpid* returns the process ID of the current process. Most often it is used to generate uniquely-
named temporary files.

**SEE ALSO**

–

**DIAGNOSTICS**

–

**NAME**

        getuid  −  get user identifications

**SYNOPSIS**

        (getuid = 24.)

        **sys getuid**

        **getuid( )**

**DESCRIPTION**

        *Getuid* returns a word (in r0), the low byte of which contains the real user ID of the current process.  The high byte contains the effective user ID of the current process.  The real user ID identifies the person who is logged in, in contradistinction to the effective user ID, which determines his access permission at the moment.  It is thus useful to programs which operate using the ''set user ID'' mode, to find out who invoked them.

**SEE ALSO**

        setuid (II)

**DIAGNOSTICS**

        −

**NAME**

gtty − get typewriter status

**SYNOPSIS**

(gtty = 32.)
(file descriptor in r0)
**sys gtty; arg**
**...**
**arg: .=.+6**

**gtty(fildes, arg)**
**int arg[3];**

**DESCRIPTION**

*Gtty* stores in the three words addressed by *arg* the status of the typewriter whose file descriptor is given in r0 (resp. given as the first argument).  The format is the same as that passed by *stty.*

**SEE ALSO**

stty (II)

**DIAGNOSTICS**

Error bit (c-bit) is set if the file descriptor does not refer to a typewriter.  From C, a −1 value is returned for an error, 0, for a successful call.

**NAME**

indir − indirect system call

**SYNOPSIS**

(indir = 0.; not in assembler)
**sys indir; syscall**

**DESCRIPTION**

The system call at the location *syscall* is executed. Execution resumes after the *indir* call.

The main purpose of *indir* is to allow a program to store arguments in system calls and execute them out of line in the data segment. This preserves the purity of the text segment.

If *indir* is executed indirectly, it is a no-op. If the instruction at the indirect location is not a system call, the executing process will get a fault.

**SEE ALSO**

−

**DIAGNOSTICS**

−

-

**NAME**

        kill  −  send signal to a process

**SYNOPSIS**

        (kill = 37.; not in assembler)
        (process number in r0)
        **sys kill; sig**

        **kill(pid, sig);**

**DESCRIPTION**

        *Kill* sends the signal *sig* to the process specified by the process number in r0.  See signal (II) for a list of signals.

        The sending and receiving processes must have the same effective user ID, otherwise this call is restricted to the super-user.

        If the process number is 0, the signal is sent to all other processes which have the same controlling typewriter and user ID.

        In no case is it possible for a process to kill itself.

**SEE ALSO**

        signal (II), kill (I)

**DIAGNOSTICS**

        The error bit (c-bit) is set if the process does not have the same effective user ID and the user is not super-user, or if the process does not exist.  From C, −1 is returned.

**NAME**

link − link to a file

**SYNOPSIS**

(link = 9.)

**sys link; name1; name2**

**link(name1, name2)**
**char *name1, *name2;**

**DESCRIPTION**

A link to *name1* is created; the link has the name *name2*. Either name may be an arbitrary path name.

**SEE ALSO**

link (I), unlink (II)

**DIAGNOSTICS**

The error bit (c-bit) is set when *name1* cannot be found; when *name2* already exists; when the directory of *name2* cannot be written; when an attempt is made to link to a directory by a user other than the super-user; when an attempt is made to link to a file on another file system; when more than 127 links are made. From C, a −1 return indicates an error, a 0 return indicates success.

**NAME**

      mknod – make a directory or a special file

**SYNOPSIS**

      (mknod = 14.; not in assembler)
      **sys  mknod; name; mode; addr**

      **mknod(name, mode, addr)**
      **char \*name;**

**DESCRIPTION**

      *Mknod* creates a new file whose name is the null-terminated string pointed to by *name.* The mode of the new file (including directory and special file bits) is initialized from *mode.* The first physical address of the file is initialized from *addr.* Note that in the case of a directory, *addr* should be zero. In the case of a special file, *addr* specifies which special file.

      *Mknod* may be invoked only by the super-user.

**SEE ALSO**

      mkdir (I), mknod (VIII), fs (V)

**DIAGNOSTICS**

      Error bit (c-bit) is set if the file already exists or if the user is not the super-user. From C, a −1 value indicates an error.

**NAME**

mount − mount file system

**SYNOPSIS**

(mount = 21.)

**sys  mount; special; name; rwflag**

**mount(special, name, rwflag)**
**char *special, *name;**

**DESCRIPTION**

*Mount* announces to the system that a removable file system has been mounted on the block-structured special file *special;* from now on, references to file *name* will refer to the root file on the newly mounted file system.  *Special* and *name* are pointers to null-terminated strings containing the appropriate path names.

*Name* must exist already. Its old contents are inaccessible while the file system is mounted.

The *rwflag* argument determines whether the file system can be written on; if it is 0 writing is allowed, if non-zero no writing is done.  Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

**SEE ALSO**

mount (VIII), umount (II)

**DIAGNOSTICS**

Error bit (c-bit) set if: *special* is inaccessible or not an appropriate file; *name* does not exist; *special* is already mounted; *name* is in use; there are already too many file systems mounted.

**BUGS**

−

**NAME**

nice – set program priority

**SYNOPSIS**

(nice = 34.)
(priority in r0)
**sys nice**

**nice(priority)**

**DESCRIPTION**

The scheduling *priority* of the process is changed to the argument. Positive priorities get less service than normal; 0 is default. Only the super-user may specify a negative priority. The valid range of *priority* is 20 to −220. The value of 4 is recommended to users who wish to execute long-running programs without flak from the administration.

The effect of this call is passed to a child process by the *fork* system call. The effect can be cancelled by another call to *nice* with a *priority* of 0.

The actual running priority of a process is the *priority* argument plus a number that ranges from 100 to 119 depending on the cpu usage of the process.

**SEE ALSO**

nice (I)

**DIAGNOSTICS**

The error bit (c-bit) is set if the user requests a *priority* outside the range of 0 to 20 and is not the super-user.

-

**NAME**

open − open for reading or writing

**SYNOPSIS**

(open = 5.)
**sys open; name; mode**
(file descriptor in r0)

**open(name, mode)**
**char \*name;**

**DESCRIPTION**

*Open* opens the file *name* for reading (if *mode* is 0), writing (if *mode* is 1) or for both reading and writing (if *mode* is 2). *Name* is the address of a string of ASCII characters representing a path name, terminated by a null character.

The returned file descriptor should be saved for subsequent calls to *read, write,* and *close.*

**SEE ALSO**

creat (II), read (II), write (II), close (II)

**DIAGNOSTICS**

The error bit (c-bit) is set if the file does not exist, if one of the necessary directories does not exist or is unreadable, if the file is not readable (resp. writable), or if too many files are open. From C, a −1 value is returned on an error.

-

**NAME**

pipe − create an interprocess channel

**SYNOPSIS**

(pipe = 42.)
**sys pipe**
(read file descriptor in r0)
(write file descriptor in r1)

**pipe(fildes)**
**int fildes[2];**

**DESCRIPTION**

The *pipe* system call creates an I/O mechanism called a pipe.  The file descriptors returned can
be used in read and write operations.  When the pipe is written using the descriptor returned in r1
(resp. fildes[1]), up to 4096 bytes of data are buffered before the writing process is suspended.  A
read using the descriptor returned in r0 (resp. fildes[0]) will pick up the data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by
subsequent *fork* calls) will pass data through the pipe with *read* and *write* calls.

The Shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors
closed) return an end-of-file.  Write calls under similar conditions generate a fatal signal (signal
(II)); if the signal is ignored, an error is returned on the write.

**SEE ALSO**

sh (I), read (II), write (II), fork (II)

**DIAGNOSTICS**

The error bit (c-bit) is set if too many files are already open.  From C, a −1 returned value indi-
cates an error.  A signal is generated if a write on a pipe with only one end is attempted.

**BUGS**

**NAME**

profil − execution time profile

**SYNOPSIS**

(profil = 44.; not in assembler)
**sys       profil; buff; bufsiz; offset; scale**

**profil(buff, bufsiz, offset, scale)**
**char buff[ ];**
**int bufsiz, offset, scale;**

**DESCRIPTION**

*Buff* points to an area of core whose length (in bytes) is given by *bufsiz.* After this call, the user's program counter (pc) is examined each clock tick (60th second); *offset* is subtracted from it, and the result multiplied by *scale.* If the resulting number corresponds to a word inside *buff,* that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 177777(8) gives a 1-1 mapping of pc's to words in *buff;* 77777(8) maps each pair of instruction words together. 2(8) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is also turned off when an *exec* is executed but remains on in child and parent both after a *fork.*

**SEE ALSO**

monitor (III), prof (I)

**DIAGNOSTICS**

−

**NAME**

ptrace − process trace

**SYNOPSIS**

(ptrace = 26.; not in assembler)

(data in r0)

**sys        ptrace; pid; addr; request**

(value in r0)

**ptrace(request, pid, addr, data);**

**DESCRIPTION**

*Ptrace* provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging, but it should be adaptable for simulation of non-UNIX environments. There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process. A process being traced behaves normally until it encounters some signal whether internally generated like ''illegal instruction'' or externally generated like ''interrupt.'' See signal (II) for the list. Then the traced process enters a stopped state and its parent is notified via *wait* (II). When the child is in the stopped state, its core image can be examined and modified using *ptrace.* If desired, another *ptrace* request can then cause the child either to terminate or to continue, possibly ignoring the signal.

The value of the *request* argument determines the precise action of the call:

0      This request is the only one used by the child process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

1,2    The word in the child process's address space at *addr* is returned (in r0). Request 1 indicates the data space (normally used); 2 indicates the instruction space (when I and D space are separated). *addr* must be even. The child must be stopped. The input *data* is ignored.

3      The word of the system's per-process data area corresponding to *addr* is returned. *Addr* must be even and less than 512. This space contains the registers and other information about the process; its layout corresponds to the *user* structure in the system.

4,5    The given *data* is written at the word in the process's address space corresponding to *addr,* which must be even. No useful value is returned. Request 4 specifies data space (normally used), 5 specifies instruction space. Attempts to write in pure procedure result in termination of the child, instead of going through or causing an error for the parent.

6      The process's system data is written, as it is read with request 3. Only a few locations can be written in this way: the general registers, the floating point status and registers, and certain bits of the processor status word.

7      The *data* argument is taken as a signal number and the child's execution continues as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal which caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop.

8      The traced process terminates.

As indicated, these calls (except for request 0) can be used only when the subject process has stopped. The *wait* call is used to determine when a process stops; in such a case the ''termination'' status returned by *wait* has the value 0177 to indicate stoppage rather than genuine termination.

To forestall possible fraud, *ptrace* inhibits the set-user-id facility on subsequent *exec* (II) calls.

**SEE ALSO**

      wait (II), signal (II), cdb (I)

**DIAGNOSTICS**

      From assembler, the c-bit (error bit) is set on errors; from C, −1 is returned and *errno* has the error code.

**BUGS**

      The request 0 call should be able to specify signals which are to be treated normally and not cause a stop. In this way, for example, programs with simulated floating point (which use ''illegal instruction'' signals at a very high rate) could be efficiently debugged.

      Also, it should be possible to stop a process on occurrence of a system call; in this way a completely controlled environment could be provided.

**NAME**

read – read from file

**SYNOPSIS**

(read = 3.)
(file descriptor in r0)
**sys read; buffer; nbytes**

**read(fildes, buffer, nbytes)**
**char \*buffer;**

**DESCRIPTION**

A file descriptor is a word returned from a successful *open, creat, dup,* or *pipe* call. *Buffer* is the location of *nbytes* contiguous bytes into which the input will be placed. It is not guaranteed that all *nbytes* bytes will be read; for example if the file refers to a typewriter at most one line will be returned. In any event the number of characters read is returned (in r0).

If the returned value is 0, then end-of-file has been reached.

**SEE ALSO**

open (II), creat (II), dup (II), pipe (II)

**DIAGNOSTICS**

As mentioned, 0 is returned when the end of the file has been reached. If the read was otherwise unsuccessful the error bit (c-bit) is set. Many conditions can generate an error: physical I/O errors, bad buffer address, preposterous *nbytes,* file descriptor not that of an input file. From C, a −1 return indicates the error.

**NAME**

seek − move read/write pointer

**SYNOPSIS**

(seek = 19.)
(file descriptor in r0)
**sys seek; offset; ptrname**

**seek(fildes, offset, ptrname)**

**DESCRIPTION**

The file descriptor refers to a file open for reading or writing.  The read (resp. write) pointer for the file is set as follows:

if *ptrname* is 0, the pointer is set to *offset.*

if *ptrname* is 1, the pointer is set to its current location plus *offset.*

if *ptrname* is 2, the pointer is set to the size of the file plus *offset.*

if *ptrname* is 3, 4 or 5, the meaning is as above for 0, 1 and 2 except that the offset is multiplied by 512.

If *ptrname* is 0 or 3, *offset* is unsigned, otherwise it is signed.

**SEE ALSO**

open (II), creat (II)

**DIAGNOSTICS**

The error bit (c-bit) is set for an undefined file descriptor.  From C, a −1 return indicates an error.

-

**NAME**
      setgid − set process group ID

**SYNOPSIS**
      (setgid = 46.; not in assembler)
      (group ID in r0)
      **sys setgid**

      **setgid(gid)**

**DESCRIPTION**
      The group ID of the current process is set to the argument. Both the effective and the real group
      ID are set. This call is only permitted to the super-user or if the argument is the real group ID.

**SEE ALSO**
      getgid (II)

**DIAGNOSTICS**
      Error bit (c-bit) is set as indicated; from C, a −1 value is returned.

**NAME**
  setuid − set process user ID

**SYNOPSIS**
  (setuid = 23.)
  (user ID in r0)
  **sys setuid**

  **setuid(uid)**

**DESCRIPTION**
  The user ID of the current process is set to the argument.  Both the effective and the real user ID
  are set.  This call is only permitted to the super-user or if the argument is the real user ID.

**SEE ALSO**
  getuid (II)

**DIAGNOSTICS**
  Error bit (c-bit) is set as indicated; from C, a −1 value is returned.

**NAME**

signal − catch or ignore signals

**SYNOPSIS**

(signal = 48.)
**sys  signal; sig; label**
(old value in r0)

**signal(sig, func)**
**int (*func)( );**

**DESCRIPTION**

A *signal* is generated by some abnormal event, initiated either by user at a typewriter (quit, inter-
rupt), by a program error (bus error, etc.), or by request of another program (kill).  Normally all
signals cause termination of the receiving process, but this call allows them either to be ignored
or to cause an interrupt to a specified location.  Here is the list of signals:

    1     hangup
    2     interrupt
    3*    quit
    4*    illegal instruction (not reset when caught)
    5*    trace trap (not reset when caught)
    6*    IOT instruction
    7*    EMT instruction
    8*    floating point exception
    9     kill (cannot be caught or ignored)
    10*   bus error
    11*   segmentation violation
    12*   bad argument to system call
    13    write on a pipe with no one to read it

In the assembler call, if *label* is 0, the process is terminated when the signal occurs; this is the
default action.  If *label* is odd, the signal is ignored.  Any other even *label* specifies an address in
the process where an interrupt is simulated.  An RTI or RTT instruction will return from the in-
terrupt.  Except as indicated, a signal is reset to 0 after being caught.  Thus if it is desired to catch
every such signal, the catching routine must issue another *signal* call.

In C, if *func* is 0, the default action for signal *sig* (termination) is reinstated.  If *func* is 1, the sig-
nal is ignored.  If *func* is non-zero and even, it is assumed to be the address of a function entry
point.  When the signal occurs, the function will be called.  A return from the function will con-
tinue the process at the point it was interrupted.  As in the assembler call, *signal* must in general
be called again to catch subsequent signals.

When a caught signal occurs during certain system calls, the call terminates prematurely.  In par-
ticular this can occur during a *read* or *write* on a slow device (like a typewriter; but not a file);
and during or *wait*.  When such a signal occurs, the saved user status is arranged in such a way
that when return from the signal-catching takes place, it will appear that the system call returned
a characteristic error status.  The user's program may then, if it wishes, re-execute the call.

The starred signals in the list above cause a core image if not caught or ignored.

The value of the call is the old action defined for the signal.

After a *fork* (II) the child inherits all signals.  *Exec* (II) resets all caught signals to default action.

**SEE ALSO**

kill (I), kill (II), ptrace (II), reset (III)

**DIAGNOSTICS**

The error bit (c-bit) is set if the given signal is out of range.  In C, a −1 indicates an error; 0 indi-
cates success.

-

**BUGS**

**NAME**

        sleep − stop execution for interval

**SYNOPSIS**

        (sleep = 35.; not in assembler)
        (seconds in r0)
        **sys sleep**

        **sleep(seconds)**

**DESCRIPTION**

        The current process is suspended from execution for the number of seconds specified by the argument.

**SEE ALSO**

        sleep (I)

**DIAGNOSTICS**

        −

**NAME**

        stat − get file status

**SYNOPSIS**

        (stat = 18.)
        **sys stat; name; buf**

        **stat(name, buf)**
        **char *name;**
        **struct inode *buf;**

**DESCRIPTION**

        *Name* points to a null-terminated string naming a file; *buf* is the address of a 36(10) byte buffer
        into which information is placed concerning the file.  It is unnecessary to have any permissions
        at all with respect to the file, but all directories leading to the file must be readable.  After *stat,*
        *buf* has the following structure (starting offset given in bytes):

        struct inode {
                char      minor;                    /* +0: minor device of i-node */
                char      major;                    /* +1: major device */
                int       inumber;                  /* +2 */
                int       flags;                    /* +4: see below */
                char      nlinks;                   /* +6: number of links to file */
                char      uid;                      /* +7: user ID of owner */
                char      gid;                      /* +8: group ID of owner */
                char      size0;                    /* +9: high byte of 24-bit size */
                int       size1;                    /* +10: low word of 24-bit size */
                int       addr[8];                  /* +12: block numbers or device number */
                int       actime[2];                /* +28: time of last access */
                int       modtime[2];               /* +32: time of last modification */
        };

        The flags are as follows:

          100000      i-node is allocated
          060000      2-bit file type:
                  000000      plain file
                  040000      directory
                  020000      character-type special file
                  060000      block-type special file.
          010000      large file
          004000      set user-ID on execution
          002000      set group-ID on execution
          001000      save text image after execution
          000400      read (owner)
          000200      write (owner)
          000100      execute (owner)
          000070      read, write, execute (group)
          000007      read, write, execute (others)

**SEE ALSO**

        ls (I), fstat (II), fs (V)

**DIAGNOSTICS**

        Error bit (c-bit) is set if the file cannot be found.  From C, a −1 return indicates an error.

**NAME**

  stime − set time

**SYNOPSIS**

  (stime = 25.)
  (time in r0-r1)
  **sys stime**

  **stime(tbuf)**
  **int tbuf[2];**

**DESCRIPTION**

  *Stime* sets the system's idea of the time and date.  Time is measured in seconds from 0000 GMT
  Jan 1 1970.  Only the super-user may use this call.

**SEE ALSO**

  date (I), time (II), ctime (III)

**DIAGNOSTICS**

  Error bit (c-bit) set if user is not the super-user.

**NAME**

stty − set mode of typewriter

**SYNOPSIS**

(stty = 31.)
(file descriptor in r0)
**sys stty; arg**

**...**
**arg:  .byte ispeed, ospeed; .byte erase, kill; mode**

**stty(fildes, arg)**
**struct {**
    **char  ispeed, ospeed;**
    **char  erase, kill;**
    **int   mode;**
**} *arg;**

**DESCRIPTION**

*Stty* sets mode bits and character speeds for the typewriter whose file descriptor is passed in r0
(resp. is the first argument to the call).  First, the system delays until the typewriter is quiescent.
The input and output speeds are set from the first two bytes of the argument structure as indi-
cated by the following table, which corresponds to the speeds supported by the DH-11 interface.
If DC-11, DL-11 or KL-11 interfaces are used, impossible speed changes are ignored.

| | |
|---|---|
| 0 | (hang up dataphone) |
| 1 | 50 baud |
| 2 | 75 baud |
| 3 | 110 baud |
| 4 | 134.5 baud |
| 5 | 150 baud |
| 6 | 200 baud |
| 7 | 300 baud |
| 8 | 600 baud |
| 9 | 1200 baud |
| 10 | 1800 baud |
| 11 | 2400 baud |
| 12 | 4800 baud |
| 13 | 9600 baud |
| 14 | External A |
| 15 | External B |

In the current configuration, only 110, 150 and 300 baud are really supported on dial-up lines, in
that the code conversion and line control required for IBM 2741's (134.5 baud) must be imple-
mented by the user's program, and the half-duplex line discipline required for the 202 dataset
(1200 baud) is not supplied.

The next two characters of the argument structure specify the erase and kill characters respec-
tively.  (Defaults are # and @.)

The *mode* contains several bits which determine the system's treatment of the typewriter:

| | |
|---|---|
| 100000 | Select one of two algorithms for backspace delays |
| 040000 | Select one of two algorithms for form-feed and vertical-tab delays |
| 030000 | Select one of four algorithms for carriage-return delays |
| 006000 | Select one of four algorithms for tab delays |
| 001400 | Select one of four algorithms for new-line delays |
| 000200 | even parity allowed on input (e. g. for M37s) |
| 000100 | odd parity allowed on input |
| 000040 | raw mode: wake up on all characters |
| 000020 | map CR into LF; echo LF or CR as CR-LF |

000010  echo (full duplex)
000004  map upper case to lower on input (e. g. M33)
000002  echo and print tabs as spaces
000001  hang up (remove 'data terminal ready,' lead CD) after last close

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal.  In all cases a value of 0 indicates no delay.

Backspace delays are currently ignored but will be used for Terminet 300's.

If a form-feed/vertical tab delay is specified, it lasts for about 2 seconds.

Carriage-return delay type 1 lasts about .08 seconds and is suitable for the Terminet 300.  Delay type 2 lasts about .16 seconds and is suitable for the VT05 and the TI 700.  Delay type 3 is unimplemented and is 0.

New-line delay type 1 is dependent on the current column and is tuned for Teletype model 37's. Type 2 is useful for the VT05 and is about .10 seconds.  Type 3 is unimplemented and is 0.

Tab delay type 1 is dependent on the amount of movement and is tuned to the Teletype model 37.  Other types are unimplemented and are 0.

Characters with the wrong parity, as determined by bits 200 and 100, are ignored.

In raw mode, every character is passed immediately to the program without waiting until a full line has been typed.  No erase or kill processing is done; the end-of-file character (EOT), the interrupt character (DEL) and the quit character (FS) are not treated specially.

Mode 020 causes input carriage returns to be turned into new-lines; input of either CR or LF causes LF-CR both to be echoed (used for GE TermiNet 300's and other terminals without the newline function).

The hangup mode 01 causes the line to be disconnected when the last process with the line open closes it or terminates.  It is useful when a port is to be used for some special purpose; for example, if it is associated with an ACU used to place outgoing calls.

This system call is also used with certain special files other than typewriters, but since none of them are part of the standard system the specifications will not be given.

**SEE ALSO**

stty (I), gtty (II)

**DIAGNOSTICS**

The error bit (c-bit) is set if the file descriptor does not refer to a typewriter.  From C, a negative value indicates an error.

**NAME**

        sync − update super-block

**SYNOPSIS**

        (sync = 36.; not in assembler)

        **sys  sync**

**DESCRIPTION**

        *Sync* causes all information in core memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

        It should be used by programs which examine a file system, for example *icheck, df,* etc. It is mandatory before a boot.

**SEE ALSO**

        sync (VIII), update (VIII)

**DIAGNOSTICS**

        −

**NAME**

       time − get date and time

**SYNOPSIS**

       (time = 13.)

       **sys  time**

       **time(tvec)**

       **int tvec[2];**

**DESCRIPTION**

       *Time* returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds.  From *as,* the high order word is in the r0 register and the low order is in r1.  From C, the user-supplied vector is filled in.

**SEE ALSO**

       date (I), stime (II), ctime (III)

**DIAGNOSTICS**

       −

-

**NAME**

        times − get process times

**SYNOPSIS**

        (times = 43.; not in assembler)
        **sys  times; buffer**

        **times(buffer)**
        **struct tbuffer *buffer;**

**DESCRIPTION**

        *Times* returns time-accounting information for the current process and for the terminated child
        processes of the current process.  All times are in 1/60 seconds.

        After the call, the buffer will appear as follows:

        struct tbuffer {
                int       proc_user_time;
                int       proc_system_time;
                int       child_user_time[2];
                int       child_system_time[2];
        };

        The children times are the sum of the children's process times and their children's times.

**SEE ALSO**

        time (I)

**DIAGNOSTICS**

        −

**BUGS**

        The process times should be 32 bits as well.

**NAME**

umount − dismount file system

**SYNOPSIS**

(umount = 22.)

**sys  umount; special**

**DESCRIPTION**

*Umount* announces to the system that special file *special* is no longer to contain a removable file system.  The file associated with the special file reverts to its ordinary interpretation; see *mount* (II).

**SEE ALSO**

umount (VIII), mount (II)

**DIAGNOSTICS**

Error bit (c-bit) set if no file system was mounted on the special file or if there are still active files on the mounted file system.

**NAME**

      unlink − remove directory entry

**SYNOPSIS**

      (unlink = 10.)

      **sys  unlink; name**

      **unlink(name)**

      **char \*name;**

**DESCRIPTION**

      *Name* points to a null-terminated string. *Unlink* removes the entry for the file pointed to by *name* from its directory. If this entry was the last link to the file, the contents of the file are freed and the file is destroyed. If, however, the file was open in any process, the actual destruction is delayed until it is closed, even though the directory entry has disappeared.

**SEE ALSO**

      rm (I), rmdir (I), link (II)

**DIAGNOSTICS**

      The error bit (c-bit) is set to indicate that the file does not exist or that its directory cannot be written. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super-user). From C, a −1 return indicates an error.

**NAME**

        wait − wait for process to terminate

**SYNOPSIS**

        (wait = 7.)
        **sys  wait**
        (process ID in r0)
        (status in r1)

        **wait(status)**
        **int *status;**

**DESCRIPTION**

        *Wait* causes its caller to delay until one of its child processes terminates.  If any child has died since the last *wait,* return is immediate; if there are no children, return is immediate with the error bit set (resp. with a value of −1 returned).  The normal return yields the process ID of the terminated child (in r0).  In the case of several children several *wait* calls are needed to learn of all the deaths.

        If no error is indicated on return, the r1 high byte (resp. the high byte stored into *status* ) contains the low byte of the child process r0 (resp. the argument of *exit* ) when it terminated.  The r1 (resp. *status* ) low byte contains the termination status of the process.  See signal (II) for a list of termination statuses (signals); 0 status indicates normal termination.  A special status (0177) is returned for a stopped process which has not terminated and can be restarted.  See ptrace (II).  If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

        If the parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

**SEE ALSO**

        exit (II), fork (II), signal (II)

**DIAGNOSTICS**

        The error bit (c-bit) is set if there are no children not previously waited for.  From C, a returned value of −1 indicates an error.

**NAME**

write − write on a file

**SYNOPSIS**

(write = 4.)
(file descriptor in r0)
**sys  write; buffer; nbytes**

**write(fildes, buffer, nbytes)**
**char \*buffer;**

**DESCRIPTION**

A file descriptor is a word returned from a successful *open, creat, dup,* or *pipe* call.

*Buffer* is the address of *nbytes* contiguous bytes which are written on the output file. The number of characters actually written is returned (in r0). It should be regarded as an error if this is not the same as requested.

Writes which are multiples of 512 characters long and begin on a 512-byte boundary in the file are more efficient than any others.

**SEE ALSO**

creat (II), open (II), pipe (II)

**DIAGNOSTICS**

The error bit (c-bit) is set on an error: bad descriptor, buffer address, or count; physical I/O errors. From C, a returned value of −1 indicates an error.