A Partial Tour Through the UNIX† Shell

Geoff Collyer

Department of Statistics
University of Toronto
Toronto, Ontario, Canada
M5S 1A1
`utzoo!utstat!geoff`
`geoff@utstat.toronto.edu`

ABSTRACT

We have recently completed protracted surgery on the UNIX command interpreter or 'shell' [Bourne1978a] to make it use the standard UNIX memory allocator (*malloc* (3) and relatives) for its internal memory management instead of the original scheme (catching its own memory faults, using the *sbrk* (2) system call to grow its memory allocation and restarting faulting instructions). We also fixed some bugs, *lint* (1) complaints and suboptimal performance. This paper describes the lessons learned about the internal workings of the shell. Much of this information is oral folklore or is simply not generally known, and requires a determined effort to learn, yet is essential to correct understanding and maintenance of the shell.

1. Introduction

A very sketchy overview of the shell is that it parses its input into a parse tree, then walks the tree, executing the tree nodes by creating pipes, *fork* ing, redirecting I/O descriptors, *exec* ing commands, and the like. Interwoven with this are macro expansion; honouring quoting; coping with keyboard, alarm clock, and other interrupts (via signals); and maintaining variables and functions. One can think of the shell as a macro processor which also interprets commands.

The original Seventh Edition shell had to run in a small address space (64k bytes of instructions and 64k bytes of data, including stack segment), yet places no arbitrary limits on lengths of strings or input lines (which may explain some of the contortions in the code). With the exceptions of *eval* and quoting, which are incompletely specified, the external specification of the shell is simple, rational, and clean. No comments in this paper should be taken as denigrating these achievements.

The shell source code is opaque and under-commented in spots, which causes maintainers to attempt only minimal changes and fixes. The shell was the last program ported to the Interdata during the original UNIX port, [Johnson1978a] due to the difficulty of getting the details of restarting faulting instructions just right, which is why the Seventh Edition (also known as "V7") distribution tape includes `/bin/osh`, the Sixth Edition shell. [Ritchie1987a] One cliched complaint about the original shell source, that it was written in a dialect of the C language resembling Algol 68, is not a problem once one gets

† UNIX is a trademark of Bell Laboratories.

used to it, particularly if one has a reading knowledge of Algol 68. In any case, recent System V shells are written in ordinary C. However, this is the least of the problems.

Upon attempting to make the Ninth Edition shell (derived from the System V Release 2 shell) run on a Sun-3 under SunOS 3.x, we ran into trouble with the shell's peculiar internal memory management. The shell often failed in a spectacular and annoying way: it grew its stack segment to maximum size and then dumped core. We discovered many previously-undocumented characteristics of the shell in the process of converting the shell to use *malloc* (3) and relatives. The result of this work is referred to throughout this paper as ‘‘the new shell’’, for lack of a better name. This paper discusses only the parts of the shell visited in the course of making it work correctly on the Sun-3; the rest of the shell is relatively straightforward.

The rest of the paper consists of six sections: section 2 describes the design of the old shell, section 3 the problems in implementation of the old shell, section 4 the fixes we applied to produce the new shell, section 5 our methods, section 6 our conclusions, and section 7 acknowledgements. Those readers interested only in the internals of the old shell may safely skip sections 4 and 5.

2. Design

2.1. Memory Management

The first subtlety encountered by most shell maintainers is the shell's internal memory management, which has been characterised as ‘extremely elegant, but a house of cards’. The shell contains its own memory allocator, a variant of the Seventh Edition *malloc* , which maintains two distinct kinds of storage: *heap* storage, which has an indefinite lifetime and resembles ordinary *malloc* ed memory; and *“stak”* storage (so spelled to distinguish it from the shell's stack segment) which is allocated and deallocated in strict last-in, first-out order. Heap and *stak* storage blocks are intermingled in the data segment.

A fundamental abstraction of the shell is a stack of *stak* storage, in which the top item on the stack is typically a growing string. The top item may be moved at the convenience of the memory allocator, so it should (in theory, if not always in practice) only be referred to by the functions and macros of `stak.h`; keeping private pointers into the top item is forbidden. Once the top item has been grown to its maximum extent, it may be made permanent and immovable, and a new, empty top item is begun.

The interface to the *stak* storage manipulators, `stak.h`, declares several functions and macros, notably *pushstak(byte)* , which appends a *byte* to the top item and advances the top past it, acquiring more memory from UNIX as needed. *relstak()* yields the integer offset of the top of the top *stak* item, i.e. the size of the top item. *absstak(offset)* yields a temporary pointer to the top item, *offset* bytes in; this pointer must not be retained. *setstak(offset)* sets the top of the top item to be *offset* bytes in. *zerostak()* stores a zero on the top of the top item but does not move the top. *curstak()* yields the a temporary pointer to the top of the top item; this pointer must not be retained. *usestak()* calls *locstak()* and ignores the result. *fixstak()* calls *endstak* with a pointer to the top of the top item.

locstak() returns a temporary pointer of the bottom of the new top *stak* item, which will be big enough for any structure used in the shell (notably `struct fileblk` or `2*CPYSIZ` from `io.c`); this pointer may be used until one of *pushstak*, *endstak*, or *fixstak* is called. *endstak(argp)* terminates the top item at *argp* with a zero byte, makes the top item permanent, starts a new top item, and returns the address of the terminated and now permanent item. *getstak(n)* returns a permanent item of size *n* bytes by growing the current top item. *savstak()* asserts that the top item is empty and returns the address of the bottom of the top item. *tdystak(ptr)* removes temporary files (e.g. from here documents) described by structures on the *stak* down to address *ptr*, and pops the *stak* down to but not including *ptr*. *stakchk()* reduces the data break if possible. *cpystak(string)* copies the *string* to a new permanent item and returns its address.

The memory allocator and other parts of the old shell assume that the address of newly-allocated *stak* storage will be greater than the addresses of all other still-active *stak* storage. (This is not true of storage obtained from arbitrary *malloc* s.) This property is exploited by tricks such as recording a single ‘‘watermark’’ pointer, to mark the points in several intertwined stacks of *stak* storage above which data may be eventually discarded, then later popping the top items from the stacks by popping each stack until its stack pointer reaches the watermark pointer, or drops below it.

Heap storage is allocated by *alloc* (also known as *malloc* in the old shell). The shell assumes fundamentally that *free* will ignore attempts to free the address zero (‘‘null pointers’’), addresses in the shell’s stack segment (automatic variables, command-line arguments, and environment variables), and addresses of *stak* storage not yet made permanent and immobile; the shell’s *free* is meant to free only heap storage and permanent *stak* storage.

The old shell catches its own memory faults (via the SIGSEGV signal, typically caused by heap allocation beyond the data break or growth of the current *stak* item beyond the data break), grows the data segment with *sbrk*(2) by *brkincr* bytes, and returns control, thus resuming the faulting instruction.

2.2. No use of C library

The shell makes no use of the C library beyond system calls, perhaps because the C library was not well-developed when the shell was written, perhaps to make the shell self-contained, or perhaps to avoid dangerous interactions among the shell, the shell’s *malloc*, the C library, and the C library’s *malloc*(3). The effect has been to make the shell fragile and less portable than if it did rely on the C library. For example, the shell’s memory allocator does not co-exist with the C library, so it is not safe to call the *directory*(3) directory-reading routines, which call *malloc*(3).

3. Implementation Problems

3.1. Memory Management

The heap allocator, in `blok.c` (a modified V7 *malloc*(3)), and the *stak* allocator, in `stak.c`, together form the shell’s memory allocator. They are intimate with each other’s internals, in part because the heap allocator must move the top item on the *stak* when allocating heap storage, in order to keep the top item of *stak* storage at the top of the data segment.

alloc moves the top *stak* item to above the new top of the heap arena when the arena is grown, and promotes other *stak* items to *free* able permanent storage, chained together. `blok.c` rounds the number of bytes to allocate down by anding it with the complement of (*brkincr* minus one); this only works correctly if *brkincr* is a power of 2, yet `stak.c` adds 256 to *brkincr*, which starts off at 512! Thus the rounded-down value will be too large, which only hurts performance, fortunately. `stak.c` should probably double *brkincr* instead.

The shell's allocator cannot co-exist with some *malloc* implementations because it assumes that only it allocates storage, and some *malloc* s do likewise. [Korn1985a] Further, the shell contains *malloc* and *free* definitions, but not a *realloc* definition, so uses of *realloc* in the C library will drag in the C library's *realloc*, which will refer to the wrong *malloc* and *free*. More subtly, because the old shell allocates storage above its data break,‡ even a tolerant *malloc* (3) which tried to co-operate with programs which do their own allocation via *brk* (2) or *sbrk* (2) would be misled and would likely step on the old shell's storage above its data break. Perhaps due in part to this problem, the old shell goes to great pains to avoid using the C library, except to execute system calls, necessitating reinvention of parts of it, notably the string functions. The new shell relaxes this restriction and so can use the C library freely.

The assumption that faulting instructions can be (and are) restarted by return from the appropriate signal handler does not hold on all machines of interest. In particular, the Sun-2, Sun-3, [Shannon1988a] and Cray-1 kernel-and-hardware combination are known not to correctly restart faulting instructions, which can lead to failure to initialise memory, and thus to use of the address zero. Furthermore, on the Sun-3, and other machines which do not permit reference to address zero, the old shell's naive assumption that growing the heap will cure a memory fault does not hold for references to address zero, and the strategy of growing the heap on each fault merely grows the heap indefinitely, often until swap or page space is exhausted, when the old shell's memory allocator blows an assertion and dumps core, slowly.

Unfortunately the `stak.h` macros, especially *pushstak*, were not used everywhere that they should have been used, and in places the equivalent code was written out in-line, or other internal programming conventions were violated. Cray Research found and fixed the most troublesome of the abuses of *pushstak*, and these fixes found their way into the Ninth Edition shell. We found and fixed the rest.

3.2. Here Documents

Here documents are a means of supplying standard input to a command from a script and are denoted by '<<'.

Here documents appear to have been added to the original Seventh Edition shell at the last moment. The code is localised, but careless about error-checking and

‡ The *data break* or just *break* is the address of the lowest-numbered byte of the data segment not allocated to a UNIX process. There may be accessible memory between the break and the bottom of the stack segment, but touching it is bad form and may result in a memory fault ("segmentation violation").

performance. Here documents are implemented by copying lines from the shell's input to a temporary file during parsing until a line containing only the delimiter is seen; then later, during execution of the command, if the delimiter was not quoted, copying the first temporary file to a second temporary file while processing macros (e.g. expanding `$DMD`). If the second temporary file was created, it is opened as the command's standard input and unlinked, so it will not have to be removed later; otherwise the first temporary file will be opened as the command's standard input. In either case, the first temporary file will have to be unlinked later, but the shell may not get the chance if it is killed first or if the block containing the command with the here document was terminated via the `exec` built-in command, which replaces the shell with the command.

write system calls to the first temporary file were unchecked, so creating a here document when the file system containing `/tmp` is full may lead to odd behaviour and no diagnostic from the old shell. (*writes* to the second temporary file use a more general mechanism, *flush* in `macro.c`, and are still unchecked in the new shell. Oops!) Also, in the old shell, when copying input to the first temporary file, lines are collected until at least CPYSIZ (512) bytes are present, then are written to disk as whole lines, so CPYSIZ+ε bytes are written at a time, causing *write* s to be unaligned with file system block boundaries, and contributing to the slowness of here documents and thus to the slowness of unbundling of shell “bundles” or “archives”. [Kernighan1984a]

3.3. Directory Reading and Wildcard Expansion

The Seventh Edition shell reads directories to expand wildcards (“generate filenames”), using the *read* system call to read 16 bytes at a time and assuming a Seventh Edition directory layout. The Ninth Edition (and presumably System V Release 2) shell used some of the *directory* (3) routines from the C library, but used private versions of others. This was done so that memory allocation would be under the control of the shell's private *opendir* and *closedir*. Unfortunately, it did require the shell to know details of buffer allocation in the *directory* (3) routines, and those details changed between 4BSD and SunOS 3.0, for example. (We believe 4BSD used a static buffer but SunOS 3.0 allocates the buffer dynamically.)

3.4. I/O Redirection

When the old shell executes a redirected built-in command such as `set`, it saves the redirected descriptor by using *dup2* (2) to make a duplicate descriptor, on a fixed descriptor, USERIO, which is typically 10 and must be above the shell's user-accessible descriptor range (0-9). Unfortunately, the old shell isn't prepared to deal with multiple redirectors of built-in commands, so `set </etc/passwd >/dev/null` causes `set` to execute and then causes the old shell to read `/etc/passwd`(!).

When applied to any command, built-in or not, `<''` and `>''` have no effect in the old shell. This appears to be a relic from the days before `$*` and `$@`.†

† This undocumented misfeature of the shell was discovered by a naive and serendipitous user who was pleasantly surprised to find that `<$1` in a shell script ‘did the right thing’ whether the script was invoked with no arguments or with one, and commented upon this surprise.

3.5. Name-to-i-number translations

The code to run down `$PATH` looking for a command executed more system calls which translate file names to i-numbers than necessary; upon finding a command, it would *access* (2) the file, then *stat* (2) it.

3.6. Exit

On many (by intent, all) UNIX systems, a program which does not use the standard I/O library (*stdio*) [Kernighan1979a] will not cause any part of *stdio* to be loaded with it. This is not true on SunOS 3.0, for example; a program such as the shell which does not use *stdio* still gets some of *stdio* loaded with it, due to *exit* calling *fflush*, and that in turn causes some *malloc* to be loaded. Sun's *malloc* includes 8,192 bytes of BSS (uninitialised data segment) containing its initial free block headers. This seems excessive, given that programs often use *malloc* in an attempt to *conserve* memory.

4. Fixes in the new shell

4.1. Memory Management

Our original fix to the memory allocator, to make it co-exist with the C library and work in general, was to delete `blok.c` thus invoking *malloc* (3) and to rewrite `stak.c` from scratch to use *malloc* (3). Much later we discovered that an alternative, less clean and less robust fix is to just delete private *directory* (3) functions, make *chkid* reject zero addresses, and provide a private *realloc* in `blok.c` which implements the semantics of *realloc* (3) using the private *malloc* and *free*, though one must also increase the values of `BRKINCR` and `BRKMAX` to at least the page size on some systems. This apparently works by causing every memory fault to increase the data segment enough to cover the largest allocation request normally seen inside the shell. The new shell does not use this fix.

The new shell uses *pushstak* and the other interface macros and functions where needed, and *pushstak* now arranges to grow the top item of *stak* storage as needed. The performance impact of this has not been measured, but appears to be insignificant; in any case, this checking is necessary. `stak.c` has been completely rewritten from scratch (see the Appendix).

We now simulate the single pointer to several interwoven stacks of *stak* storage by attaching a pointer to the previous *stak* item to each new *stak* item as it is allocated, and retaining one watermark pointer per stack.

We layer another function (*shfree*), on top of *free* (3), and the new shell is compiled with `#define free shfree` and without the old shell's `#define alloc malloc`, *alloc* being the name by which the heap allocator is invoked. *shfree* rejects attempts to free the address zero, addresses in the stack segment or of *stak* storage; *free* (3) is used directly to free *stak* storage. To distinguish heap storage from *stak* storage, the new shell's allocator attaches an integer containing a magic number, different for heap and *stak* storage, to each item of storage allocated. This costs a little bit of memory, but experiments on a PDP-11 suggest that this is not a serious problem.

We simply use *malloc* (3) and related functions, and thus require none of the complicated machinery for restarting faulting instructions. This has the pleasant side-effect that a buggy shell wielding a wild pointer typically dumps core immediately, instead of growing its stack segment until the kernel kills it (producing a multi-megabyte core dump) minutes later.

4.2. Here Documents

We have repaired both of the here document bugs, with one minor loss of generality: the here document delimiter must not exceed CPYSIZ bytes. CPYSIZ bytes are now written to the first temporary file (until end-of-file is read), and the remaining fractional line is copied back to the start of the copying buffer, which is 2*CPYSIZ bytes long.

4.3. Directory Reading and Wildcard Expansion

We solved the messy problems of reading directories by deleting the private functions and using only the C library *directory*(3) functions to read directories. The shell was modified to call the new function *openqdir* instead of calling *opendir* directly; *openqdir* passes to *opendir* a copy of the file name with the 0200 bit, used by the shell internally to mark quoted characters (e.g. the first character of a command argument such as **\?***), stripped from each character.

We also discovered that the code that implemented negated character classes (for example, `[!a-z]`) in the old shell was incorrect and had only worked by chance; Henry Spencer replaced the incorrect code with robust, working code.

4.4. I/O Redirection

The new shell is prepared to save multiple standard descriptors by duplicating them to whichever descriptors above the normal range are free.

I/O redirections now behave as one would expect: since the empty filename refers to the current directory, <`''` will open the current directory on some systems, and >`''` will, one hopes, fail with an error message.

4.5. Name-to-i-number translations

Use of *access* is inappropriate in the shell, as one wants to check against the shell's effective ids, and unnecessary, as one can easily check the permission bits obtained from the *stat*. This is faster because each system call, such as *access* or *stat*, which takes a filename as a parameter must translate it to the (device-number, i-number) pair used internally by UNIX to refer to files. This translation is relatively slow because it typically requires disk accesses, even on systems with *namei* caches. Still faster would be to simply try to *exec*(2) each filename in turn, and examine *errno* afterward; this will not work for the *type* (a.k.a. *whatis*) built-in, though, and we have not done this.

4.6. Exit

malloc is not an issue in the new shell, but unwanted static *stdio* buffers do take quite a bit of data space. Defining `exit(n) { _exit(n); }` to avoid *stdio* significantly reduces the shell's size, thus reducing the time needed to *fork(2)* and speeding command execution.

5. Methods

Debugging the shell is more difficult than one might expect. Initial debugging was largely by inspired guesses and tedious experimentation, due to the difficulty of examining multi-megabyte core dumps, which tend to be uninformative anyway.

Once the shell was made to stop catching SIGSEGV, it was possible to use debuggers to examine core dumps produced by buggy shells and produce stack traces, but lacking a truly useful debugger (such as *pi* (9.1)), [Cargill1986a] we resorted, in the main, to printing interesting variables (with the shell's *prs* and *prn*, not *printf* (3)) and thinking about the output. One other helpful technique was to insert magic numbers into each instance of each relevant data structure when the instance was created, then check periodically for the presence of the number, and clear the number upon destruction of the instance. This simple technique alone was a great help in keeping the shell sane by detecting corruption and confusion early. We also linked the shell with a debugging version of *malloc* supplied by Sun (`/usr/lib/debug/malloc.o`), which checks the arena for consistency.

6. Conclusions

Debugging would certainly have been easier if the assumptions in the old shell code had been documented; we hope that this paper will save shell maintainers many hours. The new shell appears to be quite portable and has been run on the DEC PDP-11 under V7, Sun-3 under SunOS 3.x, Sun-4 under SunOS 4.0, and MIPS M/1000.

Our new shell now contains comments which describe most of the newly-discovered assumptions which had been hidden in the old shell.

Unfortunately, this version is not generally available, as it is derived from Ninth Edition code. The new version of `stak.c`, however, is not licensed and is reprinted in the Appendix to this paper.

7. Acknowledgements

Henry Spencer of the University of Toronto's Department of Zoology hired the author to perform the work described above, commented on drafts of this paper, ran various versions of the new shell as `/bin/sh` on his machines while we discovered new undocumented properties of the shell, and was patient while bugs were found and fixed. The Department funded this work.

Cray Research found some of the places in which *pushstak* should have been used in the old shell and repaired them, and fixed *pushstak*, in order to make the shell run on Crays, at least some models of which are incapable of restarting instructions which abort due to memory faults.

Dennis Ritchie incorporated Cray's fixes into the Ninth Edition shell, and urged the author to continue attempting to fix the shell rather than throwing it out and reimplementing it. (He was of course right, in part due to the subtleties of getting details such as quoting and *eval* just right. Nevertheless, a future reimplementation of the shell would benefit by using more of the tools available in the C library and elsewhere, possibly including *yacc* and *lex*.) Dennis also provided very helpful comments on a draft of this paper.

Ian Darwin, Beverly Erlebacher and Tom Glinos proof-read drafts of this paper and contributed helpful suggestions.

Any errors remaining in this paper are the responsibility of the author.

References

Bourne1978a.
S. R. Bourne, ‘‘UNIX Time-Sharing System: The UNIX Shell,’’ *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1971-1990, 1978.

Cargill1986a.
T. A. Cargill, ‘‘The Feel of Pi,’’ *Proc. Winter Usenix Conf. 1986*, 1986.

Johnson1978a.
S. C. Johnson and D. M. Ritchie, ‘‘UNIX Time-Sharing System: Portability of C Programs and the UNIX System,’’ *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 2021-2048, 1978.

Kernighan1979a.
Brian W. Kernighan and Dennis M. Ritchie, ‘‘UNIX Programming − Second Edition,’’ *UNIX Programmer's Manual, Seventh Edition*, January, 1979.

Kernighan1984a.
Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment,* Prentice-Hall, 1984.

Korn1985a.
David G. Korn and Kiem-Phong Vo, ‘‘In Search of a Better Malloc,’’ *Proc. Summer Usenix Conf. 1985*, pp. 489-506, June 1985.

Ritchie1987a.
Dennis Ritchie, *private communication*, 1987.

Shannon1988a.
Bill Shannon, *private communication*, November, 1988.

Appendix: the new stak.c, minus debugging #ifdefs

This code was all written by the author; it is not subject to any source licences.

```
/* replaces @(#)stak.c 1.4 */

/*
 * UNIX shell
 *
 * Stacked-storage allocation.
 *
 * Maintains a linked stack (of mostly character strings), the top (most
 * recently allocated item) of which is a growing string, which pushstak()
 * inserts into and grows as needed.
 *
 * Each item on the stack consists of a pointer to the previous item
 * (the "stakbsy" pointer; stakbsy points to the top item on the stack), an
 * optional magic number, and the data.  There may be malloc overhead storage
 * on top of this.
 *
 * Pointers returned by these routines point to the first byte of the data
 * in a stack item; users of this module should be unaware of the "stakbsy"
 * pointer and the magic number.  To confuse matters, stakbsy points to the
 * "stakbsy" linked list pointer of the top item on the stack, and the
 * "stakbsy" linked list pointers each point to the corresponding pointer
 * in the next item on the stack.  This all comes to a head in tdystak().
 *
 * Geoff Collyer
 */

/* see also stak.h */

#include "defs.h"

#undef free				/* refer to free(3) here */

#define STMAGICNUM 0x1235		/* stak item magic */
#define HPMAGICNUM 0x4276		/* heap item magic */
#define MAGICSIZE BYTESPERWORD		/* was once zero */

/* imports from libc */
extern char *malloc(), *realloc();
extern char *memcpy(), *strcpy();

/* forwards */
char *stalloc(), *growstak(), *getstak();

unsigned brkincr = BRKINCR;		/* used in stak.h only */

static char *
tossgrowing()				/* free the growing stack */
{
	if (stakbsy != 0) {		/* any growing stack? */
		register struct blk *nextitem;

		/* verify magic before freeing */
		if (((int *)Rcheat(stakbsy))[1] != STMAGICNUM)
			error("tossgrowing: bad magic on stack");
		((int *)Rcheat(stakbsy))[1] = 0;	/* erase magic */

		/* about to free the ptr to next, so copy it first */
		nextitem = stakbsy->word;
		free((char *)Rcheat(stakbsy));
		stakbsy = nextitem;
	}
}

static char *
stalloc(asize)			/* allocate requested stack space (no frills) */
int asize;
{
	register char *newstack;
	register int size = asize;

	newstack = malloc((unsigned)(sizeof(struct blk) + MAGICSIZE + size));
	if (newstack == 0)
		error(nostack);

	/* stack this item */
	*((struct blk **)Rcheat(newstack)) = stakbsy; /* point back at old stack top */
	stakbsy = (struct blk *)Rcheat(newstack);	/* make this new stack top */
	newstack += sizeof(struct blk);		/* point at the data */

	/* add magic number for verification */
	*((int *)Rcheat(newstack)) = STMAGICNUM;
	newstack += MAGICSIZE;
	return newstack;
}

static char *
grostalloc()			/* allocate growing stack */
{
	register int size = BRKINCR;

	/* fiddle global variables to point into this (growing) stack */
	staktop = stakbot = stakbas = stalloc(size);
	stakend = stakbas + size - 1;
}

/*
 * allocate requested stack.
 * staknam() assumes that getstak just realloc's the growing stack,
 * so we must do just that.  Grump.
 */
char *
getstak(asize)
int asize;
{
	register char *newstack;
	register int staklen;

	/* + 1 is because stakend points at the last byte of the growing stack */
	staklen = stakend + 1 - stakbas;	/* # of usable bytes */
	newstack = growstak(asize - staklen);		/* grow growing stack to asize */
	grostalloc();	/* allocate new growing stack */
	return newstack;
}

/*
 * set up stack for local use (i.e. make it big).
 * should be followed by 'endstak'
 */
char *
locstak()
{
	if (stakend + 1 - stakbot < BRKINCR)
		(void) growstak(BRKINCR - (stakend + 1 - stakbot));
	return stakbot;
}

/*
 * return an address to be used by tdystak later,
 * so it must be returned by getstak because it may not be
 * a part of the growing stack, which is subject to moving.
 */
char *
savstak()
{
	assert(staktop == stakbot);	/* assert empty stack */
	return getstak(1);
}

/*
 * tidy up after 'locstak'.
 * make the current growing stack a semi-permanent item and
 * generate a new tiny growing stack.
 */
char *
endstak(argp)
register char *argp;
{
	register char *oldstak;

	*argp++ = 0;	/* terminate the string */
	oldstak = growstak(-(stakend + 1 - argp));	/* reduce growing stack size */
	grostalloc();	/* alloc. new growing stack */
	return oldstak; /* perm. addr. of old item */
}

/*
 * Try to bring the "stack" back to sav,
 * and bring iotemp's stack back to iosav.
 */
tdystak(sav, iosav)
register char *sav;		/* returned by growstak(): points at data */
register struct ionod *iosav;	/* an old copy of iotemp (may be zero) */
{
	rmtemp(iosav);			/* pop temp files */
	if (sav != 0 && ((int *)Rcheat(sav))[-1] != STMAGICNUM) /* sav -> data */
		error("tdystak: bad magic in argument");

	/*
	 * pop stack to sav (if zero, pop everything).
	 * sav is a pointer to data, not magic nor stakbsy link.
	 * stakbsy points at the ptr before the data & magic.
	 */
	while (stakbsy != 0 && (sav == 0 ||
	   (char *)stakbsy != sav - sizeof(struct blk) - MAGICSIZE))
		tossgrowing();		/* toss the stack top */
	grostalloc();		/* new growing stack */
```

```
}

stakchk()			/* reduce growing-stack size if feasible */
{
	if (stakend - staktop > 2*BRKINCR) /* lots of unused stack headroom */
		(void) growstak(-(stakend - staktop - BRKINCR));
}

char *			/* address of copy of newstak */
cpystak(newstak)
char *newstak;
{
	return strcpy(getstak(strlen(newstak) + 1), newstak);
}

char *			/* new address of grown stak */
growstak(incr)		/* grow the growing stack by incr */
int incr;
{
	register char *oldbsy;
	unsigned topoff, botoff, basoff;
	int staklen;

	if (stakbsy == 0)	/* paranoia */
		grostalloc();	/* make a trivial stack */

	/* paranoia: during realloc, point at previous item in case of signals */
	oldbsy = (char *)stakbsy;
	stakbsy = stakbsy->word;

	topoff = staktop - oldbsy;
	botoff = stakbot - oldbsy;
	basoff = stakbas - oldbsy;

	/* + 1 is because stakend points at the last byte of the growing stack */
	staklen = stakend + 1 + incr - oldbsy;

	if (staklen <= sizeof(struct blk) + MAGICSIZE)	/* paranoia */
		staklen = sizeof(struct blk) + MAGICSIZE;

	if (incr < 0) {
		/*
		 * V7 realloc wastes the memory given back when
		 * asked to shrink a block, so we malloc new space
		 * and copy into it in the hope of later reusing the old
		 * space, then free the old space.
		 */
		register char *new = malloc((unsigned)staklen);

		if (new == NIL)
			error(nostack);
		(void) memcpy(new, oldbsy, staklen);
		free(oldbsy);
		oldbsy = new;
	} else {
		/* get realloc to grow the stack to match the stack top */
		if ((oldbsy = realloc(oldbsy, (unsigned)staklen)) == NIL)
			error(nostack);
	}
	stakend = oldbsy + staklen - 1;		/* see? points at the last byte */
	staktop = oldbsy + topoff;
	stakbot = oldbsy + botoff;
	stakbas = oldbsy + basoff;

	/* restore stakbsy after realloc */
	stakbsy = (struct blk *)Rcheat(oldbsy);
	return stakbas;		/* addr of 1st usable byte */
}

/* ARGSUSED reqd */
addblok(reqd)			/* called from main at start only */
unsigned reqd;
{
	if (stakbot == 0)	/* called from main, 1st time */
		grostalloc();	/* allocate initial arena */
	/* else won't happen */
}

/*
 * Heap allocation.
 */
char *
alloc(size)
unsigned size;
{
	register char *p = malloc(MAGICSIZE + size);

	if (p == NIL)
		error(nospace);

	*(int *)Rcheat(p) = HPMAGICNUM;
	p += BYTESPERWORD;		/* fiddle ptr for the user */
	return p;
}

/*
 * the shell's private "free" - frees only heap storage.
 * only works on non-null pointers to heap storage
 * (below the data break and stamped with HPMAGICNUM).
 * so it is "okay" for the shell to attempt to free data on its
 * (real) stack, including its command line arguments and environment,
 * or its fake stak.
 * this permits a quick'n'dirty style of programming to "work".
 * the use of sbrk is relatively slow, but effective.
 */
shfree(p)
register char *p;
{
	extern char *sbrk();

	if (p != 0 && p < sbrk(0)) {		/* plausible data seg ptr? */
		register int *magicp = (int *)Rcheat(p) - 1;

		/* ignore attempts to free non-heap storage */
		if (*magicp == HPMAGICNUM) {
			*magicp = 0;	/* erase magic */
			p -= BYTESPERWORD; /* get orig. ptr back */
			free(p);
		}
	}
}
```