-

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.