1234

Bell Laboratories Cover Sheet for Technical Memorandum

The information contained herein is for the use of employees of Bell Laboratories and is not for publication (see GEI 13.9-3)

Title: **Interprocess Communication Mechanisms in CB-UNIX**

Date: **November 21, 1977**

TM: **77-5223-1**
5223-771121.01TM

Other Keywords: **UNIX File System**
Pipes
Named Pipes
Signals
Semaphores
MAUS
Messages

Author(s)	Location	Extension	Charging Case: 49359-20
J. C. Kaufeld Jr.	**CB 2C-249**	**4522**	Filing Case: **49075-01**

ABSTRACT

A discussion of interprocess communication mechanisms in CB-UNIX[1] is presented. In particular: files, pipes, named pipes, signals, semaphores, MAUS and messages are discussed. For each mechanism, a general explanation is given, the user interface is detailed and then limitations and potential problems are presented. The discussion applies specifically to CB-UNIX, a version of the UNIX operating system developed in Columbus for use in real-time oriented applications. The explanations do not necessarily apply to UNIX operating systems in general, however, most versions of the UNIX operating system have very similar implementations of files, pipes, signals and messages.

ERRATTA

This memo is a corrected copy issued June 19, 1978. This copy corrects a number of misleading comments and outright errors. In addition, references to the CB-UNIX programmers manual are corrected to refer to the correct section number.

Pages Text: 22	Other: 1	Total: 23	
No. Figures: 0	No. Tables: 0	No. Refs.: 5	

CB-UNIX Interprocess Communication Mechanisms

CONTENTS

Bell Laboratories

subject: **Interprocess Communication Mechanisms in CB-UNIX**
Case: **49359-20**
File: **49075-01**

date: **November 21, 1977**

from: **J. C. Kaufeld Jr.**
CB-5223
2C-249 x4522

TM: **77-5223-1**
5223-771121.01TM

ABSTRACT

A discussion of interprocess communication mechanisms in CB-UNIX[1] is presented. In particular: files, pipes, named pipes, signals, semaphores, MAUS and messages are discussed. For each mechanism, a general explanation is given, the user interface is detailed and then limitations and potential problems are presented. The discussion applies specifically to CB-UNIX, a version of the UNIX operating system developed in Columbus for use in real-time oriented applications. The explanations do not necessarily apply to UNIX operating systems in general, however, most versions of the UNIX operating system have very similar implementations of files, pipes, signals and messages.

ERRATTA

This memo is a corrected copy issued June 19, 1978. This copy corrects a number of misleading comments and outright errors. In addition, references to the CB-UNIX programmers manual are corrected to refer to the correct section number.

MEMORANDUM FOR FILE

1. OVERVIEW

1.1 INTRODUCTION

The discussion of interprocess communication mechanisms presented in this paper is directly applicable only to CB-UNIX. CB-UNIX is a version of the UNIX operating system which was developed in Columbus (hence the CB prefix) for use in real-time applications. However, the discussions of files, pipes, signals and messages are representative of UNIX operating systems in general. The discussions of named pipes, semaphores and MAUS apply only to CB-UNIX because those mechanisms, as described, exist only in CB-UNIX. Be cautioned, however, that even for mechanisms which exist in both CB-UNIX and UNIX operating systems in general, the CB-UNIX mechanism is likely to differ subtly because of the emphasis on real-time.

1. UNIX is a Trademark of Bell Laboratories.

As a final disclaimer, this discussion does not include implementation within CB-UNIX, system configuration instructions or an analysis of the overhead involved. Also, it is hoped that the extensives revisions and reviews which this document has undergone have lessened the number of errors and omissions but it's unlikely that this paper is "bug free".

1.2 MECHANISMS

The following is a synopsis of the major interprocess communication mechanisms available in CB-UNIX. Each mechanism is discussed in greater detail later in this paper. References are of the form: name(*section*) - where *section* refers to a tab in the CB-UNIX Programmer's Manual[1].

FILES CB-UNIX files, which are identical to UNIX/TS files[2], can be used as a general purpose data communication mechanism. A process can create or modify files as necessary; given that a second process is aware of the existence of the file, it can read information from the file put there by the first process. In fact, the mere existence of a file can be used as an interprocess flag.

References: open(2A), close(2A), dup(2A), creat(2A), lseek(3A), seek(2A), read(2A), write(2A), stat(2A), link(2A), unlink(2A), chmod(2A), chown(2A), access(2A), smdate(2A)

PIPES A pipe is a unidirectional data transfer mechanism which has a maximum capacity of PIPSIZ+[1] bytes. Pipes are usually used to transfer data between processes but the pipe mechanism allows a single process to use a pipe as, for instance, a temporary storage area with a PIPSIZ byte capacity. Processes which wish to use the same pipe must be related; implying that either one process is the parent of the other processes or that all processes have had a common ancestor. Pipes are created using the *pipe* system call; the file descriptors returned by *pipe* may then be passed by a process to its children using the *call* or *fork* system calls. Once set up, standard CB-UNIX calls such as *read*, *write*, and *close* may be used to manipulate the pipe. Unless all I/O to a pipe is in terms of the same block size, having more than one reader of the pipe is certainly confusing; multiple pipe writers with a single pipe reader establishes a funnel.

References: pipe(2A), close(2A), read(2A), write(2A), fork(2A), call(2A), exec(2A), fstat(2A)

NAMED PIPES Named pipes are similar to pipes except that the processes involved need not be related. Rather than using the *pipe* system call to set up the pipe, the pipe is pre-defined at system configuration time; *open* is used to set up the pipe. Two major differences between pipes and named pipes are that named pipes are never allowed to exceed PIPSIZ bytes and named pipe write requests never sleep.

References: open(2A), close(2A), read(2A), write(2A), fstat(2A)

SIGNALS Enables processes to interrupt each other and to do interrupt processing specific to the signal received. Before the signal is received, the target process can choose to ignore the signal, catch the signal, or let the system handle it as it sees fit (the default action is usually to abort the receiving process).

References: kill(2A), signal(2A), setpgrp(2A), getpid(2A), getppid(2A), sleep(2A), alarm(2A), reset(2A)

SEMAPHORES The classic resource protection mechanism. Enables cooperating processes to control access to a resource or to synchronize their actions.

References: event(2A)

MAUS Allows processes to share memory[3]. MAUS is divided into sections; each of which may be individually attached/detached from a process. Attaching a section requires a free memory management register. Since processes are limited to 8 data space registers[2] and

1. PIPSIZ is a system defined parameter. Its standard value is 4096.

since at least two data register are needed for the normal data and stack; there will be at most 6 registers available for MAUS sections.

References: maus(2A), break(2A)

MESSAGES Enables processes to communicate in terms of "typed" data packets. The messages are sent by type and queued at the receiving process. The receiver may request any message (in which case he gets the oldest message on the queue) or a particular message type (in which case he gets the oldest message of that type).

References: msg(2A)

1.3 RESTRICTIONS

As each mechanism is presented in detail, the problems and restrictions of that mechanism will be extensively discussed. There is, however, another problem which is inherent in the use of messages and signals; in order to be used, the process id of the target process must be known. Since process ids are assigned by CB-UNIX at the start of execution, process ids must be communicated by some interprocess communication mechanism. At present there is no direct way for this process id to be communicated to another process unless the processes in question are related (see call(2A), fork(2A), getpid(2A), and getppid(2A)). There is a proposal to remedy this situation[4] but no implementation date has been set. However, it is possible to use named pipes, files, MAUS or even semaphores to get around this problem.

Most of the interprocess communication mechanisms to be discussed are system wide resources. This implies that the system designers must plan for the amount of resource to be configured into the system and set up conventions for its use. Signals, semaphores, named pipes, and MAUS fall into this category. With the exception of pipes, which exist separately for each process which creates them, all interprocess communication mechanisms require system wide agreements as to their use.

1.4 SYSTEM CALLS

The system calls through which the user interfaces with interprocess communication mechanisms follow certain conventions. A −1 return means that some sort of error has occurred; the only exception to this convention is for *setpgrp* and only because a −1 is a legal return from *setpgrp* (see section 5.1). When a −1 is returned by a system call, a global variable, *errno*, is filled in by the CB-UNIX system call interface. This global variable, available in any CB-UNIX process, is used to return specific error codes to supplement the −1 return. Further documentation on *errno* is available in the CB-UNIX Programmer's Manual. In particular, the introduction to section 2, which lists error code mnemonics and the actual error code values, is recommended.

1.5 TRADEOFFS

A companion memorandum which describes the tradeoffs between the various interprocess communication mechanisms described here is in preparation[5].

---

2. A process can operate in either separated I&D space mode in which case 8 memory management registers are available for data and 8 are available for text; or in non-separated mode in which a total of 8 memory management registers are available for text + data.

2. FILES

CB-UNIX files represent the most general way that data can be shared between processes. Since most readers are probably familiar with the concept of files it would be unwise to go into any great depth in describing the various ways in which files can be manipulated. To quickly summarize, CB-UNIX files can be read, written, positioned, created, opened, etc.,.; given that a group of processes all know of the same file, and have established conventions for its use, any type of data can be communicated using the file. Synchronizing the use of the file can be accomplished through one of the other interprocess communication mechanisms or even through another file.

2.1 File Manipulation Subroutines (partial list)

open(name,access)

char *name; int access;

The file *name* is opened with the permissions specified in *access*. The file descriptor returned by the *open* system call must be saved for later use by the *read*, *write*, *close*, etc.,. system calls. *Access* may take on the following values:

access: 0 = read permission.
1 = write permission.
2 = read/write permission.

returns: ⩾0 = file descriptor.
−1 = error.

See Also: open(2A), chmod(2A), chown(2A), access(2A), stat(2A), dup(2A), ls(1A)

close(fildes)

int fildes;

The file associated with *fildes* is closed. The file descriptor *fildes* is then available for re-allocation by the system.

returns: −1 = error.

See Also: close(2A)

dup(value)

struct { char lobyte; char hibyte; } value;

The file descriptor in *value.lobyte* (must have been returned by a previous *open*, *creat*, etc.,.) is duplicated. *Dup* will attempt to allocate the new file descriptor starting at *value.hibyte*[1].

returns: ⩾0 = file descriptor.
−1 = error. (invalid input file descriptor or no free file descriptor ⩾ *value.hibyte*).

See Also: dup(2A)

creat(name,mode)

char *name; int mode;

1. In order to understand the nature of *dup* it is necessary to know that for each *open* a file structure is allocated within the system and a pointer to this structure is entered into an array maintained on a per process basis. The index associated with this array entry is a file descriptor. When a file descriptor is duplicated, the file structure pointer in the array at the index associated with the original file descriptor is copied to the new file descriptor index. Both file descriptors then point to the same file structure, i.e. they refer to exactly the same file. For instance, in order to close the file both file descriptors must be *closed*. *Dup* in conjunction with *close* and *open* may be used to manipulate file descriptors as desired. For instance it is often necessary to make some specific file the standard output (assumed to be file descriptor 1 by the CB-UNIX support routines like printf(3A) and putchar(3A)).

The file *name* is created. The mode of the file is *mode*. If the file existed its mode is not changed but the file is truncated to 0 length. *Mode* is created by ORing together some combination of the following bits:

04000 set user ID on execution.
02000 set group ID on execution.
01000 save text after execution (root only).
00400 readable by owner.
00200 writable by owner.
00100 executable by owner.
00070 read/write/execute by group.
00007 read/write/execute by others.

See Also: creat(2A), chmod(2A), chmod(1A), chown(2A), chown(1A), ls(1A), access(2A), passwd(5A), group(5A)

unlink(name)

char *name;

The link[2] *name* is removed. If *name* was the last link to a file, the file is removed.

returns: −1 = error.

See Also: unlink(2A), ln(1A), link(2A), rm(1A)

link(name1,name2)

char *name1,*name2;

A link with *name2* to *name1* is created. *Name1* must already exist and *name2* must not exist.

returns: −1 = error.

See Also: unlink(2A), link(2A), creat(2A), ln(1A), rm(1A)

read(fildes,buffer,bytcnt)

int fildes; char *buffer; int bytcnt;

Data is read from the file corresponding to *fildes* and placed in the core area starting at *buffer* until either *bytcnt* bytes are read, an error is detected or an end of file is found.

returns: >0 = # of bytes read into buffer.
≡0 = end of file (no data placed in buffer).
−1 = error.

See Also: read(2A), getc(3A), getchar(3A)

write(fildes,buffer,bytcnt)

int fildes; char *buffer; int bytcnt;

Data is written from core starting at *buffer* for *bytcnt* bytes. Writing continues until *bytcnt* bytes have been written or an error is detected.

---

2. Linking is a subject whose understanding requires an in-depth discussion of the CB-UNIX file system. However, by way of explanation, any file may have many names which are created using link(2A) or ln(1A). All of these names refer to the same logical data storage area, i.e. they are different names for the same file. When multiple names exist, the file is not removed until the last name is removed. Link cannot create a file, the file must already exist; link can only make another name for the file.

returns: ≥0 = # of bytes written.
−1 = error.

See Also: write(2A), putc(3A), putchar(3A), printf(3A)

seek(fildes,offset,type)

int fildes,offset,type;

The file corresponding to *fildes* has its read/write pointer positioned as specified by *offset* and *type*. The meaning of *offset* is modified by *type* as follows: (pos = current position of read/write pointer)

type	New File Position
0	offset
1	pos + offset
2	file size + offset
3	offset × 512.
4	pos + (offset × 512.)
5	file size + (offset × 512.)

returns: −1 = type invalid, seek error or bad *fildes*.

See Also: seek(2A), lseek(3A)

lseek(fildes,lngoffset,type)

int fildes; long lngoffset; int type;

Lseek positions the file associated with *fildes* at *lngoffset* according to *type*. *Type* has the same meaning as for *seek* except that only 0, 1 and 2 are valid.

returns: −1 = type invalid, seek error or bad *fildes*.

See Also: lseek(3A), seek(2A)

2.2 Notes:

- NOFILE and FCLFILE, system defined parameters, control the number of files which are simultaneously available to a single process. FCLFILE, whose default value is 15, is the number of open files which may be passed to a child process. NOFILE, whose default value is 20, is the number of files which a single process may have open.
- In order to enable a process to open more than FCLFILE files, an open call of the form: *open(−1,0)* must be done.
- Maximum file size is 65535 blocks, 512 bytes/block.
- The name of a file is restricted to 14 characters.
- Files past 8 blocks (4096 bytes) are inherently slower because a level of indirection is added by the system. The file structure maintained by CB-UNIX has room to store 8 block addresses; once 8 is exceeded, the blocks in the file structure become blocks of block addresses.
- Once a file gets large it remains large until removed. Files cannot be shortened.
- Space allocation in a file is deferred until data is read or written. This means that if you create a file, *seek* to block 4000, and then write some data; no space is allocated for blocks 1 to 3999. Their block addresses in the block map for the file are 0 - which signifies an all zero block. If you read one of those blocks, data values of zero will be returned and space allocated.
- If you mount a file system read-only and then access a non-allocated block on a file which is less than the length of the file (see the above description), the system will allocate space for the block

on the disk and write a block of zeros to that space, update the file system super block, and then not update the file inode. The above sequence results in a bad free list on the read-only file system. **NOTE: This is considered a bug and will be fixed.**

3. PIPES

A pipe is a uni-directional communication device used to pass information between related processes - i.e. parent-child, siblings, parent-grandchild, etc.,. Process relation is necessary because the file descriptors for the read and write ends of the pipe can only be passed on by the *fork*, *call*, or *exec* system calls. In using pipes, information is written into the write end of the pipe using the write file descriptor with any standard CB-UNIX write routine (e.g. *write*, *printf*, *putchar*, etc.,.) and is read from the read end of the pipe using the read file descriptor and any standard CB-UNIX read routine (e.g. *read*, *readf*, *getchar*, *getc*, etc.,.).

3.1 Pipe Manipulation Subroutines:

pipe(fildes)

int fildes[2];

Opens a pipe for reading and writing. The pipe read file descriptor is returned in *fildes[0]*; the write file descriptor is returned in *fildes[1]*.

See Also: pipe(2A), read(2A), write(2A), close(2A), fstat(2A)

3.2 Notes:

- PIPSIZ, the maximum byte size of a pipe, is a system define. Normally, PIPSIZ is set to 4096 bytes which is the maximum size of a small file. If PIPSIZ is made larger then 4096, than pipes will always become large files if the writer stays ahead of the reader.
- Be careful when using buffered writes to a pipe. Remember that a pipe reader will not see any of the buffered data until the writer has flushed a buffer to the pipe. For instance, when using the standard library routine *putchar*, to a file descriptor other than 1 (by definition, putchar to file descriptor 1 is unbuffered); the data will be buffered until 512 characters have been *putchar*ed or until a *flush* is done. Any process attempting to read the other end of the pipe will roadblock until data actually appears in the pipe.
- Pipes are implemented as non-associated CB-UNIX files. That is, they are normal files with disk space allocated when necessary on the root device of the system. However, the inode allocated for the pipe has no appearance in any directory and no name. Thus, if a system crashes with a pipe open, its inode will show up in a subsequent check(1B) as an unreferenced inode.
- Reading a pipe whose writing end is closed returns an end of file.
- Writing a pipe with no read file descriptors (all read file descriptors have been closed) causes a signal 13, SIGPIPE, to be sent to the writing process. After the signal has been processed, the write will return an error (errno = EPIPE).
- The system call fstat(2A) can be used with the file descriptor for either the read or write end of the pipe. When used with the write end of the pipe, the size returned is the number of bytes left until PIPSIZ bytes is reached. When used with the read end of the pipe, the size returned is the number of bytes of data in the pipe.
- In actuality, a read does not remove data from a pipe; it merely advances the read pointer. This fact has implications for pipe disk overhead which will be discussed later. In a similar fashion, a write advances the pipe write pointer. When the write pointer reaches PIPSIZ the write roadblocks; it does not matter that reads may be trailing the writes by only 1 byte and that effectively there might be only 2 bytes in the pipe. Whenever the read pointer catches the write pointer, the read and write pointers are reset to zero and any disk blocks associated with the pipe are released. Thus it is guaranteed that at least once every PIPSIZ bytes, the read and write pointers will be equal and the pipe pointers will be reset to zero.
- Once a process writing to a pipe roadblocks, it remains roadblocked until the pipe completely empties, i.e. the read pointer catches the write pointer, unless a signal is received.

- All writes, of less than PIPSIZ bytes, are guaranteed to be atomic. This means that the buffer of data referenced in a single write will be put onto the pipe contiguously thereby preventing multiple writers from having their data scrambled as it is put onto the pipe. A result of this principle is that the size of a pipe can exceed PIPSIZ bytes. Single writes of more than PIPSIZ bytes are broken up into PIPSIZ byte blocks and the writing process is roadblocked until all blocks have been written. Writing more than PIPSIZ bytes in a single write from a process - when more than one process is writing the pipe - could scramble data in the pipe.
- Pipes use the same mechanism for I/O as any other CB-UNIX file. A consequence of this fact is that whenever a full block of data (512 bytes) is in the pipe, the system will initiate I/O on the data. This means that the data in the pipe will be written to disk if the pipe is not **completely** emptied before the I/O reaches the top of the disk queue (which occurs very quickly). Additionally, even if less than 512 bytes of data have been written to the pipe, if the data rests in a pipe for longer than 30 seconds it will be written to disk by a program known as *update*.
- If the root file system runs out of space during a pipe write, the pipe write may return an error after some, but not all, of the data has been placed into the pipe. Unfortunately, there is no way of determining the amount of data in the pipe after an error of this sort occurs.

4. NAMED PIPES

A named pipe is a pipe which can be used by non-related processes. The existence of a named pipe is established by the system configuration and access is obtained by opening the CB-UNIX path name associated with the named pipe. Conventionally, named pipe path names are of the form: /dev/pipe/*name*, the remainder of this section will use *nmpipe* for the path name of a named pipe.

4.1 Named Pipe Manipulation Subroutines:

The *open* system call is used to set up the named pipe, *nmpipe*. The first *open* of *nmpipe* causes the system to initialize a pipe and associate it with *nmpipe*, subsequent *opens* of *nmpipe* establish new file descriptors which refer to the same pipe. Only one file descriptor is necessary for both reading and writing a named pipe; however, the standard case is to use separate file descriptors, usually in different processes.

open(nmpipe,access)

char *nmpipe; int access;

The named pipe associated with the path name *nmpipe* is opened for I/O based on *access*:

access: 0 = reading.
1 = writing.
2 = reading/writing.

returns: ≥0 = file descriptor of named pipe.
−1 = error.

See Also: open(2A), mknod(1A), mknod(2A), pipe(2A),stty(2A),fstat(2A)

4.2 Notes:

Named pipes have many of the same restrictions and problems as pipes. The differences are as noted in the following list.

- NNAMPIPE, a system define parameter, controls the number of named pipes available. Its default value is 10.
- By default, named pipes should never return an EOF; however, some kinds of disk errors will cause an EOF to be returned. Also, if it is not desired to block when reading an empty named pipe it is possible to get and immediate EOF returned. See stty(2A).
- A write on a named pipe will never cause a SIGPIPE.
- By default, writes on named pipes do not sleep: if there is not enough room in the pipe to hold the data, no data is placed in the pipe and a zero is returned from the write. This feature may be disabled. See stty(2A).
- If a named pipe is set to return a zero when a write to a full pipe is attempted, it will never be allowed to fill past PIPSIZ bytes. An implication of this fact is that a write of more than PIPSIZ bytes to a named pipe will always return a zero.
- Named pipes are intended for communication between unrelated processes and are well suited for the problem of funneling output from several processes into a single reading process.
- The number of named pipes is defined when the system is configured and nodes (see mknod(1A) and mknod(2A)) must be created for each pipe using the path name convention outlined above.

5. SIGNALS

Signals are used to provide interprocess interrupt capability. When a process receives a signal, its current processing is interrupted and control is given to a signal processing routine. A process may choose to ignore a signal, catch a signal or allow system default action (usually abort the process). Most of the allowable signals have specific purposes. The following mnemonics have been established in order to aid in remembering those purposes.

No.	Mnemonic	Description	Default Action
1	SIGHUP	hangup	abort
2	SIGINT	rubout (DEL)	abort
3	SIGQIT	quit (FS)	abort/core
4	SIGINS	illegal instruction	abort/core
5	SIGTRC	trace/breakpoint	abort/core
6	SIGIOT	IOT instruction	abort/core
7	SIGEMT	EMT instruction	abort/core
8	SIGFPT	floating point error	abort/core
9	SIGKIL	uncatchable termination	abort
10	SIGBUS	bus error	abort/core
11	SIGSEG	segmentation violation	abort/core
12	SIGSYS	bad system call	abort/core
13	SIGPIPE	end of pipe	abort
14	SIGCLK	alarm clock expired	abort
15	SIGTRM	catchable termination	abort
16	unused		abort
17	unused		abort
18	SIGCLD	death of a child	ignore
19	SIGPWR	power-fail restart	ignore

Under the default action column, abort means that the process is terminated, core means that a core image of the process, at the time of the signal, is produced, and ignore means that the signal is ignored.

The SIGHUP, SIGINT, and SIGQIT signals are produced by the system as a result of terminal action for processes which are associated with a terminal. They are provided so that:

SIGHUP: A process will not continue to run after the controlling terminal has been hung up.

SIGINT: A process can be forced into another processing mode or aborted when it is not accepting commands from the terminal, i.e. you may strike the 'del' or 'rubout' key on the terminal at any time and the signal will immediately be produced.

SIGQIT: An alternate signal, usually used as an unconditional kill, may be produced at the terminal to stop terminal controlled processes which have run amok. The core produced by SIGQIT is often used in debugging. Like the 'del' key, the 'fs' key may be struck at any time and the corresponding signal will be produced immediately.

SIGCLD, the child death signal, is used to inform a parent process immediately upon the death of a child. will be immediately aware of the death of a child. Because a parent may not care about a child. its default action is to be ignored. SIGCLD is discussed in detail in the **Notes** section.

Normally, in order to send a signal, the process id of the target process must be known. There are a variety of ways in which process ids can be communicated. For instance, a file could be established in which each process puts its process id in a particular spot, a named pipe could be used to pass all process ids to a signal-dispensing process, etc.,.. For related processes, i.e. processes with common ancestry or parent-child relations, fork(2A), getpid(2A), and getppid(2A) system calls can be used to obtain process ids.

Signals can also be sent to a process group. For instance, normally all processes started at a particular terminal belong to the same process group. When a terminal signal, such as SIGHUP, is generated by a terminal, it is sent to the process group associated with the terminal. Therefore, all processes

started at the terminal will get the signal. A process may change its process group to any number it desires using setpgrp(2A). Process groups defined by system designers should be in the range −101 to −32767 to avoid conflict with CB-UNIX conventions.[1]

5.1 Signal Manipulation Subroutines:

signal(sgn,&catch)

int sgn; CATCH catch(); (typedef *type* CATCH)

Notifies CB-UNIX that the subroutine *catch* is to be called when the signal *sgn* occurs. *Catch* will be called with the argument *sgn* when that signal is caught. Thus, one signal catch routine can catch multiple signals. To ignore a signal, call *signal* with *&catch* = 1. To restore the system default action on a signal, call *signal* with *&catch* = 0. For catching signals, no distinction is made at the receiving process between signals sent through the process group to which the process belongs and signals sent to that process through its process id. In other words, the receiver cannot determine the identity of the sender.

returns: old value of *catch*.
−1 = error (errno = EINVAL if illegal *sgn*).

See Also: signal(2A), reset(2A)

kill(signo,target)

int signo,target;

There are several different cases of kill:

kill(target,+signo) Sends the signal *signo* to the process with process id *target* if the sending process's uid matches the uid of *target*. If the sending process's uid is **root** the signal is sent unconditionally.

kill(0,+signo) Sends the signal *signo* to all members of the process group of the sending process whose uids match the uid of the sending process. If the sending process's uid is **root** all members of the sending process's process group receive the signal. Note that the sending process also receives the signal.

kill(−1,+signo) Sends the signal *signo* to all processes whose uids match the sending process's uid. If the sending process's uid is **root**, all processes except 0 and 1 will receive the signal.

kill(target,−signo) Sends the signal *signo* to all processes in the process group *target* whose uids match the sending process's uid. If the sending process's uid is **root**, all processes in the process group *target* will receive the signal.

kill(0,−signo) Sends the signal *signo* to all processes whose process group matches the sending process's process group and whose uids match the sending process's uid - except that the sending process will not receive the signal. If the sending process's uid is **root**, all processes whose process group matches the sending process's process group - except the sending process - will receive the signal.

kill(target,0) Reserved for future expansion.

returns: ≡0 = all ok.
−1 = no signal sent (errno = ESRCH).

See Also: kill(1A), kill(2A)

---

1 Positive process groups are used as default process group numbers by CB-UNIX and process groups in the range −1 to −100 are reserved for system programming.

setpgrp(group)

int group;

Tells the system to set the process group of the calling process to *group*. The old process group of the process is returned. If *group* = 0, then the group is not changed and the only action is to return the current process group.

returns: old process group.

5.2 Notes:

- NSIG, a system define parameter, controls the number of signals. The maximum allowable value of NSIG is currently 33, the minimum value is 19, the default value is 20. The number of signals actually available is NSIG−1, a number that only the most picayune rationale can justify.
- In the case of simultaneous signals, CB-UNIX will process SIGINS first and then the remaining signals in signal number order.
- Except for SIGINS and SIGTRC, caught signals are restored to default action when processed by CB-UNIX. Thus a signal catch subroutine should probably reset the signal catch as its first instruction. SIGINS is used by processes which emulate floating point on non-floating point machines. SIGTRC is used by adb(1A) in process tracing.
- Except for SIGCLD, signals are not queued. For instance, if two processes send a third process the same signal simultaneously, the third process will see only one instance of the signal.
- SIGCLD, used to notify a process of the death of a child, is a very special signal. When a process dies, SIGCLD is sent to its parent process and the dying process enters the zombie state, that twilight zone between life and death. At this point, the process is really dead but certain cleanup activity must be performed by its parent in order to allow its soul to rest. The default action on SIGCLD is to ignore the signal; this leaves the child process in its zombie condition, i.e. not really fully dead. If, at some time in the future, the parent process specifies an action on SIGCLD, the signal will be sent to the parent again. If the process has asked the system to ignore SIGCLD, the action is a little different. In this case, the child process is mercifully put to rest and does not remain in the zombie state. Note that there is a difference between the default ignore and a process requested ignore. Finally, if the process is catching SIGCLD, the *catch* routine is activated by SIGCLD. In this case, the child remains in the zombie state until the parent does a *wait*. If a parent process dies without *wait*ing, leaving children in the zombie state; those children are inherited by the CB-UNIX *init* process which then becomes responsible for their well being. In fact, whenever a parent dies, all of its children are inherited by *init* regardless of their current state.
- SIGTRM is used by init(1B) in changing the run state of the machine. If a particular process needs to be killed because it is not supposed to be running in the current run level of the machine, *init* will send it a SIGTRM signal. If the process does not die within 20 seconds, *init* will send it a SIGKIL signal. For more on run levels see lines(5A) and init(1B).
- SIGTRM is the default signal used by the *kill* command. See kill(1A).
- Caught signals may interfere with certain system calls. In order to understand the consequences of this statement it will be necessary to go into some detail about how signals are actually handled. When the signal is sent, the receiving process has a bit set which 'posts' the signal. The signal is posted without regard to its eventual disposal by the receiving process - because the data necessary for that determination is not available until the receiving process is active. If the receiving process is in the *wait* state,[2] when a signal is posted, it is placed in the *run* state so that signal disposal can be determined. If the requested action was to ignore the signal, the process will reenter the *wait* state. If the process is in the *sleep* state,[3] when a signal is posted, it is left in that state. After the *sleep*ing

2. A process is put into the *wait* state on low priority system calls; i.e. ones which are expected to be of long duration or are for general system resources. Some examples are I/O to a character device such as a terminal, pipe roadblocks, and semaphore activity.

process enters the *run* state, which occurs after the reason for the *sleep* has disappeared, the system examines its signal action area to determine signal disposal. If the requested action was to ignore, the system causes processing to resume without regard to the signal. If the process was in the run state when the signal was received, all is as expected: the signal is either ignored or processed. The potential problem only occurs when the process is in the *wait* state and is catching the signal which was posted. In this case the process cannot reenter the *wait* state. Instead, the system call is aborted and a −1 is returned, with errno = EINTR, after the users catch routine is executed. Any system call so interrupted may be reissued from the beginning, unfortunately, in the case of a write to a character device some of the characters may have already been written; it is not currently possible for the user to determine how many. This is the only case in which an interrupted system call may not be restarted successfully.

- SIGPWR is sent to all processes upon completion of the CB-UNIX power-fail restart procedure.
- SIGCLK is used by sleep(2A). Before using it read the *sleep* manual page. Any signal may be used by a process for its own purposes. Of course, if the process uses one of the pre-defined signals some confusion could result if the system happened to send the signal at an unexpected time. Signals 16, 17, and 19 are never sent by the system.
- SIGKIL cannot be caught.
- CAUTION: Because of the way CB-UNIX works, it is impossible to guarantee that a signal can be reset before the same signal is received again except as noted above.

3. A process is put into the *sleep* state on high priority system calls; i.e. ones which are expected to take only a short period of time or which require locking the process in core. Some examples are CB-UNIX file activity, disk I/O, and raw I/O to a character device.

6. INTERPROCESS MESSAGES

Interprocess messages are used to pass small amounts of data between processes. When a message is sent it is placed on a message queue associated with the receiving process until read; however, before a message can be placed on the receiving process's message queue, the receiving process must enable message reception. To get a message, the receiver requests that a message of a particular type be taken from the message queue. All message activity occurs in core, no disk activity is involved.

6.1 Message Manipulation Subroutines:

In the following subroutine descriptions, the following structure definition is assumed:

```
struct mstruct {
        int frompid;            /* process id of sender */
        int mtype;              /* type of message */
} mstructp[1];
```

For more documentation on messages see msg(2A) in the CB-UNIX Programmer's Manual.

msgenab()

Enables message reception for the calling process. The basic action here is to allocate a message Q header for the process which calls *msgenab*. Until messages are enabled within a process, another process cannot send messages to it.

returns: ≡0 = messages enabled.
−1 = error.

msgdisab()

Disables message reception for the calling process. All messages currently on the message Q are flushed; those requiring acknowledgments are acknowledged (see **Notes** section). The message Q header allocated by msgenab is de-allocated.

returns: ≡0 = messages disabled.
−1 = error.

send(buffer,size,topid,type)

char *buffer; int size,topid,type;

A message of *size* bytes is taken from *buffer* and placed on the message queue of process *topid*. The message will have the type *type*. If the system message queue space is full or if the addressed process message queue length is too long, *send* will return without sending the message.

returns: ⩾0 = number of bytes sent.
−1 = error.

sendw(buffer,size,topid,type)

Same as *send* except that it will not return until the message is actually sent unless an error occurs. Errors are as follows: receiving process has not enabled messages, illegal message type or message too long.

recv(buffer,maxsize,mstructp,type)

char *buffer; int maxsize,type; struct mscruct mstructp[1];

Gets the first message in the process message queue of type *type* and places the first *maxsize* bytes of the message into *buffer*. If the message is longer than *maxsize* the rest of the message is lost. If no message of *type* exists, return immediately. If *type* = 0 then recv returns the

first message on the message of any type. If no messages are on the queue and *type* = 0, recv returns immediately.

returns: ≥0 = number of bytes received.
−1 = error.

recvw(buffer,maxsize,mstructp,type)

Same as *recv* except that recvw will not return until a message of the correct type is available. If *type* = 0, *recvw* will not return until any message is available.

6.2 Notes:

- The following system define parameters are used to control message storage allocation:

Value	Mnemonic	Description
212	MAXMLEN	Maximum message length in bytes
5	MAXMSG	Maximum number of messages on receiving process queue.
10	NMQHDR	number of message q headers available system wide.
52	MMAPSIZ	size of map used for message space allocation.
50	MSGMEM	number of 32 word blocks allocated for message storage.

- NMQHDR is the number of processes which can simultaneously enable message reception.
- The amount of memory allocated within the system for messages is:

  (NMQHDR × sizeof(msgqhdr)) + (MMAPSIZ × 2) + (MSGMEM × 32);

  msgqhdr is currently 8 bytes long.
- A zero length message is legal. However, a message always requires at least 8 bytes of storage.
- Types have built in meanings for the system. The CB-UNIX implementation of messages assumes that for types 1-63 an acknowledgment is desired; 64-128 imply no acknowledgment. If a process disables message reception or terminates with messages on its message queue, CB-UNIX changes all messages of types 1-63 to type 128 and returns them to the sender (if there is room on his queue); messages of types 64-128 are discarded.
- A process may not send a message to a non-existent process or processes which have not enabled message reception. If it tries, a −1 will be returned.
- When a message system call returns, the possible values of errno are:

  ETABLE System out of message Q headers (*msgenab*) or receiving process message q full (*send*). Messages not enabled at receiver (*send, sendw*).
  EINVAL Message too long (*send, sendw*) or illegal message type (*send, sendw, recv, recvw*).
  ESRCH Process id not found (*send, sendw*).
  ENOMEM No memory to store message (*send*).
  ENOALOC Messages not enabled (*recv, recvw, msgdisab*).
  ENOMSG Message type not on Q (*recv*).
  EFAULT Cannot fill in mstructp data (*recv, recvw*).

7. SEMAPHORES

Semaphores are used to control access to system resources or to synchronize actions between processes. Typically, a non-zero semaphore value indicates that the resource being controlled is available; zero indicates that the resource is in use. In synchronization mode, a semaphore value of one tells the checking process to proceed, a semaphore value of zero tells the checking process to wait.

7.1 Semaphore Manipulation Subroutines:

For more documentation see sema(2A) in the CB-UNIX Programmer's Manual.

v(sema)

post(sema)

int sema;

The semaphore *sema* is incremented and any processes waiting on *sema* are awakened.

returns: ⩾0 = value of *sema* before increment.
−1 = illegal semaphore number.

p(sema)

int sema;

If *sema* is zero, process doing *p* is roadblocked; otherwise, *sema* is decremented and process continues at statement following *p*.

returns: ⩾0 = value of *sema* before decrement.
−1 = illegal semaphore number.

block(sema)

int sema;

Process is put to sleep on *sema* until a *post(sema)* or *v(sema)* is done by some other process.

returns: ⩾0 = value of *sema* before roadblock.
−1 = illegal semaphore number.

test(sema)

int sema;

Semaphore *sema* is decremented if it is greater than zero.

returns: ⩾0 = value of *sema* before decrement (if any).
−1 = illegal semaphore number.

7.2 Notes:

- NEVT, a system define parameter, controls the number of semaphores in the system. Of these NEVT semaphores, NSSEM of them belong to special processes like the line printer spooler. The special semaphores have negative numbers, −1 to −NSSEM; the remainder of the semaphores have positive numbers, 0 to (NEVT−NSSEM−1) and are used by application programs.
- Semaphores are initialized to 0 at system startup. For further initialization, a program must be run after the system has been started in order to set values using a series of *v* subroutine calls.
- Any of the above calls will return a −1 with errno = EINVAL if an illegal semaphore number is used.

- There are no races in the semaphore mechanism. That is, correct interlocking of semaphore checks is guaranteed by CB-UNIX.
- A typical sequence of semaphore system calls to be included in each process accessing some controlled resource, for instance, a data file, is:

```
p(sema)        /* initialized to 1 elsewhere */
---code to manipulate file---
v(sema);
```

- If a semaphore is being used by a pair of processes to synchronize some critical section of code, a typical loop control in the two processes might be:

```
while(p(sema) >= 0) {
        ---critical code section---
        v(sema);
}
```

- Note that if several processes *block* on a single semaphore, say *sema1*, a single *v(sema1)* or *post(sema1)* will wake them up and allow all to continue processing.
- Semaphores have a 16 bit value. If you were to continuously *post* or *v* a semaphore 32767 times without any intervening *p*, the semaphore would overflow without indicating an error.
- Beware of allowing a process to terminate after a *p* but before a *v* or *post*.
- Similarly, avoid sequences like *p(1) ... p(2)* in one process and *p(2) ... p(1)* in another process. This condition is known as a deadly embrace and could cause both processes to hang. Always *p* semaphores in the same fixed order in all processes.

8. MAUS

MAUS is a dedicated core area subdivided into sections which can be separately attached by any process. These sections are referenced by names in the CB-UNIX file system which must be used as arguments to *getmaus* in order to attach the corresponding section of MAUS. These names may also be used by the *open* system call after which the file descriptor returned by *open* can be used by other system calls such as *read*, *write*, *close*, and *seek* in the normal manner.

8.1 MAUS Manipulation Subroutines:

The description in this section applies to the C interface to MAUS. For a more complete coverage of the MAUS interface see maus(2A) in the CB-UNIX Programmer's Manual.

getmaus(section,access)

char *section; int access;

Returns a MAUS descriptor which references the MAUS section with the CB-UNIX name *section*. This maus descriptor is analogous to the file descriptor returned by the *open* system call and is used by the other MAUS manipulation subroutines. The second argument, *access*, describes the permissions which the caller would like with respect to the MAUS section *section*.

access: 0 = read permission.
1 = write permission (due to hardware restrictions, write implies read/write).
2 = read/write permission.

returns: ≥0 = maus descriptor.
−1 = error.

freemaus(mausdes)

int mausdes;

De-allocates the maus descriptor *mausdes* so that it may be reused. If the MAUS section corresponding to *mausdes* is active, it remains active.

returns: −1 = error.

enabmaus(mausdes)

int mausdes;

The MAUS section corresponding to *mausdes* is attached to the first available memory management register. The virtual address of the MAUS section is returned. *Mausdes* must have been obtained through a *getmaus* system call; in addition, the process must have at least one free memory management register.

returns: ≥0 = address of MAUS sections in the process's address space.
−1 = error.

dismaus(vaddr)

char *vaddr;

If a MAUS section is attached at *vaddr* it is detached. If there is no MAUS section attached to *vaddr*, an error is returned.

returns: ≥0 = *mausdes* of MAUS section attached.
−1 = error.
−2 = no MAUS section associated with *vaddr*.

switmaus(mausdes,vaddr)

int mausdes; char *vaddr;

This is a generalization of *enabmaus* and *dismaus*. There are four cases:

switmaus(−1,−1)

In this case, *switmaus* is merely a check for a free memory management register. −1 is returned if there are no free memory management registers; a return of anything else implies that there is a free memory management register.

switmaus(−1,vaddr)

The MAUS section attached to *vaddr* is disabled (same as *dismaus*).

switmaus(mausdes,−1)

The MAUS section associated with *mausdes* is attached to the first available memory management register (same as *enabmaus*).

switmaus(mausdes,vaddr)

The MAUS section associated with *mausdes* is attached to *vaddr*. If there is already a MAUS section attached to *vaddr*, its *mausdes* is returned. If no MAUS section was associated with *vaddr*, a −2 is returned. If an error condition exists, e.g. illegal *mausdes* or bad *vaddr*, a −1 is returned.

returns: −1 = error (for other returns, see above descriptions).

8.2 Notes:

- The amount of core to be used for MAUS and the assignment of sections within MAUS are part of the CB-UNIX system configuration and are completely arbitrary. MAUS sections must be allocated at system configuration time and for each section a special file must be created (see *mknod*(1A)).
- MAUS descriptors are inherited across forks and execs.
- MAUS virtual address to *mausdes* association is inherited across forks.
- If a *freemaus* is done on a *mausdes* which is associated with a MAUS section currently attached to process's address space, the MAUS section remains attached. If a *dismaus* is later done on the *vaddr* associated with the freed *mausdes*, *mausdes* is returned anyway.
- The virtual address returned by MAUS manipulation calls, *vaddr*, never has any of it low order 13 bits set. In other words it always represents the start of a 4096 word segment. By the same token, virtual addresses passed into MAUS manipulation routines have their low 13 bits ignored.
- The number of MAUS sections which may be simultaneously attached to a single process varies. For a process with separated text and data (i.e. I&D space separated) as many as 6 MAUS sections might be attached simultaneously. For a process with combined text and data (I&D space not separated) as many as 5 MAUS sections might be attached simultaneously. For instance, a I&D space separated program with 15k bytes of text, 6k bytes of data and a 1k byte stack could have 6 MAUS sections simultaneously attached. The same program without I&D space separation could have 4 MAUS sections attached.
- If a program wishes to access a MAUS section but has no free memory management registers, the MAUS section can be opened with the *open* system call after which standard UNIX I/O commands such as *read*, *write* and *seek* can be used to access the MAUS section.
- When a MAUS system call returns a −1, the possible errno values are:

  EINVAL Illegal *mausdes* or *vaddr*.

EMFILE Bad MAUS descriptor.
ENOMEM No free memory management registers.

9. REFERENCES

[1] "CB-UNIX Programmer's Manual"

[2] "A Description of the UNIX Fil
1975.

[3] "Shared Memory in UNIX", Dale

[4] "Proposal for UNIX Interproces
1976.

[5] "Interprocess Communication Pe