Regenerating System Software

Charles B. Haley

Dennis. M. Ritchie

Introduction

This document discusses how to assemble or compile various parts of the UNIX® system software. This may be necessary because a command or library is accidentally deleted or otherwise destroyed; also, it may be desirable to install a modified version of some command or library routine. A few commands depend to some degree on the current configuration of the system; thus in any new system modifications to some commands are advisable. Most of the likely modifications relate to the standard disk devices contained in the system. For example, the df(1) ('disk free') command has built into it the names of the standardly present disk storage drives (e.g. '/dev/rf0', '/dev/rp0'). Df(1) takes an argument to indicate which disk to examine, but it is convenient if its default argument is adjusted to reflect the ordinarily present devices. The companion document 'Setting up UNIX' discusses which commands are likely to require changes.

Where Commands and Subroutines Live

The source files for commands and subroutines reside in several subdirectories of the directory /usr/src. These subdirectories, and a general description of their contents, are

cmd	Source files for commands.
libc/stdio	Source files making up the 'standard i/o package'.
libc/sys	Source files for the C system call interfaces.
libc/gen	Source files for most of the remaining routines described in section 3 of the manual.
libc/crt	Source files making up the C runtime support package, as in call save-return and long arithmetic.
libc/csu	Source for the C startup routines.
games	Source for (some of) the games. No great care has been taken to try to make it obvious how to compile these; treat it as a game.
libF77	Source for the Fortran 77 runtime library, exclusive of IO.
libI77	Source for the Fortran 77 IO runtime routines.
libdbm	Source for the 'data-base manager' package *dbm* (3).
libfpsim	Source for the floating-point simulator routine.
libm	Source for the mathematical library.
libplot	Source for plotting routines.

Commands

The regeneration of most commands is straightforward. The 'cmd' directory will contain either a source file for the command or a subdirectory containing the set of files that make up the command. If it is a single file the command

```
cd /usr/src/cmd
cmake cmd_name
```

suffices. (Cmd_name is the name of the command you are playing with.) The result of the cmake command will be an executable version. If you type cmake −cp cmd_name

the result will be copied to /bin (or perhaps /etc or other places if appropriate).

If the source files are in a subdirectory there will be a 'makefile' (see make(1)) to control the regeneration. After changing to the proper directory (cd(1)) you type one of the following:

make all The program is compiled and loaded; the executable is left in the current directory.

make cp The program is compiled and loaded, and the executable is installed. Everything is cleaned up afterwards; for example .o files are deleted.

make cmp The program is compiled and loaded, and the executable is compared against the one in /bin.

Some of the makefiles have other options. Print (cat(1)) the ones you are interested in to find out.

The Assembler

The assembler consists of two executable files: /bin/as and /lib/as2. The first is the 0-th pass: it reads the source program, converts it to an intermediate form in a temporary file '/tmp/atm0?', and estimates the final locations of symbols. It also makes two or three other temporary files which contain the ordinary symbol table, a table of temporary symbols (like 1:) and possibly an overflow intermediate file. The program /lib/as2 acts as an ordinary multiple pass assembler with input taken from the files produced by /bin/as.

The source files for /bin/as are named '/usr/src/cmd/as/as1?.s' (there are 9 of them); /lib/as2 is produced from the source files '/usr/src/cmd/as/as2?.s'; they likewise are 9 in number. Considerable care should be exercised in replacing either component of the assembler. Remember that if the assembler is lost, the only recourse is to replace it from some backup storage; a broken assembler cannot assemble itself.

The C Compiler

The C compiler consists of seven routines: '/bin/cc', which calls the phases of the compiler proper, the compiler control line expander '/lib/cpp', the assembler ('as'), and the loader ('ld'). The phases of the C compiler are '/lib/c0', which is the first phase of the compiler; '/lib/c1', which is the second phase of the compiler; and '/lib/c2', which is the optional third phase optimizer. The loss of the C compiler is as serious as that of the assembler.

The source for /bin/cc resides in '/usr/src/cmd/cc.c'. Its loss alone (or that of c2) is not fatal. If needed, prog.c can be compiled by

```
/lib/cpp prog.c >temp0
/lib/c0 temp0 temp1 temp2
/lib/c1 temp1 temp2 temp3
as − temp3
ld −n /lib/crt0.o a.out −lc
```

The source for the compiler proper is in the directory /usr/src/cmd/c. The first phase (/lib/c0) is generated from the files c00.c, ..., c05.c, which must be compiled by the C compiler. There is also c0.h, a header file *included* by the C programs of the first phase. To make a new /lib/c0 use

make c0

Before installing the new c0, it is prudent to save the old one someplace.

The second phase of C (/lib/c1) is generated from the source files c10.c, ..., c13.c, the include-file c1.h, and a set of object-code tables combined into table.o. To generate a new second phase use

make c1

It is likewise prudent to save c1 before installing a new version. In fact in general it is wise to save the object files for the C compiler so that if disaster strikes C can be reconstituted without a working version of the compiler.

In a similar manner, the third phase of the C compiler (/lib/c2) is made up from the files c20.c and c21.c together with c2.h. Its loss is not critical since it is completely optional.

The set of tables mentioned above is generated from the file table.s. This '.s' file is not in fact assembler source; it must be converted by use of the *cvopt* program, whose source and object are located in the C directory. Normally this is taken care of by make(1). You might want to look at the makefile to see what it does.

UNIX

The source and object programs for UNIX are kept in four subdirectories of */usr/sys.* In the subdirectory *h* there are several files ending in '.h'; these are header files which are picked up (via '#include ...') as required by each system module. The subdirectory *dev* consists mostly of the device drivers together with a few other things. The subdirectory *sys* is the rest of the system. There are files of the form LIBx in the directories sys and dev. These are archives (ar(1)) which contain the object versions of the routines in the directory.

Subdirectory *conf* contains the files which control device configuration of the system. *L.s* specifies the contents of the interrupt vectors; *c.c* contains the tables which relate device numbers to handler routines. A third file, *mch.s*, contains all the machine-language code in the system. A fourth file, *mch0.s*, is generated by mkconf(1) and contains flags indicating what sort of tape drive is available for taking crash dumps.

There are two ways to recreate the system. Use

```
cd /usr/sys/conf
make unix
```

if the libraries /usr/sys/dev/LIB2 and /usr/sys/sys/LIB1, and also c.o and l.o, are correct. Use

```
cd /usr/sys/conf
make all
```

to recompile everything and recreate the libraries from scratch. This is needed, for example, when a header included in several source files is changed. See 'Setting Up UNIX' for other information about configuration and such.

When the make is done, the new system is present in the current directory as 'unix'. It should be tested before destroying the currently running '/unix', this is best done by doing something like

```
mv /unix /ounix
mv unix /unix
```

If the new system doesn't work, you can still boot 'ounix' and come up (see boot(8)). When you have satisfied yourself that the new system works, remove /ounix.

To install a new device driver, compile it and put it into its library. The best way to put it into the library is to use the command

```
ar uv LIB2 x.o
```

where x is the routine you just compiled. (All the device drivers distributed with the system are already in the library.)

Next, the device's interrupt vector must be entered in l.s. This is probably already done by the routine mkconf(1), but if the device is esoteric or nonstandard you will have to massage l.s by hand. This involves placing a pointer to a callout routine and the device's priority level in the vector. Use some other device (like the console) as a guide. Notice that the entries in l.s must be in order as the assembler does not permit moving the location counter '.' backwards. The assembler also does not permit assignation of an absolute number to '.', which is the reason for the '. = ZERO+100' subterfuge. If a constant smaller than 16(10) is added to the priority level, this number will be available as the first argument of the interrupt routine. This stratagem is used when several similar devices share the same interrupt routine (as in dl11's).

If you have to massage l.s, be sure to add the code to actually transfer to the interrupt routine. Again use the console as a guide. The apparent strangeness of this code is due to running the kernel in separate

I&D space. The *call* routine saves registers as required and prepares a C-style call on the actual interrupt routine named after the 'jmp' instruction. When the routine returns, *call* restores the registers and performs an rti instruction. As an aside, note that external names in C programs have an underscore ('_') prepended to them.

The second step which must be performed to add a device unknown to mkconf is to add it to the configuration table /usr/sys/conf/c.c. This file contains two subtables, one for block-type devices, and one for character-type devices. Block devices include disks, DECtape, and magtape. All other devices are character devices. A line in each of these tables gives all the information the system needs to know about the device handler; the ordinal position of the line in the table implies its major device number, starting at 0.

There are four subentries per line in the block device table, which give its open routine, close routine, strategy routine, and device table. The open and close routines may be nonexistent, in which case the name 'nulldev' is given; this routine merely returns. The strategy routine is called to do any I/O, and the device table contains status information for the device.

For character devices, each line in the table specifies a routine for open, close, read, and write, and one which sets and returns device-specific status (used, for example, for stty and gtty on typewriters). If there is no open or close routine, 'nulldev' may be given; if there is no read, write, or status routine, 'nodev' may be given. Nodev sets an error flag and returns.

The final step which must be taken to install a device is to make a special file for it. This is done by mknod(1), to which you must specify the device class (block or character), major device number (relative line in the configuration table) and minor device number (which is made available to the driver at appropriate times).

The documents 'Setting up Unix' and 'The Unix IO system' may aid in comprehending these steps.

The Library libc.a

The library /lib/libc.a is where most of the subroutines described in sections 2 and 3 of the manual are kept. This library can be remade using the following commands:

```
cd /usr/src/libc
sh compall
sh mklib
mv libc.a /lib/libc.a
```

If single routines need to be recompiled and replaced, use

```
cc −c −O x.c
ar vr /lib/libc.a x.o
rm x.o
```

The above can also be used to put new items into the library. See ar(1), lorder(1), and tsort(1).

The routines in /usr/src/cmd/libc/csu (C start up) are not in libc.a. These are separately assembled and put into /lib. The commands to do this are

```
cd /usr/src/libc/csu
as − x.s
mv a.out /lib/x
```

where x is the routine you want.

Other Libraries

Likewise, the directories containing the source for the other libraries have files compall (that recompiles everything) and mklib (that recreates the library).

System Tuning

There are several tunable parameters in the system. These set the size of various tables and limits. They are found in the file /usr/sys/h/param.h as manifests ('#define's). Their values are rather generous in the system as distributed. Our typical maximum number of users is about 20, but there are many daemon processes.

When any parameter is changed, it is prudent to recompile the entire system, as discussed above. A brief discussion of each follows:

NBUF This sets the size of the disk buffer cache. Each buffer is 512 bytes. This number should be around 25 plus NMOUNT, or as big as can be if the above number of buffers cause the system to not fit in memory.

NFILE This sets the maximum number of open files. An entry is made in this table every time a file is 'opened' (see open(2), creat(2)). Processes share these table entries across forks (fork(2)). This number should be about the same size as NINODE below. (It can be a bit smaller.)

NMOUNT This indicates the maximum number of mounted file systems. Make it big enough that you don't run out at inconvenient times.

MAXMEM This sets an administrative limit on the amount of memory a process may have. It is set automatically if the amount of physical memory is small, and thus should not need to be changed.

MAXUPRC This sets the maximum number of processes that any one user can be running at any one time. This should be set just large enough that people can get work done but not so large that a user can hog all the processes available (usually by accident!).

NPROC This sets the maximum number of processes that can be active. It depends on the demand pattern of the typical user; we seem to need about 8 times the number of terminals.

NINODE This sets the size of the inode table. There is one entry in the inode table for every open device, current working directory, sticky text segment, open file, and mounted device. Note that if two users have a file open there is still only one entry in the inode table. A reasonable rule of thumb for the size of this table is

NPROC + NMOUNT + (number of terminals)

SSIZE The initial size of a process stack. This may be made bigger if commonly run processes have large data areas on the stack.

SINCR The size of the stack growth increment.

NOFILE This sets the maximum number of files that any one process can have open. 20 is plenty.

CANBSIZ This is the size of the typewriter canonicalization buffer. It is in this buffer that erase and kill processing is done. Thus this is the maximum size of an input typewriter line. 256 is usually plenty.

CMAPSIZ The number of fragments that memory can be broken into. This should be big enough that it never runs out. The theoretical maximum is twice the number of processes, but this is a vast overestimate in practice. 50 seems enough.

SMAPSIZ Same as CMAPSIZ except for secondary (swap) memory.

NCALL This is the size of the callout table. Callouts are entered in this table when some sort of internal system timing must be done, as in carriage return delays for terminals. The number must be big enough to handle all such requests.

NTEXT The maximum number of simultaneously executing pure programs. This should be big enough so as to not run out of space under heavy load. A reasonable rule of thumb is about

(number of terminals) + (number of sticky programs)

NCLIST The number of clist segments. A clist segment is 6 characters. NCLIST should be big enough so that the list doesn't become exhausted when the machine is busy. The characters that have arrived from a terminal and are waiting to be given to a process live here. Thus enough space should be left so that every terminal can have at least one average line pending (about 30 or 40 characters).

TIMEZONE	The number of minutes westward from Greenwich. See ‘Setting Up UNIX’.
DSTFLAG	See ‘Setting Up UNIX’ section on time conversion.
MSGBUFS	The maximum number of characters of system error messages saved. This is used as a circular buffer.
NCARGS	The maximum number of characters in an exec(2) arglist. This number controls how many arguments can be passed into a process. 5120 is practically infinite.
HZ	Set to the frequency of the system clock (e.g., 50 for a 50 Hz. clock).