-

NAME

crfork, crexit, crread, crwrite, crexch, crprior − coroutine scheme

SYNOPSIS

int crfork([stack, nwords])
int stack[];
int nwords;

crexit()

int crread(connector, buffer, nbytes)
int *connector[2];
char *buffer;
int nbytes;

crwrite(connector, buffer, nbytes)
int *connector[2];
char *buffer;
int nbytes;

crexch(conn1, conn2, i)
int *conn1[2], *conn2[2];
int i;

#define logical char *
crprior(p)
logical p;

DESCRIPTION

These functions are named by analogy to *fork, exit, read, write* (II). They establish and synchronize ‘coroutines’, which behave in many respects like a set of processes working in the same address space. The functions live in */usr/lib/cr.a.*

Coroutines are placed on queues to indicate their state of readiness. One coroutine is always distinguished as ‘running’. Coroutines that are runnable but not running are registered on a ‘ready queue’. The head member of the ready queue is started whenever no other coroutine is specifically caused to be running.

Each connector heads two queues: *Connector[0]* is the queue of unsatisfied *crreads* outstanding on the connector. *Connector[1]* is the queue of *crwrites.* All queues must start empty, *i.e.* with heads set to zero.

Crfork is normally called with no arguments. It places the running coroutine at the head of the ready queue, creates a new coroutine, and starts the new one running. *Crfork* returns immediately in the new coroutine with value 0, and upon restarting of the old coroutine with value 1.

Crexit stops the running coroutine and does not place it in any queue.

Crread copies characters from the *buffer* of the *crwrite* at the head of the *connector’s* write queue to the *buffer* of *crread.* If the write queue is empty, copying is delayed and the running coroutine is placed on the read queue. The number of characters copied is the minimum of *nbytes* and the number of characters remaining in the write *buffer,* and is returned as the value of *crread.* After copying, the location of the write *buffer* and the corresponding *nbytes* are updated appropriately. If zero characters remain, the coroutine of the *crwrite* is moved to the head of the ready queue. If the write queue remains nonempty, the head member of the read queue is moved to the head of the ready queue.

Crwrite queues the running coroutine on the *connector’s* write queue, and records the fact that *nbytes* (zero or more) characters in the string *buffer* are available to *crreads.* If the read queue is not empty, its head member is started running.

Crexch exchanges the read queues of connectors *conn1* and *conn2* if *i*=0; and it exchanges the write queues if *i*=1. If a nonempty read queue that had been paired with an empty write queue becomes paired with a nonempty write queue, *crexch* moves the head member of that read queue to the head of the ready queue.

-

Crprior sets a priority on the running coroutine to control the queuing of *crreads* and *crwrites.* When queued, the running coroutine will take its place before coroutines whose priorities exceed its own priority and after others. Priorities are compared as logical, *i.e.* unsigned, quantities. Initially each coroutine's priority is set as large as possible, so default queuing is FIFO.

Storage allocation. The old and new coroutine share the same activation record in the function that invoked *crfork,* so only one may return from the invoking function, and then only when the other has completed execution in that function.

The activation record for each function execution is dynamically allocated rather than stacked; a factor of 3 in running time overhead can result if function calls are very frequent. The overhead may be overcome by providing a separate stack for each coroutine and dispensing with dynamic allocation. The base (lowest) address and size of the new coroutine's stack are supplied to *crfork* as optional arguments *stack* and *nwords.* Stacked allocation and dynamic allocation cannot be mixed in one run. For stacked operation, obtain the coroutine functions from */usr/lib/scr.a* instead of */usr/lib/cr.a.*

FILES

/usr/lib/cr.a
/usr/lib/scr.a

DIAGNOSTICS

'rsave doesn't work' − an old C compilation has called 'rsave'. It must be recompiled to work with the coroutine scheme.

BUGS

Under /usr/lib/cr.a each function has just 12 words of anonymous stack for hard expressions and arguments of further calls, regardless of actual need. There is no checking for stack overflow.
Under /usr/lib/scr.a stack overflow checking is not rigorous.

-

NAME

ms − macros for formatting manuscripts

SYNOPSIS

nroff −ms [options] file ...
troff −ms [options] file ...

DESCRIPTION

This package of *nroff* and *troff* macro definitions provides a canned formatting facility for technical papers. When producing 2-column output on a terminal, its output should be filtered through *col* (I).

The package supports three different formats: BTL technical memorandum with cover sheet, released paper with cover sheet, and an abbreviated ‘debugging’ form without cover sheet.

The macro requests are defined in the attached Request Reference. Many *nroff* and *troff* requests are unsafe in conjunction with this package, however the requests listed below may be used with impunity after the first .PP.

.bp	begin new page
.br	break output line here
.sp n	insert n spacing lines
.ls n	(line spacing) n=1 single, n=2 double space
.na	no alignment of right margin

Output of the *eqn, neqn* and *tbl* (I) preprocessors for equations and tables is acceptable as input.

FILES

/usr/lib/tmac.s

SEE ALSO

eqn (I), nroff (I), troff (I), tbl (VI)

BUGS

REQUEST REFERENCE

Request Form	Initial Value	Cause Break	Explanation
.1C	yes	yes	One column format on a new page.
.2C	no	yes	Two column format.
.AB	no	yes	Begin abstract.
.AE	-	yes	End abstract.
.AI	no	yes	Author's institution follows. Suppressed in TM.
.AU *x y*	no	yes	Author's name follows. *x* is location and *y* is extension, ignored except in TM.
.B	no	no	Boldface text follows.
.CS *x*...	-	yes	Cover sheet info if TM format, suppressed otherwise. Arguments are number of text pages, other pages, total pages, figures, tables, references.
.DA *x*	nroff	no	'Date line' at bottom of page is *x*. Default is today.
.DE	-	yes	End displayed text. Implies .KE.
.DS *x*	no	yes	Start of displayed text, to appear verbatim line-by-line. *x*=I for indented display (default), *x*=L for left-justified on the page, *x*=C for centered. Implies .KS.
.EN	-	yes	Space after equation produced by *eqn* or *neqn.*
.EQ *x*	-	yes	Space before equation. Equation number is *x*.
.FE	-	yes	End footnote.
.FS	no	no	Start footnote. The note will be moved to the bottom of the page.
.HO	-	no	'Bell Laboratories, Holmdel, New Jersey 07733'.
.I	no	no	Italic text follows.
.IP *x y*	no	yes	Start indented paragraph, with hanging tag *x*. Indentation is *y* ens (default 5).
.KE	-	yes	End keep. Put kept text on next page if not enough room.
.KF	no	yes	Start floating keep. If the kept text must be moved to the next page, float later text back to this page.
.KS	no	yes	Start keeping following text.
.LG	no	no	Make letters larger.
.LP	yes	yes	Start left-blocked paragraph.
.MH	-	no	'Bell Laboratories, Murray Hill, New Jersey 07974'.
.ND	troff	no	No date line at bottom of page.
.NH *n*	-	yes	Same as .SH, with section number supplied automatically. Numbers are multilevel, like 1.2.3, where *n* tells what level is wanted (default is 1).
.NL	yes	no	Make letters normal size.
.OK	-	yes	'Other keywords' for TM cover sheet follow.
.PP	no	yes	Begin paragraph. First line indented.
.R	yes	no	Roman text follows.
.RE	-	yes	End relative indent level.
.RP	no	-	Cover sheet and first page for released paper. Must precede other requests.
.RS	-	yes	Start level of relative indentation. Following .IP's measured from current indentation.
.SG *x*	no	yes	Insert signature(s) of author(s), ignored except in TM. *x* is the reference line (initials of author and typist).
.SH	-	yes	Section head follows, font automatically bold.
.SM	no	no	Make letters smaller.
.TL	no	yes	Title follows.
.TM *x y z*	no	-	BTL TM cover sheet and first page, *x*=TM number, *y*=(quoted list of) case number(s), *z*=file number. Must precede other requests.
.WH	-	no	'Bell Laboratories, Whippany, New Jersey 07981'.

-

NAME

plot: openpl et al. − graphics interface

SYNOPSIS

openpl( )

erase( )

label(s)
char s[ ];

line(x1, y1, x2, y2)

circle(x, y, r)

arc(x, y, x0, y0, x1, y1)

dot(x, y, dx, n, pattern)
int pattern[ ];

move(x, y)

point(x, y)

linemod(s)
char s[ ];

space(x0, y0, x1, y1)

closepl( )

DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. See *plot* (VI) for a description of the meaning of the subroutines.

There are four libraries containing these routines, one that produces general graphics commands on the standard output, and one each for the vt0 storage scope, the Diablo plotting terminal and the Tektronix 4014 terminal. *Openpl* must be used before any of the others to open the device for writing. *Closepl* flushes the output.

FILES

/usr/lib/plot.a	produces output for plotting filters
/usr/lib/vt0.a	produces output on vt0 storage scope
/usr/lib/gsip.a	produces output on Diablo terminal
/usr/lib/tek.a	produces output for the Tektronix 4014 terminal

SEE ALSO

plot (VI), graph (VI)

BUGS

NAME

salloc − string allocation and manipulation

SYNOPSIS

(get size in r0)
jsr pc,allocate
(header address in r1)

(get source header address in r0,
destination header address in r1)
jsr pc,copy

jsr pc,wc

(all following routines assume r1 contains header address)

jsr pc,release

(get character in r0)
jsr pc,putchar

jsr pc,lookchar
(character in r0)

jsr pc,getchar
(character in r0)

(get character in r0)
jsr pc,alterchar

(get position in r0)
jsr pc,seekchar

jsr pc,backspace
(character in r0)

(get word in r0)
jsr pc,putword

jsr pc,lookword
(word in r0)

jsr pc,getword
(word in r0)

(get word in r0)
jsr pc,alterword

jsr pc,backword
(word in r0)

jsr pc,length
(length in r0)

jsr pc,position
(position in r0)

jsr pc,rewind

jsr pc,create

jsr pc,fsfile

jsr pc,zero

DESCRIPTION

This package is a complete set of routines for dealing with almost arbitrary length strings of words and bytes. It lives in */lib/libs.a.* The strings are stored on a disk file, so the sum of their lengths can be consid-

-

erably larger than the available core. A small buffer cache makes for reasonable speed.

For each string there is a header of four words, namely a write pointer, a read pointer and pointers to the beginning and end of the block containing the string. Initially the read and write pointers point to the beginning of the string. All routines that refer to a string require the header address in r1. Unless the string is destroyed by the call, upon return r1 will point to the same string, although the string may have grown to the extent that it had to be be moved.

Allocate obtains a string of the requested size and returns a pointer to its header in r1.

Release releases a string back to free storage.

Putchar and *putword* write a byte or word respectively into the string and advance the write pointer.

Lookchar and *lookword* read a byte or word respectively from the string but do not advance the read pointer.

Getchar and *getword* read a byte or word respectively from the string and advance the read pointer.

Alterchar and *alterword* write a byte or word respectively into the string where the read pointer is pointing and advance the read pointer.

Backspace and *backword* read the last byte or word written and decrement the write pointer.

All write operations will automatically get a larger block if the current block is exceeded. All read operations return with the error bit set if attempting to read beyond the write pointer.

Seekchar moves the read pointer to the offset specified in r0.

Length returns the current length of the string (beginning pointer to write pointer) in r0.

Position returns the current offset of the read pointer in r0.

Rewind moves the read pointer to the beginning of the string.

Create returns the read and write pointers to the beginning of the string.

Fsfile moves the read pointer to the current position of the write pointer.

Zero zeros the whole string and sets the write pointer to the beginning of the string.

Copy copies the string whose header pointer is in r0 to the string whose header pointer is in r1. Care should be taken in using the copy instruction since r1 will be changed if the contents of the source string is bigger than the destination string.

Wc forces the contents of the internal buffers and the header blocks to be written on disc.

An in-core version of this allocator exists in *dc* (I), and a permanent-file version exists in *form* and *fed* (VI).

FILES

/lib/libs.a	library, accessed by *ld ... -ls*
alloc.d	temporary file for string storage

SEE ALSO

alloc (III)

DIAGNOSTICS

'error in copy' − disk write error encountered in *copy*.
'error in allocator' − routine called with bad header pointer.
'cannot open output file' − temp file *alloc.d* cannot be created or opened.
'out of space' − no sufficiently large block or no header is available for a new or growing block.

BUGS