System V Enhancements Overview

Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part Number: 800-1541-03
Revision: A of 19 September 1986

Contents

Tables

1

System V Compatibility Package

1

System V Compatibility Package

This overview document is intended for both users and programmers who want to learn about System V enhancements in Release 3.2. The chapters on system calls and library routines, and the chapters on System V programming considerations, are of interest only to programmers.

1.1. Introduction

Release 3.2 offers Sun users nearly complete System V compatibility. The compatibility package allows programmers to write software that meets the Base Level of the System V Interface Definition (SVID) in all but a few minor cases, documented below. This release represents the first phase of joint efforts by AT&T and Sun to unify the different versions of UNIX.

System V and Berkeley UNIX are not radically different, either in the interface they present to the user, or the routines they provide for the programmer. They are derived from UNIX systems written by Ken Thompson and Dennis Ritchie in the mid-seventies, and many features are essentially unchanged since then. Both versions have their merits: because of support by AT&T, System V is well-suited to the commercial marketplace; and because of its advanced features, Berkeley UNIX is well-suited to the technical marketplace. Nonetheless, a converged version suited to both markets is even better.

The System V compatibility package permits programmers to write and test software targeted for either System V or Berkeley UNIX. Users who acquire software that runs only on System V can run it by means of the compatibility library. Commands, system calls, and library routines can be drawn concurrently from either the Berkeley or System V set. It's possible to have one window that uses Berkeley UNIX by preference, and another window that runs System V by preference.

1.2. A Brief History

In early 1985, AT&T released the *System V Interface Definition* (SVID). This was a major step because it made explicit exactly what was standard about System V, and by omission, what was not. In late 1985, Sun and AT&T agreed to work together to converge the two major strands of UNIX into a single system. Sun's Release 3.2 combines System V with Berkeley UNIX. In the second phase of Sun's System V compatibility package, System V programs will be able to make use of SunView and other Sun-enhanced libraries (now this is possible only with BSD programs).

There were seven cases to be considered in the process of convergence:

1. Features common to both versions – do nothing.
2. Features with minor differences – do nothing (no difference in typical usage).
3. Features where System V is superset of Berkeley UNIX – add to regular release.
4. Features where Berkeley UNIX is a superset of System V – leave it that way.
5. Features that exist only in System V – put in a compatibility directory.
6. Features that exist only in Berkeley UNIX – leave it that way.
7. Features that cannot coexist – resolve on a case-by-case basis.

1.3. How the Compatibility Tools Work

System V programs that are upwards compatible with those in Berkeley UNIX have already been added to the regular system directories. For example, `/bin/sh` is the new improved Bourne shell, and `/bin/make` has all the System V enhancements.

Programs that existed only on System V have been added to a regular system directories as well. For example, the text manipulation programs `cut` and `paste` both reside in `/usr/bin`.

System V programs that are incompatible with those in Berkeley UNIX reside in the directory `/usr/5bin`. For example, `/usr/5bin/stty` has an entirely different set of options from `/bin/stty`. If you want to use System V programs by preference, simply include `/usr/5bin` early in your path, as in these lines from the `.login` or `.profile` files:

```
(csh) set path = (/usr/5bin /bin /usr/bin /usr/ucb .)

(sh)  PATH=/usr/5bin:/bin:/usr/bin:/usr/ucb::
      export PATH
```

The directories `/usr/5bin`, `/usr/5lib`, and `/usr/5include` contain material that can't yet be converged.

Libraries and include files for compiling System V software reside in `/usr/5lib` and `/usr/5include` respectively. If you want to compile a program written for System V, don't use `/bin/cc` but rather `/usr/5bin/cc`, which will read all the correct include files and load the correct libraries. You may want to make an alias or shell function that invokes the System V compiler, to obviate the need for changing your `PATH`:

```
(csh) alias cc5 /usr/5bin/cc

(sh)  cc5() {
          /usr/5bin/cc $*
      }
```

The directories that constitute the System V compatibility package are optional, requiring 3½ MB of disk space. The `setup` program lets you decide whether or not to load these directories. Because of the merging of System V programs and the kernel additions required to support System V libraries, the root filesystem is about ¼ MB larger than for Sun's Release 3.0.

1.4. Conformance to Base-Level SVID

In what ways does Sun's System V compatibility package not meet the requirements specified by the Base Level of the SVID? The package does have important System V features such as record locking, named pipes, shared memory, semaphores, messages, and an emulation of the revised terminal driver. Here are the only known ways in which Sun's Release 3.2 does not conform to the Base Level of the SVID:

□ The `creat()` and `open()` system calls use the Berkeley semantics to assign files the group of their parent directory. System V assigns files the group of the creating process.

□ For security reasons, the `chown()` system requires root privileges. On System V the owner of a file can give it away. This would make the Berkeley quota mechanism completely unenforceable.

□ The `utime()` system call can't set file time stamps to the current time on NFS-mounted files, and only works on files owned by the caller. System V allows file time stamps to be set by any process with write permission on a file.

□ The `kill()` system call only allows processes to send signals to other processes with the same effective UID. The SVID specifies that a process can send a signal to processes with an effective or real UID that matches the effective or real UID of the sender. Super-user processes can send signals to any other process.

□ The `mknod()` system call cannot be used to create directories, as specified by the SVID; use the `mkdir()` system call instead.

□ The `fcntl()` system call with the `F_SETFL` command setting the `O_NDELAY` flag affects all references to the underlying file. On System V, this `fcntl()` call affects only file descriptors associated with the same file table entry.

Also, the first phase of the System V compatibility package does not fully support some System V terminal interface specifications:

□ 5-bit and 6-bit characters are not supported.

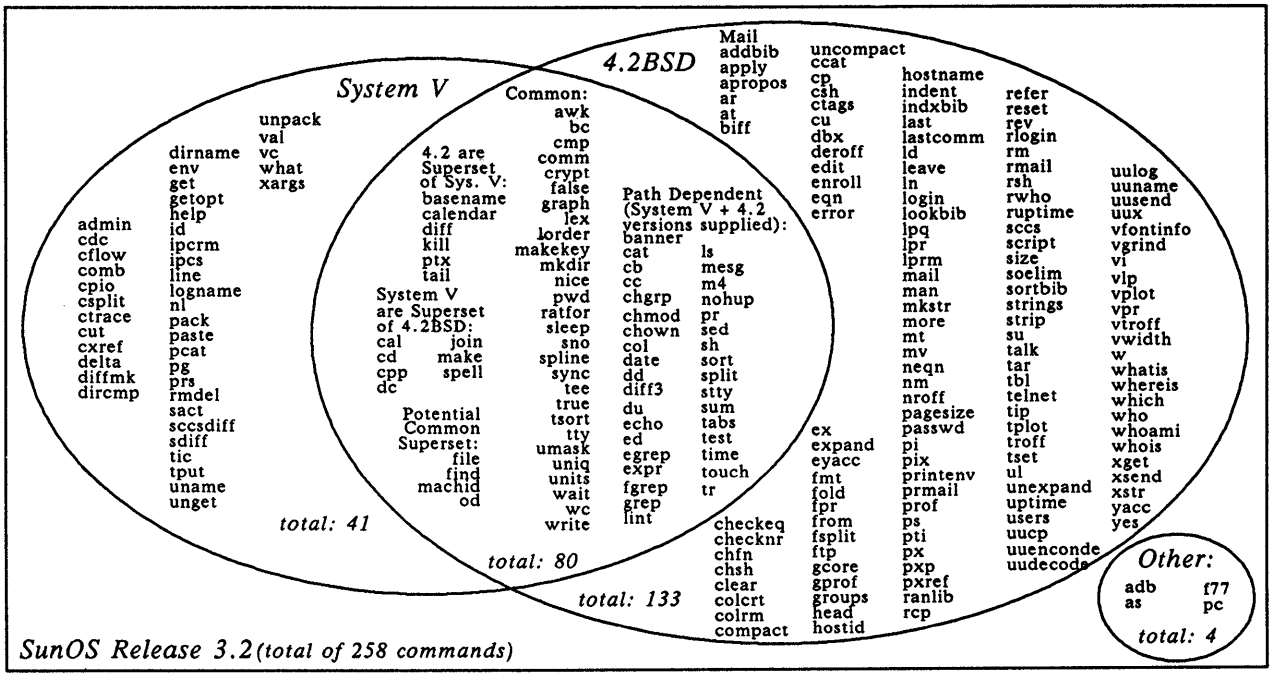
□ `VMIN` is always set to 1 and `VTIME` is always set to 0.

□ The initial default erase and kill characters are not # and @ respectively, but rather [DEL] and [CTRL-U].

1.5. Commands in Release 3.2

This table shows the intersection and union of commands in the System V and Berkeley sets.

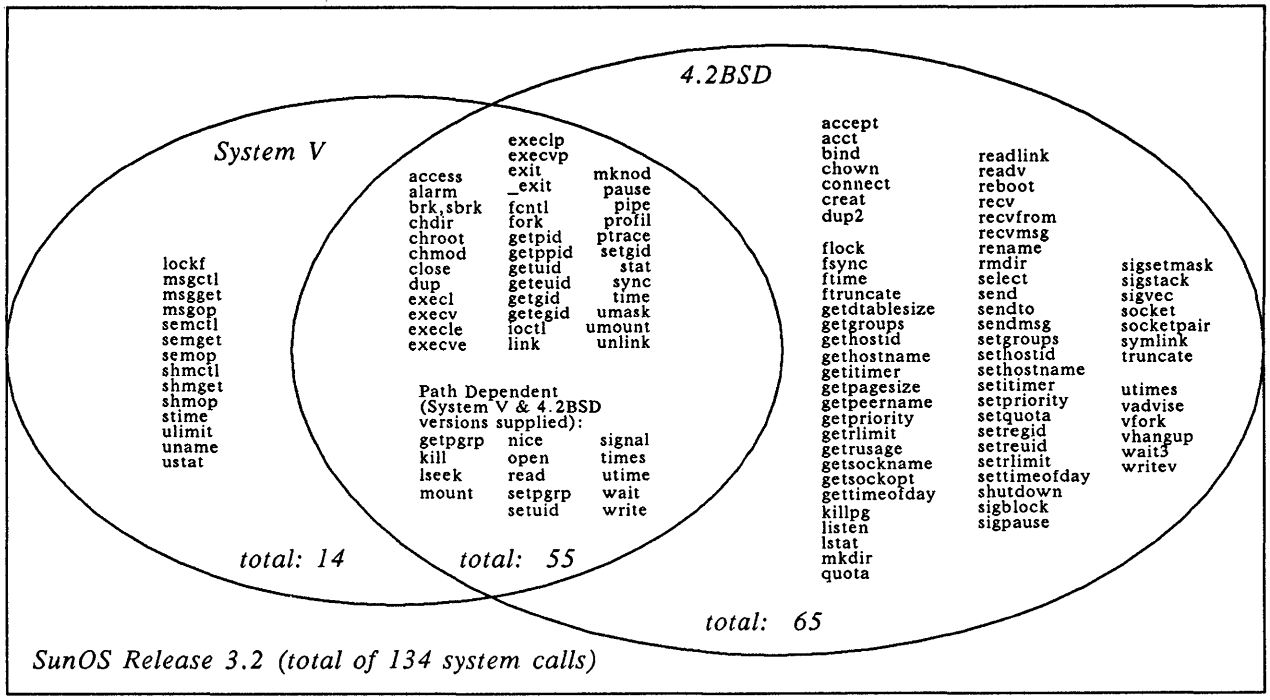
Table 1-1 *Commands in Release 3.2*



1.6. System Calls in Release 3.2

This table shows the intersection and union of system calls in the System V and Berkeley sets.

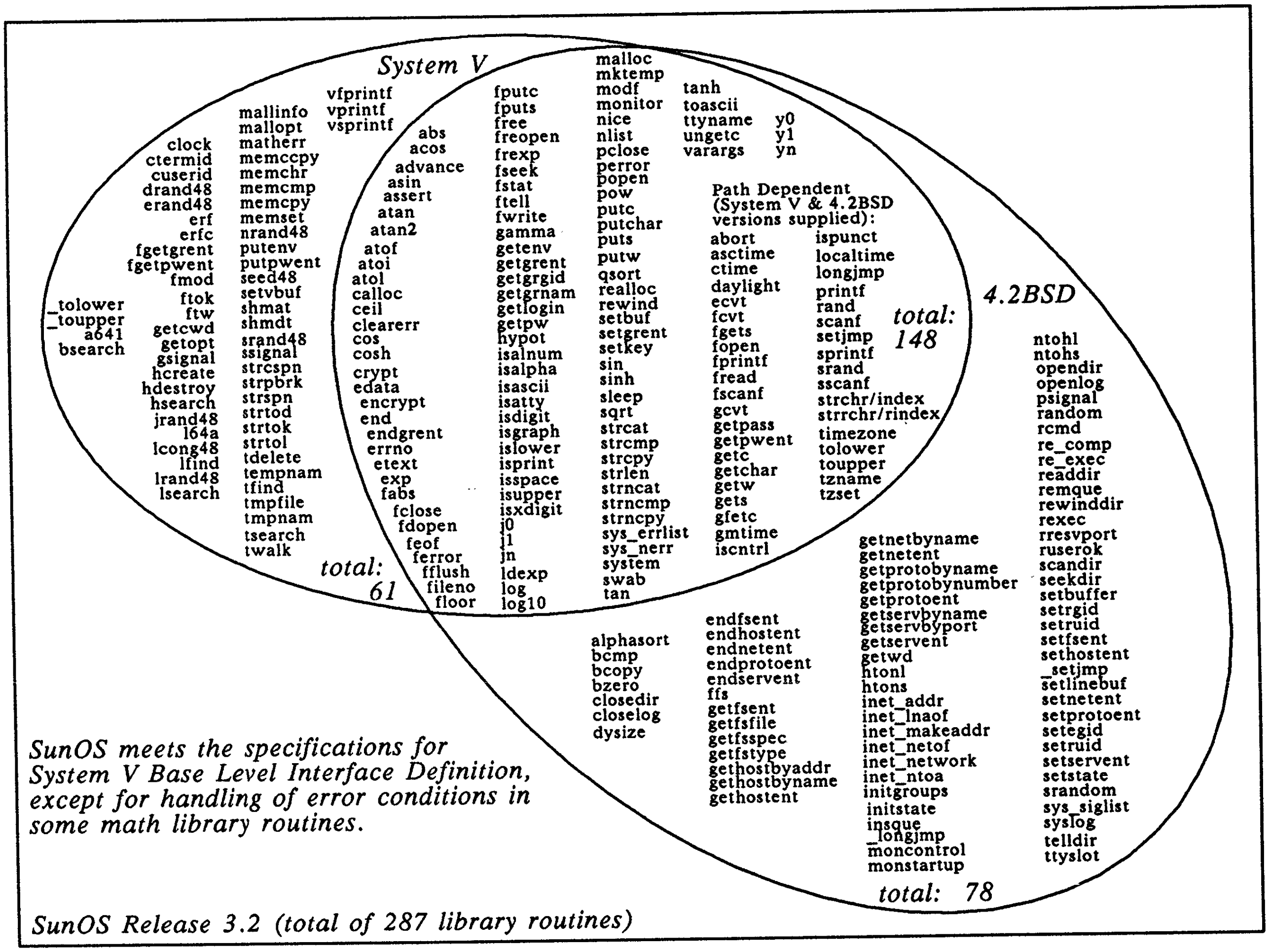
Table 1-2 *System Calls in Release 3.2*



1.7. Library Routines in Release 3.2

This table shows the intersection and union of library routines in the System V and Berkeley sets.

Table 1-3 *Library Routines in Release 3.2*



SunOS meets the specifications for System V Base Level Interface Definition, except for handling of error conditions in some math library routines.

SunOS Release 3.2 (total of 287 library routines)

2

Library Compatibility Issues

2

Library Compatibility Issues

This chapter describes differences between the System V and Berkeley UNIX programming environments in Release 3.2. Programmers who wish to write software that works in either environment will be interested in the material here.

When using System V compatibility libraries, programmers may use Berkeley library routines as long as names don't conflict with any System V routines. This means that System V environment programs can use sockets and other Berkeley features, even when they are linked using the System V library.

2.1. System Calls

The following list describes differences in Release 3.2 between the System V and Berkeley system calls.

`getpgrp()` and `setpgrp()`. There is currently no way to get the Berkeley behavior of `getpgrp()` or `setpgrp()` using the same code in both environments. The undocumented routines `_getpgrp()` and `_setpgrp()` in the System V library provide Berkeley functionality, but those routines are not in the Berkeley library. There is a proposal before the IEEE P1003.1 committee for a job control facility that provides `getpgrp2()` and `setpgrp2()` calls to provide the Berkeley functionality. If the Berkeley `getpgrp()` and `setpgrp()` functionality is required, we can provide these routines in both libraries.

`open()`. In the Berkeley environment, opening a file with the `O_NDELAY` bit set in the open mode does not leave the file descriptor returned, nor the object referred to by that file descriptor, in non-blocking mode. In the System V environment it does.

`read()`. In the Berkeley environment, a `read()` on a file descriptor or from an object in no-delay mode will return –1 and set `errno` to `EWOULDBLOCK` if no data are available. In the System V environment, the `read()` will return a count of 0, which is indistinguishable from end-of-file. The P1003.1 standard has a separate ''non-blocking'' mode that acts like the Berkeley no-delay mode, except that it sets `errno` to `EAGAIN`, not `EWOULDBLOCK`, if no data are available.

`write()`. As with `read()`, the two environments differ in how a `write()` in no-delay mode indicates that there is no buffer space available to store the data to be written.

2.2. Library Routines

The following list describes differences in Release 3.2 between the System V and Berkeley library routines.

`toupper()` and `tolower()`. In the Berkeley environment, they are macros that do not check that their argument is in the domain of the function in question. In the System V environment, they are functions that do check, and `_toupper()` and `_tolower()` are the names of the macros. Using `toupper()` and `tolower()` in either environment as if they were the Berkeley versions is safe but inefficient in the System V environment.

`assert()`. In the Berkeley environment, `_assert()` is an (undocumented) synonym for `assert()`. This is not the case in the System V environment. (This is probably not a concern, since `assert()` is a macro and the version to be used is selected when the library routine is compiled, not when the library is linked with the C library.)

`timezone()`. In the Berkeley environment, this routine returns the name of the time zone associated with a particular offset from GMT and an indication of whether daylight savings time is in effect or not. This is done in a different fashion in the System V environment, and there is no way to do this in a way that works in both environments.

`fopen()`. In the Berkeley environment, opening a file using `fopen()` with an append argument (`"a"` or `"a+"`) causes the file pointer to be placed at the end of the file when it is opened, but performs no other special functions. In the System V environment, the file descriptor for the stream returned has the forced append flag `O_APPEND` set, so that *all* subsequent writes to that stream will be forced to append to the file. It is as if an `fseek(stream,0L,2);` were to be performed before every write.

`fread()`. In the Berkeley environment, if the standard output or error output is line-buffered, `fread()` will write out any buffered data for that stream before reading from the standard input, whether the standard input is line-buffered or not. In the System V environment, if input is read from *any* line-buffered stream, all data for *all* line-buffered streams is written out before the input is read. This is not done if the stream being read from is not line-buffered.

`scanf()`. Note: the Berkeley behavior, in all the cases below, can be considered buggy. Future releases of Sun may be changed to match the System V behavior.

1. In the Berkeley environment, `scanf()` considers space, tab, and newline to be the only whitespace characters; in the System V environment, it also considers formfeed and vertical-tab to be whitespace.
2. In the Berkeley environment, if a match against ordinary characters in the format string (that is, not against a format specification) gets an end-of-file, `scanf()` returns EOF, regardless of whether any matches against a format specification have occurred or not. In the System V environment, it returns the number of matches that have already occurred.
3. In the Berkeley environment, if a match against a format specification fails, `scanf()` doggedly continues matching subsequent fields until it runs out of

input or out of format string. In the System V environment, it quits immediately, leaving the stream pointer positioned to the character that failed to match, and returning the number of successful matches.

4. In the Berkeley environment, `scanf()` considers a string consisting only of a decimal point to match a `%f`, `%e`, or `%g` specification, and returns the value 0.0 for that field. In the System V environment, `scanf()` does not consider this to be a match.

`getpass()`. In the Berkeley environment, if `getpass()` can't open `/dev/tty`, it reads from the standard input. In the System V environment it returns an error.

`nice()`. In the Berkeley environment, `nice()` returns either 0 for success, or –1 for failure, and permits the priority to be put into the range –20 to 20. In the System V environment, `nice()` returns either the new value of the priority for success, or –1 for failure, and does not permit the priority to be outside the range –20 to 19.

`sprintf()`. In the Berkeley environment, `sprintf()` returns a pointer to the string which was its first argument. In the System V environment, it returns the number of characters placed into that string (not counting the terminating null character).

`rand()`. In the Berkeley environment, `rand()` returns a value in the range 0 to $2^{31} - 1$. In the System V environment, on the other hand, it returns a value in the range 0 to $2^{15} - 1$.

Standard I/O Buffering. In the Berkeley environment, if the streams `stdout` or `stderr` refer to a terminal, they are line-buffered. All other streams are fully buffered. In the System V environment, *any* stream that refers to a terminal is line-buffered.

`setuid()`. In the Berkeley environment, `setuid()` sets both the real and effective user ID to the given argument, if this is permitted. In the System V environment, if the current process has an effective user ID of super-user, both the real and effective user ID are set; otherwise, just the effective user ID is set. In this case, the effective user ID can be repeatedly switched between the real user ID and the user ID that the process had when it last did an `exec()`; this permits a setuid program to switch between its real user ID and the user ID to which it was setuid. The `setreuid()` call can be used to perform this function in the same way in both environments (including switching).

`signal()`. In the Berkeley environment, if you catch a signal with `signal()`, the signal action is not changed when the signal handler is entered, but the signal is blocked while in the signal handler. Also, any reads on terminals and other slow devices will, if interrupted, be re-entered as long as no data has been transferred. In the System V environment, the signal action will be reset to `SIG_DFL` when the signal handler is entered, and the signal is *not* blocked. There is a possible race condition here; the `sigvec()` call can be used to establish race-free condition handling in the same way in both environments. Reads on slow devices will return –1 and set `errno` to `EINTR` if interrupted.

sleep(). In the Berkeley environment, signals other than SIGALRM cause that signal's action to take place, but will not break out of the sleep(). In the System V environment, they will break out of the sleep(), and sleep() will return the amount of time remaining before it finishes.

times(). In the Berkeley environment, times() returns either 0 for success, or –1 for failure. In the System V environment, it returns either the number of 60ths of a second since some point in the past for success, or –1 for failure; the point in the past does not change unless the machine is rebooted.

ttyslot(). In the Berkeley environment, ttyslot() returns 0 if it cannot find the slot in /etc/utmp for the terminal. In the System V environment, it returns –1.

Return from main(). A C program's main() routine that falls off the end with no explicit exit() exits with a status of 0 in the Berkeley environment. In the System V environment, it returns a random exit status.

2.3. Execution Environment

There are several programs that do not function the same way in the Berkeley and System V environments. If you plan to execute a specific system command (as opposed to executing a command string supplied by the user), you should consider either hard-coding the path (so that a particular version is used) or setting the PATH environment variable before running the command. The former would cause problems if the command were to move later. The latter, in addition to ensuring that the version from the Berkeley environment, rather than the System V environment, is run, also ensures that a version that the user has in a private bin directory is not run. The latter is the recommended approach, and can be accomplished by doing:

```
char *path_val, *save_path, *malloc();

path_val = getenv("PATH");
if (path_val != NULL) {
    save_path = malloc((unsigned)(strlen(path_val) + 6));
    (void)strcpy(save_path, "PATH=");
    (void)strcat(save_path, path_val);
    putenv("PATH=/usr/ucb:/bin:/usr/bin");
}
/*
 * run the program
 */
if (path_val != NULL) {
    putenv(save_path);  /* restore previous value of PATH */
}
```

Any shell scripts should also set PATH to a standard value (again, the recommended path is /usr/ucb:/bin:/usr/bin), but without exporting it. That way, any interactive shells invoked by that script will have the proper PATH. This can be done in the Bourne shell by saying

```
PATH=/usr/ucb:/bin:/usr/bin
```

without also saying: `export PATH`. Both of these changes are actually a good idea under any circumstances, as they prevent private commands with the same name as the official command from being run. It would be painful, for example, if a user had a special editor interactive editor command named `sed` in his private `bin` directory.

3

System Calls in 4.2 BSD and System V

3

System Calls in 4.2 BSD and System V

3.1. Introduction

This chapter compares system calls in System V to those in 4.2 BSD. It is intended as background information for programmers. If you are already familiar with the two strains of UNIX, or if you are not a programmer, skip this chapter.

In universities, Berkeley UNIX predominates over System V. There are historic, economic, and technical reasons for this. Back when AT&T was prohibited from selling computer systems, they could not legally support UNIX as a product. Because talented people at Berkeley cared about it, and because DARPA chose UNIX as their standard operating system, the University of California took over the development, and in a weird way the support, of UNIX. UC Berkeley never performed normal support functions such as consulting, but they did act as a clearing-house for bug reports and enhancements. Berkeley Software Distributions (BSD) require a Version 7 license from AT&T (the VAX requires a 32V license, which is similar). This license used to cost only $200 for educational institutions. When AT&T came out with System III and System V, they lowered the license fee for commercial organizations, but raised it to $800 for educational institutions. Berkeley UNIX continued to be based on Version 7, so that it would be more affordable for universities. Corporations can purchase less expensive System V licenses, and still obtain Berkeley UNIX.

3.2. Comparison of System Calls

The table on the following pages compares the system calls available on System V and 4.2 BSD. The remainder of this chapter describes, and in some cases justifies, extensions made to these two releases.

Table 3-1 *Comparison of System Calls*

Comparison of System Calls			
Version 7	*System V*	*4.2 BSD*	*description*
		`accept()`	accept a connection on a socket
`access()`	`access()`	`access()`	test file accessibility
`acct()`	`acct()`	`acct()`	turn process accounting on or off
`alarm()`	`alarm()`		schedule an alarm signal for a process
		`bind()`	bind a name to a socket
`brk()`	`brk()`	`brk()`	allocate memory by setting data segment size
`chdir()`	`chdir()`	`chdir()`	change the current working directory
`chmod()`	`chmod()`	`chmod()`	change file access protection mode
		`fchmod()`	ditto, given file descriptor
`chown()`	`chown()`	`chown()`	change file owner and group assignment
		`fchown()`	ditto, given file descriptor
`chroot()`	`chroot()`	`chroot()`	change the root directory for path name searches
`close()`	`close()`	`close()`	close a file descriptor
		`connect()`	initiate a connection on a socket
`creat()`	`creat()`	`creat()`	create a new file or overwrite an existing file
`dup()`	`dup()`	`dup()`	duplicate an open file descriptor
`dup2()`		`dup2()`	duplicate a specific open file descriptor
`exec()`	`exec()`	`exec()`	execute a file
`_exit()`	`_exit()`	`_exit()`	terminate a process
	`fcntl()`	`fcntl()`	control open file descriptors
		`flock()`	apply or remove an advisory lock on an open file
`fork()`	`fork()`	`fork()`	create a new process
		`fsync()`	synchronize file's in-core state with that on disk
`fstat()`	`fstat()`	`fstat()`	locate file status information by file descriptor
`getuid()`	`getuid()`	`getuid()`	get real user identity
`geteuid()`	`geteuid()`	`geteuid()`	get effective user identity
`getgid()`	`getgid()`	`getgid()`	get real group identity
`getegid()`	`getegid()`	`getegid()`	get effective group identity
		`getgroups()`	get group access list
		`gethostid()`	get unique identifier of current host
		`gethostname()`	get name of current host
		`sethostname()`	set name of current host
		`getitimer()`	get value of interval timer
		`setitimer()`	set value of interval timer
		`getpagesize()`	get system page size
		`getpeername()`	get name of connected peer
	`getpgrp()`	`getpgrp()`	get process group ID
`getpid()`	`getpid()`	`getpid()`	obtain process ID of the current process
	`getppid()`	`getppid()`	obtain process ID of the parent process
		`getpriority()`	get program scheduling priority
		`setpriority()`	set program scheduling priority
		`getrlimit()`	get limit on system resource consumption
		`setrlimit()`	set limit on system resource consumption
		`getrusage()`	get information about resource usage
		`getsockname()`	get socket name

Table 3-1 *Comparison of System Calls— Continued*

Comparison of System Calls			
Version 7	*System V*	*4.2 BSD*	*description*
		`getsockopt()`	get options on sockets
		`setsockopt()`	set options on sockets
		`gettimeofday()`	get date and time
		`settimeofday()`	set date and time
`ioctl()`	`ioctl()`	`ioctl()`	perform device control maintenance functions
`kill()`	`kill()`	`kill()`	send a signal to a process or a group of processes
		`killpg()`	send a signal to a process group
`link()`	`link()`	`link()`	create a new link to a file
		`listen()`	listen for connections on a socket
`lock()`			lock a process in primary memory
`lseek()`	`lseek()`	`lseek()`	reposition random read/write pointer
		`mkdir()`	make a directory file
`mknod()`	`mknod()`	`mknod()`	make a directory, ordinary file, or special file
		`mmap()`	map pages of memory onto a file
`mount()`	`mount()`	`mount()`	mount a removable file system for use
`mpx()`			create and manipulate multiplexed files
	`msgctl()`		maintain a message queue access control structure
	`msgget()`		create message queue, ID, and access control structure
	`msgrcv()`		retrieve a message from a message queue
	`msgsnd()`		add a message to a message queue
`nice()`	`nice()`		change process priority
`open()`	`open()`	`open()`	open a file descriptor
`pause()`	`pause()`		suspend a process until receiving a signal
`phys()`			allow a process to access physical addresses
`pipe()`	`pipe()`	`pipe()`	create an interprocess data stream
`pkon()`			establish packet protocol
`pkoff()`			turn off packet driver
	`plock()`		lock and unlock program segments in memory
`profil()`	`profil()`	`profil()`	turn profiling on or off
`ptrace()`	`ptrace()`	`ptrace()`	process tracing for breakpoint debugging
		`quota()`	manipulate disk quotas
`read()`	`read()`	`read()`	read from a file
		`readv()`	read from a file into vector of buffers
		`readlink()`	read value of a symbolic link
		`reboot()`	reboot system or halt processor
		`recv()`	receive a message from a socket
		`rename()`	change the name of a file
		`rmdir()`	remove a directory file
`sbrk()`	`sbrk()`	`sbrk()`	allocate memory by increasing data segment size
		`select()`	synchronous I/O multiplexing
		`send()`	send a message from a socket
	`semctl()`		maintain a semaphore access control structure
	`semget()`		create semaphores, semaphore ID, with access control
	`semop()`		perform array of semaphore maintenance operations
		`setgroups()`	set group access list

Table 3-1 *Comparison of System Calls— Continued*

Comparison of System Calls			
Version 7	*System V*	*4.2 BSD*	*description*
`setuid()`	`setuid()`	`setruid()`	set real and effective user ID
		`setuid()`	set effective user ID
`setgid()`	`setgid()`	`setrgid()`	set real and effective group ID
		`setgid()`	set effective group ID
	`setpgrp()`	`setpgrp()`	set process group ID
		`setquota()`	enable and disable quota on file system
	`shmat()`		attach shared memory segment to calling process
	`shmctl()`		maintain shared memory access control structure
	`shmdt()`		detach shared memory segment from calling process
	`shmget()`		create shared memory ID and access control structure
		`shutdown()`	shut down part of full-duplex socket
`signal()`	`signal()`		specify action to perform upon receipt of a signal
		`sigblock()`	block signals
		`sigpause()`	release blocked signals and wait for interrupt
		`sigsetmask()`	set current signal mask
		`sigstack()`	set and get signal stack context
		`sigvec()`	establish signal vectors
		`socket()`	create an endpoint for communication
		`socketpair()`	create a pair of connected sockets
`stat()`	`stat()`	`stat()`	locate file status information by file name
		`lstat()`	ditto, for symbolic link
`stime()`	`stime()`		set system time
		`swapon()`	add a swap device for interleaved paging
		`symlink()`	make a symbolic link to a file
`sync()`	`sync()`	`sync()`	update the super-block
`syscall()`	`syscall()`		indirect system call
`tell()`		`tell()`	return seek position in file
`time()`	`time()`		obtain the current system time
`ftime()`			fill in structure with system time
`times()`	`times()`	`times()`	obtain a process' time accounting data
		`truncate()`	truncate a file to a specified length
	`ulimit()`		establish a process' file size and data segment limits
`umask()`	`umask()`	`umask()`	set default access mode for file creation
`umount()`	`umount()`	`umount()`	remove a file system from use
	`uname()`		obtain system name, release, version, machine ID
`unlink()`	`unlink()`	`unlink()`	remove a directory entry for a link
	`ustat()`		obtain mounted file system status
`utime()`	`utime()`	`utimes()`	set file update and access time fields
		`vadvise()`	give advice to the paging system
		`vfork()`	span new process in virtual memory
`wait()`	`wait()`	`wait()`	wait for a process to signal or terminate
		`wait3()`	non-blocking wait
`write()`	`write()`	`write()`	write to a file
		`writev()`	write to a file from vector of buffers

3.3. New System Calls on System V

What follows is an brief overview of system calls on System V.

The `ulimit()` system call allows a process to establish a maximum file size for itself and for any child processes. A subsequent `write()` system call returns an error status when it would increase a file's size beyond the maximum. This is a cheap form of resource limit or quota facility.

The `ustat()` system call obtains information about mounted filesystems. This information includes the total number of free blocks and free inodes, as well as filesystem and disk pack names.

The `uname()` system call allows programs to obtain the network node name, UNIX system release and version number, and a hardware identification string. This is similar in function to the `gethostname()` system call on 4.2 BSD.

Processes can lock text segments, data segments, or both, into memory. Process segments are immune to swapping when locked using the `plock()` system call. Process locking is useful for real-time applications.

The `fcntl()` system call allows programs to obtain new file descriptors, and retrieve or set status information for open files. It is such a useful facility that it was picked up on 4.2 BSD.

The concept of process group was added in System III, and also by Berkeley. Each process has a process group identity associated with it. The process group identity is really the process identity of an ancestor that invoked the `setpgrp()` system call. The ancestor is the group leader. The group identity is inherited when new processes are created, and like `tty` groups, process groups establish commonality for sending signals.

System V includes first-in-first-out (FIFO) files, which are also called named pipes. This allows processes to open this special file, using it for communication just like a pipe, but between possibly unrelated processes. FIFO files are created using the `mknod()` system call.

The System V shared-memory facility provides common areas in memory for sharing data between processes. Facilities are also provided to control access to shared memory, and to synchronize updating by multiple processes. Shared memory is useful for database applications, among other things.

The System V semaphore facility provides a process synchronization mechanism, which can be used to schedule processes that modify shared system resources. Resources are locked for updating by one process at a time, and update ordering is supported.

The System V message queue facility provides another form of inter-process communication. Messages are a convenient way for unrelated processes to share data. Since only the message is shared, processes can remain independent of each other's internal data structures. The message queue facility is used to establish one or more queues for inter-process communication. Messages are placed on specific queues for subsequent retrieval. A control data structure is included to allow restricted access to individual message queues.

Many internal performance improvements were included in System V, some inspired by Berkeley UNIX. File system block sizes were doubled to implement a 1K filesystem. A physical I/O buffer facility, and a larger buffer pool are available. Improved system table search algorithms, faster `fork()` and `open()` system calls, C compiler improvements, and faster versions of C library I/O routines all contributed to system performance gains. However, the 4.2 BSD fast filesystem is significantly faster than the System V 1K filesystem in many applications.

3.4. New System Calls on 4.2 BSD

The networking enhancements in 4.2 BSD were one reason for its success. The Berkeley inter-process communication (IPC) facilities have already gained wide acceptance. Here are the system calls that relate to networking: `accept()`, `bind()`, `connect()`, `getpeername()`, `getsockname()`, `getsockopt()`, `setsockopt()`, `listen()`, `recv()`, `select()`, `send()`, `shutdown()`, `socket()`, and `socketpair()`. Sockets and IPC facilities are discussed in the "IPC Primer" chapter of the manual, *Networking on the Sun Workstation.*

The `fchmod()` and `fchown()` system calls are like the traditional `chmod()` and `chown()` system calls, except they work with file descriptors instead of file names.

The `flock()` system call performs advisory file locking. Because it does not deal with record locking, it is inferior to, and should be replaced by, `lockf()` in the *System V Interface Definition* (SVID).

The `fsync()` system call is like `sync()`, except it writes a single file's buffer, rather than all system buffers, to disk. It is useful for database work.

On Berkeley UNIX, it is possible to be a member of more than one group at a time, which makes the group mechanism more useful than on System V. The `getgroups()` system call yields the group access list for a process, while `setgroups()` changes the group access list.

Like the `uname()` system call on System V, `gethostname()` retrieves the name of the host machine, while `sethostname()` changes its name. This is particularly valuable in a network environment.

The `getitimer()` and `setitimer()` system calls provide user access to interval timing. This is useful for program measurement and instrumentation. No good facility for this exists on System V, although `times()` provides a rough approximation.

The `nice()` system call was replaced by `setpriority()`, which provides finer control over scheduling. Previously it was impossible to ask for a process' priority, but this is now possible with `getpriority()`.

Limits can be set on system resources such as file size, data size, stack size, coredump size, and memory usage. These limits can be obtained with `getrlimit()` and changed with `setrlimit()`. The system's resource usage can also be queried with `getrusage()`.

The system clock keeps time in thousandths of a second, rather than in sixtieths of a second. This was an attempt to make timing more accurate, but given the current state of hardware, it is an idea whose time hasn't come yet. The clock can be queried with `gettimeofday()`, and set with `settimeofday()`. Library routines provide backward compatibility with the old `time()` system call.

On 4.2 BSD, filenames are no longer limited to 14 characters, because of a new filesystem and directory format. The new systems calls `mkdir()` and `rmdir()` provide atomic means to create and remove directories. Files can be moved with the `rename()` system call. It is also possible to truncate files to a specified length with `truncate()`.

The `mmap()` system call is an attempt to provide shared memory by mapping virtual memory into a file. Unfortunately it is incompletely implemented.

Unlike System V, 4.2 BSD provides disk quotas, which are valuable for heavily used filesystems. The `setquota()` system call enables and disables quotas for a filesystem, while `quota()` allows these quotas to be altered.

Symbolic links are like hard links except they can cross filesystems. They are created with `symlink()`, manipulated with `readlink()`, and queried with `lstat()`. Symbolic links are useful for networking, because they can span not only filesystems, but machine boundaries as well.

It is possible to set just the real user or group ID with `setruid()` and `setrgid()`, or just the effective user or group ID with `seteuid()` and `setegid()`. Of course, both may still be set with `setuid()` and `setgid()`. All these library routines are based on the system calls `setreuid()` and `setregid()`.

The designers of 4.2 BSD felt that the old signal mechanism was broken, because signals could interrupt previous signals, sometimes causing dangerous race conditions.† Signal delivery on 4.2 BSD resembles the model of hardware interrupts: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a signal handler with `sigvec()`, or specify that a signal be blocked, or restore a default action on receipt of signal. Signals are sent to a process with `kill()`, or to an entire process group with `killpg()`. Signals can be blocked for a section of code with `sigblock()`, and restored with `sigsetmask()`. In blocked conditions, programs can wait for a signal with `sigpause()`. Layered signal handlers can be built with `sigstack()`. Fortunately, the old `signal()` facility is implemented as a library routine, so old programs don't need to be modified (although signal behavior is incompatible).

Since 4.2 systems have virtual memory, the `vfork()` system call is provided to prevent unnecessary copying on forks, and the `vadvise()` call is available to influence paging behavior. However, the System V `fork()` is fater then the 4.2 `vfork()` and doesn't have a problem with children modifying parents' address space.

† Interestingly, IBM expropriated the Berkeley signal facility for their implementation of System V.

3.5. Conclusion

Both Berkeley UNIX and System V contain many new system calls, some of which may gain wider acceptance in the long run. To achieve maximum portability, application programmers should avoid any system calls not available on both major versions of UNIX. System programmers, on the other hand, may require non-standard facilities to get the job done. Look before you leap.

4

Library Routines in 4.2 BSD and System V

4

Library Routines in 4.2 BSD and System V

How can a piece of code that is an order of magnitude too large be considered reliable? There is that much more that must be understood in order to make changes. Library functions ... are one way to reduce the apparent complexity of a program; they help to keep program size manageable, and they let you build on the work of others, instead of starting from scratch each time.

Kernighan and Plauger, *The Elements of Programming Style*

4.1. Introduction

Anyone who peruses old UNIX code is shocked by the sheer number of instances where programmers obviously have coded their own functions in preference to using common library routines. Especially prevalent are private string-handling routines, analagous to `strcpy()` and `strcmp()`. One reason for this is that many programs were written before such library routines became available. Another is that the UNIX system continues to grow, making it difficult to keep up with all the new library functions that appear. To counter this trend, here is a comparison of library routines available on Version 7, System V, and 4.2BSD.

Years from now, we will look back on today's software with a jaundiced eye. No doubt, we will be surprised to see custom binary search algorithms, tree management routines, and record locking schemes, despite the existence of perfectly good library routines for these purposes. In particular, the *System V Interface Definition* (SVID) includes specifications for each of these functions.

There are good reasons to use library routines in preference to ''rolling your own''. First, programs are easier to understand and maintain when other programmers are already familiar with – and can trust – calls to library routines. Second, library routines are often faster than hand-coded functions because of library optimizations. Sun, for example, delivers improved versions of `malloc()` and the Standard I/O Library. Third, library routines are maintained by other people, and changes can be coordinated with those made to other parts of the system. Finally, library routines are documented, and get improved. Private versions of functions tend to be undocumented and hidden in specific programs, waiting to surprise users when new releases are installed.

4.2. Comparison of Library Routines

As is evident from the table below, System V has the most library routines, with 4.2BSD placing a distant second. More than half of the System V additions were actually part of System III, but they are credited to System V here. The `curses` package and the `termcap` library routines are listed as one-line items.

The C compilers on System V and 4.2BSD (almost identical versions of the Portable C Compiler) have been improved since Version 7. Both compilers support enumeration data types, non-unique structure member names, and the `void` data type (for functions not returning a value). Long program identifiers are supported in 4.2, but were not added to System V until release 2. Both System V and 4.2BSD offer profiled function libraries as an aid for software performance analysis.

Table 4-1 *Comparison of Library Routines*

Comparison of Library Routines

routine	*V7*	*SV*	*4.2*	*description*
`a64l()`		x		convert base 64 ASCII string to long integer
`abort()`	x	x	x	create a program fault
`asctime()`	x	x	x	convert a time zone data structure to string format
`atof()`	x	x	x	convert character string to floating point
`atoi()`	x	x	x	convert character string to integer
`atol()`	x	x	x	convert character string to long integer
`bcmp()`			x	compare a byte array
`bcopy()`			x	copy a byte array
`bzero()`			x	zero a byte array
`bsearch()`		x		binary table search routine
`calloc()`	x	x	x	allocate an initialized array space
`clock()`		x		obtain process CPU time
`crypt()`	x	x	x	encrypt a password using setkey and encrypt functions
`ctime()`	x	x	x	convert datlpe and time to string format
`ecvt()`	x	x	x	convert floating point to string format
`encrypt()`	x	x	x	encrypt a key using the DES algorithm
`endgrent()`	x	x	x	end group file processing
`endpwent()`	x	x	x	end password file processing
`endutent()`		x		end processing of the accounting file
`fcvt()`	x	x	x	convert floating point to Fortan F string format
`free()`	x	x	x	free an allocated storage block
`frexp()`	x	x	x	split a number into mantissa and exponent
`ftok()`		x		construct access key for IPC using messages or semaphores
`ftw()`		x		descend directory hierarchy applying a function everywhere
`gcvt()`	x	x	x	convert floating point to Fortan F or E string format
`getcwd()`		x		obtain the current directory name in string format
`getenv()`	x	x	x	obtain values for process environment variables
`getgrent()`	x	x	x	read group file entries sequentially
`getgrgid()`	x	x	x	read group file entries by group ID
`getgrnam()`	x	x	x	read group file entries by group name
`getlogin()`	x	x	x	obtain a pointer to a user login name entry
`getopt()`		x		obtain command lne options

Table 4-1 *Comparison of Library Routines— Continued*

Comparison of Library Routines				
routine	*V7*	*SV*	*4.2*	*description*
`getpass()`	x	x	x	read a password from a terminal without echoing
`getpw()`	x	x	x	obtain a user name from a user ID
`getpwent()`	x	x	x	read password file entries sequentially
`getpwnam()`	x	x	x	read password file entries by group name
`getpwuid()`	x	x	x	read password file entries by group ID
`getutent()`		x		read accounting file entries sequentially
`getutid()`		x		search an accounting file by type
`getutline()`		x		search an accounting file by device
`gmtime()`	x	x	x	obtain a time data structure containing the GMT time
`gsignal()`		x		send a signal to a process or a group of processes
`hcreate()`		x		create a hash-table
`hdistroy()`		x		remove a hash-table
`hsearch()`		x		search for an entry in a hash-table
`initgroups()`			x	initialize group access list
`irand48()`		x		return double precision random numbers from 0.0 to 1.0
`isalnum()`	x	x	x	test for alphanumeric character
`isalpha()`	x	x	x	test for alphabetic character
`isascii()`	x	x	x	test for ASCII character
`isatty()`	x	x	x	test whether a file is associated with a terminal
`iscntrl()`	x	x	x	test for control character
`isdigit()`	x	x	x	test for digit character
`isgraph()`		x		test for printable character excluding spaces
`islower()`	x	x	x	test for lower case character
`isprint()`	x	x	x	test for printable character
`ispunct()`	x	x	x	test for punctuation character
`isspace()`	x	x	x	test for white space character
`isupper()`	x	x	x	test for upper case character
`isxdigit()`		x		test for hexadecimal format data
`jrand48()`		x		return long integer random numbers from –2^31 to 2^31
`krand48()`		x		return double precision random numbers from 0.0 to 1.0
`l3tol()`	x	x	x	convert from 3 byte integers to long integers
`l64a()`		x		convert long integer to base 64 ASCII string
`ldexp()`	x	x	x	combine mantissa and exponent
`localtime()`	x	x	x	obtain a time data structure adjusted for local time
`longjmp()`	x	x	x	restore stack environment information
`lrand48()`		x		return long integer random numbers from 0 to 2^31
`lsearch()`		x		linear table search and update routine
`ltol3()`	x	x	x	convert from long integers to 3 byte integers
`malloc()`	x	x	x	allocate a storage block
`memccpy()`		x		copy memory stopping after a specified character
`memchr()`		x		search memory for characters
`memcmp()`		x		compare memory locations lexicographically
`memcpy()`		x		copy memory to memory
`memset()`		x		initialize memory to a constant value
`mktemp()`	x	x	x	make a unique file name using a template

Table 4-1 *Comparison of Library Routines— Continued*

Comparison of Library Routines

routine	*V7*	*SV*	*4.2*	*description*
modf()	x	x	x	split mantissa into integer and fraction
monitor()	x	x	x	prepare execution profile for a program
mrand48()		x		return long integer random numbers from –2^31 to 2^31
nlist()	x	x	x	get entries from an executable file's symbol table
nrand48()		x		return long integer random numbers from 0 to 2^31
perror()	x	x	x	produce error messages using standard output
pkopen()	x			packet driver simulator
putpwent()		x		write a password file entry
pututline()		x		write accounting file entries
qsort()	x	x	x	quicker sort algorithm
rand()	x	x	x	obtain successive pseudo random numbers range (0,32767)
random()			x	better random number generator than rand()
realloc()	x	x	x	change the size of a storage block
setgrent()	x	x	x	reposition to the start of the group file
setjmp()	x	x	x	save stack environment information
setkey()	x	x	x	initialize a key for use in encryption
setpwent()	x	x	x	reposition to the start of the password file
setutent()		x		reposition to the start of an accounting file
sleep()	x	x	x	suspend process execution for an interval of time
srand()	x	x	x	reset random number generator at a random starting point
swab()	x	x	x	exchange adjacent bytes
tdelete()		x		remove a binary tree node
timezone()	x		x	get the name of the timezone
toascii()		x		convert integer values to ASCII
tolower()	x	x	x	translate characters to lower case (function in System V)
toupper()	x	x	x	translate characters to upper case (function in System V)
_tolower()		x		macro version of the tolower function
_toupper()		x		macro version of the toupper function
tsearch()		x		create and search a binary tree
ttyname()	x	x	x	obtain the file name of a terminal in string format
ttyslot()	x	x	x	locate the accounting file entry for a terminal user
twalk()		x		traverse (walk) through nodes of a binary tree
tzset()		x		set time zone variables using an environment variable
utmpnane()		x		specify the accounting file to be examined
string functions				
index()	x		x	search for occurrence of character
rindex()	x		x	search backwards for occurrence of character
strcat()	x	x	x	concatenate two full strings
strchr()		x		search for occurrence of character: index()
strcmp()	x	x	x	lexical comparison of two full strings
strcpy()	x	x	x	copy a string into a second string
strlen()	x	x	x	obtain the length of a string
strncat()	x	x	x	append up to *n* characters to a string
strncmp()	x	x	x	lexical comparison of no more than *n* characters
strncpy()	x	x	x	copy *n* characters of a string

Table 4-1 *Comparison of Library Routines— Continued*

Comparison of Library Routines				
routine	*V7*	*SV*	*4.2*	*description*
`strnspn()`		x		obtain length of 1st string not matching characters of 2nd string
`strpbrk()`		x		search for a member of a set of characters
`strrchr()`		x		search backwards for occurrence of character: `rindex()`
`strspn()`		x		obtain length of 1st string matching characters of 2nd string
`strtok()`		x		search a string one token at a time
standard I/O				
`clearerr()`	x	x	x	reset error, end of file indicators
`ctermid()`		x		obtain the file name for a terminal
`cuserid()`		x		obtain the login name of user as a string
`exit()`	x	x	x	terminate process after cleaning up
`fclose()`	x	x	x	close a data stream
`fdopen()`	x	x	x	connect a data stream to an open file
`feof()`	x	x	x	test for an end of file condition
`ferror()`	x	x	x	test for error conditions
`fflush()`	x	x	x	flush a data stream without closing it
`fgetc()`	x	x	x	read a character from an input data stream
`fgets()`	x	x	x	read a string, but no more than *n* characters
`fileno()`	x	x	x	obtain the file descriptor for a data stream
`fopen()`	x	x	x	open a data stream
`fprintf()`	x	x	x	place output in a named output stream
`fputc()`	x	x	x	write a character on a data stream
`fputs()`	x	x	x	write a string onto an output stream
`fread()`	x	x	x	read buffered input from a data stream
`freopen()`	x	x	x	redirect output of an open data stream
`fscanf()`	x	x	x	scan input data from a named input stream
`fseek()`	x	x	x	reposition random read/write pointer
`ftell()`	x	x	x	determine current positio in a data stream
`fwrite()`	x	x	x	write buffered output to a data stream
`getc()`	x	x	x	read a character (macro version of `fgetc`)
`getchar()`	x	x	x	read a character from the standard input (macro)
`gets()`	x	x	x	read a string up to a newline
`getw()`	x	x	x	read a word from an input stream
`pclose()`	x	x	x	close an interprocess data stream
`popen()`	x	x	x	open an interprocess data stream
`printf()`	x	x	x	place output in the standard output
`putc()`	x	x	x	write a character (macro version of `fputc`)
`putchar()`	x	x	x	write a character to the standard output (macro)
`puts()`	x	x	x	write a string and append a newline
`putw()`	x	x	x	write a word on an output stream
`rewind()`	x	x	x	reposition to the beginning of a data stream
`scanf()`	x	x	x	scan input data from the standard input
`setbuf()`	x	x	x	assign a buffer to a data stream
`sprintf()`	x	x	x	place output in a character stream
`sscanf()`	x	x	x	scan input data from a character string
`ssignal()`		x		specify action to perform upon receipt of a signal

Table 4-1 *Comparison of Library Routines— Continued*

Comparison of Library Routines				
routine	*V7*	*SV*	*4.2*	*description*
`system()`	x	x	x	issue a shell command
`tempnam()`		x		obtain file name for temporary file in any directory
`tmpfile()`		x		create a temporary file
`tmpnam()`		x		obtain file name for temporary file in /tmp
`ungetc()`	x	x	x	put a character back into the input data stream
math library				
`hypot()`	x	x	x	Euclidean distance
`acos()`	x	x	x	arccosine function
`asin()`	x	x	x	arcsine function
`ceil()`	x	x	x	ceiling function
`log10()`	x	x	x	common logarithm
`cos()`	x	x	x	cosine function
`exp()`	x	x	x	exponential
`floor()`	x	x	x	floor function
`cosh()`	x	x	x	hyperbolic cosine function
`sinh()`	x	x	x	hyperbolic sine function
`tanh()`	x	x	x	hyperbolic tangent function
`fabs()`	x	x	x	floating point absolute value
`abs()`	x	x	x	integer absolute value
`matherr()`		x		math library error handling function
`log()`	x	x	x	natural logarithm function
`pow()`	x	x	x	raise a value to a given power
`sin()`	x	x	x	sine function
`sqrt()`	x	x	x	square root
`tan()`	x	x	x	tangent function
`gamma()`		x	x	log gamma function
`fmod()`		x		remainder function - modulo
`erfc()`		x		complementary error function: 1 – *erf*(x)
`erf()`		x		error function: *erf*(x)
`atan()`	x	x	x	arctangent function
`atan2()`	x	x	x	arctangent function
`j[01n]()`	x	x	x	Bessel functions of the first kind
`y[01n]()`	x	x	x	Bessel functions of the second kind
miscellaneous				
`arc()`	x	x	x	draw arc given the center and end points
`assert()`	x	x	x	debugging macro for embedding diagnostic code
`circle()`	x	x	x	draw circle given center and radius
`closepl()`	x	x	x	close a plotting device, writing buffered output
`cont()`	x	x	x	draw line between current position and second point
`curses`		x	x	cursor addressing and screen updating library
`dbm`	x		x	database management subroutines
`directory`		x	x	directory operations
`erase()`	x	x	x	clear the plotting area
`label()`	x	x	x	supply labels for plotting

Table 4-1 *Comparison of Library Routines— Continued*

Comparison of Library Routines				
routine	*V7*	*SV*	*4.2*	*description*
`line()`	x	x	x	connect two data points with a line
`linemod()`	x	x	x	specify style for connecting lines
`logname()`		x		obtain the login name of a user
`move()`	x	x	x	reposition the cursor
`mp`	x		x	multiple precision integer arithmetic library
`openpl()`	x	x	x	prepare plotting device to receive data
`point()`	x	x	x	plot a data point
`compile()`		x		compile a regular expression
`step()`		x		match regular expression anywhere in string
`advance()`		x		match regular expression at beginning of string
`re_comp()`			x	compile a regular expression
`re_exec()`			x	execute regular expression for a pattern match
`space()`	x	x	x	define the perimeter of a plotting space
`termlib`		x	x	terminal-independent operation library
`varargs`			x	variable argument list
networking				
`byteorder`			x	convert values between host and network byte order
`gethostent()`			x	get network host entry
`getnetent()`			x	get network entry
`getprotoent()`			x	get protocol entry
`getservent()`			x	get service entry
`inet_addr()`			x	Internet address manipulation
`rcmd()`			x	return stream to remote command (superuser)
`rexec()`			x	return stream to remote command

5

The System V Terminal Driver

5

The System V Terminal Driver

5.1. Introduction

The terminal driver for System V is completely incompatible with the Version 7 terminal driver, and hence with the 4.2 BSD terminal driver. This chapter gives some tips for programmers who want to know the differences between these terminal drivers.

The System V terminal driver provides the following advantages over the ''old'' Version 7 and the ''new'' Berkeley terminal drivers:

- Input modes, output modes, control modes, and local modes, can all be controlled separately. However, different input and output baud rates are not supported.
- The programming interface is more elegant. Naming is more consistent, and everything is together in one structure. On 4.2 BSD there are three separate structures!
- The command set is orthogonal. You can, for example, send and receive 8 bit wide data, without having to resort to RAW mode.†

It is only fair to mention that Berkeley's ''new'' terminal driver has a much better user interface than the System V terminal driver. Here are some advantages of the Berkeley interface:

- Line erase (or kill) and word erase using CTRL-W remove characters as they are erased. On System V, only character erase does this. There is no word erase in System V.
- Control characters are displayed as printable characters when they are typed, which helps naive users to identify typing errors. Erasing of tabs and control characters is done properly. On System V, column alignment is not attempted.
- The new terminal driver gives the user access to job control facilities, so that processes may be suspended with CTRL-Z, placed in the background, or brought back into the foreground.
- Separate input/output baud rates are theoretically possible, although it usually doesn't work in practice, because of device limitations.

† On 4.3 BSD this can be done by using the new PASS8 bit.

From a programming standpoint, the System V terminal driver is easier to work with, but from a user's standpoint, the Berkeley interface is greatly superior.

The rest of this chapter contains pairs of programs that show how to perform several common operations on the terminal interface. Compile the first program in a pair with `/bin/cc`, and the second program with `/usr/5bin/cc`. The Version 7 (and 4.2 BSD) header file for terminal control, `sgtty.h`, is replaced on System V by `termio.h`. The System V header file `fcntl.h` gives definitions for controlling open files; 4.2 BSD provides the same facilities and include file, but earlier Berkeley UNIX systems did not.

In the programs below, the Berkeley features were tested with `/bin/cc`, and the System V features with `/usr/5bin/cc`.

5.2. Setting the Interrupt Character

Over noisy transmission lines, the DEL character (which was the default interrupt character before 4.2 BSD) looks too much like a string of ones (line closed). If you work over phone lines during a storm, for example, you may find your work getting interrupted by transmission errors, which generate extraneous DEL characters. One solution is to change your interrupt character to something else, such as control-C. That's what the following program does. Of course, this could also be accomplished with the `stty` command, but that's too easy. First the program for 4.2 BSD:

```
#include <sgtty.h>

main()      /* set interrupt to control-C */
{
    struct tchars ctls;

    ioctl(0, TIOCGETC, &ctls);
    ctls.t_intrc = 003;     /* should use argv[1] */
    ioctl(0, TIOCSETC, &ctls);
}
```

Now the program for System V:

```
#include <sgtty.h>

main()      /* set interrupt to control-C */
{
    struct termio term;

    ioctl(0, TCGETA, &term);
    term.c_cc[VINTR] = 003;     /* should use argv[1] */
    ioctl(0, TCSETA, &term);
}
```

There was no way to reset your interrupt character from the shell on vanilla Version 7 systems. On Berkeley UNIX, the `tchars` structure contains the actual interrupt character, along with the quit signal, the start and stop characters, and the end-of-file character. On System V, these characters (with the exception of start and stop) are part of an array in the `termio` structure. This arrangement is more elegant than having two structures, `sgttyb` and `tchars`. However, System V provides no way to reset the stop (^S) and start (^Q) characters.

5.3. Read Without Waiting

When writing interactive software, and packet-based communications packages, it is often necessary to perform a read to look for input, returning immediately if there is nothing to be read. On System V, this is called read-no-delay. On 4.2 BSD, there is a special `ioctl` call that tells how many characters are waiting in the input queue. The following program sleeps two seconds, reads a line without hanging if there's nothing to read, and prints what has been typed before exiting. First the version for 4.2 BSD:

```
#include <stdio.h>
#include <sgtty.h>
#include <fcntl.h>

main(argc, argv)     /* read from terminal with no delay */
int argc;
char *argv[];
{
    char buf[BUFSIZ];
    long n = 0;
    int fd;

    puts("You have 2 seconds to type a line:");
    sleep(2);
    fd = open("/dev/tty", O_RDONLY);
    ioctl(fd, FIONREAD, &n);
    if (n == 0)
        puts("You didn't type anything.");
    else {
        read(fd, buf, sizeof(buf));
        printf("You typed: %s", buf);
    }
}
```

Now the version for System V:

```
#include <stdio.h>
#include <termio.h>
#include <fcntl.h>

main(argc, argv)    /* read from terminal with no delay */
int argc;
char *argv[];
{
    char buf[BUFSIZ];
    long n = 0;     /* so on v7 sure to read nothing */
    int fd;

    puts("You have 2 seconds to type a line:");
    sleep(2);
    fd = open("/dev/tty", O_RDONLY);
    fcntl(fd, F_SETFL, O_NDELAY);
    n = (long)read(fd, buf, sizeof(buf));
    if (n == 0)
        puts("You didn't type anything.");
    else
        printf("You typed: %s", buf);
}
```

On 4.2 BSD, you call `ioctl(FIONREAD)` to check if there are characters waiting to be read; if there are, you read them. On System V, you call `fcntl(O_NDELAY)` to specify that further reads should be done without waiting. This can also be accomplished from the `open()` call. Vanilla Version 7 systems provide no features for doing non-blocking reads, so on such systems, the above program will always say, "You didn't type anything" (if it compiles at all).

5.4. Two-Way Control Flow

The following program causes the terminal driver to send a control-S when its buffer is half full, and a control-Q when it is ready to receive data again. The terminal driver does not normally do control flow. This is helpful when you are uploading data from a microcomputer onto a UNIX system. Unless two-way control flow is enabled, data will probably be lost when buffers overflow. You can set this mode from the shell simply by typing `stty tandem` on 4.2 BSD, or `stty ixoff` on System V. However, for writing file transfer software, it would be best to set this mode from within a C program, for the sake of efficiency. First the program for 4.2 BSD:

```
#include <stdio.h>
#include <sgtty.h>

main(argc, argv)    /* enable 2way XON/XOFF control flow */
int argc;
char *argv[];
{
    struct sgttyb term;

    if (argc == 1) {
        fprintf(stderr, "Usage: %s [on|off]\n", argv[0]);
        exit(1);
    }
    if (strcmp(argv[1], "on") == 0) {
        ioctl(0, TIOCGETP, &term);
        term.sg_flags |= TANDEM;
        ioctl(0, TIOCSETP, &term);
    }
    if (strcmp(argv[1], "off") == 0) {
        ioctl(0, TIOCGETP, &term);
        term.sg_flags &= ~TANDEM;
        ioctl(0, TIOCSETP, &term);
    }
}
```

Now the program for System V.

```
#include <stdio.h>
#include <termio.h>

main(argc, argv)    /* enable 2way XON/XOFF control flow */
int argc;
char *argv[];
{
    struct termio term;

    if (argc == 1) {
        fprintf(stderr, "Usage: %s [on|off]\n", argv[0]);
        exit(1);
    }
    if (strcmp(argv[1], "on") == 0) {
        ioctl(0, TCGETA, &term);
        term.c_iflag |= IXOFF;
        ioctl(0, TCSETA, &term);
    }
    if (strcmp(argv[1], "off") == 0) {
        ioctl(0, TCGETA, &term);
        term.c_iflag &= ~IXOFF;
        ioctl(0, TCSETA, &term);
    }
}
```

On Version 7, the terminal control structure was called `sgttyb`; on System V it is called termio. Most of the differences here are matters of name only. What used to be called TANDEM is now called IXOFF. The Version 7 field sg_flags was separated for System V into c_iflag for input, and c_oflag for output. The arguments to ioctl are now TCGETA and TCSETA rather than TIOCGETP and TIOCSETP. Note that we could have used gtty and stty instead of the ioctl calls for 4.2 BSD; however, gtty and stty are considered obsolete.

5.5. Full Screen Software

When writing screen-oriented software such as editors, database forms editors, and visual games, it's best to use a terminal control state somewhere between cooked and raw mode. This half-baked mode is sometimes called ''rare'' mode. On Version 7 and all BSD systems, you set CBREAK mode. On System V, you disable canonical input processing.

The following program sets the terminal line to rare mode, after saving the original state. It then calls a function that would ordinarily handle the screen. In this example, however, it just executes the stty command, to show the terminal line settings. Finally, it resets the terminal to its original state, and exits. The screen function would normally be filled in with code to handle screen input and output. Note that for most programming applications, however, it would be better to use curses, if this package is available at your site. First the program for 4.2 BSD:

```
#include <stdio.h>
#include <sgtty.h>

main(argc, argv)     /* set tty to rare for screen work */
int argc;
char *argv[];
{
    struct sgttyb tty, save;

    ioctl(0, TIOCGETP, &save);
    ioctl(0, TIOCGETP, &tty);
    tty.sg_flags |= CBREAK;
    tty.sg_flags &= ~(ECHO|XTABS|CRMOD);
    ioctl(0, TIOCSETP, &tty);
    screen();
    ioctl(0, TIOCSETP, &save);
    exit(0);
}

screen()        /* handle screen-oriented functions */
{
    system("stty");     /* just test for now */
}
```

Now the program for System V:

```
#include <stdio.h>
#include <termio.h>

main(argc, argv)    /* set tty to rare for screen work */
int argc;
char *argv[];
{
    struct termio tty, save;

    ioctl(0, TCGETA, &save);
    ioctl(0, TCGETA, &tty);
    tty.c_iflag &= ~ICRNL;
    tty.c_oflag &= ~(TAB3|ONLCR);
    tty.c_lflag &= ~(ECHO|ICANON);
    ioctl(0, TCSETA, &tty);
    screen();
    ioctl(0, TCSETA, &save);
    exit(0);
}

screen()        /* handle screen-oriented functions */
{
    system("stty");     /* just test for now */
}
```

On System V, you turn off the input mapping of carriage return to newline by doing a bitwise and of the one's complement of ICRNL into the input flags. You turn off the output mapping of tab to spaces, and the output mapping of carriage return to newline, by doing likewise with TAB3 and OCRNL, into the output flags. Finally, you turn off local echoing and canonical input processing by doing the same thing with ECHO and ICANON, into the local flags.

On Version 7 and all BSD systems, input and output modes are together, so turning off CRMOD accomplishes the same thing as turning off ICRNL and OCRNL. Echoing is turned off by twiddling the bit for ECHO, and the mapping of tab to spaces is turned off by using XTABS (which is a slightly better name for this than TAB3 on System V). On the other hand, CBREAK has to be turned on, so you should bitwise or it into the tty flags.

6

File and Record Locking

6

File and Record Locking

6.1. Introduction

Locking prevents multiple processes from modifying a file at the same time. File locking works on an entire file, while record locking works on a region of a file; actually file locking is an extreme case of record locking. Records can be locked either against reading or writing. Generally read locks are shared (allowing other processes to read the record), while write locks are exclusive (allowing no other processes to read or write the record). Locks can be blocking or non-blocking – that is, they can wait until the record is no longer locked, or they can return immediately with an error.

Locks may be advisory, in which case the cooperation of all processes is required, or mandatory, so that no process can violate the lock. Mandatory locks pose serious security problems: for example, what would happen if `/etc/passwd` were locked against reading? Sun's locking scheme is advisory rather than mandatory.

The most primitive locking method is to create a temporary file that acts as a lock. This is inelegant, inefficient, and insecure. Without inter-process communication (IPC), the best alternative is to create a lock device driver, which must be configured into the kernel. With the IPC facilities of 4.2 BSD, locking could be done with sockets. With the IPC facilities of System V, locking can be done quite easily with semaphores. Neither is necessary, because release 3.2 includes the standard file and record locking facilities from the *System V Interface Definition* (SVID).

6.2. File and Record Locking on System V

In the spring of 1981, John Bass published a paper in the Usenix newsletter *;login:*, detailing the interface and implementation of a `locking()` system call for file and record locking. His proposal called for mandatory locks. About a year later, /usr/group published a standard including the `lockf()` system call, with the same parameters and locking modes. The difference was that the /usr/group standard allowed for both mandatory and advisory locks. Advisory locks may be circumvented by programs not using `lockf()`, while mandatory locks cannot. Only files with the `setgid` bit set are subject to mandatory locking under this standard.

Mandatory locks are probably not necessary. Both OS/360 and VAX/VMS have survived for years with advisory locks only. Many large databases, including airline reservation systems, have been implemented on these operating systems. Furthermore, mandatory locks are a potential security problem. Some user program could lock /etc/passwd (if it had the setgid bit set) and then go to sleep, causing the entire protection subsystem to hang.

The lockf() system call allows a process to lock sections of a file. Other processes that attempt to lock that section will either return an error value, or block until the section becomes unlocked. All locks on a file are removed when the file is closed, and all locks for a process are removed when the process terminates. The call looks like this:

```
lockf(fd, mode, length)
int fd, mode;
long length;
```

The file descriptor fd must come from a successful open(), creat(), pipe(), or dup() system call. The mode may be F_LOCK to lock a region for exclusive use, F_TEST to test for other locks, F_TLOCK to both test and lock, and F_ULOCK to unlock a region. Actually F_TLOCK is a non-blocking lock: if a region is locked, it returns an error, rather than sleeping. The third parameter length is the number of bytes to lock, specified from the current position in the file. This can of course be changed with the lseek() system call. Negative values indicate how far back to lock, from the current position. Locking past end of file protects against appending. If locked regions overlap, they are combined into a single region.

The potential for deadlock occurs if a process controlling a locked resource gets put to sleep by accessing another process's locked resource. Thus, calls to lockf() scan for a deadlock before sleeping on a locked resource. An error is returned if sleeping on a locked resource would cause a deadlock. Sleeping on a resource is interrupted by any signal. Thus, the alarm() system call can be used to provide a timeout facility if necessary.

Record locking for a simple ISAM database is relatively straightforward: lock the data, and lock the index pointer for the data. But in a B-tree database, record locking is much harder. Modifying a record may require shuffling the leaves of the tree. The safest and easiest thing is to lock the entire B-tree, but this may not be acceptable in highly sophisticated applications.

Originally, System V had no file or record locking features. However, the SVID finally included the /usr/group locking standard, with advisory locks only. Mandatory locking will be included in future specifications. The programming usage is the same as in the /usr/group standard, as are the modes F_LOCK, F_TEST, F_TLOCK, and F_ULOCK.

The most interesting part of the AT&T specification is that locking can be controlled with the fcntl() system call. This affords a distinction between read locks and write locks, something not present in the /usr/group standard.

Locks may also be established with the F_SETLK or F_SETLKW command to fcntl(); the distinction is that F_SETLKW blocks (or waits), whereas F_SETLK is non-blocking. Either command takes the arguments F_RDLCK and F_WRLCK to lock, and F_UNLCK to unlock. A read lock using F_RDLCK prevents other processes from write-locking the protected area. More than one read lock may exist for a given region at any given time. The file descriptor in question must have been opened with read access. A write lock with F_WRLCK prevents any process from read-locking or write-locking the protected area. Only one write lock may exist for a given segment of a file at any time. The file descriptor in question must have been opened with write access.

In a production database system, it is best to place a read lock on a record during browsing. If the record gets modified, the read lock can be upgraded to a write lock (if no other read locks exist), the record quickly updated, and the write lock removed. The system must arbitrate race conditions, as when two processes are both waiting for a write lock.

6.3. File Locking on 4.2 BSD

The first major release to include file locking was 4.2 BSD with the flock() system call, which allows processes to place advisory locks on files. Since advisory locks are not enforced by the operating system, flock() is useful primarily for cooperating processes that have already agreed upon a locking protocol. When a process attempts to lock a file already locked by another process, flock() blocks until the first process releases the lock. If called with the LOCK_NB (noblock) option, however, flock() returns the error EWOULDBLOCK when a file is already locked, instead of blocking. Both exclusive and shared locks are available. At any one time, a file may have only one exclusive lock, but multiple shared locks are permitted. This call establishes an exclusive lock, and will block until the lock in effect is released:

```
if (flock(fd, LOCK_EX) < 0)
    perror("fatal error: flock");
```

This call establishes a shared lock. It will block if the file has an exclusive lock, but not if there are multiple shared locks:

```
if (flock(fd, LOCK_SH) < 0)
    perror("fatal error: flock");
```

This call establishes an exclusive, non-blocking lock. It can be used when it is acceptable to try again later, rather then waiting until a lock is released:

```
if (flock(fd, LOCK_EX|LOCK_NB) < 0)
    perror("try again later: flock");
```

This call releases any of the above locks:

```
(void)flock(fd, LOCK_UN);
```

In all the examples above, the file descriptor fd is obtained from a system call such as open(). The flock() system call returns –1 if the file descriptor is invalid, or if it does not refer to a file.

Some Berkeley UNIX commands that establish locks are `tip` (when writing a log of the call), `dump` (when recording information about incremental dumps), and some versions of `mail` (for locking the spool file during mail browsing).

The main problem with the `flock()` facility is that it lacks record locking. Database applications could perform record locking by using a socket-based lock manager, but this would be slow compared to record locking done by the kernel. Another problem with `flock()` is that its semantics fail to generalize to a networked environment.

6.4. Locking in a Network Environment

In order for `lockf()` to work on remote-mounted filesystems, the network lock daemon `rpc.lock` has to be running on the NFS server machine. See *lockd*(8c) for details.

7

Tuning IPC System Parameters

7

Tuning IPC System Parameters

7.1. Introduction

The System V Inter-Process Communications (IPC) extensions to UNIX are now implemented in the Sun UNIX operating system. These IPC extensions provide mechanisms for message passing, semaphores, and shared memory.

Data structures that describe and control various IPC functions are allocated in the Sun UNIX kernel at system initialization and remain resident in memory as long as the operating system is running. The size of these structures is determined by a variety of tunable parameters in the system configuration file. Some IPC parameters also exist to control and limit the resources dynamically allocated by IPC subsystems.

This document describes these parameters and gives some guidelines for tuning them. Before attempting to modify any of these parameters, you should be thoroughly familiar with the capabilities provided by the IPC system, and with your particular application's needs.

Refer to the *AT&T System V Interface Definition* (SVID) and Sun's *UNIX Interface Reference Manual* for a detailed description of the IPC system interface. But note that there are now two separate IPC subsystems – the one from 4.2 BSD involving sockets, and the one from System V discussed here.

7.2. Why Reconfigure IPC Parameters?

The Sun Operating System, as shipped, is configured to support a modest level of IPC activity. Applications that require more IPC resources than are provided in the distributed system must be run with reconfigured kernels.

Note: in general, Sun recommends that all system kernels be reconfigured in order to reduce the base-level memory requirements. Instructions on doing this may be found in the manual *Installing UNIX On The Sun Workstation.* The following paragraphs assume a working knowledge of the system configuration procedure.

IPC parameters may be tuned by editing the system configuration file `/usr/sys/conf/SYSTEM_NAME` and rebuilding the kernel. In order to adjust the default setting of a particular parameter, insert a line of the following form into the configuration file:

```
options  OPTION_NAME = OPTION_VALUE
```

Default values are assumed for all unspecified parameters. Note: do not change the `options` lines that refer to the IPC subsystems unless you wish to disable the entire subsystem.

Users of System V may note that there is not an exact correspondence between Sun's tunable parameters and those found in the AT&T System V release. This is because several implementation algorithms have been changed, rendering some of the parameters meaningless. In particular, certain static structures in System V are allocated dynamically by the Sun kernel, obviating the need for configurable limits. In future Sun releases, there are likely to be even fewer tunable parameters. It should be noted that although some parameters have been omitted, there are no semantic differences in the IPC system call operation.

For the remaining tunable parameters, reasonable values may be estimated using information about the application programs' IPC usage. The following sections specify the default values and attempt to give tuning guidelines. It is generally a good idea to write application programs that recover gracefully from resource allocation failures, so that system configuration requirements surface early in the development cycle. Applications that require large amounts of IPC resources are often poorly designed and should be carefully reviewed before making drastic increases in the IPC parameters.[1]

7.3. IPC Message Parameters

These parameters control system characteristics associated with inter-process message passing.

MSGPOOL

MSGPOOL defines the size (in kilobytes) of the kernel message memory pool. All queued IPC messages are stored in this pool. The behavior of the *msgsnd*(2) system call when the message pool is full depends on the value of the msgflg argument; see *msgop*(2). Attempts to queue messages larger than the message pool return EINVAL.

The IPC message pool is allocated at system initialization and is never made available for other uses. Thus, tuning the MSGPOOL parameter involves a tradeoff between the performance of processes that depend upon a high level of message activity, and the degradation of overall system performance due to wasted memory. This parameter may not exceed 255.

```
options     MSGPOOL=8
```

MSGMNB

MSGMNB defines the maximum number of message bytes that may be queued on any particular message queue. Attempts to queue messages that would exceed this limit either sleep or return EAGAIN; see *msgop*(2). MSGMNB is used as a default value for msg_qbytes when message queues are created; this limit may be lowered by any process but only the super-user may raise the limits on a particular message queue; see the *msgctl*(2) option named IPC_SET.

```
options     MSGMNB=2048
```

[1] The total amount of memory available for user processes may be estimated using the *vmstat*(8) program.

MSGMNI

MSGMNI defines the maximum number of uniquely identifiable message queues that may exist simultaneously. Attempts to create more than MSGMNI message queues return ENOSPC; see *msgget*(2). Each increment of MSGMNI reserves 49 bytes of kernel memory.

```
options     MSGMNI=50
```

MSGTQL

MSGTQL defines the total number of undelivered messages that may exist at any instant. Attempts to queue more than MSGTQL messages either sleep or return EAGAIN; see *msgop*(2). Since zero-length messages are allowed, this limit could theoretically be set arbitrarily high. Each increment of MSGTQL reserves 12 bytes.

```
options     MSGTQL=50
```

7.4. IPC Semaphore Parameters

These parameters control system characteristics associated with inter-process semaphores.

SEMMNI

SEMMNI defines the maximum number of uniquely identifiable semaphore clusters that may exist simultaneously. Attempts to create more than SEMMNI semaphore clusters return ENOSPC; see *semget*(2). Although SEMMNI may be set arbitrarily high, there is no reason to set it to be larger than SEMMNS. Each increment of SEMMNI reserves 32 bytes of kernel memory.

```
options     SEMMNI=10
```

SEMMNS

SEMMNS defines the maximum number of semaphores in the system. Attempts to create semaphore clusters when there are not enough semaphores available result in an ENOSPC error; see *semget*(2). Attempts to create semaphore clusters with more than SEMMNS semaphores return EINVAL. Each increment of SEMMNS reserves 8 bytes.

```
options     SEMMNS=60
```

SEMUME

SEMUME defines the maximum number of semaphores (per process) that may simultaneously have non-zero adjust-on-exit values. The adjust-on-exit values are manipulated when semaphore operations are requested in conjunction with the SEM_UNDO flag. Attempts to exceed this limit return EINVAL; see *semop*(2). The value of SEMUME affects the number of bytes allocated for semaphore undo structures (see SEMMNU below). The value of SEMUME must be less than 32768.

```
options     SEMUME=10
```

SEMMNU

SEMMNU defines the maximum number of processes that may simultaneously be using the IPC SEM_UNDO feature. Attempts to exceed this limit result in an ENOSPC error; see *semop*(2). There is no reason to set SEMMNU larger than the maximum number of processes that can run on the system (approximately 16*maxusers, where maxusers is configurable, defaulting to 4). Therefore, SEMMNU need not exceed 64, under normal circumstances. Each increment of SEMMNU allocates 14+(8*SEMUME) bytes.

```
options     SEMMNU=30
```

7.5. IPC Shared Memory Parameters

These parameters control system characteristics associated with inter-process shared memory.

SHMPOOL

SHMPOOL defines the total amount of shared memory (in kilobytes) that may be in use in the system at any given time. Attempts to exceed this limit result in an ENOSPC error. In the current implementation, shared memory is allocated in non-paging memory, and can lead to system lockup if indiscriminately allocated. This restriction will be removed in a future Sun release. SHMPOOL is specified in kilobytes, but is rounded up to the page size of the target machine (2048 bytes on Sun-2, and 8192 bytes on Sun-3).

```
options     SHMPOOL=512
```

SHMSEG

SHMSEG defines the maximum number of shared memory segments that may be attached to a single task. Attempts to attach to more than SHMSEG segments return EMFILE; see *shmop*(2). Each increment of SHMSEG reserves 4 bytes for each process that can run on the system (approximately 16*maxusers, where maxusers is configurable, defaulting to 4). Therefore, each increment of SHMSEG ordinarily allocates 256 bytes.

```
options     SHMSEG=6
```

SHMMNI

SHMMNI defines the maximum number of shared memory segments that may simultaneously exist in the system. Attempts to exceed this limit return ENOSPC; see *shmget*(2). Each increment of SHMMNI reserves 42 bytes.

```
options     SHMMNI=100
```

Corporate Headquarters
Sun Microsystems, Inc.
2250 Garcia Avenue
Mountain View, CA 94043
415 960-1300
TLX 287815

For U.S. Sales Office locations, call:
800 821-4643
In CA: 800 821-4642

European Headquarters
Sun Microsystems Europe, Inc.
Sun House
31-41 Pembroke Broadway
Camberley
Surrey GU15 3XD
England
0276 62111
TLX 859017

Australia: 61-2-436-4699
Canada: 416 477-6745
France: (1) 46 30 23 24
Germany: (089) 95094-0
Japan: (03) 221-7021
The Netherlands: 02155 24888
UK: 0276 62111

Europe, Middle East, and Africa, call European Headquarters:
0276 62111

Elsewhere in the world, call Corporate Headquarters:
415 960-1300
Intercontinental Sales